Sprawozdanie

Grafika komputerowa i komunikacja człowiek-komputer

Laboratorium 1

Katsiaryna Kolyshko 276708

Dr. Inż. Jan Nikodem

Wstęp teorytyczny

Zadanie laboratoryjne polega na narysowaniu dywanu Sierpińskiego z dodanymi perturbacjami.

Dywan Sierpińskiego to fraktal tworzony poprzez rekurencyjne podzielenie kwadratu na dziewięć mniejszych, identycznych kwadratów, wycięcie środkowego z nich i powtarzanie tej procedury dla każdego z pozostałych ośmiu kwadratów na każdym poziomie iteracji. Proces ten prowadzi do fascynującego efektu, w którym pole powierzchni fraktala dąży do zera wraz ze wzrostem liczby iteracji, pomimo że fraktal obejmuje nieskończoną liczbę punktów w przestrzeni.

Teoria Dywanu Sierpińskiego

Struktura i Tworzenie:

- Proces zaczyna się od jednego kwadratu.
- Kwadrat jest dzielony na siatkę 3x3, co tworzy dziewięć równych części.
- Następnie usuwa się środkowy kwadrat, pozostawiając osiem mniejszych kwadratów.
- Każdy z pozostałych ośmiu kwadratów podlega tej samej procedurze dzielenia 3x3 i usuwania środkowego kwadratu, a proces ten powtarza się rekurencyjnie przez określoną liczbę iteracji.

Wymiar Fraktalny i Pole:

Dywan Sierpińskiego to fraktal o zaskakujących właściwościach geometrycznych. Jedną z nich jest to, że jego pole powierzchni dąży do zera, natomiast obwód staje się nieskończenie długi, co stanowi paradoks w klasycznym rozumieniu geometrii.

Dlaczego pole dąży do zera?

W każdym kroku tworzenia dywanu Sierpińskiego usuwany jest środkowy kwadrat, stanowiący 1/9 pierwotnego kwadratu. Na początku zostaje usunięta tylko niewielka część figury, ale ponieważ proces jest kontynuowany nieskończoną liczbę razy, coraz większa część powierzchni znika.

W każdym etapie iteracji pozostaje 8/9 powierzchni z poprzedniego poziomu. Można to opisać jako sumę geometryczną:

$$\text{Pozostała powierzchnia} = 1*(8/9) + (8/9)^2 + (8/9)^3 + \ldots$$

Ta suma dąży do zera, ponieważ każdy kolejny etap zachowuje tylko ułamek powierzchni poprzedniego etapu. W matematyce taki nieskończony proces sumowania nazywamy szeregiem geometrycznym, a dla odpowiednich wartości może on zbiegać do wartości granicznej – w tym przypadku jest to zero. Stąd pole powierzchni dywanu Sierpińskiego przy nieskończonej liczbie iteracji równa się 0.

Dlaczego obwód jest nieskończony?

Na każdym poziomie iteracji dywan Sierpińskiego jest dzielony na nowe, mniejsze kwadraty, co oznacza wprowadzenie większej liczby linii i krawędzi. Na każdym poziomie zwiększa się liczba kwadratów, z których każdy ma swoje dodatkowe krawędzie. W ten sposób obwód, zamiast maleć, rośnie wraz z liczbą iteracji.

Obwód dywanu rośnie wykładniczo w nieskończoność, ponieważ każdy kolejny poziom dodaje więcej odcinków krawędziowych do figury. Pomimo tego, że kwadraty stają się coraz mniejsze,

liczba ich krawędzi wzrasta tak szybko, że długość obwodu wciąż rośnie, a proces nie wykazuje zbieżności, co skutkuje nieskończonym obwodem.

Fraktalny paradoks – zero pole i nieskończony obwód

Dzięki swoim właściwościom dywan Sierpińskiego jest klasycznym przykładem fraktala, który łamie intuicyjne zasady klasycznej geometrii. Fraktale cechują się tym, że ich wymiar fraktalny jest ułamkowy (w przypadku dywanu Sierpińskiego wynosi około 1,89), co oznacza, że ich struktura mieści się między jedną a dwoma wymiarami. To dlatego dywan Sierpińskiego ma zerowe pole powierzchni (jak figura jednowymiarowa) oraz nieskończony obwód (jak struktura dwuwymiarowa, która nigdy się nie kończy).

Rozmiar Płótna i Potęgi Trójki

Proces podziału opiera się na dzieleniu kwadratu na trzy części zarówno w poziomie, jak i w pionie. Aby zapewnić równomierne podziały, rozmiar płótna powinien być najlepiej potęgą liczby 3 (np. 3^2=9, 3^4=81 itd.).

Wybór płótna, którego rozmiar jest potęgą liczby 3, umożliwia równomierne podzielenie bez wprowadzania frakcyjnych rozmiarów kwadratów, co mogłoby zakłócić wzór fraktala. Właściwość ta jest kluczowa dla uzyskania geometrycznej spójności wzoru fraktalnego.

Wprowadzenie Perturbacji

W tym zadaniu wprowadza się perturbacje, które dodają ciekawy element losowości do tradycyjnego dywanu Sierpińskiego:

Kolor każdego kwadratu może być losowany, co nadaje dywanowi różnorodność wizualną. Ta losowość wzbogaca złożoność wzoru i sprawia, że każdy przebieg tworzenia fraktala ma unikalny wygląd.

Tworzenie Perturbowanego Dywanu Sierpińskiego

Aby narysować dywan Sierpińskiego z perturbacjami, zazwyczaj wykonuje się następujące kroki:

1. Inicjalizacja Płótna: Ustaw rozmiar płótna na wartość będącą potęgą trójki, aby zapewnić równomierny podział.
2. Ustawienie Parametrów: Zdefiniuj liczbę iteracji, zakres perturbacji oraz zakres kolorów.
3. Rekurencyjne Rysowanie z Perturbacjami:
    - Zacznij od początkowego kwadratu.
    - Na każdym poziomie rekurencji podziel każdy kwadrat na dziewięć równych części.
    - Nałóż losową perturbację na każdy narożnik powstałych kwadratów.
    - Wybierz losowy kolor dla każdego kwadratu.
    - Powtarzaj proces dla pozostałych kwadratów, pomijając środkowy na każdym poziomie.

Dzięki tym krokom można stworzyć dywan Sierpińskiego, który zachowuje podstawową strukturę fraktalną, ale jednocześnie wprowadza unikalne zakłócenia, dając fascynujące połączenie porządku i chaosu.

Kolorowanie w OpenGL

W OpenGL, kolor jest zazwyczaj określany w przestrzeni barw RGB (Red, Green, Blue), gdzie wartości każdego kanału (czerwonego, zielonego i niebieskiego) mogą przyjmować zakres od 0 do 1 w liczbach rzeczywistych. Przykładowo, kolor biały to (1.0,1.0,1.0), a kolor czarny to (0.0, 0.0, 0.0). Ten zakres jest szczególnie użyteczny, ponieważ można bezpośrednio określić stopień natężenia każdego z kolorów, operując na wartościach zmiennoprzecinkowych.

Ale w wielu programach graficznych lub interfejsach wartości kolorów są jednak podawane w zakresie od 0 do 255 dla każdego kanału. OpenGL również akceptuje te wartości, ale w celu ich wykorzystania w przestrzeni 0−10-10−1, każdą wartość trzeba podzielić przez 255. Na przykład, aby przekształcić kolor RGB (128,64,255) do formatu OpenGL, dzielimy każdy kanał przez 255, czyli

R: 128/255 = 0.502;

G: 64/255 = 0.251;

B: 255/255 = 1;

Otrzymamy następny kolor:

(następna informacja i przedstawione foto są ze strony „coolors”)

Ale istnieje jeszcze parametr Alfa.

Parametr alfa w ustawieniach kolorów reprezentuje przezroczystość. W OpenGL kolory są zwykle określane jako (R, G, B, A), gdzie A (alfa) kontroluje przezroczystość:

A = 1,0 oznacza całkowitą nieprzezroczystość.

A = 0,0 oznacza całkowitą przezroczystość.

Istnieje kilka popularnych przestrzeni barw, które pozwalają na reprezentację koloru w różny sposób:

1. **RGB (Red, Green, Blue)** – Każdy kolor jest złożony z trzech kanałów (czerwony, zielony, niebieski). To standardowa przestrzeń kolorów używana w grafice komputerowej i modelowana jako dodawanie światła – dodając wszystkie kolory otrzymujemy biały.
2. **HSB (Hue, Saturation, Brightness)** – Kolor definiowany jest przez tonację (hue), nasycenie (saturation) i jasność (brightness). Ten model jest intuicyjny dla użytkowników, ponieważ łatwiej wybrać odpowiedni kolor, kontrolując stopień nasycenia i jasności.
3. **HSL (Hue, Saturation, Lightness)** – Podobny do HSB, ale zamiast jasności (brightness) mamy tu pojęcie światłości (lightness), które bardziej równomiernie oddaje intensywność koloru. Stosowany w grafice do kontrolowania bardziej subtelnych odcieni.
4. **CMYK (Cyan, Magenta, Yellow, Key - Black)** – Model używany w druku, gdzie kolory są tworzone poprzez mieszanie tuszów (barwników). Jest to model subtraktywny, co oznacza, że dodanie wszystkich kolorów tworzy czarny, a nie biały.
5. **LAB** – Model przestrzeni barw LAB opiera się na ludzkim postrzeganiu kolorów i używa trzech kanałów: L (jasność), a (od zielonego do czerwonego) i b (od niebieskiego do żółtego).

LAB jest często używany w profesjonalnej obróbce zdjęć i pozwala na bardziej precyzyjną kontrolę nad wyglądem kolorów.

Każdy z tych modeli umożliwia reprezentację kolorów w inny sposób, co ułatwia ich dostosowanie w zależności od zastosowania – od bardziej technicznego RGB w komputerach, przez bardziej wizualny HSL, aż po CMYK w druku.

W OpenGL kolory są zazwyczaj reprezentowane za pomocą wartości w zakresie od 0,0 do 1,0 (dla reprezentacji float) lub od 0 do 255 (dla reprezentacji integer). Ujemne wartości kolorów, takie jak (−1,0, −0,75, −0,3), nie są prawidłowe dla standardowych kanałów kolorów i nie dadzą znaczących ani widocznych rezultatów.

Rasteryzacja:

W grafice komputerowej kolory są przypisywane do pikseli, ale renderowanie kształtu (takiego jak trójkąt) polega na wypełnieniu wszystkich pikseli w jego granicach odpowiednimi kolorami.

Kiedy definiujesz kształt (np. trójkąt) i przypisujesz kolory do jego wierzchołków lub konkretnych pikseli, system graficzny (OpenGL) nie ogranicza się do kolorowania tylko tych wierzchołków czy pikseli. Zamiast tego, cały obszar wewnątrz granic kształtu jest wypełniany kolorem, w procesie rasteryzacji. Rasteryzacja polega na przekształceniu grafiki wektorowej (czyli punktów i linii definiujących kształt) w siatkę kolorowych pikseli na ekranie.

- Jeśli definiować trójkąt o wierzchołkach w kolorze czerwonym, rasteryzacja wypełni każdy piksel znajdujący się wewnątrz tego trójkąta na czerwono.
- Procesor graficzny oblicza, które piksele znajdują się wewnątrz granic trójkąta, i przypisuje każdemu z tych pikseli informacje o kolorze na podstawie danych z wierzchołków lub tekstury.

Kolory Jednolite i Gradientowe

Możesz wybrać, czy trójkąt będzie miał jednolity kolor, czy gradient:

Jednolity Kolor: Jeśli przypiszesz wszystkim wierzchołkom ten sam kolor, rasteryzacja wypełni każdy piksel wewnątrz trójkąta tym kolorem, więc trójkąt będzie miał jednolite wypełnienie.

Gradient (Interpolacja Koloru): Jeśli przypisać różne kolory do wierzchołków, system graficzny interpoluje te kolory na powierzchni figury. Oznacza to, że stopniowo łączy kolory od jednego wierzchołka do drugiego, tworząc płynne przejście lub gradient. Proces ten, zwany interpolacją koloru, polega na obliczeniu średniej wartości kolorów między sąsiednimi wierzchołkami.

Systemy graficzne używają interpolacji i rasteryzacji, aby kształty wyglądały gładko, bez widocznych przerw między pikselami, i odpowiadały zamierzonemu projektowi. Kiedy przypisujemy kolor do piksela wewnątrz figury, ten kolor rozprzestrzenia się zgodnie z zasadami wypełnienia lub cieniowania, dzięki czemu cała figura (trójkąt, kwadrat itd.) wygląda jak spójna, wypełniona przestrzeń, a nie zbiór pojedynczych, kolorowych punktów.

Rozumienie pokrycia pikseli

Podczas rasteryzacji każdy piksel albo znajduje się całkowicie wewnątrz figury, albo poza nią. Jeśli znajduje się wewnątrz, zostaje mu przypisany kolor odpowiadający wyliczonemu kolorowi na tej pozycji (na podstawie jednolitego lub gradientowego wypełnienia). Dlatego figura, mimo że jest zdefiniowana jedynie przez kilka pikseli na wierzchołkach, jest wypełniana kolorem na całej powierzchni, a nie tylko na wybranych punktach.

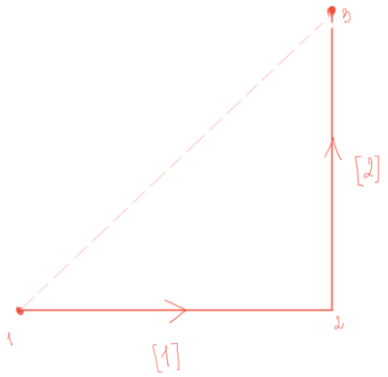
Podsumowując, cała figura jest wypełniona kolorem, ponieważ rasteryzacja stosuje zasady przypisywania kolorów do wszystkich pikseli znajdujących się w granicach figury. Interpolacja koloru (jeśli jest używana) zapewnia płynne przejścia między kolorami przypisanymi do różnych wierzchołków, co sprawia, że figury wyglądają na jednolicie wypełnione lub gładko cieniowane.

Analiza Kodu

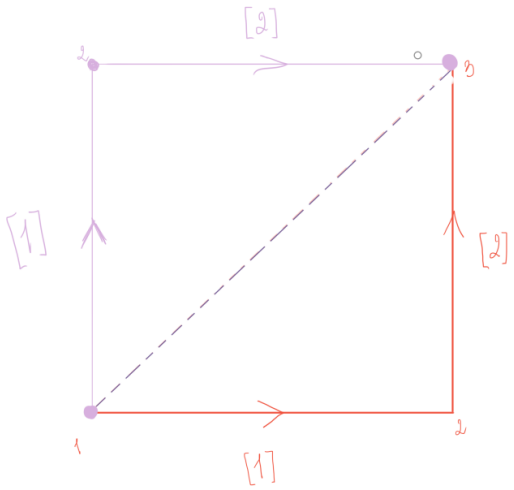
Napisałam program, który wyświetla i pokazuje dywan Sierpieńskiego;

Opiszemy główne kroki jak to działa, żeby lepiej zrozumieć budowę dywanu.

1. Narysuj trójkąt (grafika działą na trójkątach, dlatego ze prowadzimy 3 pixeli)

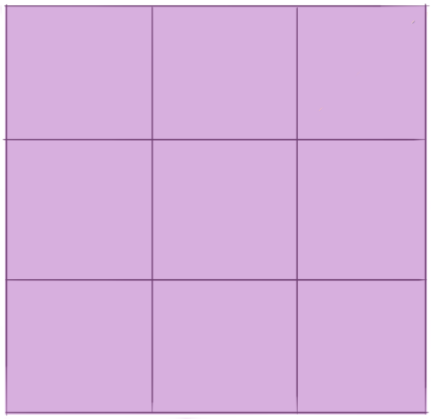


2. Narysuj drugi trójkąt

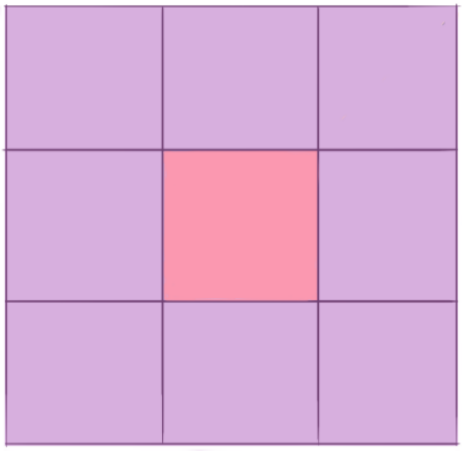


2.2) Można ominąć ten krok z używaniem GL_QUADS, który od razu rysuje kwadrat

3.Podziel kwadraty na 9 równych części liniami

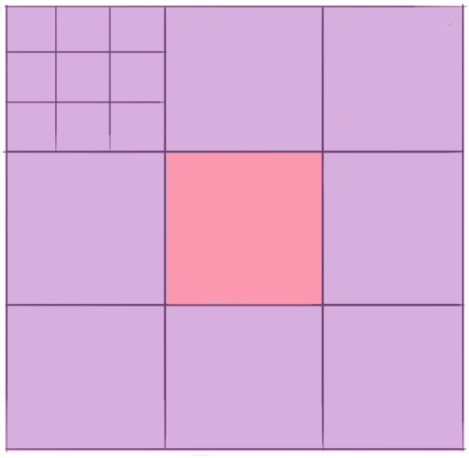


4.Usuń środkowy

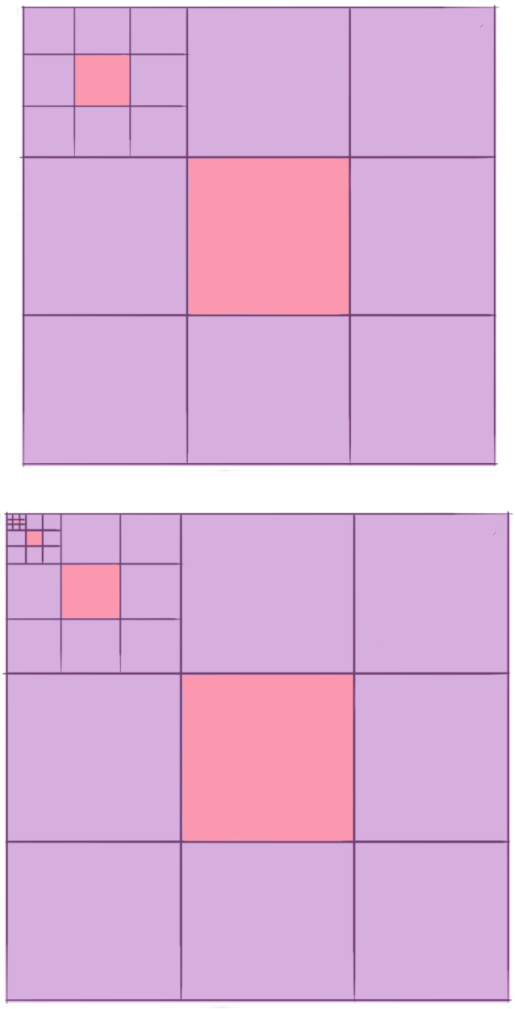


5. Przejdź do podzielonej części

6. Podziel go ponownie przez 9



7. Usuń środek, przejdź do następnego kwadratu i powtórz proces, zgodnie z liczbą podanych iteracji



8. Przejdź do innego kwadratu i powtórz proces

Kod:

Kod był napisany i skompilowany w programie PyCharm.

Funkcja startup( ):

Ta funkcja inicjuje środowisko OpenGL, ustawiając obszar widoku i ustawiając kolor czysty na czarny. Określa rozmiar obszaru widoku jako 2187 x 2187 pikseli, aby dopasować go do rozmiaru dywanu Sierpińskiego i przygotowuje kolor tła okna jako czarny.

Funkcja shutdown( ):

Ta funkcja ma obsługiwać wszelkie niezbędne zadania czyszczenia po zakończeniu programu, ale w tym przypadku jest pusta, ponieważ nie są potrzebne żadne specjalne zadania zamykania.

```
# Set up the dark gray color for the window with the size 2187 x 2187 pixels [2187 because nicely divided by 3]
def startup():  1 usage  new *
    update_viewport( window: None,  width: 2187,  height: 2187)
    glClearColor( red: 0.07,  green: 0.086,  blue: 0.098,  alpha: 0.0)  # Set background to black for main square

# Close the program when finished
def shutdown():  1 usage  new *
    pass
```

Funkcja draw_square( ): Ta funkcja rysuje kwadrat w określonej pozycji (x, y) o danym rozmiarze. Jeśli podano parametr koloru, ustawia kwadrat na ten kolor; w przeciwnym razie używa czerni. Kwadrat jest rysowany przy użyciu prymitywu GL_QUADS OpenGL, tworząc prosty wypełniony kwadrat.

```
def draw_square(x, y, size, color=None):  2 usages  new *
    if color:
        glColor3f(*color)  # Use provided color for the "deleted" square
    else:
        glColor3f( red: 0.0,  green: 0.0,  blue: 0.0)  # Default black color for the main square

    glBegin(GL_QUADS)
    glVertex2f(x, y)
    glVertex2f(x + size, y)
    glVertex2f(x + size, y + size)
    glVertex2f(x, y + size)
    glEnd()
```

Funkcja drawFractal( ):

Ta rekurencyjna funkcja rysuje wzór dywanu Sierpińskiego. Zaczyna od dużego kwadratu i dzieli go na dziewięć mniejszych kwadratów w siatce 3x3, pomijając środkowy kwadrat na każdym poziomie (kwadrat „usunięty"). Jeśli kwadrat jest „usunięty", jest on kolorowany losowo i przechowywany w słowniku, aby zapewnić spójność kolorów w przypadku przerysowań. Funkcja kontynuuje dzielenie i pomijanie środkowego kwadratu na każdym poziomie, aż do osiągnięcia przypadku bazowego (depth == 0), w którym to momencie rysowany jest czarny kwadrat.

```
# Recursively draw the Sierpiński carpet pattern
def drawFractal(x, y, size, depth):  2 usages  new *
    # Base case: if depth is 0, draw a single black square
    if depth == 0:
        draw_square(x, y, size)
    else:
        # Calculate the size of each sub-square by dividing by 3
        new_size = size / 3
        for i in range(3):
            for j in range(3):
                # Skip the center square (this is the "deleted" square)
                if i == 1 and j == 1:
                    # Generate a unique key for this square's position and recursion depth
                    key = (x + new_size * i, y + new_size * j, depth)
                    # If the square doesn't already have a color, assign a random color
                    if key not in deleted_square_colors:
                        deleted_square_colors[key] = (random.random(), random.random(), random.random())
                    # Retrieve the color from the dictionary and use it to draw the "deleted" square
                    color = deleted_square_colors[key]
                    draw_square(x + new_size * i, y + new_size * j, new_size, color=color)
                else:
                    # Recursively draw smaller squares for each non-center position
                    drawFractal(x + new_size * i, y + new_size * j, new_size, depth - 1)
```

Funkcja render( ):

Ta funkcja czyści ekran i inicjuje rysowanie dywanu Sierpińskiego poprzez wywołanie drawFractal( ) w odpowiednim początkowym rozmiarze i pozycji. Zaczyna od środka i określa głębokość 4, aby utworzyć fraktal. Po narysowaniu opróżnia bufor, aby wyświetlić fraktal na ekranie.

```
# Render function to clear the screen and start drawing the fractal pattern
def render(time):  1 usage  new *
    glClear(GL_COLOR_BUFFER_BIT)
    # Begin drawing the carpet from the center with the initial size and recursion depth
    drawFractal(-1093.5, -1093.5,  size: 2187,  depth: 4)  # Start at (-1093.5, -1093.5) to center the 2187 square
    glFlush()
```

Funkcja update_viewport( ):

Ta funkcja jest wywoływana zawsze, gdy okno jest zmieniane w celu zachowania prawidłowego współczynnika proporcji kwadratowego wzoru. Oblicza współczynnik proporcji i aktualizuje macierz projekcji OpenGL, aby zachować proporcje kwadratu 1:1 dywanu Sierpińskiego, odpowiednio dostosowując projekcję ortograficzną.

```
# Handle window resizing and maintain the correct aspect ratio
def update_viewport(window, width, height):  2 usages  new *
    if width == 0:
        width = 1
    if height == 0:
        height = 1
    aspect_ratio = width / height

    glMatrixMode(GL_PROJECTION)
    glViewport( x: 0,  y: 0, width, height)
    glLoadIdentity()

    if width <= height:
        glOrtho(-1093.5,  right: 1093.5, -1093.5 / aspect_ratio, 1093.5 / aspect_ratio,  zNear: 1.0, -1.0)
    else:
        glOrtho(-1093.5 * aspect_ratio, 1093.5 * aspect_ratio, -1093.5,  top: 1093.5,  zNear: 1.0, -1.0)

    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
```

Funkcja main( ):

Inicjuje GLFW, ustawia okno i wchodzi do pętli głównej, w której render( ) jest wielokrotnie wywoływany, aż użytkownik zamknie okno. Zajmuje się również konfiguracją i czyszczeniem kontekstu OpenGL, a funkcja glfwTerminate() jest wywoływana na końcu, aby zwolnić wszystkie zasoby GLFW.

```python
# Main function to initialize GLFW, set up the window, and begin the rendering loop
def main():  1 usage  new *
    if not glfwInit():
        sys.exit(-1)

    # Set up the window with 2187 x 2187 size to match the fractal dimensions
    window = glfwCreateWindow( width: 2187,  height: 2187,  title: "Sierpiński Carpet",  monitor: None,  share: None)
    if not window:
        glfwTerminate()
        sys.exit(-1)

    glfwMakeContextCurrent(window)
    glfwSetFramebufferSizeCallback(window, update_viewport)
    glfwSwapInterval(1)

    # Initialize OpenGL settings and begin the render loop
    startup()
    while not glfwWindowShouldClose(window):
        render(glfwGetTime())
        glfwSwapBuffers(window)
        glfwPollEvents()
    shutdown()

    glfwTerminate()
```

Wynik działania programu: