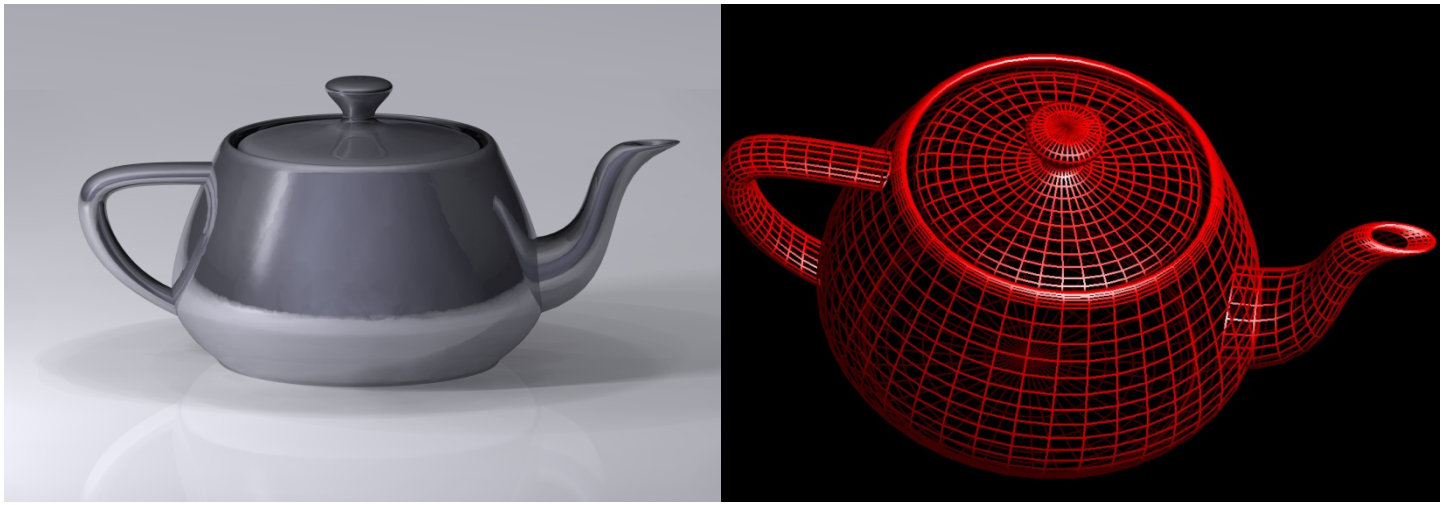


Foundations of Computer Graphics

Saurabh Ray

Reading

Reading for lecture 10: Sections 6.5 - 6.10

Reading for lectures 11, 12: Sections 7.1 - 7.9

Light Sources

In the Phong Model, we add the results from each light source.

Each light source has diffuse, specular, and ambient components.

This is for flexibility. No physical justification.

$I_d = (I_{dr}, I_{dg}, I_{db})$: diffuse component

$I_a = (I_{ar}, I_{ag}, I_{ab})$: ambient component

$I_s = (I_{sr}, I_{sg}, I_{sb})$: specular component

Material Properties

RECAP

Materials also have diffuse, ambient and specular reflectance and a shininess coefficient.

$k_d = (k_{dr}, k_{dg}, k_{db})$: diffuse reflectance

$k_a = (k_{ar}, k_{ag}, k_{ab})$: ambient reflectance

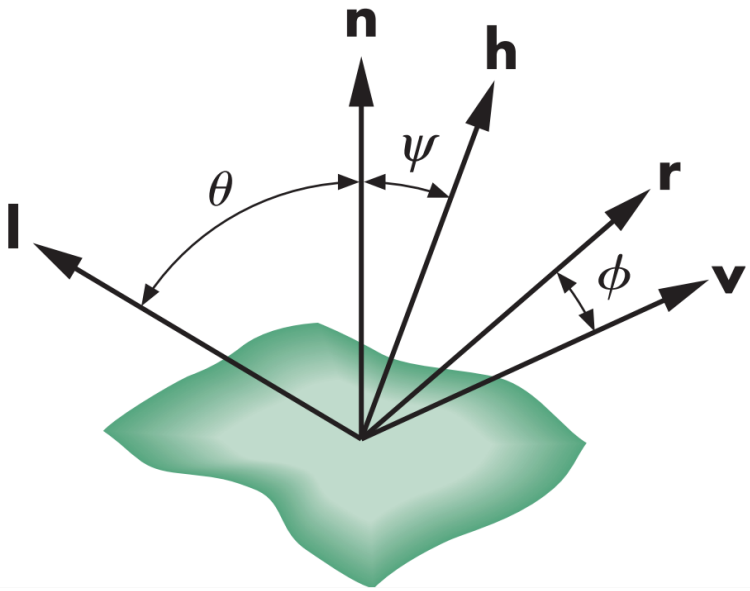
$k_s = (k_{sr}, k_{sg}, k_{sb})$: specular reflectance

α: shininess coefficient

Phong Lighting Model

RECAP

$$\mathbf{r} = -\mathbf{l} + 2(\mathbf{l} \cdot \mathbf{n})\mathbf{n}$$



$$I = k_d I_d \max(\mathbf{l} \cdot \mathbf{n}, 0) + k_a I_a + k_s I_s (\max(\mathbf{v} \cdot \mathbf{r}, 0))^{\alpha}$$ PHONG

$$I = k_d I_d \max(\mathbf{l} \cdot \mathbf{n}, 0) + k_a I_a + k_s I_s (\max(\mathbf{n} \cdot \mathbf{h}, 0))^{\beta}$$ BLINN - PHONG

Diffuse component

Ambient component

Specular component

Attenuation: divide by $a + bt + ct^2$

t: distance from light source

a, b, c: constants

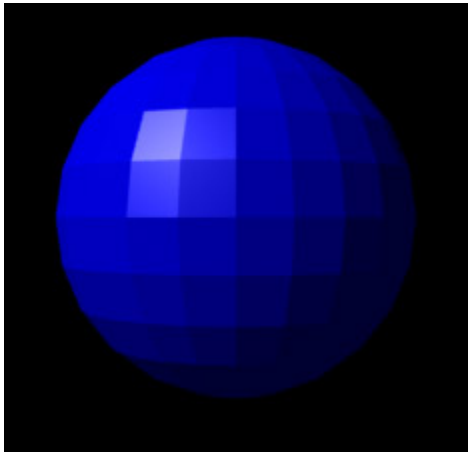
Shading

How do we compute the color points in the interior of a face?

Flat Shading:

If we assume that the viewer and the light source are far away compared to the size of a face, then **l** and **v** are constant over a face.

n is also constant.



Shading calculations need to be done only once for the triangle.

Problem: Mach Bands

Exaggerated contrast.

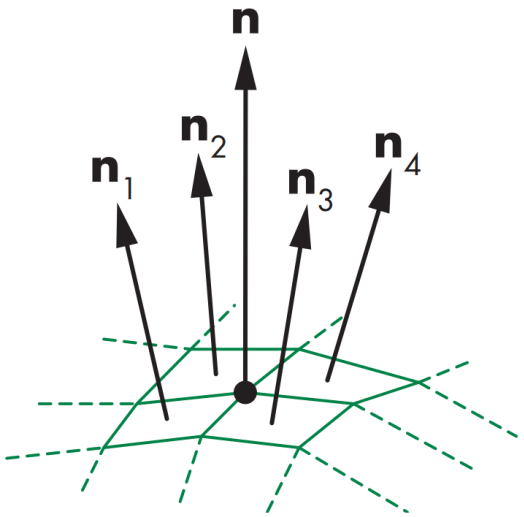
Gouraud Shading:

The normal **n** at a vertex makes sense if we have an underlying surface that our mesh is approximating.

However, if we just have mesh, then there are multiple faces meeting at a vextex. Each of the faces have different normals.

We define the normal at the vertex as the average of the normals of incident faces:

$$\mathbf{n} = \frac{\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4}{|\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4|}$$



Thus different vertices of the same face have different colors.

Colors of interior points are obtained by interpolation.

Phong Shading:

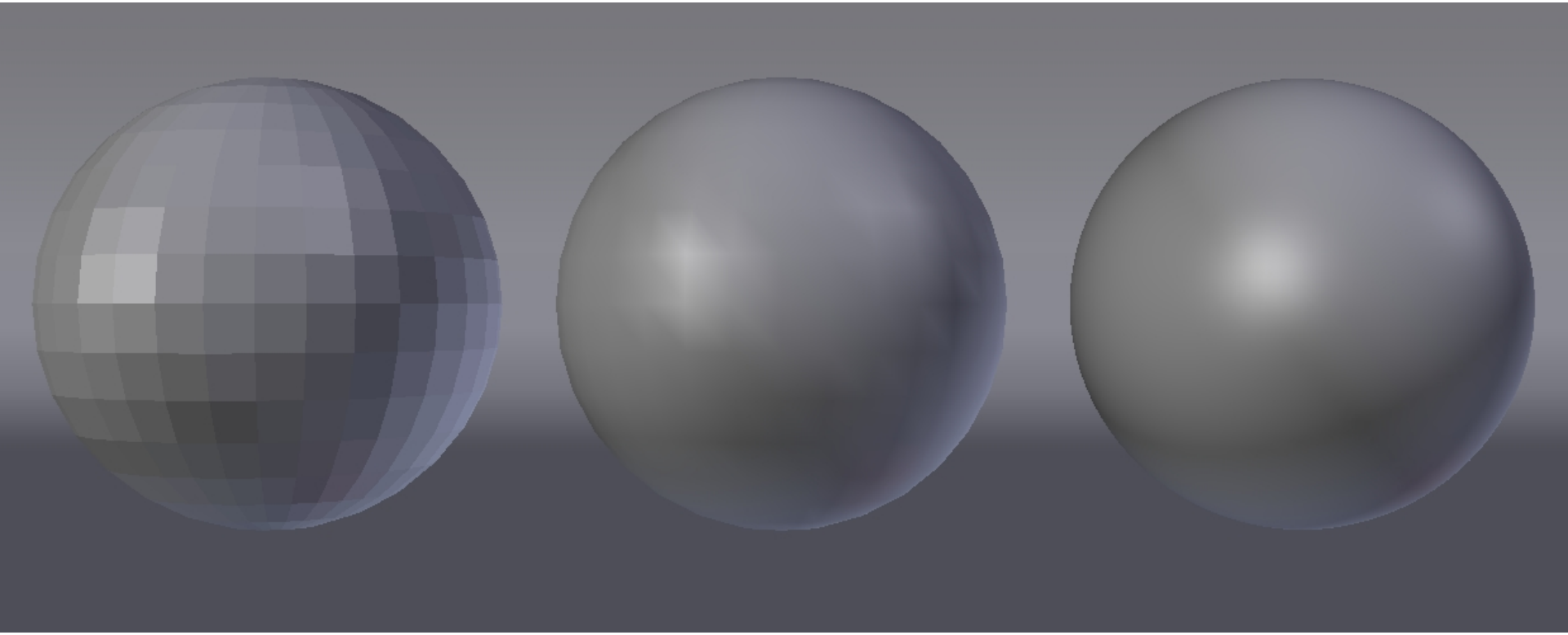
Interpolate the normals at vertices to get the normal at an interior point.

Compute the color using the lighting equations.

Note that we are doing **per fragment lighting**.

In Flat and Gouraud shading we do **per vertex lighting**.

Shading



Flat

Same normal for all vertices on a face.

Vertex colors interpolated.

Gouraud

Fixed normal vector for each vertex.

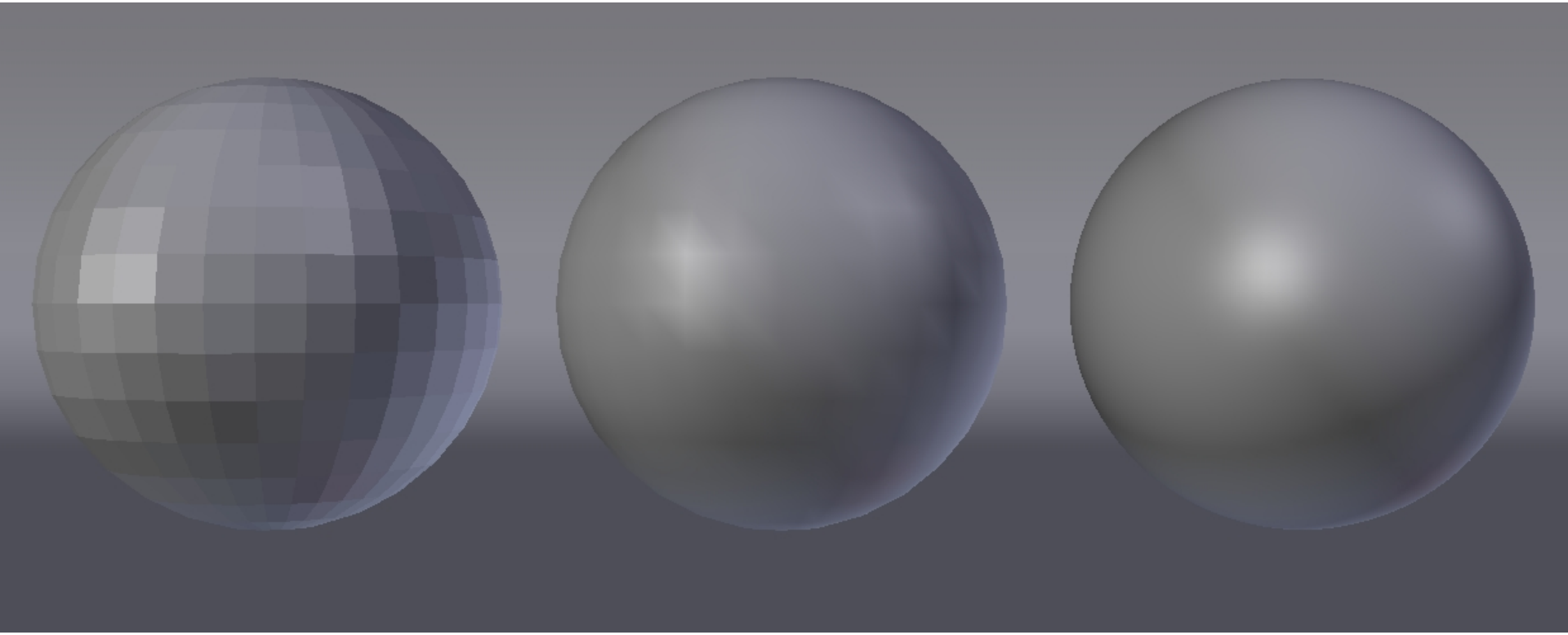
Vertex colors interpolated.

Phong

Each vertex has a fixed normal vector.

Vertex normals interpolated.

Shading



Flat **Gouraud** **Phong**

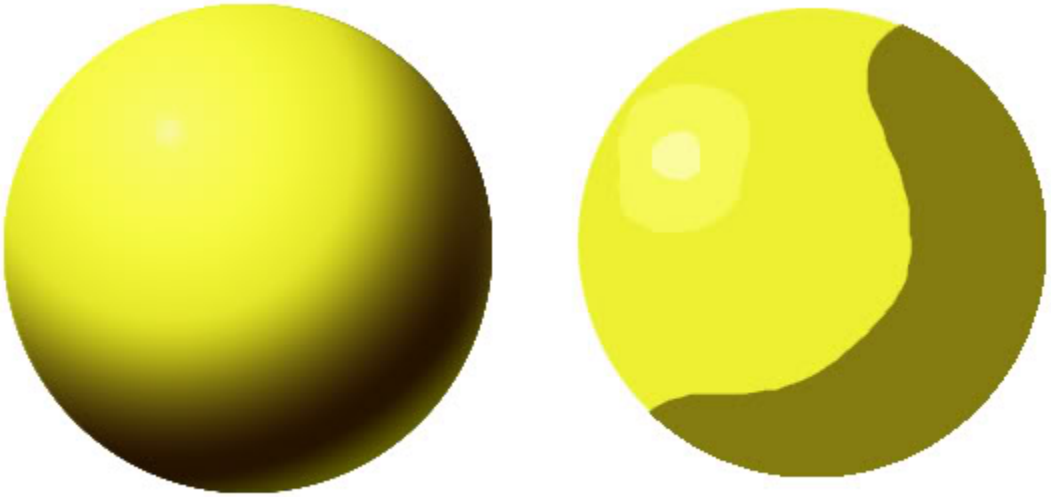
Shading and Lighting are different terms.

Shading: how interpolation is done.

Lighting: the equations used to compute color.

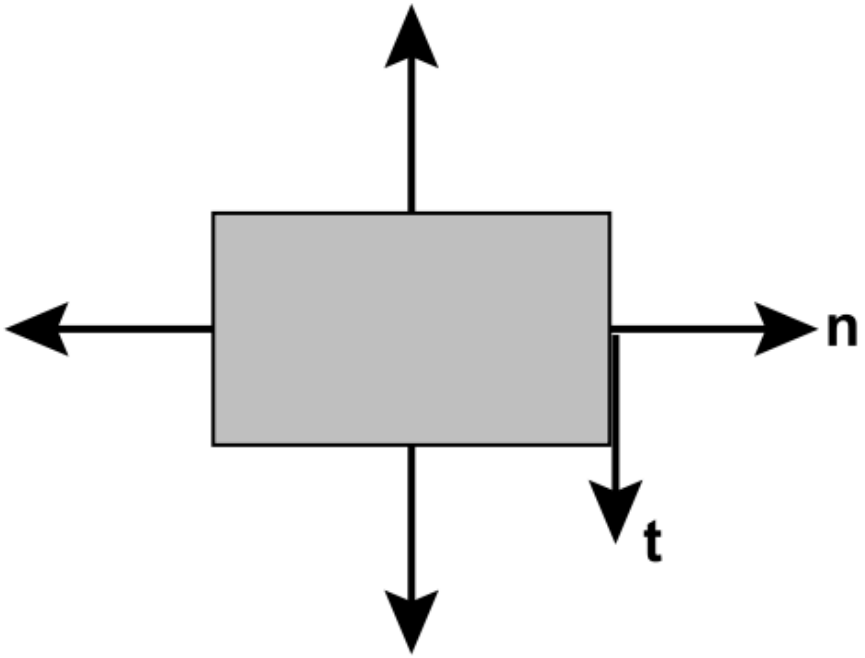
Non Photorealistic Shading

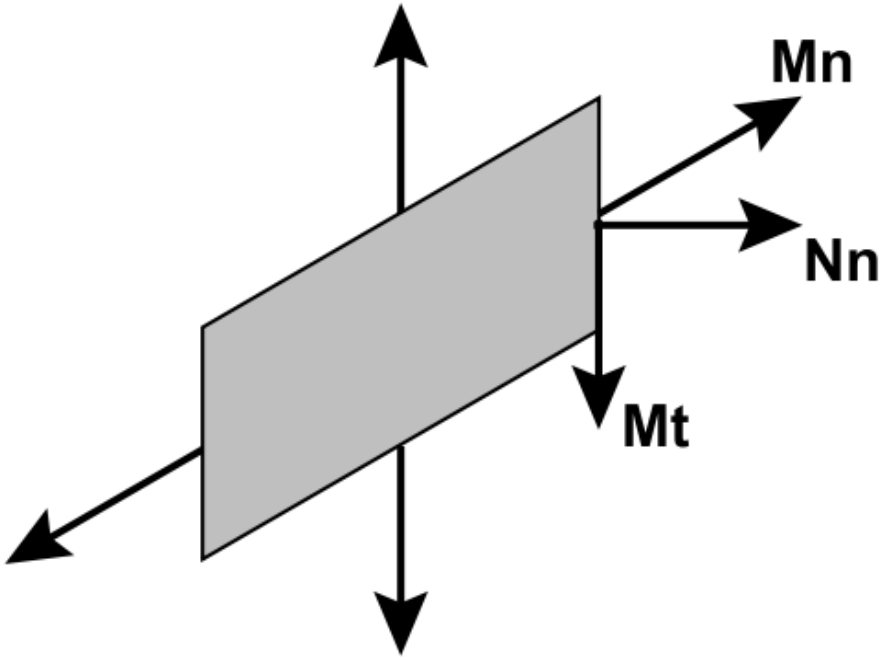
Cartoon Shading (called toon shading or cel shading)



Use only a few discrete shades.

Transformation of Normal Vectors



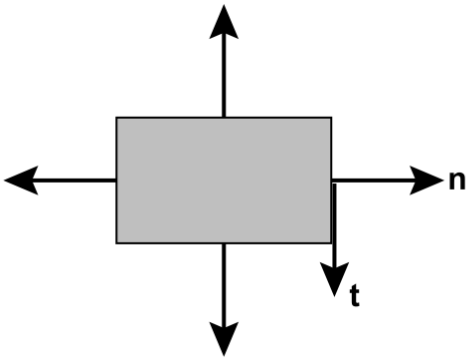


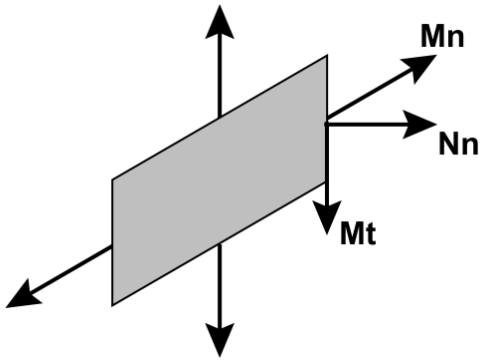
$$\mathbf{n}^T\mathbf{t} = 0$$

$$\mathbf{Mn}^T\mathbf{Mt} \neq 0$$

The tangent vector is transformed correctly but the normal vector is not.

Transformation of Normal Vectors





$$\mathbf{n}^{\mathrm{T}}\mathbf{t} = \mathbf{n}^{\mathrm{T}}\mathbf{I}\mathbf{t} = \mathbf{n}^{\mathrm{T}}\mathbf{M}^{-1}\mathbf{M}\mathbf{t} = \mathbf{0}$$

$$\underbrace{\left(\mathbf{n}^{\mathrm{T}}\mathbf{M}^{-1}\right)}\left(\mathbf{M}\mathbf{t}\right) = \left(\mathbf{n}^{\mathrm{T}}\mathbf{M}^{-1}\right)\mathbf{t}_M = \mathbf{0}$$

This is the transpose of the new normal.

New normal $\quad \mathbf{n}_N = \left(\mathbf{n}^{\mathrm{T}}\mathbf{M}^{-1}\right)^{\mathrm{T}} = \left(\mathbf{M}^{-1}\right)^{\mathrm{T}}\mathbf{n}$

Closures in Javascript

```
var counter = 0;

function increment(){
   counter++;
   console.log(counter);
}
```

Familiar concept. From an inner scope, we can access variables and functions defined in an outer scope.

Consider the following:

```
function init(){
    var counter = 0;

    function increment(){
       counter++;
       console.log(counter);
    }

    return increment;
}
```

```
var inc = init();

inc();      prints 1

inc();      prints 2

console.dir(inc);
```

Try this in the javascript console.

Closures in Javascript

How does this work?

The variable `counter` goes out of scope when the function `init` returns.

How is the function `inc` still able to access it?

Answer: via a **Closure.**

A closure is an object that consists of:

- a function
- and the environment in which it is defined

any local variables in scope

So, functions remember the environment in which they are defined.

Closures in Javascript

Closures can be used to implement data privacy.

```
function createObject(){

    var x=0;

    var obj = {
            get: function ()  { return x; },
            set: function (v) { x = v;    }
        };

    return obj;
}

var obj = createObject();

obj.get();     returns 0

obj.set(5);

obj.get();     returns 5
```

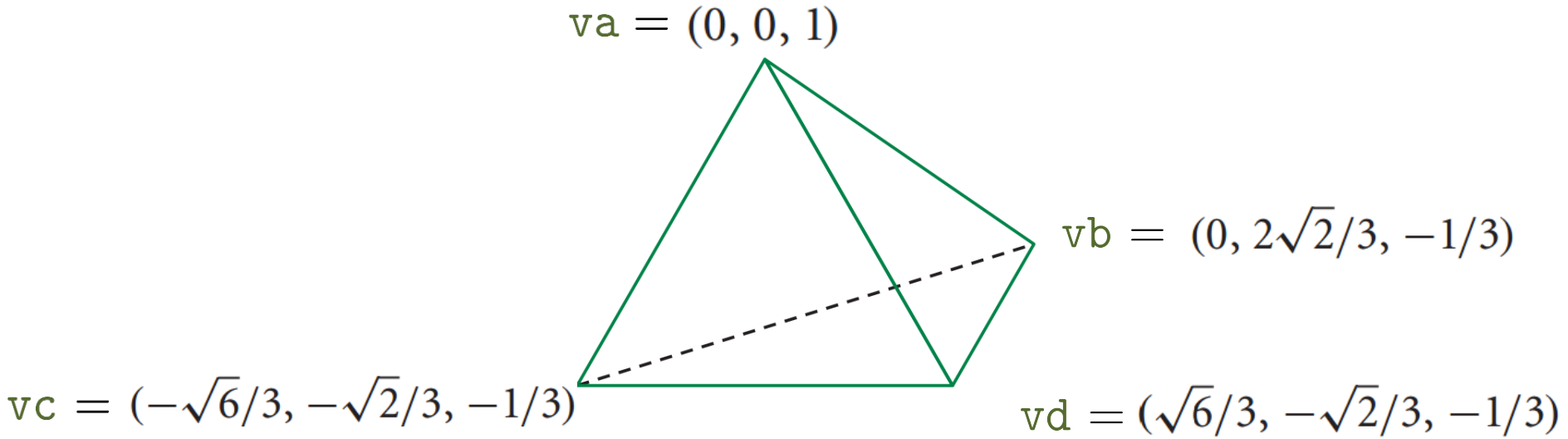
We don't have access to `x` *directly. We can only access it through the functions* `get` *and* `set`.

Coding

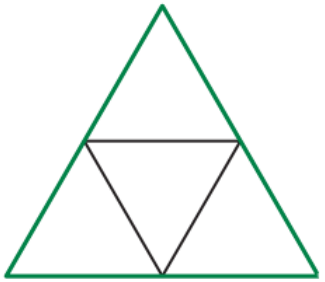
Want: Display a sphere with lighting.

How do we model a sphere?

Idea: Start with a tetrahederon and refine it.



Subdivide each face into four triangles:



Coding

```
function Sphere(n){
    // n is the number of times to
    // subdivide the faces recursively.

    var S = {   positions: [],
                normals: [],
            };



    var s2 = Math.sqrt(2);
    var s6 = Math.sqrt(6);

    var va = vec3(0,0,1);
    var vb = vec3(0, 2*s2/3, -1/3);
    var vc = vec3(-s6/3, -s2/3, -1/3);
    var vd = vec3(s6/3, -s2/3, -1/3);

    tetrahedron(va, vb, vc, vd, n);



    function tetrahedron(a,b,c,d,n){
        divideTriangle(d,c,b,n);
        divideTriangle(a,b,c,n);
        divideTriangle(a,d,b,n);
        divideTriangle(a,c,d,n);
    }
```

```
    function divideTriangle(a,b,c,n){
        if(n>0){
            var ab = normalize(mix(a,b,0.5));
            var ac = normalize(mix(a,c,0.5));
            var bc = normalize(mix(b,c,0.5));

            n--;

            divideTriangle(a,ab,ac,n);
            divideTriangle(ab,b,bc,n);
            divideTriangle(bc,c,ac,n);
            divideTriangle(ab,bc,ac,n);
        }
        else{
            triangle(a,b,c);
        }
    }


    function triangle(a,b,c){
        var norm = normalize(cross(subtract(b,a),
                                   subtract(c,a)));
        S.positions.push(a,b,c);
        S.normals.push(norm, norm, norm);
        //S.normals.push(a,b,c);
    }

    return S;
}
```

Coding

It will be useful to write a function `objInit` which attaches several functions to the sphere object we created.

```
function objInit(Obj){

    /* Initialization Code.
    (declare and initialize variables,
    create buffers and transfer data to buffers,
    get locations of attributes and uniforms, etc) */
    // ...

    /*----- Helper functions -----*/
    // Define any helper functions you need.
    // ...

    /*----- Attach functions to Obj -----*/

    Obj.setModelMatrix = function(m){
        // use m as the modeling matrix
        // ...
    }

    Obj.draw = function(){
        // draw the object
        // ...
    }
}
```

We will create a file called `Object.js` containing this function.

Coding

To make it easy to use the virtual trackball in future projects we create a file `Trackball.js` containing a function `Trackball` that returns a trackball object.

```
function Trackball(canvas){

    var tbMatrix = mat4(); /* initialize virtual trackball matrix */

    var trackball = { /* object to be returned */
        getMatrix: function() { return tbMatrix; }
    };

    /* other variables and helper functions */

    // ...

    function mousedown(event){ /* ... */ }
    function mouseup(event)  { /* ... */ }
    function mousemove(event){ /* ... */ }
    function wheel(event)    { /* ... */ }

    //set event handlers
    canvas.addEventListener("mousedown", mousedown);
    canvas.addEventListener("mouseup", mouseup);
    canvas.addEventListener("mousemove", mousemove);
    canvas.addEventListener("wheel", wheel, {passive: true});

    return trackball;
}
```

Coding

To make it easy to set up the camera, we create a file `Camera.js` containing the function `Camera()` which returns a camera object.

```
function Camera(){

    var Mcam = mat4(); // camera matrix
    var P = mat4();    // projection matrix

    var camFrame = { // camera frame
            e: vec3(0,0,0), // camera location
            u: vec3(1,0,0), // unit vector to "right"
            v: vec3(0,1,0), // unit vector in "up" direction
            w: vec3(0,0,1)  // unit vector opposite "gaze" direction
    };

    addEventHandlers(); // add event handlers for moving the camera


    var Cam = { /* object to be returned */

            lookAt: function (eye, at, up) {
                // set camFrame and Mcam
                // ...
            },

            setPerspective: function(fovy, aspect, near, far){
                // set the projection matrix P
                // ...
            },
```

Coding

```
        setOrthographic: function (r,l,t,b,n,f){
            // set the projection matrix P
            // ...
        },

        getCameraTransformationMatrix: function(){
            return Mcam;
        },

        getProjectionMatrix: function(){
            return P;
        },

        getMatrix: function(){
            // combines camera transformation and projection
            return mult(P,Mcam);
        },

        getFrame: function (){
            // returns camera frame (e, u, v, w)
            return camFrame;
        }
    };


    /* Helper functions */

    function addEventHandlers(){
        // ...
    }

    // ...

    return Cam;
}
```

Coding

The Javascript code for displaying a sphere with lighting looks like this:

```
"use strict";

// global variables
var gl, canvas, program;

var camera;     // camera object
var trackball;  // virtual trackball

var sphere;

var Locations;  // object containing locations of shader variables

window.onload = function init() {
    // Set up WebGL
    canvas = document.getElementById("gl-canvas");
    gl = WebGLUtils.setupWebGL( canvas );
    if(!gl){alert("WebGL setup failed!");}

    // set clear color
    gl.clearColor(0.0, 0.0, 0.0, 1.0);

    //Enable depth test
    gl.enable(gl.DEPTH_TEST);
    gl.depthFunc(gl.LEQUAL);
    gl.clearDepth(1.0);

    // Load shaders and initialize attribute buffers
    program = initShaders( gl, "vertex-shader", "fragment-shader" );
    gl.useProgram( program );
```

Coding

```
    var Attributes = [];
    var Uniforms = ["VP", "TB", "TBN", "cameraPosition", "Ia", "Id", "Is", "lightPosition"];
    Locations = getLocations(Attributes, Uniforms);

    // set up virtual trackball
    trackball = Trackball(canvas);

    // set up Camera
    camera = Camera();          // defined in Camera.js
    var eye = vec3(0,0, 3);
    var at = vec3(0, 0 ,0);
    var up = vec3(0,1,0);
    camera.lookAt(eye,at,up);
    camera.setPerspective(90,1,0.1,10);

    sphere = Sphere(6);

    objInit(sphere);// defined in Object.js
    sphere.setModelMatrix(scalem(2,1,2));

    // set light source
    var Light = {
        position: vec3(-5,10,20),
        Ia: vec3(0.2, 0.2, 0.2),
        Id: vec3(1,1,1),
        Is: vec3(0.8,0.8,0.8)
    };

    gl.uniform3fv( Locations.lightPosition, flatten(Light.position) );
    gl.uniform3fv( Locations.Ia, flatten(Light.Ia) );
    gl.uniform3fv( Locations.Id, flatten(Light.Id) );
    gl.uniform3fv( Locations.Is, flatten(Light.Is) );

    var shading = 3.0; // flat: 1.0, Gouraud: 2.0, Phong: 3.0
    gl.uniform1f(gl.getUniformLocation(program, "shading"), shading);

    requestAnimationFrame(render);
};
```

Coding

```
function render(now){

    requestAnimationFrame(render);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    var TB = trackballWorldMatrix(trackball, camera);
    gl.uniformMatrix4fv(Locations.TB, gl.FALSE, flatten(TB));

    var TBN = normalTransformationMatrix(TB);
    gl.uniformMatrix3fv(Locations.TBN, gl.FALSE, flatten(TBN));

    var VP = camera.getMatrix();
    gl.uniformMatrix4fv(Locations.VP, gl.FALSE, flatten(VP));

    var cameraPosition = camera.getFrame().e;
    gl.uniform3fv(Locations.cameraPosition, flatten(cameraPosition));

    sphere.draw();

}
```

Vertex Shader:

```
precision highp float;

attribute vec4 vPosition;
attribute vec3 vNormal;

uniform mat4 M, TB, VP;
uniform mat3 N, TBN;
uniform vec3 Ka, Kd, Ks, Ia, Id, Is, lightPosition, cameraPosition;
uniform float shininess;
uniform float shading;

varying vec4 fColor;
varying vec3 fNormal, fPosition;

vec4 computeColor(vec3 position, vec3 normal) {
      // we are doing lighting in world coordinate frame

      vec3 lightDir = normalize(lightPosition - position);
      vec3 viewDir = normalize(cameraPosition - position);

      vec3 ambient = Ia*Ka ;
      vec3 diffuse = Id*Kd* max(0.0, dot(normal, lightDir));

      vec3 halfVector = normalize(lightDir + viewDir);
      vec3 specular = Is*Ks* pow( max(dot(halfVector, normal), 0.0), shininess);

      vec3 color = ambient + diffuse + specular;

      return vec4(color, 1.0);
}
```

Coding

```
void main(){
    vec4 wPos = TB*M*vPosition; // world position

    gl_Position = VP*wPos;
    gl_Position.z *= -1.0; // convert to Left Handed Coordinate System

    fPosition = wPos.xyz;

    vec3 normal;

    mat3 NT = TBN*N;

    if(shading == 1.0){ // Flat
     normal = normalize(NT*vNormal);
    }
    else { // Gouraud or Phong
     normal = normalize(NT*vPosition.xyz);
    }
    /* Note: we need to scale the normal vectors to
       unit length since the normal transformation
       matrix does not guarantee that. */

    if(shading==3.0){ // Phong
        fNormal = normal;
    }
    else{ // Flat or Gouraud
        fColor = computeColor(fPosition, normal);
    }
}
```

Fragment Shader:

```
precision highp float;

uniform vec3 Ka, Kd, Ks, Ia, Id, Is, lightPosition, cameraPosition;
uniform float shininess;
uniform float shading;

varying vec4 fColor;
varying vec3 fPosition, fNormal;

vec4 computeColor(vec3 position, vec3 normal) {
    // we are doing lighting in world coordinate frame

    vec3 lightDir = normalize(lightPosition - position);
    vec3 viewDir = normalize(cameraPosition - position);

    vec3 ambient = Ia*Ka ;
    vec3 diffuse = Id*Kd* max(0.0, dot(normal, lightDir));

    vec3 halfVector = normalize(lightDir + viewDir);
    vec3 specular = Is*Ks* pow( max(dot(halfVector, normal), 0.0), shininess);

    vec3 color = ambient + diffuse + specular;

    return vec4(color, 1.0);
}
```

Coding

```
void main(){
    if(shading == 3.0){ // Phong
        gl_FragColor = computeColor(fPosition, normalize(fNormal));
    }
    else{
        gl_FragColor =  fColor;
    }
}
```