**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №3**

**по дисциплине «Компьютерная графика»**

Тема: 3D освещение

|  |  |  |
| --- | --- | --- |
| Студент гр. 9303 |  | Колованов Р.А. |
| Преподаватель |  | Герасимова Т.В. |

Санкт-Петербург

2023

**Цель работы.**

Целью работы является освоение трехмерного освещения.

**Задание.**

* Определить трехмерный объект на основе простой геометрической фигуры;
* Добавить нормали к трехмерному объекту, чтобы сделать его подходящим для освещения;
* Добавить позиционный и направленный источник света;
* Попробовать сформировать затенение по Гуро и Фонгу, и сравнить их;
* Узнать больше о взаимодействии материалов и свойств освещения и примените это на вашей сцене.

**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).

1. **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.**  /\*\*  \* 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.

|  |
| --- |
| **Листинг 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 3.

|  |
| --- |
| **Листинг 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 4.

|  |
| --- |
| **Листинг 4. Функция createProgram.**  /\*\*  \* 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 *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.**  /\*\*  \* 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 *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.**  /\*\*  \* 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 *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.**  /\*\*  \* Инициализирует WebGL.  \*/  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);  } |

Функция *setClearColor* устанавливает цвет очистки окна рендеринга. На вход принимает цвет. Функция *setClearColor* представлена в листинге 8.

|  |
| --- |
| **Листинг 8. Функция setClearColor.**  /\*\*  \* Устанавливает цвет очистки окна рендеринга.  \* @param {Color} color  \*/  export function setClearColor(color) {  GL.clearColor(color.r, color.g, color.b, color.a);  } |

1. **Шейдеры**

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

* 1. **Вершинный шейдер**

Принимает координаты вершин, их нормали и цвет при помощи атрибутов a\_*vertexPosition*, a\_*vertexNormal* 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*.

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

Для определения освещения на сцене используется униформа *u\_lighting* (определяет использования 3D-освещения), униформа-массив *u\_lights* (содержит массив источников света) и униформа *u\_material* (определяет параметры материала объектов). 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 каждой вершины в вершинном шейдере – используется затенение по Гуро.

Шейдер представлен в листинге 9.

|  |
| --- |
| **Листинг 9. Вершинный шейдер.**  export default  `  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;  attribute vec3 a\_vertexPosition;  attribute vec3 a\_vertexNormal;  uniform vec3 u\_cameraPosition;  uniform vec4 u\_vertexColor;  uniform mat4 u\_mMatrix;  uniform mat4 u\_vMatrix;  uniform mat4 u\_pMatrix;  uniform mat4 u\_nMatrix;  uniform bool u\_lighting;  uniform Light u\_lights[LIGHT\_NUMBER];  uniform Material u\_material;  varying vec4 v\_vertexColor;  vec4 globalPosition;  vec3 globalNormal;  vec4 calculateLight();  void main() {  globalPosition = u\_mMatrix \* vec4(a\_vertexPosition, 1.0);    v\_vertexColor = u\_vertexColor;  if (u\_lighting) {  globalNormal = normalize((u\_nMatrix \* vec4(a\_vertexNormal, 0.0)).xyz);  v\_vertexColor.xyz \*= calculateLight().xyz;  }    gl\_Position = u\_pMatrix \* u\_vMatrix \* globalPosition;  }  vec4 calculateLight() {  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 - globalPosition.xyz);  }  float Kd = max(dot(lightDirection, globalNormal), 0.0);  float Ks = 0.0;  if (Kd > 0.0) {  vec3 eyeDirection = normalize(u\_cameraPosition - globalPosition.xyz);  vec3 halfVector = normalize(eyeDirection + lightDirection);  Ks = pow(max(dot(halfVector, globalNormal), 0.0), u\_material.shininess);  }    resultColor.xyz += Ks \* u\_material.specular \* light.specular  + Kd \* u\_material.diffuse \* light.diffuse  + u\_material.ambient \* light.ambient;  }    resultColor.a = 1.0;    return resultColor;  }  `; |

* 1. **Фрагментный шейдер**

Принимает цвета вершин при помощи varying-переменной *v\_vertexColor*. Цвет устанавливается в *gl\_FragColor*. Шейдер представлен в листинге 10.

|  |
| --- |
| **Листинг 10. Фрагментный шейдер.**  export default  `  precision mediump float;  varying vec4 v\_vertexColor;  void main() {  gl\_FragColor = v\_vertexColor;  }  `; |

1. **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, 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.

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 11.

|  |
| --- |
| **Листинг 11. Класс SceneObject.**  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.position = position;  this.rotation = rotation;  this.scale = scale;  this.color = color;  this.vertices = new Float32Array(0);  this.normals = new Float32Array(0);  this.visible = true;  }  /\*\*  \* 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);  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];  }  callback();  });  }  /\*\*  \* Возвразщает 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);  }  /\*\*  \* Возвразщает 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 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 12.

|  |
| --- |
| **Листинг 12. 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 = 90;  this.view\_distance = 1000;  }  /\*\*  \* Возвразщает 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());  }  /\*\*  \* Возвразщает 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, 0.1, 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();  }  }  } |

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 13.

|  |
| --- |
| **Листинг 13. 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", () => {});  }  } |

1. **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 14.

|  |
| --- |
| **Листинг 14. 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 lightingUniform = GL.getUniformLocation(PROGRAM, "u\_lighting");  let vertexPositionAttribute = GL.getAttribLocation(PROGRAM, "a\_vertexPosition");  let vertexNormalAttribute = GL.getAttribLocation(PROGRAM, "a\_vertexNormal");  let vertexColorUniform = GL.getUniformLocation(PROGRAM, "u\_vertexColor");  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 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);  if(renderParameters["drawPolygons"]) {  let objectColor = object.color.asVector();  GL.uniform1i(lightingUniform, renderParameters["drawLight"] && !(object instanceof PointLight));  GL.uniform4fv(vertexColorUniform, objectColor);  GL.polygonOffset(0, 0);  GL.drawArrays(GL.TRIANGLES, 0, verticesCount);  }  if(renderParameters["drawEdges"]) {  let edgeColor = hexToColor(renderParameters["edgeColor"]).asVector();  GL.uniform1i(lightingUniform, 0);  GL.uniform4fv(vertexColorUniform, edgeColor);  GL.polygonOffset(1, 1);  GL.drawArrays(GL.LINE\_LOOP, 0, verticesCount);  }  GL.deleteBuffer(verticesBuffer);  });  } |

1. **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 15.

|  |
| --- |
| **Листинг 15. 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);  }  } |

1. **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;
* Цвет примитива;
* Прозрачность цвета.

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).

Интерфейс программы представлен на рисунке 1.

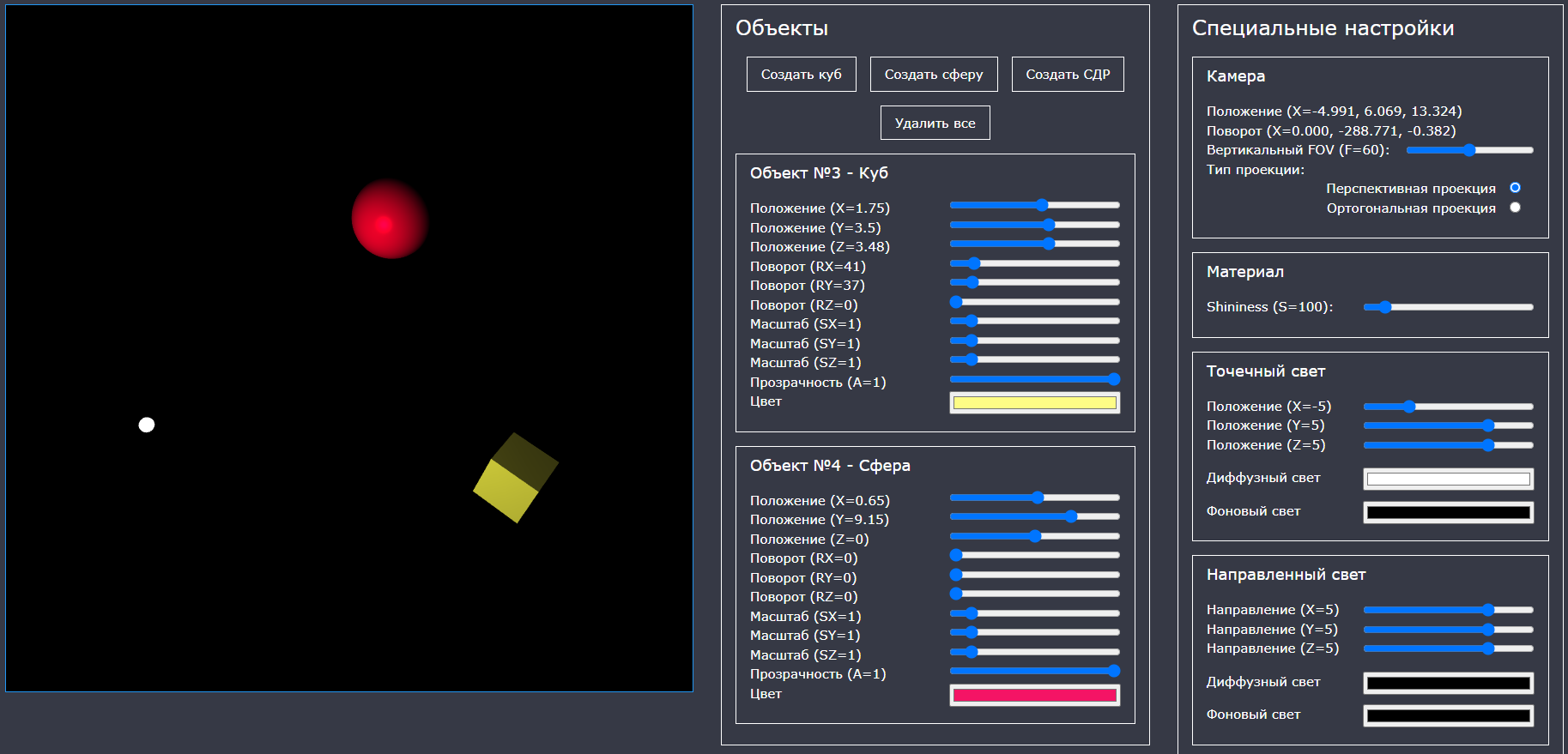


Рисунок 1 – Интерфейс программы.

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 16.

|  |
| --- |
| **Листинг 16. 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);  }  // 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 17.

|  |
| --- |
| **Листинг 17. 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>  <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 = 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("");  }  /\*\*  \* 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 освещением на глобальной сцене относительно объекта наблюдения – камеры.

Была изучена модель освещения, состоящая из трех компонент: диффузный свет, зеркальный свет и фоновый свет. Было изучено предназначение нормалей поверхностей модели для расчёта освещения. Также были изучены затенения по Гуро и Фонга. На основе полученных знаний было реализовано трехмерное освещение, базирующееся на трехкомпонентной модели освещения и затенении Гуро.

При сравнении затенения по Гуро и Фонгу обнаружилось, что затенение по Фонгу выглядит более реалистично и не имеет проблем с смазанными бликами. Стоит отметить, что проблема со смазанными бликами при затенении Гуро наблюдалась только на мало детализированных моделях (с малым количеством треугольников).