Politechnika Wrocławska

Grafika komputerowa

Projekt zaliczeniowy: WebGL -podstawy oraz fraktal 2D

Autor:
Bartosz Pogoda 225988

Prowadzący:
mgr inż. Szymon Datko

15 stycznia 2018

Spis treści

1 Wstęp teoretyczny

WebGL jest biblioteką pozwalającą na renderowanie grafiki 2D oraz 3D przy użyciu języka JavaScript. Wyrenderowane sceny są prezentowane w przeglądarce przy pomocu elementu <canvas> wprowadzonego wraz ze standardem HTML5. Implementacja WebGL zapewnia wszystkie możliwośći biblioteki OpenGL poznanej na zajęciach, tj. renderowanie 2D, renderowanie 3D, oświetlanie, teksturowanie.

2 Założenia projektowe

W ramach projektu zostały zrealizowane następujące zadania:

- zapoznanie się z instrukcją dostępną na stronie kursu,
- wyrysowanie czworościanu,
- warunek blokujący funkcję uruchom,
- wyrenderowanie fraktala 2D,
- umożliwienie regulacji parametrów renderowania przy pomocy UI.

3 Realizacja projektu

3.1 Zapoznanie się z instrukcją

Przeczytanie instrukcji zamieszczonej na stronie kursu oraz przetestowanie załączonych przykładów pozwoliło na poznanie technik realizacji różnych możliwości biblioteki OpenGL, poznanych na wcześniejszych zajęciach, w obrębie technologii WebGL.

3.1.1 Kontekst graficzny

Biblioteka WebGL oferuje interfejs WebGLRenderingContext, który zawiera wszelkie operacje potrzebne do rysowania scen 2D oraz 3D w ramach elementu <canvas>. Przed rozpoczęciem renderowania scen należy pobrać kontekst graficzny.

Listing 1: Pobranie kontekstu graficznego

```
1 var canvas = document.getElementById('myCanvas');
2 var gl_ctx = canvas.getContext('webgl');
```

Tak pobrany kontekst udostępnia wszystkie możliwości biblioteki WebGL, a więc między innymi:

- pozyskiwanie informacji o stanie,
- zarządzanie buforami,
- zarządzanie teksturami,
- zarządzanie programami oraz shaderami,
- zarządzanie atrybutami,
- renderowanie.

3.1.2 Bufory wierzchołków oraz indeksów

Bardzo ważne z punktu widzenia dalszej realizacji projektu było opanowanie renderowania obiektów 3D (oraz analogicznie 2D) podając ich wierzchołki. W WebGL tworzy się dwa bufory. Jeden z buforów zawiera współrzędne kolejnych wierzchołków w przestrzeni, wraz z informacją o kolorach przypisanych tym wierzchołkom. Drugi bufor określa kolejność występowania wierzchołków zdefiniowanych w pierwszym buforze (podawane jako indeksy 0, 1, ...) w rysowanych figurach. Poniżej zostały zamieszczone dwie przykładowe tablice, które mogą zostać użyte jako bufory w celu wyrysowania na scenie 3D czworościanu.

Listing 2: Definicja tablicy opisującej bufor wierzchołków dla czworościanu

```
var tetahedronVerticles = [
        1, 1, 1,
        1, 0, 0,

        -1, -1, 1,
        0, 1, 0,

        -1, 1, -1,
        0, 0, 1,

        1, -1, -1,
        1, 1, 0
];
```

Listing 3: Definicja tablicy opisującej bufor indeksów dla czworościanu

```
  var tetahedronFaces = [
        0, 1, 2,
        0, 2, 3,
        0, 1, 3,
        1, 2, 3,
];
```

3.1.3 Rejestracja buforów

Wyżej zdefiniowane struktury nie są jednak widziane jako bufory w bibliotece WebGL. Tak zdefiniowane tablice zawierają wszelkie potrzebne dane, jednak bufory muszą zostać utworzone oraz zarejestrowane przy użyciu konteksktu graficznego (typu *WebGLRenderingContext*).

Listing 4: Utworzenie oraz przypięcie bufora wierzchołków

```
_tetahedronVertexBuffer = gl_ctx.createBuffer();

gl_ctx.bindBuffer(gl_ctx.ARRAY_BUFFER, _tetahedronVertexBuffer);

gl_ctx.bufferData(gl_ctx.ARRAY_BUFFER,
                  new Float32Array(tetahedronVerticles),
                  gl_ctx.STATIC_DRAW);
```

Bufor dla kontekstu *gl_ctx* został utworzony za pomocą metody *createBuffer*. Następnie bufor ten został przypisany do kontekstu wraz z informacją o typie. Ostatecznie bufor został wypełniony danymi zawartymi we wcześniej zdefiniowanej tablicy *tetahedronVerticles*.

Listing 5: Utworzenie oraz przypięcie bufora indeksów

```
1  _tetahedronFacesBuffer = gl_ctx.createBuffer();
2
3  gl_ctx.bindBuffer(gl_ctx.ELEMENT_ARRAY_BUFFER,
4                    _tetahedronFacesBuffer);
5
6  gl_ctx.bufferData(gl_ctx.ELEMENT_ARRAY_BUFFER,
7                    new Uint16Array(tetahedronFaces),
8                    gl_ctx.STATIC_DRAW);
```

W sposób analogiczny został utworzony oraz przypisany bufor indeksów. Warto zwrócić uwagę na typ *WebGLRenderingContext.ELEMENT_ARRAY_BUFFER*, który przekazuje informację, że dany bufor zawiera informację o kolejności wierzchołków, a nie tak jak w przypadku *WebGLRenderingContext.ARRAY_BUFFER* - o położeniu wierzchołków oraz ich kolorach.

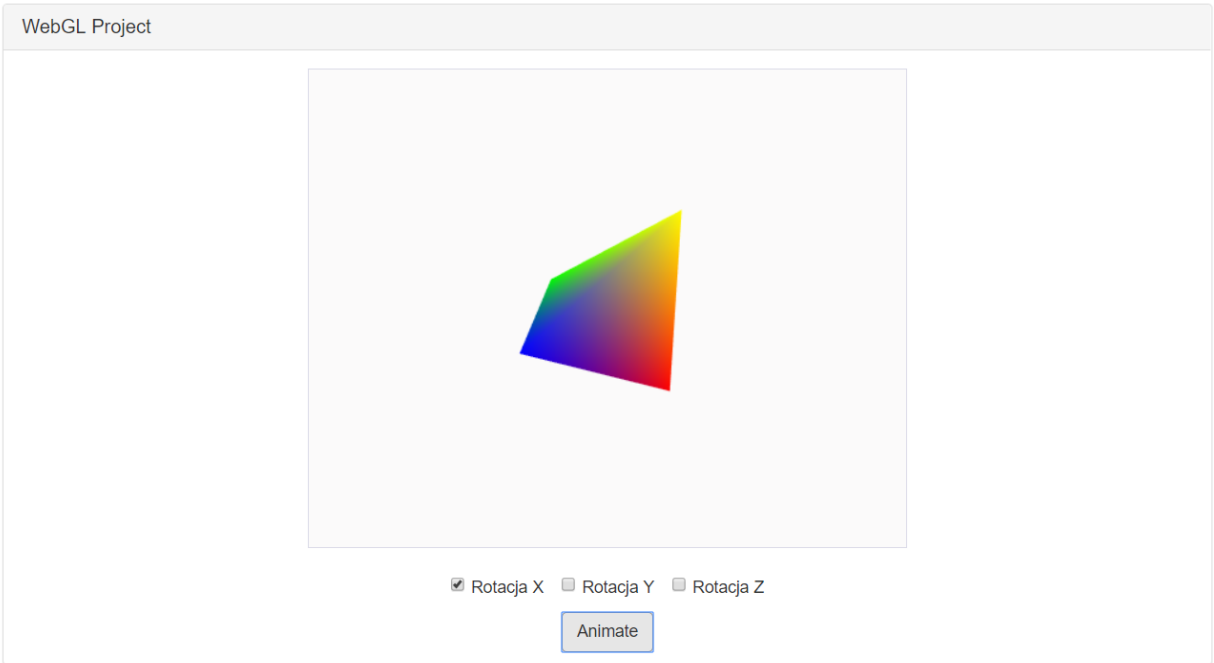
3.2 Wyrysowanie czworościanu

Skrypt zamieszczony na stronie kursu został zmodyfikowany tak, aby na ekranie został wyrysowany czworościan. W tym celu współrzedne przykładowego czworościanu (1,1,1), (-1,-1,1), (-1,1,-1), (1,-1,-1) zostały zapisane w buforach, tak jak zdefiniowano na wyżej zamieszczonych listingach.

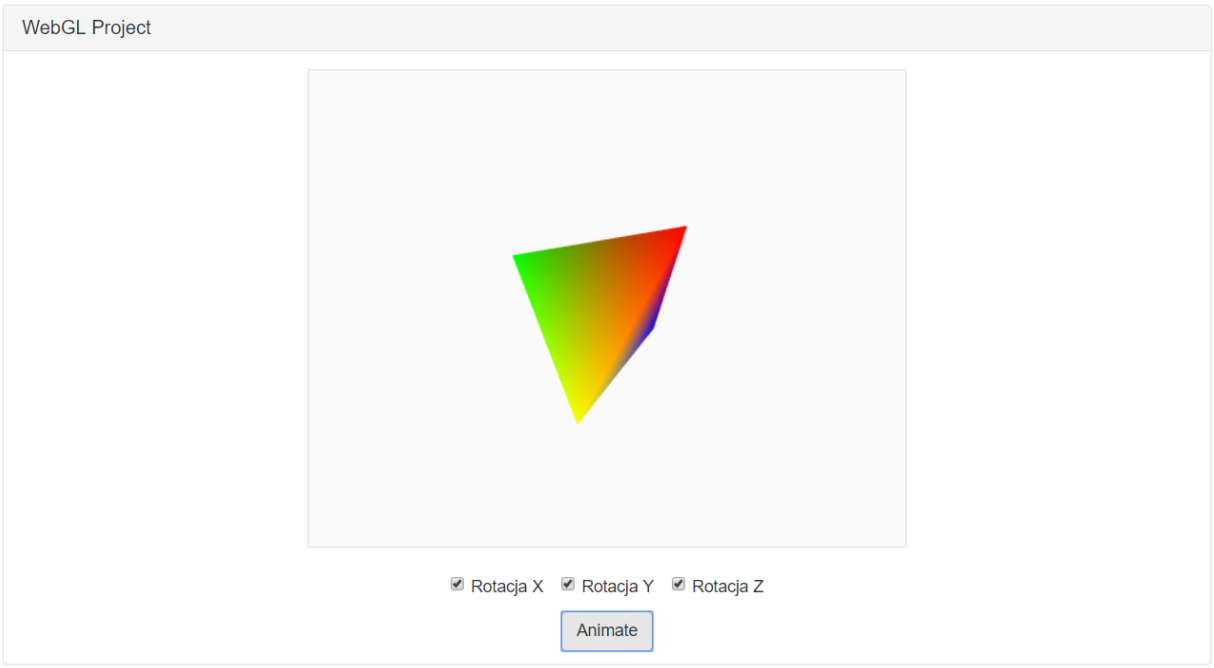
Listing 6: Wyrysowanie elemetów zawartych w buforach

```
1  gl_ctx.drawElements(gl_ctx.TRIANGLES, 4*3,
2                      gl_ctx.UNSIGNED_SHORT, 0);
3  gl_ctx.flush();
```

Czworościan został uzyskany poprzez narysowanie **czterech trójkątów** (każdy zdefiniowany przez **trzy** wierzchołki). Pogrubione parametry są istotne i można je odnaleźć przy wywołaniu funkcji *drawElements*. Warto zaznaczyć, że funkcja rysuje obiekty na podstawie aktualnie przypisanych buforów.



Rysunek 1: Czworościan wyrenderowany w oknie przeglądarki



Rysunek 2: Czworościan w innej fazie rotacji

3.3 Warunek blokujący funkcję uruchom

Przykład, na podstawie którego została stworzona wyżej opisana funkcjonalność, zawierał pewną niedoskonałość (chociaż można by równie dobrze nazwać ją "dodatkową funkcjonalnością") związaną z przyśpieszaniem animacji każdorazowo po wciśnięciu przycisku "Animate".

Niedoskonałość wynikała z każdorazowego dodawania funkcji zwrotnej przy wciśnięciu "Animate" za pomocą funkcji *requestAnimationFrame* obiektu typu *Window*. Funkcja ta przy wywołaniu rejestruje kolejną funkcję (zadaną argumentem), która zostanie uruchomiona przed przerysowaniem zawartości ekranu przez przeglądarkę, czyli zazwyczaj około 60 razy na sekundę. Rejestrowanie kolejnych callback'ów za każdym razem powodowało wywołanie tej samej funkcji wiele razy, co skutkowało widocznym zwielokrotnieniem szybkości animacji.

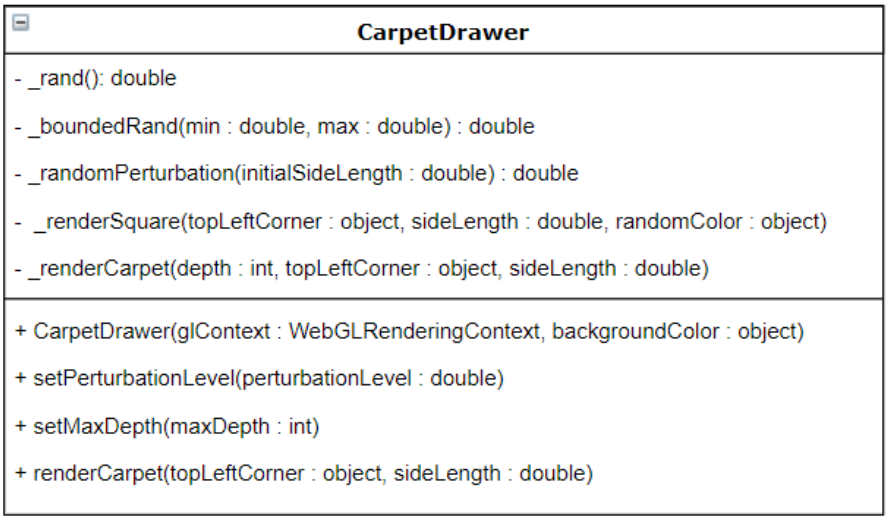
Przykładowym obejściem problemu, które zostało zrealizowane w ramach projektu, było utworzenie flagi, która zostaje ustawiona przy pierwszym uruchomieniu animacji. Przy następnych wciśnięciach przycisku "Animate" funkcje zwrotne nie są rejestrowane na nowo, ze względu na wykrycie ustawionej flagi.

Listing 7: Realizacja obejścia problemu przyśpieszania animacji

```
var alreadyAnimating = false;

function gl_draw() {
    // logika rysowania sceny

    var animate = function (time) {
      // logika animacji

      window.requestAnimationFrame(animate);
    };

  if(!alreadyAnimating) {
    animate(0);
    alreadyAnimating = true;
  }
}
```

3.4 Wyrenderowanie fraktala 2D

Projekt został rozbudowany o wyrenderowanie dywanu Sierpińskiego. Cała logika renderowania została umieszczona w klasie *CarpetDrawer*, której deklaracje metod zostały przedstawione na poniższym diagramie.



Rysunek 3: Deklaracja metod klasy CarpetDrawer

3.4.1 Losowość

W pierwszej kolejności zostały zdefiniowane metody służace do generowania liczb pseudolosowych, które następnie zostały użyte do losowania kolorów wierzchołków oraz zaburzeń we współrzędnych wierzchołków - czyli perturbacji.

3.4.2 Renderowanie kwadratu

Kluczowym etapem implementacji było utworzenie metody odpowiedzialnej za wyrysowanie na ekranie kwadratu w zadanym miejscu, o zadanej długości boku oraz trybie kolorowania.

Listing 8: Realizacja funkcji renderującej pojedyńczy kwadrat

```
1  // Renders square for given topLeftCorner, sideLength.
2  // It has random color if randomColor is set true,
3  // backgoundColor otherwise
4  _renderSquare(topLeftCorner, sideLength, randomColor) {
5    var _rand = this._rand;
6    var glContext = this.glContext;
7    var backgroundColor = this.backgroundColor;
8
9    if (randomColor) {
10
11     var squareVerticles = [
12       topLeftCorner.x, topLeftCorner.y,
13       _rand(), _rand(), _rand(),
14       topLeftCorner.x + sideLength, topLeftCorner.y,
15       _rand(), _rand(), _rand(),
16       topLeftCorner.x + sideLength, topLeftCorner.y - sideLength,
17       _rand(), _rand(), _rand(),
18       topLeftCorner.x, topLeftCorner.y - sideLength,
19       _rand(), _rand(), _rand()
20     ];
```

```
21
22    } else {
23      var squareVerticles = [
24        topLeftCorner.x, topLeftCorner.y,
25        backgroundColor.r, backgroundColor.g, backgroundColor.b,
26        topLeftCorner.x + sideLength, topLeftCorner.y,
27        backgroundColor.r, backgroundColor.g, backgroundColor.b,
28        topLeftCorner.x + sideLength, topLeftCorner.y - sideLength,
29        backgroundColor.r, backgroundColor.g, backgroundColor.b,
30        topLeftCorner.x, topLeftCorner.y - sideLength,
31        backgroundColor.r, backgroundColor.g, backgroundColor.b,
32      ];
33    }
34
35    var squareVertexBuffer = glContext.createBuffer();
36    glContext.bindBuffer(glContext.ARRAY_BUFFER,
37                         squareVertexBuffer);
38    glContext.bufferData(glContext.ARRAY_BUFFER,
39                         new Float32Array(squareVerticles),
40                         glContext.STATIC_DRAW);
41
42    var squareFaces = [
43      0, 1, 2,
44      0, 2, 3
45    ];
46
47    var squareFacesBuffer = glContext.createBuffer();
48    glContext.bindBuffer(glContext.ELEMENT_ARRAY_BUFFER,
49                         squareFacesBuffer);
50    glContext.bufferData(glContext.ELEMENT_ARRAY_BUFFER,
51                         new Uint16Array(squareFaces),
52                         glContext.STATIC_DRAW);
53
54    gl_ctx.vertexAttribPointer(_position, 2, gl_ctx.FLOAT, false,
55                               4 * (2 + 3), 0);
56    gl_ctx.vertexAttribPointer(_color, 3, gl_ctx.FLOAT, false,
57                                  4 * (2 + 3), 2 * 4);
58
59    glContext.drawElements(glContext.TRIANGLES, 2 * 3,
60                           glContext.UNSIGNED_SHORT, 0);
61  }
```

3.4.3 Renderowanie dywanu Sierpińskiego

Po zaimplementowaniu oraz przetestowaniu poprawności działania powyższych funkcji pomocniczych, zostały utworzone funkcje *renderCarpet* oraz *_renderCarpet*. Funkcja *renderCarpet* startuje rekurencyjną procedurę renderowania kolejnych poziomów głębokości dywanu Sierpińskiego. W każdym etapie rekurencji wyrenderowanych zostało osiem kwadratów dookoła wypełnionych losowymi kolorami oraz jeden kwadrat wewnątrz wypełniony kolorem tła. Do wyliczonych pozycji kwadratów zostały zaaplikowane losowe perturbacje.

Listing 9: Definicja funkcji renderCarpet

```
renderCarpet(topLeftCorner, sideLength) {
  this._renderCarpet(0, topLeftCorner, sideLength)
  this.glContext.flush();
}
```

Listing 10: Definicja funkcji rekurencyjnej _renderCarpet

```
_renderCarpet(depth, topLeftCorner, sideLength) {
  if (depth >= this.maxDepth) return;

  var subSquareSideLength = sideLength / 3;

  // Draw empty square
  var emptySquareTopLeftCorner = {
    x: topLeftCorner.x + subSquareSideLength,
    y: topLeftCorner.y - subSquareSideLength
  };
  this._renderSquare({
    x: topLeftCorner.x + subSquareSideLength,
    y: topLeftCorner.y - subSquareSideLength
  }, subSquareSideLength, false);

  // Draw remaining (randomly filled) squares
  for (var i = 0; i < 3; i++) {
    for (var j = 0; j < 3; j++) {
      if (!(i == 1 && i == j)) { // skip one which should be empty
        var deformation = this._randomPerturbation
                                              (subSquareSideLength);
        var subSquareTopLeftCorner = {
            x: topLeftCorner.x +
         (i * subSquareSideLength) + deformation,
            y: topLeftCorner.y -
        (j * subSquareSideLength) + deformation
        };

        this._renderSquare(subSquareTopLeftCorner,
                           subSquareSideLength, true);
        // render carpet recursive call
        this._renderCarpet(depth + 1, subSquareTopLeftCorner,
                           subSquareSideLength);
      }
    }
  }
}
```

3.5 Umożliwienie regulacji parametrów renderowania przy pomocy UI

W ramach interfejsu użytkownika utworzono dwa suwaki - do sterowania poziom perturbacji oraz głębokością wywołań rekurencyjnych. Suwaki zostały utworzone przy pomocy biblioteki jQueryUI.

Listing 11: Utworzenie suwaka dla zmian poziomu perturbacji

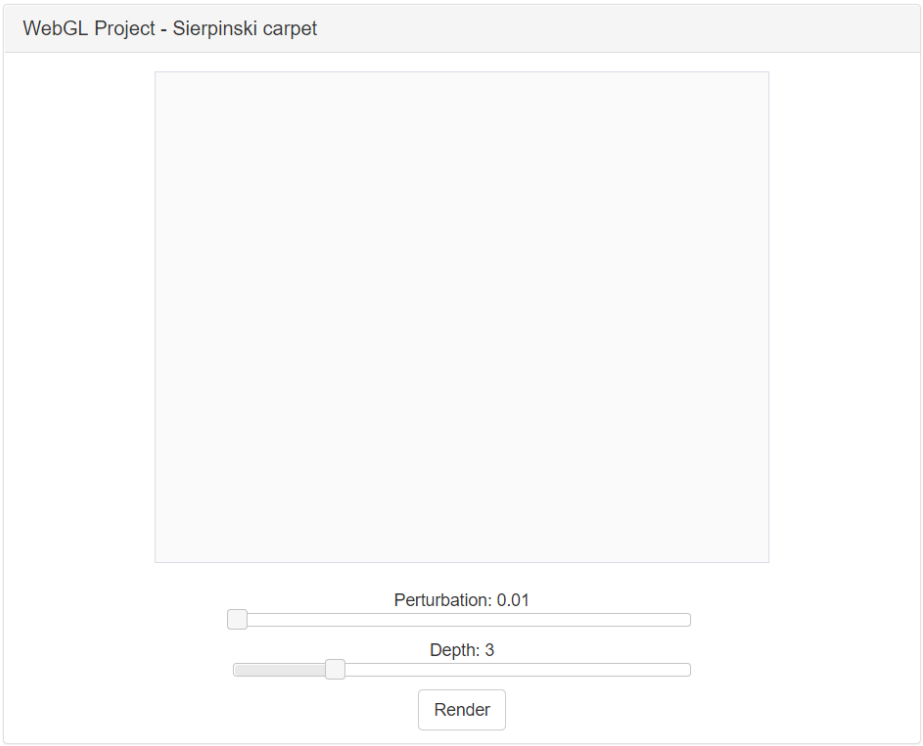
```
var perturbationPreview = $("#perturbationPreview");
$("#perturbationSlider").slider({
  range: "min",
  min: 0,
  max: 150,
  value: 1,
  create: function() {
    perturbationPreview.text($(this).slider("value") / 100);
  },
  slide: function(event, ui) {
    perturbationPreview.text(ui.value / 100);
  }
});
```

Parametry ustawione za pomocą suwaków są przekazywane do modelu (obiektu klasy *CarpetDrawer*) po wciśnięciu przycisku "Render". Następnie wywoływana jest funkcja *renderCarpet*, która renderuje dywan o lewym górnym wierzchołku (-1, 1) i długości boku 2, co przy domyślnym ustawieniu pozycji "kamery" skutkuje pokryciem całego obszaru rysowania.

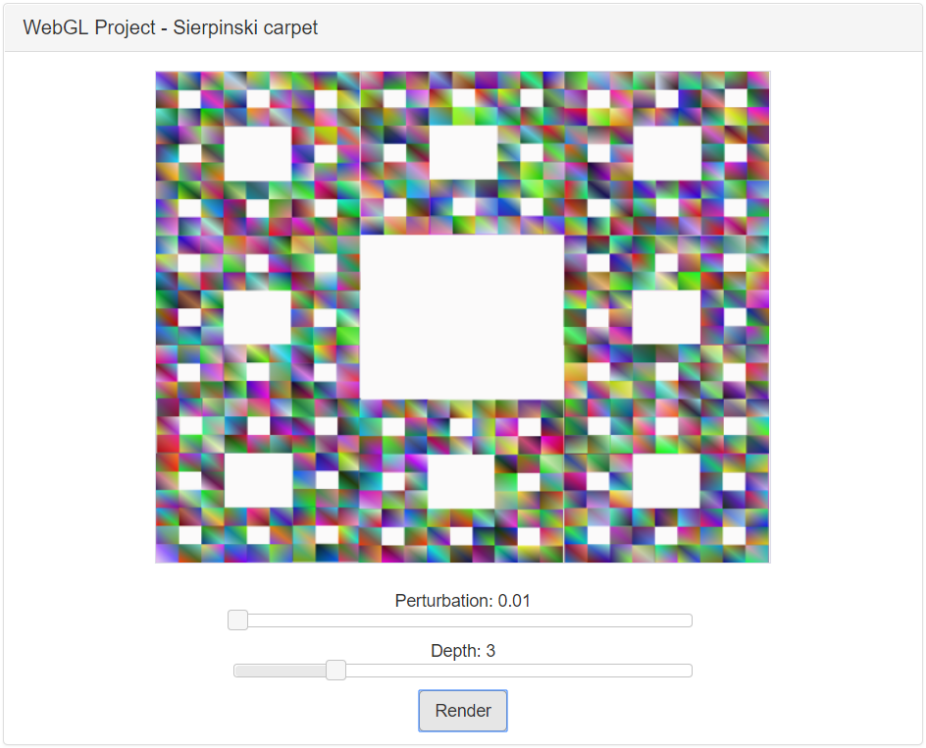
Listing 12: Metoda wywołująca się przy nacisnięciu przycisku "Render"

```
function gl_draw() {
  var backgroundColor = {
    r: 0.984,
    g: 0.980,
    b: 0.980
  };

  gl_ctx.clearColor(backgroundColor.r, backgroundColor.g,
                    backgroundColor.b, 1.0);

  let carpetDrawer = new CarpetDrawer(gl_ctx, backgroundColor);
  carpetDrawer.setPeturbationLevel($("#perturbationPreview")
                                     .text());
  carpetDrawer.setMaxDepth($("#depthPreview").text());

  carpetDrawer.renderCarpet({ x: -1, y: 1 }, 2);
}
```

4 Zrzuty ekranu

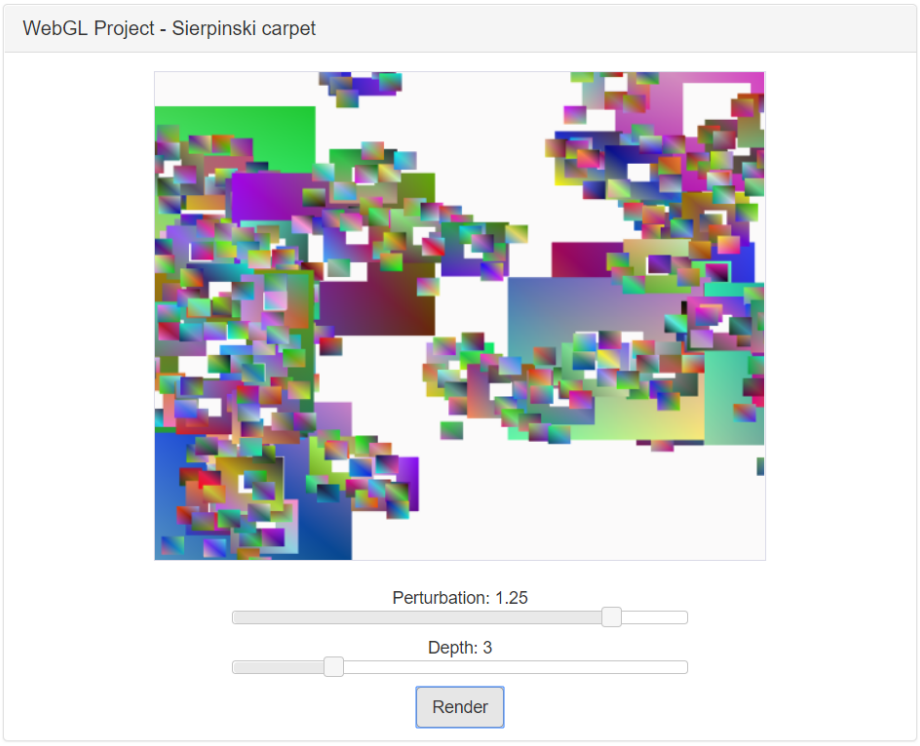


Rysunek 4: Interfejs użytkownika



Rysunek 5: Wyrenderowany podstawowy dywan Sierpińskiego, o bardzo niskim poziomie perturbacji

Rysunek 6: Wyrenderowany dywan Sierpińskiego, o średnim poziomie perturbacji oraz głębokości 4



Rysunek 7: Wyrenderowany dywan Sierpińskiego, o wysokim poziomie perturbacji oraz głębokości 3

Rysunek 8: Wyrenderowany dywan Sierpińskiego, o bardzo wysokim poziomie perturbacji oraz głębokości 5

5 Podsumowanie

Realizacja projektu pozwoliła mi nabyć wiedzę teoretyczną oraz praktyczną na temat technologii WebGL.

Literatura

[1] Instrukcja laboratoryjna. http://www.zsk.ict.pwr.wroc.pl/zsk/dyd/intinz/gk/lab/cw_8_dz/

[2] WebGL API. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

[3] Różnice WebGL i OpenGL. https://www.khronos.org/webgl/wiki/WebGL_and_OpenGL_Differences

[4] Kontekst graficzny. https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext