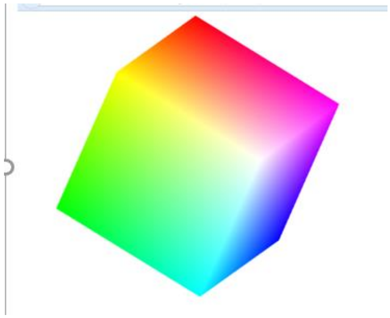
Outlines

- Transformation Cube（*）
  - Model Cube
  - ModelView Matrix
  - Program Realize
- Analog tracking ball（*）
- Quaternions for Rotation

chap4.6 建模一个彩色立方体
chap 4.11~4.14: WEBGL如何实现变换~cube,trackball实例

colorcube

See cube.js and cubev.js

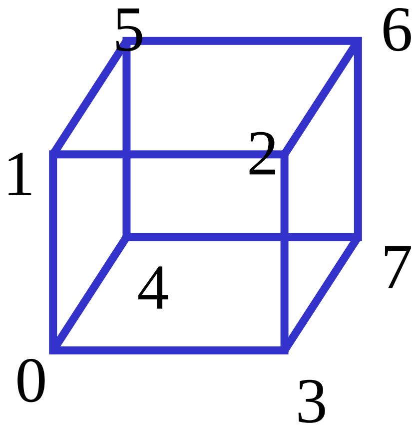


```
var canvas, gl;
var numVertices  = 36;
var points = [];
var colors = [];

window.onload = function init(){
    canvas = document.getElementById( "gl-canvas" );
    gl = WebGLUtils.setupWebGL( canvas );

    colorCube();
    ......
```

Quad(四边形）

See cube.js

- vertices are ordered, obtain correct outward facing normal

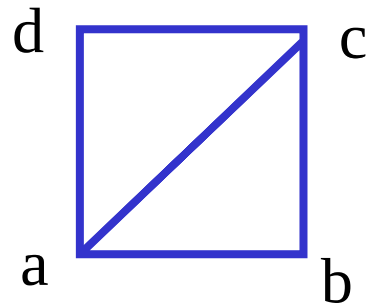



```
function colorCube( )
{    quad(0,3,2,1);
     quad(2,3,7,6);
     quad(0,4,7,3);
     quad(1,2,6,5);
     quad(4,5,6,7);
     quad(0,1,5,4);
}
```

The quad Function

See cube.js

Each quad(a,b,c,d) generates two triangles（webgl）

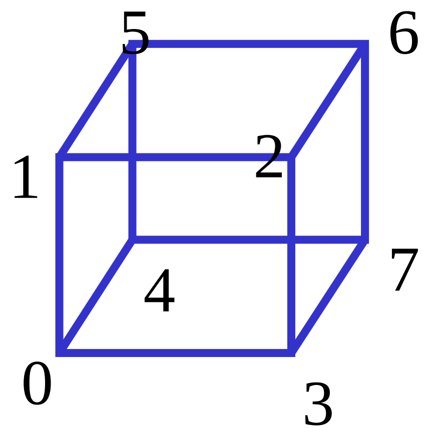



```
var quad(a, b, c, d)
{
   var indices = [ a, b, c, a, c, d ];
   for ( var i = 0; i < indices.length; ++i ) {
      points.push( vertices[indices[i]]);
      colors.push( vertexColors[indices[i]] );
      //上句是smooth插值填色时使用，如果一个面填一色，用下句
      //colors.push(vertexColors[a]);
 }
}
```

Vetext locations

See cube.js

- Define global array for vertices




```
var vertices = [
        vec3( -0.5, -0.5,  0.5 ),
        vec3( -0.5,  0.5,  0.5 ),
        vec3(  0.5,  0.5,  0.5 ),
        vec3(  0.5, -0.5,  0.5 ),
        vec3( -0.5, -0.5, -0.5 ),
        vec3( -0.5,  0.5, -0.5 ),
        vec3(  0.5,  0.5, -0.5 ),
        vec3(  0.5, -0.5, -0.5 )
    ];
```

Vetext Colors

- Define global array for colors

See cube.js

```
var vertexColors = [
        [ 0.0, 0.0, 0.0, 1.0 ],  // black
        [ 1.0, 0.0, 0.0, 1.0 ],  // red
        [ 1.0, 1.0, 0.0, 1.0 ],  // yellow
        [ 0.0, 1.0, 0.0, 1.0 ],  // green
        [ 0.0, 0.0, 1.0, 1.0 ],  // blue
        [ 1.0, 0.0, 1.0, 1.0 ],  // magenta
        [ 0.0, 1.0, 1.0, 1.0 ],  // cyan
        [ 1.0, 1.0, 1.0, 1.0 ]   // white
    ];
```

Render By Arrays

- *See cube.js* by arrays

- **Send Vertexs to GPU**

```
var vBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW );
```

- **Render**

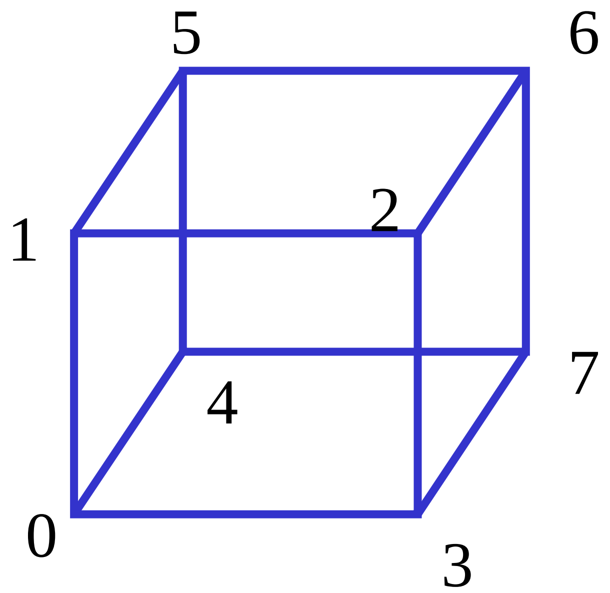
```
function render(){
    gl.clear( gl.COLOR_BUFFER_BIT |gl.DEPTH_BUFFER_BIT);
    gl.drawArrays( gl.TRIANGLES, 0, numVertices );
    requestAnimFrame( render );}
```

Mapping indices to faces

- 使用元素数组绘制网格*(see cubev.js, 书* **4.6.7***)*

✓ ***定义Indices数组，包含立方体所有拓扑信息6*2=12个三角形***

✓ ***不再需要定义quad,也不需要points和colors数组***




```
var indices =[
1,0,3, 3,2,1,
2,3,7, 7,6,2,
3,0,4, 4,7,3,
6,5,1, 1,2,6,
4,5,6, 6,7,4,
5,4,0, 0,1,5
];
```

Rendering by Elements

- 使用元素数组绘制网格*(see cubev.js,* 书 **4.6.7***)*

- **Send indices to GPU**

```
var iBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, iBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
       new Uint8Array(indices), gl.STATIC_DRAW);
```

- **Render**

```
gl.drawElements(gl.TRIANGLES, numVertices,
                gl.UNSIGNED_BYTE, 0 );
```

Outlines

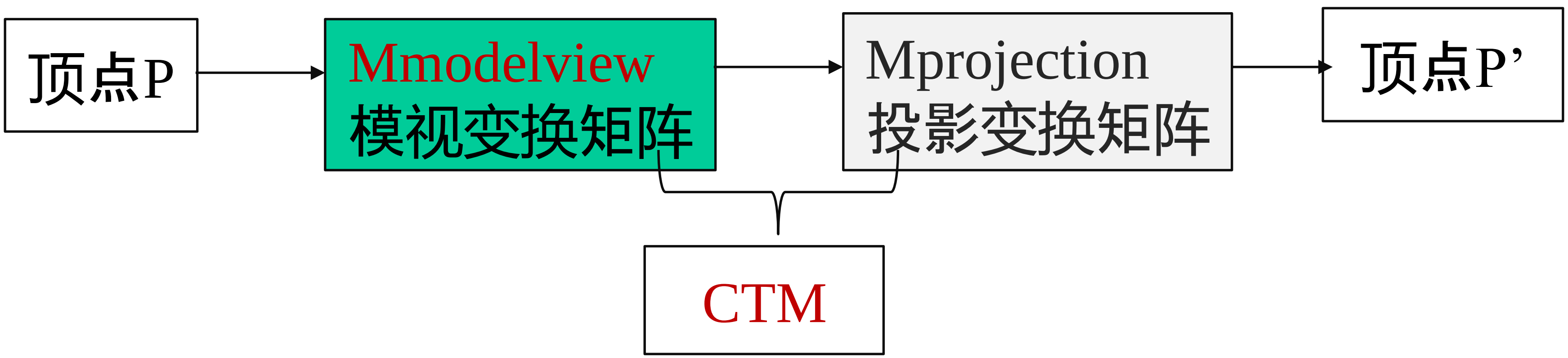
- Transformation Cube（*）
  - Model Cube
  - ModelView Matrix
  - Program Realize
- Analog tracking ball（*）
- Quaternions for Rotation

chap4.6 建模一个彩色立方体

chap 4.11~4.14: WEBGL如何实现变换~cube,trackball实例

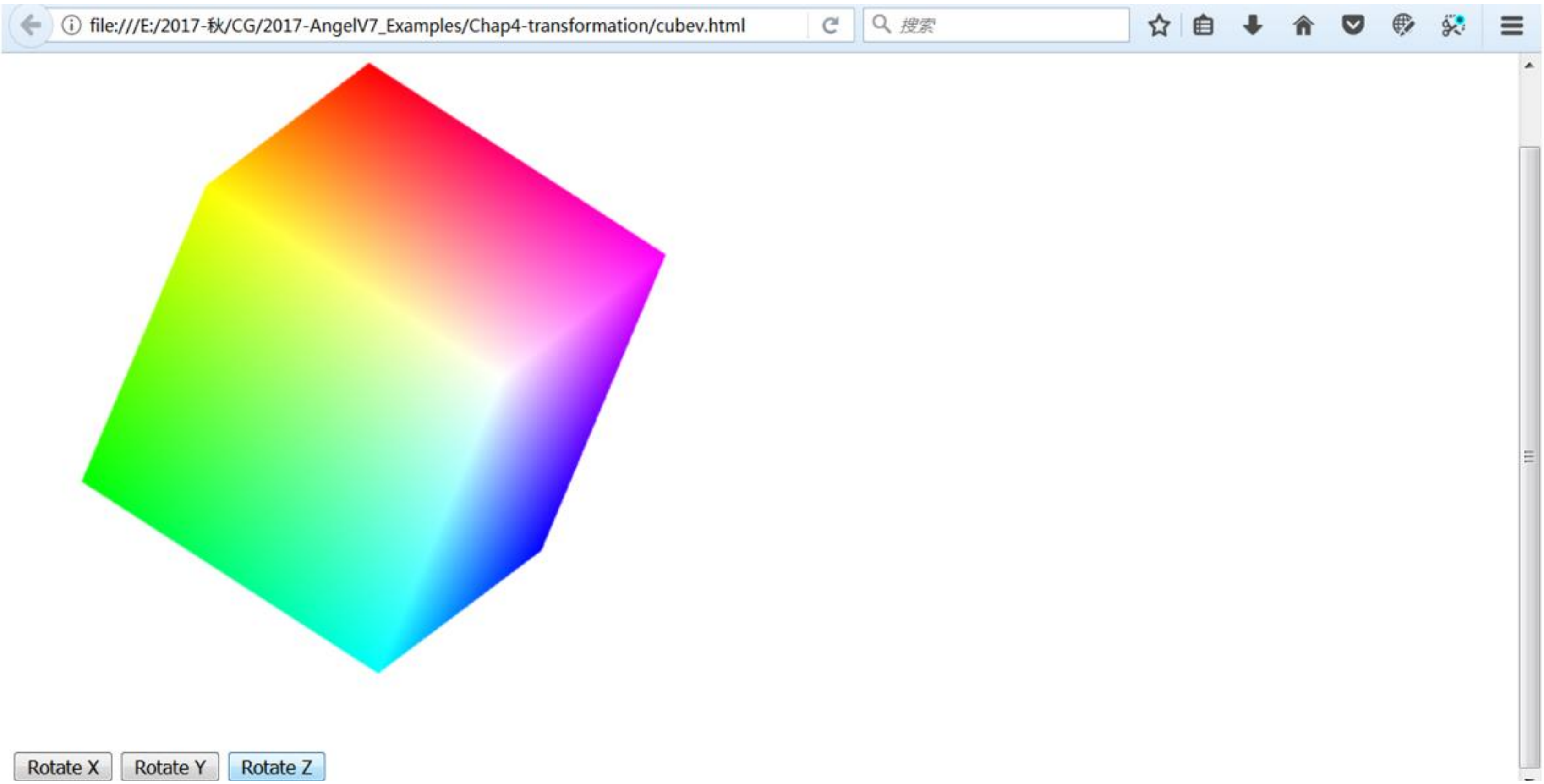
CTM

- P’= CTM * P
  - Current Transformation Matrix(当前变换矩阵)
  - CTM=Mprojection * Mmodelview



演示matrixModelView.exe

Cube / Cubev中的旋转实现



Cube中的变换

- 任何绕原点固定点的旋转，都可以分解成三个绕坐标轴的旋转，顺序可能不同，但是都可以分解实现。

$$R=R_X(thetaX)\cdot R_Y(thetaY)\cdot R_Z(thetaZ)$$

$$P'=RP \qquad (CTM=R)$$

- 每次绘制时，三个旋转角中的某一个会增加一定的角度，至于增加哪一个旋转角由用户交互决定。

? Where apply transformation?

see cube.js

Adding Buttons for Rotation

- 分别控制三个旋转方向

```
var xAxis = 0;
var yAxis = 1;
var zAxis = 2;
var axis = 0;
var theta = [ 0, 0, 0 ];
var thetaLoc;

document.getElementById( "xButton" ).onclick =
function () {axis = xAxis;     };
document.getElementById( "yButton" ).onclick =
function () {axis = yAxis;     };
document.getElementById( "zButton" ).onclick =
function () {axis = zAxis;     };
```

Where apply transformation?

1. in application: compute new vertexs
2. in vertex shader : send ModelView matrix
3. in vertex shader: send angles

第一种方法

在JS中计算CTM，对图形进行变换生成新顶点，再发送给GPU显示处理。
当顶点数据量大时，IO负荷太大，不采纳。

第二种方法

在JS中计算CTM当前变换矩阵的16个数值并发送给GPU。
传送数据大大减少，着色器中只需要用当前模式变换矩阵计算新顶点。

第三种方法（cube采用方法）

在JS中计算三个方向的旋转角累积量的3个欧拉角数值，并发送给GPU。
传送的数据更少，着色器需要根据参数生成当前变换矩阵，再计算新顶

第三种方法非常适合现代GPU
因为矩阵中三角函数运算在GPU中是硬编码（计算时间几乎可以忽略不计）

方法1 JS中计算新顶点

- 第一种方法
  - 在JS中计算CTM，对图形进行变换生成新顶点再发送给GPU显示处理。(原来的固定流水线法，数据传送量大，不采纳）

//JS中实现的参考伪代码:

```
modelviewMatrix = mat4();
modelviewMatrix=mult(modelviewMatrix ,rotate(theta[xAxis] ));
modelviewMatrix=mult(modelviewMatrix ,rotate(theta[yAxis] ));
modelviewMatrix=mult(modelviewMatrix ,rotate(theta[zAxis] ));
// CTM= modelviewMatrix =I* RXRYRZ

P= modelviewMatrix*P;

Render(P);
```

方法2：JS中计算矩阵并发送GPU

- 在JS中计算CTM当前变换矩阵(16个数值)，并发送给GPU。

```
modelviewMatrix = mat4();//初始化未单位矩阵
```

```
Function render()
{    gl.clear(gl.COLOR_BUFFER_BIT|gl.DEPTH_BUFFER_BIT);
     theta[axis]+=2;
      modelviewMatrix=mult(modelviewMatrix ,rotate(theta[xAxis]));
      modelviewMatrix=mult(modelviewMatrix ,rotate(theta[yAxis]));
      modelviewMatrix=mult(modelviewMatrix ,rotate(theta[zAxis]));

    gl.uniformMatrix4fv(modeViewMatrixLoc,false,flatten(modelviewMatrix));

    gl.drawArrays(gl.TRIANGLES,0,numVertices);
     requesetAnimFrame(render);
}
```

注：flatten把JS中基于行主顺的mat矩阵转换为列主序的Float32Array类型数据发送给vertex shader

方法2：着色器中计算顶点

```
attribute  vec4 vPosition;
attribute  vec4 vColor;
varying vec4 fColor;
uniform mat4 modelViewMatrix;

Void main()
{
      gl_Position = modelViewMatrix * vPosition
}
```

方法3：JS中计算连续旋转角度并发送GPU

- 在JS中计算三个方向的旋转角累积量数组(3个数值)，并发送给shader。

```
function render(){
    gl.clear( gl.COLOR_BUFFER_BIT |gl.DEPTH_BUFFER_BIT);

    //当前旋转轴的旋转角度增加2，其它轴的角度不变
    theta[axis] += 2.0;
    gl.uniform3fv(thetaLoc, theta);//发送角度向量



    gl.drawArrays( gl.TRIANGLES, 0, numVertices );
    requestAnimFrame( render );
}
```

see cube.js

方法3 Shader中计算旋转矩阵

- 顶点着色器中根据输入的旋转角度计算旋转变换矩阵

```
attribute  vec4 vPosition;
attribute  vec4 vColor;
varying vec4 fColor;
uniform vec3 theta;
void main() {
 // Compute the sines and cosines of theta for each of
 //  the three axes in one computation.
     vec3 angles = radians( theta );     //转换为弧度
     vec3 c = cos( angles );
     vec3 s = sin( angles );
```

see cube.html

方法3 Shader中计算旋转矩阵(cont.)

```
    //Remeber: thse matrices are column-major
    mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
            0.0,  c.x,  s.x, 0.0,
            0.0, -s.x,  c.x, 0.0,
            0.0,  0.0,  0.0, 1.0 );

    mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,
                    0.0, 1.0,  0.0, 0.0,
                    s.y, 0.0,  c.y, 0.0,
                    0.0, 0.0,  0.0, 1.0 );


    mat4 rz = mat4( c.z, s.z, 0.0, 0.0,
                   -s.z,  c.z, 0.0, 0.0,
                    0.0,  0.0, 1.0, 0.0,
                    0.0,  0.0, 0.0, 1.0 );

    gl_Position = rz * ry * rx * vPosition;

    fColor = vColor;
}
```

注意矩阵表示

列优先！

see cube.html

Outlines

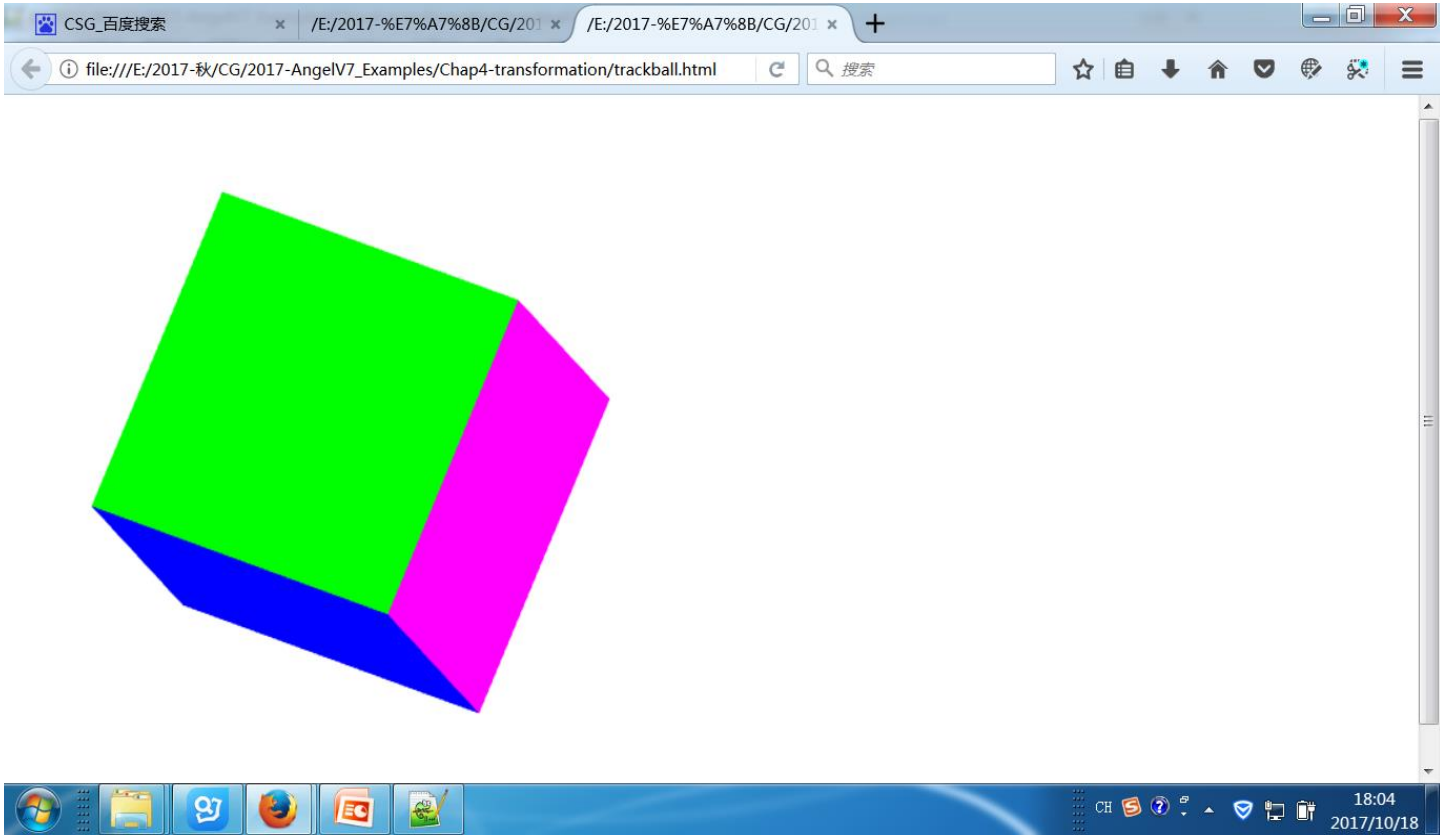
- Transformation Cube（*）
- Analog tracking ball（*）
- Quaternions for Rotation

chap4.6 建模一个彩色立方体
chap 4.11~4.14: WEBGL如何实现变换~cube,trackball实例

Trackball

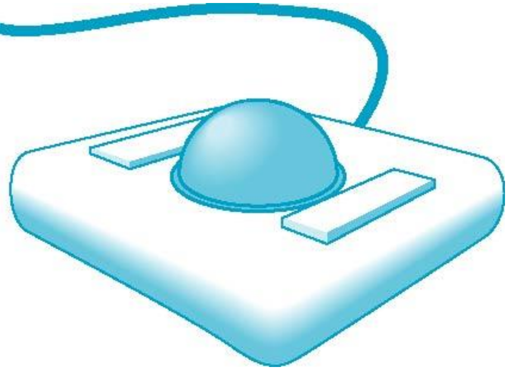
- 按用户希望的方向旋转并且是平滑连贯的旋转

See "trackball-correct"，原书给的例子有错？



Interfaces

- how to use a two-dimensional device（such as a mouse）to interface with three dimensional objects



- Some alternatives
  - Virtual trackball（虚拟跟踪球）
  - 3D input devices such as the spaceball
  - Use areas of the screen

Smooth Rotation

- A Trackball from a Mouse

use transformations to move and reorient an object smoothly

Problem: find a sequence of modelview matrices $\mathbf{M_0},\mathbf{M_1},\ldots\ldots,\mathbf{M_n}$
when they are applied successively to objects, we see a smooth transition.

- Find the axis of rotation and angle of rotation of each Mi = Rotate（angle,axis))

Specifying Rotation

- Mi = Rotate(angle,axis)) Pre 3.1 OpenGL had a function

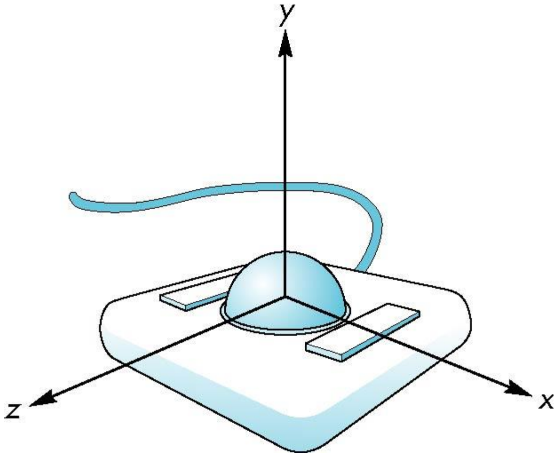
  glRotate (theta, dx, dy dz) //一般CG API 都会提供这一个函数！

the author~Angel implemented glRotate (theta, dx, dy dz) in \common\MV.js

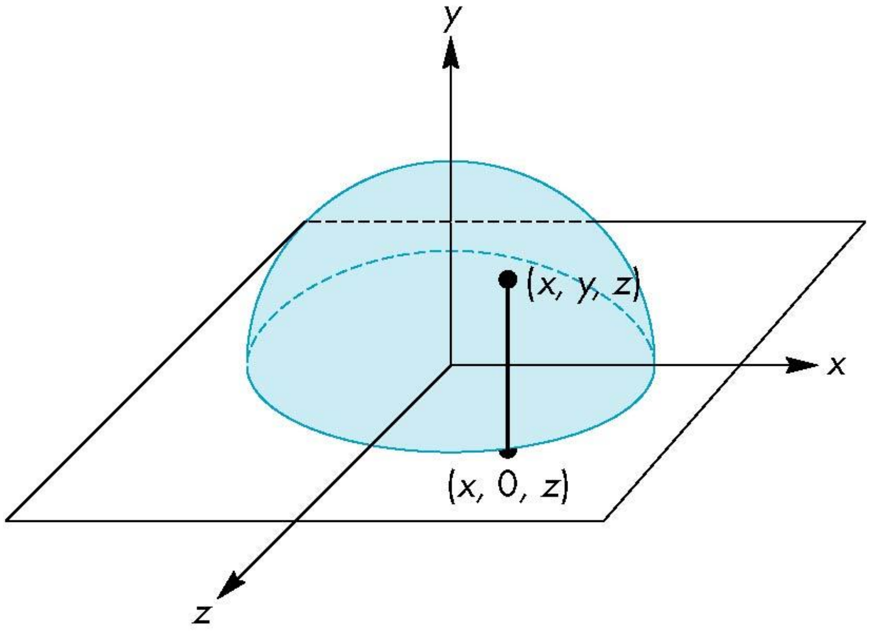
```
/**生成绕任意轴旋转的变换矩阵*/
function rotate( angle, axis )
{
    if ( !Array.isArray(axis) ) {
        axis = [ arguments[1], arguments[2], arguments[3] ];
    }

    var v = normalize( axis );
    var x = v[0];
    var y = v[1];
    var z = v[2];

    var c = Math.cos( radians(angle) );
    var omc = 1.0 - c;
    var s = Math.sin( radians(angle) );

    var result = mat4(
        vec4( x*x*omc + c,   x*y*omc - z*s, x*z*omc + y*s, 0.0 ),
        vec4( x*y*omc + z*s, y*y*omc + c,   y*z*omc - x*s, 0.0 ),
        vec4( x*z*omc - y*s, y*z*omc + x*s, z*z*omc + c,   0.0 ),
        vec4()
    );

    return result;
}
```

Projection of Trackball Position

- relate position on trackball to position on a normalized mouse pad by projecting orthogonally onto pad
  将轨迹球的位置和在正常鼠标垫上的位置联系起来



origin at center of ball



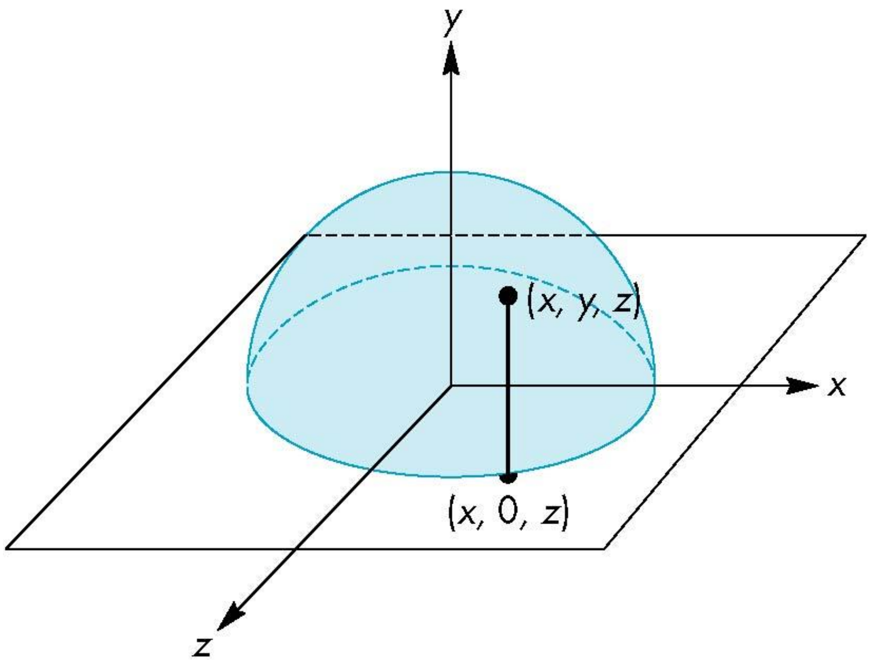
Reversing Projection

A point (x,z) on the mouse pad

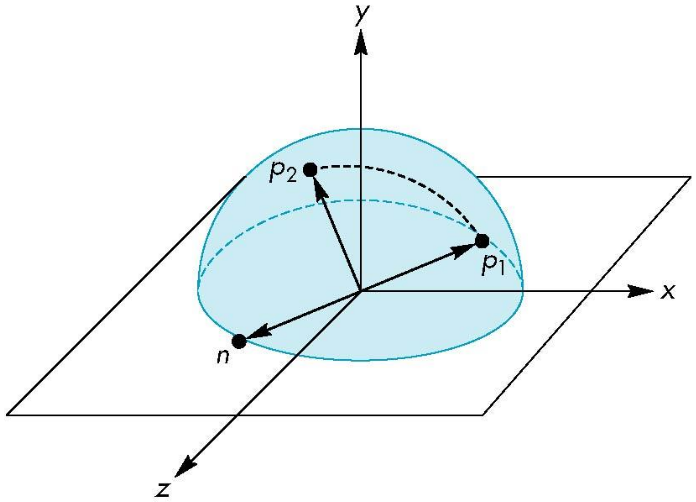
corresponds to

the point (x,y,z) on the upper hemisphere

where $y = \sqrt{r^2 - x^2 - z^2}$ if $r \geq |x| \geq 0$, $r \geq |z| \geq 0$



Computing Rotations



- two points that were obtained from the mouse.
- project them up to the hemisphere to points $\mathbf{p}_1$ and $\mathbf{p}_2$, These points determine a great circle on the sphere.

Rotate from $\mathbf{p}_1$ to $\mathbf{p}_2$ by finding

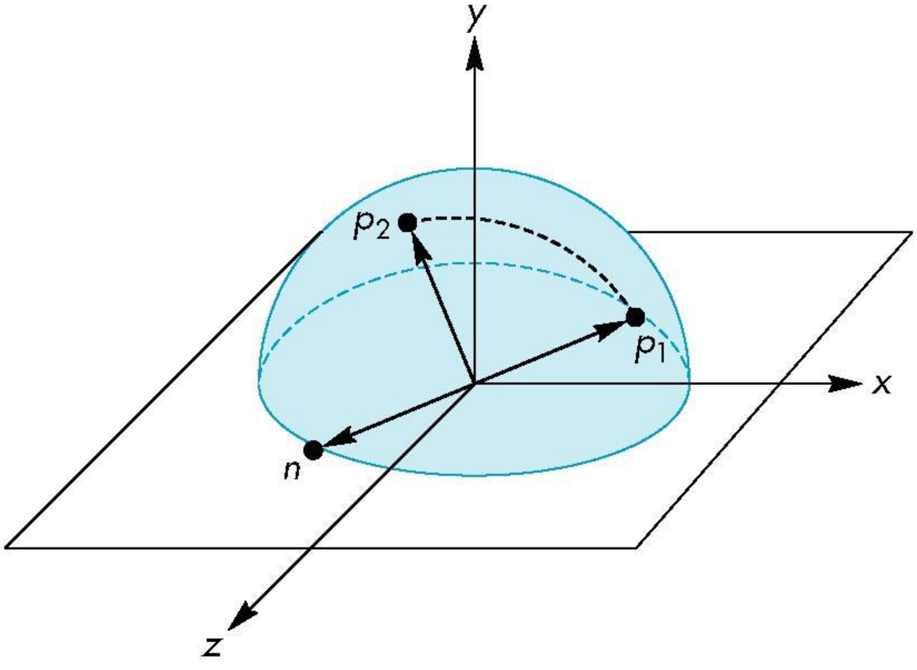
- ➢the proper axis n of rotation
- ➢the angle between the points

Using the cross product

- 旋转轴n:

The axis of rotation is given by the normal to the plane determined by the origin, $\mathbf{p}_1$, and $\mathbf{p}_2$

$$\mathbf{n} = \mathbf{p}_1 \times \mathbf{p}_2$$



Obtaining the angle

- 旋转角度The angle

between $\mathbf{p}_1$ and $\mathbf{p}_2$ is given by

$$|\sin\theta| = \frac{|\mathbf{n}|}{|\mathbf{p}_1||\mathbf{p}_2|}$$

If we move the mouse slowly or sample its position frequently, then θ will be small and we can use the approximation

$$\theta \approx \sin\theta$$

Implementing with WebGL

- Interaction Control~Mouse

```
/***用鼠标键的按下和放开，作为鼠标开始运动和结束运动的触发事件***/
canvas.addEventListener("mousedown", function(event){
  var x = 2*event.clientX/canvas.width-1;
  var y = 2*(canvas.height-event.clientY)/canvas.height-1;
  startMotion(x, y);
});

canvas.addEventListener("mouseup", function(event){
  var x = 2*event.clientX/canvas.width-1;
  var y = 2*(canvas.height-event.clientY)/canvas.height-1;
  stopMotion(x, y);
});
```

```
/***用鼠标键的按下后，开始移动时的触发事件，主要用来获取移动的屏幕位置，转换为半球上的坐标，计算转动轴***/
canvas.addEventListener("mousemove", function(event){

  var x = 2*event.clientX/canvas.width-1;
  var y = 2*(canvas.height-event.clientY)/canvas.height-1;
  mouseMotion(x, y);
} );
```

mouseDown回调函数

- Define actions in terms of two booleans
- **trackingMouse**: if true, update trackball position
- **trackballMove**: if true, update rotation matrix

//开始跟踪鼠标移动:计算新的旋转轴和角度重新绘制图形

```
function startMotion( x,  y)
{
    trackingMouse = true;
    startX = x;
    startY = y;
    //开始位置需要转换为3D半球上的坐标，并保留在lastPos***/
    lastPos = trackballView(x, y);//计算当前结束点的3D位置
    trackballMove=true;//开始跟踪鼠标的移动
}
```

mouseUp回调函数

- Define actions in terms of two booleans
- **trackingMouse**: if true, update trackball position
- **trackballMove**: if true, update rotation matrix

//结束跟踪鼠标移动:不再重新计算，按原来的旋转方向速度继续旋转

```
function stopMotion( x,  y)
{
    trackingMouse = false;//不再跟踪鼠标的移动
    if (startX != x || startY != y) {
        /*这时trackballMove还是true,表示跟踪球继续按先前的速度方向继续转动,
        这时render () 中还是会计算rotationMatrix = mult(rotationMatrix, rotate(angle, axis));
        顶点着色器中的旋转矩阵会继续更新，所以物体还是会继续转动*/
    }
    else {
        /*当鼠标停止移动时，如果鼠标按下时的开始位置start和当前鼠标弹起时的位置相同,
         表示鼠标按下又弹起来了，即鼠标没有作任何移动，物体也不会旋转! */
         angle = 0.0;
         trackballMove = false;//只有鼠标点下和弹起时的坐标相同时，物体才会停止转动
    }
}
```

mouseMotion回调函数

计算旋转轴和旋转角度，并调用绘制图形进行绘制

```
/*mouseMotion当鼠标按下了并移动时，根据新的位置X,Y,转换为半球上三维坐标后，
结合保留的上一次的三维位置，计算得出旋转轴及旋转角度，并且保留本次3D 位置。*/
function mouseMotion( x,  y)
{
    var dx, dy, dz;

    var curPos = trackballView(x, y);
    if(trackingMouse) {
      dx = curPos[0] - lastPos[0];
      dy = curPos[1] - lastPos[1];
      dz = curPos[2] - lastPos[2];

      if (dx || dy || dz) {
           angle = -0.1 * Math.sqrt(dx*dx + dy*dy + dz*dz);
           axis[0] = lastPos[1]*curPos[2] - lastPos[2]*curPos[1];
           axis[1] = lastPos[2]*curPos[0] - lastPos[0]*curPos[2];
           axis[2] = lastPos[0]*curPos[1] - lastPos[1]*curPos[0];

           lastPos[0] = curPos[0];
           lastPos[1] = curPos[1];
           lastPos[2] = curPos[2];
      }
    }
    render();
}
```

Trackball.js

？红色代码是否正确

```
function render(){
      gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

 if(trackballMove) {
//注意：rotate(angle, axis)是本次旋转，rotationMatrix是前面累积的旋转乘积
        axis = normalize(axis);
        rotationMatrix = mult(rotationMatrix, rotate(angle, axis));
        gl.uniformMatrix4fv(rotationMatrixLoc, false, flatten(rotationMatrix));
  }
  gl.drawArrays( gl.TRIANGLES, 0, NumVertices );
   requestAnimFrame( render );
}
```

Trackball shader

```
<script id="vertex-shader" type="x-shader/x-vertex">
attribute  vec4 vPosition;
attribute  vec4 vColor;
varying vec4 fColor;
uniform mat4  rotationMatrix;
Void main(){
        vec4 p;
        gl_Position = rotationMatrix *vPosition;
        fColor = vColor;
}
</script>
```

Outlines

- Transformation Cube（*）
- Analog tracking ball（*）
- Quaternions for Rotation（option）

chap4.6 建模一个彩色立方体
chap 4.11~4.14: WEBGL如何实现变换~cube,trackball实例

Quaternions四元数

- Extension of imaginary numbers from two to three dimensions（虚数的扩展）
- Requires one real and three imaginary components **i**, **j**, **k**（**一个实数，三个虚数**）

$$q=q_0+q_1\mathbf{i}+q_2\mathbf{j}+q_3\mathbf{k}$$

- Quaternions can express rotations on sphere smoothly and efficiently.

  （可有效平滑得表达球体转动）

  四元数处理旋转，Quaternion Rotation

Using Quaternions

- Quaternion arithmetic works well for representing rotations around the origin
- Can use directly avoiding rotation matrices in the virtual trackball
- Code was made available long ago (pre shader) by SGI
- Quaternion shaders are simple

Quaterion Rotation

- 四元数的计算公式

Definition: $a=(q_0,q_1,q_2,q_3)=\left(q_0,\mathbf{q}\right)$

Quaterian Arithmetic: $a+b=(a_0+b_0,\mathbf{a}+\mathbf{b})$

$$ab=(a_0b_0-\mathbf{a}\bullet\mathbf{b},\ a_0\mathbf{b}+b_0\mathbf{a}+\mathbf{a}\times\mathbf{b})$$

$$|a|^2=(q_0^2,\mathbf{q}\bullet\mathbf{q}) \qquad a^{-1}=\frac{1}{|a|^2}\left(q_0,\textbf{-q}\right)$$

- 四元数表示:

Representing a 3D point: $p=\left(0,\mathbf{p}\right)$

Representing a Rotation: $r=\left(\cos\frac{\theta}{2},\sin\frac{\theta}{2}\mathbf{v}\right)$

Rotating a Point: $p'=rpr^{-1}$

Incremental Rotation

- Consider the two approaches
  - For a sequence of rotation matrices $\mathbf{R_0},\mathbf{R_1},\ldots\ldots,\mathbf{R_n}$,
  - find the Euler angles for each and use $\mathbf{R_i} = \mathbf{R_{iz}}\,\mathbf{R_{iy}}\,\mathbf{R_{ix}}$
    - Not very efficient
  - Use the final positions to determine the axis and angle of rotation, then increment only the angle
- *Quaternions can be more efficient than either*

trackballQuaterion.js


```
function render(){
  gl.clear( gl.COLOR_BUFFER_BIT |
  gl.DEPTH_BUFFER_BIT);
  if(trackballMove) {
        axis = normalize(axis);
        var c = Math.cos(angle/2.0);
        var s = Math.sin(angle/2.0);
        var rotation = vec4(c, s*axis[0], s*axis[1], s*axis[2]);
        rotationQuaternion = multq(rotationQuaternion, rotation);
          gl.uniform4fv(rotationQuaternionLoc,flatten(rotationQuaternion));
   }
   gl.drawArrays( gl.TRIANGLES, 0, NumVertices );
  requestAnimFrame( render );
}
```

$$r = \left(\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\mathbf{v} \right)$$

Vertex Shader

- 传递入shader的是一次旋转矩阵的四元数表示 r

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform vec4 rquat; // rotation quaternion

//定义四元数乘法quaternion multiplier
//
```

$$\boxed{ab = \left(a_0 b_0 - \mathbf{a} \bullet \mathbf{b},\ a_0 \mathbf{b} + b_0 \mathbf{a} + \mathbf{a} \times \mathbf{b}\right)}$$

```
vec4 multq(vec4 a, vec4 b)
{
  return(vec4(a.x*b.x - dot(a.yzw, b.yzw),
      a.x*b.yzw+b.x*a.yzw+cross(b.yzw, a.yzw)));
}
```

Vertex Shader(cont.)

- **用四元数表示的旋转矩阵rquat,实现顶点的旋转变换**

$$a^{-1} = \frac{1}{|a|^2}\left(q_0, \mathbf{-q}\right)$$

$$p' = rpr^{-1}$$

```
// 定义四元数求逆inverse quaternion, 为了求r-1
vec4 invq(vec4 a)
{ return(vec4(a.x, -a.yzw)/dot(a,a)); }

 void main() {
 vec3 axis = rquat.yxw;
 float theta = rquat.x;
 vec4 r, p;
 p = vec4(0.0, vPosition.xyz);  // input point quaternion
 p = multq(rquat, multq(p, invq(rquat))); // rotated point quaternion
 gl_Position = vec4( p.yzw, 1.0); // back to homogeneous coordinates
 color = vColor;
}
```

Working with Quaternians

- Quaternion arithmetic works well for representing rotations around the origin
- There is no simple way to convert a quaternion to a matrix representation
- Usually copy elements back and forth between quaternions and matrices
- Can use directly without rotation matrices in the virtual trackball
- Quaternion shaders are simple

Quaternions and Computer Graphics

- (Re)discovered by both aerospace and animation communities
- Used for head mounted display in virtual and augmented reality
- Used for smooth camera paths
- Caveat(警告):
  - quaternions do not preserve up direction