Grafika komputerowa i komunikacja człowiek-komputer

Sprawozdanie z laboratorium

Data	Tytuł zajęć	Uczestnicy
09.10.2017 8:00	Podstawy OpenGL	Iwo Bujkiewicz (226203)

Zadania

Na zajęciach należało napisać na podstawie instrukcji laboratoryjnej 4 proste programy z użyciem API OpenGL oraz GLUT, realizujące:

1. wyświetlanie pustej sceny,
2. wyświetlanie wypełnionego kwadratu,
3. wyświetlanie dwóch wypełnionych trójkątów,
4. wyświetlanie wielokolorowego trójkąta.

Dodatkowo, na zajęciach lub poza nimi, należało napisać program wyświetlający z użyciem API OpenGL oraz GLUT dywan Sierpińskiego, generowany proceduralnie na podstawie zadanej szczegółowości. Narysowany dywan miał składać się z wielokolorowych kwadratów o 'losowych' kolorach i 'losowym' (w pewnym zakresie) odchyleniu współrzędnych wierzchołków od idealnego dywanu Sierpińskiego.

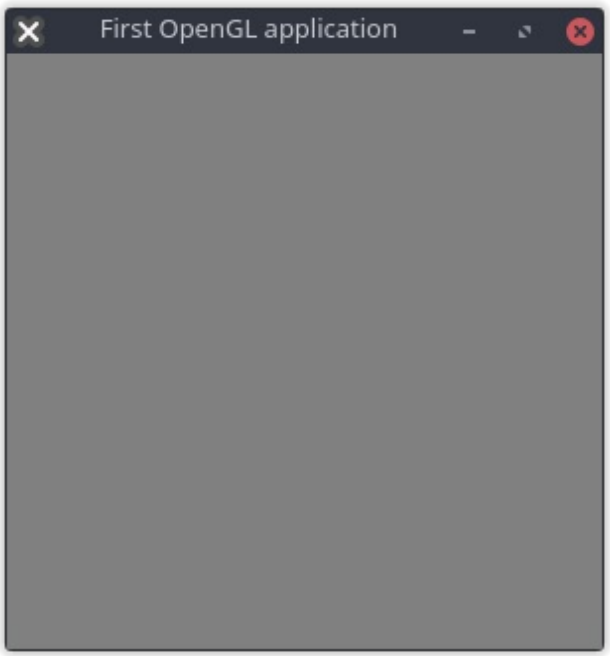
Kolejne etapy realizacji

Zadanie 1

Program miał za zadanie utworzyć okno, zainicjować środowisko renderowania, a następnie wyrenderować pustą scenę. Poniższy listing prezentuje funkcję `render_scene()`, używaną przez GLUT jako funkcję wyświetlania zawartości sceny.

```
void render_scene() {
    // Clear the stage using the current clear colour
    glClear(GL_COLOR_BUFFER_BIT);
    // Flush draw calls to execution
    glFlush();
}
```

Efekt działania programu prezentuje poniższy zrzut ekranu.

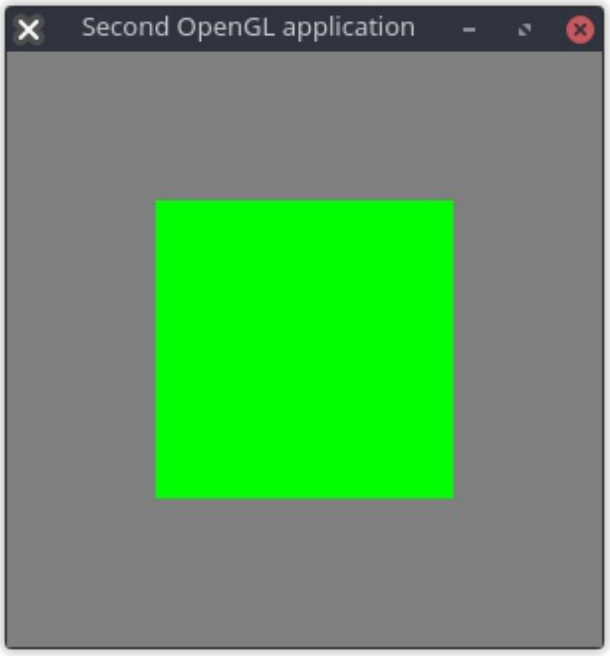


Zadanie 2

Kolejny program miał za zadanie renderować scenę z pojedynczym, płaskim kwadratem. Program miał również dostosowywać renderowany obraz do wymiarów okna. Poniższy listing prezentuje odpowiednio rozszerzoną funkcję `render_scene()` programu do zadania 1.

```
void render_scene() {
    // Clear the stage using the current clear colour
    glClear(GL_COLOR_BUFFER_BIT);
    // Set the drawing colour to green
    glColor3f(0.0f, 1.0f, 0.0f);
    // Draw a rectangle
    glRectf(-50.0f, 50.0f, 50.0f, -50.0f);
    // Flush draw calls to execution
    glFlush();
}
```

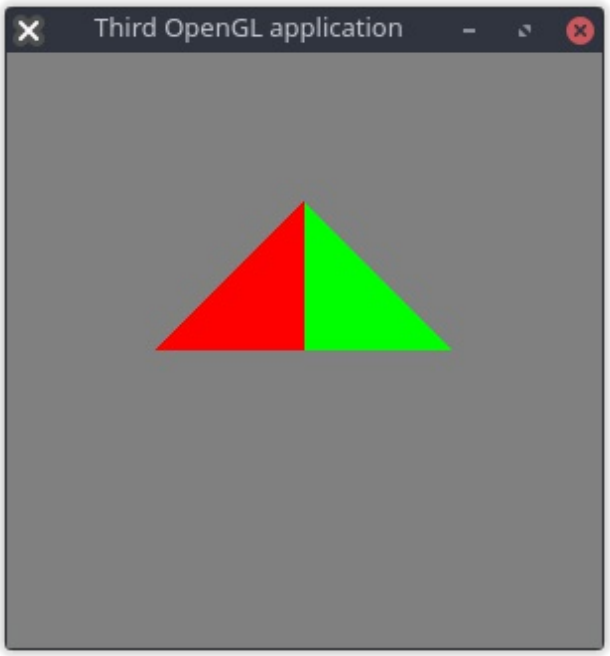
Efekt działania programu prezentuje poniższy zrzut ekranu.



Zadanie 3

Zadanie 3 polegało na zmodyfikowaniu funkcji `render_scene()` z programu do zadania 2 w taki sposób, aby zamiast kwadratu rysowane były dwa trójkąty, każdy wypełniony innym kolorem. Zamiast pojedynczym wywołaniem funkcji (jak w przypadku kwadratu - `glRectf()`), trójkąty definiowane były przez podawane kolejno współrzędne wierzchołków.

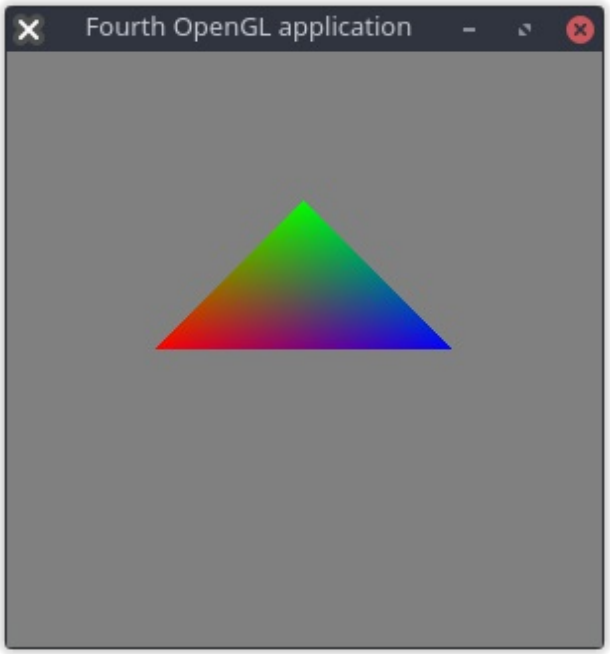
```
void render_scene() {
    // Clear the stage using the current clear colour
    glClear(GL_COLOR_BUFFER_BIT);

    // Set the drawing colour to green
    glColor3f(0.0f, 1.0f, 0.0f);
    // Draw a triangle
    glBegin(GL_TRIANGLES);
    glVertex2f(0.0f, 0.0f);
    glVertex2f(0.0f, 50.0f);
    glVertex2f(50.0f, 0.0f);
    glEnd();

    // Set the drawing colour to red
    glColor3f(1.0f, 0.0f, 0.0f);
    // Draw a triangle
    glBegin(GL_TRIANGLES);
    glVertex2f(0.0f, 0.0f);
    glVertex2f(0.0f, 50.0f);
    glVertex2f(-50.0f, 0.0f);
    glEnd();

    // Flush draw calls to execution
    glFlush();
}
```



Zadanie 4

Zadanie 4 polegało na kolejnej modyfikacji funkcji `render_scene()`, tym razem w taki sposób, aby zamiast dwóch jednolicie wypełnionych trójkątów rysowany był jeden wielokolorowy trójkąt. W tym celu każdy z wierzchołków nowego trójkąta rysowany był w innym kolorze. OpenGL rysował w ten sposób trójkąt cieniowany, w którym odcienie płynnie przechodziły między kolorami poszczególnych wierzchołków.

```
void render_scene() {
    // Clear the stage using the current clear colour
    glClear(GL_COLOR_BUFFER_BIT);

    // Draw a triangle
    glBegin(GL_TRIANGLES);
    // Set the drawing colour to red and create a vertex
    glColor3f(1.0f, 0.0f, 0.0f);
    glVertex2f(-50.0f, 0.0f);
    // Set the drawing colour to green and create a vertex
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex2f(0.0f, 50.0f);
    // Set the drawing colour to blue and create a vertex
    glColor3f(0.0f, 0.0f, 1.0f);
    glVertex2f(50.0f, 0.0f);
    glEnd();

    // Flush draw calls to execution
    glFlush();
}
```



Dywan Sierpińskiego

Program rysujący dywan Sierpińskiego z aberracjami rozpoczynał działanie od zaseedowania generatora serii liczb wartością aktualnego czasu.

```
    srand(time(NULL));
```

Następnie wywoływana była funkcja `carpet()`, generująca matrycę wypełnienia poszczególnych czworokątów składających się na dywan na podstawie zadanej szczegółowości (liczby iteracji). Rozmiar matrycy zależał od szczegółowości - im wyższy był zadany poziom szczegółowości, tym więcej czworokątów składało się na cały dywan. Użyta tu funkcja `powi()` jest własną, skrajnie uproszczoną implementacją potęgowania liczb całkowitych.

```
void carpet(int iterations) {
    cells = powi(3, iterations);

    points = realloc(points, cells * cells * sizeof(bool));
    for (int i = 0; i < cells * cells; ++i)
        points[i] = true;

    for (int square_size = cells; iterations > 0; --iterations) {
        square_size /= 3;
        for (int x = cells - 1; x >= 0; --x) {
            int y = cells - 1;
            for (int y = cells - 1; y >= 0; --y) {
                if ((x / square_size) % 3 == 1 && (y / square_size) % 3 == 1)
                    points[x * cells + y] = false;
            }
        }
    }
}
```

Szczegółowość zadawana była stałą `iterations`. Wskaźnikiem matrycy wypełnienia czworokątów była zmienna `points`, a rząd matrycy (ilość czworokątów w rzędzie) przechowywała zmienna `cells`.

```
const int iterations = 4;
bool * points = NULL;
int cells = 0;
```

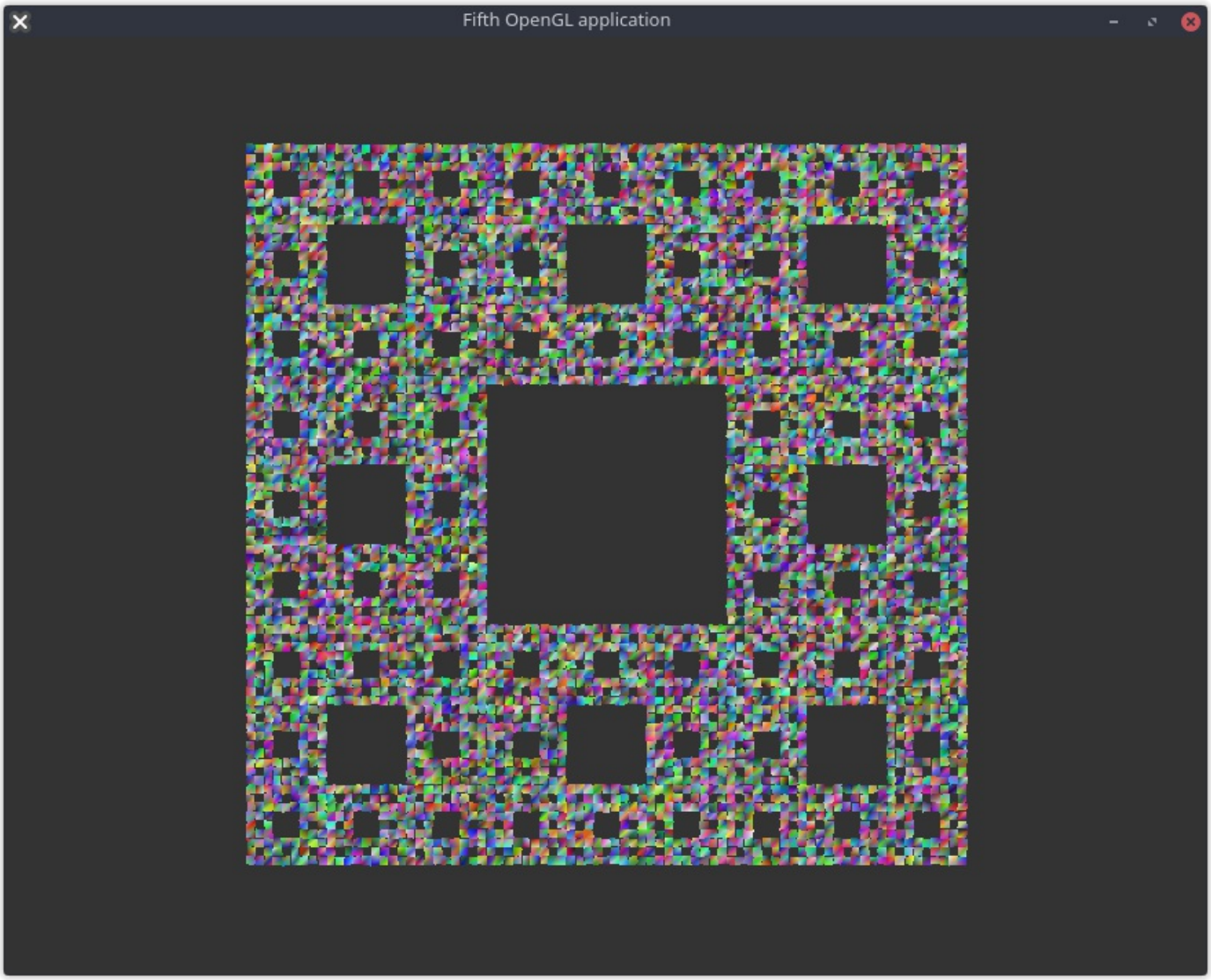
Po wygenerowaniu matrycy wypełnienia tworzone było okno aplikacji i inicjalizowane było środowisko renderowania. Za rysowanie czworokątów na podstawie matrycy odpowiedzialna była funkcja `render_scene()`.

```
void render_scene() {
    // Clear the stage using the current clear colour
    glClear(GL_COLOR_BUFFER_BIT);

    for (int x = 0; x < cells; ++x) {
        for (int y = 0; y < cells; ++y) {
            if (points[x * cells + y])
                draw_square((point2f) { (-50.0f + x * (100.0f / cells)), (-50.0f + y
* (100.0f / cells)) }, 100.0f / cells);
        }
    }

    // Flush draw calls to execution
    glFlush();
}
```

Do rysowania pojedynczego czworokąta funkcja `render_scene()` używała funkcji `draw_square()`, która, na podstawie współrzędnych czworokąta i jego zadanego rozmiaru, generowała współrzędne jego wierzchołków z aberracjami i rysowała je kolejno wygenerowanymi kolorami. Użyta tu funkcja `randf()` jest własną, skrajnie uproszczoną implementacją uzyskiwania liczb zmiennoprzecinkowych od `0.0f` do `1.0f` z serii liczb generowanych przez `rand()`.

```
void draw_square(point2f point, float size) {
    float deviation = size / 4;

    glBegin(GL_POLYGON);
    glColor3f(randf(), randf(), randf());
    glVertex2f(point[0]    + randf() * deviation, point[1]  + randf() * deviation);
    glColor3f(randf(), randf(), randf());
    glVertex2f(point[0] + size + randf() * deviation, point[1]  + randf() *
deviation);
    glColor3f(randf(), randf(), randf());
    glVertex2f(point[0] + size + randf() * deviation, point[1] + size + randf() *
deviation);
    glColor3f(randf(), randf(), randf());
    glVertex2f(point[0]    + randf() * deviation, point[1] + size + randf() *
deviation);
    glEnd();
}
```

Przykładowy efekt działania tak napisanego programu prezentuje zrzut ekranu poniżej.



Kod źródłowy

Kompletny kod opisanych tu programów został załączony do sprawozdania w osobnych plikach.