ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA KHOA HỌC MÁY TÍNH

BÀI TIỂU LUẬN NHÓM
ĐỒ HOẠ MÁY TÍNH
TIỂU LUẬN VỀ FRACTAL

Giảng viên hướng dẫn: ThS. Cáp Phạm Đình Thăng

Nhóm: 01

Sinh viên thực hiện:

Phạm Phương Minh Trí	**21522708**
Nguyễn Hoàng Gia An	**22520021**
Huỳnh Thị Hải Châu	**22520148**
Hoàng Đức Chung	**22520161**

Lớp: Đồ hoạ máy tính - CS105.P21

TP. Hồ Chí Minh, 14 tháng 4 năm 2025

1. Giới thiệu chung về Fractal

1.1. Fractal là gì?

Fractal có thể được định nghĩa là một hình hình học phức tạp, sở hữu tính tự đồng dạng ở các tỷ lệ khác nhau. Điều này có nghĩa là, khi bạn phóng to một phần bất kỳ của Fractal, bạn sẽ thấy một cấu trúc tương tự như toàn bộ hình, hoặc ít nhất là có những đặc điểm lặp lại.Fractal có thể được định nghĩa là một hình hình học phức tạp, sở hữu tính tự đồng dạng ở các tỷ lệ khác nhau. Điều này có nghĩa là, khi bạn phóng to một phần bất kỳ của Fractal, bạn sẽ thấy một cấu trúc tương tự như toàn bộ hình, hoặc ít nhất là có những đặc điểm lặp lại.

1.2. Các đặc tính cơ bản của fractal

1.2.1. Tính tự đồng dạng (Self-similarity)

Fractal chứa các bản sao thu nhỏ của chính nó (tuyệt đối, thống kê, gần đúng), ví dụ như tam giác Sierpinski và cây Fractal.

Có nhiều loại tự đồng dạng khác nhau:

- **Tự đồng dạng tuyệt đối (Exact self-similarity):** Các phần nhỏ hoàn toàn giống với toàn bộ hình sau khi được thu nhỏ hoặc phóng to. Ví dụ: tam giác Sierpinski, nơi mỗi tam giác nhỏ hơn là một bản sao chính xác của tam giác lớn hơn.
- **Tự đồng dạng thống kê (Statistical self-similarity):** Các phần nhỏ có các thuộc tính thống kê tương tự như toàn bộ hình (ví dụ: độ gồ ghề, mật độ phân bố). Ví dụ: đường bờ biển.
- **Tự đồng dạng gần đúng (Approximate self-similarity):** Các phần nhỏ có hình dạng tương tự nhưng không hoàn toàn giống với toàn bộ hình. Nhiều cấu trúc tự nhiên như cây cối và đám mây thể hiện tính tự đồng dạng gần đúng.

1.2.2. Chiều fractal (Fractal dimension)

Là một số không nguyên, thể hiện mức độ phức tạp và khả năng **"lấp đầy không gian"** của Fractal, tính bằng công thức $D = \frac{log(N)}{log(1/r)}$. Ví dụ, đường cong Koch có chiều > 1, Sierpinski < 2.

1.2.3. Tính không khả vi (Non-differentiability)

Đường cong và bề mặt Fractal thường không có đạo hàm tại mọi điểm do cấu trúc phức tạp và chi tiết vô hạn (ví dụ: đường cong Koch).

1.3. Tại sao fractal quan trọng trong đồ họa máy tính? (Tạo hình ảnh tự nhiên phức tạp, nén ảnh, nghệ thuật…)

1.3.1. Tạo hình ảnh tự nhiên phức tạp

Thuật toán Fractal hiệu quả trong việc mô phỏng địa hình, mây, cây cối, lửa, nước và các kết cấu tự nhiên phức tạp với độ chân thực cao, giảm độ phức tạp so với phương pháp truyền thống.

1.3.2. Nén ảnh

Phương pháp nén ảnh dựa trên việc tìm kiếm và lưu trữ các phép biến đổi của các phần tự đồng dạng trong ảnh, có tiềm năng đạt tỷ lệ nén cao nhưng còn hạn chế.

1.3.3. Ứng dụng trong nghệ thuật và thiết kế

Fractal là nguồn cảm hứng và công cụ mạnh mẽ để tạo ra các tác phẩm nghệ thuật kỹ thuật số độc đáo, hình nền, họa tiết và các yếu tố thiết kế sáng tạo.

1.3.4. Các ứng dụng khác

Fractal còn được ứng dụng trong việc tạo hiệu ứng đặc biệt và mô phỏng các quá trình tự nhiên động.

1.4. Mục tiêu và cấu trúc của tiểu luận

- **Cung cấp kiến thức nền tảng:** Trình bày một cách hệ thống và dễ hiểu về khái niệm Fractal, bao gồm định nghĩa, nguồn gốc lịch sử với vai trò của Benoît Mandelbrot, và các đặc tính cơ bản như tính tự đồng dạng, chiều Fractal và tính không khả vi.
- **Làm nổi bật tầm quan trọng:** Khẳng định vai trò và giá trị của Fractal trong lĩnh vực đồ họa máy tính hiện đại.
- **Khảo sát các ứng dụng chính:** Phân tích và minh họa các ứng dụng tiêu biểu của Fractal trong đồ họa máy tính, đặc biệt tập trung vào việc tạo ra hình ảnh tự nhiên phức tạp, tiềm năng trong nén ảnh và vai trò trong nghệ thuật kỹ thuật số.

2. Giới thiệu về WebGL

2.1. WebGL là gì?

WebGL (Web Graphics Library) là Javascript API có thể sử dụng được trong HTML5, WebGL được code trong tag <canvas> của HTML 5, điều này cho phép trình duyệt có thể truy cập và sử dụng GPU để xuất ra các đồ họa. WebGL được hỗ trợ bởi đa số các trình duyệt hiện đại: Chrome, FireFox, IE, Opera...

2.2. Kiến trúc cơ bản

2.2.1. Vertex Shader

Vertex Shader là giai đoạn đầu tiên trong pipeline lập trình được. Nó hoạt động trên từng đỉnh (**vertex**) của mô hình 3D và thực hiện các chức năng chính như:

- Biến đổi tọa độ từ không gian mô hình sang không gian màn hình (sử dụng các ma trận biến đổi: Model, View, Projection).
- Tính toán thuộc tính đỉnh như: tọa độ, màu sắc, vector pháp tuyến, tọa độ texture,...
- Truyền dữ liệu từ đỉnh tới các giai đoạn sau (interpolated đến Fragment Shader).

Ví dụ: Một tam giác có 3 đỉnh sẽ được gọi Vertex Shader 3 lần, mỗi lần xử lý 1 đỉnh.

2.2.2. Fragment Shader

Fragment Shader hoạt động sau khi các đỉnh được nội suy và raster hóa thành các điểm ảnh (**fragments**). Nó quyết định màu sắc cuối cùng của từng pixel trên màn hình, với các chức năng:

- Tính toán màu sắc của pixel dựa trên ánh sáng, vật liệu, và texture.
- Áp dụng các hiệu ứng như bóng, trong suốt, phản xạ,...
- Có thể bị loại bỏ nếu pixel nằm ngoài vùng nhìn hoặc bị che khuất (depth test, stencil test).

Fragment Shader rất quan trọng trong việc tạo ra các hiệu ứng hình ảnh chân thực.

2.2.3. Buffers

Buffers là bộ nhớ được sử dụng để lưu trữ dữ liệu trong quá trình vẽ, đặc biệt là:

- **Vertex Buffer:** Lưu thông tin về các đỉnh (tọa độ, màu sắc, texture, v.v.).
- **Index Buffer:** Chỉ định cách các đỉnh kết nối với nhau thành hình khối (tam giác, đường thẳng).
- **Frame Buffer:** Lưu trữ ảnh đầu ra cuối cùng, bao gồm thông tin màu sắc và độ sâu.
- **Depth Buffer:** Lưu thông tin độ sâu để xác định pixel nào hiển thị ở phía trước.

2.2.4. Textures

Textures là các hình ảnh được "dán" lên bề mặt của mô hình 3D để tạo chi tiết. Trong pipeline:

- Vertex Shader gửi tọa độ texture đến Fragment Shader.
- Fragment Shader sử dụng tọa độ đó để tra cứu màu tại điểm tương ứng trong Texture Sampler.
- Texture có thể là ảnh màu, bản đồ phản xạ, bản đồ độ cao, v.v.

Texture giúp mô hình trở nên chân thực hơn mà không cần tăng số lượng đỉnh.

2.3. Tại sao sử dụng WebGL cho đồ họa fractal?

2.3.1. Tận dụng sức mạnh xử lý song song của GPU

- WebGL cho phép truy cập trực tiếp tới GPU thông qua JavaScript.
- Các thuật toán sinh fractal như Mandelbrot, Julia, hoặc các hệ sinh L-system, Koch, đảo Minkowski,... thường cần xử lý hàng triệu điểm ảnh hoặc đỉnh một cách song song.
- GPU được thiết kế cho các phép tính vector/matrix trên quy mô lớn, giúp tăng tốc đáng kể so với xử lý bằng CPU truyền thống.

  Ví dụ: Với WebGL, ta có thể tính toán và hiển thị bộ Mandelbrot ở độ phân giải 4K trong vài mili giây thay vì vài giây nếu dùng Canvas 2D hoặc CPU đơn thuần.

2.3.2. Tương tác thời gian thực (Real-time interaction)

- WebGL hỗ trợ rendering theo khung hình (frame-based), giúp cập nhật liên tục khi người dùng phóng to, xoay, di chuyển hoặc thay đổi tham số fractal.
- Kết hợp với các input như chuột, bàn phím hoặc cảm ứng, người dùng có thể **trực tiếp điều chỉnh tham số** và xem kết quả tức thì.

  Tính tương tác này đặc biệt quan trọng khi khám phá các fractal động, như Julia biến thiên theo thời gian hoặc Mandelbrot zoom sâu nhiều lớp.

2.3.3. Không cần plugin, chạy trực tiếp trên trình duyệt

- WebGL là một chuẩn **gốc của HTML5**, được hỗ trợ bởi tất cả các trình duyệt hiện đại (Chrome, Firefox, Edge, Safari,...).
- Không cần cài đặt plugin hay phần mềm bổ trợ, chỉ cần mở một trang web là có thể sử dụng – giúp **triển khai dễ dàng và phổ biến.**
- Tính chất này phù hợp với các ứng dụng học tập, minh hoạ trực quan, hoặc trình diễn nghệ thuật số.

2.3.4. Hỗ trợ shader linh hoạt

- Với WebGL, ta có thể viết **Fragment Shader** để thực hiện tính toán fractal trực tiếp trên mỗi pixel.
- Các hiệu ứng như gradient màu, phản chiếu, đổ bóng, hiệu ứng thời gian… đều có thể tích hợp trực tiếp trong shader.
- Điều này giúp fractal không chỉ chính xác về hình dạng mà còn **đẹp mắt và sống động** về mặt thị giác.

2.3.5. Khả năng mở rộng và tích hợp dễ dàng

- WebGL có thể kết hợp với thư viện đồ họa như **Three.js, regl, p5.js**, cho phép dễ dàng tích hợp fractal vào các cảnh 3D, game, hoặc ứng dụng tương tác lớn hơn.
- Có thể xuất fractal thành hình ảnh, video, hoặc tích hợp vào VR/AR.

3. Bông tuyết Von Koch (Koch Snowflake)

3.1. Lịch sử và Mô tả (Niels Fabian Helge von Koch)

Bông tuyết Von Koch được xây dựng bằng cách đệ quy chia nhỏ các đoạn thẳng, tạo ra hình dạng như bông tuyết với các mép răng cưa càng lúc càng phức tạp.

3.2. Quy tắc tạo hình (L-System hoặc phương pháp đệ quy):

- Ở bước một, chia mỗi cạnh của tam giác thành ba đoạn bằng nhau, dựng tam giác đều trên đoạn ở giữa (ở bên ngoài tam giác đã cho) rồi xóa cạnh đáy của tam giác đều này thì được một đường gấp khúc kín.
- Ở mỗi bước tiếp theo, chia mỗi đoạn của đường gấp khúc kín thành ba đoạn con bằng nhau, dựng tam giác đều trên đoạn con ở giữa (ở bên ngoài đường gấp khúc kín đó) rồi xóa cạnh đáy.
- Cứ lặp lại các bước đó nhiều lần ta sẽ tạo ra được "Bông tuyết Von Koch".

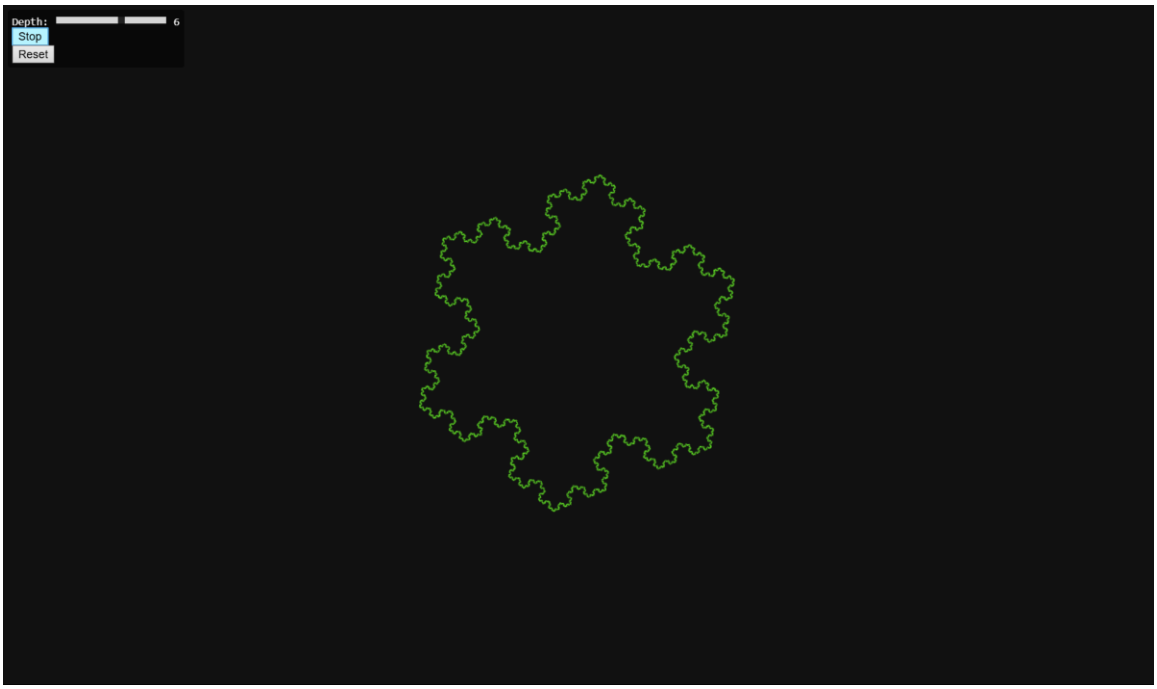
3.3. Tính chất hình học:

- **Chiều dài tăng theo cấp số nhân:** Với mỗi bước lặp, số cạnh tăng gấp 4 lần, mỗi cạnh bằng 1/3 độ dài lần trước.
- **Diện tích hữu hạn, nhưng chu vi tiến tới vô hạn.**
- **Dimension fractal là** $\frac{log(4)}{log(3)} \approx 1.2619$.

3.4. Triển khai bằng WebGL:

- Cách lưu trữ và cập nhật danh sách các đỉnh qua mỗi bước lặp.
- Sử dụng Vertex Buffer Object (VBO) để gửi dữ liệu đỉnh lên GPU.
- Vertex Shader và Fragment Shader cơ bản (ví dụ: vẽ đường màu trắng/đen).
- Sử dụng gl.LINE_STRIP hoặc gl.LINES để vẽ.
- **(Nâng cao)** Cho phép người dùng điều chỉnh số lần lặp (recursion depth), chạy ảnh động cho hình xoay vòng và đổi màu liên tục.

3.5. Hình ảnh kết quả



4. Đảo Minkowski

4.1. Mô tả:

Đảo Minkowski là một fractal được xây dựng bằng cách lặp đi lặp lại việc thay thế các cạnh của một hình vuông bằng các chuỗi đoạn thẳng gấp khúc, tạo thành những "răng cưa" nhỏ. Bắt đầu từ một hình vuông ban đầu, qua mỗi bước lặp, biên của hình vuông trở nên phức tạp hơn, tạo ra hiệu ứng như một hòn đảo có đường bờ biển gồ ghề, nhiều chi tiết.

4.2. Quy tắc tạo hình:

- Ở bước đầu tiên, bắt đầu từ một hình vuông. Mỗi cạnh của hình vuông được thay thế bằng một đoạn gấp khúc gồm 8 đoạn thẳng nhỏ, được tạo ra bằng cách chia cạnh đó thành bốn phần bằng nhau. Trên đoạn thứ hai (từ bốn đoạn đã chia), dựng một răng cưa hình chữ nhật gồm ba đoạn thẳng vuông góc lần lượt theo hướng trái, phải, phải để tạo hình gập góc nhô ra bên ngoài hình vuông ban đầu. Khi đó, toàn bộ cạnh ban đầu sẽ được thay thế bởi một chuỗi gấp khúc gồm tám đoạn thẳng.
- Ở mỗi bước tiếp theo, lặp lại quy trình trên cho **mọi đoạn thẳng** của đường gấp khúc hiện tại: chia mỗi đoạn thành bốn phần bằng

nhau, thay phần thứ hai bằng một răng cưa chữ nhật gồm ba đoạn vuông góc, và nối lại thành tám đoạn mới.

- Cứ lặp lại các bước đó nhiều lần, ta sẽ tạo ra được hình "**Đảo Minkowski**" với đường viền gồ ghề ngày càng chi tiết như bờ biển.

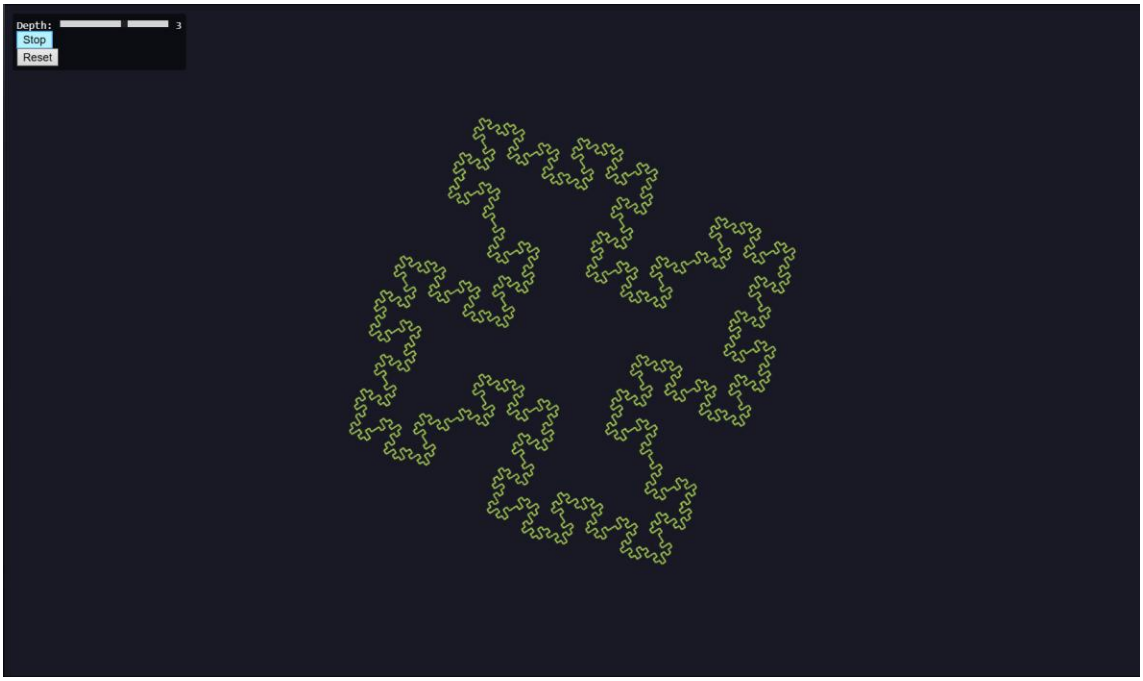
4.3. Tính chất hình học:

- **Chu vi tăng theo cấp số nhân:** Mỗi cạnh ban đầu tạo thành 8 đoạn, mỗi đoạn có độ dài bằng 1/4 → tổng chiều dài cạnh gấp đôi sau mỗi bước.
- **Diện tích hữu hạn:** Mặc dù chu vi tăng mãi, hình không vượt quá phạm vi bao quanh ban đầu.
- **Dimension fractal là** $\frac{log(8)}{log(4)} = \frac{3}{2} = 1.5$.

4.4. Triển khai bằng WebGL:

- Tương tự Koch Snowflake (lưu trữ đỉnh, VBO, shaders, gl.LINE_STRIP/gl.LINES).
- Xử lý các góc và hướng của quy tắc sinh phức tạp hơn Koch.
- **(Nâng cao)** Cho phép người dùng điều chỉnh số lần lặp (recursion depth), chạy ảnh động cho hình xoay vòng và đổi màu liên tục.

4.5. Hình ảnh kết quả



5. Tam giác Sierpinski (Sierpinski Triangle)

5.1. Lịch sử và Mô tả (Wacław Sierpiński)

Tam giác Sierpinski là một trong những hình dạng fractal cổ điển và nổi tiếng nhất trong toán học. Nó được đặt theo tên của nhà toán học người Ba Lan **Wacław Sierpiński**, người đã chính thức mô tả hình dạng này vào năm 1915. Tuy nhiên, các dạng hình tương tự đã từng xuất hiện sớm hơn trong nghệ thuật và hoa văn trang trí châu Á từ nhiều thế kỷ trước.

Về mặt hình học, tam giác Sierpinski là một tập hợp **tự đồng dạng** (self-similar set), nghĩa là toàn bộ hình có thể được chia thành các bản sao nhỏ hơn của chính nó. Mỗi bản sao có cùng hình dạng với toàn thể, chỉ khác về kích thước.

5.2. Quy tắc tạo hình (Đệ quy tam giác):

- Ở bước đầu tiên, bắt đầu từ một **tam giác đều**. Chia tam giác này thành bốn tam giác nhỏ bằng cách nối trung điểm của ba cạnh, rồi **loại bỏ tam giác nhỏ ở giữa** (tạo thành một hình tam giác rỗng ở tâm).
- Ở mỗi bước tiếp theo, thực hiện **quy trình tương tự** cho **mọi tam giác còn lại**: chia tiếp mỗi tam giác thành bốn tam giác con và loại bỏ tam giác trung tâm.
- Cứ lặp lại các bước đó nhiều lần, ta sẽ tạo ra được "**Tam giác Sierpinski**" – một mô hình tam giác đệ quy với cấu trúc rỗng ngày càng phức tạp.

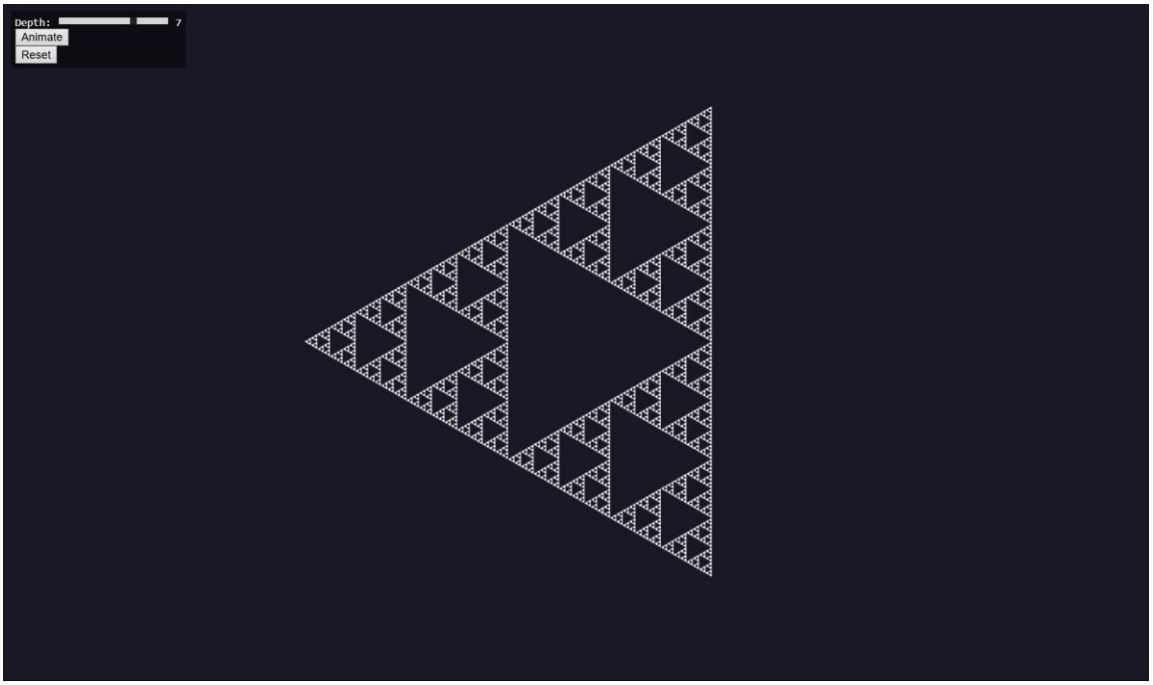
5.3. Tính chất hình học:

- **Chu vi tăng theo cấp số nhân:** Mỗi bước chia một tam giác thành 3 tam giác con mới (bỏ phần giữa), tạo ra thêm các cạnh nhỏ hơn. Dù mỗi cạnh ngắn đi, tổng chu vi tăng sau mỗi bước.
- **Diện tích hữu hạn:** Ở mỗi bước, ta bỏ đi 1/4 diện tích hình tam giác ban đầu. Sau vô hạn bước, diện tích tiến dần về 0, dù hình vẫn còn vô số điểm.
- **Fractal dimension là** $D = \frac{\log(3)}{\log(2)} \approx 1.58496$. Giá trị này nằm giữa đường thẳng (1D) và mặt phẳng (2D).

5.4. Triển khai bằng WebGL:

- Với phương pháp Đệ quy: Sinh ra rất nhiều đỉnh của các tam giác nhỏ. Dùng gl.TRIANGLES để vẽ. Cần quản lý số lượng đỉnh lớn.
- Với Chaos Game: Sinh ra các điểm. Dùng gl.POINTS để vẽ. Hiệu quả hơn về bộ nhớ đỉnh.
- Thiết lập VBO, shaders (có thể tô màu các điểm/tam giác).
- **(Nâng cao)** Cho phép người dùng điều chỉnh số lần lặp (recursion depth), chạy ảnh động cho hình xoay vòng và đổi màu liên tục.

5.5. Hình ảnh kết quả



6. Thảm Sierpinski (Sierpinski Carpet)

6.1. Mô tả:

Thảm Sierpinski là một dạng fractal hai chiều được xây dựng từ quy trình đệ quy loại bỏ các phần tử trung tâm trong một lưới vuông. Đây là phần mở rộng tự nhiên của **tam giác Sierpinski** sang không gian hai chiều dạng hình vuông, lần đầu tiên được mô tả bởi **Wacław Sierpiński** vào năm **1916** – chỉ một năm sau khi ông công bố tam giác mang tên mình.

Thảm Sierpinski là một hình vuông được chia lặp đi lặp lại thành lưới 3×3, rồi liên tục loại bỏ ô vuông trung tâm ở mỗi bước. Kết quả tạo

thành một hoa văn fractal có vô số lỗ rỗng, với hình dạng ngày càng phức tạp và tự đồng dạng ở mọi cấp độ thu phóng.

6.2. Quy tắc tạo hình:

- Ở bước đầu tiên, bắt đầu từ một **hình vuông**. Chia hình vuông thành **9 ô vuông nhỏ (3×3)**, sau đó **loại bỏ ô vuông ở giữa**.
- Ở mỗi bước tiếp theo, thực hiện **quy trình chia và xóa trung tâm** tương tự cho **mọi ô vuông còn lại**.
- Cứ lặp lại các bước đó nhiều lần, ta sẽ tạo ra "**Thảm Sierpinski**" – một dạng lưới vuông đệ quy với phần trung tâm ngày càng rỗng hơn.

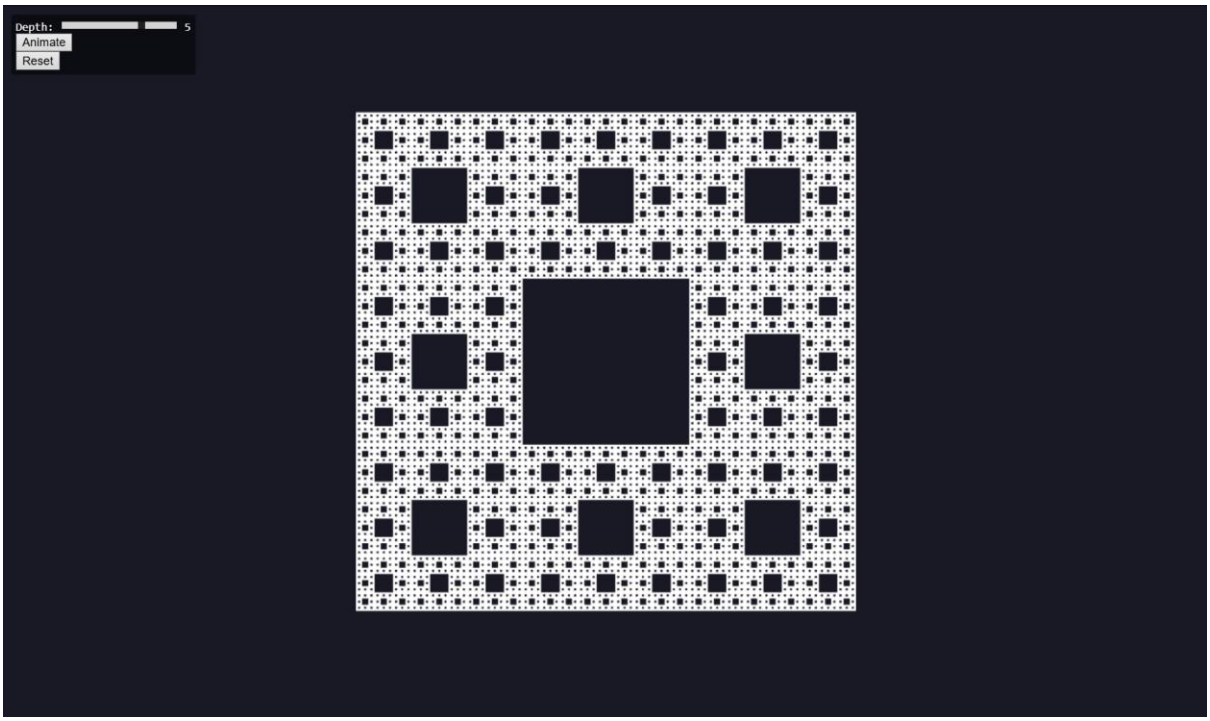
6.3. Tính chất hình học:

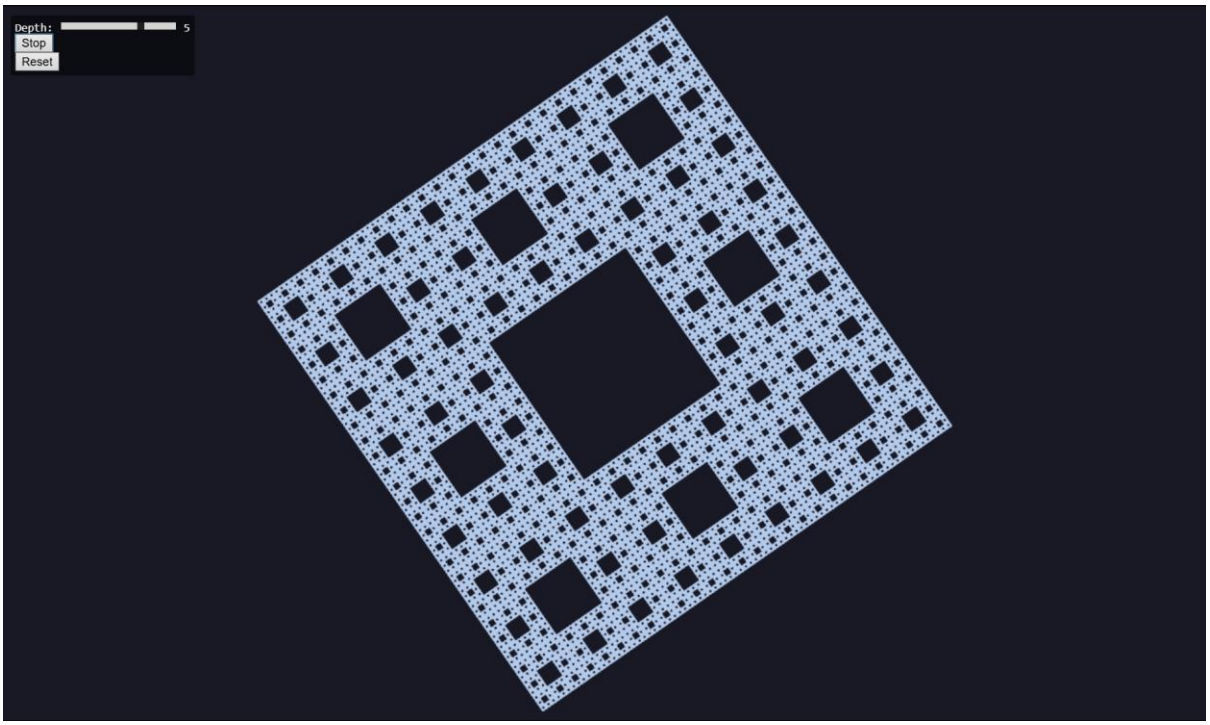
- **Chu vi tăng theo cấp số nhân:** Mỗi bước chia một hình vuông thành 8 ô vuông con (loại bỏ ô ở giữa), làm tăng tổng chu vi theo lũy thừa của 8, trong khi độ dài mỗi đoạn giảm theo hệ số 1/3.
- **Diện tích hữu hạn:** Ở mỗi bước lặp, 1/9 diện tích bị loại bỏ. Sau vô hạn bước, tổng diện tích tiến dần về **0** nhưng không bao giờ bằng 0 hoàn toàn.
- **Fractal dimension là** $D = \frac{\log(8)}{\log(3)} \approx 1.8928$. Tức là có độ phức tạp hình học cao hơn tam giác Sierpinski, gần hơn với mặt phẳng 2D.

6.4. Triển khai bằng WebGL:

- Sinh đỉnh cho tất cả các hình vuông nhỏ cần vẽ ở mỗi cấp độ đệ quy.
- Sử dụng gl.TRIANGLES (mỗi hình vuông vẽ bằng 2 tam giác) hoặc gl.POINTS (nếu xấp xỉ bằng điểm ảnh).
- Quản lý VBO, shaders.
- **(Nâng cao)** Cho phép người dùng điều chỉnh số lần lặp (recursion depth), chạy ảnh động cho hình xoay vòng và đổi màu liên tục.

6.5. Hình ảnh kết quả





7. Tập Mandelbrot (Mandelbrot Set)

7.1. Lịch sử và Mô tả (Benoît Mandelbrot)

Tập Mandelbrot là một trong những biểu tượng nổi bật và gây ấn tượng mạnh mẽ nhất trong lĩnh vực hình học fractal. Hình dạng này được đặt theo tên của nhà toán học gốc Pháp gốc Ba Lan **Benoît B. Mandelbrot**, người đã nghiên cứu sâu rộng về hình học phân mảnh và công bố rộng rãi tập hợp này vào những năm 1980.

Khi vẽ trên máy tính, quá trình kiểm tra phân kỳ được thực hiện trong một số bước lặp hữu hạn (thường từ 100 đến 1000 lần), và mỗi điểm được tô màu tùy theo tốc độ phân kỳ. Kết quả tạo nên một hình ảnh fractal tuyệt đẹp với các hoa văn vô hạn chi tiết, có **biên giới phức tạp và tự đồng dạng** ở nhiều cấp độ phóng đại.

7.2. Định nghĩa toán học:

- Dãy lặp: $z_{n+1} = z_n^2 + c$, với $z_0 = 0$ và c là một số phức.
- Tập Mandelbrot là tập hợp tất cả các điểm c trong mặt phẳng phức sao cho dãy z_n không tiến ra vô cùng (bounded).
- Điều kiện thoát (Escape Condition): Nếu $|z_n| > 2$ tại một bước lặp nào đó, thì dãy chắc chắn sẽ tiến ra vô cùng.

7.3. Thuật toán sinh Tập Mandelbrot (Escape Time Algorithm):

- Ánh xạ mỗi pixel trên màn hình (hoặc vùng vẽ) tới một điểm c trong mặt phẳng phức.
- Với mỗi c, lặp công thức $z_{n+1} = z_n^2 + c$ từ $z_0 = 0$ một số lần tối đa (max_iterations).
- Kiểm tra điều kiện thoát $|z_n| > 2$ ở mỗi bước.
- Nếu thoát, tô màu pixel dựa trên số bước lặp đã thực hiện trước khi thoát.
- Nếu không thoát sau max_iterations, điểm c thuộc tập Mandelbrot (thường tô màu đen).

7.4. Tính chất hình học:

- **Biên phức tạp vô hạn:** Ranh giới của Mandelbrot Set cực kỳ phức tạp và có tính chất **tự đồng dạng** ở mọi cấp độ phóng đại (zoom). Dù bạn phóng to bao nhiêu, vẫn luôn xuất hiện các chi

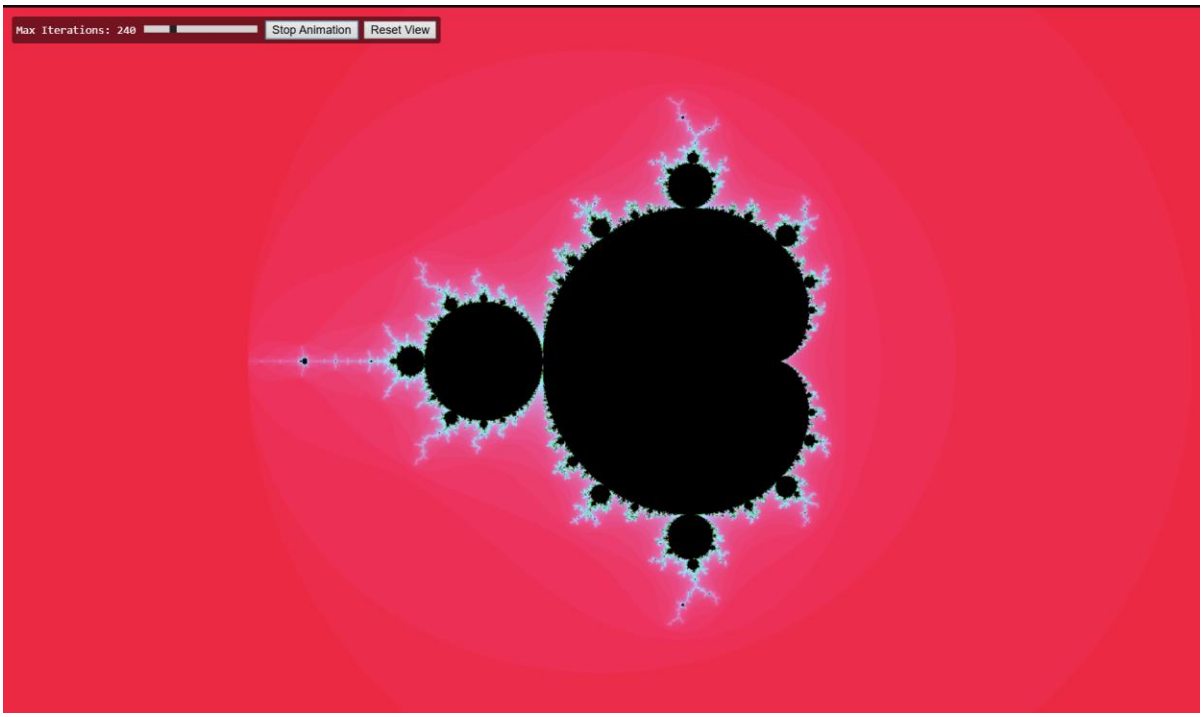
tiết mới – điều này phản ánh bản chất **vô hạn** của fractal.

- **Không liên thông đơn giản:** Tập Mandelbrot là **liên thông** (connected set), nhưng biên của nó không phải là một đường trơn tru. Ngược lại, nó **có cấu trúc gập ghềnh vô hạn**, xen kẽ các "bọ" nhỏ (minibrots) tái hiện toàn bộ tập chính ở kích thước nhỏ hơn.
- **Diện tích hữu hạn:** Tập Mandelbrot nằm gọn trong đĩa phức có bán kính 2 (tức là mọi điểm ccc trong tập đều thỏa ∣c∣≤2|c| \le 2∣c∣≤2), nên diện tích tổng thể là **hữu hạn**, dù ranh giới thì vô hạn độ chi tiết.
- **Fractal dimension (ước tính) là** $D \approx 2$.
  Ranh giới Mandelbrot có **chiều phân dạng gần bằng 2**, tức là mặc dù không phủ kín mặt phẳng, nó gần như làm điều đó ở phần rìa.

7.5. Triển khai bằng WebGL:

- Cách tiếp cận hiệu quả: Tính toán thực hiện trong Fragment Shader.
- Vertex Shader: Rất đơn giản, chỉ cần truyền tọa độ (ví dụ: vẽ một hình chữ nhật lớn phủ toàn bộ canvas). Tọa độ này (thường từ -1 đến 1) sẽ được dùng trong fragment shader.
- Fragment Shader:
  + Nhận tọa độ từ vertex shader (varying variable).
  + Ánh xạ tọa độ này thành số phức c.
  + Thực hiện vòng lặp Escape Time Algorithm.
  + Xác định màu sắc dựa trên số lần lặp trước khi thoát (hoặc màu đen nếu không thoát).
  + Gán màu cuối cùng cho gl_FragColor.
- Uniform Variables: Sử dụng uniform để truyền các tham số như max_iterations, tâm của vùng nhìn, mức độ zoom vào shader.
- **(Nâng cao)** Cho phép người dùng thay đổi số max_iteration bằng cách kéo thanh trượt, cùng với các thuật toán đổi màu mượt mà.

7.6. Hình ảnh kết quả



8. Tập Julia (Julia Set)

8.1. Mô tả và Mối liên hệ với Tập Mandelbrot:

- Cũng dựa trên dãy lặp $z_{n+1} = z_n^k + c$.
- Điểm khác biệt: Tham số c được cố định cho toàn bộ tập Julia. Biến số là điểm bắt đầu z_0 (mỗi pixel tương ứng với một z_0 khác nhau trong mặt phẳng phức).
- Tập Julia J_c là tập hợp các điểm z_0 sao cho dãy không tiến ra vô cùng với một c cho trước.
- Hình dạng của tập Julia phụ thuộc vào giá trị c. Nếu c nằm trong tập Mandelbrot, tập Julia J_c thường là liên thông (connected). Nếu c nằm ngoài, J_c thường là một "đám mây" các điểm không liên thông (Cantor set).

8.2. Thuật toán sinh Tập Julia (Escape Time Algorithm):

- Chọn một giá trị c cố định.
- Ánh xạ mỗi pixel trên màn hình tới một điểm z_0 trong mặt phẳng phức.
- Với mỗi z_0, lặp công thức $z_{n+1} = z_n^2 + c$ từ z_0 đó.

- Kiểm tra điều kiện thoát $|z_n| > 2$.
- Tô màu pixel dựa trên số bước lặp trước khi thoát (hoặc màu đen nếu không thoát).

8.3. Tính chất hình học:

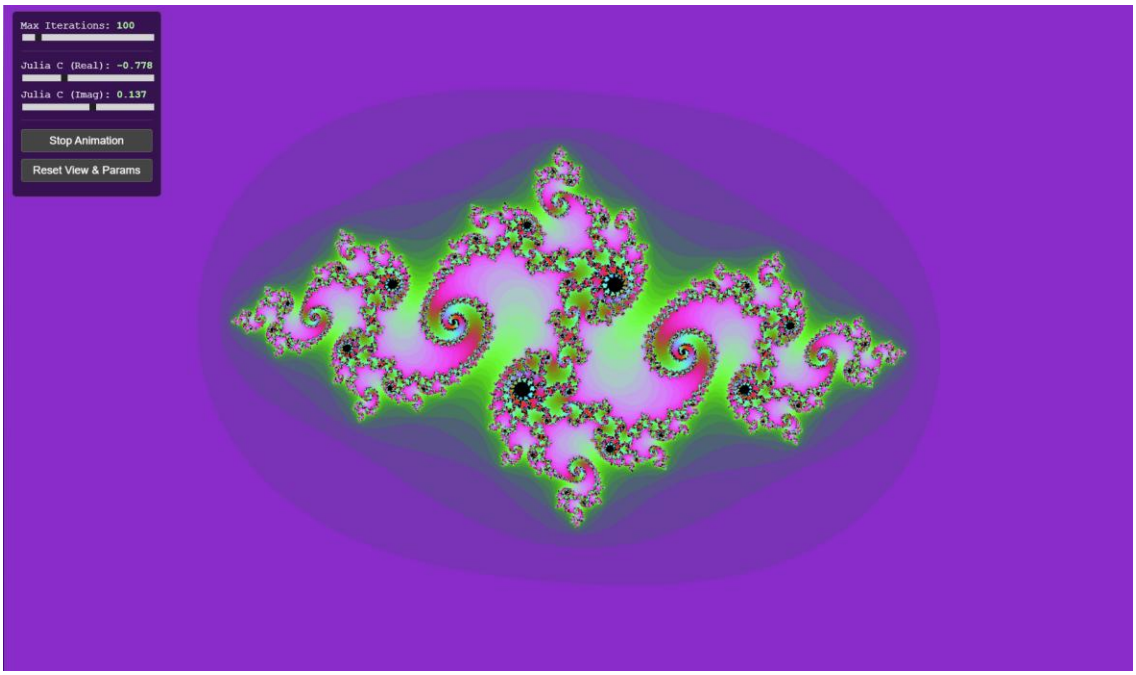
- **Hình dạng phụ thuộc tham số ccc:** Với mỗi giá trị phức ccc, ta có một tập Julia khác nhau. Có thể là **liên thông** hoặc **rời rạc** (dust-like), tùy thuộc vào vị trí của ccc trong Mandelbrot Set.
- **Tự đồng dạng (self-similarity):** Nhiều Julia Set biểu hiện rõ tính chất **tự lặp lại**, nghĩa là các phần nhỏ của hình giống với toàn thể, đặc trưng cho fractal.
- **Biên phức tạp vô hạn:** Ranh giới của Julia Set là **vô cùng chi tiết**, và càng phóng to càng thấy rõ các cấu trúc phân nhánh hoặc xoắn ốc.
- **Diện tích thay đổi:** Có Julia Set có diện tích lớn (liên thông), và cũng có cái có diện tích gần như bằng 0 (trường hợp rời rạc). Không có một diện tích cố định cho tất cả.
- **Fractal dimension (phụ thuộc ccc):**
  Giá trị chiều phân dạng của Julia Set thay đổi theo ccc, thường nằm trong khoảng: $D \in (1, 2)$
  Gần 2 khi hình rất "dày", và gần 1 khi hình như bụi.

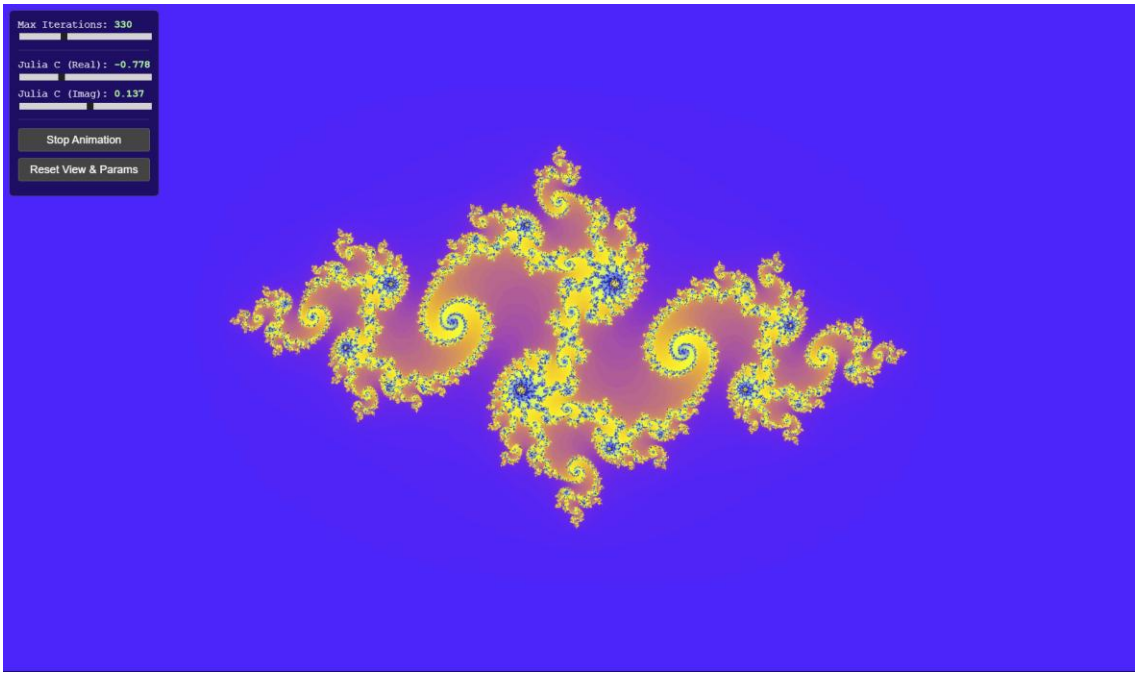
8.4. Triển khai bằng WebGL:

- Rất giống với Mandelbrot, tính toán chính nằm trong Fragment Shader.
- Vertex Shader: Tương tự Mandelbrot.
- Fragment Shader:
  + Nhận tọa độ, ánh xạ thành z_0.
  + Nhận giá trị c cố định thông qua uniform variable.
  + Thực hiện vòng lặp Escape Time Algorithm cho z_n.
  + Xác định màu sắc.
  + Gán gl_FragColor.
- Uniform Variables: max_iterations, tâm, zoom, và quan trọng nhất là giá trị c (phần thực và phần ảo).
- **(Nâng cao)** Cho phép người dùng thay đổi số max_iteration, JuliaCx hay phần thực của hằng số c (điều khiển độ lệch trái/phải

của fractal) và JuliaCy hay phần ảo của hằng số c (điều khiển độ lệch trên/dưới của fractal) bằng cách kéo thanh trượt, cùng với các thuật toán đổi màu mượt mà.

8.5. Hình ảnh kết quả





9. Tổng kết Triển khai WebGL

9.1. Cấu trúc dự án chung (HTML, CSS, JavaScript file(s))

cs105_lab04
├──juliaset
│ ├── index.html
│ ├── juliaset.js
│ └── webgl-utils.js
├──Koch-Snowflake
│ ├── index.html
│ ├── snow.js
│ └── webgl-utils.js
├──Mandelbrot
│ ├── index.html
│ ├── Mandelbrot.js
│ └── webgl-utils.js
├──Minkowski-Island
│ ├── index.html
│ ├── island.js
│ └── webgl-utils.js
├──Sierpinski-Carpet
│ ├── index.html
│ ├── Sierpinski-Carpet.js
│ └── webgl-utils.js
└──Sierpinski-Triangle
  ├── index.html
  ├── Sierpinski-Triangle.js
  └── webgl-utils.js

9.2. Những khó khăn gặp phải (ví dụ: quản lý bộ nhớ đỉnh, tối ưu shader, xử lý tương tác người dùng)

9.2.1. Quản lý bộ nhớ đỉnh (Vertex Memory Management)

Khó khăn:

Fractal thường có cấu trúc đệ quy, dẫn đến số lượng đỉnh tăng theo cấp số nhân ở mỗi cấp độ chi tiết (level of detail). Với các cấp độ cao (depth >= 5), số lượng đỉnh rất lớn, gây quá tải bộ

đệm vertex buffer (VBO), làm chậm tốc độ vẽ hoặc thậm chí gây crash trình duyệt.

Giải pháp:

- Giới hạn cấp độ đệ quy phù hợp với hiệu suất thiết bị (thường <= 5).
- Sử dụng dữ liệu động (dynamic buffer) để cập nhật VBO chỉ khi cần thiết.
- Áp dụng L-System để sinh đỉnh một cách tuyến tính thay vì dùng đệ quy thuần, giúp dễ kiểm soát bộ nhớ.
- Nếu có thể, tận dụng instancing để vẽ các phần tử lặp lại nhiều lần thay vì sinh đỉnh mới.

9.2.2. Tối ưu shader (Vertex & Fragment Shader Optimization)

Khó khăn:

Shader không được thiết kế để xử lý logic phức tạp như đệ quy hoặc tính toán fractal sâu. Shader phức tạp sẽ khiến hiệu năng giảm mạnh, đặc biệt trên thiết bị di động.

Giải pháp:

- Tách phần tính toán hình học (như sinh đỉnh của fractal) ra khỏi shader, thực hiện trên CPU và truyền vào GPU qua buffer.
- Tối giản fragment shader bằng cách hạn chế tính toán màu, ánh sáng nếu không cần thiết.
- Dùng precision thấp hơn (mediump thay vì highp) trong fragment shader nếu không cần độ chính xác cao.

10. Kết luận

10.1. Tóm tắt các kết quả đạt được

- Hiểu biết lý thuyết về các hình dạng fractal.
- Triển khai thành công các fractal bằng WebGL.

10.2. Hướng phát triển hoặc cải tiến có thể

Tăng thêm tương tác với các fractal (tạo thêm nhiều chuyển động phức tạp hơn cho các fractal, tạo hiệu ứng đổ bóng cho các fractal, …)

11. Tài liệu tham khảo

11.1. https://adammurray.link/webgl/#basic-examples

11.2. http://www.shodor.org/interactivate/activities/KochSnowflake/

11.3. http://www.shodor.org/interactivate/activities/SierpinskiTriangle/

12. Video demo WebGL

https://drive.google.com/drive/folders/1t9WPjQvl2WhrP_sq8z7tv0tApcDcVUAf?usp=drive_link

13. Phân công công việc

Thành viên	Công việc	Mức độ hoàn thành
Phạm Phương Minh Trí	- Tìm hiểu lý thuyết bông tuyết Von Koch, đảo Minkowski - Lập trình	100%
Nguyễn Hoàng Gia An	- Tìm hiểu lý thuyết Julia Set - Lập trình - Soạn file doc tiểu luận	100%
Huỳnh Thị Hải Châu	- Tìm hiểu lý thuyết bông tuyết Tam giác Sierpinski - Lập trình	100%
Hoàng Đức Chung	- Tìm hiểu lý thuyết Thảm Sierpinski, Mandelbrot Set - Lập trình - Quay demo các thuật toán - Soạn file doc tiểu luận	100%