Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

The University of New Mexico

Programming with WebGL
Part 5: More GLSL

Ed Angel

Professor Emeritus of Computer Science

University of New Mexico

Objectives

- Coupling shaders to applications
    - Reading
    - Compiling
    - Linking
- Vertex Attributes
- Setting up uniform variables
- Example applications

The University of New Mexico

Linking Shaders with Application

- Read shaders
- Compile shaders
- Create a program object
- Link everything together
- Link variables in application with variables in shaders
  - Vertex attributes
  - Uniform variables

Program Object

- Container for shaders
  - Can contain multiple shaders
  - Other GLSL functions

```
var program = gl.createProgram();

gl.attachShader( program, vertShdr );
gl.attachShader( program, fragShdr );
gl.linkProgram( program );
```

The University of New Mexico

Reading a Shader

- Shaders are added to the program object and compiled
- Usual method of passing a shader is as a null-terminated string using the function
- gl.shaderSource(fragShdr, fragElem.text);
- If shader is in HTML file, we can get it into application by getElementById method
- If the shader is in a file, we can write a reader to convert the file to a string

Adding a Vertex Shader

```
var vertShdr;
var vertElem =
   document.getElementById( vertexShaderId );

vertShdr = gl.createShader( gl.VERTEX_SHADER );

gl.shaderSource( vertShdr, vertElem.text );
gl.compileShader( vertShdr );

// after program object created
gl.attachShader( program, vertShdr );
```

Shader Reader

- Following code may be a security issue with some browsers if you try to run it locally
  - Cross Origin Request

```
function getShader(gl, shaderName, type) {
        var shader = gl.createShader(type);
        shaderScript = loadFileAJAX(shaderName);
        if (!shaderScript) {
            alert("Could not find shader source:
                    "+shaderName);
        }
}
```

Precision Declaration

- In GLSL for WebGL we must specify desired precision in fragment shaders
  - artifact inherited from OpenGL ES
  - ES must run on very simple embedded devices that may not support 32-bit floating point
  - All implementations must support mediump
  - No default for float in fragment shader
- Can use preprocessor directives (#ifdef) to check if highp supported and, if not, default to mediump

Pass Through Fragment Shader

```
#ifdef GL_FRAGMENT_SHADER_PRECISION_HIGH
  precision highp float;
#else
  precision mediump float;
#endif

varying vec4 fcolor;
void main(void)
{
    gl_FragColor = fcolor;
}
```

Introduction to Computer Graphics with WebGL

The University of New Mexico

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

Programming with WebGL
Part 6: Three Dimensions

Ed Angel

Professor Emeritus of Computer Science

University of New Mexico

Objectives

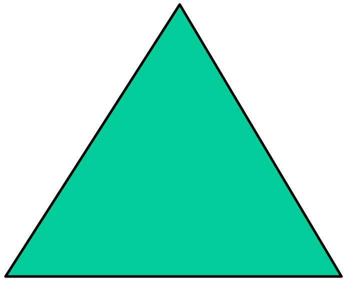
- Develop a more sophisticated three-dimensional example
  - Sierpinski gasket: a fractal
- Introduce hidden-surface removal

Three-dimensional Applications

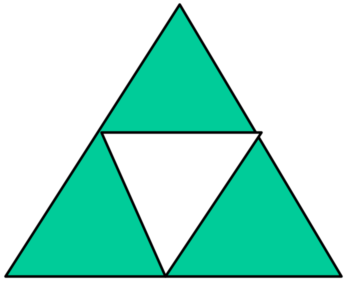
- In WebGL, two-dimensional applications are a special case of three-dimensional graphics
- Going to 3D
  - Not much changes
  - Use **`vec3, gl.uniform3f`**
  - Have to worry about the order in which primitives are rendered or use hidden-surface removal

Sierpinski Gasket (2D)

- Start with a triangle



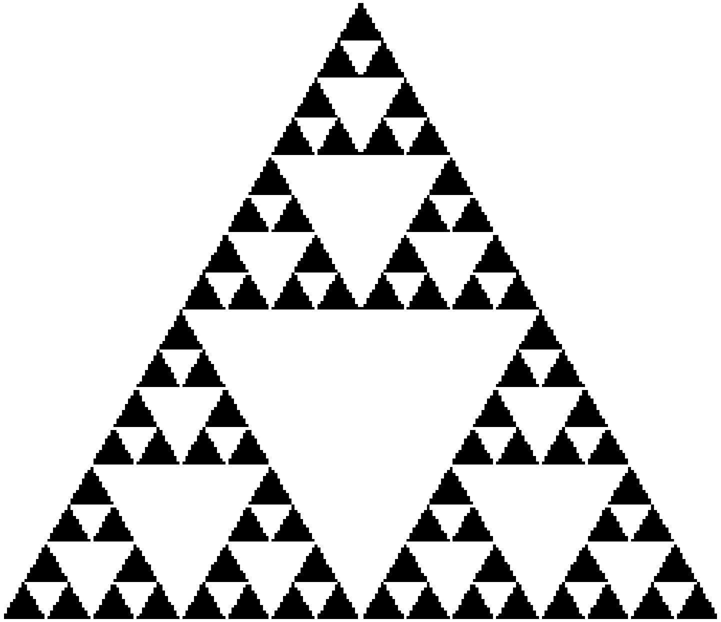
- Connect bisectors of sides and remove central triangle



- Repeat

Example

- Five subdivisions



The gasket as a fractal

- Consider the filled area (black) and the perimeter (the length of all the lines around the filled triangles)
- As we continue subdividing
  - the area goes to zero
  - but the perimeter goes to infinity
- This is not an ordinary geometric object
  - It is neither two- nor three-dimensional
- It is a *fractal* (fractional dimension) object

Gasket Program

- HTML file
  - Same as in other examples
  - Pass through vertex shader
  - Fragment shader sets color
  - Read in JS file

Gasket Program

```
var points = [];
var NumTimesToSubdivide = 5;

/* initial triangle */

 var vertices = [
        vec2( -1, -1 ),
        vec2(  0,  1 ),
        vec2(  1, -1 )
    ];

divideTriangle( vertices[0],vertices[1],
     vertices[2], NumTimesToSubdivide);
```

Draw one triangle

```
/* display one triangle  */

function triangle( a, b, c ){
    points.push( a, b, c );
}
```

Triangle Subdivision

```
function divideTriangle( a, b, c, count ){
 // check for end of recursion
    if ( count === 0 ) {
    triangle( a, b, c );
    }
    else {
//bisect the sides
    var ab = mix( a, b, 0.5 );
    var ac = mix( a, c, 0.5 );
    var bc = mix( b, c, 0.5 );
    --count;
// three new triangles
    divideTriangle( a, ab, ac, count-1 );
    divideTriangle( c, ac, bc, count-1 );
    divideTriangle( b, bc, ab, count-1 );
    }
}
```

init()

```
var program = initShaders( gl, "vertex-
 shader", "fragment-shader" );
    gl.useProgram( program );
    var bufferId = gl.createBuffer();
 gl.bindBuffer( gl.ARRAY_BUFFER, bufferId )
 gl.bufferData( gl.ARRAY_BUFFER,
 flatten(points), gl.STATIC_DRAW );
var vPosition = gl.getAttribLocation(
 program, "vPosition" );
    gl.vertexAttribPointer( vPosition, 2,
 gl.FLOAT, false, 0, 0 );
    gl.enableVertexAttribArray( vPosition );
    render();
```

Render Function

```
function render(){
    gl.clear( gl.COLOR_BUFFER_BIT );
    gl.drawArrays( gl.TRIANGLES, 0, points.length
 );
}
```

The University of New Mexico

Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

Programming with WebGL
Part 6: Three Dimensions

Ed Angel

Professor Emeritus of Computer Science

University of New Mexico

Moving to 3D

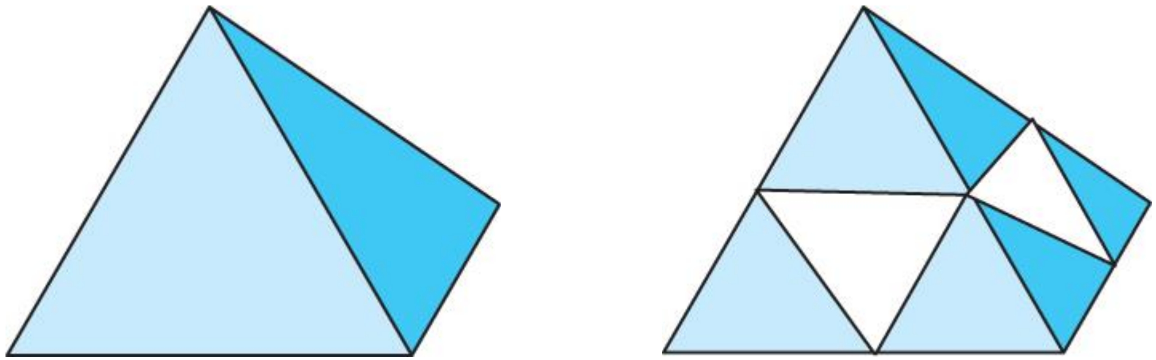
- We can easily make the program three-dimensional by using three dimensional points and starting with a tetrahedron

```
var vertices = [
  vec3(  0.0000,  0.0000, -1.0000 ),
  vec3(  0.0000,  0.9428,  0.3333 ),
  vec3( -0.8165, -0.4714,  0.3333 ),
  vec3(  0.8165, -0.4714,  0.3333 )    ];
```

subdivide each face

3D Gasket

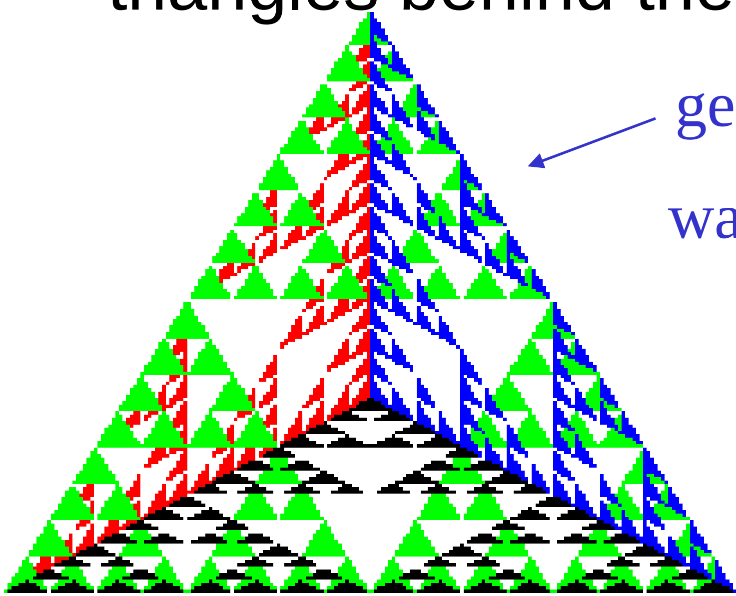
- We can subdivide each of the four faces



- Appears as if we remove a solid tetrahedron from the center leaving four smaller tetrahedra
- Code almost identical to 2D example

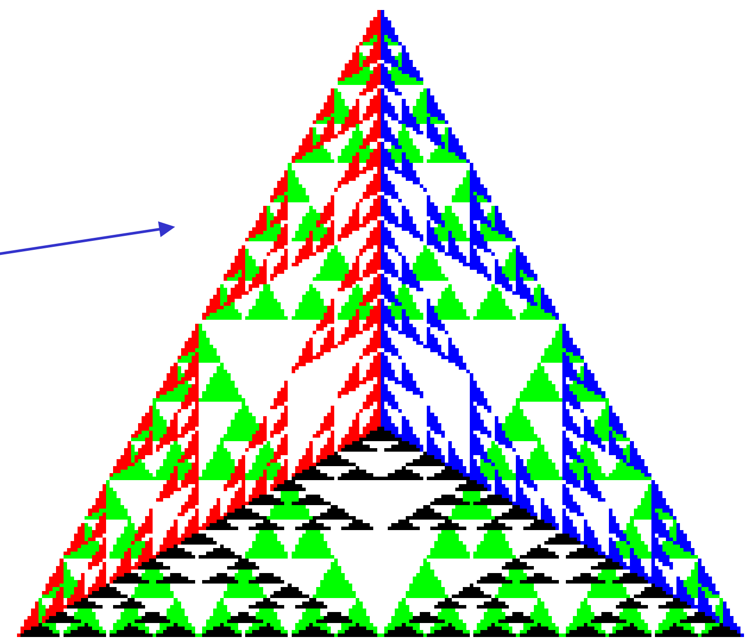
Almost Correct

- Because the triangles are drawn in the order they are specified in the program, the front triangles are not always rendered in front of triangles behind them



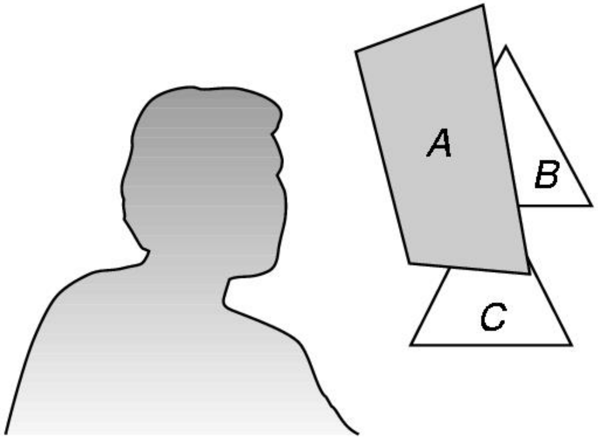
get this

want this



Hidden-Surface Removal

- We want to see only those surfaces in front of other surfaces
- OpenGL uses a *hidden-surface* method called the *z*-buffer algorithm that saves depth information as objects are rendered so that only the front objects appear in the image



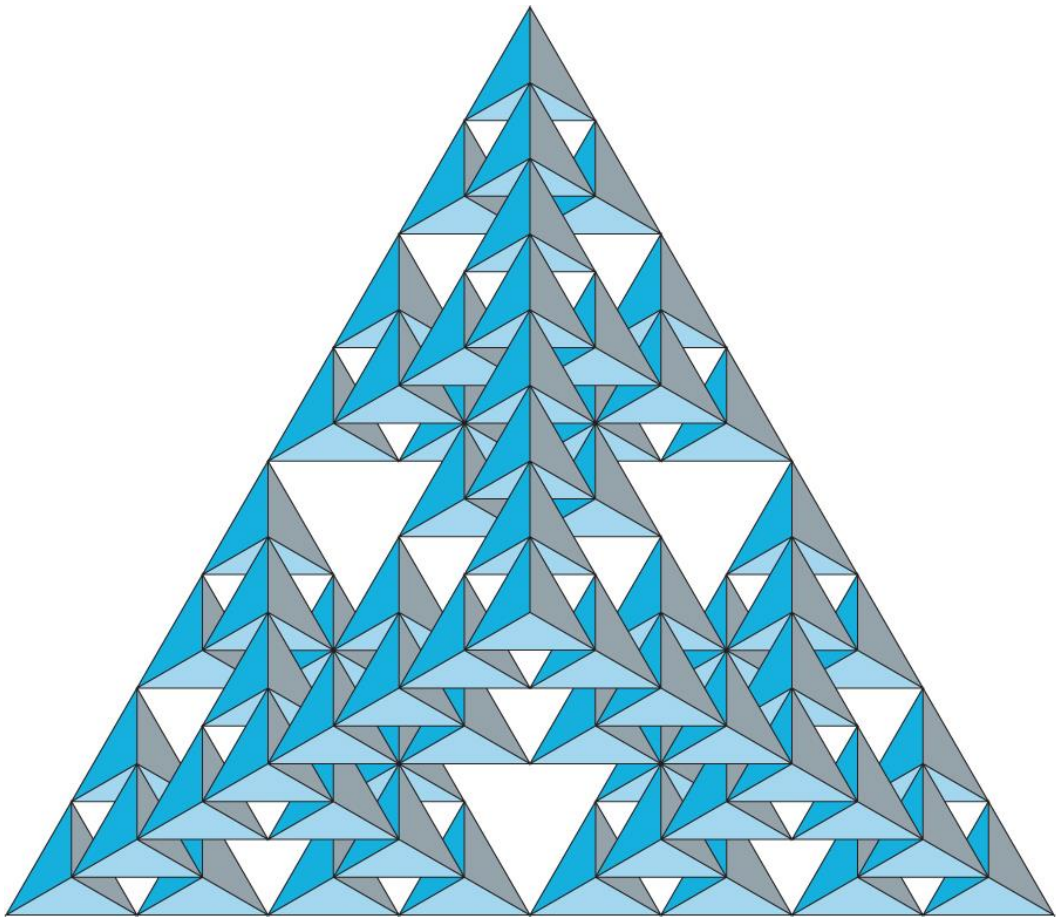
Using the *z*-buffer algorithm

- The algorithm uses an extra buffer, the z-buffer, to store depth information as geometry travels down the pipeline
- Depth buffer is required to be available in WebGL
- It must be
  - Enabled
    - `gl.enable(gl.DEPTH_TEST)`
  - Cleared in for each render
    - `gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT)`

Surface vs Volume Subdvision

- In our example, we divided the surface of each face
- We could also divide the volume using the same midpoints
- The midpoints define four smaller tetrahedrons, one for each vertex
- Keeping only these tetrahedrons removes a *volume* in the middle
- See text for code

Volume Subdivision



Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

Incremental and Quaternion Rotation

Ed Angel
Professor Emeritus of Computer Science
University of New Mexico

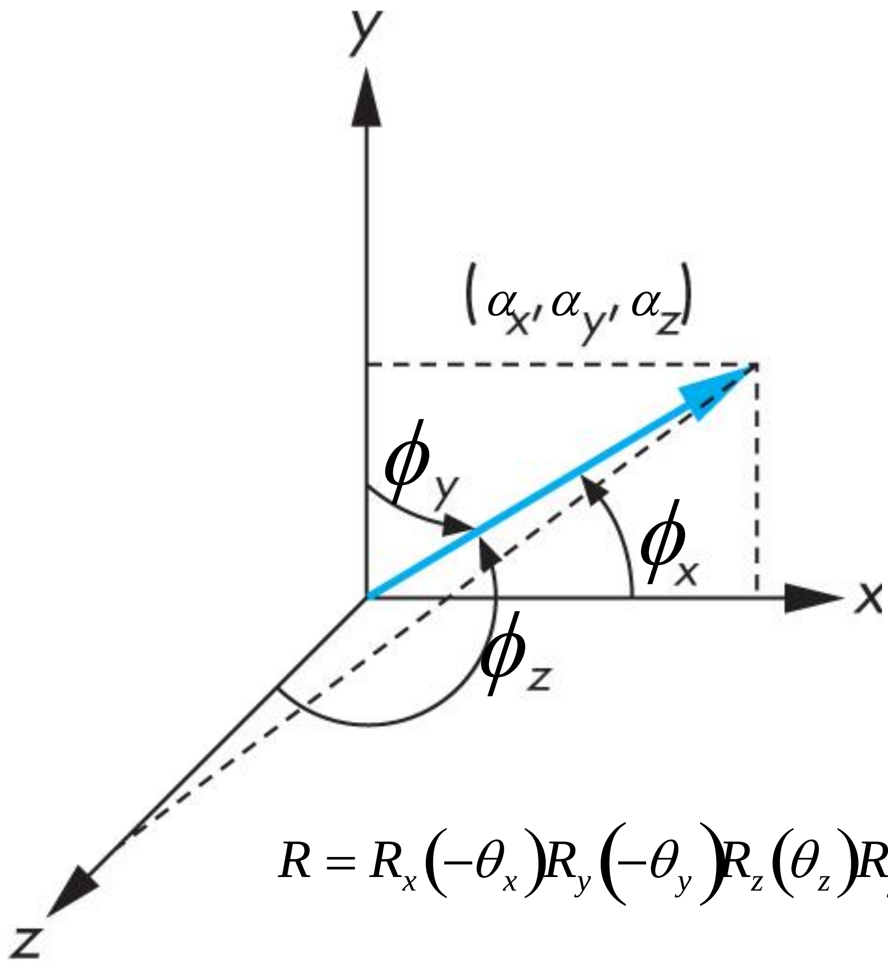
Objectives

- This is an optional lecture that
  - Illustrates the difference between using direction angles and Euler angles
  - Considers issues with incremental rotation
  - Introduces quaternions as an alternate to rotation matrices

Specifying a Rotation

- Pre 3.1 OpenGL had a function glRotate (theta, dx, dy dz) which incrementally changed the current rotation matrix by a rotation with fixed point of the origin about a vector in the direction (dx, dy, dz)
- We implemented rotate in MV.js
- Implementations of Rotate often decompose the general rotation into a sequence of rotations about the coordinate axes as in Chapter 4.

Euler from Direction Angles



$$R = R_x(-\theta_x)R_y(-\theta_y)R_z(\theta_z)R_y(\theta_y)R_x(\theta_x)$$

Efficiency

$$R = R_x(-\theta_x)R_y(-\theta_y)R_z(\theta_z)R_y(\theta_y)R_x(\theta_x)$$

should be able to write as

$$R = R_x(\varphi_x)R_y(\varphi_y)R_z(\varphi_z)$$

If we knew the angles, we could use RotateX, RotateY and RotateZ from mat.h

But is this an efficient method?
No, we can do better with quaterions

Incremental Rotation

$$R(t+dt) = R(t)R_z(\theta_z)R_y(\theta_y)R_x(\theta_x)$$

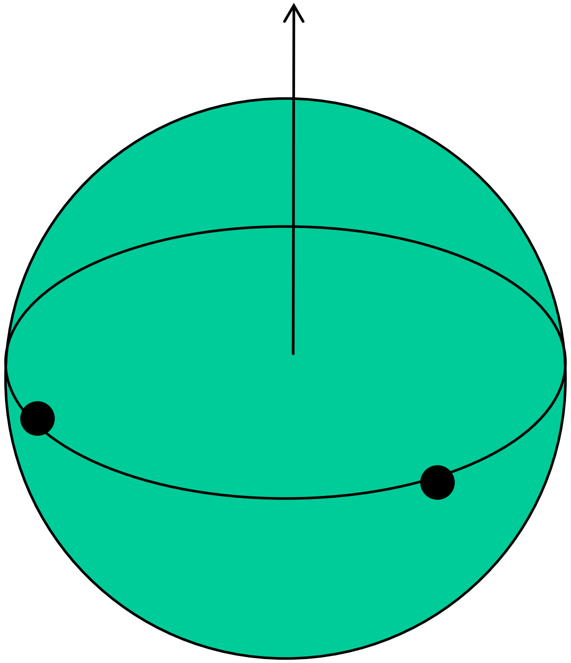
where θ_x, θ_y and θ_z are small angles

For small angles $\sin\theta \approx \theta$

$$\cos\theta \approx 1$$

$$R_z(\theta_z)R_y(\theta_y)R_x(\theta_x) \approx \begin{bmatrix} 1 & -\theta_z & \theta_y & 0 \\ \theta_z & 1 & -\theta_x & 0 \\ -\theta_y & \theta_x & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Great Circles



Rotation and Great Circles

- Shortest path between two points on a sphere is the great circle passing through the two points
- Corresponding to each great circle is vector normal to the circle
- Rotation about this vector carries us from the first point to the second

The University of New Mexico

Quaternion Rotation

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

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

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

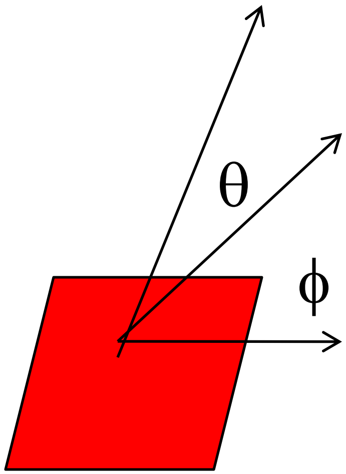
$$|a|^2 = (q_0^2, \mathbf{q} \bullet \mathbf{q}) \qquad a^{-1} = \frac{1}{|a|^2}(q_0, \mathbf{-q})$$

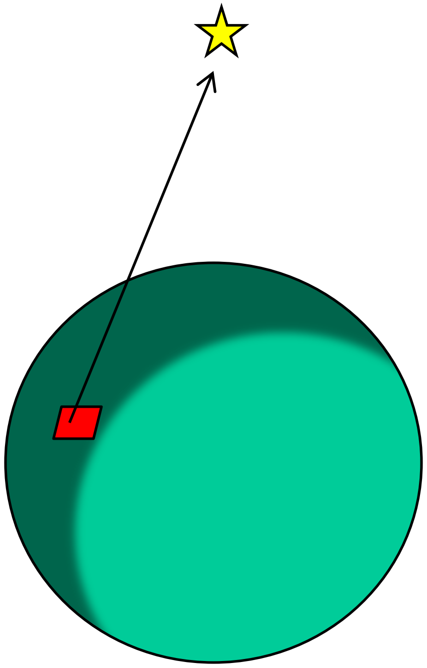
Representing a 3D point: $p = (0, \mathbf{p})$

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

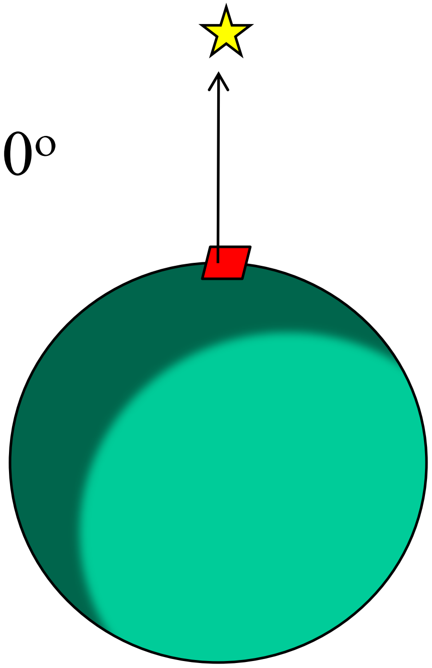
Rotating a Point: $p' = r p r^{-1}$

Looking at the North Star





At North Pole



$\theta = 90^o$

$\phi = ?$

Gimbal Lock

- Suppose you rotate about the y axis by 90º
- This action removes a degree of freedom

$$R_z(\theta_z)R_y(\theta_y)R_x(\theta_x) \approx \begin{bmatrix} 0 & \sin(\theta_x - \theta_z) & \cos(\theta_x - \theta_z) & 0 \\ 0 & \cos(\theta_x - \theta_z) & -\sin(\theta_x - \theta_z) & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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: quaternions do not preserve up direction

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