Grafika komputerowa

Sprawozdanie

Damian Koper, 241292

24 listopada 2019

Spis treści

Spis rysunków

Spis listingów

1 OpenGL - podstawy

OpenGL (*Open Graphics Library*) stanowi otwarty i uniwersalny interfejs umożliwiający renderowanie grafiki 2D i 3D. Obliczenia realizowane są dzięki bezpośredniej interakcji z GPU, który, mając zaimplementowane potrzebne operacje, może szybko przeprowadzać obliczenia. Natura abstrakcyjnego interfejsu jakim jest OpenGL pozwala tworzyć przenośne programy renderujące grafikę bez zważania na platformę uruchomienia, gdzie obliczenia mogą być realizowane programowo jak i sprzętowo.

1.1 Bazowa aplikacja

Biblioteką, która tworzy środowisko uruchomieniowe dla wyświetlania wyrenderowanej grafiki i realizuje operacje wejścia-wyjścia jest GLUT (*OpenGL Utility Toolkit*). Prosty program wyświetlający okno można stworzyć niewielkim nakładem pracy.

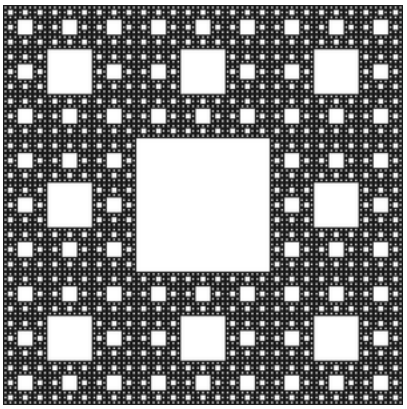
```
#include <GL/glut.h>
void draw()
{
  glClearColor(0, 0, 0, 1);
  glClear(GL_COLOR_BUFFER_BIT);
  glFlush();
}

int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
  glutInitWindowPosition(50, 50);
  glutInitWindowSize(800, 800);
  glutCreateWindow("Lab GK");
  glutDisplayFunc(draw);
  glutMainLoop();
  return 0;
}
```

Listing 1: Bazowy program wyświetlający czarne okno.

W funkcji `main` w pierwszej kolejności inicjalizowana jest sama aplikacja, a następnie wszystkie jej parametry. Do biblioteki GLUT przekazywana poprzez wskaźnik jest funkcja `draw`. Jest ona odpowiedzialna za bezpośrednią interakcję z API OpenGL, a zatem za rysowanie właściwych elementów w wyświetlonym oknie. W tej wersji funkcja `draw` czyści okno kolorem czarnym i opróżnia bufor przekazując dane na ekran.

1.2 Dywan Sierpińskiego

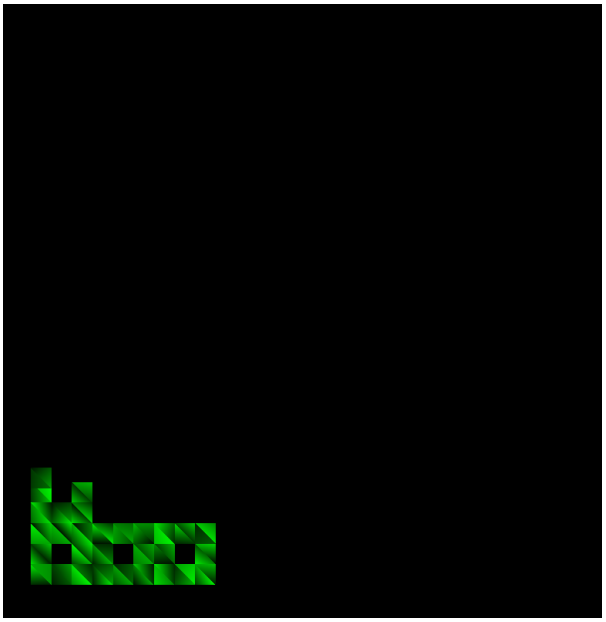
Dywan Sierpińskiego jest jest fraktalem otrzymanym z kwadratu podzielonego na 9 mniejszych kwadratów, z których usuwany jest środkowy. Procedura ta jest rekurencyjnie powtarzana dla pozostałych ośmiu kwadratów.



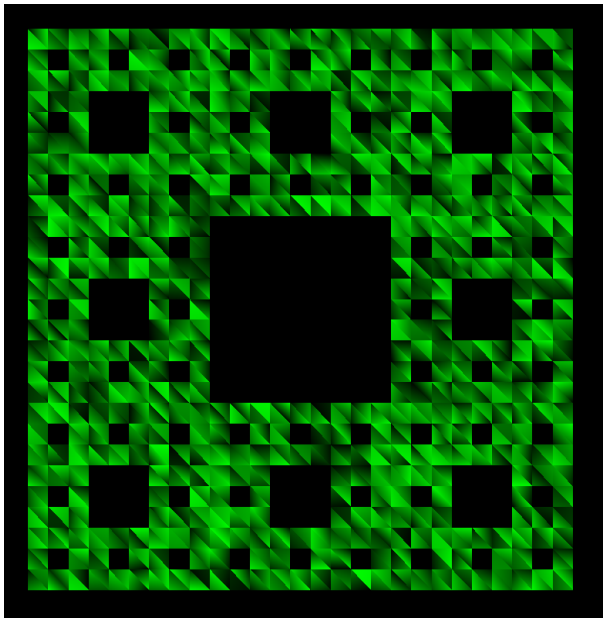
Rysunek 1: Dywan Sierpińskiego po 6 iteracjach.

1.2.1 Rysowanie rekurencyjne

Rysowanie dywanu Sierpińskiego zrealizowane za pomocą rekurencji zakłada rekurencyjne wywołanie funkcji dla każdego z ośmiu zewnętrznych kwadratów fraktalu. Poziomów wywoływań następuje tyle ile zostało zdefiniowane dla pierwszego wywołania funkcji. Gdy zostanie spełniony warunek zakończenia rekurencji, rysowany jest kwadrat. Funkcja po raz pierwszy zostaje wywołana w metodzie `draw` zaraz po wyczyszczeniu okna.



Rysunek 2: Częściowo narysowany dywan Sierpińskiego po zakończeniu 30 gałęzi rekurencji.



Rysunek 3: Dywan Sierpińskiego narysowany w całości.

```
1  void carpet(int levels, float a, float dx = 0, float dy = 0)
2  {
3    if (levels != 0)
4    {
5      a = a / 3;
6      carpet(levels - 1, a, dx - a, dy - a);
7      ...
8      carpet(levels - 1, a, dx + a, dy + a);
9    }
10   else
11   {
12     //Pomocnicza funkcja rysujaca kwadrat
13     rect(dx, dy, a);
14   }
15 }
```

Listing 2: Funkcja rysująca dywan Sierpińskiego rekurencyjnie. Pominięto niektóre wywołania.

1.2.2 Rysowanie iteracyjne

Rysowanie iteracyjne możliwe jest do osiągnięcia przez matematyczne sprawdzenie czy dany kwadrat ma być wypełniony czy nie, albo zamianę wersji rekurencyjnej do wersji iteracyjnej z wykorzystaniem kolejki i struktury danych opisującej aktualny poziom rysowania.

```
struct CarpetLevelData
{
  int level;
  float a;
  float dx = 0;
  float dy = 0;
};
```

Listing 3: Struktura danych przechowująca aktualny stan rysowania.

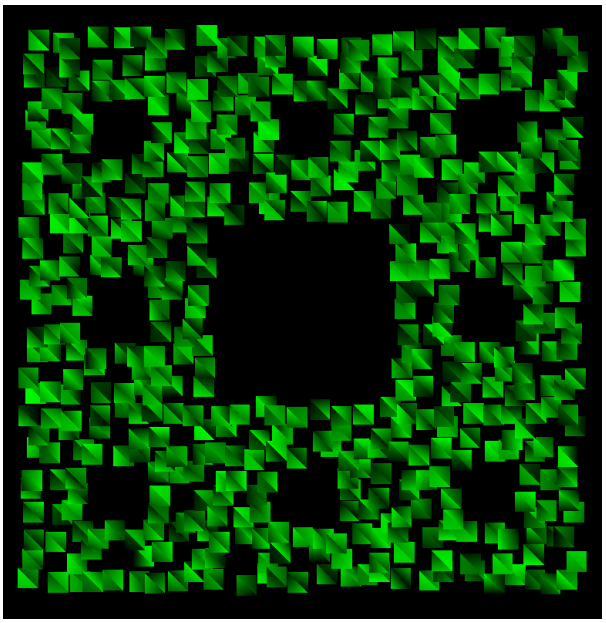
```
void carpetIt(int levels, float a)
{
  queue<CarpetLevelData> q = queue<CarpetLevelData>();
  q.push({levels, a});
  while (!q.empty())
  {
    CarpetLevelData data = q.front();
    if (data.level > 0)
    {
      data.a = data.a / 3;
      q.push({data.level - 1, data.a, data.dx - data.a, data.dy - data.a});
      ...
      q.push({data.level - 1, data.a, data.dx + data.a, data.dy + data.a});
    }
    else
    {
      rect(data.dx, data.dy, data.a);
    }
    q.pop();
  }
```

Listing 4: Funkcja rysująca dywan Sierpińskiego iteracyjnie. Pominięto niektóre wywołania.

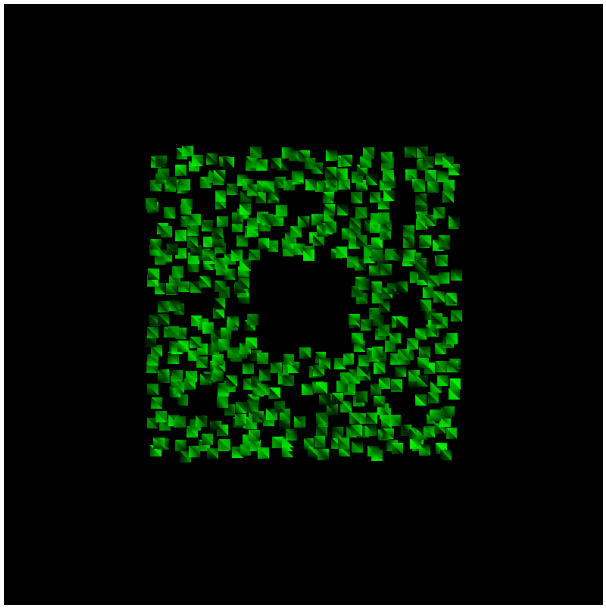
1.2.3 Losowe kolory, perturbacje i skalowanie

Losowe kolory osiągnięto poprzez wywołanie funkcji `glColor3ub(0, randChar(), 0)` przed wywołaniem funkcji interfejsu OpenGL definiującą składowy wierzchołek trójkąta tworzącego kwadrat, gdzie funkcja `randChar` zwraca liczbę z przedziału $< 0; 255 >$. Mając kontrolę nad rysowaniem każdego kwadratu można wprowadzić również losowe jego przesunięcie w osi X i Y.

Funkcja rysująca w swoim drugim parametrze przyjmuje długość boku największego kwadratu, co pozwala na skalowanie figury.



Rysunek 4: Dywan Sierpińskiego po 3 iteracjach z losowym kolorem zielonym oraz perturbacjami.



Rysunek 5: Dywan Sierpińskiego po 3 iteracjach z losowym kolorem zielonym, perturbacjami oraz w innej skali.

2 Modelowanie obiektów 3D

2.1 Bazowa aplikacja

Aplikacja bazowa, po zainicjowaniu niezbędnych elementów biblioteki *GLUT*, oddaje sterowanie do utworzonej klasy `ViewEngine`. Przechowuje one referencje na klasy, będące niezależnymi widokami. Widoki te posiadają zdefiniowane własne funkcje odpowiedzialne za renderowanie, obsługę zdarzeń, animację oraz obsługę zdarzeń związanych z cyklem życia widoku. Poprzez zmianę wartości wskaźnika na aktualny widok w klasie `ViewEngine`, funkcje przekazane do biblioteki *GLUT* zastępowane są funkcjami właściwego widoku. Każdy widok musi implementować interface `iView` zdefiniowany następująco:

```
class IView
{
public:
    virtual std::string getName() = 0;
    virtual void init() = 0;
    virtual void onEnter() = 0;
    virtual void render() = 0;
    virtual void idle() = 0;
    virtual void timer() = 0;
    virtual void onKey(unsigned char key, int x, int y) = 0;
    virtual void onLeave() = 0;

    virtual ~IView(){};
};
```

Listing 5: Interface IView.

Widoku identyfikowane są za pomocą nazw zwracanych przez funkcję `getName()`. Dzięki zaimplementowaniu wzorca Singleton w klasie `ViewEngine` możliwe jest przełączanie widoku w dowolnym miejscu wywołania metody w programie.

```
ViewEngine::g().add(new TeapotView());
ViewEngine::g().add(/*inne widoki*/);
ViewEngine::g().setCurrent("complexEgg");
```

Listing 6: Tworzeznie instancji widoków i ustawianie obecnego. Funkcja `g` zwraca instancję klasy.

2.2 Model i generowanie punktów

2.3 Rysowanie punktów i siatki

2.4 Rysowanie bryły