**Липецкий государственный технический университет**  
Факультет автоматизации и информатики  
Кафедра автоматизированных систем управления

Лабораторная работа №3

по дисциплине «Организация графических систем и систем мультимедиа»

Построение 3D-сцен с использованием WebGL

Пархомов А.С.

Студент

Группа М-АС-21

Кургасов В.В.

Руководитель

Доцент

Липецк 2022 г.

Цель работы

Изучить построение 3D-сцен в браузере с помощью технологии WebGL.

Задание кафедры

Создать 3D-сцену (не менее 3-х объектов) с использованием WebGL. Разработанная сцена должна быть управляемой и применяемой/используемой (демонстрация физического процесса, природного явления и т.д.).

Содержание

[1 Теоретические сведения 5](#_Toc103542718)

[2 Ход работы 6](#_Toc103542719)

[2.1 Использование приложения 7](#_Toc103542720)

[2.2 Создание Docker-контейнера 9](#_Toc103542721)

[Вывод 10](#_Toc103542722)

[Приложение А 11](#_Toc103542723)

1. Теоретические сведения

В качестве языка программирования для реализации проекта был выбран язык программирования JavaScript со следующим набором библиотек: dat.gui, Detector, JQuery, Mustache, OrbitControls, ShaderLoader, stats, three. Кроме того, был написан шейдер трассировки лучей на языке GLSL. Данные инструменты были выбраны для достижения высокой производительности с использованием современных технологий. Разработанное приложение упаковано в Docker-контейнер, что позволяет запустить его на любой системе с установленным Docker Engine и браузером, поддерживающим WebGL. Исходный код основной части программы представлен в приложении А.

WebGL – программная библиотека для языка JavaScript предназначенная для визуализации интерактивной трёхмерной графики и двухмерной графики в пределах совместимости веб-браузера без использования плагинов. WebGL приносит в веб трёхмерную графику, вводя API, который построен на основе OpenGL ES 2.0, что позволяет его использовать в элементах Canvas HTML5.

В контексте работы с библиотекой WebGL, графический процессор выполняет две функции: обработка положений (массива данных) и вершин пространства отсечения и отрисовка пикселей на основе результатов первой задачи.

Вершинный шейдер – это функция, написанная на GLSL. Она вызывается один раз на каждую вершину. Разработчик описывает математическую часть и устанавливает специальную переменную gl\_Position со значением координат пространства отсечения для текущей вершины. Видеокарта принимает это значение и хранит его во внутренней памяти.

Например, когда необходимо отрисовать треугольник, первая задача создаёт 3 вершины, видеокарта использует их для создания треугольника. Она определяет, какие пиксели соответствуют 3 точкам треугольника и затем растеризует треугольник. Растеризация означает "отрисовка объекта пикселями". Для каждого пикселя будет вызван фрагментный шейдер, где вас спросят, каким цветом нужно закрасить этот пиксель. Чтобы ответить на этот вопрос, ваш фрагментный шейдер должен установить специальной переменной gl\_FragColor значение цвета для данного пикселя.

1. Ход работы

Разработанное приложение представляет 3D-сцену галактики с чёрной дырой и планетой, вращающейся вокруг неё. Третьим основным объектом сцены является аккреционный диск, который возникает из-за поглощения большого количества материи чёрной дырой. Пользователь может управлять сценой с помощью меню, представленного на рисунке 1.

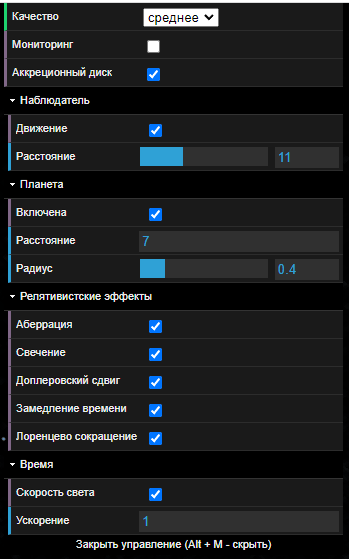


Рисунок 1 – Меню управления сценой

Кроме того, пользователь может перемещаться (приближать, вращать) по сцене с помощью мыши.

Физические законы, по которым функционирует чёрная дыра, достаточно сложны, поэтому в приложении учтены только основные релятивистские эффекты:

* аберрация – изменение направления распространения [света](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D0%B5%D1%82) (излучения) при переходе из одной [системы отсчёта](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BE%D1%82%D1%81%D1%87%D1%91%D1%82%D0%B0) к другой;
* доплеровский сдвиг – изменение частоты и длины волны излучения, воспринимаемой наблюдателем, вследствие движения источника излучения относительно наблюдателя;
* Лоренцево сокращение – эффект релятивистской кинематики, заключающийся в том, что с точки зрения наблюдателя движущиеся относительно него предметы и пространство имеют меньшую длину (линейные размеры) в направлении движения, чем их собственная длина.

Разработанное приложение можно использовать для демонстрации физических законов начальным курсам с целью привлечения внимания к науке и программированию.

* 1. Использование приложения

На рисунках 2-4 показана работа приложения с разными параметрами.



Рисунок 2 – Стандартный угол обзора и скрытое меню

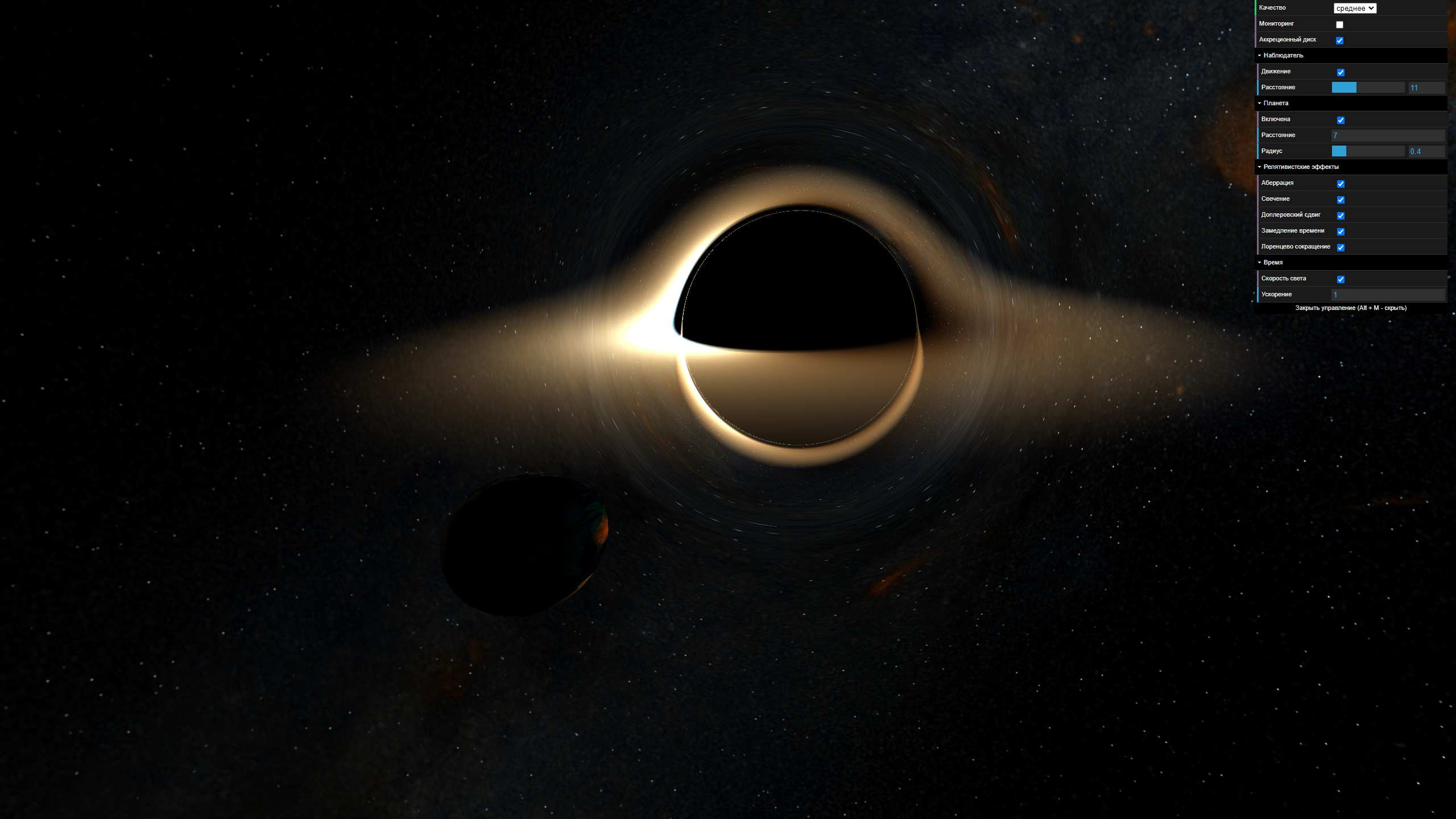


Рисунок 3 – Стандартный угол обзора и меню приложения

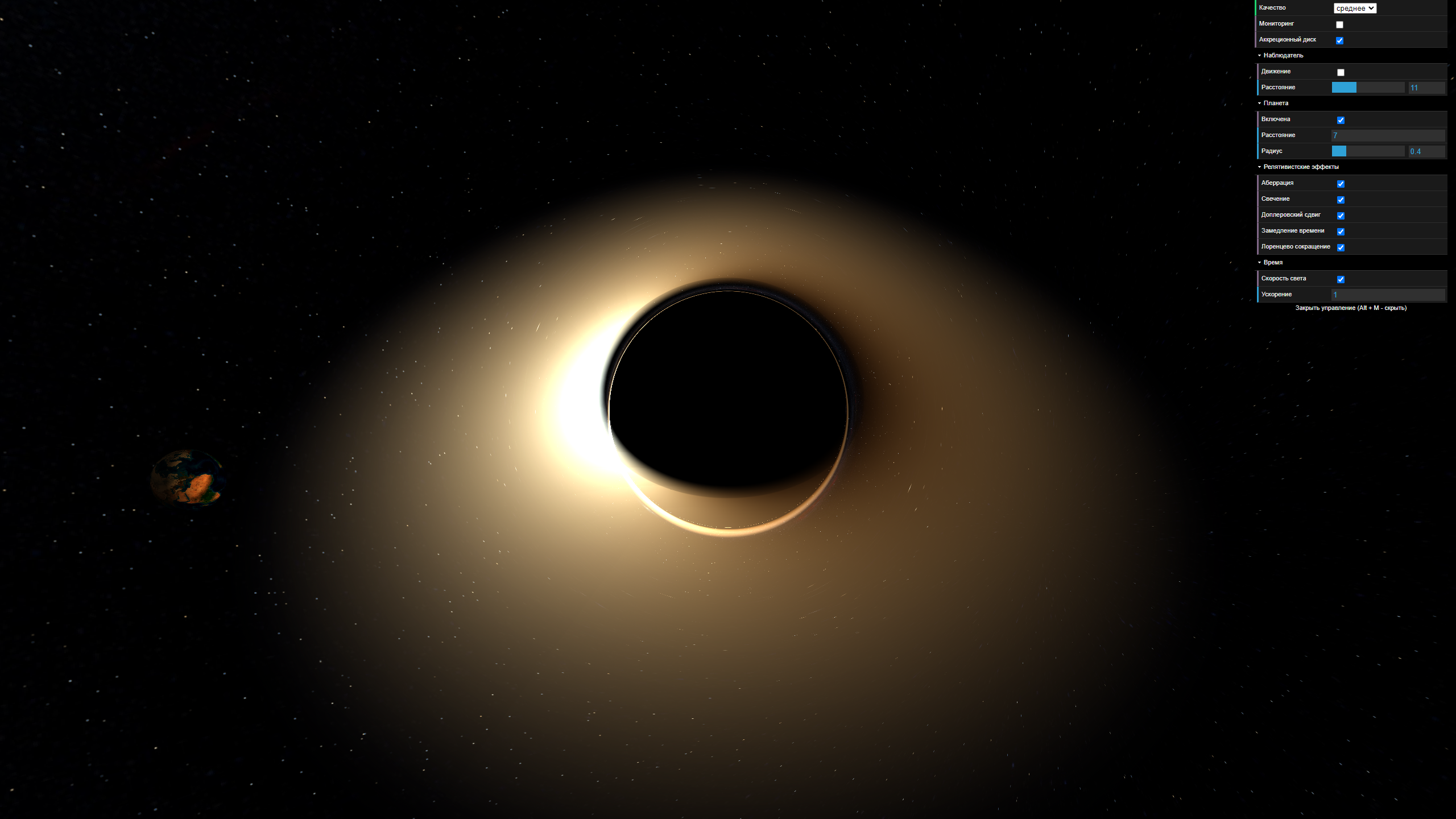


Рисунок 4 – Взгляд «сверху» относительно главной плоскости аккреционного диска

* 1. Создание Docker-контейнера

Для удобного распространения разработанного приложения был создан Docker-контейнер. Файл сборки образа контейнера:

FROM nginx:1.21.5

COPY . /usr/share/nginx/html

EXPOSE 80

Для запуска проекта необходимо выполнить следующие команды:

1. Сборка docker-образа: *docker build . -t graphicslb3\_black-hole*.
2. Создание и запуск контейнера с проектом на порту 8080: *docker run -d -p 8080:80 --name black-hole graphicslb3\_black-hole*.

Вывод

В ходе выполнения данной лабораторной работы было изучено построение 3D-сцен в браузере с помощью технологии WebGL и разработано приложение, представляющее собой 3D-сцену галактики с чёрной дырой и планетой, вращающейся вокруг неё. Пользователь может управлять сценой с помощью меню и мыши. Использованные при разработке приложения физические законы могут быть продемонстрированы начальным курсам с целью привлечения внимания к науке и программированию.

Также был создан Docker-контейнер для удобного распространения разработанного приложения.

Приложение А

Исходный код основного файла приложения

Исходный код файла main.js:

"use strict"

// если нет поддержки WebGL - вывести сообщение

if ( ! Detector.webgl ) Detector.addGetWebGLMessage();

// функция настройки параметров наблюдателя

function Observer() {

    this.position = new THREE.Vector3(10,0,0);

    this.velocity = new THREE.Vector3(0,1,0);

    this.orientation = new THREE.Matrix3();

    this.time = 0.0;

}

// функция инициализации кадра орбиты для наблюдателя

Observer.prototype.orbitalFrame = function() {

    var orbital\_y = (new THREE.Vector3())

        .subVectors(observer.velocity.clone().normalize().multiplyScalar(4.0),

            observer.position).normalize();

    var orbital\_z = (new THREE.Vector3())

        .crossVectors(observer.position, orbital\_y).normalize();

    var orbital\_x = (new THREE.Vector3()).crossVectors(orbital\_y, orbital\_z);

    return (new THREE.Matrix4()).makeBasis(

        orbital\_x,

        orbital\_y,

        orbital\_z

    ).linearPart();

};

// функция расчёта перемещения наблюдателя

Observer.prototype.move = function(dt) {

    dt \*= shader.parameters.time\_scale;

    var r;

    var v = 0;

    if (shader.parameters.observer.motion) {

        r = shader.parameters.observer.distance;

        v =  1.0 / Math.sqrt(2.0\*(r-1.0));

        var ang\_vel = v / r;

        var angle = this.time \* ang\_vel;

        var s = Math.sin(angle), c = Math.cos(angle);

        this.position.set(c\*r, s\*r, 0);

        this.velocity.set(-s\*v, c\*v, 0);

        var alpha = degToRad(shader.parameters.observer.orbital\_inclination);

        var orbit\_coords = (new THREE.Matrix4()).makeRotationY(alpha);

        this.position.applyMatrix4(orbit\_coords);

        this.velocity.applyMatrix4(orbit\_coords);

    }

    else {

        r = this.position.length();

    }

    if (shader.parameters.gravitational\_time\_dilation) {

        dt = Math.sqrt((dt\*dt \* (1.0 - v\*v)) / (1-1.0/r));

    }

    this.time += dt;

};

// объявление основных глобальных переменных

var container, stats, isStats = false;

var camera, scene, renderer, cameraControls, shader = null;

var observer = new Observer();

// функция инициализации шейдера

function Shader(mustacheTemplate) {

    this.parameters = {

        n\_steps: 100,

        quality: 'среднее',

        accretion\_disk: true,

        planet: {

            enabled: true,

            distance: 7.0,

            radius: 0.4

        },

        lorentz\_contraction: true,

        gravitational\_time\_dilation: true,

        aberration: true,

        beaming: true,

        doppler\_shift: true,

        light\_travel\_time: true,

        time\_scale: 1.0,

        observer: {

            motion: true,

            distance: 11.0,

            orbital\_inclination: -10

        },

        planetEnabled: function() {

            return this.planet.enabled && this.quality !== 'низкое';

        },

        observerMotion: function() {

            return this.observer.motion;

        }

    };

    var that = this;

    this.needsUpdate = false;

    this.hasMovingParts = function() {

        return this.parameters.planet.enabled || this.parameters.observer.motion;

    };

    this.compile = function() {

        return Mustache.render(mustacheTemplate, that.parameters);

    };

}

// функция перевода градусов в радианы

function degToRad(a) { return Math.PI \* a / 180.0; }

// функция загрузки текстур

(function(){

    var textures = {};

    function whenLoaded() {

        init(textures);

        $('#loader').hide();

        $('.initially-hidden').removeClass('initially-hidden');

        animate();

    }

    function checkLoaded() {

        if (shader === null) return;

        for (var key in textures) if (textures[key] === null) return;

        whenLoaded();

    }

    SHADER\_LOADER.load(function(shaders) {

        shader = new Shader(shaders.raytracer.fragment);

        checkLoaded();

    });

    var texLoader = new THREE.TextureLoader();

    function loadTexture(symbol, filename, interpolation) {

        textures[symbol] = null;

        texLoader.load(filename, function(tex) {

            tex.magFilter = interpolation;

            tex.minFilter = interpolation;

            textures[symbol] = tex;

            checkLoaded();

        });

    }

    loadTexture('galaxy', 'img/milkyway.jpg', THREE.NearestFilter);

    loadTexture('spectra', 'img/spectra.png', THREE.LinearFilter);

    loadTexture('planet', 'img/planet.png', THREE.LinearFilter);

    loadTexture('stars', 'img/stars.png', THREE.LinearFilter);

    loadTexture('accretion\_disk', 'img/accretion-disk.png', THREE.LinearFilter);

})();

var updateUniforms;

// функция инициализации текстур

function init(textures) {

    container = document.createElement( 'div' );

    document.body.appendChild( container );

    scene = new THREE.Scene();

    var geometry = new THREE.PlaneBufferGeometry( 2, 2 );

    var uniforms = {

        time: { type: "f", value: 0 },

        resolution: { type: "v2", value: new THREE.Vector2() },

        cam\_pos: { type: "v3", value: new THREE.Vector3() },

        cam\_x: { type: "v3", value: new THREE.Vector3() },

        cam\_y: { type: "v3", value: new THREE.Vector3() },

        cam\_z: { type: "v3", value: new THREE.Vector3() },

        cam\_vel: { type: "v3", value: new THREE.Vector3() },

        planet\_distance: { type: "f" },

        planet\_radius: { type: "f" },

        star\_texture: { type: "t", value: textures.stars },

        accretion\_disk\_texture: { type: "t",  value: textures.accretion\_disk },

        galaxy\_texture: { type: "t", value: textures.galaxy },

        planet\_texture: { type: "t", value: textures.planet },

        spectrum\_texture: { type: "t", value: textures.spectra }

    };

    updateUniforms = function() {

        uniforms.planet\_distance.value = shader.parameters.planet.distance;

        uniforms.planet\_radius.value = shader.parameters.planet.radius;

        uniforms.resolution.value.x = renderer.domElement.width;

        uniforms.resolution.value.y = renderer.domElement.height;

        uniforms.time.value = observer.time;

        uniforms.cam\_pos.value = observer.position;

        var e = observer.orientation.elements;

        uniforms.cam\_x.value.set(e[0], e[1], e[2]);

        uniforms.cam\_y.value.set(e[3], e[4], e[5]);

        uniforms.cam\_z.value.set(e[6], e[7], e[8]);

        function setVec(target, value) {

            uniforms[target].value.set(value.x, value.y, value.z);

        }

        setVec('cam\_pos', observer.position);

        setVec('cam\_vel', observer.velocity);

    };

    var material = new THREE.ShaderMaterial( {

        uniforms: uniforms,

        vertexShader: $('#vertex-shader').text(),

    });

    scene.updateShader = function() {

        material.fragmentShader = shader.compile();

        material.needsUpdate = true;

        shader.needsUpdate = true;

    };

    scene.updateShader();

    var mesh = new THREE.Mesh( geometry, material );

    scene.add( mesh );

    renderer = new THREE.WebGLRenderer();

    renderer.setPixelRatio( window.devicePixelRatio );

    container.appendChild( renderer.domElement );

    camera = new THREE.PerspectiveCamera( 45, window.innerWidth / window.innerHeight, 1, 80000 );

    initializeCamera(camera);

    cameraControls = new THREE.OrbitControls( camera, renderer.domElement );

    cameraControls.target.set( 0, 0, 0 );

    cameraControls.addEventListener( 'change', updateCamera );

    updateCamera();

    onWindowResize();

    window.addEventListener( 'resize', onWindowResize, false );

    setupGUI();

}

// функция инициализации панели управления

function setupGUI() {

    var hint = $('#hint-text');

    var p = shader.parameters;

    function updateShader() {

        hint.hide();

        scene.updateShader();

    }

    var gui = new dat.GUI();

    gui.add(p, 'quality', ['низкое', 'среднее', 'высокое']).name('Качество').onChange(function (value) {

        $('.planet-controls').show();

        switch(value) {

        case 'низкое':

            p.n\_steps = 40;

            $('.planet-controls').hide();

            break;

        case 'среднее':

            p.n\_steps = 100;

            break;

        case 'высокое':

            p.n\_steps = 200;

            break;

        }

        updateShader();

    });

    gui.add({is\_show\_stats: isStats}, 'is\_show\_stats').name('Мониторинг').onChange(function (value) {

        if (value) {

            stats = new Stats();

            stats.domElement.style.position = 'absolute';

            stats.domElement.style.top = '0px';

            container.appendChild( stats.domElement );

            $(stats.domElement).addClass('hidden-phone');

        }

        else {

            document.getElementById('stats').remove()

        }

        isStats = value

    });

    gui.add(p, 'accretion\_disk').name('Аккреционный диск').onChange(updateShader);

    var folder = gui.addFolder('Наблюдатель');

    folder.add(p.observer, 'motion').name('Движение').onChange(function(motion) {

        updateCamera();

        updateShader();

        if (motion) {

            hint.text('Подвижное наблюдение. Используйте мышь для перемещения камеры');

        } else {

            hint.text('Стационарное наблюдение. Используйте мышь для перемещения по орбите');

        }

        hint.fadeIn();

    });

    folder.add(p.observer, 'distance').min(1.5).max(30).name('Расстояние').onChange(updateCamera);

    folder.open();

    folder = gui.addFolder('Планета');

    folder.add(p.planet, 'enabled').name('Включена').onChange(function(enabled) {

        updateShader();

        var controls = $('.indirect-planet-controls').show();

        if (enabled) controls.show();

        else controls.hide();

    });

    folder.add(p.planet, 'distance').name('Расстояние').min(1.5).onChange(updateUniforms);

    folder.add(p.planet, 'radius').min(0.01).max(2.0).name('Радиус').onChange(updateUniforms);

    $(folder.domElement).addClass('planet-controls');

    function setGuiRowClass(guiEl, klass) {

        $(guiEl.domElement).parent().parent().addClass(klass);

    }

    folder.open()

    folder = gui.addFolder('Релятивистские эффекты');

    folder.add(p, 'aberration').name('Аберрация').onChange(updateShader);

    folder.add(p, 'beaming').name('Свечение').onChange(updateShader);

    folder.add(p, 'doppler\_shift').name('Доплеровский сдвиг').onChange(updateShader);

    setGuiRowClass(

        folder.add(p, 'gravitational\_time\_dilation').name('Замедление времени').onChange(updateShader),

        'planet-controls indirect-planet-controls');

    setGuiRowClass(

        folder.add(p, 'lorentz\_contraction').name('Лоренцево сокращение').onChange(updateShader),

        'planet-controls indirect-planet-controls');

    folder.open();

    folder = gui.addFolder('Время');

    folder.add(p, 'light\_travel\_time').name('Скорость света').onChange(updateShader);

    folder.add(p, 'time\_scale').min(0).name('Ускорение');

    folder.open()

    gui.width = 340

    // обработчик комбинации Alt + M для добавления/удаления панели управления

    window.addEventListener('keydown', function (event) {

        if (event.altKey && event.code === 'KeyM') {

            dat.GUI.toggleHide();

        }

    });

}

// обработчик события изменения размера окна

function onWindowResize( event ) {

    renderer.setSize( window.innerWidth, window.innerHeight );

    updateUniforms();

}

// функция инициализации камеры

function initializeCamera(camera) {

    var pitchAngle = 3.0, yawAngle = 0.0;

    camera.matrixWorldInverse.makeRotationX(degToRad(-pitchAngle));

    camera.matrixWorldInverse.multiply(new THREE.Matrix4().makeRotationY(degToRad(-yawAngle)));

    var m = camera.matrixWorldInverse.elements;

    camera.position.set(m[2], m[6], m[10]);

}

// функция обновления камеры

function updateCamera( event ) {

    var zoom\_dist = camera.position.length();

    var m = camera.matrixWorldInverse.elements;

    var camera\_matrix;

    if (shader.parameters.observer.motion) {

        camera\_matrix = new THREE.Matrix3();

    }

    else {

        camera\_matrix = observer.orientation;

    }

    camera\_matrix.set(

        m[0], m[1], m[2],

        m[8], m[9], m[10],

        m[4], m[5], m[6]

    );

    if (shader.parameters.observer.motion) {

        observer.orientation = observer.orbitalFrame().multiply(camera\_matrix);

    } else {

        var p = new THREE.Vector3(

            camera\_matrix.elements[6],

            camera\_matrix.elements[7],

            camera\_matrix.elements[8]);

        var dist = shader.parameters.observer.distance;

        observer.position.set(-p.x\*dist, -p.y\*dist, -p.z\*dist);

        observer.velocity.set(0,0,0);

    }

}

// функция вычисления расстояния Фробениуса

function frobeniusDistance(matrix1, matrix2) {

    var sum = 0.0;

    for (var i in matrix1.elements) {

        var diff = matrix1.elements[i] - matrix2.elements[i];

        sum += diff\*diff;

    }

    return Math.sqrt(sum);

}

// функция анимации объектов

function animate() {

    requestAnimationFrame( animate );

    camera.updateMatrixWorld();

    camera.matrixWorldInverse.getInverse( camera.matrixWorld );

    if (shader.needsUpdate || shader.hasMovingParts() ||

        frobeniusDistance(camera.matrixWorldInverse, lastCameraMat) > 1e-10) {

        shader.needsUpdate = false;

        render();

        lastCameraMat = camera.matrixWorldInverse.clone();

    }

    if (isStats) {

        stats.update();

    }

}

var lastCameraMat = new THREE.Matrix4().identity();

// функция для расчёта продолжительности кадра

var getFrameDuration = (function() {

    var lastTimestamp = new Date().getTime();

    return function() {

        var timestamp = new Date().getTime();

        var diff = (timestamp - lastTimestamp) / 1000.0;

        lastTimestamp = timestamp;

        return diff;

    };

})();

// функция рендера объектов

function render() {

    observer.move(getFrameDuration());

    if (shader.parameters.observer.motion) updateCamera();

    updateUniforms();

    renderer.render( scene, camera );}