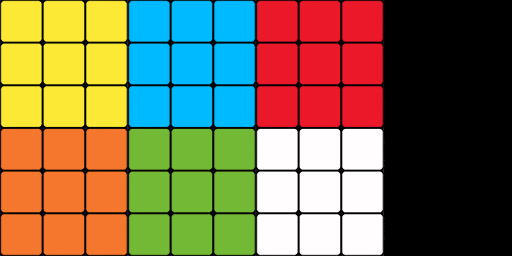
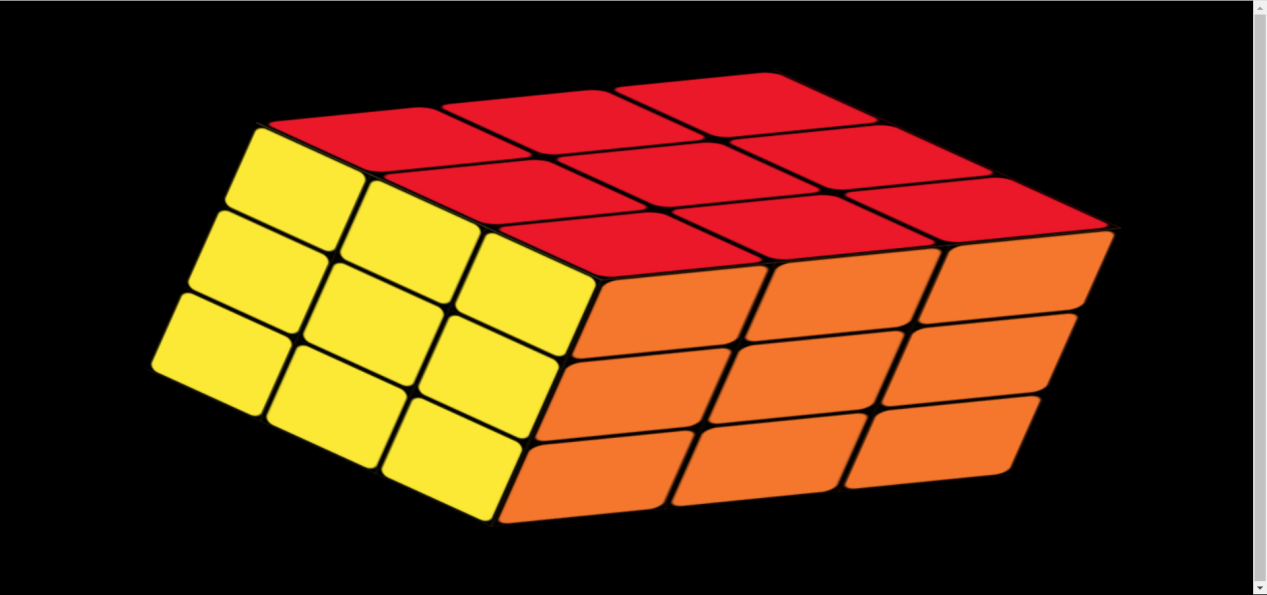
看一下，我们如何将下面的图贴到正方体中





我们的正方体是由6个面组成，而每个面则是由2个三角形（也就是6个点）组成，只要我们指定3角形的三个点，然后我们再指定其对应的贴图的3个点位置，那么webgl就可以将这3角贴图贴到我们3角形上面了，原理就是这样，看一下示例代码

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

    <script src="http://www.yanhuangxueyuan.com/versions/threejsR92/build/three.js"></script>

</head>

<body style="margin: 0px;">

    <canvas id="canvas"></canvas>

    <!-- 顶点着色器 -->

    <script id="vertexShader" type="x-shader/x-vertex">

        // 导出属性

        attribute vec4 a\_Position;

        // 纹理坐标

        attribute vec2 a\_Pin;

        // 声明全局指针

        varying vec2 v\_Pin;

        // 物体模型

        uniform mat4 u\_ModelMatrix;

        void main(){

            gl\_Position = u\_ModelMatrix\*a\_Position;

            // 将v\_Pin指向a\_Pin

            v\_Pin = a\_Pin;

        }

    </script>

    <!-- 片元着色器 -->

    <script id="fragmentShader" type="x-shader/x-fragment">

        precision mediump float;

        // 二维取样器

        // 这里需要导出取样器

        uniform sampler2D u\_Sampler;

        varying vec2 v\_Pin;

        void main() {

            // 设置颜色从纹理中进行获取

            gl\_FragColor = texture2D(u\_Sampler, v\_Pin);

        }

    </script>

    <script>

        // 初始化

        function init(context) {

            //创建程序对象

            const program = context.createProgram();

            //创建顶点着色器

            const vertexShader = context.createShader(context.VERTEX\_SHADER);

            // 获取顶点着色器代码

            const vsSource = document.getElementById('vertexShader').innerText;

            //将着色器源文件传入着色器对象中

            context.shaderSource(vertexShader, vsSource);

            //编译着色器对象

            context.compileShader(vertexShader);

            //把顶点着色对象装进程序对象中

            context.attachShader(program, vertexShader);

            //创建片元着色器

            const fragmentShader = context.createShader(context.FRAGMENT\_SHADER);

            // 获取片元着色器代码

            const fsSource = document.getElementById('fragmentShader').innerText;

            //将着色器源文件传入着色器对象中

            context.shaderSource(fragmentShader, fsSource);

            //编译着色器对象

            context.compileShader(fragmentShader)

            //把片元着色对象装进程序对象中

            context.attachShader(program, fragmentShader);

            //连接webgl上下文对象和程序对象

            context.linkProgram(program);

            //启动程序对象

            context.useProgram(program);

            //将程序对象挂到上下文对象上

            context.program = program;

        }

        // canvas画布

        const canvas = document.getElementById('canvas');

        canvas.width = document.body.clientWidth;

        canvas.height = document.body.clientHeight;

        // webgl上下文

        const context = canvas.getContext('webgl');

        // 初始化上下文

        init(context);

        // 指定将要用来清理绘图区的颜色

        context.clearColor(0.0, 0.0, 0.0, 1.0);

        context.enable(context.CULL\_FACE);

        context.enable(context.DEPTH\_TEST);

        // 清理绘图区

        context.clear(context.COLOR\_BUFFER\_BIT);

        // 获取坐标指针

        const a\_Position = context.getAttribLocation(context.program, 'a\_Position');

        // 获取纹理坐标指针

        const a\_Pin = context.getAttribLocation(context.program, 'a\_Pin');

        // 获取取样器

        const u\_Sampler = context.getUniformLocation(context.program, 'u\_Sampler');

        // 物体运动矩阵

        const u\_ModelMatrix = context.getUniformLocation(context.program, 'u\_ModelMatrix')

        //模型矩阵

        const modelMatrix = new THREE.Matrix4();

        const mx = new THREE.Matrix4().makeRotationX(0.02)

        const my = new THREE.Matrix4().makeRotationY(0.02)

        //缓冲对象

        const sourceBuffer = context.createBuffer();

        //绑定缓冲对象

        context.bindBuffer(context.ARRAY\_BUFFER, sourceBuffer);

        // 指定绘图的顶点坐标和对应的纹理坐标

        const points = [

            -0.5, -0.5, -0.5, 0, 0,

            -0.5, 0.5, -0.5, 0, 0.5,

            0.5, -0.5, -0.5, 0.25, 0,

            -0.5, 0.5, -0.5, 0, 0.5,

            0.5, 0.5, -0.5, 0.25, 0.5,

            0.5, -0.5, -0.5, 0.25, 0,

            -0.5, -0.5, 0.5, 0.25, 0,

            0.5, -0.5, 0.5, 0.5, 0,

            -0.5, 0.5, 0.5, 0.25, 0.5,

            -0.5, 0.5, 0.5, 0.25, 0.5,

            0.5, -0.5, 0.5, 0.5, 0,

            0.5, 0.5, 0.5, 0.5, 0.5,

            -0.5, 0.5, -0.5, 0.5, 0,

            -0.5, 0.5, 0.5, 0.5, 0.5,

            0.5, 0.5, -0.5, 0.75, 0,

            -0.5, 0.5, 0.5, 0.5, 0.5,

            0.5, 0.5, 0.5, 0.75, 0.5,

            0.5, 0.5, -0.5, 0.75, 0,

            -0.5, -0.5, -0.5, 0, 0.5,

            0.5, -0.5, -0.5, 0.25, 0.5,

            -0.5, -0.5, 0.5, 0, 1,

            -0.5, -0.5, 0.5, 0, 1,

            0.5, -0.5, -0.5, 0.25, 0.5,

            0.5, -0.5, 0.5, 0.25, 1,

            -0.5, -0.5, -0.5, 0.25, 0.5,

            -0.5, -0.5, 0.5, 0.25, 1,

            -0.5, 0.5, -0.5, 0.5, 0.5,

            -0.5, -0.5, 0.5, 0.25, 1,

            -0.5, 0.5, 0.5, 0.5, 1,

            -0.5, 0.5, -0.5, 0.5, 0.5,

            0.5, -0.5, -0.5, 0.5, 0.5,

            0.5, 0.5, -0.5, 0.75, 0.5,

            0.5, -0.5, 0.5, 0.5, 1,

            0.5, -0.5, 0.5, 0.5, 1,

            0.5, 0.5, -0.5, 0.75, 0.5,

            0.5, 0.5, 0.5, 0.75, 1,

        ];

        const source = new Float32Array(points);

        // 写入数据

        context.bufferData(context.ARRAY\_BUFFER, source, context.STATIC\_DRAW);

        // 设置a\_Position的指针属性，以便其能够从缓存中正确提前数据

        context.vertexAttribPointer(

            a\_Position,

            3,

            context.FLOAT,

            false,

            5 \* 4,

            0 \* 4

        );

        // 赋能-批处理

        context.enableVertexAttribArray(a\_Position);

        // 设置a\_Pin的指针属性，以便其能够从缓存中正确提前数据

        context.vertexAttribPointer(

            a\_Pin,

            2,

            context.FLOAT,

            false,

            5 \* 4,

            3 \* 4

        )

        //赋能-批处理

        context.enableVertexAttribArray(a\_Pin);

        // 对纹理图像垂直翻转

        context.pixelStorei(context.UNPACK\_FLIP\_Y\_WEBGL, 1);

        // 激活纹理单元

        context.activeTexture(context.TEXTURE0);

        // 创建纹理对象

        const texture = context.createTexture();

        // 把纹理对象装进纹理单元里

        context.bindTexture(context.TEXTURE\_2D, texture);

        // image 对象

        const image = new Image();

        image.src = './images/mf.jpg';

        image.onload = function () {

            //配置纹理图像

            context.texImage2D(

                context.TEXTURE\_2D,

                0,

                context.RGB,

                context.RGB,

                context.UNSIGNED\_BYTE,

                image

            );

            //配置纹理参数

            context.texParameteri(

                context.TEXTURE\_2D,

                context.TEXTURE\_WRAP\_S,

                context.CLAMP\_TO\_EDGE

            );

            context.texParameteri(

                context.TEXTURE\_2D,

                context.TEXTURE\_WRAP\_T,

                context.CLAMP\_TO\_EDGE

            );

            context.texParameteri(

                context.TEXTURE\_2D,

                context.TEXTURE\_MIN\_FILTER,

                context.LINEAR

            );

            //将0号纹理分配给着色器，0是纹理单元编号

            context.uniform1i(u\_Sampler, 0);

            //渲染

            context.clear(context.COLOR\_BUFFER\_BIT);

            // 建立倒带上角形

            context.drawArrays(context.TRIANGLES, 0, points.length / 5);

            // 连续渲染

            (function ani() {

                modelMatrix.multiply(my).multiply(mx)

                context.uniformMatrix4fv(u\_ModelMatrix, false, modelMatrix.elements)

                //渲染

                context.clear(context.COLOR\_BUFFER\_BIT);

                // 建立倒带上角形

                context.drawArrays(context.TRIANGLES, 0, points.length / 5);

                // 下一帧动画时调用ani

                requestAnimationFrame(ani)

            })()

        }

    </script>

</body>

</html>

知识点

- 开启深度测试

context.enable(context.DEPTH\_TEST);

默认情况下，WebGL的片元并不是谁里屏幕近就显示谁，而是谁最后渲染就显示谁。开启深度测试后，如果2个片元的xy相同，那么会比较他们的z值，z值小的片元将会被隐藏