ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Департамент программной инженерии

СОГЛАСОВАНО
Профессор
департамента программной инженерии
кандидат технических наук

_____ Е.М. Гринкруг
«___» _____ 2018 г.

УТВЕРЖДАЮ
Академический руководитель
образовательной программы
«Программная инженерия»

_____ В.В. Шилов
«___» _____ 2018 г.

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подл	RU.17701729.03.05-01

РЕАЛИЗАЦИЯ ПОДМНОЖЕСТВА СТАНДАРТА ТРЕХМЕРНОЙ ГРАФИКИ СРЕДСТВАМИ БИБЛИОТЕКИ WEBGL

Текст программы

ЛИСТ УТВЕРЖДЕНИЯ

RU.17701729.03.05-01 12 01-1-ЛУ

Исполнитель:
студентка группы БПИ162
_____ / Казанцева А.Р. /
«____»_____ 2018 г.

2018

УТВЕРЖДЕНО
RU.17701729.03.05-01 12 01-1-ЛУ

РЕАЛИЗАЦИЯ ПОДМНОЖЕСТВА СТАНДАРТА ТРЕХМЕРНОЙ ГРАФИКИ СРЕДСТВАМИ БИБЛИОТЕКИ WEBGL

Текст программы

RU.17701729.03.05-01 12 01-1

Листов 44

Инв. № подл	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата
RU.17701729.03.05-01				

2018

СОДЕРЖАНИЕ

Аннотация

В данном документе приведет текст курсовой работы «Реализация подмножества стандарта трехмерной графики средствами библиотеки WebGL», результатом которой является библиотека, написанная на языке программирования JavaScript.

Библиотека предназначена для предоставления возможности работы с 3D-графикой непосредственно средствами Web-браузера, без установки каких-либо иных специальных программных средств.

Настоящий документ разработан в соответствии с требованиями:

1) ГОСТ 19.101-77 Виды программ и программных документов [1];

2) ГОСТ 19.102-77 Стадии разработки [2];

3) ГОСТ 19.103-77 Обозначения программ и программных документов [3];

4) ГОСТ 19.104-78 Основные надписи [4];

5) ГОСТ 19.105-78 Общие требования к программным документам [5];

6) ГОСТ 19.106-78 Требования к программным документам, выполненным печатным способом[6];

7) ГОСТ 19.401-78 Текст программы. Требования к содержанию и оформлению [7].

Изменения к данному документу оформляются согласно ГОСТ 19.603-78 [8], ГОСТ 19.604-78 [9].

1 Текст программы

Программа состоит из девяти основных и трех вспомогательных классов, а также дополнительного автономного файла с кодом для WebWorker[1]

Текст основной программы:

```
/*
* Библиотека для удобного декларирование простейших 3D объектов без
использования плагинов.
*
* Технологии: WebGL
*
* Казанцева Анастасия Романовна
*
* Выполнено в рамках Курсовой работы НИУ ВШЭ ФКН ПИ.
*
* 2018 г.
* */
```

Описание класса Algebra

```
/**
 * Класс вспомогательных алгебраических функций, совершающий операции над
матрицами 4*4 и векторами 3.
 */
class Algebra {
    /**
     * Создает матрицу переноса.
     * @param tx {number} Перенос по оси X.
     * @param ty {number} Перенос по оси Y.
     * @param tz {number} Перенос по оси Z.
     * @returns {number[]} Матрица переноса.
     */
    static translation(tx, ty, tz) {
        return [
            1, 0, 0, 0,
            0, 1, 0, 0,
            0, 0, 1, 0,
            tx, ty, tz, 1,
        ];
    }

    /**
     * Создает матрицу поворота по оси X.
     * @param angleInRadians {number} Угол поворота в радианах по оси X.
     * @returns {number[]} Матрица поворота.
     */
    static xRotation(angleInRadians) {
        let c = Math.cos(angleInRadians);
        let s = Math.sin(angleInRadians);

        return [
            1, 0, 0, 0,
            0, c, s, 0,
            0, -s, c, 0,
```

[1] Программный интерфейс, позволяющий запускать на WEB-странице фоновые задачи, не влияющие на производительность страницы.

```
            0, 0, 0, 1,
        ];
    }

    /**
     * Создает матрицу поворота по оси Y.
     * @param angleInRadians {number} Угол поворота в радианах по оси Y.
     * @returns {number[]} Матрица поворота.
     */
    static yRotation(angleInRadians) {
        let c = Math.cos(angleInRadians);
        let s = Math.sin(angleInRadians);

        return [
            c, 0, -s, 0,
            0, 1, 0, 0,
            s, 0, c, 0,
            0, 0, 0, 1,
        ];
    }

    /**
     * Создает матрицу поворота по оси Z.
     * @param angleInRadians {number} Угол поворота в радианах по оси Z.
     * @returns {number[]} Матрица поворота.
     */
    static zRotation(angleInRadians) {
        let c = Math.cos(angleInRadians);
        let s = Math.sin(angleInRadians);

        return [
            c, s, 0, 0,
            -s, c, 0, 0,
            0, 0, 1, 0,
            0, 0, 0, 1,
        ];
    }

    /**
     * Создает матрицу масштабирования.
     * @param sx {number} Масштаб по оси X.
     * @param sy {number} Масштаб по оси Y.
     * @param sz {number} Масштаб по оси Z.
     * @returns {number[]} Матрица масштабирования.
     */
    static scaling(sx, sy, sz) {
        return [
            sx, 0, 0, 0,
            0, sy, 0, 0,
            0, 0, sz, 0,
            0, 0, 0, 1,
        ];
    }

    /**
     * Выполняет перенос матрицы на заданные значения.
     * @param m {number[]} Исходная матрица
     * @param tx {number} Перенос по оси X.
```

```
     * @param ty {number} Перенос по оси Y.
     * @param tz {number} Перенос по оси Z.
     * @returns {number[]} Результирующая матрица.
     */
    static translate(m, tx, ty, tz) {
        return Algebra.multiply(m, Algebra.translation(tx, ty, tz));
    }

    /**
     * Выполняет поворот матрицы по оси X.
     * @param m {number[]} Исходная матрица.
     * @param angleInRadians {number} Угол поворота по оси X в радианах.
     * @returns {number[]} Результирующая матрица.
     */
    static xRotate(m, angleInRadians) {
        return Algebra.multiply(m, Algebra.xRotation(angleInRadians));
    }

    /**
     * Выполняет поворот матрицы по оси Y.
     * @param m {number[]} Исходная матрица.
     * @param angleInRadians {number} Угол поворота по оси Y в радианах.
     * @returns {number[]} Результирующая матрица.
     */
    static yRotate(m, angleInRadians) {
        return Algebra.multiply(m, Algebra.yRotation(angleInRadians));
    }

    /**
     * Выполняет поворот матрицы по оси Z.
     * @param m {number[]} Исходная матрица.
     * @param angleInRadians {number} Угол поворота по оси Z в радианах.
     * @returns {number[]} Результирующая матрица.
     */
    static zRotate(m, angleInRadians) {
        return Algebra.multiply(m, Algebra.zRotation(angleInRadians));
    }

    /**
     * Выполняет масшабирование матрицы на заданные параметры.
     * @param m {number[]} Исходная матрица
     * @param sx {number} Масштабирование по оси X.
     * @param sy {number} Масштабирование по оси Y.
     * @param sz {number} Масштабирование по оси Z.
     * @returns {number[]} Результирующая матрица.
     */
    static scale(m, sx, sy, sz) {
        return Algebra.multiply(m, Algebra.scaling(sx, sy, sz));
    }

    /**
     * Создает матрицу перспективы по заданным параметрам.
     * @param fieldOfViewInRadians {number} Угол отображения области
видимости в радианах.
     * @param aspect {number} Отношение ширины сцены к высоте.
     * @param near {number} Наименьший индекс, видимый камере, по Z.
     * @param far {number} Наибольший индекс, видимый камере, по Z.
     * @returns {number[]} Матрица перспективы.
```

```
     */
    static perspective(fieldOfViewInRadians, aspect, near, far) {
        let f = Math.tan(Math.PI * 0.5 - 0.5 * fieldOfViewInRadians);
        let rangeInv = 1.0 / (near - far);

        return [
            f / aspect, 0, 0, 0,
            0, f, 0, 0,
            0, 0, (near + far) * rangeInv, -1,
            0, 0, near * far * rangeInv * 2, 0
        ];
    }

    /**
     * Перемножает две матрицы в обратном порядке.
     * @param b {number[]} Матрица-правый множитель.
     * @param a {number[]} Матрица-левый множитель.
     * @returns {number[]} Результат матричного умножения.
     */
    static multiply(b, a) {
        let result = [];

        for (let i = 0; i < a.length / 4; i++) {
            let row = [];
            for (let j = 0; j < (b.length / 4); j++) {
                let sum = 0;
                for (let t = 0; t < 4; t++)
                    sum += a[4 * i + t] * b[(b.length / 4) * t + j];
                row.push(sum);
            }
            result = [...result, ...row];
        }

        return result;
    }

    /**
     * Создает единичную матрицу.
     * @returns {number[]} Единичная матрица.
     */
    static identity() {
        return [
            1, 0, 0, 0,
            0, 1, 0, 0,
            0, 0, 1, 0,
            0, 0, 0, 1
        ];
    }

    /**
     * Создает обратную матрицу.
     * @param m {number[]} Исходная матрица.
     * @returns {number[]} Матрица обратная исходной.
     */
    static inverse(m) {
        let m00 = m[0 * 4 + 0];
        let m01 = m[0 * 4 + 1];
        let m02 = m[0 * 4 + 2];
```

```
let m03 = m[0 * 4 + 3];
let m10 = m[1 * 4 + 0];
let m11 = m[1 * 4 + 1];
let m12 = m[1 * 4 + 2];
let m13 = m[1 * 4 + 3];
let m20 = m[2 * 4 + 0];
let m21 = m[2 * 4 + 1];
let m22 = m[2 * 4 + 2];
let m23 = m[2 * 4 + 3];
let m30 = m[3 * 4 + 0];
let m31 = m[3 * 4 + 1];
let m32 = m[3 * 4 + 2];
let m33 = m[3 * 4 + 3];
let tmp_0 = m22 * m33;
let tmp_1 = m32 * m23;
let tmp_2 = m12 * m33;
let tmp_3 = m32 * m13;
let tmp_4 = m12 * m23;
let tmp_5 = m22 * m13;
let tmp_6 = m02 * m33;
let tmp_7 = m32 * m03;
let tmp_8 = m02 * m23;
let tmp_9 = m22 * m03;
let tmp_10 = m02 * m13;
let tmp_11 = m12 * m03;
let tmp_12 = m20 * m31;
let tmp_13 = m30 * m21;
let tmp_14 = m10 * m31;
let tmp_15 = m30 * m11;
let tmp_16 = m10 * m21;
let tmp_17 = m20 * m11;
let tmp_18 = m00 * m31;
let tmp_19 = m30 * m01;
let tmp_20 = m00 * m21;
let tmp_21 = m20 * m01;
let tmp_22 = m00 * m11;
let tmp_23 = m10 * m01;

let t0 = (tmp_0 * m11 + tmp_3 * m21 + tmp_4 * m31) -
    (tmp_1 * m11 + tmp_2 * m21 + tmp_5 * m31);
let t1 = (tmp_1 * m01 + tmp_6 * m21 + tmp_9 * m31) -
    (tmp_0 * m01 + tmp_7 * m21 + tmp_8 * m31);
let t2 = (tmp_2 * m01 + tmp_7 * m11 + tmp_10 * m31) -
    (tmp_3 * m01 + tmp_6 * m11 + tmp_11 * m31);
let t3 = (tmp_5 * m01 + tmp_8 * m11 + tmp_11 * m21) -
    (tmp_4 * m01 + tmp_9 * m11 + tmp_10 * m21);

let d = 1.0 / (m00 * t0 + m10 * t1 + m20 * t2 + m30 * t3);

return [
    d * t0,
    d * t1,
    d * t2,
    d * t3,
    d * ((tmp_1 * m10 + tmp_2 * m20 + tmp_5 * m30) -
        (tmp_0 * m10 + tmp_3 * m20 + tmp_4 * m30)),
    d * ((tmp_0 * m00 + tmp_7 * m20 + tmp_8 * m30) -
        (tmp_1 * m00 + tmp_6 * m20 + tmp_9 * m30)),
```

```
            d * ((tmp_3 * m00 + tmp_6 * m10 + tmp_11 * m30) -
                (tmp_2 * m00 + tmp_7 * m10 + tmp_10 * m30)),
            d * ((tmp_4 * m00 + tmp_9 * m10 + tmp_10 * m20) -
                (tmp_5 * m00 + tmp_8 * m10 + tmp_11 * m20)),
            d * ((tmp_12 * m13 + tmp_15 * m23 + tmp_16 * m33) -
                (tmp_13 * m13 + tmp_14 * m23 + tmp_17 * m33)),
            d * ((tmp_13 * m03 + tmp_18 * m23 + tmp_21 * m33) -
                (tmp_12 * m03 + tmp_19 * m23 + tmp_20 * m33)),
            d * ((tmp_14 * m03 + tmp_19 * m13 + tmp_22 * m33) -
                (tmp_15 * m03 + tmp_18 * m13 + tmp_23 * m33)),
            d * ((tmp_17 * m03 + tmp_20 * m13 + tmp_23 * m23) -
                (tmp_16 * m03 + tmp_21 * m13 + tmp_22 * m23)),
            d * ((tmp_14 * m22 + tmp_17 * m32 + tmp_13 * m12) -
                (tmp_16 * m32 + tmp_12 * m12 + tmp_15 * m22)),
            d * ((tmp_20 * m32 + tmp_12 * m02 + tmp_19 * m22) -
                (tmp_18 * m22 + tmp_21 * m32 + tmp_13 * m02)),
            d * ((tmp_18 * m12 + tmp_23 * m32 + tmp_15 * m02) -
                (tmp_22 * m32 + tmp_14 * m02 + tmp_19 * m12)),
            d * ((tmp_22 * m22 + tmp_16 * m02 + tmp_21 * m12) -
                (tmp_20 * m12 + tmp_23 * m22 + tmp_17 * m02))
        ];
    }

    /**
     * Преобразует матрицу вида(4*4) в нормальную матрицу(3*3).
     *
     * @param a {number[]} Исходная матрица вида.
     * @returns {number[]} Нормальная матрица для исходной.
     */
    static normalFromMat4(a) {
        let out = [
            0, 0, 0,
            0, 0, 0,
            0, 0, 0
        ];

        let a00 = a[0], a01 = a[1], a02 = a[2], a03 = a[3];
        let a10 = a[4], a11 = a[5], a12 = a[6], a13 = a[7];
        let a20 = a[8], a21 = a[9], a22 = a[10], a23 = a[11];
        let a30 = a[12], a31 = a[13], a32 = a[14], a33 = a[15];

        let b00 = a00 * a11 - a01 * a10;
        let b01 = a00 * a12 - a02 * a10;
        let b02 = a00 * a13 - a03 * a10;
        let b03 = a01 * a12 - a02 * a11;
        let b04 = a01 * a13 - a03 * a11;
        let b05 = a02 * a13 - a03 * a12;
        let b06 = a20 * a31 - a21 * a30;
        let b07 = a20 * a32 - a22 * a30;
        let b08 = a20 * a33 - a23 * a30;
        let b09 = a21 * a32 - a22 * a31;
        let b10 = a21 * a33 - a23 * a31;
        let b11 = a22 * a33 - a23 * a32;

        // Подсчет детерминанта
        let det = b00 * b11 - b01 * b10 + b02 * b09 + b03 * b08 - b04 * b07 +
    b05 * b06;
```

```
        if (!det) {
            return null;
        }
        det = 1.0 / det;

        out[0] = (a11 * b11 - a12 * b10 + a13 * b09) * det;
        out[1] = (a12 * b08 - a10 * b11 - a13 * b07) * det;
        out[2] = (a10 * b10 - a11 * b08 + a13 * b06) * det;

        out[3] = (a02 * b10 - a01 * b11 - a03 * b09) * det;
        out[4] = (a00 * b11 - a02 * b08 + a03 * b07) * det;
        out[5] = (a01 * b08 - a00 * b10 - a03 * b06) * det;

        out[6] = (a31 * b05 - a32 * b04 + a33 * b03) * det;
        out[7] = (a32 * b02 - a30 * b05 - a33 * b01) * det;
        out[8] = (a30 * b04 - a31 * b02 + a33 * b00) * det;

        return out;
    }

    /**
     * Нормирует исходный вектор.
     * @param v {number[]} Исходный вектор.
     * @returns {number[]} Нормированный вектор.
     */
    static normalize(v) {
        let length = Math.sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
        // Проверяем, что мы не делим на 0.
        if (length > 0.00001) {
            return [v[0] / length, v[1] / length, v[2] / length];
        } else {
            return [0, 0, 0];
        }
    }
}
```

Описание класса Appearance

```
/**
 * Тег my-appearance. Определяет внешний вид фигуры.
 */
class Appearance {
    /**
     * Запускает инициализацию тега, описывающего цвет текущей формы.
     *
     * @param appearanceElement Ссылка на элемент, с которым мы в данный
момент работаем.
     * @param {Number} vertexCount Количество вершин, которые необходимо
покрасить.
     */
    static init(appearanceElement, vertexCount) {
        // Определяем, какой материал получен и запускаем его инициализацию.
        let material;
        if (material = appearanceElement.getElementsByTagName("my-color")[0])
            return new Color(material, vertexCount);
        else
            throw new Error("Отсутствует тег, задающий внешний вид формы! В
```

```
тег my-appearance необходимо добавить тег my-color.");
    }
}
```

Описание класса Box

```
/**
 * Тег my-box. Определяет форму прямоугольного параллелепипеда.
 */
class Box {
    /**
     * Создает экземпляр Box.
     * @constructor
     * @this  {Box}
     *
     * @param boxElement Ссылка на DOM-элемент, который иллюстрирует этот
объект.
     * @param func {Function} Функция-колбэк, которая должна оповестить об
окончании загрузки данных фигуры.
     */
    constructor(boxElement, func) {
        // Определим все неодходимые поля.
        /**
         * Размер по трем осям.
         * @type {number[]}
         */
        this.size = [10, 10, 10];
        /**
         * Ссылка на DOM-элемент, который иллюстрирует этот объект.
         */
        this.boxElement = boxElement;
        //
        /**
         * Массив вершин фигуры.
         * @type {number[]}
         */
        this.vertices = [];
        /**
         * Массив нормалей фигуры.
         * @type {number[]}
         */
        this.normals = [];
        /**
         * Массив индексов поверхностей фигуры.
         * @type {number[]}
         */
        this.indices = undefined;

        // Запустим инициализацию полей атрибутами.
        this.init();
        // Вызываемколл-бэк.
        func(this);
    }

    /**
     * Инициализирует трехмерный объект Box, используя атрибуты тега my-box.
     * Задает вершины и нормали, необходимые для отрисовки прямоугольного
```

```
параллелепипеда.
     *
     * Из атрибута size вида массив получаются размеры параллелепипеда по
соответствующим сторонам.
     * По умолчанию size="10 10 10".
     * Центр прямоугольного параллелепипеда совпадает с началом координат.
     */
    init() {
        // Получаем из атрибута размеры фигры или оставляем их значения по
умолчанию.
        if (this.boxElement.attributes["size"])
            this.size =
                this.boxElement.attributes["size"].value.split(" ").map(value
=> parseFloat(value));

        // Выполняем проверку полученных данных.
        Utils.checkArrayAttribute(this.size, "my-box", "size");

        // Задаем вершины нашей фигуры.
        let x = this.size[0] / 2;
        let y = this.size[1] / 2;
        let z = this.size[2] / 2;

        this.vertices = [
            // передняя
            -x, y, z,
            -x, -y, z,
            x, -y, z,
            -x, y, z,
            x, -y, z,
            x, y, z,

            // правая
            x, y, z,
            x, -y, z,
            x, -y, -z,
            x, y, z,
            x, -y, -z,
            x, y, -z,

            // задняя
            x, y, -z,
            x, -y, -z,
            -x, -y, -z,
            x, y, -z,
            -x, -y, -z,
            -x, y, -z,

            // левая
            -x, y, -z,
            -x, -y, -z,
            -x, -y, z,
            -x, y, -z,
            -x, -y, z,
            -x, y, z,

            // верхняя
            -x, y, -z,
```

```
        -x, y, z,
        x, y, z,
        -x, y, -z,
        x, y, z,
        x, y, -z,

        // нижняя
        -x, -y, -z,
        x, -y, z,
        -x, -y, z,
        -x, -y, -z,
        x, -y, -z,
        x, -y, z,
    ];

    // Задаем нормали для нашей фигуры.
    this.normals = [
        // передняя
        0, 0, 1,
        0, 0, 1,
        0, 0, 1,
        0, 0, 1,
        0, 0, 1,
        0, 0, 1,

        // правая
        1, 0, 0,
        1, 0, 0,
        1, 0, 0,
        1, 0, 0,
        1, 0, 0,
        1, 0, 0,

        // задняя
        0, 0, -1,
        0, 0, -1,
        0, 0, -1,
        0, 0, -1,
        0, 0, -1,
        0, 0, -1,

        // левая
        -1, 0, 0,
        -1, 0, 0,
        -1, 0, 0,
        -1, 0, 0,
        -1, 0, 0,
        -1, 0, 0,

        // верхняя
        0, 1, 0,
        0, 1, 0,
        0, 1, 0,
        0, 1, 0,
        0, 1, 0,
        0, 1, 0,

        // нижняя
```

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

Описание класса Camera

```
/**
 * Тег my-camera. Определяет камеру в нашей сцене.
 */
class Camera {
    /**
     * Создает экземпляр Camera.
     * @constructor
     * @this  {Camera}
     *
     * @param cameraElement Ссылка на DOM-элемент, который иллюстрирует этот
объект.
     * @param scene {Scene} Трехмерная сцена, в которой определен объект.
     */
    constructor(cameraElement, scene) {
        // Определим все неодходимые поля.
        /**
         * Ссылка на DOM-элемент, который иллюстрирует этот объект.
         */
        this.cameraElement = cameraElement;
        /**
         * Ссылка на трехмерную сцену, которой принадлежит камера.
         * @type {Scene}
         */
        this.scene = scene;

        /**
         * Отношение ширины окна отрисовки к высоте.
         * @type {number}
         */
        this.aspect = this.scene.gl.canvas.clientWidth /
this.scene.gl.canvas.clientHeight;
        /**
         * Угол отображения области видимости в радианах.
         * @type {number}
         */
        this.fieldOfViewDegrees = 60;
        /**
         * Модуль самого дальнего индекса, видимого камере, по Z.
         * @type {number}
         */
        this.zFar = 2000;
        /**
         * Модуль ближайшего индекса, видимого камере, по Z.
         * @type {number}
         */
```

```
        this.zNear = 1;
        /**
         * Углы поворота камеры в градусах.
         * @type {number[]}
         */
        this.cameraRotation = [0, 0, 0];
        /**
         * Позиция камеры в мире.
         * @type {number[]}
         */
        this.cameraPosition = [0, 0, 0];
        /**
         * Тип навигации по трехмерной сцене.
         * @type {string}
         */
        this.navigationType = "";

        /**
         * Матрица камеры.
         * @type {number[]}
         */
        this.cameraMatrix = Algebra.identity();
        /**
         * Матрица, формирующая перспективу.
         * @type {number[]}
         */
        this.projectionMatrix = undefined;

        // Запустим инициализацию полей атрибутами.
        this.init();
    }

    /**
     * Инициализирует данные для шедеров, используя атрибуты тега my-camera.
     *
     * Определяет парамтры фрустума:
     *      - атрибут "view-angle" - число - угол фрустума в градусах - по
умолчанию 60;
     *      - атрибут "z-far" - число - максимальное видимое значение
координат по оси z - по умолчанию 2000;
     *      - атрибут "z-near" - число - минимальное видимое значение
координат по оси z - по умолчанию 1.
     * И параметры камеры:
     *      - атрибут "camera-rotation" - массив - углы вращения по осям
x,y,z для камеры - по умолчанию "0 0 0";
     *      - атрибут "camera-position" - массив - начальная позиция камеры -
по умолчанию "0 0 0".
     * А так же "navigation-type", принимающий одно из двух значений: object
или camera.
     * По умолчанию навигация по сцене не производится.
     *
     */
    init() {
        // Получаем из атрибутов данные или оставляем их значения по
умолчанию.
        if (this.cameraElement.attributes["view-angle"])
            this.fieldOfViewDegrees =
parseFloat(this.cameraElement.attributes["view-angle"].value);
```

```
        if (this.cameraElement.attributes["z-far"])
            this.zFar = parseFloat(this.cameraElement.attributes["z-
far"].value);
        if (this.cameraElement.attributes["z-near"])
            this.zNear = parseFloat(this.cameraElement.attributes["z-
near"].value);
        if (this.cameraElement.attributes["camera-rotation"])
            this.cameraRotation =
                this.cameraElement.attributes["camera-
rotation"].value.split(" ").map(value =>
Utils.getRadians(parseFloat(value)));
        if (this.cameraElement.attributes["camera-position"])
            this.cameraPosition = this.cameraElement.attributes["camera-
position"].value.split(" ").map(value => parseFloat(value));
        if (this.cameraElement.attributes["navigation-type"])
            this.navigationType = this.cameraElement.attributes["navigation-
type"].value;


        // Выполняем проверку полученных данных.
        Utils.checkNumberAttribute(this.fieldOfViewDegrees, "my-camera",
"view-angle");
        Utils.checkNumberAttribute(this.zFar, "my-camera", "z-far");
        Utils.checkNumberAttribute(this.zNear, "my-camera", "z-near");
        Utils.checkArrayAttribute(this.cameraRotation, "my-camera", "camera-
rotation");
        Utils.checkArrayAttribute(this.cameraPosition, "my-camera", "camera-
position");

        // Определяем способ навигации по сцене.
        this.defineNavigationType();

        // Задаем позицию камеры.
        this.setCameraParams();
    }


    /**
     * Пересчитывает матрицу камеры в случае изменения параметров.
     */
    setCameraParams() {
        // Задаем перспективу.
        this.projectionMatrix =
Algebra.perspective(Utils.getRadians(this.fieldOfViewDegrees), this.aspect,
this.zNear, this.zFar);

        // "Перносим" камеру на нужную позицию
        this.cameraMatrix = Algebra.identity();
        this.cameraMatrix = Algebra.translate(this.cameraMatrix,
this.cameraPosition[0], this.cameraPosition[1], this.cameraPosition[2]);
        this.cameraMatrix = Algebra.zRotate(this.cameraMatrix,
this.cameraRotation[2]);
        this.cameraMatrix = Algebra.yRotate(this.cameraMatrix,
this.cameraRotation[1]);
        this.cameraMatrix = Algebra.xRotate(this.cameraMatrix,
this.cameraRotation[0]);
        this.cameraMatrix = Algebra.inverse(this.cameraMatrix);
    }
```

```
    /**
     * По атрибуту navigation-type тега my-camera задает тип навигации по
трехмерной сцене.
     */
    defineNavigationType() {
        function getShift(shift) {
            let rot = Algebra.identity();
            rot = Algebra.yRotate(rot, this.cameraRotation[1]);
            rot = Algebra.xRotate(rot, this.cameraRotation[0]);
            rot = Algebra.inverse(rot);
            return Algebra.multiply(shift, rot);
        }

        let shift;

        if (this.navigationType === "object") {
            // Добавление вращения и приближения объектов в сцене.
            document.addEventListener("keydown", (event) => {
                switch (event.keyCode) {
                    case 39:
                        // Поворот по оси Y впаво
                        for (let obj of scene.objects)
                            obj.transform.rotation[1] += Utils.getRadians(5);
                        scene.drawScene();
                        break;
                    case 37:
                        // Поворот по оси Y влево
                        for (let obj of scene.objects)
                            obj.transform.rotation[1] -= Utils.getRadians(5);
                        scene.drawScene();
                        break;
                    case 38:
                        // Поворот по оси X вверх
                        for (let obj of scene.objects)
                            obj.transform.rotation[0] -= Utils.getRadians(5);
                        scene.drawScene();
                        break;
                    case 40:
                        // Поворот по оси X вниз
                        for (let obj of scene.objects)
                            obj.transform.rotation[0] += Utils.getRadians(5);
                        scene.drawScene();
                        break;
                    case 34:
                        // Поворот по оси Z против часовой
                        for (let obj of scene.objects)
                            obj.transform.rotation[2] += Utils.getRadians(5);
                        scene.drawScene();
                        break;
                    case 33:
                        // Поворот по оси Z по часовой
                        for (let obj of scene.objects)
                            obj.transform.rotation[2] -= Utils.getRadians(5);
                        scene.drawScene();
                        break;
                    case 36:
                        // Home: приближение
```

```
                        shift = getShift.call(scene.activeCamera, [0, 0, -1,
0]);
                        scene.activeCamera.cameraPosition[0] += shift[0];
                        scene.activeCamera.cameraPosition[1] += shift[1];
                        scene.activeCamera.cameraPosition[2] += shift[2];
                        scene.activeCamera.setCameraParams();
                        scene.drawScene();
                        break;
                    case 35:
                        // End: отдаление
                        shift = getShift.call(scene.activeCamera, [0, 0, 1,
0]);
                        scene.activeCamera.cameraPosition[0] += shift[0];
                        scene.activeCamera.cameraPosition[1] += shift[1];
                        scene.activeCamera.cameraPosition[2] += shift[2];
                        scene.activeCamera.setCameraParams();
                        scene.drawScene();
                        break;
                }
            });
        } else if (this.navigationType === "camera") {
            // Добавление бродящего передвижения к сцене.
            document.addEventListener("keydown", (event) => {
                if (event.shiftKey) {
                    switch (event.keyCode) {
                        case 39:
                            scene.activeCamera.cameraRotation[1] -=
Utils.getRadians(1);
                            scene.activeCamera.setCameraParams();
                            scene.drawScene();
                            break;
                        case 37:
                            scene.activeCamera.cameraRotation[1] +=
Utils.getRadians(1);
                            scene.activeCamera.setCameraParams();
                            scene.drawScene();
                            break;
                        case 38:
                            scene.activeCamera.cameraRotation[0] +=
Utils.getRadians(1);
                            scene.activeCamera.setCameraParams();
                            scene.drawScene();
                            break;
                        case 40:
                            scene.activeCamera.cameraRotation[0] -=
Utils.getRadians(1);
                            scene.activeCamera.setCameraParams();
                            scene.drawScene();
                            break;
                        case 34:
                            scene.activeCamera.cameraRotation[2] -=
Utils.getRadians(1);
                            scene.activeCamera.setCameraParams();
                            scene.drawScene();
                            break;
                        case 33:
                            scene.activeCamera.cameraRotation[2] +=
Utils.getRadians(1);
```

```
                        scene.activeCamera.setCameraParams();
                        scene.drawScene();
                        break;
                    }
                } else {
                    switch (event.keyCode) {
                        case 39:
                            shift = getShift.call(scene.activeCamera, [1, 0,
0, 0]);
                            scene.activeCamera.cameraPosition[0] += shift[0];
                            scene.activeCamera.cameraPosition[1] += shift[1];
                            scene.activeCamera.cameraPosition[2] += shift[2];
                            scene.activeCamera.setCameraParams();
                            scene.drawScene();
                            break;
                        case 37:
                            shift = getShift.call(scene.activeCamera, [-1, 0,
0, 0]);
                            scene.activeCamera.cameraPosition[0] += shift[0];
                            scene.activeCamera.cameraPosition[1] += shift[1];
                            scene.activeCamera.cameraPosition[2] += shift[2];
                            scene.activeCamera.setCameraParams();
                            scene.drawScene();
                            break;
                        case 38:
                            shift = getShift.call(scene.activeCamera, [0, 1,
0, 0]);
                            scene.activeCamera.cameraPosition[0] += shift[0];
                            scene.activeCamera.cameraPosition[1] += shift[1];
                            scene.activeCamera.cameraPosition[2] += shift[2];
                            scene.activeCamera.setCameraParams();
                            scene.drawScene();
                            break;
                        case 40:
                            shift = getShift.call(scene.activeCamera, [0, -1,
0, 0]);
                            scene.activeCamera.cameraPosition[0] += shift[0];
                            scene.activeCamera.cameraPosition[1] += shift[1];
                            scene.activeCamera.cameraPosition[2] += shift[2];
                            scene.activeCamera.setCameraParams();
                            scene.drawScene();
                            break;
                        case 35:
                            shift = getShift.call(scene.activeCamera, [0, 0,
1, 0]);
                            scene.activeCamera.cameraPosition[0] += shift[0];
                            scene.activeCamera.cameraPosition[1] += shift[1];
                            scene.activeCamera.cameraPosition[2] += shift[2];
                            scene.activeCamera.setCameraParams();
                            scene.drawScene();
                            break;
                        case 36:
                            shift = getShift.call(scene.activeCamera, [0, 0,
-1, 0]);
                            scene.activeCamera.cameraPosition[0] += shift[0];
                            scene.activeCamera.cameraPosition[1] += shift[1];
                            scene.activeCamera.cameraPosition[2] += shift[2];
                            scene.activeCamera.setCameraParams();
```

```
                            scene.drawScene();
                        }
                    }
                });
            }
        }
    }
```

Описание класса Color

```
/**
 * Тег my-color. Определяет цвет однотонной фигуры.
 */
class Color {
    /**
     * Создает экземпляр Color.
     * Получает атрибут "color" - массив - rgb-цвет (значения в пределах
0..255) - по умолчанию белый
     * @constructor
     * @this  {Color}
     *
     * @param colorElement Ссылка на DOM-элемент, который иллюстрирует этот
объект.
     * @param vertexCount {number}
     */
    constructor(colorElement, vertexCount) {
        /**
         * Список цветов вершин фигуры.
         * @type {number[]}
         */
        this.colors = [];

        // Получаем из атрибута данные или задаем их по умолчанию.
        let currentColor = colorElement.attributes["color"] ?
            colorElement.attributes["color"].value.split(" ").map(value =>
parseInt(value)) : [255, 255, 255];

        // Выполняем проверку полученных данных.
        Utils.checkArrayAttribute(currentColor, "my-color", "color");

        // Задаем цвет для текущей фигуры.
        for (let i = 0; i < vertexCount; ++i) {
            this.colors.push(currentColor[0], currentColor[1],
currentColor[2]);
        }
    }
}
```

Описание класса DirectedLight

```
/**
 * Тег my-directed-light. Определяет направленный свет для трехмерной сцены.
 */
class DirectedLight {
    /**
     * Создает экземпляр DirectedLight.
     * @constructor
```

```
     * @this  {DirectedLight}
     *
     * @param directedLightElement Ссылка на DOM-элемент, который
иллюстрирует этот объект.
     * @param scene {Scene} Трехмерная сцена, в которой определен объект.
     */
    constructor(directedLightElement, scene) {
        // Определим все неодходимые поля.
        /**
         * Ссылка на DOM-элемент, который иллюстрирует этот объект.
         */
        this.directedLightElement = directedLightElement;
        /**
         * Трехмерная сцена, в которой определен объект.
         * @type {Scene}
         */
        this.scene = scene;

        /**
         * Цвет фонового освещения.
         * @type {number[]}
         */
        this.fonLightColor = [100, 100, 100].map(value => value / 255);
        /**
         * Цвет направленного освещения.
         * @type {number[]}
         */
        this.directedLightColor = [200, 200, 200].map(value => value / 255);
        /**
         * Направление света.
         * @type {number[]}
         */
        this.lightDirection = [0, 0, -1];

        // Запустим инициализацию полей атрибутами.
        this.init();
    }

    /**
     * Инициализирует объект, используя данные атрибутов.
     *
     * Если есть устанавливает параметры этого освещения:
     *      - атрибут "fon-light-color" - массив - rgb-цвет фонового
освещения (значения в пределах 0..255)
     *                                    - по умолчанию "100 100 100";
     *      - атрибут "directed-light-color" - массив - rgb-цвет
направленного освещения(значения в пределах 0..255)
     *                                    - по умолчанию "200 200 200";
     *      - атрибут "direction" - массив - направление направленного
освещения - по умолчанию "0, 0, -1".
     */
    init() {
        if (this.directedLightElement.attributes["fon-light-color"])
            this.fonLightColor =
                this.directedLightElement.attributes["fon-light-
color"].value.split(" ").map(value => parseFloat(value)).map(value => value /
255);
        if (this.directedLightElement.attributes["directed-light-color"])
```

```
            this.directedLightColor =
                this.directedLightElement.attributes["directed-light-
color"].value.split(" ").map(value => parseFloat(value)).map(value => value /
255);
        if (this.directedLightElement.attributes["direction"])
            this.lightDirection =

this.directedLightElement.attributes["direction"].value.split(" ").map(value
=> parseFloat(value));


        // Выполняем проверку полученных данных.
        Utils.checkArrayAttribute(this.fonLightColor, "my-directed-light",
"fon-light-color");
        Utils.checkArrayAttribute(this.directedLightColor, "my-directed-
light", "directed-light-color");
        Utils.checkArrayAttribute(this.lightDirection, "my-directed-light",
"direction");

        // Нармализуем вектор направления света.
        this.lightDirection = Algebra.normalize(this.lightDirection);

        // Получим вектор, обратный вектору направления света.
        this.lightDirection[0] *= -1;
        this.lightDirection[1] *= -1;
        this.lightDirection[2] *= -1;
    }
}
```

Описание класса IndexedFaceSet

```
/**
 * Тег my-indexed-face-set. Определяет произвольную форму на сцене.
 */
class IndexedFaceSet {
    /**
     * Создает экземпляр IndexedFaceSet.
     * @constructor
     * @this  {IndexedFaceSet}
     *
     * @param indexedFaceSetElement Ссылка на DOM-элемент, который
иллюстрирует этот объект.
     * @param func {Function} Функция-колбэк, которая должна оповестить об
окончании загрузки данных фигуры.
     */
    constructor(indexedFaceSetElement, func) {
        // Определим все неодходимые поля.
        /**
         * Путь к модели, которую будет отрисовывать данный объект.
         * @type {string}
         */
        this.src = "";
        /**
         * Ссылка на DOM-элемент, который иллюстрирует этот объект.
         */
        this.indexedFaceSetElement = indexedFaceSetElement;
```

```
        // Запустим инициализацию полей атрибутами.
        this.init();

        // Запустим асинхронную загрузку полей данного объекта.
        if (typeof(Worker) !== "undefined") {
            //Браузер пользователя имеет поддержку web worker
            let worker = new Worker("task.js");
            worker.onmessage = function (e) {
                func(e.data);
            };
            worker.postMessage(this.src);
        }
        else {
            //Браузер пользователя не поддерживает web worker
            {
                // Переменная для хранения текста .obj файла.
                let modelSource;
                // Переменная для хранения пути к .obj файлу.
                let src = this.src;

                // Вспомогательная функция, определяющая способ загрузки
файла.
                function getXmlHttp() {
                    let xmlhttp;
                    try {
                        xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");
                    } catch (e) {
                        try {
                            xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
                        } catch (E) {
                            xmlhttp = false;
                        }
                    }
                    if (!xmlhttp && typeof XMLHttpRequest !== 'undefined') {
                        xmlhttp = new XMLHttpRequest();
                    }
                    return xmlhttp;
                }

                // Запрос на синхронное считывание информации из файла.
                (function () {
                    let xmlhttp = getXmlHttp();
                    xmlhttp.open('GET', src, false);
                    xmlhttp.send(null);
                    if (xmlhttp.status === 200) {
                        modelSource = xmlhttp.responseText;
                    }
                })();

                // Парсинг файла формата .obj
                if (!modelSource)
                    throw new Error("Неверно указано имя .obj файла!");
                let info = new OBJLoader();
                info.parse(modelSource);
                let vertices = info.vertices;
                let normals = info.normals;
                let indices = info.indices;
```

```
                // Вызыв колбэка, оповещающего об окончании загрузки.
                func({
                    vertices,
                    normals,
                    indices
                });
            }
        }

    }

    /**
     * Инициализирует объект, используя данные атрибутов.
     *
     * Получает путь к .obj файлу - модели, которую необходимо отрендерить,
через атрибут model.
     */
    init() {
        // Получаем путь к модели, которую необходимо отрендерить, через
атрибут model.
        if (this.indexedFaceSetElement.attributes["model"])
            this.src = this.indexedFaceSetElement.attributes["model"].value;
        else
            throw new Error("Отсутствует путь к 3D модели, ожидаемый в теге
my-indexed-face-set!");
    }
}
```

Описание класса OBJLoader

```
/**
 * Парсер файлов формата .obj.
 */
class OBJLoader {
    /**
     * Создает экземпляр OBJLoader.
     * @constructor
     * @this  {OBJLoader}
     */
    constructor() {
        // Данные необходимые к загрузке.
        /**
         * Массив вершин загружаемой модели.
         * @type {number[]}
         */
        this.vertices = [];
        /**
         * Массив нормалей загружаемой модели.
         * @type {number[]}
         */
        this.normals = [];
        /**
         * Массив индексов поверхностей загружаемой модели.
         * @type {number[]}
         */
        this.indices = [];
    }
```

```
    /**
     *  По заданным индексам поверхностей и вершинам формирует массив
нормалей.
     */
    calculateNormals() {
        // Если необходимые исходные данные не определены - не делать ничего.
        if (!(this.vertices.length && this.indices.length))
            return;

        // Число нормалей эквивалентно числу вершин в фигуре.
        this.normals = new Array(this.vertices.length);
        // Идем по массиву поверхностей и для вершин каждой поверхности
считаем нормаль.
        for (let i = 0; i < this.indices.length; i += 3) {
            // Определим два вектора, принадлежащих поверхности.
            let v1 = [
                this.vertices[this.indices[i + 2] * 3] -
this.vertices[this.indices[i + 1] * 3],
                this.vertices[this.indices[i + 2] * 3 + 1] -
this.vertices[this.indices[i + 1] * 3 + 1],
                this.vertices[this.indices[i + 2] * 3 + 2] -
this.vertices[this.indices[i + 1] * 3 + 2]
            ];
            let v2 = [
                this.vertices[this.indices[i] * 3] -
this.vertices[this.indices[i + 1] * 3],
                this.vertices[this.indices[i] * 3 + 1] -
this.vertices[this.indices[i + 1] * 3 + 1],
                this.vertices[this.indices[i] * 3 + 2] -
this.vertices[this.indices[i + 1] * 3 + 2]
            ];
            // Найдем их векторное произведение.
            let result = [
                v1[1] * v2[2] - v1[2] * v2[1],
                v1[2] * v2[0] - v1[0] * v2[2],
                v1[0] * v2[1] - v1[1] * v2[0]
            ];
            // Нормализуем полученный вектор.
            result = Algebra.normalize(result);

            // Заполним нужные элементы массива нормалей результатом.
            this.normals[this.indices[i] * 3] = result[0];
            this.normals[this.indices[i] * 3 + 1] = result[1];
            this.normals[this.indices[i] * 3 + 2] = result[2];

            this.normals[this.indices[i + 1] * 3] = result[0];
            this.normals[this.indices[i + 1] * 3 + 1] = result[1];
            this.normals[this.indices[i + 1] * 3 + 2] = result[2];

            this.normals[this.indices[i + 2] * 3] = result[0];
            this.normals[this.indices[i + 2] * 3 + 1] = result[1];
            this.normals[this.indices[i + 2] * 3 + 2] = result[2];
        }
    }

    /**
     * Преобразует текстовое представление .obj файла в массив вершин,
```

```
нормалей и индексов.
     * @param objectData {string} Текстовое представление .obj файла.
     */
    parse(objectData) {
        let self = this;
        /*
        .obj формат содержит три типа важных нам строк:
          * 'v': строка определяющая вершину (имеет вид "v 1.23 4.5 6" или "v
1.23 4.5 6 7");
          * 'vn': строка определяющая нормаль (имеет вид "vn 0.001 -0.001
0.99");
          * 'f': строка определяющая поверхность.
                Может принимать один из следующих видов:
            f 16/92/11 14/101/22 1/69/1 - даны и вершина, и текстура, и
нормаль
            f 16//11 14//22 1//1 - даны вершина и нормаль
            f 16/92 14/101 1/69 - даны вершина и текстура
            f 16 14 1 - дана только вершина

            На выходе:
            Массив индексов поверхностей будет содержать данные о том, какую
точку и нормаль использовать.
            Массив индексов - это плоский массив, где каждые три элемента
образуют треугольник.
            К примеру: треугольник, которому в массиве индексов соответствуют
числа 3, 5, 15 - это
            треугольник которому для отрисовки нобходимо взять вершины под
индексами 3, 5, 15 и нормали
            под теми же номерами.
       */

        // Временный пул всех данных.
        const verticesList = [];
        const normalsList = [];
        const unpacked = {};

        // Итоговые массивы данных.
        unpacked.verts = [];
        unpacked.norms = [];
        unpacked.indices = [];
        // Ассоциативный массив для переиспользования пар вершина нормаль.
        unpacked.heshes = {};

        // Регулярные выражения для определения типа строки.
        const VERTEX_RE = /^v\s/;
        const NORMAL_RE = /^vn\s/;
        const FACE_RE = /^f\s/;
        const WHITESPACE_RE = /\s+/;
        const all_meaning_re = [VERTEX_RE, NORMAL_RE, FACE_RE];

        // Массив строк нашей модели.
        const lines = objectData.split("\n");

        for (let i = 0; i < lines.length; i++) {
            const line = lines[i].trim();

            // Отбрасываем все неподходящие строки.
            if (!line || line.startsWith("#") || !all_meaning_re.some((re) =>
```

```
re.test(line))) {
                continue;
            }

            // Делим строку по пробелу и получаем массив необходимых нам
значений
            const elements = line.split(WHITESPACE_RE);
            elements.shift();

            if (VERTEX_RE.test(line)) {
                // Обрабатываем вершину.
                elements.length = 3; // т.к. иногда вершина может содержать
дополнительную, ненужную нам информацию.
                verticesList.push(...(elements).map(value =>
parseFloat(value)));
            } else if (NORMAL_RE.test(line)) {
                // Обрабатываем нормаль.
                normalsList.push(...(elements).map(value =>
parseFloat(value)));
            } else if (FACE_RE.test(line)) {
                // Обрабатываем поверхность.
                let current;

                let newElements = [];

                // Если поверхность не треугольник,
                // то предтавляем ее как массив треугольников, добавляя
недостающие вершины.
                for (let k = 2; k < elements.length; ++k) {
                    newElements.push(elements[0], elements[k - 1],
elements[k]);
                }

                for (let j = 0; j < newElements.length; ++j) {

                    let data = newElements[j].split('/');

                    if (data.length > 2) {
                        let hash = data[0] + "/" + data[2];
                        if (unpacked.heshes[hash]) {
                            current = unpacked.heshes[hash];
                        } else {
                            current = parseInt(unpacked.verts.length / 3);

                            unpacked.verts.push(
                                verticesList[(parseInt(data[0]) - 1) * 3],
                                verticesList[(parseInt(data[0]) - 1) * 3 +
1],
                                verticesList[(parseInt(data[0]) - 1) * 3 + 2]
                            );

                            unpacked.norms.push(
                                normalsList[(parseInt(data[2]) - 1) * 3],
                                normalsList[(parseInt(data[2]) - 1) * 3 + 1],
                                normalsList[(parseInt(data[2]) - 1) * 3 + 2]
                            );

                            unpacked.heshes[hash] = current;
```

```
                    }
                } else {
                    current = parseInt(unpacked.verts.length / 3);

                    unpacked.verts.push(
                        verticesList[(parseInt(data[0]) - 1) * 3],
                        verticesList[(parseInt(data[0]) - 1) * 3 + 1],
                        verticesList[(parseInt(data[0]) - 1) * 3 + 2]
                    );
                }
                unpacked.indices.push(current);
            }
        }
    }

    self.vertices = unpacked.verts;
    self.normals = unpacked.norms;
    self.indices = unpacked.indices;

    // Если в нашей модели были поверхности, не содержащие информации о
нормалях, -
    // пересчитаем все нормали.
    if (self.vertices.length !== self.normals.length)
        this.calculateNormals(self.vertices, self.indices);
  }
}
```

Описание класса Scene

```
/**
 * Определяет трехмерную сцену.
 */
class Scene {
    /**
     * Создает экземпляр Scene.
     * @constructor
     * @this  {Scene}
     *
     * @param sceneElement Ссылка на DOM-элемент, который иллюстрирует этот
объект.
     */
    constructor(sceneElement) {
        /**
         * Ссылка на DOM-элемент, который иллюстрирует этот объект.
         */
        this.sceneElement = sceneElement;
        /**
         * Графический контекст.
         */
        this.gl = undefined;

        /**
         * Фоновый цвет сцены.
         * @type {number[]}
         */
        this.color = [1, 1, 1, 0];
```

```
        /**
         * Все объекты сцены.
         * @type {Array}
         */
        this.objects = [];
        /**
         * Все камеры сцены.
         * @type {Camera[]}
         */
        this.cameras = [];
        /**
         * Активная камера.
         * @type {Camera}
         */
        this.activeCamera = undefined;
        /**
         * Напрвленный свет.
         * @type {DirectedLight}
         */
        this.light = undefined;

        /**
         * Текст фрагментного шейдера на GLSL.
         * @type {string}
         */
        this.fragmentShaderSource =
            "  precision mediump float;\
            \
              varying vec4 v_color;\
             varying vec3 v_light;\
            \
            uniform vec3 u_reverseLightDirection;\
            \
              void main() {\
               gl_FragColor = vec4(v_color.rgb * v_light, v_color.a);\
              }";
        /**
         * Текст вершинного шейдера на GLSL.
         * @type {string}
         */
        this.vertexShaderSource = "attribute vec4 a_position;\
    attribute vec4 a_color;\
    attribute vec3 a_normal;\
\
    uniform mat4 u_matrix;\
    uniform mat3 u_normal_matrix;\
    uniform vec3 u_fon_light_color;\
    uniform vec3 u_light_direction;\
    uniform vec3 u_directed_light_color;\
    uniform int u_use_light;\
\
    varying vec4 v_color;\
    varying vec3 v_light;\
\
    void main() {\
      gl_Position = u_matrix * a_position;\
\
      v_color = a_color;\
```

```
    \
         if (u_use_light == 0) {\
           v_light = vec3(1,1,1);\
         } else {\
           vec3 transformedNormal = u_normal_matrix * a_normal;\
    \
           float directed_light_weight = max(dot(transformedNormal,
u_light_direction), 0.0);\
    \
           v_light = u_fon_light_color + u_directed_light_color *
directed_light_weight;\
    }\
        }";

        // Регистрируем все введеные библиотекой html-элементы.
        this.registerAll();
        // Запускаем инициализацию сцены и ее дочерних элементов.
        this.init();
        // Отрисовываем сцену.
        this.drawScene();
    }

    /**
     * Регистрация всех кастомных HTML-элементов.
     */
    registerAll() {
        if (!document.registerElement)
            return;

        // Регистрируем непосредственно сцену.
        let mySceneProto = Object.create(HTMLCanvasElement.prototype);
        document.registerElement("my-scene", {
            prototype: mySceneProto,
            extends: 'canvas'
        });

        // Все кастомные теги.
        let tags = [
            "my-camera",
            "my-transform",
            "my-shape",
            "my-appearance",
            "my-indexed-face-set",
            "my-box",
            "my-color",
            "my-cone",
            "my-cylinder",
            "my-sphere",
            "my-directed-light"
        ];

        // Регистрируем остальные теги.
        for (let tag of tags) {
            let myProto = Object.create(HTMLElement.prototype);
            document.registerElement(tag, {
                prototype: myProto
            });
        }
```

```
    }


    /**
     * Инициализирует данные, используя атрибуты тега my-scene.
     * Проверяет наличие обязательных тегов.
     * Запускает инициализацию дочерних элементов.
     */
    init() {
        this.sceneElement.width = this.sceneElement.width || "0";
        this.sceneElement.height = this.sceneElement.height || "0";

        // Задаем для нее графический контекст.
        this.gl = this.sceneElement.getContext("webgl") ||
this.sceneElement.getContext("experimental-webgl");
        if (!this.gl)
            throw new Error("В данном браузере недоступен WebGL");

        // Компилируем шейдеры.
        this.fragmentShader = this.createShader(this.gl.FRAGMENT_SHADER,
this.fragmentShaderSource);
        this.vertexShader = this.createShader(this.gl.VERTEX_SHADER,
this.vertexShaderSource);
        // Создаем шейдерную программу.
        this.program = this.createProgram(this.vertexShader,
this.fragmentShader);

        // Проверяем на наличие камеры.
        if (!this.sceneElement.getElementsByTagName("my-camera").length)
            throw new Error("Отсутствует дочерний элемент - my-camera");
        for (let cam of this.sceneElement.getElementsByTagName("my-camera"))
{
            this.cameras.push(new Camera(cam, this));
            if (this.cameras[this.cameras.length - 1].isActive)
                this.activeCamera = this.cameras[this.cameras.length - 1];
        }
        if (!this.activeCamera)
            this.activeCamera = this.cameras[0];

        // Подготовим пространство для отрисовки
        this.gl.enable(this.gl.CULL_FACE); // разрешим отрисовку только
лицевых треугольников
        this.gl.enable(this.gl.DEPTH_TEST); // включим проверку z-индекса

        // Достаем из атрибутов указанный цвет фона
        this.color = this.sceneElement.attributes["fon-color"] ?
            this.sceneElement.attributes["fon-color"].value.split("
").map(value => parseFloat(value)).map(value => value / 255)
            : this.color;

        // Запускаем инициализацию дерева элемнтов.
        for (let trans of this.sceneElement.getElementsByTagName("my-
transform")) {
            new Transform(trans, this);
        }

        // Проверяем наличие света в сцене.
        this.light = this.sceneElement.getElementsByTagName("my-directed-
```

```
light").length ?
            new DirectedLight(this.sceneElement.getElementsByTagName("my-
directed-light")[0], this) : null;
    }

    /**
     * Подгоняет размер canvas под экран
     */
    resize() {
        // получаем размер HTML-элемента canvas
        let displayWidth = this.sceneElement.clientWidth;
        let displayHeight = this.sceneElement.clientHeight;

        // проверяем, отличается ли размер canvas
        if (this.sceneElement.width !== displayWidth ||
            this.sceneElement.height !== displayHeight) {

            // подгоняем размер буфера отрисовки под размер HTML-элемента
            this.sceneElement.width = displayWidth;
            this.sceneElement.height = displayHeight;
        }
    }

    /**
     * Добавление трехмерного объекта на сцену.
     * @param transform {Transform} Положение в пространстве добавляемого
объекта.
     * @param shape {Shape} Форма добавляемого объекта.
     * @param appearance {Appearance} Внешний вид добавляемого объекта.
     */
    addObject({transform, shape, appearance}) {
        // Добавляем ко всем объектам сцены.
        this.objects.push({
            transform,
            shape,
            appearance
        });
        // Перерисовываем сцену.
        this.drawScene();
    }

    /**
     * Рендерит сцену.
     */
    drawScene() {
        // Подгоняем размер окна прорисовки под канвас.
        this.resize(this.gl.canvas);
        this.gl.viewport(0, 0, this.gl.canvas.width, this.gl.canvas.height);

        // Очищаем canvas.
        this.gl.clearColor(...this.color);
        // Очищаем буферы цветов и глубины.
        this.gl.clear(this.gl.COLOR_BUFFER_BIT | this.gl.DEPTH_BUFFER_BIT);

        // Указываем, какую программу использовать.
        this.gl.useProgram(this.program);

        // Запускаем прорисовку каждой фигуры сцены по порядку.
```

```
        for (let obj of this.objects) {
            // Передаем данные а атрибуты и буферы.
            // Передаем позиции вершин фигуры.
            let size = 3;                 // 3 компоненты на итерацию
            let type = this.gl.FLOAT; // наши данные - 32-битные числа с
плавающей точкой
            let normalize = false;      // не нормализовать данные
            let stride = 0;               // 0 = перемещаться на size *
sizeof(type) каждую итерацию для получения следующего положения
            let buf_offset = 0;         // начинать с начала буфера
            this.fillAttribute(this.program, "a_position", new
Float32Array(obj.shape.vertices),
                size, type, normalize, stride, buf_offset);

            // Передаем координаты нормалей фигуры.
            size = 3;                   // 3 компоненты на итерацию
            type = this.gl.FLOAT;    // наши данные - 32-битные числа с
плавающей точкой
            normalize = false;          // не нормализовать данные
            stride = 0;                 // 0 = перемещаться на size *
sizeof(type) каждую итерацию для получения следующего положения
            buf_offset = 0;             // начинать с начала буфера
            this.fillAttribute(this.program, "a_normal", new
Float32Array(obj.shape.normals),
                size, type, normalize, stride, buf_offset);

            // Передаем цвета вершин фигуры.
            size = 3;                           // 3 компоненты на итерацию
            type = this.gl.UNSIGNED_BYTE;  // данные - 8-битные беззнаковые
целые
            normalize = true;                   // нормализовать данные
(конвертировать из 0-255 в 0-1)
            stride = 0;                         // 0 = перемещаться на size *
sizeof(type) каждую итерацию для получения следующего положения
            buf_offset = 0;                     // начинать с начала буфера
            this.fillAttribute(this.program, "a_color", new
Uint8Array(obj.appearance.colors),
                size, type, normalize, stride, buf_offset);

            // Передаем данные в Uniform-переменные.
            // Передача матрицы смещения.
            let matrixLocation = this.gl.getUniformLocation(this.program,
"u_matrix");
            this.gl.uniformMatrix4fv(matrixLocation, false,
                obj.transform.getMatrix(this.activeCamera.cameraMatrix,
this.activeCamera.projectionMatrix));

            // Передача матрицы нормалей.
            let nMatrixLocation = this.gl.getUniformLocation(this.program,
"u_normal_matrix");
            this.gl.uniformMatrix3fv(nMatrixLocation, false,
obj.transform.getNormalMatrix());

            // Передача флага использования света.
            let useLightLocation =
                this.gl.getUniformLocation(this.program, "u_use_light");
            this.gl.uniform1i(useLightLocation, Number(!!this.light));
            if (this.light) {
```

```
                // Передача направления освещения.
                let lDirectionLocation =
                    this.gl.getUniformLocation(this.program,
"u_light_direction");
                this.gl.uniform3fv(lDirectionLocation,
Algebra.normalize(this.light.lightDirection));

                // Передача цвета фонового освещения.
                let fColorLocation =
                    this.gl.getUniformLocation(this.program,
"u_fon_light_color");
                this.gl.uniform3fv(fColorLocation, this.light.fonLightColor);

                // Передача цвета направленного освещения.
                let dColorLocation =
                    this.gl.getUniformLocation(this.program,
"u_directed_light_color");
                this.gl.uniform3fv(dColorLocation,
this.light.directedLightColor);
            }
            if (obj.shape.indices) {
                // Передаем индексы поверхностей фигуры.
                this.fillIndexAttribute(obj.shape.indices);

                // Отрисовка сцены.
                let primitiveType = this.gl.TRIANGLES; // рисовать
треугольники.
                let offset = 0; // начинать с начала буферов
                let type = this.gl.UNSIGNED_SHORT;
                let count = obj.shape.indices.length; // количество
треугольников передаваемых для отрисовки.
                this.gl.drawElements(primitiveType, count, type, offset);

            } else {
                // Отрисовка сцены.
                let primitiveType = this.gl.TRIANGLES; // рисовать
треугольники.
                let offset = 0; // начинать с начала буферов
                let count = obj.shape.vertices.length / 3; // количество
треугольников передаваемых для отрисовки.
                this.gl.drawArrays(primitiveType, offset, count);
            }

        }
    }

    /**
     * Создание и компиляция шейдера.
     *
     * @param type Тип шейдера.
     * @param source {String} Код шейдера.
     * @returns {WebGLShader} Шейдер.
     */
    createShader(type, source) {
        // Создание шейдера
        let shader = this.gl.createShader(type);

        // Устанавливаем шейдеру его программный код
```

```
        this.gl.shaderSource(shader, source);

        // Компилируем шейдер
        this.gl.compileShader(shader);

        // В случае успешной компиляции возвращаем шейдер
        let success = this.gl.getShaderParameter(shader,
this.gl.COMPILE_STATUS);
        if (success) {
            return shader;
        }

        // В случае ошибки - сообщаем о ней
        console.log(this.gl.getShaderInfoLog(shader));
        this.gl.deleteShader(shader);
    }

    /**
     * Собираем программу для видеокарты из двух шейдеров.
     *
     * @param vertexShader {WebGLShader} Вершинный шейдер.
     * @param fragmentShader {WebGLShader} Фрагментный шейдер.
     * @returns {WebGLProgram} Программа для видеокарты.
     */
    createProgram(vertexShader, fragmentShader) {
        // Создаем программую
        let program = this.gl.createProgram();

        // Закрепляем за ней шейдеры.
        this.gl.attachShader(program, vertexShader);
        this.gl.attachShader(program, fragmentShader);

        // Указываем, что именно это программу надо выполнять в текущем
графическом контексте.
        this.gl.linkProgram(program);

        // В случае успеха линковки - вернуть программу.
        let success = this.gl.getProgramParameter(program,
this.gl.LINK_STATUS);
        if (success) {
            return program;
        }

        // В случае ошибки - вывести информацию о ней.
        console.log(this.gl.getProgramInfoLog(program));
        this.gl.deleteProgram(program);
    }

    /**
     * Заполняет атрибут и буфер даннми.
     *
     * @param program {WebGLProgram} Текущая программа для видеокарты.
     * @param name {String} Наименование атрибута в шейдере.
     * @param data Массив данных для передачи, приведенный к нужному типу.
     * @param size {Number} Количество компонент массива на итерацию.
     * @param type Тип массива данных.
     * @param normalize {Boolean} Нуждаются ли данные в нормализации.
     * @param stride {Number} Дополнительное перемещение по данным
```

```
относительно итерации.
     * @param buf_offset {Number} Элемент массива, с которого начнется первая
итерация.
     */
    fillAttribute(program, name, data, size, type, normalize, stride,
buf_offset) {
        // Инициализируем атрибут и буфер.
        let attributeLocation = this.gl.getAttribLocation(program, name);
        let buffer = this.gl.createBuffer();

        // Привязываем атрибут и буфер.
        this.gl.bindBuffer(this.gl.ARRAY_BUFFER, buffer);
        this.gl.enableVertexAttribArray(attributeLocation);

        // Передаем данные в атрибут и в буфер.
        this.gl.bufferData(this.gl.ARRAY_BUFFER, data, this.gl.STATIC_DRAW);
        this.gl.vertexAttribPointer(
            attributeLocation, size, type, normalize, stride, buf_offset);
    }

    /**
     * Заполняет атрибут индексов.
     * @param indices {number[]} Данные для передачи
     */
    fillIndexAttribute(indices) {
        // Инициализируем буфер.
        let indexBuffer = this.gl.createBuffer();
        // Привязываем буфер.
        this.gl.bindBuffer(this.gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
        // Передаем данные в буфер.
        this.gl.bufferData(this.gl.ELEMENT_ARRAY_BUFFER, new
Uint16Array(indices), this.gl.STATIC_DRAW);
    }
}
```

Описание класса Shape

```
/**
 * Тег my-shape. Определяет геометрию.
 */
class Shape {
    /**
     * Запускает асинхронную инициализацию дочерних элементов.
     *
     * @param shapeElement Ссылка на DOM-элемент, который иллюстрирует этот
объект.
     * @param transform {Transform} Данные о положении фигуры в пространстве
сцены.
     * @param scene {Scene} Трехмерная сцена, в которой определен объект.
     */
    static init(shapeElement, transform, scene) {
        // Инициализируем форму фигуры и ее внешний вид.
        let indexedFaceSetElement = shapeElement.getElementsByTagName("my-
indexed-face-set")[0];
        let boxElement = shapeElement.getElementsByTagName("my-box")[0];
        let appearanceElement = shapeElement.getElementsByTagName("my-
appearance")[0];
```

```
        // Проверяем наличие обязательного элемента.
        if (!appearanceElement)
            throw new Error("Отсутствует обязательный тег my-appearance!");

        // Формируем колбек для асинхронной загрузки.
        function func(shape) {
            let promise = new Promise(function (resolve, reject) {
                resolve({
                    transform,
                    shape,
                    appearance: Appearance.init(appearanceElement,
shape.vertices.length / 3)
                })
            });

            promise.then(value => {
                scene.addObject(value)
            })
        }

        // Запускаем асинхронную загрузку найденного дочернего элемента.
        let promise;
        if (indexedFaceSetElement)
            promise = new Promise(function () {
                new IndexedFaceSet(indexedFaceSetElement, func);
            });
        else if (boxElement)
            promise = new Promise(function () {
                new Box(boxElement, func);
            });
        else
            throw new Error("Отсутствует тег, задающий форму");

        promise.then();
    }
}
```

Описание класса Transform

```
/**
 * Тег my-transform. Определяет положение геометрии в пространстве сцены.
 */
class Transform {
    /**
     * Создает экземпляр Transform.
     * @constructor
     * @this  {Transform}
     *
     * @param transformElement Ссылка на DOM-элемент, который иллюстрирует
этот объект.
     * @param scene {Scene} Трехмерная сцена, в которой определен объект.
     * @param info Данные о дополнительном смещении, если есть родительский
Transform.
     */
    constructor(transformElement, scene, info = {translation: [0, 0, 0],
rotation: [0, 0, 0], scale: [1, 1, 1]}) {
```

```
        /**
         * Ссылка на DOM-элемент, который иллюстрирует этот объект.
         */
        this.transformElement = transformElement;
        /**
         * Трехмерная сцена, в которой определен объект.
         * @type {Scene}
         */
        this.scene = scene;
        /**
         * Перенос оюъекта относительно начала координат сцены.
         * @type {number[]}
         */
        this.translation = info.translation;
        /**
         * Поворот объекта относительно его осей X, Y, Z.
         * @type {number[]}
         */
        this.rotation = info.rotation;
        /**
         * Масштабирование объекта относительно его осей X, Y, Z.
         * @type {number[]}
         */
        this.scale = info.scale;

        // Запустим инициализацию полей атрибутами.
        this.init();
    }

    /**
     * Инициализирует данные, используя атрибуты тега my-transform.
     * Атрибуты:
     *           translation - массив - определяет смещение относительно
центра сцены - по умолчанию "0 0 0"
     *           rotation - массив - определяет поворот относительно осей
сцены - по умолчанию "0 0 0"
     *           scale - массив - определяет масштаб объекта - по умолчанию "1
1 1"
     */
    init() {
        // Получаем из атрибутов данные или задаем их по умолчанию.
        let translation = this.transformElement.attributes["translation"] ?
            this.transformElement.attributes["translation"].value.split("
").map(value => parseFloat(value)) : [0, 0, 0];
        let anglesInDegrees = this.transformElement.attributes["rotation"] ?
            this.transformElement.attributes["rotation"].value.split("
").map(value => parseFloat(value)) : [0, 0, 0];
        let scale = this.transformElement.attributes["scale"] ?
            this.transformElement.attributes["scale"].value.split("
").map(value => parseFloat(value)) : [1, 1, 1];

        // Выполняем проверку полученных данных.
        Utils.checkArrayAttribute(translation, "my-transform",
"translation");
        Utils.checkArrayAttribute(anglesInDegrees, "my-transform",
"rotation");
        Utils.checkArrayAttribute(scale, "my-transform", "scale");
```

```
        // Дополняем наши внутренние свойства.
        this.translation[0] += translation[0];
        this.translation[1] += translation[1];
        this.translation[2] += translation[2];

        this.rotation[0] += Utils.getRadians(anglesInDegrees[0]);
        this.rotation[1] += Utils.getRadians(anglesInDegrees[1]);
        this.rotation[2] += Utils.getRadians(anglesInDegrees[2]);

        this.scale[0] *= scale[0];
        this.scale[1] *= scale[1];
        this.scale[2] *= scale[2];

        // Создаем фигур с текущими параметрами.
        for (let trans of this.transformElement.getElementsByTagName("my-
transform")) {
            new Transform(trans, this.scene);
        }

        for (let shape of this.transformElement.getElementsByTagName("my-
shape")) {
            Shape.init(shape, this, this.scene);
        }
    }

    /**
     * Считает матрицу вида для трехмерного объекта.
     * @param cameraMatrix {number[]} Матрица камеры.
     * @param projectionMatrix {number[]} Матрица проекции.
     * @returns {number[]} Матрица вида.
     */
    getMatrix(cameraMatrix, projectionMatrix) {
        let matrix = cameraMatrix;
        matrix = Algebra.translate(matrix, this.translation[0],
this.translation[1], this.translation[2]);
        matrix = Algebra.xRotate(matrix, this.rotation[0]);
        matrix = Algebra.yRotate(matrix, this.rotation[1]);
        matrix = Algebra.zRotate(matrix, this.rotation[2]);
        matrix = Algebra.scale(matrix, this.scale[0], this.scale[1],
this.scale[2]);
        matrix = Algebra.multiply(projectionMatrix, matrix);

        return matrix;
    }

    /**
     * Считает матрицу нормалей для трехмерного объекта.
     * @returns {number[]} Матрица нормалей.
     */
    getNormalMatrix() {
        let matrix = Algebra.identity();
        matrix = Algebra.translate(matrix, this.translation[0],
this.translation[1], this.translation[2]);
        matrix = Algebra.xRotate(matrix, this.rotation[0]);
        matrix = Algebra.yRotate(matrix, this.rotation[1]);
        matrix = Algebra.zRotate(matrix, this.rotation[2]);
        matrix = Algebra.normalFromMat4(matrix);
```

```
        return matrix;
    }
}
```

Описание класса Utils

```
/**
 * Класс вспомогательных функций.
 */
class Utils {
    /**
     * Перевод из градусов в радианы.
     *
     * @param angle {number} Угол в градусах.
     * @returns {number} Угол в радианах.
     */
    static getRadians(angle) {
        return angle * Math.PI / 180;
    }

    /**
     * Проверка значения на принадлежность к множеству массивов из трех
вещественных чисел.
     * Бросает исключение в случае не принадлежности.
     *
     * @param value Проверяемое значение.
     * @param tagName {string} Имя тега, откуда значение пришло.
     * @param attributeName {string} Имя атрибута, откуда значение пришло.
     */
    static checkArrayAttribute(value, tagName, attributeName) {
        if (value.length !== 3 || !Utils.checkNumber(value[0]) ||
!Utils.checkNumber(value[1]) || !Utils.checkNumber(value[2]))
            throw new Error("Ошибка при задании атрибута " + attributeName +
" тега " + tagName +
                ". Данный атрибут должен принимать значения \"x y z\", где x,
y, z - это вещественные числа.");
    }

    /**
     * Проверка значения на принадлежность к вещественным числам.
     * Бросает исключение в случае не принадлежности.
     *
     * @param value Проверяемое значение.
     * @param tagName {string} Имя тега, откуда значение пришло.
     * @param attributeName {string} Имя атрибута, откуда значение пришло.
     */
    static checkNumberAttribute(value, tagName, attributeName) {
        if (!Utils.checkNumber(value))
            throw new Error("Ошибка при задании атрибута " + attributeName +
" тега " + tagName +
                ". Данный атрибут должен принимать значения \"x\", где x -
это вещественное число.");
    }

    /**
     * Проверка значения на принадлежность к вещественным числам.
     * @param value Проверяемое значение.
```

```
     * @returns {boolean} Флаг принадлежности к вещественным числам.
     */
    static checkNumber(value) {
        return !(value !== value || typeof value === "undefined" || (typeof
value === "object" && !value));
    }
}
```

Текст вспомогательного файла task.js

```
// Функция, которую Worker запустит, получив сообщение.
onmessage = function (e) {
    // Подключим недостающие модули
    importScripts("../src/easy_webgl.js");

    // Переменная для хранения текста .obj файла.
    let modelSource;

    // Вспомогательная функция, определяющая способ загрузки файла.
    function getXmlHttp() {
        let xmlhttp;
        try {
            xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");
        } catch (e) {
            try {
                xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
            } catch (E) {
                xmlhttp = false;
            }
        }
        if (!xmlhttp && typeof XMLHttpRequest !== 'undefined') {
            xmlhttp = new XMLHttpRequest();
        }
        return xmlhttp;
    }

    // Запрос на синхронное считывание информации из файла.
    (function () {
        let xmlhttp = getXmlHttp();
        xmlhttp.open('GET', e.data, false);
        xmlhttp.send(null);
        if (xmlhttp.status === 200) {
            modelSource = xmlhttp.responseText;
        }
    })();

    // Парсинг модели.
    if (!modelSource)
        throw new Error('Неверно указано имя .obj файла!');
    let info = new OBJLoader();
    info.parse(modelSource);
    let vertices = info.vertices;
    let normals = info.normals;
    let indices = info.indices;

    // Отправка загруженных данных обратно в программу.
    postMessage({
```

```
            vertices,
            normals,
            indices
        });
    };
```

ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ

Лист регистрации изменений									
Номера листов (страниц)					Всего листов (страниц в докум.)	№ документа	Входящий № сопроводительного докум. и дата	Подп.	Дата
Изм.	Измененных	Замененных	Новых	Аннулированных					

Изм.	Лист	№ докум.	Подп.	Дата
RU.17701729.03.05-01 12				
Инв. № подл.	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата