**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №2**

**по дисциплине «Компьютерная графика»**

Тема: 3D трансформации

|  |  |  |
| --- | --- | --- |
| Студент гр. 9381 |  | Колованов Р.А. |
| Преподаватель |  | Герасимова Т.В. |

Санкт-Петербург

2023

**Цель работы.**

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

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

**Задание.**

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

**Выполнение работы.**

1. **Используемые технологии**

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

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

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

|  |
| --- |
| **Листинг 2. Функция getWebGlContext.**  function getWebGlContext() {  let canvas = getCanvas();  if (!canvas) {  return null;  }  let context = canvas.getContext("webgl");  if (!context) {  console.error("Не удалось получить контекст WebGL.");  return null;  }  return context;  } |

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

|  |
| --- |
| **Листинг 3. Функция createShader.**  function createShader(gl, type, source) {  let shader = gl.createShader(type);  if (!shader) {  console.error("Не удалось создать шейдер с типом '" + type + "'");  return null;  }  gl.shaderSource(shader, source);  gl.compileShader(shader);  let compiled = gl.getShaderParameter(shader, gl.COMPILE\_STATUS);  if (!compiled) {  let error = gl.getShaderInfoLog(shader);  console.error("Ошибка компиляции шейдера: " + error);  gl.deleteShader(shader);  return null;  }  return shader;  } |

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

|  |
| --- |
| **Листинг 4. Функция createProgram.**  function createProgram(gl, shaders) {  let program = gl.createProgram();  if (!program) {  console.error("Не удалось создать шейдерную программу");  return null;  }  shaders.forEach((shader) => {  gl.attachShader(program, shader);  });  gl.linkProgram(program);  let linked = gl.getProgramParameter(program, gl.LINK\_STATUS);  if (!linked) {  let error = gl.getProgramInfoLog(program);  console.error("Ошибка линковки программы: " + error);  gl.deleteProgram(program);  return null;  }  return program;  } |

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

|  |
| --- |
| **Листинг 5. Функция initializeShaderProgram.**  function initializeShaderProgram(gl) {  let vertexShader = createShader(gl, gl.VERTEX\_SHADER, VERTEX\_SHADER\_SOURCE);  let fragmentShader = createShader(gl, gl.FRAGMENT\_SHADER, FRAGMENT\_SHADER\_SOURCE);  if (PROGRAM !== null) {  gl.deleteProgram(PROGRAM);  }  PROGRAM = createProgram(gl, [vertexShader, fragmentShader]);  gl.useProgram(PROGRAM);  } |

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

|  |
| --- |
| **Листинг 6. Функция initializeViewport.**  function initializeViewport(gl) {  let canvas = getCanvas();  if (canvas === null) {  return;  }  gl.viewport(0, 0, canvas.width, canvas.height);  } |

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

|  |
| --- |
| **Листинг 7. Функция initializeWebGl.**  export function initializeWebGl() {  GL = getWebGlContext();  initializeShaderProgram(GL);  initializeViewport(GL);  // Дополнительные функции  GL.enable(GL.DEPTH\_TEST);  GL.enable(GL.POLYGON\_OFFSET\_FILL);  GL.enable(GL.SAMPLE\_ALPHA\_TO\_COVERAGE);  } |

1. **Шейдеры**

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

* 1. **Вершинный шейдер**

Принимает координаты вершин и их цвет при помощи атрибута a\_*vertexPosition* и униформы u\_*vertexColor* соответственно. Помимо этого, принимает униформы матриц модели *u\_mMatrix*, вида *u\_vMatrix* и перспективы *u\_pMatrix*, с помощью которых осуществляется перевод локальных координат вершин объекта сначала к мировым координатам (с использованием матрицы модели), после чего к видовым координатам (с использованием матрицы вида), и в завершение к проекционным координатам (с использованием матрицы проекции). Преобразованные координаты устанавливаются в *gl\_Position*, а цвет передается фрагментному шейдеру через varying-переменную *v\_vertexColor*. Шейдер представлен в листинге 8.

|  |
| --- |
| **Листинг 8. Вершинный шейдер.**  export default  `  attribute vec3 a\_vertexPosition;  uniform vec4 u\_vertexColor;  uniform mat4 u\_mMatrix;  uniform mat4 u\_vMatrix;  uniform mat4 u\_pMatrix;  varying vec4 v\_vertexColor;  void main() {  gl\_Position = u\_pMatrix \* u\_vMatrix \* u\_mMatrix \* vec4(a\_vertexPosition, 1);  v\_vertexColor = u\_vertexColor;  }  `; |

* 1. **Фрагментный шейдер**

Принимает цвета вершин при помощи varying-переменной *v\_vertexColor*. Цвет устанавливается в *gl\_FragColor*. Шейдер представлен в листинге 9.

|  |
| --- |
| **Листинг 9. Фрагментный шейдер.**  export default  `  precision mediump float;  varying vec4 v\_vertexColor;  void main() {  gl\_FragColor = v\_vertexColor;  }  `; |

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 трех методов: *getVertices*, *getTransformMatrix* и *loadFromObjFile*.

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

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

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

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

Класс *SceneObject* представлен в листинге 10.

|  |
| --- |
| **Листинг 10. Класс SceneObject.**  export class SceneObject {  constructor(position = vec3.create(), rotation = vec3.create(), scale = vec3.fromValues(1, 1, 1), color = new Color(1, 1, 1, 1)) {  this.position = position;  this.rotation = rotation;  this.scale = scale;  this.color = color;  this.vertices = new Float32Array(0);  }  loadFromObjFile(name, callback) {  K3D.load(`/public/models/${name}.obj`, (rawdata) => {  let result = K3D.parse.fromOBJ(rawdata);  this.vertices = new Float32Array(3 \* result.i\_verts.length);  for(let i = 0; i < result.i\_verts.length; ++i)  {  this.vertices[3 \* i] = result.c\_verts[3 \* result.i\_verts[i]];  this.vertices[3 \* i + 1] = result.c\_verts[3 \* result.i\_verts[i] + 1];  this.vertices[3 \* i + 2] = result.c\_verts[3 \* result.i\_verts[i] + 2];  }  callback();  });  }  getTransformMatrix() {  let rotationQuad = quat.create();  quat.rotateX(rotationQuad, rotationQuad, this.rotation[0]);  quat.rotateY(rotationQuad, rotationQuad, this.rotation[1]);  quat.rotateZ(rotationQuad, rotationQuad, this.rotation[2]);  return mat4.fromRotationTranslationScale(mat4.create(), rotationQuad, this.position, this.scale);  }  getVertices() {  return new Float32Array(this.vertices);  }  } |

Было разработано две 3D-модели:

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

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

* getForwardViewTarget – для получения позиции в мировой системе координат, куда смотрит камера на данный момент;
* getUpVector – возвращает вектор направления, указывающий вверх относительно камеры;
* getViewMatrix – возвращает матрицу вида, используя текущую позицию камеры, а также два предыдущих метода.
* getProjectionMatrix – возвращает матрицу проекции (может быть перспективной или ортогональной), используя поля *view\_projection* (хранит тип проекции камеры), *view\_vfog* (вертикальный угол обзора камеры), *view\_distance* (дальность видимости камеры).

Класс *Camera* представлен в листинге 11.

|  |
| --- |
| **Листинг 11. Класс PlaneObject.**  export class Camera extends SceneObject  {  constructor(position = vec3.create(), rotation = vec3.create(), scale = vec3.fromValues(1, 1, 1), color = new Color(1, 1, 1, 1)) {  super(position, rotation, scale, color);  this.view\_projection = "perspective"  this.view\_vfov = 90;  this.view\_distance = 1000;  }  getForwardViewTarget() {  let target = vec3.copy(vec3.create(), [1, 0, 0]);  vec3.rotateX(target, target, [0, 0, 0], this.rotation[0]);  vec3.rotateY(target, target, [0, 0, 0], this.rotation[1]);  vec3.rotateZ(target, target, [0, 0, 0], this.rotation[2]);  vec3.normalize(target, target);  vec3.scale(target, target, this.view\_distance);  return vec3.add(vec3.create(), this.position, target);  }  getUpVector() {  let upVector = vec3.copy(vec3.create(), [0, 1, 0]);  vec3.rotateX(upVector, upVector, [0, 0, 0], this.rotation[0])  vec3.rotateY(upVector, upVector, [0, 0, 0], this.rotation[1])  vec3.rotateZ(upVector, upVector, [0, 0, 0], this.rotation[2])  vec3.normalize(upVector, upVector);  return upVector;  }  getViewMatrix() {  return mat4.lookAt(mat4.create(), this.position, this.getForwardViewTarget(), this.getUpVector());  }  getProjectionMatrix() {  if(this.view\_projection === "perspective") {  let aspectRatio = getCanvas().width / getCanvas().height;  return mat4.perspective(mat4.create(), this.view\_vfov, aspectRatio, 0.1, this.view\_distance);  } else if(this.view\_projection === "orthogonal") {  let boxSize = 5;  return mat4.ortho(mat4.create(), -boxSize, boxSize, -boxSize, boxSize, 0.1, this.view\_distance);  } else {  return mat4.create();  }  }  } |

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*. Функция *renderScene* представлена в листинге 12.

|  |
| --- |
| **Листинг 12. Функция renderScene.**  export function renderScene(sceneObjects, camera, renderParameters) {  let backgroundColor = hexToColor(renderParameters["backgroundColor"]);  GL.clearColor(backgroundColor.r, backgroundColor.g, backgroundColor.b, backgroundColor.a);  GL.clear(GL.COLOR\_BUFFER\_BIT);  GL.clear(GL.DEPTH\_BUFFER\_BIT);  let vertexPositionAttribute = GL.getAttribLocation(PROGRAM, "a\_vertexPosition");  let vertexColorUniform = GL.getUniformLocation(PROGRAM, "u\_vertexColor");  let modelMatrixUniform = GL.getUniformLocation(PROGRAM, "u\_mMatrix");  let viewMatrixUniform = GL.getUniformLocation(PROGRAM, "u\_vMatrix");  let projectionMatrixUniform = GL.getUniformLocation(PROGRAM, "u\_pMatrix");  let vMatrix = camera.getViewMatrix();  GL.uniformMatrix4fv(viewMatrixUniform, false, vMatrix);  GL.enableVertexAttribArray(viewMatrixUniform);  let pMatrix = camera.getProjectionMatrix();  GL.uniformMatrix4fv(projectionMatrixUniform, false, pMatrix);  GL.enableVertexAttribArray(projectionMatrixUniform);  sceneObjects.forEach((object) => {  let mMatrix = object.getTransformMatrix();  GL.uniformMatrix4fv(modelMatrixUniform, false, mMatrix);  GL.enableVertexAttribArray(modelMatrixUniform);  let vertices = object.getVertices();  let verticesCount = vertices.length / 3;  let vertexBuffer = GL.createBuffer();  GL.bindBuffer(GL.ARRAY\_BUFFER, vertexBuffer);  GL.bufferData(GL.ARRAY\_BUFFER, vertices, GL.STATIC\_DRAW);  GL.vertexAttribPointer(vertexPositionAttribute, 3, GL.FLOAT, false, 0, 0);  GL.enableVertexAttribArray(vertexPositionAttribute);  if(renderParameters["drawPolygons"]) {  let objectColor = object.color.asVector();  GL.uniform4fv(vertexColorUniform, objectColor);  GL.polygonOffset(0, 0);  GL.drawArrays(GL.TRIANGLES, 0, verticesCount);  }  if(renderParameters["drawEdges"]) {  let edgeColor = hexToColor(renderParameters["edgeColor"]);  GL.uniform4fv(vertexColorUniform, edgeColor.asVector());  GL.polygonOffset(1, 1);  GL.drawArrays(GL.LINE\_LOOP, 0, verticesCount);  }  GL.deleteBuffer(vertexBuffer);  });  } |

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 и параметры рендеринга.

Объекты сцены имеют следующие параметры:

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

Камера помимо основных параметров объекта сцены, имеет:

* Вертикальный угол обзора;
* Тип проекции.

Интерфейс программы представлен на рисунке 1.

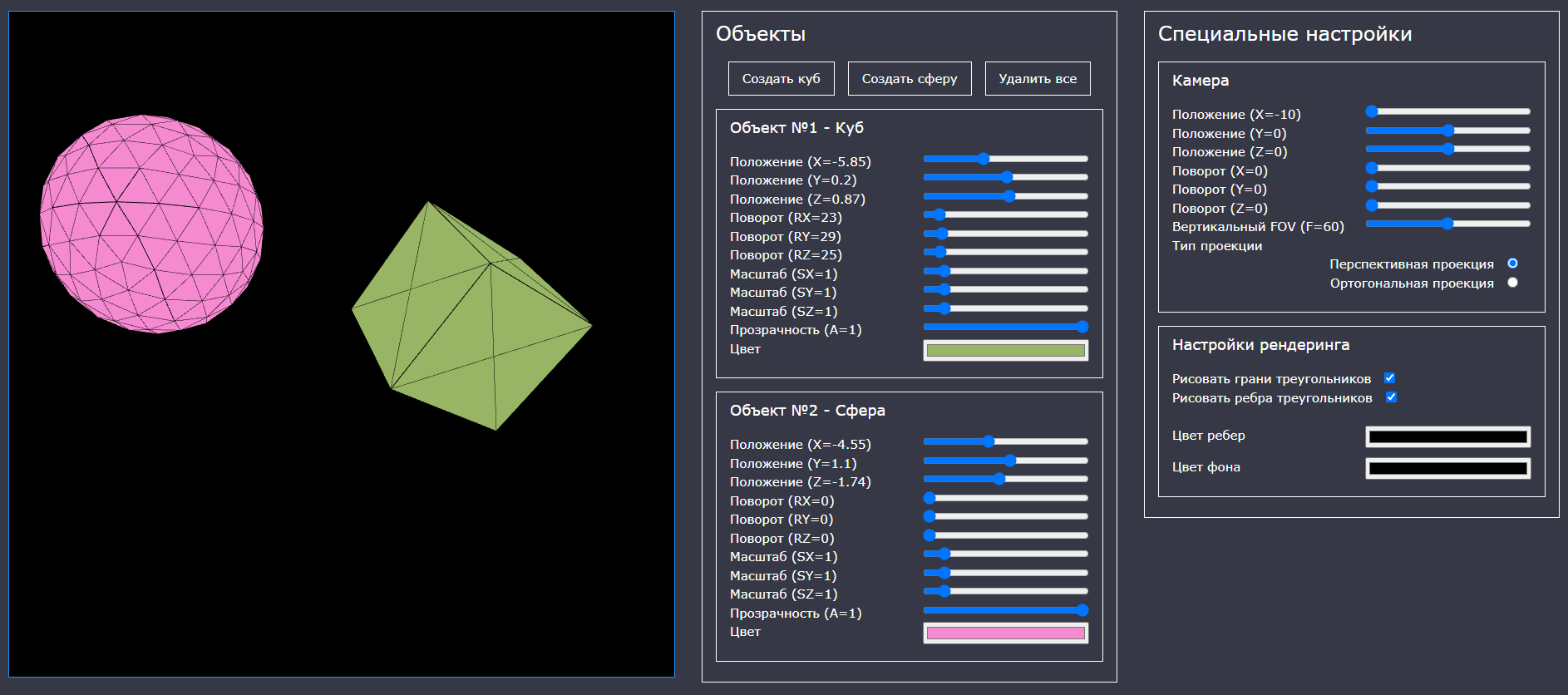


Рисунок 1 – Интерфейс программы.

Настройки рендеринга имеют следующие параметры:

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

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

|  |
| --- |
| **Листинг 14. Функция onParametersChanged.**  export function onParametersChanged() {  // Координаты камеры  let e\_x = document.getElementById(`input-x-camera`);  let e\_y = document.getElementById(`input-y-camera`);  let e\_z = document.getElementById(`input-z-camera`);  CAMERA\_OBJECT.position[0] = Number(e\_x.value);  CAMERA\_OBJECT.position[1] = Number(e\_y.value);  CAMERA\_OBJECT.position[2] = Number(e\_z.value);  $(`#output-x-camera`).text(e\_x.value);  $(`#output-y-camera`).text(e\_y.value);  $(`#output-z-camera`).text(e\_z.value);  // Вращение камеры  let e\_r\_x = document.getElementById(`input-x-rotation-camera`);  let e\_r\_y = document.getElementById(`input-y-rotation-camera`);  let e\_r\_z = document.getElementById(`input-z-rotation-camera`);  CAMERA\_OBJECT.rotation[0] = toRadian(Number(e\_r\_x.value));  CAMERA\_OBJECT.rotation[1] = toRadian(Number(e\_r\_y.value));  CAMERA\_OBJECT.rotation[2] = toRadian(Number(e\_r\_z.value));  $(`#output-x-rotation-camera`).text(e\_r\_x.value);  $(`#output-y-rotation-camera`).text(e\_r\_y.value);  $(`#output-z-rotation-camera`).text(e\_r\_z.value);  // Вертикальный FOV  let fov = document.getElementById(`input-fov-camera`);  CAMERA\_OBJECT.view\_vfov = toRadian(Number(fov.value));  $(`#output-fov-camera`).text(fov.value);  // Тип проекции  let projection = document.getElementById(`pers-projection`);  CAMERA\_OBJECT.view\_projection = (projection.checked) ? "perspective" : "orthogonal";  // Объекты  for (let i = 0; i < SCENE\_OBJECTS.length; ++i) {  // Позиция  let e\_x = document.getElementById(`input-x-${i+1}`);  let e\_y = document.getElementById(`input-y-${i+1}`);  let e\_z = document.getElementById(`input-z-${i+1}`);  SCENE\_OBJECTS[i].position[0] = Number(e\_x.value);  SCENE\_OBJECTS[i].position[1] = Number(e\_y.value);  SCENE\_OBJECTS[i].position[2] = Number(e\_z.value);  $(`#output-x-${i+1}`).text(e\_x.value);  $(`#output-y-${i+1}`).text(e\_y.value);  $(`#output-z-${i+1}`).text(e\_z.value);  // Вращение  let e\_r\_x = document.getElementById(`input-rotation-x-${i+1}`);  let e\_r\_y = document.getElementById(`input-rotation-y-${i+1}`);  let e\_r\_z = document.getElementById(`input-rotation-z-${i+1}`);  SCENE\_OBJECTS[i].rotation[0] = toRadian(Number(e\_r\_x.value));  SCENE\_OBJECTS[i].rotation[1] = toRadian(Number(e\_r\_y.value));  SCENE\_OBJECTS[i].rotation[2] = toRadian(Number(e\_r\_z.value));  $(`#output-rotation-x-${i+1}`).text(e\_r\_x.value);  $(`#output-rotation-y-${i+1}`).text(e\_r\_y.value);  $(`#output-rotation-z-${i+1}`).text(e\_r\_z.value);  // Масштаб  let e\_s\_x = document.getElementById(`input-scale-x-${i+1}`);  let e\_s\_y = document.getElementById(`input-scale-y-${i+1}`);  let e\_s\_z = document.getElementById(`input-scale-z-${i+1}`);  SCENE\_OBJECTS[i].scale[0] = Number(e\_s\_x.value);  SCENE\_OBJECTS[i].scale[1] = Number(e\_s\_y.value);  SCENE\_OBJECTS[i].scale[2] = Number(e\_s\_z.value);  $(`#output-scale-x-${i+1}`).text(e\_s\_x.value);  $(`#output-scale-y-${i+1}`).text(e\_s\_y.value);  $(`#output-scale-z-${i+1}`).text(e\_s\_z.value);  // Цвет  let e\_c = document.getElementById(`input-color-${i+1}`);  let e\_a = document.getElementById(`input-alpha-${i+1}`);  SCENE\_OBJECTS[i].color = hexToColor(e\_c.value);  SCENE\_OBJECTS[i].color.a = Number(e\_a.value);  $(`#output-alpha-${i+1}`).text(e\_a.value);  }  RENDER\_PARAMETERS["drawEdges"] = document.getElementById(`draw-edges`).checked;  RENDER\_PARAMETERS["drawPolygons"] = document.getElementById(`draw-polygons`).checked;  RENDER\_PARAMETERS["backgroundColor"] = document.getElementById(`input-background-color`).value;  RENDER\_PARAMETERS["edgeColor"] = document.getElementById(`input-edge-color`).value;  renderScene(SCENE\_OBJECTS, CAMERA\_OBJECT, RENDER\_PARAMETERS);  } |

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

|  |
| --- |
| **Листинг 14. Функции createSceneObject и clearSceneObjects.**  export function createSceneObject(type) {  let id = SCENE\_OBJECTS.length + 1;  let typeName = null;  switch (type) {  case "cube": typeName = "Куб"; break;  case "sphere": typeName = "Сфера"; break;  }  if (typeName === null) {  return;  }  let objectHtml = `  <div id="object-${id}" class="w3-padding w3-margin-bottom w3-border w3-border-white">  <div class="w3-large" onclick='changeAccordionVisibility("accordion-${id}")'>Объект №${id} - ${typeName}</div>  <div id="accordion-${id}" class="w3-margin-top w3-hide w3-show">  <div>  <label for="input-x-${id}" class="w3-left">Положение (X=<span id="output-x-${id}"></span>)</label>  <input id="input-x-${id}" class="w3-right" type="range" min="-20" max="20" step="0.05" value="0" style="width: 200px;" oninput="onParametersChanged()"/>  </div>  <br/>  <div>  <label for="input-y-${id}" class="w3-left">Положение (Y=<span id="output-y-${id}"></span>)</label>  <input id="input-y-${id}" class="w3-right" type="range" min="-20" max="20" step="0.05" value="0" style="width: 200px;" oninput="onParametersChanged()"/>  </div>  <br/>  <div>  <label for="input-z-${id}" class="w3-left">Положение (Z=<span id="output-z-${id}"></span>)</label>  <input id="input-z-${id}" class="w3-right" type="range" min="-20" max="20" step="0.01" value="0" style="width: 200px;" oninput="onParametersChanged()"/>  </div>  <br/>  <div>  <label for="input-rotation-x-${id}" class="w3-left">Поворот (RX=<span id="output-rotation-x-${id}"></span>)</label>  <input id="input-rotation-x-${id}" class="w3-right" type="range" min="0" max="360" step="1" value="0" style="width: 200px;" oninput="onParametersChanged()"/>  </div>  <br/>  <div>  <label for="input-rotation-y-${id}" class="w3-left">Поворот (RY=<span id="output-rotation-y-${id}"></span>)</label>  <input id="input-rotation-y-${id}" class="w3-right" type="range" min="0" max="360" step="1" value="0" style="width: 200px;" oninput="onParametersChanged()"/>  </div>  <br/>  <div>  <label for="input-rotation-z-${id}" class="w3-left">Поворот (RZ=<span id="output-rotation-z-${id}"></span>)</label>  <input id="input-rotation-z-${id}" class="w3-right" type="range" min="0" max="360" step="1" value="0" style="width: 200px;" oninput="onParametersChanged()"/>  </div>  <br/>  <div>  <label for="input-scale-x-${id}" class="w3-left">Масштаб (SX=<span id="output-scale-x-${id}"></span>)</label>  <input id="input-scale-x-${id}" class="w3-right" type="range" min="0" max="10" step="0.05" value="1" style="width: 200px;" oninput="onParametersChanged()"/>  </div>  <br/>  <div>  <label for="input-scale-y-${id}" class="w3-left">Масштаб (SY=<span id="output-scale-y-${id}"></span>)</label>  <input id="input-scale-y-${id}" class="w3-right" type="range" min="0" max="10" step="0.05" value="1" style="width: 200px;" oninput="onParametersChanged()"/>  </div>  <br/>  <div>  <label for="input-scale-z-${id}" class="w3-left">Масштаб (SZ=<span id="output-scale-z-${id}"></span>)</label>  <input id="input-scale-z-${id}" class="w3-right" type="range" min="0" max="10" step="0.05" value="1" style="width: 200px;" oninput="onParametersChanged()"/>  </div>  <br/>  <div>  <label for="input-alpha-${id}" class="w3-left">Прозрачность (A=<span id="output-alpha-${id}"></span>)</label>  <input id="input-alpha-${id}" class="w3-right" type="range" min="0" max="1" step="0.01" value="1" style="width: 200px;" oninput="onParametersChanged()"/>  </div>  <br/>  <div class="w3-margin-bottom">  <label for="input-color-${id}" class="w3-left">Цвет</label>  <input id="input-color-${id}" class="w3-right" type="color" value="${colorToHex(Math.round(Math.random() \* 255), Math.round(Math.random() \* 255), Math.round(Math.random() \* 255))}" style="width: 200px;" oninput="onParametersChanged()"/>  </div>  <br/>  </div>  </div>  `;  let objectsDiv = $("#objects");  objectsDiv.append(objectHtml);  let object = null;  switch (type) {  case "cube": object = new SceneObject(); break;  case "sphere": object = new SceneObject(); break;  }  if (object !== null) {  object.loadFromObjFile(type, () => {  SCENE\_OBJECTS.push(object);  onParametersChanged();  });  }  }  export function clearSceneObjects() {  SCENE\_OBJECTS = [];  $("#objects").html("");  onParametersChanged();  renderScene(SCENE\_OBJECTS, CAMERA\_OBJECT, RENDER\_PARAMETERS);  } |

**Выводы.**

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