<table>
<tr><td>  <br>Politechnika Wrocławska</td><td>Imie, nazwisko oraz numer albumu studenta:<br><br>Hubert Twardowski 264446</td><td>Wydział: Telekomunikacji i informatyki<br><br>Rok:2023<br><br>Rok akadem.: 2022/23</td></tr>
<tr><td colspan="3">Grafika komputerowa i komunikacja człowiek-komputer</td></tr>
<tr><td>Data ćwiczenia: 29.11.2023</td><td rowspan="2"><u>Temat ćwiczenia laboratoryjnego:</u><br>Laboratorium 3 - Modelowanie obiektów 3D</td><td>Ocena:</td></tr>
<tr><td>Nr ćwiczenia: 3</td><td>Podpis prowadzącego:</td></tr>
</table>

# *Cele ćwiczeń*

Zrozumienie różnych sposobów definiowania modeli 3D
Nabranie wprawy w definiowaniu brył przy pomocy wierzchołków
Poznanie zasady działania mechanizmu bufora głębi

# Zadanie. 1

Treść: Zadaniem jest zadeklarowanie tablicy [N]x[N]x[3], wyznaczenie parametrów u i v (w przedziale 0.0 do 1.0, obliczenie dla każdej pary parametrów u i v wartości tablicy x,y,z.

W funkcji render() wyświetlić współrzędne oraz użycie prymitywu GL_POINTS

Wykonanie: Na początku zdefiniowaliśmy zmienną N oraz tablicę w języku *C++* w następujący sposób:

```
const int N = 100;

float vertices[N][N][3];
```

Aby wypełnić tablicę stworzyliśmy w pętle for, obliczyliśmy parametry u i v aby zaczynały się na 0.0 i kończyły na 1.0, wewnątrz pętli przypisaliśmy wartości tablicy oraz użyliśmy następujący wzór z wykładów

- Współrzędne wierzchołków można określić za pomocą układu równań,

$$x(u,v) = \left(-90 \cdot u^5 + 225 \cdot u^4 - 270 \cdot u^3 + 180 \cdot u^2 - 45 \cdot u\right) \cdot \cos(\pi \cdot v),$$
$$y(u,v) = 160 \cdot u^4 - 320 \cdot u^3 + 160 \cdot u^2 - 5,$$
$$z(u,v) = \left(-90 \cdot u^5 + 225 \cdot u^4 - 270 \cdot u^3 + 180 \cdot u^2 - 45 \cdot u\right) \cdot \sin(\pi \cdot v),$$

gdzie dziedziny $u$ i $v$ to przedziały $0 \leq u \leq 1$ oraz $0 \leq v \leq 1$.

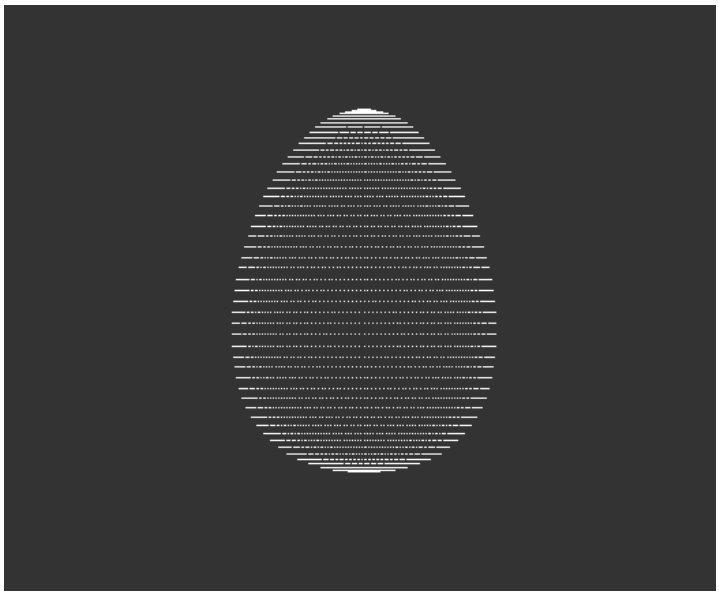
```
void startup() {
    glClearColor(0.2, 0.2, 0.2, 1.0);
    for (int i = 0; i < N; ++i) {
        float u = static_cast<float>(i) / static_cast<float>(N - 1);
        for (int j = 0; j < N; ++j) {
            float v = static_cast<float>(j) / static_cast<float>(N - 1);
            vertices[i][j][0] = ((-90 * pow(u, 5)) + (225 * pow(u, 4)) - (270 * pow(u, 3)) + (180 * pow(u, 2)) - (45 * u)) * (cos(M_PI * v));
            vertices[i][j][1] = (160 * pow(u, 4)) - (320 * pow(u, 3)) + (160 * pow(u, 2));
            vertices[i][j][2] = ((-90 * pow(u, 5)) + (225 * pow(u, 4)) - (270 * pow(u, 3)) + (180 * pow(u, 2)) - (45 * u)) * (sin(M_PI * v));
        }
    }
}
```

Następnie aby narysować punkty oraz wypisać współrzędne również stworzyliśmy pętlę, tym razem pojedyńczą wywołującą funkcje glVertex3f(), rysującą każdy element tablicy oraz funkcję printf do wypisywania współrzędnych.

```
void render(double time){
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBegin(GL_POINTS);
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            glVertex3fv(vertices[i][j]);
            printf("x %f : y %f : z %f \n",vertices[i][j][0], vertices[i][j][1], vertices[i][j][2]);
        }
    }
    glEnd();
    glFlush();
}
```

Rezultat:

Jajko narysowane za pomocą punktów w oknie:



Współrzędne wypisane w konsoli z której uruchomiono program

```
x -0.372072 : y 0.015682 : z 0.220043
x -0.378800 : y 0.015682 : z 0.208247
x -0.385154 : y 0.015682 : z 0.196246
x -0.391128 : y 0.015682 : z 0.184051
x -0.396716 : y 0.015682 : z 0.171675
x -0.401913 : y 0.015682 : z 0.159129
x -0.406713 : y 0.015682 : z 0.146426
x -0.411112 : y 0.015682 : z 0.133578
x -0.415105 : y 0.015682 : z 0.120599
x -0.418688 : y 0.015682 : z 0.107501
x -0.421858 : y 0.015682 : z 0.094296
x -0.424612 : y 0.015682 : z 0.080999
x -0.426947 : y 0.015682 : z 0.067622
x -0.428860 : y 0.015682 : z 0.054178
x -0.430350 : y 0.015682 : z 0.040680
x -0.431416 : y 0.015682 : z 0.027142
x -0.432055 : y 0.015682 : z 0.013578
```

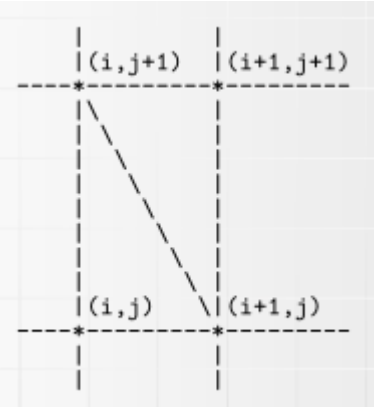
# Zadanie. 2

Treść: Zbudowanie modelu jajka przy pomocy linii, połączenie elementów sąsiadujących oraz obracanie jajka.

Wykonanie:

Modyfikujemy poprzedni program, zamiast GL_POINTS wykorzystujemy primityw GL_LINES.

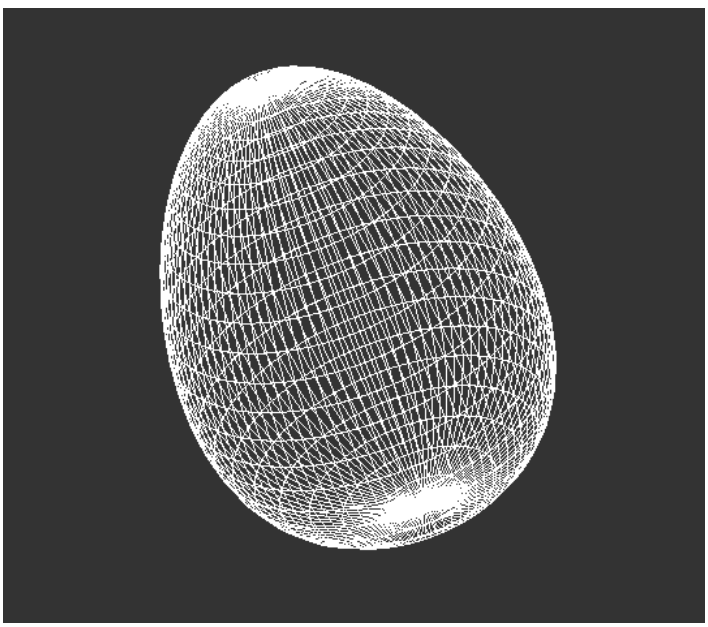
```
void render(double time)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glRotatef(static_cast<float>(time) * 40.f, 1, 1, 1);
    glBegin(GL_LINES);
    for (int i = 0; i < N; ++i)
    {
        for (int j = 0; j < N; ++j)
        {
            glVertex3fv(vertices[i][j]);
            glVertex3fv(vertices[i][j + 1]);
            glVertex3fv(vertices[i + 1][j]);
            glVertex3fv(vertices[i + 1][j]);
            glVertex3fv(vertices[i + 1][j + 1]);
            glVertex3fv(vertices[i][j + 1]);
        }
    }
    glEnd();
    glFlush();
}
```




Równierz łączymy sąsiadujące elementy, jest to dosyć proste gdyż postępujemy analogicznie do treści zadania czyli definiujemy wierzchołki w sposób jaki pokazany jest wyżej. W funkcji render() wywołujemy funkcję glRotatef(), przyjmuje ona 4 argumenty: kąt obrotu (w radianach) oraz zmienne x,y,z które definują koordynaty wektora obrotu. Aby funkcja ta zadziałała przy wywołaniu funkcji render() podajemy argument glfwGetTime() który wzraca czas tym samym inkrementując kąt obrotu.

```
while (!glfwWindowShouldClose(window))
{
    render(glfwGetTime());
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

Rezultat:



# Zadanie. 3

Treść: Zbudowanie modelu jajka przy pomocy trójkątów używając prymitywu GL_TRIANGLES, połączenie elementów (i,j) z dwoma sąsiednimi elementami oraz przypisanie każdemu z wierzchołków losowego koloru.

Wykonanie:

Aby uprościć przechowywanie kolorów tworzymy następujący struct, zawiera on 3 zmienne typu double, a po zdefiniowaniu owego structa tworzymy tablicę dla każdego wierzchołka (w naszym przypadku [N]x[N]x[6] gdyż każdy element jajka ma 6 wierzchołków lub 2 kwadraty.

```
typedef struct
{
    double values[3];
} Vertex3f;

Vertex3f color_rand[N][N][6];
```

```
srand(static_cast<unsigned>(time(0)));
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < N; j++)
    {
        for (int k = 0; k < 6; k++)
        {
            color_rand[i][j][k] = {random_normalized(), random_normalized(), random_normalized()};
        }
    }
}
```

W funkcji main wypełniamy tablicę losowymi kolorami za pomocą funkcji random_normalized() którą stworzyliśmy na poprzednich labolatoriach. Zwraca ona losowe wartości z zakresu 0.0 do 1.0

```
// Zmienna losowa normalizowana
float random_normalized()
{
    return static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
}
```

Podobnie do poprzedniego zadania rysujemy wierzchołki jednak dla każdego przydzielamy kolor funkcją glColor3f z parametrem z tablicy kolorów zdefiniowanej na początku pliku.

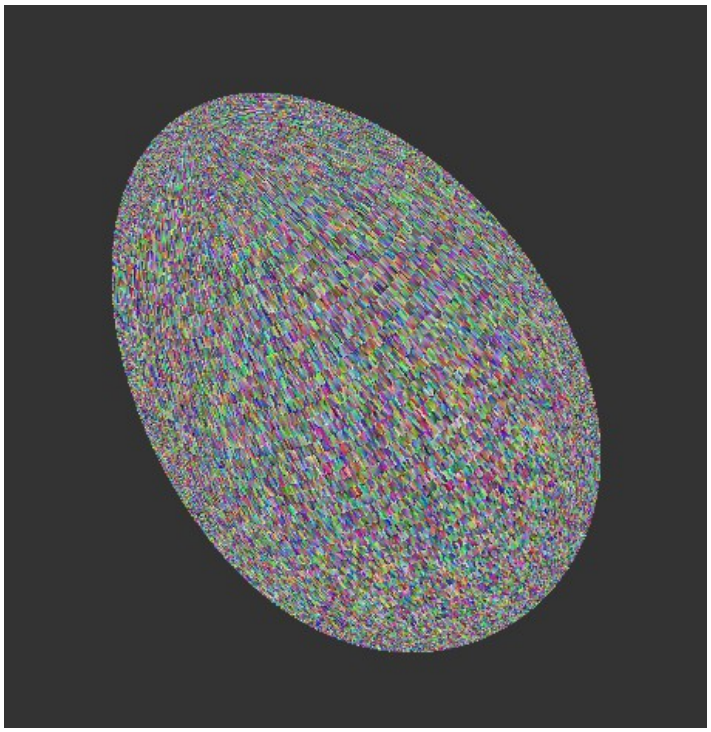
```
for (int i = 0; i < N - 1; ++i)
{
    for (int j = 0; j < N - 1; ++j)
    {
        glColor3f(color_rand[i][j][0].values[0], color_rand[i][j][0].values[1], color_rand[i][j][0].values[2]);
        glVertex3fv(vertices[i][j]);

        glColor3f(color_rand[i][j][1].values[0], color_rand[i][j][1].values[1], color_rand[i][j][1].values[2]);
        glVertex3fv(vertices[i][j + 1]);

        glColor3f(color_rand[i][j][2].values[0], color_rand[i][j][2].values[1], color_rand[i][j][2].values[2]);
        glVertex3fv(vertices[i + 1][j]);

        glColor3f(color_rand[i][j][3].values[0], color_rand[i][j][3].values[1], color_rand[i][j][3].values[2]);
        glVertex3fv(vertices[i + 1][j]);

        glColor3f(color_rand[i][j][4].values[0], color_rand[i][j][4].values[1], color_rand[i][j][4].values[2]);
        glVertex3fv(vertices[i + 1][j + 1]);

        glColor3f(color_rand[i][j][5].values[0], color_rand[i][j][5].values[1], color_rand[i][j][5].values[2]);
        glVertex3fv(vertices[i][j + 1]);
    }
}
```

Aby jednak uniknąć migotania wywołujemy 2 (3 opcjonalnie w zależności od systemu operacyjnego):
glfwWindowHint(GLFW_DEPTH_BITS, 32) – Przydzielamy 32 bity dla buforu głębi
glEnable(GL_DEPTH_TEST) – Włączamy depth testing. Wykonuje on test wartości fragmentu z buforem głębi i jeśli jest on poprawny to fragment jest rysowany.
glDisable(GL_CULL_FACE) – Wyłączamy przycinanie elementów które są ‘schowane’ za innymi elementami

```
glfwWindowHint(GLFW_DEPTH_BITS, 32);
glEnable(GL_DEPTH_TEST);
glDisable(GL_CULL_FACE);
```

Ważne jest też aby kolory które przydzielamy naszemu modelowi były jednolite czyli nie zmieniały się w trakcie działania programu. Jest to też przyczyna migotania.

Rezultat:



# Wnioski:

W drodze aby zaimplementować poprawny model jajka napotkaliśmy wiele problemów. Jednymi z nich były znikające tylne ściany (BACK FACE CULLING) przy braku włączonego DEPTH TESTINGU. Problemem był również tzw Z FIGHTING który można było rozwiązać przydzieleniem większej ilości bitów dla bufora głębi. W drodze aby uzyskać spójny model elipsy musieliśmy pokonać problem z implementacją algorytmu gdyż u i v powinny zaczynać na 0.0 i kończyć na 1.0 bądź w przeciwnym wypadku będą ‘ucięte na pół’, kolejnym problemem były złe indeksy, wystarczył błąd w byle jakiej pętli a na ekranie mogliśmy widzieć wszelakie graficzne błędy. Jednak nauczyliśmy się najprzeróżniejszych prymitywów w OpenGL czy też obsługi bufora głębi który jest ważny przy implementowaniu modelów 3D. Prezentacja dostarczona na laboratorium oraz dokumentacja OpenGL była wystarczającym źródłem wiedzy aby wykonać powieżone nam zadania.