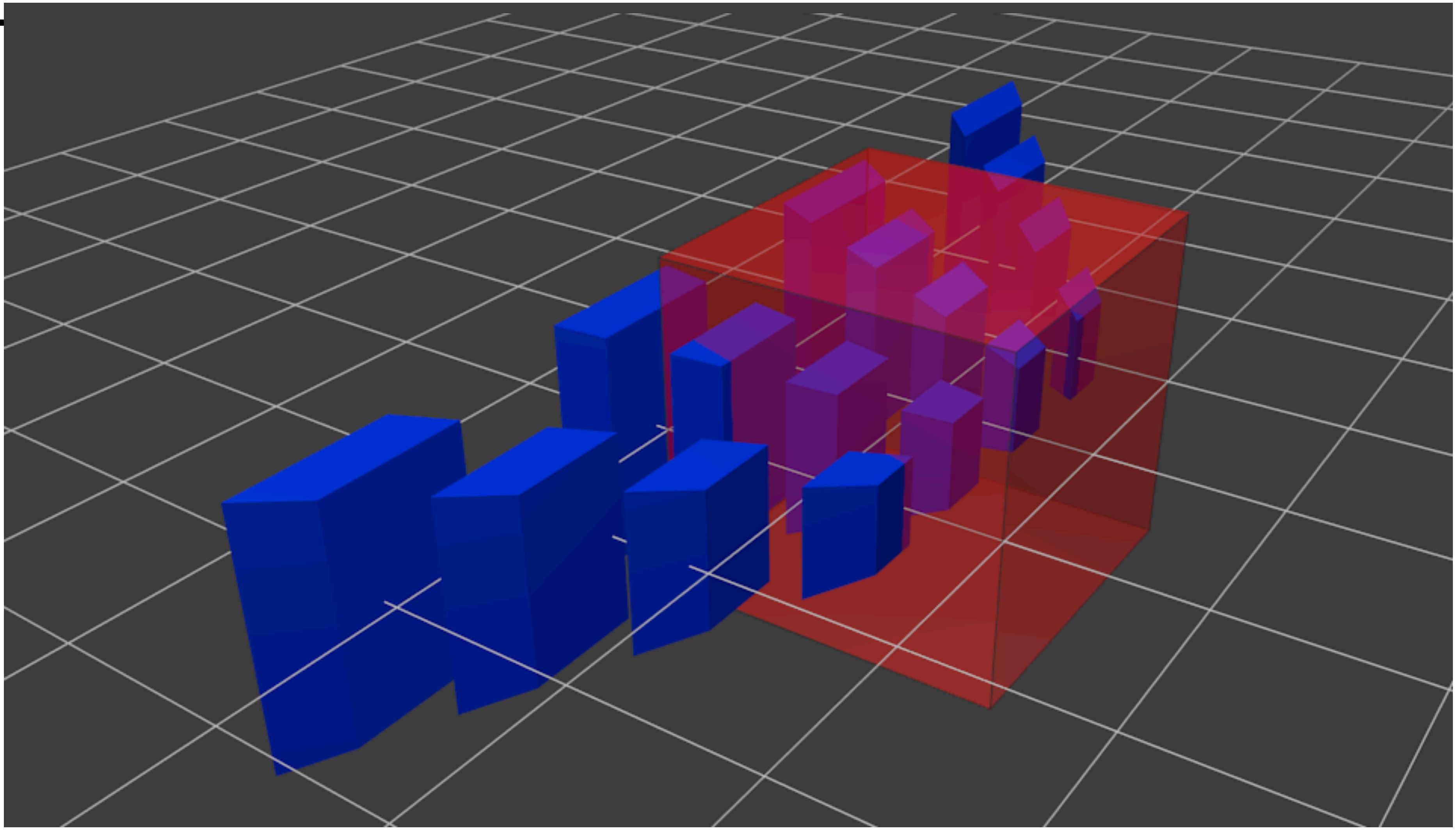
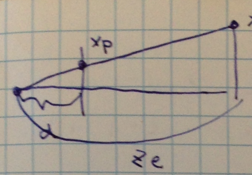
CSE 5542 - Real Time Rendering
Week 6-7-8

OpenGL Perspective Matrix

Courtesy: Prof. H-W. Shen

Perspective Transform





$\cdot (x_e, y_e, z_e)$

$$\frac{x_e}{x_p} = \frac{-z_e}{d} \Rightarrow x_p = \frac{x_e}{\frac{-z_e}{d}}$$

Same for y

$$\frac{y_e}{y_p} = \frac{-z_e}{d} \Rightarrow y_p = \frac{y_e}{\frac{-z_e}{d}}$$

Basic perspective projection matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{-d} & 0 \end{bmatrix} = M_{p_0} \quad \text{why?}$$

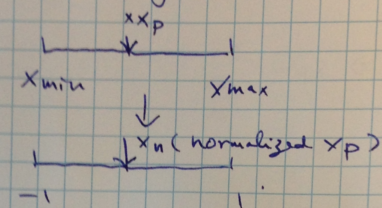
Because

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{-d} & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = \begin{bmatrix} x_e \\ y_e \\ z_e \\ \frac{z_e}{-d} \end{bmatrix}$$

Normalize: $x_p = \frac{x_e}{\frac{z_e}{-d}}$ $\quad y_p = \frac{y_e}{\frac{z_e}{-d}}$ $\quad z_e = -d.$

But we need normalize $(x_{min}, x_{max}] \rightarrow [-1, 1]$
$[y_{min}, y_{max}] \rightarrow [-1, 1]$

Visualize this.



We have:

$$\frac{x_p - x_{min}}{x_{max} - x_{min}} = \frac{x_n - (-1)}{1 - (-1)} = \frac{x_n + 1}{2}$$

$$\Rightarrow x_n = \frac{2(x_p - x_{min})}{x_{max} - x_{min}} - 1$$

$$= \frac{2}{x_{max} - x_{min}} x_p - \frac{x_{max} + x_{min}}{x_{max} - x_{min}}$$

Remember earlier we have

$$x_p = \frac{x_e}{\frac{z}{-d}}.$$

So Now we have $x_n = \frac{2}{x_{max} - x_{min}} \times \frac{x_e}{\frac{z}{-d}} - \frac{x_{max} + x_{min}}{x_{max} - x_{min}}$

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
THE OHIO STATE UNIVERSITY

So modify the basic perspective matrix

$$\begin{bmatrix} \frac{2}{x_{max}-x_{min}} & 0 & \frac{1}{d}\left(\frac{x_{max}+x_{min}}{x_{max}-x_{min}}\right) & 0 \\ 0 & \frac{2}{y_{max}-y_{min}} & \frac{1}{d}\left(\frac{y_{max}+y_{min}}{y_{max}-y_{min}}\right) & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{-d} & 0 \end{bmatrix} = M_{p_1}$$

Why? Because

$$M_{p_1} \times \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2x_e}{x_{max}-x_{min}} + \frac{z_e}{d} \cdot \frac{x_{max}+x_{min}}{x_{max}-x_{min}} \\ \frac{2y_e}{y_{max}-y_{min}} + \frac{z_e}{d} \frac{y_{max}+y_{min}}{y_{max}-y_{min}} \\ z_e \\ \frac{z_e}{-d} \end{bmatrix} = \begin{bmatrix} x_n' \\ y_n' \\ z_n' \\ w_n' \end{bmatrix}$$

Normalize (xn', yn', zn', wn')

$$x_n = \frac{2x_e}{x_{max}-x_{min}} \times \frac{z}{-d} - \frac{x_{max}+x_{min}}{x_{max}-x_{min}}$$

$$y_n = \frac{2y_e}{y_{max}-y_{min}} * \frac{z}{-d} - \frac{y_{max}+y_{min}}{y_{max}-y_{min}}$$

$$z_n = -d$$

$$w_n = 1$$

But we are not done yet.

We need to map z range between [-near, -far] to [-1, 1]. This requires change of M_{p_1}

$$\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ 0 & 0 & A & B \\ 0 & 0 & \frac{1}{-d} & 0 \end{bmatrix} = M_{p_2}$$

(copy x from M_{p_1} in the previous page)

We need to find A & B

$$\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ 0 & 0 & A & B \\ 0 & 0 & \frac{1}{-d} & 0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ -near \\ 1 \end{bmatrix} = \begin{bmatrix} xx \\ xx \\ A \cdot -near + B \\ \frac{near}{d} \end{bmatrix}$$

(xx means "don't care")

$$\Rightarrow \frac{A \cdot -near + B}{\frac{near}{d}} = -1$$

$$\Rightarrow -Ad + \frac{B \cdot d}{near} = -1 \rightarrow (1)$$

Also:

$$\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ 0 & 0 & A & B \\ 0 & 0 & -\frac{1}{d} & 0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ -far \\ 1 \end{bmatrix} = \begin{bmatrix} x\,x \\ x\,x \\ A\cdot -far + B \\ \frac{far}{d} \end{bmatrix}$$

$$\Rightarrow \frac{A\cdot -far + B}{\frac{far}{d}} = +1$$

$$\Rightarrow -Ad + \frac{B\cdot d}{far} = 1 \quad \text{—— (2)}$$

Solve A, B from (1) & (2):

$$A = \frac{N+F}{(N-F)d} \qquad B = \frac{2N\times F}{d(N-F)}$$

where $N = near$ $F = far$

so we have the new projection matrix.

$$\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ 0 & 0 & \frac{-(N+F)}{(F-N)d} & \frac{-2NF}{d(F-N)} \\ 0 & 0 & \frac{1}{-d} & 0 \end{bmatrix}$$

That is:

$$\begin{bmatrix} \frac{2}{X_{max}-X_{min}} & 0 & \frac{X_{max}+X_{min}}{X_{max}-X_{min}} & 0 \\ 0 & \frac{2}{y_{max}-y_{min}} & \frac{y_{max}+y_{min}}{y_{max}-y_{min}} & 0 \\ 0 & 0 & \frac{-(N+F)}{(F-N)d} & \frac{-2NF}{(F-N)d} \\ 0 & 0 & \frac{1}{-d} & 0 \end{bmatrix} = M_{p_2}$$

Scale ~~Multiply~~ the whole matrix by d.

$$\begin{bmatrix} \frac{2d}{X_{max}-X_{min}} & 0 & \frac{X_{max}+X_{min}}{X_{max}-X_{min}} & 0 \\ 0 & \frac{2d}{y_{max}-y_{min}} & \frac{y_{max}+y_{min}}{y_{max}-y_{min}} & 0 \\ 0 & 0 & -\frac{(N+F)}{F-N} & \frac{-2NF}{F-N} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

now, if we set the image plane at the near plane, that is, $d = N$.

then we have the following Final matrix

$$
\begin{bmatrix}
\frac{2N}{X_{max}-X_{min}} & 0 & \frac{X_{max}+X_{min}}{X_{max}-X_{min}} & 0 \\
0 & \frac{2N}{Y_{max}-Y_{min}} & \frac{Y_{max}+Y_{min}}{Y_{max}-Y_{min}} & 0 \\
0 & 0 & \frac{-(N+F)}{F-N} & \frac{-2NF}{F-N} \\
0 & 0 & -1 & 0
\end{bmatrix}
$$

↑

OpenGL perspective projection Matrix

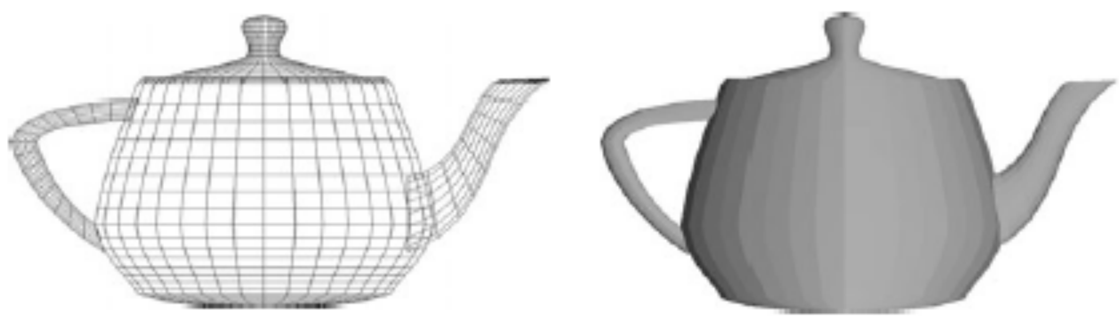
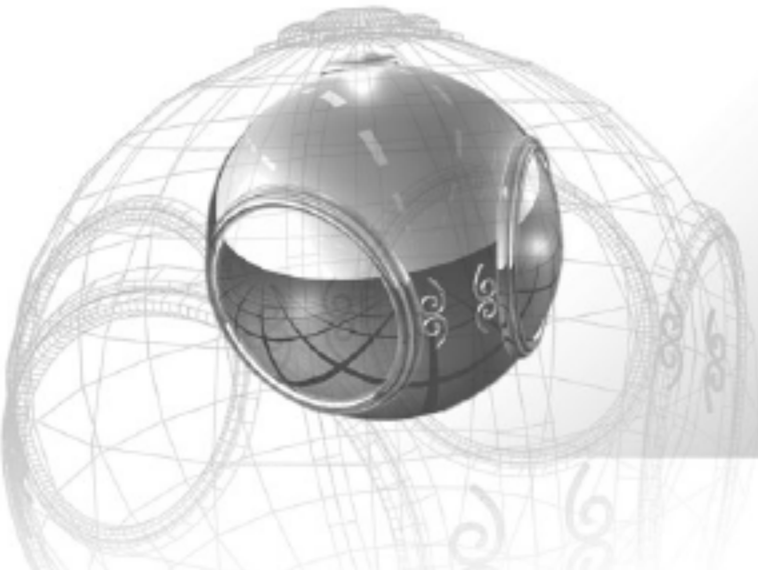


FIGURE 10.41 Rendered teapots.



CHAPTER 10

CURVES AND SURFACES

Modeling

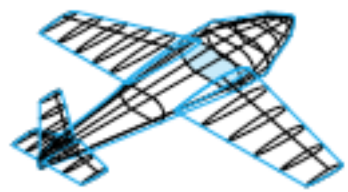


FIGURE 10.5 Model airplane

FIGURE 10.6 Cross-section curve.

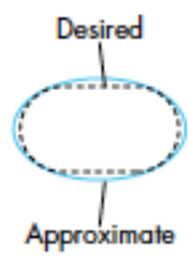


FIGURE 10.7 Approximation of cross-section curve.

Parametric Curve

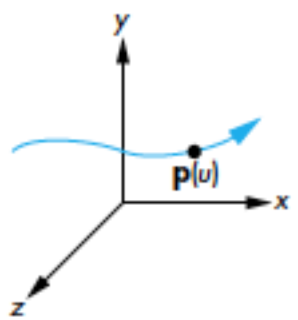


FIGURE 10.1 Parametric

$$x = x(u),$$
$$y = y(u),$$
$$z = z(u).$$

$$\frac{d\mathbf{p}(u)}{du} = \begin{bmatrix} \frac{dx(u)}{du} \\ \frac{dy(u)}{du} \\ \frac{dz(u)}{du} \end{bmatrix}$$

Parametric Curve

Consider a curve of the form[2]

$$\mathbf{p}(u) = \begin{bmatrix} x(u) \\ y(u) \\ z(u) \end{bmatrix}.$$

A polynomial parametric curve of degree[3] n is of the form

$$\mathbf{p}(u) = \sum_{k=0}^{n} u^k \mathbf{c}_k,$$

where each $\mathbf{c}_k$ has independent x, y, and z components; that is,

$$\mathbf{c}_k = \begin{bmatrix} c_{xk} \\ c_{yk} \\ c_{zk} \end{bmatrix}.$$

The $n+1$ column matrices $\{\mathbf{c}_k\}$ are the coefficients of $\mathbf{p}$; they give us $3(n+1)$ degrees of freedom in how we choose the coefficients of a particular $\mathbf{p}$. There is no coupling, however, among the x, y, and z components, so we can work with three independent equations, each of the form

$$p(u) = \sum_{k=0}^{n} u^k c_k,$$

where p is any one of x, y, or z. There are $n+1$ degrees of freedom in $p(u)$. We can define our curves for any range interval of u:

$$u_{min} \le u \le u_{max};$$

however, with no loss of generality (see Exercise 10.3), we can assume that $0 \le u \le 1$. As the value of u varies over its range, we define a **curve segment**, as shown in Figure 10.3.

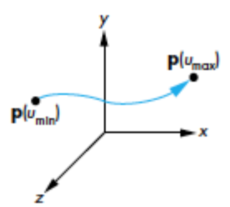


FIGURE 10.3 Curve segment.

Cubic Parametric Curves

$$\mathbf{p}(u) = \mathbf{c}_0 + \mathbf{c}_1 u + \mathbf{c}_2 u^2 + \mathbf{c}_3 u^3 = \sum_{k=0}^{3} \mathbf{c}_k u^k = \mathbf{u}^T \mathbf{c},$$

where

$$\mathbf{c} = \begin{bmatrix} \mathbf{c}_0 \\ \mathbf{c}_1 \\ \mathbf{c}_2 \\ \mathbf{c}_3 \end{bmatrix}, \qquad \mathbf{u} = \begin{bmatrix} 1 \\ u \\ u^2 \\ u^3 \end{bmatrix}, \qquad \mathbf{c}_k = \begin{bmatrix} c_{kx} \\ c_{ky} \\ c_{kz} \end{bmatrix}.$$

Control Points

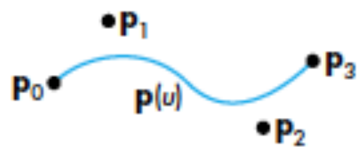


FIGURE 10.9 Curve segment and control points.

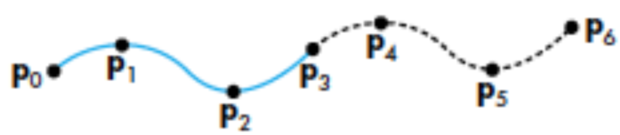


FIGURE 10.10 Joining of interpolating segments.

Bezier

$$\mathbf{p}(u) = \sum_{i=0}^{3} b_i(u)\mathbf{p}_i,$$

$$\mathbf{p}(u) = \mathbf{b}(u)^T\mathbf{p},$$

$$\mathbf{b}(u) = \mathbf{M}_B^T\mathbf{u} = \begin{bmatrix} (1-u)^3 \\ 3u(1-u)^2 \\ 3u^2(1-u) \\ u^3 \end{bmatrix}.$$

$$\mathbf{M}_B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}.$$

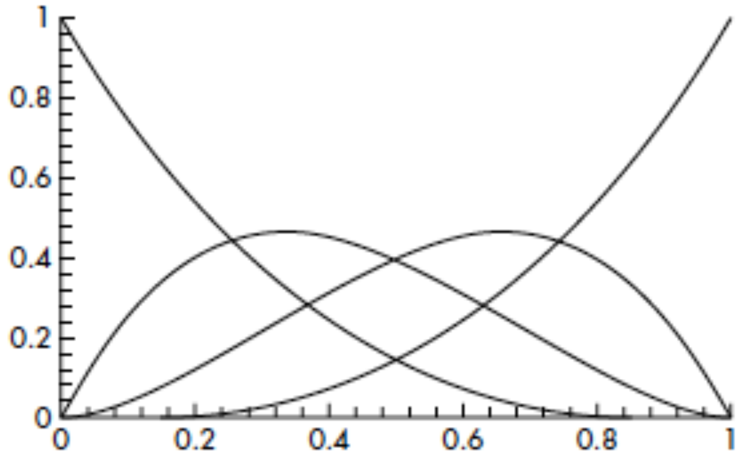


FIGURE 10.18 Blending polynomials for the Bézier cubic.

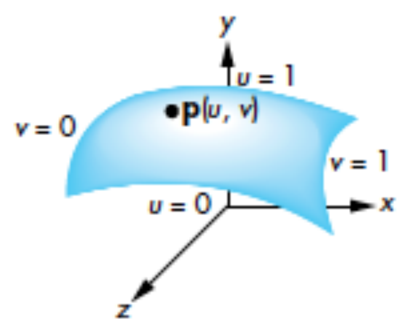
Parametric Surface

$$\mathbf{p}(u, v) = \begin{bmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{bmatrix}$$

$$x = x(u, v), \quad y = y(u, v), \quad z = z(u, v),$$

$$\frac{\partial \mathbf{p}}{\partial u} = \begin{bmatrix} \frac{\partial x(u,v)}{\partial u} \\ \frac{\partial y(u,v)}{\partial u} \\ \frac{\partial z(u,v)}{\partial u} \end{bmatrix} \quad \frac{\partial \mathbf{p}}{\partial v} = \begin{bmatrix} \frac{\partial x(u,v)}{\partial v} \\ \frac{\partial y(u,v)}{\partial v} \\ \frac{\partial z(u,v)}{\partial v} \end{bmatrix}$$

$/\partial v.$



$\mathbf{p}(u, v) =$

FIGURE 10.4 Surface patch.

Parametric Surface

$$\mathbf{p}(u, v) = \begin{bmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{bmatrix} = \sum_{i=0}^{n} \sum_{j=0}^{m} \mathbf{c}_{ij} u^i v^j .$$

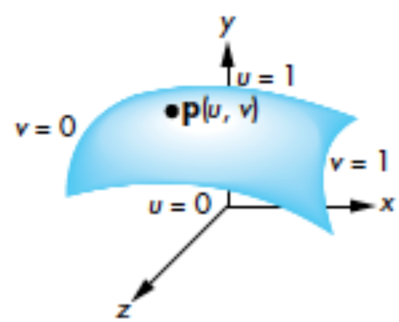


FIGURE 10.4 Surface patch.

Bezier Surface Patches

$$\mathbf{p}(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(u) b_j(v) \mathbf{p}_{ij} = \mathbf{u}^T \mathbf{M}_B \mathbf{P} \mathbf{M}_B^T \mathbf{v}.$$

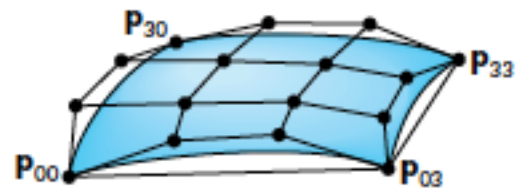


FIGURE 10.20 Bézier patch.

Subdivision

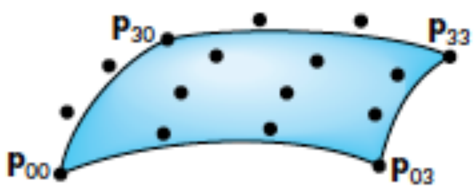


FIGURE 10.37 Cubic Bézier surface.

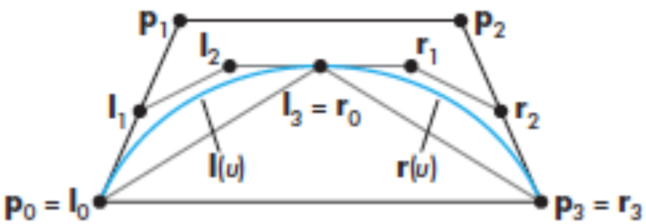


FIGURE 10.34 Convex hulls and control points.

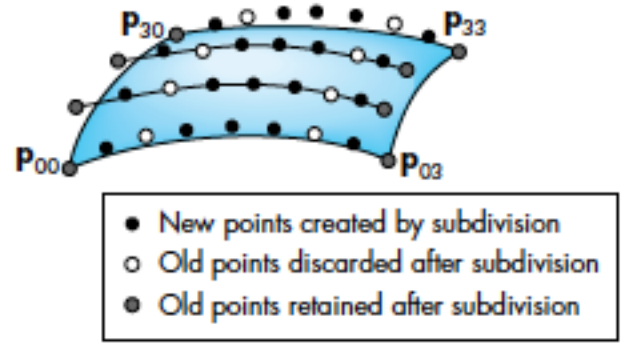


FIGURE 10.38 First subdivision of surface.

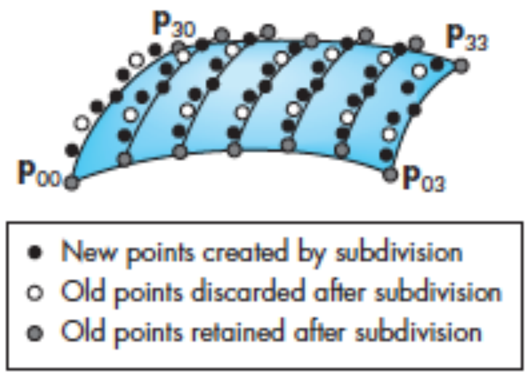


FIGURE 10.39 Points after second subdivision.

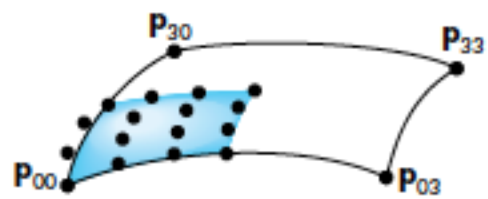


FIGURE 10.40 Subdivided quadrant.

Code

```
void draw_patch(point4 p[4][4])
{
    points[n] = p[0][0];
    n++;
    points[n] = p[3][0];
    n++;
    points[n] = p[3][3];
    n++;
    points[n] = p[0][3];
    n++;
}
```

```
void divide_curve(point4 c[4], point4 r[4], point4 l[4])
{

/* division of convex hull of Bezier curve */

   int i;
   point4 t;
   for(i=0;i<3;i++)

       l[0][i]=c[0][i];
       r[3][i]=c[3][i];
       l[1][i]=(c[1][i]+c[0][i])/2;
       r[2][i]=(c[2][i]+c[3][i])/2;
       t[i]=(l[1][i]+r[2][i])/2;
       l[2][i]=(t[i]+l[1][i])/2;
       r[1][i]=(t[i]+r[2][i])/2;
       l[3][i]=r[0][i]=(l[2][i]+r[1][i])/2;

   for(i=0; i<4; i++) l[i][3] = r[i][3] = 1.0;
}
```

```
void divide_patch(point4 p[4][4], int n)
{
   point4 q[4][4], r[4][4], s[4][4], t[4][4];
   point4 a[4][4], b[4][4];
   int k;
   if(n==0) draw_patch(p); /* draw patch if recursion done */

/* subdivide curves in u direction, transpose results, divide
in u direction again (equivalent to subdivision in v) */

   else
       {
       for(k=0; k<4; k++) divide_curve(p[k], a[k], b[k]);
       transpose4(a);
       transpose4(b);
       for(k=0; k<4; k++)
          {
          divide_curve(a[k], q[k], r[k]);
          divide_curve(b[k], s[k], t[k]);
          }

/* recursive division of 4 resulting patches */

       divide_patch(q, n-1);
       divide_patch(r, n-1);
       divide_patch(s, n-1);
       divide_patch(t, n-1);
      }
}
```

Code for GL

Courtesy:
http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

GLM

OpenGL Mathematics (GLM) is a header only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL).

Provides classes and functions designed and implemented following as strictly as possible the GLSL conventions and functionalities.

When a programmer knows GLSL, he knows GLM as well, making it really easy to use.

C++

```
glm::mat4 myMatrix;
glm::vec4 myVector;

// fill myMatrix and myVector somehow

glm::vec4 transformedVector = myMatrix * myVector;

 // Again, in this order ! this is important.
```

GLSL

```
mat4 myMatrix;
vec4 myVector;

// fill myMatrix and myVector somehow
vec4 transformedVector = myMatrix * myVector;

// Yeah, it's pretty much the same than GLM
```

Identity

```
glm::mat4 myIdentityMatrix = glm::mat4(1.0f);
```

Translate

GLM -

```
#include <glm/transform.hpp> // after <glm/glm.hpp>
glm::mat4 myMatrix = glm::translate(10.0f, 0.0f, 0.0f);
glm::vec4 myVector(10.0f, 10.0f, 10.0f, 0.0f);
glm::vec4 transformedVector = myMatrix * myVector;
```

GLSL -

```
vec4 transformedVector = myMatrix * myVector;
```

Scaling

```
// Use #include <glm/gtc/matrix_transform.hpp> and #include
<glm/gtx/transform.hpp>

glm::mat4 myScalingMatrix = glm::scale(2.0f, 2.0f ,2.0f);
```

Rotation

```
// Use #include <glm/gtc/matrix_transform.hpp> and #include
<glm/gtx/transform.hpp>

glm::vec3 myRotationAxis( ??, ??, ??);

glm::rotate( angle_in_degrees, myRotationAxis );
```

Accumulating Transforms

TransformedVector =
TranslationMatrix * RotationMatrix * ScaleMatrix * OriginalVector;

In Code

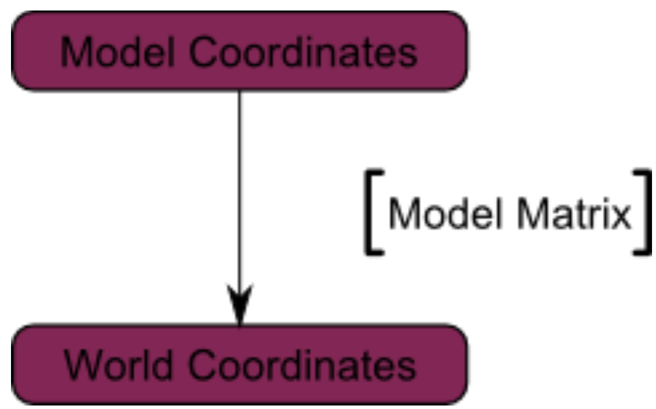
GLM

```
glm::mat4 myModelMatrix = myTranslationMatrix * myRotationMatrix * myScaleMatrix;

glm::vec4 myTransformedVector = myModelMatrix * myOriginalVector;
```

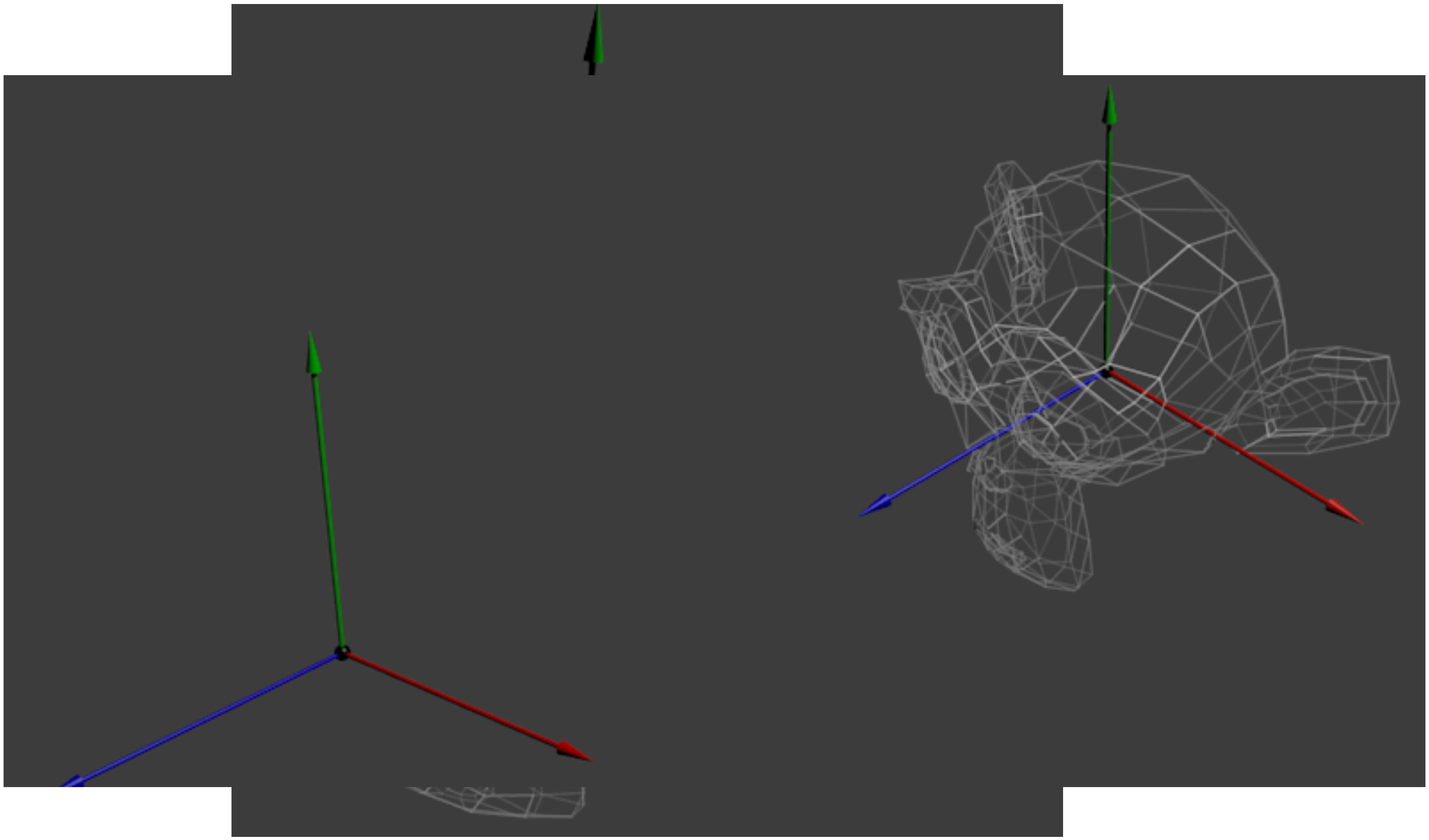
GLSL

```
mat4 transform = mat2 * mat1;
vec4 out_vec = transform * in_vec;
```

In Diagrams

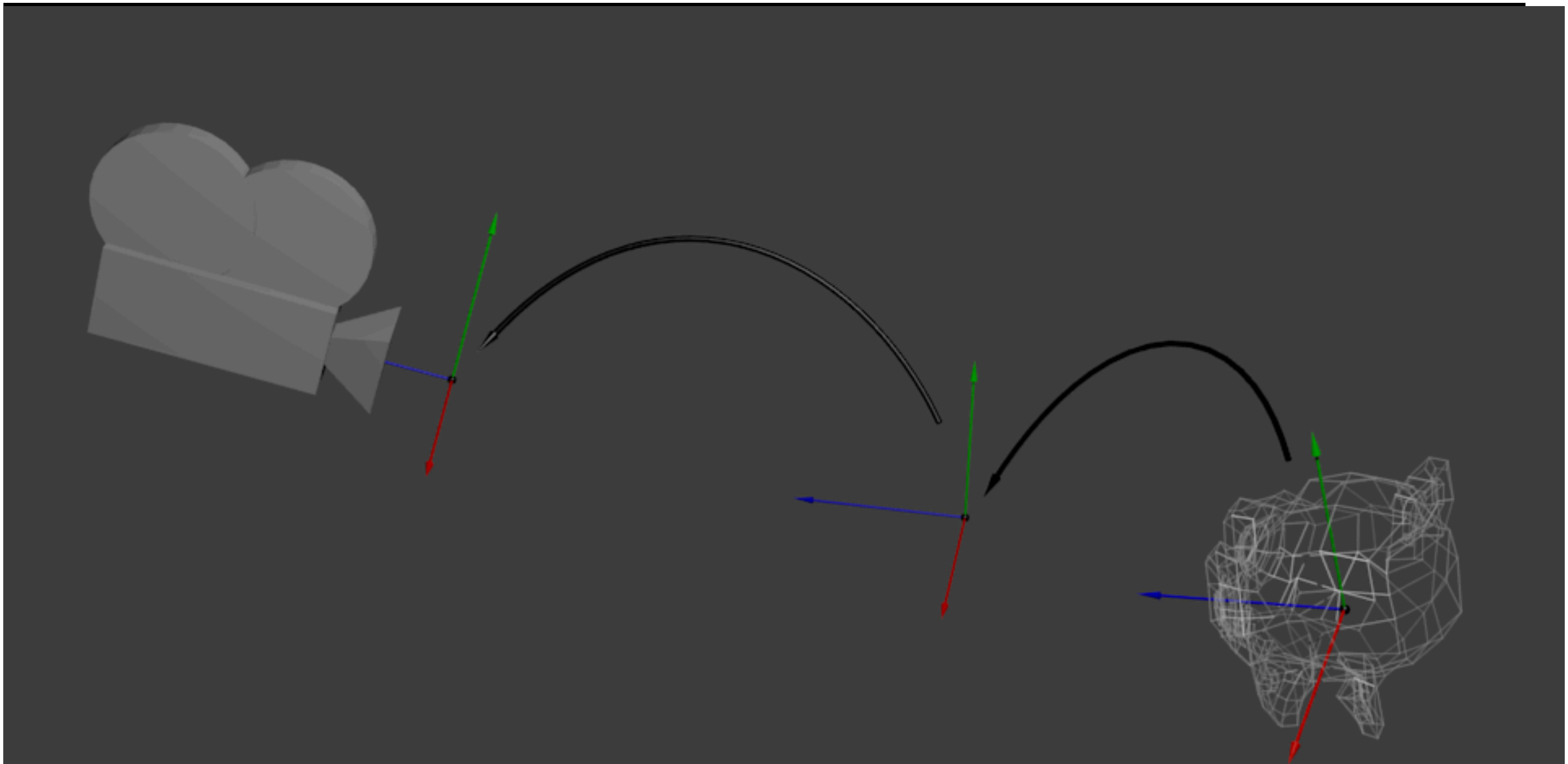


In Pictures




Camera

Camera/Eye Space

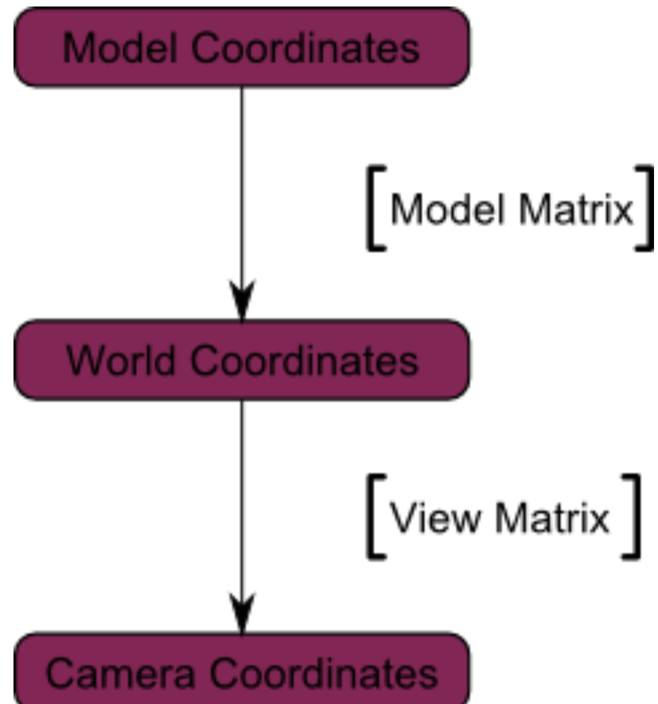


```
glm::mat4 ViewMatrix = glm::translate(-3.0f, 0.0f ,0.0f);
```

Camera/Eye Space

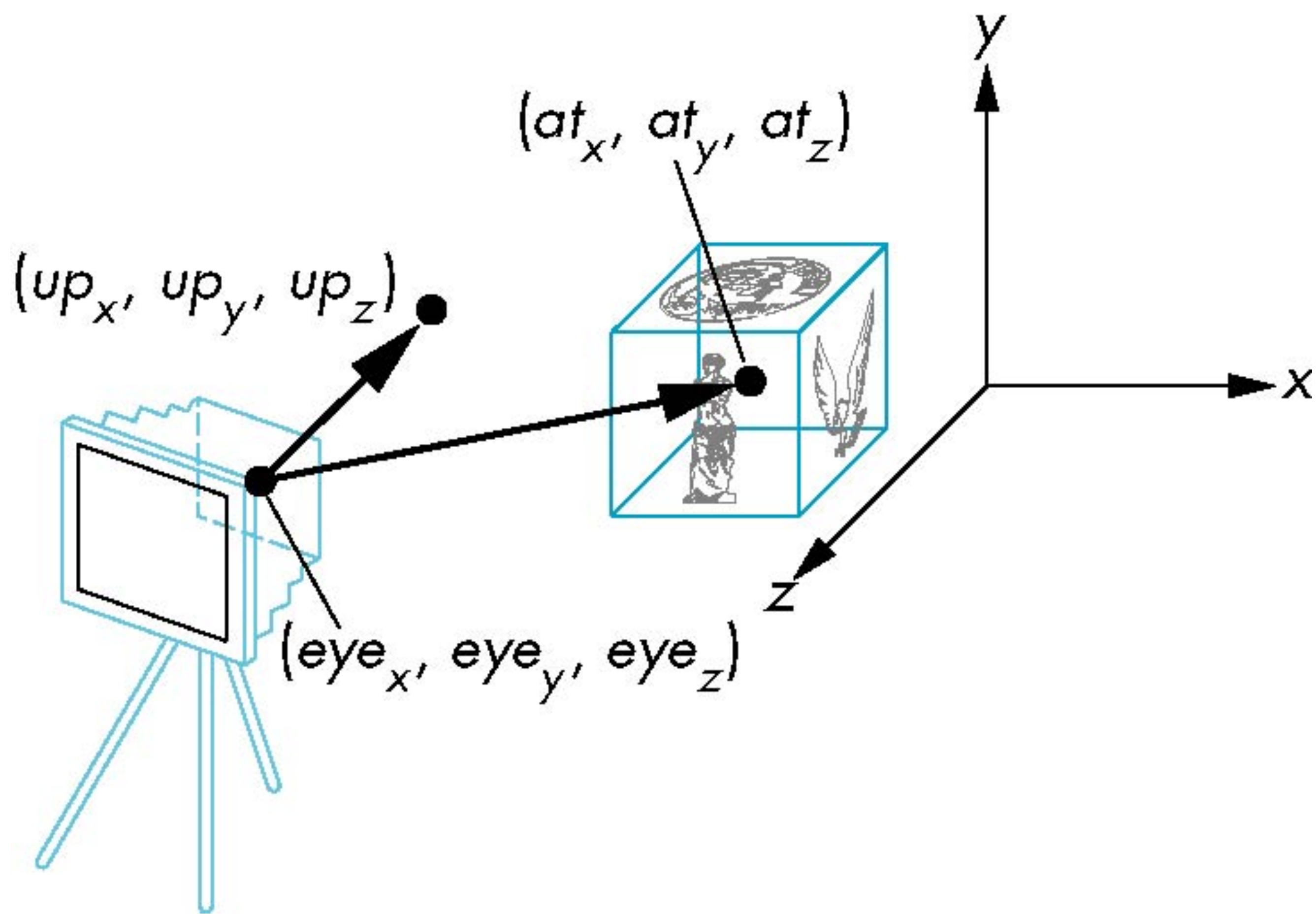
```
glm::mat4 CameraMatrix = glm::LookAt (
    cameraPosition, // the position of your camera, in world space
    cameraTarget,   // where you want to look at, in world space
    upVector        // probably glm::vec3(0,1,0),
                    // but (0,-1,0) would make you looking upside-down
);
```

<ins>Transform objects from world to eye space</ins>

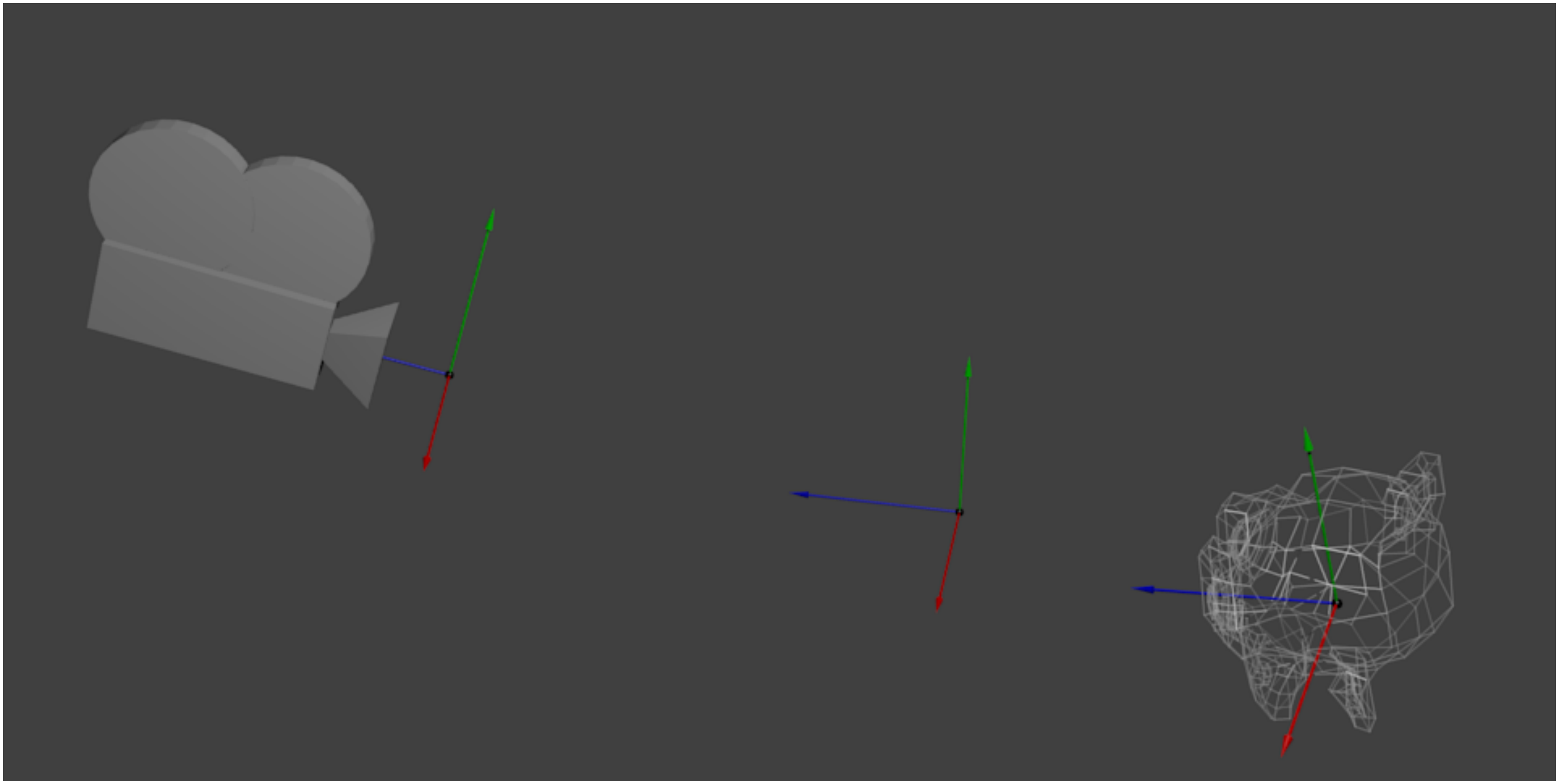


gluLookAt

LookAt(eye, at, up)



Camera Coordinate Frame



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Camera Space

Right hand coordinate system

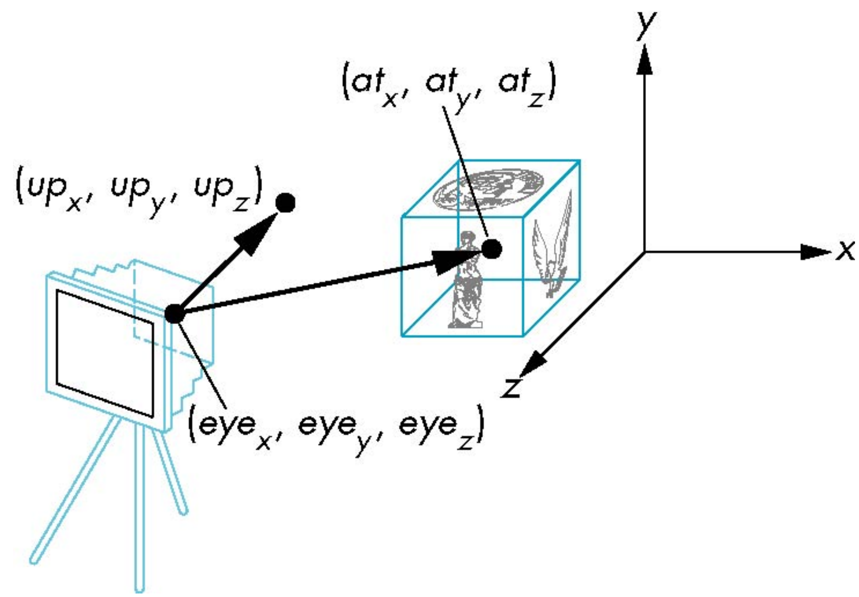
$$\vec{n} = at - eye$$

$$\vec{n} = \frac{\vec{n}}{\|\vec{n}\|}$$

$$\vec{u} = up \times \vec{n}$$

$$v = \vec{n} \times \vec{u}$$

$$\boldsymbol{V} = \begin{pmatrix} u_x & u_y & u_z & -eye \cdot \mathbf{u} \\ v_x & v_y & v_z & -eye \cdot \mathbf{v} \\ n_x & n_y & n_z & -eye \cdot \mathbf{n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Old Style

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,1,0,0,0,0,1,0);
    …
}
```

New World

- Create a view matrix

```
view = glm::lookAt(glm::vec3(0.0, 2.0, 2.0), glm::vec3(0.0, 0.0, 0.0), glm::vec3(0.0, 1.0, 0.0));
```

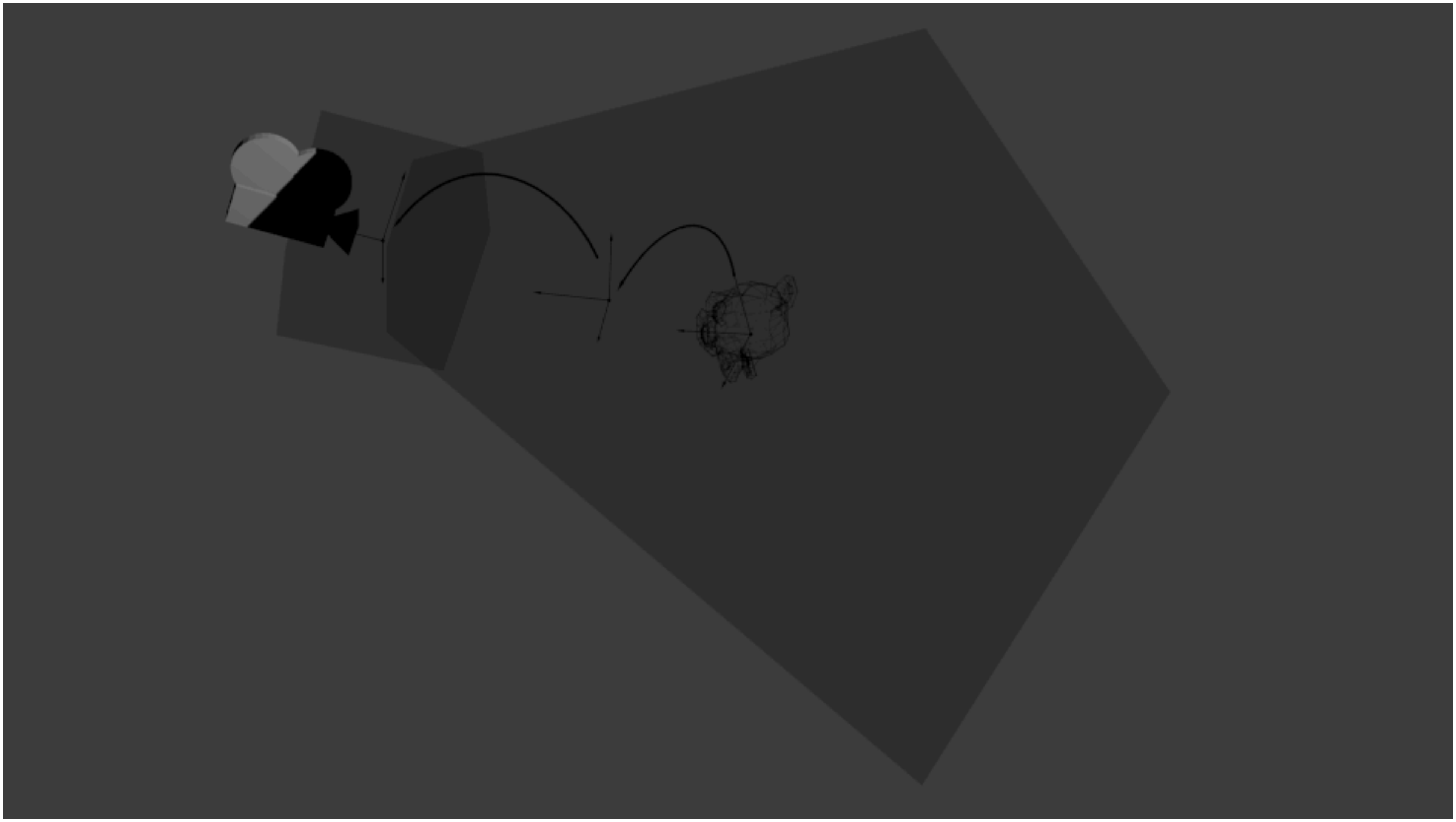
- Combine with modeling matrices

```
glm::mat4 model = glm::mat4(1.0f);
model = glm::rotate(model, angle, glm::vec3(0.0f, 0.0f, 1.0f));
model = glm::scale(model, scale_size, scale_size, scale_size);

glm::mat4 modelview = view * model;
```

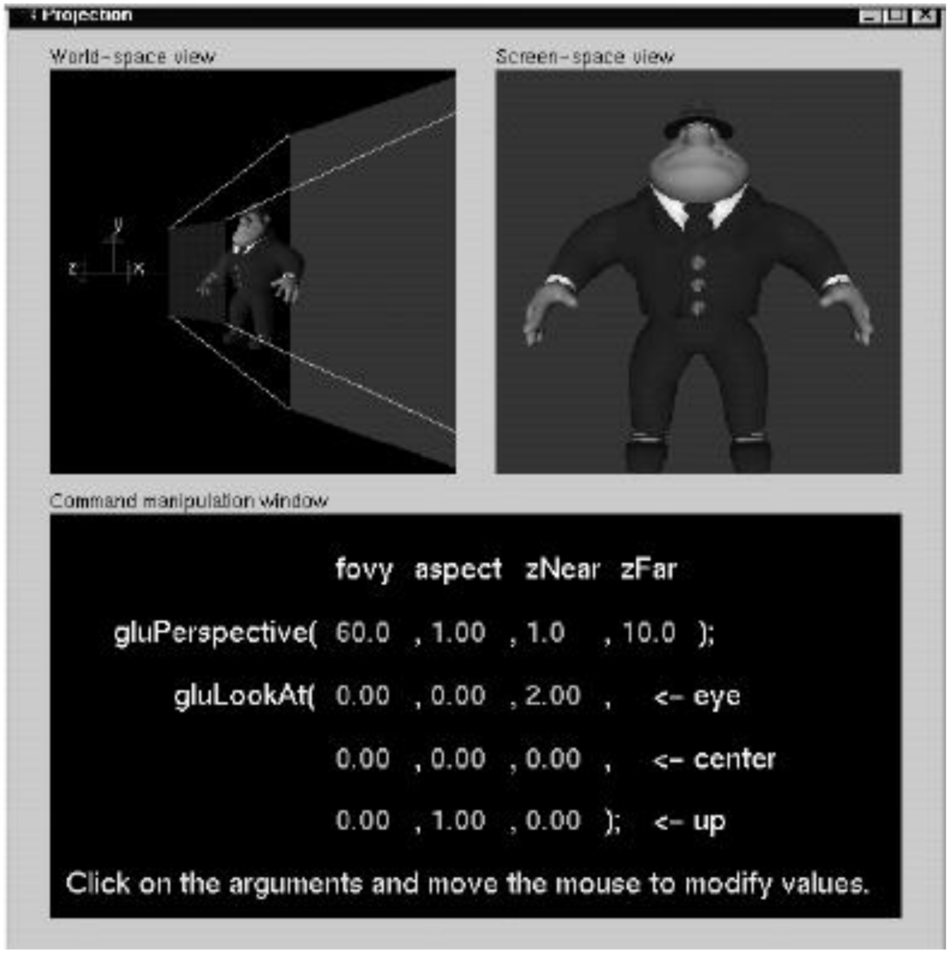
Working with Old World

```
glMatrixMode(GL_MODELVIEW);
glLoadMatrixf(&modelview[0][0]);

// begin to draw your geometry
…
```

Projection Matrices



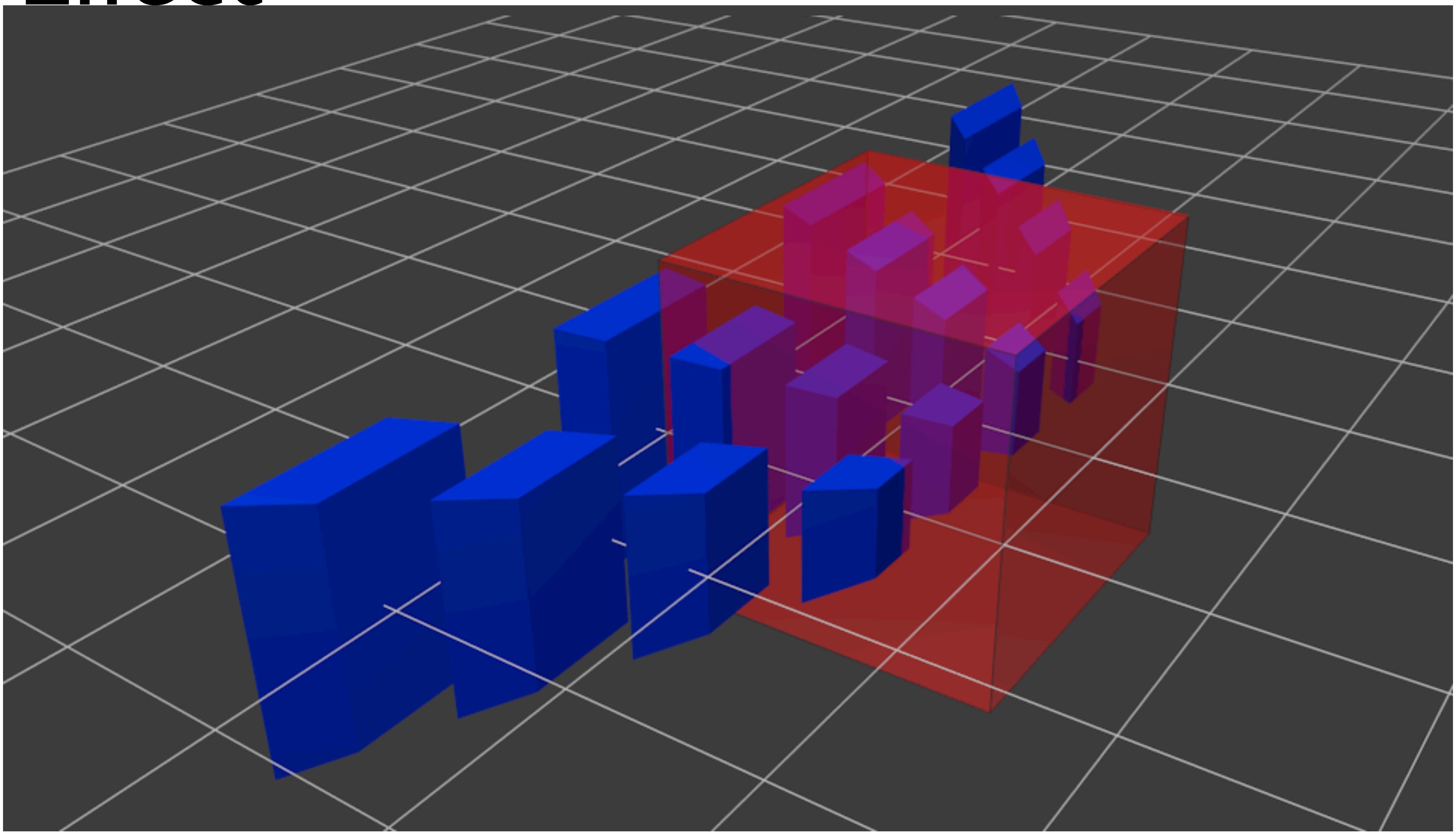
Demo


Projection
World-space view
Screen-space view
Command manipulation window
fovy aspect zNear zFar
gluPerspective(60.0 , 1.00 , 1.0 , 10.0);
gluLookAt(0.00 , 0.00 , 2.00 , <- eye
0.00 , 0.00 , 0.00 , <- center
0.00 , 1.00 , 0.00); <- up
Click on the arguments and move the mouse to modify values.

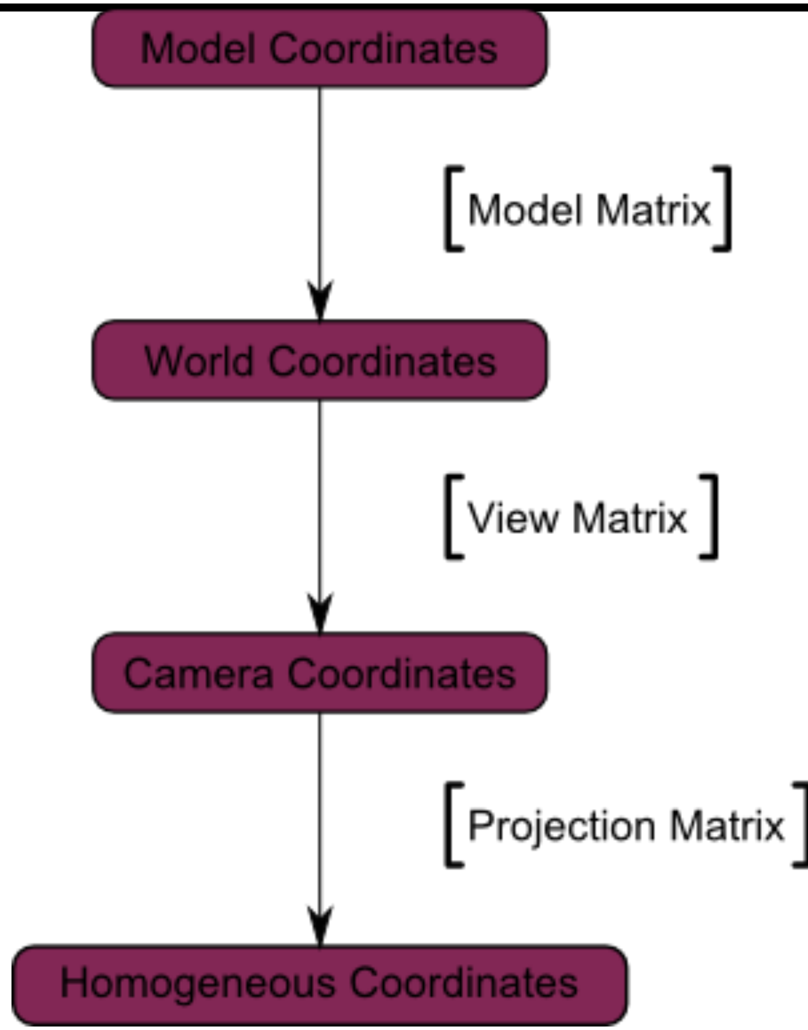
In Code

```
// Generates a really hard-to-read matrix, but a normal, standard 4x4 matrix nonetheless

glm::mat4 projectionMatrix = glm::perspective(

    FoV,        // The horizontal Field of View, in degrees : the amount of "zoom".
                // Think "camera lens". Usually between 90° (extra wide) and 30° (quite zoomed in)
    4.0f / 3.0f, // Aspect Ratio. Depends on the size of your window.
                //Notice that 4/3 == 800/600 == 1280/960, sounds familiar ?
    0.1f,       // Near clipping plane. Keep as big as possible, or you'll get precision issues.

    100.0f      // Far clipping plane. Keep as little as possible.
);
```

Effect



In Diagrams



Model Coordinates
Model Matrix
World Coordinates
View Matrix
Camera Coordinates
Projection Matrix
Homogeneous Coordinates

More Code

C++ : compute the matrix

```
glm::mat4 MVPmatrix = projection * view * model;
// Remember : inverted !
```

```
// GLSL : apply it
transformed_vertex = MVP * in_vertex;
```

Combined

Generate Matrix

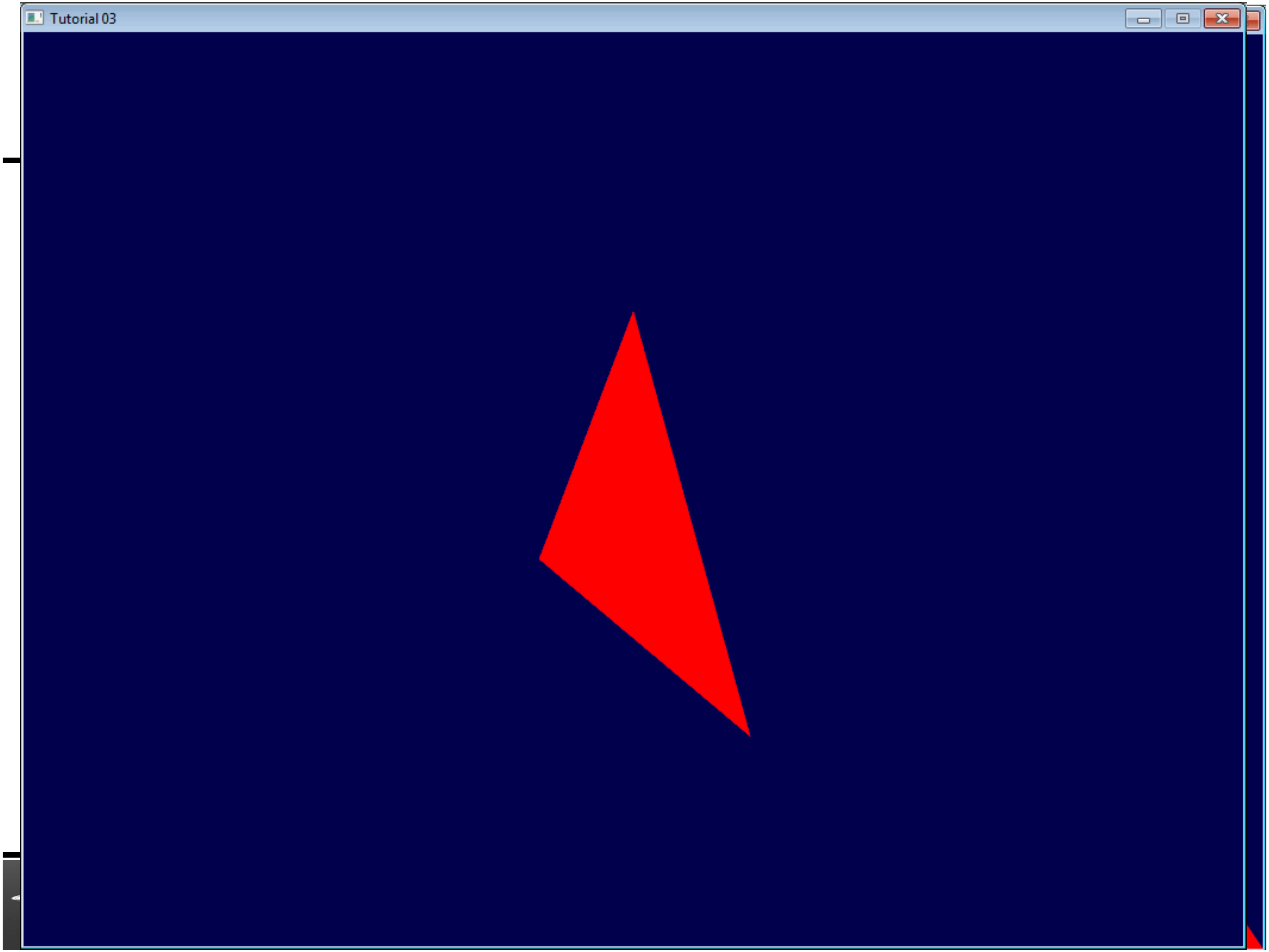
```
// Projection matrix : 45°
//Field of View, 4:3 ratio, display range : 0.1 unit <-> 100 units
glm::mat4 Projection = glm::perspective(45.0f, 4.0f / 3.0f, 0.1f, 100.0f);

// Camera matrix
glm::mat4 View       = glm::lookAt(
    glm::vec3(4,3,3), // Camera is at (4,3,3), in World Space
    glm::vec3(0,0,0), // and looks at the origin
    glm::vec3(0,1,0)  // Head is up (set to 0,-1,0 to look upside-down)
);
// Model matrix : an identity matrix (model will be at the origin)
glm::mat4 Model      = glm::mat4(1.0f);  // Changes for each model !
// Our ModelViewProjection : multiplication of our 3 matrices
glm::mat4 MVP        = Projection * View * Model;
// Remember, matrix multiplication is the other way around
```

GLSL Takes Over

```
// Get a handle for our "MVP" uniform.
// Only at initialisation time.
GLuint MatrixID = glGetUniformLocation(programID, "MVP");

// Send our transformation to the currently bound shader,
// in the "MVP" uniform
// For each model you render, since the MVP will be different
// (at least the M part)

glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
```

Use It

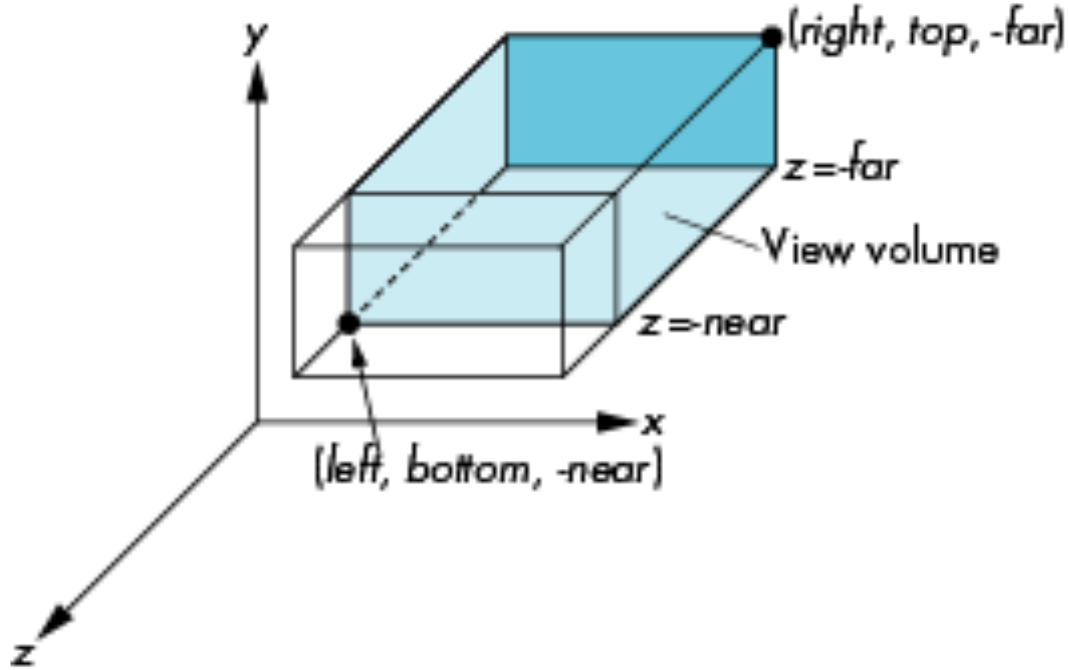
```
in vec3 vertexPosition_modelspace;
uniform mat4 MVP;

void main(){
// Output position of the vertex, in clip space : MVP * position

    vec4 v = vec4(vertexPosition_modelspace,1);

// Transform an homogeneous 4D vector, remember ?
    gl_Position = MVP * v;

}
```


Tutorial 03

Old Style

OpenGL Orthogonal Viewing

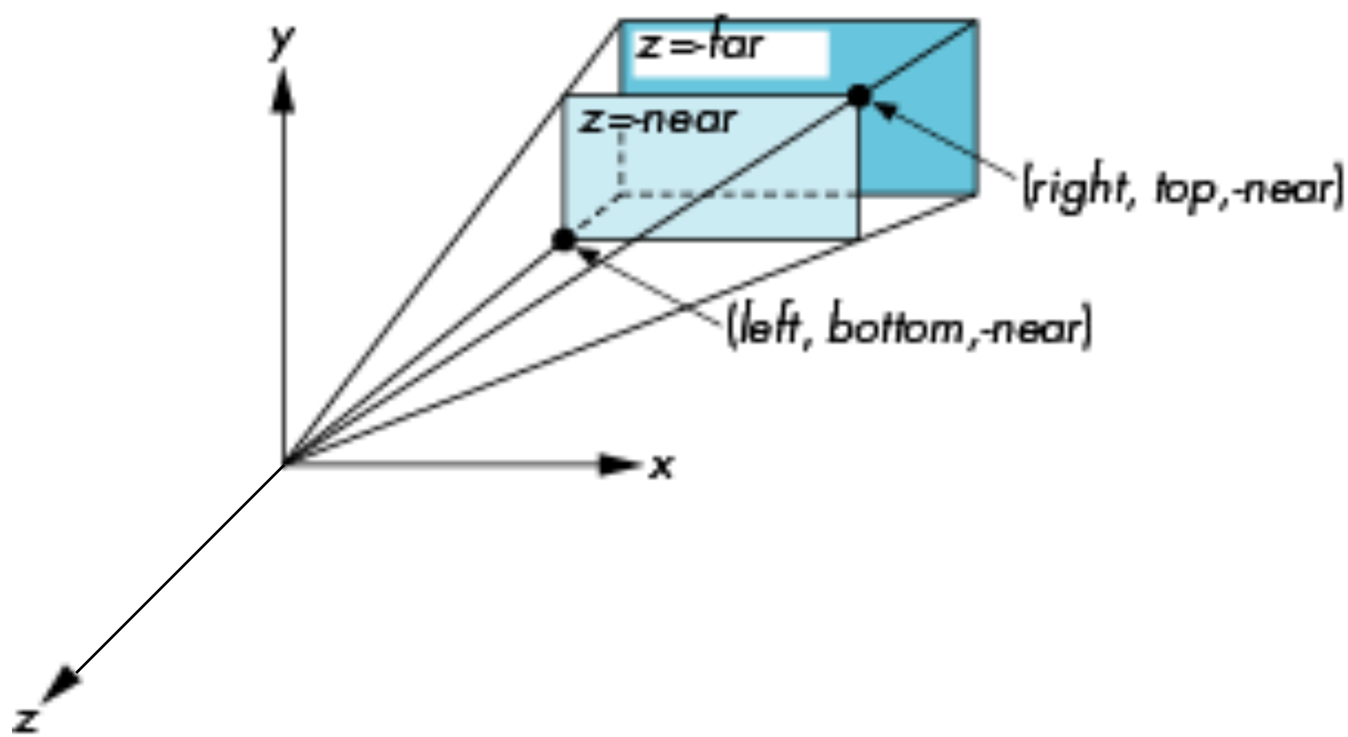
`Ortho(left,right,bottom,top,near,far)`



`near` and **`far`** measured <u>from</u> camera

OpenGL Perspective

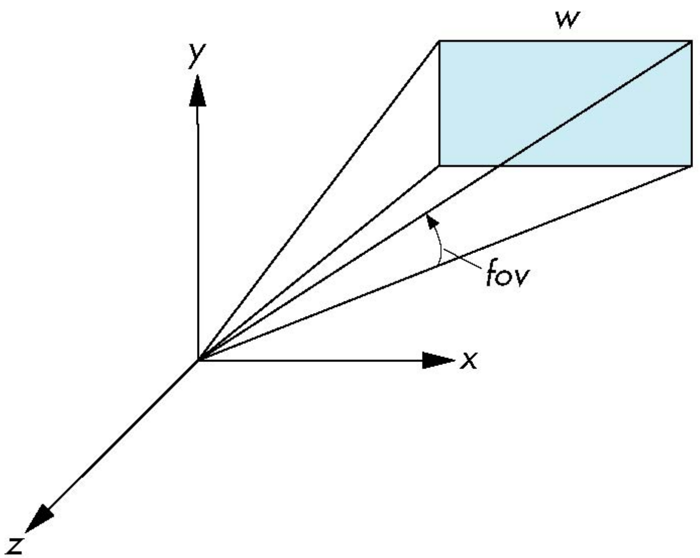
Frustum(left,right,bottom,top,near,far)



THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Using Field of View

- With Frustum it is often difficult to get the desired view
- Perpective(fovy, aspect, near, far) often provides a better interface



front plane

aspect = w/h

Old Style

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJETION);
    glLoadIdentity();
    gluPerspective(fove, aspect, near, far);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,1,0,0,0,0,1,0);
    my_display();    // your display routine
}
```

Can Still GLM

- Set up the projection matrix

```
glm::mat4 projection = glm::mat4(1.0f);
projection = glm::perspective(60.0f,1.0f,.1f,100.0f);
```

- Load the matrix to GL_PROJECTION

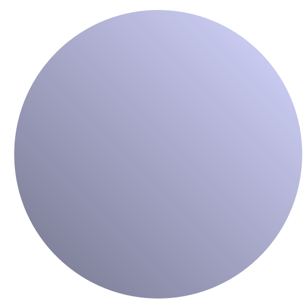
```
glMatrixMode(GL_PROJECTION);
glLoadMatrixf(&projection[0][0]);
```

Next

Why we need shading

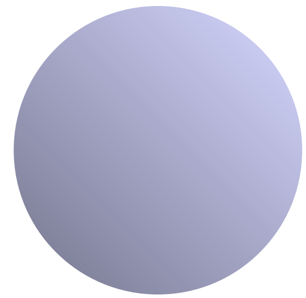
- Just attach color *glColor*

- But



Shading

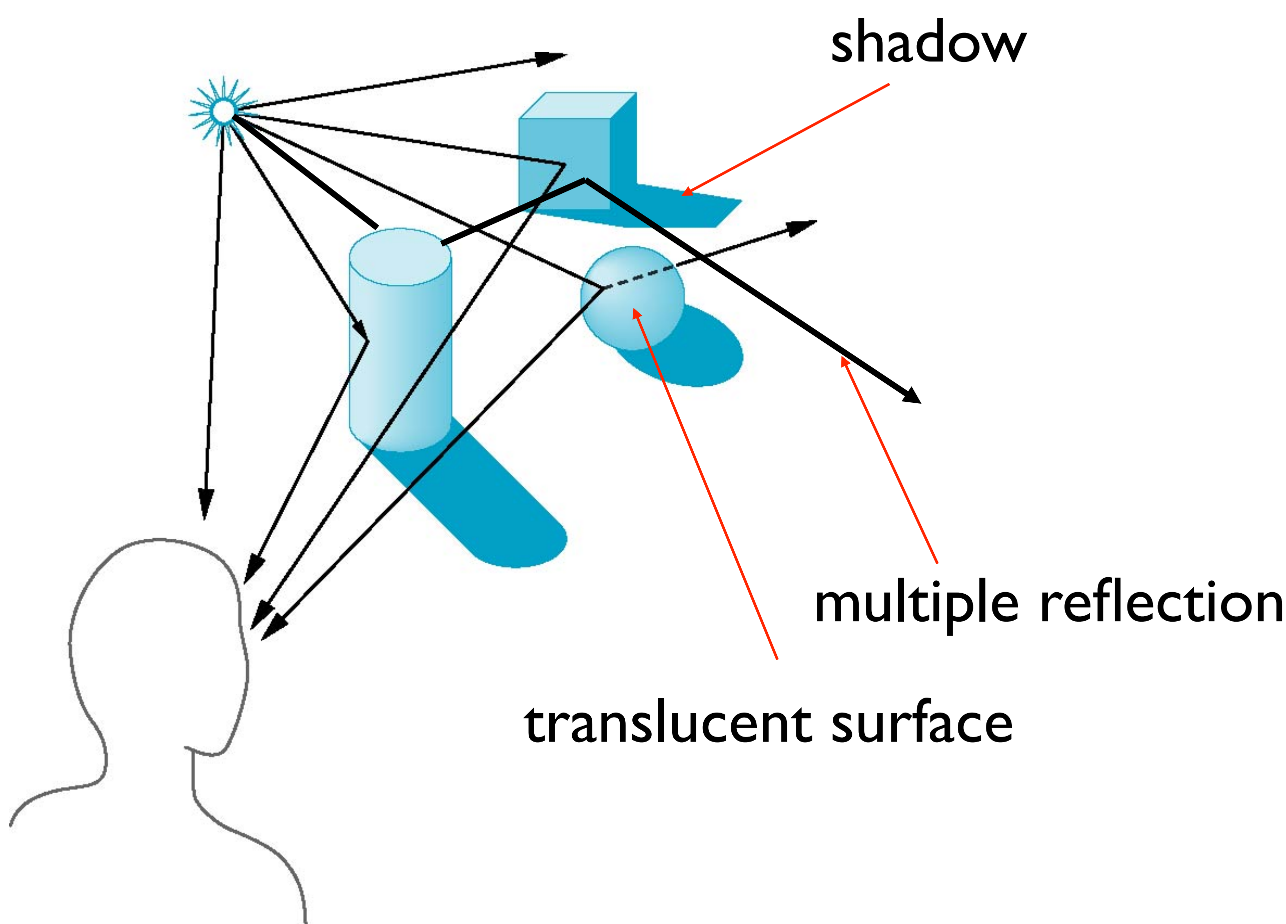
- Why does the shape ?



- Light-material interaction at points -> different color or shade

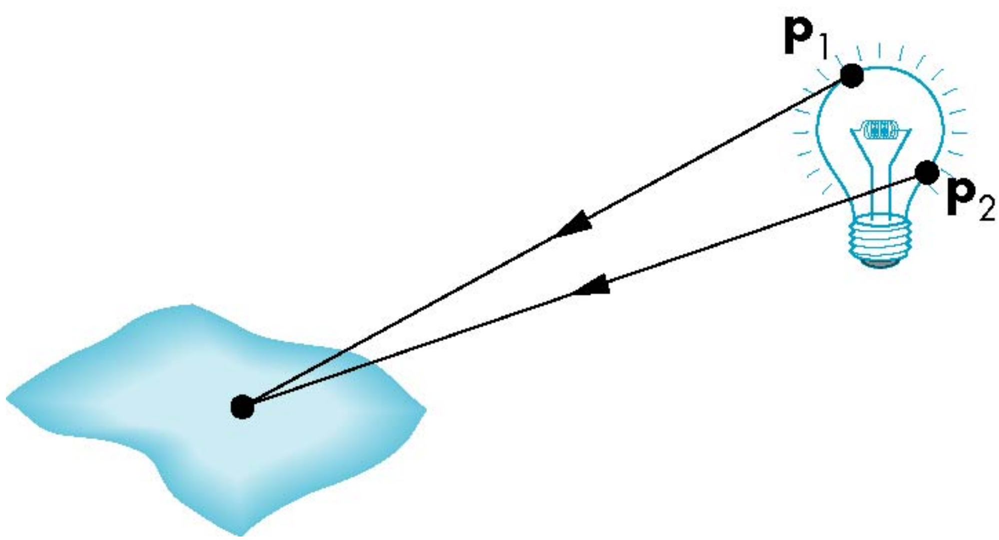
- Factor
  - Light sources
  - Material properties
  - Location of viewer
  - Surface orientation

Global Effects



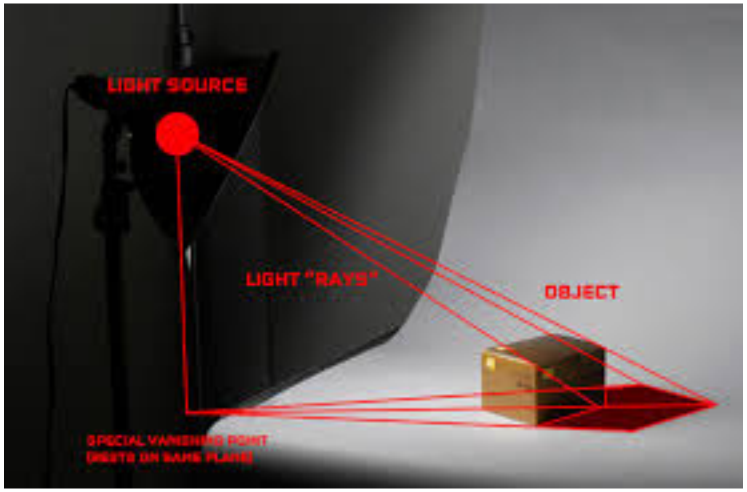
Light Sources

General Difficult !



Simple Light Sources

Point Sources



Point source

- Model with position and color
- Distant source = infinite distance away (parallel)

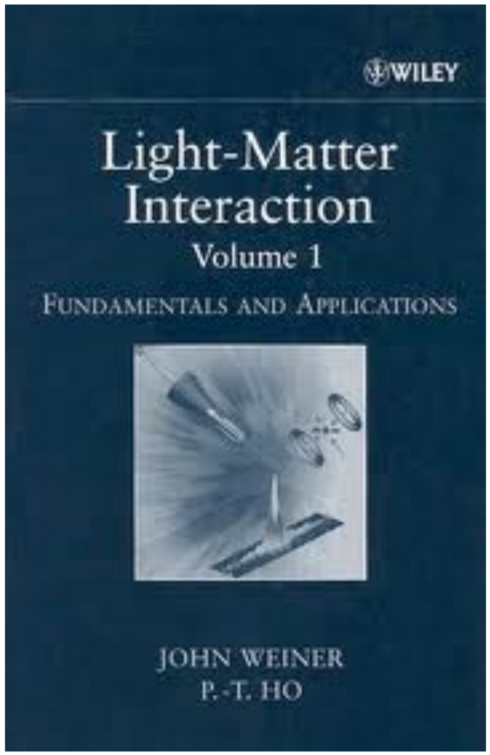
Spot Light

Spotlight
- Restrict light from ideal point source

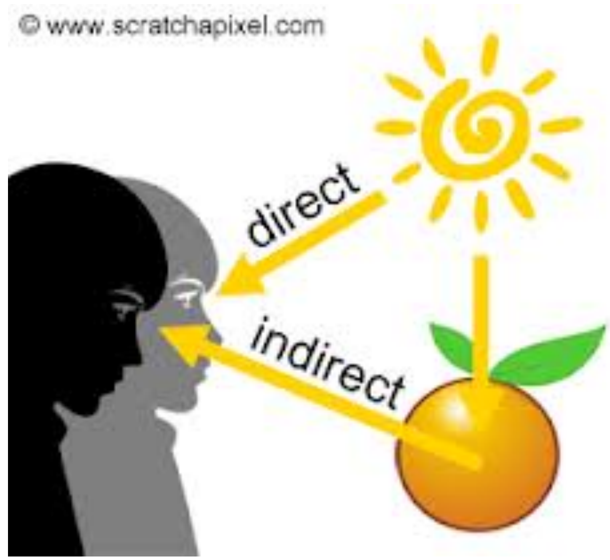
Ambient

Ambient light

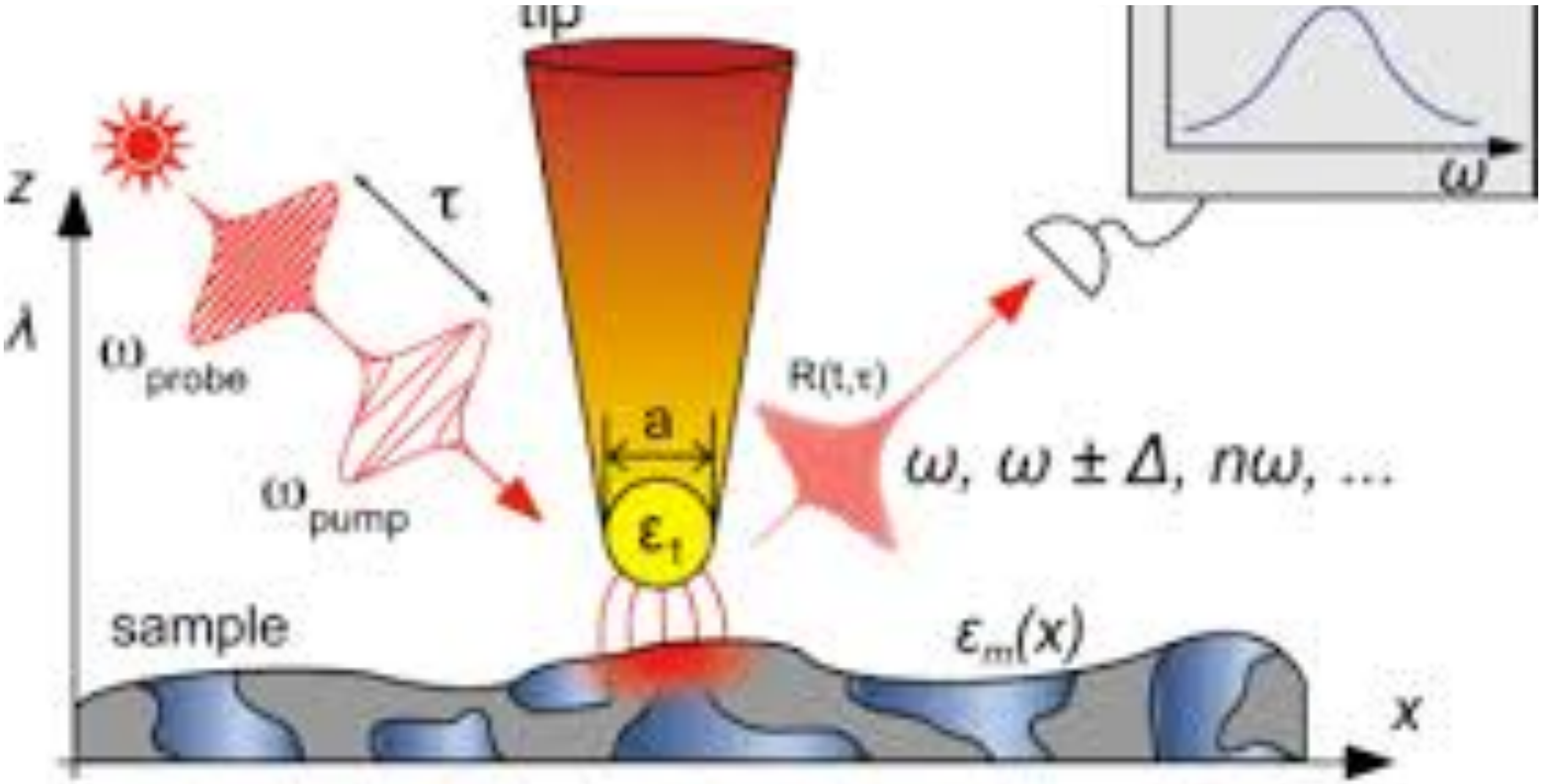
- Same amount of light in scene
- Model contribution of all sources and reflecting surfaces


WILEY
Light-Matter
Interaction
Volume 1
FUNDAMENTALS AND APPLICATIONS
JOHN WEINER
P.-T. HO

Indirect/Direct Light

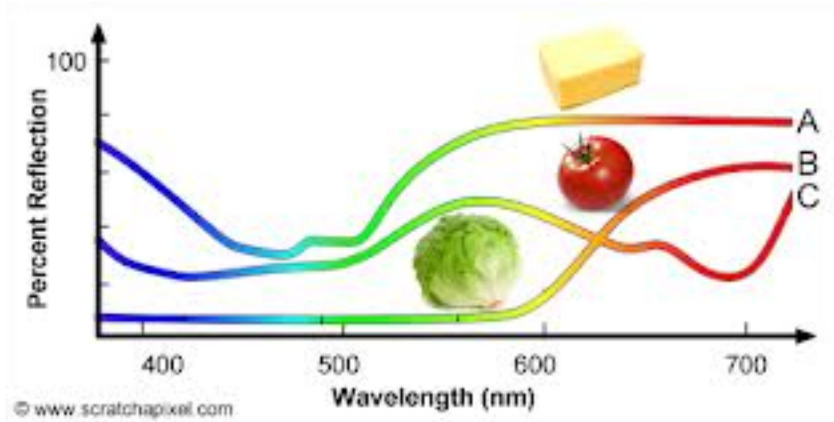


Scatter (reflect) & Absporb



Light strikes object - is partially absorbed & partially scattered (reflected)

Color !

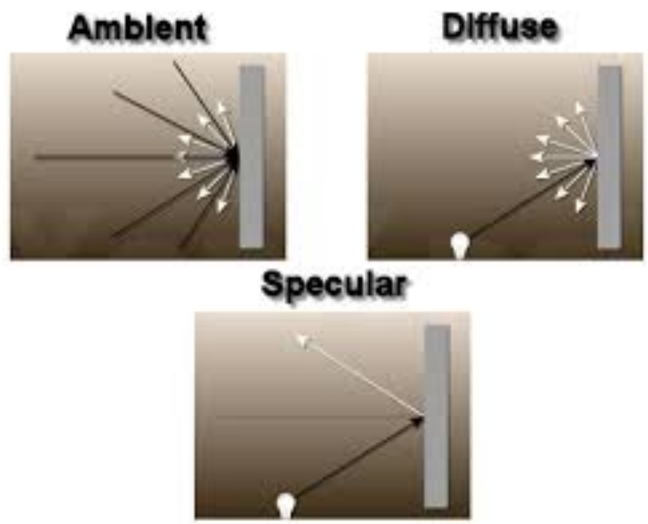


Amount reflected determines the color and brightness of the object

- Red surface appears red in white light - red component is reflected and rest is absorbed

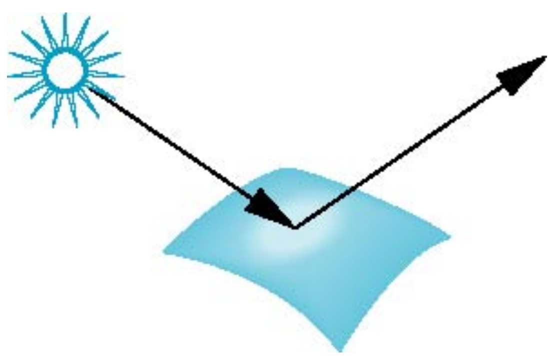
The Surface

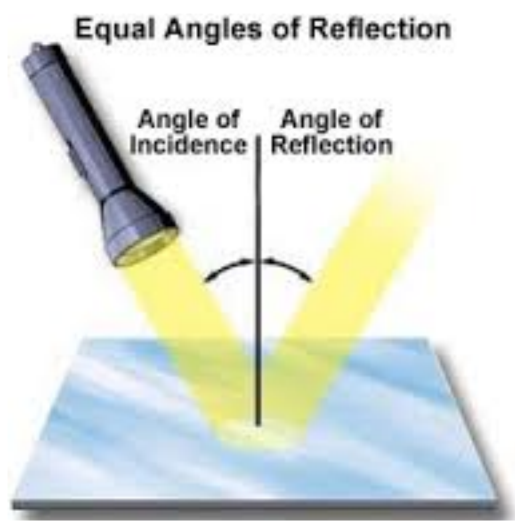
Reflected light is scattered depending on smoothness and orientation of the surface



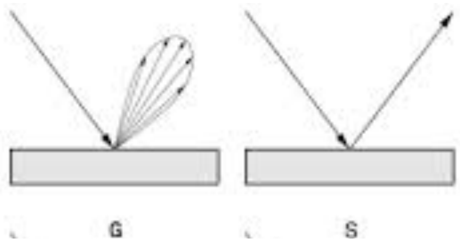
Surface Type - Smooth

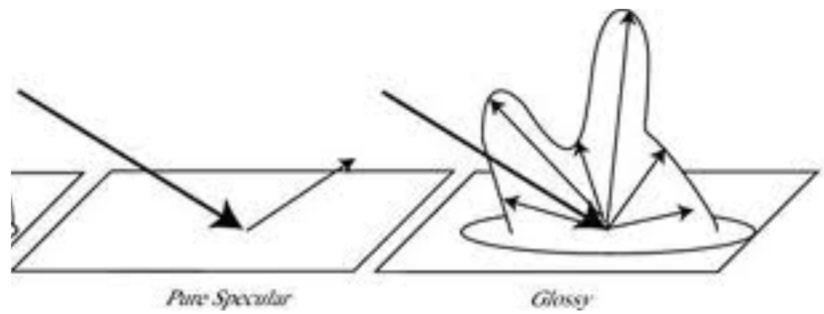
- Very Smooth - more reflected light concentrated in one direction – like a perfect mirror





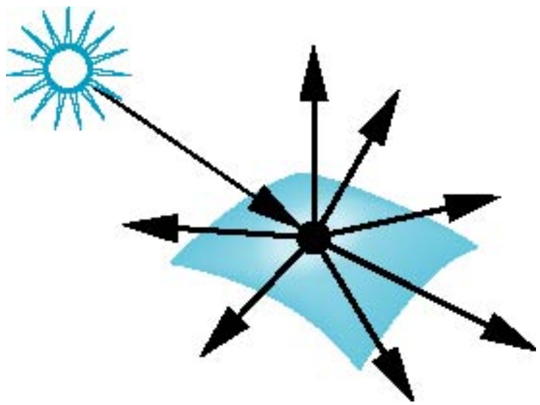
TBT - specular





Surface Type - Rough

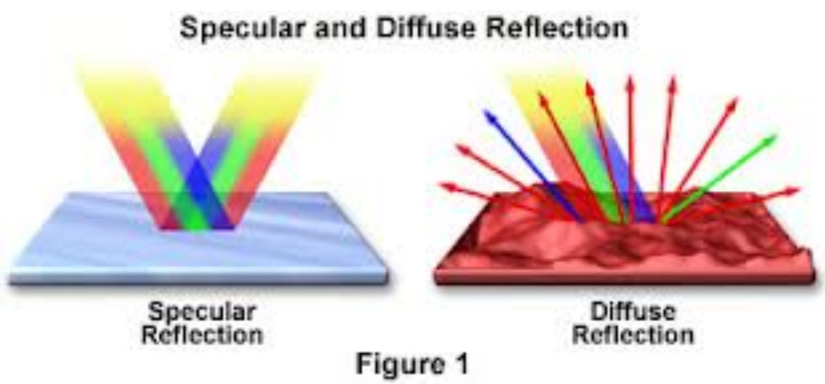
Scatters light in all directions

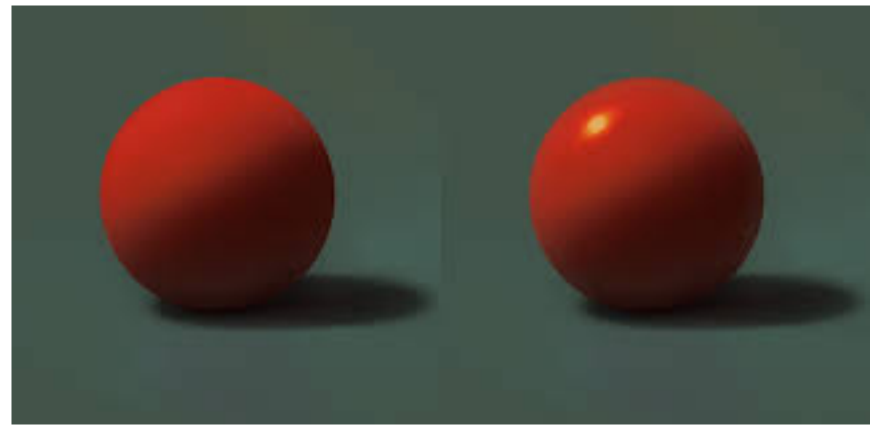


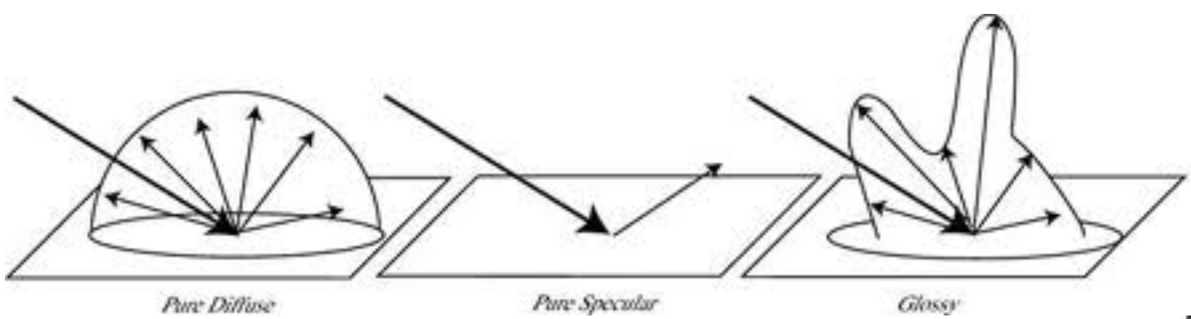
rough surface

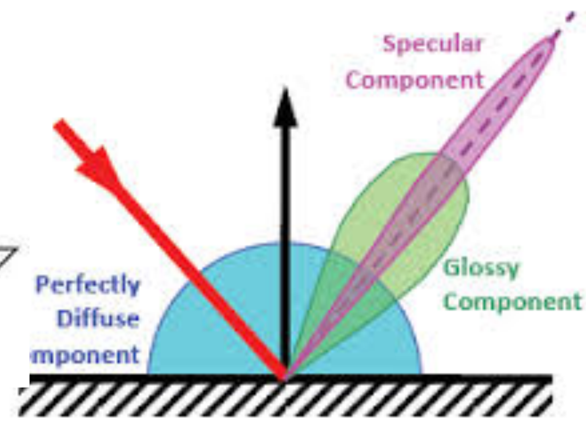
THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Smooth vs. Rough

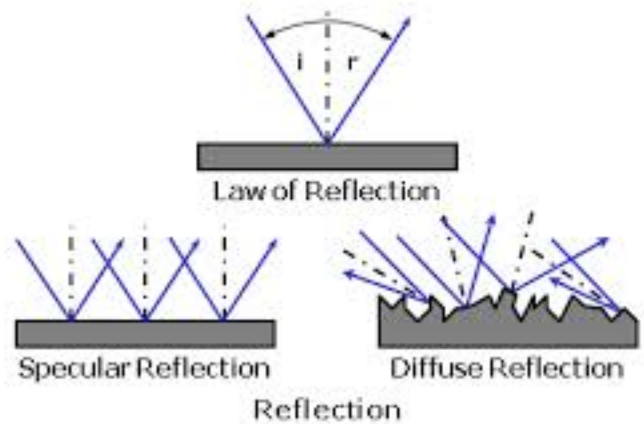








Smooth vs. Rough

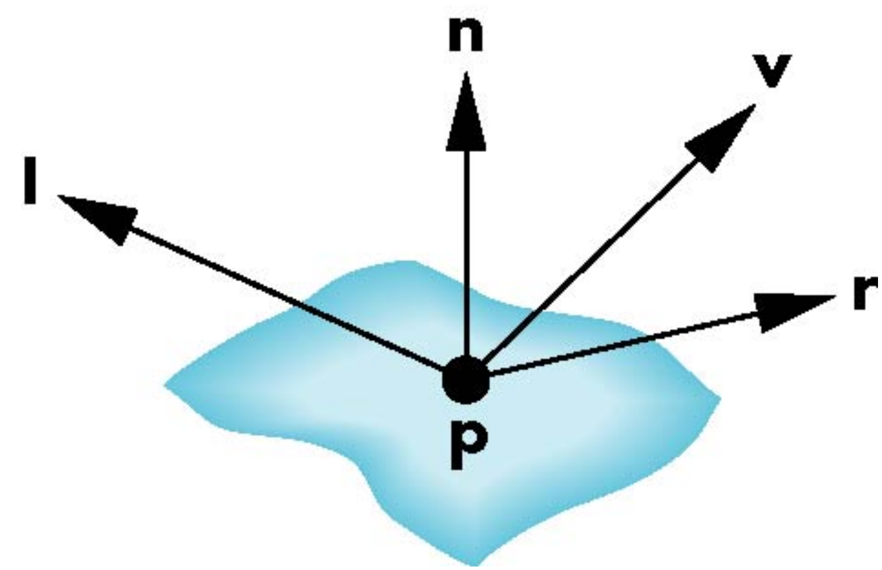


The Phong Illumination Model

Phong Model

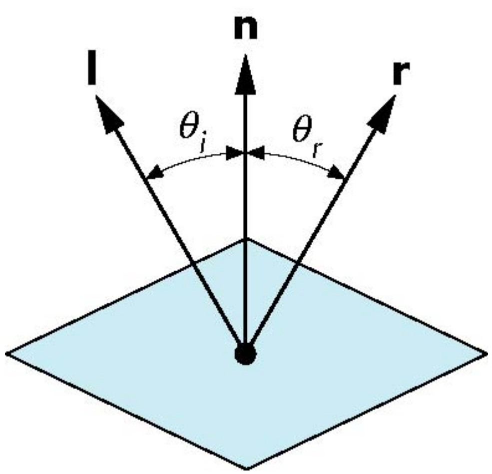
A simple local model that can be computed rapidly

- Has three components
  - Diffuse
  - Specular
  - Ambient
- Uses four vectors
  - To source
  - To viewer
  - Normal
  - Perfect reflector



Ideal Reflector

- Normal is determined by local orientation
- Angle of incidence = angle of relection
- The three vectors must be coplanar

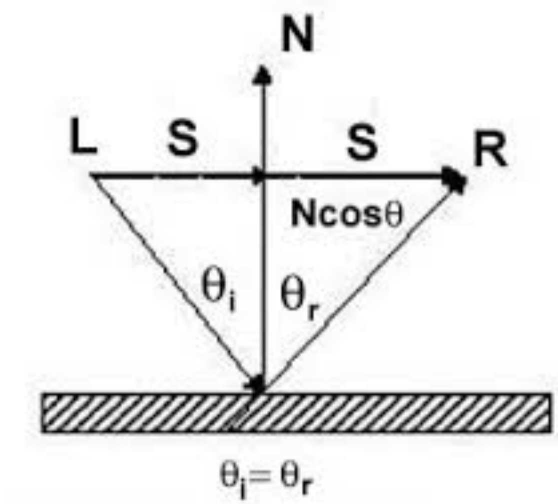


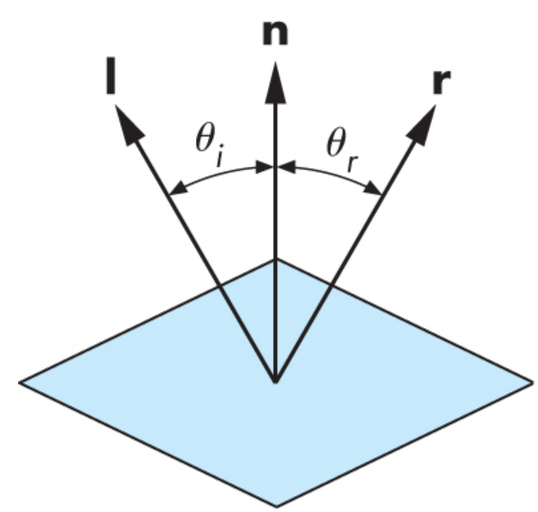
THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Computing r

Want all three to be unit length

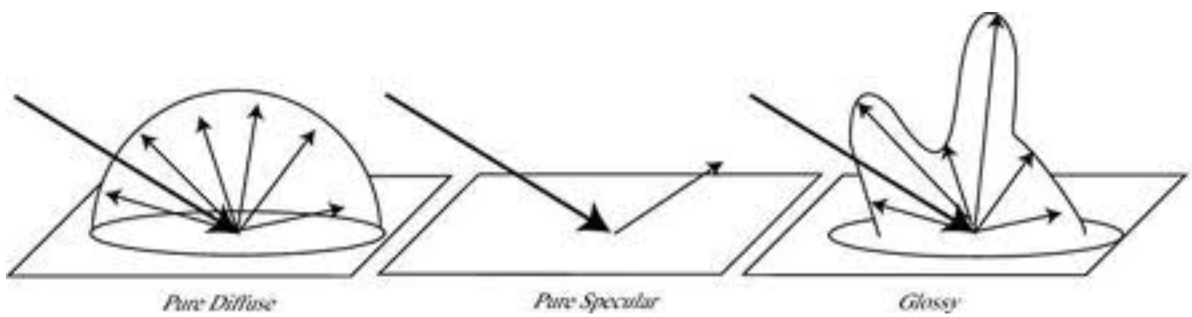
$$r = 2(l \bullet n)n - l$$





THE OHIO STATE UNIVERSITY
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

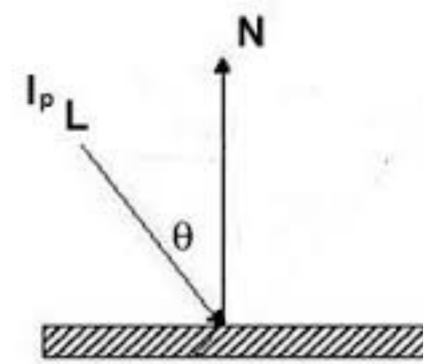
Diffuse

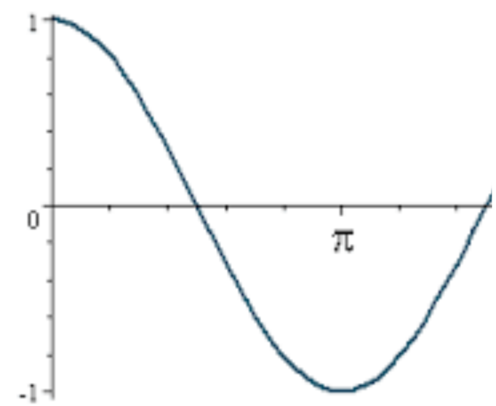


Lambertian Surface

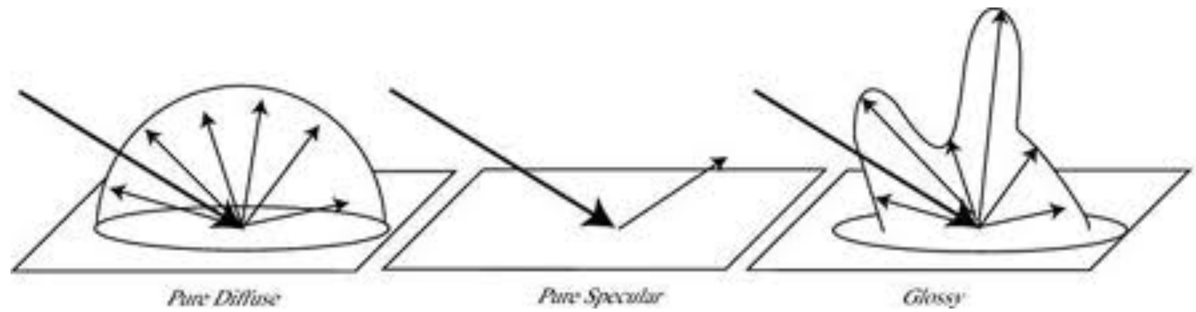
Amount reflected is proportional to vertical component of incoming light

- reflected light ~cos θ_i
- cos θ_i = $\mathbf{l} \cdot \mathbf{n}$ if vectors normalized
- Three coefficients, k_r, k_b, k_g that measure each color component is reflected





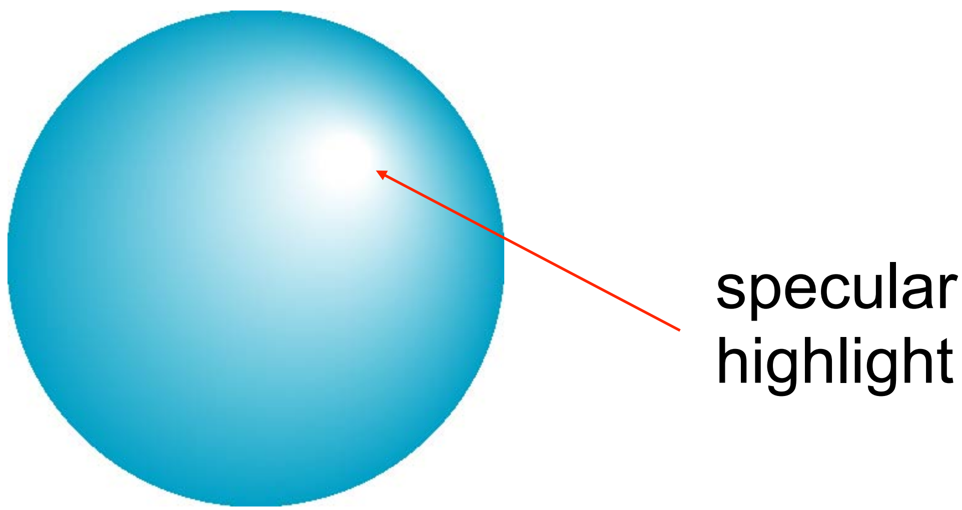
Specular or Glossy Surface



Specular Surfaces

Specular highlights due to incoming light being reflected in directions close to the direction of a perfect reflection

Not Ideal Mirror



Specular Reflections

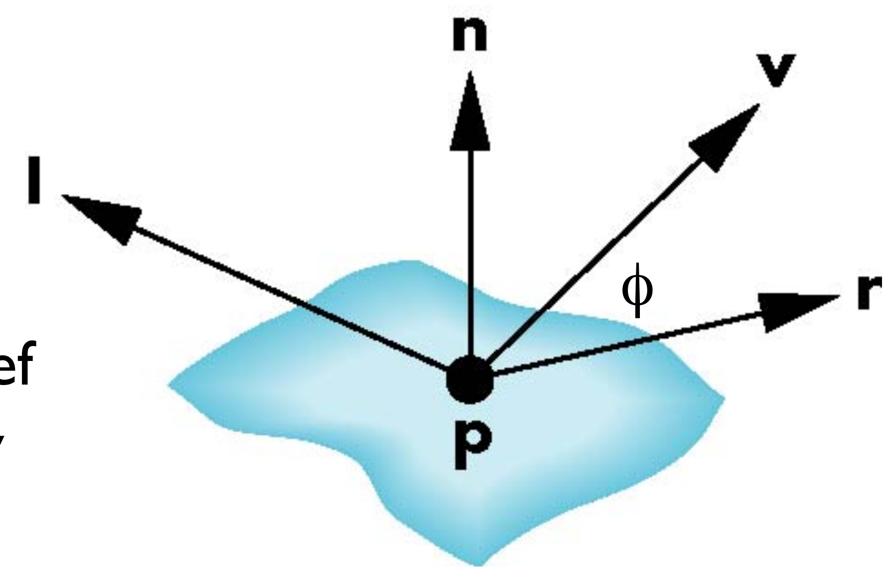
$\mathbf{I}_r \sim k_s \, \mathbf{I} \cos^{\alpha}\phi$

reflected intensity

absorption coef

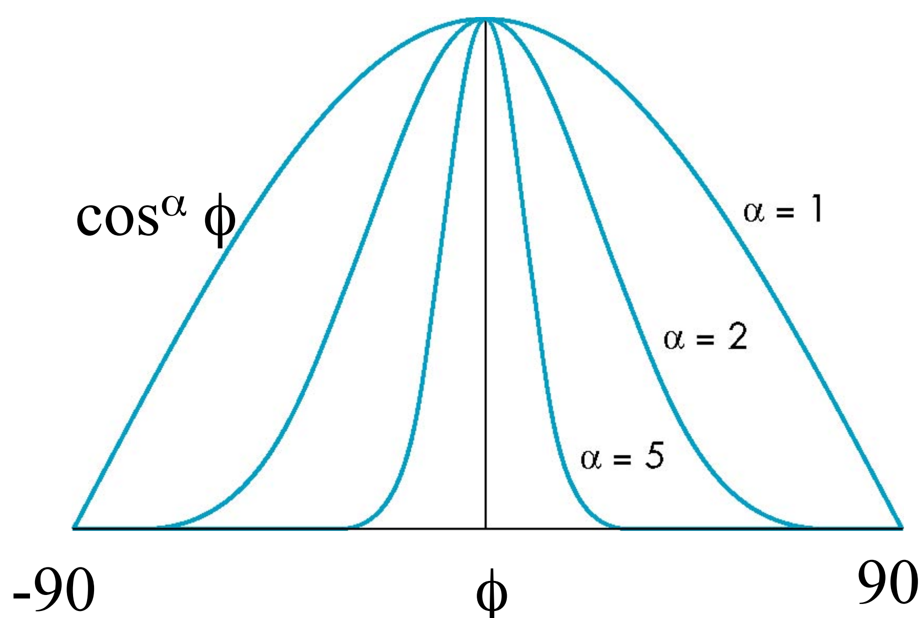
incoming intensity

shininess coef



The Shininess Coefficient

- α between
  - 100 and 200 correspond to metals
  - 5 and 10 give surface that look like plastic



THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

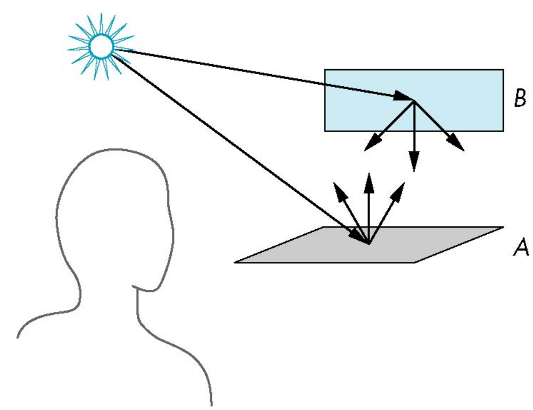
Ambient Light

- Result of multiple interactions between (large) light sources and objects in environment
- Amount and color depend on both color of light(s) and material properties of the object
- Add k_a I_a to diffuse and specular terms

reflection coef — intensity of ambient light

Distance Terms

- Light from a point source that reaches a surface is inversely proportional to the square of the distance between them
- We can add a factor of the form $1/(ad + bd + cd^2)$ to the diffuse and specular terms
- The constant and linear terms soften the effect of the point source



Light Source As

- We add results from each light source
- Each light source has separate diffuse, specular, and ambient terms to allow for maximum flexibility even though this form does not have a physical justification
- Separate red, green and blue components
- Hence, 9 coefficients for each point source
  - I_{dr}, I_{dg}, I_{db}, I_{sr}, I_{sg}, I_{sb}, I_{ar}, I_{ag}, I_{ab}

Material Properties

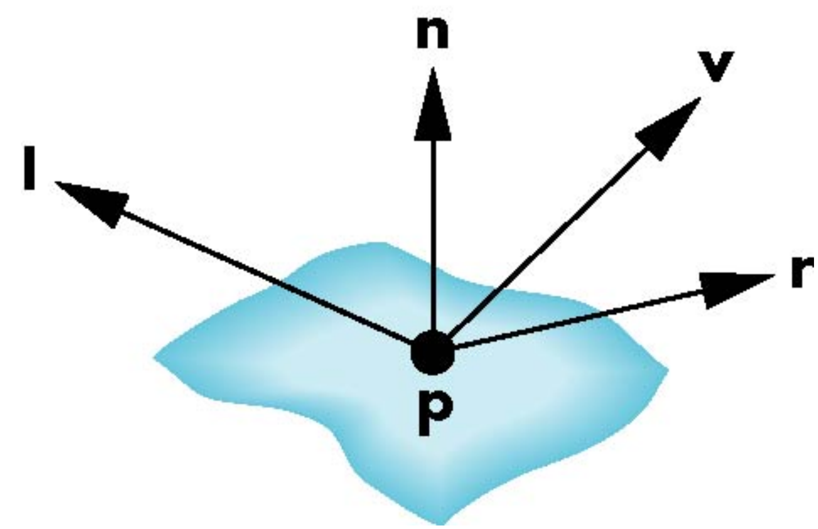
- Material properties match light source properties
  - Nine absorbtion coefficients
    - k_{dr}, k_{dg}, k_{db}, k_{sr}, k_{sg}, k_{sb}, k_{ar}, k_{ag}, k_{ab}
  - Shininess coefficient a

Adding Components

For each light source and each color component, the Phong model can be written (without the distance terms) as

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

For each color component we add contributions from all sources



Modified Phong Model

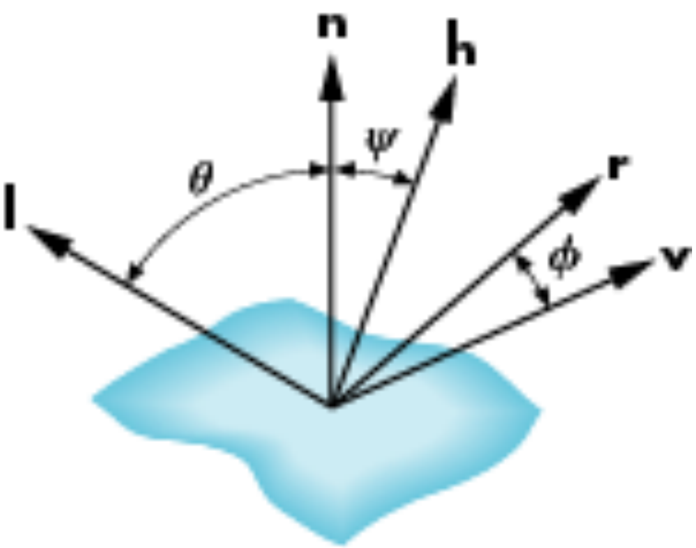
- The specular term in the Phong model is problematic because it requires the calculation of a new reflection vector and view vector for each vertex
- Blinn suggested an approximation using the halfway vector that is more efficient

More to Come.....

The Halfway Vector

- **h** is normalized vector halfway between **l** and **v**

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



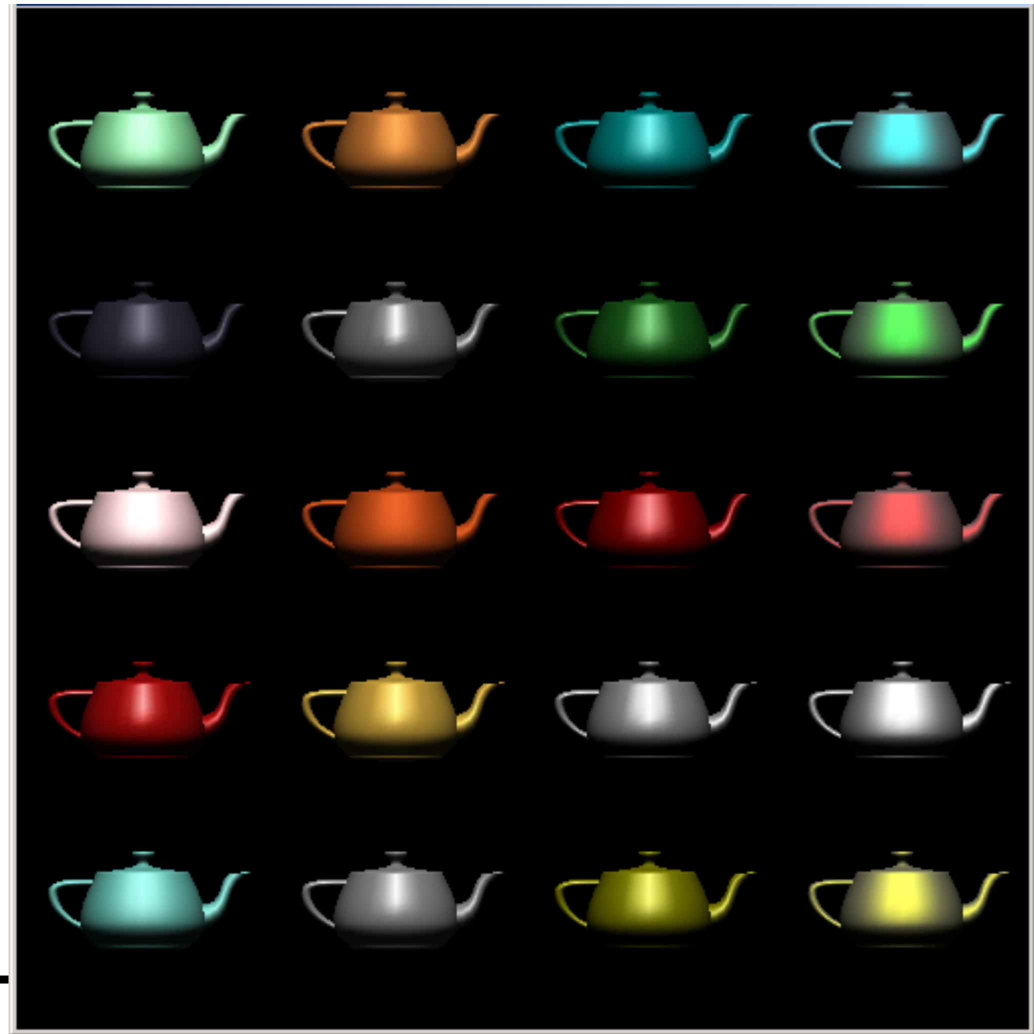
Using the halfway vector

- Replace $(\mathbf{v} \cdot \mathbf{r})^{\alpha}$ by $(\mathbf{n} \cdot \mathbf{h})^{\beta}$
- β is chosen to match shineness
- Note that halfway angle is half of angle between $\mathbf{r}$ and $\mathbf{v}$ if vectors are coplanar
- Resulting model is known as the modified Phong or Blinn lighting model
  - Specified in OpenGL standard

THE OHIO STATE UNIVERSITY DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Example

Only differences in these teapots are the parameters in the modified Phong model



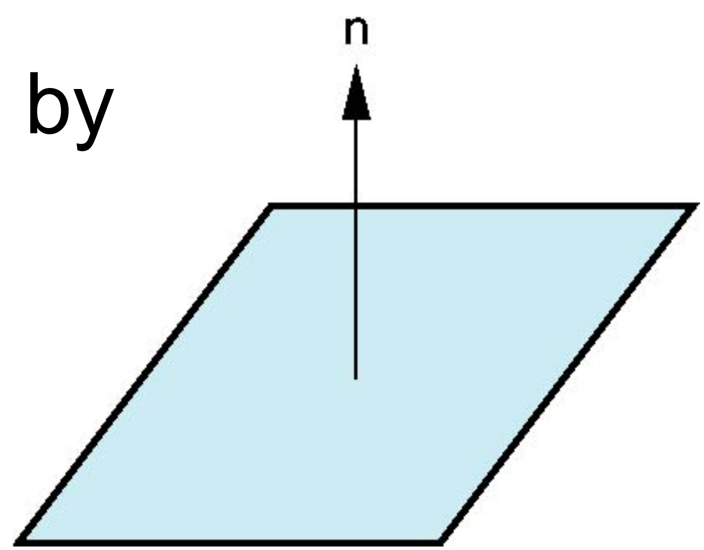
Computation of Vectors

- **l** and **v** are specified by the application
- Can computer **r** from **l** and **n**
- Problem is determining **n**
- For simple surfaces is can be determined but how we determine **n** differs depending on underlying representation of surface
- OpenGL leaves determination of normal to application
  - Exception for GLU quadrics and Bezier surfaces was deprecated

Plane Normals

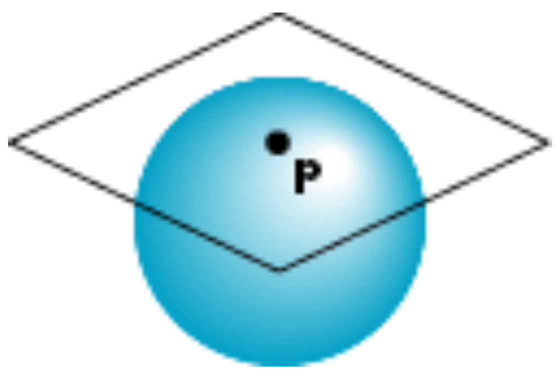
- Equation of plane: $ax+by+cz+d = 0$
- From Chapter 3 we know that plane is determined by three points p_0, p_2, p_3 or normal $\mathbf{n}$ and p_0
- Normal can be obtained by

$$\mathbf{n} = (p_2 - p_0) \times (p_1 - p_0)$$



Normal to Sphere

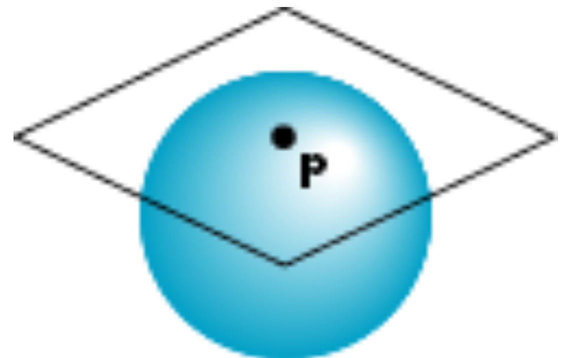
- Implicit function $f(x,y.z)=0$
- Normal given by gradient
- Sphere $f(\mathbf{p})=\mathbf{p}\cdot\mathbf{p}-1$
- $n = [\partial f/\partial x, \partial f/\partial y, \partial f/\partial z]^T=\mathbf{p}$



Parametric Form

- For sphere

$$x=x(u,v)=\cos u \sin v$$
$$y=y(u,v)=\cos u \cos v$$
$$z= z(u,v)=\sin u$$



- Tangent plane determined by vectors

$$\partial \mathbf{p}/\partial u = [\partial x/\partial u, \partial y/\partial u, \partial z/\partial u]T$$
$$\partial \mathbf{p}/\partial v = [\partial x/\partial v, \partial y/\partial v, \partial z/\partial v]T$$

- Normal given by cross product

$$\mathbf{n} = \partial \mathbf{p}/\partial u \times \partial \mathbf{p}/\partial v$$

General Case

- We can compute parametric normals for other simple cases
  - Quadrics
  - Parameteric polynomial surfaces
    - Bezier surface patches (Chapter 10)

Shading in OpenGL

Ed Angel
Professor Emeritus of Computer Science
University of New Mexico

Objectives

- Introduce the OpenGL shading methods
  - per vertex vs per fragment shading
  - Where to carry out
- Discuss polygonal shading
  - Flat
  - Smooth
  - Gouraud

OpenGL shading

- Need
  - Normals
  - material properties
  - Lights
- State-based shading functions have been deprecated (glNormal, glMaterial, glLight)
- Get computer in application or send attributes to shaders

Normalization

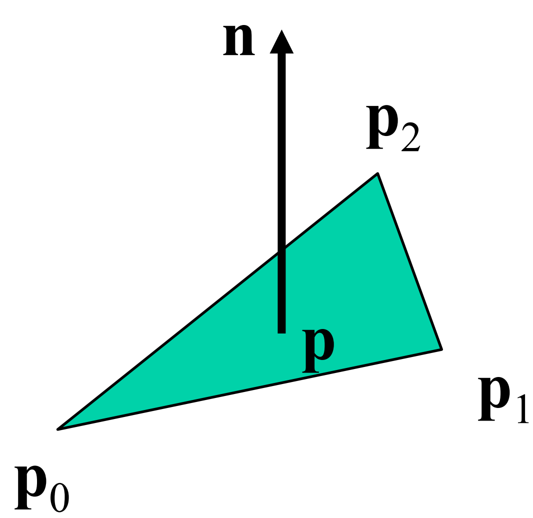
- Cosine terms in lighting calculations can be computed using dot product
- Unit length vectors simplify calculation
- Usually we want to set the magnitudes to have unit length but
  – Length can be affected by transformations
  – Note that scaling does not preserved length
- GLSL has a normalization function

Normal for Triangle

plane $\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0$

$\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)$

normalize $\mathbf{n} \leftarrow \mathbf{n} / |\mathbf{n}|$



Note that right-hand rule determines outward face

Specifying a Point Light Source

- For each light source, we can set an RGBA for the diffuse, specular, and ambient components, and for the position

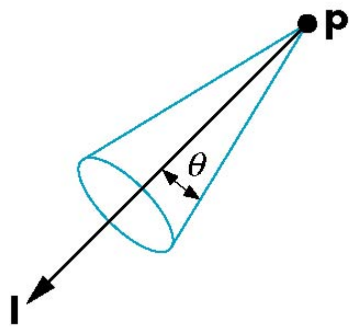
```
vec4 diffuse0 =vec4(1.0, 0.0, 0.0, 1.0);
vec4 ambient0 = vec4(1.0, 0.0, 0.0, 1.0);
vec4 specular0 = vec4(1.0, 0.0, 0.0, 1.0);
vec4 light0_pos =vec4(1.0, 2.0, 3,0, 1.0);
```

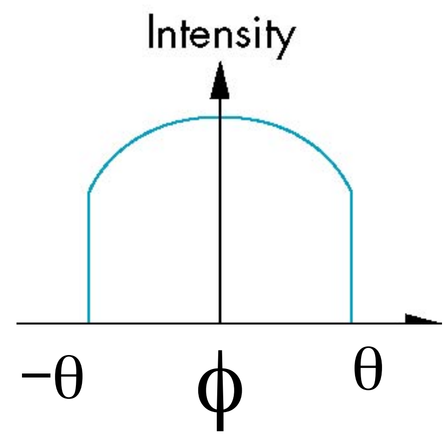
Distance and Direction

- The source colors are specified in RGBA
- The position is given in homogeneous coordinates
  - If w =1.0, we are specifying a finite location
  - If w =0.0, we are specifying a parallel source with the given direction vector
- The coefficients in distance terms are usually quadratic (1/(a+b*d+c*d*d)) where d is the distance from the point being rendered to the light source

Spotlights

- Derive from point source
  - Direction
  - Cutoff
  - Attenuation Proportional to $\cos^{\alpha}\phi$





Global Ambient Light

- Ambient light depends on color of light sources
  - A red light in a white room will cause a red ambient term that disappears when the light is turned off
- A global ambient term that is often helpful for testing

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
THE OHIO STATE UNIVERSITY

Moving Light Sources

- Light sources are geometric objects whose positions or directions are affected by the model-view matrix
- Depending on where we place the position (direction) setting function, we can
  - Move the light source(s) with the object(s)
  - Fix the object(s) and move the light source(s)
  - Fix the light source(s) and move the object(s)
  - Move the light source(s) and object(s) independently

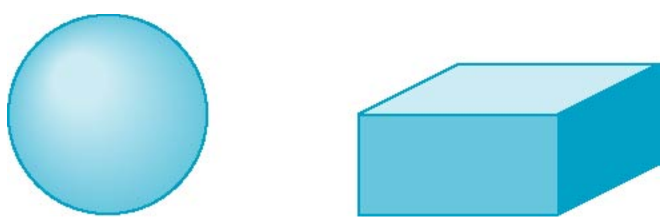
Material Properties

- Material properties should match the terms in the light model
- Reflectivities
- w component gives opacity

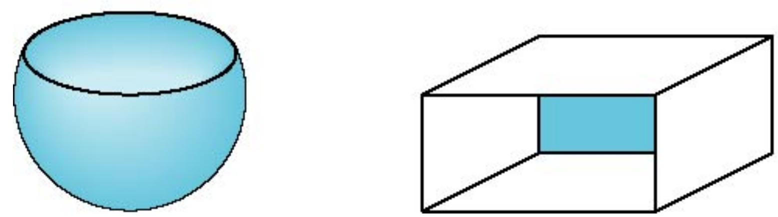
```
vec4 ambient = vec4(0.2, 0.2, 0.2, 1.0);
vec4 diffuse = vec4(1.0, 0.8, 0.0, 1.0);
vec4 specular = vec4(1.0, 1.0, 1.0, 1.0);
GLfloat shine = 100.0
```

Front and Back Faces

- Every face has a front and back
- For many objects, we never see the back face so we don't care how or if it's rendered
- If it matters, we can handle in shader



back faces not visible



back faces visible

Emissive Term

- We can simulate a light source in OpenGL by giving a material an emissive component
- This component is unaffected by any sources or transformations

Transparency

- Material properties are specified as RGBA values
- The A value can be used to make the surface translucent
- The default is that all surfaces are opaque regardless of A
- Later we will enable blending and use this feature

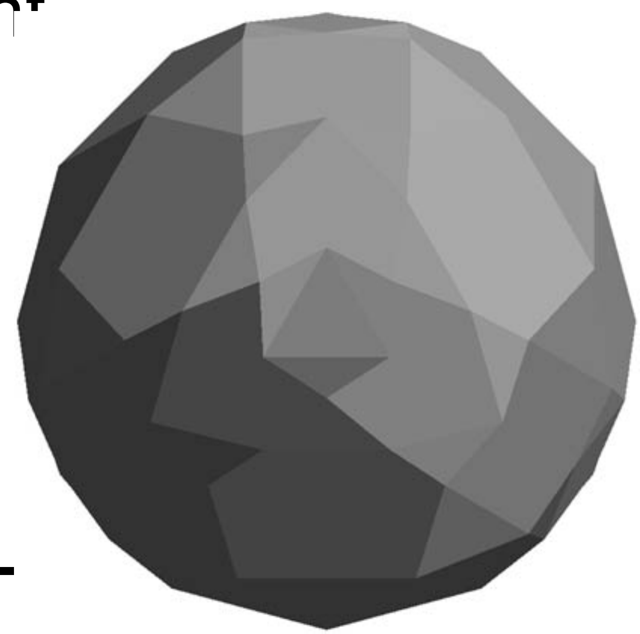
Polygonal Shading

- In per vertex shading, shading calculations are done for each vertex
  - Vertex colors become vertex shades and can be sent to the vertex shader as a vertex attribute
  - Alternately, we can send the parameters to the vertex shader and have it compute the shade
- By default, vertex shades are interpolated across an object if passed to the fragment shader as a varying variable (smooth shading)
- We can also use uniform variables to shade

Polygon Normals

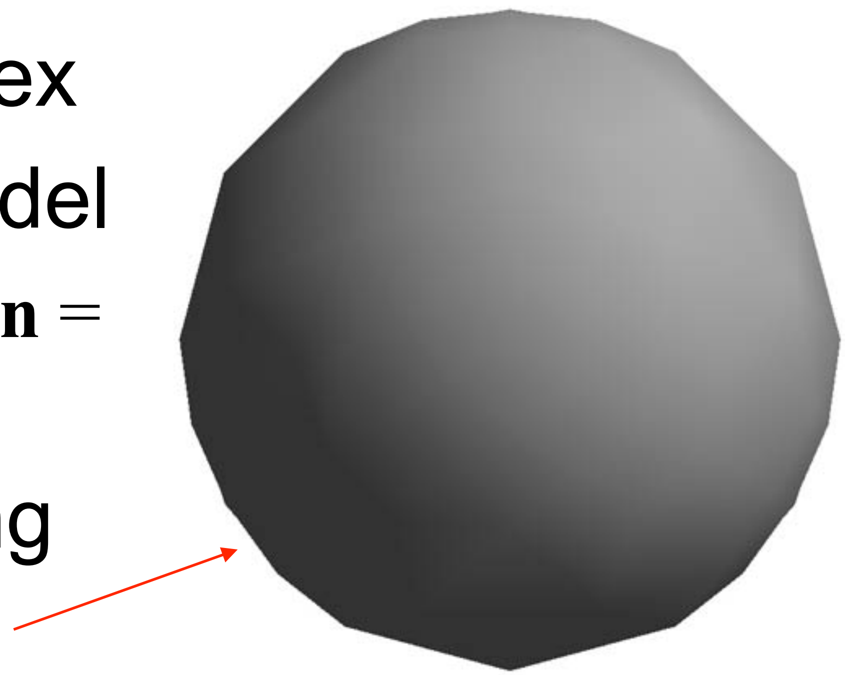
- Triangles have a single normal
  - Shades at the vertices as computed by the Phong model can be almost same
  - Identical for a distant viewer (default) or if there is no specular component
- Consider model of sphere
- Want different normals at

each vertex even though this concept is not quite correct mathematically



Smooth Shading

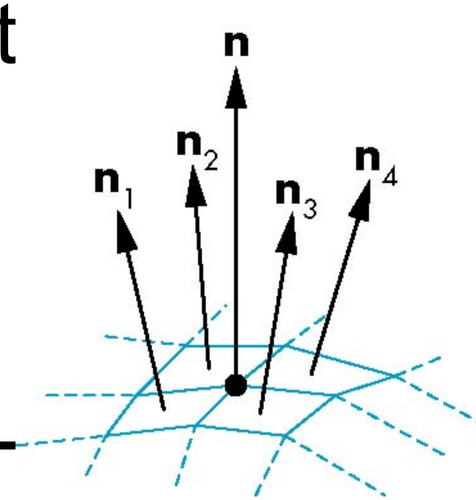
- We can set a new normal at each vertex
- Easy for sphere model
  - If centered at origin $\mathbf{n} = \mathbf{p}$
- Now smooth shading works
- Note *silhouette edge*



Mesh Shading

- The previous example is not general because we knew the normal at each vertex analytically
- For polygonal models, Gouraud proposed we use the average of the normals around a mesh vert

$$\mathbf{n} = (\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4) / |\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4|$$



Gouraud and Phong Shading

- Gouraud Shading
  - Find average normal at each vertex (vertex normals)
  - Apply modified Phong model at each vertex
  - Interpolate vertex shades across each polygon
- Phong shading
  - Find vertex normals
  - Interpolate vertex normals across edges
  - Interpolate edge normals across polygon
  - Apply modified Phong model at each fragment

Comparison

- If the polygon mesh approximates surfaces with a high curvatures, Phong shading may look smooth while Gouraud shading may show edges
- Phong shading requires much more work than Gouraud shading
  – Until recently not available in real time systems
  – Now can be done using fragment shaders
- Both need data structures to represent meshes so we can obtain vertex normals

Vertex Lighting Shaders I

```
// vertex shader
in vec4 vPosition;
in vec3 vNormal;
out vec4 color;  //vertex shade

// light and material properties
uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform vec4 LightPosition;
uniform float Shininess;
```

Vertex Lighting Shaders II

```
void main()
{
   // Transform vertex  position into eye coordinates
   vec3 pos = (ModelView * vPosition).xyz;

   vec3 L = normalize( LightPosition.xyz - pos );
   vec3 E = normalize( -pos );
   vec3 H = normalize( L + E );

   // Transform vertex normal into eye coordinates
   vec3 N = normalize( ModelView*vec4(vNormal, 0.0) ).xyz;
```

Vertex Lighting Shaders III

```
// Compute terms in the illumination equation
    vec4 ambient = AmbientProduct;

    float Kd = max( dot(L, N), 0.0 );
    vec4  diffuse = Kd*DiffuseProduct;
    float Ks = pow( max(dot(N, H), 0.0), Shininess );
    vec4  specular = Ks * SpecularProduct;
    if( dot(L, N) < 0.0 )  specular = vec4(0.0, 0.0, 0.0, 1.0);
    gl_Position = Projection * ModelView * vPosition;

    color = ambient + diffuse + specular;
    color.a = 1.0;
}
```

Vertex Lighting Shaders IV

```
// fragment shader

in vec4 color;

void main()
{
    gl_FragColor = color;
}
```

Fragment Lighting Shaders I

```
// vertex shader
in vec4 vPosition;
in vec3 vNormal;

// output values that will be interpolatated per-fragment
out vec3 fN;
out vec3 fE;
out vec3 fL;

uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform mat4 Projection;
```

Fragment Lighting Shaders II

```
void main()
{
    fN = vNormal;
    fE = vPosition.xyz;
    fL = LightPosition.xyz;

    if( LightPosition.w != 0.0 ) {
            fL = LightPosition.xyz - vPosition.xyz;
    }

    gl_Position = Projection*ModelView*vPosition;
}
```

Fragment Lighting Shaders III

```
// fragment shader

// per-fragment interpolated values from the vertex shader
in vec3 fN;
in vec3 fL;
in vec3 fE;

uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform float Shininess;
```

Fragment Lighting Shaders IV

```
void main()
{
   // Normalize the input lighting vectors

  vec3 N = normalize(fN);
   vec3 E = normalize(fE);
   vec3 L = normalize(fL);

   vec3 H = normalize( L + E );
   vec4 ambient = AmbientProduct;
```

Fragment Lighting Shaders V

```
float Kd = max(dot(L, N), 0.0);
    vec4 diffuse = Kd*DiffuseProduct;

    float Ks = pow(max(dot(N, H), 0.0), Shininess);
    vec4 specular = Ks*SpecularProduct;

    // discard the specular highlight if the light's behind the vertex
    if( dot(L, N) < 0.0 )
        specular = vec4(0.0, 0.0, 0.0, 1.0);

    gl_FragColor = ambient + diffuse + specular;
    gl_FragColor.a = 1.0;
}
```