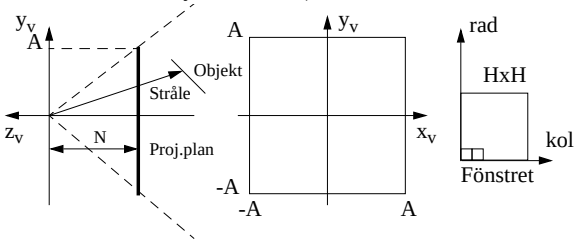
Strålföljning: Bildpunkt till värld 1(1)

Med bildens beteckningar får vi (för enkelhets skull antas att fönstret är kvadratiskt och att synkvoten är 1)



att en bildpunkt (kol,rad) motsvarar i vykoordinatsystemet $P_v = (x_v,y_v,z_v)^T$, där

$x_v = A(-1 + 2*kol/H)$, $y_v = A(-1 + 2*rad/H)$, $z_v = -N$

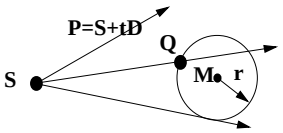
Uttryckt i världskoordinater blir punkten

$$\mathbf{P}_w = \mathbf{P}_0 - N\hat{\mathbf{z}}_\mathbf{v} + x_v\hat{\mathbf{x}}_\mathbf{v} + y_v\hat{\mathbf{y}}_\mathbf{v}$$

där P_0 är ögats placering i världskoordinatsystemet och $\hat{\mathbf{x}}_v$ etc avser de normerade vykoordinataxlarna uttryckta i världskoordinater (se "Från värld till skärm").

Strålföljning: Skärning stråle och sfär 1(1)

En sfär med mittpunkt M och radie r är given. Vi vill veta om strålen med utgångspunkt i punkten S och riktningen D (gärna normerad så att A nedan är 1; då är t avståndet från utgångspunkten) träffar sfären. Vi kommer att bestämma t för att få ett svar på frågan. I figuren syns



tre olika fall. Notera att lösningar med t<0 saknar intresse (de ligger bakom betraktaren eller strålens utgångspunkt). Punkten Q skall ligga på både strålen och sfären, dvs

$$\begin{cases} Q = S + tD \\ |Q - M| = r \end{cases}$$

Insättning av första ekvationen i andra ger $|S + tD - M|^2 = r^2$.

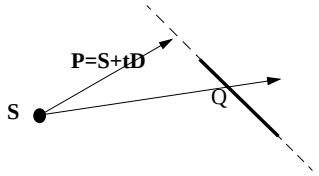
Med S=(S_1,S_2,S_3) etc och beteckningarna (vektornotation kortare)

$$A = \sum_{i=1}^{3} D_i^2 \qquad B = \sum_{i=1}^{3} (S_i - M_i)D_i \qquad C = \sum_{i=1}^{3} (S_i - M_i)^2 - r^2$$

får vi andragradsekvationen $At^2 + 2Bt + C = 0$ med lösningar $t = \frac{1}{A}\left(-B \pm \sqrt{B^2 - AC}\right)$. Diskriminanten avgör huruvida det finns 0, 1 eller 2 lösningar. Vi accepterar bara t>0.

Strålföljning: Skärning stråle och polygon 1(1)

En plan polygon med hörn V_1, ..., V_N är given. Vi vill veta om strålen med utgångspunkt i punkten S och riktningen D träffar polygonen.



Vi kan som vanligt lätt bestämma ekvationen F(x,y,z)=Ax+By+Cz+D=0 för det plan i vilket polygonen ligger. Vi börjar med att leta efter eventuella skärningar mellan strålen och detta plan. I figurens fall skär båda strålarna planet, men bara den understa skär polygonen. Vi får ekvationerna

$$\begin{cases} Q = S + tD \\ F(Q) = 0 \end{cases}$$

och efter insättning av den första i den andra

$$t = \frac{AS_1 + BS_2 + CS_3 + D}{-(AD_1 + BD_2 + CD_3)}$$

Som förut accepteras bara värden t>0. Återstår sedan att kolla att punkten verkligen hamnat inom polygonen. Vi tar upp detta problem senare under Beräkningsgeometri.

Strålföljning

Detta var en rask introduktion till strålföljning. Det finns åtskilliga andra problemställningar förknippade med strålföljning, som man inte anar omedelbart.

Eftersom metoden i sig är långsam är det extra viktigt att kunna göra olika slag av effektiviseringar, som t ex hindrar att vi försöker beräkna skärningar med objekt som ligger utanför synpyramiden.
Realtidsförsök: Parallellism, algoritmer, lägre upplösning vid rörelse, ny hårdvara.

Ett strålföljningsprogram PovRay 1(3)

Finns i såväl UNIX- som PC-version (snyggare användargränssnitt). Fritt. Det använder sig av ett scenbeskrivningsspråk. Parametrarna har namn som påminner om de vi mött, men betydelsen är inte alltid lika. En utmärkt manual finns. Koordinatsystemet är vänsterorienterat - och inte som hittills högerorienterat - om du börjar undra. Nu version 3.7.

Om scenbeskrivningen ligger i filen `Ex3.pov` (i `$DG/EXEMPEL_MB`) startar man programmet med

```
povray +Ex3.pov +X +D +W512 +H512
```

Då växer en fotorealistisk bild fram i sakta mak. Den sparas automatiskt som `Ex3.png` (från 3.6; tidigare användes det gammaldagsa formatet tga)). P g a rättighetsmekanismer bäst att vara i den mapp där pov-filen finns.

Vill man ha radiositet lägger man i version 3.7 till en rad `radiosity{ }` i `global_settings`. Allt tar då längre tid.

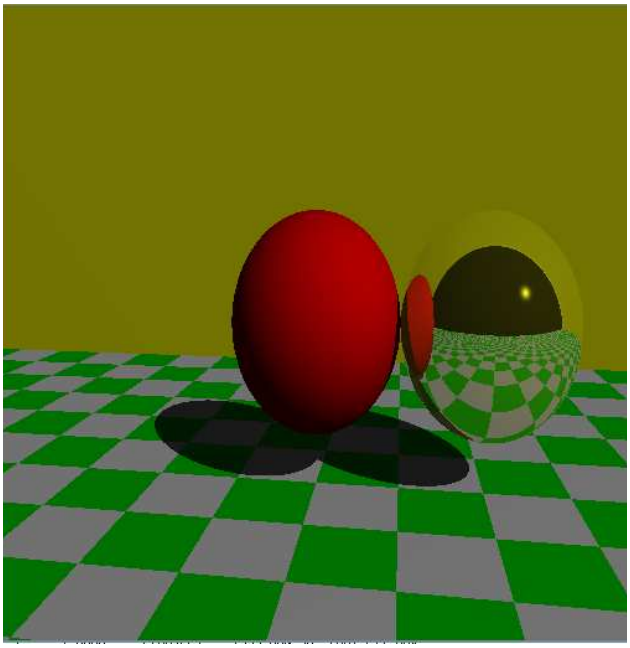
Exempel (nästa OH**)**: Scen med två klot på ett plan. Bild efter koden.

Ett strålföljningsprogram PovRay 2(3)

```
> cat Ex3.pov
// En scen för PovRay
#include "colors.inc"
global_settings { assumed_gamma 2.2 }
// En kamera i (1,1,-7) som tittar mot (0,0,0)
camera { location <1, 1, -7>
         look_at <0, 0, 0>
         angle 56
}

// En vit ljuskälla snett och långt bakom kameran
light_source { <1000, 1000, -1000> White }

// Ett oändligt gult plan med normalen (0,0,1)
// och 6 enheter i den riktningen
plane { <0,0,1>, 6
        pigment {color rgb <1, 1, 0>}
}

// Ett horisontellt plan -1.5 under xy-planet
// med ett schackrutemönster
plane { <0, 1, 0>, -1.5
         pigment { checker Green, White }
}// En röd sfär i origo med radien 1
sphere { <0,0,0>, 1
         pigment { Red }
}

// En sfär med radien 1 och materialegenskaper
sphere { <2,0,0>, 1
         pigment { BrightGold }
         finish {
                 ambient .1  diffuse .1
                 specular 1 roughness .001
                 reflection .75 metallic
         }
}
```


Ett strålföljningsprogram PovRay 3(3)

Bilden



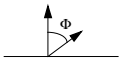
Exempel: Ett vitt ägg på ett bord med och utan radiositet (`Ex4R.pov` resp `Ex4.pov` i samma mapp).

Fotorealism: Radiositetsmetoden 1(3)

Vid strålföljning tar vi hänsyn till spegelreflektionen, men struntar helt i den diffusa reflektionen mellan objekt. Detta är inte korrekt. T ex kan ett vitt föremål placerat i ett rum med röda väggar få en rödaktig ton. **Radiositetsmetoden** tar hänsyn till den diffusa reflektionen mellan objekten och struntar i spegelreflektionen, men kan på olika sätt kombineras med strålföljning.

Metoden i stora drag

1. Förutsättningar



Alla ljuskällor och alla ytor är diffusa (dvs ingen spegelreflektion). Begreppet diffus innebär att ljuset sprids i de olika riktningarna med energin proportionell mot cos Φ, där Φ är vinkeln mot ytans normal. Intensiteten blir därmed (se figur på OH om diffus reflektion) lika oberoende av riktning, som vi antog i samband med belysningsmodell-resonemanget. Maximalt ljusflöde fås alltså i normalens riktning och inget alls längs ytan.

Ljuskällorna kan vara utbredda. Ytorna kan vara krökta. Scenen skall vara **sluten**.

2. De olika stegen

1. Samtliga ytor delas in i mindre delar A_1, A_2, A_3, ..., A_N. Intensiteten B_i på en sådan yta A_i antages vara konstant, vilket naturligtvis innebär en approximation och betyder att indelningen måste göras adaptivt, dvs så att delarna är små där intensiteten kan förväntas variera raskt.

Fotorealism: Radiositetsmetoden 2(3)

2. Nu beräknas alla B_i genom att man
   först bestämmer N^2 formfaktorer F_{ij}, $1 \le i,j \le N$, som ger ett rent geometriskt samband mellan ytorna A_i och A_j.
   sedan löser ett linjärt ekvationssystem med N ekvationer och N obekanta.
3. Nu kan observatören placeras godtyckligt utan att beräkningarna i 3 behöver göras om. Man löser bara synlighetsproblemet.
4. Ytorna ritas upp med flat toning eller Gouraud-toning.

3. Ekvationssystemet

Intensiteten B_j hos ytan A_j mäts i W/m^2. Om A_j får beteckna inte bara själva ytan utan även dess area, betyder det att den totala ljuseffekten från ytan A_j är B_jA_j. B_jA_j byggs upp av två komponenter. Dels kan ytan vara en ljuskälla med den emitterande intensiteten E_j (också med sorten W/m^2). Dels reflekterar den diffust ljus som når ytan A_j från andra ytor. Reflektionskoefficienten, som är en materialkonstant och antages vara konstant över A_j, betecknar vi med ρ_j. Detta innebär att

$$B_jA_j = E_jA_j + \rho_j \sum_{i=1}^{N} EnerginSomNarA_jFranA_i$$

Termen innanför summatecknet kan skrivas $F_{ij}B_iA_i$, där F_{ij} anger hur stor del av den totala energin från A_i som når A_j. Vi kommer senare att visa att $A_iF_{ij}/A_j = F_{ji}$, varav

$$B_j = E_j + \rho_j \sum_{i=1}^{N} F_{ji}B_i$$

Detta är vårt ekvationssystem (kallas radiositetsekvationen), som kan lösas med t ex gausselimination eller något iterativt förfarande (Gauss-Seidel brukar nämnas; systemet är diagonaldominant dvs den

Fotorealism: Radiositetsmetoden 3(3)

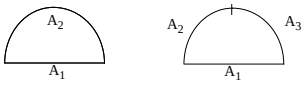
metoden konvergerar; diagonaldominansen följer av att $\sum_j F_{ij} \le 1$

Man kan ange komplicerade formler för formfaktorerna, t ex

$$F_{ij} = \frac{1}{\pi A_i} \int_{A_i} \int_{A_j} H_{ij} \frac{\cos\Phi_i \cos\Phi_j}{r^2} dA_i dA_j$$

men i praktiken måste de ändå approximeras.

4. Ett exempel



I båda fallen är ytorna i praktiken för stora och borde delats in i mindre. Men vi vill bara få ett begrepp om vad formfaktorerna innebär.
I vänstra fallet når allt ljuset från ytan A_1 ytan A_2, dvs F_{12}=1 och F_{11}=0. Sambandet mellan de båda formfaktorerna F_{ij} och F_{ji} ger sedan $F_{21}=A_1F_{12}/A_2=A_1/A_2$ (där nu A_i betecknar area, varvid A_1=ytan av en enhetscirkel=π och A_2=ytan av en enhetshemisfär=2π). Resten av ljuset från A_2 måste nå A_2 själv, varför $F_{22}=1-A_1/A_2$. I högra fallet ger symmetrin att $F_{12}=F_{13}=1/2$ och som förut är F_{11}=0. Härur får vi som förut $F_{21}=A_1F_{12}/A_2=0.5A_1/A_2$. Men att bestämma F_{22} och F_{23} (och därmed F_{32} och F_{33}) förefaller inte helt lätt, så vi ger upp.

Radiositetsmetoden är en dyr metod, varför många system, bl a *Pov-Ray*, använder en approximativ metod, som innebär att det från varje bildpunkt skickas strålar i olika riktningar som undersöker om det finns näraliggande ytor som skall ge ett "diffust" bidrag.

Kurvor och ytor

Det finns två huvudmetoder för konstruktion av kurvor och ytor:

- Olika former av **splines**, framför allt B-splines och NURBS. Utgående från ett antal punkter sätts ett uttryck för kurvan eller ytan upp på parameterform. Kurvan ritas utifrån denna parameterframställning. Upplevs av de flesta som rätt matematiskt och krångligt. Stöds av OpenGL. Behandlas i det separata pappret *Kurv- och ytapproximation*, samt i OpenGL-häftet, avsnitt 24. Sammanfattas och kompletteras med ett antal följande OH.
- **Uppdelningsmetoder** (eng. subdivision). Ytligt sett mera praktiskt. Utgående från ett antal punkter inför man successivt nya punkter och modifierar samtidigt de gamla. Därefter ritas kurvan genom ett polygontåg genom punkterna. Man får automatiskt sina objekt i flera upplösningar. Att analysera metoderna matematiskt är däremot inte lätt. Inget direkt stöd i OpenGL. Liksom alla kommersiella modelleringsprogram har *Blender* och *Art Of Illusion* verktyg (om än begränsade) för uppdelning. Användningsområdet är ytor. Blev populära i slutet av 1990-talet. Vi ger en kort introduktion i form av några OH.

I båda fallen skiljer man på **interpolerande** kurvor/ytor och **approximerande** kurvor/ytor. Vi ägnar oss mest åt den senare typen. Kurvan/ytan går då inte säkert igenom de olika styrpunkter som man utgår ifrån.

B-splines: Sammanfattning 1(2)

Givet: n+1 punkter $\mathbf{P}_0$, $\mathbf{P}_1$, ..., $\mathbf{P}_n$.

Approximera med en kurva som interpolerar startpunkten $\mathbf{P}_0$ och slutpunkten $\mathbf{P}_n$ och styrs av övriga punkter. Kurvan skall vara styckevis sammansatt av tredjegradskurvor. Resultatet blir en kurva med kontinuerlig andraderivata.

Allmänt behövs n-2 kurvsegment $\mathbf{P}_i(t)$, $t_i \le t \le t_{i+1}$, i=1,..., n-2. Segmenten $\mathbf{P}_{i-1}(t)$ och $\mathbf{P}_i(t)$ går ihop vid $t=t_i$, som kallas **skarv** (eng knot). Speciellt sätter vi t_1=0 (första segmentets start) och vid likformiga B-splines t_i=i-2, dvs t_{n-1}=n-2 (sista segmentets slut).

Man kan skriva

$$\boldsymbol{P}_i(t) = B_{i-1}(t)\boldsymbol{P}_{i-1} + B_i(t)\boldsymbol{P}_i + B_{i+1}(t)\boldsymbol{P}_{i+1} + B_{i+2}(t)\boldsymbol{P}_{i+2}$$

I inre punkter gäller i likformiga fallet $B_i(t) = B(t-t_i)$, där

$$B(t) = \begin{cases} \frac{1}{6}(2-|t|)^3 & 1 \le |t| \le 2 \\ \frac{1}{6}[1 + 3(1-|t|) + 3(1-|t|)^2 - 3(1-|t|)^3] & |t| \le 1 \\ 0 & 2 \le |t| \end{cases}$$

B-splines: Sammanfattning 2(2)

Allmänt är basfunktionen $B_i(t)$, $0 \le i \le n$, bestämd av **skarvföljden** (eng. knot vector) $[t_{i-2},t_{i-1},t_i,t_{i+1},t_{i+2}]$. Den är "centrerad" kring t_i (i likformiga fallet t_i=i-1) och 0 utanför intervallet $[t_{i-2},t_{i+2}]$. För detta behövs dock 3 extra skarvar $t_{-2} = t_{-1} = t_0 = 0$ före de andra och 3 extra $t_n = t_{n+1} = t_{n+2}$ =n-2 i slutet. $B_0(t)$ bestäms då av [0,0,0,0,1], medan $B_1(t)$ bestäms av [0,0,0,1,2] (om antalet punkter är minst 5) och $B_2(t)$ bestäms av [0,0,1,2,3]. På motsvarande sätt i den andra änden. Den ursprungliga skarvföljden om n-1 värden $[t_1,t_2,...,t_{n-1}]$ utökas på detta vis med 6 skarvar, dvs omfattar totalt (n+1)+4 skarvar eller 4 mer än antalet punkter. För interpolation i ändpunkterna skall de fyra första och de fyra sista i den nya följden vara sinsemellan lika.
Motsvarande NURBS-approximation är

$$\boldsymbol{P}_i(t) = \frac{\sum_{j=i-1}^{i+2} w_j B_j(t) \boldsymbol{P}_j}{\sum_{j=i-1}^{i+2} w_j B_j(t)}$$

Ett annat allmännare sätt att uttrycka basfunktionerna är med en **rekursionsformel** (detaljer i småskriften *Kurv- och ytapproximation*) över gradtalet (våra B_i är $B_{3,i}$, $B_{0,i}(t) = 1$ för $t_{i-1} \le t < t_i$ och 0 för övrigt).

$$B_{k,i}(t) = \frac{t - t_v}{t_{v+k} - t_v} B_{k-1,r}(t) + \frac{t_h - t}{t_h - t_{h-k}} B_{k-1,r+1}(t)$$

Med denna klaras godtyckligt gradtal och godtyckliga intervall-längder bekvämt, men det blir ju litet mer att göra för datorn. Och vi har inte härlett formeln.

OpenGL: Kubiska B-Splines och NURBS

OpenGL (snarare GLU) har stöd för B-splines med godtyckligt gradtal och de allmännare NURBS=**N**on-**U**niform **R**ational **B**-**S**plines (skämtsamt Nobody Understands Rational B-Splines).
OBS! Alla vektorer skall vara av float-typ (double duger inte).

- **Arbetsarea**:
```
GLUnurbsObj* theNurb;    (Java: long theNurb)
theNurb =  gluNewNurbsRenderer();
```
- **Styrpunkter**:(3->4 om NURBS)
```
GLfloat styrpunkter[antal_pkter][3];
// alt. GLfloat styrpunkter[3*antal_pkter];
```
- **Skarvar**:
```
int antal_skarv_punkter = antal_punkter+4;
GLfloat skarvar[antal_skarv_punkter] = {0,0,0,0,1,.. };
```
- **Vikter i NURBS-fallet**:
```
GLfloat vikter[antal_pkter];
```
- **Uppritning**:
```
gluBeginCurve(theNurb);
    gluNurbsCurve(theNurb, antal_skarv_punkter,
          skarvar,
           3, // avstånd i GLfloats mellan styrpunkter,
              // dvs 4 i NURBS-fallet
           styrpunkter,
           4, // gradtalet+1
           GL_MAP1_VERTEX_3 // eller motsv
    );
gluEndCurve(theNurb);
```

Exempel 20 i OGL-häftet.

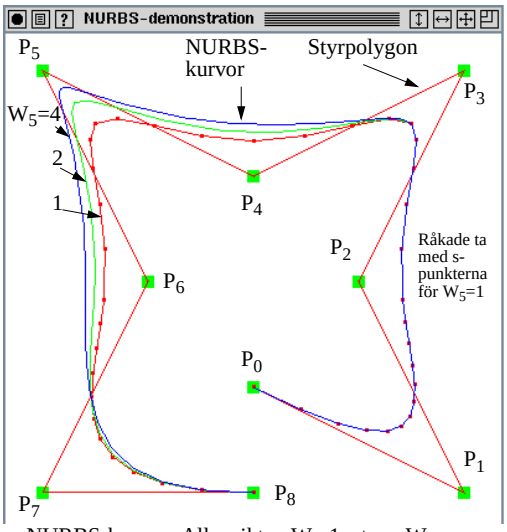
Exempel 20 i OGL-häftet med JOGL

JOGL har av någon anledning missat stödet för B-splines/NURBS, varför jag inte bifogar någon kod. Fungerade däremot i en av föregångarna, nämligen *OpenGL for Java*. Officiellt anges att man kommer att skriva om den del av GLU-koden som behövs i Java, i stället för att försöka anropa motsvarande kod skriven i C. Men man har inte lyckats hitta någon som är villig att göra jobbet. Att skriva en klass för sådana kurvor och ytor är inte svårt, men skall den ha lika många metoder som i GLU och utnyttja det stöd som OpenGL har för Bezierkurvor blir det värre. F ö frodas JOGL och i kombination med kommande Java 6.0 (våren 2006 kanske; betaversion går att hämta) fungerar OpenGL bra i Swing-fönster (GLJPanel; se sid 52 Anmärkningar i OpenGL-häftet).

C#/Tao verkar inte heller klara B-splines/NURBS. Det går visserligen att kompilera program med sådana, men vid körningen skapas inte de objekt som behövs utan man får nollreferenser. Tao tycks för övrigt stå still just nu.

NURBS i OpenGL 1(2)

Se OpenGL-häftet och kurvskriften. Här litet till.



Kubiska NURBS-kurvor. Alla vikter W_i=1 utom W_5 som varierar. Större värde ökar punktens attraktionskraft. Samplingspunkterna (se separat OH) råkade komma med för W_5=1.

NURBS i OpenGL 2(2)

Utdrag ur koden

```
Point PV[50]; // 2D-punkter
int Nr_Of_Points=9;
GLfloat CtrlPoint[Nr_Of_Points][4];
GLfloat W[] = {1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0};
GLfloat t[] = {0,0,0,0,1,2,3,4,5,6,6,6,6};

// Bilda 3D-data i homogena koordinater
for (i=0; i<Nr_Of_Points; i++) {
  CtrlPoint[i][0]=W[i]*PV[i].x;
  CtrlPoint[i][1]=W[i]*PV[i].y;
  CtrlPoint[i][2]=0.0;
  CtrlPoint[i][3]=W[i];
}
// Ritningen (när alla vikterna lika)
gluBeginCurve(theNurb);
         gluNurbsCurve(theNurb,Nr_Of_Points+4,t,4,
         (GLfloat*)CtrlPoint,4,GL_MAP1_VERTEX_4);
gluEndCurve(theNurb);

// Ändra en av vikterna
W[5] = 2;
// Upprepa
```

Att notera betr parametrarna till *gluNurbsCurve*:

4:e parametern är nu 4 eftersom varje punkt består av 4 floats.

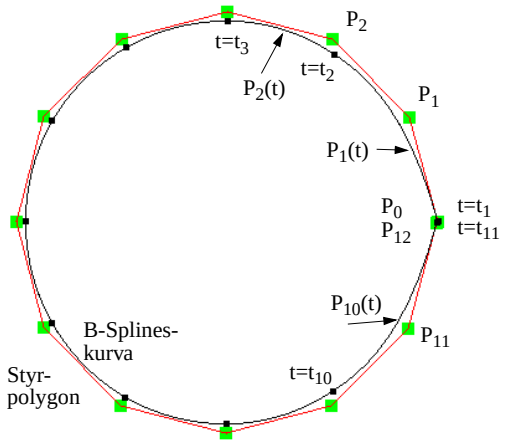
I 5:e parametern står `(GLfloat*)` enbart för att hindra en varning från kompilatorn.

6:e parametern är som vanligt gradtalet + 1 (gradtalet=3 eftersom det handlar om kubiska NURBS).

7:e parametern är `GL_MAP1_VERTEX_4` eftersom vi nu arbetar med homogena koordinater.

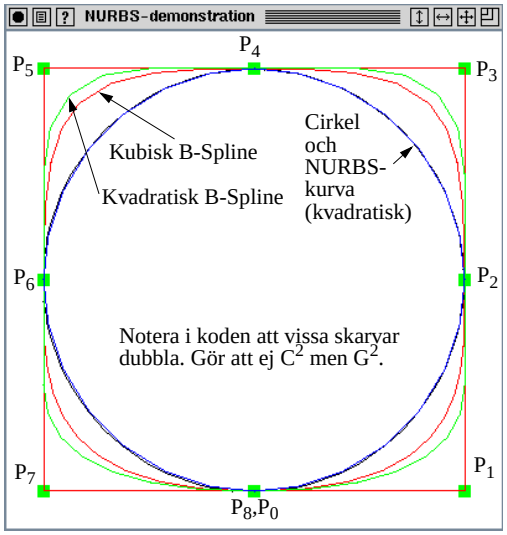
Cirkelapproximation med B-splines i OpenGL

Vi väljer 12 (13) punkter ekvidistant belägna på cirkelns omkrets. Dessa är markerade med större (gröna) fyrkanter. Approximation med kubiska B-splines (med interpolation i start- och slutpunkt) ger en kurva som
a) är "spetsig"
b) ligger innanför den riktiga cirkeln



Spetsen kan förbättras genom att man t ex inför flera identiska start- och slutpunkter. Med NURBS (se följande OH) kan man beskriva cirkeln exakt.

Cirkelapproximation med NURBS i OpenGL

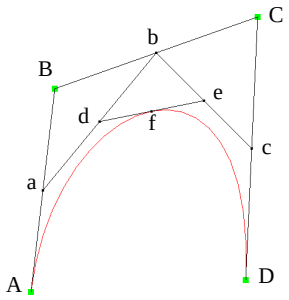



```
GLfloat t[] = {0,0,0,1,1,2,2,3,3,4,4,4};
GLfloat w = 1.0/sqrt(2.0);
GLfloat W[] = {1.0, w, 1.0, w, 1.0, w, 1.0, w, 1.0};
// Som förut och sedan ritning med kvadratiska NURBS
gluBeginCurve(theNurb);
            gluNurbsCurve(theNurb,Nr_Of_Points+3,t,4,
            (GLfloat*)CtrlPoint,3,GL_MAP1_VERTEX_4);
gluEndCurve(theNurb);
```

Det finns flera andra sätt att välja punkter. Man kan klara sig utan ett par av de vi valt. Se t ex Les Piegl: The Nurbs Book, Springer 1997.

Uppdelningsmetoder 1(7)

Vi börjar med att nämna de Casteljau's[1] geometriska metod för konstruktion av en enstaka Bezier-kurva.



I figuren (`GL_BEZIER2005.c`) har vi fyra punkter A, B, C och D och är alltså ute efter en kubisk Bezier-kurva P(t). Vi konstruera punkten f, som är P(0.5), genom att först införa mittpunkter a, b och c på tre av styrpolygonens sidor. Därefter punkterna d och e som mittpunkter på linjerna ab respektive bc. Slutligen f som mittpunkt på linjen de. Processen kan upprepas. T ex för att beräkna P(0.25) arbetar vi på samma sätt på styrpolygonen Aadf. En fördel med metoder av detta slag är att vi har geometrisk kontroll. Hade t ex polygonen Aadf varit mycket smal hade vi kanske kunnat nöja oss med ett streck från A till f.

I själva verket kan man med metoden konstruera vilken som helst punkt på kurvan genom att göra delningen litet annorlunda. En motsvarighet till detta för B-splines är de Boor's metod, som dock är mycket omständligare.

1. Paul de Casteljau och Pierre Bezier var bilingenjörer. Den förre vid Peugot och den andre vid Renault. Båda jobbade med Bezier-kurvor utan att känna till varandras arbete.

Uppdelningsmetoder 2(7)

Låt oss nu titta på en uppdelningsmetod för kurvor. Precis som i splines-fallet startar vi med n+1 punkter $\mathbf{P}_0$, $\mathbf{P}_1$, ..., $\mathbf{P}_n$. och vill ha en kurva som interpolerar första och sista punkten och går i närheten av de andra.

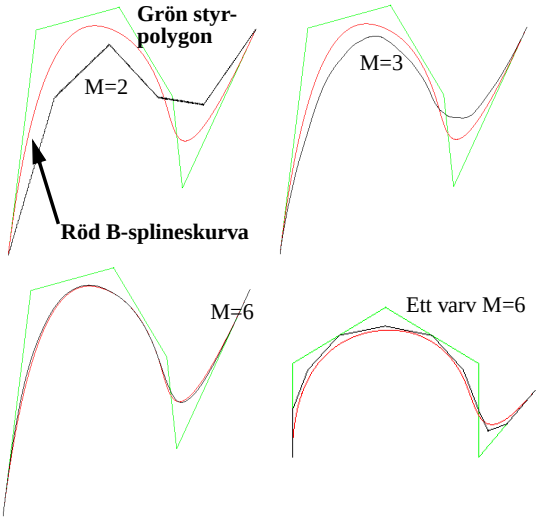
Mittpunktsmetoden innebär ett upprepande av följande steg:
1. Inför nya punkter $\mathbf{Q}_{2i+1}$, i =0, 1, ..., mittemellan de tidigare, dvs
   $\mathbf{Q}_{2i+1} = (\mathbf{P}_{i+1} + \mathbf{P}_i)/2$.
2. Modifiera de ursprungliga punkterna nu kallade $\mathbf{Q}_{2i}$, i = 0, 1, ..., enligt
   $\mathbf{Q}_{2i} = (\mathbf{P}_{i-1} + (M-2)\mathbf{P}_i + \mathbf{P}_{i+1})/M$
3. Låt $\mathbf{P}$ beteckna den nya punktvektorn $\mathbf{Q}$.

För varje steg närmar sig punkterna alltmer en kurva (kan bevisas). Dock inte i allmänhet i ett ändligt antal steg.

De två stegen är typiska för alla uppdelningsmetoder. Först har man ett **förfiningssteg**, där nya punkter tillförs och sedan ett **utjämningssteg** som innebär att vissa punkter modifieras.

I figuren (med programmet `GL_BSPLINES_2005.c`) nedan har vi en kubisk B-splineskurva (röd) hörande till styrpunkterna (6 st) som är markerade med en (grön) styrpolygon. Vi har dessutom en svart kurva från mittpunktsmetoden med olika M-värden. I samtliga fall lika många varv i mittpunktsmetoden. För M=6 noterar vi att kurvan ser ut att sammanfalla med B-splines-kurvan. Att det är så kan bevisas (jag känner mig personligen inte helt övertygad om den visuella bevisningen i figuren).

Uppdelningsmetoder 3(7)



Längst ner till höger visas ett enda varv för metoden.

På adressen `http://www.multires.caltech.edu/teaching/demos/` hittar du Java-appletar som beskriver och demonstrerar två andra uppdelningsmetoder för kurvor: Chaikin (approximerande) och 4-punktsmetoden (interpolerande).

Uppdelningsmetoder 4(7)

Vi övergår nu till yt-fallet för vilket det finns en uppsjö metoder. Först en bild hämtad från SIGGRAPHS kurs 1998 om "Subdivision" (avsnittet *Subdivision Surfaces in Character Animation* av Tony DeRose m fl från Pixar Animation Studios) .

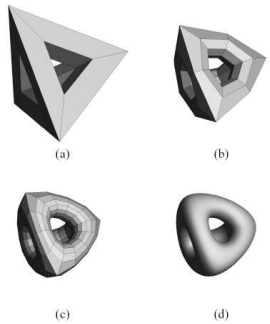


Figure 3: Recursive subdivision of a topologically complicated mesh: (a) the control mesh; (b) after one subdivision step; (c) after two subdivision steps; (d) the limit surface.

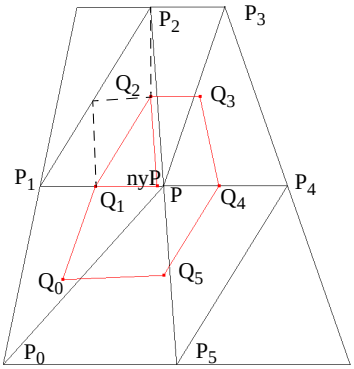
Metoden i figuren kallas *Catmull-Clark* och arbetar med fyrhörningar och är approximerande.

Låt oss se på en metod *Loop* (upphovsmannen heter Loop), som arbetar med trianglar i anslutning till följande figur.

Uppdelningsmetoder 5(7)

Förfiningssteget: För varje styrpunkt P inför ytterligare styrpunkter Q_i som bestäms av dels P, dels de punkter P_i som P är förbunden med.
Om P är förbunden med 6 punkter:
$Q_i = (3P + 3P_i + P_{i-1} + P_{i+1})/8$, varvid $P_{-1} = P_5$, $P_6 = P_0$.

Utjämningssteget: Varje styrpunkt P (ej de nya) modifieras enligt (fallet 6 även nu)
$P = (10*P + P_0 + ... + P_5)/16$



Vi inser att specialåtgärder måste vidtas vid kanterna och när styrpunkterna inte har 6 förbundna. I figuren har vi med streckade linjer markerade delar av ett par trianglar som punkten P_2 ger upphov till.

Uppdelningsmetoder 6(7)

Ett exempel (kunde varit roligare om de boolska operationerna fungerat som de borde) med *Art Of Illusion*. Vi startar med en kub. Väljer den. Konverterar till triangelnät med **Object/Convert to Triangle Mesh**. Påbörjar redigering av kuben med **Object/Edit Object**, som visar upp ett nytt fönster med enbart vår kub. Väljer FACE längst ned till vänster i det fönstret. Markerar med MK3 de fyra trianglarna på ovansidan. Nu ser det ut som i figur 1. Gör **Mesh/Bevel/Extrude Selection** två gånger. Därefter väljer vi med **Mesh/Smoothing Method Approximate** (som - inbillar jag mig - ger en approximerande uppdelning; metoden anges ej). Nu enligt figur 2.

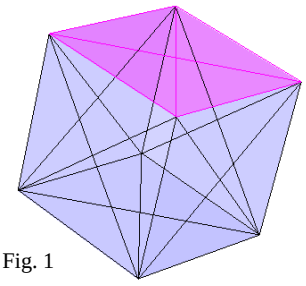


Fig. 1

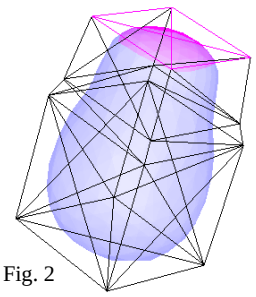


Fig. 2

Vi lämnar kubredigeringen och ser då i det vanliga fönstret något som liknar figur 3 eller figur 4 (beroende på visningssättet i **Scene/Display Mode**). Med **File/Export/VRML** (välj bort komprimeringen) får vi en massa triangeldata för det skapade objektet i en fil.