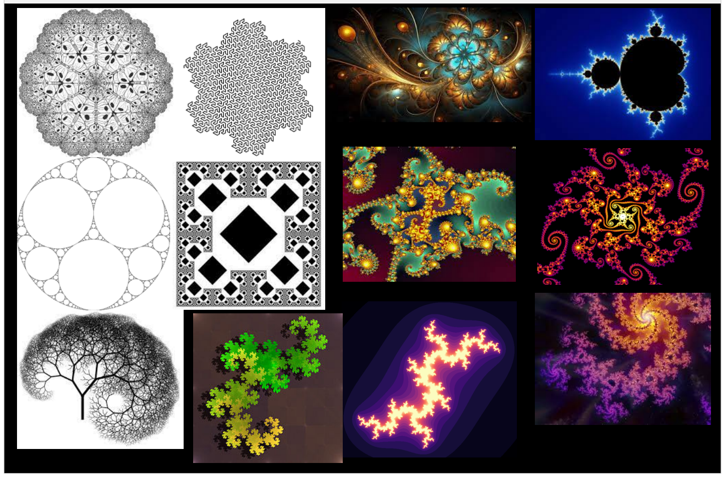
Self-Similarity and Fractals

Self-Similarity and Fractals

Self-similarity is a property where parts of an object resemble the whole – the structure looks similar to itself at different levels of magnification.

Fractals are a kind of object that exhibit self-similarity at all (or at least many) scales.

Fractals



Fractals and Self-Similarity in Nature

Credits

https://users.math.yale.edu/public_html/People/frame/Fractals/
https://mathinart.weebly.com/fractals.html
https://fractalsaco.weebly.com/fractal-tree.html
https://en.wikipedia.org/wiki/Gosper_curve
https://math.stackexchange.com/questions/1081922/self-similar-fractal-dimension-of-unsymmetrial-fractal
https://en.wikipedia.org/wiki/Dragon_curve
https://midlibrary.io/styles/fractal-art
https://people.math.rochester.edu/faculty/jnei/FRACTALS.html
https://matplotlib.org/matplotblog/posts/animated-fractals/
https://www.themarginalian.org/2021/02/22/mandelbrot-fractals-chaos/
https://www.scienceforums.net/topic/119915-fractals/
https://www.thewisemag.com/mystery/fractal-geometry/

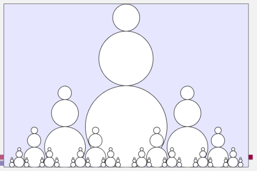
https://upload.wikimedia.org/wikipedia/commons/4/4f/Fractal_Broccoli.jpg
http://www.fernlifecenter.com/wp-content/uploads/2010/01/curled-fern-frond.jpg
http://www.asergeev.com/pictures/archives/2012/1082/jpeg/26.jpg
http://www.disassociated.com/images/posts/china_fractal.jpg
http://www.wired.com/wp-content/uploads/images_blogs/wiredscience/2010/09/fractal_12a.jpg
http://lariphotos.free.fr/googleearth%20art/rivertree01.jpg
http://www.dmc.gov.lk/hazard/hazard/Images/Lightning.jpg
http://4.bp.blogspot.com/-0q3bcXR2juk/UvVu04lWl_I/AAAAAAAAABI/XdmNTbJ_d9c/s1600/FractalClouds.jpg
http://gajitz.com/wp-content/uploads/2010/05/von-karman-vortex-street-clouds.jpg
http://www.wired.com/images_blogs/wiredscience/2010/09/fractal_11b.jpg
http://garutweb.com/wp-content/uploads/2015/04/cascade-mountain-range-silhouette-sobh1462.jpg
http://www.oddee.com/_media/imgs/articles/a302_f7.jpg
https://siwtk.files.wordpress.com/2013/01/leaf.jpg
https://radicalbotany.files.wordpress.com/2012/04/fractal-queen-annes-lace-torus.jpg
https://egregores.files.wordpress.com/2010/12/fractal-brain-2.jpg
https://egregores.files.wordpress.com/2011/03/lena_hires2.jpg
http://www.wired.com/wp-content/uploads/images_blogs/wiredscience/2010/09/fractal_13.jpg

Functions Calling Other Functions

Functions can call other functions –

- e.g. calling `size()`, `fill()`, `ellipse()` etc from `setup()` and `draw()`
- e.g. calling `drawSnowman()` from `draw()`
- e.g. calling `drawDrunkardUR()` from `drawDrunkardBlock()`
- e.g. calling `drawSnowman()` from `drawSnowman()`

Recursion – a function calling itself – is a technique for drawing self-similar patterns.



Recursive Drawing Functions

Drawing function questions –

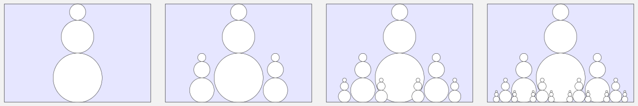
- What is being drawn?
  → function name (and comments)
- What differs from one copy to the next?
  → parameters
- How is it drawn?
  → function body

Recursive drawing function questions –

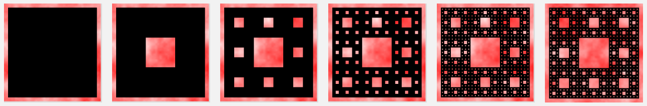
- What is *the whole pattern* being drawn? (not just one level)
  → function name (and comments)
- What differs from one *level* to the next?
  → parameters
- *Additive or replacement pattern?*
  → function body

Two Patterns for Recursive Designs

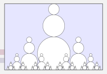
- additive
  - each level adds to what has already been drawn



- replacement
  - each level replaces what has already been drawn



Additive Pattern for Recursive Designs

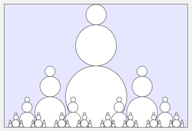


To draw the design, draw the shapes that make up one copy of the design, then draw the “smaller” copies.

- “smaller” generally means a decrease in size, but could also be a countdown or some other thing

```
if ( the design is small enough ) {
  return;                    // we're done (draw nothing)
} else {                     // draw the design
  draw the shapes for one copy of the design
  draw the smaller copies of the design  ◄──── recursive calls
}
```

Implementing the Additive Pattern



what is the whole design? a family of snowmen

what is one copy of the design? one snowman (at the top level there is one snowman)

what differs from one copy of the design to the next? position and size

```
// draw a line of snowmen
//  (x,y) is the center of the bottom circle
//  w is the width of the snowman
void drawSnowFamily ( int x, int y, int w ) {

  if ( w < 10 ) {   // "small enough" is less than 10 pixels
    return;

  } else {
    // draw one snowman
    ellipseMode(CENTER);
    fill(255);
    stroke(0);
    ellipse(x, y, w, w);
    ellipse(x, y-(w/2+w/3), 2*w/3, 2*w/3);
    ellipse(x, y-(w/2+2*w/3+w/6), w/3, w/3);

    // draw two smaller copies
    drawSnowFamily(x-(w/2+w/4), y+w/4, w/2);
    drawSnowFamily(x+(w/2+w/4), y+w/4, w/2);
  }
}
```

To draw the design, draw the shapes that make up one copy of the design, then draw the “smaller” copies.

- “smaller” generally means a decrease in size, but could also be a countdown or some other thing

```
if ( the design is small enough ) {
  return;                    // we're done (draw nothing)
} else {                     // draw the design
  draw the shapes for one copy of the design
  draw the smaller copies of the design  ◄──── recursive calls
}
```

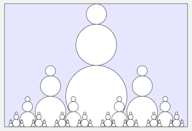
the design is small enough – based on the size of the snowman – w parameter

draw the shapes for one copy of the design – one copy is one snowman, with the position and size indicated by the parameters

draw the smaller copies – there are two smaller snowmen on either side of the one

Implementing the Additive Pattern

- call the recursive drawing function with the parameter values for the top level

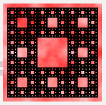


```
void draw () {
  background(230, 230, 255);

  drawSnowFamily(300, 300, 200);
}
```

```
// draw a line of snowmen
//  (x,y) is the center of the bottom circle
//  w is the width of the snowman
void drawSnowFamily ( int x, int y, int w ) {
```

Replacement Pattern for Recursive Designs

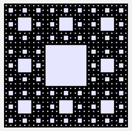


To draw the design, draw the “smaller” copies.

- “smaller” generally means a decrease in size, but could also be a countdown or some other thing

```
if ( the design is small enough ) {
  draw the base shape
} else {
  draw the smaller copies of the design  ◄──── recursive calls
}
```

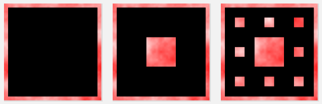
Implementing the Replacement Pattern



what is the whole design? Sierpinski carpet

what is the base shape? a black square

what differs from one copy of the base shape to the next? position and size



```
// draw a sierpinski carpet
//  (x,y) is the upper left corner of the carpet
//  w is the size of the carpet
void drawCarpet ( float x, float y, float w ) {

  if ( w < 5 ) {    // "small enough" is less than 5 pixels
    // a simple carpet is just a black rectangle
    fill(0);
    rect(x, y, w, w);

  } else {
    // draw a carpet in each of the 8 subregions around the edge
    //  (leave the middle region empty)
    drawCarpet(x, y, w/3);
    drawCarpet(x+w/3, y, w/3);
    drawCarpet(x+2*w/3, y, w/3);

    drawCarpet(x, y+w/3, w/3);
    drawCarpet(x+2*w/3, y+w/3, w/3);

    drawCarpet(x, y+2*w/3, w/3);
    drawCarpet(x+w/3, y+2*w/3, w/3);
    drawCarpet(x+2*w/3, y+2*w/3, w/3);
  }
}
```

To draw the design, draw the “smaller” copies.

- “smaller” generally means a decrease in size, but could also be a countdown or some other thing

```
if ( the design is small enough ) {
  draw the base shape
} else {
  draw the smaller copies of the design  ←—— recursive calls
}
```

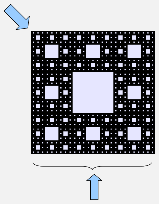
the design is small enough – based on the size of the base shape – w parameter

draw the base shape – one copy is one black square, with the position and size indicated by the parameters

draw the smaller copies – there are smaller copies in each of the 8 subregions around the edge

Implementing the Replacement Pattern

- call the recursive drawing function with the parameter values for the top level



```
void draw () {
  background(230, 230, 255);

  drawCarpet(0, 0, width);
}
```

```
// draw a sierpinski carpet
//  (x,y) is the upper left corner of the carpet
//  w is the size of the carpet
void drawCarpet ( float x, float y, float w ) {
```

Recursion

Important implementation notes –

- the “when the design is small enough” case is essential
  - called a *base case*
- the “draw smaller copies of the design” must be moving quantities closer to “small enough”
  - so the base case is reached

```
// draw a line of snowmen
//  (x,y) is the center of the bottom circle
//  w is the width of the snowman
void drawSnowFamily ( int x, int y, int w ) {

  if ( w < 10 ) {   // "small enough" is less than 10 pixels wide
    return;

  } else {
    // draw one snowman
    ellipseMode(CENTER);
    fill(255);
    stroke(0);
    ellipse(x, y, w, w);
    ellipse(x, y-(w/2+w/3), 2*w/3, 2*w/3);
    ellipse(x, y-(w/2+2*w/3+w/6), w/3, w/3);

    // draw two smaller copies
    drawSnowFamily(x-(w/2+w/4), y+w/4, w/2);
    drawSnowFamily(x+(w/2+w/4), y+w/4, w/2);
  }
}
```

```
// draw a sierpinski carpet
//  (x,y) is the upper left corner of the carpet
//  w is the size of the carpet
void drawCarpet ( float x, float y, float w ) {

  if ( w < 5 ) {    // "small enough" is less than 5 pixels
    // a simple carpet is just a black rectangle
    fill(0);
    rect(x, y, w, w);

  } else {
    // draw a carpet in each of the 8 subregions around the edge
    //  (leave the middle region empty)
    drawCarpet(x, y, w/3);
    drawCarpet(x+w/3, y, w/3);
    drawCarpet(x+2*w/3, y, w/3);

    drawCarpet(x, y+w/3, w/3);
    drawCarpet(x+2*w/3, y+w/3, w/3);

    drawCarpet(x, y+2*w/3, w/3);
    drawCarpet(x+w/3, y+2*w/3, w/3);
    drawCarpet(x+2*w/3, y+2*w/3, w/3);
  }
}
```