Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Фундаментальные науки»

КАФЕДРА «Вычислительная математика и математическая физика» (ФН-11)

ОТЧЕТ

по домашней работе № 1

по курсу «Компьютерная геометрия»

Студент ФН11-51Б (Группа) _____ (Подпись, дата) Куприн А. Д. (И. О. Фамилия)

Преподаватель _____ (Подпись, дата) Захаров А. А. (И. О. Фамилия)

2023 г.

СОДЕРЖАНИЕ

с.

1 Задание

Напишите программу, обеспечивающую воспроизведение движения планет вокруг Солнца в трёхмерном пространстве подобно той, что описана на стр. 135–138 книги [2]. Задайте угловую скорость вращения каждой планеты. Наклоните оси планет. Добавьте к паре планет их спутники (например, Луну для Земли и Фобос и Деймос для Марса).

2 Выполнение

Репозиторий: `https://github.com/Plasmaa0/webgl-solar-system`

1. Создал компонент `WebGLCanvas` содержаший основной цикл выполнения программы. В нем задаются параметры камеры, создаются планеты, создается `GUI`.

2. Для реализации функционала камеры создан отдельный файл `Camera.js`, в котором написаны функции для получаения `view` и `projection` матриц. А также реализовано получения базиса камеры, а именно трех векторов направленных вперед по взгляду, направо и вверх. Это нужно для более приятного управления камерой на клавиши `W,A,S,D,Q,E` и стрелки.

3. В файле `Planet.js` создан класс планеты содержащий информацию о планете, параметрах ее орбиты, а также ее спутниках. Основные функции этого класса это:

   (a) `update(deltaTime)` - обновляет положение планеты в пространстве, положение планеты потом используется для создания её `model` матрицы. Реализация предельно простая, вытекает из школьной тригонометрии. Эта функция рекурсивно вызывается на всех спутниках планеты, которые сами по себе тоже являются представителями класса `Planet`. Такая реализация позволяет за один вызов `update()` на Солнце отрисовать рекурсивно сразу всю солнечную систему.

   (b) `draw(gl, shaderProgram, camera, params)` - аналогично `update()` вызывается рекурсивно на всех спутниках. Такая реализация позволяет за один вызов `draw()` на Солнце отрисовать рекурсивно сразу всю солнечную систему. Внутри функции в программе загружается нужный шейдер (для Солнца или остальных планет шейдеры отличаются из-за реализации света). После этого с помощью вспомогательных функций и методов получаются `MVP` матрицы и загружаются в шейдер. Генерируются вершины изосферы с помощью `generateSphereVertices()`. Они

также используются в качестве нормалей, так как на единичной изосфере координаты её вершин равны нормалям этих вершин. Обзор шейдеров будет отдельным пунктом.

(c) `generateSphereVertices()` - генерирует вершины изосферы с заданным количеством параллелей и меридианов.

4. В файле `shaders.js` находятся шейдеры для Солнца и остальных планет. Шейдер для солнца тривиален и рассмотрен не будет. Шейдер для планет является более продвинутым, так как в нем рассчитывается свет. В шейдер в момент вызова функции `draw()` у планеты передается множество нужных параметров позволяющих реализовать освещение. Например, положение источника света, направление света, нормали поверхностей и так далее. Реализованы несколько алгоритмов освещения:

   (a) Глобальное освещение (направление одинаково везде).

   (b) Точечный (`point`) источник света - применяется для симуляции света Солнца, в данный момент выключено.

   (c) `Spot`-свет освещающий только узкий конус пространства, раствор которого можно контролировать через графический интерфейс.

   (d) `Ambient` - фоновое освещение - небольшая подсветка всех поверхностей независимо от того, попадает ли на них свет, или они находятся в темноте.

   Детали реализации взяты из лекций и сайта `webglfundamentals.org/webgl/lessons`.

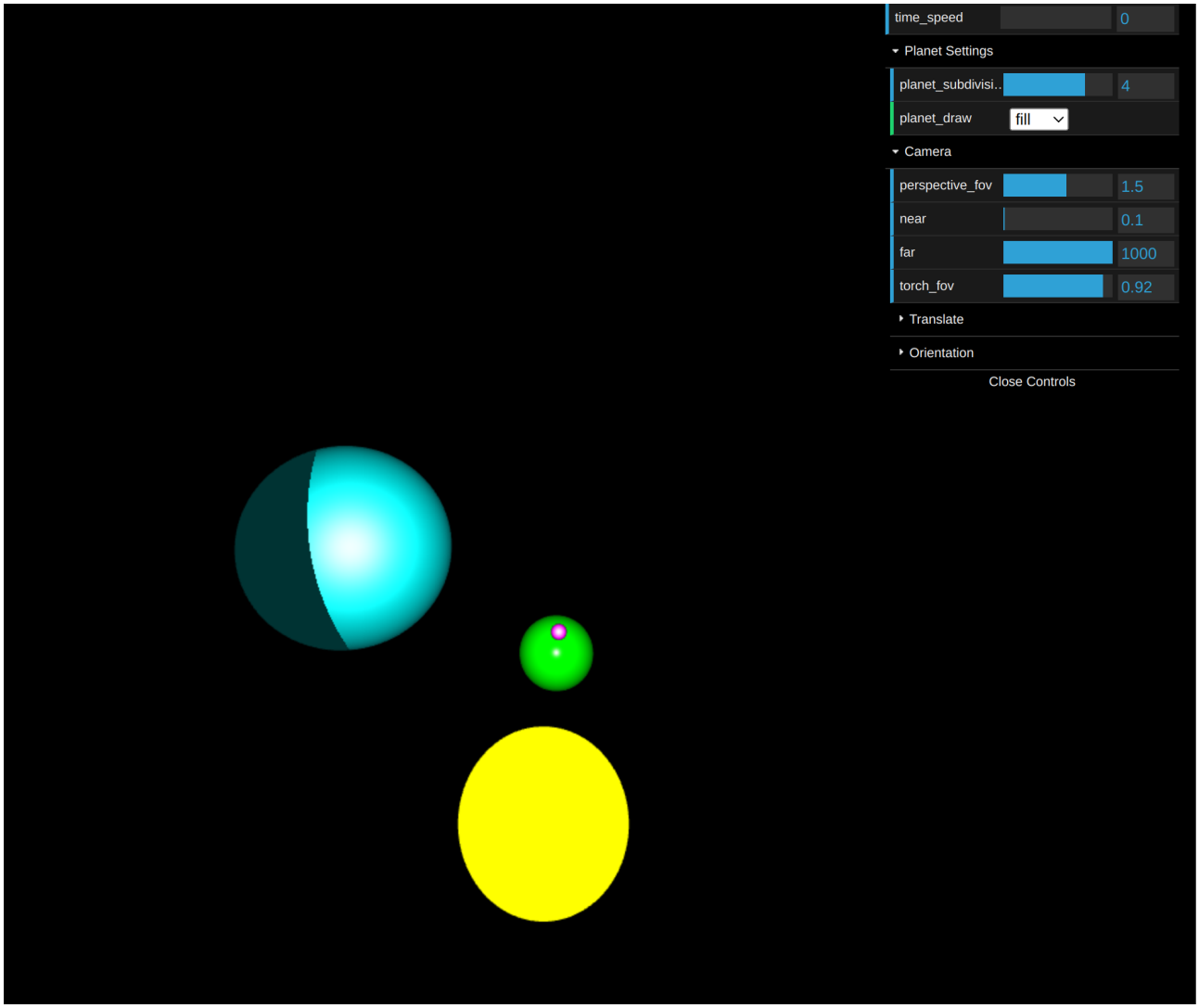

Рисунок 2.1 – Пример системы из двух планет, у одной из которых есть спутник. Освещение - spotlight с небольшим раствором направленный из камеры вперед.

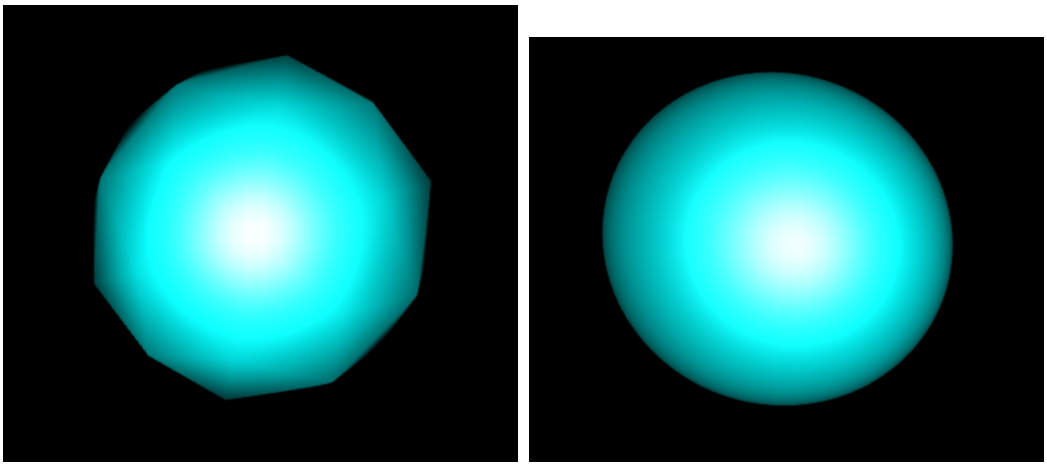

Рисунок 2.2 – Пример генерации изосферы с малым (слева) и большим (справа) количеством вершин.

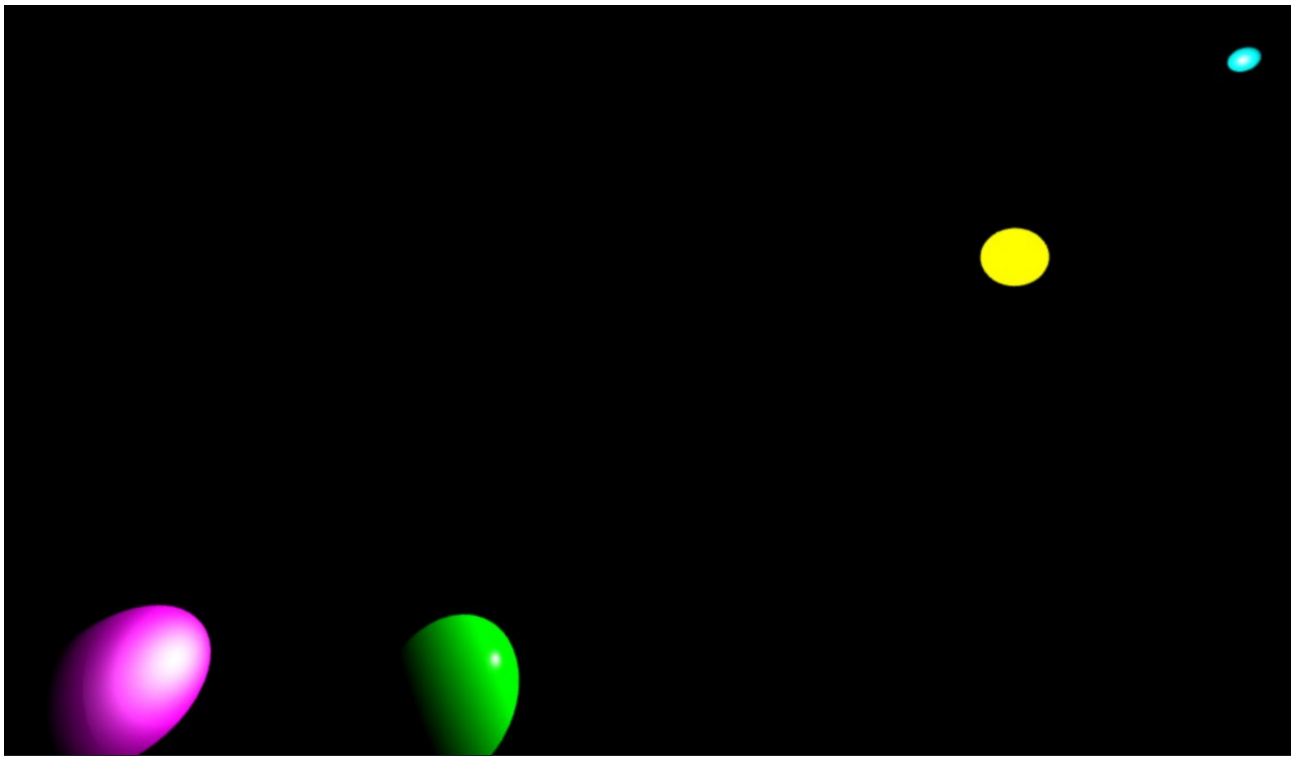

Рисунок 2.3 – Пример системы. Освещение - point-light (солнце).

ПРИЛОЖЕНИЕ А
Исходный код

A.1 index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import WebGLCanvas from './WebGLCanvas'

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(
  // <React.StrictMode>
  <WebGLCanvas style="width: 100%; height: 100%" />
  // </React.StrictMode>
);
```

Листинг А.1 – index.js

A.2 WebGLCanvas.js

```
import React, { useEffect, useRef } from 'react';
import dat from "dat-gui";
import Planet from './Planet'
import { get_camera_basis } from './Camera';
import { vertexShaderSource, fragmentShaderSource } from './shaders';

const { mat4, vec3 } = require('gl-matrix');

const WebGLCanvas = () => {
    const canvasRef = useRef(null);
    let animationFrameId = useRef(null);
    let previousTimestamp = useRef(0); // Store the previous timestamp
    let shaderProgram = null;
    let planets = [
        new Planet(10, 0, [1, 1, 0], 0, 10, 100, [
            new Planet(5, 19.6, [0, 1, 0], 0, 36, 200, [
                new Planet(1, 7, [1, 0, 1], Math.PI / 2, 1, 5)
            ]),
            new Planet(15, 45.6, [0, 1, 1], 0, 26, 10),
        ], true), //sun
    ];
    let gui = new dat.GUI();
    let world = {
        time_speed: 1,
        planet_subdivisions: 3,
        planet_draw: 'fill'
    }
```

```
    let camera = {
        perspective_fov: 1.5,
        position: {
            x: 0,
            y: 75,
            z: 0
        },
        rotation: {
            pitch: -Math.PI / 2, // тангаж (вверх вниз)
            yaw: 0, // рысканье (вправо влево)
            roll: 0 // крен
        },
        near: 0.1,
        far: 1000.0,
        torch_fov: 0.9
    }

    function setup_controls() {
        const step = 0.5
        document.addEventListener('keydown', function (event) {
            let [towards, right, up] = get_camera_basis(camera)
            vec3.scale(right, right, step)
            vec3.scale(towards, towards, step)
            if (event.code === 'KeyW') {
                camera.position.x += towards[0]
                camera.position.y += towards[1]
                camera.position.z += towards[2]
            }
            if (event.code === 'KeyA') {
                camera.position.x -= right[0]
                camera.position.y -= right[1]
                camera.position.z -= right[2]
            }
            if (event.code === 'KeyS') {
                camera.position.x -= towards[0]
                camera.position.y -= towards[1]
                camera.position.z -= towards[2]
            }
            if (event.code === 'KeyD') {
                camera.position.x += right[0]
                camera.position.y += right[1]
                camera.position.z += right[2]
            }

            if (event.code === 'ArrowLeft') {
                camera.rotation.yaw -= step
            }
            if (event.code === 'ArrowRight') {
```

```
                camera.rotation.yaw += step
            }
            if (event.code === 'ArrowUp') {
                camera.rotation.pitch += step
            }
            if (event.code === 'ArrowDown') {
                camera.rotation.pitch -= step
            }
            if (event.key === "e") {
                camera.rotation.roll += step
            }
            if (event.key === "q") {
                camera.rotation.roll -= step
            }

            if (event.shiftKey) {
                camera.position.y -= step
            }
            if (event.key === " ") {
                camera.position.y += step
            }
        });
    }

    function setup_gui() {
        gui.add(world, 'time_speed', 0, 10).listen()
        let planet_folder = gui.addFolder('Planet Settings')
        planet_folder.add(world, 'planet_subdivisions', 1, 5, 1.0)
        let camera_type = planet_folder.add(world, 'planet_draw', ['dots',
            'fill', 'lines']);
        camera_type.setValue("fill");
        let camera_folder = gui.addFolder('Camera')
        camera_folder.add(camera, 'perspective_fov', 0.1, Math.PI *
            0.8).listen();
        camera_folder.add(camera, 'near', 0.01, 10.0).listen();
        camera_folder.add(camera, 'far', 10.0, 1000.0).listen();
        camera_folder.add(camera, 'torch_fov', 0.05, 1.0).listen();
        let camera_translate_folder = camera_folder.addFolder('Translate')
        camera_translate_folder.add(camera.position, 'x', -100, 100,
            0.1).listen();
        camera_translate_folder.add(camera.position, 'y', -100, 100,
            0.1).listen();
        camera_translate_folder.add(camera.position, 'z', -100, 100,
            0.1).listen();
        let camera_orientation_folder =
            camera_folder.addFolder('Orientation')
        camera_orientation_folder.add(camera.rotation, 'pitch', -Math.PI /
            2, Math.PI / 2, 0.1).listen();
```

```
117         camera_orientation_folder.add(camera.rotation, 'yaw', -Math.PI,
                Math.PI, 0.1).listen();
118         camera_orientation_folder.add(camera.rotation, 'roll', -Math.PI /
                2, Math.PI / 2, 0.1).listen();
119     }
120
121     function init(canvas, gl) {
122         setup_gui()
123         // Set the canvas size
124         setup_controls()
125         const dimension = [document.documentElement.clientWidth,
                document.documentElement.clientHeight];
126
127         canvas.width = dimension[0];
128         canvas.height = dimension[1];
129
130         // Set clear color to black
131         gl.clearColor(0, 0, 0, 1);
132         gl.enable(gl.DEPTH_TEST)
133
134         // Clear <canvas>
135         gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT)
136
137         gl.enable(gl.BLEND);
138         gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA)
139         // // Create a buffer object to store circle data
140         // const circleBuffer = gl.createBuffer();
141
142         // Define the vertex shader
143         const vertexShader = gl.createShader(gl.VERTEX_SHADER);
144         gl.shaderSource(vertexShader, vertexShaderSource);
145         gl.compileShader(vertexShader);
146
147         // Define the fragment shader
148         const fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
149         gl.shaderSource(fragmentShader, fragmentShaderSource);
150         gl.compileShader(fragmentShader);
151
152         // Create a shader program and attach the shaders
153         shaderProgram = gl.createProgram();
154         gl.attachShader(shaderProgram, vertexShader);
155         gl.attachShader(shaderProgram, fragmentShader);
156         gl.linkProgram(shaderProgram);
157         gl.useProgram(shaderProgram);
158     }
159
160     function update(deltatime) {
161         for (const planet of planets) {
```

```
            planet.update(deltatime)
        }
    }

    function draw(gl, camera) {
        for (const planet of planets) {
            planet.draw(gl, shaderProgram, camera, world)
        }
    }

    useEffect(() => {
        const canvas = canvasRef.current;
        const gl = canvas.getContext('webgl');

        // Check if WebGL is supported
        if (!gl) {
            console.error('WebGL is not supported');
            return;
        }
        init(canvas, gl);
        let deltaTime;
        let currentTime;
        for (const planet of planets) {
            planet.update(1)
        }
        // Function to update the canvas
        const updateCanvas = (timestamp) => {
            // Clear the canvas
            gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT)

            // gui.updateDisplay();
            // Draw circles
            currentTime = timestamp;
            deltaTime = (currentTime - previousTimestamp.current) / 1000;
                // Convert to seconds
            previousTimestamp.current = currentTime;

            update(deltaTime * world.time_speed)
            draw(gl, camera)

            // Request the next frame
            animationFrameId.current = requestAnimationFrame(updateCanvas);
        };

        // Start the update loop
        updateCanvas(0);

        // Cleanup function to cancel animation frame on component unmount
```

```
209         return () => cancelAnimationFrame(animationFrameId.current);
210     }, []);
211
212     return <>
213         <canvas ref={canvasRef}
                width={document.documentElement.clientWidth}
                height={document.documentElement.clientHeight} />
214     </>;
215 };
216
217 export default WebGLCanvas;
```

Листинг А.2 – WebGLCanvas.js

A.3 shaders.js

```
1  export const vertexShaderSource = `
2              attribute vec3 aPosition;
3              attribute vec3 aColor;
4              attribute vec3 aNormal;
5
6              uniform mat4 uModel;
7              uniform mat4 uView;
8              uniform mat4 uProjection;
9              uniform mat4 u_worldInverseTranspose;
10             uniform vec3 u_lightWorldPosition;
11             uniform vec3 u_viewWorldPosition;
12             uniform vec3 u_lightDirection;
13             uniform float u_torch_fov;
14
15             varying vec3 vColor;
16             varying vec3 vNormal;
17             varying vec3 v_surfaceToLight;
18             varying vec3 v_surfaceToView;
19             varying vec3 v_lightDirection;
20             varying float v_torch_fov;
21
22             void main() {
23                 gl_Position = uProjection*uView*uModel*vec4(aPosition, 1.0);
24                 vColor = aColor;
25                 vNormal = mat3(u_worldInverseTranspose) * aNormal;
26
27                 vec3 surfaceWorldPosition = (uModel * vec4(aPosition,
                       1.0)).xyz;
28                 v_surfaceToLight = u_lightWorldPosition -
                       surfaceWorldPosition;
29                 v_surfaceToView = u_viewWorldPosition -
                       surfaceWorldPosition;
30                 v_lightDirection = u_lightDirection;
```

```
                v_torch_fov = u_torch_fov;
            }
        ';
export const fragmentShaderSource = `
            precision mediump float;

            uniform float u_shininess;

            varying vec3 vColor;
            varying vec3 vNormal;
            varying vec3 v_surfaceToLight;
            varying vec3 v_surfaceToView;
            varying vec3 v_lightDirection;
            varying float v_torch_fov;

            void main() {
                gl_FragColor = vec4(vColor, 1.0);

                vec3 normal = normalize(vNormal);
                vec3 surfaceToLightDirection = normalize(v_surfaceToLight);
                vec3 surfaceToViewDirection = normalize(v_surfaceToView);
                vec3 halfVector = normalize(surfaceToLightDirection +
                   surfaceToViewDirection);

                float ambientStrength = 0.1;
                vec3 ambient = ambientStrength * vColor;

                float light = dot(normal, surfaceToLightDirection);
                float specular = 0.0;
                if (light > 0.0) {
                    specular = pow(dot(normal, halfVector), u_shininess);
                }
                gl_FragColor = vec4(vColor, 1.0);
                gl_FragColor.rgb *= (ambient+light);
                gl_FragColor.rgb += specular;

                // vec3 normal = normalize(vNormal);

                // vec3 surfaceToLightDirection =
                   normalize(v_surfaceToLight);
                // vec3 surfaceToViewDirection = normalize(v_surfaceToView);
                // vec3 halfVector = normalize(surfaceToLightDirection +
                   surfaceToViewDirection);

                // float dotFromDirection =
                   dot(surfaceToLightDirection,-v_lightDirection);
                // float inLight = 0.0;
                // if(dotFromDirection>=v_torch_fov){
```

```
75                 //     inLight=dotFromDirection;
76                 // }
77                 // float light = inLight * dot(normal,
                      surfaceToLightDirection);
78                 // float specular = inLight * pow(dot(normal, halfVector),
                      u_shininess);
79
80                 // gl_FragColor = vec4(vColor, 1.0);
81
82                 // // Lets multiply just the color portion (not the alpha)
83                 // // by the light
84                 // gl_FragColor.rgb *= light;
85
86                 // // Just add in the specular
87                 // gl_FragColor.rgb += specular;
88                 // float ambientStrength = 0.2;
89                 // vec3 ambient = ambientStrength * vColor;
90                 // gl_FragColor.rgb += ambient;
91             }
92         `;
93
94
95 export const SunVertexShaderSource = `
96                 attribute vec3 aPosition;
97                 attribute vec3 aColor;
98
99                 uniform mat4 uModel;
100                uniform mat4 uView;
101                uniform mat4 uProjection;
102
103                varying vec3 vColor;
104
105                void main() {
106                    gl_Position = uProjection*uView*uModel*vec4(aPosition,
                          1.0);
107                    vColor = aColor;
108                }
109                `;
110 export const SunFragmentShaderSource = `
111                precision mediump float;
112
113                varying vec3 vColor;
114
115                void main() {
116                    gl_FragColor = vec4(vColor, 1.0);
117                }
118                `;
```

Листинг А.3 – shaders.js

A.4 Camera.js

```
const { mat4, vec3 } = require('gl-matrix');

export const get_camera_basis = (camera) => {
    const towardsVec = vec3.fromValues(
        Math.cos(camera.rotation.yaw) * Math.cos(camera.rotation.pitch),
        Math.sin(camera.rotation.pitch),
        Math.sin(camera.rotation.yaw) * Math.cos(camera.rotation.pitch)
    )
    const upVec = vec3.fromValues(0, 1, 0); // Define the up direction (typically [0, 1, 0])
    const rightVec = vec3.cross(vec3.create(), towardsVec, upVec);
    const UpVec_no_roll = vec3.cross(vec3.create(), rightVec, towardsVec);

    const rotationMatrix = mat4.create();
    mat4.fromRotation(rotationMatrix, camera.rotation.roll, towardsVec);

    const newUpVec = vec3.transformMat4(vec3.create(), UpVec_no_roll, rotationMatrix);

    const normalized_UP = vec3.normalize(vec3.create(), newUpVec);
    const normalized_RIGHT = vec3.normalize(vec3.create(), rightVec);
    const normalized_TOWARDS = vec3.normalize(vec3.create(), towardsVec);

    return [normalized_TOWARDS, normalized_RIGHT, normalized_UP]
}

export const getProjectionMatrix = (gl, camera) => {
    let projection = mat4.create()
    mat4.identity(projection);
    // console.log(gl.drawingBufferWidth, gl.drawingBufferHeight)
    mat4.perspective(projection, camera.perspective_fov, gl.drawingBufferWidth / gl.drawingBufferHeight, camera.near, camera.far)
    return projection
}

export const getViewMatrix = (camera) => {
    let u_view_matrix = mat4.create();
    const [towards, right, up] = get_camera_basis(camera)
    mat4.lookAt(u_view_matrix,
        [camera.position.x, camera.position.y, camera.position.z],
        [
            camera.position.x + towards[0],
            camera.position.y + towards[1],
            camera.position.z + towards[2]
        ],
```

```
        up)
    return u_view_matrix
}
```

Листинг А.4 – Camera.js

А.5 Planet.js

```
import { getProjectionMatrix, getViewMatrix } from './Camera';
import icomesh from 'icomesh';
import { SunFragmentShaderSource, SunVertexShaderSource,
   fragmentShaderSource, vertexShaderSource } from './shaders';
import { get_camera_basis } from './Camera';
const { mat4 } = require('gl-matrix');


class Planet {
    constructor(size, radius, color, inclination, orbitPeriod, shininess =
       100, satellites = [], isSun = false, centerX = 0, centerY = 0,
       centerZ = 0) {
        this.isSun = isSun
        this.size = size;
        this.radius = radius; // orbit radius
        this.color = color;
        this.inclination = inclination;
        this.x = 0;
        this.y = 0;
        this.z = 0;
        this.orbitPeriod = orbitPeriod;
        this.centerX = centerX;
        this.centerY = centerY;
        this.centerZ = centerZ;
        this.currentAngle = 0;
        this.satellites = satellites;
        this.shininess = shininess;
    }

    update(deltaTime) {
        const angleIncrement = (Math.PI * 2) / this.orbitPeriod * deltaTime;
        this.currentAngle += angleIncrement;
        if (this.currentAngle >= Math.PI * 2) {
            this.currentAngle -= Math.PI * 2
        }

        this.x = this.centerX + Math.cos(this.currentAngle) * this.radius;
        this.y = this.centerY + Math.sin(this.currentAngle) *
           Math.sin(this.inclination) * this.radius;
        this.z = this.centerZ + Math.sin(this.currentAngle) *
           Math.cos(this.inclination) * this.radius;
```

```
        for (const satellite of this.satellites) {
            satellite.centerX = this.x
            satellite.centerY = this.y
            satellite.centerZ = this.z
            satellite.update(deltaTime)
        }
    }

    loadShader(gl, vertex, fragment) {
        const vertexShader = gl.createShader(gl.VERTEX_SHADER);
        gl.shaderSource(vertexShader, vertex);
        gl.compileShader(vertexShader);

        // Define the fragment shader
        const fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
        gl.shaderSource(fragmentShader, fragment);
        gl.compileShader(fragmentShader);

        // Create a shader program and attach the shaders
        let shaderProgram = gl.createProgram();
        gl.attachShader(shaderProgram, vertexShader);
        gl.attachShader(shaderProgram, fragmentShader);
        gl.linkProgram(shaderProgram);
        gl.useProgram(shaderProgram);
        return shaderProgram
    }

    draw(gl, shaderProgram, camera, params) {
        for (const satellite of this.satellites) {
            satellite.draw(gl, shaderProgram, camera, params)
        }
        if (this.isSun) {
            shaderProgram = this.loadShader(gl, SunVertexShaderSource,
                SunFragmentShaderSource)
        } else {
            shaderProgram = this.loadShader(gl, vertexShaderSource,
                fragmentShaderSource)
        }
        // generate an icosphere with 4 subdivisions
        const { vertices, triangles } = icomesh(params.planet_subdivisions);
        const normals = vertices;
        this.verticesBuffer = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, this.verticesBuffer);
        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices),
            gl.STATIC_DRAW);
        const indexBuffer = gl.createBuffer();
        gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
        gl.bufferData(
```

```
            gl.ELEMENT_ARRAY_BUFFER,
            new Uint16Array(triangles),
            gl.STATIC_DRAW
        );

        // Create color buffer
        const colors = this.generateColors(vertices.length / 3);
        // const colors = this.generateHeightColors(vertices)
        this.colorsBuffer = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, this.colorsBuffer);
        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors),
            gl.STATIC_DRAW);

        // Bind vertex attributes
        const positionAttributeLocation =
            gl.getAttribLocation(shaderProgram, "aPosition");
        const colorAttributeLocation = gl.getAttribLocation(shaderProgram,
            "aColor");

        gl.bindBuffer(gl.ARRAY_BUFFER, this.verticesBuffer);
        gl.vertexAttribPointer(positionAttributeLocation, 3, gl.FLOAT,
            false, 0, 0);
        gl.enableVertexAttribArray(positionAttributeLocation);

        gl.bindBuffer(gl.ARRAY_BUFFER, this.colorsBuffer);
        gl.vertexAttribPointer(colorAttributeLocation, 3, gl.FLOAT, false,
            0, 0);
        gl.enableVertexAttribArray(colorAttributeLocation);

        const modelMatrix = mat4.create();

        mat4.translate(modelMatrix, modelMatrix, [this.x, this.y, this.z]);
        mat4.scale(modelMatrix, modelMatrix, [this.size, this.size,
            this.size])
        const modelUniformLocation = gl.getUniformLocation(shaderProgram,
            "uModel");
        gl.uniformMatrix4fv(modelUniformLocation, false, modelMatrix);

        const aViewLocation = gl.getUniformLocation(shaderProgram, "uView");
        const aProjectionLocation = gl.getUniformLocation(shaderProgram,
            "uProjection");

        gl.uniformMatrix4fv(aViewLocation, false, getViewMatrix(camera));
        gl.uniformMatrix4fv(aProjectionLocation, false,
            getProjectionMatrix(gl, camera));

        if (!this.isSun) {
            this.normalBuffer = gl.createBuffer();
```

```
121             gl.bindBuffer(gl.ARRAY_BUFFER, this.normalBuffer);
122             gl.bufferData(gl.ARRAY_BUFFER, normals, gl.STATIC_DRAW);
123             let normalLocation = gl.getAttribLocation(shaderProgram,
                    "aNormal");
124             gl.enableVertexAttribArray(normalLocation);
125             gl.bindBuffer(gl.ARRAY_BUFFER, this.normalBuffer);
126             gl.vertexAttribPointer(normalLocation, 3, gl.FLOAT, false, 0, 0)
127
128             ////
129             let lightWorldPositionLocation =
                    gl.getUniformLocation(shaderProgram, "u_lightWorldPosition");
130             // gl.uniform3fv(lightWorldPositionLocation,
                    [camera.position.x, camera.position.y, camera.position.z]);
131             gl.uniform3fv(lightWorldPositionLocation, [0, 0, 0]);
132
133
134             let lightDirectionLocation =
                    gl.getUniformLocation(shaderProgram, "u_lightDirection");
135             const [towards, right, up] = get_camera_basis(camera)
136             gl.uniform3fv(lightDirectionLocation, [
137                 towards[0],
138                 towards[1],
139                 towards[2]
140             ]);
141
142
143             let cameraWorldPositionLocation =
                    gl.getUniformLocation(shaderProgram, "u_viewWorldPosition");
144             gl.uniform3fv(cameraWorldPositionLocation, [camera.position.x,
                    camera.position.y, camera.position.z]);
145
146
147             let shininessLocation = gl.getUniformLocation(shaderProgram,
                    "u_shininess");
148             gl.uniform1f(shininessLocation, this.shininess);
149
150             let torch_fov_location = gl.getUniformLocation(shaderProgram,
                    "u_torch_fov");
151             // console.log(camera.torch_fov)
152             gl.uniform1f(torch_fov_location, camera.torch_fov);
153
154             let worldInverseMatrix = mat4.create()
155             mat4.invert(worldInverseMatrix, modelMatrix);
156             let worldInverseTransposeMatrix = mat4.create();
157             mat4.transpose(worldInverseTransposeMatrix, worldInverseMatrix);
158             // let worldViewProjectionLocation =
                    gl.getUniformLocation(shaderProgram,
                    "u_worldViewProjection");
```

```
159             let worldInverseTransposeLocation =
                    gl.getUniformLocation(shaderProgram,
                    "u_worldInverseTranspose");
160             // gl.uniformMatrix4fv(worldViewProjectionLocation, false,
                    worldViewProjectionMatrix);
161             gl.uniformMatrix4fv(worldInverseTransposeLocation, false,
                    worldInverseTransposeMatrix);
162         }
163         // Draw planet
164         if (params.planet_draw === 'dots') {
165             gl.drawArrays(gl.POINTS, 0, vertices.length / 3);
166         } else if (params.planet_draw === 'fill') {
167             gl.drawElements(gl.TRIANGLES, triangles.length,
                    gl.UNSIGNED_SHORT, 0);
168         } else if (params.planet_draw === 'lines') {
169             gl.drawElements(gl.LINES, triangles.length, gl.UNSIGNED_SHORT,
                    0);
170         }
171
172     }
173
174     generateSphereVertices() {
175         const latitudeBands = 64;
176         const longitudeBands = 64;
177
178         const vertices = [];
179
180         for (let latNumber = 0; latNumber <= latitudeBands; latNumber++) {
181             const theta = (latNumber * Math.PI) / latitudeBands;
182             const sinTheta = Math.sin(theta);
183             const cosTheta = Math.cos(theta);
184
185             for (let longNumber = 0; longNumber <= longitudeBands;
                    longNumber++) {
186                 const phi = (longNumber * 2 * Math.PI) / longitudeBands;
187                 const sinPhi = Math.sin(phi);
188                 const cosPhi = Math.cos(phi);
189
190                 const x = cosPhi * sinTheta;
191                 const y = cosTheta;
192                 const z = sinPhi * sinTheta;
193
194                 vertices.push(x * this.size);
195                 vertices.push(y * this.size);
196                 vertices.push(z * this.size);
197             }
198         }
199
```

```
        return vertices;
    }

    generateHeightColors(vertices) {
        const colors = [];
        for (let i = 0; i < vertices.length; i += 3) {
            colors.push(vertices[i + 0]);
            colors.push(vertices[i + 1]);
            colors.push(vertices[i + 2]);
        }
        return colors;
    }

    generateColors(numVertices) {
        const colors = [];
        for (let i = 0; i < numVertices; i++) {
            colors.push(this.color[0]);
            colors.push(this.color[1]);
            colors.push(this.color[2]);
        }
        return colors;
    }
}

export default Planet;
```

Листинг А.5 – Planet.js