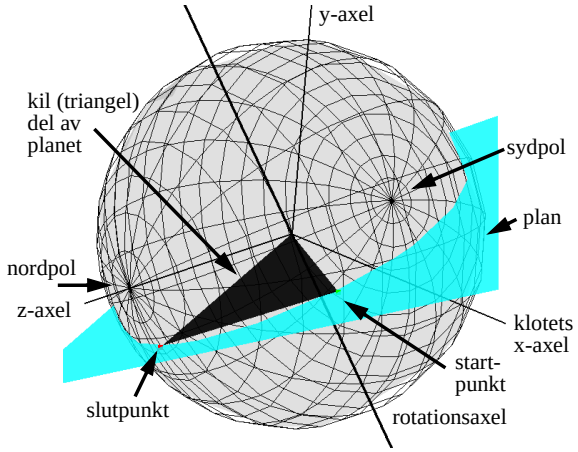
```
void myFigure(int x, int y) {
   glBegin(GL_POLYGON);
         glVertex2i(x,y);
         glVertex2i(x+50,y);
         glVertex2i(x+50,y+70);
    glEnd();
}

void display(void) {
    GLbyte Minne[30000];
    GLbyte M[48]={127,0,0,127,0,0,127,0,0,127,0,0,
           0,127,0,0,127,0,0,127,0,0,127,0,
           0,0,127,0,0,127,0,0,127,0,0,127,
           127,127,0,127,127,0,127,127,0,127,127,0};
    printf("Display called\n");
    glClearColor(1.0,1.0,1.0,1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0,0.0,1.0);
    myFigure(0,0);
    glRasterPos2d(100,100);
    glCopyPixels(0,0,50,70,GL_COLOR);
    glReadPixels(0,0,50,70,GL_RGB,GL_BYTE,Minne);
    glRasterPos2d(200,100);    glPixelZoom(2.0,3.0);
    glDrawPixels(50,70,GL_RGB,GL_BYTE,Minne);
    glRasterPos2d(200,10);    glPixelZoom(4.0,4.0);
    glDrawPixels(4,4,GL_RGB,GL_BYTE,M);
    glPixelZoom(1.0,1.0);

}

int GLOBAL_height;
void myReshape(int width, int height) {
    printf("Reshape called!\n");
    glViewport(0, 0, width, height);
    GLOBAL_height = height;
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho (0.0, width, 0.0, height, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
}
```


Rasterkopiering i JOGL

I förhållande till JOGL1_MB.java behöver vi som vanligt bara ändra *reshape* och *display*-

```
private void myFigure(int x, int y) {
   gl.glBegin(GL.GL_POLYGON);
            gl.glVertex2d(x,y);
            gl.glVertex2d(x+50,y);
            gl.glVertex2d(x+50,y+70);
   gl.glEnd();
}
public void reshape(GLDrawable drawable, int x, int y, int
width, int height) {
   gl.glViewport(0,0,width,height);
   gl.glMatrixMode(GL.GL_PROJECTION);
   gl.glLoadIdentity();
   gl.glOrtho(0.0, width, 0.0, height, -1.0, 1.0);
   gl.glMatrixMode(GL.GL_MODELVIEW); gl.glLoadIdentity();
   GLOBAL_height = height;
}
public void display(GLDrawable drawable) {
   System.out.println("Tid=: "+System.currentTimeMillis());
   byte[] Minne = new byte[30000];
   byte[] M={127,0,0, 127,0,0, 127,0,0, 127,0,0,
             0,127,0, 0,127,0, 0,127,0, 0,127,0,
             0,0,127, 0,0,127, 0,0,127, 0,0,127,
             127,127,0, 127,127,0, 127,127,0, 127,127,0};
   gl.glClearColor(1.0f,1.0f,1.0f,1.0f);
   gl.glClear(GL.GL_COLOR_BUFFER_BIT |
             GL.GL_DEPTH_BUFFER_BIT);
   gl.glColor3d(0,0,1); myFigure(0,0);
   gl.glRasterPos2d(100,100);
   gl.glCopyPixels(0,0,50,70,GL.GL_COLOR);
   gl.glReadPixels(0,0,50,70,GL.GL_RGB,GL.GL_BYTE,Minne);
   gl.glRasterPos2d(200,100); gl.glPixelZoom(2.0f,3.0f);
   gl.glDrawPixels(50,70,GL.GL_RGB,GL.GL_BYTE,Minne);
   gl.glRasterPos2d(200,10); gl.glPixelZoom(4.0f,4.0f);
   gl.glDrawPixels(4,4,GL.GL_RGB,GL.GL_BYTE,M);
   gl.glPixelZoom(1.0f,1.0f);
}
```


Bågbollsnavigering (arcball)



Om man vill kunna se på ett föremål från olika håll, kan man använda en s k bågboll (eng. arcball, virtual trackball), som är ett tänkt klot omgivande föremålet. Klotet (det tänkta) kan man sedan snurra på genom att trycka ner en mustangent och dra med musen. Detta ger en mycket intuitiv interaktion. Utgångspunkten och aktuell punkt definierar tillsammans med klotets mittpunkt ett plan (som ger en storcirkel på klotet och rörelsen kan beskrivas som en rotation kring planets normal. När vi väl känner start- och slutpunkt, vilket kräver en omvandling från muskoordinater till modellkoordinater, är det lätt att beräkna planet och dess normal och därmed rotationsaxeln och rotationsvinkeln. Det första steget görs med en metod *glUnproject*, som tas upp någon gång senare. Se även uppgift 3-4 på lab 2.

Fotorealism 1(1)

Den värld du skapar i laboration 2 liknar föga verkligheten. Vad är det som fattas?

- Ljus inkl reflektion av olika slag
- Texturering av olika slag, t ex för gräsmatta
- Skuggor

Vi skall här se på en enkel **belysningsmodell** och sedan se hur den förverkligas i OpenGL. Därefter ett par tekniker för att återge scener: **strålföljning** och **radiositet**. Sedan **texturering**. Snabba skuggor kommer separat senare.

Hittills har vi angett en färg för det som skall ritas. Men i verkligheten har objekten inte färger utan materialegenskaper. Det är dessa egenskaper i kombination med belysningen som bestämmer hur vi upplever världen.

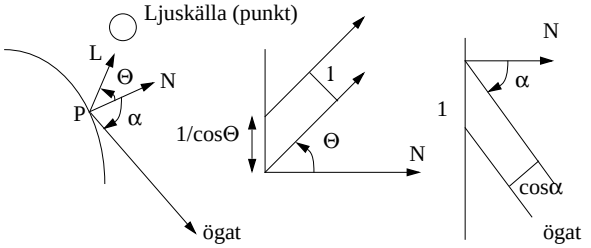
När vi säger att ett föremål har röd färg menar vi att det ser rött ut i normal belysning (vitt ljus). Orsaken är att av det vita ljus som träffar föremålet reflekteras bara den röda delen. Resten absorberas. Om vi belyser ett "rött" föremål med blått ljus, finns inget rött att reflektera och föremålet ter sig svart.

Vi byter alltså ut färgsättning mot materialegenskaper. Bland dessa finns:
Reflektionsfaktor för bakgrundsljus
Diffus reflektionsfaktor
Spegelreflektionsfaktor
Speglingskoncentrationsgrad

Fotorealism: En belysningsmodell 1(3)

Diffus reflektion

Vi vill beräkna intensiteten (energi/ytenhet) i punkten P, eller rättare sagt hur ögat upplever intensiteten i den punkten. N är den normerade normalen till ytan i P. L är en normerad vektor mot ljuskällan.



Från ljuskällan kommer i riktningen bestämd av L ljus med intensiteten I_e. Det betyder att energin i röret (genomskärningsytan 1) i mellersta figuren är I_e. Denna energi sprids på en något större area, dvs intensiteten på ytan blir $I_e\cos\Theta$. Delar av den infallande energin reflekteras. Vi antar att diffusa reflektionsfaktorn är k_d. Det betyder att den utgående intensiteten är $I_d = k_dI_e\cos\Theta = k_dI_e(L\bullet N)$. Vid diffus reflektion brukar man säga att denna upplevs lika oberoende av ögats position. Men egentligen är det litet omständligare. Ljuset sprids proportionellt mot cosα, dvs i röret i den högra figuren kommer energin I_d*cosα (vi glömmer en fördelningskonstant; vi glömmer också den tredje dimensionen). Denna fördelar sig på arean cosα (vinkelrät mot ögat), dvs den upplevda intensiteten blir I_d, precis som det sas för tre meningar sedan.

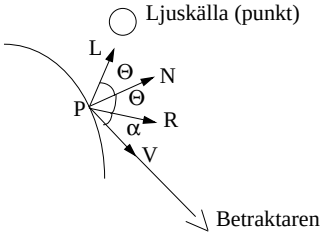
Fotorealism: En belysningsmodell 2(3)

Bakgrundsljusreflektion

$I_a = k_aI_{ambient}$

Speglingsreflektion

Samma förutsättningar som för diffus reflektion. För ett perfekt speglande material skulle allt ljus reflekteras i **huvudreflektionsriktningen R**, men i verkligheten sker en viss spridning som avtar med vinkeln α, vinkelskillnaden mellan huvudreflektionsriktningen och vektorn V (normerad) mot betraktaren



En modell (Phong) - det finns andra - för spegelreflektionen är
$I_s = k_sI_e(\cos\ \alpha)^n = k_sI_e(R_{norm}\bullet V)^n$ där $R_{norm}=R/|R|$, $R = 2(N\bullet L)N-L$
I en variant (mindre räknearbete) ersätter man R med H = L + V (riktning mittemellan L och V) och låter α vara avvikelsen från normalen.

Strålande ytor

Vi kan ta med en sådan genom att införa ett bidrag $I_{e,ytan}$.

Fotorealism: En belysningsmodell 3(3)

Totalt får vi för den upplevda intensiteten i punkten
$I = I_a + I_d + I_s + I_{e,ytan}$

Ovanstående beräkningar måste i praktiken genomföras för varje basfärg för sig. Dvs faktorerna är olika beroende på om det gäller R, G eller B. Se tabell längre fram.

I praktiken dämpas ljus med avståndet och man har en gång fått lära sig att intensiteten avtar med kvadraten på avståndet. I datorgrafiksammanhang struntar man ofta i detta beroende och de gånger man tar hänsyn till dämpningen antar man ofta att dämpningen är linjär. I OpenGL kan man arbeta med en dämpningsfaktor av formen

$$\frac{1}{a + bd + cd^2}$$

där d är avståndet.

Om vi har flera ljuskällor, N st:

$$I = I_a + \sum_1^N I_{e,i}\left(k_d(N \bullet L_i) + k_s(V \bullet R_i)^n\right)$$

Det finns mer komplicerade belysningsmodeller (se t ex Hills bok, avsnitt 14.7.3 som beskriver Cook-Torrance modell) som kan ge ett mera realistiskt resultat. Den vi beskrivit är den som vanligen förekommer i grundläggande datorgrafikböcker.

Fotorealism: Materialvärden

McReynolds och Blythe anger i sin kurs “Avancerad OpenGL” (finns på CD’n i *OPENGL2000/SGICOURSE*; se även länk på kurssidan) given vid SIGGRAPH ett par gånger lämpliga värden för ett stort antal material. När flera rader förekommer gäller de i tur och ordning R-, G- och B-komponenterna.

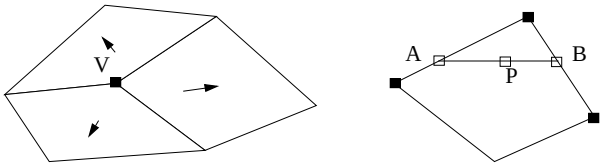
Material	$\mathbf{k_a}$	$\mathbf{k_d}$	$\mathbf{k_s}$	**n**
Silver	0.19225	0.50754	0.508273	51.2
Koppar	0.19225 0.0735 0.0225	0.7038 0.27048 0.0828	0.256777 0.137622 0.086014	12.8
Guld	0.24725 0.1995 0.0745	0.75164 0.60648 0.22648	0.628281 0.555802 0.366065	51.2
Mässing	0.329412 0.223529 0.027451	0.780392 0.568627 0.113725	0.992157 0.941176 0.807843	27.8974

Hill har en fullständigare tabell.

Fotorealism: Toningssätt (eng shading; målningssätt?)

Vi har en modell uppbyggd av polygoner och vill återge den i en värld med belysning. Tre olika sätt (jfr t ex programmet FACES):

- **Platt toning**: Man räknar ut ett belysningsvärde med hjälp av polygonens normal och belysningsmodellen. Värdet används som intensitet över hela polygonen.
- **Gouraudtoning**: I allmänhet är våra modeller approximationer av krökta modeller. Platt toning bidrar till att approximationen upplevs som kantig. Gouraud-toning gör att man får variation över ytan och samtidigt kontinuitet vid kanterna. Metoden innebär att man beräknar en medelnormal för varje hörn utifrån angränsande polygoners normaler. Med denna normal och belysningsmodellen beräknar man ett belysningsvärde i hörnet. För att beräkna ett värde i punkten P interpolerar man först fram ett värde i punkterna A och B och interpolerar sedan mellan dessa. Stöds av OpenGL.



- **Phong-toning**: Med Gouraud-toning kommer ljusintensiteten att variera linjärt över en enskild polygon. Men detta är uppåt väggarna speciellt när det gäller spegelreflektion. I Phongs metod anstränger man sig litet mer. I varje hörnpunkt beräknas som förut en medelnormal. Utifrån dessa beräknar vi med interpolation normaler i punkterna A och B och med ny interpolation en normal i P. Slutligen beräknas med denna och belysningsmodellen en intensitet i P. Phong-toning har inget stöd i OpenGL (jo med VFP).

Fotorealism: Belysning i OpenGL 1(2)

Se OpenGL-häftet. Väsentligen den modell som beskrivits. Men i ett par fall litet fler detaljer (som vi förbigår).

Materialegenskaper (ersätter tillsammans med *glNormal glColor*)

```
              GL_FRONT,          GL_AMBIENT,
glMaterialfv(GL_BACK,            GL_DIFFUSE, 4-vektor)
              GL_FRONT_AND_BACK, GL_SPECULAR,
                                 GL_EMISSION
glMaterialf(-"-,GL_SHININESS,koncentrationsgrad n)
```

Normaler (ersätter tillsammans med *glMaterial glColor*)

`glNormal3f(a,b,c)`

Skall vara normerad vid belysningsberäkningen, vilket kan kräva `glEnable(GL_NORMALIZE)` (åtminstone om *glScale* finns med i transformationskedjan).

Ljuskällor

```
                    GL_AMBIENT,
glLightfv(GL_LIGHTx,GL_DIFFUSE,4-vektor)
                    GL_SPECULAR,
                    GL_POSITION,
```

Man låter ljuset bestå av tre delar, medan vi tidigare betraktade det som enhetligt (lättast att låta alla vara lika). När det gäller positionen betyder en vektor med 0 i fjärde komponenten att en riktning till ljuskällan anges, medan ett värde 1 där betyder en äkta position.

Aktivera belysning

`glEnable(GL_LIGHTING)`+per ljuskälla `glEnable(GL_LIGHTx)`

Val av toningsmodell

`glShadeModel(`*`modell`*`)`, där *modell* kan vara `GL_FLAT` (platt) eller `GL_SMOOTH` (Gouraud).

Om vi vill se på ut- och insidor

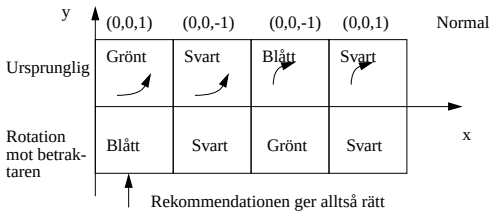
```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,GL_TRUE)
```


Fotorealism: Belysning i OpenGL 2(2)

Man ger ju ofta olika materialegenskaper åt polygoners fram- och baksida.

I en del grafiksystem låter man begreppet framsida definiera även polygonens normal. Så inte i OpenGL. Utan den (de) normaler som behövs för belysningsberäkningen är helt oberoende och anges med `glNormal` för en eller flera hörnpunkter. Vid flat toning (`glShadingMode(GL_FLAT)`) används en normal för hela polygonen. Vid Gouraud-toning (`glShadingMode(GL_SMOOTH)`) behövs en normal per hörnpunkt. **Normalerna skall vara utåtriktade från framsidan**, dvs om hörnen anges i motursordning och normalen är utåtriktad blir resultatet som vi tänkt oss. De flesta andra alternativ ger fel resultat, t ex en svart yta.

Exempel: En fyrkant i xy-planet med grön framsida och blå baksida (enligt glMaterial). Hörnen angivna i olika ordningar (enligt pilarna). Normalerna också olika. Program: GL_BELYS.

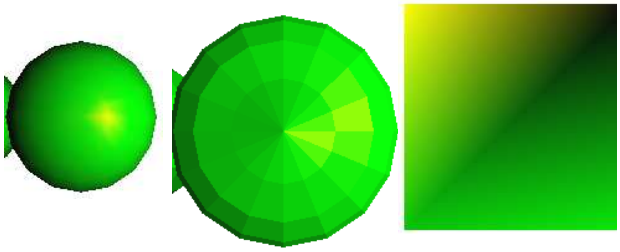


Gouraud- resp flat toning i OpenGL

I enklare fall anger man en normal per polygon (innanför `glBegin` eller utanför). Den gäller då samtliga hörn i polygonen. Eftersom samtliga hörn har samma normal ger flat toning och Gouraud-toning samma resultat.

I allmänhet anger man en normal per hörn (innanför `glBegin`). Om man begär flat toning används den först angivna för beräkning av polygonens belysningsvärde. Om Gouraud-toning begärs används i stället samtliga normalvärden på det sätt som tidigare beskrivits. För att resultatet skall bli annorlunda än vid flat toning måste normalvärdena skilja sig åt, vilket de gör om man **bildar hörnnormalerna som medelvärden av angränsande polygoners riktiga normaler. Detta steg måste du själv göra**.

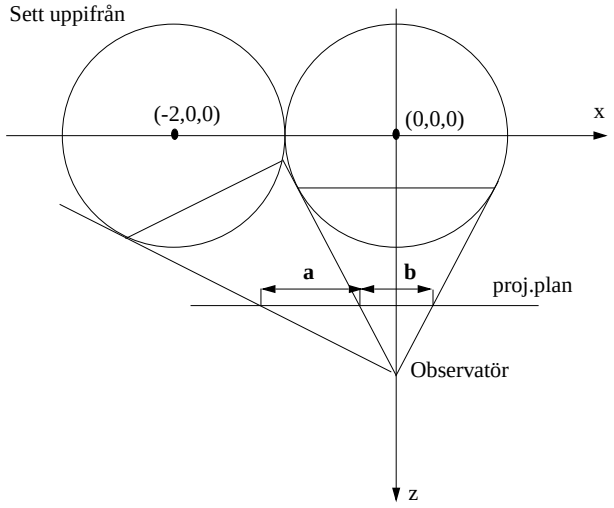
Alla GLUTs procedurer för färdiga objekt tillverkar automatiskt normaler för samtliga hörn. Alla utom `glutSolidCube` gör det så att Gouraud-toning fungerar. Detta val är naturligt, eftersom ett kantigt objekt som en kub bör se kantigt ut. T v en sfär med Gouraud-toning,



i mitten sfär med flat toning och t h fyrhörning med Gouraud-toning.

Exempel 17 i OGL-häftet: GL_ANIM.c (utv till GL_LJUS.c)

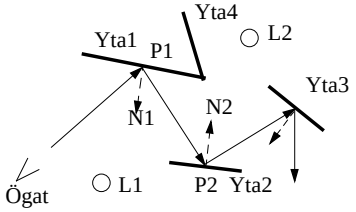
"Varför ser den vänstra sfären ut som en ellipsoid?" frågade en uppmärksam kursdeltagare. Orsaken är att vi bara ser delar av de båda sfärerna. Den högra döljer delvis den vänstra (vilket vi också noterade i datorbilden). Man kan naturligtvis räkna fram att a>b, men jag hoppas att det framgår ändå.



På längre observatörsavstånd blir det mera normalt. Blir det så här i verkligheten också? Blås två jättelika ballonger eller gjut stora betongklot och kontrollera.

Fotorealism: Strålföljning 1(3)

För att få global fotorealism får man ta i mer. En metod är **strålföljning** (eng ray tracing; eng ray casting är en förenklad variant som bara löser dolda-yt-problemet). Metoden är principiellt enkel.



Ögat tittar från sin position i en viss riktning. Vi bestämmer vilka ytor som träffas av strålen och motsvarande skärningspunkter. Vi tar reda på den närmsta av dessa. I figurens fall träffas Yta1 och Yta4. Punkten P1 på Yta1 är närmst. Vi bestämmer till att börja med ljusvärdet i P1 med den tidigare ljusmodellen. Vi tar med bidrag enbart från de ljuskällor som är synliga från P1, dvs vi måste kontrollera om linjen från P1 till ljuskällan skär något någon yta. I figurens fall döljs inte L1 men däremot L2 (av såväl Yta1 som Yta2). Vi tar alltså med enbart bidraget från ljuskälla L1. Till det nu beräknade ljusvärdet lägger vi den från P2 eller motsvarande reflekterade ljuset (efter multiplikation med sedvanlig reflektionsfaktor). Vi följer dock bara huvudreflektionsriktningarna. Vi är därmed tillbaka i nästan samma situation som från början. Men nu gäller det att bestämma hur en betraktare i P1 upplever punkten P2. Vi tar därmed lämpligen till rekursion. Rekursionen avbryts när bakgrunden (ingen yta) träffas av strålen, rekursionsdjupet är för stort, objektet är föga reflekterande eller ett strålande material påträffas (utbredd ljuskälla).

Fotorealism: Strålföljning 2(3)

Blender använder enbart OpenGL:s belysningsverktyg. Ingen strålföljning (man kan koppla en strålföljare till Render-steget).

Ett trevligt strålföljningsprogram är *PovRay.* Finns för PC och Sun/Solaris. Modellen byggs i en textfil. Eller med hjälp av en modellerare (*Moray* - bara för PC och kostar; *Art Of Illusion* - javaprogrammet).

I ett strålföljningsprogram använder man obetydligt av OpenGL. Egentligen bara 2D-grafik med ritning av en punkt. Dock görs runt om i världen diverse försök att använda fragmentprogram för strålföljning. NVIDIA har en produkt (Gelato; programvara + Quadro FX) som möjligen fungerar så (dokumentationen är oklar).

Strålföljning lämpar sig för parallellism.

Strålföljning klarar även genomskinliga föremål, men vi tar inte upp några detaljer här. Vad man gör är att man följer inte bara en reflektionsriktning utan även en brytningsriktning genom det genomskinliga materialet. Det är förhållandevis enkelt att lägga till när resten av programmet är klart. Man behöver naturligtvis nu utökade materialdata (brytningskoefficient och genomsläpplighetsfaktor).

Eftersom strålföljning i sina rena form inte är en realtidsalgoritm, finns det diverse "fusk"-sätt som kan lura den oinitierade.

Fotorealism: Strålföljning. Program 3(3)

Ett programskelett:

0. Vi har en lista (vektor) med alla objekt
1. För varje bildpunkt
   1. Beräkna motsvarande punkt i världen
   2. Beräkna strålen (startpunkt, riktning) från ögat till denna punkt
   3. Beräkna färg(stråle,anropsdjup)

Funktionen färg(stråle,anropsdjup):

1. Beräkna strålens skärningar med ytor. Träffar "bakom" strålens utgångspunkt ointressanta.
2. Om det inte finns några intressanta träffar returnera med bakgrundsfärgen.
3. Låt annars P vara den närmsta
   1. Ljusvärde = emissionsljus + omgivningsljus (vi måste själva räkna; OpenGL hjälper inte längre till).
   2. För alla punktformiga ljuskällor
      1. Om ljuskällan synlig ljusvärde = ljusvärde + diffus- och speglingsbidrag från ljuskällan (dito)
   3. Om punkten P reflekterande nog och anropsdjupet lågt nog
      ljusvärde = ljusvärde +
      färg(stråle i huvudreflektionsriktningen, anropsdjup+1)
   4. Returnera ljusvärdet

Huvudproblem: Träffar.