МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ
по дисциплине «3D Компьютерная графика»
Тема: 3D освещение, Текстуры изображения, Карты теней

Студент гр. 5304 Лянгузов А.А.

Преподаватель Герасимова Т.В.

Санкт-Петербург
2019

Введение

Требования к работе:

- Добавить источник света;
- Добавить второй источник света;
- Отредактировать картинку и сделать ее текстурной картой;
- Разложить текстуру по изображению;
- Применить методы расчета теней.

Ход работы

Шаг 1. Полное переписывание исходного кода

В результате сложности добавления в существующую архитектуру приложения текстур и источников света, было принято решение переписать существующую функциональность с нуля. В более простом формате.

Шаг 2. Добавление источников света

В данном шаге на сцену были добавлен источник света, отбрасывающий тень. Положение задается параметрами PosX, PosY, PosZ в исходном коде. А направление света - параметрами targetX, targetY, targetZ. Для реализации освещения потребовалось:

Внести изменения в шейдеры:

```
<!-- vertex shader -->
<script id="3d-vertex-shader" type="x-shader/x-vertex">
      attribute vec4 a_position;
      attribute vec2 a_texcoord;
      attribute vec3 a_normal;

      uniform mat4 u_projection;
      uniform mat4 u_view;
      uniform mat4 u_world;
      uniform mat4 u_textureMatrix;

      varying vec2 v_texcoord;
      varying vec4 v_projectedTexcoord;
      varying vec3 v_normal;

      void main() {
           // Multiply the position by the matrix.
```

```
            vec4 worldPosition = u_world * a_position;

            gl_Position = u_projection * u_view * worldPosition;

            // Pass the texture coord to the fragment shader.
            v_texcoord = a_texcoord;

            v_projectedTexcoord = u_textureMatrix * worldPosition;

            // orient the normals and pass to the fragment shader
            v_normal = mat3(u_world) * a_normal;
      }
</script>
<!-- fragment shader -->
<script id="3d-fragment-shader" type="x-shader/x-fragment">
      precision mediump float;

      // Passed in from the vertex shader.
      varying vec2 v_texcoord;
      varying vec4 v_projectedTexcoord;
      varying vec3 v_normal;

      uniform vec4 u_colorMult;
      uniform sampler2D u_texture;
      uniform sampler2D u_projectedTexture;
      uniform float u_bias;
      uniform vec3 u_reverseLightDirection;

      void main() {
            // because v_normal is a varying it's interpolated
            // so it will not be a unit vector. Normalizing it
            // will make it a unit vector again
            vec3 normal = normalize(v_normal);

            float light = dot(normal, u_reverseLightDirection);

            vec3 projectedTexcoord = v_projectedTexcoord.xyz /
v_projectedTexcoord.w;
            float currentDepth = projectedTexcoord.z + u_bias;

            bool inRange =
            projectedTexcoord.x >= 0.0 &&
            projectedTexcoord.x <= 1.0 &&
```

```
            projectedTexcoord.y >= 0.0 &&
            projectedTexcoord.y <= 1.0;

            // the 'r' channel has the depth values
            float projectedDepth = texture2D(u_projectedTexture,
projectedTexcoord.xy).r;
            float shadowLight = (inRange && projectedDepth <=
currentDepth) ? 0.3 : 1.0;

            vec4 texColor = texture2D(u_texture, v_texcoord) *
u_colorMult;
            gl_FragColor = vec4(
            texColor.rgb * light * shadowLight,
            texColor.a);
      }
</script>
```

Также потребовалось задать матрицу проекции и настройки источника света в функции отображения сцены:

```
const lightWorldMatrix = m4.lookAt(
      [settings.posX, settings.posY, settings.posZ],// position
      [settings.targetX, settings.targetY, settings.targetZ], // target
      [0, 1, 0],// up
);
const lightProjectionMatrix = m4.orthographic(
      -settings.projWidth / 2,   // left
      settings.projWidth / 2,   // right
      -settings.projHeight / 2,  // bottom
      settings.projHeight / 2,  // top
      0.01,                      // near
      20);                       // far
```

Далее, заданные значения, были переданы в шейдеры:

```
webglUtils.setUniforms(programInfo, {
            u_view: viewMatrix,
            u_projection: projectionMatrix,
            u_bias: settings.bias,
            u_textureMatrix: textureMatrix,
            u_projectedTexture: depthTexture,
            u_reverseLightDirection: lightWorldMatrix.slice(8, 11),});
```

Шаг 3. Наложение текстур

Шейдеры уже включают в себя код, используемых для наложения текстур. Для удобства шейдеры текстур и шейдеры цвета были разделены.

```
const textureProgramInfo = webglUtils.createProgramInfo(gl,
['3d-vertex-shader', '3d-fragment-shader']);
      const colorProgramInfo = webglUtils.createProgramInfo(gl,
['color-vertex-shader', 'color-fragment-shader']);
```

Для наложения текстуры на объект была написана специальная функция:

```
function initTexture(url) {
      const texture = gl.createTexture();
      gl.bindTexture(gl.TEXTURE_2D, texture);
      const level = 0;
      const internalFormat = gl.RGBA;
      const width = 1;
      const height = 1;
      const border = 0;
      const srcFormat = gl.RGBA;
      const srcType = gl.UNSIGNED_BYTE;
      const pixel = new Uint8Array([255, 255, 255, 255]);
      gl.texImage2D(gl.TEXTURE_2D, level, internalFormat,
            width, height, border, srcFormat, srcType,
            pixel);

      const image = new Image();
      image.onload = function () {
            gl.bindTexture(gl.TEXTURE_2D, texture);
            gl.texImage2D(gl.TEXTURE_2D, level, internalFormat,
                  srcFormat, srcType, image);
            if (isPowerOf2(image.width) && isPowerOf2(image.height)) {
                  gl.generateMipmap(gl.TEXTURE_2D);
            } else {
                  // Размер не соответствует
степени 2.
                  // Отключаем MIP'ы и устанавливаем
натяжение по краям
                  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
gl.CLAMP_TO_EDGE);
                  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
gl.CLAMP_TO_EDGE);
```

```
                gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.LINEAR);
            }
        };
        image.src = url;
        return texture;
}
```

Далее, с использование мэтой функции были инциализированы все необходимые текстуры:

```
const woodTexture2 = initTexture('resources/textures/wood_2.jpg');
const blackWoodTexture =
                initTexture('resources/textures/black_wood_1.jpg');
const chessboardTexture =
               initTexture('resources/textures/chess_board_3.jpg');
```

Полученные таким образом переменные, используются для отрисовки фигур:

```
drawFigure(programInfo, planeBufferInfo, {
    u_colorMult: [1, 1, 1, 1],
    u_color: [1, 0, 0, 1],
    u_texture: chessboardTexture,
    u_world: m4.translation(0, 0, 0),
});

drawFigure(programInfo, cubeBufferInfo, {
    u_colorMult: [1, 1, 1, 1],
    u_color: [1, 0, 0, 1],
    u_texture: woodTexture2,
    u_world: m4.scale(m4.translation(0, -1.0, 0), 12.5, 1, 12.5),
});
```

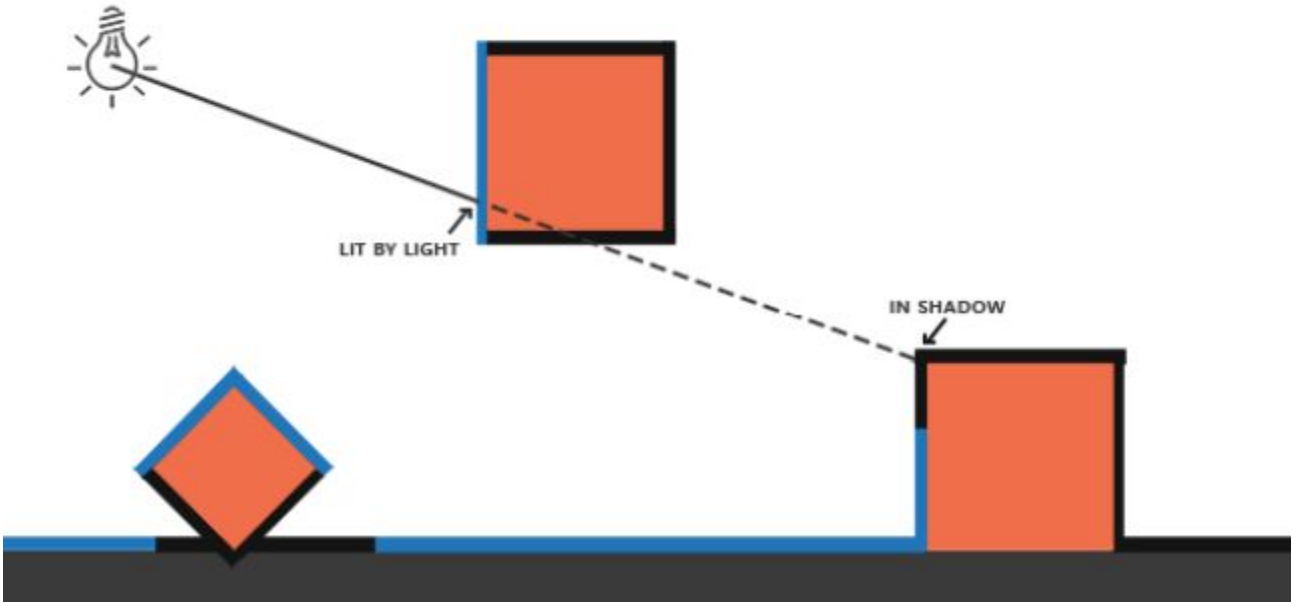
Сама отрисовка фигуры происходит следующим образом:

```
function drawFigure(program, figureBuffer, figureUniform, primitive) {
        webglUtils.setBuffersAndAttributes(gl, program, figureBuffer);

        // Set the uniforms unique to the cube
        webglUtils.setUniforms(program, figureUniform);

        // calls gl.drawArrays or gl.drawElements
        if (primitive) {
                webglUtils.drawBufferInfo(gl, figureBuffer, primitive);
        } else {
                webglUtils.drawBufferInfo(gl, figureBuffer);
        }
```

```
}
```

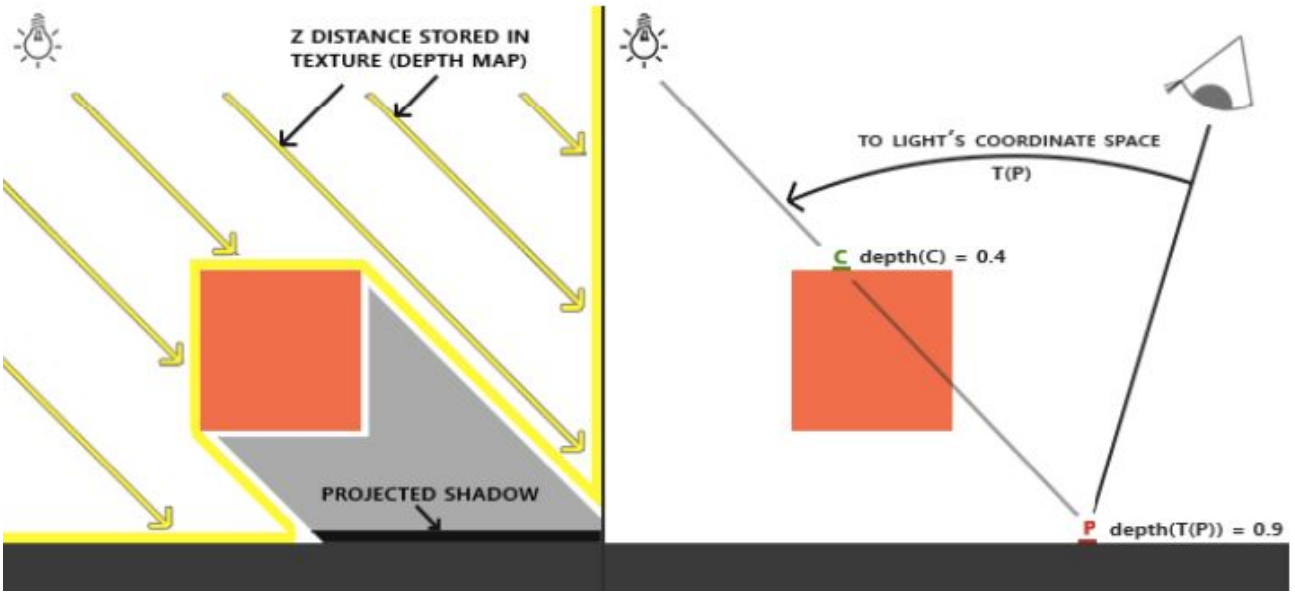
Шаг 4. Добавление теней

Идея, лежащая в основе карт теней, достаточно проста: мы рисуем сцену с точки зрения источника света. Всё, что мы видим, освещено, остальное — в тени. Представьте кусочек пола с большим кубом между ней и источником света. Так как источник света "видит" куб, а не кусочек пола, эта часть пола будет затенена.



На картинке выше синими линиями нарисованы поверхности, которые источник света может увидеть. Закрытые поверхности нарисованы чёрным — они будут нарисованы затенёнными. Если нарисовать линию (луч) от источника света вершине самого правого куба, то она сначала пересечёт висящий в воздухе кубик. Из-за этого левая поверхность висящего кубика освещена, в отличие от куба справа.

Мы хотим найти точку самого первого пересечения луча с поверхностью и сравнить её с остальными пересечениями. Если точка пересечения луча с поверхностью не совпадает с ближайшим пересечением, то она в тени. Повторение такой операции для тысяч различных лучей от источника будет крайне неэффективным и не подойдёт для рисования в каждом кадре игры. Значение в буфере глубины- это глубина фрагмента из точки зрения камеры, ограниченная значениями от 0 до 1. Если мы отрендерим сцену с точки зрения источника света и сохраним значения глубины в текстуру, то получим наименьшие значения глубины, которые видно с точки зрения источника света. Кроме того, значения глубины показывают поверхности, ближайшие для источника света. Такую текстуру называют картой глубины (depth map) или картой теней (shadow map).



На левой картинке показан направленный источник света (все лучи параллельны), отбрасывающий тень на поверхность ниже куба. С помощью значений глубины, сохранённых в текстуру, мы находим ближайшую к источнику поверхность и с её помощью определяем, что находится в тени. Мы создаём карту глубины с помощью рендеринга сцены, в качестве матриц вида и проекции используя матрицы, соответствующие нашему источнику света.
На картинке справа мы видим тот же самый свет, куб и наблюдателя. Мы рисуем фрагмент поверхности в точке P, и нам надо определить, находится ли он в тени. Для этого мы переводим P в координатное пространство источника света T(P). Так как точка P не видна из точки зрения света, её координата z в нашем примере будет 0.9. По координатам точки x, y мы можем заглянуть в карту глубины и узнать, что ближайшая к источнику света точка — C с глубиной 0.4 Это значение меньше, чем для точки P, поэтому точка P находится в тени.
Рисование теней состоит из двух проходов: сначала рисуем карту глубины, во втором проходе рисуем мир как обычно, с помощью карты глубины определяя, какие фрагменты поверхности находятся в тени.
Карта глубины — это текстура со значениями глубины, отрендеренная с точки зрения источника света. Мы потом будем использовать её для вычисления теней. Чтобы сохранить отрендеренный результат в текстуру, нам понадобится кадровый буфер (framebuffer). Шейдер используется для рендеринга в карту глубины.

Как уже было отмечено выше, тени представляют собой текстуру. Ее инициализация выглядит так:

```
// Init depth texture
const depthTextureSize = 512;
const depthTexture = createDepthTexture();
const depthFramebuffer = gl.createFramebuffer();
```

```
gl.bindFramebuffer(gl.FRAMEBUFFER, depthFramebuffer);
gl.framebufferTexture2D(
    gl.FRAMEBUFFER,        // target
    gl.DEPTH_ATTACHMENT,  // attachment point
    gl.TEXTURE_2D,         // texture target
    depthTexture,          // texture
    0);                    // mip level
```

Для создания текстуры была создана отдельная функция:

```
function createDepthTexture() {
      let texture = gl.createTexture();
      gl.bindTexture(gl.TEXTURE_2D, texture);
      gl.texImage2D(
            gl.TEXTURE_2D,   // target
            0,               // mip level
            gl.DEPTH_COMPONENT, // internal format
            depthTextureSize,   // width
            depthTextureSize,   // height
            0,               // border
            gl.DEPTH_COMPONENT, // format
            gl.UNSIGNED_INT, // type
            null);           // data
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
gl.NEAREST);
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.NEAREST);
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
gl.CLAMP_TO_EDGE);
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
gl.CLAMP_TO_EDGE);

      return texture;
}
```

Матрица камеры задается следующим образом:

```
const cameraPosition = [settings.cameraX, settings.cameraY, 15];
const target = [0, 0, 0];
const up = [0, 1, 0];
const cameraMatrix = m4.lookAt(cameraPosition, target, up);
```

И далее передается в отрисовку объектов на сцене в качестве параметра. С помощью этой матрицы формируется видовая матрица. Это происходит следующим образом:

```
// Make a view matrix from the camera matrix.
const viewMatrix = m4.inverse(cameraMatrix);
```

Значение же видовой матрицы, передается в шейдеры в качестве uniform-переменной:

```
webglUtils.setUniforms(programInfo, {
            u_view: viewMatrix,
            u_projection: projectionMatrix,
            u_bias: settings.bias,
            u_textureMatrix: textureMatrix,
            u_projectedTexture: depthTexture,
            u_reverseLightDirection: lightWorldMatrix.slice(8, 11),
});
```

Код шейдеров уже был представлен в разделе добавления источников света. Эти шейдеры поддерживают и добавление теней:

```
float shadowLight=(inRange && projectedDepth <= currentDepth)?0.3:1.0;
vec4 texColor = texture2D(u_texture, v_texcoord) * u_colorMult;
gl_FragColor = vec4(
   texColor.rgb * light * shadowLight,
   texColor.a);
```

Результат работы программы:

Исходная сцена

Рисунок 3 - Исходная сцена.

Перезагрузив, веб-страницу, получаем следующий результат.

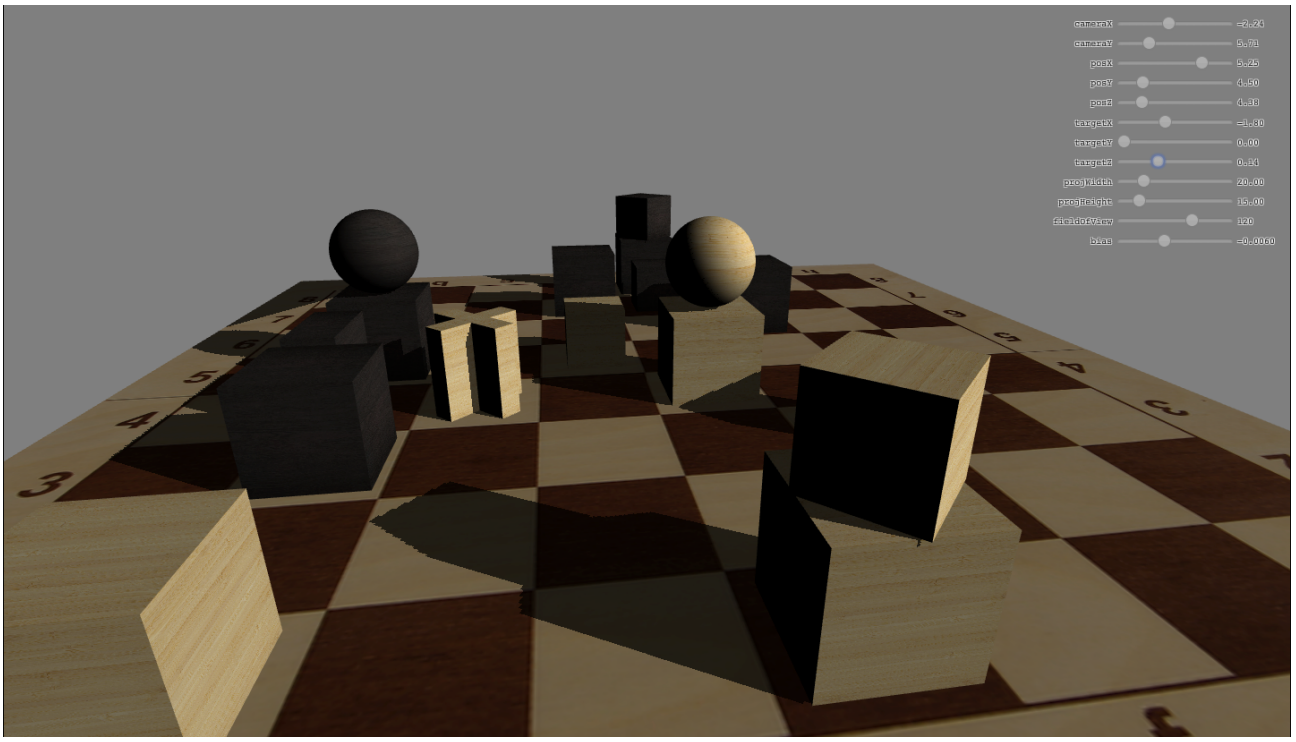


Рисунок 1 - Результат работы программы

Как можно видеть на скриншоте, на сцене присутствуют тени, источник света, а также на все объекты нанесены необходимые текстуры.

Выводы

В ходе лабораторной работы была полностью передела функиональность визуализации 3D-объектов и способ взаимодейсвтия со сценой. На сцену был добавлен источник света, на объекты наложены текстуры. Также на сцену были добавлены тени.

Исходный код

Файл app.html

```
<html xmlns="http://www.w3.org/1999/html">
<head>
	<meta charset="utf-8"/>
	<title>Great 3D scene</title>
	<meta content="text/html" http-equiv="content-type">
	<!-- vertex shader -->
	<script id="3d-vertex-shader" type="x-shader/x-vertex">
	attribute vec4 a_position;
	attribute vec2 a_texcoord;
	attribute vec3 a_normal;

	uniform mat4 u_projection;
	uniform mat4 u_view;
	uniform mat4 u_world;
	uniform mat4 u_textureMatrix;

	varying vec2 v_texcoord;
	varying vec4 v_projectedTexcoord;
	varying vec3 v_normal;

	void main() {
		// Multiply the position by the matrix.
		vec4 worldPosition = u_world * a_position;

		gl_Position = u_projection * u_view * worldPosition;

		// Pass the texture coord to the fragment shader.
		v_texcoord = a_texcoord;

		v_projectedTexcoord = u_textureMatrix * worldPosition;
```

```
            // orient the normals and pass to the fragment shader
            v_normal = mat3(u_world) * a_normal;
        }
        </script>
        <!-- fragment shader -->
        <script id="3d-fragment-shader" type="x-shader/x-fragment">
        precision mediump float;

        // Passed in from the vertex shader.
        varying vec2 v_texcoord;
        varying vec4 v_projectedTexcoord;
        varying vec3 v_normal;

        uniform vec4 u_colorMult;
        uniform sampler2D u_texture;
        uniform sampler2D u_projectedTexture;
        uniform float u_bias;
        uniform vec3 u_reverseLightDirection;

        void main() {
            // because v_normal is a varying it's interpolated
            // so it will not be a unit vector. Normalizing it
            // will make it a unit vector again
            vec3 normal = normalize(v_normal);

            float light = dot(normal, u_reverseLightDirection);

            vec3 projectedTexcoord = v_projectedTexcoord.xyz /
v_projectedTexcoord.w;
            float currentDepth = projectedTexcoord.z + u_bias;

            bool inRange =
            projectedTexcoord.x >= 0.0 &&
            projectedTexcoord.x <= 1.0 &&
            projectedTexcoord.y >= 0.0 &&
            projectedTexcoord.y <= 1.0;

            // the 'r' channel has the depth values
            float projectedDepth = texture2D(u_projectedTexture,
projectedTexcoord.xy).r;
            float shadowLight = (inRange && projectedDepth <=
currentDepth) ? 0.3 : 1.0;
```

```
            vec4 texColor = texture2D(u_texture, v_texcoord) *
u_colorMult;
            gl_FragColor = vec4(
            texColor.rgb * light * shadowLight,
            texColor.a);
      }
      </script>
      <!-- vertex shader -->
      <script id="color-vertex-shader" type="x-shader/x-vertex">
      attribute vec4 a_position;

      uniform mat4 u_projection;
      uniform mat4 u_view;
      uniform mat4 u_world;

      void main() {
            // Multiply the position by the matrices.
            gl_Position = u_projection * u_view * u_world * a_position;
      }
      </script>
      <!-- fragment shader -->
      <script id="color-fragment-shader" type="x-shader/x-fragment">
      precision mediump float;

      uniform vec4 u_color;
      void main() {
            gl_FragColor = u_color;
      }
      </script>
      <script src="./libs/webgl-lessons-ui.js"></script>
      <script src="./libs/webgl-utils.js"></script>
      <script src="./libs/m4.js"></script>
      <script src="./libs/primitives.js"></script>
      <script src="./utils.js" type="text/javascript"></script>
      <script src="./my_figures.js" type="text/javascript"></script>
      <script src="./app.js" type="text/javascript"></script>
      <link rel="stylesheet" href="./app.css">
</head>
<body onload="main()">
      <canvas id="canvas"></canvas>
      <div id="uiContainer">
      <div id="ui"></div>
      </div>
```

```
</body>
</html>
```

Файл app.js

```
'use strict';

function main() {
      // Get A WebGL context
      /** @type {HTMLCanvasElement} */
      const canvas = document.getElementById('canvas');
      if(canvas == null) return;

      const gl = canvas.getContext('webgl');
      if (!gl) return;

      const ext = gl.getExtension('WEBGL_depth_texture');
      if (!ext) {
      return alert('need WEBGL_depth_texture');  // eslint-disable-line
      }

      // setup GLSL programs
      const textureProgramInfo = webglUtils.createProgramInfo(gl,
['3d-vertex-shader', '3d-fragment-shader']);
      const colorProgramInfo = webglUtils.createProgramInfo(gl,
['color-vertex-shader', 'color-fragment-shader']);

      const planeBufferInfo = primitives.createPlaneBufferInfo(
      gl,
      25,  // width
      25,  // height
      1,   // subdivisions across
      1,   // subdivisions down
      );
      const cubeBufferInfo = primitives.createCubeBufferInfo(
      gl,
      2,  // size
      );

      const sphereBufferInfo = primitives.createSphereBufferInfo(
      gl,
      1,
      100,
```

```
      100,
      degToRad(0),
      degToRad(360),
      degToRad(0),
      degToRad(360)
      );

      //
------------------------------TEXTURES-----------------------------------
-----
      //Init my textures
      const woodTexture1 = initTexture('resources/textures/wood_1.jpg');
      const woodTexture2 = initTexture('resources/textures/wood_2.jpg');
      const blackWoodTexture =
initTexture('resources/textures/black_wood_1.jpg');
      // Init a 8x8 checkerboard texture
      const chessboardTexture =
initTexture('resources/textures/chess_board_3.jpg');
      // Init depth texture
      const depthTextureSize = 512;
      const depthTexture = createDepthTexture();
      const depthFramebuffer = gl.createFramebuffer();
      gl.bindFramebuffer(gl.FRAMEBUFFER, depthFramebuffer);
      gl.framebufferTexture2D(
      gl.FRAMEBUFFER,       // target
      gl.DEPTH_ATTACHMENT,  // attachment point
      gl.TEXTURE_2D,        // texture target
      depthTexture,         // texture
      0);                   // mip level

      //
------------------------------UI-------------------------------------
      const settings = {
      cameraX: -0.26,//-9.5,
      cameraY: 20,//12,
      posX: 1.0,
      posY: 4.8,
      posZ: 5.72,
      targetX: 3.5,
      targetY: 0,
      targetZ: 5.0,
      projWidth: 20,
      projHeight: 15,
```

```
      fieldOfView: 120,
      bias: -0.006,
      };
      webglLessonsUI.setupUI(document.querySelector('#ui'), settings, [
      {type: 'slider', key: 'cameraX', min: -20, max: 20, change:
render, precision: 2, step: 0.001,},
      {type: 'slider', key: 'cameraY', min: 1, max: 20, change: render,
precision: 2, step: 0.001,},
      {type: 'slider', key: 'posX', min: -10, max: 10, change: render,
precision: 2, step: 0.001,},
      {type: 'slider', key: 'posY', min: 1, max: 20, change: render,
precision: 2, step: 0.001,},
      {type: 'slider', key: 'posZ', min: 1, max: 20, change: render,
precision: 2, step: 0.001,},
      {type: 'slider', key: 'targetX', min: -10, max: 10, change:
render, precision: 2, step: 0.001,},
      {type: 'slider', key: 'targetY', min: 0, max: 20, change: render,
precision: 2, step: 0.001,},
      {type: 'slider', key: 'targetZ', min: -10, max: 20, change:
render, precision: 2, step: 0.001,},
      {type: 'slider', key: 'projWidth', min: 0, max: 100, change:
render, precision: 2, step: 0.001,},
      {type: 'slider', key: 'projHeight', min: 0, max: 100, change:
render, precision: 2, step: 0.001,},
      {type: 'slider', key: 'fieldOfView', min: 1, max: 179, change:
render,},
      {type: 'slider', key: 'bias', min: -0.01, max: 0.00001, change:
render, precision: 4, step: 0.0001,},
      ]);

      const fieldOfViewRadians = degToRad(60);

      function draw3DObjects(
      projectionMatrix,
      cameraMatrix,
      textureMatrix,
      lightWorldMatrix,
      programInfo) {
      // Make a view matrix from the camera matrix.
      const viewMatrix = m4.inverse(cameraMatrix);

      gl.useProgram(programInfo.program);
```

```
	// set uniforms that are the same for both the sphere and plane
	// note: any values with no corresponding uniform in the shader
	// are ignored.
	webglUtils.setUniforms(programInfo, {
		u_view: viewMatrix,
		u_projection: projectionMatrix,
		u_bias: settings.bias,
		u_textureMatrix: textureMatrix,
		u_projectedTexture: depthTexture,
		u_reverseLightDirection: lightWorldMatrix.slice(8, 11),
	});

	// ------ Draw the plane --------
	drawFigure(programInfo, planeBufferInfo,
	{
		u_colorMult: [1, 1, 1, 1],
		u_color: [1, 0, 0, 1],
		u_texture: chessboardTexture,
		u_world: m4.translation(0, 0, 0),
	});
	drawFigure(programInfo, cubeBufferInfo, {
		u_colorMult: [1, 1, 1, 1],
		u_color: [1, 0, 0, 1],
		u_texture: woodTexture2,
		u_world: m4.scale(m4.translation(0, -1.0, 0), 12.5, 1,
12.5),
	});

	// ------ Draw the cubes --------
	drawFigure(programInfo, cubeBufferInfo, {
		u_colorMult: [1, 1, 1, 1],
		u_color: [1, 0, 0, 1],
		u_texture: woodTexture2,
		u_world: m4.scale(m4.translation(1.25, 1, 9.25), 1, 1, 1),
	});

	drawFigure(programInfo, cubeBufferInfo, {
		u_colorMult: [1, 1, 1, 1],
		u_color: [1, 0, 0, 1],
		u_texture: woodTexture2,
		u_world: m4.axisRotate(m4.scale(m4.translation(1.25, 2.75,
9.25), 0.75, 0.75, 0.75), [0, 1, 0], degToRad(45))
	});
```

```
      drawFigure(programInfo, cubeBufferInfo, {
            u_colorMult: [1, 1, 1, 1],
            u_color: [1, 0, 0, 1],
            u_texture: woodTexture2,
            u_world: m4.scale(m4.translation(-6.75, 1, 9.25), 1, 1, 1),
      });

      drawFigure(programInfo, cubeBufferInfo, {
            u_colorMult: [1, 1, 1, 1],
            u_color: [1, 0, 0, 1],
            u_texture: woodTexture2,
            u_world: m4.scale(m4.translation(-1.25, 0.75, -1.25), 0.75,
0.75, 0.75)
      });

      drawFigure(programInfo, cubeBufferInfo, {
            u_colorMult: [1, 1, 1, 1],
            u_color: [1, 0, 0, 1],
            u_texture: blackWoodTexture,
            u_world: m4.scale(m4.translation(-1.25, 1, -6.5), 1, 1, 1),
      });

      drawFigure(programInfo, cubeBufferInfo, {
            u_colorMult: [1, 1, 1, 1],
            u_color: [1, 0, 0, 1],
            u_texture: blackWoodTexture,
            u_world: m4.scale(m4.translation(1.25, 0.75, -6.5), 0.75,
0.75, 0.75),
      });

      drawFigure(programInfo, cubeBufferInfo, {
            u_colorMult: [1, 1, 1, 1],
            u_color: [1, 0, 0, 1],
            u_texture: blackWoodTexture,
            u_world: m4.scale(m4.translation(1.25, 1, -9.25), 1, 1, 1),
      });

      drawFigure(programInfo, cubeBufferInfo, {
            u_colorMult: [1, 1, 1, 1],
            u_color: [1, 0, 0, 1],
            u_texture: blackWoodTexture,
            u_world: m4.axisRotate(m4.scale(m4.translation(1.25, 2.75,
```

```
-9.25), 0.75, 0.75, 0.75), [0, 1, 0], degToRad(45))
      });

      drawFigure(programInfo, cubeBufferInfo, {
            u_colorMult: [1, 1, 1, 1],
            u_color: [1, 0, 0, 1],
            u_texture: woodTexture2,
            u_world: m4.scale(m4.translation(1.25, 1, 1.25), 1, 1, 1)
      });

      drawFigure(programInfo, cubeBufferInfo, {
            u_colorMult: [1, 1, 1, 1],
            u_color: [1, 0, 0, 1],
            u_texture: woodTexture2,
            u_world: m4.axisRotate(m4.scale(m4.translation(3.75, 0.75,
3.75), 0.75, 0.75, 0.75), [0, 1, 0], degToRad(45))
      });

      drawFigure(programInfo, cubeBufferInfo, {
            u_colorMult: [1, 1, 1, 1],
            u_color: [1, 0, 0, 1],
            u_texture: blackWoodTexture,
            u_world: m4.scale(m4.translation(3.75, 1, -3.75), 1, 1, 1)
      });

      drawFigure(programInfo, cubeBufferInfo, {
            u_colorMult: [1, 1, 1, 1],
            u_color: [1, 0, 0, 1],
            u_texture: blackWoodTexture,
            u_world: m4.scale(m4.translation(-6.5, 1, 4), 1, 1, 1)
      });

      drawFigure(programInfo, cubeBufferInfo, {
            u_colorMult: [1, 1, 1, 1],
            u_color: [1, 0, 0, 1],
            u_texture: blackWoodTexture,
            u_world: m4.scale(m4.translation(-7.0, 1, 1.5), 0.5, 1, 1)
      });

      drawFigure(programInfo, cubeBufferInfo, {
            u_colorMult: [1, 1, 1, 1],
            u_color: [1, 0, 0, 1],
            u_texture: blackWoodTexture,
```

```
            u_world: m4.scale(m4.translation(-6.5, 1, -1.25), 1, 1, 1)
      });

      drawFigure(programInfo, cubeBufferInfo, {
            u_colorMult: [1, 1, 1, 1],
            u_color: [1, 0, 0, 1],
            u_texture: woodTexture2,
            u_world: m4.scale(m4.axisRotate(m4.translation(-4.0, 1,
1.5), [0, 1, 0], degToRad(45)), 0.25, 1, 1)
      });

      drawFigure(programInfo, cubeBufferInfo, {
            u_colorMult: [1, 1, 1, 1],
            u_color: [1, 0, 0, 1],
            u_texture: woodTexture2,
            u_world: m4.scale(m4.axisRotate(m4.translation(-4.0, 1,
1.5), [0, 1, 0], degToRad(-45)), 0.25, 1, 1)
      });

      drawFigure(programInfo, sphereBufferInfo, {
            u_colorMult: [1, 1, 1, 1],
            u_color: [1, 0, 0, 1],
            u_texture: woodTexture2,
            u_world: m4.scale(m4.translation(1.25, 3, 1.25), 1, 1, 1)
      });

      drawFigure(programInfo, sphereBufferInfo, {
            u_colorMult: [1, 1, 1, 1],
            u_color: [1, 0, 0, 1],
            u_texture: blackWoodTexture,
            u_world: m4.scale(m4.translation(-6.5, 3, -1.25), 1, 1, 1)
      });
      }

      function drawFigure(program, figureBuffer, figureUniform,
primitive) {
      webglUtils.setBuffersAndAttributes(gl, program, figureBuffer);

      // Set the uniforms unique to the cube
      webglUtils.setUniforms(program, figureUniform);

      // calls gl.drawArrays or gl.drawElements
      if (primitive) {
```

```
            webglUtils.drawBufferInfo(gl, figureBuffer, primitive);
        } else {
            webglUtils.drawBufferInfo(gl, figureBuffer);
        }
        }

        // Draw the scene.
        function render() {
        webglUtils.resizeCanvasToDisplaySize(gl.canvas);

        gl.enable(gl.CULL_FACE);
        gl.enable(gl.DEPTH_TEST);

        // first draw from the POV of the light
        const lightWorldMatrix = m4.lookAt(
            [settings.posX, settings.posY, settings.posZ],         //
position
            [settings.targetX, settings.targetY, settings.targetZ], //
target
            [0, 1, 0],                                             // up
        );
        const lightProjectionMatrix = m4.orthographic(
            -settings.projWidth / 2,   // left
            settings.projWidth / 2,    // right
            -settings.projHeight / 2,  // bottom
            settings.projHeight / 2,  // top
            0.01,                      // near
            20);                       // far

        // draw to the depth texture
        gl.bindFramebuffer(gl.FRAMEBUFFER, depthFramebuffer);
        gl.viewport(0, 0, depthTextureSize, depthTextureSize);
        gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

        // Depth rendering
        draw3DObjects(
            lightProjectionMatrix,
            lightWorldMatrix,
            m4.identity(),
            lightWorldMatrix,
            colorProgramInfo);

        // now draw scene to the canvas projecting the depth texture into
```

```
the scene
      gl.bindFramebuffer(gl.FRAMEBUFFER, null);
      gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);
      gl.clearColor(0, 0, 0, 0.5);
      gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

      let textureMatrix = m4.identity();
      textureMatrix = m4.translate(textureMatrix, 0.5, 0.5, 0.5);
      textureMatrix = m4.scale(textureMatrix, 0.5, 0.5, 0.5);
      textureMatrix = m4.multiply(textureMatrix, lightProjectionMatrix);
      // use the inverse of this world matrix to make
      // a matrix that will transform other positions
      // to be relative this this world space.
      textureMatrix = m4.multiply(
            textureMatrix,
            m4.inverse(lightWorldMatrix));

      // Compute the projection matrix
      const aspect = gl.canvas.clientWidth / gl.canvas.clientHeight;
      const projectionMatrix =
            m4.perspective(fieldOfViewRadians, aspect, 1, 2000);

      // Compute the camera's matrix using look at.
      const cameraPosition = [settings.cameraX, settings.cameraY, 15];
      const target = [0, 0, 0];
      const up = [0, 1, 0];
      const cameraMatrix = m4.lookAt(cameraPosition, target, up);

      // Scene visualisation
      draw3DObjects(
            projectionMatrix,
            cameraMatrix,
            textureMatrix,
            lightWorldMatrix,
            textureProgramInfo);
      }

      function initTexture(url) {
      const texture = gl.createTexture();
      gl.bindTexture(gl.TEXTURE_2D, texture);
      const level = 0;
      const internalFormat = gl.RGBA;
      const width = 1;
```

```
      const height = 1;
      const border = 0;
      const srcFormat = gl.RGBA;
      const srcType = gl.UNSIGNED_BYTE;
      const pixel = new Uint8Array([255, 255, 255, 255]);
      gl.texImage2D(gl.TEXTURE_2D, level, internalFormat,
            width, height, border, srcFormat, srcType,
            pixel);

      const image = new Image();
      image.onload = function () {
            gl.bindTexture(gl.TEXTURE_2D, texture);
            gl.texImage2D(gl.TEXTURE_2D, level, internalFormat,
                  srcFormat, srcType, image);
            if (isPowerOf2(image.width) && isPowerOf2(image.height)) {
                  gl.generateMipmap(gl.TEXTURE_2D);
            } else {
                  // Размер не соответствует
степени 2.
                  // Отключаем MIP'ы и устанавливаем
натяжение по краям
                  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
gl.CLAMP_TO_EDGE);
                  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
gl.CLAMP_TO_EDGE);
                  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.LINEAR);
            }
      };
      image.src = url;
      return texture;
      }

      function createChessBoardTexture() {
      let texture = gl.createTexture();
      gl.bindTexture(gl.TEXTURE_2D, texture);
      gl.texImage2D(
            gl.TEXTURE_2D,
            0,                // mip level
            gl.LUMINANCE,     // internal format
            8,                // width
            8,                // height
            0,                // border
```

```
            gl.LUMINANCE,    // format
            gl.UNSIGNED_BYTE, // type
            new Uint8Array([  // data
                 0xFF, 0x00, 0xFF, 0x00, 0xFF, 0x00, 0xFF, 0x00,
                 0x00, 0xFF, 0x00, 0xFF, 0x00, 0xFF, 0x00, 0xFF,
                 0xFF, 0x00, 0xFF, 0x00, 0xFF, 0x00, 0xFF, 0x00,
                 0x00, 0xFF, 0x00, 0xFF, 0x00, 0xFF, 0x00, 0xFF,
                 0xFF, 0x00, 0xFF, 0x00, 0xFF, 0x00, 0xFF, 0x00,
                 0x00, 0xFF, 0x00, 0xFF, 0x00, 0xFF, 0x00, 0xFF,
                 0xFF, 0x00, 0xFF, 0x00, 0xFF, 0x00, 0xFF, 0x00,
                 0x00, 0xFF, 0x00, 0xFF, 0x00, 0xFF, 0x00, 0xFF,
            ]));
      gl.generateMipmap(gl.TEXTURE_2D);
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
gl.NEAREST);
      return texture;
      }

      function createDepthTexture() {
      let texture = gl.createTexture();
      gl.bindTexture(gl.TEXTURE_2D, texture);
      gl.texImage2D(
            gl.TEXTURE_2D,   // target
            0,               // mip level
            gl.DEPTH_COMPONENT, // internal format
            depthTextureSize,   // width
            depthTextureSize,   // height
            0,               // border
            gl.DEPTH_COMPONENT, // format
            gl.UNSIGNED_INT, // type
            null);           // data
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
gl.NEAREST);
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.NEAREST);
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
gl.CLAMP_TO_EDGE);
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
gl.CLAMP_TO_EDGE);

      return texture;
      }
```

```
        render();

        setTimeout(() => {
        render();
        }, 100);
}
```