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

ОТЧЕТ

по лабораторной работе №4
по дисциплине «Компьютерная графика»
Тема: Текстуры изображения

Студент гр. 9303 _____ Колованов Р.А.

Преподаватель _____ Герасимова Т.В.

Санкт-Петербург
2023

Цель работы.

Целью работы является освоение текстурирования.

Задание.

- Отредактируйте свою собственную картинку и сделайте ее текстурной картой (3 разных размера);
- Разложите текстуру по изображению;
- Закрепите и отцентрируйте свое изображение на объекте.

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), K3D (для загрузки 3D моделей формата .OBJ) и load-image (для загрузки изображений).

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 2.1.

Листинг 2.1. Функция getCanvas.

```
/**
 * Возвращает найденный на странице элемент canvas.
 * @return {HTMLCanvasElement}
 */
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 2.2.

Листинг 2.2. Функция getWebGlContext.

```
/**
 * Возвращает найденный на странице контекст WebGL.
 * @return {WebGLRenderingContext}
 */
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 2.3.

Листинг 2.3. Функция createShader.

```
/**
 * Создает и компилирует шейдер WebGL.
 * @param {WebGLRenderingContext} gl
 * @param {int} type
 * @param {string} source
 * @return {WebGLShader}
 */
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 2.4.

Листинг 2.4. Функция createProgram.

```
/**
 * Создает, компилирует и линкует шейдерную программу WebGL.
 * @param {WebGLRenderingContext} gl
 * @param {WebGLShader[]} shaders
 * @return {WebGLProgram}
 */
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 2.5.

Листинг 2.5. Функция initializeShaderProgram.

```
/**
 * Создает и настраивает для работы шейдерную программу WebGL.
 * @param {WebGLRenderingContext} gl
 */
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 2.6.

Листинг 2.6. Функция initializeViewport.

```
/**
 * Инициализирует окно рендеринга WebGL.
 * @param {WebGLRenderingContext} gl
 */
function initializeViewport(gl) {
    let canvas = getCanvas();
    if (canvas === null) {
        return;
    }

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

Функция *loadTexture* загружает изображения для использования в качестве текстур. На вход принимает контекст WebGL. Из загруженных изображений создает текстурные объекты. Функция *loadTexture* представлена в листинге 2.7.

Листинг 2.7. Функция loadTexture.

```
/**
 * Загружает текстуры из файлов и передает из в шейдерную программу.
 */
function loadTextures(gl) {
    const texturesCount = 4;
    const textureIndexMap = {0: gl.TEXTURE0, 1: gl.TEXTURE1, 2:
gl.TEXTURE2, 3: gl.TEXTURE3};
    const getPixelData = (image) => {
        let canvas = document.createElement('canvas');
        canvas.width = image.width;
        canvas.height = image.height;

        let context = canvas.getContext('2d');
        context.drawImage(image, 0, 0);

        let imageData = context.getImageData(0, 0, image.width,
image.height);
        return new Uint8Array(imageData.data.buffer);
    };

    for (let i = 0; i < texturesCount; ++i) {
        loadImage(`/public/textures/${i}.jpg`, (image) => {
            let texture = gl.createTexture();
            gl.activeTexture(textureIndexMap[i]);
            gl.bindTexture(gl.TEXTURE_2D, texture);

            gl.texParameteri(gl.TEXTURE_2D,        gl.TEXTURE_MAG_FILTER,
gl.NEAREST);
            gl.texParameteri(gl.TEXTURE_2D,        gl.TEXTURE_MIN_FILTER,
gl.NEAREST);

            let pixelData = getPixelData(image);
            gl.texImage2D(gl.TEXTURE_2D,    0,    gl.RGBA,    image.width,
image.height, 0, gl.RGBA, gl.UNSIGNED_BYTE, pixelData);
        });
    }
}
```

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 2.8.

Листинг 2.8. Функция initializeWebGl.

```
/**
 * Инициализирует WebGL.
 */
export function initializeWebGl() {
    GL = getWebGlContext();

    initializeShaderProgram(GL);
    initializeViewport(GL);

    loadTextures(GL);

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

Функция *setClearColor* устанавливает цвет очистки окна рендеринга. На вход принимает цвет. Функция *setClearColor* представлена в листинге 2.9.

Листинг 2.9. Функция setClearColor.

```
/**
 * Устанавливает цвет очистки окна рендеринга.
 * @param {Color} color
 */
export function setClearColor(color) {
    GL.clearColor(color.r, color.g, color.b, color.a);
}
```

3. Шейдеры

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

3.1 Вершинный шейдер

Принимает координаты вершин, нормали и текстурные координаты при помощи атрибутов a*_vertexPosition,* a*_vertexNormal* и *a_texturePosition* 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 вершин устанавливаются в *gl_Position*.

Для корректного преобразования нормалей от локальных координат к глобальным используется матрица нормалей (является обратной транспонированной матрицей модели), которая передается через униформу *u_nMatrix*.

Мировые координаты вершин и нормалей, а также координаты текстур передаются во фрагментный шейдер через varying-переменные *v_vertexColor*, *v_vertexNormal* и *v_texturePosition*.

Шейдер представлен в листинге 3.1.

Листинг 3.1. Вершинный шейдер.

```
export default
`#version 300 es

in vec3 a_vertexPosition;
in vec3 a_vertexNormal;
in vec2 a_texturePosition;

uniform mat4 u_mMatrix;
uniform mat4 u_vMatrix;
uniform mat4 u_pMatrix;
uniform mat4 u_nMatrix;

out vec3 v_vertexPosition;
out vec3 v_vertexNormal;
out vec2 v_texturePosition;

void main() {
    vec4 globalPosition = u_mMatrix * vec4(a_vertexPosition, 1.0);
    vec3 globalNormal = normalize((u_nMatrix * vec4(a_vertexNormal,
0.0)).xyz);

    gl_Position = u_pMatrix * u_vMatrix * globalPosition;

    v_vertexPosition = globalPosition.xyz;
    v_vertexNormal = globalNormal;
    v_texturePosition = a_texturePosition;
}
`;
```

3.2 Фрагментный шейдер

Для определения освещения на сцене используется униформа *u_useLighting* (определяет использование 3D-освещения), униформа-массив *u_lights* (содержит массив источников света *Light*) и униформа *u_material* (определяет параметры материала объектов *Material*). Для вычисления бликов от источников света (зеркального света) дополнительно передается глобальная позиция камеры через униформу *u_cameraPosition*. Используемая модель света состоит из трех компонентов: диффузный свет (diffuse), зеркальный свет (specular) и фоновый свет (ambient). Цвет освещаемого объекта рассчитывается для каждого фрагмента во фрагментном шейдере – используется затенение по Фонгу.

Для определения текстурирования используется униформа *u_useTexture* (определяет использование текстур), униформа u_texture (содержит текстурную единицу) и униформа *u_textureScale* (определяет масштаб текстуры, на этот коэффициент умножаются UV-координаты текстур).

Цвет устанавливается в *finalColor*. Шейдер представлен в листинге 3.2.

Листинг 3.2. Фрагментный шейдер.

```
export default
`#version 300 es

precision mediump float;

struct Light
{
    vec3 diffuse;
    vec3 ambient;
    vec3 specular;
    vec4 position;
};

struct Material
{
    vec3 diffuse;
    vec3 ambient;
    vec3 specular;
    float shininess;
};

const int LIGHT_NUMBER = 2;

in vec3 v_vertexPosition;
```

```
in vec3 v_vertexNormal;
in vec2 v_texturePosition;

uniform vec3 u_cameraPosition;
uniform bool u_useLighting;
uniform Light u_lights[LIGHT_NUMBER];
uniform Material u_material;
uniform bool u_useTexture;
uniform sampler2D u_texture;
uniform float u_textureScale;

out vec4 finalColor;

vec3 globalNormal;

vec4 calculateLight();

void main() {
    vec4 resultColor = vec4(u_material.diffuse, 1.0);
    globalNormal = normalize(v_vertexNormal);

    if (u_useTexture) {
        resultColor     =     texture(u_texture,     u_textureScale     *
v_texturePosition);
    }

    if (u_useLighting) {
        resultColor = calculateLight();
    }

    finalColor = resultColor;
}

vec4 calculateLight() {
    float alpha = 1.0;
    vec4 resultColor = vec4(0, 0, 0, 0);

    for(int i = 0; i < LIGHT_NUMBER; ++i) {
        Light light = u_lights[i];

        vec3 lightDirection;
        if (light.position.w == 0.0) {
            lightDirection = normalize(light.position.xyz);
        } else {
            lightDirection     =     normalize(light.position.xyz     -
v_vertexPosition.xyz);
        }

        float Kd = max(dot(lightDirection, globalNormal), 0.0);

        vec3     eyeDirection     =     normalize(u_cameraPosition     -
v_vertexPosition.xyz);
        vec3 halfVector = normalize(eyeDirection + lightDirection);
        float   Ks   =   pow(max(dot(halfVector,   globalNormal),   0.0),
u_material.shininess);

        vec3 materialDiffuse = u_material.diffuse;
```

```
        if (u_useTexture) {
            vec4    t    =    texture(u_texture,    u_textureScale    *
v_texturePosition);
            materialDiffuse.rgb = t.rgb;
            alpha = t.a;
        }

        resultColor.xyz += Ks * materialDiffuse * light.specular
                         + Kd * materialDiffuse * light.diffuse
                         + u_material.ambient * light.ambient;
    }

    resultColor.a = alpha;

    return resultColor;
}
`;
```

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 следующих методов: *getVertices*, *getNormals*, *getTransformMatrix*, *getNormalMatrix*, *getTextureCoordinates*, *getForwardVector*, *getUpVector*, *getRightVector* 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 *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*.

Метод *getNormals* возвращает массив нормалей 3D-объекта в локальной системе координат объекта. Вершины упорядочены по тройкам, где каждая тройка определяет нормаль для соответствующей вершины.

Метод *getNormalMatrix* возвращает матрицу для преобразования нормалей к мировым координатам в соответствии с положением, поворотом и масштабом объекта на сцене.

Метод *getTextureCoordinates* возвращает UV-координаты текстур для граней объекта.

Методы *getForwardVector*, *getUpVector* и *getRightVector* являются вспомогательными и возвращают нормализированные векторы-направления (вектор, направленный вперед, вверх и вправо соответственно) в мировых координатах в соответствии с поворотом объекта.

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

Цвет объекта хранится в поле *color*, нормали – в поле *normal*, вершины – в поле *vertices*, видимость объекта – в поле *visible*.

Класс *SceneObject* представлен в листинге 4.1.

Листинг 4.1. Класс SceneObject.

```
/**
 * 3D-объект сцены
 */
export class SceneObject {
    /**
     * @param {vec3} position
     * @param {vec3} rotation
     * @param {vec3} scale
     * @param {Color} color
     */
    constructor(position = vec3.create(), rotation = vec3.create(), scale
= vec3.fromValues(1, 1, 1), color = new Color(1, 1, 1, 1)) {
        this.visible = true;
        this.position = position;
        this.rotation = rotation;
        this.scale = scale;
        this.color = color;
        this.vertices = new Float32Array(0);
        this.normals = new Float32Array(0);
        this.textureCoordinates = new Float32Array(0);
        this.textureScale = 1.0;
        this.texture = -1;
    }

    /**
     * Загружает 3D-модель из файла формата .OBJ
```

```
     * @param {string} name
     * @param {function} callback
     */
    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);
            this.normals = new Float32Array(3 * result.i_norms.length);
            this.textureCoordinates = new Float32Array(2 *
result.i_uvt.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];
            }

            for(let i = 0; i < result.i_norms.length; ++i) {
                this.normals[3 * i] = result.c_norms[3 *
result.i_norms[i]];
                this.normals[3 * i + 1] = result.c_norms[3 *
result.i_norms[i] + 1];
                this.normals[3 * i + 2] = result.c_norms[3 *
result.i_norms[i] + 2];
            }

            for(let i = 0; i < result.i_uvt.length; ++i) {
                this.textureCoordinates[2 * i] = result.c_uvt[2 *
result.i_uvt[i]];
                this.textureCoordinates[2 * i + 1] = result.c_uvt[2 *
result.i_uvt[i] + 1];
            }

            callback();
        });
    }

    /**
     * Возвращает матрицу модели 4x4 (ModelMatrix)
     * @return {mat4}
     */
    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);
    }

    /**
     * Возвращает матрицу преобразования нормалей 4x4 (NormalMatrix)
     * @return {mat4}
     */
```

```
    getNormalMatrix() {
        let matrix = this.getTransformMatrix();
        mat4.invert(matrix, matrix);
        mat4.transpose(matrix, matrix);
        return matrix;
    }

    /**
     * Возвращает вершины 3D-модели в локальной системе координат объекта
     * @return {Float32Array}
     */
    getVertices() {
        return new Float32Array(this.vertices);
    }

    /**
     * Возвращает вершины 3D-модели в локальной системе координат объекта
     * @return {Float32Array}
     */
    getNormals() {
        return new Float32Array(this.normals);
    }

    /**
     * Возвращает UV координаты текстуры 3D-модели
     * @return {Float32Array}
     */
    getTextureCoordinates() {
        return new Float32Array(this.textureCoordinates);
    }

    /**
     * Возвращает направление в мировой СК, куда смотрит камера
     * @return {vec3}
     */
    getForwardVector() {
        let forwardVector = vec4.copy(vec4.create(), [1, 0, 0, 0]);
        vec4.transformMat4(forwardVector,                 forwardVector,
this.getTransformMatrix());
        vec4.normalize(forwardVector, forwardVector);
        return vec3.copy(vec3.create(), forwardVector);
    }

    /**
     * Возвращает направление в мировой СК, куда смотрит верх камеры
     * @return {vec3}
     */
    getUpVector() {
        let upVector = vec4.copy(vec4.create(), [0, 1, 0, 0]);
        vec4.transformMat4(upVector,                              upVector,
this.getTransformMatrix());
        vec4.normalize(upVector, upVector);
        return vec3.copy(vec3.create(), upVector);
    }

    /**
     * Возвращает направление в мировой СК, куда смотрит правая сторона
```

```
камеры
     * @return {vec3}
     */
    getRightVector() {
        let rightVector = vec4.copy(vec4.create(), [0, 0, 1, 0]);
        vec4.transformMat4(rightVector,                    rightVector,
this.getTransformMatrix());
        vec4.normalize(rightVector, rightVector);
        return vec3.copy(vec3.create(), rightVector);
    }
}
```

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 2 метода:

- 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 4.2.

Листинг 4.2. Класс Camera.

```
/**
 * Объект камеры на сцене
 */
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 = toRadian(60);
        this.view_distance = 1000;
    }
```

```
    /**
     * Возвращает матрицу просмотра 4x4 (ViewMatrix)
     * @return {mat4}
     */
    getViewMatrix() {
        let target = this.getForwardVector();
        vec3.scale(target, target, this.view_distance);
        vec3.add(target, target, this.position);
        return    mat4.lookAt(mat4.create(),    this.position,    target,
this.getUpVector());
    }

    /**
     * Возвращает матрицу проекции 4x4 (ProjectionMatrix)
     * @return {mat4}
     */
    getProjectionMatrix() {
        if(this.view_projection === "perspective") {
            let aspectRatio = getCanvas().width / getCanvas().height;
            return    mat4.perspective(mat4.create(),    this.view_vfov,
aspectRatio, 0.01, this.view_distance);
        } else if(this.view_projection === "orthogonal") {
            let boxSize = 5;
            return mat4.ortho(mat4.create(), -boxSize, boxSize, -boxSize,
boxSize, 0.01, this.view_distance);
        } else {
            return mat4.create();
        }
    }
}
```

Дополнительно был создан класс *Light*, унаследованный от класса SceneObject, который представляет собой источник света на сцене. Класс содержит 3 новых поля: ambient (фоновый свет), diffuse (диффузный свет) и specular (зеркальный свет). Было реализовано 2 метода:

- setColor – устанавливает цвет диффузного и зеркального света;
- setAmbientColor – устанавливает цвет фонового света.

От класса *Light* были унаследованы классы *PointLight* и *DirectionalLight*, которые представляют собой точечный и направленный источник света на сцене соответственно.

Классы *Light*, *PointLight*, *DirectionalLight* представлены в листинге 4.3.

Листинг 4.3. Классы Light, PointLight, DirectionalLight.

```
/**
 * Объект света на сцене
 */
```

```
export class Light extends SceneObject {
    constructor(position = vec3.create()) {
        super(position, vec3.fromValues(0, 0, 0), vec3.fromValues(1, 1,
1), new Color());
        this.ambient = new Color(0, 0, 0, 1);
        this.diffuse = new Color(0, 0, 0, 1);
        this.specular = new Color(0, 0, 0, 1);
    }

    /**
     * Устанавливает цвет света (диффузный и зеркальный)
     * @param {Color} color
     */
    setColor(color) {
        this.diffuse = color;
        this.specular = color;
    }

    /**
     * Устанавливает цвет фонового света
     * @param {Color} color
     */
    setAmbientColor(color) {
        this.ambient = color;
    }
}

/**
 * Объект точечного света на сцене
 */
export class PointLight extends Light {
    constructor(position = vec3.create()) {
        super(position);
        this.scale = vec3.fromValues(0.1, 0.1, 0.1);
        this.loadFromObjFile("sphere", () => {});
    }
}

/**
 * Объект направленного света на сцене
 */
export class DirectionalLight extends Light {
    constructor(rotation = vec3.create()) {
        super(rotation);
        this.visible = false;
        this.scale = vec3.fromValues(0.1, 0.1, 0.1);
        this.loadFromObjFile("sphere", () => {});
    }
}
```

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*, а также освещение, представляемое объектами *POINT_LIGHT* и *DIRECTIONAL_LIGHT*.

Функция *renderScene* представлена в листинге 5.1.

Листинг 5.1. Функция renderScene.

```
/**
 * Выполняет рендеринг сцены на окно рендеринга.
 * @param {SceneObject[]} sceneObjects
 * @param {Camera} camera
 * @param {PointLight} pointLight
 * @param {DirectionalLight} directionalLight
 * @param {{}} renderParameters
 */
export function renderScene(sceneObjects, camera, pointLight,
directionalLight, renderParameters) {
    GL.clear(GL.COLOR_BUFFER_BIT);
    GL.clear(GL.DEPTH_BUFFER_BIT);

    let cameraPositionUniform = GL.getUniformLocation(PROGRAM,
"u_cameraPosition");
    GL.uniform3f(cameraPositionUniform, camera.position[0],
camera.position[1], camera.position[2]);

    let lightsUniform = GL.getUniformLocation(PROGRAM,
"u_lights[0].diffuse");
    GL.uniform3f(lightsUniform, pointLight.diffuse.r,
pointLight.diffuse.g, pointLight.diffuse.b);
    lightsUniform = GL.getUniformLocation(PROGRAM,
"u_lights[0].ambient");
    GL.uniform3f(lightsUniform, pointLight.ambient.r,
pointLight.ambient.g, pointLight.ambient.b);
    lightsUniform = GL.getUniformLocation(PROGRAM,
"u_lights[0].specular");
    GL.uniform3f(lightsUniform, pointLight.specular.r,
pointLight.specular.g, pointLight.specular.b);
    lightsUniform = GL.getUniformLocation(PROGRAM,
"u_lights[0].position");
    GL.uniform4f(lightsUniform, pointLight.position[0],
pointLight.position[1], pointLight.position[2], 1);

    lightsUniform = GL.getUniformLocation(PROGRAM,
"u_lights[1].diffuse");
    GL.uniform3f(lightsUniform, directionalLight.diffuse.r,
directionalLight.diffuse.g, directionalLight.diffuse.b);
    lightsUniform = GL.getUniformLocation(PROGRAM,
"u_lights[1].ambient");
    GL.uniform3f(lightsUniform, directionalLight.ambient.r,
directionalLight.ambient.g, directionalLight.ambient.b);
    lightsUniform = GL.getUniformLocation(PROGRAM,
```

```
"u_lights[1].specular");
    GL.uniform3f(lightsUniform,             directionalLight.specular.r,
directionalLight.specular.g, directionalLight.specular.b);
    lightsUniform           =           GL.getUniformLocation(PROGRAM,
"u_lights[1].position");
    GL.uniform4f(lightsUniform,             directionalLight.position[0],
directionalLight.position[1], directionalLight.position[2], 0);

    let viewMatrixUniform = GL.getUniformLocation(PROGRAM, "u_vMatrix");
    let vMatrix = camera.getViewMatrix();
    GL.uniformMatrix4fv(viewMatrixUniform, false, vMatrix);
    GL.enableVertexAttribArray(viewMatrixUniform);

    let    projectionMatrixUniform    =    GL.getUniformLocation(PROGRAM,
"u_pMatrix");
    let pMatrix = camera.getProjectionMatrix();
    GL.uniformMatrix4fv(projectionMatrixUniform, false, pMatrix);
    GL.enableVertexAttribArray(projectionMatrixUniform);

    let    vertexPositionAttribute    =    GL.getAttribLocation(PROGRAM,
"a_vertexPosition");
    let     vertexNormalAttribute     =     GL.getAttribLocation(PROGRAM,
"a_vertexNormal");
    let    texturePositionAttribute    =    GL.getAttribLocation(PROGRAM,
"a_texturePosition");

    let       lightingUniform       =       GL.getUniformLocation(PROGRAM,
"u_useLighting");
    let      useTextureUniform      =      GL.getUniformLocation(PROGRAM,
"u_useTexture");
    let textureUniform = GL.getUniformLocation(PROGRAM, "u_texture");
    let     textureScaleUniform     =     GL.getUniformLocation(PROGRAM,
"u_textureScale");
    let modelMatrixUniform = GL.getUniformLocation(PROGRAM, "u_mMatrix");
    let     normalMatrixUniform     =     GL.getUniformLocation(PROGRAM,
"u_nMatrix");

    sceneObjects.forEach((object) => {
        if(!object.visible) {
            return;
        }

        let mMatrix = object.getTransformMatrix();
        GL.uniformMatrix4fv(modelMatrixUniform, false, mMatrix);
        GL.enableVertexAttribArray(modelMatrixUniform);

        let nMatrix = object.getNormalMatrix();
        GL.uniformMatrix4fv(normalMatrixUniform, false, nMatrix);
        GL.enableVertexAttribArray(normalMatrixUniform);

        let     materialUniform     =     GL.getUniformLocation(PROGRAM,
"u_material.diffuse");
        GL.uniform3f(materialUniform,   object.color.r,   object.color.g,
object.color.b);
        materialUniform         =         GL.getUniformLocation(PROGRAM,
"u_material.ambient");
        GL.uniform3f(materialUniform, 1, 1, 1);
```

```
        materialUniform          =          GL.getUniformLocation(PROGRAM,
"u_material.specular");
        GL.uniform3f(materialUniform,   object.color.r,   object.color.g,
object.color.b);
        materialUniform          =          GL.getUniformLocation(PROGRAM,
"u_material.shininess");
        GL.uniform1f(materialUniform,
renderParameters["materialShininess"]);

        let useTexture = (object.texture === -1) ? 0 : 1;
        GL.uniform1i(useTextureUniform, useTexture);
        if (useTexture) {
            GL.uniform1i(textureUniform, object.texture);
            GL.uniform1f(textureScaleUniform, object.textureScale);
        }

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

        let normals = object.getNormals();
        let normalsBuffer = GL.createBuffer();
        GL.bindBuffer(GL.ARRAY_BUFFER, normalsBuffer);
        GL.bufferData(GL.ARRAY_BUFFER, normals, GL.STATIC_DRAW);
        GL.vertexAttribPointer(vertexNormalAttribute, 3, GL.FLOAT, false,
0, 0);
        GL.enableVertexAttribArray(vertexNormalAttribute);

        let textureCoordinates = object.getTextureCoordinates();
        let textureCoordinatesBuffer = GL.createBuffer();
        GL.bindBuffer(GL.ARRAY_BUFFER, textureCoordinatesBuffer);
        GL.bufferData(GL.ARRAY_BUFFER,                textureCoordinates,
GL.STATIC_DRAW);
        GL.vertexAttribPointer(texturePositionAttribute,   2,   GL.FLOAT,
false, 0, 0);
        GL.enableVertexAttribArray(texturePositionAttribute);

        if(renderParameters["drawPolygons"]) {
            let objectColor = object.color.asVector();
            GL.uniform1i(lightingUniform,   renderParameters["drawLight"]
&& !(object instanceof PointLight));
            GL.polygonOffset(0, 0);
            GL.drawArrays(GL.TRIANGLES, 0, verticesCount);
        }

        if(renderParameters["drawEdges"]) {
            let edgeColor = hexToColor(renderParameters["edgeColor"]);
            let    materialUniform    =    GL.getUniformLocation(PROGRAM,
"u_material.diffuse");
            GL.uniform3f(materialUniform,    edgeColor.r,    edgeColor.g,
edgeColor.b);
            GL.uniform1i(lightingUniform, 0);
```

```
            GL.uniform1i(useTextureUniform, 0);
            GL.polygonOffset(1, 1);
            GL.drawArrays(GL.LINE_LOOP, 0, verticesCount);
        }

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

6. Обработка ввода пользователя

Для удобного перемещения по сцене было реализовано управление перемещением и поворотом камеры через клавиатуру и мышь. Для этого был создан класс *Input*. Он отслеживает нажатия клавиш клавиатуры и мыши, а также перемещение мыши, и исходя из этого меняет положение и поворот камеры. Объект класс Input хранится в глобальной переменной *USER_INPUT*.

Класс *Input* представлен в листинге 6.1.

Листинг 6.1. Класс Input.

```
let MOVEMENT_SPEED = 5;
let ROTATION_SPEED = 0.4;

/**
 * Класс для обработки нажатий клавиш и мыши.
 */
export class Input {
    constructor() {
        this.binds = {};
        this.actions = {}
        this.mousePressed = false;
        this.mousePositionDelta = [0, 0];
        this.previousMousePosition = null;
        this.mousePositionUpdated = false;
    }

    /**
     * Инициализирует действия и слущателей событий.
     */
    initialize() {
        this.binds[87] = "forward";
        this.binds[65] = "left";
        this.binds[83] = "backward";
        this.binds[68] = "right";
        this.binds[69] = "up";
        this.binds[81] = "down";
        this.actions["forward"] = false;
        this.actions["left"] = false;
        this.actions["backward"] = false;
        this.actions["right"] = false;
        this.actions["up"] = false;
```

```
        this.actions["down"] = false;

        document.body.addEventListener("keyup",         (event)         =>
{this.onKeyUp(event);});
        document.body.addEventListener("keydown",        (event)        =>
{this.onKeyDown(event);});

        let canvas = $("#canvas");
        canvas.bind("mouseup", (event) => {this.onMouseUp(event);});
        canvas.bind("mousedown", (event) => {this.onMouseDown(event);});
        canvas.bind("mousemove", (event) => {this.onMouseMove(event);});
    }

    /**
     * Вызывается на нажатие клавиши клавиатуры.
     */
    onKeyUp(event) {
        let action = this.binds[event.keyCode];
        if (action) {
            this.actions[action] = false;
        }
    }

    /**
     * Вызывается на отпускание клавиши клавиатуры.
     */
    onKeyDown(event) {
        let action = this.binds[event.keyCode];
        if (action) {
            this.actions[action] = true;
        }
    }

    /**
     * Вызывается на нажатие клавиши мыши.
     */
    onMouseUp(_) {
        this.mousePressed = false;
    }

    /**
     * Вызывается на отпускание клавиши мыши.
     */
    onMouseDown(_) {
        this.mousePressed = true;
    }

    /**
     * Вызывается на перемещение курсора мыши.
     */
    onMouseMove(event) {
        let x = event.offsetX;
        let y = event.offsetY;

        if (this.previousMousePosition === null) {
            this.previousMousePosition = [x, y];
        }
```

```
        this.mousePositionDelta = [x - this.previousMousePosition[0], y -
this.previousMousePosition[1]];
        this.previousMousePosition = [x, y];
        this.mousePositionUpdated = true;
    }

    /**
     * Обрабатывает текущее состояние клавиатуры и мыши, и исходя из него
меняет параметры объектов.
     */
    processInput(camera, deltaSeconds) {
        let movementDelta = MOVEMENT_SPEED * deltaSeconds;
        let rotationDelta = ROTATION_SPEED * deltaSeconds;

        let deltaP = vec3.create();
        let forward = camera.getForwardVector();
        let up =  camera.getUpVector();
        let right =  camera.getRightVector();

        if(this.actions["forward"]) {
            vec3.add(deltaP,  deltaP,  vec3.scale(vec3.create(),  forward,
movementDelta));
        }
        if(this.actions["backward"]) {
            vec3.add(deltaP,       deltaP,       vec3.scale(vec3.create(),
vec3.negate(vec3.create(), forward), movementDelta));
        }
        if(this.actions["right"]) {
            vec3.add(deltaP,  deltaP,  vec3.scale(vec3.create(),  right,
movementDelta));
        }
        if(this.actions["left"]) {
            vec3.add(deltaP,       deltaP,       vec3.scale(vec3.create(),
vec3.negate(vec3.create(), right), movementDelta));
        }
        if(this.actions["up"]) {
            vec3.add(deltaP,   deltaP,   vec3.scale(vec3.create(),   up,
movementDelta));
        }
        if(this.actions["down"]) {
            vec3.add(deltaP,       deltaP,       vec3.scale(vec3.create(),
vec3.negate(vec3.create(), up), movementDelta));
        }

        camera.position   =   vec3.add(vec3.create(),   camera.position,
deltaP);

        let deltaR = vec3.create();
        if(this.mousePressed && this.mousePositionUpdated)
        {
            deltaR[1] = -this.mousePositionDelta[0] * rotationDelta;
            deltaR[2] = -this.mousePositionDelta[1] * rotationDelta;
        }
        this.mousePositionUpdated = false;

        camera.rotation   =   vec3.add(vec3.create(),   camera.rotation,
```

```
deltaR);
    }
}
```

7. 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, параметры источников освещения, параметры материала объектов 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;
- Диффузный цвет;
- Прозрачность цвета;
- Использовать текстуру;

- Масштаб текстуры;
- Текстура.

Камера помимо основных параметров объекта сцены, имеет:

- Вертикальный угол обзора;
- Тип проекции.

Имеется два типа источников освещения: точечный и направленный. Для точечного источника можно настраивать параметры положения, диффузного и фонового света (зеркальный свет равен диффузному). Для направленного источника можно настраивать параметры направления, диффузного и фонового света (зеркальный свет равен диффузному). Для материала можно менять его блеск (shininess).

Интерфейс программы представлен на рисунке 1.

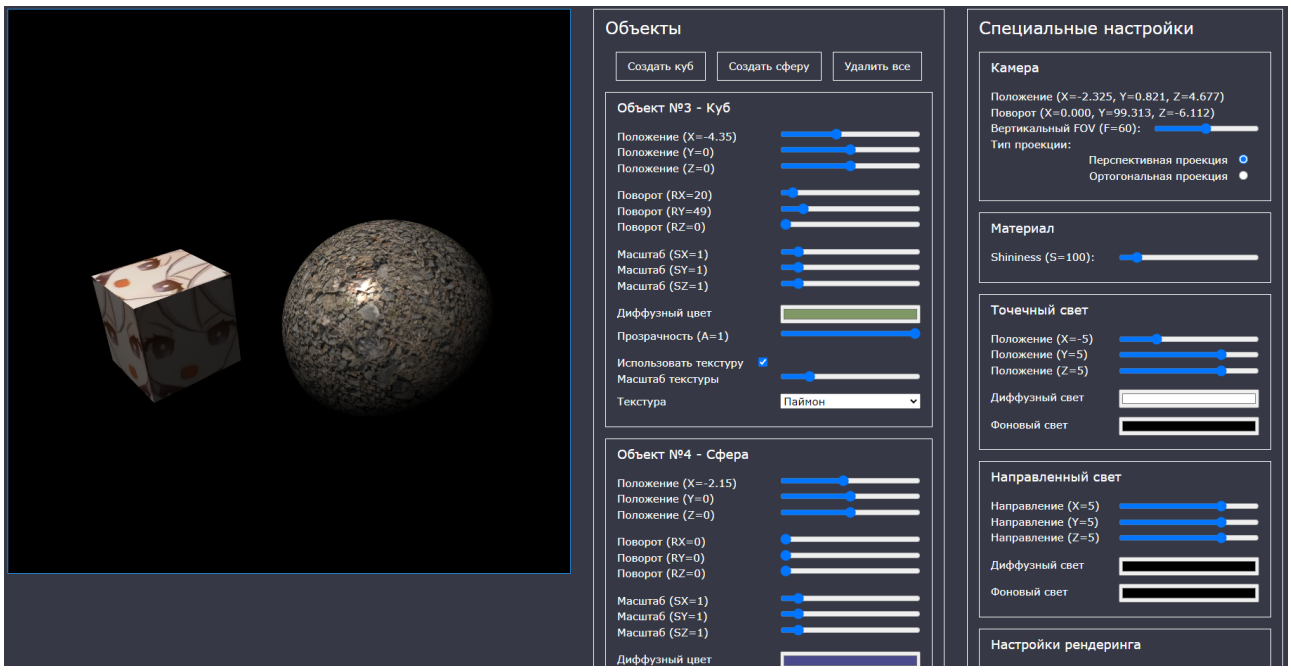

Рисунок 1 – Интерфейс программы.

Настройки рендеринга имеют следующие параметры:

- Использование освещения;
- 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*, объектам *CAMERA_OBJECT*, *POINT_LIGHT*, *DIRECTIONAL_LIGHT* и *RENDER_PARAMETERS*, после чего вызывается функция *renderScene*. Функция *onParametersChanged* представлена в листинге 7.1.

Листинг 7.1. Функция onParametersChanged.

```
/**
 * Обрабатывает изменения в input-элементах от пользователя и применяет изменения к сцене.
 */
export function onParametersChanged() {
    // Параметры рендеринга
    RENDER_PARAMETERS["drawLight"]    =    document.getElementById(`draw-light`).checked;
    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;

    // Координаты камеры
    $(`#output-x-camera`).text(CAMERA_OBJECT.position[0].toFixed(3));
    $(`#output-y-camera`).text(CAMERA_OBJECT.position[1].toFixed(3));
    $(`#output-z-camera`).text(CAMERA_OBJECT.position[2].toFixed(3));

    // Вращение камеры
    $(`#output-x-rotation-camera`).text(toDegree(CAMERA_OBJECT.rotation[0]).toFixed(3));
    $(`#output-y-rotation-camera`).text(toDegree(CAMERA_OBJECT.rotation[1]).toFixed(3));
    $(`#output-z-rotation-camera`).text(toDegree(CAMERA_OBJECT.rotation[2]).toFixed(3));

    // 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";

    // Материал
    let e_m_s = document.getElementById(`input-shininess`);
    RENDER_PARAMETERS["materialShininess"] = Number(e_m_s.value);
    $(`#output-shininess`).text(e_m_s.value);

    // Позиция точечного света
    let e_pl_x = document.getElementById(`input-x-point-light`);
    let e_pl_y = document.getElementById(`input-y-point-light`);
    let e_pl_z = document.getElementById(`input-z-point-light`);
    POINT_LIGHT.position[0] = Number(e_pl_x.value);
    POINT_LIGHT.position[1] = Number(e_pl_y.value);
    POINT_LIGHT.position[2] = Number(e_pl_z.value);
    $(`#output-x-point-light`).text(e_pl_x.value);
    $(`#output-y-point-light`).text(e_pl_y.value);
    $(`#output-z-point-light`).text(e_pl_z.value);

    // Цвет точечного света
    let e_pl_d = document.getElementById(`input-diffuse-point-light`);
    let c = hexToColor(e_pl_d.value);
    POINT_LIGHT.setColor(c);
    POINT_LIGHT.color = c;
    $(`#output-diffuse-point-light`).text(e_pl_d.value);

    // Фоновый цвет точечного света
    let e_pl_a = document.getElementById(`input-ambient-point-light`);
    POINT_LIGHT.setAmbientColor(hexToColor(e_pl_a.value));
    $(`#output-ambient-point-light`).text(e_pl_a.value);

    // Поворот направленного света
    let e_dl_x = document.getElementById(`input-x-directional-light`);
    let e_dl_y = document.getElementById(`input-y-directional-light`);
    let e_dl_z = document.getElementById(`input-z-directional-light`);
    DIRECTIONAL_LIGHT.position[0] = Number(e_dl_x.value);
    DIRECTIONAL_LIGHT.position[1] = Number(e_dl_y.value);
    DIRECTIONAL_LIGHT.position[2] = Number(e_dl_z.value);
    $(`#output-x-directional-light`).text(e_dl_x.value);
    $(`#output-y-directional-light`).text(e_dl_y.value);
    $(`#output-z-directional-light`).text(e_dl_z.value);

    // Цвет направленного света
    let   e_dl_d   =   document.getElementById(`input-diffuse-directional-
light`);
    c = hexToColor(e_dl_d.value);
    DIRECTIONAL_LIGHT.setColor(c);
    DIRECTIONAL_LIGHT.color = c;
    $(`#output-diffuse-directional-light`).text(e_dl_d.value);

    // Фоновый цвет направленного света
    let   e_dl_a   =   document.getElementById(`input-ambient-directional-
```

```
light`);
    DIRECTIONAL_LIGHT.setAmbientColor(hexToColor(e_dl_a.value));
    $(`#output-ambient-directional-light`).text(e_dl_a.value);

    // Объекты
    for (let i = 0; i < SCENE_OBJECTS.length; ++i) {
        if(SCENE_OBJECTS[i]   instanceof   Camera   ||   SCENE_OBJECTS[i]
instanceof Light)
        {
            continue;
        }

        // 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);

        // Текстура
        let e_ut = document.getElementById(`use-texture-${i+1}`);
        let e_t = document.getElementById(`texture-${i+1}`);
        let e_uv = document.getElementById(`uv-scale-${i+1}`);
        SCENE_OBJECTS[i].texture = (e_ut.checked) ? Number(e_t.value) : -
```

```
1;
        SCENE_OBJECTS[i].textureScale = Number(e_uv.value);
    }

    // Цвет заднего фона
    let                          backgroundColor                          =
hexToColor(RENDER_PARAMETERS["backgroundColor"]);
    setClearColor(backgroundColor);
}
```

Для добавления и удаления примитивов сцены используются функции *createSceneObject* и *clearSceneObjects* соответственно. Функции представлены в листинге 7.2.

Листинг 7.2. Функции createSceneObject и clearSceneObjects.

```
/**
 * Добавляет новый объект на сцену.
 * @param {string} type
 */
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 class="w3-margin-bottom">
                <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-${id}"    class="w3-right"
type="range" min="0" max="360" step="1" value="0" style="width: 200px;"
oninput="onParametersChanged()"/>
            </div>
            <br/>
            <div class="w3-margin-bottom">
                <label      for="input-rotation-z-${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 class="w3-margin-bottom">
                <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 style="margin-bottom: 10px;">
                <label for="input-color-${id}" class="w3-left">Диффузный
цвет</label>
                <input id="input-color-${id}" class="w3-right"
type="color" value="${colorToHex(Math.round(Math.random() * 255),
Math.round(Math.random() * 255), Math.round(Math.random() * 255))}"
style="width: 200px;" oninput="onParametersChanged()"/>
            </div>
            <br/>
            <div class="w3-margin-bottom">
                <label for="input-alpha-${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>
                <label for="use-texture-${id}">Использовать
текстуру   </label>
                <input id="use-texture-${id}" type="checkbox"
oninput="onParametersChanged()"/>
            </div>
            <div style="margin-bottom: 10px;">
                <label for="uv-scale-${id}" class="w3-left">Масштаб
текстуры</label>
                <input id="uv-scale-${id}" class="w3-right" type="range"
min="0.1" max="5" step="0.1" value="1" style="width: 200px;"
oninput="onParametersChanged()"/>
            </div>
            <br/>
            <div class="w3-margin-bottom">
                <label for="texture-${id}" class="w3-
left">Текстура</label>
                <select id="texture-${id}" class="w3-right" type="text"
style="width: 200px;" oninput="onParametersChanged()">
                    <option value="0">Кирпич</option>
                    <option value="1">Дерево</option>
                    <option value="2">Гравий</option>
                    <option value="3">Паймон</option>
                </select>
            </div>
            <br/>
        </div>
    </div>
    `;

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

    let object = new SceneObject();
    object.loadFromObjFile(type, () => {
        SCENE_OBJECTS.push(object);
    });
}
```

```
/**
 * Удаляет все объекты со сцены.
 */
export function clearSceneObjects() {
    SCENE_OBJECTS = [POINT_LIGHT, DIRECTIONAL_LIGHT];
    $("#objects").html("");
}
```

Прочие основные функции, а именно onPageLoad (срабатывает при загрузке станицы), exec (основной цикл приложения) и changeAccordionVisibility (срабатывает при нажатии на блок настроек объекта), представлены в листинге 7.3.

Листинг 7.3. Функции createSceneObject и clearSceneObjects.

```
/**
 * Подготавливает приложение к работе. Выполняется в момент полной заггрузки страницы.
 */
export function onPageLoad() {
    initializeWebGl();
    USER_INPUT.initialize();
    setInterval(exec, 1000 / FPS);
}

/**
 * Главный цикл обработки. Обрабатывает события с клавиатуры и мыши, получает значения из input-ов, после чего рендерит сцену.
 */
function exec() {
    USER_INPUT.processInput(CAMERA_OBJECT, 1 / FPS);
    onParametersChanged();
    renderScene(SCENE_OBJECTS, CAMERA_OBJECT, POINT_LIGHT, DIRECTIONAL_LIGHT, RENDER_PARAMETERS);
}

/**
 * Меняет видимость контейнера с настройками параметров объекта.
 * @param {string} id
 */
export function changeAccordionVisibility(id) {
    let e = document.getElementById(id);
    if (e.className.indexOf("w3-show") === -1) {
        e.className += " w3-show";
    } else {
        e.className = e.className.replace(" w3-show", "");
    }
}
```

Выводы.

В результате выполнения лабораторной работы была разработана программа на языке JavaScript, которая осуществляет рисование 3D-объектов с текстурами и освещением на глобальной сцене относительно объекта наблюдения – камеры.

Было изучено текстурирование 3D-объектов при помощи языка шейдеров GLSL ES 3.0 (WebGL 2.0). Рассмотрены создание и использование текстурных объектов, использование текстурных единиц, создание и использование семплер-переменных, а также загрузка и работа с изображениями. На основе полученных знаний было реализовано текстурирование 3D-объектов. Для каждой текстуры используется текстурный объект и соответствующая ему отдельная текстурная единица.