L-Systems Pattern

- create a drawing function for the F production rule
- create drawing functions for any other production rules

```
// production rule: F -> F-F++F-F
// angle: 60 degrees, scale factor 1/3
// depth is the number of levels of recursion remaining
// len is the length of the line to draw
void drawF ( int depth, float len ) {
  if ( depth == 0 ) {
    // do the 'F' action - draw line, move turtle
    line(0, 0, len, 0);
    translate(len, 0);
  } else {
    // otherwise, do what the rule states: F-F++F-F
    drawF(depth-1, len/3);
    rotate(radians(-60));
    drawF(depth-1, len/3);
    rotate(radians(60));
    rotate(radians(60));
    drawF(depth-1, len/3);
    rotate(radians(-60));
    drawF(depth-1, len/3);
  }
}
```

DIVISION WARNING! Writing (1/3)*len, while mathematically correct, will result in 0! If you want division to include decimal points, use floats – (1.0/3.0)*len

scale factor 1/3 means len/3
rotation angle 60 degrees

depth decreases by 1

- the F production rule becomes a drawing function

```
// production rule: F -> …
void drawF ( int depth, float len ) {
  if ( depth == 0 ) {
    // 'F' means draw line, move turtle
    line(0, 0, len, 0);
    translate(len, 0);
  } else {
    do what the right side of the rule states: …
  }
}
```

coordinates are interpreted relative to the turtle so this will draw a line in front of the turtle rather than in the upper left corner of the window

- reference to a production symbol → call to that function (depth decreases by 1, len may be adjusted by fractal's scale factor)
- +, - → rotate(*a*) or rotate(-*a*) (angle *a* depends on the fractal)
- [,] → pushMatrix(), popMatrix()

CPSC 120: Principles of Computer Science • Fall 2

- any other production rules also become drawing functions

```
// production rule: S → …
void drawS ( int depth, float len ) {
  if ( depth == 0 ) {
    do the action for symbol S (may be nothing)
  } else {
    do what the right side of the rule states: …
  }
}
```

- reference to a production symbol → call to that function (depth decreases by 1, len may be adjusted by fractal's scale factor)
- +, - → rotate(*a*) or rotate(-*a*) (angle *a* depends on the fractal)
- [,] → pushMatrix(), popMatrix()

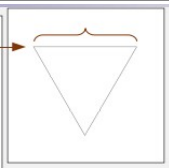
- create a drawing function for the whole pattern
- call the drawing function to actually draw the fractal

```
// draw a Koch snowflake
//  (x,y) is the upper left corner of the initial triangle
//  len is the width of the initial triangle
//  depth is the maximum depth of recursion
void drawSnowflake ( int x, int y, float len, int depth ) {
  pushMatrix();

  // set up turtle initial position and orientation -
  //  turtle should start at the upper left corner of the
  //  initial triangle, facing right
  translate(x, y);
  rotate(radians(0));  // turtle is already facing right

  // generator: F++F++F++
  drawF(depth, len);
  rotate(radians(60));
  rotate(radians(60));
  drawF(depth, len);
  rotate(radians(60));
  rotate(radians(60));
  drawF(depth, len);
  rotate(radians(60));
  rotate(radians(60));

  popMatrix();
}
```



- create a drawing function for the whole pattern
- parameters for position, size, and maximum depth

```
void drawFractal ( int x, int y, float len, int depth ) {
  pushMatrix();

  // set up turtle initial position and orientation
  translate(x,y);
  rotate(radians(…));

  carry out the generator: …

  popMatrix();
}
```

turtle starts at (0,0), facing right
- translate(*dx*,*dy*) to move turtle by *dx*,*dy*
- rotate(*a*) to turn turtle by angle *a*

note: write translate step before rotate step

- reference to a production symbol → call to that function (with max depth and overall size)
- +, - → rotate(*a*) or rotate(-*a*) (angle *a* depends on the fractal)
- [,] → pushMatrix(), popMatrix()

CPSC 120: Principles of Computer Science • F

- call the whole-fractal drawing function to actually draw the fractal

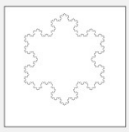
```
void setup () {
  …
}

void draw () {
  …
  drawFractal(…);
  …
}
```

```
void setup () {
  size(600, 600);
}

void draw () {
  background(255);

  drawSnowflake(100, 150, 400, 4);
}
```



At the End of Class

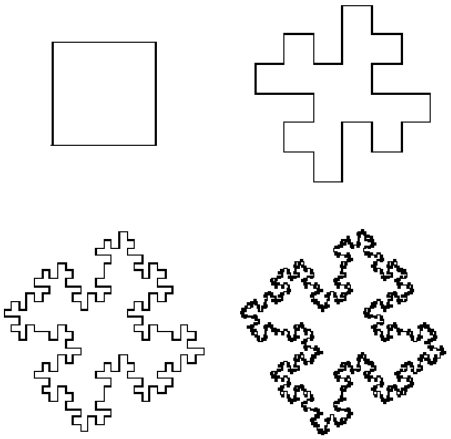
Hand in whatever you have done during class, even if a sketch is incomplete.

- Make sure each sketch is named as directed and has a comment with the names of your group. Also be sure to save your sketches! (in Linux, this should be in your sketchbook **~/cs120/sketchbook**)
- Copy the entire directory for each sketch (not only the .pde file) into your handin directory (**/classes/cs120/handin/*username***). You only need to hand in one copy for the group. (If you are running Processing on your computer instead of using the Linux virtual desktop, you will need to use FileZilla to copy the sketches.)

Exercises

1. Create a new sketch called **sketch_241016a** which draws a quadratic Koch island. This fractal is generated as follows:

   - generator: F+F+F+F+
   - F → F+F-F-FF+F+F-F
   - angle: 90 degrees
   - scale factor: ¼
   - turtle starting point: upper left corner of the initial square facing right

   The results with several successive values of depth are shown.

   Choose a reasonable initial size and a not-too-large maximum depth. (Start with a depth of 3 or 4 and increase slowly from there as desired.) Position the fractal so that it fits nicely in the drawing window.

2. Continue to work on the in-class exercises from last Friday (additive and replacement pattern fractals).

If you have time, experiment a bit: save a copy of your sketch from #1 as **sketch_241016b** and –

- Add some color: set the stroke color to black before you start the generator and at the beginning of the production rule in `drawF`, then set it to red just before the last call to `drawF` in the production rule. What happens?
- Add randomness: instead of always using the same angle, use a random angle near that value (add or subtract a small random amount from the desired angle). What happens? (Use `random()` to generate random numbers and put `randomSeed()` in `draw()` to generate the same sequence of random numbers in each frame. See the Processing API: `https://processing.org/reference`)
- Add randomness: instead of always moving/drawing by the same amount when a line segment is drawn, add or subtract a small random amount from the desired length. What happens? (You'll need to use a local variable to temporarily store the random number so the line length and the amount moved are the same. Use the pattern shown below – this is the same pattern introduced for the local variables in parametric equations. Hint: you can avoid having to choose between adding or subtracting by generating a random number between a negative value and a positive value – replace *amt* in the example below with your desired value.)

```
{
   float amount = random(-amt,amt);
   line(0,0,len+amount,0);
   translate(len+amount,0);
}
```

- Try some different angles. (Start with a value not too different from the angle given in the exercise.) What happens?