UNIT IV – RENDERING

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

4.1 Introduction to Shading Models

The mechanism of light reflection from an actual surface is very complicated it depends on many factors. Some of these factors are geometric and others are related to the characteristics of the surface.

A shading model dictates how light is scattered or reflected from a surface. 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.

A shading model uses two types of light source to illuminate the objects in a scene : **point light sources** and **ambient light**. 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

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.

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

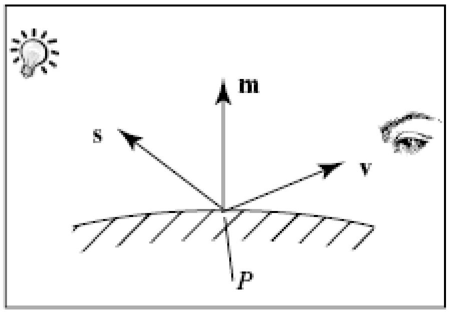
- **Diffuse scattering** occurs when some of the incident light slightly penetrates the surface and is re-radiated uniformly in all directions. Scattered light interacts strongly with the surface and so its color is usually affected by the nature of the surface material.
- **Specular reflections** are more mirrorlike and highly directional. Incident light is directly reflected from its outer surface. This makes the surface looks shinny. In the simplest model the reflected light has the same color as the incident light, this makes the material look like plastic. In a more complex model the color of the specular light varies , providing a better approximation to the shininess of metal surfaces.

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.

4.1.1 Geometric Ingredients For Finding Reflected Light

We need to find three vectors in order to compute the diffuse and specular components. The below fig. shows 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.

The angles between these three vectors form the basis of computing light intensities. These angles are normally calculated using world coordinates.

Each face of a mesh object has two sides. If the object is solid , one is inside and the other is outside. The eye can see only the outside and it is this side for which we must compute light contributions.

We shall develop the shading model for a given side of a face. If that side of the face is turned away from the eye there is no light contribution.

4.1.2 How to Compute the Diffuse Component

Suppose that a light falls from a point source onto one side of a face , a fraction of it is re-radiated diffusely in all directions from this side. Some fraction of the re-radiated part reaches the eye, with an intensity denoted by $\mathbf{I_d}$.

An important property assumed for 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.

On the other hand the amount of light that illuminates the face does not depend on the orientation of the face relative to the point source. The amount of light is proportional to the area of the face that it sees.

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

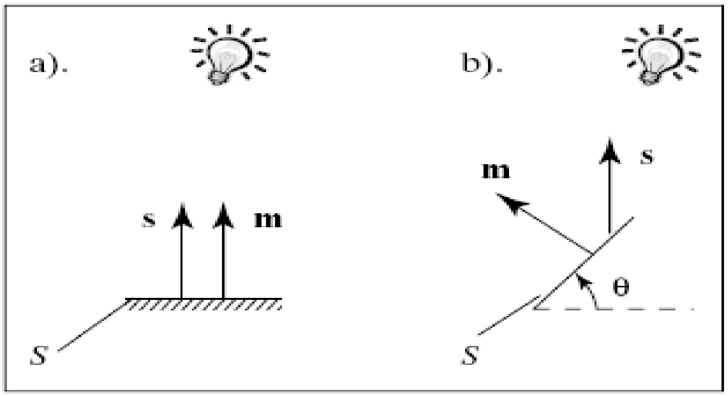


Fig (a) shows the cross section of a point source illuminating a face S when m is aligned with s.

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

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

$$I_d = I_s \rho_d \frac{s.m}{|s||m|}$$

I_s is the intensity of the light source and ρ_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. A more precise computation of the diffuse component is :

$$I_d = I_s \rho_d \max \frac{s.m}{|s||m|},0$$

The reflection coefficient ρ_d depends on the wavelength of the incident light , the angle θ and various physical properties of the surface. But for simplicity and to reduce computation time, these effects are usually suppressed when rendering images. A reasonable value for ρ_d is chosen for each surface.

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

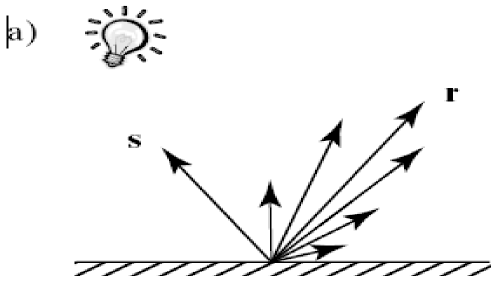
Phong Model

It is easy to apply and the highlights generated by the phong model given an plasticlike appearance , so the phong model is good when the object is made of shinny plastic or glass.

The Phong model is less successful with objects that have a shinny metallic surface.

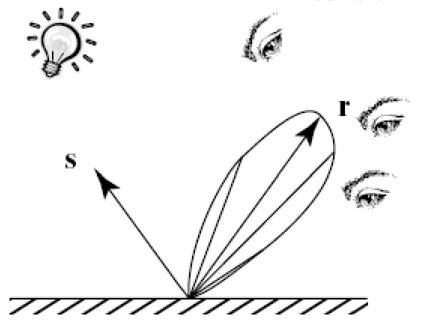
Fig a) shows a situation where light from a source impinges on a surface and is reflected in different directions.

a)

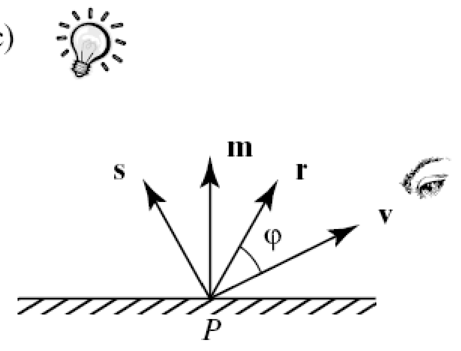


In this model we discuss the amount of light reflected is greatest in the direction of perfect mirror reflection , r, where the angle of incidence θ equals the angle of reflection. This is the direction in which all light would travel if the surface were a perfect mirror. At the other nearby angles the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

b)



c)



Fig(c) shows how to quantify this beam pattern effect . The direction r of perfect reflection depends on both s and the normal vector m to the surface, according to:

$$r = -s + 2\frac{(s.m)}{|m|} m \text{ (the mirror – reflection direction)}$$

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

4.1.4 The Role of Ambient Light and Exploiting Human Perception

The diffuse and specular components of reflected light are found by simplifying the rules by which physical light reflects from physical surfaces. The dependence of these components on the relative position of the eye , model and light sources greatly improves the reality of a picture.

The simple reflection model does not perfectly renders a scene. An example: shadows are unrealistically deep and harsh, to soften these shadows we add a third light component called **ambient light**.

With only diffuse and specular reflections, any parts of a surface that are shadowed from the point source receive no light and so are drawn black but in real, the scenes around us are always in some soft nondirectional light. 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 $\mathbf{I_a}$. Each face in the model is assigned a value for its **ambient reflection coefficient** $\boldsymbol{\rho_d}$**,** and the term $\mathbf{I_a\ \rho_a}$ is added to the diffuse and specular light that is reaching the eye from each point P on that face. $\mathbf{I_a}$ and $\boldsymbol{\rho_a}$ are found experimentally.

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

4.1.5 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:

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

$$I = I_a \rho_a + I_d \rho_d \times \text{lambert} + I_{sp} \rho_s \times \text{phong}^f$$

Where we define the values

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

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

4.1.6 To Add Color

Colored light can be constructed by adding certain amounts of red, green and blue light. When dealing with colored sources and surfaces we calculate each color component individually and simply add them to from the final color of the reflected light.

$I_r = I_{ar}\,\rho_{ar} + I_{dr}\,\rho_{dr} \times \text{lambert} + I_{spr}\,\rho_{sr} \times \text{phong}^f$
$I_g = I_{ag}\,\rho_{ag} + I_{dg}\,\rho_{dg} \times \text{lambert} + I_{spg}\,\rho_{sg} \times \text{phong}^f$
$I_b = I_{ab}\,\rho_{ab} + I_{db}\,\rho_{db} \times \text{lambert} + I_{spb}\,\rho_{sb} \times \text{phong}^f$ --------------- (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_{ar},I_{ag},I_{ab}) , diffuse=(I_{dr},I_{dg},I_{db}) and specular=(I_{spr},I_{spg},I_{spb}). Usually the diffuse and the specular light colors are the same. The terms lambert and phong^f 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: ρ_{ar} , ρ_{ag} and ρ_{ab}
diffuse reflection coefficients: ρ_{dr} , ρ_{dg} and ρ_{db}
specular reflection coefficients: ρ_{sr} , ρ_{sg} and ρ_{sb}

The ambient and diffuse reflection coefficients are based on the color of the surface itself.

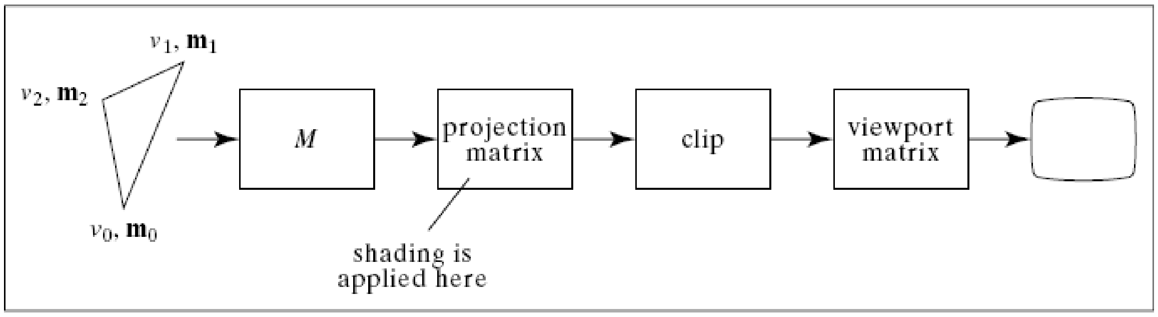
The Color of Specular Light

Specular light is mirrorlike , the color of the specular component is same as that of the light source.

Example: A specular highlight seen on a glossy red apple when illuminated by a yellow light is yellow and not red. This is the same for shiny objects made of plasticlike material.

To create specular highlights for a plastic surface the specular reflection coefficients ρ_{sr} , ρ_{sg} and ρ_{sb} are set to the same value so that the reflection coefficients are gray in nature and do not alter the color of the incident light.

4.1.7 Shading and the Graphics Pipeline



The key idea is that the vertices of a mesh are sent down the pipeline along with their associated vertex normals, and all shading calculations are done on vertices.

The above fig. shows a triangle with vertices v_0,v_1 and v_2 being rendered. Vertex v_i has the normal vector m_i associated with it. These quantities are sent down the pipeline with calls such as :

```
glBegin(GL_POLYGON);
        for( int i=0 ;i< 3; i++)
        {
            glNormal3f(m[i].x, m[i].y, m[i].z);
            glVertex3f(v[i].x, v[i].y, v[i].z);
        }
glEnd();
```

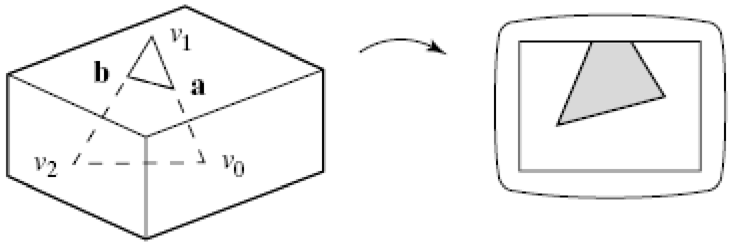
The call to glNormal3f() sets the “current normal vector” which is applied to all vertices sent using glVertex3f(). The current normal remains current until it is changed with another call to glNormal3f().

The vertices are transformed by the modelview matrix, M so they are then expressed in camera coordinates. The normal vectors are also transformed. Transforming points of a surface by a matrix M causes the normal **m** at any point to become the normal $\mathbf{M^{-T}m}$ on the transformed surface, where M^{-T} is the transpose of the inverse of M.

All quantities after the modelview transformation are expressed in camera coordinates. At this point the shading model equation (1) is applied and a color is attached to each vertex.

The clipping step is performed in homogenous coordinates. This may alter some of the vertices. The below figure shows the case where vertex v_1 of a triangle is clipped off and two new vertices a and b are created. The triangle becomes a quadrilateral. The color at each new vertices must be computed, since it is needed in the actual rendering step.

Clipping a polygon against the view volume



The vertices are finally passed through the viewport transformation where they are mapped into the screen coordinates. The quadrilateral is then rendered.

4.1.8 To Use Light Sources in OpenGL

OpenGL provides a number of functions for setting up and using light sources, as well as for specifying the surface properties of materials.

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

```
   GLfloat myLightPosition[]={3.0 , 6.0,5.0,1.0 };
   glLightfv(GL_LIGHT0, GL-POSITION, myLightPosition);
   glEnable(GL_LIGHTING);      //enable lighting in general
   glEnable(GL_LIGHT0);        //enable source GL_LIGHT0
```

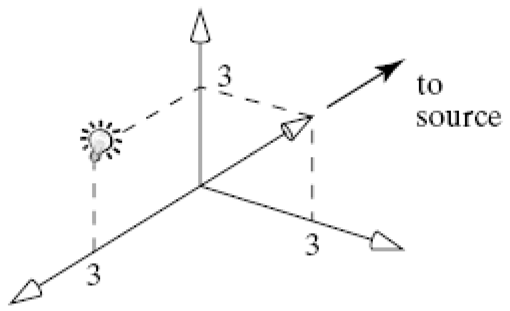
The array myLightPosition[] specifies the location of the light source. This position is passed to glLightfv() along with the name GL_LIGHT0 to attach it to the particular source GL_LIGHT0.

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:

```
   GLfloat amb0[]={ 0.2 , 0.4, 0.6, 1.0 };            // define some colors
   GLfloat diff0[]= { 0.8 ,0.9 , 0.5 ,1.0 };
   GLfloat spec0[]= { 1.0 , 0.8 , 1.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_AMBIENT, amb0); //attach them to LIGHT0
   glLightfv(GL_LIGHT0, GL_DIFFUSE, diff0);
   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);   dimmest possible :black

For light source LIGHT0:

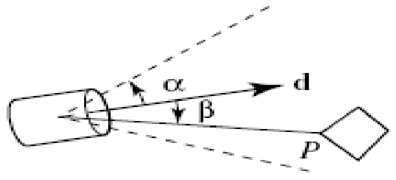
Default diffuse= (1,1,1,1)   brightest possible:white

Default specular=(1,1,1,1)   brightest possible:white

Spotlights

Light sources are point sources by default, meaning that they emit light uniformly in all directions. But OpenGL allows you to make them into spotlights, so they emit light in a restricted set of directions. The fig. shows a spotlight aimed in direction d with a "cutoff angle" of α.

Properties of an OpenGL spotlight



No light is seen at points lying outside the cutoff cone. For vertices such as P, which lie inside the cone, the amount of light reaching P is attenuated by the factor $\cos^{\varepsilon}(\beta)$, where β is the angle between d and a line from the source to P and is the exponent chosen by the user to give the desired falloff of light with angle.

The parameters for a spotlight are set by using **glLightf()** to set a single value and **glLightfv()** to set a vector:

```
 glLightf(GL_LIGHT0, GL_SPOT_CUTOFF,45.0); //a cutoff angle 45degree
 glLightf(GL_LIGHT0, GL_SPOT_EXPONENT,4.0);          //ε=4.0
GLfloat dir[]={2.0, 1.0, -4.0};          // the spotlight's direction
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION,dir);
```

The default values for these parameters are d= (0,0,-1) , α=180 degree and ε=0, which makes a source an omni directional point source.

OpenGL allows three parameters to be set that specify general rules for applying the lighting model. These parameters are passed to variations of the function **glLightModel**.

The color of global Ambient Light:

The global ambient light is independent of any particular source. To create this light , specify its color with the statements:

```
GLfloat amb[]={ 0.2, 0.3, 0.1, 1.0};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT,amb);
```

This code sets the ambient source to the color (0.2, 0.3, 0.1). The default value is (0.2, 0.2, 0.2,1.0) so the ambient is always present. Setting the ambient source to a non-zero value makes object in a scene visible even if you have not invoked any of the lighting functions.

Is the Viewpoint local or remote?

OpenGL computes specular reflection using halfway vector **h= s + v**. The true directions s and v are different at each vertex. If the light source is directional then s is constant but v varies from vertex to vertex. The rendering speed is increased if v is made constant for all vertices.

As a default OpenGL uses v =(0,0,1),which points along the positive z axis in camera coordinates. The true value of v can be computed by the following statement:

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
```

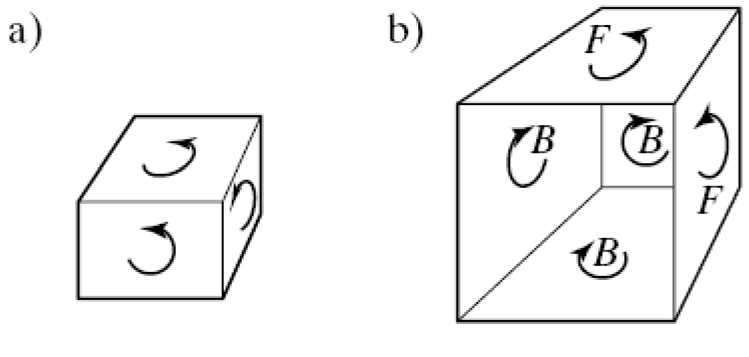
Are both sides of a Polygon Shaded Properly?

Each polygon faces in a model has two sides, inside and outside surfaces. The vertices of a face are listed in counterclockwise order as seen from outside the object. The camera can see only the outside surface of each face. With hidden surfaces removed, the inside surface of each face is hidden from the eye by some closer face.

In OpenGL the terms “front faces” and “back faces” are used for “inside” and “outside”. A face is a front face if its vertices are listed in counterclockwise order as seen by the eye.

The fig.(a) shows a eye viewing a cube which is modeled using the counterclockwise order notion. The arrows indicate the order in which the vertices are passed to OpenGL. For an object that encloses that some space, all faces that are visible to the eye are front faces, and OpenGL draws them with the correct shading. OpenGL also draws back faces but they are hidden by closer front faces.

OpenGL’s definition of a front face



Fig(b) shows a box with a face removed. Three of the visible faces are back faces. By default, OpenGL does not shade these properly. To do proper shading of back faces we use:

```
glLightModeli (GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```

When this statement is executed, OpenGL reverses the normal vectors of any back face so that they point towards the viewer, and then it performs shading computations properly. Replacing GL_TRUE with GL_FALSE will turn off this facility.

Moving Light Sources

Lights can be repositioned by suitable uses of **glRotated()** and **glTranslated()**. The array position, specified by using

glLightfv(GL_LIGHT0,GL_POSITION,position)

is modified by the modelview matrix that is in effect at the time glLightfv() is called. To modify the position of the light with transformations and independently move the camera as in the following code:

```
void display()
{
   GLfloat position[]={2,1,3,1};        //initial light position
   clear the color and depth buffers
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glPushMatrix();
               glRotated(….);              //move the light
               glTranslated(…);
               glLightfv(GL_LIGHT0,GL_POSITION,position);
    glPopMatrix();

    gluLookAt(….);                      //set the camera position
    draw the object
    glutSwapBuffers();
}
```

To move the light source with camera we use the following code:

```
GLfloat pos[]={0,0,0,1};
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glLightfv(GL_LIGHT0,GL_POSITION,position); //light at (0,0,0)
gluLookAt(….);                   //move the light and the camera
draw the object
```

This code establishes the light to be positoned at the eye and the light moves with the camera.

4.1.9 Working With Material Properties In OpenGL

The effect of a light source can be seen only when light reflects off an object's surface. OpenGL provides methods for specifying the various reflection coefficients. The coefficients are set with variations of the function glMaterial and they can be specified individually for front and back faces. The code:

```
Glfloat myDiffuse[]={0.8, 0.2, 0.0, 1.0 };
glMaterialfv(GL_FRONT,GL_DIFFUSE,myDiffuse);
```

sets the diffuse reflection coefficients(ρ_{dr} , ρ_{dg} ,ρ_{db}) equal to (0.8, 0.2, 0.0) for all specified front faces. The first parameter of glMaterialfv() can take the following values:

GL_FRONT:Set the reflection coefficient for front faces.
GL_BACK:Set the reflection coefficient for back faces.
GL_FRONT_AND_BACK:Set the reflection coefficient for both front and back faces.

The second parameter can take the following values:

GL_AMBIENT: Set the ambient reflection coefficients.
GL_DIFFUSE: Set the diffuse reflection coefficients.
GL_SPECULAR: Set the specular reflection coefficients.
GL_AMBIENT_AND_DIFFUSE: Set both the ambient and the diffuse reflection coefficients to the same values.
GL_EMISSION: Set the emissive color of the surface.

The emissive color of a face causes it to "glow" in the specified color, independently of any light source.

4.1.10 Shading of Scenes specified by SDL

The scene description language SDL supports the loading of material properties into objects so that they can be shaded properly.

```
light 3 4 5 .8 .8 ! bright white light at (3,4,5)
background 1 1 1 ! white background
globalAmbient .2 .2 .2 ! a dark gray global ambient light
ambient .2 .6 0
diffuse .8 .2 1 ! red material
specular 1 1 1 ! bright specular spots – the color of the source
specularExponent 20 !set the phong exponent
scale 4 4 4 sphere
```

The code above describes a scene containing a sphere with the following material properties:

- ambient reflection coefficients: (ρ_{ar} , ρ_{ag} , ρ_{ab})= (.2, 0.6, 0);
- diffuse reflection coefficients: (ρ_{dr} , ρ_{dg} , ρ_{db})= (0.8,0.2,1.0);
- specular reflection coefficients: (ρ_{sr} , ρ_{sg} , ρ_{sb}) = (1.0,1.0,1.0);
- Phong exponent f = 20.

The light source is given a color of (0.8,0.8,0.8) for both its diffuse and specular component. The global ambient term (I_{ar} , I_{ag} , I_{ab})= (0.2, 0.2, 0.2).

The current material properties are loaded into each object's **mtrl** field at the time the object is created.

When an object is drawn using drawOpenGL(), it first passes its material properties to OpenGL, so that at the moment the object is actually drawn, OpenGL has those properties in its current state.

4.2 FLAT SHADING AND SMOOTH SHADING

Different objects require different shading effects. In the modeling process we attached a normal vector to each vertex of each face. If a certain face is to appear as a distinct polygon, we attach the same normal vector to all of its vertices; the normal vector chosen is that indicating the direction normal to the plane of the face. If the face is approximate an underlying surface, we attach to each vertex the normal to the underlying surface at that plane.

The information obtained from the normal vector at each vertex is used to perform different kinds of shading. The main distinction is between a shading method that accentuates the individual polygons (**flat shading**) and a method that blends the faces to de-emphasize the edges between them (**smooth shading**).

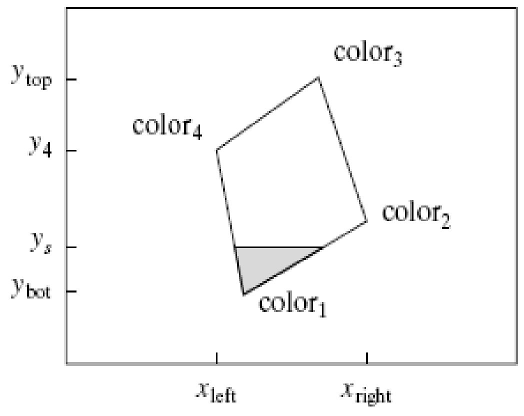
In both kinds of shading, the vertices are passed down the graphics pipeline, shading calculations are performed to attach a color to each vertex and the vertices are converted to screen coordinates and the face is "painted" pixel by pixel with the appropriate color.

Painting a Face

A face is colored using a polygon fill routine. A polygon routine is sometimes called as a **tiler** because it moves over a polygon pixel by pixel, coloring each pixel. The pixels in a polygon are visited in a regular order usually from bottom to top of the polygon and from left to right.

Polygons intersect are convex. A tiler designed to fill only convex polygons can be very efficient because at each scan line there is unbroken run of pixels that lie inside the polygon. OpenGL uses this property and always fills convex polygons correctly whereas nonconvex polygons are not filled correctly.

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_{bott} and y_{top}. The tiler first fills in the row at y= y_{bott} , then at y_{bott} + 1, etc. At each scan line y_s, there is a leftmost pixel x_{left} and a rightmost pixel x_{right}. The toler moves from x_{left} to x_{right}, placing the desired color in each pixel. The tiler is implemented as a simple double loop:

```
for (int y= $y_{bott}$ ; y<= $y_{top}$; y++)  // for each scan line
 {
     find $x_{left}$ and $x_{right}$
     for( int x= $x_{left}$ ; x<= $x_{right}$; 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.

4.2.1 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. In this mode, although a color is passed down the pipeline as part of each vertex of the face, the painting algorithm uses only one color value. So the command **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);**

When objects are rendered using flat shading. The individual faces are clearly visible on both sides. 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**. When there is a discontinuity across an object the eye manufactures a **Mach Band** at the discontinuity and a vivid edge is seen.

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

4.2.2 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

$$color_{left} = lerp(color_1, color_4, f), \qquad ----------(1)$$

where the fraction

$$f = \frac{y_5 - y_{bott}}{y_4 - y_{bott}}$$

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:

$$C(x) = lerp\left(color_{left}, color_{right}, x - x_{left} | x_{right} - x_{left}\right)$$

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= $y_{bott}$; y<=$y_{top}$ ; y++)          //for each scan line
{
    find $x_{left}$ and $x_{right}$
    find $color_{left}$ and $color_{right}$
    $color_{inc}$=( $color_{right}$ - $color_{left}$) / ($x_{right}$ - $x_{left}$);
    for(int x= $x_{left}$, c=$color_{left}$; x<=$x_{right}$; x++, c+=$color_{inc}$)
    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);

When a sphere and a bucky ball are rendered using Gouraud shading, the bucky ball looks the same as it was rendered with flat shading because the same color is associated with each vertex of a face. But the sphere looks smoother, as there are no abrupt jumps in color between the neighboring faces and the edges of the faces are gone , replaced by a smoothly varying colors across the object.

Continuity of color across a polygonal edge

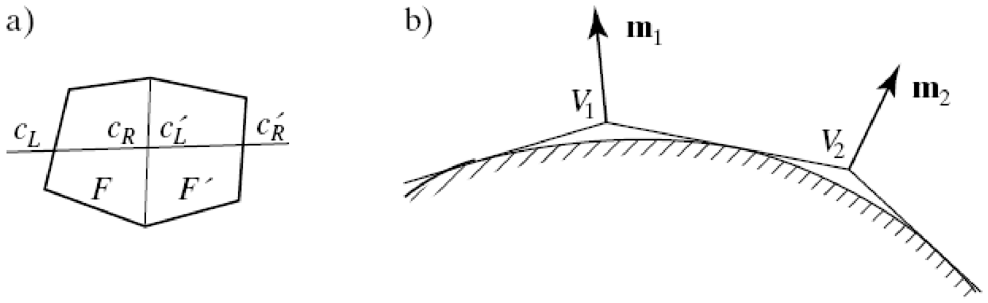


Fig.(a) shows two faces F and F' that share an edge. In rendering F, the colors C_L and C_R are used and in rendering F', the colors C'_L and C'_R are used. But since C_R equals C'_L, there is no abrupt change in color at the edge along the scan line.

Fig.(b) shows how Gouraud shading reveals the underlying surface. The polygonal surface is shown in cross section with vertices V_1 and V_2. The imaginary smooth surface is also represented. Properly computed vertex normals m_1,m_2 point perpendicularly to this imaginary surface so that the normal for correct shading will be used at each vertex and the color there by found will be correct. The color is then made to vary smoothly between the vertices.

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

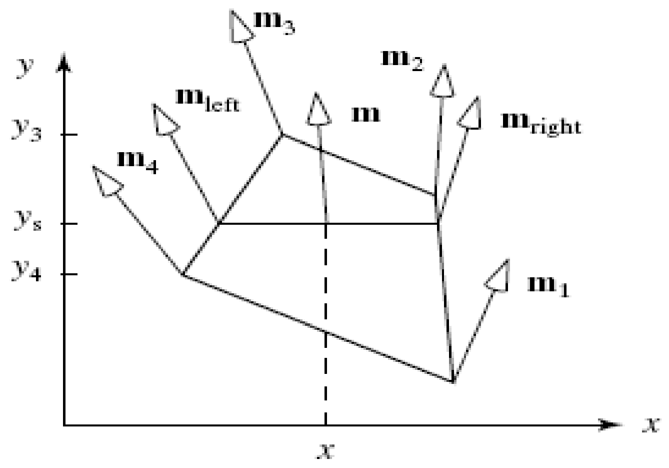
Phong Shading

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 find the normal vector at each point on the face of the object and we apply the shading model there to fig the color 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 *m1,* m2, m3 and m4 indicated at the four vertices.

Interpolating normals



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

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

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

4.3 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:

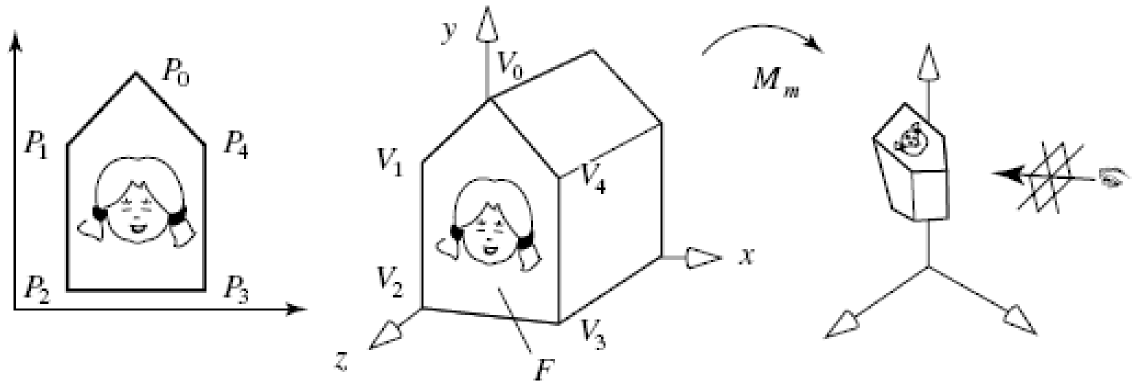
```
float fakesphere( float s, float t)
{
  float r= sqrt((s-0.5) * (s-0.5)+ (t-0.5) * (t-0.5));
  if (r < 0.3) return 1-r/0.3;      //sphere intensity
  else return 0.2;                  //dark background
}
```

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

4.3.1 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



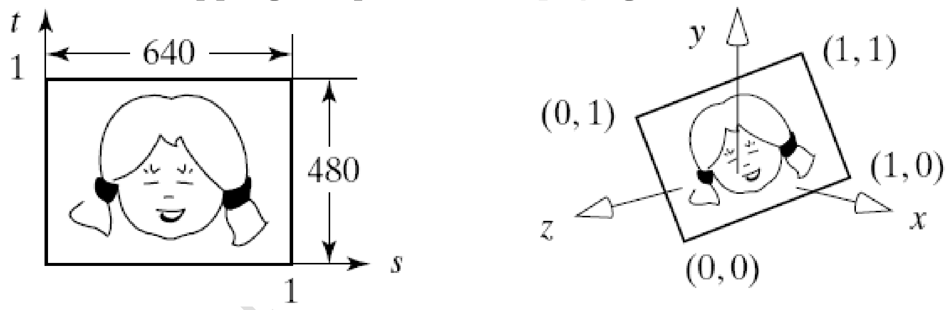
The fig. shows a texture image mapped to a portion of a planar polygon,F. We need to specify how to associate points on the texture with points on F.

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:

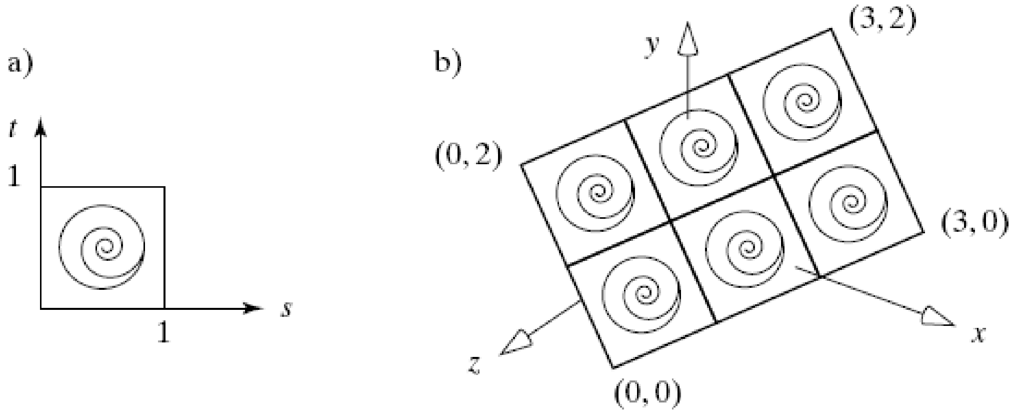
```
 glBegin(GL_QUADS);                //defines a quadrilateral face
        glTexCoord2f(0.0 ,0.0); glVertex3f(1.0 , 2.5, 1.5);
        glTexCoord2f(0.0 ,0.6); glVertex3f(1.0 , 3.7, 1.5);
        glTexCoord2f(0.8 ,0.6); glVertex3f(2.0 , 3.7, 1.5);
        glTexCoord2f(0.8 ,0.0); glVertex3f(2.0 , 2.5, 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.

Adding Texture Coordinates to Mesh Objects

A mesh objects has three lists

- The vertex list
- The normal vector list
- The face list

We need to add texture coordinate to this list, which stores the coordinates (s_i, t_i) to be associated with various vertices. We can add an array of elements of the type

class TxtrCoord(public : float s,t;);

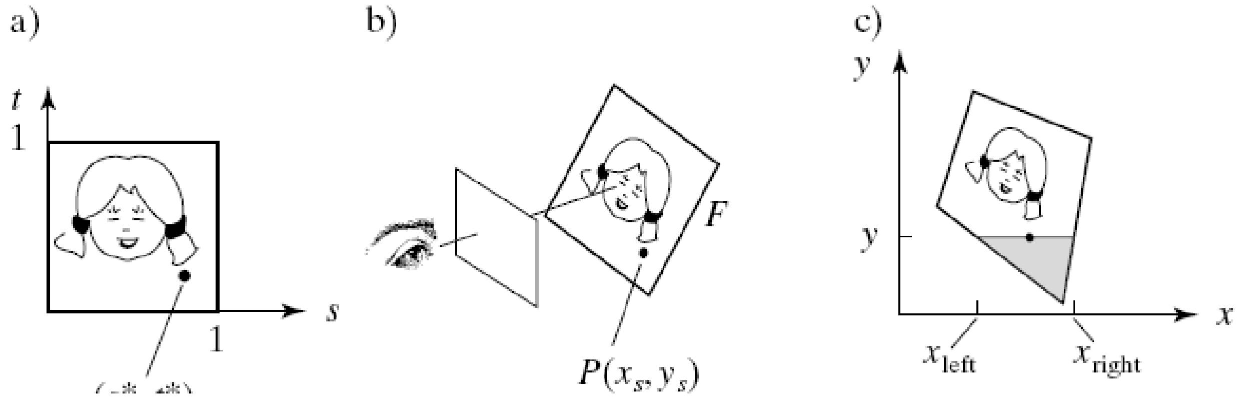
to hold all of the coordinate pairs of the mesh. The two important techniques to treat texture for an object are:

1. The mesh object consists of a small number of flat faces, and a different texture is to be applied to each. Each face has only a sigle normal vector, but its own list of texture coordinates. So the following data are associated with each face:
   - the number of vertices in the face.
   - the index of normal vector to the face.
   - a list of indices of the vertices.
   - a list of indices of the texture coordinates.
2. The mesh represents a smooth underlying object and a single texture is to wrapped around it. Each vertex has associated with it a specific normal vector and a particular texture coordinate pair. A single index into the vertex, normal vector and texture lists is used for each vertex. The data associated with the face are:
   - the number of vertices in the face.
   - list of indices of the vertices.

4.3.2 Rendering the Texture

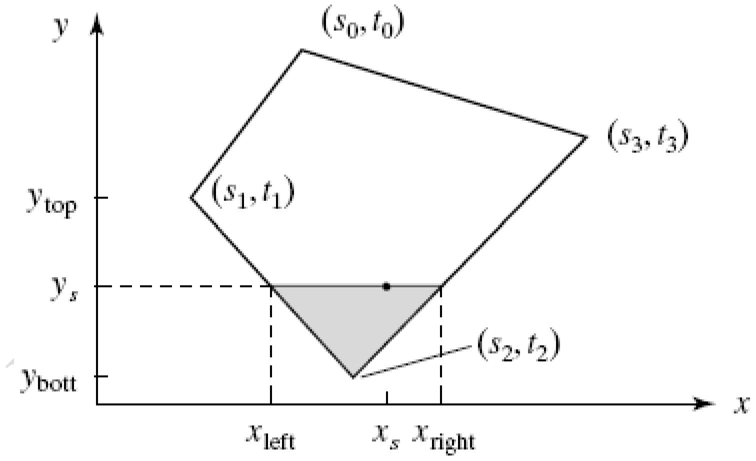
Rendering texture in a face F is similar to Gouraud Shading. It proceeds across the face pixel by pixel. For each pixel it must determine the corresponding texture coordinates (s,t), access the texture and set the pixel to the proper texture color. Finding the coordinated (s,t) should be done carefully.

Rendering a face in a camera snapshot



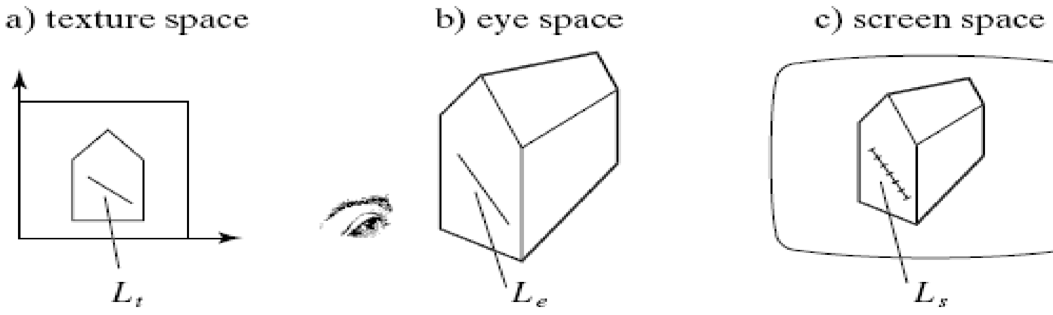
The fig shows the camera taking a snapshot of a face F with texture pasted onto it and the rendering in progress. The scan line y is being filled from x_{left} to x_{right}. For each x along this scan line, we compute the correct position on the face and from that , obtain the correct position (s^*,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

Incremental calculation of texture coordinates



We compute (s_{left},t_{left}) and (s_{right},t_{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.

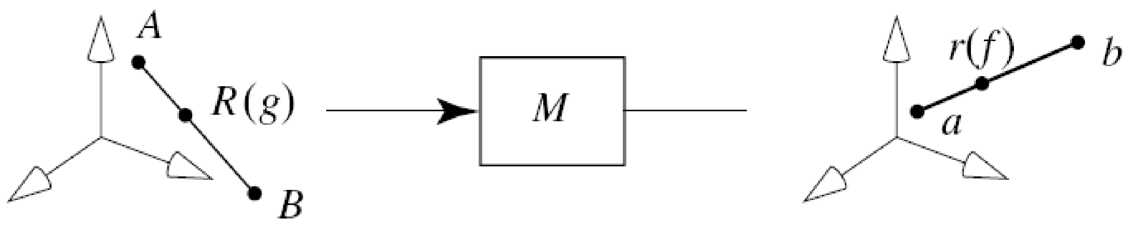
Lines in one space map to lines in another



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?

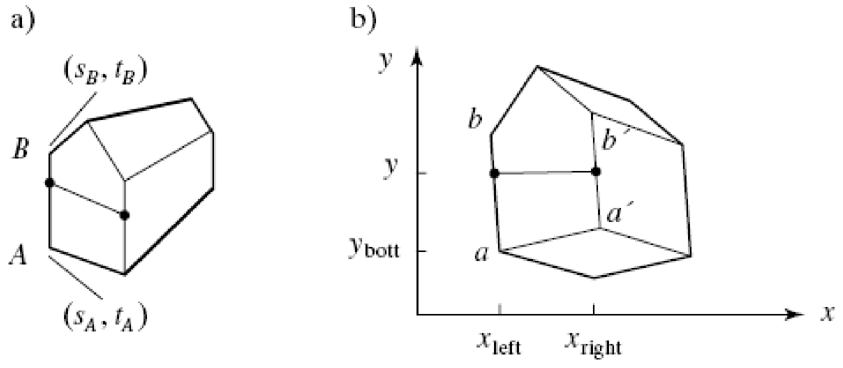


The fig. shows a line AB in 3D being transformed into the line ab in 3D by the matrix M. A maps to a, B maps to b. Consider the point R(g) that lies a fraction g of the way between A and B. This point maps to some point r(f) that lies a fraction f of the way from a to b. The fractions f and g are not 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?

Rendering Images Incrementally

We now find the proper texture coordinates (s,t) at each point on the face being rendered.

Rendering the texture on a face



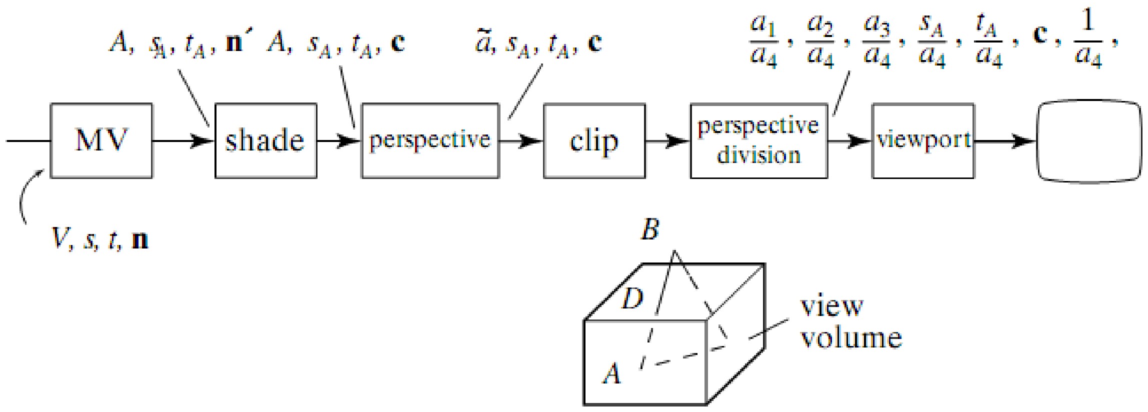
The fig. shows the face of a barn. The left edge of the projected face has endpoints a and b. The face extends from x_{left} to x_{right} across scan line y. We need to find appropriate texture coordinates (s_{left}, t_{left}) and (s_{right}, t_{right}) to attach to x_{left} and x_{right}, which we can then interpolate across the scan line

Consider finding $s_{left}(y)$, the value of s_{left} at scan line y.We know that texture coordinate s_A is attached to point a and s_B is attached to point b. If the scan line at y is a fraction f of the way between y_{bott} and y_{top} so that f=(y – y_{bott})/ (y_{top} – y_{bott}), the proper texture coordinate to use is

$$s_{left}(y) = \left(lerp\left(\frac{s_A}{a_4},\frac{s_B}{b_4},f\right)\right| lerp\left(\frac{1}{a_4},\frac{1}{b_4},f\right)$$

and similarly for t_{left}.

Implications for the Graphics Pipeline


$A, s_A, t_A, \mathbf{n}'$
$A, s_A, t_A, \mathbf{c}$
$\tilde{a}, s_A, t_A, \mathbf{c}$
$\frac{a_1}{a_4}, \frac{a_2}{a_4}, \frac{a_3}{a_4}, \frac{s_A}{a_4}, \frac{t_A}{a_4}, \mathbf{c}, \frac{1}{a_4},$
MV
shade
perspective
clip
perspective division
viewport
$V, s, t, \mathbf{n}$
B
D
A
view volume

The figure shows a refinement of the pipeline. Each vertex V is associated with a texture pair (s,t) and a vertex normal. The vertex is transformed by the modelview matrix, producing vertex A=(A_1, A_2, A_3) and a normal n' in eye coordinates.

Shading calculations are done using this normal, producing the color c=(c_r, c_g, c_b). The texture coordinates (s_A, t_A) are attached to A. Vertex A then goes perspective transformation, producing a =(a_1,a_2, a_3,a_4). The texture coordinates and color c are not altered.

Next clipping against the view volume is done. Clipping can cause some vertices to disappear and some vertices to be formed. When a vertex D is created, we determine its position (d_1, d_2, d_3, d_4) and attach it to appropriate color and texture point. After clipping the face still consists of a number of verices, to each of which is attached a color and a texture point. For a point A, the information is stored in the array (a_1, a_2, a_3, a_4, s_A, t_A, c,1). A final term of 1 has been appended; this is used in the next step.

Perspective division is done, we need hyberbolic interpolation so we divide every term in the array that we wish to interpolate hyperbolically by a_4, to obtain the array $(x, y, z, 1, s_A/a_4, t_4/a_4, c, 1/a_4)$. The first three components of the array $(x, y, z)=(a_1/a_4, a_2/a_4, a_3/a_4)$.

Finally, the rendering routine receives the array $(x, y, z, 1, s_A/a_4, t_4/a_4, c, 1/a_4)$ for each vertex of the face to be rendered.

4.3.3 What does the texture Modulate?

There are three methods to apply the values in the texture map in the rendering calculations

Creating a Glowing Object

This is the simplest method. The visibility intensity I is set equal to the texture value at each spot:

I=texture(s,t)

The object then appears to emit light or glow. Lower texture values emit less light and higher texture values emit more light. No additional lighting calculations are needed. OpenGL does this type of texturing using

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
          GL_REPLACE);
```

Painting the Texture by Modulating the Reflection Coefficient

The color of an object is the color of its diffuse light component. Therefore we can make the texture appear to be painted onto the surface by varying the diffuse reflection coefficient. The texture function modulates the value of the reflection coefficient from point to point. We replace eq(1) with

$I = \text{texture}(s,t)\,[I_a \rho_a + I_d \rho_d \times \text{lambert}] + I_{sp} \rho_s \times \text{phong}^f$

For appropriate values of s and t. Phong specular reflections are the color of the source and not the object so highlights do not depend on the texture. OpenGL does this type of texturing using

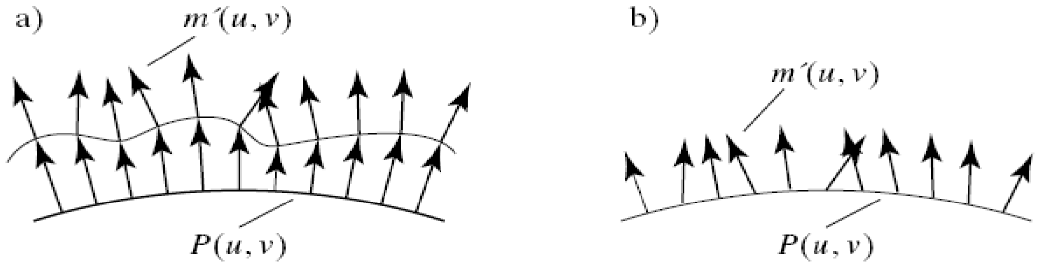
```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
           GL_MODULATE);
```

Simulating Roughness by Bump Mapping

Bump mapping is a technique developed by Blinn, to give a surface a wrinkled or dimpled appearance without struggling to model each dimple itself. One problem associated with applying bump mapping to a surface like a teapot is that since the model does not contain the dimples , the object's outline caused by a shadow does not show dimples and it is smooth along each face.

The goal is to make a scalar function texture(s,t) disturb the normal vector at each spot in a controlled fashion. This disturbance should depend only on the shape of the surface and the texture.

On the nature of bump mapping



The fig. shows in cross section how bump mapping works. Suppose the surface is represented parametrically by the function P(u,v) and has unit normal vector m(u,v). Suppose further that the 3D point at(u*,v*) corresponds to texture at (u*,v*).

Blinn's method simulates perturbing the position of the true surface in the direction of the normal vector by an amount proportional to the texture (u*,v*);that is

P'(u*,V*) = P(u*,v*)+texture(u*,v*)m(u*,v*).

Figure(a) shows how this techniques adds wrinkles to the surface. The disturbed surface has a new normal vector m'(u*,v*)at each point. The idea is to use this disturbed normal as if it were "attached" to the original undisturbed surface at each point, as shown in figure (b). Blinn has demonstrated that a good approximation to m'(u*,v*) is given by

m'(u*,v*) =m(u*,v*)+d(u*,v*)

Where the perturbation vector d is given by

d(u*,v*) = (m X p_v) $texture_u$ – (m X p_u) $texture_v$.

In which $texture_u$, and $texture_v$ are partial derivatives of the texture function with respect to u and v respectively. Further p_u and p_v are partial derivative of P(u,v) with respect to u and v, respectively. all functions are evaluated at(u*,V*).Note that the perturbation function depends only on the partial derivatives of the texture(),not on texture()itself.

4.3.4 Reflection Mapping

This technique is used to improve the realism of pictures , particularly animations. The basic idea is to see reflections in an object that suggest the world surrounding that object.

The two types of reflection mapping are

- Chrome mapping

  A rough and blurry image that suggests the surrounding environment is reflected in the object as you would see in an object coated with chrome.
- Environment mapping

  A recognizable image of the surrounding environment is seen reflected in the object. Valuable visual clues are got from such reflections particularly when the object is moving.

4.4 ADDING SHADOWS OF OBJECTS

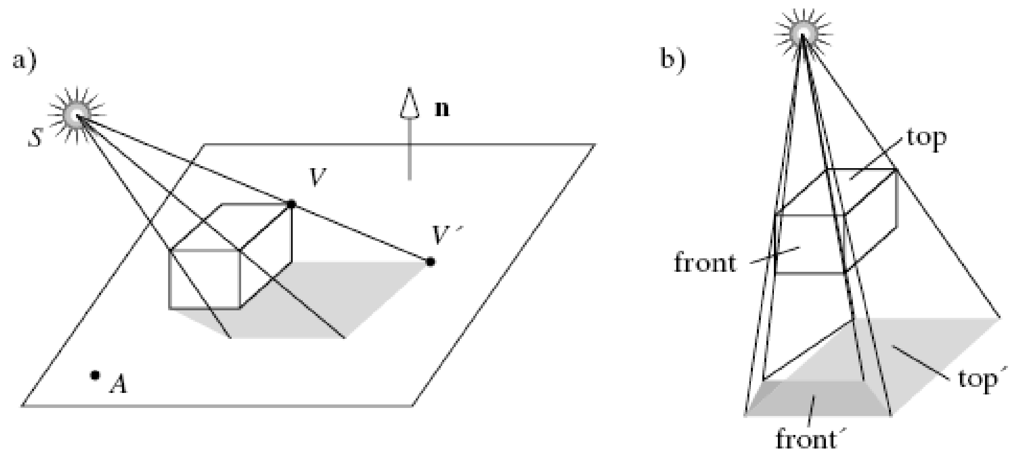
Shadows make an image more realistic. The way one object casts a shadow on another object gives important visual clues as to how the two objects are positioned with respect to each other. Shadows conveys lot of information as such, you are getting a second look at the object from the view point of the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

- Shadows as Texture
- Creating shadows with the use of a shadow buffer

4.4.1 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 faces 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

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

4.4.2 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 lot of memory.

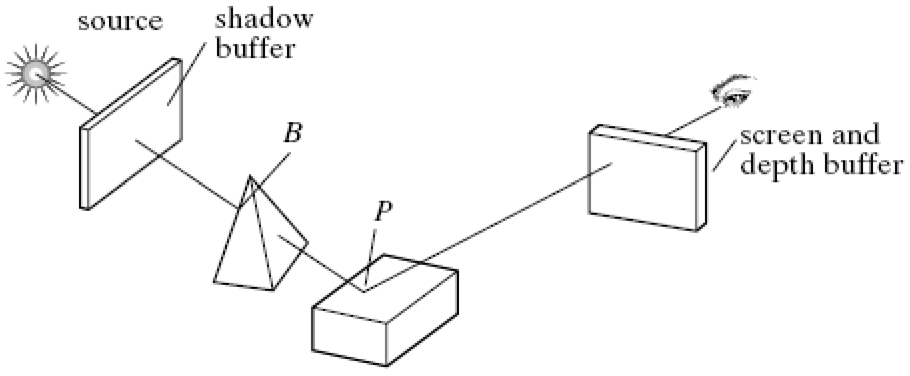
This method is based on the principle that any points in a scene that are hidden from the light source must be in shadow. If no object lies between a point and the light source, the point is not in shadow.

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.

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 the 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.

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
```

The Camera class definition:

```
class Camera {
      private:
        Point3 eye;
        Vector3 u, v, n;
        double viewAngle, aspect, nearDist, farDist; //view volume shape
        void setModelViewMatrix();   //tell OpenGL where the camera is

public:
        Camera();                        //default constructor
        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 it
        void setShape(float vAng, float asp, float nearD, float farD);
};
```

The Camera class definition contains fields for eye and the directions u, v and n. Point3 and Vector3 are the basic data types. It also has fields that describe the shape of the view volume: viewAngle, aspect, nearDist and farDist.

The utility routine setModelViewMatrix() communicates the modelview matrix to OpenGL. 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

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

This matrix V accounts for the transformation of world points into camera coordinates. 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().

The utility routines set() and setModelViewMatrix()

```
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]= y.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
}
void Camera :: set (Point3 eye, Point3 look, Vector3 up)
{ // Create a modelview matrix and send it to OpenGL
          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.set(up.cross(n));                        //make u= up X n
          n.normalize();                             // make them  unit length
          u.normalize();
          v.set(n.cross(u));                         // make v= n X u
          setModelViewMatrix();                      // tell OpenGL
}
```

The method set() acts like gluLookAt(): It uses the values of eye, look and up to compute u, v and n according to equation:

n= eye – look,

u = up X n

and

v = n X u. It places this information in the camera's fields and communicates it to OpenGL.

The routine setShape() is simple. It puts the four argument values into the appropriate camera fields and then calls

```
gluPerspective(viewangle, aspect, nearDist, farDist)
```

along with

```
glMatrixMode(GL_PROJECTION)
```

and

```
glLoadIdentity()
```

to set the projection matrix.

The central camera functions are slide(), roll(), yaw() and pitch(), which makes relative changes to the camera's position and orientation.

4.5.1 Flying the camera

The user flies the camera through a scene interactively by pressing keys or clicking the mouse. For instance,

- pressing u will slide the camera up some amount
- pressing y will yaw the camera to the left
- pressing f will slide the camera forward

The user can see different views of the scene and then changes the camera to a better view and produce a picture. Or the user can fly around a scene taking different snapshots. If the snapshots are stored and then played back, an animation is produced of the camera flying around the scene.

There are six degrees of freedom for adjusting a camera: It can be slid in three dimensions and it can be rotated about any of three coordinate axes.

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:

```
void Camera : : 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

Two new axes are formed u' and v' that lie in the same plane as u and v, and have been rotated through the angle ɑ radians.

We form u' as the appropriate linear combination of u and v and similarly for v':

u' = cos (ɑ)u + sin(ɑ)v ;
v' = -sin (ɑ)u + cos(ɑ)v

The new axes u' and v' then replace u and v respectively in the camera. The angles are measured in degrees.

Implementation of roll()

```
void Camera :: 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 Camera :: 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 Camera :: 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();
}
```

The Camera class can be used with OpenGL to fly a camera through a scene. The scene consists of only a teapot. The camera is a global object and is set up in main(). When a key is pressed **myKeyboard()** is called and the camera is slid or rotated, depending on which key was pressed.

For instance, if P is pressed, the camera is pitched up by 1 degree. If CTRL F is pressed , the camera is pitched down by 1 degree. After the keystroke has been processed, **glutPostRedisplay()** causes **myDisplay()** to be called again to draw the new picture.

This application uses double buffering to produce a fast and smooth transition between one picture and the next. Two memory buffers are used to store the pictures that are generated. The display switches from showing one buffer to showing the other under the control of **glutSwapBuffers()**.

Application to fly a camera around a teapot

```
#include "camera.h"
Camera cam;                //global camera object
//--------------------- myKeyboard------------------------------
void myKeyboard(unsigned char key, int x, int y)
{
   switch(key)
      {
          //controls for the camera
             case 'F':                          //slide camera forward
                       cam.slide(0, 0, 0.2);
                       break;
            case 'F'-64:                        //slide camera back
                       cam.slide(0, 0,-0.2);
                       break;
            case 'P':
                       cam.pitch(-1.0);
                       break;
            case ‘P’-64:
                        cam.pitch(1.0);
                        break;
          //add roll and yaw controls
          }
                glutPostRedisplay();                //draw it again
}
//-------------------------myDisplay-----------------------------
void myDisplay(void)
{
          glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
          glutWireTeapot(1,0);             // draw the teapot
          glFlush();
          glutSwapBuffers();               //display the screen just made
}
//-------------------------main---------------------------
void main(int argc, char  **argv)
{
   glutInit(&argc, argv);
 glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);  //double buffering
 glutInitWindowSize(640, 480);
 glutInitWindowPosition(50, 50);
 glutCreateWindow(“fly a camera around a teapot”);
 glutKeyboardFunc(myKeyboard);
 glutDisplayFunc(myDisplay);
 glClearColor(1.0f, 1.0f, 1.0f, 1.0f);                    //background is white
 glColor3f(0.0f, 0.0f, 0.0f);                            //set color of stuff
 glViewport(0, 0, 640, 480);
 cam.set(4, 4, 4, 0, 0, 0, 0, 1, 0);                //make the initial camera
 cam.setShape(30.0f, 64.0f/48.0f, 0.5f, 50.0f);
 glutMainLoop();
}
```