Sierpinski’s Gasket: Fractals in OpenGL

Jonathon Moore

CST-310

College of Engineering: Software Engineering

Professor Citro

Description

1. A description of the project

This project demonstrates the power of OpenGL in 2D and 3D environments. Specifically the ability to create infinite and complex shapes with only primitive objects. These being triangles and their related counterparts. We will be going over three different fractals which are infinite recursive objects with repeating patterns. The first is a classic Sierpinski Triangle in 2D, where a triangle is split into three triangles, and then each is split into three triangles, and so on. The second is a Sierpinski triangle in 3D space where it can be rotated and is made of consecutive tetrahedrons and each is split into four respective tetrahedrons and so on. The last is a 2D implementation of Koch’s Snowflake which uses a beginning triangle and each consecutive iteration rotates the lines producing the triangle at a certain angle and so on. From these we will gain a basic understanding of objects in OpenGL and how they live in their 2D and 3D space within the application.

1. A description of the methodology/approach used

All three of the fractals are implemented using a recursive approach. For the 2D Triangle it begins with a triangle with three points and from there is split into three triangles, each with an endpoint halfway between each side of the original triangle. From there if there are more iterations then the generated triangles are again split into three smaller triangles, with an endpoint created halfway between the three triangles sides and used for the new triangles. For the 3D Triangle it is very similar, however as a tetrahedron has four sides each side is split into its respective halfway point and is used to create four new tetrahedrons, in this case each side is a different color to differentiate which side you are looking at as it is rotated. One thing to note is here we are using the depth buffer which is necessary to only display the closest object to the user and avoid unwanted overlap displayed to the user. For the Snowflake you begin with an equilateral triangle that is created using three line segments. From there we divide the line segment into three equal parts, and the, using the middle section as its base, create an equilateral triangle from it. Then this is processed recursively to the desired depth.

1. The algorithm for drawing the gasket, including a flowchart

2D Triangle:

A screenshot of a computer screen

Description automatically generated

1. You first have an initial triangle with vertices *A1, A2, A3.*
2. Calculate the midpoints *B1, B2, B3* such that *B1* is the midpoint of *A2A3, B2* is the midpoint of *A1A3*, and *B3* is the midpoint of *A1A2*.
3. Then remove the triangle formed by *B1, B2, B3* from the original triangle by creating new triangles *B2B1B3, B3A2B1, A1B3B2.*
4. Repeat this process until desired depth.

*A red triangle with a white background

Description automatically generated with medium confidence*

3D Triangle:

A screenshot of a computer screen

Description automatically generated

1. The vertices of the initial tetrahedron will be *T1, T2, T3, T4*.
2. Calculate the midpoints *M(12), M(13), M(14), M(23), M(24), M(34)* for each of the six edges.
3. Remove the tetrahedron created by these midpoints by only drawing the respective tetrahedrons left from those points.
4. Repeat this process until desired depth.

2D Snowflake:

A screenshot of a computer screen

Description automatically generated

For each segment *L* with endpoints *(x1, y1)* and *(x2, y2)*:

1. Calculate points *P1* and *P2* that divide *L* into three equal segments.
2. Let *M* be the point that forms an equilateral triangle with *P1* and *P2* with its side length equal to |*P1P2*|.
3. Replace the middle third *P1P2* of *L* with two line segments *P1M* and *MP2*.
4. Repeat this process until desired depth.

A diagram of a process flow

Description automatically generated

1. Key code-related explanations or important and/or interesting things, accompanied by relevant code snippets

Some of the more important things of note here are with the 3D Triangle, as you need to use the Depth buffer to read only the nearest value. To do this you need to enable the buffer and in your code when writing the object you need to start and end the depth check:

A screenshot of a computer program

Description automatically generated

Also, with the 3D Triangle we are able to rotate the image by applying a transformation matrix. To do this we are using glRotatef(Angle, x-vector, y-vector, z-vector). With the Angle being the desired rotation between 1 and 360, and the vectors being how much it applies to that axis. For example, glRotatef(90, 0.0, 1.0, 0.0) would rotate the objects around the y-axis 90 degrees. This is it in our case for a more even rotation in all axis:



Note we are loading in the identity matrix first to apply fresh transformations.

One thing that I tried to do was to use the perspective view instead of the orthogonal view for the 3D Triangle. However I ran into an issue where I could not get the objects to move backwards, so I need to do more research on that no matter how far I moved the perspective away.

For keyboard inputs we are using ASCII values as well as the arrow keys. For the 3D Triangle the ASCII input method looks like:

A computer screen shot of a program code

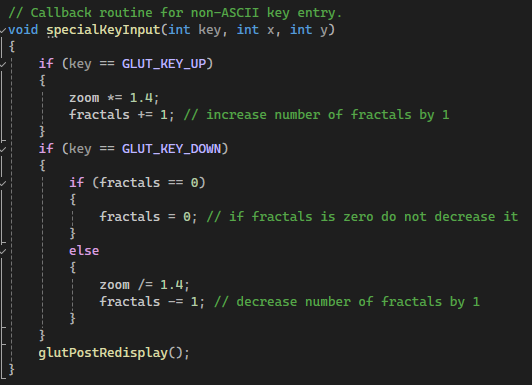
Description automatically generated

With ‘r’ resetting to the original display. Note that if you continuously add fractals it does become exponentially resource dependent so it will lag significantly, this is where the reset comes in handy.

An interesting thing implemented into the 2D Snowflake is a zoom transformation. Using the function glScalef(x-zoom, y-zoom, z-zoom). With the inputs being floats and acting as a percentage so glScalef(1.1, 1.1, 1.0) would zoom the x axis and y axis in by 10% and leave the z axis in place. This is how it was implemented in my code with the zoom variable being continuously increased when adding depth:



Note again loading in the identity matrix. Here is how the zoom variable is manipulated using non-ASCII keyboard inputs:



Note here that when we make a change with input we are always redisplaying the objects as to apply any changes.