**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №1**

**по дисциплине «Компьютерная графика»**

Тема: Рисование геометрических объектов

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

Санкт-Петербург

2023

**Цель работы.**

Целью работы является ознакомление с основными примитивами WebGL. Требования и рекомендации к выполнению задания:

* Проанализировать полученное задание, выделить информационные объекты и действия;
* Разработать программу с использованием требуемых примитивов и атрибутов.

**Задание.**

Получите удобное рисование с буферами вершин, униформой и шейдерами:

* Рисование нескольких вещей с отдельными командами рисования;
* Используя разные примитивы;
* Изменение размеров линий и точек по умолчанию;
* Изменение цвета на лету.

**Выполнение работы.**

1. **Используемые технологии**

Для реализации программы использовались язык программирования JavaScript, библиотека jQuery, API WebGL для 3D-графики и набор стилей W3.CSS.

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.**  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.**  function initializeWebGl() {  GL = getWebGlContext();  initializeShaderProgram(GL);  initializeViewport(GL);  // Дополнительные функции  GL.enable(GL.DEPTH\_TEST);  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 цвета при помощи атрибутов *vertexPosition* и *vertexColor*. Координаты устанавливаются в *gl\_Position*, а цвет передается фрагментному шейдеру. Шейдер представлен в листинге 8.

|  |
| --- |
| **Листинг 8. Вершинный шейдер.**  let VERTEX\_SHADER\_SOURCE = `  attribute vec3 vertexPosition;  attribute vec4 vertexColor;  varying lowp vec4 vColor;  void main() {  gl\_Position = vec4(vertexPosition, 1.0);  vColor = vertexColor;  }  `; |

* 1. **Фрагментный шейдер**

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

|  |
| --- |
| **Листинг 9. Фрагментный шейдер.**  let FRAGMENT\_SHADER\_SOURCE = `  precision mediump float;  varying lowp vec4 vColor;  void main() {  gl\_FragColor = vColor;  }  `; |

1. **Объекты графических примитивов**

Классы всех графических примитивов наследуются от базового абстрактного класса *SceneObject*. Данный класс определяет интерфейс для производных классов в виде трех методов: *getMeshTriangles*, *getVerticesForDrawing* и *getColors*.

Метод *getMeshTriangles* возвращает массив вершин 2D-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 *getVerticesForDrawing* берет локальные координаты вершин 2D-объекта (при помощи метода *getMeshTriangles*) и преобразует их 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 *getColors* возвращает массив цветов для вершин 2D-объекта. Размер возвращаемого массива соответствует размеру массива вершин.

Класс *SceneObject* представлен в листинге 10.

|  |
| --- |
| **Листинг 10. Класс SceneObject.**  class SceneObject {  constructor(position, rotation, scale, color) {  if (this.constructor === SceneObject)  {  throw new Error("Abstract classes cannot be initialized.");  }  this.position = position;  this.rotation = rotation;  this.scale = scale;  this.color = color;  }  getMeshTriangles() {  return [new Vector3D(), new Vector3D(), new Vector3D()];  }  getVerticesForDrawing() {  let vertices = this.getMeshTriangles();  let result = [];  for(let i = 0; i < vertices.length; ++i)  {  // Scale  vertices[i].x \*= this.scale.x;  vertices[i].y \*= this.scale.y;  vertices[i].z \*= this.scale.z;  // 2D rotation  let angle = Math.radians(this.rotation.z);  let new\_x = vertices[i].x \* Math.cos(angle) - vertices[i].y \* Math.sin(angle);  let new\_y = vertices[i].x \* Math.sin(angle) + vertices[i].y \* Math.cos(angle);  vertices[i].x = new\_x;  vertices[i].y = new\_y;  // Position  vertices[i].x += this.position.x;  vertices[i].y += this.position.y;  vertices[i].z += this.position.z;  result.push(vertices[i].x, vertices[i].y, vertices[i].z);  }  return result;  }  getColors() {  let count = this.getMeshTriangles().length;  let result = [];  for(let i = 0; i < count; ++i)  {  result.push(this.color.r, this.color.g, this.color.b, this.color.a);  }  return result;  }  } |

Было разработано два графических примитива:

* Квадрат (класс *PlaneObject*);
* Круг (класс *CircleObject*).

Класс *PlaneObject* представлен в листинге 11.

|  |
| --- |
| **Листинг 11. Класс PlaneObject.**  class PlaneObject extends SceneObject {  constructor(position = new Vector3D(), rotation = new Vector3D(), scale = new Vector3D(1.0, 1.0, 1.0), color = new Color()) {  super(position, rotation, scale, color);  }  getMeshTriangles() {  let base\_size = 0.1;  return [new Vector3D(base\_size, base\_size, 0.0),  new Vector3D(-base\_size, base\_size, 0.0),  new Vector3D(-base\_size, -base\_size, 0.0),  new Vector3D(-base\_size, -base\_size, 0.0),  new Vector3D(base\_size, -base\_size, 0.0),  new Vector3D(base\_size, base\_size, 0.0)];  }  } |

Класс *CircleObject* представлен в листинге 12.

|  |
| --- |
| **Листинг 12. Класс CircleObject.**  class CircleObject extends SceneObject {  constructor(position = new Vector3D(), rotation = new Vector3D(), scale = new Vector3D(1.0, 1.0, 1.0), color = new Color()) {  super(position, rotation, scale, color);  }  getMeshTriangles() {  let base\_size = 0.1;  let verticesCount = 32;  let angle = 0;  let d\_angle = 360.0 / verticesCount;  let result = [];  for (let i = 0; i < verticesCount; ++i) {  result.push(new Vector3D());  result.push(new Vector3D(base\_size \* Math.cos(Math.radians(angle)), base\_size \* Math.sin(Math.radians(angle)), 0.0));  angle += d\_angle  result.push(new Vector3D(base\_size \* Math.cos(Math.radians(angle)), base\_size \* Math.sin(Math.radians(angle)), 0.0));  }  return result;  }  } |

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 графических примитивов сцены, хранящихся в глобальной переменной *SCENE\_OBJECTS*. Функция *renderScene* представлена в листинге 13.

|  |
| --- |
| **Листинг 13. Функция renderScene.**  function renderScene() {  GL.clearColor(0.0, 0.0, 0.0, 1.0);  GL.clear(GL.COLOR\_BUFFER\_BIT);  GL.clear(GL.DEPTH\_BUFFER\_BIT);  SCENE\_OBJECTS.forEach((object) => {  let vertices = object.getVerticesForDrawing();  let colors = object.getColors();  let verticesCount = vertices.length / 3;  let colorsCount = colors.length / 4;  let vertexBuffer = GL.createBuffer();  GL.bindBuffer(GL.ARRAY\_BUFFER, vertexBuffer);  GL.bufferData(GL.ARRAY\_BUFFER, new Float32Array(vertices), GL.STATIC\_DRAW);  let vertexPositionAttr = GL.getAttribLocation(PROGRAM, "vertexPosition");  GL.vertexAttribPointer(vertexPositionAttr, 3, GL.FLOAT, false, 0, 0);  GL.enableVertexAttribArray(vertexPositionAttr);  GL.bindBuffer(GL.ARRAY\_BUFFER, null);  let colorBuffer = GL.createBuffer();  GL.bindBuffer(GL.ARRAY\_BUFFER, colorBuffer);  GL.bufferData(GL.ARRAY\_BUFFER, new Float32Array(colors), GL.STATIC\_DRAW);  let vertexColorAttr = GL.getAttribLocation(PROGRAM, "vertexColor");  GL.vertexAttribPointer(vertexColorAttr, 4, GL.FLOAT, false, 0, 0);  GL.enableVertexAttribArray(vertexColorAttr);  GL.bindBuffer(GL.ARRAY\_BUFFER, null);  GL.drawArrays(GL.TRIANGLES, 0, verticesCount);  GL.deleteBuffer(vertexBuffer);  GL.deleteBuffer(colorBuffer);  });  } |

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 графические 2D-примитивы на сцены, редактировать параметры созданных графических 2D-примитивов, а также удалять все объекты со сцены для ее очистки.
* Справа располагается панель с краткой справкой для пользователя, где описаны параметры графических примитивов.

Графические примитивы имеют следующие параметры:

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

Интерфейс программы представлен на рисунке 1.

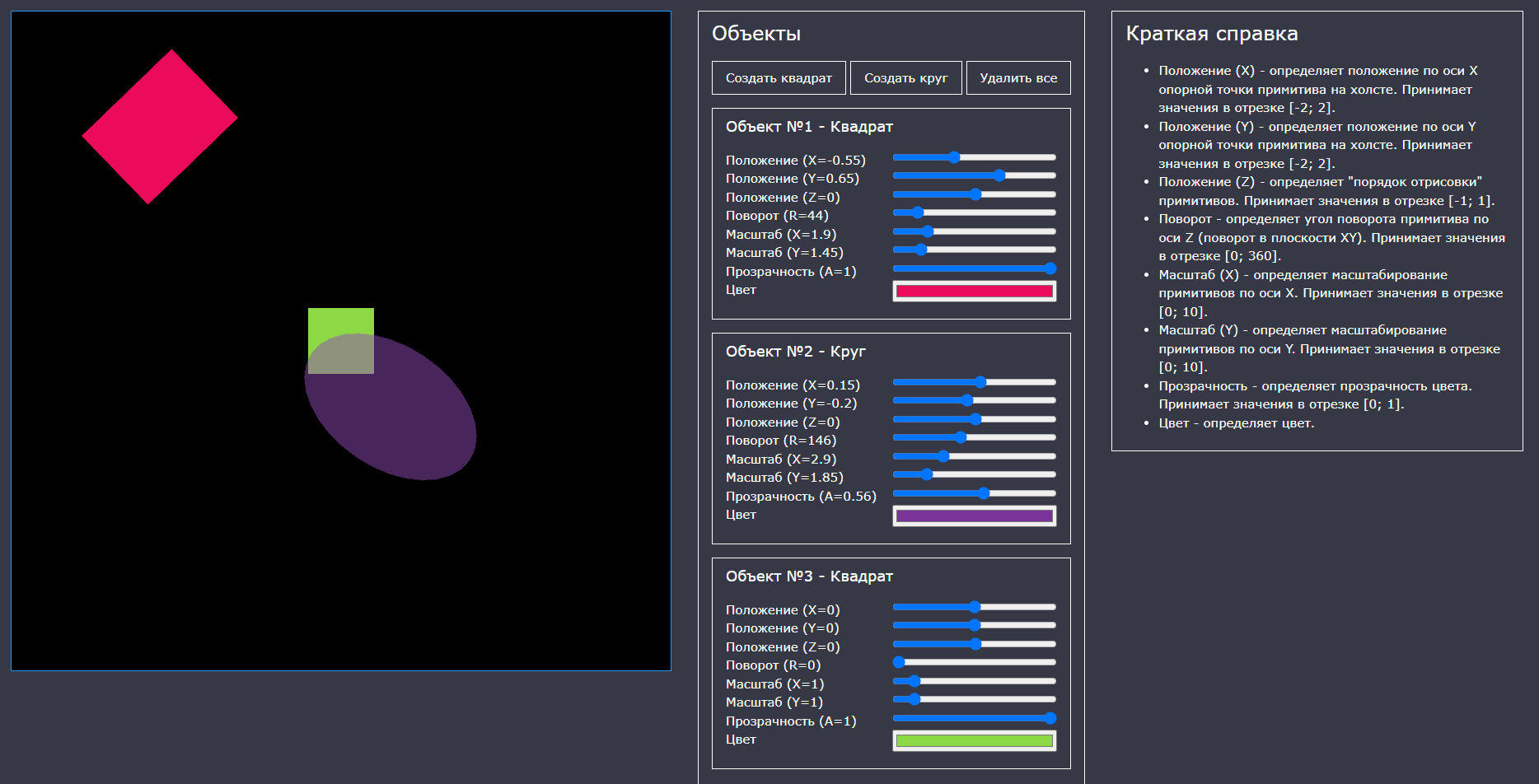


Рисунок 1 – Интерфейс программы.

Для изменения значений параметров примитивов используются HTML-элементы *input* типа *range* и *color*. При любых изменениях значений параметров вызывается функция *onParametersChanged*, которая извлекает значения из HTML-элементов, устанавливает их соответствующим объектам из массива *SCENE\_OBJECTS*, после чего вызывается функция *renderScene*. Функция *onParametersChanged* представлена в листинге 14.

|  |
| --- |
| **Листинг 14. Функция onParametersChanged.**  function onParametersChanged() {  for (let i = 0; i < SCENE\_OBJECTS.length; ++i) {  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.x = Number(e\_x.value);  SCENE\_OBJECTS[i].position.y = Number(e\_y.value);  SCENE\_OBJECTS[i].position.z = 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);  let e\_r = document.getElementById(`input-rotation-${i+1}`);  SCENE\_OBJECTS[i].rotation.z = Number(e\_r.value);  $(`#output-rotation-${i+1}`).text(e\_r.value);  let e\_s\_x = document.getElementById(`input-scale-x-${i+1}`);  let e\_s\_y = document.getElementById(`input-scale-y-${i+1}`);  SCENE\_OBJECTS[i].scale.x = Number(e\_s\_x.value);  SCENE\_OBJECTS[i].scale.y = Number(e\_s\_y.value);  $(`#output-scale-x-${i+1}`).text(e\_s\_x.value);  $(`#output-scale-y-${i+1}`).text(e\_s\_y.value);  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);  }  renderScene();  } |

Для добавления и удаления примитивов сцены используются функции *createSceneObject* и *clearSceneObjects* соответственно. Функции представлены в листинге 15.

|  |
| --- |
| **Листинг 15. Функции createSceneObject и clearSceneObjects.**  function createSceneObject(type) {  let id = SCENE\_OBJECTS.length + 1;  let typeName = null;  switch (type) {  case "square": typeName = "Квадрат"; break;  case "circle": typeName = "Круг"; 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-${id}" class="w3-left">Положение (X=<label id="output-x-${id}"></label>)</label>  <input id="input-x-${id}" class="w3-right" type="range" min="-2" max="2" step="0.05" value="0" style="width: 200px;" oninput="onParametersChanged()"/>  </div>  <br/>  <div>  <label for="input-y-${id}" class="w3-left">Положение (Y=<label id="output-y-${id}"></label>)</label>  <input id="input-y-${id}" class="w3-right" type="range" min="-2" max="2" step="0.05" value="0" style="width: 200px;" oninput="onParametersChanged()"/>  </div>  <br/>  <div>  <label for="input-z-${id}" class="w3-left">Положение (Z=<label id="output-z-${id}"></label>)</label>  <input id="input-z-${id}" class="w3-right" type="range" min="-1" max="0.99" step="0.01" value="0" style="width: 200px;" oninput="onParametersChanged()"/>  </div>  <br/>  <div>  <label for="input-rotation-${id}" class="w3-left">Поворот (R=<label id="output-rotation-${id}"></label>)</label>  <input id="input-rotation-${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-${id}" class="w3-left">Масштаб (X=<label id="output-scale-x-${id}"></label>)</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 (Y=<label id="output-scale-y-${id}"></label>)</label>  <input id="input-scale-y-${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-alpha-${id}" class="w3-left">Прозрачность (A=<label id="output-alpha-${id}"></label>)</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 "square": object = new PlaneObject(); break;  case "circle": object = new CircleObject(); break;  }  if (object !== null) {  SCENE\_OBJECTS.push(object);  onParametersChanged();  }  }  function clearSceneObjects() {  SCENE\_OBJECTS = [];  $("#objects").html("");  renderScene();  } |

**Выводы.**

В результате выполнения лабораторной работы была разработана программа на языке JavaScript, которая осуществляет рисование графических 2D-примитивов (квадрат и круг) с параметрами положения, вращения, масштабирования и цвета.