Debreceni Egyetem
Informatika Kar

CSG MODELLEK TESSZELLÁCIÓJA

Témavezető:
Dr. Schwarcz Tibor
egyetemi adjunktus

Készítette:
Szakács László
programtervező matematikus

Debrecen
2008

Tartalomjegyzék

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőmnek, Dr. Schwarcz Tibornak tanácsaiért és ötleteiért, amelyekkel irányította ezen diplomamunka létrejöttét, barátaimnak, akiktől sok szakmai segítséget és erkölcsi támogatást kaptam, és elsősorban menyasszonyomnak, akinek bíztatása és lelkesedése a legfőbb mozgatóereje volt munkámnak.

Bevezetés

A háromdimenziós testmodellezés az élet sok terültén nagyon fontos szerepet játszik napjainkban. A mérnökök a régi, papíron történő tervezés helyett a számítógép adta lehetőségeket kihasználva terveiket modern szoftverekkel készítik el, ezzel sokkal magasabb színvonalú, látványosabb, és könnyebben kezelhető modelleket hozva létre. A mai animációs filmek elkészítése elképzelhetetlen lenne modern technikai háttér, és komoly modellező szoftver nélkül. Az orvosi informatikában is egyre nagyobb jelentősége van a számítógépes modelleknek.

A konstruktív tömörtest geometria (CSG) a számítógépi grafikának azon területe, ahol a tervezők egyszerű testekből – primitívekből – CSG halmazműveletek segítségével (unió, metszet, különbség) összetett testeket hoznak létre. Ezek ún. regularizált halmazműveletek, ami azt jelenti, hogy eredményként nem jöhetnek létre degenerált, alacsonyabb dimenziószámú testek. Jelen esetben, mivel 3 dimenzióban történik a tervezés, az eredmény is mindig 3 dimenziós test lesz, soha nem lehet benne 1 vagy 2 dimenziós alakzat (pont, szakasz, vagy sokszög). Egy ilyen modell mindig leírható egy ún. CSG-fával. Ez egy bináris fa, amelynek az adott test a gyökere, a levélelemei egyszerű testek (kocka, gömb, kúp, tórusz, stb.), egy közbülső elem pedig úgy jön létre, hogy a két gyermekére alkalmazunk valamilyen halmazműveletet. A kapcsolat egy test és az őt leíró fa között nem egyértelmű, ez azt jelenti, hogy egy objektumot nem csak egyféleképpen hozhatunk létre. Például egy téglatest létrejöhet két kisebb kocka uniójaként, de két nagyobb kocka metszeteként is. A faelemekben eltárolhatjuk az adott elemen elvégzett transzformációkat, ezáltal teljes képet kaphatunk egy összetett test létrejöttének folyamatáról.

Egy felület tesszellációján azt a folyamatot értjük, amikor azt egy sokszöghálóval teljese egészében lefedjük. Jelen esetben ez azt jelenti, hogy meghatározzuk azt a háromszöghálót, amely lefedi az adott CSG test felszínét. Ez a következőképpen történik: az alaptestek hálóját ismerjük, a halmazműveletek után létrejött összetett testek hálóját pedig úgy határozzuk meg, hogy az alaphálókat alkotó háromszögeket végigvezetjük azokon a halmazműveleteken, amelyeken az eredeti testeket. Ennek megoldása számos problémát vet fel, mint például két térbeli háromszöglap egymással történő metszése, vagy a megmaradó és eldobandó lapok kiválogatása, stb.

Elméletben a 3D-s testeket elképzelhetjük úgy, mint pontok halmazát, ilyenkor nagyon egyszerű közöttük definiálni a halmazműveleteket. Azonban a gyakorlati alkalmazásokban egy testet mindig sokszöglapokból építünk fel, egy sokszöglapot pedig a csúcsaival adunk meg. Egy ilyen reprezentáció mellett már nem olyan könnyű implementálni a műveleteket.

Ezen diplomamunka célja egy olyan szoftver létrehozása, amelynek segítségével a CSG halmazműveletek könnyen elvégezhetők. A fejlesztéskor elsődleges szempont volt a könnyű kezelhetőség, a többféle megjelenítési mód, illetve a létrehozott testek tárolhatósága későbbi felhasználás céljából. Ezeket a célokat rendre sikerült megvalósítani: a könnyű kezelhetőséget biztosítja a grafikus ablakozó rendszeren alapuló, eseményvezérelt felhasználói felület, melyben az egérrel és néhány billentyűvel mindent el lehet végezni; a program négyféle megjelenítési módot használ, a testet alkotó ponthalmaz kirajzolásától egészen a Gouraud-árnyalásig; a modelleket lemezre lehet menteni, és bármikor visszatölteni, későbbi felhasználás céljából.

A mai fejlett tervező szoftverek, CAD programok is ismerik a CSG műveleteket, de gyakran meglehetősen bonyolult őket használni, és mivel olyan sok egyéb funkciójuk van, a felhasználóknak meg kell tanulniuk kezelni őket. Az én célom az volt, hogy azok is tudják használni ezt a programot, akik nem jártasak más tervezőprogramokban. A fejlesztéshez a Microsoft Visual Studio 2005 keretrendszert és a C# programnyelvet használtam, illetve a megjelenítéshez egy C#-hoz készült OpenGL motort, a CsGL-t. A program futtatásához szükséges a Microsoft .NET Framework 2.0, a grafikus motorhoz pedig a „csgl.dll” és a „csgl.native.dll” nevű fájlok, amelyek megtalálhatók az indítási könyvtárban.

A diplomamunka első, nagyobb részében ismertetem a szoftver által használt alaptesteket, transzformációkat, megjelenítési módokat, illetve a halmazműveletek elvégzésének algoritmusát. Röviden kitérek néhány implementációs kérdésre, úgymint a fontosabb alaposztályok és felületek ismertetése, valamint az OpenGL motor C#-ban történő használata.

A második rész tulajdonképpen egyetlen fejezetből áll, és a program használati kézikönyveként is felfogható. Itt helyet kap a menürendszer ismertetése, a fontosabb menüpontok; az ablakok felépítésének logikája; egyszerű testek létrehozása, ezek transzformálása, a különböző megjelenítési módok használata; a transzformált alaptestek összekapcsolása halmazműveletekkel; a modell mentése, betöltése és végül a gyorsbillentyűk használata.

1. Alapfogalmak

1.1. Pontok, vektorok

A diplomamunka további részeiben a „pont” fogalom mindig a háromdimenziós euklideszi tér egy pontját jelenti, amelyet 3 koordinátával írhatunk le: x, y, z. Ugyanezt a pontot jellemezhetjük az origóból a pontba mutató helyvektorral is. Implementáció szempontjából a pont és a vektor ugyanazon adatszerkezettel kezelhetők, hiszen mindkettőt három darab lebegőpontos értékkel adhatjuk meg. Legyen $P(x_1, y_1, z_1)$ és $Q(x_2, y_2, z_2)$ két vektor, ezekkel a következő műveletek végezhetők:

- Két vektor összege: $P + Q = (x_1 + x_2, y_1 + y_2, z_1 + z_2)$
- Két vektor különbsége: $P - Q = (x_1 - x_2, y_1 - y_2, z_1 - z_2)$
- Vektor skalárszorosa: $dP = (dx_1, dy_1, dz_1)$, ahol $d \in R$
- Két vektor skaláris (belső) szorzata: $(P, Q) = x_1x_2 + y_1y_2 + z_1z_2$
- Két vektor vektoriális szorzata: $P \times Q = (y_1z_2 - z_1y_2, z_1x_2 - x_1z_2, x_1y_2 - y_1x_2)$
- Vektor hossza (normája): $\|P\| = \sqrt{x_1^2 + y_1^2 + z_1^2}$
- Vektor normáltja: $P' = \frac{P}{\|P\|}$

1.2. Mátrixok

Az *n* sorból és *m* oszlopból álló számtáblázatot *nxm*-es mátrixnak nevezzük. Az *1xm* típusú mátrixokat sorvektoroknak, az *nx1* típusúakat pedig oszlopvektoroknak nevezzük. A mátrixok közötti műveletek a jól ismert módon történnek, az összeadás elemenként, a szorzás pedig sor-oszlop kompozícióval. A transzformációknál és leképezéseknél szükség van arra, hogy pontokat tudjunk szorozni mátrixokkal. Ezt egyszerűen elvégezhetjük, ha beírjuk a pontot egy oszlopvektorba, majd az adott mátrixszal balról megszorozzuk.

Implementációs szempontból az egységes kezelhetőség érdekében a transzformációk leírása 4x4-es mátrixokkal történik. A pontok megadása inhomogén alakban történik, a kettő közötti átmenet azonban könnyen biztosítható, az alábbi módon:

Legyen adott egy $P(px, py, pz)$ pont, valamint egy 4x4-es M mátrix. A $Q = M * P$ szorzást a következőképpen implementálhatjuk:

$$qx = px * M[0,0] + py * M[0,1] + pz * M[0,2] + M[0,3]$$

$$qy = px * M[1,0] + py * M[1,1] + pz * M[1,2] + M[1,3]$$

$$qz = px * M[2,0] + py * M[2,1] + pz * M[2,2] + M[2,3]$$

A pontok homogén koordinátás reprezentációjától a plusz memóriaigény és a Descartes-alakba történő állandó konverzió okozta sebességcsökkenés miatt tekintettem el, ugyanakkor a transzformációk és leképezések egységes kezelését megkönnyíti a 4x4-es mátrix reprezentáció. Mivel ezen mátrixok utolsó sora mindig [0, 0, 0, 1] alakú, így a fenti számítás eredménye ekvivalens azzal, mintha homogén alakban megadott pontokkal végeznénk el az $M * P$ szorzást, azaz helyes eredményt szolgáltat, és egy kicsivel gyorsabb a másik formánál.

2. Primitívek

2.1. A testek felépítése

A testmodellek reprezentációjához a B-Rep adatszerkezetet használtam. A CSG modellek ilyen módon történő tárolását hívják CSR-nek, azaz „Constructive Shell Representation”-nek. A B-Rep adatszerkezet három fő részből áll:

1. Pontok tömbje: egy pont az x, y, z koordinátájával adott.
2. Élek tömbje: egy él a kezdő- és végpontjával adott, ezek azonban nem konkrét háromdimenziós pontok, csak az első tömbben lévő pontok indexei.
3. Lapok tömbje: az élekhez hasonlóan egy lapot a csúcsainak indexei határoznak meg.

A modellezésnél praktikussági okokból háromszöglapokat használtam, ez megkönnyíti az algoritmusok implementálását és a láthatósági vizsgálatokat.

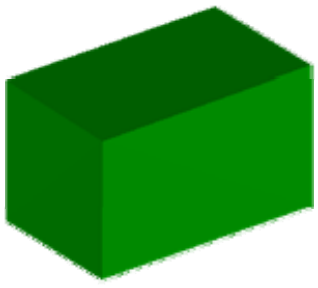
2.2. Alaptestek

A programban használt alaptestek, vagy más néven primitívek egyszerű geometriai objektumok, amelyeknek ismert a megadási módja. Ez kétféleképpen történik: vagy a test paraméteres egyenletrendszere segítségével (pl. gömb, tórusz esetén), vagy direkt módon megadjuk a pontokat, éleket, és lapokat (pl. téglatest, kúp, henger esetén). Alapértelmezés szerint a test középpontját a térbeli derékszögű koordinátarendszer origója jelenti, de megadhatunk más középpontot is. A különféle méreteket (szélesség, élhossz, sugár, stb.) mindig pixelekben kell érteni.

2.2.1. Téglatest

Ez a legegyszerűbb test, van szélessége, magassága és mélysége.

- ✓ Pontok száma: 8
- ✓ Élek száma: 12
- ✓ Lapok száma: 12



1. ábra Téglatest

2.2.2. Ellipszoid

Ezt a testet 5 paraméter határozza meg, ezek közül három az ellipszoid X, Y és Z koordinátatengely irányába eső tengelyét jelenti (*a, b, c*), míg a másik kettő a vízszintes és függőleges felosztását jelenti a felületének. Mivel ezt a testet paraméteres egyenlet írja le, az utolsó két paraméter (*hdiv, vdiv*) tulajdonképpen azt mondja meg, hogy hány részre osszuk a [0..2π) paramétertartományt. Az ellipszoidot leíró paraméteres egyenlet (koordinátákra lebontva):

$$x(u,v) = a\cos u\cos v$$
$$y(u,v) = b\cos u\sin v$$
$$z(u,v) = c\sin u$$

$$-\frac{\pi}{2} \le u \le \frac{\pi}{2};\ -\pi \le v \le \pi$$

- ✓ Pontok száma: $hdiv(vdiv-1)+2$
- ✓ Élek száma: $hdiv(2vdiv-1)$
- ✓ Lapok száma: $2hdiv(vdiv-1)$

2. ábra Ellipszoid

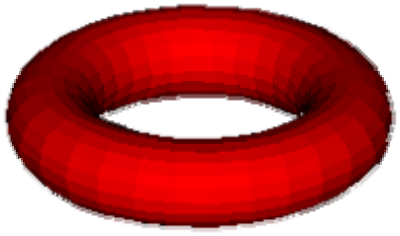
2.2.3. Tórusz

Ha egy kört megforgatunk egy olyan egyenes körül, amely vele egy síkban van, és nem metszi, akkor eredményül tóruszt kapunk. Ezt a testet 4 paraméter határozza meg: a tórusz középpontja és a perem középpontja közötti távolság (*R*), a perem sugara (*r*), illetve a vízszintes és függőleges felosztás (*hdiv*, *vdiv*), mint az ellipszoidnál. A paraméteres egyenlet:

$$x(u,v) = (R + r\cos v)\cos u$$
$$y(u,v) = (R + r\cos v)\sin u$$
$$z(u,v) = r\sin v$$

$$u,v \in [0,2\pi)$$

- ✓ Pontok száma: $vdiv * hdiv$
- ✓ Élek száma: $2vdiv * hdiv$
- ✓ Lapok száma: $2vdiv * hdiv$

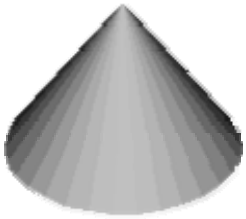


3. ábra Tórusz

2.2.4. Egyenes körkúp

Ezt a geometriai formát három paraméter határozza meg: az alapkörének a sugara, a kúp magassága, illetve, hogy hány részre osszuk fel az alapkört (*div*).

- ✓ Pontok száma: $div + 2$
- ✓ Élek száma: $3div$
- ✓ Lapok száma: $2div$



4. ábra Kúp

2.2.5. Egyenes körhenger

A hengernél ugyanazokat a paramétereket adhatjuk meg, mint a kúpnál: alapkör sugara, magasság és felosztás (*div*).

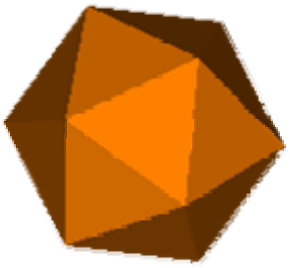
- ✓ Pontok száma: $2div + 2$
- ✓ Élek száma: $3div$
- ✓ Lapok száma: $4div$

5. ábra Henger

2.2.6. Ikozaéder

Ez a test a tervezőprogramokban általában nem tartozik az alaptestek közé, de egyszerűségénél fogva úgy gondoltam itt helyet kaphat. Mindössze egyetlen paramétert kell hozzá megadni, egy lapjának az élhosszát.

- ✓ Pontok száma: 12
- ✓ Élek száma: 30
- ✓ Lapok száma: 20



6. ábra Ikozaéder

3. Transzformációk

A program két egybevágósági transzformációt ismer: az eltolást, a forgatást, illetve egy affin transzformációt, a skálázást. Fontos, hogy ezeknek a transzformációknak a viszonyítási pontja mindig a térbeli koordinátarendszer origója, kivéve a forgatásnál, ahol megadhatunk saját forgatási tengelyt. Ezt különösen a skálázásnál kell figyelembe venni, mert ha előzőleg már elvégeztünk valamilyen transzformációt a testen, akkor az a skálázással torzulhat. Tetszőlegesen sok transzformáció alkalmazható egy testre, és ezek utólag is bármikor módosíthatóak, törölhetőek. A könnyebb és gyorsabb transzformálást gyorsbillentyűk segítik.

3.1. Eltolás

Ennél a transzformációnál meg kell adni az eltolás mértékét a térbeli koordinátarendszer X, Y és Z tengelye mentén. Ezután a program a test minden pontjának x, y, z komponenséhez hozzáadja a megfelelő értéket. Ezt a következő mátrixszal lehet leírni:

$\begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$, ahol *dx, dy, dz* rendre az X, Y és Z tengely mentén vett eltolásérték.

3.2. Forgatás

Itt választhatunk, hogy valamely koordinátatengely körül forgatjuk a testet, vagy saját forgatási tengelyt adunk meg. Az előbbi esetben csak a forgatás mértékét kell megadnunk fokokban (ez egy 0° és 360° közötti valós érték lehet), utóbbi esetben pedig két térbeli pont megadásával definiálhatjuk a forgatás tengelyét is. A forgatási transzformációt egy-egy mátrix írja le, ezt a mátrixot kell balról megszorozni az összes ponttal.

3.2.1. Forgatás valamely koordinátatengely körül

Legyen a forgatás szöge α.

Az X tengely körüli forgatás mátrixa: $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Az Y tengely körüli forgatás mátrixa: $\begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Az Z tengely körüli forgatás mátrixa: $\begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

3.2.2. Forgatás tetszőleges tengely körül

Adott két pont a térben, $P_0(x_0, y_0, z_0)$ és $P_1(x_1, y_1, z_1)$, amelyek meghatároznak egy szakaszt. Ezen szakaszt használva forgatási tengelyként, szeretnénk elforgatni egy pontot. Ehhez egy olyan transzformációt fogunk használni, amely először a forgatási tengelyt elmozgatja úgy, hogy egybeessen a Z tengellyel, majd a Z tengely körül elforgatja a pontot, végül az előbbi transzformáció inverzét használva visszamozgatja a pontot a végső helyére, a tengelyt pedig az eredeti helyére.

Legyen a $\underline{c}$ vektor a $\overrightarrow{P_0P_1}$ vektor normáltja, azaz

$$c_x = \frac{x_1 - x_0}{h}$$

$$c_y = \frac{y_1 - y_0}{h} \qquad \text{ahol } h = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2}$$

$$c_z = \frac{z_1 - z_0}{h};$$

Az első lépés egy eltolás, amely a forgatási tengely kezdőpontját az origóba viszi, ezt a következő mátrix írja le:

$$T = \begin{pmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Ezután az X tengely körül forgatunk α szöggel, ahol α a Z tengelynek és a $\underline{c}$ vektor YZ síkon vett merőleges vetületének a közbezárt szöge. Legyen d a $\underline{c}$ vektor YZ síkon vett merőleges vetületének hossza, vagyis $d = \sqrt{{c_z}^2 + {c_y}^2}$. Ekkor α szögfüggvényeit, amelyek a forgatási mátrixba kellenek, kiszámíthatjuk a következőképpen:

$$\cos\alpha = \frac{c_z}{d}; \qquad \sin\alpha = \frac{c_y}{d}$$

A X tengely körüli forgatás mátrixa pedig már ismert (*Rx*).

Most az Y tengely körül forgatunk β szöggel, amely a Z tengelynek és a $\underline{c}$ vektor XZ síkon vett merőleges vetületének a közbezárt szöge. β szögfüggvényeit hasonlóképpen számíthatjuk:

$$\cos\beta = d; \qquad \sin\beta = c_x$$

Ennek a forgatásnak a mátrixát jelöljük *Ry*-nal.

Most következhet a tényleges forgatás a Z tengely körül a megadott szöggel, majd elvégezzük az előző 3 transzformáció inverzét, persze fordított sorrendben, és kész. Mivel ezek a mátrixok ortogonálisak, így az inverzük megegyezik a transzponáltjukkal. A végső mátrix tehát a következőképpen alakul:

$$M = Rx^T Ry^T RzRyRx$$

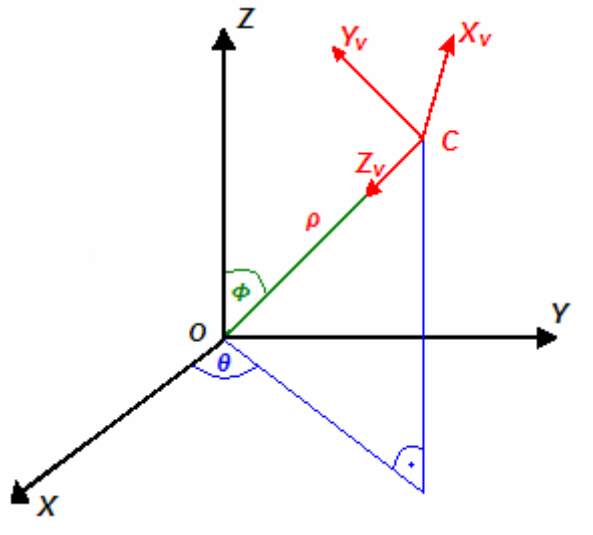
3.3. Skálázás

Ez egy olyan affin transzformáció, amely az objektumot az egyes tengelyek mentén eltérő mértékben nyújtja meg. Szintén leírható egy mátrixszal, amely $\begin{pmatrix} \lambda & 0 & 0 & 0 \\ 0 & \mu & 0 & 0 \\ 0 & 0 & \nu & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ alakú, ahol λ, μ, ν pozitív valós értékek.

4. Megjelenítés

Ez a folyamat több lépésből tevődik össze. Először a test pontjaira alkalmazzuk a megadott transzformációkat, majd elvégezzük a pontok leképezését. Az eredeti B-Rep modell eközben változatlan marad, a módosított modellt egy új adatszerkezetben tároljuk.

A leképezés alapja egy olyan centrális vetítés, amelyben a vetítés centruma az origóban van, és a pozitív Z tengely irányába néz. A képsík az XY síkkal párhuzamos sík, amely *d* távolságra van az origótól a Z tengelyen. Ebben a rendszerben egy *P(x, y, z)* pont *P'* képe a párhuzamos szelők tétele alapján könnyen megkapható:



7. ábra A kétféle koordinátarendszer

$$x' = x\frac{d}{z}; \qquad y' = y\frac{d}{z}$$

Ennek a továbbfejlesztett változata, amikor a kamerát egy gömb felszínén mozgatjuk. Jelöljük a kamerát C-vel; a pozícióját egyértelműen meghatározzák az ún. gömbi koordinátái, azaz az origótól való távolsága, vagyis a gömb sugara (ρ), a sugárnak a Z tengellyel bezárt szöge (ϕ), valamint a pozitív X tengelynek és a sugárnak az XY síkra eső vetülete közötti szög (θ). Megkülönböztetünk világ-koordinátarendszert (X, Y és Z tengelyekkel) és nézőponti koordinátarendszert (X_v, Y_v és Z_v tengelyekkel), amely a következőképpen néz ki: a kamera pozíciója a térben a nézőponti koordinátarendszer origója, a Z_v tengely pedig a kamerából a világ-koordinátarendszer origójába mutat. Y_v-t úgy adjuk meg, hogy Z_v, Z és Y_v egy síkba essen, X_v-t pedig úgy állítjuk elő, hogy Y_v-re és Z_v-re merőleges legyen, és ez a három tengely balsodrású rendszert alkosson. A célunk, hogy ezt a rendszert visszavezessük a fenti egyszerű centrális vetítési rendszerre. Ehhez meg kell adni azt a transzformációt, amely a nézőponti koordinátarendszert a világ-koordinátarendszerbe viszi át, vagyis a kamera a világ-koordinátarendszer origójába kerül, az X_v, Y_v, Z_v tengelyek pedig egybeesnek az X, Y, Z tengelyekkel. Ennek lépései a következők:

1. lépés: A nézőponti koordinátarendszert el kell tolni a $\underline{d} = \overrightarrow{OC}$ vektorral, amelynek komponensei a következők:

$$d_x = \rho \sin\phi \cos\theta$$
$$d_y = \rho \sin\phi \sin\theta$$
$$d_z = \rho \cos\phi$$

Mátrixos alakban: $T = \begin{pmatrix} 1 & 0 & 0 & -d_x \\ 0 & 1 & 0 & -d_y \\ 0 & 0 & 1 & -d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$

2. lépés: Z_v tengely körüli forgatás $90^\circ - \theta$ szöggel. Az ezt leíró mátrix:

$$R_z = \begin{pmatrix} \cos(90^\circ - \theta) & -\sin(90^\circ - \theta) & 0 & 0 \\ \sin(90^\circ - \theta) & \cos(90^\circ - \theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \sin\theta & -\cos\theta & 0 & 0 \\ \cos\theta & \sin\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3. lépés: X_v tengely körüli forgatás $-180^\circ + \phi$ szöggel:

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(-180+\phi) & -\sin(-180+\phi) & 0 \\ 0 & \sin(-180+\phi) & \cos(-180+\phi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -\cos\phi & \sin\phi & 0 \\ 0 & -\sin\phi & -\cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

4. lépés: Jobbsodrású rendszer kialakítása, azaz tükrözés az YZ síkra:

$$M_x = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Ezt a négy transzformációs mátrixot összevonhatjuk egybe, így a végső mátrix a következőképpen fog kinézni:

$$A = M_x R_x R_z T = \begin{pmatrix} -\sin\phi & \cos\phi & 0 & 0 \\ -\cos\phi\cos\theta & -\sin\phi\cos\theta & \sin\theta & 0 \\ -\cos\phi\sin\theta & -\sin\phi\sin\theta & -\cos\theta & \rho \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Ezek után a leképezés végrehajtása annyit jelent, hogy a modell minden csúcspontját megszorozzuk ezzel az A mátrixszal, majd alkalmazzuk rá az egyszerű centrális leképezést.

Bizonyos nézeteknél a megjelenítés felgyorsítása érdekében a kirajzolás előtt szükség van a hátsó, nem látható lapok eldobására, konkáv objektumoknál pedig a helyes megjelenítéshez a mélységi rendezésre is. A következőkben ismertetem ezt a két egyszerű algoritmust, majd a program által ismert 5 féle megjelenítési módot.

4.1. Hátsó lapok eldobása (Backface culling)

Ez az algoritmus arra szolgál, hogy még a kirajzolás előtt eltávolítsuk a testnek azokat a lapjait, amelyek biztosan nem láthatóak, ezzel jelentősen lecsökkentve a megjelenítendő lapok számát (átlagosan körülbelül a felére). Legyen adott egy F háromszöglap a három csúcspontjával (P_1, P_2, P_3), illetve a kamera pozíciója a térben (C). Az algoritmus menete a következő:

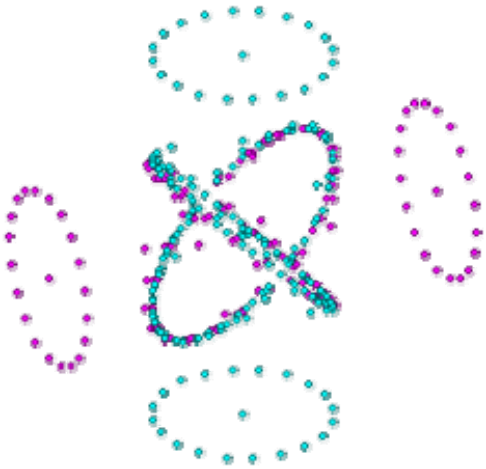
1. Számítsuk ki F normálvektorát. Ez alatt azt a vektort kell érteni, amely F síkjára merőleges, és a testből kifelé mutat, azaz ha a lap csúcspontjai olyan sorrendben vannak felvéve, hogy a körüljárási irány az óramutató járásával ellenkező irányú, akkor ezt a $v_1 = P_2 - P_1$ és $v_2 = P_3 - P_1$ vektorok vektoriális szorzata adja meg, vagyis $n = v_1 \times v_2$.
2. Számítsuk ki F valamely belső pontjából a kamerába mutató vektort (célszerű a lap középpontját venni). $CP = (P_1 + P_2 + P_3)/3;\ v = C - CP$
3. Ha a lap normálvektora és a kamera felé mutató vektor 90°-nál kisebb szöget zárnak be egymással, akkor az adott lap látható. Ez pontosan akkor teljesül, ha a közbezárt szög koszinusza pozitív előjelű.
4. Használjuk ki a belső szorzat egyik kiszámítási módját, miszerint $(\underline{a},\underline{b}) = \|\underline{a}\|\|\underline{b}\|\cos\alpha$, ahol α a két vektor közbezárt szöge. Ezt átrendezve kapjuk, hogy $\cos\alpha = \frac{(\underline{a},\underline{b})}{\|\underline{a}\|\|\underline{b}\|}$. A jobb oldalon mindent ismerünk, így ki tudjuk számítani a koszinusz értéket. Sőt, mivel a nevező értéke mindig pozitív (vektor hossza soha nem lehet negatív), így az előjel eldöntéséhez elegendő a számlálót kiszámítani.
5. Ha a kapott érték pozitív, a lap látható, különben nem.

4.2. Mélységi rendezés

Konvex testek esetén erre az algoritmusra nincs szükség, ha elvégezzük a hátsó lapok eldobását, ugyanis azután már nem fordulhatnak elő egymást takaró lapok. Konkáv modelleknél viszont igen, ezért előbb a lapokat rendezni kell, majd hátulról visszafelé megrajzolni őket. Mi szerint rendezzünk? Legcélszerűbb a lap középpontjának és a

kamerának a távolságát alapul venni, ugyanis ha valamelyik csúcspontot választjuk a középpont helyett, hosszú lapoknál helytelen eredményre juthatunk.

4.3. Megjelenítési módok



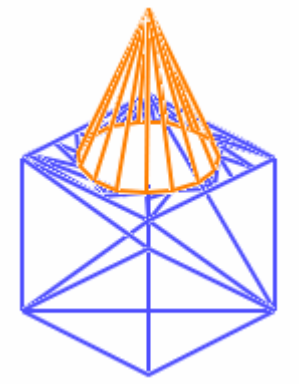
8. ábra Két henger uniójának pontjai

4.3.1. Csak pontok

Ebben a módban csak a test csúcspontjai kerülnek megrajzolásra. Minden csúcspont olyan színt kap, amilyen színű a lap, amely hozzá kapcsolódik. Mivel egy csúcs több laphoz is tartozik, a színe az összetett testeknél nem egyértelmű, viszont a primitíveknél igen. Azonban ha egy összetett testet átszínezünk, akkor minden lapja egyforma színű lesz, következésképpen a csúcspontjai is.

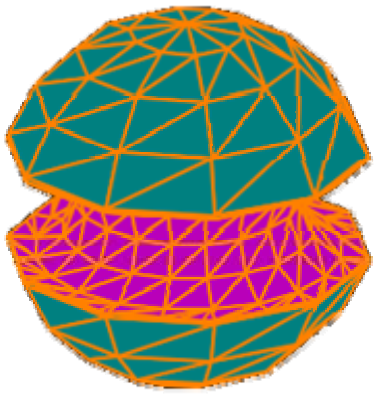
4.3.2. Drótváz

Ebben a módban a test drótváz modelljét láthatjuk, ilyenkor csak az élek kerülnek kirajzolásra, mégpedig az OpenGL motor által simítva (antialiasing). Az élek színét külön meg lehet adni a primitíveknél, illetve az összetett testeknél is, de alapértelmezésként az összetett test élei az alkotó testek éleinek színeit kapják.



9. ábra Kúp és kocka uniójának drótváz modellje

4.3.3. Kitöltés színnel



10. ábra Gömb és tórusz különbsége kitöltve

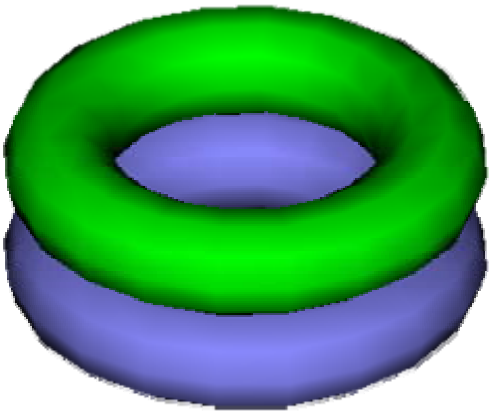
Ha ezt a módot választjuk, megjelennek a lapok, saját színükkel kitöltve, illetve az élek, szintén saját színükkel megrajzolva és simítva. Itt már van hátsó lap eldobás és mélységi rendezés is.

4.3.4. Konstans árnyalás (Flat shading)

11. ábra Gömb és ikozaéder uniója konstans árnyalással

Ez a legegyszerűbb árnyalási mód. Minden laphoz hozzárendelünk egy intenzitásértéket, és ezzel az értékkel megszorozzuk a lap színének R, G, B színösszetevőit. Ezáltal annak a színnek egy világosabb, vagy sötétebb változatához jutunk. Algoritmusa hasonlít a hátsó lap eldobás algoritmusához, mindössze a végén van eltérés. Van egy fényforrásunk, amellyel egy adott irányból megvilágítjuk a testet, ezért a kamera helyett itt a fényforrás pozícióját vesszük figyelembe a számításoknál. Lényege, hogy kiszámítsuk az adott lap normálvektora és a fényforrásba mutató vektor által bezárt szög koszinuszát (a pontos koszinuszát, itt nem elég csak az előjelet), majd mivel ez egy -1 és 1 közötti érték, a negatív részt nullának vesszük. Ezzel a 0 és 1 közötti értékkel súlyozzuk a színkomponenseket. Látható, hogy minél nagyobb szöget zár be a két vektor, a koszinusz annál kisebb lesz, azaz a lap annál sötétebb, hiszen ez azt jelenti, hogy egyre jobban elfordul a fényforrástól. Nyílván annak a lapnak lesz a legnagyobb a világosságértéke, amelynek a normálvektora pontosan a fényforrás felé mutat.

4.3.5. Gouraud-árnyalás



12. ábra Két tórusz uniója Gouraud-árnyalással

Ez az árnyalási módszer sokkal szebb képet ad, viszont nagyobb a számításigénye is, ugyanis egy háromszöglapot nem egy színnel töltünk ki. Ehelyett kiszámítjuk a lap három csúcspontjához tartozó színértékeket, majd ezeket interpoláljuk a háromszög belsejében. Implementációs szempontból ez azt jelenti, hogy végig kell mennünk a test összes csúcspontján (mivel nincs olyan csúcspont, amelyhez ne tartozna lap, ezért egyet sem hagyhatunk ki), és megnézni, hogy az adott csúcshoz mely lapok kapcsolódnak. Az adott pontban az összes kapcsolódó lapnak ki kell számítani a normálvektorát, majd e vektorokat átlagolni. Ez egy egyszerű számtani átlag kiszámítását jelenti minden koordinátára. Az így kapott vektor lesz az ún. csúcspontbeli normális. Ezután a már ismert módon kiszámítjuk a csúcspontbeli normális és a csúcspontból a nézőpontba mutató vektor által bezárt szög koszinuszát, és ebből a csúcshoz tartozó színintenzitást. Amikor rajzoljuk a lapokat, mindhárom csúcsának intenzitásával megszorozzuk az adott lap színét, és így megkapjuk a színeket a csúcspontokban. A színek interpolációját a háromszögön belül már az OpenGL motor elvégzi.

4.4. Néhány szó a sebességről

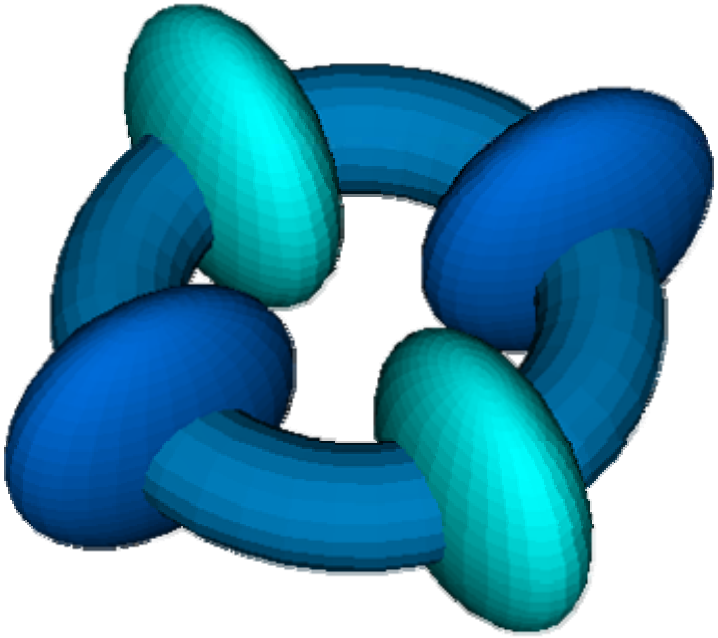
Nyilvánvaló, hogy az első három megjelenítési mód nem igényel extra számításokat, így ezek még nagy (néhány ezer) csúcs-, él-, és poligonszám esetén is folyamatos animációt adnak a kamera forgatásakor. Persze a kitöltés színnel kissé lassabb művelet, mivel a kiszínezett lapokat is ki kell rakni, plusz még a körvonalukat is.

A konstans árnyalás már igényel előzetes számításokat, ráadásul sok szorzás van benne a vektorműveleteknél (skaláris- és vektoriális szorzat), ami a négy alapművelet közül a legidőigényesebb a processzor számára. Viszont ha eltekintünk ezektől a számításoktól, a grafikus megjelenítés gyorsabb, mint a „kitöltés színnel” mód esetén, ugyanis itt tényleg csak kitöltött poligonokat kell rajzolni, éleket nem. A számításokat fel lehet gyorsítani, például azzal, ha előre letároljuk az egyes lapokhoz tartozó normálvektorokat. Mivel ez a művelet

magában 6 darab lebegőpontos szorzást igényel, ami a ciklusmag szorzásainak fele, jelentősen felgyorsíthatjuk ezzel az algoritmust.

A leginkább számításigényes megjelenítési mód természetesen a Gouraud-árnyalás. Eleve azért lassú, mert minden csúcspontnál végig kell mennünk az összes lapon, és ellenőrizni, hogy az adott pont nem az ő csúcspontja-e. Továbbá minden találatnál meg kell határozni a lap normálvektorát (kivéve, ha előre el van tárolva), majd ezeket a vektorokat átlagolni, végül a koszinusz értéket kiszámítani. A legtöbb időt azonban a teljes keresés végrehajtása jelenti a lapok tömbjében, ezt viszont csak úgy lehetne megkerülni, ha minden pontnál megjegyeznénk azoknak a lapoknak az indexeit, amelyekhez tartozik. Ez a megoldás viszont több memóriát követelne. Érdemes megfontolni, hogy a kisebb memóriaigényre vagy a nagyobb sebességre törekszünk, az ugyanis általános szabály, hogy a kettő együtt nem megy.

A tesztek azt mutatják, hogy egy mai középkategóriás számítógépen a Gouraud-árnyalási mód kb. 2500 lap esetén még elfogadható sebességet produkál, 3000 lap fölött viszont már eléggé akadozik. A konstans árnyalásnál ez a határ körülbelül 7000 poligonnál van, akárcsak a színezett lapoknál, a drótvázas megjelenítés illetve a csúcspontok megrajzolása pedig még 8500-9000 lapnál is elég jó sebességet ad. Ez utóbbiakat természetesen leginkább a használt OpenGL motor implementációja határozza meg, de a két árnyalt modellnél inkább a megvilágítási számítások gyorsasága a döntő.



13. ábra Egy tórusz és 4 ellipszoid uniója (10384 lap)

5. A halmazműveletek algoritmusa

5.1. Térbeli vizsgálatok, algoritmusok

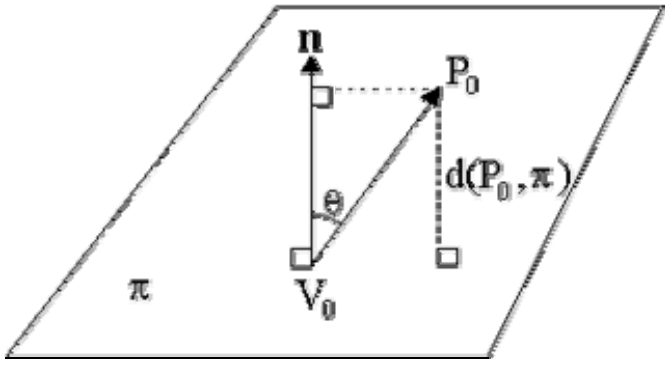
Itt következzenek azok a fontos geometriai algoritmusok, amelyek megoldják a 3D halmazműveletekkel kapcsolatban felmerülő problémákat. Van köztük, amelyik nagyon egyszerű, de az egyértelműség és átláthatóság kedvéért ezeket is ismertetem.

5.1.1. Pont a téglatestben

Hogyan döntsük el, hogy egy pont benne van-e egy téglatestben? Legyen adott a $P(x,y,z)$ pont és egy téglatest, amelyet két átellenes csúcsa reprezentál, amelyeket $Q_1(x_1,y_1,z_1)$ és $Q_2(x_2,y_2,z_2)$ jelöl. Tegyük fel, hogy a megfelelő koordinátapárok rendezve vannak, azaz $x_1 \le x_2, y_1 \le y_2, z_1 \le z_2$. A P pont akkor és csakis akkor van benne ebben a téglatestben, ha $x_1 \le x \le x_2, y_1 \le y \le y_2, z_1 \le z \le z_2$ teljesül.

5.1.2. Pont és sík távolsága

Most bevezetjük az ún. előjeles távolság fogalmát. Ez nagyon hasznos, mert azt is megmondja, hogy az adott pont a sík melyik oldalán van. Legyen adott egy $P(x,y,z)$ pont, és egy π sík egy tetszőleges pontjával (V_0) és a normálvektorával ($\underline{n}$). A pont és a sík távolsága nem lesz más, mint a P_0-V_0 vektor $\underline{n}$ vektorra eső merőleges vetületének a hossza.



14. ábra Pont és sík távolsága

$d(P_0,\pi) = \|P_0 - V_0\| \cos\theta = \dfrac{(\underline{n},(P_0 - V_0))}{\|\underline{n}\|}$, ahol θ a két vektor közbezárt szöge.

Ha a normálvektor egységnyi hosszú, akkor a fenti képlet egyszerűsödik az alábbira:

$d(P_0,\pi) = (\underline{n},(P_0 - V_0))$

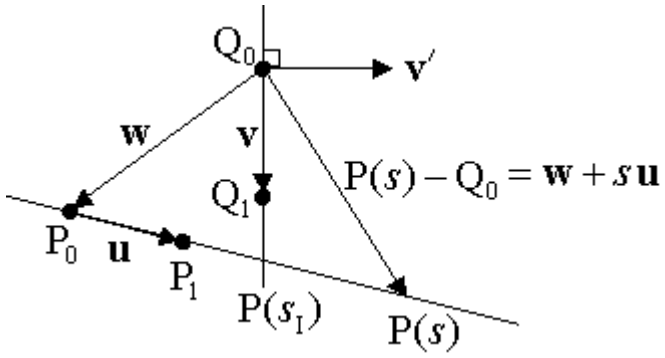
5.1.3. Két szakasz metszéspontja

Két egyenes helyzete a térben sokféle lehet. Lehetnek párhuzamosak, metszők, vagy kitérők. Két szakasz esetén pedig még több lehetőség van, például ha egy egyenesre esnek, akkor sem biztos, hogy van közös részük.

Legyen adott a szakaszok két végpontja, P_0, P_1 és Q_0, Q_1. Számítsuk ki a szakaszok irányvektorát: $\underline{u} = P_1 - P_0$ és $\underline{v} = Q_1 - Q_0$, majd írjuk fel a megfelelő egyenesek paraméteres egyenletét:

$P(s) = P_0 + s\underline{u}$, illetve $Q(t) = Q_0 + t\underline{v}$

A párhuzamosságot egyszerűen tesztelhetjük, ugyanis ekkor létezik olyan C valós szám, hogy $\underline{u} = C\underline{v}$. Ha $\underline{u} = (u_1,u_2,u_3)$ és $\underline{v} = (v_1,v_2,v_3)$ akkor ez pontosan azt jelenti, hogy $\dfrac{u_1}{v_1} = \dfrac{u_2}{v_2} = \dfrac{u_3}{v_3} = C$.



15. ábra Két szakasz metszéspontja a síkban

Ha a két egyenes egybeesik, akkor P_0 rajta van $Q(t)$-n, azaz $P_0 = Q_0 + t_0\underline{v}$ valamely t_0 valós számra. Ha $\underline{w} = P_0 - Q_0$, akkor a $\underline{w} = t_0\underline{v}$-ből megkaphatjuk t_0-t, mivel az előző esethez hasonlóan $\dfrac{w_1}{v_1} = \dfrac{w_2}{v_2} = \dfrac{w_3}{v_3} = t_0$. Két szakasz esetén azt kell eldöntenünk, hogy van-e egybeeső

részük, átfedésük. Ehhez ki kell számítani t_1-et is a $P_1 = Q_0 + t_1 \underline{v}$ egyenletből, majd meg kell vizsgálni, hogy a $[t_0, t_1]$ és $[0,1]$ intervallumok metszete üres-e. Ha igen, akkor a két szakasznak nincs átfedése, különben a közös részt ez a metszet adja: $[r_0, r_1] = [t_0, t_1] \cap [0,1]$. Ezután r_0-t és r_1-et visszahelyettesítve megkapjuk a közös szakaszdarab $Q(r_0)$ és $Q(r_1)$ végpontjait.

A szakaszokra illeszkedő egyenesek a térben vagy metszik egymást, vagy kitérők. Azonban, ha metszik egymást, akkor biztos, hogy a síkon vett vetületüknek is van metszéspontja, ezért a számítást elvégezhetjük két dimenzióban, majd ha megkaptuk a metszésponthoz tartozó s_I és t_I paramétereket, visszahelyettesítjük őket a paraméteres egyenletekbe, és ellenőrizzük, hogy $P(s_I) = Q(t_I)$ igaz-e minden koordinátára.

Ahhoz, hogy megkapjuk s_I-t, induljunk ki a $P(s) - Q_0 = \underline{w} + s\underline{u}$ egyenletből. Ahogy az ábrán is látszik, a metszéspontban a $P(s) - Q_0$ vektor merőleges a $\underline{v'}$ vektorra, azaz a belső szorzatuk nulla lesz: $(\underline{v'}, (\underline{w} + s\underline{u})) = 0$. Ebből az összefüggésből kiszámolhatjuk s_I-t:

$$s_I = \frac{(\underline{w}, -\underline{v'})}{(\underline{u}, \underline{v'})}, \text{ ahol } \underline{v'} = (-v_2, v_1)$$

A t_I paraméter esetében hasonlóan járunk el, és megkapjuk, hogy

$$t_I = \frac{(\underline{w}, \underline{u'})}{(\underline{v}, \underline{u'})}, \text{ ahol } \underline{u'} = (-u_2, u_1)$$

Már csak vissza kell helyettesíteni a paramétereket a megfelelő egyenletekbe, és ellenőrizni a $P(s_I) = Q(t_I)$ egyenlőséget a koordinátákra, hogy megkapjuk a térbeli metszéspontot. Mivel szakaszokról van szó, itt is ellenőrizni kell a paraméterek értékét, ugyanis az eddigiekkel csak a két egyenes metszéspontját kaptuk meg. A két szakasz ténylegesen akkor metszi egymást, ha s_I és t_I is a [0, 1] intervallumban van.

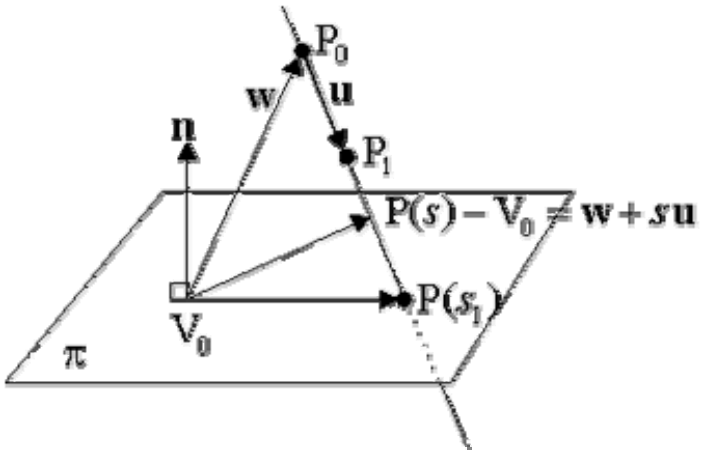
5.1.4. Szakasz és sík metszéspontja

A probléma a következő: adott egy szakasz a két végpontjával (P_0 és P_1), illetve egy sík egy tetszőleges pontjával (V_0) és a normálvektorával ($\underline{n}$). Döntsük el, hogy metszik-e egymást, és ha igen, mely pontban?

Számítsuk ki a P_0 és P_1 pontokon átmenő egyenes irányvektorát: $\underline{u} = P_1 - P_0$. Az egyenes paraméteres egyenletét fogjuk használni, melynek alakja: $P(s) = P_0 + s\underline{u}$. Ha

$(\underline{n},\underline{u}) = 0$, az azt jelenti, hogy az egyenes és a sík egymással párhuzamos, és vagy nincs metszéspontjuk, vagy az egyenes teljes egészében a síkon fekszik. Ezt úgy lehet ellenőrizni, hogy kiszámítjuk az egyenes bármely pontjának és a síknak a távolságát. Ha ez 0, akkor a második eset teljesül, ezt valamiképpen külön jelezni kell.

Ha nem párhuzamosak, akkor pontosan egy közös metszéspontjuk van, amelyet $P(s_I)$ -vel jelölünk.



16. ábra Szakasz és sík metszéspontja

Legyen $\underline{w} = P_0 - V_0$. A metszéspontra teljesül a következő egyenlőség:

$$P(s_I) - V_0 = P_0 - V_0 + s_I \underline{u} = \underline{w} + s_I \underline{u},$$

amely az egyenes egyenletéből adódik. Továbbá a baloldalon álló $P(s_I) - V_0$ vektor merőleges $\underline{n}$-re, vagyis a belső szorzatuk nulla: $(\underline{n},(\underline{w} + s_I \underline{u})) = 0$. Ebből kifejezhetjük s_I -t, majd ezt visszahelyettesítve az egyenes paraméteres egyenletébe, megkapjuk a metszéspont koordinátáit:

$$s_I = \frac{(-\underline{n},\underline{w})}{(\underline{n},\underline{u})} = \frac{(\underline{n},(V_0 - P_0))}{(\underline{n},(P_1 - P_0))}$$

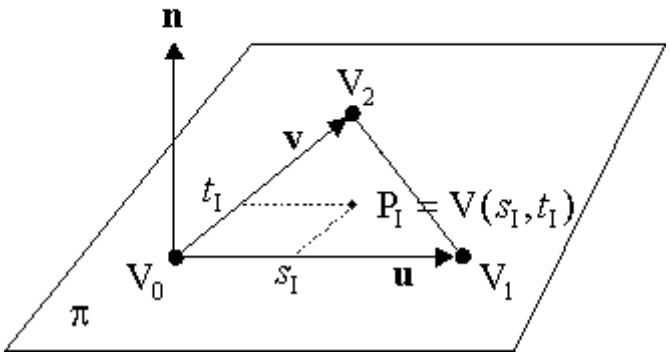
Mivel itt egy szakaszról volt szó, még egy ellenőrzés szükséges. A metszéspont akkor és csakis akkor lesz rajta a szakaszon, ha $0 \le s_I \le 1$. Amennyiben egy kezdőpontjával és irányvektorával adott sugárról van szó, ugyanígy kell eljárni, de az utolsó feltétel $0 \le s_I$ -re módosul.

5.1.5. Pont a háromszögben

Többször is felmerül az algoritmus során az a probléma, hogy el kell dönteni, hogy egy pont rajta van-e egy térbeli háromszöglapon. Legyen V_0, V_1, V_2 a háromszög három csúcspontja, P pedig a vizsgált pont, és tegyük fel, hogy a pont a háromszög síkjában van. Induljunk ki a sík paraméteres egyenletéből, azaz

$$V(s,t) = V_0 + s(V_1 - V_0) + t(V_2 - V_0) = V_0 + s\underline{u} + t\underline{v}$$

Egy $P = V(s,t)$ pont benne van a háromszöglapban, ha $s \geq 0, t \geq 0, s + t \leq 1$ teljesül. Így csak meg kell keresnünk az adott P_I ponthoz tartozó s_I és t_I paramétereket, és ellenőrizni az egyenlőtlenségeket.



17. ábra Pont a háromszögben

Legyen $\underline{w} = P_I - V_0$. A fenti síkegyenletet átírhatjuk a következő formába:

$\underline{w} = s_I \underline{u} + t_I \underline{v}$, ahol $\underline{u} = V_1 - V_0$ és $\underline{v} = V_2 - V_0$

Továbbá legyen $\underline{v'} = \underline{n} \times \underline{v}$ és $\underline{u'} = \underline{n} \times \underline{u}$. Látható, hogy ez a két új vektor merőleges $\underline{n}$-re, és a háromszög síkjában van, valamint $(v, \underline{v'}) = 0$ és $(u, \underline{u'}) = 0$. Először vegyük az egyenlet mindkét oldalának belső szorzatát $\underline{v'}$-vel: $(\underline{w}, \underline{v'}) = s_I(\underline{u}, \underline{v'}) + t_I(\underline{v}, \underline{v'}) = s_I(\underline{u}, \underline{v'})$. Ebből megkaphatjuk s_I-t, majd hasonlóképpen vesszük mindkét oldal belső szorzatát $\underline{u'}$-vel, és megkapjuk t_I-t.

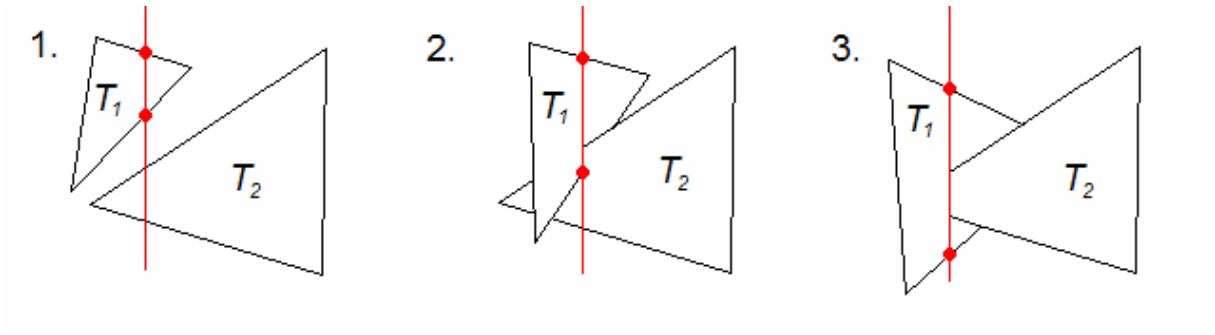
$$s_I = \frac{(\underline{w}, \underline{v'})}{(\underline{u}, \underline{v'})} \qquad t_I = \frac{(\underline{w}, \underline{u'})}{(\underline{v}, \underline{u'})}$$

Ezután már csak ellenőrizni kell a fent megadott feltételeket a kiszámított s_I-re és t_I-re. Ha valamelyik nevező nulla, az azt jelenti, hogy a háromszög degenerált, ezt külön esetként kell kezelnünk.

5.1.6. Két háromszög metszése

Van két háromszöglapunk, T_1 és T_2. Ha metszik egymást, akkor a metszetük egy térbeli szakasz lesz. A mi szempontunkból azonban most nem ez a fontos, hanem az, hogy ha tényleg metszik egymást, akkor adjuk meg T_1 megfelelő éleinek metszéspontját T_2 síkjával. Ez azért kell, mert ennek alapján tudjuk majd T_1-et felosztani újabb háromszögekre, és ezen alapul majd a halmazműveletek algoritmusa.

Első lépésként megvizsgáljuk, hogy a két háromszög hogyan helyezkedik el egymáshoz képest. Előfordulhat, hogy a két lap egy síkban van, ekkor csak az éleket kell elmetszeni egymással az adott síkon (lásd. 5.1.3. Két szakasz metszéspontja), és az algoritmus véget ér. Ha T_2 teljes egészében T_1 síkjának valamelyik oldalán helyezkedik el, vagy fordítva, azaz T_1 csúcsai vannak T_2 valamelyik oldalán, akkor a két háromszög biztosan nem metszi egymást. Ellenkező esetben T_1 egyik csúcsa T_2 síkjának egyik oldalán van, a másik kettő pedig a másik oldalán. A csúcsok elhelyezkedését úgy tudjuk meghatározni, hogy vesszük a csúcs és a sík előjeles távolságának előjelét (lásd. 5.1.2. Pont és sík távolsága). Ha megtaláltuk ezt a különálló csúcsot, akkor a belőle induló éleket el kell metszeni T_2 síkjával (lásd. 5.1.4. Szakasz és sík metszéspontja). Miután megvannak a metszéspontok, még egy ellenőrzést kell végezni, mert itt még mindig nem biztos, hogy a két lap metszi egymást. Gondoljunk például arra az esetre, ha az egyik háromszög az XZ síkon van, a másik pedig az YZ síkon, úgymond „egymás fölött” ha a Z tengelyt tekintjük „felfelé” mutatónak. Más szavakkal az egyik háromszög csúcsainak legkisebb Z koordinátája nagyobb, mint a másik háromszög csúcsainak legnagyobb Z koordinátája. Ennél az esetnél az első két teszten gond nélkül átmehet a két lap, de mégsem metszik egymást. Ezért kell a végén még megnézni, hogy a kapott metszéspontok hol helyezkednek el T_2-höz képest. A következő ábra mutatja a lehetséges eseteket (a többit az előző tesztek már kizárták):



18. ábra Két térbeli háromszöglap lehetséges helyzete

A piros vonal mutatja a két sík metszésvonalát, a két piros pont pedig T_1 metszéspontjait T_2 síkjával.

1. A háromszögek nem metszik egymást.
2. Az egyik metszéspont beleesik T_2-be. Ezt könnyen tudjuk tesztelni a „Pont a háromszögben” algoritmussal.
3. T_2 a két metszéspont között helyezkedik el. Ebben az esetben a metszéspontok közötti szakasz és a háromszög valamelyik oldalszakasza metszi egymást (lásd. 5.1.3. Két szakasz metszéspontja).

Mivel a célunk az volt, hogy a metszést T_1 szempontjából figyeljük, a 2. és 3. esetben az algoritmus sikeresen visszaadja a két metszéspontot.

5.1.7. Pont a poliéderben

Ez az algoritmus eldönti egy *P* pontról, hogy benne van-e egy *S* poliéderben, amely a B-Rep modelljével adott. A visszatérési érték 1, ha a pont benne van a testben, -1, ha nincs benne, és 0, ha a pont a test valamelyik lapjára vagy élére, azaz határfelületére esik.

Egy egyszerű, de hatékony teszttel kezdjük: ellenőrizzük, hogy a pont benne van-e a test ún. befoglaló dobozában (bounding box). Ez egy téglatest, amelynek két átellenes csúcsát úgy kapjuk, hogy vesszük a test pontjainak legkisebb és legnagyobb koordinátáit. Elvégezzük az **„5.1.1. Pont a téglatestben”** tesztet, és ha ez hamis eredményt ad, akkor azonnal elmondhatjuk, hogy *P* nincs benne a testben, és az algoritmus itt véget ér.

Különben *P*-ből indítunk véletlenszerű irányba egy *R* sugarat, és megszámoljuk, hogy ez a sugár *S*-nek hány lapját metszi. Ha ez az érték páratlan, akkor a pont benne van a testben, különben nincs benne. Közben először mindig ellenőrizzük, hogy a pont nincs-e rajta az adott

lapon (**„5.1.5. Pont a háromszögben”** teszt), mert ekkor visszaadhatjuk a 0-t eredményül, és vége. A sugár és a háromszöglap metszéspontját úgy határozhatjuk meg, hogy először vesszük a *R*-nek és a lap síkjának a metszéspontját (**„5.1.4. Szakasz és sík metszéspontja”**, az utolsó feltételnél $0 \le s_1$-re tesztelünk), majd megnézzük, hogy ez benne van-e a háromszögben (**„5.1.5. Pont a háromszögben”** teszt).

5.2. Az algoritmus főszála

Tulajdonképpen a három halmazművelet (unió, metszet, különbség) jelentős része ugyanaz, csak a végén van eltérés, ezért az implementációban is egyetlen függvény végzi el mindhármat. Ez alatt azt értem, hogy mindegyik azon alapul, hogy háromszöglapokat kell elvágni síkokkal a térben, és a halmazművelet típusa csak akkor játszik szerepet, amikor el kell dönteni, hogy az eredményként létrehozott testben mely lapok szerepeljenek, és melyek ne. A különbség műveletnél meg kell határoznunk egy sorrendet a testek között, hiszen ez nem kommutatív művelet, nem mindegy, hogy melyik testből vonjuk ki a másikat. Az egyik testet jelöljük S_1-gyel, a másikat pedig S_2-vel.

Van tehát két testmodellünk, mindkettő B-Rep adatszerkezettel reprezentálva. Az eljárás azzal kezdődik, hogy megvizsgáljuk az egymáshoz viszonyított elhelyezkedésüket. Ehhez rendelkezésre áll mindkét testnek a befoglaló doboza. Első lépésben megvizsgáljuk, hogy ez a két doboz metszi-e egymást. Ez azt jelenti, hogy tekintjük a dobozok vetületét az egyes koordinátatengelyeken, ezek meghatároznak egy-egy intervallumot, és ezen intervallumoknak keressük páronként a metszetét. Ha bármely két ilyen pár diszjunkt, azaz nincs közös részük, a két doboz nem metszi egymást, vagyis a két test sem metszheti egymást. Ebben az esetben a művelet típusától függ a továbblépés:

- **unió**: A két test B-Rep-jét összefűzzük, és ez lesz az eredmény.
- **metszet**: Az eredmény üres halmaz.
- **különbség**: Az eredmény az S_1 test lesz.

Ha a befoglaló dobozok metszik egymást, akkor lehet, hogy a testek is metszeni fogják, ekkor következnek a térbeli vágások.

Az ötlet, hogy először tekintsük S_1 lapjait, és messük el őket S_2 lapjaival, ahol szükséges. Ezután a halmazművelet típusától függően válogassuk ki S_1-nek azon lapjait, amelyek bekerülnek az eredmény testbe. A második menetben ugyanezt csináljuk, csak a másik test szemszögéből, azaz S_2 lapjait kell metszeni S_1 lapjaival, és elvégezni a válogatást

S_2 lapjaira is. A kiválogatás úgy történik, hogy megnézzük, az adott lap három csúcsa a másik testen kívül vagy belül helyezkedik el.

Következzen az algoritmus pszeudo-kódja:

Első menet:

C1. Ciklus S_1 összes lapjára {

$F_1 = S_1$ i-edik lapja (Csúcsai: V_0, V_1, V_2)

C2. Ciklus S_2 összes lapjára {

$F_2 = S_2$ j-edik lapja

Ha F_2 metszi F_1-et {

Kiszámítjuk F_1 megfelelő éleinek metszéspontját F_2 síkjával az **„5.1.6. Két háromszög metszése”** algoritmus szerint.

F_1-et felosztjuk három új háromszögre: G_1, G_2, G_3

A lapok tömbjének végére felvesszük G_1, G_2, G_3-at.

Egy logikai változóval jelöljük, hogy volt metszés, és C2 ciklusból kilépünk.

}

}

Ha volt metszés, folytatjuk S_1 következő lapjával.

Ha nem volt metszés, az azt jelenti, hogy F_1 teljes egészében S_2-n kívül vagy belül van. Ezért meghívjuk az **„5.1.7. Pont a poliéderben”** algoritmust V_0, V_1, V_2-re, illetve a lap középpontjára, ami legyen CP.

B_0 = A teszt eredménye V_0-ra.

B_1 = A teszt eredménye V_1-re.

B_2 = A teszt eredménye V_2-re.

B_3 = A teszt eredménye CP-re.

Ezután a művelet típusának megfelelően meghatározzuk azt a feltételt, ami ahhoz kell, hogy a vizsgált lap bekerülhessen az eredményhalmazba. A feltételt C-vel jelöljük. (Az „és” illetve „vagy” műveletet a logikában alkalmazott $\wedge$ valamint $\vee$ jelek jelentik.)

- unió: $C = B_0 \le 0 \wedge B_1 \le 0 \wedge B_2 \le 0 \wedge B_3 \le 0$
- metszet: $C = B_0 \ge 0 \wedge B_1 \ge 0 \wedge B_2 \ge 0 \wedge B_3 \ge 0$

- különbség: $C = B_0 \leq 0 \wedge B_1 \leq 0 \wedge B_2 \leq 0 \wedge B_3 \leq 0$

Ha C igaz, akkor F_1-et felvesszük az eredmény lapjai közé.

}

A feltételek magyarázata:

- **unió:** Ennél a műveletnél az S_1 test azon lapjai kerülnek bele az eredményhalmazba, amelyeknek a csúcsai a másik testen kívül, vagy annak határfelületén vannak. Ez azonban nem elegendő feltétel, hiszen előfordulhat olyan eset, amikor mindhárom csúcs a test felületén van, de a lap többi pontja a testen belül helyezkedik el. Ezért kell tesztelni a lap középpontját is. Ha ez a pont is S_2 felszínén van, itt akkor is felvesszük az eredménybe. (A második menetben nem fogjuk, lásd ott.)
- **metszet:** Itt a feltétel majdnem az unió ellentéte, de az egyenlőséget itt is meg kell engedni, valamint a lapközéppont vizsgálata is fontos a fent említett okokból.
- **különbség:** A feltétel megegyezik az unió művelet feltételével, hiszen S_1-ből vonjuk ki S_2-t, ezért nyilván azokat a lapokat kell meghagyni, amelyek nincsenek benne S_2-ben.

Látszik, hogy az első menetben módosítottuk S_1 lapjainak a tömbjét, mivel felvettük a végére a szétvágott lapok részeit. Hogy egy kicsit felgyorsítsuk a futást, a második menet előtt a tömböt visszaállítjuk eredeti állapotába, azaz töröljük a végéről az új lapokat. Az eredményt ez nem befolyásolja, hiszen a most törölt lapok már amúgy is léteztek a tömbben, csak egy darabban, viszont jóval kevesebb metszési tesztet kell elvégezni, és ez szignifikáns különbséget okoz a futási sebesség terén.

Második menet:

C1. Ciklus S_2 összes lapjára {

$F_1 = S_2$ i-edik lapja (Csúcsai: V_0, V_1, V_2)

C2. Ciklus S_1 összes lapjára {

$F_2 = S_1$ j-edik lapja

Ha F_2 metszi F_1-et {

Kiszámítjuk F_1 megfelelő éleinek metszéspontját F_2 síkjával az **„5.1.6. Két háromszög metszése”** algoritmus szerint.

F_1-et felosztjuk három új háromszögre: G_1, G_2, G_3

A lapok tömbjének végére felvesszük G_1, G_2, G_3-at.

Egy logikai változóval jelöljük, hogy volt metszés, és C2 ciklusból kilépünk.

}

}

Ha volt metszés, folytatjuk S_2 következő lapjával.

Ha nem volt metszés, az azt jelenti, hogy F_1 teljes egészében S_1-en kívül vagy belül van. Ezért meghívjuk az **„5.1.7. Pont a poliéderben”** algoritmust V_0, V_1, V_2-re, illetve a lap középpontjára, ami legyen CP.

B_0 = A teszt eredménye V_0-ra.

B_1 = A teszt eredménye V_1-re.

B_2 = A teszt eredménye V_2-re.

B_3 = A teszt eredménye CP-re.

Ezután a művelet típusának megfelelően meghatározzuk azt a feltételt, ami ahhoz kell, hogy a vizsgált lap bekerülhessen az eredményhalmazba. A feltételt C-vel jelöljük. (Az „és” illetve „vagy” műveletet a logikában alkalmazott $\wedge$ valamint $\vee$ jelek jelentik.)

- unió: $C = B_0 \leq 0 \wedge B_1 \leq 0 \wedge B_2 \leq 0 \wedge B_3 < 0$
- metszet: $C = B_0 \geq 0 \wedge B_1 \geq 0 \wedge B_2 \geq 0 \wedge B_3 > 0$
- különbség: $C = B_0 \geq 0 \wedge B_1 \geq 0 \wedge B_2 \geq 0 \wedge B_3 > 0$

Ha C igaz, akkor F_1-et felvesszük az eredmény lapjai közé.

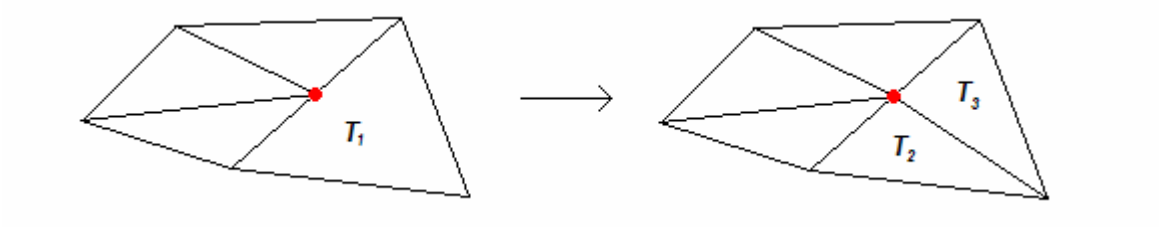
}

A feltételek magyarázata:

- **unió:** A művelet lényege ugyanaz, mint az első menetben, mindössze a feltétel végén van eltérés, itt ugyanis nem engedjük meg az egyenlőséget. Tegyük fel, hogy az első menetben volt olyan lap, amelyre $B_i = 0, \quad i = 0,1,2,3$, vagyis olyan lapot teszteltünk, amely teljesen a másik test felületén volt. Ez esetben a két test felülete egy bizonyos részen egybeesik, vagyis S_2-ben is lesz ilyen lap. Ha itt is megengednénk az egyenlőséget, akkor egymást fedő lapok kerülhetnének bele az eredményhalmazba, ami rossz megoldáshoz vezet.
- **metszet:** Az első menethez képest itt is csak a feltétel vége alakul át szigorú egyenlőtlenséggé, a már ismert okból.
- **különbség:** S_2 részéről azokat a lapokat kell meghagyni, amelyek S_1-en belül vannak, ez pedig pontosan a metszet művelet feltételét jelenti. Fontos, hogy ennél a műveletnél a lap körüljárási irányát meg kell fordítani, mielőtt belerakjuk az eredményhalmazba.

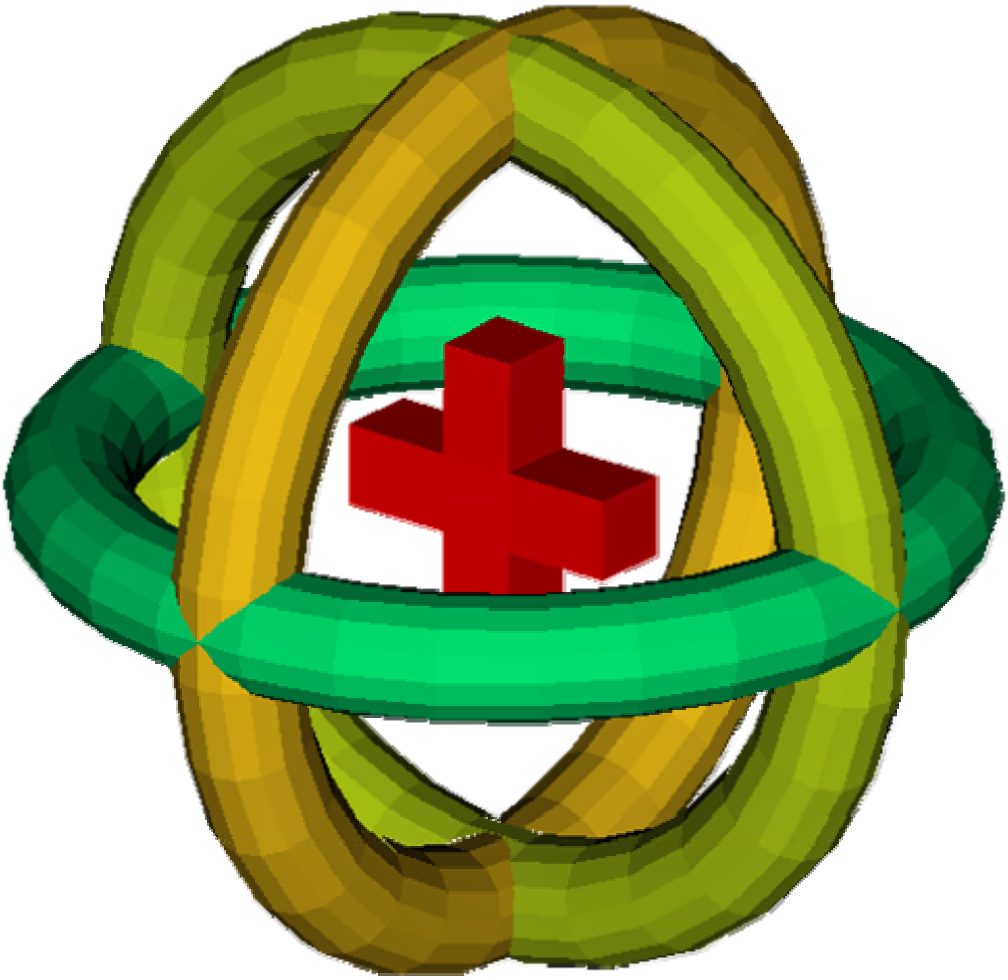
Egy lap felvétele az eredményhalmazba úgy történik, hogy először minden csúcsát ellenőrizzük, hogy nem szerepel-e már az eredményben. Amelyik már szerepel, annak egyszerűen tároljuk az indexét, amelyik nem, azt felvesszük, és az indexe a tömb aktuális elemszáma – 1 lesz (mivel 0-ról indul a számozás). Ezután felvesszük a lapot az ismert indexekkel, illetve a három éle közül azokat, amelyek még nem szerepeltek. Így nem lesz egyik tömbben sem redundancia, amelyet a képernyőn ugyan nem látnánk, viszont lelassítaná a megjelenítést.

Ezzel gyakorlatilag megkonstruáltuk a halmazművelet eredményét a két test között, azonban van még egy dolog, amire oda kell figyelnünk. Az első menetben, amikor egy lapnak mind a négy tesztelt pontja ráesik a test felszínére, belevesszük az eredménybe. Ez okozhat olyan hibát, hogy ezen lap éleire ráeshet a test valamelyik csúcspontja. Ilyet a B-Rep adatszerkezet nem enged meg, ezért a végén ki kell küszöbölni. Minden pontot le kell tesztelni, hogy nem esik-e rá valamelyik lap élére. Ha igen, azt a lapot két részre bontjuk az alábbi ábra szerint:



19. ábra A háromszöglap darabolása

T_1-et töröljük a lapok közül, és felvesszük helyette T_2-t és T_3-at. Ezt a műveletet hívom konzisztencia ellenőrzésnek, ez az algoritmus utolsó lépése.



20. ábra Három tórusz és két téglatest uniója

6. Az implementációról

Amint azt a bevezetésben említettem, a szoftver C# programnyelven íródott, a fejlesztéshez pedig a Microsoft Visual Studio 2005 fejlesztői keretrendszert használtam. A C# egy objektumorientált nyelv, és egyszerű használhatósága miatt kiválóan alkalmas e diplomamunka témájához tartozó szoftver elkészítésére, a keretrendszerrel pedig könnyen és gyorsan létre lehet hozni grafikusan a szükséges ablakokat, vezérlő elemeket. Ebben a fejezetben néhány implementációs problémának a megoldását ismertetem, amely hasznos lehet mindazoknak, akik hasonló témájú programon dolgoznak. Az osztályneveket ***félkövér, dőlt*** betűtípussal írom, az adattagokat és a metódusokat pedig sima *dőlt* betűkkel (a metódusnevek után ()-lel), hogy ezzel is kiemeljem őket.

6.1. Az osztályhierarchia

A osztályokat alapvetően négy csoportba sorolhatjuk:

1. A testeket reprezentáló osztályhierarchia
2. A transzformációkat leíró osztályhierarchia
3. Különféle alapvető adatszerkezeteket és algoritmusokat leíró segédosztályok
4. A grafikus ablakok osztályai, ún. „Form”-ok

6.1.1. A testeket reprezentáló osztályhierarchia

A hierarchia gyökere a ***Solid*** osztály, amely bármilyen testet jelenthet, és ennek megfelelően olyan adattagjai vannak, amelyek minden testre jellemzőek, vagyis a testet leíró B-Rep adatstruktúra (***BRep***), a test középpontja (***Point3D***), és a befoglaló dobozának két szemben lévő csúcspontja. Ebből származnak a primitívek osztályai, amelyek már speciális testeket reprezentálnak, ennélfogva olyan adattagokat tartalmaznak, amelyek csak az adott testre jellemzőek. Például a paraméteres egyenletekkel leírt primitíveknél van egy vagy két paraméter, amely a paramétertartomány(ok) felosztásának finomságát adja meg, ezáltal meghatározva a test pontjainak, éleinek és lapjainak számát, és persze a megjelenítés minőségét is. Ilyen osztályok a ***Cuboid*** (Téglatest), ***Ellipsoid*** (Ellipszoid), ***Torus*** (Tórusz), ***Cylinder*** (Henger), ***Cone*** (Kúp) és ***Icosahedron*** (Ikozaéder), amelyek mind a ***Solid*** leszármazottai. Főbb adattagjaik megegyeznek a **„2.2. Alaptestek”** c. fejezetben leírt megadható paraméterekkel, és természetesen öröklik az ősosztályuk mezőit, a viselkedésükről

pedig csak annyit, hogy mindegyik képes a paraméterei alapján elkészíteni a saját poligonhálóját, vagy szükség esetén újragenerálni, ha megváltoznak ezek a paraméterek, és mindegyik tudja, hogy milyen testet ír le, ezt a *SolidType()* függvényükkel lehet lekérdezni. Ez primitívek esetén az osztály neve, összetett testeknél pedig egyszerűen „Solid”. Ezen kívül minden osztályban van egy statikus *counter* adattag, amely tulajdonképpen azt mutatja, hogy ahhoz az osztályhoz hány objektumpéldány tartozik. Minden példányosításnál nő az értéke.

Íme egy rövid kódrészlet, a tórusz poligonhálójának felépítése:

```
public override void MeshTesselation() {
    mesh = new BRep();
    double hstep = 360.0 / (double)hdiv;
    double vstep = 360.0 / (double)vdiv;
    double a = hstep * Math.PI / 180.0;
    double b = vstep * Math.PI / 180.0;

    // A tórusz pontjainak kiszámítása
    for (double u1 = 0; u1 < hdiv; u1 += 1) {
        for (double u2 = 0; u2 < vdiv; u2 += 1) {
            mesh.Vertices.Add(new Point3D(
             (irad + mrad * Math.Cos(u2 * b)) * Math.Cos(u1 * a),
             (irad + mrad * Math.Cos(u2 * b)) * Math.Sin(u1 * a),
              mrad * Math.Sin(u2 * b)));
        }
    }

    // Az élek meghatározása
    for (int i = 0; i < hdiv; ++i) {
        for (int j = 0; j < vdiv; ++j) {
            mesh.Edges.Add(new Edge(i * vdiv + j, ((i + 1) % hdiv) * vdiv + j));
            mesh.Edges.Add(new Edge(i * vdiv + j, i * vdiv + ((j + 1) % vdiv)));
        }
    }

    // A lapok meghatározása
    for (int i = 0; i < hdiv; ++i) {
        for (int j = 0; j < vdiv; ++j) {
            mesh.Faces.Add(new Face(((i + 1) % hdiv) * vdiv + ((j + 1) % vdiv),
                i * vdiv + j, ((i + 1) % hdiv) * vdiv + j));
            mesh.Faces.Add(new Face(i * vdiv + ((j + 1) % vdiv), i * vdiv + j,
                 ((i + 1) % hdiv) * vdiv + ((j + 1) % vdiv)));
        }
    }
}
```

6.1.2. A transzformációkat leíró osztályhierarchia

Az ősosztály az absztrakt ***Transformation***, amelynek meglehetősen egyszerű a felépítése, mindössze a transzformációs mátrixot tárolja, és van egy *SetMatrix()* nevű absztrakt metódusa, amelyet a leszármazottak fognak felüldefiniálni, hogy beállítsák vele a mátrixot. Ezek, a három transzformációtípusnak megfelelően a ***Translation*** (Eltolás), ***Rotation*** (Forgatás) és ***Scale*** (Skálázás) osztályok. A konstruktoraiknak átadható paraméterek megegyeznek a **„3. Transzformációk”** c. fejezetben leírtakkal. A közös ősre azért van szükség, hogy egységesen lehessen tárolni és kezelni a transzformációkat egy generikus listában, amelyet a C#-ban a ***System.Collections.Generic.List<>*** osztály reprezentál.

Következzen az (*sp, ep*) végpontokkal leírt tengely körül *angle* szöggel történő forgatás mátrixának elkészítése:

```
public override void SetMatrix() {
    double delta = angle * Math.PI / 180; // A forgatás szöge radiánban
    // A tengely két végpontjának távolsága h
    double h = Math.Sqrt((ep.X - sp.X) * (ep.X - sp.X) +
        (ep.Y - sp.Y) * (ep.Y - sp.Y) + (ep.Z - sp.Z) * (ep.Z - sp.Z));
    double cx = (ep.X - sp.X) / h;
    double cy = (ep.Y - sp.Y) / h;
    double cz = (ep.Z - sp.Z) / h;
    double d = Math.Sqrt(cz * cz + cy * cy);
    // Az eltolás mátrixa
    Matrix t = new Matrix(4, 4); t.LoadIdentity();
    t[0, 3] = -sp.X; t[1, 3] = -sp.Y; t[2, 3] = -sp.Z;
    // Az X tengely körüli forgatás mátrixa
    Matrix rx = new Matrix(4, 4); rx.LoadIdentity();
    if (d != 0) {
        rx[1, 1] = cz / d; rx[1, 2] = -cy / d;
        rx[2, 1] = cy / d; rx[2, 2] = cz / d;
    }
    // Az Y tengely körüli forgatás mátrixa
    Matrix ry = new Matrix(4, 4); ry.LoadIdentity();
    ry[0, 0] = d; ry[0, 2] = -cx; ry[2, 0] = cx; ry[2, 2] = d;
    // A Z tengely körüli forgatás mátrixa
    Matrix rz = new Matrix(4, 4); rz.LoadIdentity();
    rz[0, 0] = Math.Cos(delta); rz[0, 1] = -Math.Sin(delta);
    rz[1, 0] = Math.Sin(delta); rz[1, 1] = Math.Cos(delta);
    // A végső forgatási mátrix
    TrMatrix = t.Trans() * rx.Trans() * ry.Trans() * rz * ry * rx * t;
}
```

6.1.3. Segédosztályok

Ide tartoznak azok az osztályok, amelyek a program alapvető eszközeit írják le, amelyek nélkülözhetetlenek az adatok tárolásához, vagy a megjelenítéshez.

A ***Point3D*** osztály a háromdimenziós euklideszi tér egy pontját reprezentálja, de ugyanezt az osztályt használom a vektorok tárolására is, ezért implementálva vannak benne az **„1.1. Pontok, vektorok”** c. fejezetben leírt vektorműveletek, valamint tárolja az adott pont színét és koordinátáit.

A ***Matrix*** osztály az alapvető mátrixműveleteken kívül (összeadás, kivonás, szorzás, transzponálás) megvalósítja a „mátrix és pont szorzata” műveletet is, amely nagyon hasznos a pontok transzformálásánál és leképezésénél. Ezen felül automatikusan képes egységmátrixot és nullmátrixot generálni.

A ***Face*** és ***Edge*** osztályok írják le a lap és él grafikus objektumokat. Nagyon hasonlítanak egymáshoz, mindkettő tárolja az adott objektum színét, a különbség, hogy az ***Edge*** egy kezdő- és egy végpontot tárol, a ***Face*** pedig három csúcspontot.

A ***BRep*** osztály képezi a testmodellezés alapját, mivel egyrészt a B-Rep modell felépítését reprezentálja, másrészt hasznos metódusokat tartalmaz a megjelenítéshez, például a hátsó lapok eldobását, a mélységi rendezést, illetve a konstans- és Gouraud-árnyalás fényeinek számítását. A legfontosabb adattagjai a 3D-s pontok, élek és lapok listája, ezeken felül tárolja még a 2D-be leképezett pontok listáját, a „hátsó lapok eldobása” algoritmus által kiválogatott látható lapok listáját, valamint a lapok súlypontjait, valamint egy listát a fények számításához. Ez utóbbi konstans árnyalás esetén a lapokhoz tartozó fényerősséget tárolja, Gouraud-árnyalás esetén pedig a csúcspontokhoz tartozó intenzitást.

A következő kódrészlet a lapok színének intenzitását számítja ki a konstans árnyaláshoz:

```
public void CalcLights(Point3D lamp) {
    // Az intenzitást csak a látható lapokra számoljuk!
    for (int i = 0; i < visibleFaces.Count; i++) {
        Face f = (Face)visibleFaces[i];
        // A lap középpontja
        Point3D cp = (vertices[f[0]] + vertices[f[1]] + vertices[f[2]]) / 3;
        // A lap normálvektora
        Point3D n = (vertices[f[1]] - vertices[f[0]]) %
                    (vertices[f[2]] - vertices[f[0]]);
        // A lapközéppontból a fényforrásba mutató vektor
        Point3D v = lamp - cp;
        // v és n közbezárt szögének koszinusza
        double d = (v * n) / (v.Norm() * n.Norm());
        if (d < 0) d = 0;
        lights.Add(d);
    }
}
```

Végül pedig a legfontosabb osztály, az ***Operations***. Ez tartalmazza az összes olyan függvényt, amely az **„5. A halmazműveletek algoritmusa”** c. fejezetben le van írva. Fő metódusa a *Construct()*, amelynek bemenő paramétere a két testet leíró objektum (két ***Solid*** vagy annak leszármazottja) és a halmazművelet típusát meghatározó karakterlánc, amely lehet „union”, „intersection” vagy „difference”; ezek ismeretében a már leírt módon elvégzi a kijelölt halmazműveletet, eredményként pedig szintén egy ***Solid*** típusú objektumot ad vissza. Az ***Operations*** összes többi metódusa tulajdonképpen ennek a műveletnek a részfeladatait látja el.

Az implementálásnál itt találkozunk egy nagyon fontos problémával, mégpedig a számítási pontatlanság jelenségével. Előfordulhat, hogy két nagy pontosságú lebegőpontos érték összehasonlításakor az elvárttól különböző eredményt kapunk. Ez azért van, mert a két szám akár 10^{-5}-en nagyságrendig megegyezik, utána viszont különböző, és a rendszer ezeket különbözőként is kezeli, holott nekünk az lenne a jó, ha egyformának látná. Sok-sok tesztelés után derült ki ez a probléma, mivel ilyen esetben a program állandóan végtelen ciklusba esett. Ennek megoldására bevezettem az *EPS* konstanst, amely egy nullától különböző, de nagyon kicsi szám, értéke 10^{-6}. Ezután az egyenlőségvizsgálatokat átírtam $a = b$ helyett $|a - b| < EPS$ alakúra, és így tökéletesen működött. Az eredmény szempontjából ez a változtatás irreleváns, mindössze programozástechnikai jelentősége van.

Végül következzen az ***Operations*** osztály egy statikus függvénye, amely megadja, hogy egy *p* pont benne van-e egy vele egy síkban lévő *a, b, c* csúcsok által meghatározott háromszögben.

```
private static bool PointInTriangle(
      Point3D p, Point3D a, Point3D b, Point3D c) {
    Point3D u = b - a;
    Point3D v = c - a;
    Point3D w = p - a;
    double si, ti;
    double uv = (u * v), wv = (w * v), vv = (v * v),
           wu = (w * u), uu = (u * u);
    double denom = (uv * uv - uu * vv);
    if (Math.Abs(denom) < EPS) return false;
    si = (uv * wv - vv * wu) / denom;
    ti = (uv * wu - uu * wv) / denom;
    if (si >= -EPS && ti >= -EPS && si + ti <= 1.0 + EPS) return true;
      else return false;
}
```

6.1.4. „Form” osztályok

Ezek összessége képezi a program ablakozó rendszerét. Alapja a főablak, ebben történik minden. Ez egy ún. MDI form, ami azt jelenti, hogy mintegy konténer objektumként szolgál a benne elhelyezhető kisebb ablakok számára. Ezen található a főmenü is.

Minden primitívhez tartozik egy form, amely a test paramétereinek megadására szolgál. Ezek neve a New előtaggal kezdődik, majd jön a megfelelő test neve (pl. ***NewTorus***). Hasonlóképpen minden transzformációhoz is tartozik egy paraméterező ablak, amelynek neve a transzformáció neve + Form utótag (pl. ***ScaleForm***).

A legfontosabb a testek megjelenítésére szolgáló ablak, a ***SolidForm***. Ezen lehet beállítani a megjelenítési módot, transzformációkat és színeket megadni, illetve van egy gomb, amely a test paramétereinek megváltoztatására szolgál, természetesen ez csak primitívek esetén aktív, összetett testeknél értelmetlen lenne. Itt található továbbá a rajzolási terület, amelyen az OpenGL motor végzi el a megjelenítést, erről a következő részben lesz szó. Egy ***SolidForm*** csak egy testet képes megjeleníteni.

Ennek egyszerűsített változata a ***Preview*** form, amely, mint az a nevéből is látszik, előnézetre való. Egészen pontosan a halmazművelet eredményét lehet vele megtekinteni, és eldönthetjük, hogy megtartjuk, vagy elvetjük a kapott testet. Ha megtartjuk, akkor az megjelenik egy ***SolidForm***-on, ha nem, elvész. Funkcióit tekintve meg lehet adni itt is a megjelenítési módot, illetve ki- és bekapcsolni a koordinátarendszer megrajzolását (ez a ***SolidForm***-on is megvan).

6.2. CsGL – OpenGL motor C#-hoz

A CsGL tulajdonképpen egy ingyenes, C#-hoz készült grafikus függvénykönyvtár, amely OpenGL technológiát használ a megjelenítéshez, ennek megfelelően megtalálható benne szinte mindegyik OpenGL függvény. A használatához szükségesek a `csgl.dll` és a `csgl.native.dll` nevű fájlok.

Ahhoz, hogy egy ilyen felületre rajzolhassunk, egyszerűen készíteni kell egy saját komponenst, amelyet az ***OpenGLControl*** osztályból kell származtatni, majd ezt a komponenst elhelyezhetjük egy ablakban, és a *Draw()* metódusát felüldefiniálva rajzolhatunk rá, OpenGL utasításokat használva. A következő kódrészlet megmutatja, hogyan is néz ki egy ilyen felhasználói komponens:

```
using System;
using System.ComponentModel;
using System.Windows.Forms;
using CsGL.OpenGL;

namespace CSGBuilder {
    public partial class MyControl : OpenGLControl {

        public MyControl() {
            InitializeComponent();
        }

        // A rajzolás itt történik
        public override void glDraw() {
            // Törlés
            GL.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
            // Egy háromszög kirajzolása
            GL.glColor3d(0, 0, 255);
            GL.glBegin(GL.GL_TRIANGLES);
            GL.glVertex2d(30, 30);
            GL.glVertex2d(50, 80);
            GL.glVertex2d(60, 10);
            GL.glEnd();
        }

        // OpenGL inicializáció
        protected override void InitGLContext() {
            GL.glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
            GL.glShadeModel(GL.GL_SMOOTH);
            GL.glEnable(GL.GL_DEPTH_TEST);
        }

        // Méretezéskor módosítja az ablakot
        protected override void OnSizeChanged(EventArgs e) {
            base.OnSizeChanged(e);
            GL.glMatrixMode(GL.GL_PROJECTION);
            GL.glLoadIdentity();
            GL.gluOrtho2D(0.0, Size.Width, 0.0, Size.Height);
        }
    }
}
```

Azért választottam ezt a technológiát, mert gyors megjelenítést tesz lehetővé, és a segítségével egyszerűen megvalósítható például a Gouraud-árnyalás, mivel képes a háromszöglap csúcspontjai között interpolálni a színeket. Ugyanez saját kezűleg lekódolva sokkal lassabb futást produkálna. Ezen kívül beállítható az élsimítás (antialiasing), amely nagyobb mértékben javítja a kép minőségét, mint amennyivel lassítja a megjelenítést.

6.3. Szerializáció

A program képes az elkészített testeket lemezre menteni, illetve visszatölteni, mindezt az objektumszerializáció segítségével. Ez a technológia a következőképpen működik:
Minden osztályban, amelynek valamely információját tárolni szeretnénk, implementálni kell a `System.Runtime.Serialization.ISerializable` interfészt, és ki kell bővíteni egy *GetObjectData()* nevű metódussal, ebben adhatjuk meg a szerializálandó adattagokat. Vegyük példának a ***Matrix*** osztályt, amelynek a *rows, cols* mezői egész értékűek, és rendre a sorok és az oszlopok számát tárolják, az *m* adattagja pedig egy kétdimenziós tömb, amelyben a lebegőpontos adatok vannak. Így a ***Matrix*** osztályban ez a metódus így néz ki:

```
public void GetObjectData(
    System.Runtime.Serialization.SerializationInfo info,
    System.Runtime.Serialization.StreamingContext context) {
    info.AddValue("Rows", rows);
    info.AddValue("Cols", cols);
    info.AddValue("Data", m);
}
```

A `System.Runtime.Serialization.SerializationInfo` osztály tárolja a megadott adattagokat kulcs-érték párok formájában. Ha értékként egy objektumot adunk meg, akkor annak az objektumnak is szerializálhatónak kell lennie, különben futás közben kivétel keletkezik. C#-ban a beépített egyszerű típusok mind szerializálhatóak.
Ezután definiálni kell az osztálynak egy új konstruktort, amely ugyanilyen paramétereket kap, és pont a fordítottját teszi, mint a *GetObjectData()*, azaz a szerializációs információk alapján értékeket rendel az adattagokhoz.

```
public Matrix(System.Runtime.Serialization.SerializationInfo info,
              System.Runtime.Serialization.StreamingContext context) {
    rows = info.GetInt32("Rows");
    cols = info.GetInt32("Cols");
    m = (double[,])info.GetValue("Data", typeof(double[,]));
}
```

Ezután a tényleges mentés és betöltés a ***BinaryFormatter*** osztály *Serialize()* és *Deserialize()* metódusával történik. Az alábbi példa ezt egy ***Matrix*** objektumon keresztül szemlélteti:

```
// Szerializáció
Stream s = File.Open(FileName, FileMode.Create);
BinaryFormatter b = new BinaryFormatter();
Matrix m = new Matrix(3, 3);
b.Serialize(s, m);
s.Close();

// Deszerializáció
Stream s = File.Open(FileName, FileMode.Open);
BinaryFormatter b = new BinaryFormatter();
Matrix m = (Matrix)b.Deserialize(s);
s.Close();
```

7. A szoftver használata

Ez a fejezet afféle útmutatóként szolgál a program kezeléséhez. Szeretném bemutatni a főbb funkcióit, illetve tippeket adni az egyszerű és gyors kezeléshez.

A program a „CSGBuilder.exe”-vel indítható, amely mellett ott kell lennie a „csgl.dll” és „csgl.native.dll” állományoknak is, illetve legyen telepítve a számítógépen Microsoft .NET Framework 2.0.

A szoftver az univerzitás kedvéért teljesen angol nyelvű, de meglehetősen egyszerű nyelvezetű, egy informatikai angoltudás bőven elegendő a kezeléséhez.

7.1. A főablak és a menürendszer

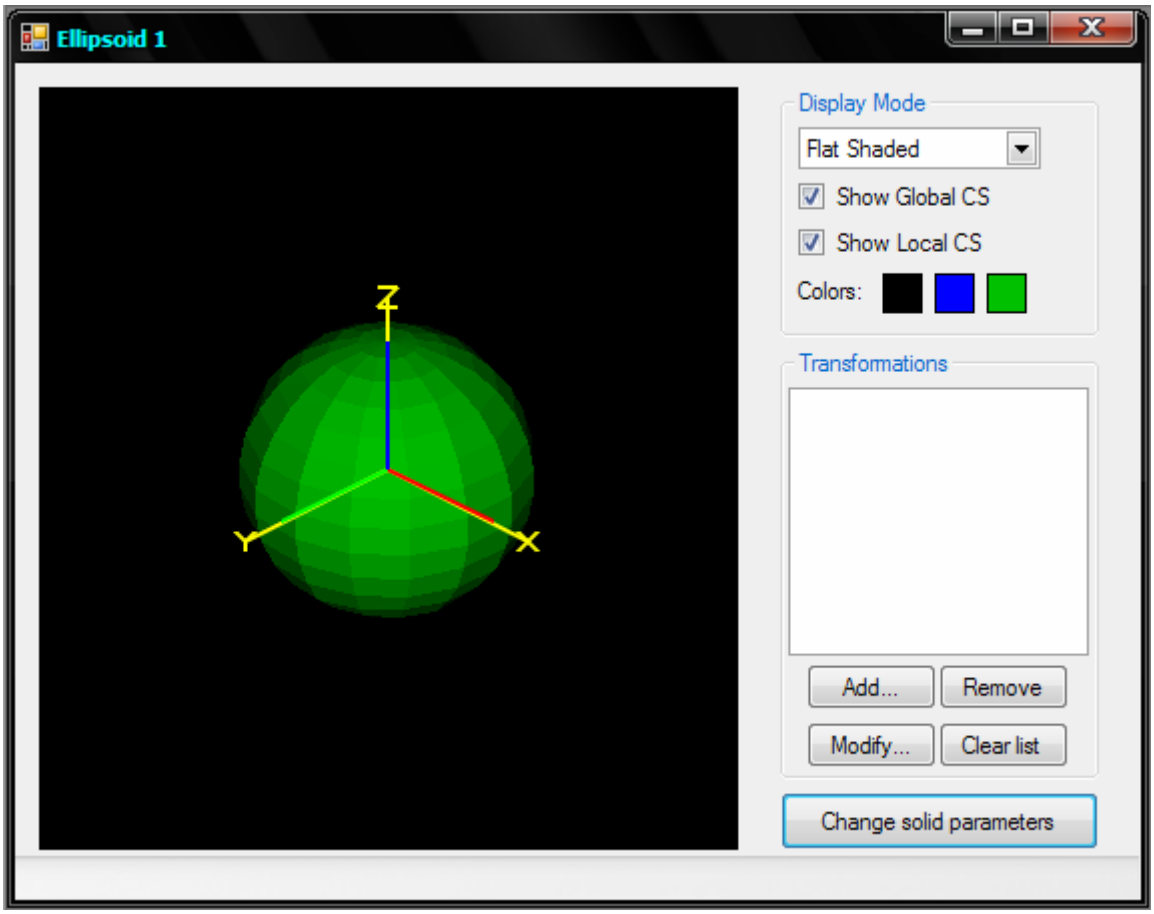
Első lépésként lássuk a főablak felépítését. Felül a főmenüsor található, a fejlécben pedig az éppen megnyitott fájl neve olvasható, indításkor ez „untitled.csg”. Ez egy MDI ablak, azaz minden ebből megnyitott kisebb ablak úgymond „beleágyazódik”, ezért mintegy tárolóként szolgál, és összefogja a kisebb ablakokat, amelyekben a testek fognak megjelenni. Ezen kisebb ablakok együttesét, a bennük lévő testekkel együtt hívjuk modellnek ebben a környezetben.

A menüsorban a szokványos „File”, „Window” és „Help” menüpontokon kívül megtalálható a „Solids” és az „Operations” menü. Következzen most ezeknek a bemutatása.

- **File:** Itt a szokásos fájlműveleteket találjuk.
  - **New model:** Eltünteti az összes megnyitott ablakot, lezárja az éppen megnyitott fájlt, és új modellt kezd. Ha a munkánk nem volt elmentve, figyelmeztet, és felajánlja a mentési lehetőséget.
  - **Open:** Megnyithatunk egy elmentett modellt, amelyet hozzáad az aktuális modellhez. Ezáltal több modellt is összekapcsolhatunk. Ha csak a megnyitott modellt szeretnénk látni, akkor a megnyitás előtt válasszuk a **New model** menüpontot.
  - **Save:** Elmenti az aktuális modellt. Ha még nem volt elnevezve, akkor rákérdez a nevére, különben felülírja a régebbi mentést.
  - **Save as…:** Ugyanaz, mint a **Save**, de mindig rákérdez a fájl nevére.
  - **Exit:** Bezárja a programot. Ha a munka nem volt elmentve, figyelmeztet.

- **Solids:** Ez a menüpont tartalmazza a testek létrehozására és törlésére szolgáló műveleteket.
    - **Add primitive:** Itt adhatunk hozzá egy új testet a modellünkhöz. Ha kiválasztjuk a **Cuboid, Ellipsoid, Torus, Cone, Cylinder, Icosahedron** menüpont valamelyikét, akkor megjelenik a megfelelő testhez tartozó paraméterező dialógusablak, ahol megadhatjuk a test jellemzőit, majd elkészíti az objektumot, és megjeleníti egy ablakban.
    - **Delete current solid:** Megerősítés után eltünteti a legfelül lévő ablakot, ezáltal törli a benne lévő testet.
    - **Delete all:** Törli az összes testet.
- **Operations:** Ebből a menüpontból érhetjük el a halmazműveleteket. Ha kiválasztunk egy halmazműveletet, meg kell adni a két operandusát, azaz a két testet (különbségnél fontos a sorrendjük!). A művelet elvégzése után megjelenik egy ***Preview*** ablak, ahol megtekinthetjük az eredményt, és eldönthetjük, hogy megtartjuk vagy nem.
    - **Union:** Az únió művelet.
    - **Intersection:** A metszet művelet.
    - **Difference:** A különbség művelet.
- **Window:** Itt az átláthatóság kedvéért különféle elrendezési lehetőségek vannak az ablakok számára, például a lépcsőzetes vagy mozaik elrendezés.
- **Help:** Súgó menü.
    - **Hotkeys:** Az ablakokban használható gyorsbillentyűket írja le, amelyek megkönnyítik a testek gyors transzformálását vagy a megjelenítési módok közötti váltást.
    - **About CSGBuilder:** A program névjegye.

7.2. A testek megjelenítése – *SolidForm*



21. ábra SolidForm

Amikor egy új testet hozunk létre, vagy egy halmazművelet eredményét elfogadjuk, megjelenik egy ablak, amelyben azután kezelhetjük a testmodellt, ez a ***SolidForm***. Az ablak fejlécében a benne lévő test típusa van és sorszáma. A típust a *SolidType()* függvény adja meg, a sorszámát pedig az objektum osztályának már említett statikus adattagja határozza meg (pl. Ellipsoid 1, Solid 3, stb.). Ez az információ minden ablaknál egyedi, így ezzel azonosíthatóak a halmazműveleteknél.

A „form” jobb oldalán különféle vezérlők találhatók. A *„Display Mode”* csoportban van egy lenyíló lista, amelyből kiválaszthatjuk a megjelenítés módját. Ezt megtehetjük úgy is, hogy megnyomjuk a megfelelő gyorsbillentyűt, ez pedig a Ctrl+1-től a Ctrl+5-ig terjed a megjelenítési módoknak megfelelően. Alatta két jelölőnégyzet található, amelyekkel ki- és bekapcsolhatjuk a „globális” és „lokális” koordinátarendszer megjelenítését. A globális

koordinátarendszer a háromdimenziós euklideszi tér ortonormált bázisvektorainak elhelyezkedését mutatja, és mindig változatlan. A lokális koordinátarendszer igazából csak arra való, hogy mutassa a test állapotát a kiinduláshoz képest, ugyanis ez a testtel együtt transzformálódik.

A csoportban még van három gomb, amelyekkel a pontok, élek és lapok színét módosíthatjuk. Ha egy test valamely alkotóelemének a színét módosítjuk, akkor a régi színét már nem lehet visszaállítani. Ez akkor lehet fontos, ha összetett testről van szó. Például, ha a modellt egy kék és egy zöld lapokból álló testből raktunk össze, akkor a megfelelő testből „örökölt" lapok is kék, illetve zöld színűek lesznek. Azonban, ha megváltoztatjuk ezen test lapjainak a színét, mondjuk pirosra, akkor az összes lapja piros lesz, és többé nem lehet látni, hogy melyik lap melyik eredeti testből való.

A *„Transformations"* csoportban van egy lista, amelybe a testre alkalmazott transzformációk kerülnek szöveges formában, és alatta az ezt kezelő gombok. Az *„Add…"* gombra kattintva megjelenik egy kis helyi menü, amelyből kiválaszthatjuk a transzformáció típusát, majd kapunk egy dialógusablakot, amelyben megadhatjuk a paramétereket. A *„Modify…"* gombbal a listában kiválasztott transzformáció paramétereit módosíthatjuk egy ugyanilyen dialógusablakban. A *„Remove"* gombbal a kijelölt listaelemet törölhetjük, a *„Clear list"* gombbal pedig az összest, természetesen csak megerősítés után.

Az ablak jobb alsó sarkában található még egy gomb, a *„Change solid parameters"*, amellyel a test paramétereit változtathatjuk meg. Ez a gomb csak primitívek esetén használható, összetett testeknél inaktív, hiszen ott nincs is értelme.

Az ablak bal oldalán található a megjelenítő felület. Ha itt kattintunk, majd az egér bal gombját lenyomva tartva valamilyen irányba húzzuk, akkor változtatjuk a kamera pozícióját, és más szögből nézhetjük a testet. Ha ugyanezt a jobb egérgomb lenyomása mellett tesszük, akkor nagyíthatjuk és kicsinyíthetjük a testet, azaz „zoomolhatunk".

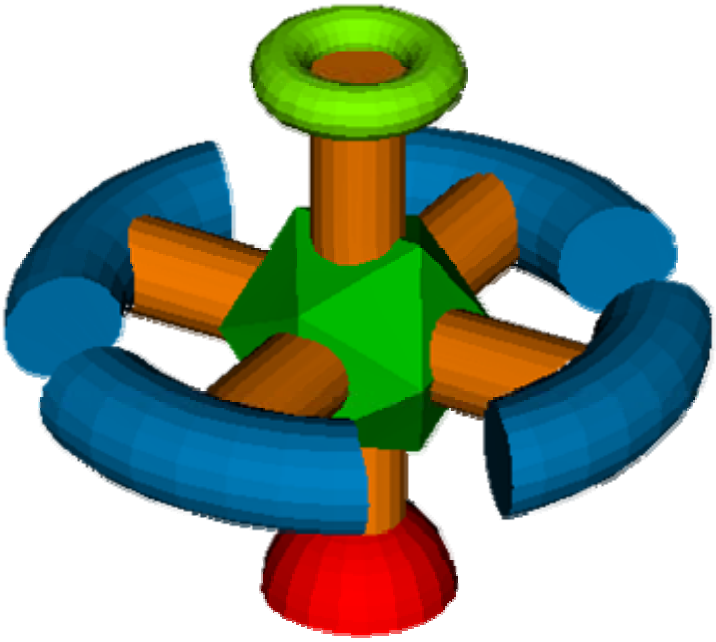
A gyors transzformálás érdekében is léteznek gyorsbillentyűk. Az eltoláshoz a 'T' billentyű tartozik, a forgatáshoz az 'R', a skálázáshoz az 'S', ezek valamelyikét kell lenyomva tartani a kívánt transzformációhoz, és mellé még valamelyik koordinátatengely betűjelét, azaz az 'X', 'Y' vagy 'Z' gombot. Ezután a bal egérgombot lenyomva és az egeret húzva balra és jobbra azonnal transzformálhatjuk a testet. Tehát ha például el akarjuk forgatni a testet az X tengely körül, akkor nyomva tartjuk az 'R' és az 'X' gombokat, lenyomjuk az egér bal gombját, és húzzuk vízszintesen valamelyik irányba. Ezzel a módszerrel persze nem lehet

pontosan megadni a transzformáció paramétereit, arra használjuk a jobb oldalon lévő *„Add…”* gombot, de ha nem fontos a nagy pontosság, akkor így nagyon gyorsan megadhatjuk a transzformációkat.

7.3. A halmazműveletek elvégzése

A három CSG halmazműveletet az *„Operations”* menüpontból érhetjük el. Válasszuk ki a művelet típusát, ezután egy kis dialógusablakban legördülő listákból kiválaszthatjuk a két testet, amelyen a műveletet el szeretnénk végezni. Vigyázzunk, a különbség műveletnél számít a sorrend, mivel ez nem kommutatív, a másik kettőnél viszont tetszőleges. Ha rákattintottunk az **OK** gombra, megjelenik egy folyamatjelző, amelyen jelzi, hogy hol tart a művelet elvégzése. Nagy, sok lapból álló testeknél ez több másodpercet is igénybe vehet.

Ezután megjelenik az előnézeti ablak, benne a halmazművelet eredményével. Itt ugyanúgy lehet forgatni a testet és nagyítani / kicsinyíteni, mint a ***SolidForm***-on, illetve ki- és bekapcsolhatjuk a koordinátarendszerek megrajzolását, és váltogathatjuk a megjelenítési módokat. Ha elfogadjuk a létrejött modellt, kattintsunk az **Accept** gombra, ekkor a testmodellünk megjelenik egy ***SolidForm***-on, ahol már alkalmazhatunk rá transzformációkat, és át is színezhetjük, illetve felhasználhatjuk egy újabb halmazművelet egyik operandusaként.



22. ábra Egy összetett test

Utószó

A CSGBuilder egy meglehetősen összetett program, amelynek elsődleges célja az volt, hogy egy megfelelő algoritmust implementáljon a CSG halmazműveletek elvégzésére háromdimenziós testeken. Úgy gondolom, hogy ez sikerült is. A testek többféle megjelenítési módja csak kisebb cél volt, de ezt is sikerült megoldani. Továbbá létrehoztam olyan osztályhierarchiákat, amelyekkel mind a testmodellek, mind pedig a transzformációk könnyen kezelhetőek. Ezek az osztályok, csakúgy, mint a halmazműveleteknél szereplő algoritmusok önállóan, minden további módosítás nélkül átvihetőek más szoftverekbe, így az újrafelhasználhatóság szempontjából is hatékony lett a program.

Az előszóban azt mondtam, hogy célom egy olyan szoftver létrehozása, amely mindenki számára könnyen kezelhető. Ezt sikerült elérnem, hiszen egérrel szinte minden elvégezhető, például a kamera forgatása és a „zoomolás”, csak néhány számot kell kézzel beütni a transzformációknál, illetve a testek paraméterezésénél.

Egyetlen dolog maradt ki a programból, az exportálhatóság lehetősége. Ez tulajdonképpen egy olyan cél volt, amelyről úgy gondoltam, hogy elegendő idő esetén bekerülhet az implementációba, de ez sajnos nem így történt. Viszont a jövőben mindenképpen bele fogom építeni a szoftverbe, hogy az elkészített modellek ezáltal átvihetőek legyenek más CAD rendszerekbe, és ott tovább lehessen fejleszteni őket.

Irodalomjegyzék

Antal György, Csonka Ferenc, Szirmay-Kalos László: Háromdimenziós grafika, animáció és játékfejlesztés, ComputerBooks Kft., 2003.

A felhasznált alapvető algoritmusok a következő webcímeken érhetők el:

http://softsurfer.com/Archive/algorithm_0102/algorithm_0102.htm

http://softsurfer.com/Archive/algorithm_0104/algorithm_0104.htm

http://softsurfer.com/Archive/algorithm_0104/algorithm_0104B.htm

http://softsurfer.com/Archive/algorithm_0105/algorithm_0105.htm

A CsGL honlapja:

http://csgl.sourceforge.net/index.html

A használatáról további hasznos információk és kódrészletek találhatók itt:

http://www.developerfusion.co.uk/show/3823/3/