МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

Отчет
по лабораторной работе №3
по дисциплине «КОМПЬЮТЕРНАЯ 3D-ГРАФИКА»
Тема: Освещение.

Студентка гр. 5381 _____ Буздина М.А.

Преподаватель _____ Герасимова Т.В.

Санкт-Петербург
2019

Цель работы.

Представить 3D сцену и в ней один или несколько объектов из вашего задания (стараться представить все). Добавить освещение к сцене.

Задание.

1. Определить трехмерный объект на основе простой геометрической фигуры (задается преподавателем)
2. Добавьте нормали к трехмерному объекту, чтобы сделать его подходящим для освещения
3. Добавить второй источник света
4. Добавить позиционный и направленный источник света
5. Попробуйте сформировать затенение по Гуро и Фонгу, сравните их.
6. Узнайте больше о взаимодействии материалов и свойств освещения и примените это на вашей сцене.

Основные теоретические сведения.

Детали языка затенения

Векторные компоненты. Векторы, которые вы отправляете в свои шейдеры, обычно представляют цвета, координаты вершин, нормаль поверхности и координаты текстуры. Поскольку компоненты этих векторов имеют разные значения, GLSL предоставляет специальные средства доступа, которые можно использовать для ссылки на компоненты.

- r, g, b, a используется для цветов. красный, зеленый, синий, альфа (коэффициент смешивания)
- x, y, z, w используется для пространственных координат, таких как векторы и точки.
- s, t, p, q используется для поиска текстур.

Затенение.

Если задано плоское затенение, одна вершина выбирается как репрезентативная для всех вершин; таким образом, весь примитив отображается одним цветом. Для всех примитивов это последняя указанная вершина в каждом многоугольнике или отрезке. Плоское затенение задается ключевым словом flat перед типом данных на выходе из вашего вершинного шейдера и соответствующим входом для вашего фрагментного шейдера.

Ваш вершинный шейдер может выводить больше, чем просто цвет и положение вершин. Любой атрибут вершины может быть интерполирован по мере его отправки в ваш фрагментный шейдер. Если вы решили отправлять и интерполировать только цвета, вы делаете затенение по Гуро. Вы также можете интерполировать нормали и выполнять расчеты освещения в фрагментном шейдере, а не в вершинном шейдере. Это называется затенением Фонг.

Модели освещения

В модели закрашивания, часто используемой в компьютерной графике, предполагается, что объекты сцены освещаются двумя типами источников: точечными источниками света и фоновым светом. Эти источники света "сверкают" на различных поверхностях объектов, и падающий свет взаимодействует с поверхностью одним из трех возможных способов:

- некоторая часть поглощается поверхностью и превращается в тепло;
- некоторая часть отражается от поверхности;
- некоторая часть проходит внутрь объекта, как в случае куска стекла.

Различают два типа отражения падающего света:

- диффузное рассеяние
- зеркальные отражения

Геометрические составляющие отраженного света

Для того чтобы вычислить диффузный и зеркальный компоненты света, необходимо найти три вектора.

На рис. 1 показаны три главных вектора, необходимых для нахождения количества света, попадающего в глаз из точки P:

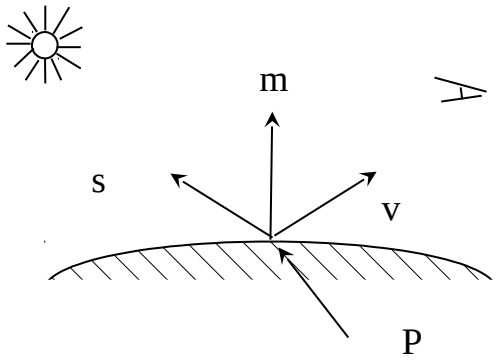

Рисунок 1

- нормаль m к поверхности в точке P;
- вектор v, соединяющий точку P с глазом наблюдателя;
- вектор s, соединяющий точку P с источником света

Углы между этими тремя векторами составляют основу для вычисления интенсивностей освещения. Обычно эти углы вычисляются в мировых координатах, поскольку при некоторых преобразованиях (таких, как перспективное преобразование) углы не сохраняются.

Ход работы.

Определим 4 трехмерных объекта: куб, сферу, конус и цилиндр. Для них рассчитаем нормали и текстурные координаты (см. Приложение 1). На рис. 2 представлены эти объекты без освещения.

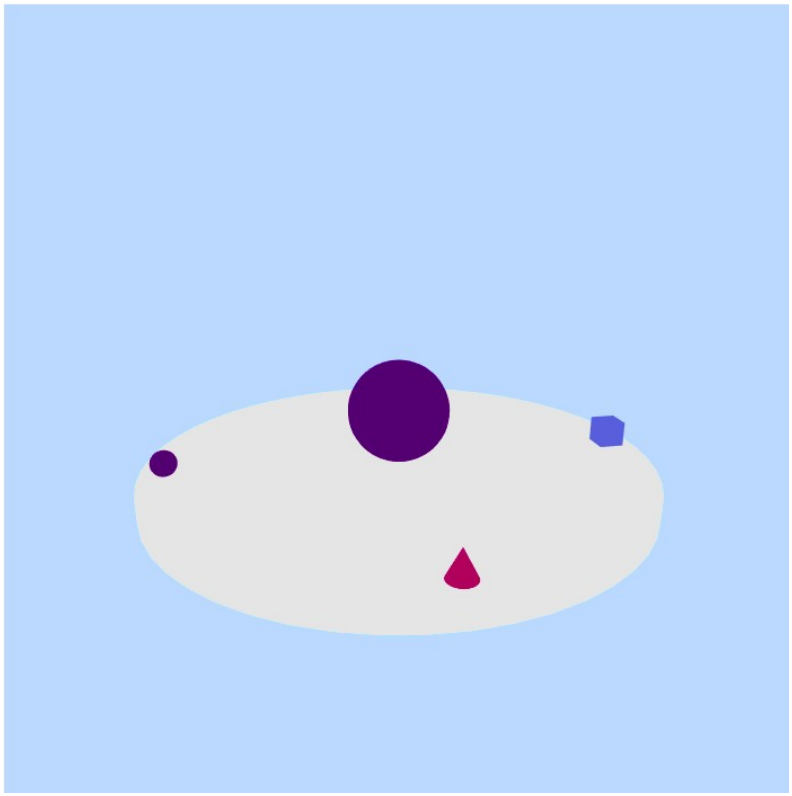

Рисунок 2 — 3D объекты. Без освещения.

Добавим позиционное освещение. На рисунке 3 представлена сцена с только позиционным освещением.

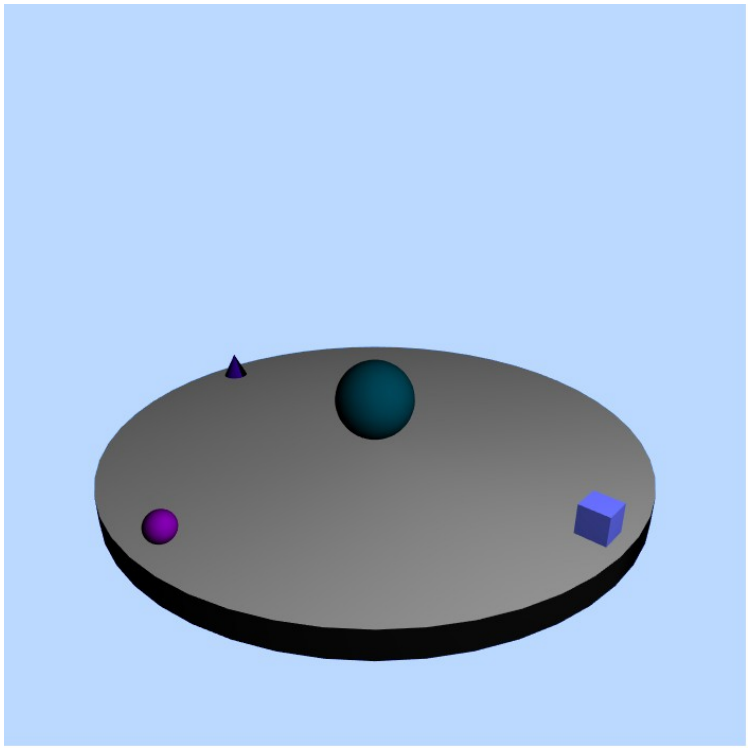

Рисунок 3 — 3D объекты. Позиционное освещение.

Добавим фоновое освещение к позиционному. На рисунке 4 представлена сцена с позиционным и фоновым освещением. В приложении 2 представлен код шейдеров.

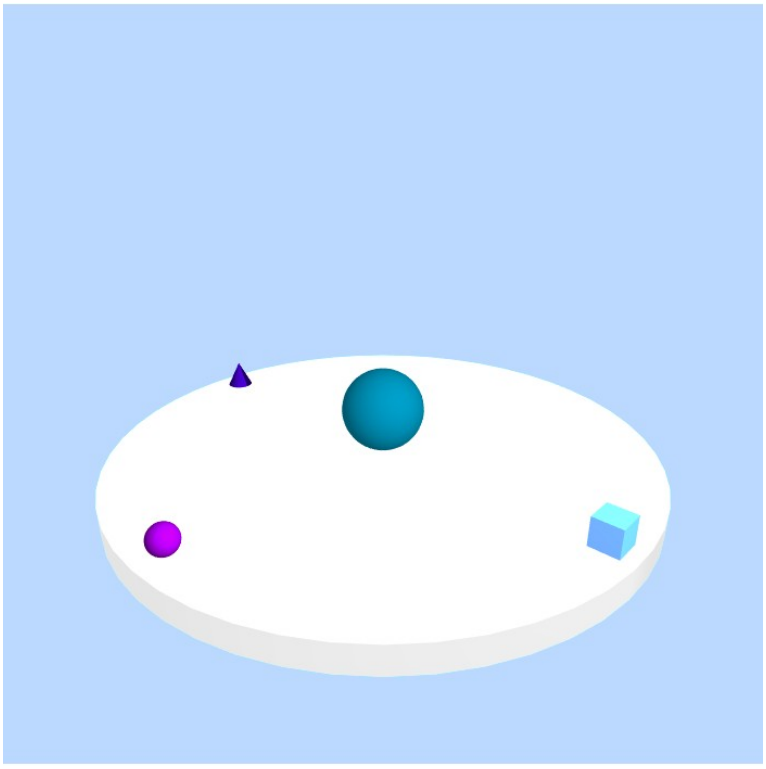

Рисунок 4 — 3D объекты. Позиционное и фоновое освещения.

Изменим цветовые составляющие освещения (см. рис. 5-6)

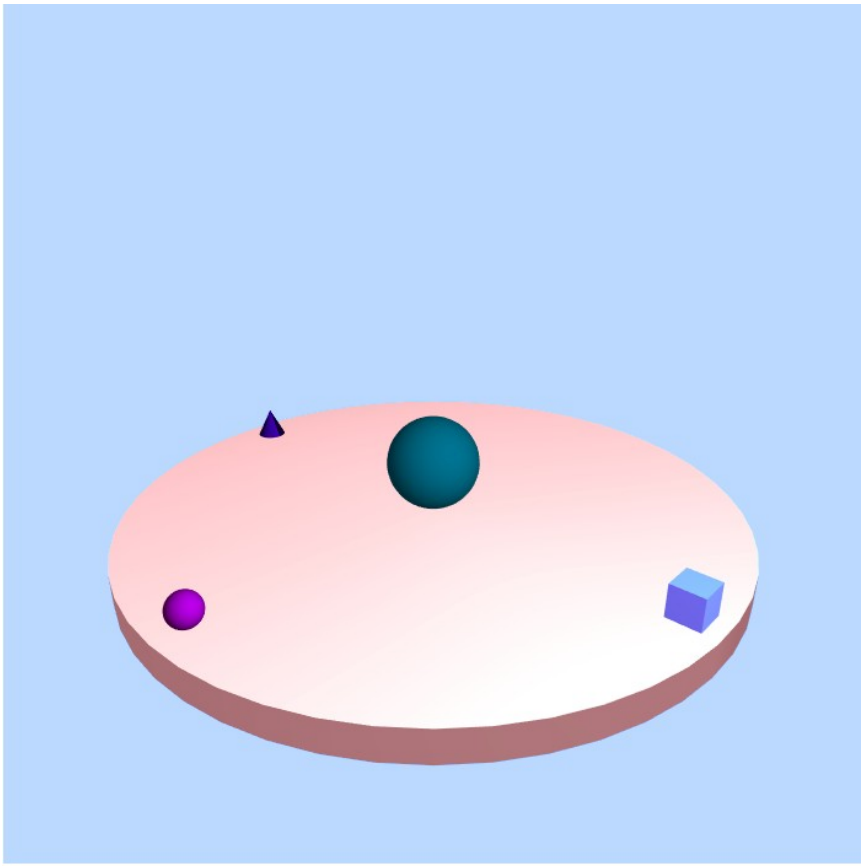

Рисунок 5 — 3D объекты. Позиционное красное освещение.

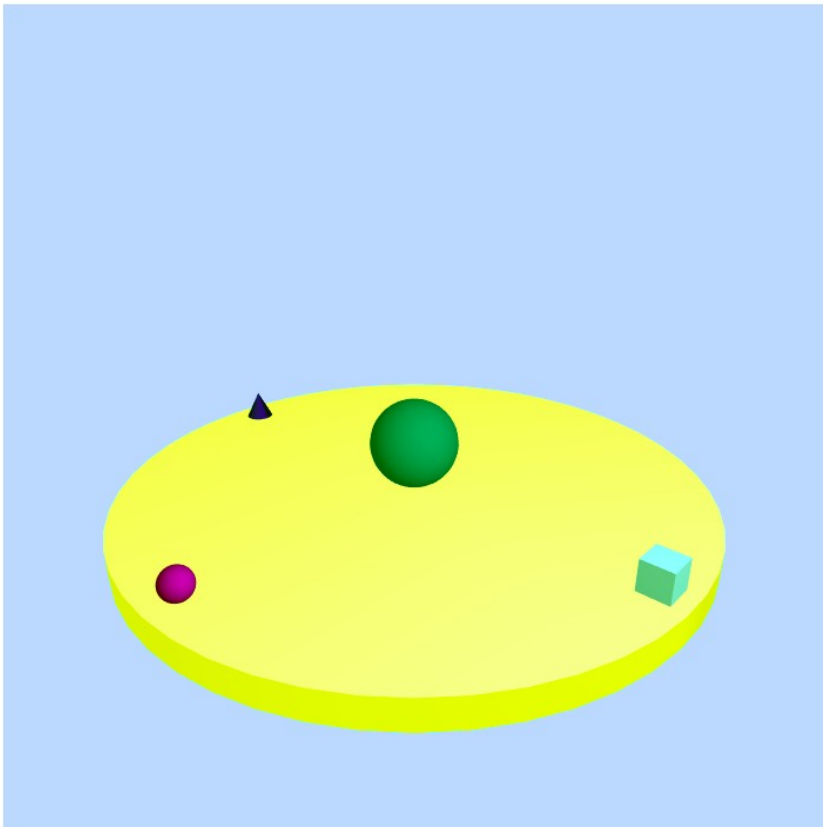

Рисунок 6 — 3D объекты. Позиционное красное освещение и фоновое синее.

Выводы.

В ходе данной работы было определены несколько трехмерных объектов с финальной сцены. Для них были рассчитаны нормали, для корректного отражения света. Были добавлены фоновое и позиционное освещение.

Приложение 1

Cube.js

```
class Cube extends Figure {
    constructor(center, angle, scale, color) {
        super(center, angle, scale, color);
        this.x = 1;
        this.y = 1;
        this.z = 1;
    }
    initBuffers() {
        this.generateVerticesMatrix();
        this.generateTextureCoords();
        this.generateNormalesMatrix();
        this.generateIndexesMatrix();
        this.initPositionBuffer();
        this.initTextureCoordsBuffer();
        this.initIndexBuffer();
        this.initNormalesBuffer();
    }
    generateNormalesMatrix(){
        this.vertexNormales = [
            // Front face
            0.0,  0.0,  1.0,
            0.0,  0.0,  1.0,
            0.0,  0.0,  1.0,
            0.0,  0.0,  1.0,
            // Back face
            0.0,  0.0, -1.0,
            0.0,  0.0, -1.0,
            0.0,  0.0, -1.0,
            0.0,  0.0, -1.0,
            // Top face
            0.0,  1.0,  0.0,
            0.0,  1.0,  0.0,
            0.0,  1.0,  0.0,
            0.0,  1.0,  0.0,
            // Bottom face
            0.0, -1.0,  0.0,
            0.0, -1.0,  0.0,
            0.0, -1.0,  0.0,
            0.0, -1.0,  0.0,
            // Right face
            1.0,  0.0,  0.0,
            1.0,  0.0,  0.0,
            1.0,  0.0,  0.0,
            1.0,  0.0,  0.0,
            // Left face
            -1.0,  0.0,  0.0,
            -1.0,  0.0,  0.0,
            -1.0,  0.0,  0.0,
            -1.0,  0.0,  0.0,
        ];
    }
    generateTextureCoords(){
        this.textureCoords = [
            // Front face
            0.0, 0.0,
            1.0, 0.0,
            1.0, 1.0,
```

```
            0.0, 1.0,
            // Back face
            1.0, 0.0,
            1.0, 1.0,
            0.0, 1.0,
            0.0, 0.0,
            // Top face
            0.0, 1.0,
            0.0, 0.0,
            1.0, 0.0,
            1.0, 1.0,
            // Bottom face
            1.0, 1.0,
            0.0, 1.0,
            0.0, 0.0,
            1.0, 0.0,
            // Right face
            1.0, 0.0,
            1.0, 1.0,
            0.0, 1.0,
            0.0, 0.0,
            // Left face
            0.0, 0.0,
            1.0, 0.0,
            1.0, 1.0,
            0.0, 1.0,
        ];
    }
    // Special color like on picture
    generateColorMatrix() {
        let colors = [];
        for (let i=0; i < this.vertices.length / 3; i++) {
            colors = colors.concat(this.color);
        }
        this.colors = colors;
    }
    generateVerticesMatrix() {
        this.vertices = [
            // Front face
            -this.x, -this.y, this.z,
            this.x, -this.y, this.z,
            this.x, this.y, this.z,
            -this.x, this.y, this.z,
            // Back face
            -this.x, -this.y, -this.z,
            -this.x, this.y, -this.z,
            this.x, this.y, -this.z,
            this.x, -this.y, -this.z,
            // Top face
            -this.x, this.y, -this.z,
            -this.x, this.y, this.z,
            this.x, this.y, this.z,
            this.x, this.y, -this.z,
            // Bottom face
            -this.x, -this.y, -this.z,
            this.x, -this.y, -this.z,
            this.x, -this.y, this.z,
            -this.x, -this.y, this.z,
            // Right face
            this.x, -this.y, -this.z,
            this.x, this.y, -this.z,
```

```
            this.x, this.y, this.z,
            this.x, -this.y, this.z,
            // Left face
            -this.x, -this.y, -this.z,
            -this.x, -this.y, this.z,
            -this.x, this.y, this.z,
            -this.x, this.y, -this.z,
        ];
    }
    generateIndexesMatrix(){
        this.indices = [
            0, 1, 2,      0, 2, 3,    // Front face
            4, 5, 6,      4, 6, 7,    // Back face
            8, 9, 10,     8, 10, 11,  // Top face
            12, 13, 14,   12, 14, 15, // Bottom face
            16, 17, 18,   16, 18, 19, // Right face
            20, 21, 22,   20, 22, 23  // Left face
        ]
    }
}
```

Cone.js

```
class Cone extends Figure {
    constructor(center, angle, scale, color) {
        super(center, angle, scale, color);
        this.radius = 1;
        this.height = 1;
        this.heightSegments = 1;
        this.radialSegments = 50;
        this.index = 0;
    }
    initBuffers() {
        //this.generateCap(true);// крышка цилиндра верх
        this.generateCap(false); // крышка цилинда низ
        this.generateTorso(); // цилиндр
        this.initPositionBuffer();
        this.initIndexBuffer();
        this.initTextureCoordsBuffer();
        this.initNormalesBuffer();
    }
    generateTorso() {
        let x, y;
        let indexArray = [];
        let halfHeight = this.height / 2;
        let slope = (this.radius)/this.height;
        // this will be used to calculate the normal
        // generate vertices, normals and uvs
        for (y = 0; y <= this.heightSegments; y++) {
            let indexRow = [];
            let v = y / this.heightSegments;
            let radius = v * this.radius;
            // calculate the radius of the current row
            //let radius = v * (radiusBottom - radiusTop) + radiusTop;
            for (x = 0; x <= this.radialSegments; x++) {
                let u = x / this.radialSegments;
                let theta = u * 2 * Math.PI;
                let sinTheta = Math.sin(theta);
                let cosTheta = Math.cos(theta);
                // vertex
```

```
                this.vertices.push(radius * sinTheta, -v * this.height +
halfHeight, radius * cosTheta);
                // normal
                let length = Math.pow(Math.pow(Math.sqrt(sinTheta), 2) +
Math.pow(Math.sqrt(cosTheta), 2), 1/2)
                this.vertexNormales.push(sinTheta/length, 0, cosTheta/length);
                this.textureCoords.push( u, 1 - v );
                // save index of vertex in respective row
                indexRow.push(this.index++);
            }
            // now save vertices of the row in our index array
            indexArray.push(indexRow);
        }
        // generate indices
        for (x = 0; x < this.radialSegments; x++) {
            for (y = 0; y < this.heightSegments; y++) {
                // we use the index array to access the correct indices
                let a = indexArray[y][x];
                let b = indexArray[y + 1][x];
                let c = indexArray[y + 1][x + 1];
                let d = indexArray[y][x + 1];
                // faces
                this.indices.push(a, b, d);
                this.indices.push(b, c, d);
            }
        }
    }
    generateCap(top) {
        let x, centerIndexStart, centerIndexEnd;
        let halfHeight = this.height / 2;
        let sign = (top === true) ? 1 : -1;
        let thetaStart = 0.0;
        let thetaLength = 2 * Math.PI;
        // save the index of the first center vertex
        centerIndexStart = this.index;
        // first we generate the center vertex data of the cap.
        // because the geometry needs one set of uvs per face,
        // we must generate a center vertex per face/segment
        for (x = 1; x <= this.radialSegments; x++) {
            // vertex
            this.vertices.push(0, halfHeight * sign, 0);
            // normal
            this.vertexNormales.push(0, sign, 0);
            this.textureCoords.push( 0.5, 0.5);
            // increase index
            this.index++;
        }
        // save the index of the last center vertex
        centerIndexEnd = this.index;
        // now we generate the surrounding vertices, normals and uvs
        for (x = 0; x <= this.radialSegments; x++) {
            let u = x / this.radialSegments;
            let theta = u * thetaLength + thetaStart;
            let cosTheta = Math.cos(theta);
            let sinTheta = Math.sin(theta);
            // vertex
            this.vertices.push(this.radius * sinTheta, halfHeight * sign,
this.radius * cosTheta);
            // normal
            this.textureCoords.push( cosTheta * 0.5  + 0.5, sinTheta * 0.5 *
sign  + 0.5 );
```

```
            this.vertexNormales.push(0, sign, 0);
            // increase index
            this.index++;
        }
        // generate indices
        for (x = 0; x < this.radialSegments; x++) {
            let c = centerIndexStart + x;
            let i = centerIndexEnd + x;
            if (top === true) {
                // face top
                this.indices.push(i, i + 1, c);
            }
            else {
                // face bottom
                this.indices.push(i + 1, i, c);
            }
        }
    }
}
```

Cylinder.js

```
class Cylinder extends Figure {
    constructor(center, angle, scale, color) {
        super(center, angle, scale, color);
        this.radius = 1;
        this.height = 1;
        this.heightSegments = 1;
        this.radialSegments = 50;
        this.index = 0;
    }
    initBuffers() {
        this.generateCap(true);// крышка цилиндра верх
        this.generateCap(false); // крышка цилинда низ
        this.generateTorso(); // цилиндр
        this.initPositionBuffer();
        this.initIndexBuffer();
        this.initTextureCoordsBuffer();
        this.initNormalesBuffer();
    }
    generateTorso() {
        let x, y;
        let indexArray = [];
        let halfHeight = this.height / 2;
        // this will be used to calculate the normal
        // generate vertices, normals and uvs
        for (y = 0; y <= this.heightSegments; y++) {
            let indexRow = [];
            let v = y / this.heightSegments;
            // calculate the radius of the current row
            //let radius = v * (radiusBottom - radiusTop) + radiusTop;
            for (x = 0; x <= this.radialSegments; x++) {
                let u = x / this.radialSegments;
                let theta = u * 2 * Math.PI;
                let sinTheta = Math.sin(theta);
                let cosTheta = Math.cos(theta);
                // vertex
                this.vertices.push(this.radius * sinTheta, -v * this.height +
halfHeight, this.radius * cosTheta);
                // normal
```

```
                let length = Math.pow(Math.pow(Math.sqrt(sinTheta), 2) +
Math.pow(Math.sqrt(cosTheta), 2), 1/2)
                this.vertexNormales.push(sinTheta/length, 0, cosTheta/length);
                this.textureCoords.push( u, 1 - v );
                // save index of vertex in respective row
                indexRow.push(this.index++);
            }
            // now save vertices of the row in our index array
            indexArray.push(indexRow);
        }
        // generate indices
        for (x = 0; x < this.radialSegments; x++) {
            for (y = 0; y < this.heightSegments; y++) {
                // we use the index array to access the correct indices
                let a = indexArray[y][x];
                let b = indexArray[y + 1][x];
                let c = indexArray[y + 1][x + 1];
                let d = indexArray[y][x + 1];
                // faces
                this.indices.push(a, b, d);
                this.indices.push(b, c, d);
            }
        }
    }
    generateCap(top) {
        let x, centerIndexStart, centerIndexEnd;
        let halfHeight = this.height / 2;
        let sign = (top === true) ? 1 : -1;
        let thetaStart = 0.0;
        let thetaLength = 2 * Math.PI;
        // save the index of the first center vertex
        centerIndexStart = this.index;
        // first we generate the center vertex data of the cap.
        // because the geometry needs one set of uvs per face,
        // we must generate a center vertex per face/segment
        for (x = 1; x <= this.radialSegments; x++) {
            // vertex
            this.vertices.push(0, halfHeight * sign, 0);
            // normal
            this.vertexNormales.push(0, sign, 0);
            this.textureCoords.push( 0.5, 0.5);
            // increase index
            this.index++;
        }
        // save the index of the last center vertex
        centerIndexEnd = this.index;
        // now we generate the surrounding vertices, normals and uvs
        for (x = 0; x <= this.radialSegments; x++) {
            let u = x / this.radialSegments;
            let theta = u * thetaLength + thetaStart;
            let cosTheta = Math.cos(theta);
            let sinTheta = Math.sin(theta);
            // vertex
            this.vertices.push(this.radius * sinTheta, halfHeight * sign,
this.radius * cosTheta);
            // normal
            this.textureCoords.push( cosTheta * 0.5  + 0.5, sinTheta * 0.5 *
sign  + 0.5 );
            this.vertexNormales.push(0, sign, 0);
            // increase index
            this.index++;
```

```
        }
        // generate indices
        for (x = 0; x < this.radialSegments; x++) {
            let c = centerIndexStart + x;
            let i = centerIndexEnd + x;
            if (top === true) {
                // face top
                this.indices.push(i, i + 1, c);
            }
            else {
                // face bottom
                this.indices.push(i + 1, i, c);
            }
        }
    }
}
```

Sphere.js

```
class Sphere extends Figure {

    constructor(center, radius, angle,  scale,  color) {
        super(center, angle,  scale, color);
        this.radius = radius;
        this.latitudeBands = 30;
        this.longitudeBands = 30;
    }
    initBuffers(){
        this.generateVerticesMatrix();
        this.generateIndexesMatrix();
        this.initPositionBuffer();
        this.initIndexBuffer();
        this.initTextureCoordsBuffer();
        this.initNormalesBuffer();
        this.initColorBuffer();
    }
    generateColorMatrix(){
        let colors = [];
        for (let i=0; i < this.vertices.length / 3; i++) {
            colors = colors.concat(this.color);
        }
        this.colors = colors;
    }
    generateIndexesMatrix(){
        for (let latNumber = 0; latNumber < this.latitudeBands; latNumber++) {
            for (let longNumber = 0; longNumber < this.longitudeBands;
longNumber++) {
                let first = (latNumber * (this.longitudeBands + 1)) +
longNumber;
                let second = first + this.longitudeBands + 1;
                this.indices.push(first);
                this.indices.push(second);
                this.indices.push(first + 1);
                this.indices.push(second);
                this.indices.push(second + 1);
                this.indices.push(first + 1);
            }
        }
    }
    generateVerticesMatrix(){
        for (let latNumber=0; latNumber <= this.latitudeBands; latNumber++) {
            let theta = latNumber * Math.PI / this.latitudeBands;
```

```
            let sinTheta = Math.sin(theta);
            let cosTheta = Math.cos(theta);
            for (let longNumber=0; longNumber <= this.longitudeBands;
longNumber++) {
                let phi = longNumber * 2 * Math.PI / this.longitudeBands;
                let sinPhi = Math.sin(phi);
                let cosPhi = Math.cos(phi);
                let x = cosPhi * sinTheta;
                let y = cosTheta;
                let z = sinPhi * sinTheta;
                let u = 1 - (longNumber / this.longitudeBands);
                let v = 1 - (latNumber / this.latitudeBands);
                this.vertexNormales.push(x);
                this.vertexNormales.push(y);
                this.vertexNormales.push(z);
                this.textureCoords.push(u);
                this.textureCoords.push(v);
                this.vertices.push(this.radius * x);
                this.vertices.push(this.radius * y);
                this.vertices.push(this.radius * z);
            }
        }
    }
}
```

Приложение 2

Вершинный шейдер

```
  <script id="shader-vs" type="x-shader/x-vertex">
    attribute vec3 aVertexPosition;
    attribute vec3 aVertexNormal;
    attribute vec2 aTextureCoord;
    uniform mat4 uMVMatrix;
    uniform mat4 uPMatrix;
    uniform mat3 uNMatrix;
    uniform vec3 uAmbientColor;
    uniform vec3 uPointLightingLocation;
    uniform vec3 uPointLightingColor;
    uniform bool uUseLighting;
    varying vec2 vTextureCoord;
    varying vec3 vLightWeighting;
    void main(void) {
        vec4 mvPosition = uMVMatrix * vec4(aVertexPosition, 1.0);
        gl_Position = uPMatrix * mvPosition;
        vTextureCoord = aTextureCoord;
        if (!uUseLighting) {
            vLightWeighting = vec3(1.0, 1.0, 1.0);
        } else {
            vec3 lightDirection = normalize(uPointLightingLocation -
mvPosition.xyz);
            vec3 transformedNormal = uNMatrix * aVertexNormal;
            float directionalLightWeighting = max(dot(transformedNormal,
lightDirection), 0.0);
            vLightWeighting = uAmbientColor + uPointLightingColor *
directionalLightWeighting;
        }
    }
</script>
```