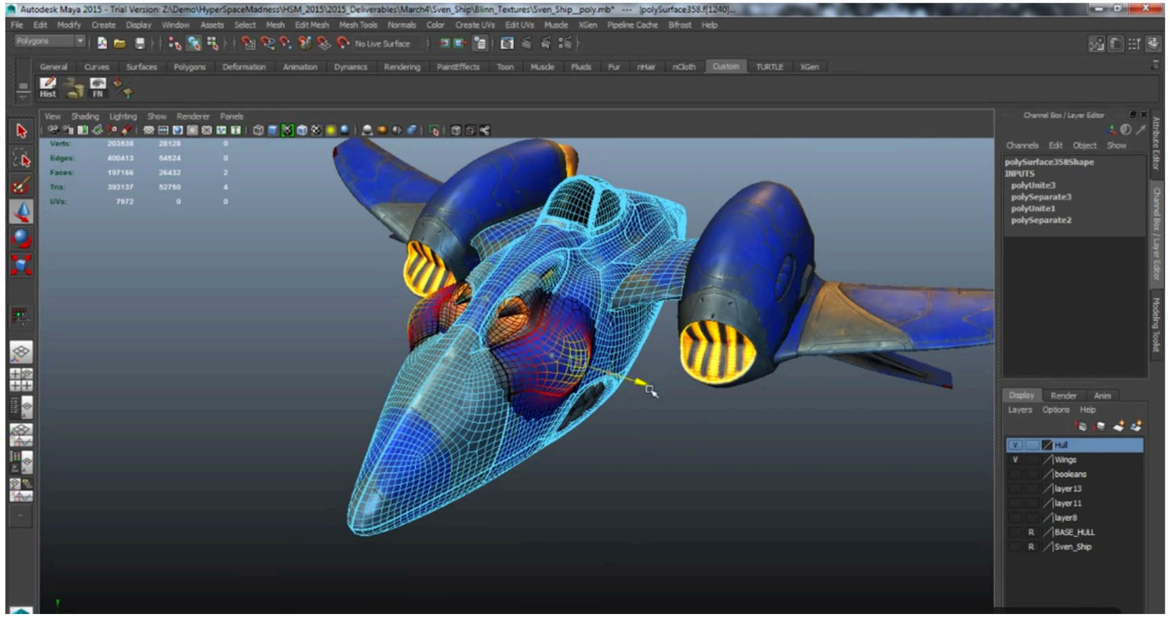
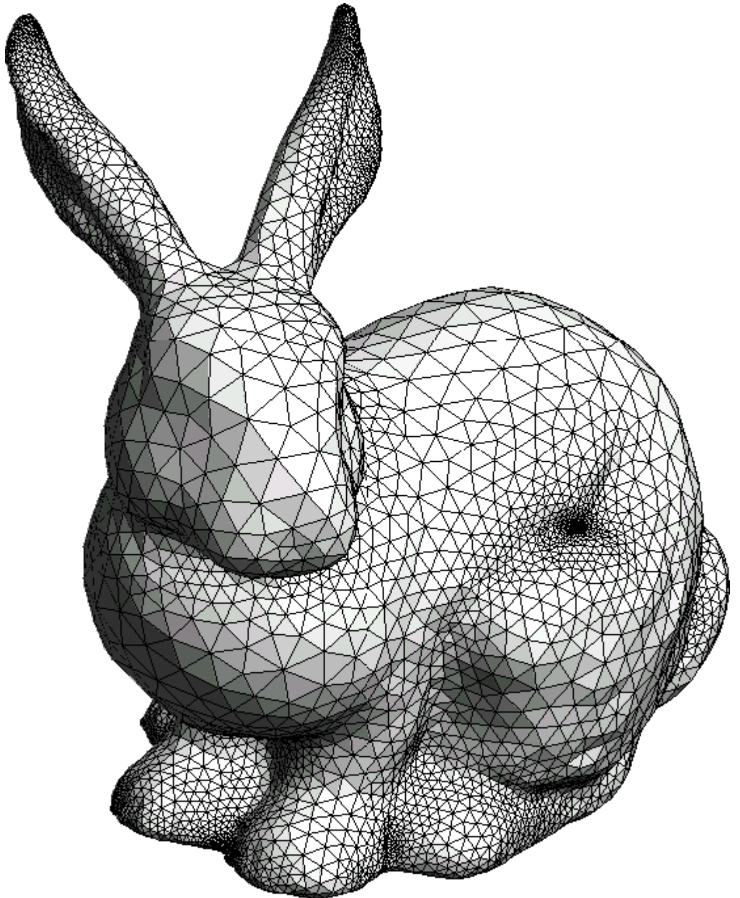
Geometric Modeling

- You may have noticed we have drawn only simple shapes
- Geometric modeling is not easy
- Even with sophisticated computational tools like Maya
    - Still labor intensive



Geometric Modeling

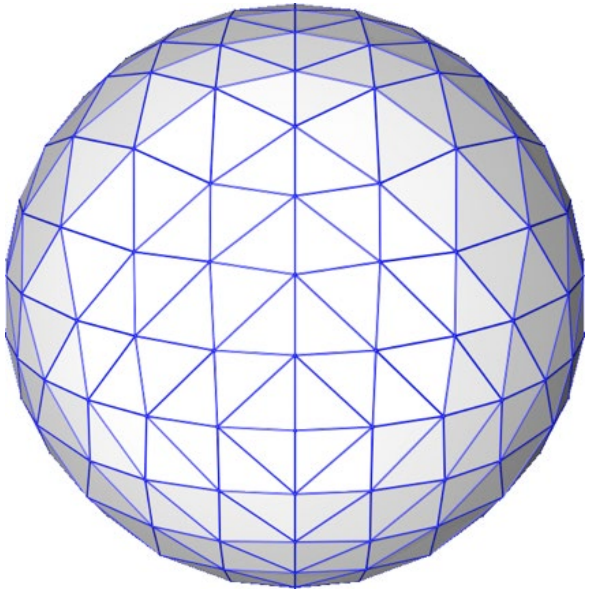
- So how do you get geometric models for this course?
- Well…you can get models in files and implement a file reader
  - Some browsers prevent file reading for security reasons
  - We'll see how to work around that later



Geometric Modeling

- You can type the geometry in by hand
  - Hard code it into the .js file
  - Useful for testing
  - Obviously not scalable

- You can procedurally generate geometry
  - Write code to produce a bunch of triangles
  - We'll write code to do that for one type of surface today

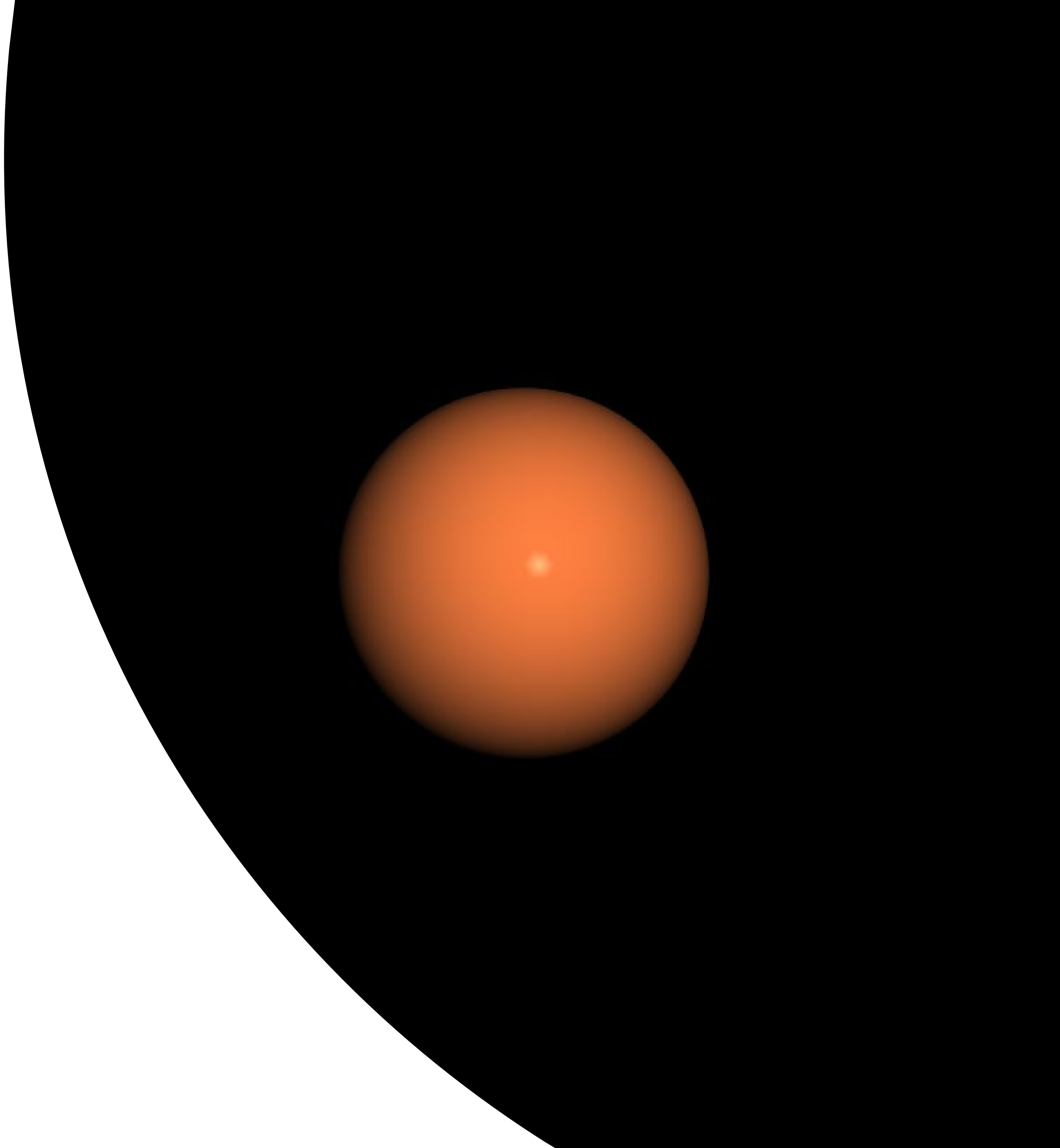


Generating a Mesh for a Sphere

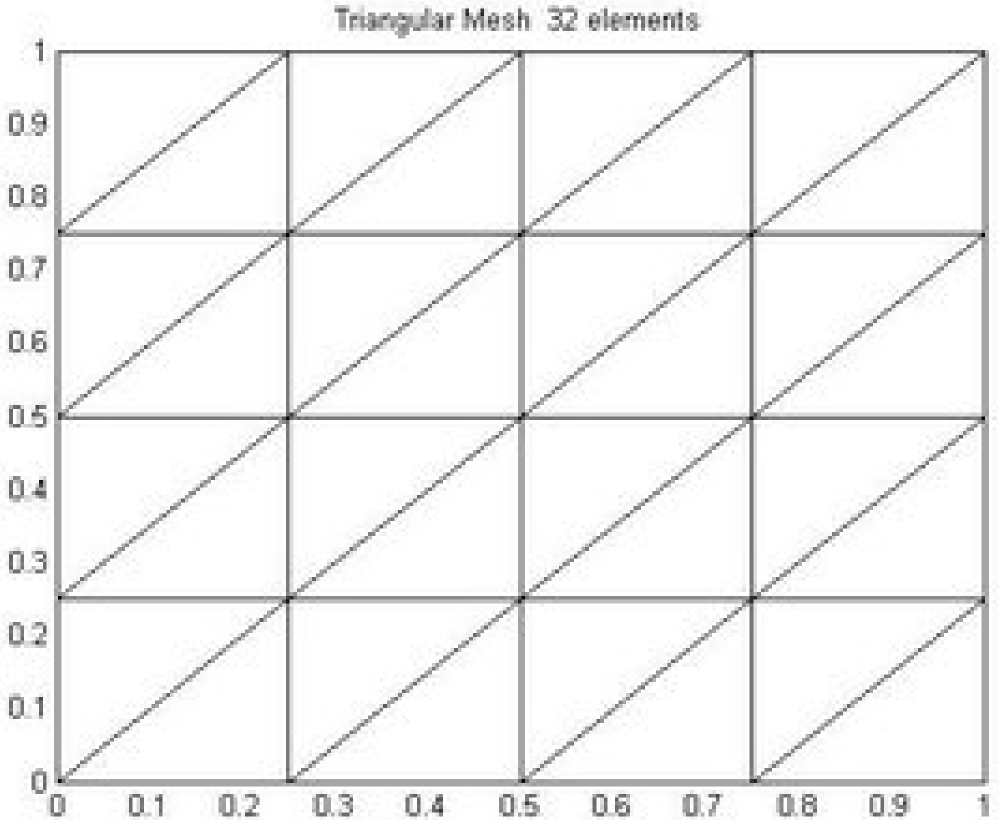
Today we’ll review our first really 3D image

Provided code will do two important things

- It will render the scene in perspective
- It will shade the sphere as if lit by a light-source



Generating a Tessellated Quadrilateral



- Let’s look at how to generate the triangles for a simpler shape
- Technically, a **tessellation** is a tiling of a plane using geometric shapes
- We will divide a rectangle up into triangles

Generating a Tessellated Quad

Recursion (or iteration) can be used to generate refined geometry

- Meaning lots of triangles…

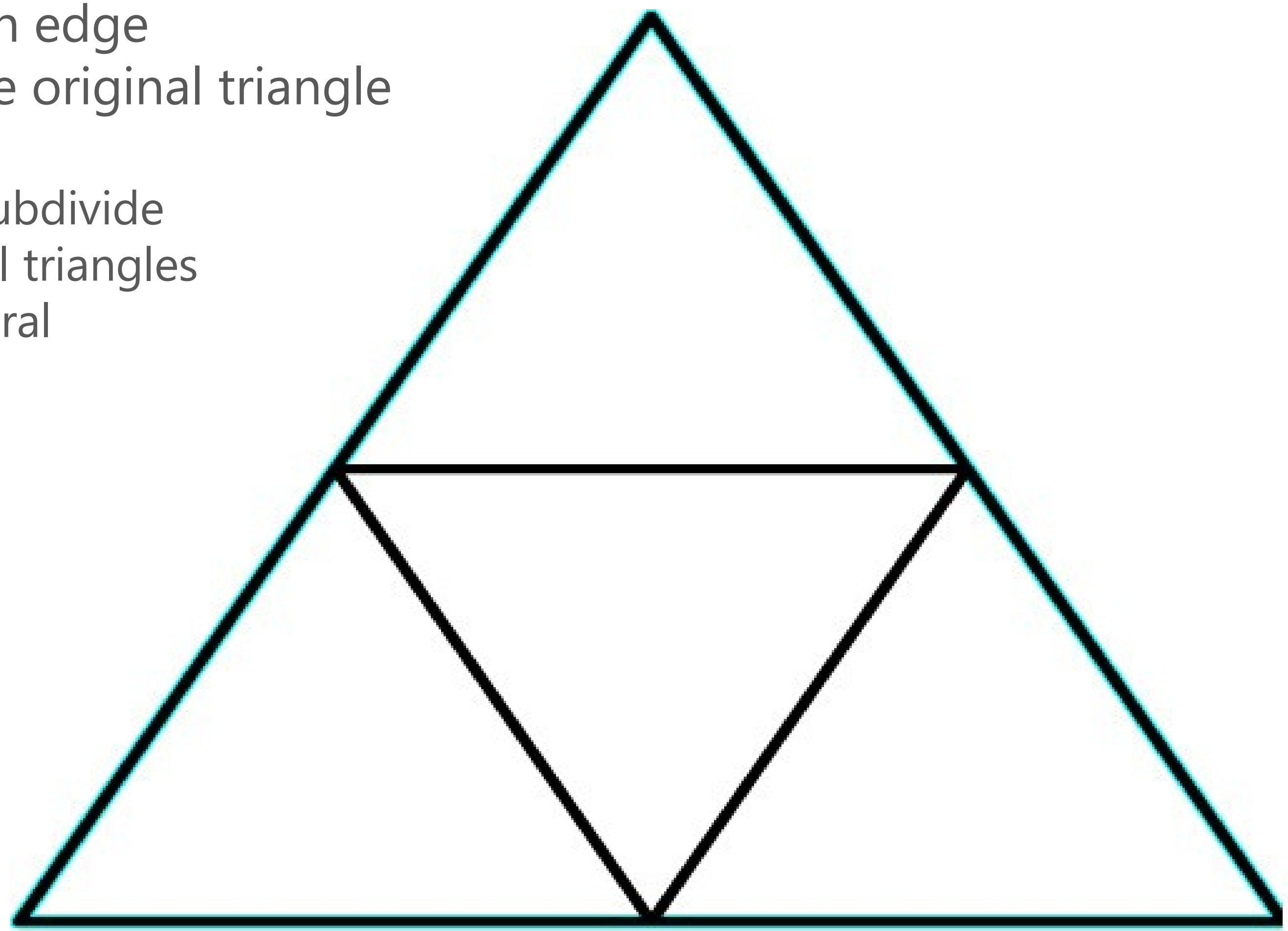
How can we generate a tessellated plane recursively?
What does this code do?

```
function planeFromSubdivision(n, minX,maxX,minY,maxY, vertexArray)
{
    var numT=0;
    var va = vec4.fromValues(minX,minY,0,1);
    var vb = vec4.fromValues(maxX,minY,0,1);
    var vc = vec4.fromValues(maxX,maxY,0,1);
    var vd = vec4.fromValues(minX,maxY,0,1);

    numT+=divideTriangle(va,vb,vd,n, vertexArray);
    numT+=divideTriangle(vb,vc,vd,n, vertexArray);
    return numT;

}
```

Subdividing a Triangle

1. Find the midpoints of each edge
2. Create 4 triangles from the original triangle

There are other ways you could subdivide
This method generates equilateral triangles
...if you start out with an equilateral



Subdividing a Triangle

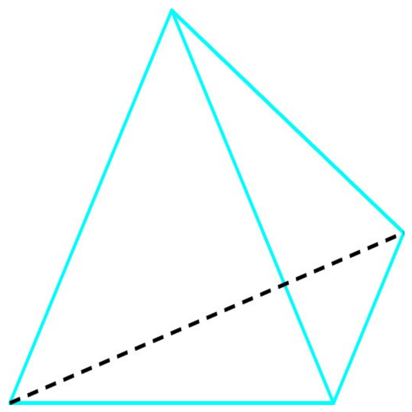
```
function divideTriangle(a,b,c,numSubDivs, vertexArray){
    if (numSubDivs>0){
        var numT=0;
        var ab =  vec4.create(); vec4.lerp(ab,a,b,0.5);
        var ac =  vec4.create(); vec4.lerp(ac,a,c,0.5);
        var bc =  vec4.create(); vec4.lerp(bc,b,c,0.5);

        numT+=divideTriangle(a,ab,ac,numSubDivs-1, vertexArray);
        numT+=divideTriangle(ab,b,bc,numSubDivs-1, vertexArray);
        numT+=divideTriangle(bc,c,ac,numSubDivs-1, vertexArray);
        numT+=divideTriangle(ab,bc,ac,numSubDivs-1, vertexArray);
        return numT;
    }
    else …
```

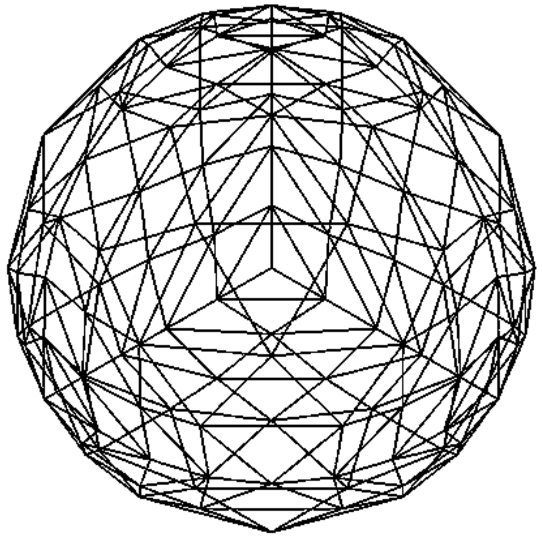
Subdividing a Triangle

```
else
    {
        // Add 3 vertices to the array
        pushVertex(a,vertexArray);
        pushVertex(b,vertexArray);
        pushVertex(c,vertexArray);
        return 1;

    }
}

function pushVertex(v, vArray)
{
 for(i=0;i<3;i++)
 {
     vArray.push(v[i]);
 }
}
```

Generating a Sphere

1. Start with a tetrahedron centered on the origin
2. Vertices at a distance of 1 from the origin
3. Recursively subdivide the triangular faces
4. Normalize the new vertices that get introduced
   normalize means move the vertex to a distance of 1 around the origin
   keep same direction



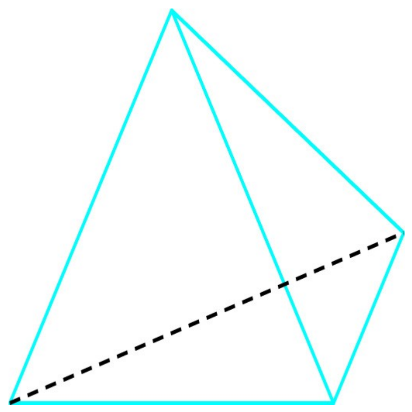


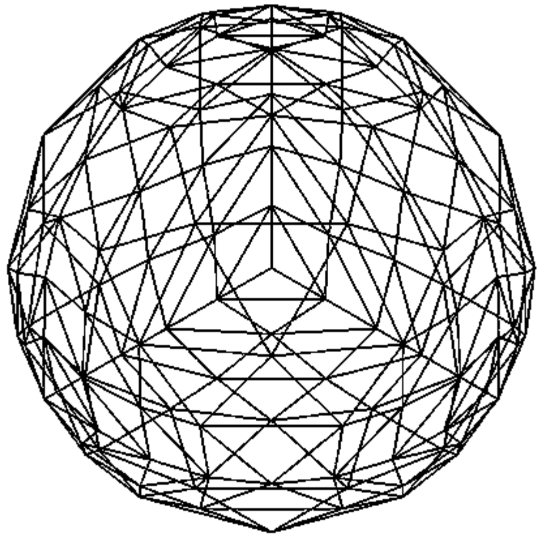
Generating a Sphere

In the file `SimpleModeling.js`

`sphDivideTriangle(a,b,c,numSubDivs, vertexArray,normalArray)`

- `a` `b` and `c` are the corners of triangle…each is a vec4 specifying (x,y,z,w)
- `numSubDivs` is how many times to subdivide the triangles
- The `vertexArray` is where you add the vertices you compute
- The `normalArray` is where you add per-vertex normal vectors





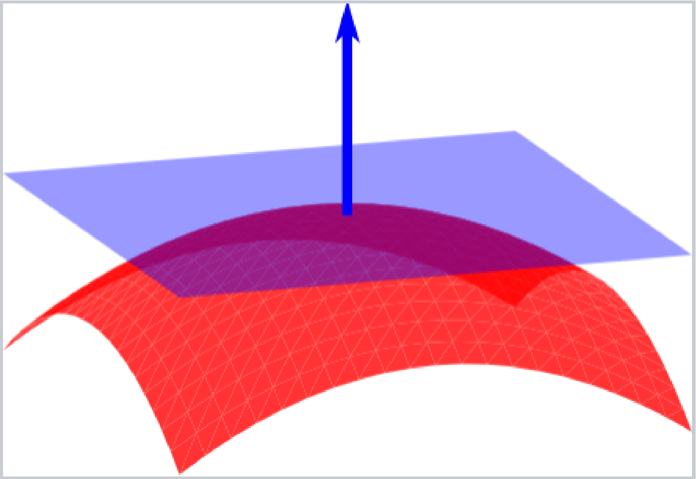
Generating Normals

To ***shade*** the mesh we need ***per-vertex normal vectors***

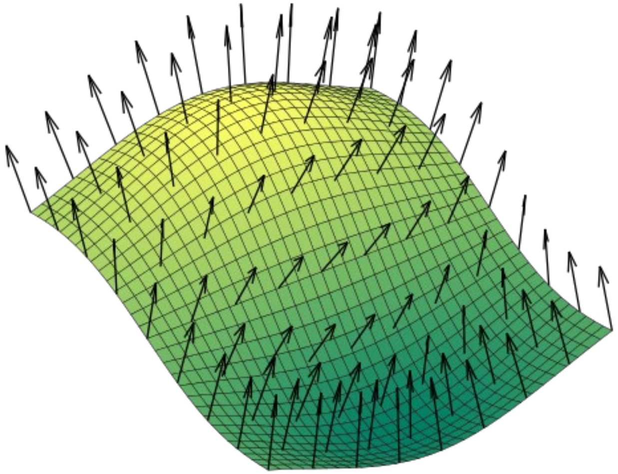
Normal vectors are vectors perpendicular to the surface

For the sphere, what are the normals?

In general, if we have no other information about a surface mesh,
we compute a normal for each vertex
by averaging the normal of the surrounding faces



A normal to a surface at a point is the same as a normal to the tangent plane to the same surface at the same point.

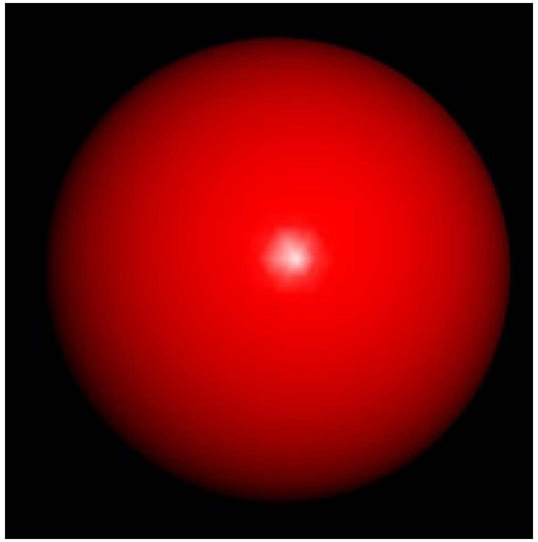


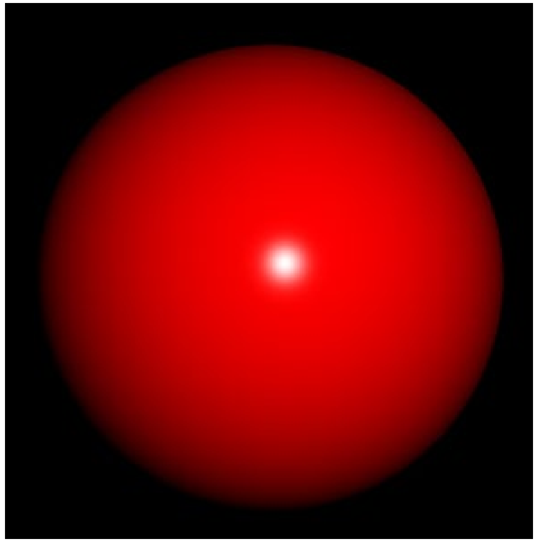
Phong Shading

You also need to implement Phong shading

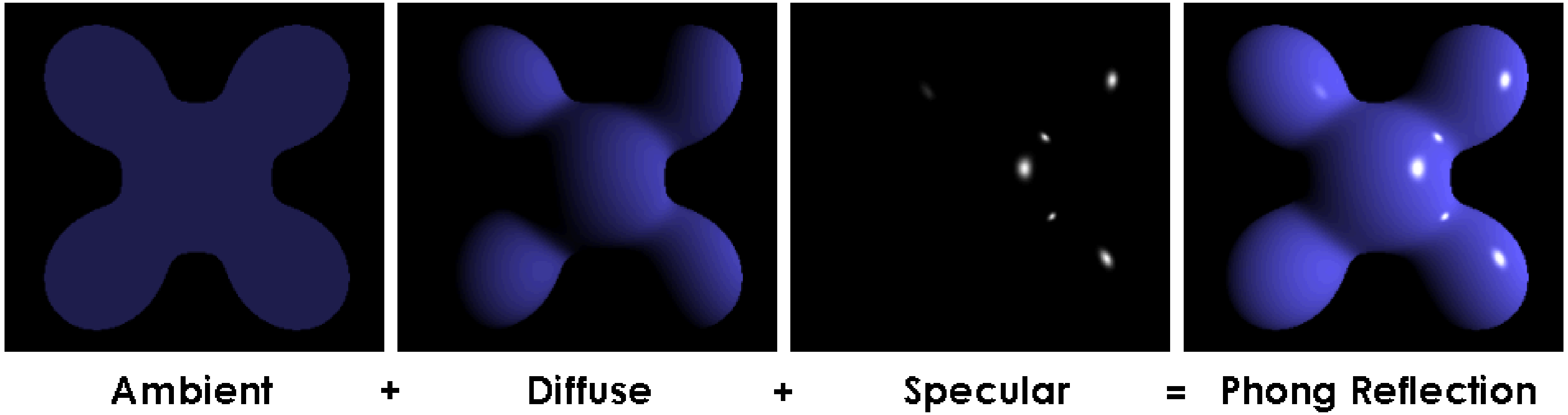
- This means the shading calculation is done in the fragment shader
- We will use the Phong reflectance model for the MP

Gouraud Shading vs. Phong Shading





Phong Reflectance Model



$$I_p = k_a i_a + \sum_{m \in \text{lights}} (k_d (\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s (\hat{R}_m \cdot \hat{V})^\alpha i_{m,s})$$

Complete the Phong Shader

In HelloPhong.html

1. Complete the vertex shader code
    1. Implement shader-phong-phong-vs
2. Complete the fragment shader code
    1. Implement shader-phong-phong-fs

More Specifically

You can use the existing shader code as a reference.

It implements Gouraud shading

1. The colors are computed per-vertex in the vertex shader
2. Sent to the Fragment Shader as varyings
3. And so interpolated across the fragments

You need to implement Phong shading

1. Compute gl_Position in the vertex shader like usual
2. Send the vertex normal and position to the fragment shader
    1. Which means they will be interpolated across fragments
3. Compute the color for the fragment
    1. You'll need to send the uniforms you need to fragment shader for this

One More Hint…

```
<script id="shader-phong-phong-vs" type="x-shader/x-vertex">
   attribute vec3 aVertexNormal;
   attribute vec3 aVertexPosition;

    uniform mat4 uMVMatrix;
    uniform mat4 uPMatrix;
    uniform mat3 uNMatrix;
    varying vec3 vNormal;
    varying vec3 vPosition;

    void main(void) {

    // Get the vertex position in eye coordinates
    vec4 vertexPositionEye4 = uMVMatrix * vec4(aVertexPosition, 1.0);
    vPosition = vertexPositionEye4.xyz / vertexPositionEye4.w;
    //Calculate the normal
    vNormal = normalize(uNMatrix * aVertexNormal);

    gl_Position = uPMatrix*uMVMatrix*vec4(aVertexPosition, 1.0);

 }
</script>
```

Here's the vertex shader you need to implement…code up a fragment shader to match….

Some Final Questions….

1. What coordinate space is the shading calculation performed in?
    1. Model?World?View?Clip?
2. You’ll notice that the light position is not transformed by the view transformation
    1. In an animated scene, this could be a bug. Why?
3. In the js code, the shaders are recompiled and linked every time we switch shader programs.
    1. How could we implement this more efficient?