

## UNIT IV – RENDERING

Introduction to shading models – Flat and smooth shading – Adding texture to faces – Adding shadows of objects – Building a camera in a program – Creating shaded objects – Rendering texture – Drawing shadows.

### Topic1:

#### Introduction to Shading Models

- 1) The mechanism of light reflection from an actual surface is very complicated it depends on many factors.
- 2) Some of these factors are geometric and others are related to the characteristics of the surface.
- 3) A shading model dictates how light is scattered or reflected from a surface.
- 4) Incident light interacts with the surface in three different ways:  
Some is absorbed by the surface and is converted to heat.  
Some is reflected from the surface  
Some is transmitted into the interior of the object
- 5) If all incident light is absorbed the object appears black and is known as a **black body**. If all of the incident light is transmitted the object is visible only through the effects of reflection.
- 6) The shading models described here focuses on achromatic light. **Achromatic light** has brightness and no color, it is a shade of gray so it is described by a single value its intensity.
- 7) A shading model uses two types of light source to illuminate the objects in a scene : **point light sources** and **ambient light**.
- 8) Some amount of the reflected light travels in the right direction to reach the eye causing the object to be seen. The amount of light that reaches the eye depends on the orientation of the surface, light and the observer.

There are two different types of reflection of incident light

**Diffuse scattering** occurs when some of the incident light slightly penetrates the surface and is re-radiated uniformly in all directions.

- **Specular reflections** are more mirrorlike and highly directional. Incident light is directly reflected from its outer surface. This makes the surface looks shinny

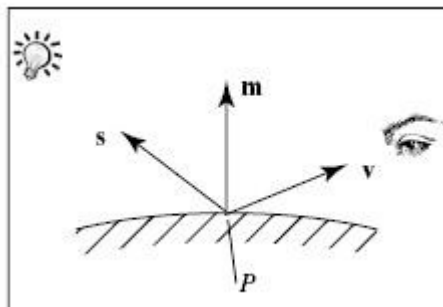
The total light reflected from the surface in a certain direction is the sum of the diffuse component and the specular component. For each surface point of interest we compute the size of each component that reaches the eye.

#### Computing Diffuse and Specular Components Geometric

##### Ingredients For Finding Reflected Light

three principal vectors (  $s$ ,  $m$  and  $v$ ) required to find the amount of light that reaches the eye from a point  $P$ .

##### Important directions in computing the reflected light



1. The normal vector ,  $m$  , to the surface at  $P$ .
2. The vector  $v$  from  $P$  to the viewer's eye.
3. The vector  $s$  from  $P$  to the light source.

#### How to Compute the Diffuse Component

Some fraction of the re-radiated part reaches the eye, with an intensity denoted by  $I_d$ .

diffuse scattering is that it is independent of the direction from the point  $P$ , to the location of the viewer's eye. This is called **omnidirectional scattering** , because scattering is uniform in all directions.

Therefore  $I_d$  is independent of the angle between  $m$  and  $v$ .

**The brightness depends on the area of the face that it sees**

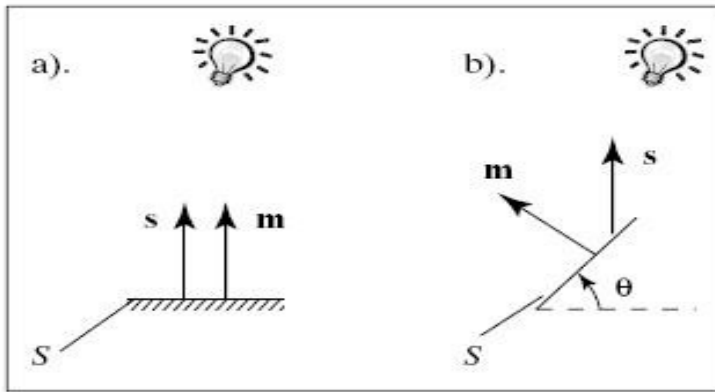


Fig (a) shows the cross section of a point source illuminating a face  $S$  when  $m$  is aligned with  $s$ .

Fig (b) the face is turned partially away from the light source through angle  $\theta$ . The area subtended is now only  $\cos(\theta)$ , so that the brightness of  $S$  is reduced by this same factor. This relationship between the brightness and surface orientation is called **Lambert's law**.

$\cos(\theta)$  is the dot product between the normalized versions of  $s$  and  $m$ . Therefore the strength of the diffuse component:

$$I_d = I_s \rho_d \max \left( \frac{s \cdot m}{|s||m|}, 0 \right)$$

$I_s$  is the intensity of the light source and  $\rho_d$  is the diffuse reflection coefficient.

If the facet is aimed away from the eye this dot product is negative so we need to evaluate  $I_d$  to 0.

$$I_d = I_s \rho_d \max \left( \frac{s \cdot m}{|s||m|}, 0 \right)$$

The reflection coefficient  $\rho_d$  depends on the wavelength of the incident light, the angle  $\theta$  and various physical properties of the surface.

### Specular Reflection

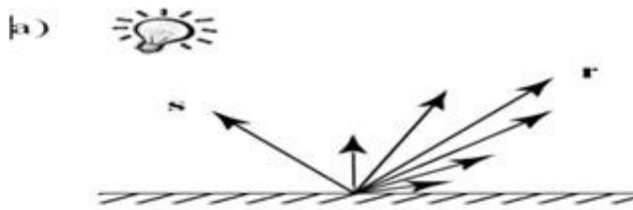
Real objects do not scatter light uniformly in all directions and so a specular component is added to the shading model.

Specular reflection causes highlights which can add reality to a picture when objects are shiny.

The behavior of specular light can be explained with Phong model.

### Phong Model

The Phong model is good when the object is made of shiny plastic or glass. The Phong model is less successful with objects that have a shiny metallic surface.





In this model we discuss the amount of light reflected is greatest in the direction of perfect mirror reflection ,  $r$ , where the angle of incidence  $\theta$  equals the angle of reflection. This is the direction in which all light would travel if the surface were a perfect mirror.

At the other nearby angles the amount of light reflected diminishes rapidly

fig (b) shows this with beam patterns. The distance from  $P$  to the beam envelope shows the relative strength of the light scattered in that direction.

( the mirror – reflection direction)

$$r = -s + (s \cdot m) |m| / |m|^2$$

For surfaces that are shiny but are not true mirrors, the amount of light reflected falls off as the angle  $\phi$  between  $r$  and  $v$  increases. In Phong model the  $\phi$  is said to vary as some power  $f$  of the cosine of  $\phi$  i.e., (  $\cos(\phi)^f$  ) in which  $f$  is chosen experimentally and usually lies between 1 and 200.

$$I_{sp} = I_s \rho_s (r \cdot v / |r| \cdot |v|)^f \quad \text{is the specular reflection coefficient}$$

$\cos(\phi)$  is the dot product between  $r$  and  $v$

### The Role of Ambient Light and Exploiting Human Perception

This light arrives by multiple reflections from various objects in the surroundings. But it would be computationally very expensive to model this kind of light.

#### Ambient Sources and Ambient Reflections

To overcome the problem of totally dark shadows we imagine that a uniform background glow called **ambient light** exists in the environment. The ambient light source spreads in all directions uniformly.

The source is assigned an intensity  $I_a$ . Each face in the model is assigned a value for its **ambient reflection coefficient**  $p_a$ , and the term  $I_a p_a$  is added to the diffuse and specular light that is reaching the eye from each point  $P$  on that face.  $I_a$  and  $p_a$  are found experimentally.

Too little ambient light makes shadows appear too deep and harsh., too much makes the picture look washed out and bland.

### How to combine Light Contributions

We sum the three light contributions –diffuse, specular and ambient to form the total amount of light  $I$  that reaches the eye from point  $P$ :

$$I = \text{ambient} + \text{diffuse} + \text{specular}$$

$$I = I_a p_a + I_d p_d \times \text{lambert} + I_s p_s \times \text{phong}^f \quad \text{Where we define the values}$$

$$\text{lambert} = \max \left( 0, \frac{s \cdot m}{|s| |m|} \right) \quad \text{and} \quad \text{phong} = \max \left( 0, \frac{h \cdot m}{|h| |m|} \right)$$

$I$  depends on various source intensities and reflection coefficients and the relative positions of the point  $P$ , the eye and the point light source.

### To Add Color

When dealing with colored sources and surfaces we calculate each color component individually and simply add them to form the final color of the reflected light.

$$I_r = I_{ar} \rho_{ar} + I_{dr} \rho_{dr} \times \text{lambert} + I_{spr} \rho_{sr} \times \text{phong}_f$$

$$I_g = I_{ag} \rho_{ag} + I_{dg} \rho_{dg} \times \text{lambert} + I_{sg} \rho_{sg} \times \text{phong}_f$$

$$I_b = I_{ab} \rho_{ab} + I_{db} \rho_{db} \times \text{lambert} + I_{sb} \rho_{sb} \times \text{phong}_f \text{ ----- (1)}$$

The above equations are applied three times to compute the red, green and blue components of the reflected light.

The light sources have three types of color :

ambient =(I<sub>ar</sub>,I<sub>ag</sub>,I<sub>ab</sub>) , diffuse=(I<sub>dr</sub>,I<sub>dg</sub>,I<sub>db</sub>) and specular=(I<sub>spr</sub>,I<sub>spg</sub>,I<sub>spb</sub>).

Usually the diffuse and the specular light colors are the same. The terms lambert and phong<sub>f</sub> do not depends on the color component so they need to be calculated once. To do this we need to define

nine reflection coefficients:

ambient reflection coefficients:  $\rho_{ar}$  ,  $\rho_{ag}$  and  $\rho_{ab}$

diffuse reflection coefficients:  $\rho_{dr}$  ,  $\rho_{dg}$  and  $\rho_{db}$

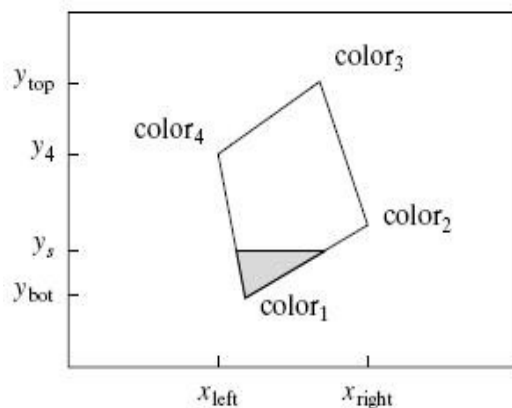
specular reflection coefficients:  $\rho_{sr}$  ,  $\rho_{sg}$  and  $\rho_{sb}$

## Topic2:

### FLAT SHADING AND SMOOTH SHADING

#### Painting a Face:

**A convex quadrilateral whose face is filled with color**



The screen coordinates of each vertex is noted.

The lowest and highest points on the face are y<sub>bott</sub> and y<sub>top</sub>. The tiler first fills in the row at y= y<sub>bott</sub> , then at y<sub>bott</sub> + 1, etc.

At each scan line y<sub>s</sub>, there is a leftmost pixel x<sub>left</sub> and a rightmost pixel x<sub>right</sub>. The tiler moves from x<sub>left</sub> to x<sub>right</sub>, placing the desired color in each pixel.

The tiler is implemented as a simple double loop:

```
for (int y= ybott ; y<= ytop; y++) // for each scan line
{
    find xleft and xright

    for( int x= xleft ; x<= xright; x++) // fill across the scan line
    {
        find the color c for this pixel put c into the pixel at
        (x,y)
    }
}
```

The main difference between flat and smooth shading is the manner in which the color c is determined in each pixel.

## Flat Shading

When a face is flat, like a roof and the light sources are distant, the diffuse light component varies little over different points on the roof. In such cases we use the same color for every pixel covered by the face.

OpenGL offers a rendering mode in which the entire face is drawn with the same color **find the color c for this pixel** is not inside the loops, but appears before the loop, setting c to the color of one of the vertices.

Flat shading is invoked in OpenGL using the command

**glShadeModel(GL\_FLAT);**

### Drawback : 1) MachBandEffect

When objects are rendered using flat shading. Edges between faces actually appear more pronounced than they would on an actual physical object due to a phenomenon in the eye known as **lateral inhibition**.

2) Specular highlights are rendered poorly with flat shading because the entire face is filled with a color that was computed at only one vertex.

### Smooth Shading

Smooth shading attempts to de-emphasize edges between faces by computing colors at more points on each face.

The two types of smooth shading

Gouraud shading Phong shading

## Gouraud Shading

Gouraud shading computes a different value of c for each pixel. For the scan line  $y_s$  in the fig. , it finds the color at the leftmost pixel,  $color_{left}$ , by linear interpolation of the colors at the top and bottom of the left edge of the polygon. For the same scan line the color at the top is  $color_4$ , and that at the bottom is  $color_1$ , so  $color_{left}$  will be calculated as

$color_{left} = \text{lerp}(color_1, color_4, f),$  -----(1) where the fraction

$$f = \frac{y_s - y_{bott}}{y_4 - y_{bott}}$$

varies between 0 and 1 as  $y_s$  varies from  $y_{bott}$  to  $y_4$ . The eq(1) involves three calculations since each color quantity has a red, green and blue component.

$Color_{right}$  is found by interpolating the colors at the top and bottom of the right edge. The tiler then fills across the scan line , linearly interpolating between  $color_{left}$  and  $color_{right}$  to obtain the color at pixel x:

$$c(x) = \text{lerp}(color_{left}, color_{right}, x - x_{left} / x_{right} - x_{left})$$

To increase the efficiency of the fill, this color is computed incrementally at each pixel . that is there is a constant difference between  $c(x+1)$  and  $c(x)$  so that

$$C(x+1) = c(x) + \frac{color_{right} - color_{left}}{x_{right} - x_{left}}$$

The incremented is calculated only once outside of the inner most loop. The code:

```
For ( int y= ybott; y<=ytop ; y++)          //for each scan line
{
    find xleft and xright

    find colorleft and colorright

    colorinc=( colorright - colorleft) / (xright - xleft);
```

```
for(int x= xleft, c=colorleft; x<=xright; x++, c+=colorinc) put c into the pixel
at (x,y)
```

```
}
```

Computationally Gouraud shading is more expensive than flat shading. Gouraud shading is established in OpenGL using the function:

```
glShadeModel(GL_SMOOTH);
```

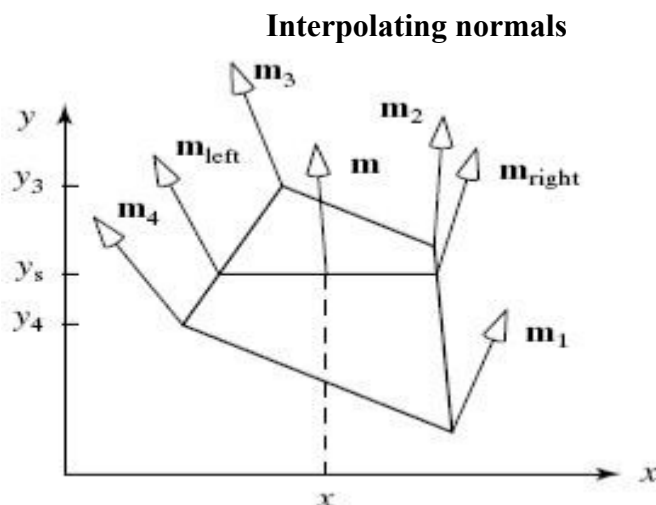
Gouraud shading does not picture highlights well because colors are found by interpolation. Therefore in Gouraud shading the specular component of intensity is suppressed.

### Phong Shading

Highlights are better reproduced using Phong Shading. Greater realism can be achieved with regard to highlights on shiny objects by a better approximation of the normal vector to the face at each pixel this type of shading is called as Phong Shading

When computing Phong Shading we compute the normal vector at each pixel by interpolating the normal vectors at the vertices of the polygon.

The fig shows a projected face with the normal vectors  $m_1$ ,  $m_2$ ,  $m_3$  and  $m_4$  indicated at the four vertices.



For the scan line  $y_s$ , the vectors  $m_{left}$  and  $m_{right}$  are found by linear interpolation

$$m_{left} = \text{lerp} \left( m_4, m_3, \frac{y_s - y_4}{y_3 - y_4} \right)$$

This interpolated vector must be normalized to unit length before it is used in the shading formula once  $m_{left}$  and  $m_{right}$  are known they are interpolated to form a normal vector at each  $x$  along the scan line that vector is used in the shading calculation to form the color at the pixel.

In Phong Shading the direction of the normal vector varies smoothly from point to point and more closely approximates that of an underlying smooth surface the production of specular highlights are good and more realistic renderings produced.

### Drawbacks of Phong Shading

Relatively slow in speed

More computation is required per pixel

**Note:** OpenGL does not support Phong Shading

### Topic3:

#### Adding texture to faces

The realism of an image is greatly enhanced by adding surface texture to various faces of a mesh object. The basic technique begins with some texture function, **texture(s,t)** in **texture space**, which has two parameters s and t. The function texture(s,t) produces a color or intensity value for each value of s and t between 0(dark)and 1(light).

The two common sources of textures are

Bitmap Textures

Procedural Textures

**Bitmap Textures** Textures are formed from bitmap representations of images, such as digitized photo. Such a representation consists of an array **txtr[c][r]** of color values. If the array has C columns and R rows, the indices c and r vary from 0 to C-1 and R-1 resp. The function texture(s,t) accesses samples in the array as in the code:

```
Color3 texture (float s, float t)
{ return txtr[ (int) (s * C)][(int) (t * R)]; }
```

Where Color3 holds an RGB triple. Example: If R=400 and C=600, then the texture (0.261, 0.783) evaluates to txtr[156][313]. Note that a variation in s from 0 to 1 encompasses 600 pixels, the variation in t encompasses 400 pixels.

To avoid distortion during rendering , this texture must be mapped onto a rectangle with aspect ratio 6/4.

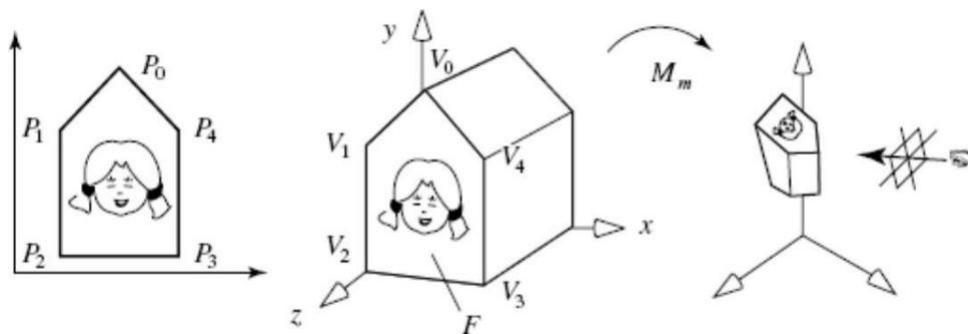
**Procedural Textures** Textures are defined by a mathematical function or procedure. For example a spherical shape could be generated by a function:

```
float fakesphere( float s, float t)
{ float r= sqrt((s-0.5) * (s-0.5)+ (t-0.5) * (t-0.5)); if (r < 0.3) return 1-r/0.3; //sphere intensity else return 0.2; //dark background }
```

This function varies from 1(white) at the center to 0 (black) at the edges of the sphere.

**Painting the Textures onto a Flat Surface** Texture space is flat so it is simple to paste texture on a flat surface.

#### Mapping texture onto a planar polygon



The fig. shows a texture image mapped to a portion of a planar polygon,  $F$ . We need to specify how to associate points on the texture with points on  $F$ .

In OpenGL we use the function

### **glTexCoord2f()**

to associate a point in texture space  $P_i=(s_i,t_i)$  with each vertex  $V_i$  of the face. the function

**glTexCoord2f(s,t)** sets the current texture coordinate to  $(s,y)$ . All calls to **glVertex3f()**

is called after a call to **glTexCoord2f()**, so each vertex gets a new pair of texture coordinates. Example to define a quadrilateral face and to position a texture on it,

we send OpenGL four texture coordinates and four 3D points, as follows:

```
glBegin(GL_QUADS); //defines a quadrilateral face
```

```
glTexCoord2f(0.0,0.0); glVertex3f(1.0,2.5,1.5);
```

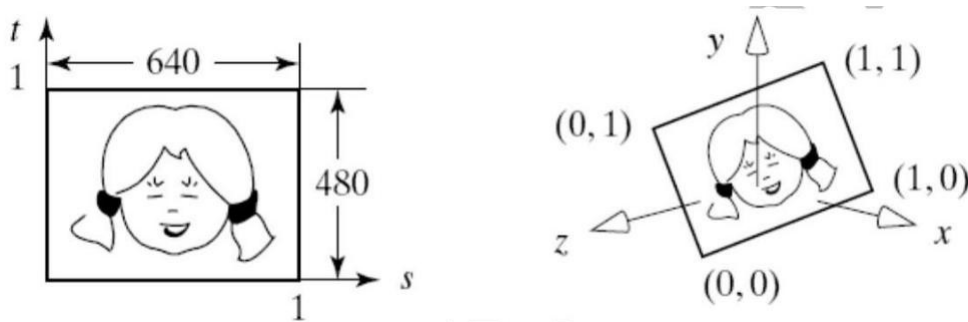
```
glTexCoord2f(0.0,0.6);glVertex3f(1.0,3.7,1.5);
```

```
glTexCoord2f(0.8,0.6);glVertex3f(2.0,3.7,1.5);
```

```
glTexCoord2f(0.8,0.0);glVertex3f(2.0,2.5,1.5);
```

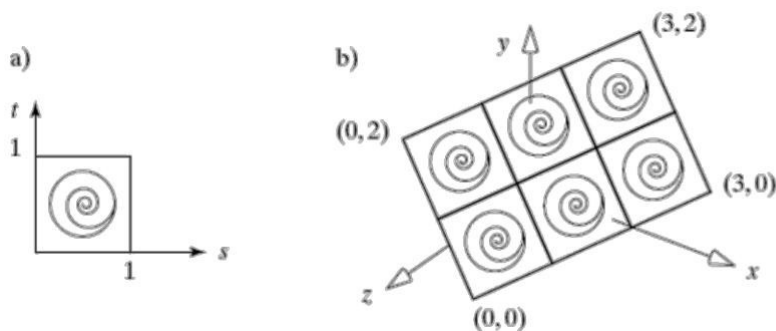
```
glEnd();
```

Mapping a Square to a Rectangle



The fig. shows the a case where the four corners of the texture square are associated with the four corners of a rectangle. In this example, the texture is a 640-by-480 pixel bit map and it is pasted onto a rectangle with aspect ratio 640/480, so it appears without distortion.

### **Producing repeated textures**

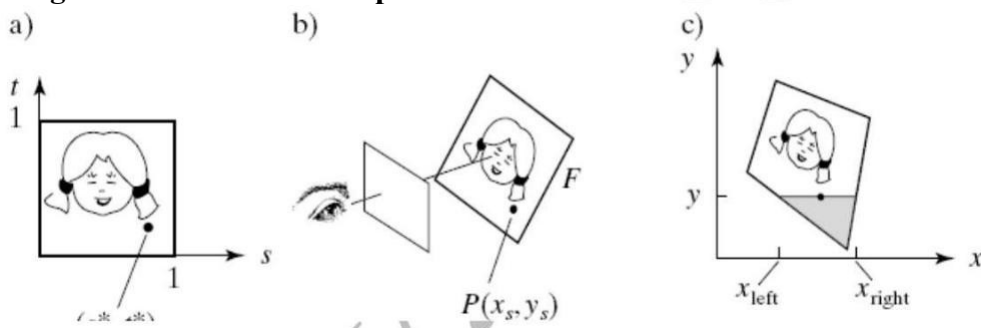




The fig. shows the use of texture coordinates, that tile the texture, making it to repeat. To do this some texture coordinates that lie outside the interval  $[0,1]$  are used. When rendering routine encounters a value of  $s$  and  $t$  outside the unit square, such as  $s=2.67$ , it ignores the integral part and uses only the fractional part 0.67. A point on a face that requires  $(s,t)=(2.6,3.77)$  is textured with texture  $(0.6,0.77)$ . The points inside  $F$  will be filled with texture values lying inside  $P$ , by finding the internal coordinate values  $(s,t)$  through the use of interpolation.

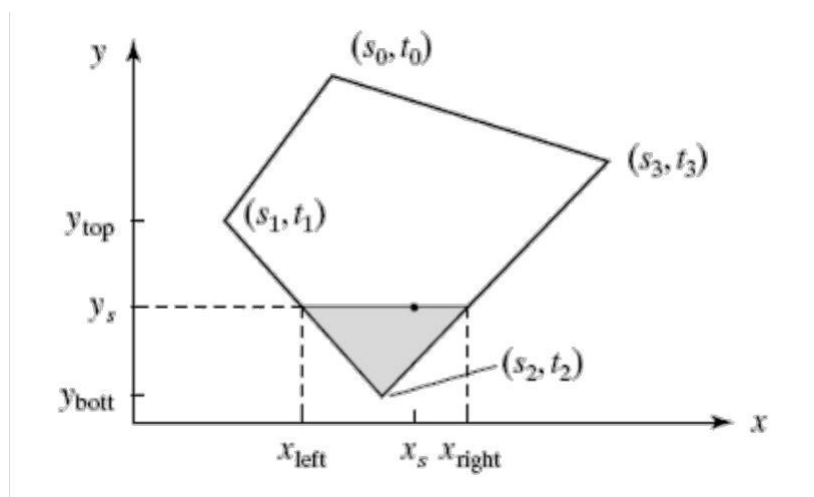
**Rendering texture** in a face  $F$  is similar to Gouraud Shading. It proceeds across the face pixel by pixel. For each pixel it must determine the corresponding texture **coordinates**  $(s,t)$ , access the texture and set the pixel to the proper texture color. Finding the coordinates  $(s,t)$  should be done carefully.

### Rendering a face in a camera snapshot



The fig shows the camera taking a snapshot of a face  $F$  with texture pasted onto it and the rendering in progress. The scan line  $y$  is being filled from  $x_{\text{left}}$  to  $x_{\text{right}}$ . For each  $x$  along this scan line, we compute the correct position on the face and from that, obtain the correct position  $(s^*, t^*)$  within the texture. **Incremental calculation of texture coordinates**

We compute  $(s_{\text{left}}, t_{\text{left}})$  and  $(s_{\text{right}}, t_{\text{right}})$  for each scan line in a rapid incremental fashion and to interpolate between these values, moving across these scan lines. Linear interpolation produces some distortion in the texture. This distortion is disturbing in an animation when the polygon is rotating. Correct interpolation produces an texture as it should be. In an animation this texture would appear to be firmly attached to the moving or rotating face

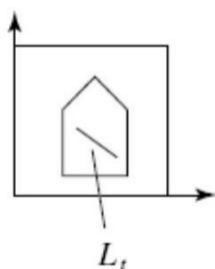


# • Linear interpolation in screen space:

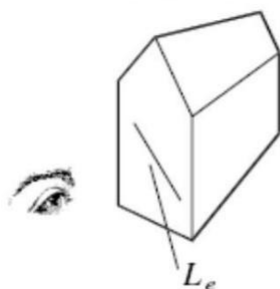


## Lines in one space map to lines in another

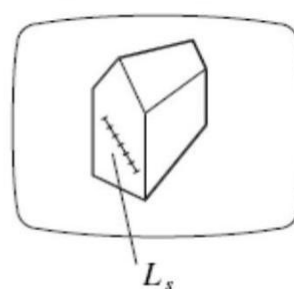
a) texture space



b) eye space



c) screen space



Affine and projective transformations preserve straightness, so line  $L_e$  in eye space projects to line  $L_s$  in screen space, and similarly the texels we wish to draw on line  $L_s$  lie along the line  $L_t$  in texture spaces, which maps to  $L_e$ . The question is : if we move in equal steps across  $L_s$  on the screen, how should we step across texels along  $L_t$  in texture space? **How does motion along corresponding lines operate?**

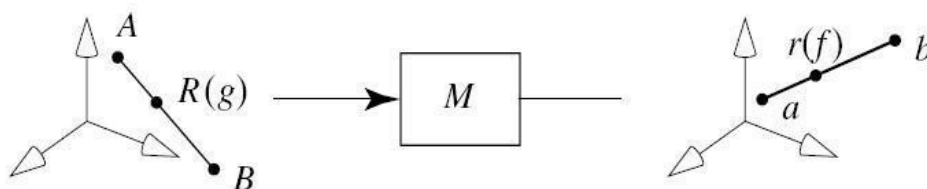


fig. shows a line  $AB$  in 3D being transformed into the line  $ab$  in 3D by the matrix  $M$ .

$A$  maps to  $a$ ,  $B$  maps to  $b$ .

Consider the point  $R(g)$  that lies a fraction  $g$  of the way between  $A$  and  $B$ .

This point maps to some point  $r(f)$  that lies a fraction  $f$  of the way from  $a$  to  $b$ .

The fractions  $f$  and  $g$  are not the same. The question is, As  $f$  varies from 0 to 1, how exactly does  $g$  vary? How does motion along  $ab$  correspond to motion along  $AB$ ?

## Topic4:

What is called a shadow buffer?

What does sliding means?

Write down the syntax for `glFramebufferRenderbufferEXT()`.

What is the function of `glCheckFramebufferStatusEXT()`?

Write down the syntax for `glGetRenderbufferParameterivEXT()`.

List out some of the rules of FBO completeness.

## Adding shadows of objects

### Explain Adding shadows of objects

Shadows make an image more realistic. The way one object casts a shadow on another object gives important visual clues as to how the two objects are positioned with respect to each other. Shadows convey a lot of information as such, you are getting a second look at the object from the view point of the light source. There are two methods for computing shadows:

Shadows as Texture

Creating shadows with the use of a shadow buffer

### Shadows as Texture

The technique of “painting” shadows as a texture works for shadows that are cast onto a flat surface by a point light source. The problem is to compute the shape of the shadow that is cast.

### Computing the shape of a shadow

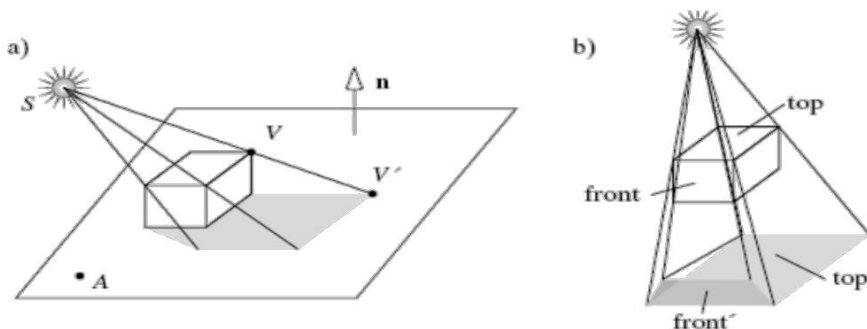
Fig(a) shows a box casting a shadow onto the floor. The shape of the shadow is determined by the projections of each of the faces of the box onto the plane of the floor, using the light source as the center of projection.

Fig(b) shows the superposed projections of two of the faces. The top face projects to “top” and the front face to “front”. This provides the key to drawing the shadow. After drawing the plane by the use of ambient, diffuse and specular light contributions, draw the six projections of the box’s faces on the plane, using only the ambient light. This technique will draw the shadow in the right shape and color. Finally draw the box.

### Building the “Projected” Face

To make the new face  $F''$  produced by  $F$ , we project each of the vertices of  $F$  onto the plane. Suppose that the plane passes through point  $A$  and has a normal vector  $n$ . Consider projecting vertex  $V$ , producing  $V''$ .  $V''$  is the point where the ray from source at  $S$  through  $V$  hits the plane, this point is

$$V'' = S + (V - S) \frac{n \cdot (A - S)}{n \cdot (V - S)}$$



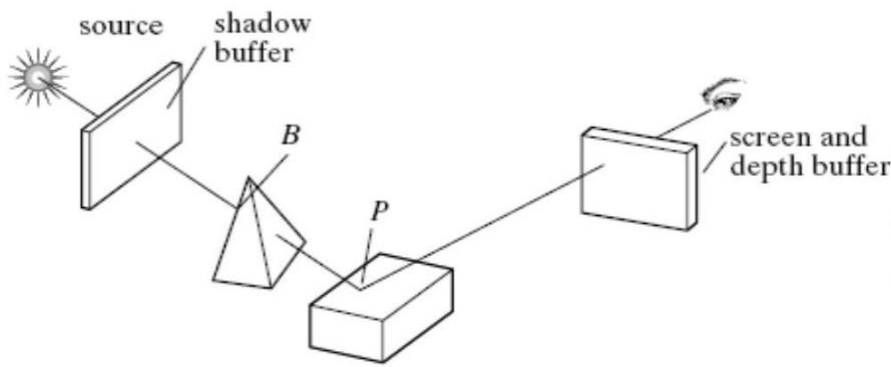
### Creating Shadows with the use of a Shadow buffer

This method uses a variant of the depth buffer that performs the removal of hidden surfaces. An auxiliary second depth buffer called a shadow buffer is used for each light source. This requires a lot of memory. This method is based on the principle that any points in a scene that are hidden from the light source must be in

shadow. If no object lies between a point and the light source, the point is not in shadow. The shadow buffer contains a depth picture of the scene from the point of view of the light source. Each of the elements of the buffer records the distance from the source to the closest object in the associated direction. Rendering is done in two stages:

**1) Loading the shadow buffer** The shadow buffer is initialized with 1.0 in each element, the largest pseudodepth possible. Then through a camera positioned at the light source, each of the scene is rasterized but only the pseudodepth of the point on the face is tested. Each element of the shadow buffer keeps track of the smallest pseudodepth seen so far.

**2) Using the shadow buffer**



The fig. shows a scene being viewed by the usual eye camera and a source camera located at the light source. Suppose that point P is on the ray from the source through the shadow buffer pixel  $d[i][j]$  and that point B on the pyramid is also on this ray.

If the pyramid is present  $d[i][j]$  contains the pseudodepth to B;

if the pyramid happens to be absent  $d[i][j]$  contains the pseudodepth to P.

The shadow buffer calculation is independent of the eye position, so in an animation in which only the eye moves, the shadow buffer is loaded only once. The shadow buffer must be recalculated whenever the objects move relative to the light source.

**2) Rendering the scene** Each face in the scene is rendered using the eye camera. Suppose the eye camera sees point P through pixel  $p[c][r]$ . When rendering  $p[c][r]$ , we need to find

The pseudodepth D from source to P

the index location  $[i][j]$  in the shadow buffer that is to be tested and the value  $d[i][j]$  stored in the shadow buffer

If  $d[i][j]$  is less than D, the point P is in the shadow and  $p[c][r]$  is set using only ambient light. Otherwise P is not in shadow and  $p[c][r]$  is set using ambient, diffuse and specular light.

### Topic5:

**Write down and explain the details to build a camera in a program?**

#### 4.5 BUILDING A CAMERA IN A PROGRAM

To have a finite control over camera movements, we create and manipulate our own camera in a program. After each change to this camera is made, the camera tells OpenGL what the new camera is. We create a Camera

class that does all things a camera does. In a program we create a Camera object called cam, and adjust it with functions such as the following:

```
cam.set(eye, look, up); // initialize the camera
cam.slide(-1, 0, -2); //slide the camera forward and to the left
cam.roll(30); // roll it through 30 degree
cam.yaw(20); // yaw it through 20 degree
```

We create a Camera class that does all things a camera does.

```
class Point3
{
public:
    float x,y,z;
    void set(float dx,float dy,float dz)
    { x=dx;y=dy;z=dz;}
    void set(Point3 &p){x=p.x;y=p.y;z=p.z;}
    Point3(float xx,float yy,float zz)
    {
        x=xx;y=yy;z=zz;
    }
    Point3()
    {
        x=y=z=0;
    }
};

class Vector3
{public:
    float x,y,z;
    Vector3()
    {
        x=y=z=0;
    }
    Vector3(float xx,float yy,float zz)
    {
        x=xx;y=yy;z=zz;
    }
    void set(float dx,float dy,float dz)
    {
        x=dx;y=dy;z=dz;
    }
    void set(Vector3 &v)
    {
        x=v.x;y=v.y;z=v.z;
    }
};

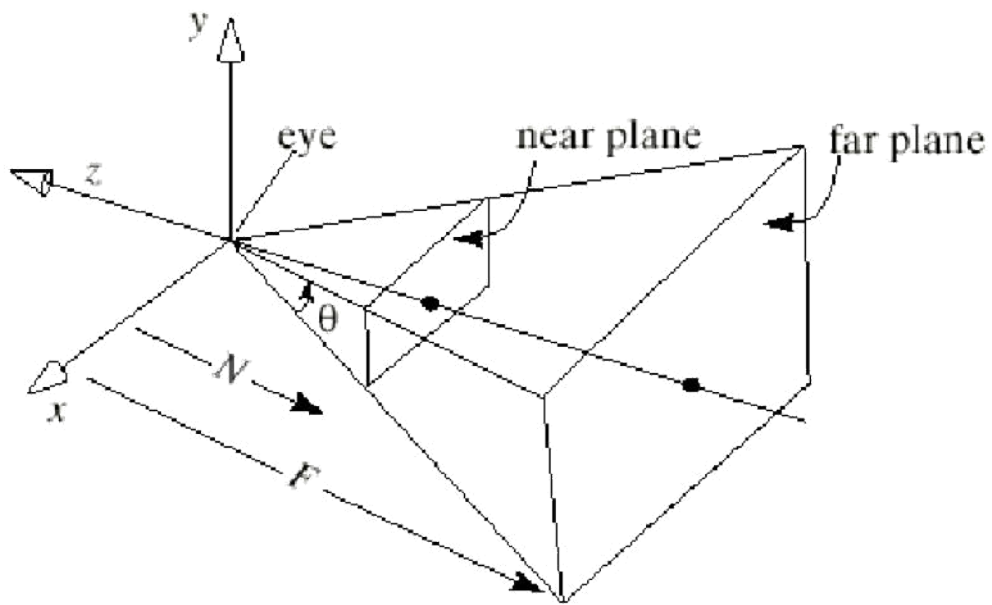
void normalize()
{
    / Calculate the magnitude of our vector float
    magnitude = sqrt((x * x) + (y * y) + (z
    * z));

    / As long as the magnitude isn't zero, divide
    each element by the magnitude
    / to get the normalised value between -1
    and +1
    if (magnitude != 0)
    {
        x /= magnitude;
        y /= magnitude;
        z /= magnitude;
    }
}

void cross( Vector3 &vec1, Vector3 &vec2)
{
    x=vec1.y * vec2.z - vec1.z * vec2.y;
    y= vec1.z * vec2.x - vec1.x * vec2.z;
    z= vec1.x * vec2.y - vec1.y * vec2.x;
}

float dot( Vector3 &vec1, Vector3 &vec2)
{
    return vec1.x * vec2.x + vec1.y * vec2.y +
    vec1.z * vec2.z;
}

};
```

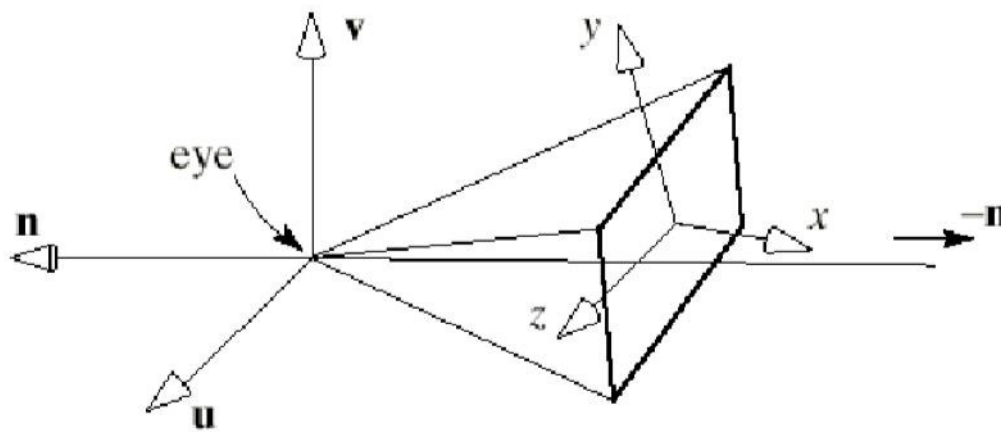


```

class camera1
{
private:
Point3 eye;
Vector3 u,v, n,Sample;
double viewAngle, aspect, nearDist, farDist; //view volume shape
void setModelViewMatrix();
public:
//default constructor
camera1(){}
void setShape(float vAng, float asp, float nearD, float farD); //set viewvolume
void set(Point3 eye, Point3 look, Vector3 up); //like gluLookAt()
void roll(float angle); //roll it
void pitch(float angle); // increase the pitch
void yaw(float angle); //yaw it
void slide(float delU, float delV, float delN); //slide
};

```

## The General Camera with Arbitrary Orientation and Position



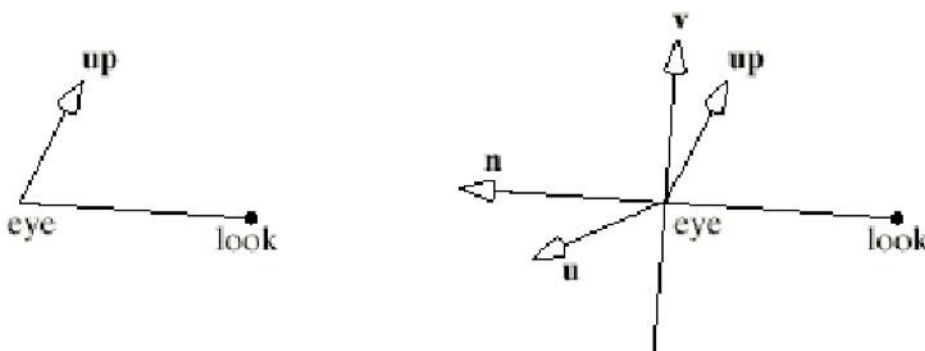
- It is useful to attach an explicit co-ordinate system to the camera as shown in the figure above.
- This co-ordinate system has its origin at the eye and has three axes, usually called the  $u$ -,  $v$ -, and  $n$ -axis which define the orientation of the camera

## The General Camera with Arbitrary Orientation and Position

- The axes are pointed in directions given by the vectors  $u$ ,  $v$ , and  $n$
- Because, by default, the camera looks down the negative  $z$ -axis, we say in general that the camera looks down the negative  $n$ -axis in the direction  $-n$
- The direction  $u$  points off “to the right of” the camera and the direction  $v$  points “upward”

### Camera Control

#### set function



- What are the directions of  $u$ ,  $v$  and  $n$  when we execute `set()` with given values for **eye**, **look** and **up**
- If given the locations of **eye**, **look** and **up**, we immediately know that  $n$  must be parallel to the vector **eye-look**, as shown above. so we can set  $n = \text{eye} - \text{look}$
- We now need to find a  $u$  and a  $v$  that are perpendicular to  $n$  and to each other.
- The  $u$  direction points “off to the side” of a camera, so it is natural to make it perpendicular to **up** which the user has said is the “upward” direction.
- An easy way to build a vector that is perpendicular to two given vector is to form their cross product, so we set  $u = \text{up} \times n$
- With  $u$  and  $n$  formed it is easy to determine  $v$  as it must be perpendicular to both and is thus the cross product of  $u$  and  $n$  thus  $v = n \times u$
- Notice that  $v$  will usually not be aligned with **up** as  $v$  must be aimed perpendicular to  $n$  whereas the user provides **up** as a suggestion of “upwardness” and the only property of **up** that is used is its cross product with  $n$

To summarise, given eye look and up, we form

$\mathbf{n} = \text{eye} - \text{look}$

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

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

we then normalize to unit length

### Viewing matrix

- The job of the View matrix is to convert world co-ordinates to camera co-ordinates it must transform the camera's coordinate system into the generic position for the camera
- An orthographic projection is specified by setting “clip” planes
- We do this using
  - near - far ○
  - left - right ○
  - top - bottom
- We can then create a matrix to scale and set the view of our

object `cam.set(Point3(4, 4, 4), Point3(0, 0, 0), Vector3(0, 1, 0));`

```
void camera1 :: set (Point3 Eye, Point3 look, Vector3 up)
{ eye.set(Eye); // store the given eye position n.set(eye.x -
look.x, eye.y - look.y, eye.z - look.z); // make n
u.cross(up,n);
u.set(u);      //make u= up X n
n.normalize(); // make them unit length
u.normalize();
v.cross(n,u);
v.set(v);      // make v= n X u
v.normalize();
setModelViewMatrix();    // tell OpenGL
}
```

### setModelViewMatrix();

It is used only by member functions of the class and needs to be called after each change is made to the camera's position. The matrix V transforms the world points into camera coordinates.

$$V = \begin{pmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$(d_x, d_y, d_z) = (-\text{eye} \cdot \mathbf{u}, -\text{eye} \cdot \mathbf{v}, -\text{eye} \cdot \mathbf{n})$

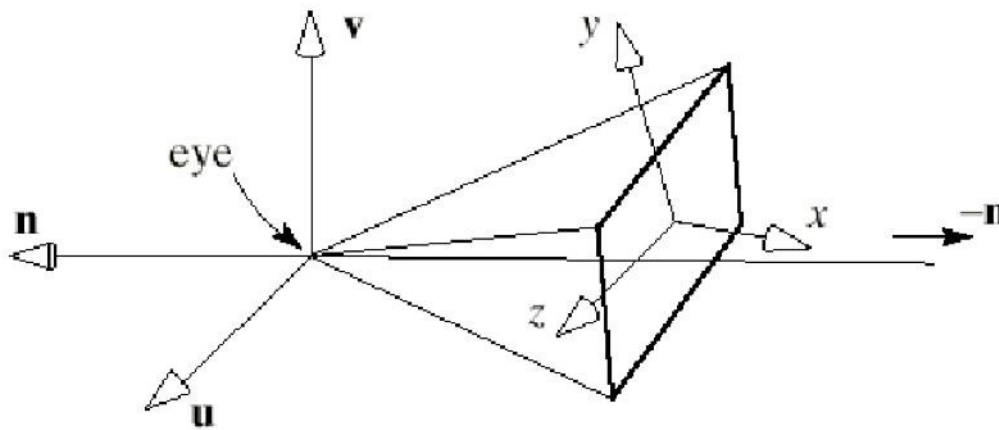
The utility routine computes the matrix V on the basis of current values of eye, u, v and n and loads the matrix directly into the modelview matrix using `glLoadMatrixf()`.

```
void Camera :: setModelViewMatrix(void)
{ //load modelview matrix with existing camera values float m[16];
  Vector3 eVec(eye.x, eye.y, eye.z); //a vector version of eye
  m[0]= u.x ; m[4]= u.y ; m[8]=      u.z ; m[12]= -eVec.dot(u);
  m[1]= v.x ; m[5]= v.y ; m[9]=      v.z ; m[13]= -eVec.dot(v);
  m[2]= n.x ; m[6]= n.y ; m[10]= n.z ; m[14]= -eVec.dot(n);
  m[3]= 0 ; m[7]= 0 ; m[11]= 0 ; m[15]= 1.0;
  glMatrixMode(GL_MODELVIEW);
  glLoadMatrixf(m); //load OpenGL's modelview matrix
}
```



The utility routine computes the matrix V on the basis of current values of eye, u, v and n and loads the matrix directly into the modelview matrix using `glLoadMatrixf()`.

## Setting the View Volume



- OpenGL 1.x provided a simple way to set the view volume in a program by setting the projection Matrix
- This was done using `gluPerspective`
- given in degrees and sets the angle between the top and bottom walls of the pyramid.
- The parameters `w` and `h` sets the aspect ratio of any window parallel to the `xy`-plane
- The value `N` is the distance from the eye to the near plane, and `F` is the distance from the eye to the far plane. `N` and `F` should be positive.
- `gluPerspective(viewAngle, aspect, N, farDist);`

**`cam.setShape(30.0f,64.0f/48.0f,0.5f,50.0f) ;`**

It is used only by member functions of the class and needs to be called after each change is made to the camera's position. The matrix V transforms the world points into camera coordinates.

`void camera1 :: setShape(float vAng, float asp, float nearD, float farD)`

```
{
    viewAngle = vAng;
    aspect = asp;
    nearDist = nearD;
    farDist = farD;
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(viewAngle, aspect, nearDist, farDist);
}
```

Example:

```
cam.set(eye, look, up);           // initialize the camera
cam.roll(30);                     // roll it through 30 degree
cam.yaw(20);                      // yaw it through 20 degree
```

## Sliding the Camera

Sliding the camera means to move it along one of its own axes that is, in the `u`, `v` and `n` direction without rotating it. Since the camera is looking along the negative `n` axis, movement along `n` is forward or back. Movement along `u` is left or right and along `v` is up or down.

To move the camera a distance `D` along its `u` axis, set `eye` to `eye + Du`. For convenience, we can combine the three possible slides in a single function:

`slide(delU, delV, delN)`

slides the camera amount `delU` along `u`, `delV` along `v` and `delN` along `n`. The code is as follows:

```
cam.slide(-1, 0, -2);             //slide the camera forward and to the left
```

```

void camera1 :: slide(float delU, float delV, float delN)
{eye.x += delU * u.x + delV * v.x + delN * n.x;
eye.y += delU * u.y + delV * v.y + delN * n.y;
eye.z += delU * u.z + delV * v.z + delN * n.z;
setModelViewMatrix();
}

```

### Rotating the Camera

Roll, pitch and yaw the camera , involves a rotation of the camera about one of its own axes.

To roll the camera we rotate it about its own n-axis. This means that both the directions u and v must be rotated as shown in fig.

#### Rolling the camera

$$\begin{aligned}
 u' &= \cos(\alpha)u + \sin(\alpha)v; \\
 v' &= -\sin(\alpha)u + \cos(\alpha)v
 \end{aligned}$$

```

void camera1 :: roll (float angle)
{ // roll the camera through angle
degrees float cs = cos (3.14159265/180 *
angle); float sn = sin (3.14159265/180 *
angle); Vector3 t = u; //remember old u
u.set(cs * t.x - sn * v.x , cs * t.y - sn * v.y, cs * t.z - sn * v.z);
v.set(sn * t.x + cs * v.x , sn * t.y + cs * v.y, sn * t.z + cs * v.z);
setModelViewMatrix();
}

```

//Implementation of pitch()

```

void camera1 :: pitch (float angle)
{ // pitch the camera through angle degrees around U
float cs = cos(3.14159265/180 * angle);
float sn = sin(3.14159265/180 * angle);
Vector3 t(v); // remember old v
v.set(cs*t.x - sn*n.x, cs*t.y - sn*n.y, cs*t.z - sn*n.z);
n.set(sn*t.x + cs*n.x, sn*t.y + cs*n.y, sn*t.z + cs*n.z);
setModelViewMatrix();
}

```

//Implementation of yaw()

```

void camera1 :: yaw (float angle)
{ // yaw the camera through angle degrees around V
float cs = cos(3.14159265/180 * angle);
float sn = sin(3.14159265/180 * angle);
Vector3 t(n); // remember old v
n.set(cs*t.x - sn*u.x, cs*t.y - sn*u.y, cs*t.z - sn*u.z);
u.set(sn*t.x + cs*u.x, sn*t.y + cs*u.y, sn*t.z + cs*u.z);
setModelViewMatrix();
}

```

```

camera1 cam; //global camera object
void myKeyboard(unsigned char key, int x, int y) //----- myKeyboard-----
{
    switch(key)
    {
        //controls for the camera
        case 'F' :      //slide camera forward
            cam.slide(0, 0, 0.2);
            break;
        case 'B':      //slide camera back
            cam.slide(0, 0,-0.2);
            break;
        case 'P':
            cam.pitch(-1.0);
            break;
        case 'Q':
            cam.pitch(1.0);
            break;
        case 'Y':
            cam.yaw(1.0);
            break;
        case 'Z':
            cam.yaw(-1.0);
            break;
        case 'R':
            cam.roll(1.0);
            break;
        case 'S':
            cam.roll(-1.0);
            break;
    }
    glutPostRedisplay(); //draw it again
}

void initGL() {
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f); // Set background color to black and opaque
    glColor3f(0.0f,0.0f,0.0f);
    glViewport(0, 0, 640,480); // Set background depth to
    farthest }

/* Handler for window-repaint event. Called back when the window first appears and
   whenever the window needs to be re-painted. */
void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear color ,depth buffers
    glutWireTeapot(1.0);
    glFlush();
    glutSwapBuffers(); // Swap the front and back frame buffers (double buffering)
}

```

```

void reshape(GLsizei width, GLsizei height) { // GLsizei for non-negative
    integer // Compute aspect ratio of the new window
    if (height == 0) height = 1; // To prevent divide by 0 GLfloat
    aspect = (GLfloat)width / (GLfloat)height;
    / Set the viewport to cover the new window
    glViewport(0, 0, width, height);
    / Set the aspect ratio of the clipping volume to match the viewport
    glMatrixMode(GL_PROJECTION); // To operate on the Projection matrix
    glLoadIdentity();           // Reset
    / Enable perspective projection with fovy, aspect, zNear and zFar
    gluPerspective(45.0f, aspect, 0.1f, 100.0f);
}
//-----main-----
int main(int argc, char **argv)
{glutInit(&argc, argv);           // Initialize GLUT
glutInitDisplayMode(GLUT_DOUBLE); // Enable double buffered mode
glutInitWindowSize(640, 480); // Set the window's initial width & height
glutInitWindowPosition(50, 50); // Position the window's initial top-left corner
glutCreateWindow("xyz");
glutKeyboardFunc(myKeyboard); // Create window with the given title
glutDisplayFunc(display);      // Register callback handler for window re-paint event
//glutReshapeFunc(reshape);      // Register callback handler for window re-size event
initGL();
cam.set(Point3(4, 4, 4), Point3(0, 0, 0), Vector3(0, 1, 0));
cam.setShape(30.0f, 64.0f/48.0f, 0.5f, 50.0f) ; // Our own OpenGL initialization
glutMainLoop();                // Enter the infinite event-processing loop
return 0;
}

```

## Topic6

### Creating shaded objects

Establishing a point lightsource in the scene and assigning various material properties , include ambient,diffuse and specular light provide key strokes that switches between flat and smooth shading

#### Step1:

#### To Use Light Sources in OpenGL

##### Create a Light Source

In OpenGL we can define upto eight sources, which are referred through names GL\_LIGHT0, GL\_LIGHT1 and so on.

Each source has properties and must be enabled. Each property has a default value. For example, to create a source located at (3,6,5) in the world coordinates

```
1)GLfloat myLightPosition[]={3.0 , 6.0,5.0,1.0 };
2)glLightfv(GL_LIGHT0, GL_POSITION, myLightPosition);
3)glEnable(GL_LIGHTING);           //enable lighting in general
4)glEnable(GL_LIGHT0);             //enable source GL_LIGHT0
```

The array myLightPosition[] specifies the location of the light source.

This position is passed to glLightfv() along with the name GL\_LIGHT0 to attach it to the particular source GL\_LIGHT0.

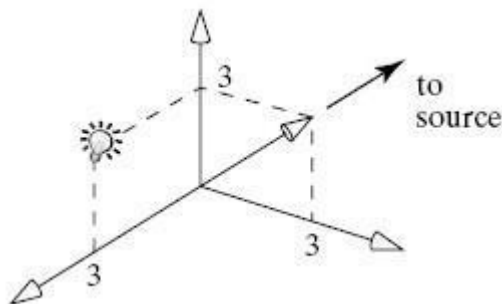
Some sources such as desk lamp are in the scene whereas like the sun are infinitely remote.

OpenGL allows us to create both types by using homogenous coordinates to specify light position:

(x,y,z,1) : a local light source at the position (x,y,z)

(x,y,z,0) a vector to an infinitely remote light source in the direction (x,y,z)

#### A local source and an infinitely remote source



The above fig., shows a local source positioned at (0,3,3,1) and a remote source “located” along vector (3,3,0,0). Infinitely remote light sources are often called “directional”.

In OpenGL you can assign a different color to three types of light that a source emits : ambient , diffuse and specular. Arrays are used to hold the colors emitted by light sources and they are passed to glLightfv() through the following code:

```
5)GLfloat amb0[]={ 0.2 , 0.4, 0.6, 1.0 };           // define some colors
6)GLfloat diff0[]={ 0.8 ,0.9 , 0.5 ,1.0 };
7)GLfloat spec0[]={ 1.0 , 0.8 , 1.0, 1.0 };
8)glLightfv(GL_LIGHT0, GL_AMBIENT, amb0); //attach them to LIGHT0
9)glLightfv(GL_LIGHT0, GL_DIFFUSE, diff0);
10)glLightfv(GL_LIGHT0, GL_SPECULAR, spec0);
```

Colors are specified in RGBA format meaning red, green, blue and alpha. The alpha value is sometimes used for blending two colors on the screen. Light sources have various default values. For all sources:

Default ambient= (0,0,0,1);  
Default diffuse= (1,1,1,1)  
specular=(1,1,1,1)

dimmet possible :black For light source LIGHT0:  
brightest possible:white Default  
brightest possible:white

### Step2:

This version of glMaterialfv takes vector-based color coordinates. The GL\_AMBIENT\_AND\_DIFFUSE parameter specifies that colorBlue will be applied to both ambient and diffuse components of the material.

// Evaluate the reflective properties of the material

```
11)float colorBlue[] = { 0.0f, 0.0f, 1.0f, 1.0f };
```

```
12)glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, colorBlue);
```

The first parameter of the glMaterialfv command described above (GL\_FRONT) indicates which face of the polygon should reflect the light specified by colorBlue.

now, draw polygon as its material properties will be affected by the glMaterialfv call.

### Step3:

```
/ Draw a polygon with current material properties being  
set 13)glBegin(GL_TRIANGLES);  
glVertex3f(-1.0f, 0.0f,  
0.0f); glVertex3f(0.0f, -  
1.0f, 0.0f); glVertex3f(1.0f,  
0.0f, 0.0f); glEnd();
```

### Step4:

```
FlatShading  
14)glShadeModel(GL_FLAT);  
SmoothShading  
15)SmoothShading(GL_SMOOTH);
```

## Topic7: Rendering Textures

## Applying Textures

[illegible]

Step1: OpenGL uses textures that are stored in **pixel maps**, or pixmaps for short.

pixmap is a simple array of pixel values, each pixel value is red, green, and blue color values of texture image so for six texture images we need six pixmaps

The method `readBMPFile()` reads a (24-bit) BMP file and stores the pixel values in its `pixmap` object;

- Our example OpenGL application will use six textures. To create them we first make an RGBpixmap object for each:  
`RGBpixmap pix[6]; // create six (empty) pixmaps`
- **Making a texture from a stored image.**
- OpenGL offers no support for reading an image file and creating the pixel map in memory.
- The method `readBMPFile()`, available on the book's companion website, provides a simple way to read a BMP image into a pixmap. For instance,  
`pix[1].readBMPFile("mandrill.bmp");`  
reads the file `mandrill.bmp` and creates the pixmap in `pix[1]`.
- Once the pixel map has been created, `pix[1].setTexture()` is used to pass the pixmap to OpenGL to make a texture.
- We then load the desired texture image into each one.
- Finally each one is passed to OpenGL to define a texture.
- **Making a procedural texture.**
- We create a checkerboard texture using the method `makeCheckerboard()` and store it in `pix[0]`:  
`pix[0].makeCheckerboard();`
- `glTexImage2D()` to create the actual texture for OpenGL.

#### Step2:

- Once we have a pixel map, we must bind it to a unique integer (its name) to refer to it in OpenGL.
- We arbitrarily assign the names 2001, 2002, ..., 2006 to our six textures.
  - OpenGL can supply unique names: If we need six unique names we can build an array to hold them: `GLuint name[6];` and then call `glGenTextures(6,name)`. OpenGL places six unique integers in `name[0],...,name[5]`, and we subsequently refer to the  $i^{\text{th}}$  texture using `name[i]`.
- The texture is created by making certain calls to OpenGL, which we encapsulate in the method: `void RGBpixmap :: setTexture(GLuint textureName)`

The intensity  $I$  is **replaced** by the value of the texture. (In color, the replacement is for each of the R, G, B components.)

In OpenGL, for  $I = \text{texture}(s,t)$  we use

`glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);`  
`GL_REPLACE` and `GL_DECAL` are equivalent.

Modulating Reflection Coefficient

- Color of an object is color of its diffuse light component; vary diffuse reflection coefficient.
- $I = \text{texture}(s,t) * [I_{\text{pa}} + I_{\text{pd}} \times \text{lambert}] + I_{\text{sp}} \rho_s \times \text{phong}$ .
  - The specular component is the color of the light, not the object.

- In OpenGL, use `glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);`



```

void RGBpixmap :: setTexture(GLuint textureName)
{
    glBindTexture(GL_TEXTURE_2D, textureName);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, nCols, nRows, 0, GL_RGB,
GL_UNSIGNED_BYTE, pixel);
}

```

- The call to `glBindTexture()` binds the given name to the texture being formed. When this call is made at a later time, it will make this texture the active texture.
- The calls to `glTexParameteri()` specify that a pixel should be filled with the texel whose coordinates are nearest the center of the pixel, for both enlarged or reduced versions. This is fast but can lead to aliasing effects.
- Finally, the call to `glTexImage2D()` associates the pixmap with this current texture. This call describes the texture as 2D consisting of RGB byte-triples, gives its width, height, and the address in memory (pixel) of the first byte of the bitmap.
- Texture mapping must also be enabled with `glEnable(GL_TEXTURE_2D)`.
- The call `glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST)` is used to request that OpenGL render the texture properly (using hyperbolic interpolation), so that it appears correctly attached to faces even when a face rotates relative to the viewer in an animation.
- Complete code is in Fig. 8.49. The texture creation, enabling, and hinting is done once, in an initialization routine.
- In `display()` the cube is rotated through angles `xAngle`, and `yAngle`, and the faces are drawn. The appropriate texture must be bound to the face, and the texture coordinates and 3D positions of the face vertices be specified inside a `glBegin()/glEnd()` pair.
- Once the rendering (off screen) of the cube is complete, `glutSwapBuffers()` is called to make the new frame visible.
- Animation is controlled by the callback “idle” function `spinner()`. Whenever there is no user input, `spinner` is called; it alters the rotation angles of the cube slightly, and calls `display()`.