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

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Компьютерная графика»
Тема: 3D трансформации

Студент гр. 9381 _____ Колованов Р.А.

Преподаватель _____ Герасимова Т.В.

Санкт-Петербург
2023

Цель работы.

Целью работы является освоение использование стандартных матричных преобразований над объектами, таких как:

- Преобразование модели (ModelMatrix);
- Преобразование вида (ViewMatrix);
- Преобразование проекции (ProjectionMatrix).

Задание.

Реализовать представление 3D сцены и объектов на ней с использованием стандартных матричных преобразований.

Выполнение работы.

1. Используемые технологии

Для реализации программы использовались язык программирования JavaScript, библиотека jQuery, API WebGL для 3D-графики, набор стилей W3.CSS, а также среда выполнения NodeJS (express, pug). В качестве вспомогательных библиотек для 3D-графики использовались библиотека gl-matrix (для работы с векторами, матрицами, кватернионами и стандартными преобразованиями над ними) и K3D (для загрузки 3D моделей формата .OBJ).

2. Функции для работы с WebGL

Для удобной работы с WebGL были созданы вспомогательные функции. Функция *getCanvas* позволяет получить элемент canvas с HTML-страницы, поиск которого осуществляется по ID. Функция *getCanvas* представлена в листинге 1.

Листинг 1. Функция getCanvas.

```
export function getCanvas() {
    let canvas = document.getElementById("canvas");
    if (!canvas) {
        console.error("Не найден HTML-элемент canvas.");
        return null;
    }
    return canvas;
```

```
}
```

Функция *getWebGlContext* позволяет получить контекст WebGL. Функция *getWebGlContext* представлена в листинге 2.

Листинг 2. Функция getWebGlContext.

```
function getWebGlContext() {
    let canvas = getCanvas();
    if (!canvas) {
        return null;
    }

    let context = canvas.getContext("webgl");
    if (!context) {
        console.error("Не удалось получить контекст WebGL.");
        return null;
    }
    return context;
}
```

Функция *createShader* позволяет создать скомпилированный шейдер WebGL. На вход принимает контекст WebGL, тип шейдера и исходный код шейдера. Возвращает объект шейдера, если он был успешно создан и скомпилирован, иначе – null. Функция *createShader* представлена в листинге 3.

Листинг 3. Функция createShader.

```
function createShader(gl, type, source) {
    let shader = gl.createShader(type);
    if (!shader) {
        console.error("Не удалось создать шейдер с типом '" + type +
"'");
        return null;
    }

    gl.shaderSource(shader, source);
    gl.compileShader(shader);

    let compiled = gl.getShaderParameter(shader, gl.COMPILE_STATUS);
    if (!compiled) {
        let error = gl.getShaderInfoLog(shader);
        console.error("Ошибка компиляции шейдера: " + error);
        gl.deleteShader(shader);
        return null;
    }

    return shader;
}
```

Функция *createProgram* позволяет создать связанную (слинкованную) шейдерную программу WebGL. На вход принимает контекст WebGL и массив шейдеров для программы. Возвращает объект программы, если она была успешно создана и связана, иначе – null. Функция *createProgram* представлена в листинге 4.

Листинг 4. Функция createProgram.

```
function createProgram(gl, shaders) {
    let program = gl.createProgram();
    if (!program) {
        console.error("Не удалось создать шейдерную программу");
        return null;
    }

    shaders.forEach((shader) => {
        gl.attachShader(program, shader);
    });

    gl.linkProgram(program);

    let linked = gl.getProgramParameter(program, gl.LINK_STATUS);
    if (!linked) {
        let error = gl.getProgramInfoLog(program);
        console.error("Ошибка линковки программы: " + error);
        gl.deleteProgram(program);
        return null;
    }

    return program;
}
```

Функция *initializeShaderProgram* инициализирует шейдерную программу WebGL. На вход принимает контекст WebGL. Создает вершинный и фрагментный шейдеры, шейдерную программу, после чего связывает их. Созданная шейдерная программа передается WebGL в качестве используемой программы. Функция *initializeShaderProgram* представлена в листинге 5.

Листинг 5. Функция initializeShaderProgram.

```
function initializeShaderProgram(gl) {
    let     vertexShader     =     createShader(gl,     gl.VERTEX_SHADER,
VERTEX_SHADER_SOURCE);
    let    fragmentShader    =    createShader(gl,    gl.FRAGMENT_SHADER,
FRAGMENT_SHADER_SOURCE);

    if (PROGRAM !== null) {
        gl.deleteProgram(PROGRAM);
```

```
    }
    PROGRAM = createProgram(gl, [vertexShader, fragmentShader]);

    gl.useProgram(PROGRAM);
}
```

Функция *initializeViewport* инициализирует размеры окна рендеринга WebGL. На вход принимает контекст WebGL. Используются размеры элемента *canvas*. Функция *initializeViewport* представлена в листинге 6.

Листинг 6. Функция initializeViewport.

```
function initializeViewport(gl) {
    let canvas = getCanvas();
    if (canvas === null) {
        return;
    }

    gl.viewport(0, 0, canvas.width, canvas.height);
}
```

Функция *initializeWebGl* подготавливает WebGL для использования. На вход принимает контекст WebGL. Инициализирует шейдерную программу и окно рендеринга, а также включает дополнительные функции WebGL. Функция *initializeWebGl* представлена в листинге 7.

Листинг 7. Функция initializeWebGl.

```
export function initializeWebGl() {
    GL = getWebGlContext();

    initializeShaderProgram(GL);
    initializeViewport(GL);

    // Дополнительные функции
    GL.enable(GL.DEPTH_TEST);
    GL.enable(GL.POLYGON_OFFSET_FILL);
    GL.enable(GL.SAMPLE_ALPHA_TO_COVERAGE);
}
```

3. Шейдеры

Для шейдерной программы были написаны два шейдера: вершинный и фрагментный.

3.1 Вершинный шейдер

Принимает координаты вершин и их цвет при помощи атрибута a_*vertexPosition* и униформы u_*vertexColor* соответственно. Помимо этого, принимает униформы матриц модели *u_mMatrix*, вида *u_vMatrix* и перспективы *u_pMatrix*, с помощью которых осуществляется перевод локальных координат вершин объекта сначала к мировым координатам (с использованием матрицы модели), после чего к видовым координатам (с использованием матрицы вида), и в завершение к проекционным координатам (с использованием матрицы проекции). Преобразованные координаты устанавливаются в *gl_Position*, а цвет передается фрагментному шейдеру через varying-переменную *v_vertexColor*. Шейдер представлен в листинге 8.

Листинг 8. Вершинный шейдер.

```
export default
`
attribute vec3 a_vertexPosition;
uniform vec4 u_vertexColor;
uniform mat4 u_mMatrix;
uniform mat4 u_vMatrix;
uniform mat4 u_pMatrix;

varying vec4 v_vertexColor;

void main() {
    gl_Position = u_pMatrix * u_vMatrix * u_mMatrix *
vec4(a_vertexPosition, 1);
    v_vertexColor = u_vertexColor;
}
`;
```

3.2 Фрагментный шейдер

Принимает цвета вершин при помощи varying-переменной *v_vertexColor*. Цвет устанавливается в *gl_FragColor*. Шейдер представлен в листинге 9.

Листинг 9. Фрагментный шейдер.

```
export default
`
precision mediump float;

varying vec4 v_vertexColor;

void main() {
```

```
    gl_FragColor = v_vertexColor;
}
`;
```

4. Объекты графических примитивов

Классы всех графических примитивов наследуются от базового абстрактного класса *SceneObject*. Данный класс определяет основные методы для представления 3D-объекта в виде трех методов: *getVertices*, *getTransformMatrix* и *loadFromObjFile*.

Метод *getVertices* возвращает массив вершин 3D-объекта в локальной системе координат объекта. Вершины упорядочены по тройкам, где каждая тройка определяет простейший примитив для отрисовки – треугольник, который в свою очередь является частью объекта. Таким образом, каждый объект представляется в виде набора треугольников.

Метод *loadFromObjFile* позволяет загрузить массив вершин 3D-объекта из файла формата .OBJ, который является стандартизированным форматом хранения 3D-моделей.

Метод *getTransformMatrix* возвращает матрицу для преобразования локальных координат к мировым координатам в соответствии с положением, поворотом и масштабом объекта на сцене. Положение объекта хранится в поле *position*, поворот объекта хранится в поле *rotation*, масштаб объекта хранится в поле *scale*.

Цвет объекта хранится в поле *color*, а вершины – в поле *vertices*.

Класс *SceneObject* представлен в листинге 10.

Листинг 10. Класс SceneObject.

```
export class SceneObject {
    constructor(position = vec3.create(), rotation = vec3.create(), scale
= vec3.fromValues(1, 1, 1), color = new Color(1, 1, 1, 1)) {
        this.position = position;
        this.rotation = rotation;
        this.scale = scale;
        this.color = color;
        this.vertices = new Float32Array(0);
    }
```

```
    loadFromObjFile(name, callback) {
        K3D.load(`/public/models/${name}.obj`, (rawdata) => {
            let result = K3D.parse.fromOBJ(rawdata);
            this.vertices = new Float32Array(3 * result.i_verts.length);

            for(let i = 0; i < result.i_verts.length; ++i)
            {
                this.vertices[3    *    i]    =    result.c_verts[3    *
result.i_verts[i]];
                this.vertices[3   *   i   +   1]   =   result.c_verts[3   *
result.i_verts[i] + 1];
                this.vertices[3   *   i   +   2]   =   result.c_verts[3   *
result.i_verts[i] + 2];
            }

            callback();
        });
    }

    getTransformMatrix() {
        let rotationQuad = quat.create();
        quat.rotateX(rotationQuad, rotationQuad, this.rotation[0]);
        quat.rotateY(rotationQuad, rotationQuad, this.rotation[1]);
        quat.rotateZ(rotationQuad, rotationQuad, this.rotation[2]);
        return           mat4.fromRotationTranslationScale(mat4.create(),
rotationQuad, this.position, this.scale);
    }

    getVertices() {
        return new Float32Array(this.vertices);
    }
}
```

Было разработано две 3D-модели:

- Куб (файл *cube.obj*);
- Сфера (файл *sphere.obj*).

Дополнительно был создан класс *Camera*, унаследованный от класса *SceneObject*, который представляет собой камеру на сцене. Было реализовано 4 метода:

- getForwardViewTarget – для получения позиции в мировой системе координат, куда смотрит камера на данный момент;
- getUpVector – возвращает вектор направления, указывающий вверх относительно камеры;

- getViewMatrix – возвращает матрицу вида, используя текущую позицию камеры, а также два предыдущих метода.
- getProjectionMatrix – возвращает матрицу проекции (может быть перспективной или ортогональной), используя поля *view_projection* (хранит тип проекции камеры), *view_vfog* (вертикальный угол обзора камеры), *view_distance* (дальность видимости камеры).

Класс *Camera* представлен в листинге 11.

Листинг 11. Класс PlaneObject.

```
export class Camera extends SceneObject
{
    constructor(position = vec3.create(), rotation = vec3.create(), scale
= vec3.fromValues(1, 1, 1), color = new Color(1, 1, 1, 1)) {
        super(position, rotation, scale, color);
        this.view_projection = "perspective"
        this.view_vfov = 90;
        this.view_distance = 1000;
    }

    getForwardViewTarget() {
        let target = vec3.copy(vec3.create(), [1, 0, 0]);
        vec3.rotateX(target, target, [0, 0, 0], this.rotation[0]);
        vec3.rotateY(target, target, [0, 0, 0], this.rotation[1]);
        vec3.rotateZ(target, target, [0, 0, 0], this.rotation[2]);
        vec3.normalize(target, target);
        vec3.scale(target, target, this.view_distance);
        return vec3.add(vec3.create(), this.position, target);
    }

    getUpVector() {
        let upVector = vec3.copy(vec3.create(), [0, 1, 0]);
        vec3.rotateX(upVector, upVector, [0, 0, 0], this.rotation[0])
        vec3.rotateY(upVector, upVector, [0, 0, 0], this.rotation[1])
        vec3.rotateZ(upVector, upVector, [0, 0, 0], this.rotation[2])
        vec3.normalize(upVector, upVector);
        return upVector;
    }

    getViewMatrix() {
        return          mat4.lookAt(mat4.create(),          this.position,
this.getForwardViewTarget(), this.getUpVector());
    }

    getProjectionMatrix() {
        if(this.view_projection === "perspective") {
            let aspectRatio = getCanvas().width / getCanvas().height;
            return    mat4.perspective(mat4.create(),    this.view_vfov,
aspectRatio, 0.1, this.view_distance);
        } else if(this.view_projection === "orthogonal") {
```

```
            let boxSize = 5;
            return mat4.ortho(mat4.create(), -boxSize, boxSize, -boxSize,
boxSize, 0.1, this.view_distance);
        } else {
            return mat4.create();
        }
    }
}
```

5. Рисование графических примитивов

Отрисовка графических примитивов сцены осуществляется при помощи функции *renderScene*. В начале происходит очистка области рендеринга, а также буферов цвета и глубины. Далее осуществляется поочередная отрисовка 3D-объектов сцены, хранящихся в глобальной переменной *SCENE_OBJECTS*, при этом учитывая текущее расположение камеры, хранящийся в глобальной переменной *CAMERA_OBJECT*. Функция *renderScene* представлена в листинге 12.

Листинг 12. Функция renderScene.

```
export function renderScene(sceneObjects, camera, renderParameters) {
    let                          backgroundColor                          =
hexToColor(renderParameters["backgroundColor"]);
    GL.clearColor(backgroundColor.r,                    backgroundColor.g,
backgroundColor.b, backgroundColor.a);
    GL.clear(GL.COLOR_BUFFER_BIT);
    GL.clear(GL.DEPTH_BUFFER_BIT);

    let    vertexPositionAttribute    =    GL.getAttribLocation(PROGRAM,
"a_vertexPosition");
    let     vertexColorUniform     =     GL.getUniformLocation(PROGRAM,
"u_vertexColor");
    let modelMatrixUniform = GL.getUniformLocation(PROGRAM, "u_mMatrix");
    let viewMatrixUniform = GL.getUniformLocation(PROGRAM, "u_vMatrix");
    let    projectionMatrixUniform    =    GL.getUniformLocation(PROGRAM,
"u_pMatrix");

    let vMatrix = camera.getViewMatrix();
    GL.uniformMatrix4fv(viewMatrixUniform, false, vMatrix);
    GL.enableVertexAttribArray(viewMatrixUniform);

    let pMatrix = camera.getProjectionMatrix();
    GL.uniformMatrix4fv(projectionMatrixUniform, false, pMatrix);
    GL.enableVertexAttribArray(projectionMatrixUniform);

    sceneObjects.forEach((object) => {
        let mMatrix = object.getTransformMatrix();
        GL.uniformMatrix4fv(modelMatrixUniform, false, mMatrix);
        GL.enableVertexAttribArray(modelMatrixUniform);
```

```
        let vertices = object.getVertices();
        let verticesCount = vertices.length / 3;
        let vertexBuffer = GL.createBuffer();
        GL.bindBuffer(GL.ARRAY_BUFFER, vertexBuffer);
        GL.bufferData(GL.ARRAY_BUFFER, vertices, GL.STATIC_DRAW);
        GL.vertexAttribPointer(vertexPositionAttribute,    3,    GL.FLOAT,
false, 0, 0);
        GL.enableVertexAttribArray(vertexPositionAttribute);

        if(renderParameters["drawPolygons"]) {
            let objectColor = object.color.asVector();
            GL.uniform4fv(vertexColorUniform, objectColor);
            GL.polygonOffset(0, 0);
            GL.drawArrays(GL.TRIANGLES, 0, verticesCount);
        }

        if(renderParameters["drawEdges"]) {
            let edgeColor = hexToColor(renderParameters["edgeColor"]);
            GL.uniform4fv(vertexColorUniform, edgeColor.asVector());
            GL.polygonOffset(1, 1);
            GL.drawArrays(GL.LINE_LOOP, 0, verticesCount);
        }

        GL.deleteBuffer(vertexBuffer);
    });
}
```

6. Пользовательский интерфейс

Для пользователя был разработан интерфейс со следующей структурой:

- Слева располагается окно отрисовки сцены (размером 800 на 800 пикселей). При помощи данного окна пользователь может в реальном времени наблюдать за изменениями на сцене.
- По центру располагается панель настройки объектов сцены. При помощи данной панели пользователь может добавлять 3D-объекты на сцену, редактировать параметры созданных 3D-объектов, а также удалять все объекты со сцены для ее очистки.
- Справа располагается панель настройки специальных параметров, куда относятся параметры камеры и параметры рендеринга.

Объекты сцены имеют следующие параметры:

- Положение по X;

- Положение по Y;
- Положение по Z;
- Поворот по X;
- Поворот по Y;
- Поворот по Z;
- Масштаб по X;
- Масштаб по Y;
- Масштаб по Z;
- Цвет примитива;
- Прозрачность цвета.

Камера помимо основных параметров объекта сцены, имеет:

- Вертикальный угол обзора;
- Тип проекции.

Интерфейс программы представлен на рисунке 1.

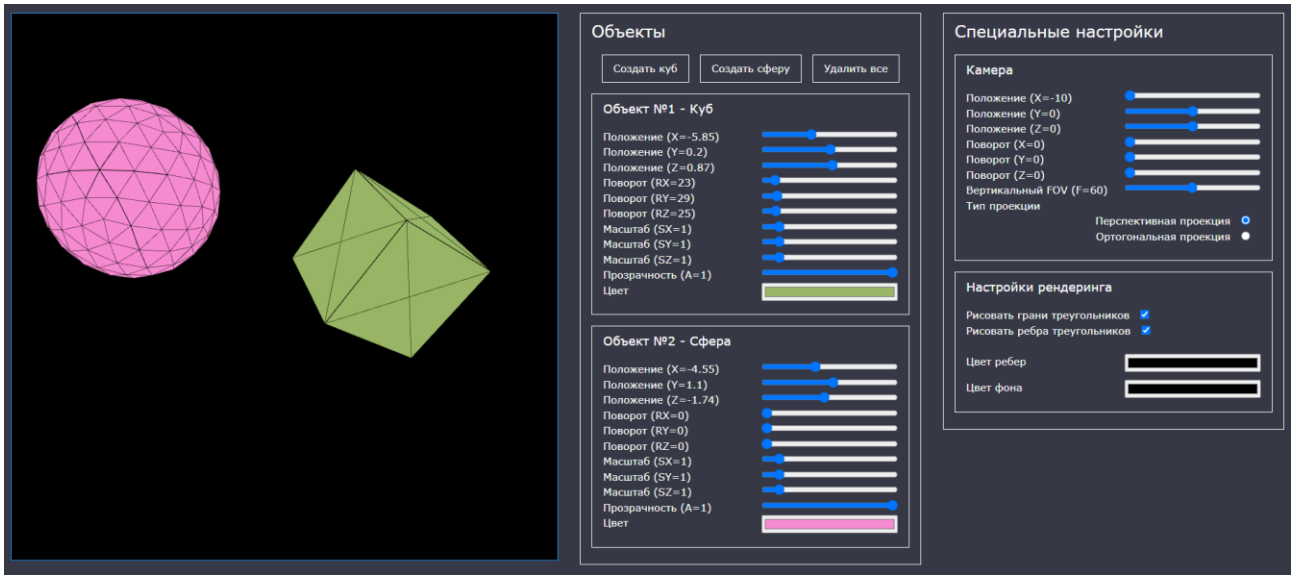

Рисунок 1 – Интерфейс программы.

Настройки рендеринга имеют следующие параметры:

- Рисование граней (треугольников);

- Рисование ребер граней;
- Цвет ребер;
- Цвет заднего фона (цвет очистки).

Для изменения значений параметров примитивов используются HTML-элементы *input* типа *range*, *checkbox* и *color*. При любых изменениях значений параметров вызывается функция *onParametersChanged*, которая извлекает значения из HTML-элементов, устанавливает их соответствующим объектам из массива *SCENE_OBJECTS*, объекту *CAMERA_OBJECT* и *RENDER_PARAMETERS*, после чего вызывается функция *renderScene*. Функция *onParametersChanged* представлена в листинге 13.

Листинг 14. Функция onParametersChanged.

```
export function onParametersChanged() {
    // Координаты камеры
    let e_x = document.getElementById(`input-x-camera`);
    let e_y = document.getElementById(`input-y-camera`);
    let e_z = document.getElementById(`input-z-camera`);
    CAMERA_OBJECT.position[0] = Number(e_x.value);
    CAMERA_OBJECT.position[1] = Number(e_y.value);
    CAMERA_OBJECT.position[2] = Number(e_z.value);
    $(`#output-x-camera`).text(e_x.value);
    $(`#output-y-camera`).text(e_y.value);
    $(`#output-z-camera`).text(e_z.value);

    // Вращение камеры
    let e_r_x = document.getElementById(`input-x-rotation-camera`);
    let e_r_y = document.getElementById(`input-y-rotation-camera`);
    let e_r_z = document.getElementById(`input-z-rotation-camera`);
    CAMERA_OBJECT.rotation[0] = toRadian(Number(e_r_x.value));
    CAMERA_OBJECT.rotation[1] = toRadian(Number(e_r_y.value));
    CAMERA_OBJECT.rotation[2] = toRadian(Number(e_r_z.value));
    $(`#output-x-rotation-camera`).text(e_r_x.value);
    $(`#output-y-rotation-camera`).text(e_r_y.value);
    $(`#output-z-rotation-camera`).text(e_r_z.value);

    // Вертикальный FOV
    let fov = document.getElementById(`input-fov-camera`);
    CAMERA_OBJECT.view_vfov = toRadian(Number(fov.value));
    $(`#output-fov-camera`).text(fov.value);

    // Тип проекции
    let projection = document.getElementById(`pers-projection`);
    CAMERA_OBJECT.view_projection = (projection.checked) ? "perspective"
: "orthogonal";

    // Объекты
```

```
    for (let i = 0; i < SCENE_OBJECTS.length; ++i) {
        // Позиция
        let e_x = document.getElementById(`input-x-${i+1}`);
        let e_y = document.getElementById(`input-y-${i+1}`);
        let e_z = document.getElementById(`input-z-${i+1}`);
        SCENE_OBJECTS[i].position[0] = Number(e_x.value);
        SCENE_OBJECTS[i].position[1] = Number(e_y.value);
        SCENE_OBJECTS[i].position[2] = Number(e_z.value);
        $(`#output-x-${i+1}`).text(e_x.value);
        $(`#output-y-${i+1}`).text(e_y.value);
        $(`#output-z-${i+1}`).text(e_z.value);

        // Вращение
        let e_r_x = document.getElementById(`input-rotation-x-${i+1}`);
        let e_r_y = document.getElementById(`input-rotation-y-${i+1}`);
        let e_r_z = document.getElementById(`input-rotation-z-${i+1}`);
        SCENE_OBJECTS[i].rotation[0] = toRadian(Number(e_r_x.value));
        SCENE_OBJECTS[i].rotation[1] = toRadian(Number(e_r_y.value));
        SCENE_OBJECTS[i].rotation[2] = toRadian(Number(e_r_z.value));
        $(`#output-rotation-x-${i+1}`).text(e_r_x.value);
        $(`#output-rotation-y-${i+1}`).text(e_r_y.value);
        $(`#output-rotation-z-${i+1}`).text(e_r_z.value);

        // Масштаб
        let e_s_x = document.getElementById(`input-scale-x-${i+1}`);
        let e_s_y = document.getElementById(`input-scale-y-${i+1}`);
        let e_s_z = document.getElementById(`input-scale-z-${i+1}`);
        SCENE_OBJECTS[i].scale[0] = Number(e_s_x.value);
        SCENE_OBJECTS[i].scale[1] = Number(e_s_y.value);
        SCENE_OBJECTS[i].scale[2] = Number(e_s_z.value);
        $(`#output-scale-x-${i+1}`).text(e_s_x.value);
        $(`#output-scale-y-${i+1}`).text(e_s_y.value);
        $(`#output-scale-z-${i+1}`).text(e_s_z.value);

        // Цвет
        let e_c = document.getElementById(`input-color-${i+1}`);
        let e_a = document.getElementById(`input-alpha-${i+1}`);
        SCENE_OBJECTS[i].color = hexToColor(e_c.value);
        SCENE_OBJECTS[i].color.a = Number(e_a.value);
        $(`#output-alpha-${i+1}`).text(e_a.value);
    }

    RENDER_PARAMETERS["drawEdges"]    =    document.getElementById(`draw-
edges`).checked;
    RENDER_PARAMETERS["drawPolygons"]   =   document.getElementById(`draw-
polygons`).checked;
    RENDER_PARAMETERS["backgroundColor"]                                    =
document.getElementById(`input-background-color`).value;
    RENDER_PARAMETERS["edgeColor"] = document.getElementById(`input-edge-
color`).value;

    renderScene(SCENE_OBJECTS, CAMERA_OBJECT, RENDER_PARAMETERS);
}
```

Для добавления и удаления примитивов сцены используются функции *createSceneObject* и *clearSceneObjects* соответственно. Функции представлены в листинге 14.

Листинг 14. Функции createSceneObject и clearSceneObjects.

```
export function createSceneObject(type) {
    let id = SCENE_OBJECTS.length + 1;
    let typeName = null;

    switch (type) {
        case "cube": typeName = "Куб"; break;
        case "sphere": typeName = "Сфера"; break;
    }

    if (typeName === null) {
        return;
    }

    let objectHtml = `
    <div id="object-${id}" class="w3-padding w3-margin-bottom w3-border
w3-border-white">
        <div                                                class="w3-large"
onclick='changeAccordionVisibility("accordion-${id}")'>Объект №${id} -
${typeName}</div>
        <div id="accordion-${id}" class="w3-margin-top w3-hide w3-show">
            <div>
                <label for="input-x-${id}" class="w3-left">Положение
(X=<span id="output-x-${id}"></span>)</label>
                <input id="input-x-${id}" class="w3-right" type="range"
min="-20" max="20" step="0.05" value="0" style="width: 200px;"
oninput="onParametersChanged()"/>
            </div>
            <br/>
            <div>
                <label for="input-y-${id}" class="w3-left">Положение
(Y=<span id="output-y-${id}"></span>)</label>
                <input id="input-y-${id}" class="w3-right" type="range"
min="-20" max="20" step="0.05" value="0" style="width: 200px;"
oninput="onParametersChanged()"/>
            </div>
            <br/>
            <div>
                <label for="input-z-${id}" class="w3-left">Положение
(Z=<span id="output-z-${id}"></span>)</label>
                <input id="input-z-${id}" class="w3-right" type="range"
min="-20" max="20" step="0.01" value="0" style="width: 200px;"
oninput="onParametersChanged()"/>
            </div>
            <br/>
            <div>
                <label for="input-rotation-x-${id}" class="w3-
left">Поворот (RX=<span id="output-rotation-x-${id}"></span>)</label>
                <input id="input-rotation-x-${id}" class="w3-right"
type="range" min="0" max="360" step="1" value="0" style="width: 200px;"
oninput="onParametersChanged()"/>
```

```
            </div>
            <br/>
            <div>
                <label      for="input-rotation-y-${id}"      class="w3-
left">Поворот (RY=<span id="output-rotation-y-${id}"></span>)</label>
                <input    id="input-rotation-y-${id}"    class="w3-right"
type="range" min="0" max="360" step="1" value="0" style="width: 200px;"
oninput="onParametersChanged()"/>
            </div>
            <br/>
            <div>
                <label      for="input-rotation-z-${id}"      class="w3-
left">Поворот (RZ=<span id="output-rotation-z-${id}"></span>)</label>
                <input    id="input-rotation-z-${id}"    class="w3-right"
type="range" min="0" max="360" step="1" value="0" style="width: 200px;"
oninput="onParametersChanged()"/>
            </div>
            <br/>
            <div>
                <label  for="input-scale-x-${id}" class="w3-left">Масштаб
(SX=<span id="output-scale-x-${id}"></span>)</label>
                <input     id="input-scale-x-${id}"     class="w3-right"
type="range" min="0" max="10" step="0.05" value="1" style="width: 200px;"
oninput="onParametersChanged()"/>
            </div>
            <br/>
            <div>
                <label  for="input-scale-y-${id}" class="w3-left">Масштаб
(SY=<span id="output-scale-y-${id}"></span>)</label>
                <input     id="input-scale-y-${id}"     class="w3-right"
type="range" min="0" max="10" step="0.05" value="1" style="width: 200px;"
oninput="onParametersChanged()"/>
            </div>
            <br/>
            <div>
                <label  for="input-scale-z-${id}" class="w3-left">Масштаб
(SZ=<span id="output-scale-z-${id}"></span>)</label>
                <input     id="input-scale-z-${id}"     class="w3-right"
type="range" min="0" max="10" step="0.05" value="1" style="width: 200px;"
oninput="onParametersChanged()"/>
            </div>
            <br/>
            <div>
                <label         for="input-alpha-${id}"         class="w3-
left">Прозрачность (A=<span id="output-alpha-${id}"></span>)</label>
                <input      id="input-alpha-${id}"      class="w3-right"
type="range" min="0" max="1" step="0.01" value="1" style="width: 200px;"
oninput="onParametersChanged()"/>
            </div>
            <br/>
            <div class="w3-margin-bottom">
                <label         for="input-color-${id}"         class="w3-
left">Цвет</label>
                <input      id="input-color-${id}"      class="w3-right"
type="color"    value="${colorToHex(Math.round(Math.random()    *    255),
Math.round(Math.random()  *  255),  Math.round(Math.random()  *  255))}"
style="width: 200px;" oninput="onParametersChanged()"/>
```

```
            </div>
            <br/>
        </div>
    </div>
    `;

    let objectsDiv = $("#objects");
    objectsDiv.append(objectHtml);

    let object = null;
    switch (type) {
        case "cube": object = new SceneObject(); break;
        case "sphere": object = new SceneObject(); break;
    }

    if (object !== null) {
        object.loadFromObjFile(type, () => {
            SCENE_OBJECTS.push(object);
            onParametersChanged();
        });
    }
}

export function clearSceneObjects() {
    SCENE_OBJECTS = [];
    $("#objects").html("");
    onParametersChanged();
    renderScene(SCENE_OBJECTS, CAMERA_OBJECT, RENDER_PARAMETERS);
}
```

Выводы.

В результате выполнения лабораторной работы была разработана программа на языке JavaScript, которая осуществляет рисование 3D-объектов на глобальной сцене относительно объекта наблюдения – камеры. Были освоены стандартные матричные преобразования координат, при помощи которых локальные координаты объекта преобразуются к мировым координатам (с использованием матрицы модели), после чего к видовым координатам (с использованием матрицы вида), после чего к проекционным координатам (с использованием матрицы проекции).