一、 题目

太阳系模拟系统

二、 概要

基于 three.js 的 WebGL API 模拟实现一个太阳系系统。
主要功能模块如下：

1. 运动模块：实现各行星的公转与自转；
2. 轨道绘制：绘制每个行星的公转和自传轨道；
3. 土星环绘制：使用自定义的几何体创建土星环；
4. 太阳绘制：着色器绘制太阳；
5. 群星绘制：很多随机的 Point 构成远处恒星；
6. 各行星视角：好似身处于行星之上，观看太阳系。

三、 细节

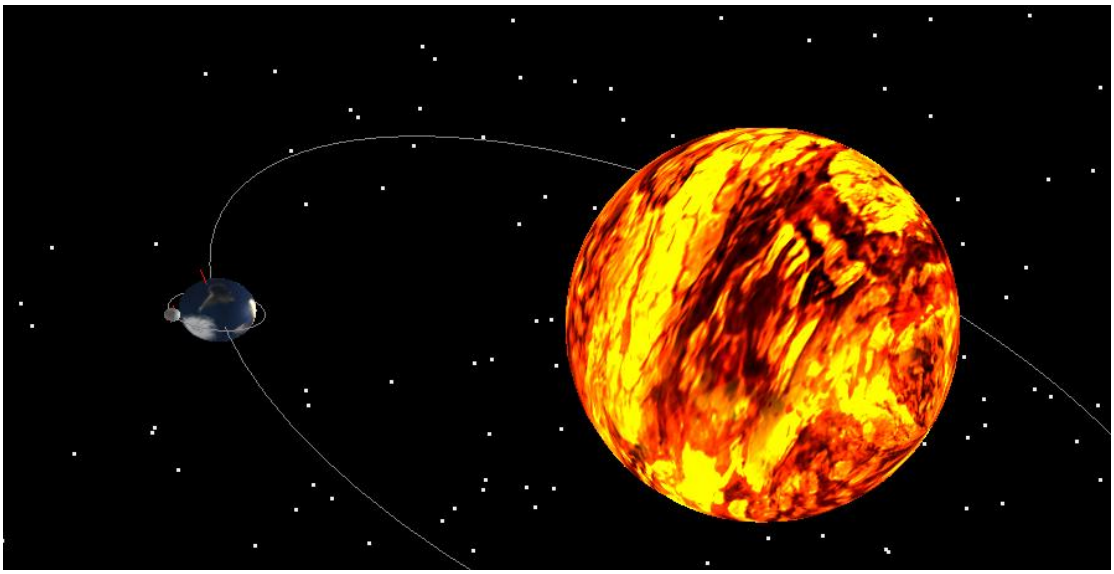
1. 运动模块：
   运动包括自转、公转、轨道倾角。
   a) 行星运动轨道是椭圆的，但偏心率太小了，如果按实际的绘制出来，难度大了不少，而效果可能也看不出来，所以没必要，最终使用圆形轨道。

偏心率
水星 0.206
金星 0.007
地球 0.017
火星 0.093
木星 0.048
土星 0.055
天王星 0.051
海王星 0.006

   b) 自转轨道的倾角很大，不可忽略。
   c) 行星在自转的同时，还在公转，直接考虑，运动比较复杂。
   在太阳系中运动最为复杂的是月球，拥有三种运动状态：自转，绕地球公转，被地球带着绕太阳公转。
   使用层次化模型解决上述的运动问题，伪代码如下：

```
var earthGroup, earth, moonGroup, moon;

moonGroup.add(moon);
earthGroup.add(earth);    // earth 可以自转,
earthGroup.add(moonGroup);// earthGroup 可以带着

moon.rotate();           // moon 自转
earth.rotate();          // earth 自转
moonGroup.rotate();      // moon 公转
earthGroup.rotate();     // earth 带着 moon 公转
```

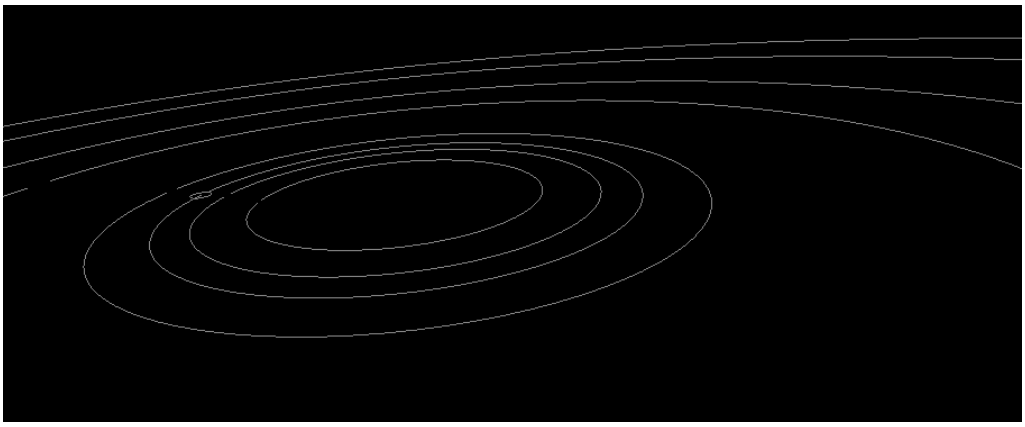
先单独创建 earth，moon,他们都是球形几何结构，几何中心设在日地距离和月地距离处，在创建 earthGroup,MoonGroup,他们是没有形状的空物体，几何中心设在太阳中心和地球中心。然后将他们按上述层次添加，最后各自独立旋转达到效果。



2. 轨道绘制：
利用行星运动轨道具有共面性和近圆性的特点，利用 three.js 中提供的 Line 类近似的绘制圆形轨道，由于不需要轨道的光学特性，所以没必要计算 normal。

```
var geometry = new THREE.Geometry();
for(var i = 0; i < points.length; i++){
    geometry.vertices.push(points[i]);
}
// 0xffffff = (ff,ff,ff)=(255,255,255)
var material = new THREE.LineBasicMaterial({color:color,
linewidth:5, opacity:.5});
return new THREE.Line(geometry, material);
```

传入函数的数据点是一个圆周上的等距采样点。



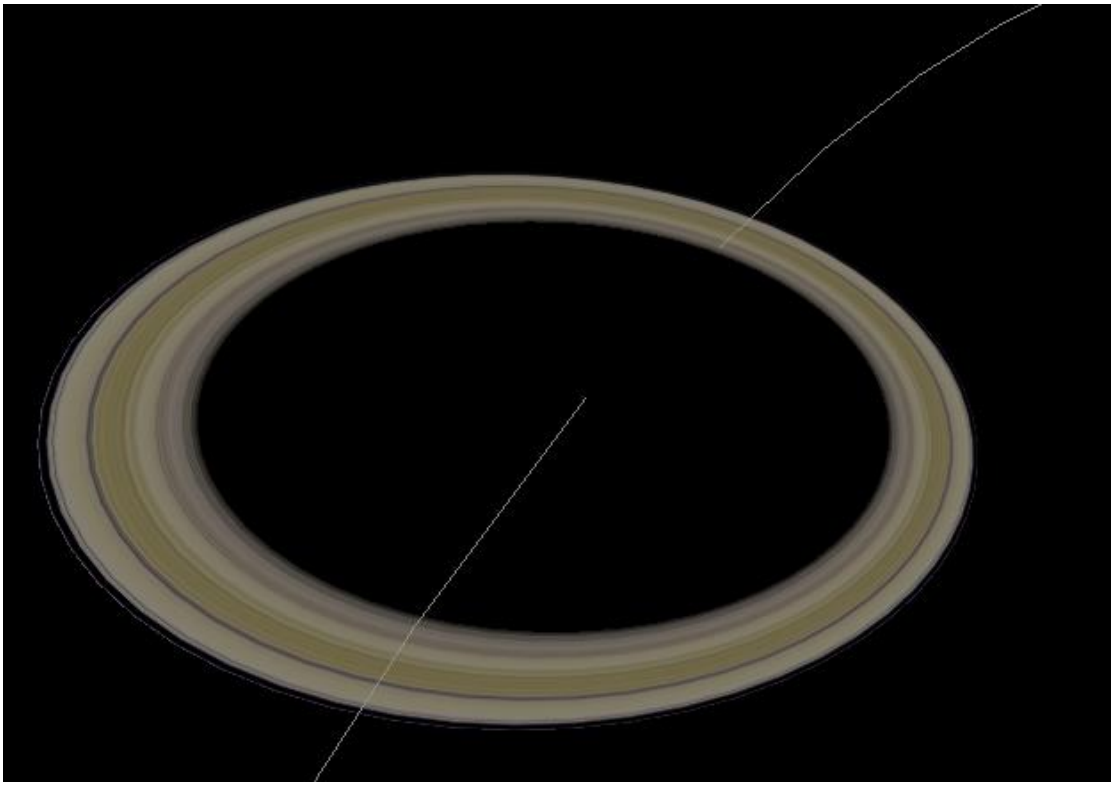
3. 土星环绘制：
土星环的绘制有一个难点就是，要自定义一个环形几何体，对环形几何体贴上合适的纹理。

```
var ring = new THREE.Geometry();
var i,frad,pos;
// 添加内圆和外圆上的等距采样点
for(i = 0; i < pointCount; i++){
    frad = i / pointCount * Math.PI * 2;
    pos = new THREE.Vector3(innerRadius * Math.cos(frad),
innerRadius * Math.sin(frad),0);
    ring.vertices.push(pos);
    pos = new THREE.Vector3(outRadius * Math.cos(frad),
outRadius * Math.sin(frad),0);
    ring.vertices.push(pos);
}
// 用添加的顶点绘制三角面片
for(i = 0; i < pointCount; i++){
    var v0 = 2 * i, v1 = 2 *i + 1, v2 = (2 * i + 2) % (2 *
pointCount), v3 = (2 * i + 3) % (2 * pointCount);
    // 建立两个三角形，模拟圆环的一部分
    ring.faces.push(new THREE.Face3(v0, v1, v2));
    ring.faces.push(new THREE.Face3(v1, v3, v2));
    // 对应于三角形的相应面的顶点的纹理坐标（）
    ring.faceVertexUvs[0].push([new THREE.Vector2(0,0),new
THREE.Vector2(1,0),new THREE.Vector2(0,1)]);
    ring.faceVertexUvs[0].push([new THREE.Vector2(1,0),new
THREE.Vector2(1,1),new THREE.Vector2(0,1)]);

    // 反向建一个面，保证正反都能看到这个面
    ring.faces.push(new THREE.Face3(v0,v2,v1));
    ring.faces.push(new THREE.Face3(v1, v2, v3));
```

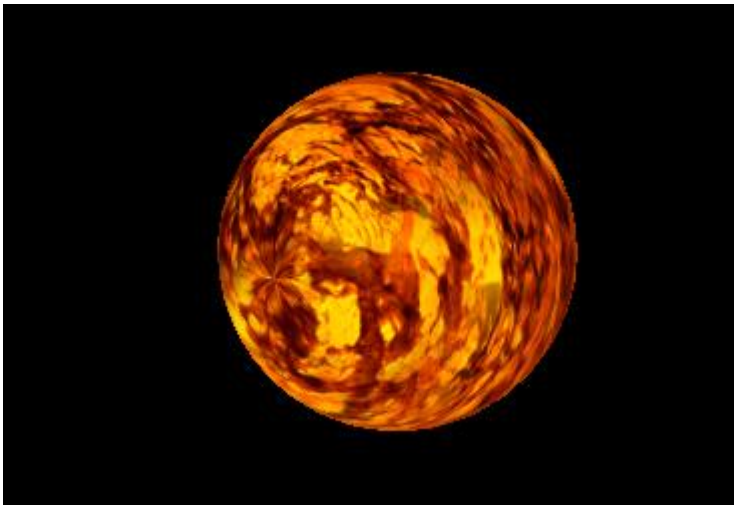
```
    ring.faceVertexUvs[0].push([new THREE.Vector2(0,0),new
THREE.Vector2(0,1),new THREE.Vector2(1,0)]);
    ring.faceVertexUvs[0].push([new THREE.Vector2(1,0),new
THREE.Vector2(0,1),new THREE.Vector2(1,1)]);
}
// 因为要利用光学特性，所以计算法向量
ring.computeFaceNormals();
return ring;
```

要注意的点是：

a. 利用顶点绘制三角面片时，一定要注意，顶点的添加顺序，按右手法则保持法向量一致。这里两个方向的法向量面都要添加，所以也么什么关系。
b. 由于为了保证土星环的正反两面都可见，所以要添加两个法向量方向相反的面。
c. 为每个添加纹理坐标，一定要对应好位置关系。而且土星环的纹理是有条纹方向的，所以如果纹理对应错误，也会使得条纹方向垂直土星。
d. 和前面的轨道不同，因为要利用光学特性，所以计算法向量

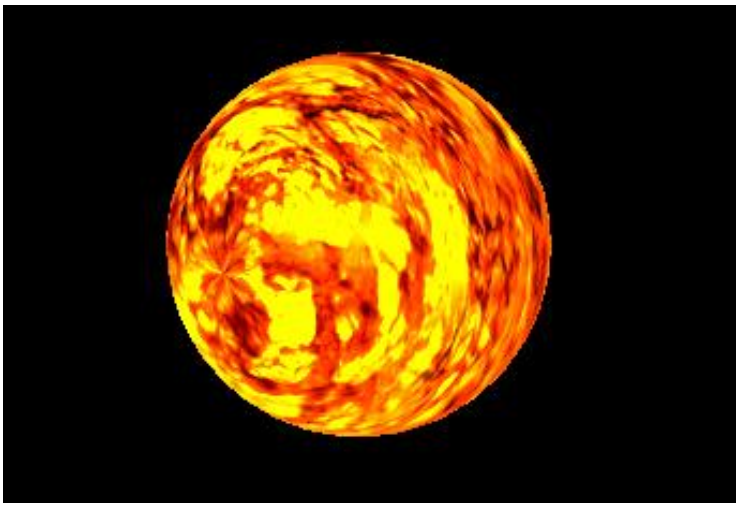


4. 太阳绘制：
提供的太阳纹理略显暗淡，没有鲜亮感。



使用 shader 对纹理的色彩进行处理：

```
vec4 temp = texture2D( texture2, T2 );
temp = temp * 2.0;
gl_FragColor = temp;
```



仅仅是将纹理色彩乘了 2 就明亮了起来

为了达到流动的太阳的效果，还要引入一个时间，声明 uniform

```
var uniforms = {
    time:{type:"f",value:1.0},//对 value 使用加法，所以不写成 string
    texture1:{type:"t",value: new
THREE.TextureLoader().load("./images/cloud.png")},
    texture2:{type:'t',value: new
THREE.TextureLoader().load("./images/lavatile.jpg")}
};
```

在帧刷新函数里调用更新时间

```
uniforms.time.value += delta;
```

在片源着色器中

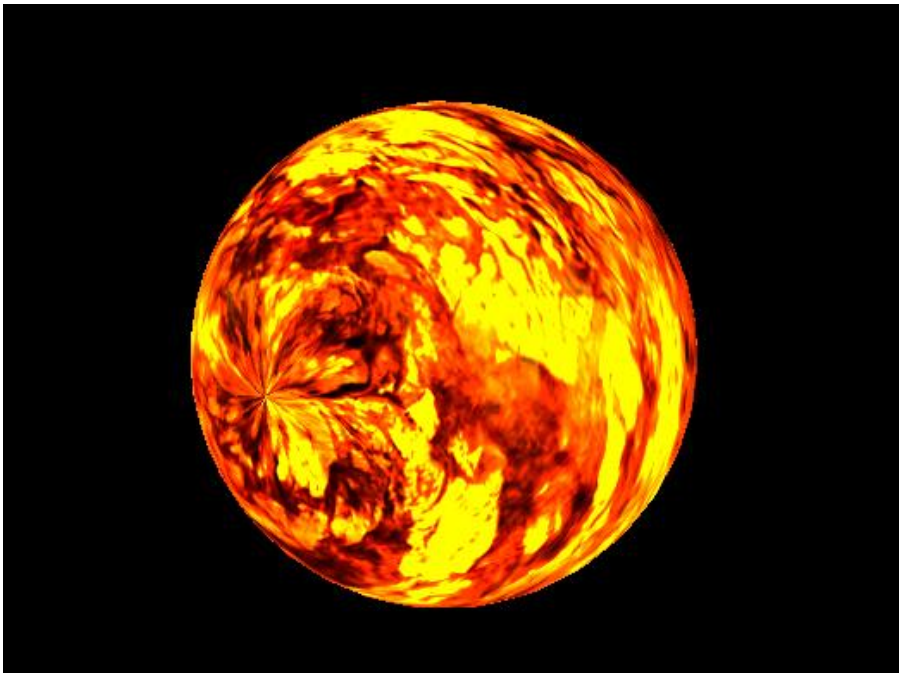
```
texCoord + vec2( 1.5, -1.5 ) * time  * 0.01;
```

引入了太阳黑子运动的 shader

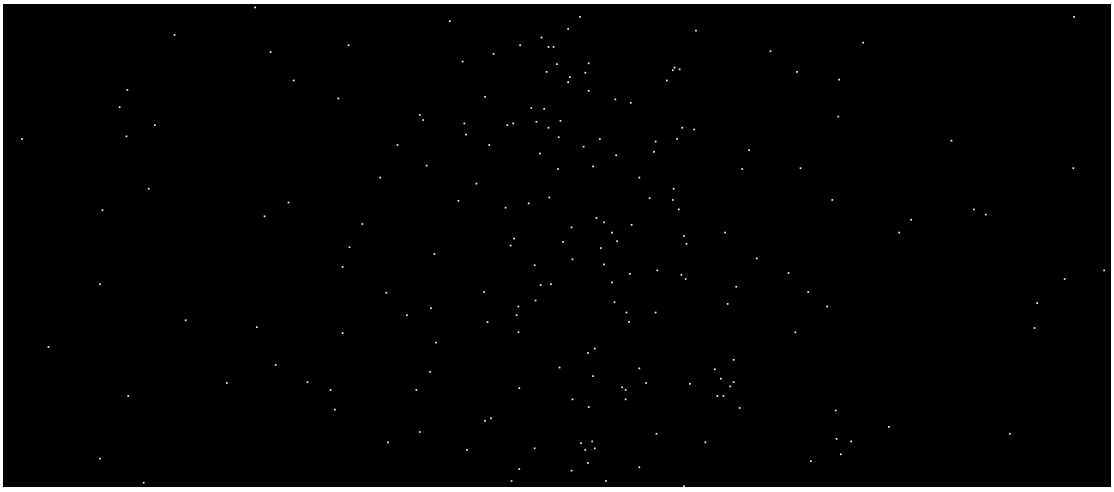
```
<script id="fragmentShader" type="x-shader/x-fragment">

        uniform float time;

        uniform sampler2D texture1;
        uniform sampler2D texture2;

        varying vec2 texCoord;

        void main( void ) {

            vec4 noise = texture2D( texture1, texCoord );
// 纹理流动
            vec2 T1 = texCoord + vec2( 1.5, -1.5 ) * time  *
0.01;
            vec2 T2 = texCoord + vec2( -0.5, 2.0 ) * time *
0.01;
//
            T1.x -= noise.r * 2.0;
            T1.y += noise.g * 4.0;
            T2.x += noise.g * 0.2;
            T2.y += noise.b * 0.2;

            float p = texture2D( texture1, T1 * 2.0 ).a +
0.25;

            vec4 color = texture2D( texture2, T2 );
            vec4 temp = color * 2.0 * ( vec4( p, p, p, p ) )
+ ( color * color );
            gl_FragColor = temp;
            //vec4 color = texture2D(texture2, texCoord);
            //vec4 temp = color * FOO;
            gl_FragColor = temp;
        }
</script>
<script id="vertexShader" type="x-shader/x-vertex">
      varying vec2 texCoord;
```

```
        void main()
        {
            texCoord = uv;
            vec4 mvPosition = modelViewMatrix * vec4( position,
1.0 ) * FOO;
            gl_Position = projectionMatrix * mvPosition;
        }

</script>
```

效果如下



5. 群星绘制：



在太阳系的星空，少了远处的恒星，自然失色了不少，远处的恒星创建就是随机创建一堆位置不小于某个距离的点，这些点颜色设为白色，汇聚起来就是星空，这里要注意的就是以下两点：

1. 远处的恒星是白色材质的点模拟的，而不是发光物体模拟。
2. 创建出的点的材质 PointsMaterial 的 sizeAttenuation 属性设为 false；表示不会随着相机的靠近远离而发生大小变化。

6. 行星视角：

由于使用前面的方法，距离与半径的缩放并不在一个数量级别上，所以当切换到地球等行星的视角上时，太阳显得格外地大。因此对行星视角，该缩放并不严谨，因此使用新的比例：

```
reRadius : function(r){
    return  r/250;
},
reDis : function(dis){
    return dis * 4;
}
```

为了创建行星视角，写了触发事件，‘e’表示切换下一视角，’L’表示切换主视角，即初始视角。
可以切换到太阳系八大行星的任意一个行星上。
比如地球视角

所谓地球视角，就是站在地球赤道上不动，沿着垂直自转轨道的方向观看星空。实现如下：
站在赤道上，赤道上一点的坐标最为相机位置

```
planet.getPosition=function(){      // subPlanet 的 position 是相对于父节点的 position
    var pos = new THREE.Vector4(-r,0,0,1);  // 赤道上的某个定点相对 subPlanet 的坐标
```

```
    return pos.applyMatrix4(subPlanet.matrixWorld);
};
```

看向垂直于自转转轴的方向，从该方向上选一点作为相机视点位置

```
planet.getLookPoint = function(){
    var pos = new THREE.Vector4(-2 * r *
Math.cos(shaftAngle),0,-2*r*Math.sin(shaftAngle),1);
    pos = pos.applyMatrix4(subPlanet.matrixWorld);
    return pos;
};
```

转轴方向作为屏幕的处置方向，即相机的 up 方向

```
planet.getUp = function(){
    return new
THREE.Vector3(Math.sin(shaftAngle),0,Math.cos(shaftAngle));
};
```

（注：其中 shaftAngle 是轨道倾角）

这样便实现了，行星视角了。

四、 问题

1. 最大的问题就是开发所使用的库与教程的库不匹配，很多函数变更了的问题，列举几个最经典的：
   a. ImageUtil 变更为 TextureLoader，使用会有警告
   b. Uniform 的 texture 的 value 属性由索引变成 texture 对象，使用不会报错，但没有效果。
   c. Vertex 类被删除，创建 geometry 时，顶点直接是 vector，而不用转成 vertex,使用会报错。

2. 其次的问题是实现地球视角时，太阳有时变成椭圆的原因，还不能很好解释。

3. 加入了太阳黑子的 shader 的实现中太阳黑子的干扰的数学原理也不是很清楚。