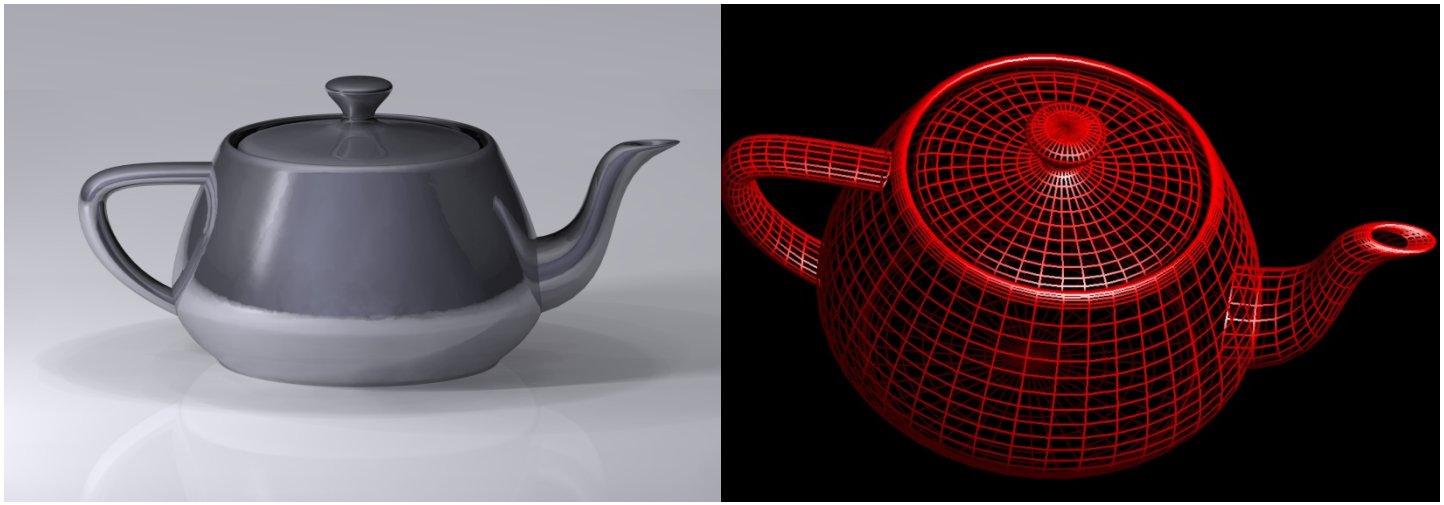
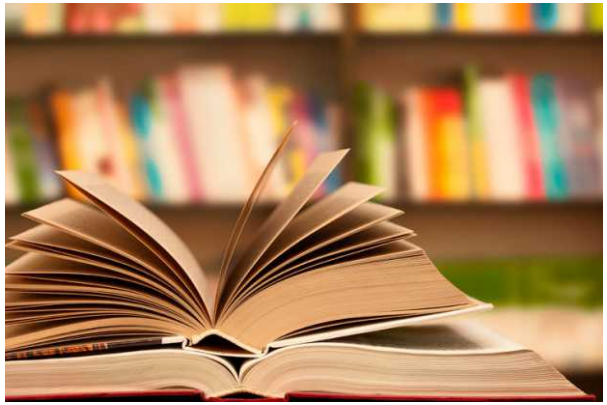


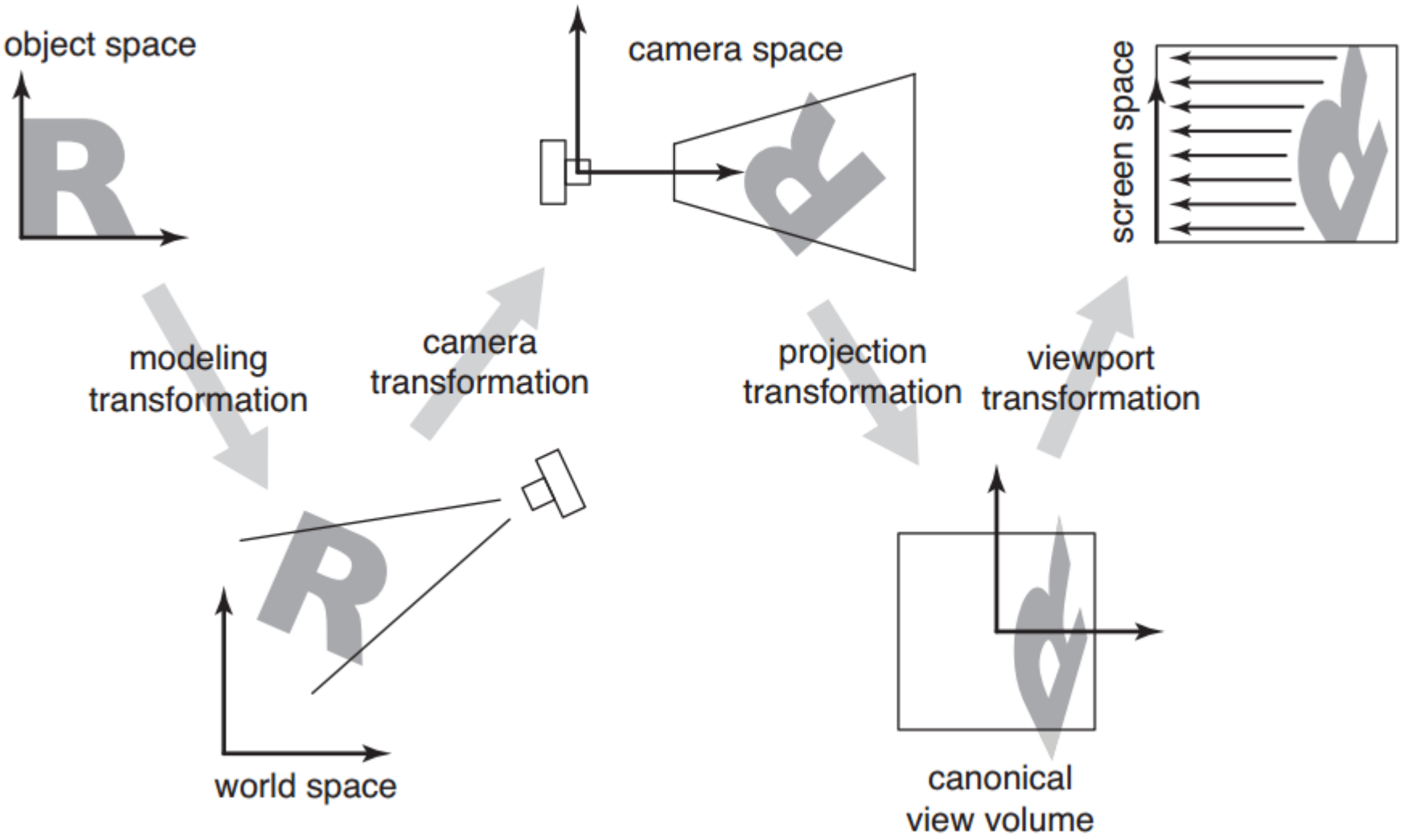
Foundations of Computer Graphics

Saurabh Ray

Reading

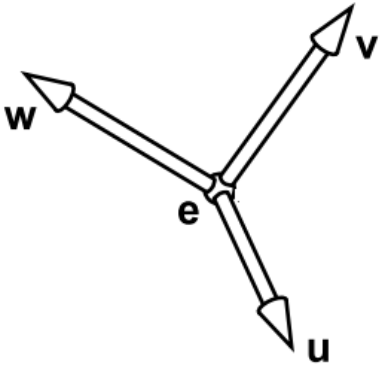


Reading for Lectures 9,10: Sections 6.1 - 6.8.

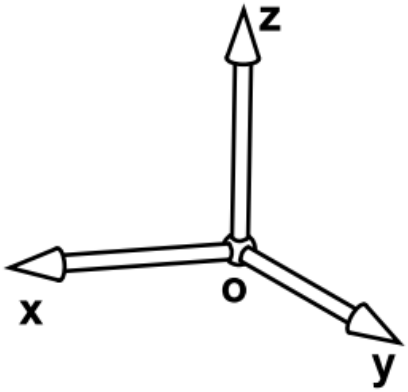

object space
camera space
screen space
modeling
transformation
camera
transformation
projection
transformation
viewport
transformation
world space
canonical
view volume

Camera or Eye Transformation

Camera Coordinate Frame



World Coordinate Frame



Coordinates in Camera Frame

columns are homogeneous coordinates w.r.t. the world frame

Coordinates in World Frame

$$\boldsymbol{\beta} = \underbrace{\begin{bmatrix} \vec{u} & \vec{v} & \vec{w} & e \end{bmatrix}^{-1}}_{M_{cam}} \boldsymbol{\alpha}$$

Matrix that implements Camera transformation

Computing Camera Transformation Matrix

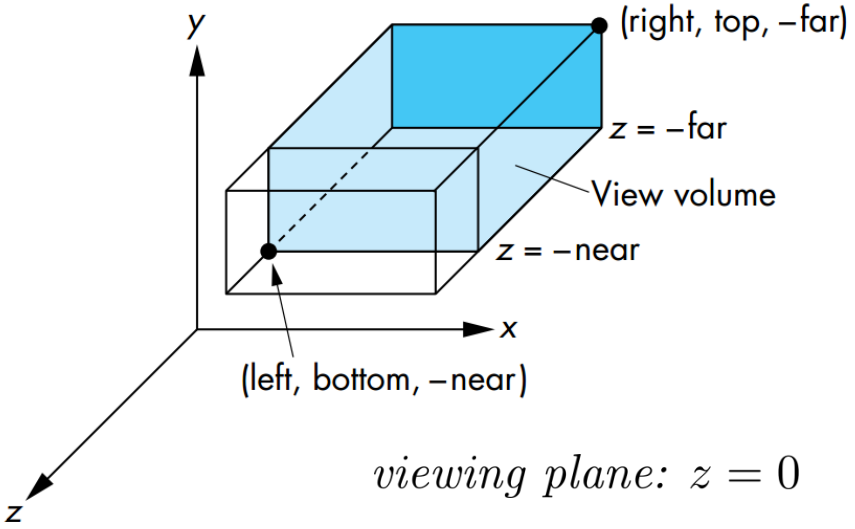
$$M_{cam} = \begin{bmatrix} \vec{u} & \vec{v} & \vec{w} & e \end{bmatrix}^{-1}$$

If we treat $\vec{u}$, $\vec{v}$ and $\vec{w}$ and 3D vectors, and let $\vec{e}$ be the 3D position vector of the eye, then

$$M_{cam} = \begin{pmatrix} \vec{u} & \vec{v} & \vec{w} & \vec{e} \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} \vec{u} & -\vec{u}\cdot\vec{e} \\ \vec{v} & -\vec{v}\cdot\vec{e} \\ \vec{w} & -\vec{w}\cdot\vec{e} \\ 0\ 0\ 0 & 1 \end{pmatrix}$$

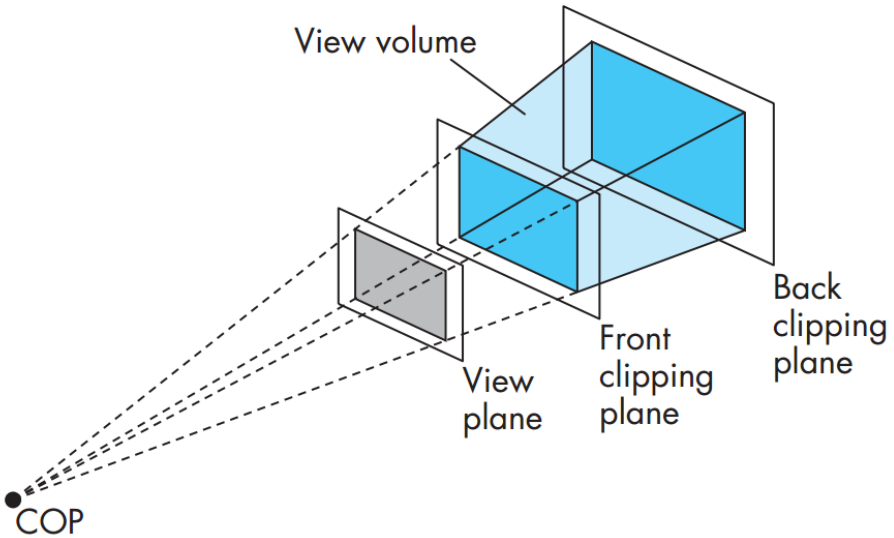
Viewing

Orthographic:



viewing plane: $z = 0$

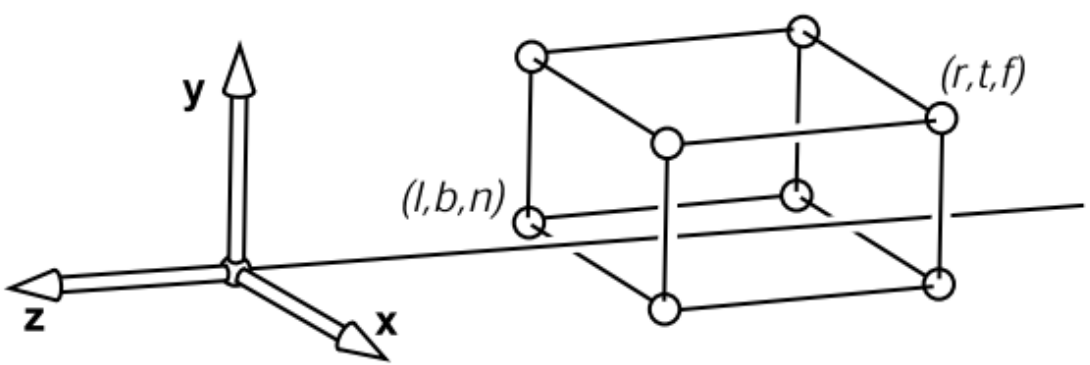
Perspective:

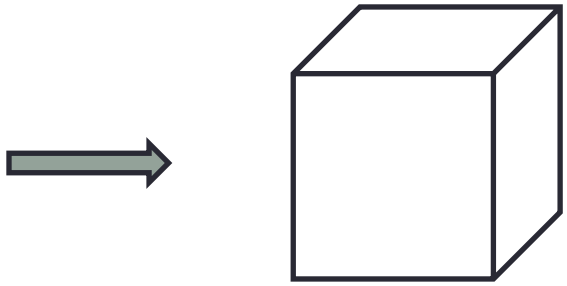


In both cases there is a notion of a view volume

Projection Normalization

Orthographic projection:

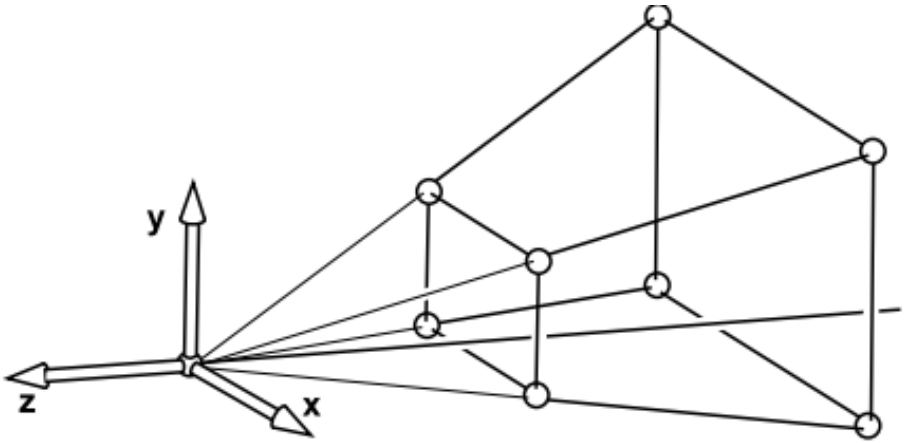




$$[-1,1] \times [-1,1] \times [-1,1]$$

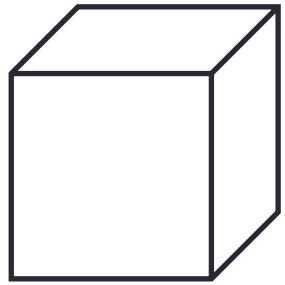
$$M_{orth} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective Projection Normalization

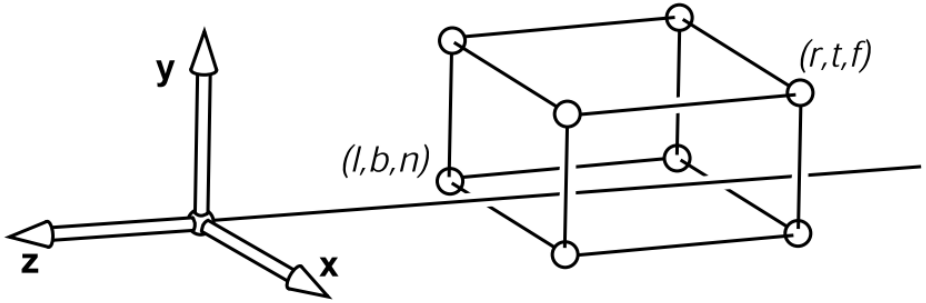


$$M_{per} = M_{orth} P$$

P



$[-1,1] \times [-1,1] \times [-1,1]$



M_{orth}

Perspective Matrix

$$M_{per} = M_{orth} P$$

$$= \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1+\frac{f}{n} & -f \\ 0 & 0 & 1/n & 0 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{2}{r-l} & 0 & -\frac{r+l}{n(r-l)} & 0 \\ 0 & \frac{2}{t-b} & -\frac{t+b}{n(t-b)} & 0 \\ 0 & 0 & \frac{n+f}{n(n-f)} & -\frac{2f}{n-f} \\ 0 & 0 & 1/n & 0 \end{bmatrix}$$

Perspective Matrix

Perspective Matrix:

$$M_{per} = \begin{bmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & -\frac{2fn}{n-f} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = M_{per} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Observe that: $w' = z$

For points in front of camera: $z < 0$.

So, for those points: $w' < 0$

Problem: WebGL clips out points with negative w-coordinate!

Perspective Normalization

Solution: Multiply every entry in the matrix by -1.

Due to perspective divide this does not change transformation but makes the w-coordinate of points in front of the camera positive.

Perspective Matrix
to be used in WebGL coding:

$$M_{per} = \begin{bmatrix} -\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & -\frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{n+f}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Note: Sometimes n and f are treated as distances to the near and far planes respectively. In that case the planes are $z = -n$ and $z = -f$.

Lighting

Objects appear more realistic to us if we can see color variations that happen because of lighting.

vs

Each point has a different color of shade due to interactions of light and material.

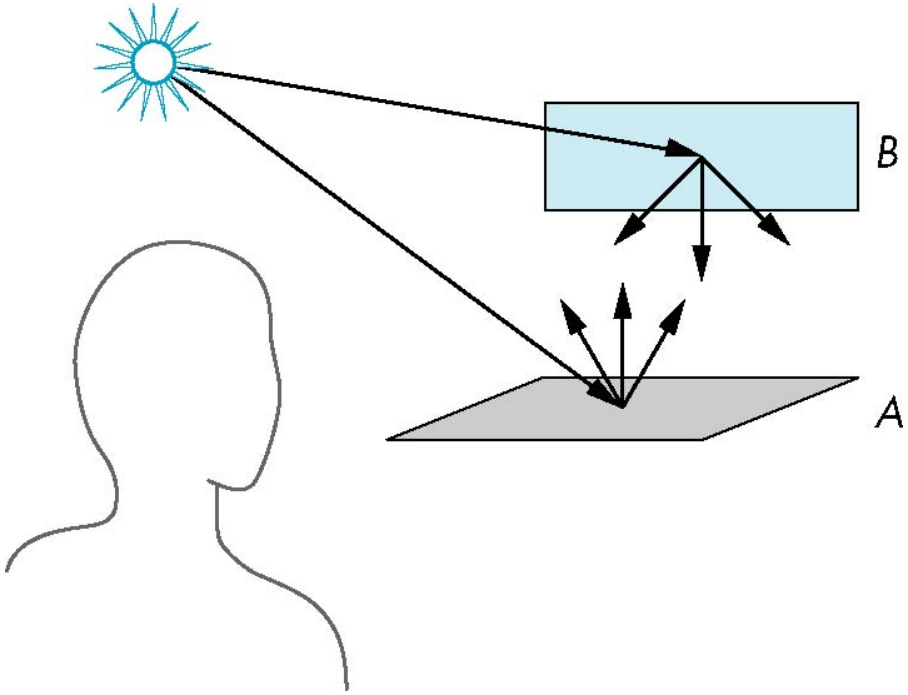
Need to consider:

- Light sources
- Material properties
- Location of viewer
- Surface orientation

Scattering

Light bounces off different surfaces before reaching our eye.

Each time it hits a surface, some of the light is absorbed and some is scattered.



Rendering Equation

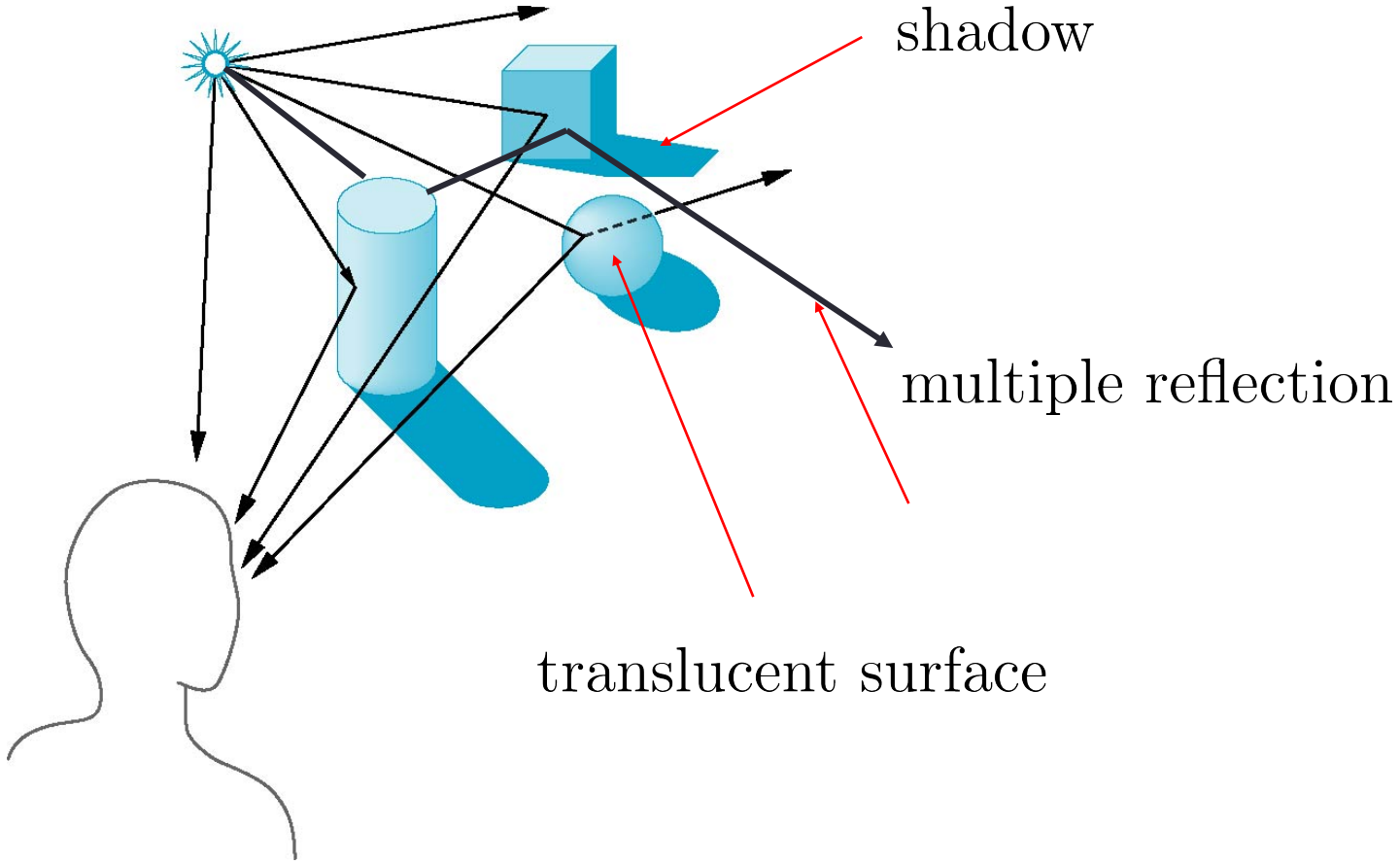
The infinite scattering and absorption of light can be described by the **rendering equation**.

- Cannot be solved in general
- Ray tracing is a special case for perfectly reflecting surfaces

Rendering equation is global and includes:

- Shadows
- multiple scattering from object to object

Global Effects



Correct shading requires a global calculation involving all objects and light sources. This is incompatible with the pipeline model.

In real time computer graphics, we use heuristics to approximate reality.

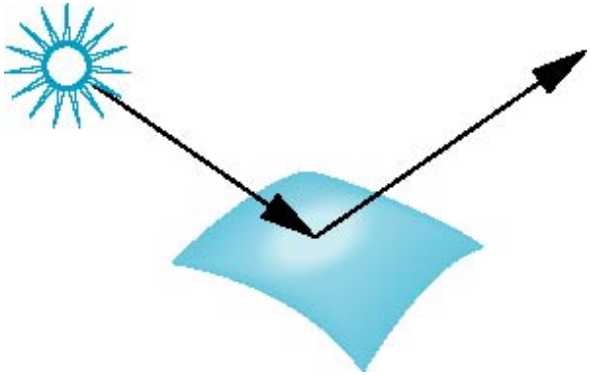
Light Material Interaction

Light that strikes an object is partially absorbed and partially scattered (reflected).

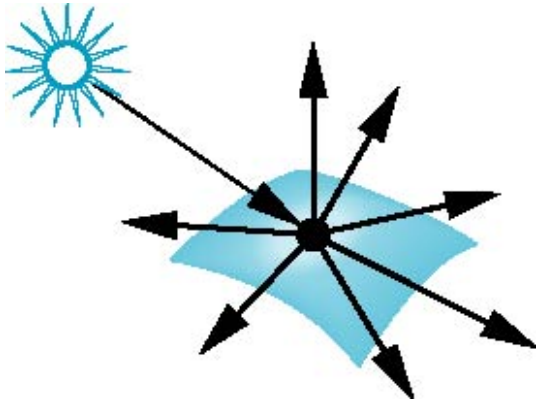
The amount reflected determines the color and brightness of the object.

> A surface appears red under white light because the red component of the light is reflected and the rest is absorbed

The reflected light is scattered in a manner that depends on the smoothness and orientation of the surface

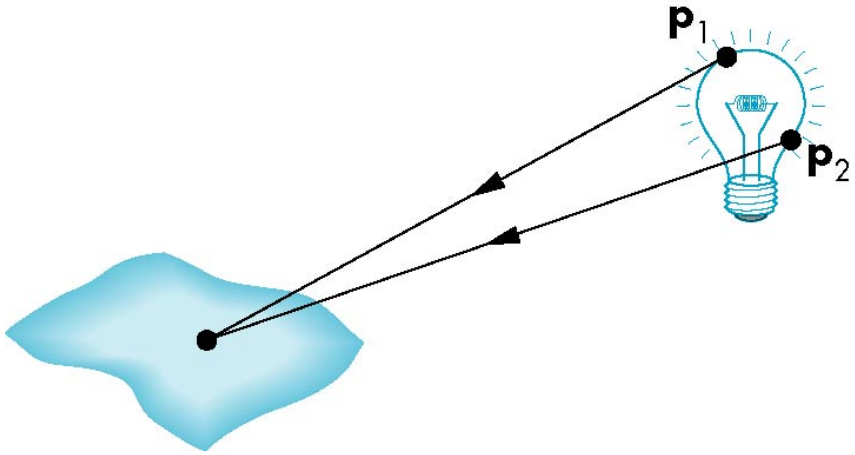


Smooth Surface



Rough Surface

Light Sources



General light sources are difficult to work with because we must integrate light coming from all points on the source

Simple Light Sources:

Directional Light modelled with direction and color

Point source: modelled with position and color

Spot Light: restricted light from point source

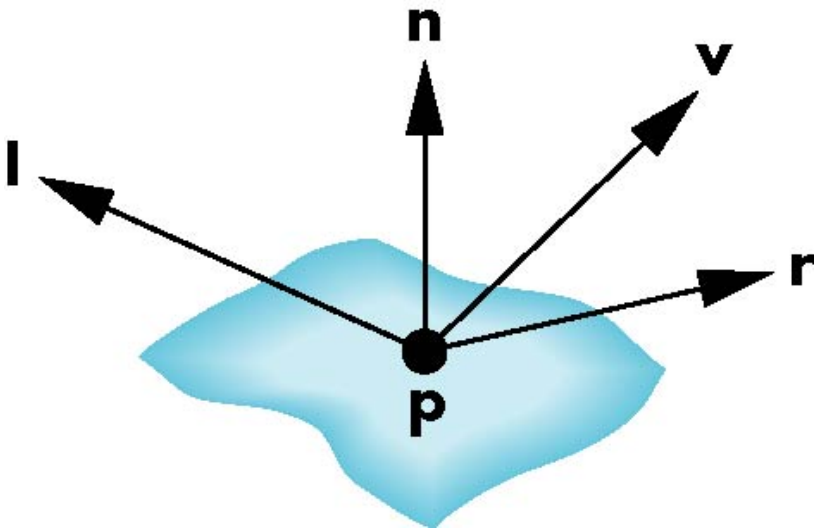
Ambient Light: same amount of light everywhere

Phong Model

A simple model that can be computed rapidly.

Has three components:

- Diffuse
- Specular
- Ambient



Uses four *unit* vectors:

Typically all vectors are normalized.

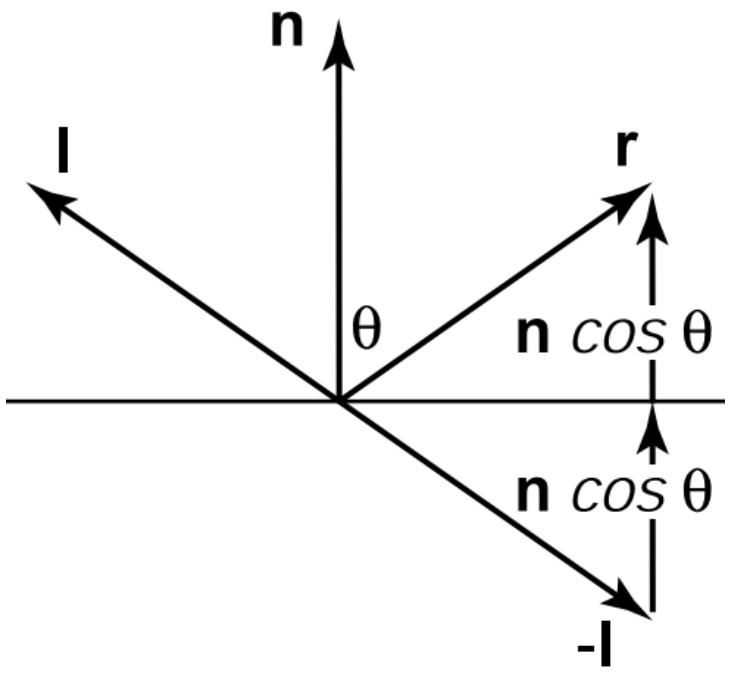
- To light source $\vec{l}$
- To viewer $\vec{v}$
- Normal $\vec{n}$ *determined by local orientation*
- Perfect reflection direction $\vec{r}$ *computed from $\vec{l}$ and $\vec{n}$*

Computing r

How do we compute $\mathbf{r}$?

$\mathbf{l}, \mathbf{n}$ and $\mathbf{r}$ lie in the same plane. Assume all are unit vectors.

Angle of incidence = Angle of reflection



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

Lambertian Surface : Diffuse Lighting

Perfectly diffuse reflector.

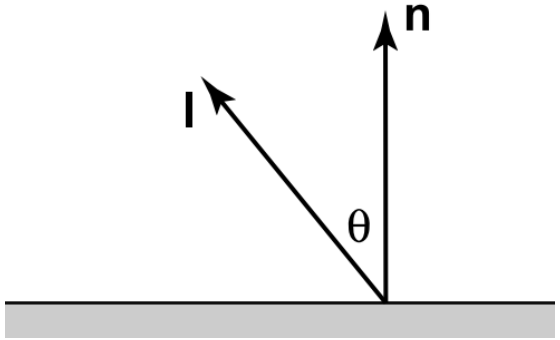
Used for objects with a "matte" (non-shiny) appearance.

For such objects (e.g. paper), color doesn't change much with the gaze direction.

Such objects are called *Lambertian*.

color $\propto cos\ \theta$

color $\propto \max(0, \mathbf{n} \cdot \mathbf{l})$



k_{dr}, k_{dg}, k_{db}: constants determining how much of red, green and blue are reflected.

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

Ambient Lighting

Ambient light is the result of multiple interactions between light sources and the objects in the environment.

Intensity and color depend on both the color of the light(s) and the material properties of the object.

$$\text{color} = k_a I_a$$

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

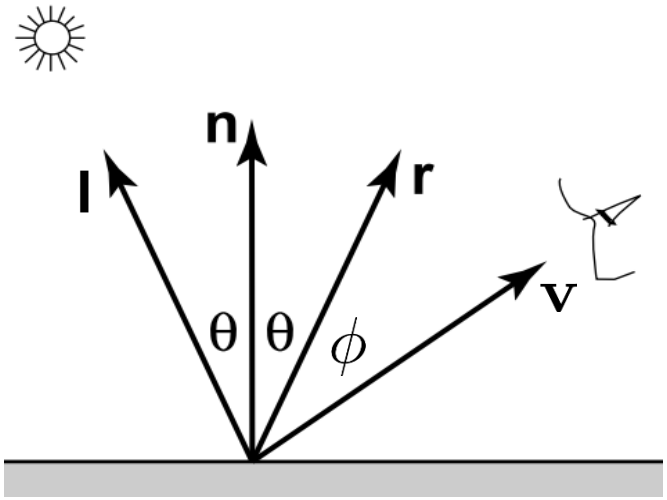
$I_a = (I_{ar}, I_{ag}, I_{ab})$: intensity of ambient light

Specular Reflection

Some surfaces (especially polished ones) have highlights.

color of highlight = color of light

almost no effect of surface color



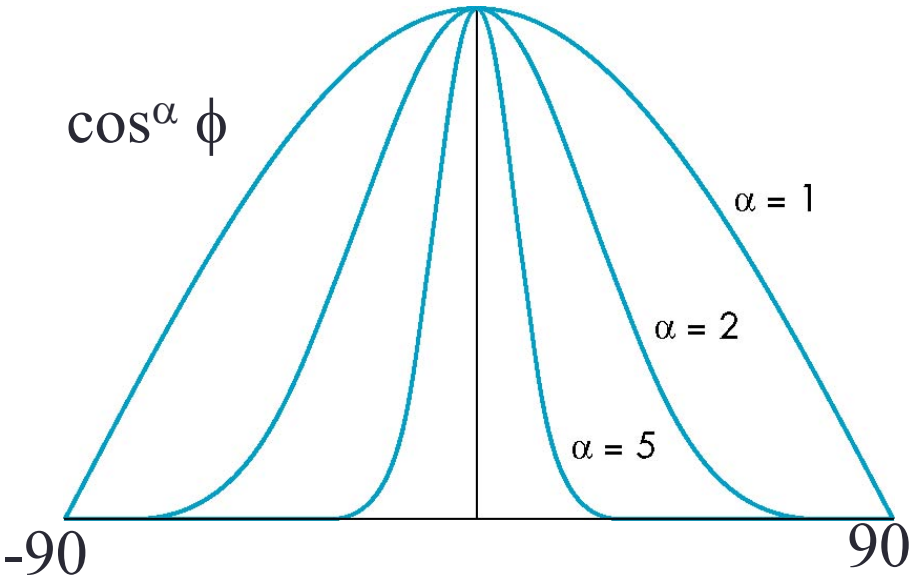
$$\text{color} \propto k_s cos^{\alpha}\phi$$

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

α: shininess coefficient

$$\text{color} \propto k_s \ (\max(0, \mathbf{v} \cdot \mathbf{r}))^{\alpha}$$

Specular Reflection



Values of α between 100 and 200 correspond to metals.

Values between 5 and 10 correspond to plastic.

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

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

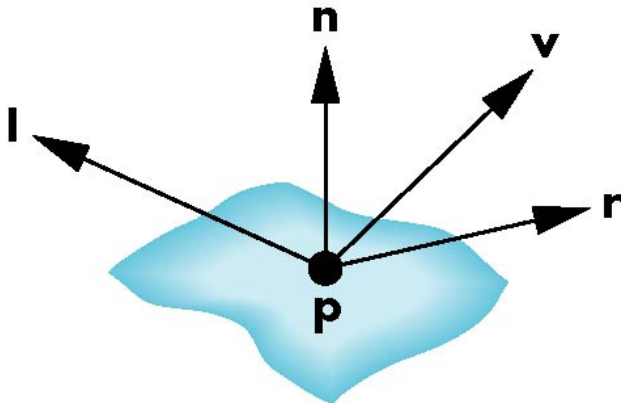
$k_d = (k_{dr}, k_{dg}, k_{db})$: diffuse reflectance

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

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

α: shininess coefficient

Phong Model



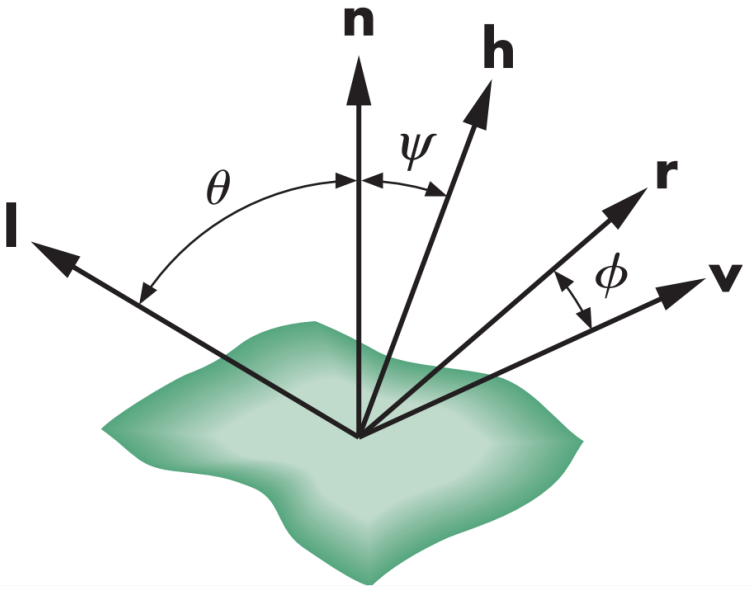
Adding up the components:

$$I = \underbrace{k_d I_d \max(\mathbf{l}\cdot\mathbf{n}, 0)}_{\text{Diffuse component}} + \underbrace{k_a I_a}_{\text{Ambient component}} + \underbrace{k_s I_s (\max(\mathbf{v}\cdot\mathbf{r}, 0))^{\alpha}}_{\text{Specular component}}$$

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

Modified Phong Model

Also called Blinn-Phong Model.



$\mathbf{h} = \mathbf{v} + \mathbf{l}/\|\mathbf{v} + \mathbf{l}\|$

Normalized halfway vector between $\mathbf{l}$ *and* $\mathbf{v}$

Note: $\psi = \phi/2$

We replace $cos(\phi)$ by $cos(\psi)$ in the specular term.

i.e., $\mathbf{v} \cdot \mathbf{r}$ is replaced by $\mathbf{n} \cdot \mathbf{h}$.

Thus we avoid computing $\mathbf{r}$.

α is replaced by a larger number β to match shininess.

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

Attenuation

The intensity of light reaching a point is inversely proportional to the square of its distance from the light source.

To model this, divide the diffuse and specular terms by $(a + bt + ct^2)$.

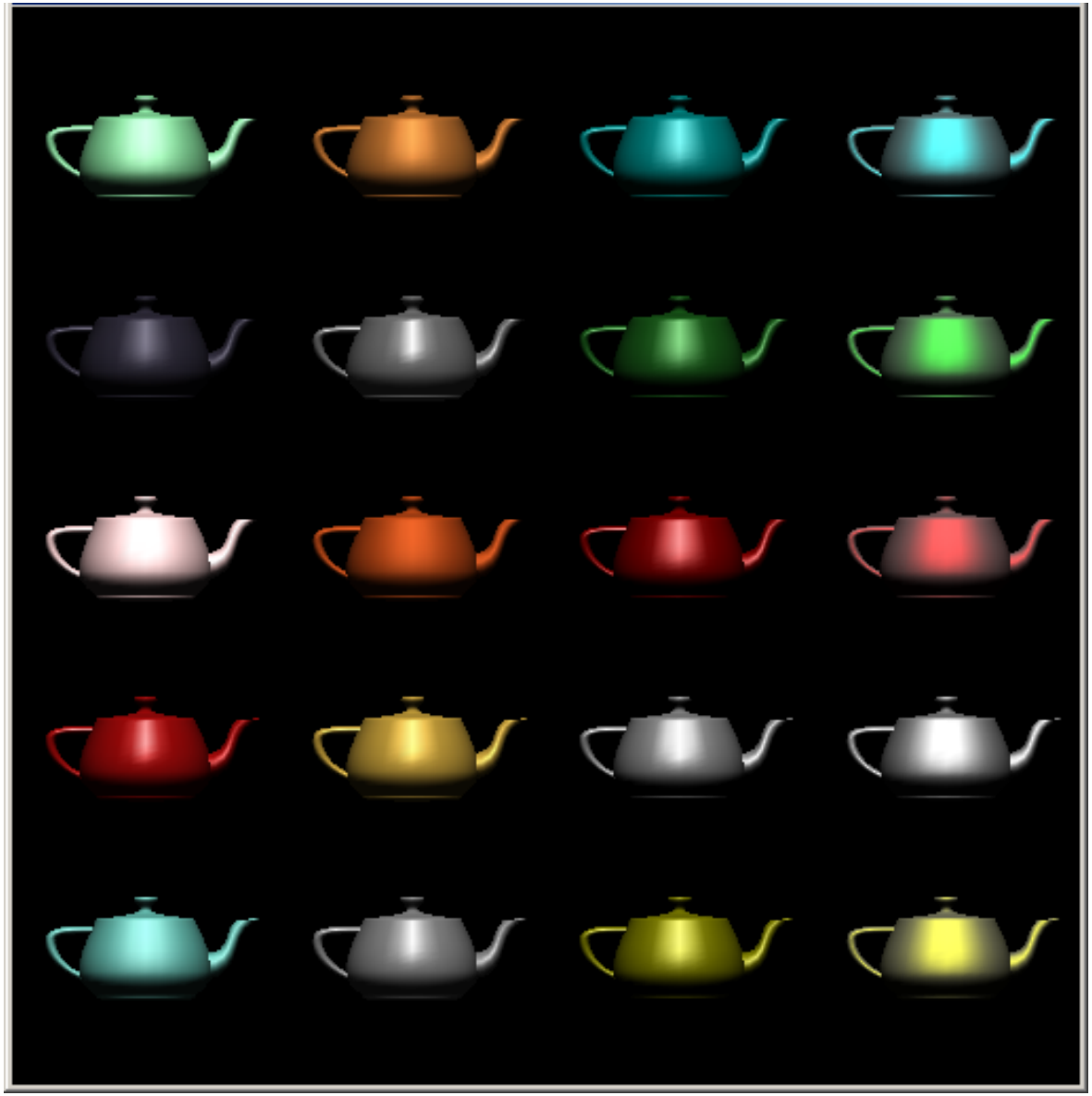
t: distance of the point from light source

a, b and c are some constants. The linear constant and linear terms are used to soften the effect of a point source.

$$I = \frac{1}{a+bt+ct^2}(k_d I_d \max(\mathbf{l} \cdot \mathbf{n}, 0) \; + \; k_s I_s (\max(\mathbf{n} \cdot \mathbf{h}, 0))^{\beta}) \; + \; k_a I_a$$

Blinn-Phong Example

Same teapot with different parameters in the modified phong model.



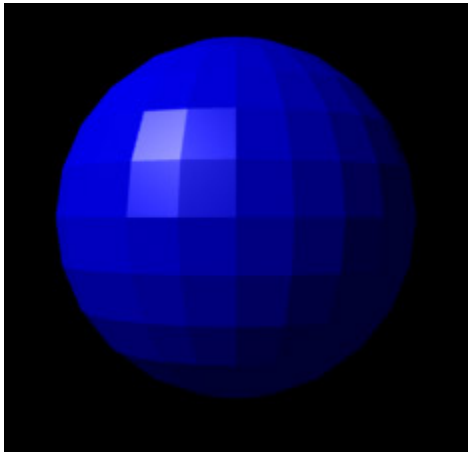
Shading

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.

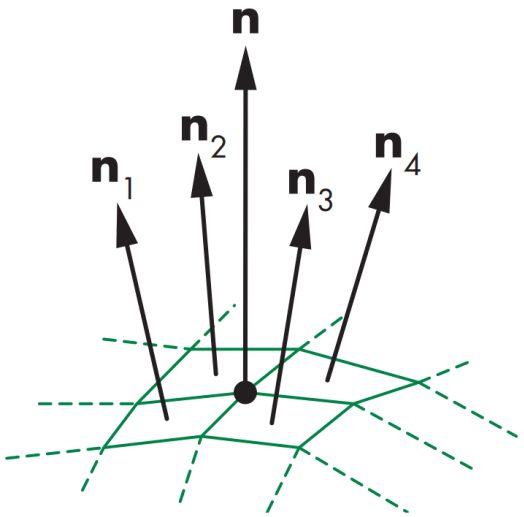
Gouraud Shading:

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:

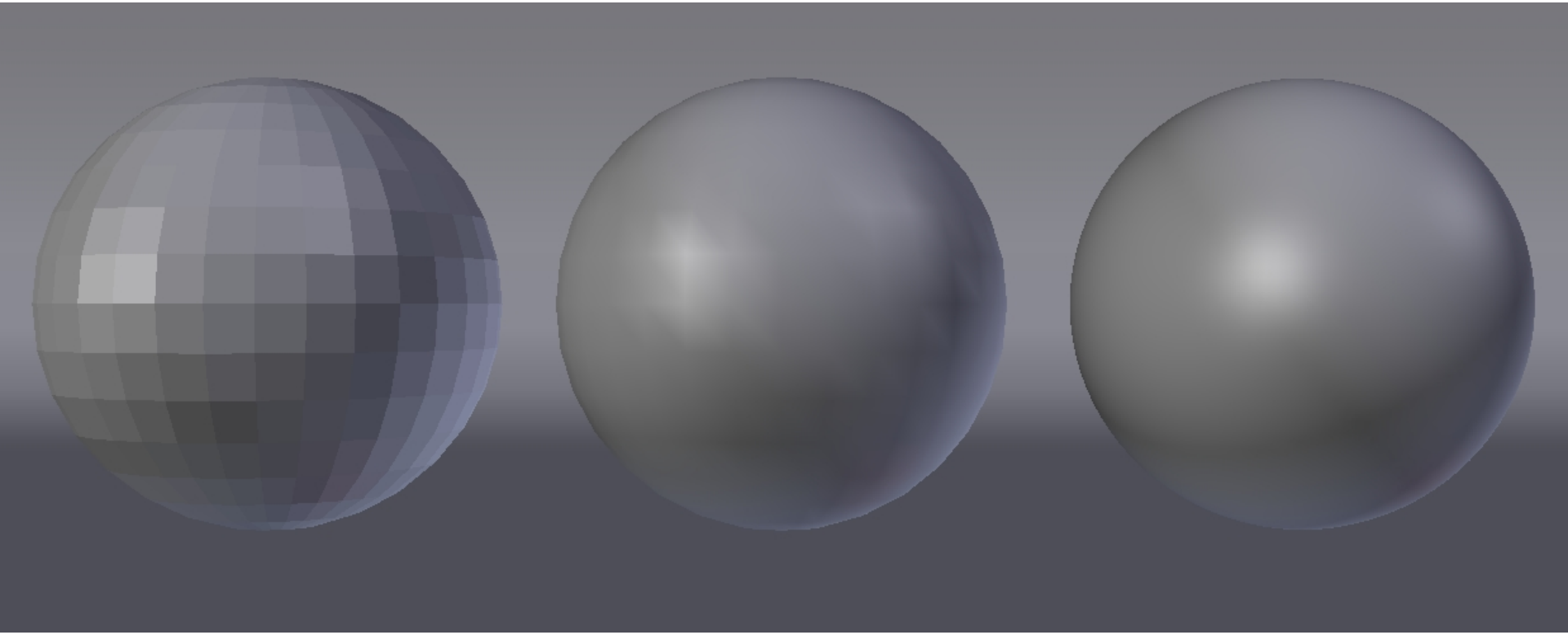
Interpolate the normals at vertices to get the normal at an interior point.

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 **Gouraud** **Phong**

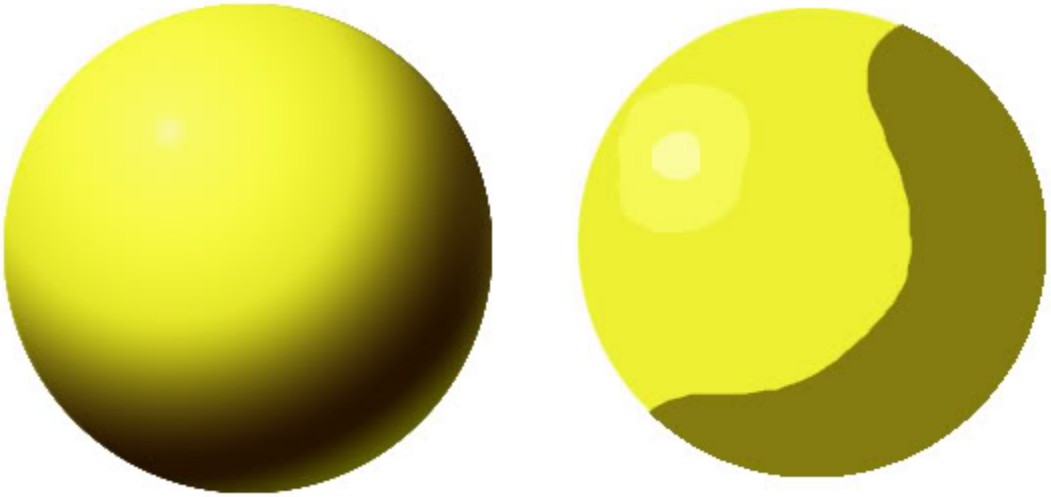
Shading and Lighting are different terms.

Shading: how interpolation is done.

Lighting: the equations used to compute color.

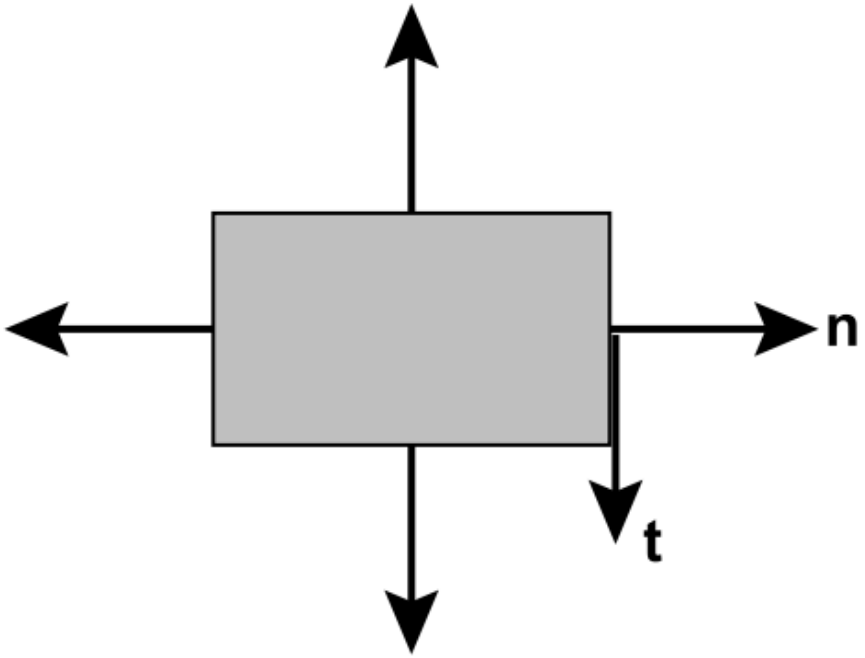
Non Photorealistic Shading

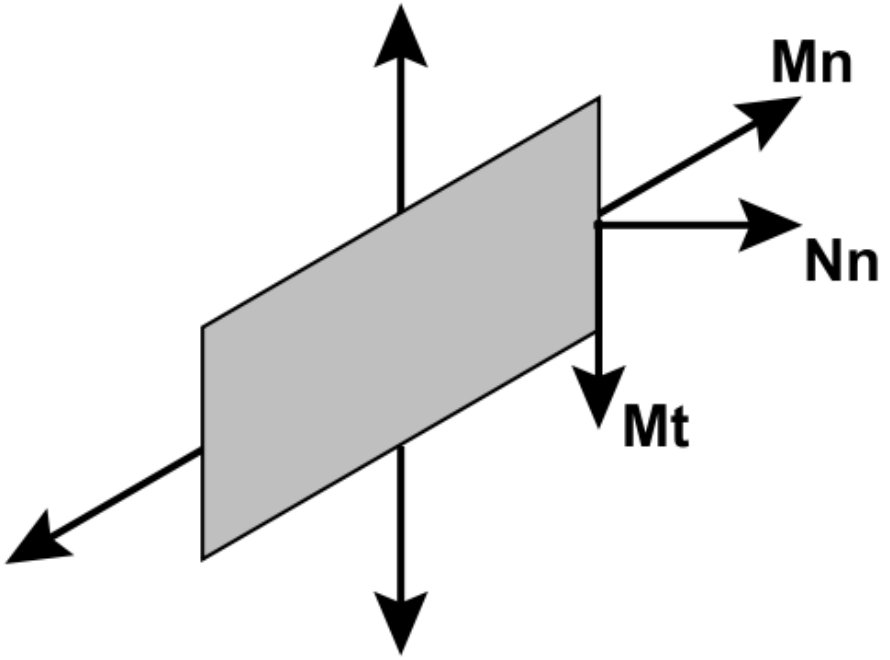
Cartoon Shading (called toon shading or cel shading)



Use only a few discrete shades.

Transformation of Normal Vectors



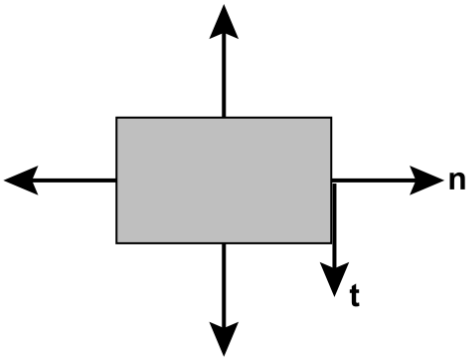


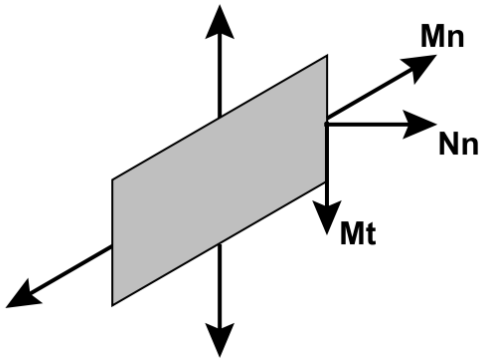
$$\mathbf{n}^T\mathbf{t} = 0$$

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

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
```

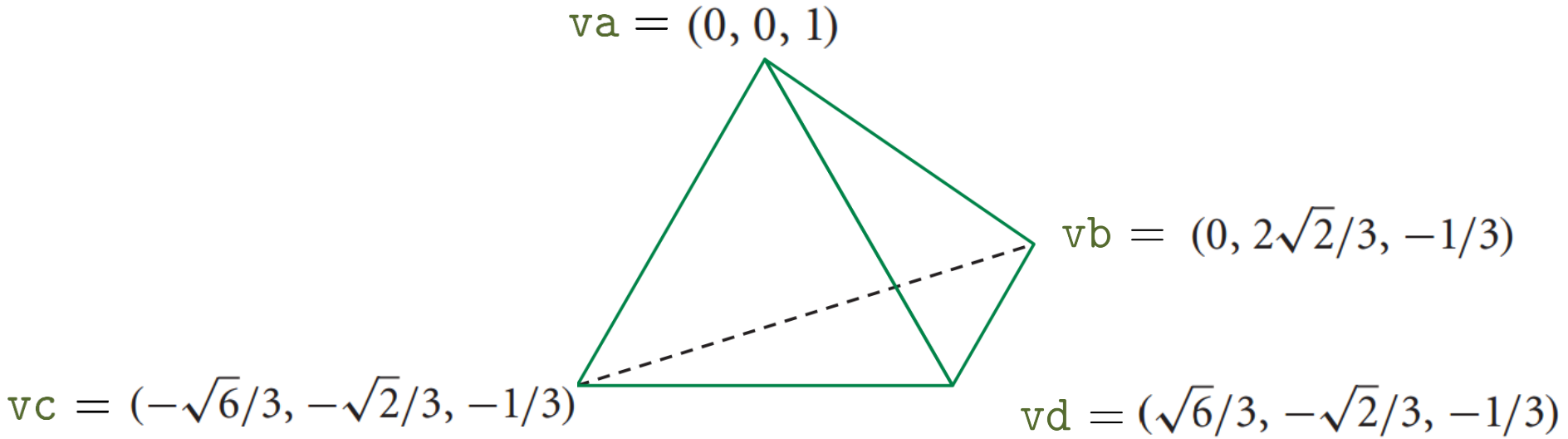
We don't have access to `x` *directly. We can only access it through the functions* `get` *and* `set`.

Coding

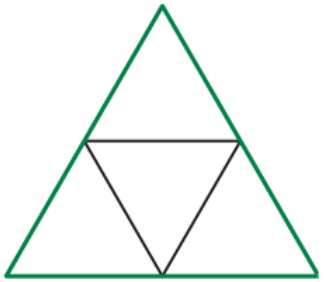
Want: Display a sphere with lighting.

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;
}
```