**Reflection:**

**Module-2 Critical Thinking - The Sierpinski Gasket**

Alejandro Ricciardi

Colorado State University Global

CSC405: Graphics and Visualization

Professor: Dr Jennifer Marquez

August 25, 2024

**Reflection:**

**Module-2 Critical Thinking - The Sierpinski Gasket**

This reflection is part of Module 2 Critical Thinking – The Sierpinski Gasket from CSC405: Graphics and Visualization at Colorado State University Global. It provides an overview and reflection on the program's functionality, including testing scenarios and output screenshots. The program is titled "Sierpinski Gasket Vertex 2D" and it is based on WebGL and is coded using GLES 3, JavaScript, and HTML. It is a very simple WebGL application that generates and displays a 2D animation of the Sierpinski Gasket being rendered.

**The Assignment**

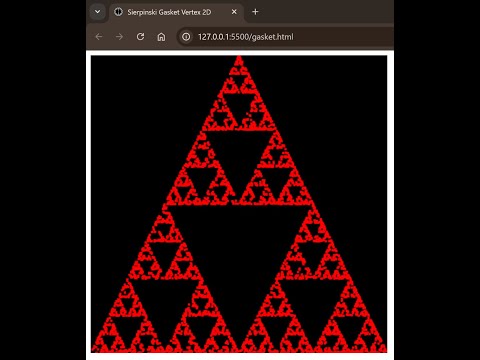
The assignment requires generating a simple Sierpinski Gasket using WebGL or OpenGL. The program needs to render the gasket utilizing a vertex shader and a fragment shader. The Sierpinski Gasket has a fractal geometry, which is an object that can be defined recursively and randomly (Angel & Shreiner, 2020). The geometry starts on a plan with three non-collinear points, which are the vertices of a triangle. To construct the gasket the following steps need to be followed: “Pick an initial point p = (𝑥, 𝑦, 0) at random inside the triangle. Select one of the three vertices at random. Find the point q halfway between p and the randomly selected vertex. Display q by putting some sort of marker, such as a small circle, at the corresponding location on the display. Replace p with q. Return to step 2 “(Angel & Shreiner, 2020, p. 41).

**Observations**

After creating the program following the assignment requirement, I observed the gasket being created almost instantly, the final geometrical shape consisted of smaller triangles recursively removed from the original larger triangle. Each smaller triangle’ sizes are proportional to the larger triangles, creating a recursive. Additionally, the triangles were composed of exactly 5,000 red vertices, which I specified in the JavaScript code. See Figure 1 for the final geometrical structure.

**Figure 1***Final Points Geometrical Shape*A screenshot of a computer screen

Description automatically generated

**Video 1***A 2D Animation of the Sierpinski Gasket* [](https://www.youtube.com/embed/qPzTHRYefV4?feature=oembed)

**Figure 2**

*A 2D Points Rendering of Sierpinski Gasket*

A screenshot of a computer

Description automatically generated

I was hoping to observe the rending of the vertexes generated by the following JavaScript code line which uses the interface gl (the OpenGL ES 3.0 rendering context) “gl.drawArrays(gl.POINTS, 0, positions.length);” within the render method (see source code). However, my PC rendered the geometry too quickly. To address this, I modified the render function by making it recursive and implementing a time delay (see source code). At first, the points appeared random, but as the process continued, the triangular geometry of the Sierpinski Gasket appeared, see Figure 2 and Video 1. The pattern had properties that were not at all random after all.

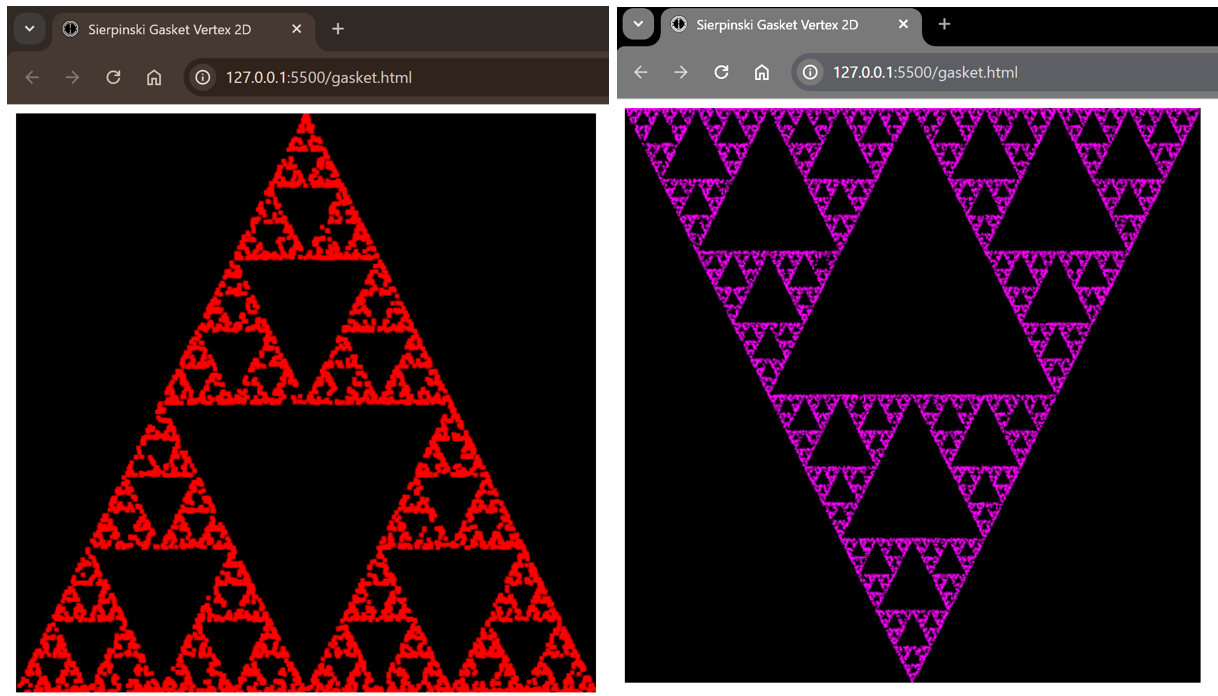
**Figure 3**

*Triangles Vertices*

Screens screenshot of a computer screen

Description automatically generated

**Figure 3**

*Points Vertices* 

**Primitive and Attributes**

In the gasket files, I used points to generate the graphic representation of the Sierpinski Gasket. In WebGL, points are primitive types; primitive types are rasterized into pixels in the framebuffer. I used the triangle primitive types to generate the graphic representation of the Sierpinski Gasket by dividing the triangle sides' length in half, see Figure 3. Additionally, I manipulated the attributes of both the points and triangles, see Figure 3 and Figure 4. Attributes are properties or characteristics that define how a primitive is rendered. In the vertex-shader, I changed the attribute value size of the points from 4 to 1 and performed reflection transformation by flipping the fractal along the x-axis position. I also changed the attribute color value of the points and triangles from red to magenta in the fragment shader. Additionally, in the JavaScript file, I changed the number of points from 5000 to 20,000.

The vertex-shader handles the vertex's transformation and processing. It transforms the vertex positions from the local coordinate space to the screen coordinate space by applying transformations such as translation, rotation, and scaling. On the other hand, the fragment shader operates on fragments, that is the pixels on the screen. It determines the final color of each pixel by processing the data passed from the vertex-shader, such as interpolated colors, textures, and lighting information. The shader GLES code is computed directly by the GPU, and the JavaScript code is processed by the CPU.

Changing the attributes significantly affects the appearance of the fractal by modifying the size and color attributes in both the vertex shader and the fragment shader directly impacting the visual outcome of the fractal. Changing the point sizes, numbers, and colors led to a finer, more detailed appearance of the fractal mostly against the black background. The triangle primitive led to a sharper and even more refined appearance of the fractal.

**Conclusion**

In summary, the project required implementing vertex and fragment shaders to render the Sierpinski Gasket. I explored changing attribute values like point sizes and color and their effects on the visual output of the fractal. I documented how changes, such as modifying the number of points and using triangle primitives, significantly influenced the fractal's appearance, resulting in a more detailed and refined final image.

**References**

Angel, E., & Shreiner, D. (2020). *Interactive computer graphics*. 8th edition. Pearson Education, Inc. ISBN: 9780135258262