Tornai Róbert

Fejezetek a számítógépi grafikából

mobiDIÁK könyvtár

Tornai Róbert

Fejezetek a számítógépi grafikából

mobiDIÁK könyvtár

SOROZATSZERKESZTŐ

Fazekas István

Tornai Róbert

Fejezetek a számítógépi grafikából

Egyetemi segédanyag
első kiadás

mobiDIÁK könyvtár
Debreceni Egyetem
Informatikai Kar

Lektor

Dr. Szabó József

Copyright © Tornai Róbert, 2004

Copyright © elektronikus közlés mobiDIÁK könyvtár, 2004

mobiDIÁK könyvtár
Debreceni Egyetem
Informatikai Kar
4010 Debrecen, Pf. 12
http://mobidiak.unideb.hu

A mű egyéni tanulmányozás céljára szabadon letölthető. Minden egyéb felhasználás csak a szerző előzetes írásbeli engedélyével történhet.
A mű a *A mobiDIÁK önszervező mobil portál* (IKTA, OMFB-00373/2003) és a *GNU Iterátor, a legújabb generációs portál szoftver* (ITEM, 50/2003) projektek keretében készült.

Tartalomjegyzék

Előszó ... 9

I. Alapvető fogalmak, irányelvek ... 11
1. Történeti áttekintés ... 11
2. Algoritmus fogalma ... 12
3. Bolondbiztosság ... 12
4. Kódolási megfontolások ... 13
5. Homogén koordináták használata ... 13

II. Inkrementális algoritmusok ... 19
1. Szakasz rajzolása ... 19
2. Kör rajzolása ... 21

III. Vágó algoritmusok ... 27
1. Cohen-Sutherland algoritmus ... 27
2. Alakzatra való lehatárolás ... 30
2.1. Kinn-benn algoritmus konvex sokszögekre ... 30
2.2. Szakasz vágása konvex sokszögre ... 33
2.3. Kinn-benn algoritmus konkáv sokszögekre ... 36
2.4. Szakasz vágása konkáv sokszögre ... 39

IV. Harmadrendű görbék ... 45
1. Hermite-ív ... 46
2. Bézier görbe ... 48
3. B-spline ... 49

V. 3D ponttranszformációk ... 53
1. Egybevágósági transzformációk ... 53
1.1. Eltolás ... 53
1.2. Forgatás ... 54
1.2.1. Forgatás az x tengely körül ... 54
1.2.2. Forgatás az y tengely körül ... 55
1.2.3. Forgatás a z tengely körül ... 55
1.3. Tükrözés koordinátasíkra ... 56
1.3.1. Tükrözés az $[y, z]$ koordinátasíkra ... 56
1.3.2. Tükrözés az $[x, z]$ koordinátasíkra ... 57
1.3.3. Tükrözés az $[x, y]$ koordinátasíkra ... 58
2. Hasonlósági transzformációk ... 59

2.1. Origó középpontú kicsinyítés és nagyítás 59
3. Affin transzformációk 59
3.1. Skálázás 59
3.2. Nyírás 60

VI. 3 dimenziós tér leképezése képsíkra és a Window to Viewport transzformáció 63
1. Leképezések 63
1.1. Centrális vetítés 63
1.2. Párhuzamos vetítés 64
1.3. Axonometria 66
2. Window to Viewport transzformáció 67

VII. Rekurzív kitöltés 71

VIII. Testmodellezés 73
1. Drótvázmodell (Wire Frame Model) 73
2. Felületmodell (B-rep adatstruktúra) 73
3. Testmodellek megjelenítése 74
3.1. Hátsó lapok eltávolítása 75
3.2. Festő algoritmus 75
3.3. Z-buffer algoritmus 77

Szójegyzet 83

Ajánlott Irodalom 85

Előszó

Mottó: „Nem számít semmi, csak az, hogy kérdezz, keresd a választ!"
(Tankcsapda)

Jelen jegyzet összeállításánál igyekeztem az évek során összegyűjtött tapasztalatokat beépíteni az egyes fejezetekbe. Mindemellett megpróbáltam a hangsúlyt a gyakorlati alkalmazásra fektetni.

A példákat C++-ban készítettem el, a kódolás során pedig a Microsoft Visual C++ 6.0-ás környezetet használtam. A számítógépes kódokat könnyen fel lehet majd ismerni `speciális` szedésükről. A kódolás során a lebegőpontos számokat a konvekcióknak megfelelően a tizedespont utáni egy darab 0-val jelöltem amennyiben nem volt tizedestört része a számnak. Például: `d = 5.0 / 4.0 - r;`.

Az algoritmusok hatékony elkészítéséhez szükséges volt néhány segédosztály implementálása. Ilyen például a síkbeli illetve térbeli pontok tárolására és kezelésére szolgáló `CPoint2D` és `CPoint3D` osztályok, melyek az alapvető műveleteken (összeadás, kivonás) kívül képesek akár a skaláris vagy a vektoriális szorzatot is előállítani, vagy a `CMatrix` osztály, mely a zérus illetve egységmátrix létrehozásán kívül pédául a mátrixok szorzását is implementálja. Nagyon hasznos továbbá a szilárd testek modellezésénél a `CB_Rep`, mely mind a geometriai, mind a topológiai információkat tárolja. Az ablaktechnikák pedig jó hasznát veszik a `CWindowBase`-nek, mely konvex alakzatok leírását valósítja meg, illetve a `CConcaveWindow`-nak, mely az előbbi osztály segítségével konkáv alakzatok leírását teszi lehetővé. A közölt kódrészletekben egy `eBuffer` felsorolt típusú változó fogja szabályozni, hogy az adott függvény közvetlenül a képernyőre (`Front`) vagy egy háttérképernyőre (`Back`) rajzol.

Ezúton mondok köszönetet Dr. Szabó Józsefnek, Dr. Schwarcz Tibornak illetve Tomán Henriettának a lektorálásért és Fazekas Saroltának a szemléletes ábrákért.

I. fejezet
Alapvető fogalmak, irányelvek

Ahogy a technika fejlődik, sok leendő programozónak illetve programtervezőnek nincs megfelelő tapasztalata és tudása a programozott hardverről. Gondolok itt az egyes utasítások időigényétől kezdve, a grafikus hardverek speciális tulajdonságáig sok mindenre. Sajnos sokan az algoritmus fogalmát sem tudják pontosan. Ebben a fejezetben egy rövid történeti áttekintés után a jegyzetben követett elveket mutatom be, illetve a homogén koordináták fogalmát ismertetem.

1. Történeti áttekintés

A számítógépi grafika annak idején nagyszámítógépeken kezdődött. Komolyabb (pl.: orvosi, tervezési) feladatokhoz a mai napig célszámítógépeket használnak. Azonban a személyi számítógépek az elmúlt időben olyan mértékű fejlődésnek indultak, hogy a valós idejű animációk kivitelezése sem lehetetlen rajtuk, vagy akár egy házimozi rendszer lelkéül is szolgálhatnak.

A grafikai megoldások fejlődését eleinte a konzol játékgépek és a személyi számítógépes játékok okozták. Napjainkban egyre több tervezői illetve szimulációs probléma oldható meg hatékonyan PC segítségével. A fejlődést a következő felsoroláson keresztül követhetjük nyomon.

- 80-as évek eleje: a felbontás 320×200 pixel, a használható színek száma 4, melyet 16 alapszínből lehet kiválasztani. CGA videókártya - CGA monitor páros. Videómemória nagysága kb. 64KB.
- 80-as évek közepe-vége: megjelentek az EGA videokártyák max. 256KB memóriával. Felbontásuk 640×480 pixel 64 szín használatával. Emellett teret hódítottak a Hercules kártyák a hozzájuk tartozó monokróm monitorokkal, ugyanis a színes monitorok abban az időben magánember számára szinte megfizethetetlenek voltak. A Hercules kártyák nagyobb (758×512) felbontást nyújtanak ugyan, de csak fekete-fehér grafika mellett. Megjelentek a különféle emulációk az egyes működési módok között.
- 90-es évek eleje: A VGA kártyák 256KB memóriától egészen 4MB kivitelig kaphatóak voltak. A 640×480-as működési módot minimum teljesítették, azonban a több memóriával rendelkező darabok akár egészen a 2048×1536-os felbontást is tudták kezelni. Itt jelent meg először a 65536 színű (16 bites) üzemmód, majd később a 16.7 millió színű (24 bites) ábrázolás. Látható, hogy a felbontás és a pixelenként tárolt egyre több színinformáció egyre nagyobb memóriát igényel.

- 90-es évek vége: megjelentek a 3D gyorsítást végző modellek. Napjainkban memóriájuk 4MB-tól 256MB-ig terjed, azonban akár még ennél is többel rendelkező modellek megjelenése sem lehetetlen a közeljövőben. Kezdetben csak célfeladatokat gyorsítottak, azonban manapság külön programozható a videokártyák GPU-ja shader programok segítségével.

A személyi számítógépek háttértára, egyéb paraméterei is jelentős fejlődésen mentek és mennek keresztül. A legszorosabban talán a CPU fejlődése kapcsolódik a videókártyán kívül a számítógépi grafikához, ugyanis a bonyolultabb grafikák létrehozásához komoly számolókapacitásra van szükség. Eleinte csak fixpontos számolások elvégzésére volt lehetőség. Ekkor az emulált lebegőpontos számolások igencsak komoly teljesítményveszteséggel jártak. Lehetőség volt ugyan coprocesszor használatára, de amellett, hogy költséges volt a beszerzése, sosem lett 100%-osan megbízható egy ilyen rendszer. Ennek köszönhető, hogy az inkrementális algoritmusok jelentős fejlődésnek indultak abban az időben. Később, a 90-es évek közepétől integrálták a lebegőpontos egységet a CPU-val. Ez újabb lendületet adott a grafikus alkalmazásoknak.

2. Algoritmus fogalma

Az algoritmusokra teljesül az alábbi négy kritérium mindegyike.

- **Helyes:** az algoritmus a kitűzött célokat valósítja meg, a célfeltételeknek megfelelő végeredményt szolgáltatja az összes lehetséges input esetén.
- **Teljes:** a problémaosztály összes problémájára megoldást nyújt, ellenőrzi az inputot, hogy az értelmezési tartományból való-e.
- **Véges:** véges sok elemi lépés eredményeképpen minden megoldási úton eredményt szolgáltat, nem kerül végtelen ciklusba.
- **Determinisztikus:** nem tartalmaz véletlenszerű elemet a megoldás során, ugyanarra a bemeneti kombinációra minden egyes futás során ugyanazt az eredményt szolgáltatja.

Az algoritmusok megalkotása során lényeges szempont, hogy az adott problémaosztály minden egyes problémájára megoldást nyújtsunk. Ez a gyakorlatban azt jelenti, hogy a megoldást szolgáltató függvény formális paraméterlistáján veszi át az összes változó paramétert. Emellett szükség szerint lokális változókat is definiálhatunk, azonban a függvény környezetét nem változtathatjuk meg. Ez biztosítja majd a létrehozott kód hordozhatóságát.

3. Bolondbiztosság

A jegyzetben közölt algoritmusok első lépése az lesz, hogy amennyiben valamelyik paraméter hibás értékkel rendelkezhet (nem az értelmezési tartományból való), kezeli azt, mielőtt a hibás érték problémát okozhatna. (Jegyzetemben általában a 0-val való osztás elkerülése a cél, ugyanis a többi matematikai művelet, ha értelmetlen is, de végrehajtható számítógépen. 0-val való osztáskor azonban az egész program futása

megszakadhat.) Mindezek tükrében fontos, hogy a bolondbiztosságot mindig szem előtt tartsuk.

4. Kódolási megfontolások

Fontos szempont az is, hogy az algoritmus milyen sebességgel hajtja végre feladatát. A felhasznált műveleteknél azt kell figyelembe venni, hogy a fixpontos műveletek nagyságrendileg gyorsabban hajthatók végre, mint a lebegőpontosak, illetve az összeadás és a kivonás sokkal gyorsabb művelet, mint a szorzás és az osztás. Ezt a legegyszerűbben változóink típusának a megválasztásával tudjuk vezérelni. Fixpontos értékek tárolására az `int` a legtermészetesebb típus, mely a -2 147 483 648. . . 2 147 483 647 intervallum ábrázolására alkalmas 32 bites mikroprocesszor esetén. Lebegőpontos számok esetében a `double` típus a leghatékonyabb. Ettől általában csak memóriaszűke miatt térünk el a kisebb pontosságú `float` felé.

Gyors algoritmusok készítésénél mindig figyelembe kell venni, hogy lényeges sebességnövekedést általában csak alapvetően új módszertől várhatunk. Amennyiben megvan a megfelelő módszer, először egy helyes függvény megalkotása a cél. A kódolás során felmerülő problémákat legtöbbször a Debugger használatával tudjuk a leggyorsabban és legkönyebben megoldani. Amennyiben már rendelkezünk egy jól működő kóddal, a 80-20 szabályt kell még szem előtt tartanunk. Ez annyit tesz, hogy egy program idejének 80 százalékát a kód 20 százalékában tölti. Ez a 20 százalék a ciklusmagokat jelenti. Bármilyen optimalizáció ebben a 20 százalékban többszörösen térül meg a futás során. A Profiler használatával pontosan meghatározhatjuk, hogy az adott függvény a futási időt az egyes részeknél hány százalékban használta. Ezáltal segít annak a pontos követésében, hogy egy átírással valóban gyorsítottunk-e az algoritmuson, és ha igen, akkor mennyit.

5. Homogén koordináták használata

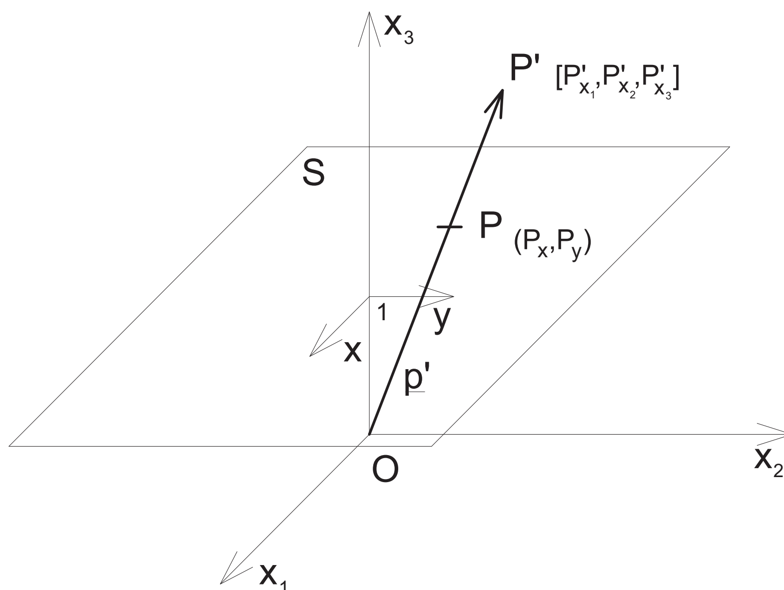
A grafikai algoritmusok elkészítése során a fő problémáink a következők lesznek: meghatározni egy sík két közönséges egyenesének metszéspontját illetve eldönteni azt, hogy egy adott közönséges pont egy adott közönséges egyenes mely oldalán helyezkedik el.

A sík kezeléséhez a közönséges térelemek körét végtelen távolinak is mondott ideális térelemekkel bővítjük ki. Ehhez olyan új koordinátákat vezetünk be, melyekkel ezeket az ideális térelemeket is kezelni tudjuk. Az euklideszi sík két egyenese vagy rendelkezik egy közös (metszés)ponttal vagy párhuzamos. A geometriában található számos tételpár, mely tulajdonképpen ugyanazt mondja ki az egyikben metsző, a másikban párhuzamos egyenesekre. Az egyenes pontjainak összességét egy *ideális pont*nak (végtelen távoli pont) mondott elemmel bővítjük, így oldva fel ezt a kétféleséget. Két egyenes pontjainak az összességét akkor és csak akkor bővítjük ugyanazzal az ideális elemmel, ha a két egyenes párhuzamos. Most már elmondhatjuk, hogy a sík bármely

két egyenesének egyetlen közös pontja van, amit metszéspontnak hívunk akkor is, ha az ideális pont.

A vizsgált S síkban vegyünk fel egy x, y koordináta-rendszert, majd válasszunk egy olyan térbeli derékszögű x_1, x_2, x_3 koordináta-rendszert, ahol az x_1, x_2 tengelyek párhuzamosak az x, y tengelyekkel, x_3 tengelyének $x_3 = 1$ pontja pedig az x, y koordinátarendszer kezdőpontja.

Tekintsük az S sík $P(P_x, P_y)$ pontját. A térbeli koordináta-rendszer O kezdőpontja és a P pont meghatároz egy egyenest. Minden O ponton áthaladó egyenes egyértelműen meghatározza az S sík egy pontját. Az OP egyenest egyértelműen meghatározhatjuk egy O ponttól különböző közönséges P' pontjának megadásával. A P pontot így meghatározó $P'[P'_{x_1}, P'_{x_2}, P'_{x_3}]$ pont koordinátáit a P pont *homogén koordinátái*nak mondjuk. P pontot megadhatjuk az $\underline{OP'} = \underline{p}'[P'_{x_1}, P'_{x_2}, P'_{x_3}]$ vektorral is. A félreértések elkerülése végett a homogén koordinátákat szögletes zárójelbe tesszük $P[P_{x_1}, P_{x_2}, P_{x_3}]$.



1. ábra. A P pont homogén koordinátás megadása

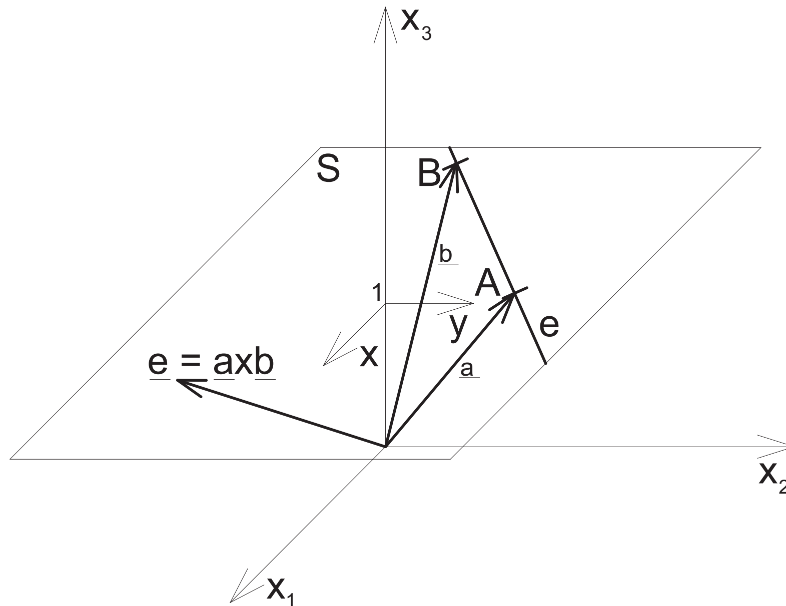
Tétel: *Ha egy pont homogén koordinátáit ugyanazzal a 0-tól különböző számmal szorozzuk meg, akkor ugyanannak a pontnak homogén koordinátáihoz jutunk, s így a pont minden homogén koordinátahármasához eljuthatunk.*

Tétel: *A közönséges $P(P_x, P_y)$ pont $P[P_{x_1}, P_{x_2}, P_{x_3}]$ homogén koordinátáira*

$$P_{x_1} : P_{x_2} : P_{x_3} = P_x : P_y : 1.$$

A közönséges pontok harmadik homogén koordinátája nem 0, és a homogén koordinátákból a közönséges koordinátákat a következőképpen számítjuk:

$$P_x = \frac{P_{x_1}}{P_{x_3}} \quad \text{és} \quad P_y = \frac{P_{x_2}}{P_{x_3}}.$$



2. ábra. Az e egyenesnek megfelelő $\underline{e}$ vektor

Az S sík e egyenesét jellemezhetjük a térbeli koordináta-rendszer O kezdőpontján és az e egyenesen átfektetett sík megadásával. Ezt a síkot legegyszerűbben pedig az $\underline{e} \neq \underline{0}$ normálvektorával adhatjuk meg. A következőkben az egyenest meghatározó vektorok esetén ilyen normálvektorokra kell gondolnunk.

Egy e egyenes vektorát a legegyszerűbben talán úgy számíthatjuk ki, hogy tekintjük az egyenes két pontját ($A[A_x, A_y, 1]$ és $B[B_x, B_y, 1]$), majd ezen pontokba mutató helyvektorok vektoriális szorzataként határozzuk meg azt.

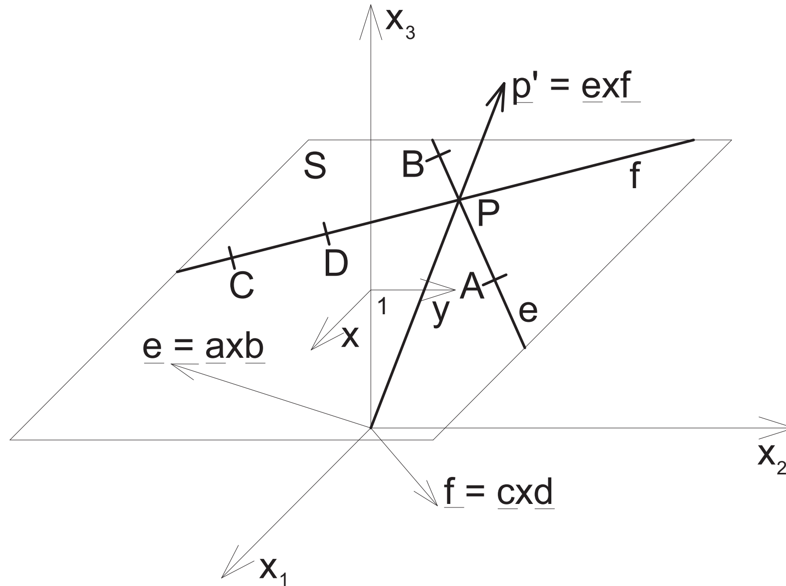
$$\underline{e} = \underline{a} \times \underline{b}$$

Mivel az $\underline{a}$ és $\underline{b}$ vektorok az S sík e egyenesének és a térbeli koordináta-rendszer O kezdőpontja által meghatározott síkjában vannak, a vektoriális szorzatuk pontosan merőleges lesz arra.

Tétel: *A $\underline{p}$ vektor által meghatározott pont és az $\underline{e}$ vektor által meghatározott egyenes akkor és csakis akkor illeszkedik egymáshoz, ha*

$$\underline{p} \cdot \underline{e} = 0.$$

Ez tulajdonképpen a két vektor merőlegességét jelenti. Az $\underline{e}$ vektorra merőleges $\underline{p}$ vektorok pedig pont az e egyenes homogén koordinátás pontjaiba mutatnak.



3. ábra. Két egyenes metszéspontjának meghatározása

Két egyenes metszéspontját a következőképpen határozhatjuk meg: az előző A, B pontok és e egyenes mellett tekintsünk még két pontot, C és D pontokat és a rájuk illeszkedő f egyenest. e-nek és f-nek a metszéspontja P lesz. Ezután felhasználjuk a skalárszorzat alaptulajdonságait. Mivel P illeszkedik e-re, ezért $\underline{p} \cdot \underline{e} = 0$. Azonban P illeszkedik f-re is, ezért $\underline{p} \cdot \underline{f} = 0$, ahol $\underline{f}$ az $\overline{OCD}$ sík normálvektora. Az előző két egyenlet azt jelenti, hogy a P-be mutató $\underline{p}$ vektor merőleges $\underline{e}$ illetve $\underline{f}$ vektorokra. Ilyen irányú $\underline{p'}$ vektort a legegyszerűbben az $\underline{e}$ és $\underline{f}$ vektorok vektoriális szorzataként hozhatunk létre: $\underline{p'} = \underline{e} \times \underline{f}$. Párhuzamos egyenesek metszéspontja esetén a harmadik koordinátában 0-át kapunk eredményül. Ilyenkor az x_1 és x_2 koordináták megmutatják a végtelen távoli $P[p'_{x_1}, p'_{x_2}, 0]$ metszéspont irányát.

Amennyiben a x_3 koordináta nem 0, akkor a $P(\frac{p'_{x1}}{p'_{x_3}}, \frac{p'_{x2}}{p'_{x_3}})$ pont közönséges koordinátáit úgy kaphatjuk meg, hogy a megismert módon az egyes koordinátákat elosztjuk a p'_{x_3} értékkel.

Annak eldöntésére, hogy egy S sík közönséges Q pontja a sík egy közönséges g egyenesének melyik oldalán helyezkedik el, a skalárszorzatot tudjuk felhasználni. Amennyiben a $\underline{q} \cdot \underline{g}$ skalárszorzat pozitív értéket szolgáltat, Q pont az S sík pozitív félsíkjában helyezkedik el g-re nézve; amennyiben negatív, Q pont az S sík negatív félsíkjában helyezkedik el g-re nézve; nulla érték esetén Q pont illeszkedik a g egyenesre.

Két vektor skaláris szorzatát a következőképpen számolhatjuk ki:

$$\underline{a} \cdot \underline{b} = a_{x_1} \cdot b_{x_1} + a_{x_2} \cdot b_{x_2} + a_{x_3} \cdot b_{x_3}$$

A `CPoint2D` és a `CPoint3D` osztályok `x, y, z`, illetve `x, y, z, w` tagokkal rendelkeznek.

```
double CPoint2D::Scalar(const CPoint2D &a) {
    return (x * a.x + y * a.y + z * a.z);
}
```

Az $\underline{e} = \underline{a} \times \underline{b}$ vektoriális szorzathoz az alábbi mátrix e_{x_1}, e_{x_2}, e_{x_3}-hez tartozó adjungált aldeterminánsait kell meghatároznunk:

$$\begin{bmatrix} e_{x_1} & e_{x_2} & e_{x_3} \\ a_{x_1} & a_{x_2} & a_{x_3} \\ b_{x_1} & b_{x_2} & b_{x_3} \end{bmatrix}$$

Így a következő eredményt kapjuk $\underline{e}$-re:

$$e_{x_1} = a_{x_2} \cdot b_{x_3} - a_{x_3} \cdot b_{x_2}$$
$$e_{x_2} = a_{x_3} \cdot b_{x_1} - a_{x_1} \cdot b_{x_3}$$
$$e_{x_3} = a_{x_1} \cdot b_{x_2} - a_{x_2} \cdot b_{x_1}$$

```
CPoint2D CPoint2D::Vectorial(const CPoint2D &a) {
    return CPoint2D(y * a.z - z * a.y,
                    z * a.x - x * a.z,
                    x * a.y - y * a.x);
}
```

Térbeli esetben a síkbelihez hasonlóan járunk el. A különbség annyi, hogy az x_1, x_2 és x_3 koordináták mellé felveszünk egy negyedik x_4 koordinátát. Egy közönséges térbeli $P(P_x, P_y, P_z)$ pont egy homogénkoordinátás felírása pedig a $P[P_x, P_y, P_z, 1]$ lesz.

II. fejezet
Inkrementális algoritmusok

Az inkrementális algoritmusok során a végrehajtandó feladatot egy ciklus felépítésével óhajtjuk megoldani, amit a probléma természetétől függően valamelyik koordináta komponensre építünk fel. A választott komponens (x, ill. y koordináta) minden egyes iterációban növekedni fog eggyel, míg a másik koordináta az algoritmus függvényében nő illetve csökken esetenként eggyel.

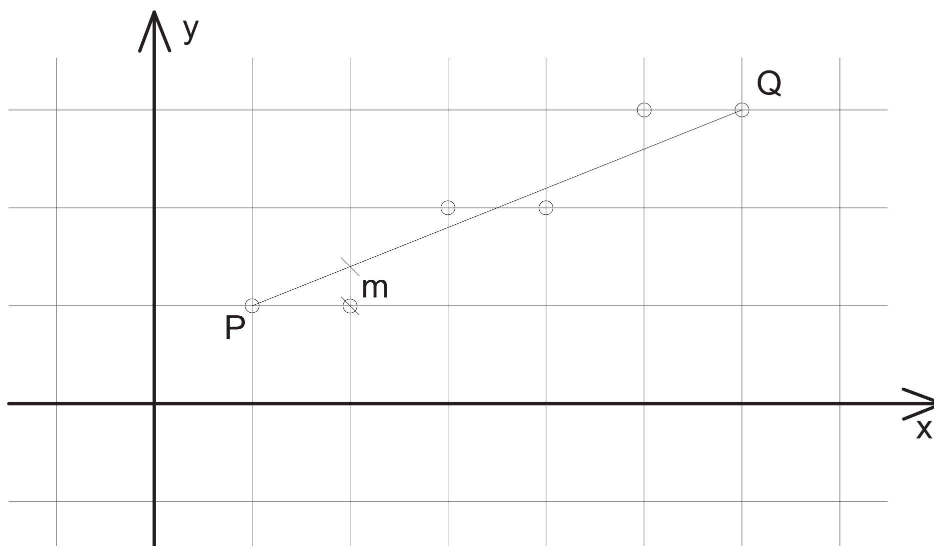
1. Szakasz rajzolása

A problémaosztály ebben az esetben a tetszőleges P és Q végpontú, illetve `color` színű szakasz rajzolása. Ez azt jelenti, hogy ezeket a paramétereket kell átvennünk a formális paraméterlistán. A bolondbiztosság során pedig a végpontok azonosságát kell vizsgálnunk. Amennyiben a végpontok megegyeznek, nem írnak le szakaszt. A mi feladatunk az, hogy eldöntsük, mi a teendő ebben az esetben. A közölt kódban a végpontok által megjelölt pixelt `color` színnel megjelenítjük a képernyőn. A rajzolás során a következő tulajdonságokat fogjuk megkövetelni.

- **A szakasz legyen szimmetrikus:** ez az esztétikai megfontolások mellett azért is jó, mivel főleg régebbi, vonalas rajzra épülő programoknál a képernyőtörlést a megrajzolt szakaszok háttérszínnel való kirakásával helyettesítették. Ebben az esetben, ha nem szimmetrikus a szakasz és az ellenkező irányból rajzoljuk meg, felesleges pixelek maradhatnak a képernyőn. (Az akkori technikával ilyen módon nagyságrendileg gyorsabb programokat lehetett írni. Tulajdonképpen a hardver fejlődésével a drótvázrajz is veszített jelentőségéből. Helyét a különböző felületmodellek veszik át manapság.) A szimmetria biztosítása érdekében a szakaszt a végpontjaiból kiindulva fogjuk megrajzolni a közepe felé haladva, ami tulajdonképpen gyorsítani is fogja az algoritmusunkat, mivel minden egyes kiszámolt pixelt kétszer tudunk megrajzolni.
- **A szakasz ne tartalmazzon szakadásokat illetve „csomókat":** ez a gyakorlatban azt jelenti, hogy ha pl. a szakaszunk meredeksége kisebb, mint 45°, akkor a rajzolást az x koordinátára kell felépítenünk. Ebben az esetben el szeretnénk kerülni, hogy a rajzolás során pixelek egymás alá kerüljenek. Mivel az inkrementális algoritmusok alaptulajdonsága miatt minden egyes iterációban egyetlen újabb pixelt gyújtunk ki, ez a kritérium is teljesülni fog.

A koordináta komponenst, melyre a ciklust felépítjük, a szakasz meredeksége alapján választjuk ki. Ennek az egyik legegyszerűbb módszere az, ha megvizsgáljuk a végpontok koordináta komponenseinek különbségének az abszolútértékét (`fabs(q.x - p.x) > fabs(q.y - p.y)`).

Ezekután a `p` és `q` végpontokat kell rendeznünk a kiválasztott koordináta szerint. Így már nem okoz problémát, hogy az egy pixelre jutó változást meghatározzuk, amit az `m` változóval jelölünk a kódban. A C++ konverziós eljárása miatt a részeredményeket tároló `mt1` és `mt2` segédváltozókat 0.5-ről indítjuk el.



1. ábra. Szakasz rajzolása a raszteres képernyőn

A kiválasztott koordinátára felépített ciklusban pedig már csak a szakasz két felének egy-egy pixelét kell kiraknunk.

Függvényünk visszatérési értékét felhasználhatjuk annak jelzésére, hogy a végpontok alapján tényleg szakaszt rajzoltunk-e (`return 0;`) vagy csak egy pixelt tettünk ki (`return -1;`).

```
int CGraphPrimitives::Line(
        eBuffer buffer, CPoint2Dint p, CPoint2Dint q, COLORREF color)
{
    CDC         *dc = (buffer == Front ? m_cDC : &m_cBackDC);
    CPoint2Dint t;
    double      m, mt1 = 0.5, mt2 = 0.5;
    int         limit;

    if (p == q) {
        dc->SetPixel(p.x, p.y, color);
        return -1;
    }

    if (abs(q.x - p.x) > abs(q.y - p.y)) {
        if (q.x < p.x)
            t = p, p = q, q = t;
```

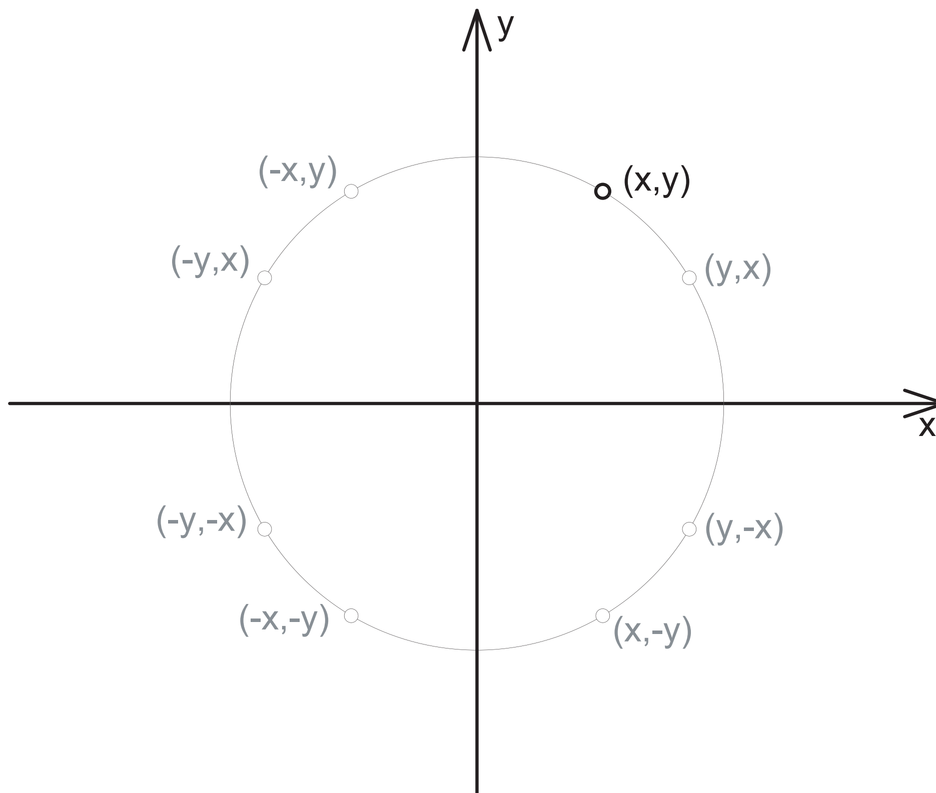
```
        m = (double)(q.y - p.y) / (double)(q.x - p.x);

        limit = (p.x + q.x) >> 1;
        for (int i = p.x, j = q.x; i <= limit; i++, j--) {
            dc->SetPixel(i, p.y + mt1, color);
            dc->SetPixel(j, q.y + mt2, color);
            mt1 += m;
            mt2 -= m;
        }
    } else {
        if (q.y < p.y)
            t = p, p = q, q = t;

        m = (double)(q.x - p.x) / (double)(q.y - p.y);

        limit = (p.y + q.y) >> 1;
        for (int i = p.y, j = q.y; i <= limit; i++, j--) {
            dc->SetPixel(p.x + mt1, i, color);
            dc->SetPixel(q.x + mt2, j, color);
            mt1 += m;
            mt2 -= m;
        }
    }

    return 0;
}
```

2. Kör rajzolása

Kör rajzolásakor a pixelezett képernyő miatt négy szimmetriatengelyt tudunk kihasználni az algoritmus írása során. (A függőleges, a vízszintes, illetve a 45° és a 135° fokos átlós szimmetriatengelyeket.) Ez azt jelenti, hogy elegendő mindössze a kör egy nyolcadához tartozó pixeleket kiszámolni és a többi részt tükrözések segítségével tudjuk előállítani. Az algoritmus során az r sugarú, origó középpontú kör É-ÉK-i nyolcadát fogjuk meghatározni az északi $(0, r)$ pontból kiindulva. A kérdés az, hogy az x koordinátát meddig kell futtatni. Kézenfekvőnek tűnik az $x_{max} = sin(45^o) \cdot r = \frac{\sqrt{2}}{2} \cdot r$ válasz. Nem túl szerencsés azonban egy egész értéket egy ciklusfejben rendszeresen egy lebegőpontos értékkel összehasonlítani, jobb lenne csak egész számokat alkalmazni. Az $y > x$ vizsgálat az eredeti elgondolással megegyező eredményt fog szolgáltatni. Az (x, y) pont mindig a soron következő pixelt jelöli, így amire az $y > x$ egyenlőtlenség hamissá válik, az É-ÉK-i nyolcadot teljes terjedelmében megrajzoltuk.

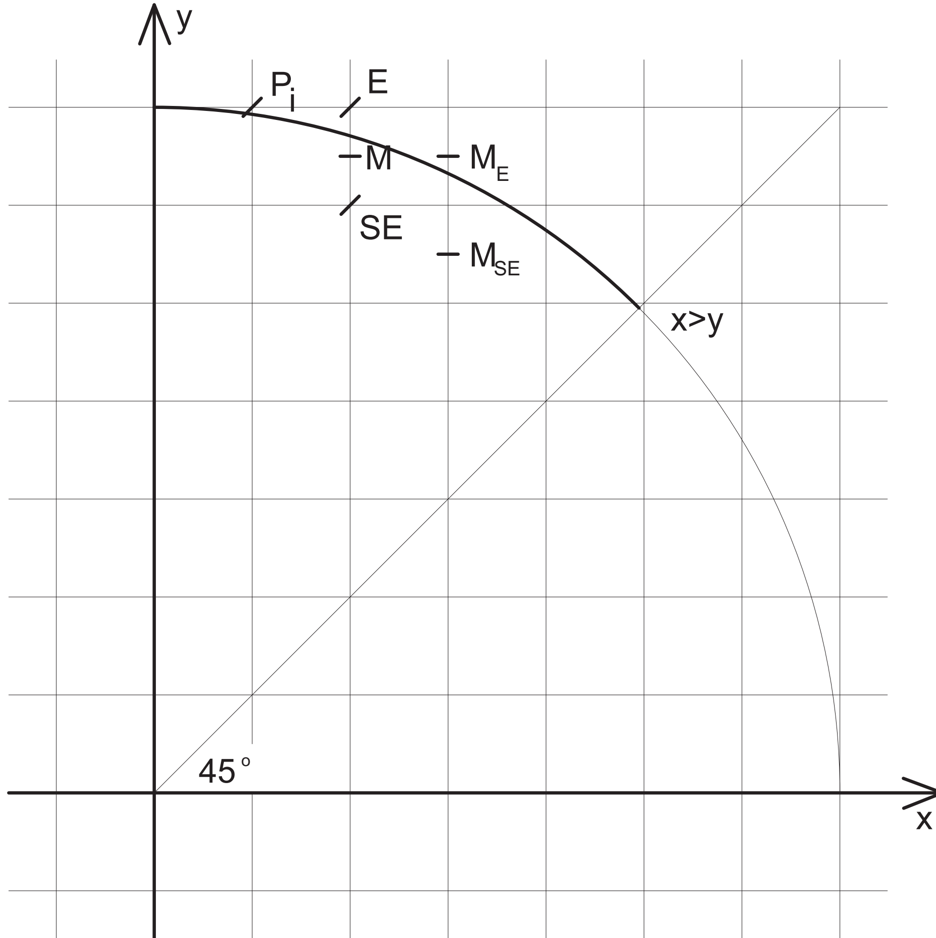
Az origó középpontú kör pozícióba tolásában és a nyolcadok létrehozásában a `CirclePoints` függvény lesz segítségünkre. A középpont és az aktuális pont koordinátái `int` típusúak.

```
void CGraphPrimitives::CirclePoints(
        eBuffer buffer, const CPoint2Dint &center,
        const CPoint2Dint &p, COLORREF color)
{
    CDC     *dc = (buffer == Front ? m_cDC : &m_cBackDC);

    dc->SetPixel(center.x + p.x, center.y + p.y, color);
    dc->SetPixel(center.x + p.x, center.y - p.y, color);
    dc->SetPixel(center.x - p.x, center.y + p.y, color);
    dc->SetPixel(center.x - p.x, center.y - p.y, color);
    dc->SetPixel(center.x + p.y, center.y + p.x, color);
    dc->SetPixel(center.x + p.y, center.y - p.x, color);
    dc->SetPixel(center.x - p.y, center.y + p.x, color);
    dc->SetPixel(center.x - p.y, center.y - p.x, color);
}
```



2. ábra. A kiválasztott nyolcad tükrözése

A körív meghatározásában egy $F(x, y) = x^2 + y^2 - r^2$ függvényt használunk majd, mely az ideális körívtől való távolságot szolgáltatja. Tehát az $F(x, y) < 0$ azt jelenti, hogy az (x, y) pont a körön belül helyezkedik el, $F(x, y) > 0$ azt, hogy a körön kívül, illetve 0 pontosan akkor lesz, ha az (x, y) pont illeszkedik a körre.



3. ábra. Az É-ÉK-i nyolcad kiszámítása

Tételezzük fel, hogy a körív kiszámítása során eljutottunk egy P_i pontba. A következő pixelt két pont — $E(P_{ix}+1, P_{iy})$ [East] és $SE(P_{ix}+1, P_{iy}-1)$ [South-East] — közül kell kiválasztanunk. Ezt az E és SE közötti $M(P_{ix}+1, P_{iy}-\frac{1}{2})$ felezőpont vizsgálatával tehetjük meg. Amennyiben $F(M) < 0$, tudjuk, hogy a körív az E és az M pontok között halad, ekkor az E ponttal haladunk tovább. Ellenkező esetben a körív az SE és M pontok között (egyenlőség esetén pontosan M-en) halad át, és ebben az esetben SE pontot választjuk. Íly módon meg is határozhatjuk a kiválasztott nyolcad pontjait. Azonban az $F(M)$ meghatározásához legalább két szorzást kell végrehajtanunk a további gyors műveletek mellett, mivel az $r \cdot r$ értéket segédváltozóban tárolhatjuk.

A továbbfejlesztés ötlete az, hogy meghatározzuk az egymás utáni $F(M)$ értékek különbségét, és ezeket felhasználva kevesebbet kell számolnunk egy új $F(M)$ érték kiszámításához. Jelöljük az $F(M)$ értékét d_{old}-al.

$$d_{old} = F(M) = F(P_{ix}+1, P_{iy}-\frac{1}{2}) = (P_{ix}+1)^2 + (P_{iy}-\frac{1}{2})^2 - r^2$$

- Ha $d_{old} < 0$, akkor az E pontot választjuk és az új felezőponthoz (M_E) tartozó $F(M_E)$ értéket, melyet d_{new}-al jelölünk, a következőképpen számolhatjuk ki:

$$d_{new} = F(M_E) = F(P_{ix} + 2, P_{iy} - \frac{1}{2}) = (P_{ix} + 2)^2 + (P_{iy} - \frac{1}{2})^2 - r^2$$

A változást (Δ_E) pedig az alábbi átalakítások után kapjuk:

$$\begin{aligned}\Delta_E = d_{new} - d_{old} &= (P_{ix} + 2)^2 + (P_{iy} - \frac{1}{2})^2 - r^2 \\ &-((P_{ix} + 1)^2 + (P_{iy} - \frac{1}{2})^2 - r^2) = (P_{ix} + 2)^2 - (P_{ix} + 1)^2 = \\ &= P_{ix}^2 + 4P_{ix} + 4 - P_{ix}^2 - 2P_{ix} - 1 = 2P_{ix} + 3\end{aligned}$$

- Ha $d_{old} \geq 0$, akkor az SE pontot választjuk és az új felezőponthoz (M_{SE}) tartozó $F(M_{SE})$ értéket, melyet d_{new}-al jelölünk, a következőképpen számolhatjuk ki:

$$d_{new} = F(M_{SE}) = F(P_{ix} + 2, P_{iy} - \frac{3}{2}) = (P_{ix} + 2)^2 + (P_{iy} - \frac{3}{2})^2 - r^2$$

A változást (Δ_{SE}) pedig az alábbi egyenlőség alapján kapjuk:

$$\begin{aligned}\Delta_{SE} = d_{new} - d_{old} &= (P_{ix} + 2)^2 + (P_{iy} - \frac{3}{2})^2 - r^2 \\ &-((P_{ix} + 1)^2 + (P_{iy} - \frac{1}{2})^2 - r^2) = P_{ix}^2 + 4P_{ix} + 4 + P_{iy}^2 - 3P_{iy} + \frac{9}{4} \\ &-P_{ix}^2 - 2P_{ix} - 1 - P_{iy}^2 + P_{iy} - \frac{1}{4} = 2(P_{ix} - P_{iy}) + 5\end{aligned}$$

Látható, hogy a szorzások számát mindkét esetben egy műveletre tudtuk csökkenteni, ráadásul fixpontos aritmetika mellett ezek a kettővel való szorzások bitléptetéssel helyettesíthetőek, ami lényegesen gyorsabb. Egy feladatunk maradt viszont, méghozzá a d változó kezdőértékének a meghatározása. Mivel az algoritmus a $(0, r)$ pontból indul, az első középpont $(1, r - \frac{1}{2})$-ben van. Így:

$$d_{start} = F(1, r - \frac{1}{2}) = 1^2 + (r - \frac{1}{2})^2 - r^2 = 1 + r^2 - r + \frac{1}{4} - r^2 = \frac{5}{4} - r$$

Amennyiben csak fixpontos változókkal dolgozunk, a d_{start} értékét a $d_{start} = 1 - r$-el közelíthetjük. Az így generált görbe nem fog lényegesen eltérni az ideális körívtől.

```
int CGraphPrimitives::Circle(
        eBuffer buffer, CPoint2Dint center,
        double r, COLORREF color)
{
    CPoint2Dint     p(0, r);
    int             d = 5.0 / 4.0 - r;

    if (r <= 0.0) return -1;

    CirclePoints(buffer, center, p, color);
```

```
    while (p.y > p.x) {
        if (d < 0) {
            d += (p.x << 1) + 3;
            p.x++;
        } else {
            d += ((p.x - p.y) << 1) + 5;
            p.x++;
            p.y--;
        }

        CirclePoints(buffer, center, p, color);
    }

    return 0;
}
```

III. fejezet
Vágó algoritmusok

A vágó algoritmusok segítségével megrajzolandó szakaszok egy meghatározott ablakban látható részeit fogjuk meghatározni vágások sorozatán keresztül. Ez az ablak egyszerűbb esetben egy téglalap (Cohen-Sutherland algoritmus), bonyolultabb esetben pedig egy konvex vagy egy konkáv alakzat lesz. A konkáv alakzat lyukakat és szigeteket is tartalmazhat, akár tetszőleges mélységben egymásba ágyazva is.

A vágó algoritmusok mellett ebben a fejezetben tárgyaljuk a tartalmazást vizsgáló algoritmusokat is.

1. Cohen-Sutherland algoritmus

A kezelendő síkot kilenc részre osztjuk fel a téglalap alakú képernyőnk négy oldalegyenesének segítségével. A kilenc síkrészhez négybites kódokat rendelünk az oldalegyenesek alapján.

- Az első bit egyes lesz, ha a vizsgált végpont a felső vízszintes oldalegyenes felett helyezkedik el, egyébként 0.
- A második bit egyes lesz, ha a vizsgált végpont az alsó vízszintes oldalegyenes alatt helyezkedik el, egyébként 0.
- A harmadik bit egyes lesz, ha a vizsgált végpont a jobb oldali függőleges oldalegyenestől jobbra helyezkedik el, egyébként 0.
- A negyedik bit egyes lesz, ha a vizsgált végpont a bal oldali függőleges oldalegyenestől balra helyezkedik el, egyébként 0.

```
int CGraphPrimitives::CSCode(
        const CRect &window, const CPoint2D &p)
{
    int code = 0;

    if (p.x < window.left) code |= 1;
    else if (p.x > window.right) code |= 2;

    if (p.y < window.bottom) code |= 4;
    else if (p.y > window.top) code |= 8;

    return code;
}
```

1	1	1
0	0	0
0	0	0

10	10	10
00	00	00
01	01	01

100	100	101
000	000	001
010	010	011

1001	1000	1010
0001	0000	0010
0101	0100	0110

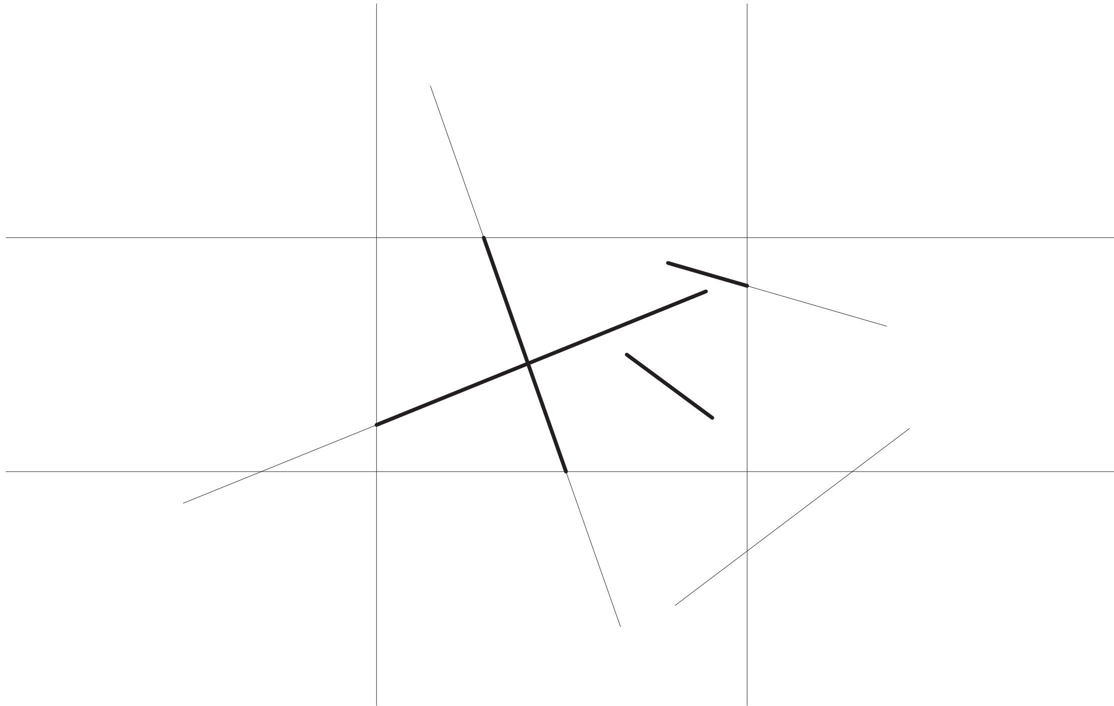
1. ábra. A bitkódok hozzárendelése

Az algoritmus során a végpontokhoz először is hozzárendeljük a négybites kódokat. A végpontok egyezőségét itt nem vizsgáljuk, ugyanis az algoritmus felépítéséből fakadóan nullával való osztáshoz nem vezethet, a szakaszrajzoló függvény pedig fel van készítve az ilyen esetekre.

Amennyiben a kódokban azonos helyen egyes jelenik meg, azt jelenti, hogy mindkét végpont ugyanazon oldalegyenesnek a külső részén helyezkedik el, tehát nem tartalmaz látható részt. Ebben az esetben az algoritmust befejezhetjük.

Ellenkező esetben meg kell néznünk, hogy a végpontok csupa nulla kóddal rendelkeznek-e. Amennyiben igen, a szakasz a végpontjai segítségével megrajzolható és az algoritmus befejezhető. Azonban ha az 1 értékű kódok nem egyező pozíciókban vannak, akkor vágásra van szükség. Vágás után a mozgatott végponthoz minden esetben újra meghatározzuk a kódot, ugyanis nem jósolható meg előre, hogy mi fog történni vele. Nem csak 1-es bit tűnhet el, hanem például új helyen be is jöhet egy másik. Vágás során a párhuzamos szelők tételét alkalmazhatjuk. Mivel az új koordinátapár egyik tagja ismert, az arányosságból a másik tag kifejezhető ismert mennyiségek segítségével.

A vágás után az új kódok segítségével a vizsgálatokat megismételjük. Ily módon vágások sorozatán keresztül meg tudjuk határozni egy szakasz látható részét egy téglalap alakú képernyőn, illetve egyértelműen meg tudjuk mondani, ha nem rendelkezik ilyen résszel.



2. ábra. A vágások eredménye

```
int CGraphPrimitives::CSLine(
        eBuffer buffer, const CRect &window, CPoint2D p,
        CPoint2D q, COLORREF color_in, COLORREF color_out)
{
    int         c1, c2, t;
    CPoint2D    r, p_old;

    c1 = CSCode(window, p);
    c2 = CSCode(window, q);

    while ((c1 & c2) == 0) {

        if (c1 == c2) {
            Line(buffer, p, q, color_in);
            return 0;
        }

        if (c1 == 0) {
            t = c1, c1 = c2, c2 = t;
            r =  p,  p =  q,  q = r;
        }
```

```
        p_old = p;
        if (c1 & 1) {
            p.y += (q.y - p.y) * (  window.left - p.x) / (q.x - p.x);
            p.x = window.left;
        } else if (c1 & 2) {
            p.y += (q.y - p.y) * ( window.right - p.x) / (q.x - p.x);
            p.x = window.right;
        } else if (c1 & 4) {
            p.x += (q.x - p.x) * (window.bottom - p.y) / (q.y - p.y);
            p.y = window.bottom;
        } else if (c1 & 8) {
            p.x += (q.x - p.x) * (   window.top - p.y) / (q.y - p.y);
            p.y = window.top;
        }
        Line(buffer, p, p_old, color_out);

        c1 = CSCode(window, p);
    }

    Line(buffer, p, q, color_out);
    return -1;
}
```

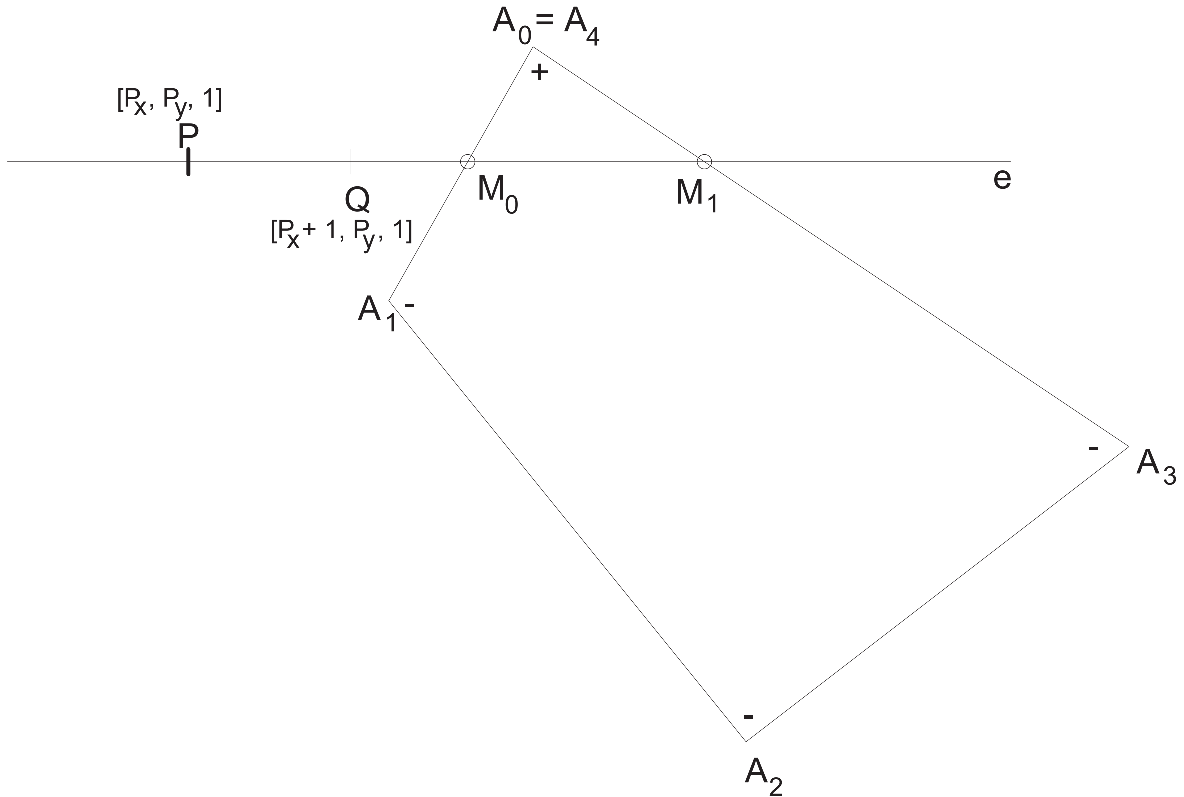
2. Alakzatra való lehatárolás

A különböző alakzatokra való lehatárolás során gyakran kell majd egyenesek metszéspontját kezelni, ami az $y = m \cdot x + b$ képlet használatával nehézségekhez vezet. Sokkal egyszerűbb lesz a dolgunk, ha bevezetjük a homogén koordinátákat.

2.1. Kinn-benn algoritmus konvex sokszögekre

Konvex sokszög esetén a vizsgálatot a legegyszerűbben úgy végezhetjük el, ha a vizsgált $P[P_x, P_y, 1]$ ponton keresztül tekintünk egy e egyenest. (Ennek egyik egyszerű előállítási módja, ha a P pont x illetve y koordinátáinak segítségével létrehozunk egy $Q[P_x+1, P_y, 1]$ pontot, majd ennek a pontnak és a P pontnak a vektoriális szorzataként az e egyenest fogjuk kapni. Érdemes az előbb ismertetett módon csak az x vagy csak az y koordinátában eltérést eszközölni, ugyanis így vízszintes illetve függőleges állású e egyeneshez jutunk. Később pontokat kell majd rendeznünk ezen egyenes mentén és ezek a speciális állású egyenesek optimalizálásra adnak majd lehetőséget.)

Ezek után szükségünk van az egyenes és az alakzat oldalszakaszainak a metszéspontjaira. A kérdés az, hogy az alakzat csúcspontjai az egyenes mely oldalán helyezkednek el. Ugyanis ahol két egymást követő csúcspont az egyenes két oldalán helyezkedik el, ott nem csak a csúcsokra illeszthető oldalegyenesnek, hanem az oldalszakasznak is



3. ábra. Kinn-benn vizsgálat konvex sokszögre

metszéspontja van az e egyenessel. Ez azt jelenti, hogy vennünk kell a csúcspontok és az e egyenest leíró számhármas skaláris szorzatát.

Most vizsgáljuk meg, hogy az értékek előjelében hol van váltás. Amennyiben nem szeretnénk külön esetként kezelni, amikor az első és az utolsó csúcsponthoz tartozó értéket vizsgáljuk, használhatjuk a régi trükköt, hogy a legelső csúcspontot redundánsan felvesszük legutolsóként is. Egy kicsivel több adatot kell ily módon tárolnunk, azonban egyetlen ciklus segítségével tudjuk most már kezelni a problémát. A 0 értékű skalárszorzatokat vagy a pozitív, vagy a negatív értékekhez kell sorolnunk. Így ha egy csúcsponton érintőleg átmegy az e egyenes, akkor vagy kettő vagy egy metszéspontot sem kapunk az algoritmus során. Az előbbiek miatt egyéb esetekben, amennyiben a P pont az alakzatot tartalmazó vízszintes sávban helyezkedik el, mindig kettő metszéspontot kapunk, amennyiben ezen a sávon kívül helyezkedik el, akkor pedig egy metszéspontot sem kapunk. Tehát ha nem kaptunk metszéspontokat, máris megállhatunk az algoritmussal és kijelenthetjük, hogy a P pont a konvex alakzaton kívül helyezkedik el.

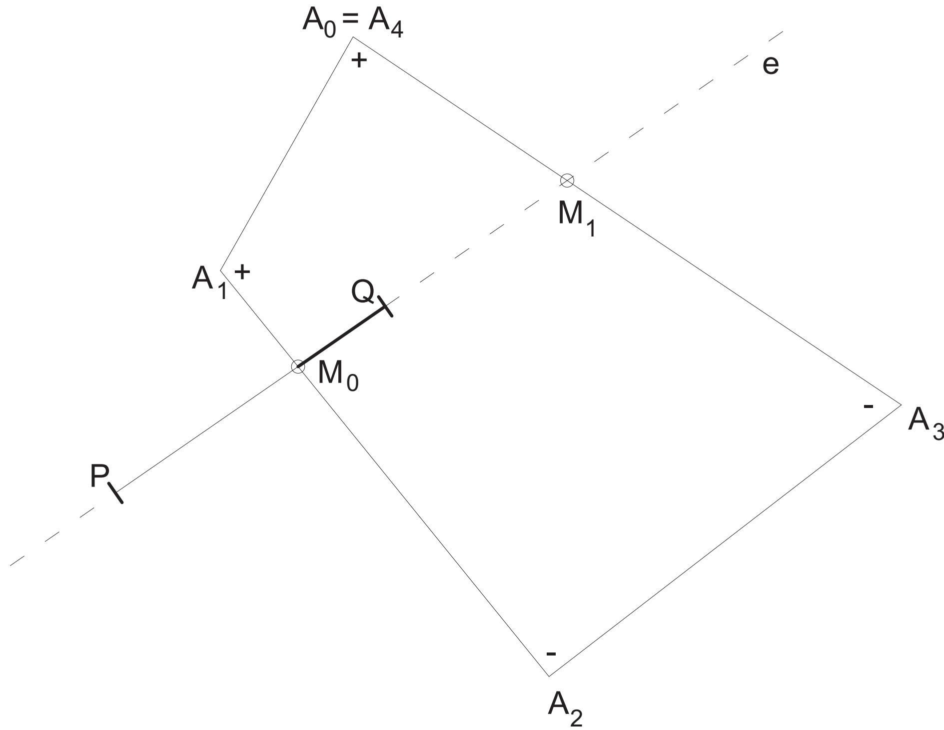
Abban az esetben, ha két metszéspontot kaptunk eredményül, meg kell vizsgálni, hogy a vizsgált pont a metszéspontok között helyezkedik-e el. (Ez a vízszintes egyenes miatt mindössze az x koordináták vizsgálatát jelenti.) Ha igen, azt mondhatjuk,

hogy a P pont az adott konvex alakzaton belül van, amennyiben nem, akkor pedig megállapíthatjuk, hogy azon kívül helyezkedik el.

```
void CGraphPrimitives::ConvexInOut(
        eBuffer buffer, CWindowBase &window, CPoint2D &p,
        COLORREF color_in, COLORREF color_out)
{
    CDC         *dc = (buffer == Front ? m_cDC : &m_cBackDC);
    int         i, position[2], *signum, posNum = 0;
    CPoint2D    q, e, f, t, m[2];

    signum = new int[window.m_iVertNum + 1];
    q = CPoint2D(p.x + 1.0, p.y, 1.0);
    e = p * q;

    for (i = 0; i <= window.m_iVertNum; i++)
        signum[i] = SGN(e.Scalar(window.m_Vertices[i]));

    for (i = 0; i < window.m_iVertNum; i++)
        if (signum[i] != signum[i + 1]) position[posNum++] = i;

    delete [] signum;

    if (posNum == 0) {
        dc->SetPixel(p.x, p.y, color_out);
        return;
    }

    for (i = 0; i < 2; i++) {
        f = window.m_Vertices[position[i]] *
            window.m_Vertices[position[i] + 1];
        m[i] = e * f;
        m[i].DescartesCoords();
    }
    if (m[1].x < m[0].x)
        t = m[0], m[0] = m[1], m[1] = t;

    if ((m[0].x <= p.x) && (p.x <= m[1].x))
        dc->SetPixel(p.x, p.y, color_in);
    else
        dc->SetPixel(p.x, p.y, color_out);
}
```

2.2. Szakasz vágása konvex sokszögre

Amennyiben szakaszokat szeretnénk konvex alakzatra lehatárolni, hasonlóan kell eljárnunk, mint a kinn-benn vizsgálatkor. A különbség annyi, hogy nem nekünk kell előállítani Q-t, hanem a szakasz másik végpontjaként adott lesz. Ezek után a két végpont segítségével meghatározzuk az e egyenest.



4. ábra. Szakasz vágása konvex sokszögre

Itt merül fel a bolondbiztosság kérdése: ugyanis ha P megegyezik Q-val, akkor nem írnak le egy szakaszt, és a rájuk illesztett e egyenessel is csak problémánk lesz. Ebben az esetben a leghelyesebben akkor járunk el, ha nem csinálunk semmit és azonnal kilépünk az algoritmusból, illetve ha a szakaszt ezentúl pontként kezeljük és P pontra meghívjuk a kinn-benn algoritmust.

Helyes paraméterek esetén megvizsgáljuk az alakzat csúcspontjainak a skalárszorzatát az e egyenessel, majd meghatározzuk a metszéspontokat, ha vannak. Amennyiben nem találunk metszéspontokat, az azt jelenti, hogy a konvex alakzat teljes egészében a $\overline{PQ}$ szakaszra illeszkedő egyenes egyik vagy másik oldalán helyezkedik el. Amennyiben kaptunk metszéspontokat, a kérdés az, hogy mi a viszonyuk a P és Q végpontokhoz képest.

A rendezést az e egyenes mentén kell elvégeznünk. Azonban ha ismerjük az e meredekségét, akkor a vizsgálatot valamelyik (x illetve y) koordináta vizsgálatára redukálhatjuk. A meredekség vizsgálatát legegyszerűbben talán a $|P_x - Q_x|$ és $|P_y - Q_y|$ összehasonlításával tudjuk elvégezni. Ha $|P_x - Q_x|$ a nagyobb, akkor az x koordiáta

vizsgálata elegendő, ellenkező esetben az y koordinátákat kell tekintenünk. Egyenlőség esetén 45 fokos dőlésszögű e egyenesről van szó, tehát a pontosság szempontjából irreleváns, hogy melyik koordinátát fogjuk felhasználni.

Ha a metszéspontokat illetve P-t és Q-t rendezzük, akkor minössze öt esetet tudunk megkülönböztetni.

- $Q \leq M[0]$ vagy $P \geq M[1]$: $\overline{PQ}$ szakasz az alakzaton kívül helyezkedik el teljes egészében, így nincs látható része.
- $P \geq M[0]$ és $Q \leq M[1]$: $\overline{PQ}$ szakasz teljes egészében az alakzaton belül van, a teljes szakaszt megrajzolhatjuk.
- $P \leq M[0]$ és $Q \geq M[1]$: $\overline{PQ}$ szakasz túlnyúlik az alakzaton mindkét irányban, így csak a metszéspontok közötti rész lesz látható.
- $P < M[0]$ és $Q > M[0]$: $\overline{PQ}$ szakasz belelóg az alakzatba, így csak az $M[0]$ és Q közötti rész lesz látható.
- $P < M[1]$ és $Q > M[1]$: $\overline{PQ}$ szakasz belelóg az alakzatba, így csak a P és $M[1]$ közötti rész lesz látható.

```
void CGraphPrimitives::ConvexWLine(
        eBuffer buffer, CWindowBase &window, CPoint2D &p,
        CPoint2D &q, COLORREF color_in, COLORREF color_out)
{
    int         i;
    int         position[2], *signum, posNum = 0;
    CPoint2D    e, f, t, m[2];

    if (p == q) return;

    signum = new int[window.m_iVertNum + 1];
    e = p * q;

    for (i = 0; i <= window.m_iVertNum; i++)
        signum[i] = SGN(e.Scalar(window.m_Vertices[i]));

    for (i = 0; i < window.m_iVertNum; i++)
        if (signum[i] != signum[i + 1]) {
            f = window.m_Vertices[i] *
                window.m_Vertices[i + 1];
            m[posNum] = e * f;
            m[posNum].DescartesCoords();
            posNum++;
        }
```

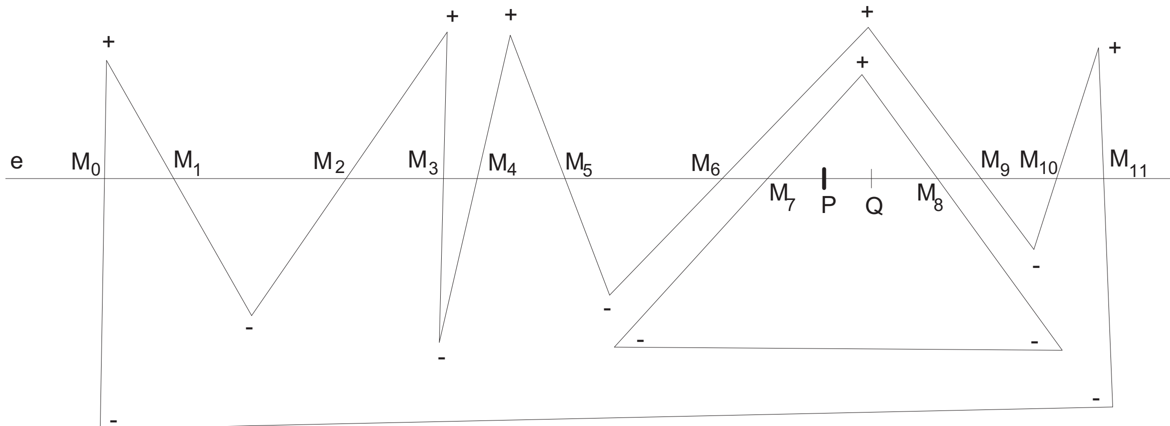
```
delete [] signum;

if (posNum == 0) {
    Line(buffer, p, q, color_out);
    return;
}

if (fabs(p.y - q.y) < fabs(p.x - q.x)) {
    if (m[1].x < m[0].x)
        t = m[0], m[0] = m[1], m[1] = t;

    if (q.x < p.x)
        t = p, p = q, q = t;

    if ((q.x < m[0].x) || (m[1].x < p.x)) {
        Line(buffer, p, q, color_out);
        return;
    }

    if ((m[0].x <= p.x) && (q.x <= m[1].x)) {
        Line(buffer, p, q, color_in);
        return;
    }

    if ((p.x <= m[0].x) && (m[1].x <= q.x)) {
        Line(buffer, m[0], p, color_out);
        Line(buffer, m[1], q, color_out);
        Line(buffer, m[0], m[1], color_in);
        return;
    }

    if ((p.x <= m[0].x) && (m[0].x <= q.x)) {
        Line(buffer, m[0], p, color_out);
        Line(buffer, m[0], q, color_in);
        return;
    }

    if ((p.x <= m[1].x) && (m[1].x <= q.x)) {
        Line(buffer, m[1], q, color_out);
        Line(buffer, m[1], p, color_in);
        return;
    }
} else {
    if (m[1].y < m[0].y)
        t = m[0], m[0] = m[1], m[1] = t;
```

```
        if (q.y < p.y)
            t = p, p = q, q = t;

        if ((q.y < m[0].y) || (m[1].y < p.y)) {
            Line(buffer, p, q, color_out);
            return;
        }

        if ((m[0].y <= p.y) && (q.y <= m[1].y)) {
            Line(buffer, p, q, color_in);
            return;
        }

        if ((p.y <= m[0].y) && (m[1].y <= q.y)) {
            Line(buffer, m[0], p, color_out);
            Line(buffer, m[1], q, color_out);
            Line(buffer, m[0], m[1], color_in);
            return;
        }

        if ((p.y <= m[0].y) && (m[0].y <= q.y)) {
            Line(buffer, m[0], p, color_out);
            Line(buffer, m[0], q, color_in);
            return;
        }

        if ((p.y <= m[1].y) && (m[1].y <= q.y)) {
            Line(buffer, m[1], q, color_out);
            Line(buffer, m[1], p, color_in);
            return;
        }
    }
}
```

2.3. Kinn-benn algoritmus konkáv sokszögekre

A továbbiakban konkáv alakzatnak tekintjük az olyan alakzatokat is, melyek lyukat tartalmaznak, illetve a lyukban szigetet tetszőleges mélységben.

Ennél az algoritmusnál sok dolgot fel tudunk használni a konvex eset tapasztalataiból. Az ott alkalmazottakhoz hasonlóan létrehozunk egy vízszintes e egyenest. Ezek után keressük az oldalszakaszokkal való metszéspontjait. Amennyiben egyet sem kapunk, az azt jelenti, hogy a vizsgált P pont az alakzatot tartalmazó legszűkebb vízszintes sávon kívülre esik, így nem lesz látható. Ha kapunk metszéspontokat, azok mindig



5. ábra. Kinn-benn algoritmus konkáv sokszögre

páros sokan lesznek (lásd: kinn-benn algoritmus konvex alakzatra). Ezeket a metszéspontokat az x vagy y koordináták alapján rendeznünk kell. Mivel a C++ tartalmaz beépített rendező függvényt, nekünk csupán a két elem összehasonlítását végző kódot kell elkészítenünk.

```
int sortx(const void *a, const void *b)
{
    CPoint2D    *p = (CPoint2D *)a, *q = (CPoint2D *)b;

    if (p->x > q->x) return 1;
    if (p->x == q->x) return 0;
//  if (p->x < q->x)
    return -1;
}

int sorty(const void *a, const void *b)
{
    CPoint2D    *p = (CPoint2D *)a, *q = (CPoint2D *)b;

    if (p->y > q->y) return 1;
    if (p->y == q->y) return 0;
//  if (p->y < q->y)
    return -1;
}
```

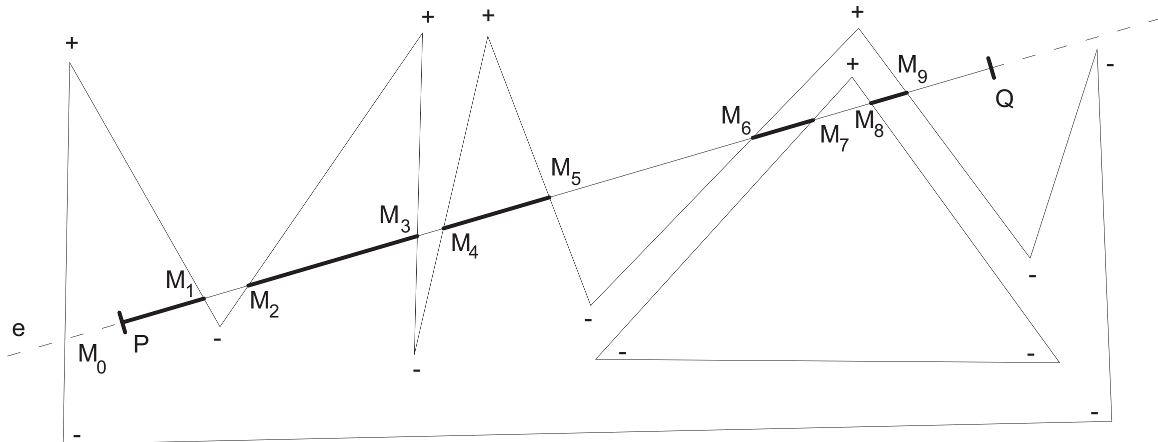
A rendezett metszéspontok között megkeressük P helyét. Amennyiben páros sok metszéspont helyezkedik el P mindkét oldalán, azt mondhatjuk, hogy a vizsgált pont a konkáv alakzaton kívül helyezkedik el. Ha mindkét oldalon páratlan sok metszéspont helyezkedik el, akkor pedig az alakzaton belül van a P pont.

```
void CGraphPrimitives::ConcaveInOut(
        eBuffer buffer, CConcaveWindow &window, CPoint2D &p,
        COLORREF color_in, COLORREF color_out)
{
    CDC         *dc = (buffer == Front ? m_cDC : &m_cBackDC);
    int         i, j, *signum, posNum = 0;
    CPoint2D    q, e, f, *m;

    signum = new int[window.m_iVertNum + 1];
    m = new CPoint2D[window.m_iVertNum + 1];

    q = CPoint2D(p.x + 1.0, p.y, 1.0);
    e = p * q;

    for (j = 0; j < window.m_iWindowNum; j++) {
        for (i = 0; i <= window.m_Windows[j]->m_iVertNum; i++)
            signum[i] = SGN(e.Scalar(window.m_Windows[j]->m_Vertices[i]));

        for (i = 0; i < window.m_Windows[j]->m_iVertNum; i++)
            if (signum[i] != signum[i + 1]) {
                f = window.m_Windows[j]->m_Vertices[i] *
                    window.m_Windows[j]->m_Vertices[i + 1];
                m[posNum] = e * f;
                m[posNum].DescartesCoords();
                posNum++;
            }
    }

    delete [] signum;

    if (posNum == 0) {
        dc->SetPixel(p.x, p.y, color_out);
        delete [] m;
        return;
    }

    qsort((void *)m, posNum, sizeof(CPoint2D), sortx);

    if ((p.x < m[0].x) || (m[posNum - 1].x < p.x)) {
        dc->SetPixel(p.x, p.y, color_out);
        delete [] m;
        return;
    }

    for (i = 0; i < posNum; i += 2) {
```

```
        if ((m[i].x <= p.x) && (p.x <= m[i + 1].x)) {
            dc->SetPixel(p.x, p.y, color_in);
            delete [] m;
            return;
        }
    }

    dc->SetPixel(p.x, p.y, color_out);
    delete [] m;
}
```

2.4. Szakasz vágása konkáv sokszögre

A konvex alakzatra való lehatároláshoz hasonlóan P és Q pontokra illesztünk egy e egyenest. (Már amennyiben nem egyezik meg a két végpont.) Ezek után meghatározzuk a metszéspontjait a konkáv alakzat oldalszakaszaival. Ha nincsenek metszéspontok, akkor a szakasznak nincs látható része, egyébként az e egyenes meredekségének függvényében rendezzük a metszéspontokat az x illetve az y koordináta szerint növekvő sorrendbe. Emellett rendezzük P és Q pontokat is ugyanezen koordináták szerint.



6. ábra. Szakasz vágása konkáv sokszögre

Amennyiben P megfelelő koordinátájánál kisebb az összes metszéspont ezen koordinátája vagy Q ezen koordinátájánál mindegyiküké nagyobb, akkor a szakasznak nem lesz látható része a konkáv ablakon. Egyébként meg kell keresnünk a P és a Q pont helyét a metszéspontok között. (A metszéspontokat a C++ szerint 0-tól indexeljük.) Keressük a P helyét az első metszésponttól felfelé illetve a Q helyét az utolsó metszésponttól csökkenő indexekkel az első felé.

A végső rajzolást egy ciklussal lenne célszerű elvégezni. Ez esetenként a metszéspontokat tároló tömb módosítását jelentheti majd. Ha a P pont két oldalán rendre páratlan és páros indexű metszéspont van, akkor a majdani ciklus `start` indexét a

páros értékre állítjuk be, ha a metszéspontok indexei páros-páratlan sorozatot alkotnak, akkor pedig nem csak a `start` indexet állítjuk be a páros értékre, hanem a hozzá tartozó metszéspont koordinátáit egyenlővé tesszük a P pont koordinátáival. (Erre azért van szükség, mivel a $\overline{PQ}$ szakasz belelóg egy látható tartományba és csak a P illetve a páratlan indexű metszéspont közötti részt kell megrajzolnunk.) Hasonlóképpen megkeressük a Q pont helyét a rendezett metszéspontok között, azonban az utolsótól indulva visszafelé tesszük ezt. Ha páratlan-páros indexű metszéspontok között van, akkor mindössze az `end` indexet kell a páratlan értékre állítanunk, ellenkező esetben az `end` index beállításán túl a páratlan indexű metszéspont koordinátáit egyenlővé kell tennünk a Q pont koordinátáival.

Ezek után mindössze a `start` indextől az `end` indexig meg kell rajzolnunk a szakasz látható darabjait. Ez a páros-páratlan indexű metszéspontok közötti részeket jelenti, tehát a ciklusváltozónkat kettesével kell majd léptetni. Egy probléma lehet csak, ha a `start` index nagyobb, mint az `end` index. Ez abban az esetben lehetséges, ha a vizsgált szakasz a konkáv ablak valamelyik közbülső, nem látható területére esik teljes hosszában. Ekkor azonban a `for` ciklus a végfeltétele miatt egyszer sem fut le, így a láthatóságnak megfelelően nem rajzol semmit.

```
void CGraphPrimitives::ConcaveWLine(
        eBuffer buffer, CConcaveWindow &window, CPoint2D &p,
        CPoint2D &q, COLORREF color_in, COLORREF color_out)
{
    int         i, j, start, end;
    int         *signum, posNum = 0;
    CPoint2D    e, f, t, *m;

    signum = new int[window.m_iVertNum + 1];
    m = new CPoint2D[window.m_iVertNum + 1];
    e = p * q;

    for (j = 0; j < window.m_iWindowNum; j++) {
        for (i = 0; i <= window.m_Windows[j]->m_iVertNum; i++)
            signum[i] =
                SGN(e.Scalar(window.m_Windows[j]->m_Vertices[i]));

        for (i = 0; i < window.m_Windows[j]->m_iVertNum; i++)
            if (signum[i] != signum[i + 1]) {
                f = window.m_Windows[j]->m_Vertices[i] *
                    window.m_Windows[j]->m_Vertices[i + 1];
                m[posNum] = e * f;
                m[posNum].DescartesCoords();
                posNum++;
            }
    }

    delete [] signum;
```

```
if (posNum == 0) {
    Line(buffer, p, q, color_out);
    delete [] m;
    return;
}

start = 0;
end = posNum - 1;

if (fabs(p.y - q.y) < fabs(p.x - q.x)) {
    qsort((void *)m, posNum, sizeof(CPoint2D), sortx);

    if (q.x < p.x)
        t = p, p = q, q = t;

    if ((q.x < m[0].x) || (m[posNum - 1].x < p.x)) {
        Line(buffer, p, q, color_out);
        delete [] m;
        return;
    }

    for (i = 0; i <= posNum - 1; i++)
        if (p.x <= m[i].x) {
            if ((i % 2) == 1) {
                m[i - 1] = p;
                start = i - 1;
                break;
            } else {
                if (q.x <= m[i].x)
                    Line(buffer, p, q, color_out);
                else
                    Line(buffer, p, m[i], color_out);
                start = i;
                break;
            }
        }

    for (i = posNum - 1; i >= 0; i--)
        if (m[i].x <= q.x) {
            if ((i % 2) == 0) {
                m[i + 1] = q;
                end = i + 1;
                break;
            } else {
```

```
                if (p.x >= m[i].x)
                    Line(buffer, q, p, color_out);
                else
                    Line(buffer, q, m[i], color_out);
                end = i;
                break;
            }
        }
} else {
    qsort((void *)m, posNum, sizeof(CPoint2D), sorty);

    if (q.y < p.y)
        t = p, p = q, q = t;

    if ((q.y < m[0].y) || (m[posNum - 1].y < p.y)) {
        Line(buffer, p, q, color_out);
        delete [] m;
        return;
    }

    for (i = 0; i <= posNum - 1; i++)
        if (p.y <= m[i].y) {
            if ((i % 2) == 1) {
                m[i - 1] = p;
                start = i - 1;
                break;
            } else {
                if (q.y <= m[i].y)
                    Line(buffer, p, q, color_out);
                else
                    Line(buffer, p, m[i], color_out);
                start = i;
                break;
            }
        }

    for (i = posNum - 1; i >= 0; i--)
        if (m[i].y <= q.y) {
            if ((i % 2) == 0) {
                m[i + 1] = q;
                end = i + 1;
                break;
            } else {
                if (p.y >= m[i].y)
                    Line(buffer, q, p, color_out);
```

```
                else
                    Line(buffer, q, m[i], color_out);
                end = i;
                break;
            }
        }
    }

    if (end < start) {
        Line(buffer, p, q, color_out);
        delete [] m;
        return;
    }

    for (i = start + 1; i < end; i += 2)
        Line(buffer, m[i], m[i + 1], color_out);
    for (i = start; i < end; i += 2)
        Line(buffer, m[i], m[i + 1], color_in);

    delete [] m;
}
```

IV. fejezet
Harmadrendű görbék

A harmadrendű görbék általános alakja a következő:

$$\underline{r}(t) = \underline{a} \cdot t^3 + \underline{b} \cdot t^2 + \underline{c} \cdot t + \underline{d} \qquad t \in R,$$

ahol $\underline{a}$, $\underline{b}$, $\underline{c}$ és $\underline{d}$ vektorok $\underline{r}(t)$-vel megegyező dimenziójú vektorok. A görbét tulajdonképpen ezen vektoroknak t hatványaival vett szorzatának kombinációjaként tudjuk előállítani.

A továbbiakban síkbeli görbéket fogunk vizsgálni $t \in [0, 1]$ tartományon, amihez vezessük be a következő jelöléseket:

$$\underline{r}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}, \quad C = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \end{bmatrix}, \quad T = \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}$$

azaz:

$$\underline{r}(t) = C \cdot T$$

A következő fejezetekben azt fogjuk megvizsgálni, hogy bizonyos speciális harmadrendű görbéket hogyan tudunk létrehozni.

Ebben a `DrawCurve` függvény lesz a segítségünkre, mely 4×4-es mátrixokkal leírt görbék rajzolására képes. A `resolution` paraméter azt határozza meg, hogy a görbét a rajzolás során hány szakasszal közelítsük.

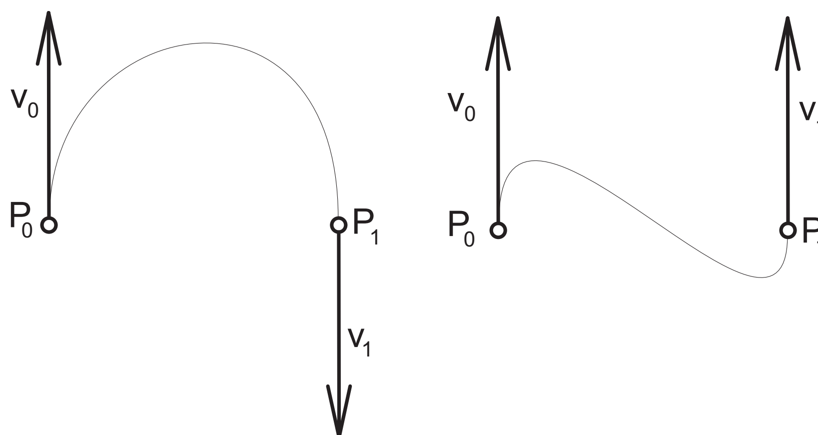
```
void CGraphPrimitives::DrawCurve(eBuffer buffer,
        const CMatrix &g, const CMatrix &m, int resolution)
{
    CDC         *dc = (buffer == Front ? m_cDC : &m_cBackDC);
    int         i;
    CMatrix     t(4, 1), c, coords;
    double      u, step = 1.0 / (double)resolution;

    c = g * m;

    t[0][0] = 0.0; t[1][0] = 0.0; t[2][0] = 0.0; t[3][0] = 1.0;

    coords = c * t;
    dc->MoveTo(coords[0][0], coords[0][1]);

    for (u = step; u <= 1.0; u += step) {
        t[0][0] = u * u * u;
```

```
        t[1][0] = u * u;
        t[2][0] = u;

        coords = c * t;
        dc->LineTo(coords[0][0], coords[0][1]);
    }
}
```

1. Hermite-ív

Adott a görbe két végpontja (P_0, P_1) és ezekben a végpontokban adottak az érintők ($\underline{v_0}$, $\underline{v_1}$).

$$\underline{H}(t) = C \cdot T$$



1. ábra. Hermite görbék

Célunk a C mátrix meghatározása. Ehhez felbontjuk $C = G_H \cdot M_H$ szorzatra, ahol G_H egy 2×4-es, M_H pedig egy 4×4-es mátrix. G_H mátrix a görbét meghatározó geometriai adatokat fogja tartalmazni.

$$G_H = \begin{bmatrix} P_0 & P_1 & \underline{v_0} & \underline{v_1} \end{bmatrix}$$

Ekkor:

$$\underline{H}(0) = P_0 = G_H \cdot M_H \cdot \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}^T$$

$$\underline{H}(1) = P_1 = G_H \cdot M_H \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}^T$$

$$\underline{H'}(0) = \underline{v_0} = G_H \cdot M_H \cdot \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}^T$$

$$\underline{H'}(1) = \underline{v_1} = G_H \cdot M_H \cdot \begin{bmatrix} 3 & 2 & 1 & 0 \end{bmatrix}^T$$

Azaz:

$$\begin{bmatrix} P_0 & P_1 & \underline{v_0} & \underline{v_1} \end{bmatrix} = G_H = G_H \cdot M_H \cdot \begin{bmatrix} 0 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

Így:

$$M_H = \begin{bmatrix} 0 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}^{-1} = \begin{bmatrix} 2 & -3 & 0 & 1 \\ -2 & 3 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix}$$

Összegezve az eddigieket:

$$C = G_H \cdot M_H = \begin{bmatrix} P_0 & P_1 & \underline{v_0} & \underline{v_1} \end{bmatrix} \cdot \begin{bmatrix} 2 & -3 & 0 & 1 \\ -2 & 3 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix}$$

$$\underline{H}(t) = C \cdot T = \begin{bmatrix} P_0 & P_1 & \underline{v_0} & \underline{v_1} \end{bmatrix} \cdot \begin{bmatrix} 2 & -3 & 0 & 1 \\ -2 & 3 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}$$

Vegyük észre, hogy $C = G_H \cdot M_H$ előre letárolható, így az iteráció során mindössze a $\underline{H}(t) = C \cdot T$ szorzatot kell kiszámítanunk a különböző $t \in [0, 1]$-ekre. A megfelelő M_H és G_H mátrixot létrehozó, és ennek segítségével Hermite ívet rajzoló C++ függvény kódja:

```
void CGraphPrimitives::HermiteCurve(
        eBuffer buffer, const CPoint2D &p0, const CPoint2D &p1,
        const CPoint2D &v0, const CPoint2D &v1, int resolution)
{
    CMatrix     g(2, 4), m(4, 4);
    CPoint2D    e, f;

    e = p0 + v0;
    f = p1 + v1;
    Line(buffer, p0, e, RGB(255, 0, 0));
    Line(buffer, p1, f, RGB(255, 0, 0));
    Circle(buffer, e, 2.0, RGB(255, 0, 0));
    Circle(buffer, f, 2.0, RGB(255, 0, 0));
    Circle(buffer, p0, 2.0, RGB(0, 255, 0));
    Circle(buffer, p1, 2.0, RGB(0, 255, 0));

    g[0][0] = p0.x; g[0][1] = p1.x; g[0][2] = v0.x; g[0][3] = v1.x;
    g[1][0] = p0.y; g[1][1] = p1.y; g[1][2] = v0.y; g[1][3] = v1.y;

    m[0][0] =  2.0; m[0][1] = -3.0; m[0][2] = 0.0; m[0][3] = 1.0;
```

```
    m[1][0] = -2.0; m[1][1] =  3.0; m[1][2] = 0.0; m[1][3] = 0.0;
    m[2][0] =  1.0; m[2][1] = -2.0; m[2][2] = 1.0; m[2][3] = 0.0;
    m[3][0] =  1.0; m[3][1] = -1.0; m[3][2] = 0.0; m[3][3] = 0.0;

    DrawCurve(buffer, g, m, resolution);
}
```

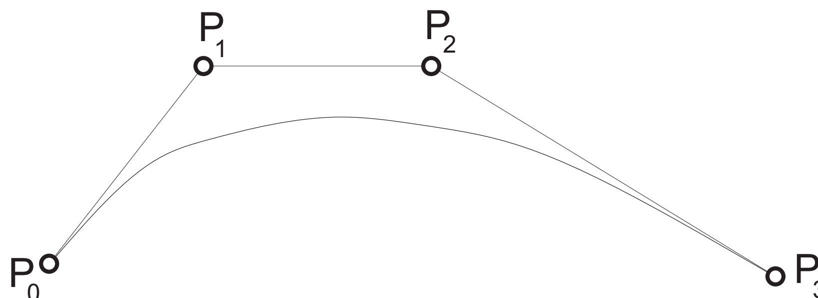
2. Bézier görbe

Adott négy kontrollpont, jelöljük őket P_i-vel ($i = 0, 1, 2, 3$). A Bézier görbe leírásához a Bernstein polinomokat használjuk.

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i} \qquad t \in [0,1]$$

ahol n a kontrollpontok száma, i pedig a vizsgált kontrollpont indexe. Ekkor:

$$\underline{B}(t) = \sum_{i=0}^{n} P_i B_i^n(t) = C \cdot T$$



2. ábra. Bézier görbe

Célunk a C mátrix meghatározása. Ehhez felbontjuk $C = G_B \cdot M_B$ szorzatra, ahol G_B egy 2×4-es, M_B pedig egy 4×4-es mátrix. G_B mátrix a görbét meghatározó geometriai adatokat fogja tartalmazni.

$$G_B = \begin{bmatrix} P_0 & P_1 & P_2 & P_3 \end{bmatrix}$$

A görbe képlete alapján a következő M_B mátrixot kapjuk:

$$M_B = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\underline{B}(t) = C \cdot T = \begin{bmatrix} P_0 & P_1 & P_2 & P_3 \end{bmatrix} \cdot \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}$$

Az G_B és M_B mátrixot megvalósító C++ kód:

```
void CGraphPrimitives::BezierCurve(
        eBuffer buffer, const CPoint2D &p0, const CPoint2D &p1,
        const CPoint2D &p2, const CPoint2D &p3, int resolution)
{
    CMatrix     g(2, 4), m(4, 4);

    Line(buffer, p0, p1, RGB(255, 0, 0));
    Line(buffer, p1, p2, RGB(255, 0, 0));
    Line(buffer, p2, p3, RGB(255, 0, 0));
    Circle(buffer, p0, 2.0, RGB(0, 255, 0));
    Circle(buffer, p1, 2.0, RGB(0, 255, 0));
    Circle(buffer, p2, 2.0, RGB(0, 255, 0));
    Circle(buffer, p3, 2.0, RGB(0, 255, 0));

    g[0][0] = p0.x; g[0][1] = p1.x; g[0][2] = p2.x; g[0][3] = p3.x;
    g[1][0] = p0.y; g[1][1] = p1.y; g[1][2] = p2.y; g[1][3] = p3.y;

    m[0][0] = -1.0; m[0][1] =  3.0; m[0][2] = -3.0; m[0][3] =  1.0;
    m[1][0] =  3.0; m[1][1] = -6.0; m[1][2] =  3.0; m[1][3] =  0.0;
    m[2][0] = -3.0; m[2][1] =  3.0; m[2][2] =  0.0; m[2][3] =  0.0;
    m[3][0] =  1.0; m[3][1] =  0.0; m[3][2] =  0.0; m[3][3] =  0.0;

    DrawCurve(buffer, g, m, resolution);
}
```

3. B-spline

Definíció. *Tekintsük az $u_i \leq u_{i+1} \in R$ $(i = -\infty, \ldots, \infty)$ skalárokat. Az*

$$N_i^1(u) = \begin{cases} 1, & \text{ha } u_i \leq u < u_{i+1}; \\ 0 & \text{egyébként} \end{cases}$$

$$N_i^k(u) = \frac{u - u_i}{u_{i+k-1} - u_i} N_i^{k-1}(u) + \frac{u_{i+k} - u}{u_{i+k} - u_{i+1}} N_{i+1}^{k-1}(u)$$

rekurzióval definiált függvényt normalizált B-spline alapfüggvénynek nevezzük, az u_i skalárokat pedig csomóértékeknek (knot values), vagy csomóvektornak (knot vector). Az előforduló $\frac{0}{0}$-t definíció szerint 0-nak tekintjük.

Definíció. *Az*

$$\underline{S}(u) = \sum_i P_i N_i^k(u) = C \cdot T$$

görbét $k-1$*-ed fokú (vagy* k*-ad rendű) B-spline görbének nevezzük, ahol* $N_i^k(u)$ *a* $k-1$*-ed fokú normalizált B-spline alapfüggvény. A* P_i *pontokat kontrollpontoknak, vagy de Boor-pontoknak, az általuk meghatározott poligont pedig kontroll- vagy de Boor-poligonnak nevezzük.*

Négy kontrollponttal fogunk rendelkezni, illetve egy olyan uniform u_i sorozattal, ahol az egymást követő értékek közötti különbség állandó és nem nulla.

Célunk a C mátrix meghatározása. Ehhez felbontjuk $C = G_{B-spline} \cdot M_{B-spline}$ szorzatra, ahol $G_{B-spline}$ egy 2×4-es, $M_{B-spline}$ pedig egy 4×4-es mátrix. $G_{B-spline}$ mátrix a görbét meghatározó geometriai adatokat fogja tartalmazni.

$$G_{B-spline} = \begin{bmatrix} P_0 & P_1 & P_2 & P_3 \end{bmatrix}$$

$$M_{B-spline} = \frac{1}{6} \cdot \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 0 & 4 \\ -3 & 3 & 3 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\underline{S}(t) = C \cdot T = \begin{bmatrix} P_0 & P_1 & P_2 & P_3 \end{bmatrix} \cdot \frac{1}{6} \cdot \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 0 & 4 \\ -3 & 3 & 3 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}$$

```
void CGraphPrimitives::B_Spline(
        eBuffer buffer, const CPoint2D &p0, const CPoint2D &p1,
        const CPoint2D &p2, const CPoint2D &p3, int resolution)
{
    CMatrix     g(2, 4), m(4, 4);

    Line(buffer, p0, p1, RGB(255, 0, 0));
    Line(buffer, p1, p2, RGB(255, 0, 0));
    Line(buffer, p2, p3, RGB(255, 0, 0));
    Circle(buffer, p0, 2.0, RGB(0, 255, 0));
    Circle(buffer, p1, 2.0, RGB(0, 255, 0));
    Circle(buffer, p2, 2.0, RGB(0, 255, 0));
    Circle(buffer, p3, 2.0, RGB(0, 255, 0));

    g[0][0] = p0.x; g[0][1] = p1.x; g[0][2] = p2.x; g[0][3] = p3.x;
    g[1][0] = p0.y; g[1][1] = p1.y; g[1][2] = p2.y; g[1][3] = p3.y;

    m[0][0] = -1.0; m[0][1] =  3.0; m[0][2] = -3.0; m[0][3] =  1.0;
    m[1][0] =  3.0; m[1][1] = -6.0; m[1][2] =  0.0; m[1][3] =  4.0;
    m[2][0] = -3.0; m[2][1] =  3.0; m[2][2] =  3.0; m[2][3] =  1.0;
    m[3][0] =  1.0; m[3][1] =  0.0; m[3][2] =  0.0; m[3][3] =  0.0;
```

```
    m *= 1.0 / 6.0;

    DrawCurve(buffer, g, m, resolution);
}
```

V. fejezet
3D ponttranszformációk

Az egyes műveleteknek meg tudjuk határozni a 4×4-es mátrixát. P pont P' transzformáltját megkapjuk, ha a P pontot megszorozzuk balról a végrehajtandó transzformációk mátrixával.

$$P' = (M_n \cdot (\ \ldots \ \cdot (M_1 \cdot P) \ldots))$$

Mivel a fenti szorzások átzárójelezhetőek és az M_i mátrixok összeszorozhatóak, a következő egyszerűsítést végezhetjük el:

$$M = M_n \cdot \ \ldots \ \cdot M_1$$

$$P' = M \cdot P$$

1. Egybevágósági transzformációk

1.1. Eltolás

Legyen az eltolás vektora $\underline{d}$. Ekkor $P' = P + \underline{d} = M_{translate} \cdot P$. Ugyanez mátrixos alakban:

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} P_x + d_x \\ P_y + d_y \\ P_z + d_z \\ 1 \end{bmatrix}$$

Az $M_{translate}$-et szolgáltató C++ függvény kódja:

```
CMatrix CGraphPrimitives::Translate(const CPoint3D &d)
{
    CMatrix     m(4, 4);

    m.LoadIdentity();
    m[0][3] = d.x;
    m[1][3] = d.y;
    m[2][3] = d.z;

    return m;
}
```

1.2. Forgatás

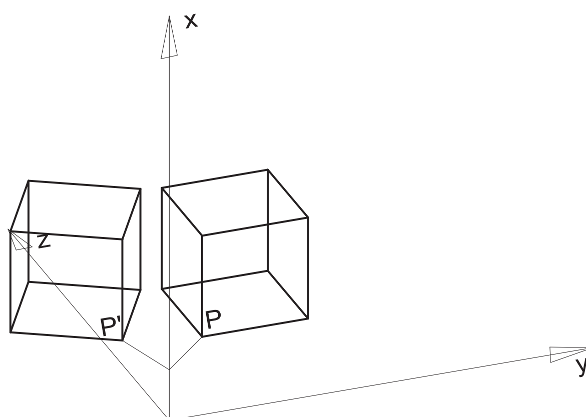
A tetszőleges irányú tengely körüli forgatást az egyes koordináta tengelyek körüli forgatásokra lehet felbontani.

1.2.1. Forgatás az x tengely körül. Az x koordináta a forgatás során vátozatlan marad, míg a P pont az $[y, z]$ síkban α szöggel elfordul.

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\alpha) & -sin(\alpha) & 0 \\ 0 & sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} P_x \\ cos(\alpha) \cdot P_y - sin(\alpha) \cdot P_z \\ sin(\alpha) \cdot P_y + cos(\alpha) \cdot P_z \\ 1 \end{bmatrix}$$

Az M_{rotate_x}-et szolgáltató C++ függvény kódja:

```
CMatrix CGraphPrimitives::RotateX(double alpha)
{
    CMatrix     m(4, 4);

    m.LoadIdentity();
    m[1][1] = cos(alpha);
    m[1][2] = -sin(alpha);
    m[2][1] = sin(alpha);
    m[2][2] = cos(alpha);

    return m;
}
```



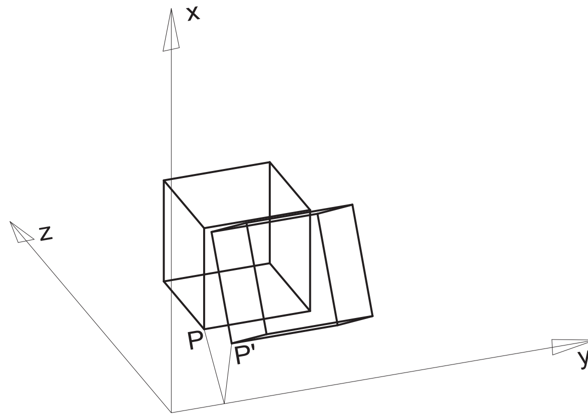
1. ábra. Kocka forgatása az x tengely körül

1.2.2. Forgatás az y tengely körül. Az y koordináta a forgatás során vátozatlan marad, míg a P pont az $[x, z]$ síkban α szöggel elfordul.

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} cos(\alpha) & 0 & sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\alpha) & 0 & cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} cos(\alpha) \cdot P_x + sin(\alpha) \cdot P_z \\ P_y \\ -sin(\alpha) \cdot P_x + cos(\alpha) \cdot P_z \\ 1 \end{bmatrix}$$

Az M_{rotate_y}-t szolgáltató C++ függvény kódja:

```
CMatrix CGraphPrimitives::RotateY(double alpha)
{
    CMatrix     m(4, 4);

    m.LoadIdentity();
    m[0][0] = cos(alpha);
    m[0][2] = sin(alpha);
    m[2][0] = -sin(alpha);
    m[2][2] = cos(alpha);

    return m;
}
```



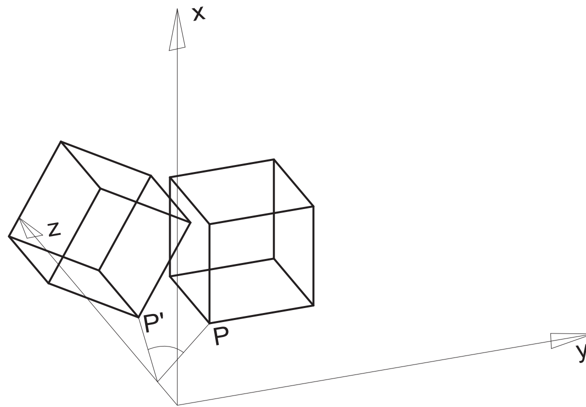
2. ábra. Kocka forgatása az y tengely körül

1.2.3. Forgatás a z tengely körül. A z koordináta a forgatás során vátozatlan marad, míg a P pont az $[x, y]$ síkban α szöggel elfordul.

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} cos(\alpha) & -sin(\alpha) & 0 & 0 \\ sin(\alpha) & cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} cos(\alpha) \cdot P_x - sin(\alpha) \cdot P_y \\ sin(\alpha) \cdot P_x + cos(\alpha) \cdot P_y \\ P_z \\ 1 \end{bmatrix}$$

Az M_{rotate_z}-t szolgáltató C++ függvény kódja:

```
CMatrix CGraphPrimitives::RotateZ(double alpha)
{
    CMatrix     m(4, 4);

    m.LoadIdentity();
    m[0][0] = cos(alpha);
    m[0][1] = -sin(alpha);
    m[1][0] = sin(alpha);
    m[1][1] = cos(alpha);

    return m;
}
```



3. ábra. Kocka forgatása az z tengely körül

1.3. Tükrözés koordinátasíkra

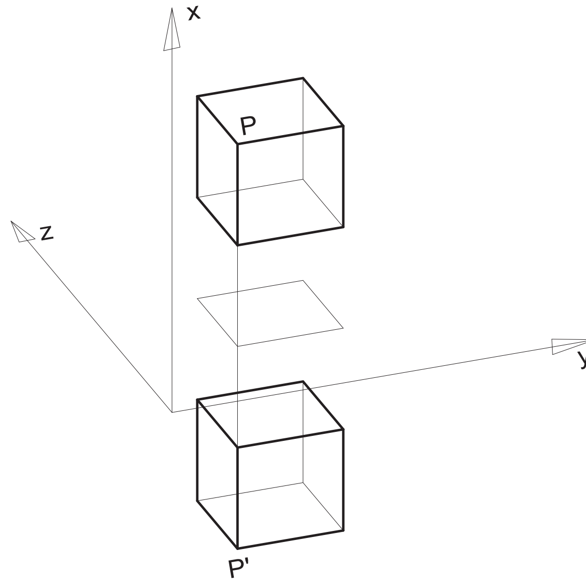
1.3.1. Tükrözés az $[y, z]$ koordinátasíkra. A tükrözés során tulajdonképpen az adott pont x koordinátáját kell -1-el megszoroznunk, hogy a tükörképet megkapjuk.

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} -P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

Az $M_{mirror_{yz}}$-t szolgáltató C++ függvény kódja:

```
CMatrix CGraphPrimitives::MirrorYZ()
{
    CMatrix     m(4, 4);

    m.LoadIdentity();
    m[0][0] = -1.0;
```

```
    return m;
}
```



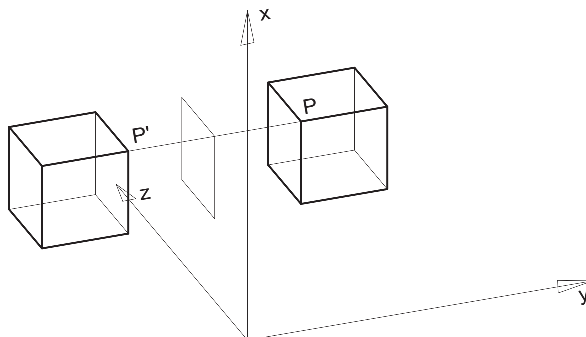
4. ábra. Kocka tükrözése az $[y, z]$ koordinátasíkra

1.3.2. Tükrözés az $[x, z]$ koordinátasíkra. A tükrözés során tulajdonképpen az adott pont y koordinátáját kell -1-el megszoroznunk, hogy a tükörképet megkapjuk.

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} P_x \\ -P_y \\ P_z \\ 1 \end{bmatrix}$$

Az $M_{mirror_{xz}}$-t szolgáltató C++ függvény kódja:

```
CMatrix CGraphPrimitives::MirrorXZ()
{
    CMatrix     m(4, 4);

    m.LoadIdentity();
    m[1][1] = -1.0;

    return m;
}
```



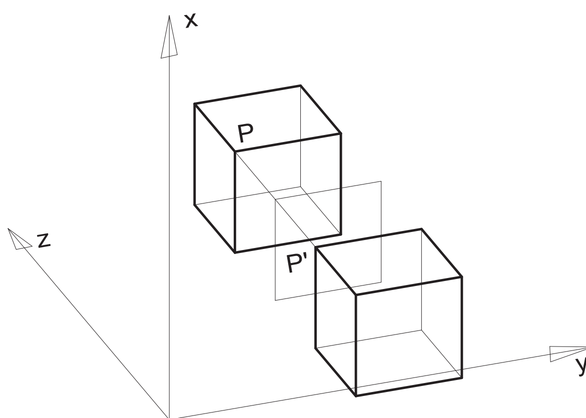
5. ábra. Kocka tükrözése az $[x, z]$ koordinátasíkra

1.3.3. Tükrözés az $[x, y]$ koordinátasíkra. A tükrözés során tulajdonképpen az adott pont z koordinátáját kell -1-el megszoroznunk, hogy a tükörképet megkapjuk.

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \\ -P_z \\ 1 \end{bmatrix}$$

Az $M_{mirror_{xy}}$-t szolgáltató C++ függvény kódja:

```
CMatrix CGraphPrimitives::MirrorXY()
{
    CMatrix     m(4, 4);

    m.LoadIdentity();
    m[2][2] = -1.0;

    return m;
}
```



6. ábra. Kocka tükrözése az $[x, y]$ koordinátasíkra

2. Hasonlósági transzformációk

2.1. Origó középpontú kicsinyítés és nagyítás

Tulajdonképpen egy olyan skálázásnak tekinthető ez a művelet, ahol mindhárom tengely mentén ugyanakkora mértékkel nyújtjuk meg, illetve nyomjuk össze az objektumokat. Programozáskor nem is szokták külön esetként kezelni. Sokszor még a tükrözésnek sincs saját függvénye (pl. OpenGL) hanem – a paramétereknek negatív értéket is megengedve – a skálázás függvényét használják rá.

Azért tárgyaljuk mégis külön, mert a két művelet eltérő transzformációs osztályba tartozik. P pont kicsinyítése illetve nagyítása λ értékkel:

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} \lambda & 0 & 0 & 0 \\ 0 & \lambda & 0 & 0 \\ 0 & 0 & \lambda & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} \lambda \cdot P_x \\ \lambda \cdot P_y \\ \lambda \cdot P_z \\ 1 \end{bmatrix},$$

ahol $0 < \lambda \in R$. $\lambda = 0$ minden pontot a $[0, 0, 0, 1]$ pontba visz át. Azért nem elegendő a nem nulla feltétel, mivel egy negatív λ érték egyben tükrözést is jelentene a három koordinátasíkra.

```
CMatrix CGraphPrimitives::Magnify(double lambda)
{
    CMatrix    m(4, 4);

    m.LoadZero();
    m[0][0] = lambda;
    m[1][1] = lambda;
    m[2][2] = lambda;
    m[3][3] = 1.0;

    return m;
}
```

3. Affin transzformációk

3.1. Skálázás

A skálázás során az objektumokat az egyes tengelyek mentén eltérő mértékben nyújthatjuk meg vagy nyomhatjuk össze.

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} \lambda & 0 & 0 & 0 \\ 0 & \mu & 0 & 0 \\ 0 & 0 & \nu & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} \lambda \cdot P_x \\ \mu \cdot P_y \\ \nu \cdot P_z \\ 1 \end{bmatrix},$$

ahol $\lambda > 0, \mu > 0, \nu > 0$ és $\lambda, \mu, \nu \in R$. A negatív értékek a kicsinyítés-nagyításhoz hasonlóan a tükrözés vonzat miatt nem megengedettek.

Az M_{scale}-t szolgáltató C++ függvény kódja:

```
CMatrix CGraphPrimitives::Scale(double lambda, double mu, double nu)
{
    CMatrix     m(4, 4);

    m.LoadZero();
    m[0][0] = lambda;
    m[1][1] = mu;
    m[2][2] = nu;
    m[3][3] = 1.0;

    return m;
}
```

3.2. Nyírás

A térbeli nyírás a tér P pontjainak egy fix síkkal párhuzamos csúsztatása. Legyen a fix síkunk origón áthaladó. A csúszás mértéke arányos a pontnak a fix síktól való d távolságával. A sík állása megadható egységhosszúságú $\underline{n}$ normálvektorával, a csúszást pedig a $\underline{t}$ irány egységvektorával, amely merőleges $\underline{n}$-re.

Egy λ mértékű nyírás:

$$P' = P + \lambda \cdot d \cdot \underline{t} = P + \lambda \cdot (\underline{n} \cdot \underline{p}) \cdot \underline{t}$$

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 + \lambda \cdot t_x \cdot n_x & \lambda \cdot t_x \cdot n_y & \lambda \cdot t_x \cdot n_z & 0 \\ \lambda \cdot t_y \cdot n_x & 1 + \lambda \cdot t_y \cdot n_y & \lambda \cdot t_y \cdot n_z & 0 \\ \lambda \cdot t_z \cdot n_x & \lambda \cdot t_z \cdot n_y & 1 + \lambda \cdot t_z \cdot n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} =$$

$$= \begin{bmatrix} P_x + \lambda \cdot t_x \cdot (P_x \cdot n_x + P_y \cdot n_y + P_z \cdot n_z) \\ P_y + \lambda \cdot t_y \cdot (P_x \cdot n_x + P_y \cdot n_y + P_z \cdot n_z) \\ P_z + \lambda \cdot t_z \cdot (P_x \cdot n_x + P_y \cdot n_y + P_z \cdot n_z) \\ 1 \end{bmatrix}$$

Az $M_{shearing}$-et szolgáltató C++ függvény kódja:

```
CMatrix CGraphPrimitives::Shearing(double lambda,
        const CPoint2D &t, const CPoint2D &n)
{
    CMatrix     m(4, 4);

    m.LoadIdentity();
    m[0][0] += lambda * t.x * n.x;
    m[0][1] = lambda * t.x * n.y;
    m[0][2] = lambda * t.x * n.z;
```

```
    m[1][0] = lambda * t.y * n.x;
    m[1][1] += lambda * t.y * n.y;
    m[1][2] = lambda * t.y * n.z;
    m[2][0] = lambda * t.z * n.x;
    m[2][1] = lambda * t.z * n.y;
    m[2][2] += lambda * t.z * n.z;

    return m;
}
```

VI. fejezet
3 dimenziós tér leképezése képsíkra és a Window to Viewport transzformáció

Ez a fejezet a háromdimenziós tér leképezése mellett azzal a transzformációval foglalkozik, mely a képsíkon létrejött ablakot képes egy eltérő méretű és oldalarányú képernyőre transzformálni.

1. Leképezések

Az ismertetendő leképezések célja az, hogy a térbeli objektumainkat leképezzük egy kétdimenziós képsíkra, ami rendszerint az $[x, y]$ sík. Az alább ismertetett centrális és párhuzamos vetítésnél is ezt a síkot fogjuk használni.

1.1. Centrális vetítés

Centrális vetítésnél a vetítés centrumát rendszerint a z tengelyen helyezzük el a pozitív oldalon. Távolságát az origótól s-el jelöljük. Egy $P[P_x, P_y, P_z, 1]$ pont vetületét, P'-t a következőképpen határozhatjuk meg a párhuzamos szelők tétele alapján:

$$\frac{P'_x}{s} = \frac{P_x}{s - P_z} \qquad \frac{P'_y}{s} = \frac{P_y}{s - P_z}$$

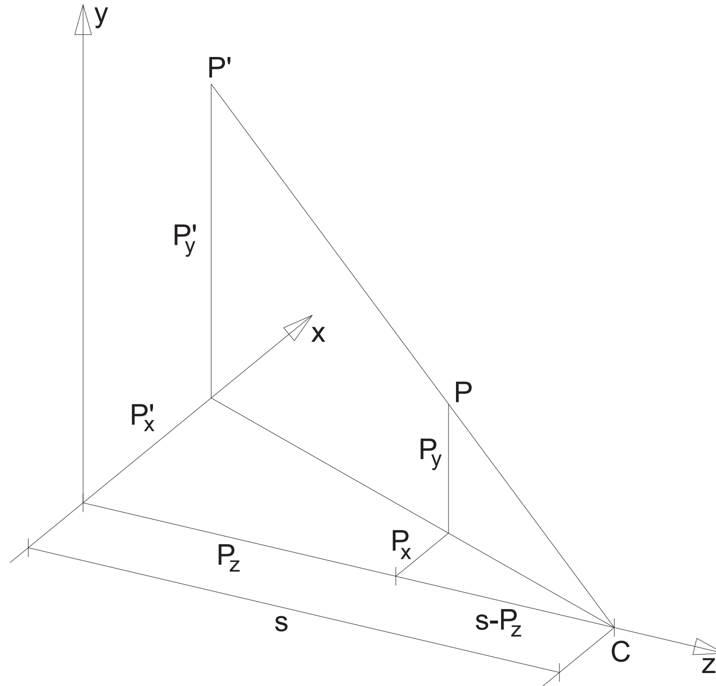
$$P'_x = P_x \cdot \frac{s}{s - P_z} \qquad P'_y = P_y \cdot \frac{s}{s - P_z}$$

Ezek alapján P_z-t bele kellene írnunk a transzformációs mátrixba, azonban ez nyilván lehetetlen, mivel a mátrix nem függhet a transzformálandó pontoktól. Megoldás az lehet, ha a negyedik homogén koordinátába próbáljuk bevinni a perspektív torzulást. Az alábbi mátrix ezen az úton próbálja megoldani a problémát.

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{-1}{s} & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \\ 0 \\ 1 - \frac{P_z}{s} \end{bmatrix} = \begin{bmatrix} P_x \cdot \frac{s}{s-P_z} \\ P_y \cdot \frac{s}{s-P_z} \\ 0 \\ 1 \end{bmatrix}$$

A fenti mátrixot megvalósító függvény:

```
CMatrix CGraphPrimitives::CentralProjection(double s)
{
    CMatrix     m(4, 4);
```



1. ábra. P pont perspektív képe

```
    m.LoadIdentity();
    m[2][2] = 0.0;
    m[3][2] = -1.0 / s;

    return m;
}
```

1.2. Párhuzamos vetítés

Ennél a leképezésnél adott a vetítés vektora $\underline{v}$. A képsíkon egy P pont képét úgy találhatjuk meg, hogy tekintjük azt a P pontot tartalmazó egyenest, mely $\underline{v}$-vel párhuzamos. Ezek után vesszük ennek az egyenesnek és a képsíknak a metszéspontját. Kiszámítása a következőképpen történik:

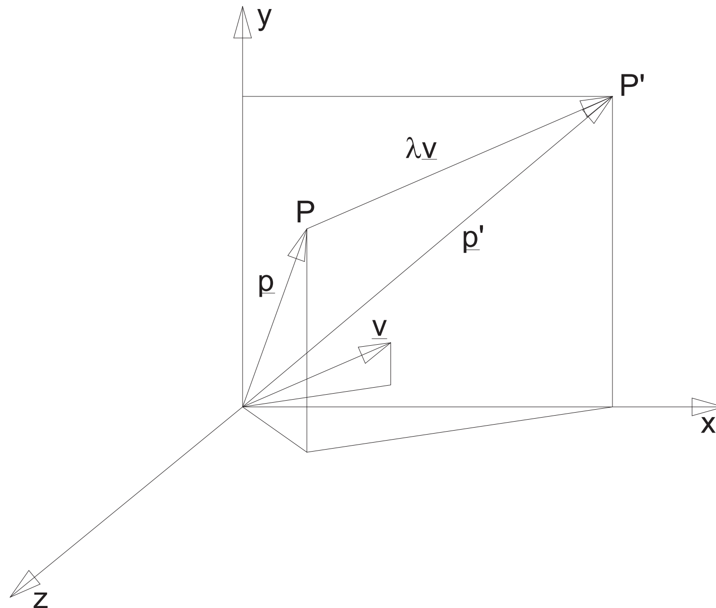
$$P' = P + \lambda \cdot \underline{v}$$

Kibontva ezt az egyenletet, a következő három egyenlőséget kapjuk:

$$P'_x = P_x + \lambda \cdot v_x$$

$$P'_y = P_y + \lambda \cdot v_y$$

$$P'_z = P_z + \lambda \cdot v_z$$



2. ábra. P pont párhuzamosan vetített képe

Ebben a három egyenletben látszólag négy ismeretlen van, azonban ha figyelembe vesszük, hogy a képsík az $[x, y]$ sík, akkor tudjuk azt, hogy $P'_z = 0$. Így mindössze egy háromismeretlenes lineáris egyenletrendszert kell megoldanunk. A harmadik egyenlet alapján:

$$\lambda = \frac{-P_z}{v_z}$$

λ-t visszahelyettesítve az első két egyenletbe a következőket kapjuk:

$$P'_x = P_x + \lambda \cdot v_x = P_x - P_z \cdot \frac{v_x}{v_z}$$

$$P'_y = P_y + \lambda \cdot v_y = P_y - P_z \cdot \frac{v_y}{v_z}$$

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\frac{v_x}{v_z} & 0 \\ 0 & 1 & -\frac{v_y}{v_z} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} P_x - P_z \cdot \frac{v_x}{v_z} \\ P_y - P_z \cdot \frac{v_y}{v_z} \\ 0 \\ 1 \end{bmatrix}$$

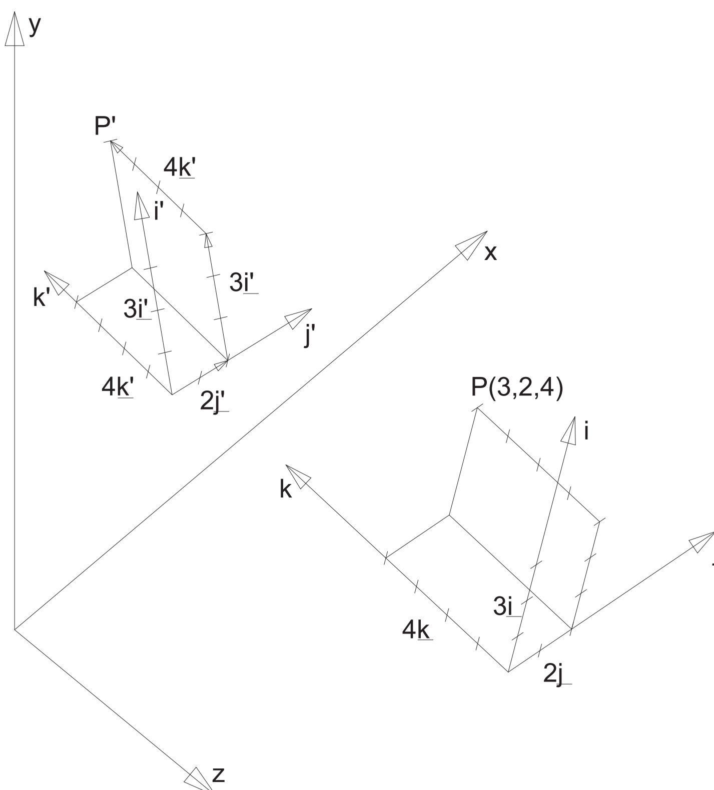
A fenti mátrixot megvalósító függvény:

```
CMatrix CGraphPrimitives::ParallelProjection(const CPoint3D &v)
{
    CMatrix     m(4, 4);

    m.LoadIdentity();
    m[0][2] = -(v.x / v.z);
    m[1][2] = -(v.y / v.z);
    m[2][2] = 0.0;
```

```
    return m;
}
```

1.3. Axonometria

Az axonometrikus ábrázolás során adottak a térbeli i, j, k Descartes-féle derékszögű koordináta-rendszer origójának és a koordinátatengelyek egységvektorainak képei a képsíkon. Egy $P(P_i, P_j, P_k)$ pont $P'(P'_x, P'_y)$ képét úgy kaphatjuk meg, hogy az egyes koordinátákat megszorozzuk a hozzájuk tartozó tengelyirányú ($\underline{i}$, $\underline{j}$, $\underline{k}$) egységvektorok $[x, y]$ képsíkon vett képének ($\underline{i'}$, $\underline{j'}$, $\underline{k'}$) vektoraival és összegezzük ezeket a vektorokat.



3. ábra. P pont axonometrikus képe

$$P' = P_i \cdot \underline{i'} + P_j \cdot \underline{j'} + P_k \cdot \underline{k'}$$

Azaz:

$$P'_x = P_i \cdot i'_x + P_j \cdot j'_x + P_k \cdot k'_x$$

$$P'_y = P_i \cdot i'_y + P_j \cdot j'_y + P_k \cdot k'_y$$

A leképezést megvalósító transzformáció mátrixos alakban:

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} i'_x & j'_x & k'_x & 0 \\ i'_y & j'_y & k'_y & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_i \\ P_j \\ P_k \\ 1 \end{bmatrix} = \begin{bmatrix} P_i \cdot i'_x + P_j \cdot j'_x + P_k \cdot k'_x \\ P_i \cdot i'_y + P_j \cdot j'_y + P_k \cdot k'_y \\ 0 \\ 1 \end{bmatrix}$$

A megfelelő 4×4-es mátrixot létrehozó függvény, ahol az `i`, `j` és `k` paraméterek a nekik megfelelő tengelyirányú egységvektorok képsíkon lévő képeinek a vektorát jelentik:

```
CMatrix CGraphPrimitives::Axonometry(
        const CPoint2D &i, const CPoint2D &j, const CPoint2D &k)
{
    CMatrix     m(4, 4);

    m.LoadZero();
    m[0][0] = i.x;
    m[0][1] = j.x;
    m[0][2] = k.x;
    m[1][0] = i.y;
    m[1][1] = j.y;
    m[1][2] = k.y;
    m[3][3] = 1.0;

    return m;
}
```

2. Window to Viewport transzformáció

Ez a transzformáció egy képsíkon levő ablakot tud transzformálni egy képernyőn megjelenítendő ablakra. Az eredeti téglalap alakú ablakot, amelyet az ablak (w_{left}, w_{bottom}) és (w_{right}, w_{top}) átellenes csúcspontjai definiálnak, először eltoljuk az origóba. Következő lépésként a két ablak oldalarányai alapján skálázást végzünk. A most már helyes méretű ablakot, eltoljuk a végleges pozíciójába. Az átellenes csúcsok a (v_{left},

v_{bottom}) és (v_{right}, v_{top}) pontokba kerülnek. A három mátrix szorzataként nyert transzformációt alkalmazva az objektumokat a képernyőn elhelyezkedő ablakban tudjuk megjeleníteni.

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & v_{left} \\ 0 & 1 & 0 & v_{bottom} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \frac{v_{right}-v_{left}}{w_{right}-w_{left}} & 0 & 0 & 0 \\ 0 & \frac{v_{top}-v_{bottom}}{w_{top}-w_{bottom}} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot$$

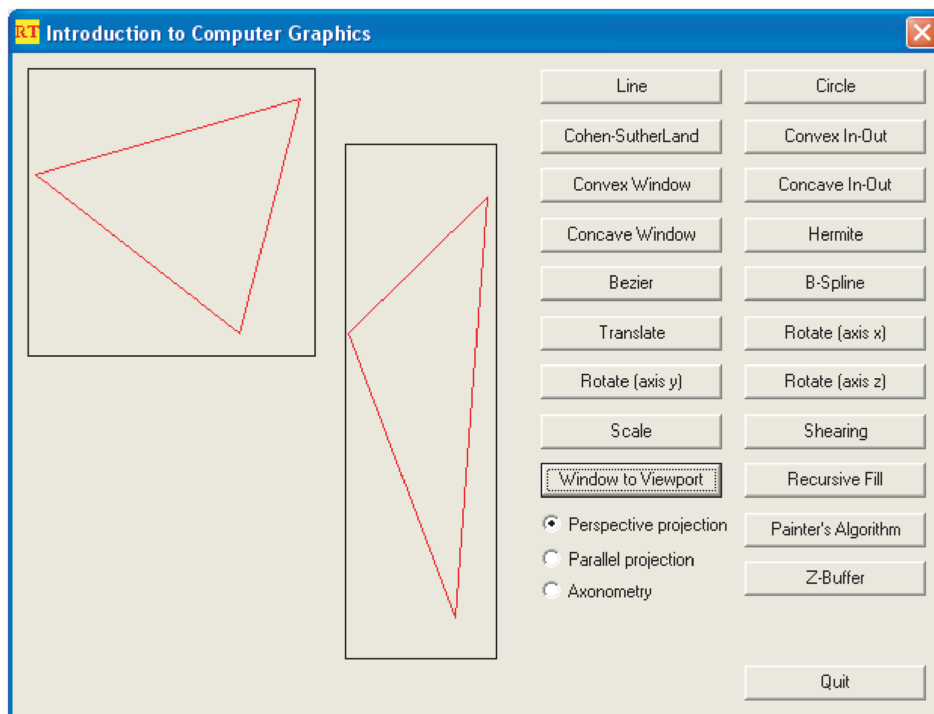
$$\cdot \begin{bmatrix} 1 & 0 & 0 & -w_{left} \\ 0 & 1 & 0 & -w_{bottom} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} =$$

$$= \begin{bmatrix} \frac{v_{right}-v_{left}}{w_{right}-w_{left}} & 0 & 0 & v_{left} - w_{left} \cdot \frac{v_{right}-v_{left}}{w_{right}-w_{left}} \\ 0 & \frac{v_{top}-v_{bottom}}{w_{top}-w_{bottom}} & 0 & v_{bottom} - w_{bottom} \cdot \frac{v_{top}-v_{bottom}}{w_{top}-w_{bottom}} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} =$$

$$= \begin{bmatrix} v_{left} + (P_x - w_{left}) \cdot \frac{v_{right}-v_{left}}{w_{right}-w_{left}} \\ v_{bottom} + (P_y - w_{bottom}) \cdot \frac{v_{top}-v_{bottom}}{w_{top}-w_{bottom}} \\ P_z \\ 1 \end{bmatrix}$$

A transzformációs mátrixot szolgáltató függvény kódja:

```
CMatrix CGraphPrimitives::WindowToViewport(
        const CRect &window, const CRect &viewport)
{
    CMatrix     m(4, 4);
    double      t1 = window.right - window.left;
    double      t2 = window.top   - window.bottom;

    m.LoadIdentity();

    if ((t1 == 0.0) || (t2 == 0.0)) return m;

    m[0][0] = (viewport.right - viewport.left  ) / t1;
    m[1][1] = (viewport.top   - viewport.bottom) / t2;
    m[0][3] = viewport.left   - window.left   * m[0][0];
    m[1][3] = viewport.bottom - window.bottom * m[1][1];

    return m;
}
```

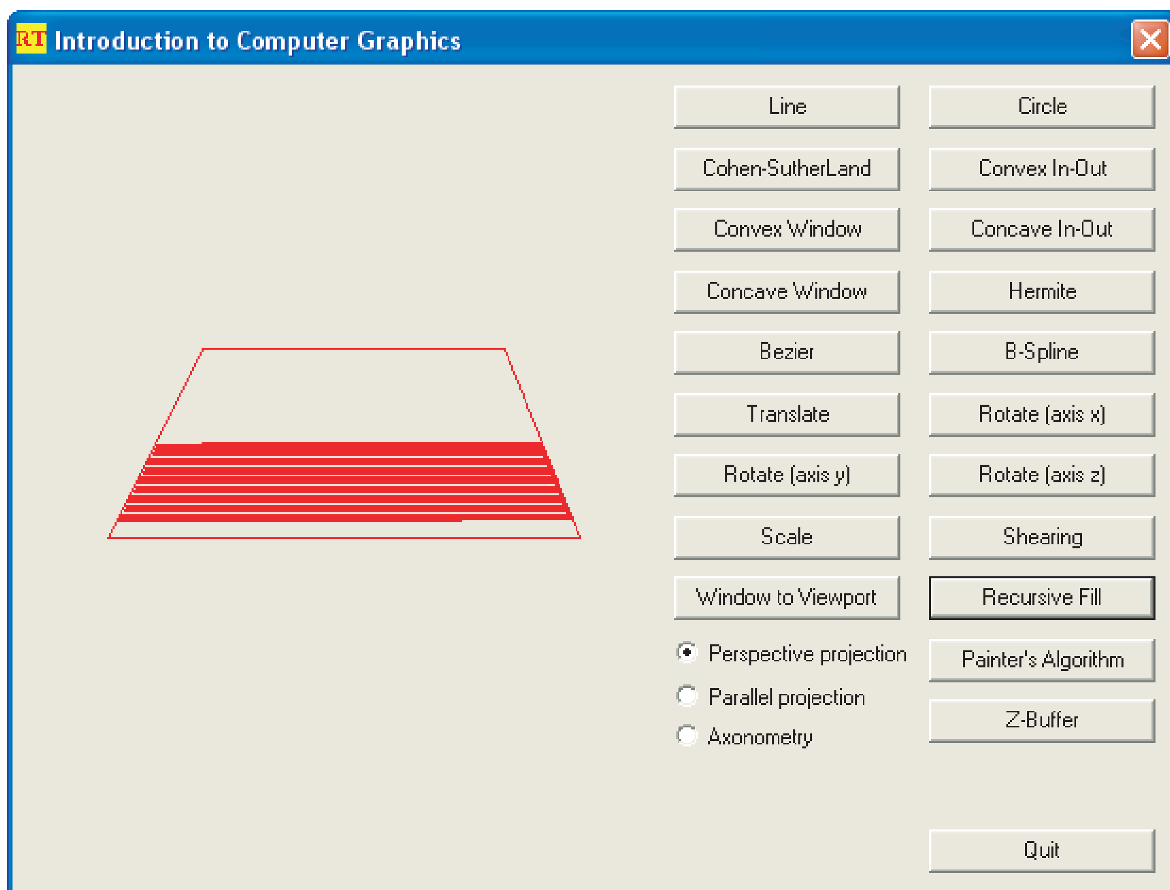


4. ábra. Háromszöget tartalmazó *window* transzformációja *viewport*ra

VII. fejezet
Rekurzív kitöltés

Ezen módszernél feltételezzük, hogy rendelkezünk a kitöltendő alakzat határvonalával a kitöltési színnel megrajzolva. Emellett ismernünk kell az alakzat egy belső pontját.

Az algoritmust ebből az ismert pontból indítjuk el. Amennyiben a vizsgált pont nem a kitöltési színnel rendelkezik, kigyújtjuk a megfelelő színnel, majd a pixel közvetlen szomszédaira meghívjuk ismételten a függvényt. Ha a vizsgált pixel színe megegyezik a kitöltési színnel, nem kell semmit sem tennünk, ugyanis elértük az alakzat határvonalát.



1. ábra. A rekurzív kitöltés egy közbülső lépése

Lassúsága mellett nagy hátránya a módszernek a hatalmas verem igény. Éppen emiatt inkább az alakzat definícóján alapuló algoritmusokat szoktak alkalmazni. A Z-buffer alkalmazhatóságához például egy hatékony háromszögrajzoló algoritmust fogunk majd implementálni.

```
void CGraphPrimitives::RecursiveFill(
        eBuffer buffer, CPoint2Dint p, COLORREF color)
{
    CDC     *dc = (buffer == Front ? m_cDC : &m_cBackDC);

    if (dc->GetPixel(p.x, p.y) != color) {
        dc->SetPixel(p.x, p.y, color);
        RecursiveFill(buffer, CPoint2Dint(p.x + 1, p.y), color);
        RecursiveFill(buffer, CPoint2Dint(p.x - 1, p.y), color);
        RecursiveFill(buffer, CPoint2Dint(p.x, p.y + 1), color);
        RecursiveFill(buffer, CPoint2Dint(p.x, p.y - 1), color);
    }
}
```

VIII. fejezet
Testmodellezés

A testek modellezéséhez szükségszerűen tovább kellett lépni a testet alkotó pontok tárolásán. A drótvázmodell ugyan képes az éleket kezelni, azonban a láthatóság szemléltetésére alkalmatlan. A láthatóság szerinti ábrázolás a felületmodellek használatával lehetséges.

1. Drótvázmodell (Wire Frame Model)

A legegyszerűbb modell, azonban a vele előállított kép nem mindig értelmezhető egyértelműen, mivel a láthatóság szerinti ábrázolás nem lehetséges. Nem teljes értékű testmodell, mivel lehetetlen testeket is leírhatunk vele.

2. Felületmodell (B-rep adatstruktúra)

A B-rep (Boundary Representation) felületmodell a drótvázmodell továbbfejlesztésének tekinthető.

Lényeges tulajdonsága, hogy geometriai és topológiai információkat tárol. Geometriai információ a lapok, élek egyenlete vagy a csúcspontok koordinátái. Topológián pedig a lapok, élek és csúcspontok kapcsolatát értjük. Az éleket például csúcsokra való mutatókkal írjuk le.

Egy élt egy `SPair` struktúrával adhatunk meg.

```
struct SPair
{
            SPair(int x = 0, int y = 0) {a = x; b = y;};

    int     a, b;
};
```

Hasonlóképp egy háromszög csúcsait egy `STriangle` struktúrával adhatjuk meg.

```
struct STriangle
{
            STriangle(int x = 0, int y = 0, int z = 0)
                {a = x; b = y; c = z;};

    int     a, b, c;
};
```

Amennyiben csak síklapokat engedünk meg, akkor csak poliédereket tudunk egzaktul reprezentálni. A görbült felületeket poliéderekkel kell közelítenünk. Az ilyen modellt poliéder modellnek hívjuk.

A modellezhető testekre vonatkozó megszorítások:

- a lapok határa egyszerű sokszög
- egyetlen csúcs sem belső pontja valamely élnek
- nincsenek egymást metsző élpárok
- minden élben pontosan két lap találkozik
- egymáshoz vagy önmagukhoz élben vagy csúcsban csatlakozó testek nem modellezhetők

Egy egyszerű B-rep modell C++ implementációja:

```
class CB_Rep
{
public:
                CB_Rep(int vertNum, int edgeNum, int triangleNum) {
                    m_iVertNum = vertNum;
                    m_iEdgeNum = edgeNum;
                    m_iTriangleNum = triangleNum;

                    m_Vertices = new CPoint3D[m_iVertNum];
                    m_Edges = new SPair[m_iEdgeNum];
                    m_Triangles = new STriangle[m_iTriangleNum];
                };
    virtual     ~CB_Rep() {
                    delete [] m_Vertices;
                    delete [] m_Edges;
                    delete [] m_Triangles;
                };

    CPoint3D    *m_Vertices;
    int         m_iVertNum;
    SPair       *m_Edges;
    int         m_iEdgeNum;
    STriangle   *m_Triangles;
    int         m_iTriangleNum;
};
```

3. Testmodellek megjelenítése

Miután valamilyen módon előállítottuk a térbeli testmodellt, annak a legvalósághűbb megjelenítése a célunk. Minimális igény a láthatóság szerinti ábrázolás. Egyszerűbb esetben a takart vonalakat távolítjuk el például (*hidden line elimination*) plotteres rajz esetén, míg raszteres megjelenítő esetén, mint a képernyő, a takart felületeket távolítjuk el (*hidden surface elimination*).

3.1. Hátsó lapok eltávolítása

A poliéder modellek megjelenítését optimálisan a hátsó lapok eltávolításával (*back-face culling*) kezdjük. Ezeknek a lapoknak a normálvektora 90°-nál nagyobb szöget zár be a nézőpontba mutató vektorral. A vektorok skalárszorzatának előjel vizsgálatával határozhatjuk meg a legegyszerűbben ezeket a hátsó lapokat, melyeket a rajzolás során egyszerűen figyelmen kívül hagyunk.

3.2. Festő algoritmus

A kirajzolandó lapokat súlypontjuk z koordinátája alapján rendezzük, majd hátulról előrefele sorban megrajzoljuk őket. Gyakran kombinálják a hátsó, nem látható lapok eltávolításával. Ekkor hatékony láthatósági algoritmushoz jutunk.

Alapelemként általában a háromszöget definiáljuk. Így két lap kölcsönös átfedése nem fordulhat elő. Az `SPainter` struktúra a csúcspontok indexei mellett a transzformált háromszögek súlypontjainak z koordinátáját is tárolni fogja.

```
struct SPainter
{
    int         a, b, c;
    double      weight;
};
```

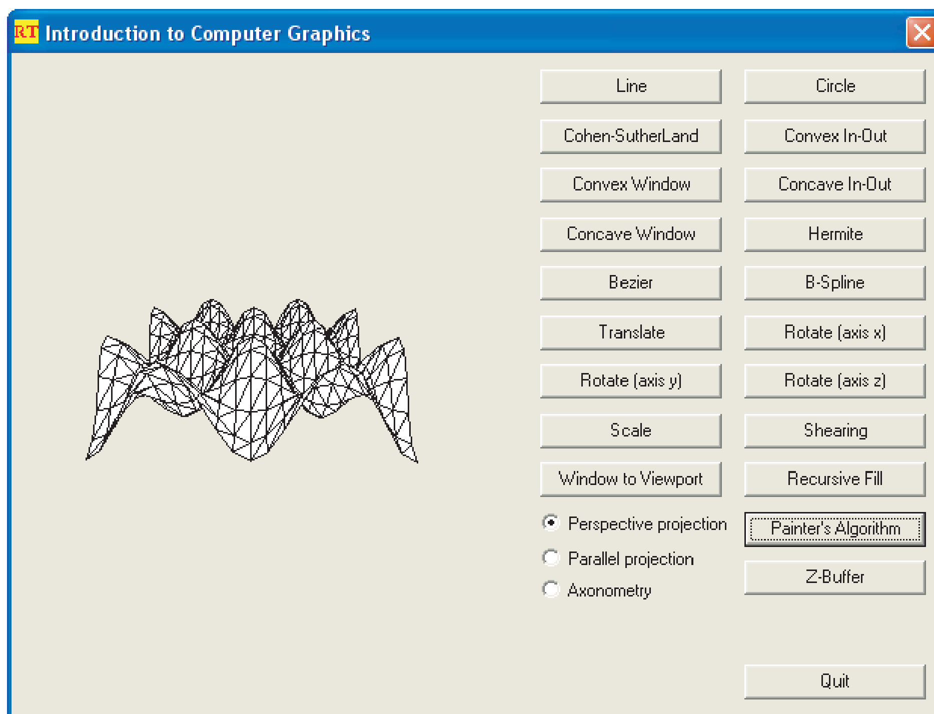
Az implementáláshoz egy speciális B-Rep osztályt érdemes definiálni. Az eredeti csúcspontok mellett azok transzformáltjait is külön fogjuk tárolni.

```
class CB_Rep_Painter
{
public:
                CB_Rep_Painter(int vertNum, int triangleNum) {
                    m_iVertNum = vertNum;
                    m_iTriangleNum = triangleNum;

                    m_Vertices = new CPoint3D[m_iVertNum];
                    m_VerticeImages = new CPoint3D[m_iVertNum];
                    m_Triangles = new SPainter[m_iTriangleNum];
                };
    virtual     ~CB_Rep_Painter() {
                    delete [] m_Vertices;
                    delete [] m_VerticeImages;
                    delete [] m_Triangles;
                };

    int         m_iVertNum;
    int         m_iTriangleNum;
    CPoint3D    *m_Vertices;
    CPoint3D    *m_VerticeImages;
```

```
    SPainter    *m_Triangles;
};
```

A rendezéshez érdemes a beépített `qsort` függvényt használni. Ez a jól ismert quick sort implementációja. Nekünk mindössze a hasonlító függvényt kell megírnunk. Használat során az első rendezés lesz a leglassabb. Mivel animáció során két kép között csekély általában a különbség, a későbbiekben egy nagyjából előrendezett tömböt kell majd ismét rendezni, ami a cserék alacsony számához vezet majd.

```
int sort_painter(const void *a, const void *b)
{
    SPainter    *p = (SPainter *)a, *q = (SPainter *)b;

    if (p->weight > q->weight) return 1;
    if (p->weight == q->weight) return 0;
//  if (p->weight < q->weight)
    return -1;
}
```



1. ábra. *Depth sorting*-al készült felület

Hátránya, hogy vannak olyan helyzetek, melyeket nem lehet vele kezelni. Ilyen például a ciklikus átfedés vagy a keskeny és hosszú lapok esete, amikor előfeldolgozási lépésre van szükség.

Mivel az alábbi kódrészlet nem kezeli az átfedéseket és a hosszúkás lapokat, igazából „Depth-sorting” algoritmusnak tekinthető.

```
void CGraphPrimitives::PainterAlgorithm(
        eBuffer buffer, CB_Rep_Painter &model)
{
    CDC     *dc = (buffer == Front ? m_cDC : &m_cBackDC);
    POINT   p[3];

    qsort((void *)model.m_Triangles, model.m_iTriangleNum,
          sizeof(SPainter), sort_painter);

    for (int i = 0; i < model.m_iTriangleNum; i++) {
        p[0].x = model.m_VerticeImages[model.m_Triangles[i].a].x;
        p[0].y = model.m_VerticeImages[model.m_Triangles[i].a].y;
        p[1].x = model.m_VerticeImages[model.m_Triangles[i].b].x;
        p[1].y = model.m_VerticeImages[model.m_Triangles[i].b].y;
        p[2].x = model.m_VerticeImages[model.m_Triangles[i].c].x;
        p[2].y = model.m_VerticeImages[model.m_Triangles[i].c].y;

        dc->Polygon(p, 3);
    }
}
```

3.3. Z-buffer algoritmus

Meglehetősen memória és számolásigényes, ennélfogva szoftveresen ritkán alkalmazzák, inkább csak a hardveres támogatás megléte mellett.

Az algoritmus élőszavas leírása:

raszter memória = háttérszín
z-buffer = $-\infty$
for minden o objektumra do
    for o objektum vetületének minden p pixelére do
        if o objektum p-ben látható pontjának
        z koordinátája $>$ zbuffer[p] then
            p színe = o színe ebben a pontban
            zbuffer[p] = o objektum p pixelben
            látható pontjának z koordinátája
        endif
    endfor
endfor

A képernyő törlését a `FillSolidRect` függvénnyel egyszerűen megtehetjük. A Z-buffert pedig legegyszerűbben egy `CMatrix` objektummal valósíthatjuk meg. Ezt az objektumot egyszerűen feltölthetjük egy megfelelően kicsi számmal, mivel a $-\infty$ értéket programozáskor nem használhatjuk.

Alapobjektumként általában a háromszöget szoktuk definiálni, mivel három nem egybeeső és nem egy egyenesen elhelyezkedő pont mindig pontosan egy síkot határoz meg. Még számolási pontatlanságok miatt sem tud megtekeredni, amire akár már egy négyszög is hajlamos. A bonyolultabb objektumok mindig felbonthatóak háromszögekre, így a függvényünket is érdemes erre az alakzatra felépíteni.

A számítások során az értékeket lineáris interpolációval határozzuk majd meg. Ez azt jelenti, hogy az objektum képe nem centrális projekcióval jött létre, hanem párhuzamos vetítéssel vagy axonometrikus leképezéssel.

Ahhoz, hogy egy háromszög pixeljeit meg tudjuk határozni, szükségünk van egy segédfüggvényre. Ez a függvény az oldalszakaszok alapján meghatározza, hogy egy vízszintes sorban hol kezdődik a háromszög határa, illetve, hogy hol végződik az. A csúcspontok z koordinátáinak lineáris interpolációjával ezekhez a végpontokhoz hozzárendelhetjük a megfelelő z értéket.

Ha a három csúcsponthoz három különböző színt rendelünk hozzá, akkor ezeket a színértékeket ugyanúgy kell interpolálni, mint a z értékeket. Mivel a fixpontos számok nem nyújtanak megfelelő pontosságot, érdemes készítenünk egy saját struktúrát, mely lebegőpontos számokkal képes kezelni a színkomponenseket. Ez lesz az `SRGB` struktúra. Használatával egy kis plusz számolás árán Gouraud árnyaláshoz is jutunk.

A függvény tulajdonképpen egy módosított szakaszrajzoló algoritmus lesz, mely pixelek rajzolása helyett az interpolált z értékeket és színeket fogja tárolni egy segédtömbben. Ezt a tömböt a Z-bufferrel együtt hozzuk létre, és majd a `Z_Triangle` tartja karban.

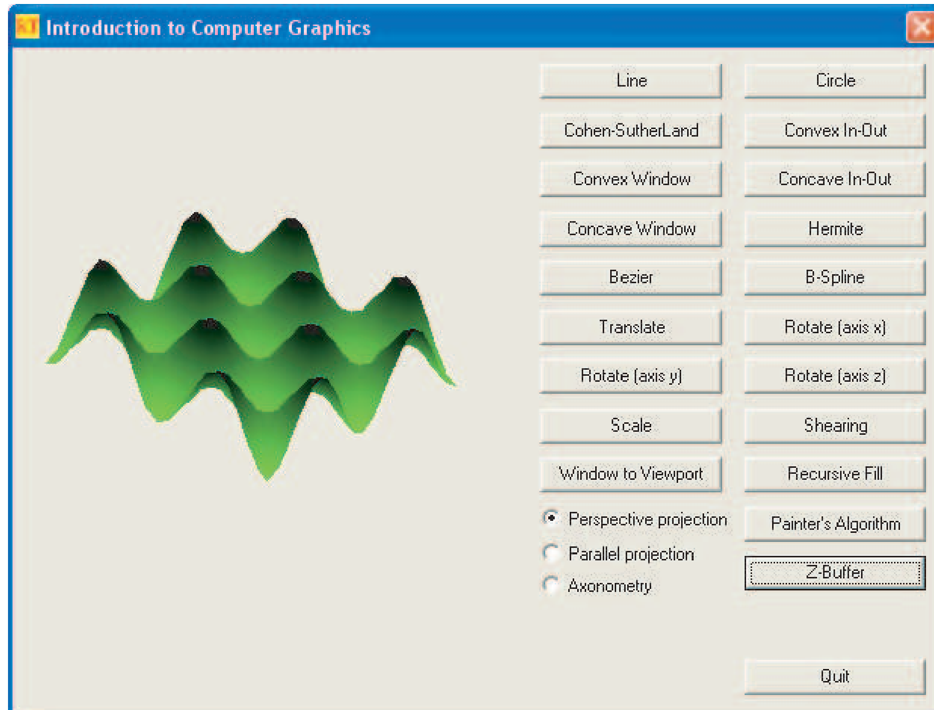
```
void CGraphPrimitives::Z_Section(
        CPoint3D &p, CPoint3D &q, COLORREF p_color, COLORREF q_color)
{
    int         i, j_int, px = p.x, py = p.y, qx = q.x, qy = q.y;
    double      j, m, z, mz;
    double      dx = qx - px, dy = qy - py, dz = q.z - p.z;
    double      dx_abs = fabs(dx), dy_abs = fabs(dy);
    SRGB        color, color_p(p_color), color_q(q_color);
    SRGB        dcolor(color_q - color_p);

    if (dx_abs > dy_abs) {
        z = p.z;
        color = color_p;

        mz = dz / dx_abs;
        dcolor /= dx_abs;
        m = dy / dx_abs;

        j = p.y; j_int = j;

        if (qx > px) {
            for (i = px; i <= qx; i++) {
                if (m_Z_Aux[j_int].left > i) {
```

```
                m_Z_Aux[j_int].left = i;
                m_Z_Aux[j_int].left_Z = z;
                m_Z_Aux[j_int].left_color = color;
            }
            if (m_Z_Aux[j_int].right < i) {
                m_Z_Aux[j_int].right = i;
                m_Z_Aux[j_int].right_Z = z;
                m_Z_Aux[j_int].right_color = color;
            }

            j += m;
            j_int = j;
            z += mz;
            color += dcolor;
        }
    } else {
        for (i = px; i >= qx; i--) {
            if (m_Z_Aux[j_int].left > i) {
                m_Z_Aux[j_int].left = i;
                m_Z_Aux[j_int].left_Z = z;
                m_Z_Aux[j_int].left_color = color;
            }
            if (m_Z_Aux[j_int].right < i) {
                m_Z_Aux[j_int].right = i;
                m_Z_Aux[j_int].right_Z = z;
                m_Z_Aux[j_int].right_color = color;
            }

            j += m;
            j_int = j;
            z += mz;
            color += dcolor;
        }
    }
} else {
    z = p.z;
    color = color_p;

    if (dy_abs != 0.0) {
        mz = dz / dy_abs;
        dcolor /= dy_abs;
        m = dx / dy_abs;
    }

    j = p.x;
```

```
        j_int = j;

        if (qy > py) {
            for (i = py; i <= qy; i++) {
                if (m_Z_Aux[i].left > j) {
                    m_Z_Aux[i].left = j;
                    m_Z_Aux[i].left_Z = z;
                    m_Z_Aux[i].left_color = color;
                }
                if (m_Z_Aux[i].right < j) {
                    m_Z_Aux[i].right = j;
                    m_Z_Aux[i].right_Z = z;
                    m_Z_Aux[i].right_color = color;
                }

                j += m;
                j_int = j;
                z += mz;
                color += dcolor;
            }
        } else {
            for (i = py; i >= qy; i--) {
                if (m_Z_Aux[i].left > j) {
                    m_Z_Aux[i].left = j;
                    m_Z_Aux[i].left_Z = z;
                    m_Z_Aux[i].left_color = color;
                }
                if (m_Z_Aux[i].right < j) {
                    m_Z_Aux[i].right = j;
                    m_Z_Aux[i].right_Z = z;
                    m_Z_Aux[i].right_color = color;
                }

                j += m;
                j_int = j;
                z += mz;
                color += dcolor;
            }
        }
    }
}
```

Egy háromszöget a `Z_Triangle` függvény képes helyesen megrajzolni. Az egyes csúcsokhoz külön egy-egy színt rendelhetünk. A háromszög határvonalának meghatározása után rendeznünk kell a három csúcspontot y koordináta szerint. A rajzolás a minimális illetve a maximális y érték között szükséges.

A feladatunk innentől mindössze annyi, hogy az adott sorban a háromszög baloldali határoló pixelétől a jobboldali határpixelig interpoláljuk a z értéket és a színt. Amennyiben a most kirajzolandó pixel közelebb van, mint a jelenleg kigyújtott, akkor kigyújtjuk azt az új színnel és tároljuk az új z értéket.



2. ábra. *Z-buffer* segítségével készült felület

A sorok végén már csak a segédtömb karbantartását kell elvégeznünk.

```
void CGraphPrimitives::Z_Triangle(
        eBuffer buffer, CPoint3D &a, CPoint3D &b, CPoint3D &c,
        COLORREF a_color, COLORREF b_color, COLORREF c_color)
{
    CDC         *dc = (buffer == Front ? m_cDC : &m_cBackDC);
    double      z, mz, width;
    int         x, y, ay = a.y, by = b.y, cy = c.y, t;
    SRGB        color, dcolor;

    Z_Section(a, b, a_color, b_color);
    Z_Section(b, c, b_color, c_color);
    Z_Section(c, a, c_color, a_color);

    if (ay > by) t = ay, ay = by, by = t;
    if (by > cy) t = by, by = cy, cy = t;
    if (ay > by) t = ay, ay = by, by = t;
```

```
    for (y = ay; y <= cy; y++) {
        mz = m_Z_Aux[y].right_Z - m_Z_Aux[y].left_Z;
        dcolor = m_Z_Aux[y].right_color - m_Z_Aux[y].left_color;

        width = m_Z_Aux[y].right - m_Z_Aux[y].left;
        if (width != 0.0) {
            mz /= width;
            dcolor /= width;
        }

        z = m_Z_Aux[y].left_Z;
        color = m_Z_Aux[y].left_color;

        for (x = m_Z_Aux[y].left; x <= m_Z_Aux[y].right; x++) {
            if (m_Z_Buffer[y][x] < z) {
                dc->SetPixel(x, y, color.GetRGB());
                m_Z_Buffer[y][x] = z;
            }
            z += mz;
            color += dcolor;
        }
        m_Z_Aux[y].left = m_iWidth;
        m_Z_Aux[y].right = -1;
    }
}
```

Szójegyzet

(1) ALU (Arithmetical Logical Unit): Aritmetikai logikai egység.
(2) CGA (Color Graphics Array): Első generációs videókártyák jelölése.
(3) CPU (Central Processing Unit): Központi feldolgozó egység.
(4) EGA (Extended Graphics Array): Második generációs videókártyák jelölése.
(5) GPU (Graphical Processing Unit): Grafikus feldolgozó egység.
(6) Pixel: négyzet vagy téglalap alakú alapegysége a képernyőnek, mely egy meghatározott színnel van kitöltve.
(7) VGA (Video Graphics Array): Második-harmadik generációs videókártyák jelölése.

Ajánlott Irodalom

(1) Dr. Juhász Imre: *Számítógépi geometria és grafika*
Miskolci egyetemi kiadó, Miskolc, 1995.

(2) Dr. Szirmay-Kalos László: *Számítógépes grafika*
Computerbooks, Budapest, 2001.

(3) James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes: *Computer Graphics: Principles and Practice in C (2nd Edition)*
Addison-Wesley Pub Co, 1995.

(4) Alan Watt: *3D Computer Graphics*
Addison-Wesley Pub Co, 1993.

(5) Hajós György: *Bevezetés a geometriába*
Tankönyvkiadó vállalat, Budapest, 1971.

(6) Reiman István: *A geometria és határterületei*
Gondolat, Budapest, 1986.