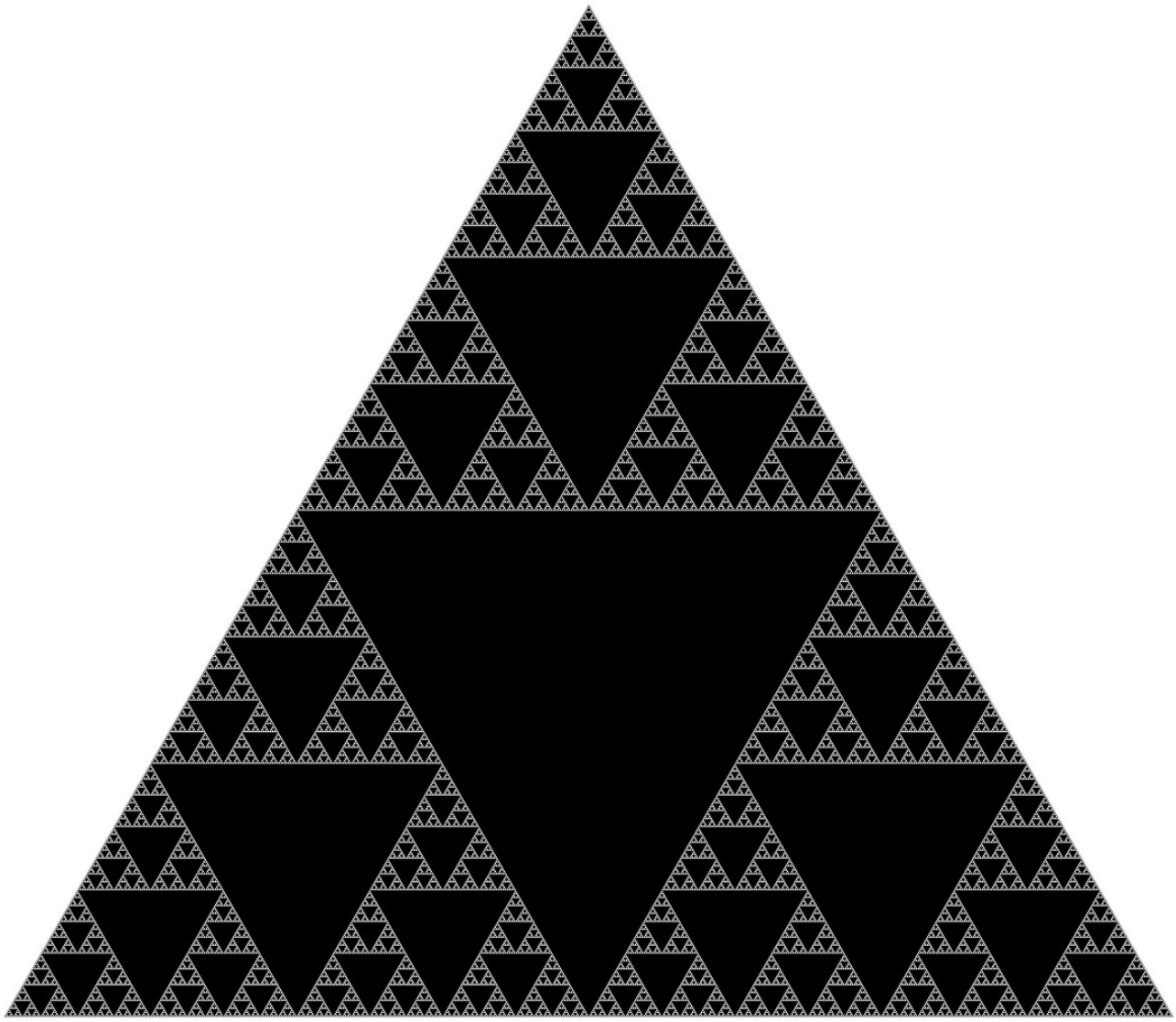
PS1 - Recursive Graphics (Sierpinski's Triangle)

This assignment involved implementing Sierpinski's triangle using recursion. Sierpinski's triangle is a fractal pattern described by a Polish mathematician named Wacław Sierpiński in 1915.



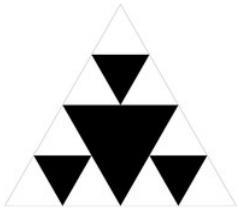
This program is done by drawing one one filled equilateral triangle (pointed upwards) and then another filled equilateral triangle (pointed downwards), which then calls itself recursively three times to create a triangle on the top, left and right of the original downwards triangle.

The executables takes in two arguments. The first argument is the number of recursions and the second argument is the initial window height. The length and width are equal and the fractal should correspond to the window size.

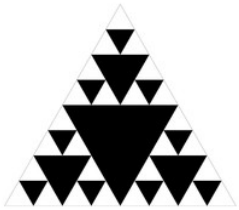
What I learned from this assignment was modular design with recursion; breaking up the programs into pieces and having them work together instead of having one large piece.

./sierpinski 1 200

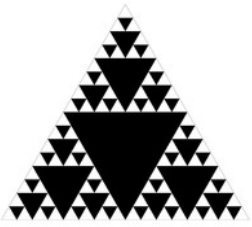
./sierpinski 2 200



./sierpinski 3 200

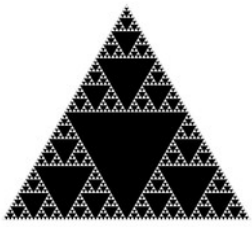


./sierpinski 4 200

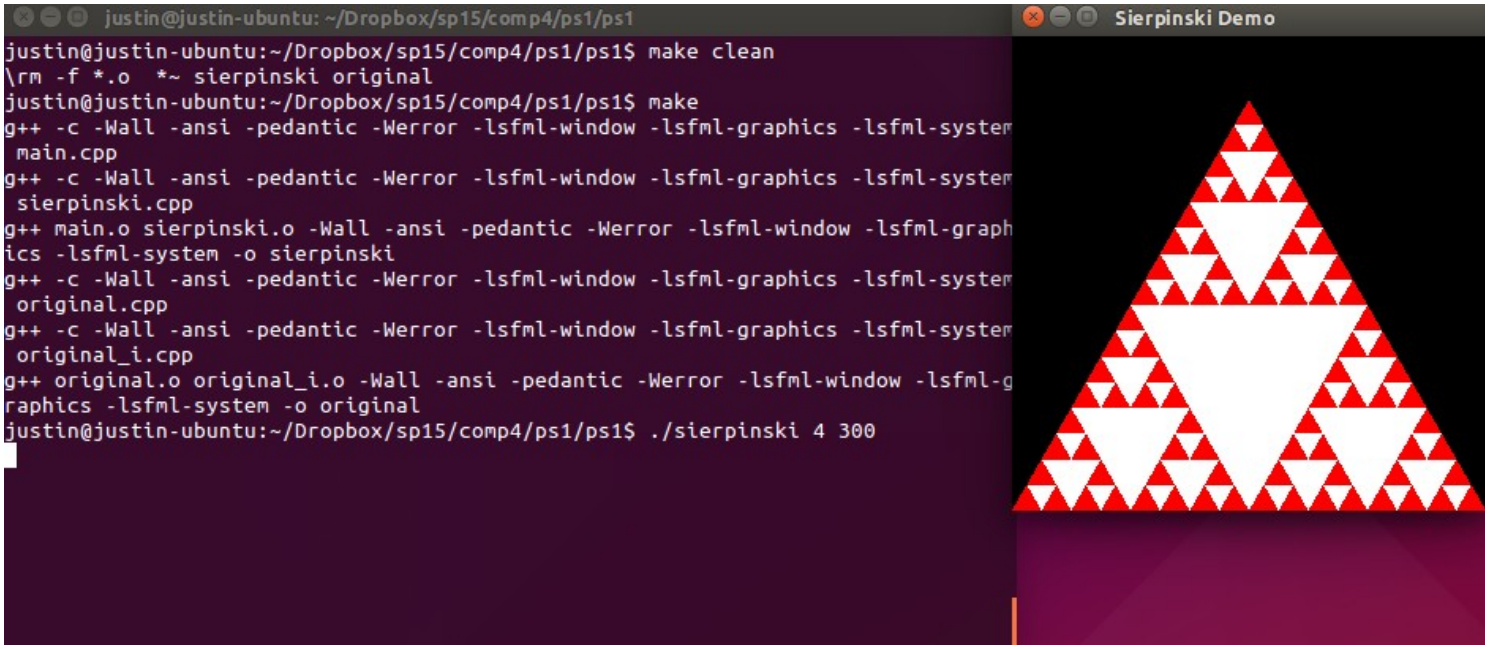


./sierpinski 5 200

./sierpinski 6 200

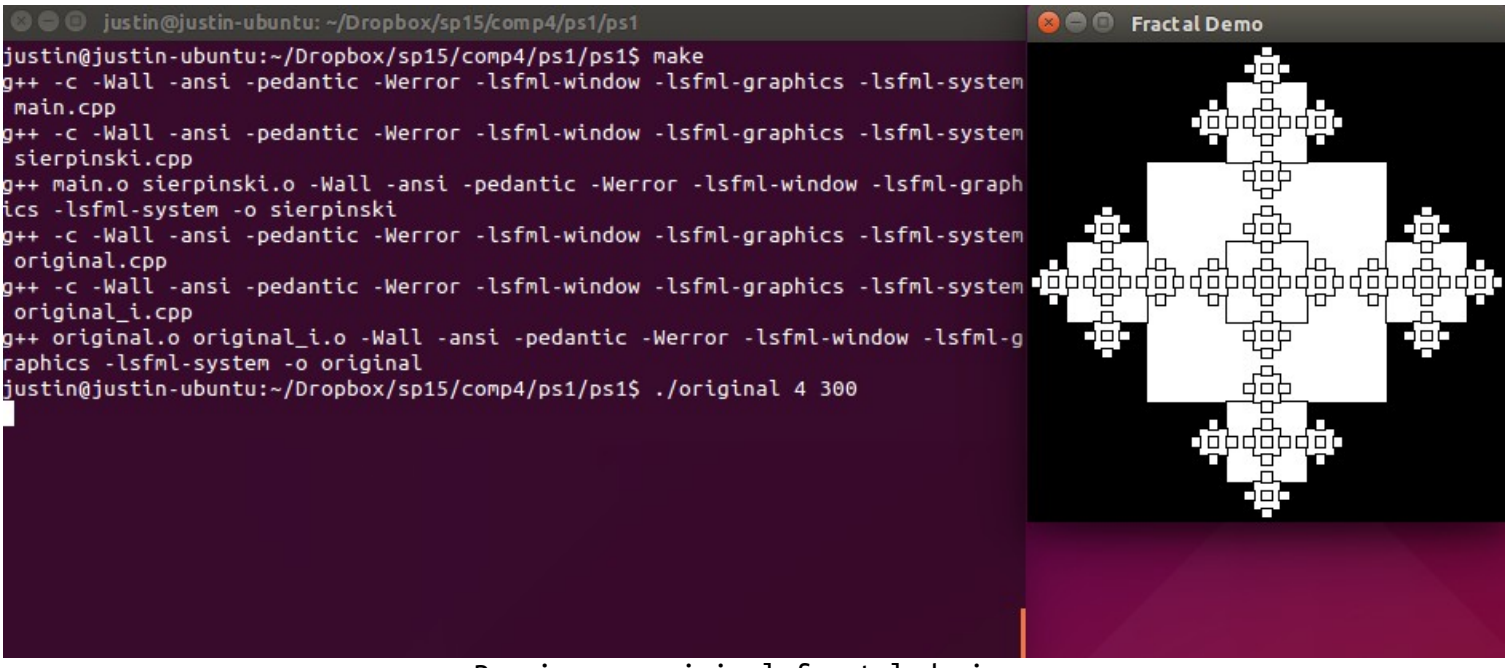


Examples of what Sierpinski should look like.



Running my implementation of Sierpinski's triangle.

After we implementing Sierpinski's triangle, we had to create our own fractal image. The idea I went for was a snowflake, so instead of triangles, I made squares and had them recursively draw on the top, bottom, left and right of the square.



Running my original fractal design.

```
 1: CC = g++
 2: OFLAGS = -c -Wall -ansi -pedantic -Werror
 3: CFLAGS = -Wall -ansi -pedantic -Werror
 4: LFLAGS = -lsfml-window -lsfml-graphics -lsfml-system
 5:
 6: all: ps1 ps1_original
 7:
 8: ps1: main.o sierpinski.o
 9:         $(CC) main.o sierpinski.o $(CFLAGS) $(LFLAGS) -o sierpinski
10:
11: ps1_original: original.o original_i.o
12:         $(CC) original.o original_i.o $(CFLAGS) $(LFLAGS) -o original
13:
14: main.o: main.cpp
15:         $(CC) $(OFLAGS) $(LFLAGS) main.cpp
16:
17: original.o: original.cpp
18:         $(CC) $(OFLAGS) $(LFLAGS) original.cpp
19:
20: sierpinski.o: sierpinski.cpp
21:         $(CC) $(OFLAGS) $(LFLAGS) sierpinski.cpp
22:
23: original_i.o: original_i.cpp
24:         $(CC) $(OFLAGS) $(LFLAGS) original_i.cpp
25:
26: clean:
27:         \rm -f *.o  *~ sierpinski original
```

```
 1: // Copyright [2015] Justin Nguyen
 2:
 3: #include "sierpinski.hpp"
 4: #include <iostream>
 5:
 6: int main(int argc, char* argv[])
 7: {
 8:   if (argc < 3) {
 9:     std::cout << "You need to enter a depth and window size.\n";
10:     exit(1);
11:   }
12:   sf::RenderWindow window(sf::VideoMode(atoi(argv[2]), atoi(argv[2])),
13:                           "Sierpinski Demo");
14:   Sierpinski sierpinski(atoi(argv[1]), atoi(argv[2]));
15:   while (window.isOpen()) {
16:     sf::Event event;
17:     while (window.pollEvent(event)) {
18:       if (event.type == sf::Event::Closed) {
19:         window.close();
20:         break;
21:       }
22:     }
23:     window.clear();
24:     window.draw(sierpinski);
25:     window.display();
26:   }
27:   return 0;
28: }
```

```
    1: // Copyright [2015] Justin Nguyen
    2:
    3: #include <SFML/Graphics.hpp>
    4: #include <SFML/Window.hpp>
    5: #include <cmath>
    6: #include "sierpinski.hpp"
    7:
    8: void Sierpinski::draw(sf::RenderTarget& target, sf::RenderStates states) con
st {
    9:   target.draw(base_triangle, states);
   10:   sierpinski(0.5 * size, 0.0, 0.5 * size, depth, target, states);
   11: }
   12:
   13: Sierpinski::Sierpinski(int N, int tri_size) {
   14:   depth = N;
   15:   size = tri_size;
   16:   length = tri_size;
   17:   // sets the base triangle to the size of the window
   18:   base_triangle.setPointCount(3);
   19:   base_triangle.setPoint(0, sf::Vector2f(0, length));
   20:   base_triangle.setPoint(1, sf::Vector2f(length, length));
   21:   base_triangle.setPoint(2, sf::Vector2f((length / 2),
   22:                                          (length - (sqrt(3) / 2) * length)))
;
   23:   base_triangle.setFillColor(sf::Color::Red);
   24: }
   25:
   26: Sierpinski::~Sierpinski() {
   27: }
   28:
   29: void Sierpinski::sierpinski(double x, double y, double s, int count,
   30:                             sf::RenderTarget& target,
   31:                             sf::RenderStates states) const {
   32:   sf::Vector2f set_next_points[3];
   33:   if (count == 0) {  // causes segmentation error without
   34:     return;
   35:   } else {
   36:     // sets coords
   37:     set_next_points[0].x = x;
   38:     set_next_points[0].y = y;
   39:     set_next_points[1].x = (x - (s / 2));
   40:     set_next_points[1].y = (y - (sqrt(3) / 2) * s);
   41:     set_next_points[2].x = (x + (s / 2));
   42:     set_next_points[2].y = (y - (sqrt(3) / 2) * s);
   43:     // makes triangle
   44:     filledTriangle(set_next_points, target, states);
   45:     // recursion
   46:     sierpinski(x, y + ( (sqrt(3) / 2) * (s)), (s / 2),
   47:                count - 1, target, states);
   48:     sierpinski(x - (s / 2), y, (s / 2), count - 1, target, states);
   49:     sierpinski(x + (s / 2), y, (s / 2), count - 1, target, states);
   50:   }
   51: }
   52:
   53: void Sierpinski::filledTriangle(sf::Vector2f set_point[3],
   54:                                 sf::RenderTarget& target,
   55:                                 sf::RenderStates states) const {
   56:   // creates a filled triangle
   57:   sf::ConvexShape triangle;
   58:   triangle.setPointCount(3);
   59:   triangle.setPoint(0, set_point[0]);
```

```
   60:   triangle.setPoint(1, set_point[1]);
   61:   triangle.setPoint(2, set_point[2]);
   62:   triangle.setFillColor(sf::Color::White);
   63:   triangle.setPosition(0, size - (2 * set_point[0].y));
   64:   target.draw(triangle, states);
   65: }
```

```
    1: // Copyright [2015] Justin Nguyen
    2:
    3: // include guard
    4: #ifndef SIERPINSKI_HPP_
    5: #define SIERPINSKI_HPP_
    6:
    7: // incluced dependencies
    8: #include <SFML/Graphics.hpp>
    9: #include <SFML/Window.hpp>
   10:
   11: class Sierpinski : public sf::Drawable {
   12:  public:
   13:   Sierpinski(int N, int tri_size);
   14:   ~Sierpinski();
   15:   void filledTriangle(sf::Vector2f set_point[3], sf::RenderTarget& target,
   16:                       sf::RenderStates states) const;
   17:   void sierpinski(double x, double y, double s, int count,
   18:                   sf::RenderTarget& target, sf::RenderStates states) const;
   19:
   20:  private:
   21:   sf::ConvexShape base_triangle;
   22:   int depth;
   23:   // 2 units so they don't change the value of each other
   24:   int size;
   25:   int length;
   26:
   27:   virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const
;
   28: };
   29:
   30: #endif
```

```
 1: // Copyright [2015] Justin Nguyen  [legal/copyright]
 2:
 3: #include "original.hpp"
 4: #include <iostream>
 5:
 6: int main(int argc, char* argv[]) {
 7:   if (argc < 3) {
 8:     std::cout << "You need to enter a depth and window size";
 9:     exit(1);
10:   }
11:   sf::RenderWindow window(sf::VideoMode(atoi(argv[2]), atoi(argv[2])),
12:                           "Fractal Demo");
13:   Frac frac(atoi(argv[1]), atoi(argv[2]));
14:   while (window.isOpen()) {
15:     sf::Event event;
16:     while (window.pollEvent(event)) {
17:       if (event.type == sf::Event::Closed) {
18:         window.close();
19:         break;
20:       }
21:     }
22:     window.clear();
23:     window.draw(frac);
24:     window.display();
25:   }
26:   return 0;
27: }
```

```
    1: // Copyright [2015] Justin Nguyen
    2:
    3: #include <SFML/Graphics.hpp>
    4: #include <SFML/Window.hpp>
    5: #include <cmath>
    6:
    7: #include "original.hpp"
    8:
    9: void Frac::draw(sf::RenderTarget& target, sf::RenderStates states) const {
   10:   target.draw(base_poly, states);
   11:   frac_re(size * 0.5, 0.5 * size, 0.25 * size, depth, target, states);
   12: }
   13: Frac::Frac(int N, int o_size) {
   14:   depth = N;
   15:   size = o_size;
   16:   length = o_size;
   17:   base_poly.setPointCount(4);
   18:   base_poly.setPoint(0, sf::Vector2f(length / 2, 0));
   19:   base_poly.setPoint(1, sf::Vector2f(0, length / 2));
   20:   base_poly.setPoint(2, sf::Vector2f(length / 2, (length)));
   21:   base_poly.setPoint(3, sf::Vector2f((length), length / 2));
   22:   base_poly.setFillColor(sf::Color::Black);
   23: }
   24:
   25: Frac::~Frac() {
   26: }
   27:
   28: void Frac::filledPoly(sf::Vector2f set_point[4], sf::RenderTarget& target,
   29:                       sf::RenderStates states) const {
   30:   sf::ConvexShape poly;
   31:   poly.setPointCount(4);
   32:   poly.setPoint(0, set_point[0]);
   33:   poly.setPoint(1, set_point[1]);
   34:   poly.setPoint(2, set_point[2]);
   35:   poly.setPoint(3, set_point[3]);
   36:   poly.setFillColor(sf::Color::White);
   37:   poly.setOutlineColor(sf::Color::Black);
   38:   poly.setOutlineThickness(1);
   39:   poly.setPosition(0, (length - (2 * set_point[0].y)));
   40:   target.draw(poly, states);
   41: }
   42:
   43: void Frac::frac_re(double x, double y, double s, int count,
   44:                    sf::RenderTarget& target, sf::RenderStates states) const
{
   45:   sf::Vector2f set_next_points[4];
   46:   if (count == 0) {
   47:     return;
   48:   } else {
   49:     // top right
   50:     set_next_points[0].x = (x + s);
   51:     set_next_points[0].y = (y + s);
   52:     // bot right
   53:     set_next_points[1].x = (x + s);
   54:     set_next_points[1].y = (y + (s * 3));
   55:     // bot left
   56:     set_next_points[2].x = (x - s);
   57:     set_next_points[2].y = (y + (s * 3));
   58:     // top left
   59:     set_next_points[3].x = (x - s);
   60:     set_next_points[3].y = (y + s);
```

```
   61:     filledPoly(set_next_points, target, states);
   62:     frac_re((x - ((s / 3) * 4)), y, (s / 3), count - 1, target, states);
   63:     frac_re(x, (y + ((s / 3) * 4)), (s / 3), count - 1, target, states);
   64:     frac_re((x + ((s / 3) * 4)), y, (s / 3), count - 1, target, states);
   65:     frac_re(x, ((y - ((s / 3) * 4))), (s / 3), count - 1, target, states);
   66:     frac_re(x, y, (s / 3), count - 1, target, states);
   67:   }
   68: }
```

```
    1: // Copyright [2015] Justin Nguyen  [legal/copyright]
    2:
    3: // include guard
    4: #ifndef FRAC_HPP_
    5: #define FRAC_HPP_
    6:
    7: // incluced dependencies
    8: #include <SFML/Graphics.hpp>
    9: #include <SFML/Window.hpp>
   10:
   11: class Frac : public sf::Drawable {
   12:  public:
   13:   Frac(int N, int o_size);
   14:   ~Frac();
   15:   void filledPoly(sf::Vector2f set_point[4], sf::RenderTarget& target,
   16:                   sf::RenderStates states) const;
   17:   void frac_re(double x, double y, double s, int count,
   18:                sf::RenderTarget& target, sf::RenderStates states) const;
   19:
   20:  private:
   21:   sf::ConvexShape base_poly;
   22:   int depth;
   23:   int size;
   24:   int length;
   25:   virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const
;
   26: };
   27:
   28: #endif
```