

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

Отчет
по лабораторной работе №5
по дисциплине «КОМПЬЮТЕРНАЯ 3D-ГРАФИКА»
Тема: Карты теней

Студентка гр. 5303

Допира В.Е.

Преподаватель

Герасимова Т.В.

Санкт-Петербург

2019

Цель работы: применить методы расчета теней к 4 лабораторной работе. В итоге выполнения пяти лабораторных работ полученный результата должен максимально приближаться к изображению, полученному в начале семестра.

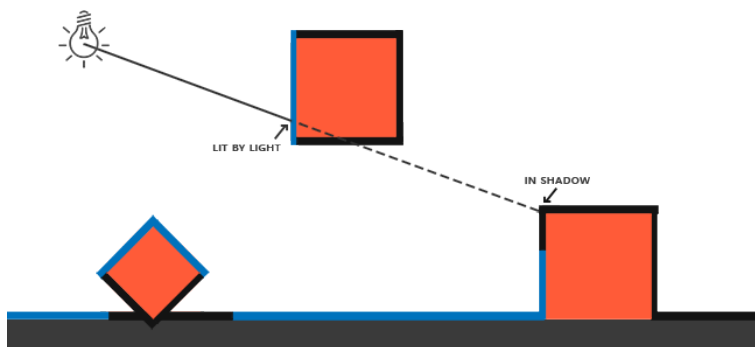
Ход работы

1. Создан шейдер, специфичный для отображения карты теней. Он использует заданную карту теней (shadowmap).

```
var shader = THREE.ShaderChunk.shadowmap_pars_fragment;  
shader = shader.replace(  
    "#ifdef USE_SHADOWMAP",  
    "#ifdef USE_SHADOWMAP" +  
document.getElementById("PCSS").textContent  
);  
THREE.ShaderChunk.shadowmap_pars_fragment = shader;
```

2. На javascript определены необходимые буферы и матрицы перспективы.

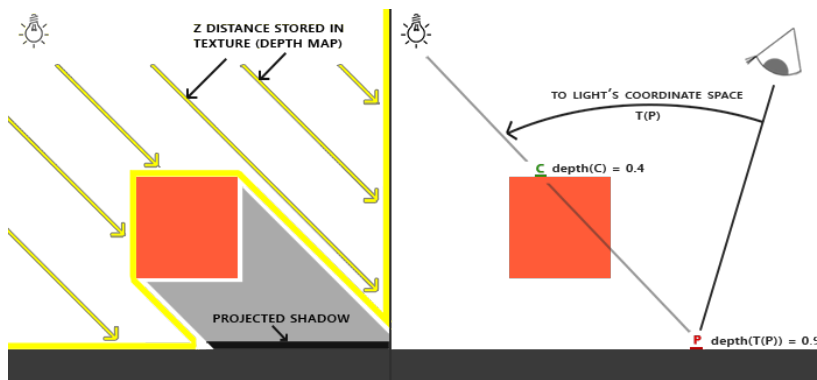
Идея, лежащая в основе карт теней, достаточно проста: мы рисуем сцену с точки зрения источника света. Всё, что мы видим, освещено, остальное — в тени. Представьте кусочек пола с большим кубом между ней и источником света. Так как источник света "видит" куб, а не кусочек пола, эта часть пола будет затенена.



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.

Ниже заданы значения расстояний для источника света.

```
#define LIGHT_WORLD_SIZE 0.005
#define LIGHT_FRUSTUM_WIDTH 3.75
#define LIGHT_SIZE_UV (LIGHT_WORLD_SIZE / LIGHT_FRUSTUM_WIDTH)
#define NEAR_PLANE 9.5

#define NUM_SAMPLES 17
#define NUM_RINGS 11
#define BLOCKER_SEARCH_NUM_SAMPLES NUM_SAMPLES
#define PCF_NUM_SAMPLES NUM_SAMPLES

vec2 poissonDisk[NUM_SAMPLES];

3. Определены плоскости и углы до источника:

void initPoissonSamples( const in vec2 randomSeed ) {
    float ANGLE_STEP = PI2 * float( NUM_RINGS ) /
    float( NUM_SAMPLES );
    float INV_NUM_SAMPLES = 1.0 / float( NUM_SAMPLES );
    float angle = rand( randomSeed ) * PI2;
    float radius = INV_NUM_SAMPLES;
    float radiusStep = radius;

    for( int i = 0; i < NUM_SAMPLES; i ++ ) {
        poissonDisk[i] = vec2( cos( angle ), sin( angle ) ) *
        pow( radius, 0.75 );
        radius += radiusStep;
        angle += ANGLE_STEP;
    }
}

float penumbraSize( const in float zReceiver, const in float
zBlocker ) {
    return (zReceiver - zBlocker) / zBlocker;}
```

4. Заданы треугольники для вычисления области карты теней. В результате будет заполнен буфер глубины, для каждого пикселя содержащий глубину ближайщего фрагмента с точки зрения света.

```
float findBlocker( sampler2D shadowMap, const in vec2 uv, const in
float zReceiver ) {
    float searchRadius = LIGHT_SIZE_UV * ( zReceiver - NEAR_PLANE
) / zReceiver;
    float blockerDepthSum = 0.0;
    int numBlockers = 0;

    for( int i = 0; i < BLOCKER_SEARCH_NUM_SAMPLES; i++ ) {
        float
            shadowMapDepth
        =
        unpackRGBAToDepth(texture2D(shadowMap, uv + poissonDisk[i] *
searchRadius));
        if ( shadowMapDepth < zReceiver ) {
            blockerDepthSum += shadowMapDepth;
            numBlockers ++;
        }
    }

    if( numBlockers == 0 ) return -1.0;
    return blockerDepthSum / float( numBlockers );
}
```

5. Реализация PCF (Percentage Closer Filtering)

Этот метод используется для сглаживания теней, чтобы изображение было более реалистичным. Его идея состоит в выборке из карты теней вокруг текущего пикселя и сравнении его глубины с этой выборкой. Получив среднее значение можно добиться гладкого перехода между светом и тенью.

Каждая ячейка матрицы хранит значение глубины для каждого пикселя относительно позиции источника света. Тогда задается граничное значение такое, что, если значение в карте теней меньше него, то оно считается тенью. А если значение больше либо равно, то объект расположен на свету. Это дает грубую границу между светом и тенью. Если рассмотреть значения близкие к границе и вычислить их среднее, то получится значение, которое поможет сгладить границу между светом и тенью.

Сначала производится деление перспективы на координаты пространства клиппера с позиции источника света, а затем преобразования отрезка. Далее поиск смещения в пространстве координат текстуры между 2 соседними

пикселями. В цикле for вычисляется вектор смещения для каждого пикселя. Получен теневой коэффициент.

Функция принимает на вход созданную в работе карту теней и возвращает границы теней.

```
float PCF_Filter(sampler2D shadowMap, vec2 uv, float zReceiver,
float filterRadius ) {
    float sum = 0.0;
    for( int i = 0; i < PCF_NUM_SAMPLES; i ++ ) {
        float depth = unpackRGBAToDepth( texture2D( shadowMap,
uv + poissonDisk[ i ] * filterRadius ) );
        if( zReceiver <= depth ) sum += 1.0;
    }
    for( int i = 0; i < PCF_NUM_SAMPLES; i ++ ) {
        float depth = unpackRGBAToDepth( texture2D( shadowMap,
uv + -poissonDisk[ i ].yx * filterRadius ) );
        if( zReceiver <= depth ) sum += 1.0;
    }
    return sum / ( 2.0 * float( PCF_NUM_SAMPLES ) );
}
```

6. Применение фильтрации, вычисление карты полутеней:

```
float PCSS ( sampler2D shadowMap, vec4 coords ) {
    vec2 uv = coords.xy;
    float zReceiver = coords.z;
    initPoissonSamples( uv );
    float avgBlockerDepth = findBlocker(shadowMap,uv,zReceiver);
    if( avgBlockerDepth == -1.0 ) return 1.0;

    float penumbraRatio = penumbraSize( zReceiver,
avgBlockerDepth);
    float filterRadius = penumbraRatio * LIGHT_SIZE_UV *
NEAR_PLANE / zReceiver;

    return PCF_Filter( shadowMap, uv, zReceiver, filterRadius );
}
```

7. Рендер сцены с картой теней:

```
renderer.outputEncoding = THREE.sRGBEncoding;
renderer.shadowMapEnabled = true;
```

Результат выполнения программы показан на рисунке 1.

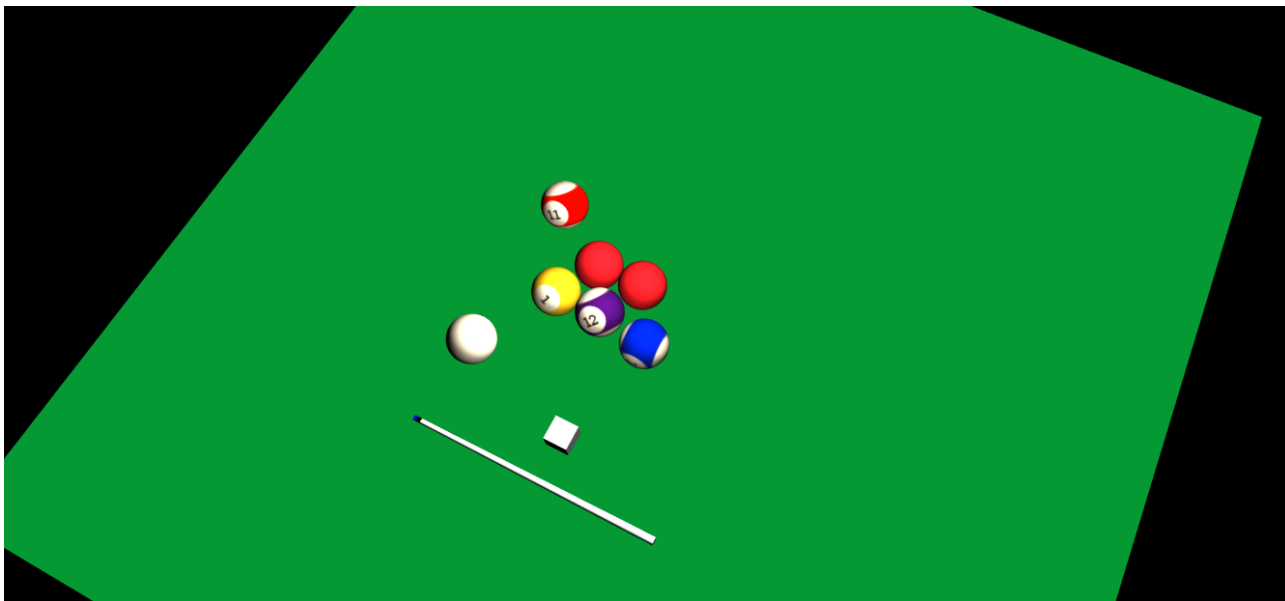


Рисунок 1 — Результат работы программы

Вывод

В ходе лабораторной работы была представлена 3D сцена и объекты куб и шар. Освоены матричные преобразования над объектами.

Приложение 1. Листинг программы

```
<!doctype html>
<html lang="en">
<head>
    <title>WebGL</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, user-
scalable=no, minimum-scale=1.0, maximum-scale=1.0">

    <script type="x-shader/x-fragment" id="PCSS">
        #define LIGHT_WORLD_SIZE 0.005
        #define LIGHT_FRUSTUM_WIDTH 3.75
        #define LIGHT_SIZE_UV (LIGHT_WORLD_SIZE /
LIGHT_FRUSTUM_WIDTH)
        #define NEAR_PLANE 9.5

        #define NUM_SAMPLES 17
        #define NUM_RINGS 11
        #define BLOCKER_SEARCH_NUM_SAMPLES NUM_SAMPLES
        #define PCF_NUM_SAMPLES NUM_SAMPLES

        vec2 poissonDisk[NUM_SAMPLES];

        void initPoissonSamples( const in vec2 randomSeed ) {
            float ANGLE_STEP = PI2 * float( NUM_RINGS ) /
float( NUM_SAMPLES );
            float INV_NUM_SAMPLES = 1.0 / float( NUM_SAMPLES );

            // jsfiddle that shows sample pattern:
https://jsfiddle.net/a16fflp7/
            float angle = rand( randomSeed ) * PI2;
            float radius = INV_NUM_SAMPLES;
            float radiusStep = radius;

            for( int i = 0; i < NUM_SAMPLES; i ++ ) {
                poissonDisk[i] = vec2( cos( angle ),
sin( angle ) ) * pow( radius, 0.75 );
                radius += radiusStep;
                angle += ANGLE_STEP;
            }
        }

        float penumbraSize( const in float zReceiver, const in
float zBlocker ) { // Parallel plane estimation
            return (zReceiver - zBlocker) / zBlocker;
        }

        float findBlocker( sampler2D shadowMap, const in vec2
uv, const in float zReceiver ) {
```



```

        // This uses similar triangles to compute what
        // area of the shadow map we should search
        float searchRadius = LIGHT_SIZE_UV * ( zReceiver -
NEAR_PLANE ) / zReceiver;
        float blockerDepthSum = 0.0;
        int numBlockers = 0;

        for( int i = 0; i < BLOCKER_SEARCH_NUM_SAMPLES; i++
) {
            float                shadowMapDepth                =
unpackRGBAToDepth(texture2D(shadowMap,    uv    +    poissonDisk[i]    *
searchRadius));
            if ( shadowMapDepth < zReceiver ) {
                blockerDepthSum += shadowMapDepth;
                numBlockers ++;
            }
        }

        if( numBlockers == 0 ) return -1.0;

        return blockerDepthSum / float( numBlockers );
    }

    float PCF_Filter(sampler2D shadowMap, vec2 uv, float
zReceiver, float filterRadius ) {
        float sum = 0.0;
        for( int i = 0; i < PCF_NUM_SAMPLES; i ++ ) {
            float                depth                =
unpackRGBAToDepth( texture2D( shadowMap, uv + poissonDisk[ i ] *
filterRadius ) );
            if( zReceiver <= depth ) sum += 1.0;
        }
        for( int i = 0; i < PCF_NUM_SAMPLES; i ++ ) {
            float                depth                =
unpackRGBAToDepth( texture2D( shadowMap, uv + -poissonDisk[ i ].yx
* filterRadius ) );
            if( zReceiver <= depth ) sum += 1.0;
        }
        return sum / ( 2.0 * float( PCF_NUM_SAMPLES ) );
    }

    float PCSS ( sampler2D shadowMap, vec4 coords ) {
        vec2 uv = coords.xy;
        float zReceiver = coords.z; // Assumed to be eye-
space z in this code

        initPoissonSamples( uv );
        // STEP 1: blocker search
        float avgBlockerDepth = findBlocker( shadowMap, uv,
zReceiver );

```

```

        //There are no occluders so early out (this saves
filtering)
        if( avgBlockerDepth == -1.0 ) return 1.0;

        // STEP 2: penumbra size
        float    penumbraRatio    =    penumbraSize(    zReceiver,
avgBlockerDepth );
        float    filterRadius    =    penumbraRatio * LIGHT_SIZE_UV
* NEAR_PLANE / zReceiver;

        // STEP 3: filtering
        //return avgBlockerDepth;
        return    PCF_Filter(    shadowMap,    uv,    zReceiver,
filterRadius );
    }
</script>
</head>

<body>
<script src="js/Three.js"></script>
<script src="js/Detector.js"></script>
<script src="js/OrbitControls.js"></script>
<script src="js/THREEEx.KeyboardState.js"></script>
<script src="js/THREEEx.WindowResize.js"></script>

<div    id="ThreeJS"    style="position:    absolute;    left:0px;
top:0px"></div>
<script>

// variables
var container, scene, camera, renderer, controls;
var keyboard = new THREEEx.KeyboardState();
var sphereCamera;

init();
animate();

// FUNCTIONS
function init()
{
    // SCENE
    scene = new THREE.Scene();

    // CAMERA
    var    SCREEN_WIDTH    =    window.innerWidth,    SCREEN_HEIGHT    =
window.innerHeight;
    var    VIEW_ANGLE    =    45,    ASPECT    =    SCREEN_WIDTH / SCREEN_HEIGHT,
NEAR    =    0.1,    FAR    =    20000;

```

```

    camera = new THREE.PerspectiveCamera( VIEW_ANGLE, ASPECT,
NEAR, FAR);
    scene.add(camera);
    camera.position.set(200, 400, 800);
    camera.rotation.x = Math.PI / 3;
    camera.lookAt(scene.position);

    // RENDERER
    if ( Detector.webgl )
        renderer = new THREE.WebGLRenderer( {antialias:true} );
    else
        renderer = new THREE.CanvasRenderer();
    renderer.setSize(SCREEN_WIDTH, SCREEN_HEIGHT);
    container = document.getElementById( 'ThreeJS' );
    container.appendChild( renderer.domElement );

    // EVENTS
    THREEEx.WindowResize(renderer, camera);

    // CONTROLS
    controls = new THREE.OrbitControls(camera,
renderer.domElement);

    // LIGHT
    var light = new THREE.PointLight(0xffffffff);
    light.position.set(100,1000,-500);
    scene.add(light);
    var light1 = new THREE.PointLight(0xffffffff);
    light1.position.set(100,500,300);

    light1.castShadow = true;
    scene.add(light1);

    // FLOOR
    var floorGeometry = new THREE.PlaneGeometry(2000,2000);
    var floorMaterial = new THREE.MeshBasicMaterial({color:
0x009933, side: THREE.DoubleSide});
    var floor = new THREE.Mesh(floorGeometry, floorMaterial);
    floor.rotation.x = Math.PI / 2;
    floor.position.set(0,-50,0);
    floor.receiveShadow = true;
    scene.add(floor);

    // SKYBOX/FOG
    var materialArray = [];
    materialArray.push(new THREE.MeshBasicMaterial( { map:
THREE.ImageUtils.loadTexture( 'images/px.png' ) }));
    materialArray.push(new THREE.MeshBasicMaterial( { map:
THREE.ImageUtils.loadTexture( 'images/nx.png' ) }));

```

```

        materialArray.push(new THREE.MeshBasicMaterial( { map:
THREE.ImageUtils.loadTexture( 'images/py.png' ) }));
        materialArray.push(new THREE.MeshBasicMaterial( { map:
THREE.ImageUtils.loadTexture( 'images/ny.png' ) }));
        materialArray.push(new THREE.MeshBasicMaterial( { map:
THREE.ImageUtils.loadTexture( 'images/pz.png' ) }));
        materialArray.push(new THREE.MeshBasicMaterial( { map:
THREE.ImageUtils.loadTexture( 'images/nz.png' ) }));
        for (var i = 0; i < 6; i++)
            materialArray[i].side = THREE.BackSide;
        var skyboxMaterial = new
THREE.MeshFaceMaterial( materialArray );
        var skyboxGeom = new THREE.CubeGeometry( 2000, 2000, 2000 );
        var skybox = new THREE.Mesh( skyboxGeom, skyboxMaterial );
        scene.add( skybox );

        var shader = THREE.ShaderChunk.shadowmap_pars_fragment;
        shader = shader.replace(
            "#ifdef USE_SHADOWMAP",
            "#ifdef USE_SHADOWMAP" +
document.getElementById("PCSS").textContent
        );
        THREE.ShaderChunk.shadowmap_pars_fragment = shader;

        ////////////
        // CUSTOM //
        ////////////

        var material;
        var sphereGeom = new THREE.SphereGeometry( 50, 32, 16 ); //
radius, segmentsWidth, segmentsHeight

        sphereCamera = new THREE.CubeCamera( 0.1, 5000, 512 );
        sphereCamera.renderTarget.minFilter =
THREE.LinearMipMapLinearFilter;
        scene.add( sphereCamera );

        geometry = new THREE.SphereGeometry( 50, 64, 30 );
        texture = new
THREE.ImageUtils.loadTexture( 'images/whiteball.png' );
        material = new THREE.MeshLambertMaterial( { map: texture } );
        var whiteball = new THREE.Mesh( geometry, material );
        whiteball.metallic = true;
        whiteball.position.set(-300,0,264);
        scene.add(whiteball);

        var redball1;
        texture = new
THREE.ImageUtils.loadTexture( 'images/redball.png' );

```

```

material = new THREE.MeshLambertMaterial( { map: texture } );
redball1 = new THREE.Mesh( geometry, material );
redball1.position.set(-150,0,10);
scene.add(redball1);

var redball2;
redball2 = new THREE.Mesh( geometry, material );
redball2.position.set(-50,0,10);
scene.add(redball2);

texture = new
THREE.ImageUtils.loadTexture( 'images/1ball.png' );
material = new THREE.MeshLambertMaterial( { map: texture } );
var ball1 = new THREE.Mesh( geometry, material );
ball1.position.set(-200,0,100);
ball1.rotation.x = -Math.PI/4;
ball1.rotation.y = Math.PI;
ball1.rotation.z = Math.PI/6;
ball1.castShadow = true;
ball1.receiveShadow = true;
scene.add(ball1);

texture = new
THREE.ImageUtils.loadTexture( 'images/12ball.png' );
material = new THREE.MeshLambertMaterial( { map: texture } );
var ball12 = new THREE.Mesh( geometry, material );
ball12.position.set(-100,0,100);
ball12.rotation.x = -Math.PI/4;
ball12.rotation.y = Math.PI;
ball12.rotation.z = -Math.PI/4;
ball12.castShadow = true;
ball12.receiveShadow = true;
scene.add(ball12);

texture = new
THREE.ImageUtils.loadTexture( 'images/10ball.png' );
material = new THREE.MeshLambertMaterial( { map: texture } );
var ball10 = new THREE.Mesh( geometry, material );
ball10.position.set(10,0,120);
ball10.rotation.y = Math.PI;
ball10.rotation.x = 0.0;
ball10.rotation.z = Math.PI/2;
ball10.castShadow = true;
ball10.receiveShadow = true;
scene.add(ball10);

texture = new
THREE.ImageUtils.loadTexture( 'images/11ball.png' );
material = new THREE.MeshLambertMaterial( { map: texture } );
;

```

```

var ball11 = new THREE.Mesh( geometry, material );
ball11.position.set(-280,0,-80);
ball11.rotation.x = -Math.PI/4+Math.PI/16;
ball11.rotation.y = Math.PI;
ball11.rotation.z = -Math.PI/6;
ball11.castShadow = true;
ball11.receiveShadow = true;
scene.add(ball11);

var cubeGeometry = new THREE.CubeGeometry(50,50,50,50);
                                texture = new
THREE.TextureLoader().load( 'images/grayball.png' );
material = new THREE.MeshLambertMaterial( { map: texture } );
var cube = new THREE.Mesh( cubeGeometry, material );
cube.position.set(-50,-24,350)
scene.add(cube);

var points = [];
for ( var i = 0; i < 10; i ++ ) {
    points.push( new THREE.Vector2( Math.sin( i * 0.2 ) * 10
+ 5, ( i - 5 ) * 2 ) );
}
var geometry1 = new THREE.LatheGeometry( points );
var lathe = new THREE.Mesh( geometry1, material );
lathe.position.set(-50,50,350)
scene.add( lathe );

var cylindergeometry = new THREE.CylinderGeometry(5, 8, 500,
512, false);
var cylindermaterial = new THREE.MeshLambertMaterial({color:
0xffffffff});
cue = new THREE.Mesh(cylindergeometry, cylindermaterial);
cue.rotation.x = Math.PI / 2;
cue.rotation.z = Math.PI / 2;
cue.position.set(-50,0,450);
scene.add(cue);

cylindergeometry = new THREE.CylinderGeometry(5, 5, 10, 512,
false);
cylindermaterial = new THREE.MeshLambertMaterial({color:
0x000000});
cue_top1 = new THREE.Mesh(cylindergeometry,
cylindermaterial);
cue_top1.rotation.x = Math.PI / 2;
cue_top1.rotation.z = Math.PI / 2;
cue_top1.position.set(-305,0,450);
scene.add(cue_top1);

```

```

        cylindergeometry = new THREE.CylinderGeometry(5, 5, 6, 512,
false);
        cylindermaterial = new THREE.MeshLambertMaterial({color:
0x0000ff});
        cue_top2 = new THREE.Mesh(cylindergeometry,
cylindermaterial);
        cue_top2.rotation.x = Math.PI / 2;
        cue_top2.rotation.z = Math.PI / 2;
        cue_top2.position.set(-312,0,450);
        scene.add(cue_top2);

    }

function animate()
{
    requestAnimationFrame( animate );
    render();
    update();
}

function update()
{
    controls.update();
}

function render()
{
    // move the CubeCamera to the position of the object
    // that has a reflective surface, "take a picture" in each
direction
    // and apply it to the surface.
    // need to hide surface before and after so that it does not
    // "get in the way" of the camera
    sphereCamera.updateCubeMap(renderer, scene);
    renderer.outputEncoding = THREE.sRGBEncoding;
    renderer.shadowMapEnabled = true;
    renderer.render(scene, camera);
}

</script>

</body>
</html>

```