CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

Scheme of Evaluation
Internal Assessment Test 3 – May.2019

Sub:	Computer Graphics and Visualization							**Code:**	15CS62
Date:	13/05/2019	Duration:	90mins	Max Marks:	50	**Sem:**	VI	**Branch:**	CSE

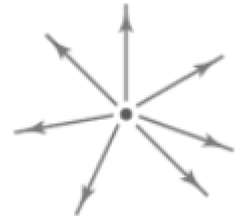
Note: Answer Any Five Questions

Question #	Marks Distribution	Max Marks
1. Explain different light sources? Define viewport and explain function to define viewport?	Light sources: 6Marks Viewport : 4Marks	10M
2. Explain normalization transformation for orthogonal projection with neat diagram?	Diagram : 4 Marks Formula & Explanation : 6 Marks	10M
3. Derive perspective projection transformation coordinates and represent in matrix format?	Diagram : 1 Mark Derivation: 7 Marks Matrix form : 2 Marks	10M
4. Write and explain depth buffer algorithm?	Algorithm : 5 Marks Explanation : 5 Marks	10M
5. Write a program to animate a flag using Bezier Curve algorithm.	Main : 2 Marks Display : 2Marks Bazier : 6Marks	10M
6. Explain basic illumination model? Explain Phong model.	Basic Illumination model: 3 Marks Phong Model : 5 Marks Diagrams : 2Marks	10M
7. Write an interactive program to rotate a square.	Main : 2 Marks Display : 2Marks Idle Function: 2 Makrs Rotate code: 4 Marks	10M
8. Write a note on logical operations. Explain XOR operation in detail.	Logical Op with diagram: 5 marks XOR Op: 5 marks	10M

SOLUTIONS

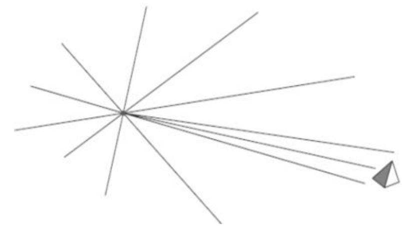
1. Explain different light sources? Define viewport and explain function to define viewport?

Point Light Sources:



- The simplest model for an object that is emitting radiant energy is a point light source with a single color, specified with three RGB components.
- A point source for a scene by giving its position and the color of the emitted light. Light rays are generated along radially diverging paths from the single-color source position. This light-source model is a reasonable approximation for sources whose dimensions are small compared to the size of objects in the scene.

Infinitely Distant Light Sources:



- A large light source, such as the sun, that is very far from a scene can also be approximated as a point emitter, but there is little variation in its directional effects.
- The light path from a distant light source to any position in the scene is nearly constant.
- We can simulate an infinitely distant light source by assigning it a color value and a fixed direction for the light rays emanating from the source.
- The vector for the emission direction and the light-source color are needed in the illumination calculations, but not the position of the source.

Radial Intensity Attenuation:

- As radiant energy from a light source travels outwards through space, its amplitude at any distance d_l from the source is attenuated by the factor $1/d^2$ a surface close to the light source receives a higher incident light intensity from that source than a more distant surface. However, using an attenuation factor of $1/d_l^2$ with a point source does not always produce realistic pictures. The factor $1/d_l^2$ tends to produce too much intensity variation for objects that are close to the light source, and very little variation when d_l is large.
- We can attenuate light intensities with an inverse quadratic function of d_l that includes a linear term:

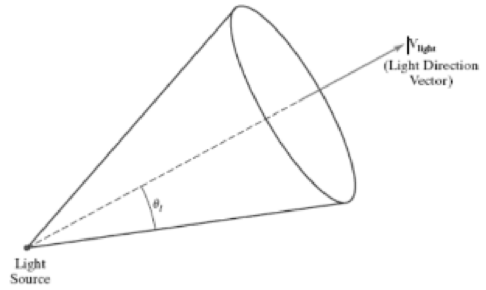
$$f_{\text{radatten}}(d_l) = \frac{1}{a_0 + a_1 d_l + a_2 d_l^2}$$

- The numerical values for the coefficients, a0, a1, and a2, can then be adjusted to produce optimal attenuation effects.
- We cannot apply intensity-attenuation calculation 1 to a point source at “infinity,” because the distance to the light source is indeterminate. We can express the intensity-attenuation function as

$$f_{l,\text{radatten}} = \begin{cases} 1.0, & \text{if source is at infinity} \\ \dfrac{1}{a_0 + a_1 d_l + a_2 d_l^2}, & \text{if source is local} \end{cases}$$

Directional Light Sources and Spotlight Effects:

- A local light source can be modified easily to produce a directional, or spotlight, beam of light.
- If an object is outside the directional limits of the light source, we exclude it from illumination by that source. One way to set up a directional light source is to assign it a vector direction and an angular limit θ_l measured from that vector direction, in addition to its position and color.

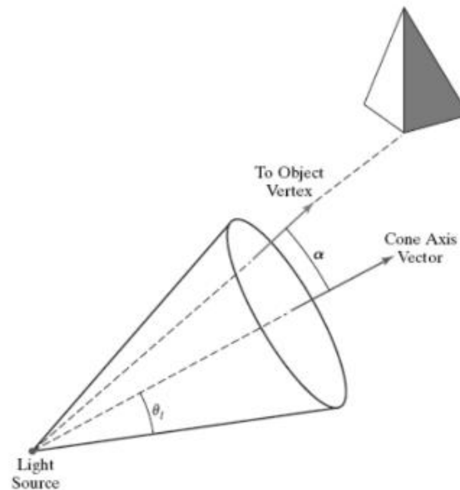


- We can denote Vlight as the unit vector in the light-source direction and Vobj as the unit vector in the direction from the light position to an object position.

$$\text{Then } V_{obj} \cdot V_{light} = \cos\alpha$$

where angle α is the angular distance of the object from the light direction vector.

- If we restrict the angular extent of any light cone so that $0^\circ < \theta_l \leq 90^\circ$, then the object is within the spotlight if $\cos\alpha \geq \cos\theta_l$, as shown



- If $V_{obj} \cdot V_{light} < \cos\theta_l$, however, the object is outside the light cone.

Angular Intensity Attenuation:

- For a directional light source, we can attenuate the light intensity angularly about the source as well as radially out from the point-source position
- This allows intensity decreasing as we move farther from the cone axis.
- A commonly used angular intensity-attenuation function for a directional light source is

$$f_{\text{angatten}}(\phi) = \cos^{a_l}\phi, \qquad 0^\circ \leq \phi \leq \theta$$

- Where the attenuation exponent al is assigned some positive value and angle φ is measured from the cone axis. The greater the value for the attenuation exponent al, the smaller the value of the angular intensity-attenuation function for a given value of angle $\varphi > 0^\circ$.
- There is no angular attenuation if the light source is not directional (not a spotlight).
- We can express the general equation for angular attenuation as

$$f_{l,\text{angatten}} = \begin{cases} 1.0, & \text{if source is not a spotlight} \\ 0.0, & \text{if } \mathbf{V}_{\text{obj}} \cdot \mathbf{V}_{\text{light}} = \cos\alpha < \cos\theta_l \\ & \quad \text{(object is outside the spotlight cone)} \\ (\mathbf{V}_{\text{obj}} \cdot \mathbf{V}_{\text{light}})^{a_l}, & \text{otherwise} \end{cases}$$

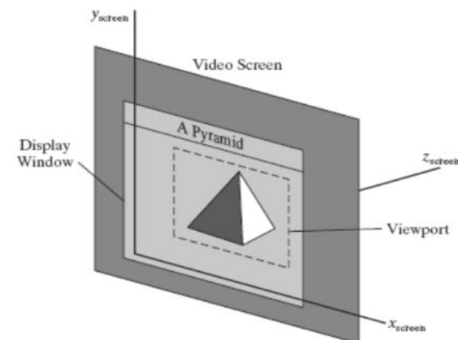
Viewport:

Graphics packages allow us also to control the placement within the display window using another "window" called the viewport. Objects inside the clipping window are mapped to the viewport, and it is the viewport that is then positioned within the display window. The clipping window selects what we want to see; the viewport indicates where it is to be viewed on the output device. By changing the position of a viewport, we can view objects at different positions on the display area of an output device. Viewport is defined using following function:

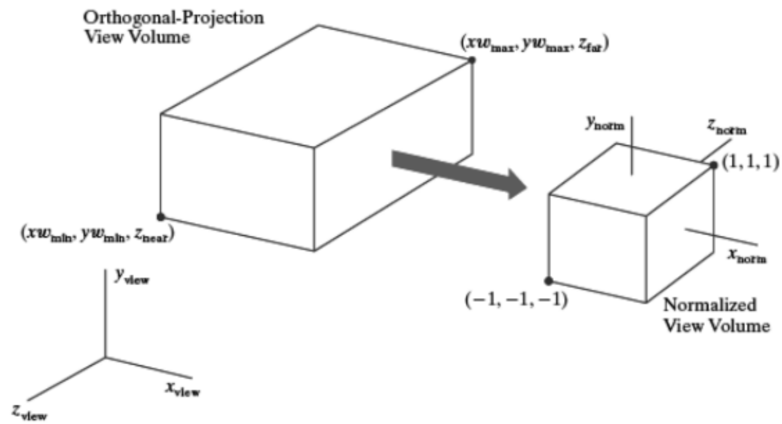
glViewport (xvmin, yvmin, vpWidth, vpHeight);

The first two parameters in this function specify the integer screen position of the lower left corner of the viewport relative to the lower-left corner of the display window. And the last two parameters give the integer width and height of the viewport. To maintain the proportions of objects in a scene, we set the aspect ratio of the viewport equal to the aspect ratio of the clipping window.

2. Explain normalization transformation for orthogonal projection with neat diagram?



- Once we have established the limits for the view volume, coordinate descriptions inside this rectangular parallelepiped are the projection coordinates, and they can be mapped into a normalized view volume without any further projection processing.
- Some graphics packages use a unit cube for this normalized view volume, with each of the x, y, and z coordinates normalized in the range from 0 to 1.
- Another normalization-transformation approach is to use a symmetric cube, with coordinates in the range from −1 to 1.
- We can convert projection coordinates into positions within a left-handed normalized coordinate reference frame, and these coordinate positions will then be transferred to left handed screen coordinates by the viewport transformation.
- To illustrate the normalization transformation, we assume that the orthogonal-projection view volume is to be mapped into the symmetric normalization cube within a left-handed reference frame.
- Also, z-coordinate positions for the near and far planes are denoted as znear and zfar, respectively. Figure below illustrates this normalization transformation. Position (xmin, ymin, znear) is mapped to the normalized position (−1, −1, −1), and position (xmax, ymax, zfar) is mapped to (1, 1, 1).

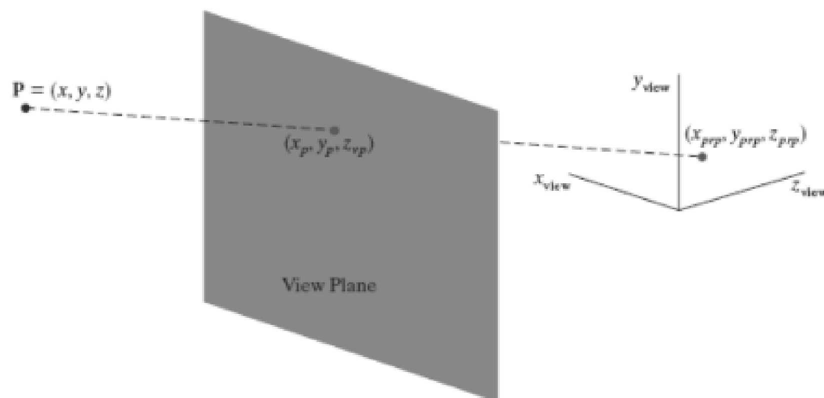


- The normalization transformation for the orthogonal view volume is

$$\mathbf{M}_{ortho,norm} = \begin{bmatrix} \frac{2}{xw_{max} - xw_{min}} & 0 & 0 & -\frac{xw_{max} + xw_{min}}{xw_{max} - xw_{min}} \\ 0 & \frac{2}{yw_{max} - yw_{min}} & 0 & -\frac{yw_{max} + yw_{min}}{yw_{max} - yw_{min}} \\ 0 & 0 & \frac{-2}{z_{near} - z_{far}} & \frac{z_{near} + z_{far}}{z_{near} - z_{far}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3. Derive perspective projection transformation coordinates and represent in matrix format?

Figure below shows the projection path of a spatial position (x, y, z) to a generalprojection reference point at (xprp, yprp, zprp).



- The projection line intersects the view plane at the coordinate position (xp, yp, zvp), where Zvp is some selected position for the view plane on the Zview axis.
- We can write equations describing coordinate positions along this perspective-projection line in parametric form as

$$\begin{aligned} x' &= x - (x - x_{prp})u \\ y' &= y - (y - y_{prp})u \qquad 0 \leq u \leq 1 \\ z' &= z - (z - z_{prp})u \end{aligned}$$

- On the view plane, z' = Zvp and we can solve the z' equation for parameter u at this position along the projection line:

$$u = \frac{z_{vp} - z}{z_{prp} - z}$$

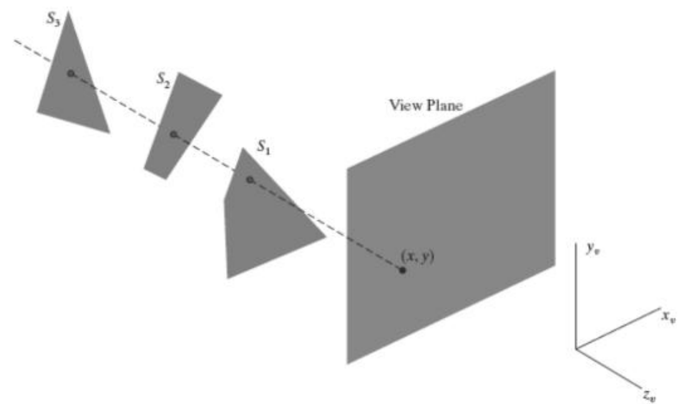
- Substituting this value of u into the equations for x' and y', we obtain the general perspective-transformation equations

$$x_p = x\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) + x_{prp}\left(\frac{z_{vp} - z}{z_{prp} - z}\right)$$

$$y_p = y\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) + y_{prp}\left(\frac{z_{vp} - z}{z_{prp} - z}\right)$$

4. Write and explain depth buffer algorithm?

- A commonly used image-space approach for detecting visible surfaces is the depth-buffer method, which compares surface depth values throughout a scene for each pixel position on the projection plane.
- The algorithm is usually applied to scenes containing only polygon surfaces, because depth values can be computed very quickly and the method is easy to implement.
- This visibility-detection approach is also frequently alluded to as the z-buffer method, because object depth is usually measured along the z axis of a viewing system.



- The figure here shows three surfaces at varying distances along the orthographic projection line from position (x, y) on a view plane.
- These surfaces can be processed in any order.
- If a surface is closer than any previously processed surfaces, its surface color is calculated and saved, along with its depth.
- The visible surfaces in a scene are represented by the set of surface colors that have been saved after all surface processing is completed.
- As implied by the name of this method, two buffer areas are required. A depth buffer is used to store depth values for each (x, y) position as surfaces are processed, and the frame buffer stores the surface-color values for each pixel position.

Depth-Buffer Algorithm:

1. Initialize the depth buffer and frame buffer so that for all buffer positions (x, y),
   depthBuff (x, y) = 1.0, frameBuff (x, y) = backgndColor
2. Process each polygon in a scene, one at a time, as follows:
   - For each projected (x, y) pixel position of a polygon, calculate the depth z (if not already known).
   - If z < depthBuff (x, y), compute the surface color at that position and set
     depthBuff (x, y) = z, frameBuff (x, y) = surfColor (x, y)

After all surfaces have been processed, the depth buffer contains depth values for the visible surfaces and the frame buffer contains the corresponding color values for those surfaces.

- Given the depth values for the vertex positions of any polygon in a scene, we can calculate the depth at any other point on the plane containing the polygon. ϖ At surface position (x, y), the depth is calculated from the plane equation as

$$z = \frac{-Ax - By - D}{C}$$

- If the depth of position (x, y) has been determined to be z, then the depth z' of the next position (x+1,y) along the scan line is obtained as,

$$z' = \frac{-A(x+1) - By - D}{C}$$

$$z' = z - \frac{A}{C}$$

- The ratio −A/C is constant for each surface, so succeeding depth values across a scan line are obtained from preceding values with a single addition.
- We can implement the depth-buffer algorithm by starting at a top vertex of the polygon.
- Then, we could recursively calculate the x-coordinate values down a left edge of the polygon.
- The x value for the beginning position on each scan line can be calculated from the beginning (edge) x value of the previous scan line as

$$x' = x - \frac{1}{m}$$ where m is the slope of the edge.

- Depth values down this edge are obtained recursively as

$$z' = z + \frac{A/m + B}{C}$$

- If we are processing down a vertical edge, the slope is infinite and the recursive calculations reduce to

$$z' = z + \frac{B}{C}$$

5. Write a program to animate a flag using Bezier Curve algorithm.

```
#include<GL/glut.h>
#include<stdio.h>
#include<math.h>
#define PI 3.1416
GLsizei winWidth = 600, winHeight = 600;
GLfloat xwcMin = 0.0, xwcMax = 130.0;
GLfloat ywcMin = 0.0, ywcMax = 130.0;
typedef struct wcPt3D
{
      GLfloat x, y, z; };
void bino(GLint n, GLint *C){
      GLint k, j;
      for(k=0;k<=n;k++){
        C[k]=1;
      for(j=n;j>=k+1; j--)
        C[k]*=j;
      for(j=n-k;j>=2;j--)
        C[k]/=j;
      }
```

```
}

void computeBezPt(GLfloat u, wcPt3D *bezPt, GLint nCtrlPts, wcPt3D *ctrlPts,
Glint *C){
      GLint k, n=nCtrlPts-1;
      GLfloat bezBlendFcn;
      bezPt ->x =bezPt ->y = bezPt->z=0.0;
      for(k=0; k< nCtrlPts; k++){
      bezBlendFcn = C[k] * pow(u, k) * pow( 1-u, n-k);
      bezPt ->x += ctrlPts[k].x * bezBlendFcn;
      bezPt ->y += ctrlPts[k].y * bezBlendFcn;
      bezPt ->z += ctrlPts[k].z * bezBlendFcn;
      }
}
void bezier(wcPt3D *ctrlPts, GLint nCtrlPts, GLint nBezCurvePts){
      wcPt3D bezCurvePt;
      GLfloat u;
      GLint *C, k;
      C= new GLint[nCtrlPts];
      bino(nCtrlPts-1, C);
      glBegin(GL_LINE_STRIP);
      for(k=0; k<=nBezCurvePts; k++){
        u=GLfloat(k)/GLfloat(nBezCurvePts);
        computeBezPt(u, &bezCurvePt, nCtrlPts, ctrlPts, C);
        glVertex2f(bezCurvePt.x, bezCurvePt.y);
      }
      glEnd();
      delete[]C;
}

void displayFcn(){
      GLint nCtrlPts = 4, nBezCurvePts =20;
      static float theta = 0;
wcPt3D ctrlPts[4] = { {20, 100, 0}, {30, 110, 0}, {50, 90, 0}, {60, 100, 0}};
      ctrlPts[1].x +=10*sin(theta * PI/180.0);
      ctrlPts[1].y +=5*sin(theta * PI/180.0);
      ctrlPts[2].x -= 10*sin((theta+30) * PI/180.0);
      ctrlPts[2].y -= 10*sin((theta+30) * PI/180.0);
      ctrlPts[3].x-= 4*sin((theta) * PI/180.0);
      ctrlPts[3].y += sin((theta-30) * PI/180.0);
      theta+=0.1;
     glClear(GL_COLOR_BUFFER_BIT);
     glColor3f(1.0, 1.0, 1.0);
     glPointSize(5);
     glPushMatrix();
     glLineWidth(5);
     glColor3f(255/255, 153/255.0, 51/255.0); //Indian flag: Orange color code
     for(int i=0;i<8;i++){
        glTranslatef(0, -0.8, 0);
      bezier(ctrlPts, nCtrlPts, nBezCurvePts);
     }
     glColor3f(1, 1, 1); //Indian flag: white color code
     for(int i=0;i<8;i++){
        glTranslatef(0, -0.8, 0);
        bezier(ctrlPts, nCtrlPts, nBezCurvePts);
      }
     glColor3f(19/255.0, 136/255.0, 8/255.0); //Indian flag: green color code
     for(int i=0;i<8;i++){
        glTranslatef(0, -0.8, 0);
```

```
        bezier(ctrlPts, nCtrlPts, nBezCurvePts);
      }
      glPopMatrix();
    glColor3f(0.7, 0.5,0.3);
    glLineWidth(5);
    glBegin(GL_LINES);
    glVertex2f(20,100);
    glVertex2f(20,40);
    glEnd();
    glFlush();
    glutPostRedisplay();
    glutSwapBuffers();
}

void winReshapeFun(GLint newWidth, GLint newHeight){
  glViewport(0, 0, newWidth, newHeight);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(xwcMin, xwcMax, ywcMin, ywcMax);
  glClear(GL_COLOR_BUFFER_BIT);
}

void main(int argc, char **argv){
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
  glutInitWindowPosition(50, 50);
  glutInitWindowSize(winWidth, winHeight);
  glutCreateWindow("Bezier Curve");
  glutDisplayFunc(displayFcn);
  glutReshapeFunc(winReshapeFun);
  glutMainLoop();
}
```

6. Explain basic illumination model? Explain Phong model.

Basic Illumination Models:

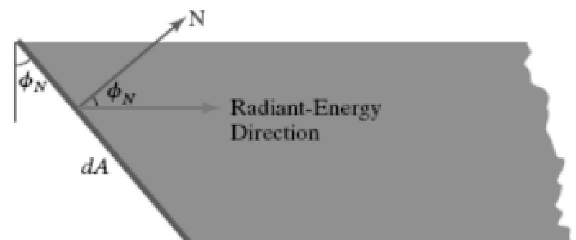
- Light-emitting objects in a basic illumination model are generally limited to point sources many graphics packages provide additional functions for dealing with directional lighting (spotlights) and extended light sources.

Ambient Light:

- This produces a uniform ambient lighting that is the same for all objects, and it approximates the global diffuse reflections from the various illuminated surfaces.
- Reflections produced by ambient-light illumination are simply a form of diffuse reflection, and they are independent of the viewing direction and the spatial orientation of a surface.
- However, the amount of the incident ambient light that is reflected depends on surface optical properties, which determine how much of the incident energy is reflected and how much is absorbed

Diffuse Reflection:

- The incident light on the surface is scattered with equal intensity in all directions, independent of the viewing position.



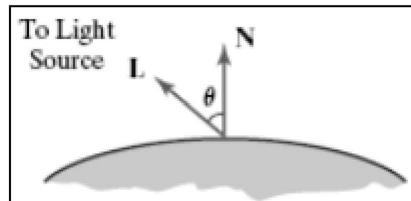
- Such surfaces are called ideal diffuse reflectors they are also referred to as Lambertian reflectors, because the reflected radiant light energy from any point on the surface is calculated with Lambert's cosine law.
- Assuming that every surface is to be treated as an ideal diffuse reflector (Lambertian), we can set a parameter kd for each surface that determines the fraction of the incident light that is to be scattered as diffuse reflections.
- This parameter is called the diffuse-reflection coefficient or the diffuse reflectivity. The ambient contribution to the diffuse reflection at any point on a surface is simply

$$I_{ambdiff} = k_d I_a$$

- We can model the diffuse reflections from a light source with intensity I_l using the calculation,

$$I_{l,diff} = k_d I_{l,incident} = k_d \cos\theta$$

- At any surface position, we can denote the unit normal vector as N and the unit direction vector to a point source as L,

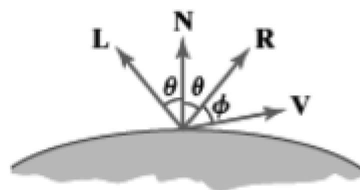


- The diffuse reflection equation for single point-source illumination at a surface position can be expressed in the form

$$I_{l,\text{diff}} = \begin{cases} k_d I_l (\mathbf{N} \cdot \mathbf{L}), & \text{if } \mathbf{N} \cdot \mathbf{L} > 0 \\ 0.0, & \text{if } \mathbf{N} \cdot \mathbf{L} \le 0 \end{cases}$$

Specular Reflection and the Phong Model:

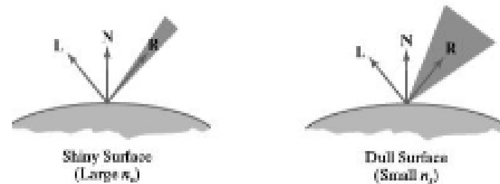
- The bright spot, or specular reflection, that we can see on a shiny surface is the result of total, or near total, reflection of the incident light in a concentrated region around the specular-reflection angle.
- The below figure shows the specular reflection direction for a position on an illuminated surface.



1. N represents: unit normal surface vector The specular reflection angle equals the angle of the incident light, with the two angles measured on opposite sides of the unit normal surface vector N
2. R represents the unit vector in the direction of ideal specular reflection,
3. L is the unit vector directed toward the point light source, and
4. V is the unit vector pointing to the viewer from the selected surface position.

- Angle φ is the viewing angle relative to the specular-reflection direction R
- An empirical model for calculating the specular reflection range, developed by Phong Bui Tuong and called the Phong specular-reflection model or simply the Phon G model, sets the intensity of specular reflection proportional to cosns φ

- Angle φ can be assigned values in the range 0◦ to 90◦, so that cos φ varies from 0 to 1.0.
- The value assigned to the specular-reflection exponent ns is determined by the type of surface that we want to display.
- A very shiny surface is modeled with a large value for ns (say, 100 or more), and smaller values (down to 1) are used for duller surfaces.
- For a perfect reflector, ns is infinite. For a rough surface, such as chalk or cinderblock, ns is assigned a value near 1.



- We can approximately model monochromatic specular intensity variations using a specular-reflection coefficient, W(θ), for each surface.
- In general, W(θ) tends to increase as the angle of incidence increases. At θ = 90◦, all the incident light is reflected (W(θ) = 1).
- Using the spectral-reflection function W(θ), we can write the Phong specular-reflection model as

$$I_{l,\text{spec}} = W(\theta) I_l \cos^{n_s} \phi$$

where Il is the intensity of the light source, and φ is the viewing angle relative to the specular reflection direction R.

- Because V and R are unit vectors in the viewing and specular-reflection directions, we can calculate the value of cos φ with the dot product V·R.
- In addition, no specular effects are generated for the display of a surface if V and L are on the same side of the normal vector N or if the light source is behind the surface.
- We can determine the intensity of the specular reflection due to a point light source at a surface position with the calculation.

$$I_{l,\text{spec}} = \begin{cases} k_s I_l (\mathbf{V} \cdot \mathbf{R})^{n_s}, & \text{if } \mathbf{V} \cdot \mathbf{R} > 0 \quad \text{and} \quad \mathbf{N} \cdot \mathbf{L} > 0 \\ 0.0, & \text{if } \mathbf{V} \cdot \mathbf{R} \leq 0 \quad \text{or} \quad \mathbf{N} \cdot \mathbf{L} \leq 0 \end{cases}$$

7. Write an interactive program to rotate a square.

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include<math.h>
#include<stdio.h>

GLfloat theta,thetar;
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POLYGON);
    thetar=theta/(3.14159/180.0);         //convert theta in degrees to
radians
```

```
    glVertex2f(cos(thetar),sin(thetar));
    glVertex2f(-sin(thetar),cos(thetar));
    glVertex2f(-cos(thetar),-sin(thetar));
    glVertex2f(sin(thetar),-cos(thetar));
    glEnd();
    glFlush();
    glutSwapBuffers();
}
void idle()
{
    theta+=2;
    if(theta>=360.0) theta-=360.0;
    glutPostRedisplay();
}

void mouse(int button,int state,int x,int y)      //  change idle
function based on
         //  mouse button pressed
{
    if(button==GLUT_LEFT_BUTTON&&state==GLUT_DOWN)
    glutIdleFunc(idle);
    if(button==GLUT_RIGHT_BUTTON&&state==GLUT_DOWN)
        glutIdleFunc(NULL);
}
int main(int argc,char **argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
    glutCreateWindow("Rotating Square");
  //  glutIdleFunc(idle);
    glutDisplayFunc(display);
    glutMouseFunc(mouse);
    glutMainLoop();
    return 0;
}
```

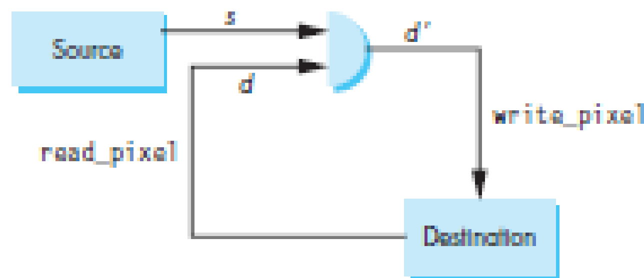
8. Write a note on logical operations. Explain XOR operation in detail.

- Two types of functions that define writing modes are:
    1. Replacement mode
    2. Exclusive OR (XOR)

- When program specifies about visible primitive then OpenGL renders it into set of color pixels and stores it in the present drawing buffer.
- In case of default mode, consider we start with a color buffer then has been cleared to black. Later we draw a blue color rectangle of size 10x10 pixels then 100 blue pixels are copied into the

color buffer, replacing 100 black pixels. Therefore, this mode is called as “copy or replacement mode”.

- Consider the below model, where we are writing single pixel into color buffer.



  - The pixel that we want to write is called as “source pixel”.
  - The pixel in the drawing buffer which gets replaced by source pixel is called as ‘destination pixel’.
  - In Exclusive-OR or (XOR) mode, corresponding bits in each pixel are combing using XOR logical operation.
  - If s and d are corresponding bits in the source and destination pixels, we can denote the new destination bit as d’. d’ = d XOR s.
  - One special property of XOR operation is if we apply it twice, it returns to the original state, it returns to the original state. So, if we draw some thing in XOR mode, we can erase it by drawing it again.

$$\mathbf{d} = (\mathbf{d} \oplus \mathbf{s}) \oplus \mathbf{s}$$

  - OpenGL supports all 16 logic modes; copy mode (GL_COPY) is the default. To change mode, we must enable logic operation, glEnable(GL_COLOR_LOGIC_OP) and then it can change to XOR mode glLogicOp(GL_XOR).