Grafika komputerowa

Sprawozdanie

Damian Koper, 241292

2 grudnia 2019

Spis treści

1 OpenGL - podstawy **4**

1.1 Bazowa aplikacja 4

1.2 Dywan Sierpińskiego 5

2 Modelowanie obiektów 3D **9**

2.1 Bazowa aplikacja 9

2.2 Model i generowanie punktów 10

2.3 Rysowanie modelu za pomocą punktów, siatki i jako bryły 12

2.4 Rysowanie siatki i bryły 13

2.5 Animacja obrotu 14

3 Interakcja z użytkownikiem **15**

3.1 Perspektywa 15

3.2 Transformacje widoku 16

4 Oświetlenie **16**

4.1 Oświetlenie imbryczka 18

Spis rysunków

1 Dywan Sierpińskiego po 6 iteracjach. 5

2 Częściowo narysowany dywan Sierpińskiego po zakończeniu 30 gałęzi rekurencji. . . 6

3 Dywan Sierpińskiego narysowany w całości. 6

4 Dywan Sierpińskiego po 3 iteracjach z losowym kolorem zielonym oraz perturbacjami. 8

5 Dywan Sierpińskiego po 3 iteracjach z losowym kolorem zielonym, perturbacjami oraz w innej skali. 8

6 Model jajka zbudowany z punktów. 13

7 Model jajka zbudowany z siatki. 13

8 Model jajka zbudowany z trójkątów. 13

9 Widok imbryczka z zastosowaną zmianą obrotu i skali modelu, zmianą położenia kamery, oświetlonego dwoma źródłami światła. 18

10 Imbryczek w innej pozycji, widziany z innego punktu. Barwa światła znajdującego się bliżej kamery została zmieniona na zieloną. 19

Spis listingów

1 Bazowy program wyświetlający czarne okno. 4
2 Funkcja rysująca dywan Sierpińskiego rekurencyjnie. Pominięto niektóre wywołania. 6
3 Struktura danych przechowująca aktualny stan rysowania. 7
4 Funkcja rysująca dywan Sierpińskiego iteracyjnie. Pominięto niektóre wywołania. . 7
5 Interfejs IView. 9
6 Tworzeznie instancji widoków i ustawianie obecnego. Funkcja `g` zwraca instancję klasy. 9
7 Nagłówek klasy Point. 10
8 Nagłówek klasy modelu Egg. 11
9 Rysowanie jajka z punktów. Widok `DottEggView` 12
10 Rysowanie jajka z punktów. Model `Egg` 12
11 Ustawianie zmiennej kątu obrotu i przerysowanie klatki. 14
12 Metody interfejsu IView do obsługi zdarzeń myszy. 15
13 Definicja i wywołanie metody `gluLookAt`. 15
14 Metoda `apply` klasy materiału. 17
15 Metody inicjujące światło na scenie. 17

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.

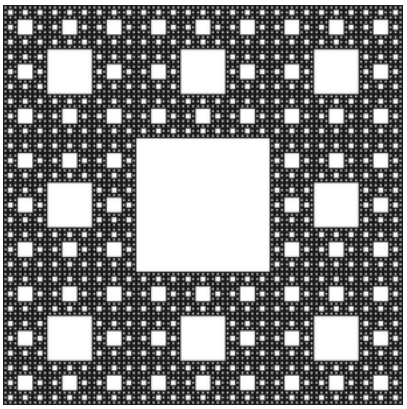
```
#include <GL/glut.h>
void draw()
{
  glClearColor(0, 0, 0, 1);
  glClear(GL_COLOR_BUFFER_BIT);
  glFlush();
}

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

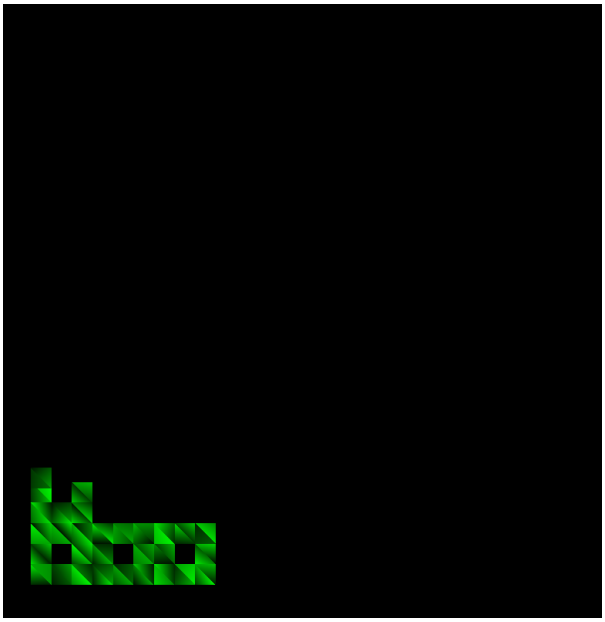
Dywan Sierpińskiego jest jest fraktalem otrzymanym z kwadratu podzielonego na 9 mniejszych kwadratów, z których usuwany jest środkowy. Procedura ta jest rekurencyjnie powtarzana dla pozostałych ośmiu kwadratów.



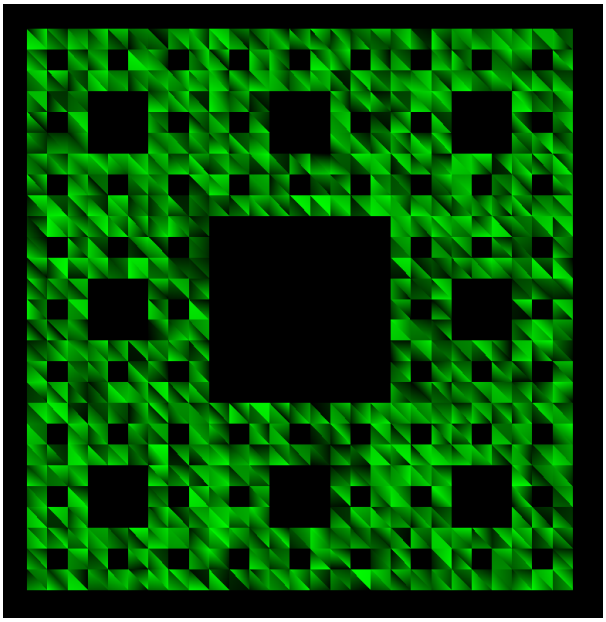
Rysunek 1: Dywan Sierpińskiego po 6 iteracjach.

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.

```
void carpet(int levels, float a, float dx = 0, float dy = 0)
{
  if (levels != 0)
  {
    a = a / 3;
    carpet(levels - 1, a, dx - a, dy - a);
    ...
    carpet(levels - 1, a, dx + a, dy + a);
  }
  else
  {
    //Pomocnicza funkcja rysujaca kwadrat
    rect(dx, dy, a);
  }
}
```

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.

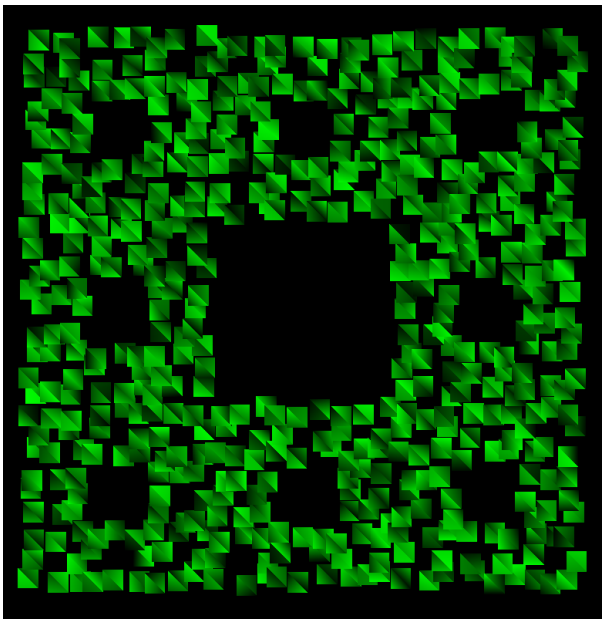
```
void carpetIt(int levels, float a)
{
  queue<CarpetLevelData> q = queue<CarpetLevelData>();
  q.push({levels, a});
  while (!q.empty())
  {
    CarpetLevelData data = q.front();
    if (data.level > 0)
    {
      data.a = data.a / 3;
      q.push({data.level - 1, data.a, data.dx - data.a, data.dy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

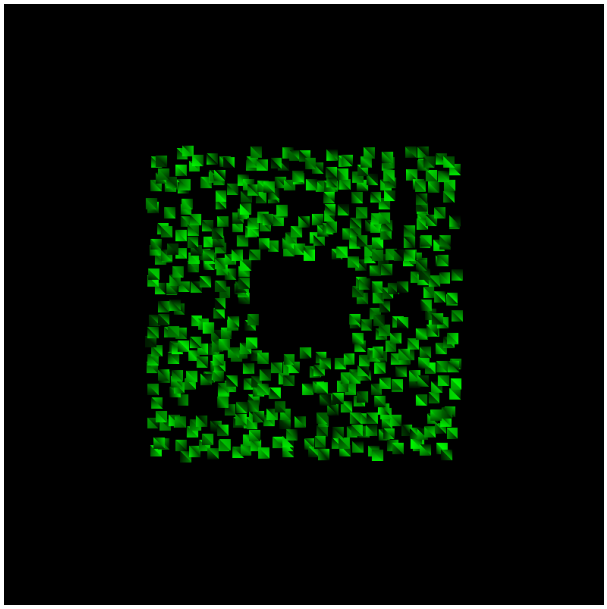
1.2.3 Losowe kolory, perturbacje i skalowanie

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 interfejs `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: Interfejs 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

2.2.1 Generowanie punktów

W ćwiczeniu wygenerowane zostały punkty modelu jajka. Uzyskane zostały poprzez obrócenie odpowiednio dobranej krzywej Beziera. Punkty opisują wzory w dziedzinie parametrycznej kwadratu jednostkowego:

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

Generowanie punktów dla wartości parametrów funkcji z przedziału $< 0; 1 >$ zakłada wygenerowanie nakładających się punktów. Dlatego w programie pominięto ostatnie iteracje pętli.

2.2.2 Model

Każdy model zdefiniowany jest za pomocą klasy, która przyjmuje parametry określające jego wygląd i zachowania. W przypadku klasy `Egg` jest to ilość iteracji pętli generowania punktów, co przekłada się na poziom wygładzenia modelu. Klasa modelu odpowiedzialna jest również za jego rysowanie w różnych wariantach.

Dla czytelności kodu stworzona została klasa `Point`, która przechowuje współrzędne punktu, jego kolor, oraz odpowiedzialna jest rysowanie samego siebie.

```
1  class Point
2  {
3  public:
4      Point(float x, float y, float z, GLubyte r, GLubyte g, GLubyte b)
5          : color({r, g, b}), x(x), y(y), z(z){};
6      ~Point();
7      void callGlVertex3f();
8      void callGlColor3f();
9      void drawWithColor();
10     struct Color
11     {
12         GLubyte r = 255;
13         GLubyte g = 255;
14         GLubyte b = 255;
```

```
    };
    float x = 0;
    float y = 0;
    float z~= 0;
    Color color;
};
```

Listing 7: Nagłówek klasy Point.

Właściwa generacja punktów odbywa się w konstruktorze klasy `Egg`.

```
class Egg
{
public:
    Egg(int n = 32);
    ~Egg();
    std::vector<std::vector<Point>> getPoints();
    void renderPoints();
    void renderMesh();
    void renderTriangles();
    void renderComplex();
private:
    int n;
    std::vector<std::vector<Point>> points;
    float calcX(float u, float v);
    float calcY(float u, float v);
    float calcZ(float u, float v);
};
```

Listing 8: Nagłówek klasy modelu Egg.

2.3 Rysowanie modelu za pomocą punktów, siatki i jako bryły

W utworzonej aplikacji rysowanie odbywa się w metodzie `IView::render()` modelu. Znajduje się tam również opisana niżej procedura transformacji i animacji modelu.

2.3.1 Rysowanie punktów

Rysowanie punktów obywa się z wykorzystanie prymitywu `GL_POINTS` biblioteki *OpenGL*. Wszystkie punkty podawane są do funkcji w dowolnej kolejności.

```
void DotEggView::render()
{
    glLoadIdentity();
    glRotated(eggRotation, 1, 1, 1);
    glTranslated(0, -5., 0);
    glPointSize(10.);
    egg.renderPoints();
}
```

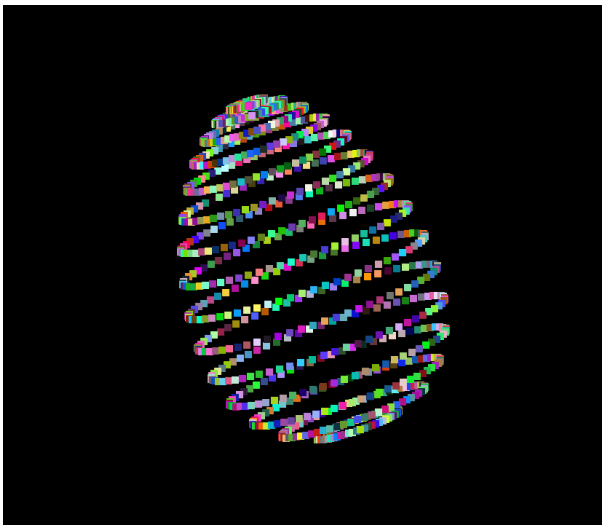
Listing 9: Rysowanie jajka z punktów. Widok `DottEggView`

```
void Egg::renderPoints()
{
    glBegin(GL_POINTS);
    for (auto &&row : points)
    {
        for (auto &&point : row)
        {
            point.drawWithColor();
        }
    }
    glEnd();
}
```

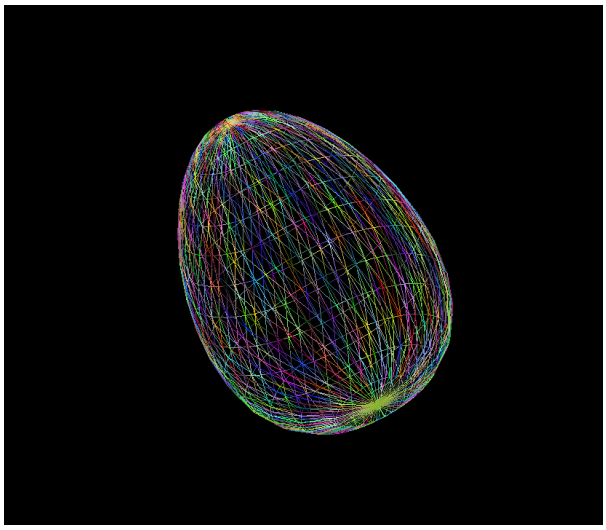
Listing 10: Rysowanie jajka z punktów. Model `Egg`

2.4 Rysowanie siatki i bryły

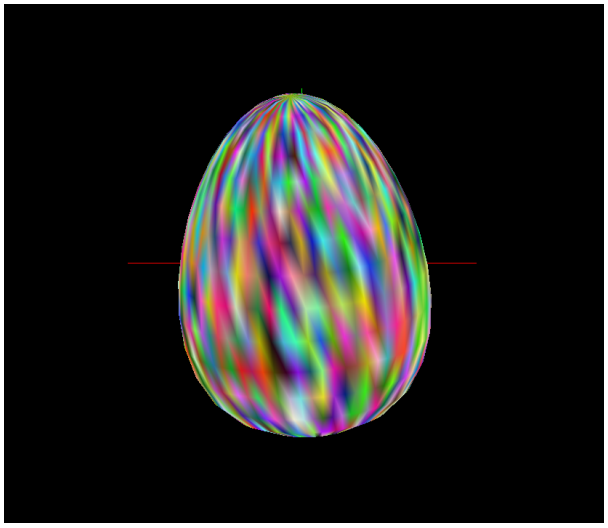
Rysowanie siatki i bryły odbywa się analogicznie do rysowania punktów. Jedyną różnicą jest użyty prymityw, oraz kolejność umieszczania punktów, która różni się w zależności od prymitywu. Dla siatki użyty został prymityw `GL_LINES`. Bryła została narysowana na dwa sposoby używając prymitywów `GL_TRIANGLES` i `GL_TRIANGLE_STRIP`, co po narysowaniu daje taki sam efekt.



Rysunek 6: Model jajka zbudowany z punktów.



Rysunek 7: Model jajka zbudowany z siatki.



Rysunek 8: Model jajka zbudowany z trójkątów.

W przypadku rysowania za pomocą siatki wymagane było ujednolicenie losowo generowanych kolorów dla punktów, które w procesie ich generowania się na siebie nakładają. Ma to miejsce na czubkach oraz, w przypadku generowania dla całego zakresu $< 0; 1 >$, dla jednego z *południków* jajka.

2.5 Animacja obrotu

Zmiana kąta obrotu modelu odbywa się w metodzie `IView::timer()` widoku, która wywoływana jest 60 razy na sekundę korzystając z funkcji `void glutTimerFunc(unsigned int time, void (*callback)( int), int value)`. Zapewnia to większą kontrolę nad animacją i jej prędkością niż podanie wskaźnika do funkcji `void glutIdleFunc(void (*callback)())`. Metoda widoku inkrementuje zmienną definiującą kąt obrotu modelu dla rysowanej klatki. W listingu 9, podczas rysowania, przeprowadzony jest proces transformacji modelu. Najpierw resetowana jest macierz transformacji, potem dokonywany jest obrót o wartość ze zmiennej widoku wokół wszystkich osi. Translacja w osi Y ustawia model *mniej więcej* po środku ekranu.

```
1 void DotEggView::timer()
2 {
3     eggRotation += 1;
4     if (eggRotation >= 360)
5         eggRotation = 0;
6     glutPostRedisplay();
7 }
```

Listing 11: Ustawianie zmiennej kątu obrotu i przerysowanie klatki.

3 Interakcja z użytkownikiem

W celu poszerzenia możliwości obsługi akcji użytkownika o zdarzenia myszy do interfejsu `IView` opisanego na listingu 5 dodano metody z listingu 12.

```
void onMouse(int btn, int state, int x, int y);
void onMotion(GLsizei x, GLsizei y) = 0;
```

Listing 12: Metody interfejsu IView do obsługi zdarzeń myszy.

W ogólnym przypadku metoda `onMouse` zapisywała stan, klawisz przycisku myszy i jej pozycję, a metoda `onMotion` odpowiadała za wykonywanie akcji odpowiedzialnej za ruch myszy.

3.1 Perspektywa

Za wyświetlenie perspektywiczne odpowiada metoda `gluPerspective`, która zastąpiła metodę `gluOrtho` odpowiedzialną za wyświetlanie ortograficzne w metodzie `changeSize`. Widok obserwatora definiuje metoda `gluLookAt`, do której poprzez parametry podane zostają pozycje kamery, celu i jej kierunku obrotu. W metodzie `render` widoku `TeapotView` dodane zostało wywołanie metody `gluLookAt`, które argumenty czerpie ze zmiennych klasy widoku, co daje możliwość dowolnego sterowania pozycją obserwatora.

```
void gluLookAt(
    GLdouble eyeX, GLdouble eyeY, GLdouble eyeZ,
    GLdouble centerX, GLdouble centerY, GLdouble centerZ,
    GLdouble upX, GLdouble upY, GLdouble upZ
);
//---
gluLookAt(
    eye.x + center.x,eye.y + center.y, eye.z + center.z,
    center.x, center.y, center.z,
    0.0, 0.0, 1.0
);
```

Listing 13: Definicja i wywołanie metody `gluLookAt`.

3.2 Transformacje widoku

Transformacje widoku podzielono na trzy tryby, które obsługiwała metoda `onMotion` w zależności od przyciśniętego klawisza.

3.2.1 Obrót i skalowanie modelu

Rysowany model imbryczka po naciśnięciu lewego przycisku myszy i jej przeciągnięciu jest obracany w osi Z i X oraz skalowany w przypadku naciśniętego prawego przycisku myszy. Użyto w tym celu parametryzowane transformacje `glTranslate` i `glRotate`. Aktualny kąty obrotu i skala są zapisywane w zmiennych klasy widoku i aktualizowana o ostatnią różnicę pozycji myszy.

3.2.2 Obrót i zmiana pozycji kamery

Pozycja kamery zapisywana jest jako zmienne określające środek sfery w położeniu C, o promieniu R, jej azymut Θ i kąt elewacji Φ. Pozycja kamery w układzie współrzędnych wyliczana jest ze wzorów 1.

$$x(\Theta, \Phi) = R \cdot cos(\Theta) \cdot cos(\Phi) + C_x \tag{1a}$$

$$y(\Theta, \Phi) = R \cdot sin(\Theta) \cdot cos(\Phi) + C_y \tag{1b}$$

$$z(\Theta, \Phi) = R \cdot sin(\Phi) + C_z \tag{1c}$$

Aplikacja pozwala na manipulowanie kątami, promieniem, przemieszczeniem oraz odległością kamery od środka sfery.

3.2.3 Obrót i zmiana pozycji świateł

Światło reprezentowane jest jako klasa `Light`. Światła, podobnie jak kamera, rozmieszczone są na powierzchni sfery ze środkiem w centrum układu współrzędnych. W trybie manipulacji źródłem światła można zmienić położenie jednego z dwóch świateł.

4 Oświetlenie

W celu odpowiedniego oddziaływania źródeł światła z powierzchnią modeli zdefiniowano dla nich materiał reprezentowany przez klasę `Material`. Zawiera ona parametry `ambient`, `diffuse`, `specular` i `shininess`, definiujące kolejno współczynniki dla światła otoczenia, rozproszonego, odbitego oraz

połysk powierzchni. Materiał używany jest poprzez wywołanie metody `apply` pokazaną na listingu 14.

```
void Material::apply()
{
  glMaterialfv(GL_FRONT, GL_SPECULAR, specular);
  glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse);
  glMaterialf(GL_FRONT, GL_SHININESS, shininess);
}
```

Listing 14: Metoda `apply` klasy materiału.

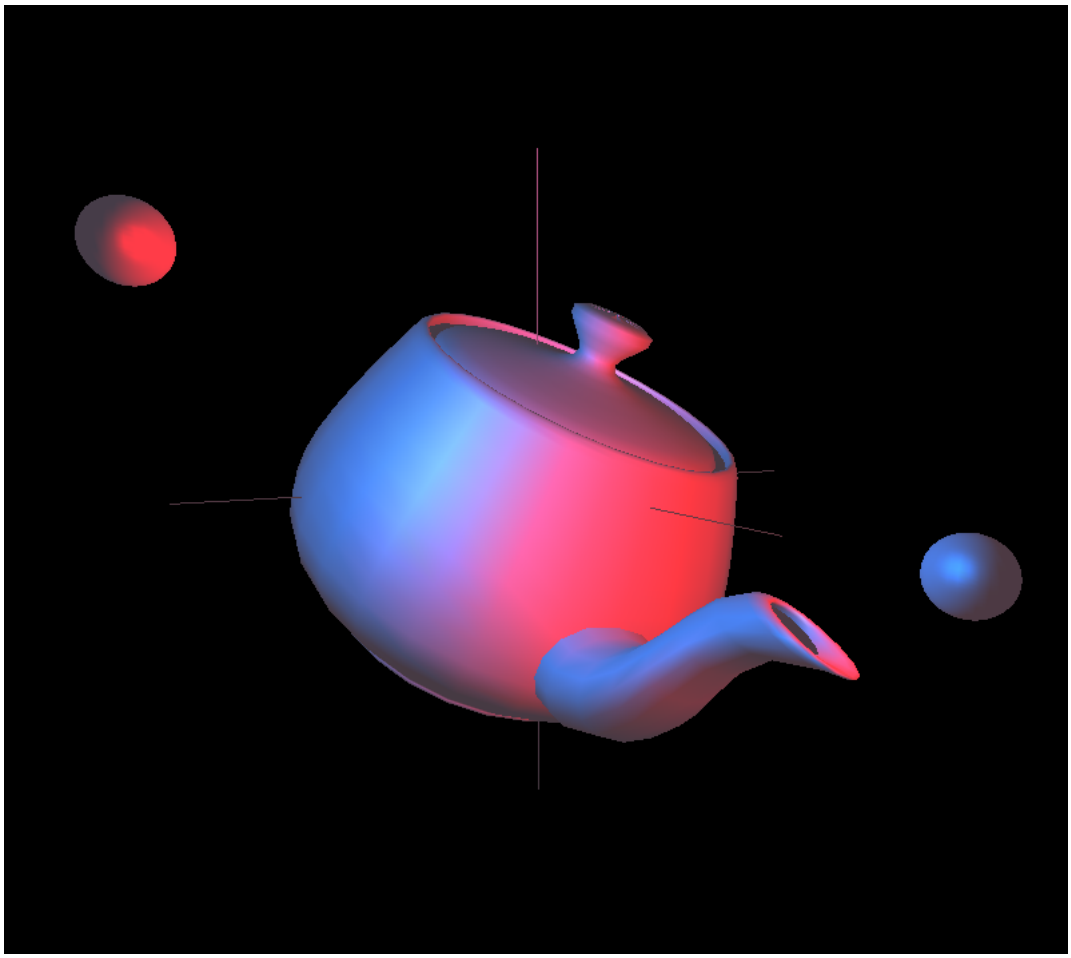
```
void Light::calcPosition()
{
    position[0] = rDistance * cos(azimuth) * cos(elevation);
    position[1] = rDistance * sin(azimuth) * cos(elevation);
    position[2] = rDistance * sin(elevation);
    glLightfv(n, GL_POSITION, position);
}
void Light::calcColor()
{
    ambient[0] = color.r / 255. * 0.1;
    ...
    specular[2] = color.b / 255.;
    glLightfv(n, GL_AMBIENT, ambient);
    glLightfv(n, GL_DIFFUSE, diffuse);
    glLightfv(n, GL_SPECULAR, specular);
}
void Light::init(int n)
{
  this->n = n;
  calcColor(); calcPosition();
  glLightf(n, GL_CONSTANT_ATTENUATION, constant);
  glLightf(n, GL_LINEAR_ATTENUATION, linear);
  glLightf(n, GL_QUADRATIC_ATTENUATION, quadratic);
  glEnable(n);
}
```

Listing 15: Metody inicjujące światło na scenie.

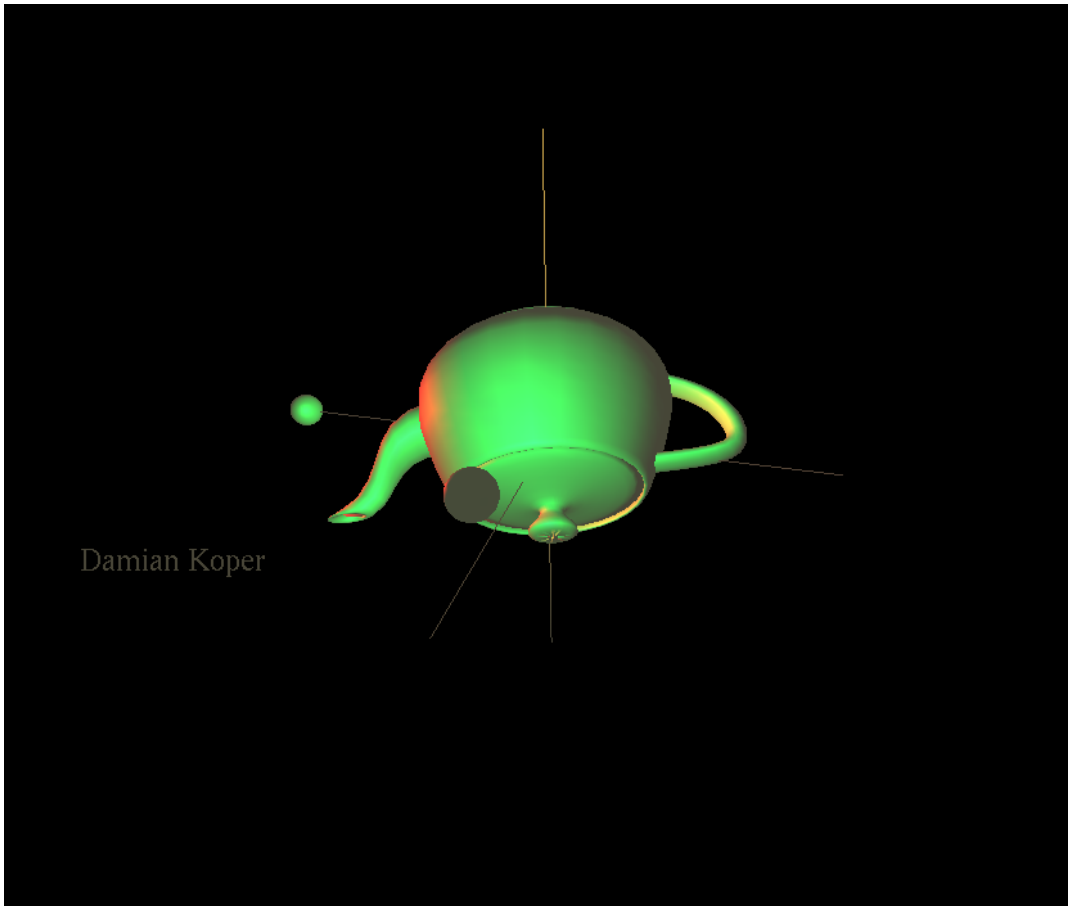
Klasę `Light` opisuje siedem parametrów: pozycja światła, współczynniki świecenia źródła światła otoczenia, światła powodującego odbicie dyfuzyjne i odbicie kierunkowe. Składają się na nie jeszcze składowa stała, liniowa i kwadratowa opisujące zmiany oświetlenia w zależności od odległości od jego źródła. Listing 15 przedstawia metody obliczające pozycję światła na powierzchni sfery, kolor dla wszystkich składowych oraz metody inicjujące te parametry na scenie.

4.1 Oświetlenie imbryczka

Na scenie został umieszczony imbryczek razem z dwoma źródłami światła. Ponieważ domyślnie nie są one reprezentowane przez żaden widzialny obiekt, ich pozycja została pokazana za pomocą sfer.



Rysunek 9: Widok imbryczka z zastosowaną zmianą obrotu i skali modelu, zmianą położenia kamery, oświetlonego dwoma źródłami światła.



Rysunek 10: Imbryczek w innej pozycji, widziany z innego punktu. Barwa światła znajdującego się bliżej kamery została zmieniona na zieloną.