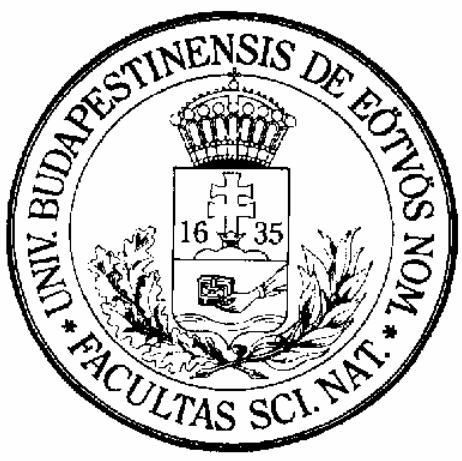


A láthatósági feladat megoldása háromdimenziós belső színtereken valósidejű alkalmazásokban

Pócza Krisztián
Eötvös Loránd Tudományegyetem, Informatikai Kar
Algoritmusok és Alkalmazásaik Tanszék

Témavezető:
Krammer Gergely

Budapest
2004

Tartalomjegyzék

1 Bevezetés

A valósidejű háromdimenziós színterek megjelenítése már hosszú idő óta fontos szerepet tölt be a számítógépes grafikában. Legfontosabb alkalmazási terület a játékok, a szimulátorok valamint a mérnöki tervezőprogramok. Megfigyelhető, hogy olyan technológiák jelennek meg egyre szélesebb körben, amelyeket eredetileg csak katonai illetve kutatási célokra alkalmaztak nagy és drága gépeken, valamint új technológiák jelennek meg az informatika minden területén, mivel most már a személyi számítógépek teljesítménye is kielégítő ekkora méretű feladatok megoldására. Ez a megfigyelés leginkább a számítógépes grafikára igaz, amely a képek számítógépes megjelenítésének eszköze. A számítógépek számítási és adatfeldolgozó teljesítménye folyamatosan nő, ezért van mód az egyre kifinomultabb, valószerűbb, részletesebb, használhatóbb illetve szórakoztatóbb grafikai megjelenítés valós időben történő elvégzésére. Gondoljunk csak arra, hogy régebben a számítógépes játékok és szimulátorok lehet, hogy gyönyörű és részletes animációkat jelenítettek meg játék közben, de maga a játék grafikai megjelenítése - mai szemmel nézve – csapnivaló volt. Az animációkat előre kiszámolták. Mostanában pedig már a játék menete közben is magasabb grafikai élménnyel szembesülhet a felhasználó, mint a régi előre kiszámolt animációk alatt.

Másrészről pedig, nemcsak a számítógépek számítási teljesítménye az, ami miatt gazdagabb grafikai élményekben lehet részünk, hanem a megjelenítés során alkalmazott kifinomultabb algoritmusok miatt is. Ezek az algoritmusok egyrészről implementálhatnak valamiféle grafikai megjelenítési módot, valamint vizsgálhatják azt is, hogy mely grafikai elemek láthatóak és melyek azok, amelyek takarásban állnak egy másik vagy több grafikai elem által. Nyilvánvaló, ha el tudjuk dönteni, hogy mely elemek álnak takarásban, valamint ezeket el tudjuk hagyni a megjelenítendő elemek sorából, akkor több idő marad a többi, látható elem megjelenítésére.

Dolgozatunkban bemutatjuk a láthatóság vizsgáló algoritmusok alapjait, főbb jellemzőik szerint csoportosítjuk őket, valamint ismertetjük, hogy napjainkban milyen algoritmusokat alkalmaznak a belső színterek látható elemeinek meghatározására.

Dolgozatunk célja az, hogy a belső terek láthatóság vizsgáló algoritmusairól egy összefoglaló képet nyújtson, amely alapján bárki, aki jártas a grafikai és geometriai programozásban, könnyen elkészíthesse ezeket az algoritmusokat.

2 A színtér fogalma és a grafikus szerelőszalag

A számítógépes grafika feladata [Szir03], hogy a felhasználó számára a modellezett világ állapotáról pillanatképeket készítsen és azt valamilyen kimeneti eszközön megjelenítse. Azt a folyamatot, amikor a valóságos vagy mesterséges világot megalkotjuk, valamilyen formában eltároljuk modellezésnek nevezzük. Azt a folyamatot, amikor a tárolt világot megjelenítjük képszintézisnek nevezzük.

A modellt létrehozhatjuk valamilyen modellező- vagy tervezőprogram segítségével, de akár valamilyen tudományos számítás során is. A modell felépítése és jellege meghatározza, hogy milyen módszerrel jelenítjük meg a képszintézis során.

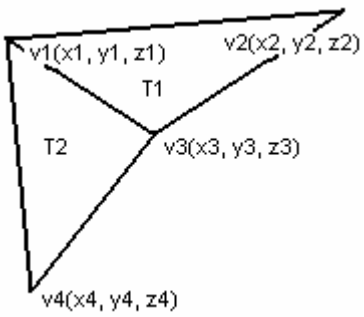
A megjelenítő eszköz leggyakrabban a számítógép monitorja, de lehet nyomtató vagy rajzgép is. Mi ebben az írásban olyan eszközt feltételezünk, amely síkban ábrázolható, vízszintesen és függőlegesen is megcímezhető pixelekből áll, amelyek színét elemi műveletekkel határozhatjuk meg. A pixel az angol „picture element” kifejezésből származik.

Ebben a fejezetben ismertetjük, hogy milyen matematikailag egyszerűen definiálható téren dolgozunk. Bemutatjuk, hogy milyen lépéseket szükséges elvégeznünk azért, hogy a bemeneti adathalmaz feldolgozása után, egy ember számára is értelmezhető képet hozzunk létre a kétdimenziós megjelenítő eszközön. Dolgozatunkban csak és kizárólag az inkrementális képszintézisről szólunk. A másik megközelítés a sugárkövető (raytracing) algoritmusok használata lenne [Szir03]. Ha sugárkövető algoritmusokat használnánk, akkor más megoldásokat is bemutatnánk, egyeseket kihagynánk, illetve az itt ismertetett megoldások egy másfajta implementációját kellene alkalmaznunk.

2.1 A színtér fogalma

A színtér (scene) az az adatszerkezet [Szir03], ahol a megjeleníteni kívánt testeket tároljuk. Ezek a testek bármilyen, a köznapi életben megszokott illetve képzeletbeli test virtuális leírásai lehetnek. Egy test felületi elemekből áll, amelyek általában kiterjedésükben korlátozott síklapok illetve görbe felületek lehetnek. Mi most csak a síklapokkal foglalkozunk, mivel a görbe felületek könnyen általánosíthatók síklapok összetételeként valamint a grafikus megjelenítő motor is így jeleníti meg őket. A síklapok leggyakrabban háromszögek, de több oldalból is állhatnak. A grafikus megjelenítő motor ezeket a több mint három oldalból álló sokszögeket háromszögekre bontja, és úgy adja át a konkrét megjelenítést végző egységnek.

A háromszögeket matematikailag a három csúcspont (vertex) háromdimenziós Euklideszi koordinátáival definiáljuk (X, Y, Z). Ha ilyen háromszögeket szorosan egymás mellé helyezünk, akkor egy virtuális testet kapunk. A szorosan egymás mellett elhelyezkedő háromszögeknek vannak közös csúcsai és élei. Tehát, elég azt letárolnunk, hogy milyen csúcsokból áll a testünk, valamint azt, hogy a háromszögeink a test melyik csúcsain fekszenek.



Háromszögek

Az ábrán látható két háromszög (T1 és T2). A T1 csúcsai a (v1, v2, v3), a T2-é pedig a (v1, v3, v4) hármassal írhatók le.

A színtereknek két különböző típusa létezik, a külső és a belső. A külső színterek mindig valamilyen nyílt teret, domborzati elemeket, természeti tájat ábrázolnak, általában szimulátorokban, tudományos számításokban jelennek meg. A belső színterek zárt épületet vagy várost jelenítenek meg. A belső színterek szektorokra (általában szobák) bonthatók, amelyek között átlátható részek (általában ajtók, átjárók) találhatók. Ez a megkülönböztetés a rajtuk alkalmazható láthatósági algoritmusok alapján alakult ki.

2.2 A grafikus szerelőszalag felépítése és működése

A grafikus szerelőszalag (graphics pipeline) feladata [Abr97, Szir03], hogy a virtuális kameráink nézőpontjából egy kétdimenziós képet alkosson a háromdimenziós térről. Működését több, szigorúan egymás után következő műveletre bonthatjuk. Az egyes műveletek kimenete bemenete a sorrendben őt követő műveletnek. Az általunk ismertetett sorrend nem kötelező, grafikus motoronként és implementációnként változhat. A lépések felcserélhetők és összevonhatók. A grafikus szerelőszalag annál hatékonyabb, minél korábbi lépésben sikerül a látótéren kívül eső illetve takart objektumokat kiszűrnünk.

A grafikus szerelőszalag a következő műveletekből áll:

1. Képelemek összeállítása
2. Modell transzformáció
3. Látótéren kívül eső térrészek elhagyása
4. Megvilágítási modell alkalmazása
5. Triviális hátsólap eldobás
6. Nézőponti transzformáció
7. Láthatóság-vizsgálat – takart objektumok elhagyása
8. Vetítés a képtérre
9. Raszter-műveletek
10. Megjelenítés

Az első lépésben, meghatározzuk, hogy milyen testekkel szeretnénk dolgozni, milyen testek találhatók a színtérben. Mint említettük, ezeket a testeket a csúcspontjaik háromdimenziós koordinátáival valamint a határoló síkok leírásával definiáljuk. Ezeket a csúcspontokat a test saját-koordináta rendszerében adjuk meg (object-space). Úgy definiáljuk ezeket a csúcspontokat, hogy (általában a test belsejében) kiválasztunk egy kitüntetett pontot, az origót, és a koordinátákat ehhez képest adjuk meg. Minden testhez felveszünk egy forgatási mátrixot, amely a test állását (orientációját) határozza meg, valamint egy eltolási vektort is. A mátrix kezdetben egy 3x3-as egységmátrix szokott lenni. Ennek a két algebrai struktúrának a használatáról a későbbiekben bővebben szót ejtünk.

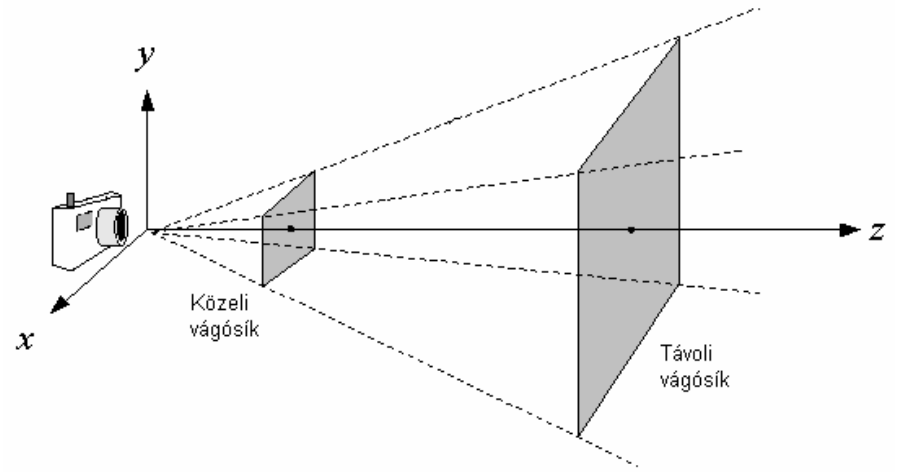
A második lépés a modell transzformáció. Láttuk, hogy minden test rendelkezik egy forgatási mátrixszal és egy eltolási vektorral. Célunk az, hogy egy közös, a háromdimenziós világunk koordináta rendszerébe helyezzünk át minden testet. A forgatási mátrix a test saját-koordináta rendszerében történő forgatást írja

le, tehát először minden csúcspontot meg kell szoroznunk ezzel a mátrixszal, majd pedig az eltolási vektor segítségével át kell transzformálnunk a világ-koordinátarendszerünk (world-space) origójához képest minden egyes csúcspontot. Tehát a képlet:

$$W_i := M * V_i + T \ .$$

M jelöli a 3x3-as forgatási mátrixot, V a test saját-koordináta rendszerében adott háromdimenziós koordinátát, T a világ koordinátarendszer origójához képest adott eltolását a test origójának, W pedig a világ koordinátarendszerbe transzformált csúcspontot.

A harmadik lépés a látótéren kívül eső térrészek elhagyása (frustum culling). A látótér egy olyan csonka gúlának tekinthető, amelynek levágtuk a tetejét esetleg az alját a csonka gúla középvonalára merőleges síkokkal. A csonka gúlához definiálható hat határoló sík, amelyeket vágósíkoknak nevezünk. Először meghatározzuk ezeket a vágósíkokat.



Látótér

A síkok meghatározásához a következő három tulajdonság ismeretére van szükségünk:

1. kamera világ-koordinátarendszerbeli helye,
2. az a világ-koordinátarendszerben adott pont, ahova a kamera néz,
3. a kamera látószöge.

A közeli és a távoli vágósík merőleges a kamera irányára, a látótér oldalsó vágósíkjai pedig a látószög felével „hajlanak” el „fel” illetve „le”. A vágósíkok

kiszámítása egyszer történik meg képkockánként, mivel minden test és sokszög esetében ugyanezeket a síkokat használjuk a vágásokkor.

Kiszámítása:

$$sh = \sin(FOVH/2)$$
$$sv = \sin(FOVV/2)$$
$$ch = \cos(FOVH/2)$$
$$cv = \cos(FOVV/2)$$
$$frustum[0].N = (ch,0,sh),\ frustum[0].D = -1$$
$$frustum[1].N = (-ch,0,sh),\ frustum[1].D = -1$$
$$frustum[2].N = (0,cv,sv),\ frustum[2].D = -1$$
$$frustum[3].N = (0,-cv,sv),\ frustum[3].D = -1$$
$$frustum[4].N = (0,0,1),\ frustum[4].D = Znear$$
$$frustum[5].N = (0,0,-1),\ frustum[5].D = -ZFar$$

Ahol a FOVH (Horizontal Field of View) és a FOVV (Vertical Field of View) a vízszintes és a függőleges látószögek, a frustum jelöli a hat vágósíkot található. A 0 indexű a bal, az 1 indexű jobb, a 2 indexű az alsó, a 3 indexű a felső, a 4 indexű a közeli az 5 indexű a távoli vágósík. A Znear és a Zfar pedig a közeli és a távoli vágósíkok távolsága a nézőponttól.

Ezek után megvizsgáljuk, hogy mely testek találhatók teljes egészében a látótérben, melyek azok, amelyeket egy vagy több vágósík elmetsz, melyek azok, amelyek teljesen kilógnak a látótérből. A legelső esetben felvesszük a testet a kirajzolandók listájába, a második esetben elvágjuk a metsző vágósíkokkal, majd a fennmaradó részt tesszük csak be a kirajzolandók listájába. Ha a test teljes egészében kilóg, akkor elhagyjuk.

Negyedik lépésben a kirajzolandó testekre alkalmazzuk a megvilágítási modellünket. Ehhez figyelembe vesszük a látótérben és a látótéren kívül található fényforrásokat is. A testeken található fény lehet előre kiszámított vagy valós időben kiszámított.

Az előre kiszámított fényértékeket (statikus fényforrásokból) meghatározhatjuk a csúcspontokban, illetve a pontosabb eredmény érdekében eltárolhatjuk őket fénytérképekben (lightmap). A fénytérképeknek az előnyük az, hogy mivel előre kiszámítjuk őket, több időt szánhatunk a kiszámítási feladatra, így pontosabb eredmény kapunk, és az árnyékokat is részletesebben megjeleníthetjük majd. A fénytérképek létrehozásánál leggyakrabban alkalmazott algoritmus a Radiosity [AWPH97], amely azon a fizikai jelenségen alapszik, hogy minden fényforrás fényenergiát sugároz, valamint minden felület sugároz valamiféle

energiát, de leginkább visszaveri és elnyeli a fényforrásokból illetve a másik felületről érkező visszavert energiát. Ezek a fizikai tulajdonságok a felülethez rendelt anyagtól (material) függnek. Az algoritmus minden egyes lépésében a legtöbb energiával rendelkező energiaforrást tekinti, és kiszámítja, hogy a felületekre mennyi energia érkezik innen. Ezt addig végzi, amíg az így továbbítható energia valamilyen előre meghatározott minimális szintre nem csökken.

A dinamikus fényforrások általában kisebb hatótávolságúak, nem annyira pontos eredményt adnak, mert minden egyes megjelenített képkockánál újra kell számítanunk őket, előnyük, hogy dinamikusan változó fényviszonyokat és árnyékokat kitűnően megjeleníthetünk segítségükkel.

Ötödik lépés a triviális hátsólap eldobás. Ezzel a lépéssel fennmaradó sokszögek (háromszögek) körülbelül felét eldobhatjuk. Ehhez minden háromszöget csúcspontjainak sorrendjét azonos módon, óramutató járásával megegyező vagy ellentétes irányban kell definiálnunk. A háromszögek normálvektorát minden esetben azonos módon kiszámítjuk a háromszög két oldalvektorának vektoriális szorzatával, majd ezt a vektoriális szorzatot skalárisan megszorozzuk a kamera irányvektorának inverzével. Valamint az előbb kiszámított normálvektort skalárisan megszorozzuk a háromszög bármelyik csúcspontjának koordinátáival. Ha az első érték kisebb, mint a második, akkor a háromszöget eldobhatjuk. Tehát a képlet:

$$N = (v2 - v1) \times (v3 - v1)$$
$$Backface = N * ICAM < N * v1$$

Ahol a v1, v2, v3 jelölik a háromszög csúcspontjainak koordinátáit, N a normálvektort, ICAM pedig a kamera irányvektorának inverzét. A Backface egy logikai érték lesz, ami ha igaz, akkor eldobhatjuk a háromszöget. Minden vektor világ-koordináta rendszerben adott.

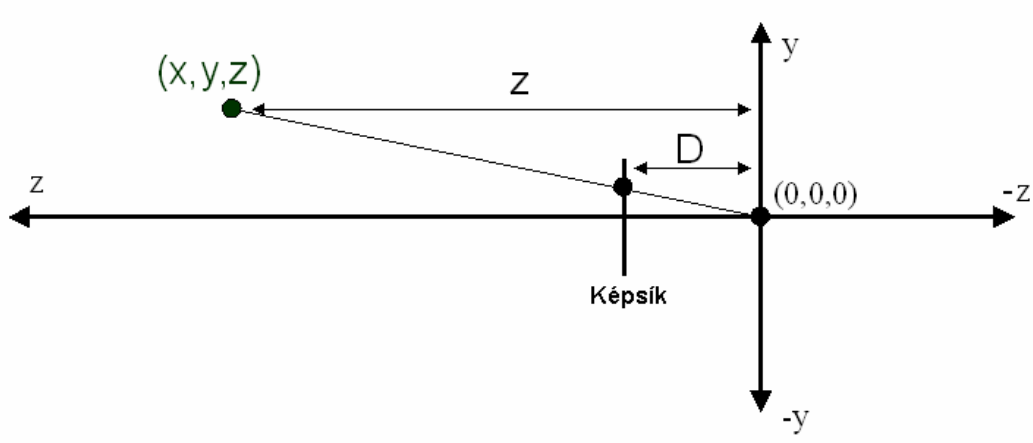
Hatodik lépés a nézőponti transzformáció, amely a világ-koordináta rendszerből a kamera koordináta rendszerébe (camera-space) transzformálja a testeket. Mint említettük, a kamera elhelyezkedése és az a pont, ahova néz világ-koordináta rendszerben adottak. Ezen két adat, valamint rögzített irányok alapján meghatározhatjuk azt a mátrixot, amely alapján a látható testeket elforgathatjuk. Majd a kamera terébe transzformáláshoz kivonjuk a kamera világ-koordináta rendszerbeli elhelyezkedését. Tehát a képletünk:

$$C_i := M * W_i - O \ .$$

M jelöli a 3x3-as forgatási mátrixot, W a test világ-koordináta rendszerében adott csúcspont koordinátáját, O a kamera világ-koordinátarendszerbeli origóját, C pedig a kamera koordinátarendszerbe transzformált csúcspontot.

A hetedik lépés a láthatóság-vizsgálat (Occlusion culling). A látótérből kilógó testeken kívül még olyan nem látható testek is jelen lehetnek az eddig felépített világunkban, amelyeket egy vagy több másik test eltakar. Ebben a lépésben ezeket a testeket próbáljuk meg minél hatékonyabban eltávolítani a kirajzolandó testek listájából. A következő fejezetekben erről szólunk részletesen.

A nyolcadik lépésben újabb koordináta transzformációt kell végrehajtanunk. Áttérünk a kamera teréből a képtérre (screen-space), melyet vetítésnek nevezünk. Végső soron kiszámítjuk azt a képsíkbeli kétdimenziós pontot, amelyet az a szakasz döf át, amely az origóból a kamera-koordináta rendszerbeli háromdimenziós pontba mutat. A következő ábrán a (0,0,0) pontban található a nézőpont, az (x, y, z) pontban található kamera-koordináta rendszerbeli pontot vetítjük le a képsíkra. A nézőpont és a képsík távolsága: D. Cél az (x, y, z) háromdimenziós koordinátákból az (X2D, Y2D) képtérbeli kétdimenziós koordináták kiszámítása. Természetesen a képernyő középpontjához képest kell ezt megtennünk, amelyet a (Xc, Yc)-vel jelölünk.



Vetítés

A képlet:

$$X2D = X * D / Z + Xc$$
$$Y2D = Y * D / Z + Yc$$

Kilencedik lépés a raszter-műveletek végrehajtása. A megvilágítási modellünk alapján kiszámítottuk a csúcspontjaink megvilágítási értékeket. Tudjuk, milyen fénytérképeket kell a felületekre helyezni, valamint ismerjük, hogy milyen anyag (material) található az adott síklapokon. A raszter-műveletek során kitöltjük a

háromszögek belső részét a fénytérképek, a csúcspontok fényértékei valamint a síklap anyagának figyelembevételével. Az anyag határozza meg a kitöltő textúrát (kétdimenziós bittérkép, amelyet ráfeszítünk a háromszögre) valamint a felület fizikai tulajdonságait (fényvisszaverő és elnyelő képesség, domborulatok).

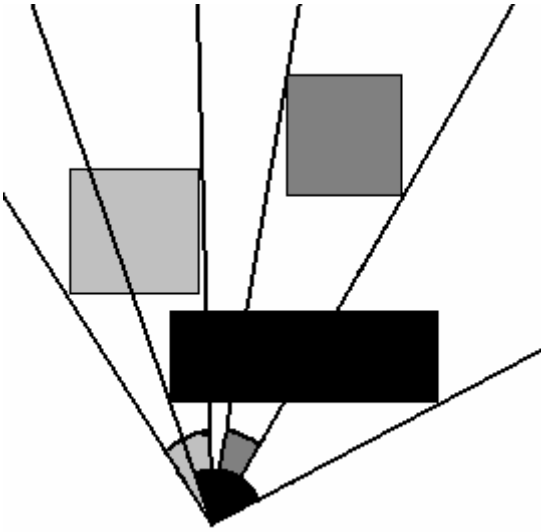
Tizedik, egyben utolsó lépés a megjelenítés. Mindig legalább kettő képernyő méretű pufferrel dolgozunk. Az egyik, amelyik a képernyőn megjelenő képet tárolja (front buffer), a másik az, amelyen dolgozunk, ahova a raszter-műveletek során az eredményt kirajzoljuk (back buffer). A grafikus szerelőszalag minden egyes végrehajtása után ennek a két puffernek a szerepe felcserélődik, így kerülhetjük el a felhasználó számára idegesítő felülrajzolás (overdraw) és villogás (flickering) látványát.

A fentiektől teljesen eltérő módon működő sugárkövető algoritmusok:

```
for each pixel p do
begin
      r = ray from the eye through p
      visible object = null
      for each object o do
      begin
            if r intersects o then
                  if intersection point is closer than previous ones then
                        visible object = o
      end
      if visible object <> null then
            p.color = color of visible object at intersection point
      else
            p.color = background color;
end
```

3 A láthatóság-vizsgálat szerepe és feladata

Az előző fejezetben ismertettük a grafikus szerelőszalag működését, ahol a hetedik lépésben a láthatóság-vizsgálatot [HZh98, MHAV] hajtottuk végre. Ennek a lépésnek tehát az volt a feladata, hogy azokat a testeket határozza meg és távolítja el, amelyeket egy vagy több másik test eltakar. Az eltakart testeket occludee-nek, az ezeket a testeket eltakaró testeket pedig occluder-nek nevezzük.



Takarások

Az ábrán látható három test közül a fekete teljes egészében látszik, nem takarja el semmi, a sötét szürkét teljesen eltakarja a fekete a világos szürkét félig. A sötét szürkét biztosan nem kell kirajzolni, a világos szürkénél két eset lehetséges: kirajzolhatjuk az égész testet, mivel esetenként időigényes lehet annak meghatározása, hogy melyik részei láthatóak. Ha ennek meghatározása nem igényel sok időt, akkor csak a látható részeit rajzoljuk ki, mivel a kirajzolás időigényes. Természetesen, ha az egész testet kirajzoljuk, akkor a későbbiekben ismertetésre kerülő Z-buffer algoritmust is alkalmaznunk kell.

Ezeknél a takarási viszonyoknál bonyolultabb esetek is létrejöhetnek, pl. amikor két takaró együttesen fed el egy másik objektumot, ezt occluder fusion-nak nevezzük. A takarásvizsgálat történhet két, illetve három dimenzióban is, így a grafikus szerelőszalag más lépésében helyet is foglalhat.

Vannak olyan láthatósági algoritmusok, amelyek külső és belső színtéren is alkalmazhatók, vannak olyanok, amelyek speciálisan csak az egyiken vagy a másikon működnek, így a következőkben e csoportosítás szerint mutatjuk be őket.

A láthatósági algoritmusokat absztrakt szinten definiáljuk először, a konkrét algoritmusok ennek megvalósításai. A bemeneti testek halmaza legyen O, a takarási viszonyok aktuális állapotát a H absztrakt változó jelzi. Ezek alapján az algoritmusunk a következő módon írható le:

```
OcclusionCullingAlgorithm(ObjectSet O, ActualOccState H)
begin
      InitAndClear(H)

      for each object o chosen front-to-back from O
      begin
            if FullyOccluded(o, H) then
                  Skip(o)
            else if PartiallyOccluded(o, H) then
            begin
                  PartiallyRender(o)
                  Update(o, H)
            end else
            begin
                  Render(o)
                  Update(o, H)
            end
      end
end
```

Első lépésben inicializáljuk a takarásokat tartalmazó absztrakt változót, majd végigmegyünk az összes testen. Ha egy test teljesen takarásban van, akkor egyszerűen kihagyjuk, ha részlegesen takart, akkor a látható részét kirajzoljuk, majd frissítjük a H absztrakt változót. Ha teljesen látszik, akkor az egész test kirajzolása után frissítjük az absztrakt változót.

3.1 Általános láthatóság vizsgáló algoritmusok

Z-buffer algoritmus [Abr97, MHAV, Szir03] a takarási feladatot a pixelek szintjén oldja meg, vagyis egzaktul oldja meg a láthatósági feladatot, becslések nélkül. De mivel kis egységeken dolgozik, ezért alacsony hatékonyságú. A következőkben ismertetett algoritmus az alapalgoritmus, léteznek gyorsításai és javításai. A Z-buffer algoritmus láthatósági algoritmus ugyan, de a grafikus szerelőszalag egy késői fázisába, a mi definíciónk szerinti kilencedik lépésbe csúszott, így még inkább látható, hogy alacsony hatékonyságú. A modern 3D grafikus gyorsító kártyák implementálják ezt az algoritmust.

Első lépésben létrehozunk egy buffer-t, amelynek dimenziója megegyezik a képernyő felbontásával, majd feltöltjük egy maximális távolságértékkel. Ezután végighaladunk az összes test összes lapján, és a 2D-s sokszögkitöltő algoritmus futása során nemcsak a színértékeket határozzuk meg, hanem karbantartjuk az éppen számított pixel távolságértékét is. Ha a Z-bufferben található távolságérték nagyobb, mint amit az aktuális pixel feldolgozása közben kiszámítottunk, akkor kirajzoljuk a pixel-t és a Z-bufferben található távolságértéket erre az új értékre módosítjuk. Ha a Z-bufferben található érték kisebb, mint az aktuális számított távolságérték, akkor ez azt jelenti, hogy a képernyőn már egy közelebbi pixel látszik, mint amit most számítunk. Tehát ilyenkor nem kell semmi tennünk. A következő pseudo-kód mutatja az implementációt:

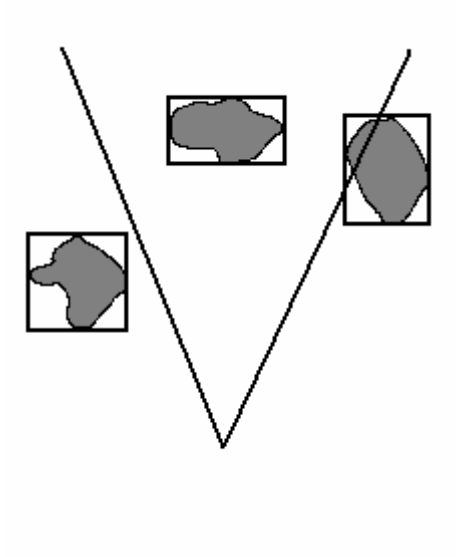
```
Z-buffer.FillWith(MAXIMUM_Z)

for each object o begin
begin
      for each pixel p covered by projected surface of o
      begin
            if p.z < zbuffer[p] then
            begin
                  screen[p] = p.color
                  zbuffer[p] = p.z
            end
      end
end
```

A triviális hátsólap eldobást (backface culling) és a látótéren kívül eső térrészek elhagyását (frustum culling) már említettük az előzőekben, a grafikus szerelőszalag bemutatását tartalmazó részben.

A befoglaló objektumok (bounding volumes) használata igen elterjedt, gyakran alkalmazzuk valamely másik láthatósági algoritmussal együtt, így sokszor a

frustum culling-al is. Ha csak a frustum culling-al dolgoznánk, akkor minden test minden síklapjára külön meg kellene vizsgálnunk, hogy a látótérben van-e vagy nem. Ha egy olyan objektumról, ami teljesen magában foglalja az egész testet, el tudjuk dönteni, hogy teljesen kívül esik-e a látótéren, akkor az egész test is kívül esik, nem kell vele a későbbiekben foglalkozni. Ha az egész befoglaló objektum a látótéren belül van (azaz nem vágja el egyik határoló sík sem), akkor az egész test is a látótérben van. Ha a befoglaló objektumot valamelyik határoló sík elmetszi, akkor a síklapok szintjén is meg kell vizsgálnunk a látótérhez vett viszonyokat.

A befoglaló objektum általában téglatest vagy gömb szokott lenni. A következő ábra mutatja az előbb vázolt viszonyokat.



Befoglaló objektumok

Csak megemlítjük az egyik legfontosabb takarásvizsgáló algoritmust, a HOM-ot (Hierarchical Occlusion Maps), mivel ennek ismertetése nem célja ennek az írásnak. Hanson Zhang alkotta [HZh98], a modern háromdimenziós gyorsító kártyák nagy részében ennek egy változatát használják is már.

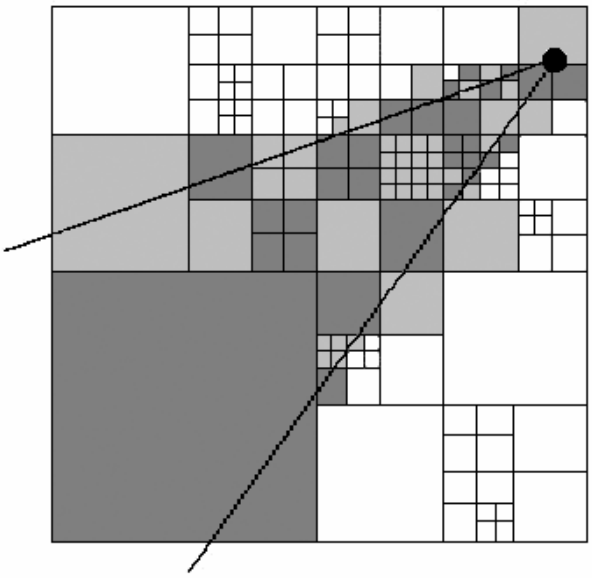
3.2 Külső színterek láthatóság vizsgáló algoritmusai

A négyesfa (Quadtree) és a nyolcasfa (Octree) [MHAV] algoritmus a két legfontosabb eljárás, amelyekkel külső színterek láthatóság-vizsgálatát megvalósíthatjuk.

A négyesfa algoritmus előfeldolgozó eljárásában a teret az Y tengellyel párhuzamos (az X-Z síkra merőleges), egymásra merőleges síkokkal osztjuk két részre. Első lépésben a teret négy egyenlő részre osztjuk a két egymásra merőleges sík felhasználásával. Az így kialakuló negyedeket tovább bontjuk további négy egyenlő részre hasonló orientációjú síkokkal. Ezt minden negyed minden negyedére rekurzívan végrehajtjuk. Addig fut az algoritmus, amíg megfelelően kevés számú test illetve háromszög lesz található a térrészekben, illetve megfelelően kicsi térrészekhez jutottunk.

```
Quad::QuarterSpaceRecursively(level)
begin
      if level = DesiredMaximumLevel or
            NumberofObjects <= DesiredMinimumObjects then return

      CreateSubspaces(quad1, quad2, quad3, quad4)

      quad1.QuarterSpaceRecursively(level+1)
      quad2.QuarterSpaceRecursively(level+1)
      quad3.QuarterSpaceRecursively(level+1)
      quad4.QuarterSpaceRecursively(level+1)
End
```

A CreateSubspaces eljárás létrehozza a negyedeket, amelyekben elhelyezi a hozzájuk tartozó testeket. Ha egy test több térrészhez is hozzátartozik, akkor el kell



Négyesfa

vágnunk. Az algoritmus hátránya, hogy nem törődik azzal, hogy minden térrészben közel azonos számú test kerüljön.

Az ábrán felülnézetből látható egy felosztott tér. A sötét pont jelöli a nézőpontot, a két szakasz, pedig a látóteret határozza meg. A szürke árnyalatokkal ábrázolt négyzetek jelölik azokat a térrészeket, amelyekhez tartozó objektumok beleesnek a látótérbe, így ezeket ki kell rajzolnunk. Látható, hogy a négyesfa magas szinten – térrészek, testcsoportok szintjén - dolgozik, ezáltal egy lépésben több testről dönti el, hogy ki kell-e rajzolni. Ez az eldöntő algoritmus szintén rekurzívan dolgozik a generáló algoritmushoz hasonlóan. Bejárja a fát, és meghatározza, hogy az éppen vizsgált negyed kívül vagy belül esik a látótéren. Ha kívül esik, akkor ezzel a térrésszel és a benne levő további kisebb térrészekkel nem kell foglalkoznunk. A látótérbe eső negyedeken pedig rekurzívan tovább kell alkalmaznunk az algoritmust.

```
Quad::RenderSpaceRecursively
begin
      if hasRenderableObjects then
      begin
            RenderObjects()
      end
      if hasQuads
      begin
            ExtractSubspaces(quad1, quad2, quad3, quad4)
            if quad1.IsInViewingFrustum then
                              quad1.RenderSpaceRecursively
            If quad2.IsInViewingFrustum then
                              quad2.RenderSpaceRecursively
            if quad3.IsInViewingFrustum then
                              quad3.RenderSpaceRecursively
            if quad4.IsInViewingFrustum then
                              quad4.RenderSpaceRecursively
      end
end
```

Ennek az algoritmusnak egy továbbfejlesztése a nyolcasfa (Octree) algoritmus, amely csak annyiban különbözik, hogy négy helyett nyolc egyenlő részre osztja a teret, tehát generáló és kirajzoló algoritmusa megegyezik.

Látható, hogy a négyesfa és a nyolcasfa algoritmus a láthatótérbe eső testek meghatározását segíti. A négyesfát akkor alkalmazzuk, amikor a tér viszonylag sík, pl. katonai szimulátor programokban, a nyolcasfát pedig olyan esetekben, amikor a tér mindhárom dimenzióban közel azonos módon kiterjedt. Mindkét algoritmus komplexitása logaritmikus.

3.3 Belső színterek láthatóság vizsgáló algoritmusai

Dolgozatunk következő három fejezetében erről a témáról lesz szó. Tárgyaljuk a BSP-fa (Binary Space Partitioning Tree) algoritmust, a Portal-okat valamint a PVS-t (Potentially Visible Set).

4 A BSP-fa

A BSP-fa (Binary Space Partitioning Tree) [Abr97, MDB00, MHAV, SRE01] fogalmát 1980 körül Fuchs, Kedem és Naylor vezette be. Két cikket írtak a témában: „Predetermining Visibility Priority in 3D Scenes” és „On Visible Surface Generation by A Priori Tree Structure”.

A BSP-fákat a ’80-as évek végén, ‘90-es évek elején arra használták, hogy belső terekben a sokszögeket közelről-távolra illetve távolról-közelre sorrendben rajzolják ki belső terekben, ezáltal elkerülhették a Z-buffer algoritmus alkalmazását. Mondhatja az olvasó, hogy ez a probléma megoldható úgy is, hogy a sokszögeket a nézőponttól való távolságuk illetve a súlypontjuk Z koordinátája szerint csökkenő sorrendbe rendezzük, és így megkapjuk a kívánt eredményt.

Ezzel az a probléma, hogy ha két háromszög metszi egymást, akkor nem lesz jó a sorba rendezés semmilyen algoritmus szerint sem. Megoldás az, hogy az ilyen háromszögeket elmetsszük. A súlyosabb probléma azonban az, hogy a háromszögek sorba rendezése időigényes feladat, még akkor is, ha a gyorsrendezést (Quick sort) [CRL99] használjuk, mivel annak a bonyolultsága optimális esetben $O(n*\log(n))$, legrosszabb esetben pedig négyzetes, tehát ha szerencsénk van, akkor is $c*n*\log(n)$ idő szükséges a sorba rendezési feladat megoldásához, ahol c konstans.

A BSP-fa algoritmus használatával ez elkerülhető. A BSP-fa megoldja a rendezési feladatot úgy, hogy a testeket illetve sokszögeket olyan fa-struktúrába [CRL99] helyezhetjük az előfeldolgozás során, amelynek bejárása során távolról-közelre, illetve közelről-távolra sorrendben rajzolhatjuk ki a teret, és még a látótérből kilógó térrészek eltávolításánál is segítségünkre van.

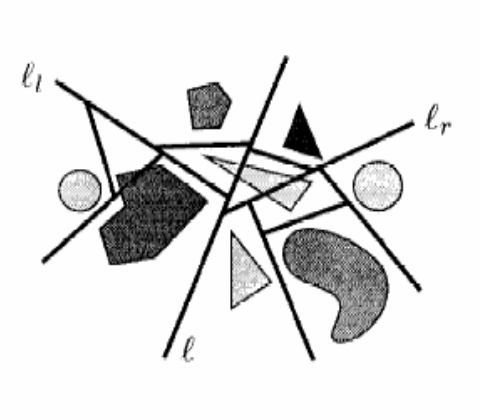
A BSP-fa algoritmus segítségünkre lehet az ütközések vizsgálatánál (collision detection), mivel $O(\log(n))$ idő alatt meghatározhatjuk, hogy melyik térrészben vagyunk, és így csak ezekre a sokszögekre kell az ütközés-vizsgálatot elvégezni.

Az algoritmus hátránya azonban az, hogy csak statikus teret tudunk segítségével így eltárolni. Tekintsünk egy tipikus belső teret! Ez általában egy épület vagy város. A falak, ajtók, ablakok és berendezési tárgyak nem mozognak, ezeket gond nélkül behelyezhetjük a BSP-fába. A dinamikus testeket a Z-buffer algoritmus segítségével rajzolhatjuk ki.

A következőkben háromszögekkel dolgozunk, azonban az algoritmus könnyű szerrel kiterjeszthető magasabb oldalszámú sokszögekre is.

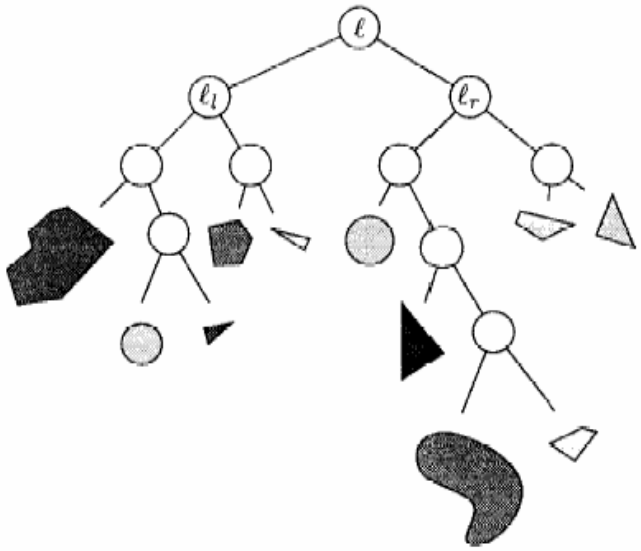
4.1 Működési elve

A BSP-fa egyszerűen fogalmazva a háromdimenziós tér egy hierarchikus felosztása konvex térrészekre. Valamint, a BSP-fa egy bináris fa, amelynek van egy gyökér eleme, amelyből csúcsok ágazhatnak ki balra és jobbra. Minden csúcsban található egy térosztó hipersík, amely a teret minden esetben két részre bontja. A hipersík mindig az általa két részre bontott tér dimenziójánál egyel kevesebb dimenziójú struktúra, tehát három dimenzióban egy sík, két dimenzióban egy egyenes, négy dimenzióban egy háromdimenziós tér, azonban ez utóbbi számunkra csak elméleti jelentőségű. A hipersík mindkét oldalán újabb két csúcs (node),



Felosztás

esetleg levél (leaf) található. A csúcsokban újabb hipersík van, és még két csúcsra illetve levélre mutat. A levél abban különbözik a csúcstól, hogy ott már nincs hipersík, nincsenek gyerekei, csak sokszögeket tartalmaz. A csúcsokban ezzel szemben nincsenek sokszögek.



BSP-fa

4.2 Generálása

Ebben a részben ismertetjük, hogy a BSP-fa generálása milyen algoritmus szerint történik. Alapvetően a következő műveleteket hajtjuk végre:

1. Egy hipersík kiválasztása, amely két részre osztja a teret
2. A hipersíkot tartalmazó térrész összes háromszögét a hipersík „előtti” illetve „mögötti” térrészbe soroljuk, ez alapján a fa bal illetve jobb ágába kerül
3. Rekurzívan belépünk a 2. lépésben keletkezett mindkét ágba, és addig hajtjuk végre az algoritmust az első lépéstől fogva, amíg minden térrészben teljesül egy előre meghatározott feltétel (később ismertetjük)

Tehát az első lépés a hipersík kiválasztása. A célunk az, hogy olyan BSP-fát hozzunk létre, amely optimális, tehát minden ága közel azonos számú leszármazottat tartalmazzon, valamint közel azonos számú sokszög legyen minden levelében. Hátrányos az is, ha a hipersík sok sokszöget metsz el. A kiválasztott hipersíkot általában úgy határozzuk meg, hogy a térrész összes sokszögén végighaladunk, és egy rá illeszkedő síkot hozunk létre. Azt a síkot választjuk ki, amelynek súlyfüggvénye a legkisebb. A súlyfüggvény értéke függ a sík két oldalán található sokszögek számának abszolút értékben vett különbségétől, valamint az elmetszett sokszögek számától. Az elmetszett sokszögeket nem mindig vesszük figyelembe.

Négy eset lehetséges, amikor egy sokszöget egy hipersíkkal particionálunk:

1. A sokszög teljesen a hipersík előtt helyezkedik el
2. Teljesen a hipersík mögött helyezkedik el
3. A hipersíkon helyezkedik el
4. A hipersík és a sokszög metszi egymást

Amikor azt mondjuk, hogy egy sokszög a hipersík előtt helyezkedik el, akkor arra gondolunk, hogy mindegyik csúcspontja a hipersík azon oldalán található, amerre a hipersík normálvektora mutat.

Megszámoljuk, hogy melyik esetbe hány sokszög tartozik. Az első és a második eset kezelése triviális: minden esetben a megfelelő oldalhoz számítjuk hozzá a sokszöget. A harmadik esetben két megoldási lehetőség kínálkozik: mindkét oldalhoz hozzászámítjuk a sokszöget vagy pedig csak az egyikhez. Mi a második esetet választottuk, a hipersík előtt található sokszögekhez számoltuk

hozzá a hipersíkon elhelyezkedő sokszögeket. A negyedik esetben mindét oldalhoz hozzá szoktuk adni az elmetszett háromszöget.

Tehát keletkezett két érték, az egyik azoknak a sokszögek számát határozza meg, amelyek a hipersík előtt találhatók (illetve oda számoltuk), a másik azon sokszögek számát határozza meg, amelyek a hipersík mögött találhatók (illetve oda számoltuk). A cél e két érték közötti különbség abszolút értékének minimalizálása (ez lesz a súlyfüggvény) és az ehhez a minimumhoz tartozó hipersík meghatározása. Több hipersík is kaphatja ugyanazt a súlyfüggvényt, ebben az esetben egyet kiválasztunk közülük, nem alkalmazunk visszalépéses keresést. Az imént ismertetett algoritmus pszeudó kódja:

```
Plane BSPGenerator::FindBestDivisorPlane()
begin
      count = 0
      absdifference = DIFF_MAX
      bestplane = NULL

      for i = 1 to numpolygons do
      begin
            numfront = 0
            numback = 0

            pln = poly[i].GetPlane()

            for j = 1 to numpolygons do
            begin
                  if i = j continue

                  res = pln.GetOrientationOf(poly[j])

                  if res = SIDE_FRONT or res = SIDE_ONPLANE
                        numfront++
                  else
                  if res = SIDE_BACK
                        numback++
                  else
                  if res = SIDE_UNDEF then
                  begin
                        numfront++;
                        numback++;
                  end
            end

            if abs(numfront - numback) < absdifference then
            begin
                  absdifference = abs(numfront - numback)
                  bestplane = pln
            end
      end

      return bestplane
end
```

A második lépés igen egyszerű. Miután tudjuk, hogy mi lesz a térosztó hipersík és tudjuk, hogy melyik sokszög melyik oldalra fog kerülni, és melyek azok,

amelyeket el kell vágnunk, egyszerűen létrehozunk egy bal és egy jobb oldali ágat, és a hozzájuk tartozó sokszögeket betesszük ide.

A továbbiakban az előző két lépést ismételgetjük rekurzívan, amíg valamilyen feltétel nem teljesül. Ez a feltétel általában kétféle lehet. Az egyik, amikor azt tartjuk szem előtt, hogy a vizsgált térrészben megfelelően kevés számú test illetve sokszög legyen megtalálható, a másik az, amikor azt figyeljük, hogy a keletkezett sokszögek konvex halmazt alkotnak-e. Mi a második esetet választjuk, mivel a későbbiekben ilyen felépítésű fára van szükségünk.

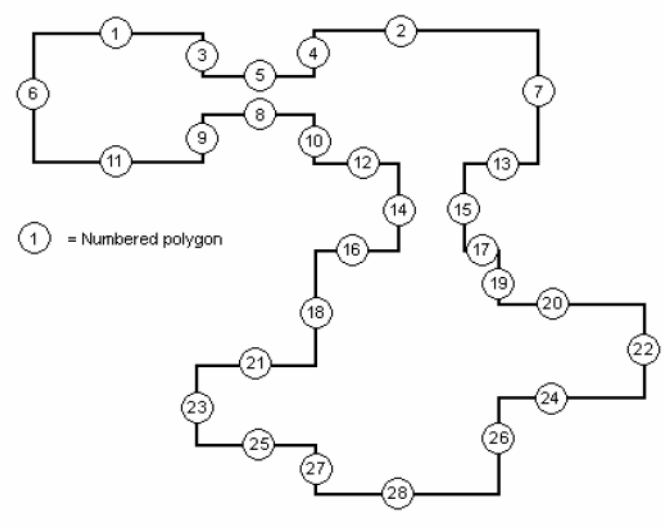
Tehát meg kell vizsgálnunk, hogy konvex-e a sokszöghalmazunk, amit vizsgálunk. Ezt megtelhetjük úgy, hogy végigmegyünk az összes sokszögön és megnézzünk, hogy az összes többi ugyanazon az oldalán található-e. Ezt a vizsgálatot azonban a FindBestDivisorPlane függvénybe is beleintegrálhatjuk, mivel az már úgyis végrehajt egy hasonló funkcionalitású egymásba ágyazott ciklust. Feladatunk tehát úgy kiegészíteni ezt az algoritmust, hogy a konvexitást is vizsgálja.

Egy számlálót akkor és csak akkor növelünk a főciklusban, ha a sokszög előtt illetve mögött található másik sokszögek száma nulla, vagyis mindegyik mögötte illetve előtte van. Ha ez a számláló a főciklus lefutása után megegyezik a sokszögek számával, akkor ez azt jelenti, hogy konvex sokszöghalmazról van szó.

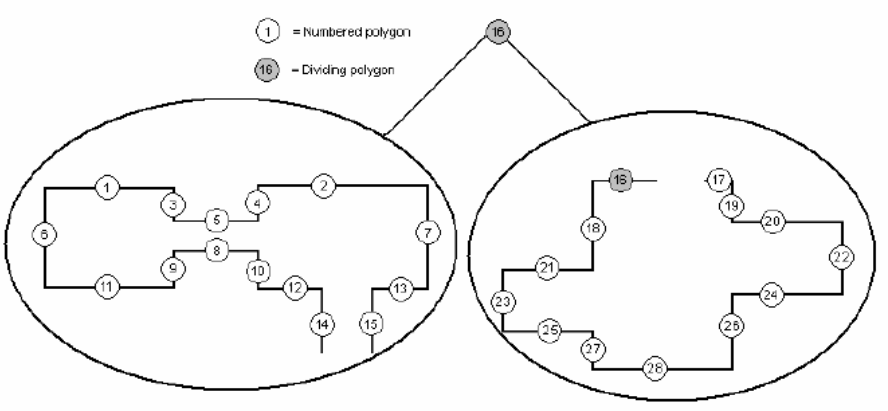
```
void CBSPGenerator::SubdivideSpaceRecursively()
begin
      if IsConvexSet then return

      if FindBestDivisorPlane() then
      begin
            CreateSideAndPutPolygons(SIDE_FRONT, frontside)
            CreateSideAndPutPolygons(SIDE_BACK, backside)

            FreeCurrentPolygonList()

            frontside.SubdivideSpaceRecursively();

            backside.SubdivideSpaceRecursively();
      end
end
```

Tehát megvizsgáljuk, hogy konvex sokszöghalmazzal van-e dolgunk, ha igen nem teszünk semmit (ez lesz a generálás megállási feltétele). Majd megkeressük a legjobb osztósíkot. Miután sikerrel jártunk, létrehozzuk a bal és jobb oldali új csúcsokat, majd mindegyikbe beletesszük az oda tartozó sokszögeket. Ha kell, vágunk. Töröljük a szülőben található sokszögeket, mivel ezek már a gyerekekben benn vannak és csak ott lesz rájuk szükség. Majd rekurzívan meghívjuk ezt az algoritmust a bal és jobb oldali ágakra is.

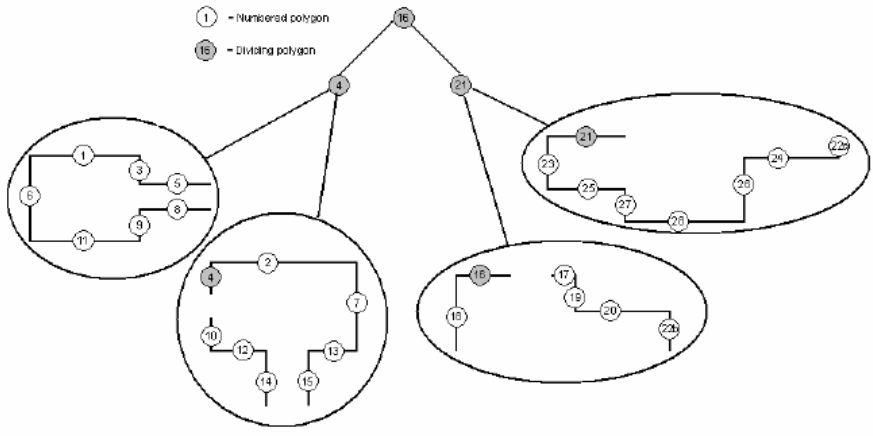
Tekintsük a következő ábrákat, amelyek a generálás lépéseit szemléltetik:



Eredeti színtér



Első lépés



Második lépés (vízszintes bejárással)

4.3 Bejárása – kirajzolása

A BSP-fa algoritmus alapvető célja az volt, hogy a teret segítségével meghatározott sorrendben, közelről-távolra illetve távolról-közelre haladva rajzoljuk ki. Először ezt ismertetjük, majd megmutatjuk, hogy hogyan lehet a BSP-fákat a láthatótérből kilógó térrészek eltávolítására használni.

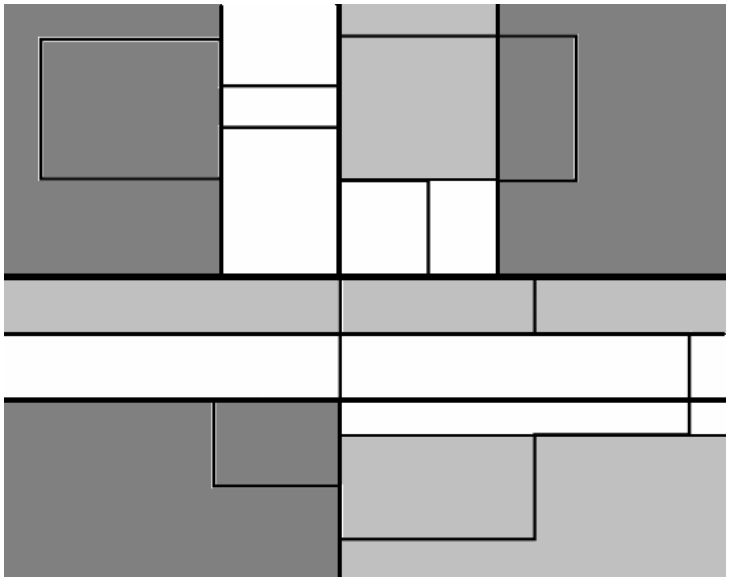
Az kirajzoló eljárás a következő:

```
BSPTree::WalkTree()
begin
      if viewpoint*hyperplane >= 0 then
      begin
            if frontnode!= NULL then
                  frontnode.WalkTree()

            DrawPolygons()

            if backnode!= NULL then
                  backnode.WalkTree()
      else
      begin
            if backnode!= NULL then
                  backnode.WalkTree()

            DrawPolygons()

            if frontnode!= NULL then
                  frontnode.WalkTree()
      end
end
```

Ha megnézzük, ez egy feltétellel kiegészített dupla inorder fabejárás [CRL99]. A feltétételben azt vizsgáljuk, hogy a nézőpont és az aktuálisan vizsgált hipersík szorzata nagyobb-egyenlő-e, mint 0. Ha igen, ez azt jelenti, hogy a nézőpont a hipersík előtt helyezkedik el, tehát a normálvektora a nézőpont felé mutat. Ha a nézőpont a hipersík előtt helyezkedik el, akkor a konstrukciónkból következően ez közelebb van, mintha a másik oldalon lenne. Ebben az esetben a hipersík előtt található térrészbe lépünk be először. Miután visszatértünk a rekurzióból, kirajzoljuk a sokszögeket. A mi konstrukciónk szerint sokszögek csak a levelekben találhatók, de a csúcsokban is meghívjuk a kirajzoló algoritmust, hogy ne sértsük meg az inorder tulajdonságot. Ezután rekurzív módon a hipersík mögötti térrész felé indulunk el. Ha a feltétel nem teljesül, akkor a „mégse” ágba lépünk, mivel ekkor a hipersík mögötti rész van közelebb a nézőponthoz. Ebből következik, hogy a fenti algoritmus közelről-távolra sorrendben rajzolja ki a teret. Ha éppen az ellenkező kirajzolási sorrendet szeretnénk elérni, akkor a feltétel két ágát meg kellene cserélnünk.

Az alkalmazásokban mindkettő kirajzolási sorrend megtalálható. Ha átlátszó felületeket is használunk, akkor először a nem átlátszó felületeket kell kirajzolnunk valamilyen sorrend alapján, de közben karban kell tartanunk a Z-buffer értékeit. Majd egy távolról-közelre kirajzolás szerint az átlátszó felületeket rajzoljuk ki, ezáltal két egymás mögött elhelyezkedő átlátszó felületet is meg tudunk jeleníteni.

A következő képen látható az előzőekben felosztott tér. A szakaszok hipersíkokat jelentenek, amelyek vastagságával a fában elfoglalt mélységüket jelöltük. A szürkeárnyalatokkal még jobban próbáltuk az ábra szemléletességét javítani. Az imént ismertetett algoritmus jobb megértése érdekében az olvasó kijelölhet nézőpontokat az ábrán és végigkövetheti az algoritmus működését:



Felosztás

A látótéren kívül eső térrészek eltávolításában a BSP-fa, struktúrája miatt, kitűnő segítségünkre lehet.

Ha ismerjük minden egyes csúcs és levél befoglaló objektumát, akkor ezek felhasználásával az egész csúcsról illetve levélről el tudjuk dönteni, hogy kívül vagy belül esik a látótéren.

A generáló algoritmusban mindössze annyi a különbség, hogy minden egyes csúcsban létrehozzuk a befoglaló objektumot.

Tudjuk, hogy egy csúcs gyerekeinek befoglaló objektumait fizikailag tartalmazza a csúcs befoglaló objektuma. Mivel hierarchikus tartalmazási viszonyról van szó, ezért ezt angolul „Bounding Volume Hierarchy”-nak nevezzük.

Tehát ha a csúcs befoglaló objektuma a látótéren kívül található, akkor nyilvánvaló, hogy a gyerekek befoglaló objektumai, így a gyerekek is teljes

egészében a látótéren kívül találhatóak. Ha a csúcs befoglaló objektuma a látótéren belül található, vagy metszi a látótér határoló síkjait, akkor ebből sajnos nem tudunk még semmire sem következtetni, egy szinttel lejjebb kell lépnünk a fában.
Az így módosított bejáró algoritmus:

```
BSPTree::WalkTree()
Begin
      if frustum.IsBoundingBoxOutside then
            return

      if viewpoint*hyperplane >= 0 then
      begin
            if frontnode!= NULL then
                  frontnode.WalkTree()

            DrawPoligons()

            if backnode!= NULL then
                  backnode.WalkTree()
      else
      begin
            if backnode!= NULL then
                  backnode.WalkTree()

            DrawPoligons()

            if frontnode!= NULL then
                  frontnode.WalkTree()
      end
end
```

Mindössze egy feltétel a különbség és ezáltal a vágást és a sorba rendezést egy lépésben el tudtuk végezni.
A módosított generáló algoritmus:

```
void CBSPGenerator::SubdivideSpaceRecursively()
begin
      CreateBoundingBox()

      if IsConvexSet then return

      if FindBestDivisorPlane() then
      begin
            CreateSideAndPutPoligons(SIDE_FRONT, frontside)
            CreateSideAndPutPoligons(SIDE_BACK, backside)

            FreeCurrentPolygonList()

            frontside.SubdivideSpaceRecursively();

            backside.SubdivideSpaceRecursively();
      end
end
```

A generáló algoritmus komplexitását nehéz meghatározni, erre most nem térünk ki. Annak meghatározása, hogy egy térrész a látótérbe esik-e illetve nem, logaritmikus feladat.

4.4 Alkalmazásának történeti áttekintése

A BSP-fa (Binary Space Partitioning Tree) fogalmát 1980 körül Fuchs, Kedem és Naylor vezette be. Két cikket írtak a témában: „Predetermining Visibility Priority in 3D Scenes” és „On Visible Surface Generation by A Priori Tree Structure”.

Az első olyan játékszoftver, amely ilyen belső téren dolgozott a Wolfenstein 3D volt. Ebben a játékban még nem alkalmazták a BSP-fákat. Nem kellett alkalmazni, mert a pályák felépítésére olyan megszorításokat alkalmaztak, amelyek ezt nem tették szükségessé. Minden fal ugyanolyan magas volt, ugyanott volt a teteje és az alja (tehát egy szinten történt minden esemény), a falak egymással csak 90 fokos szöget zárhattak be, a föld és a plafon egyszínű volt, amelyet a falaktól függetlenül ki lehetett tölteni. A kamerát vízszintesen lehetett mozgatni és forgatni a függőleges tengely körül. Ezekből a megszorításokból következik, hogy függőlegesen minden pixel ugyanahhoz a falhoz tartozott. A megjelenítés úgy történt, hogy sugarakat lőttek ki minden oszlop felé, és amelyik falat legközelebb metszette, abból rajzolta ki a grafikus motor az oszlopot. Később rajzolta ki a pálya alját és tetejét, valamint az esetleges berendezési tárgyakat és embereket is. Tehát a Wolfenstein 3D nem igazi sokszögeket jelenített meg.

Wolfenstein 3D

A következő alkalmazás a Doom már tényleges sokszögekkel dolgozó játékszoftver volt. Itt is megvolt a Wolfenstein-nél említett megszorítások nagy része. Azzal a különbséggel, hogy ugyan itt is csak egy szint volt megtalálható a játéktérben, de a falaknak nem kellett ugyanolyan magasnak lenniük. A földet és a plafont már textúrával látták el. A játékban kétdimenziós BSP-fát alkalmaztak. Az egy szinten belüli magasságkülönbségek ugyan megengedettek voltak, de a háromdimenziós BSP-fa használata még így is szükségtelen volt. Mivel a kamerát itt is csak a függőleges tengely körül forgathattuk, így most is kihasználható volt, hogy a falak függőleges oszlopokból állnak, csak most egy másképpen. A teret itt már

szektorokra osztották, melyek között átlátszó, illetve ajtóval határolt összekötő részek voltak megtalálhatók. Amikor a játék grafikus motorja közelről-távolra bejárta a BSP-fát, akkor minden olyan függőleges pixeloszlopot foglaltnak jelölt, ahol nem volt összekötő rész. Ha egy következő térrész kirajzolása közben olyan oszlophoz ért, ami foglalt volt, ott nem tett semmit, csak akkor rajzolt és tette foglalttá az oszlopot, ha még üres volt. A kirajzolás addig ment, amíg minden oszlop nem volt foglalt. A statikus berendezési tárgyak a BSP-fához tartoztak, így csak a dinamikusan mozgó ellenfeleket kellett a Z-buffer felhasználásával kirajzolni.

Doom

Az igazi áttörést a Quake jelentette, ahol nem volt semmi megszorítás a játéktér felépítésére. A falak bárhogyan elhelyezkedhettek egymáshoz képest, a pálya több szintből állhatott, a kamerát bármilyen irányba eldönthettük.

Itt már háromdimenziós BSP-fát alkalmaztak. Mivel a játéktér több tízezer sokszögből állt, és az akkori számítógépek ezt nem lettek volna képesek valós időben megjeleníteni, ezért itt alkalmazták a PVS algoritmusát is, amelynek tárgyalására dolgozatunk egy későbbi fejezetében kerül majd sor.

Quake

5 Portal-ok

A későbbiekben ismertetésre kerülő PVS technika a BSP-fáknak és a portal technika [Elm99] ötvözése. Ezért mielőtt ezt ismertetnénk, meg kell ismertetnünk az olvasóval a portal technika lényegével.

A portal technikát 1971-ben említették először, mint háromdimenziós belső színterek láthatóság vizsgáló-megjelenítő algoritmusát.

A portal szó angol jelentése: bejárat, átjáró. Bejárat, átjáró de mik között? A háromdimenziós belső tereket szobákra, szektorokra és ezek között elhelyezkedő bejáratokra, átjárókra bonthatjuk. Nyilvánvaló, hogy egyik szobából a másikba csak ilyen átjárókon keresztül nézhetünk át, de ott sem láthatunk mindent. A portal technika ezt használja ki. A portal matematikailag egy egyszerű sokszögnek tekinthető.

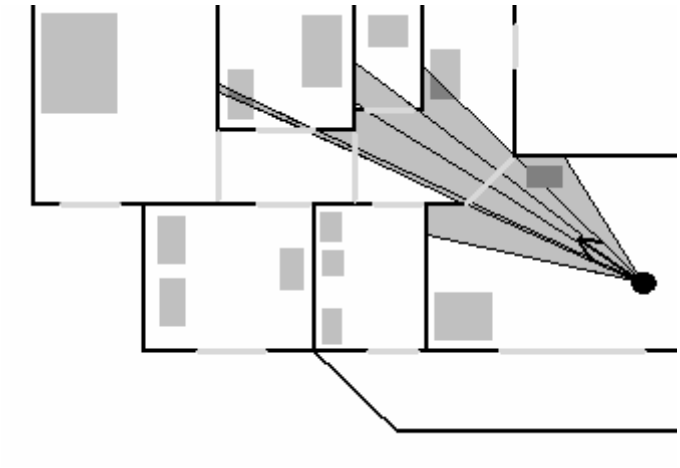
Mint említettük a portal-ok és a BSP-fák ötvözéséből alakult ki a PVS, azonban a portal algoritmus önmagában is alkalmazható.

A portal-ok segítségével egyszerre oldhatjuk meg a láthatósági feladatot, valamint a színtér kirajzolását is elvégezhetjük segítségével. A BSP-fa algoritmushoz hasonlóan, a portal-ok is kitűnően felhasználhatók az ütközésvizsgálatkor (collision detection). Könnyen meghatározhatjuk, hogy melyik szektorban vagyunk, hiszen ezek konvexek, így ha biztosítjuk, hogy minden határoló sík normálvektora a szektor középpontja felé néz, akkor egyszerű skaláris szorzatokkal eldönthetjük, hogy a nézőpont egy adott szektorban található-e.

Régebben előszeretettel alkalmazták belső színterű játékok grafikai megjelenítésénél, de nagy számításigénye, valamint a grafikus gyorsító kártyák fejlődése miatt ma már nem alkalmazzák.

5.1 Működési elve

A portal technika egy olyan rekurzív bejáró algoritmus, amely szereti, ha konvex térrészeket biztosítunk számára. Ezek a feltételek hasonlóak azokhoz, amelyeket már megismertünk a BSP-fa algoritmus tárgyalásakor. A BSP megjelenítő algoritmusa azonban megkövetelte, hogy konvex térrészeket biztosítsunk számára, a generáló algoritmusa ilyen térrészeket hozott létre.



Portal-ok

Az ábrán látható egy nézőpont, néhány szoba és az őket összekötő portal-ok. Látható, hogy abban a szobában, ahol a nézőpont található a látótér vágósíkjait használjuk a látótérből kilógó elemek eltávolítására. Ezt a szobát 1-1 portal köti össze 1-1 másik szobával. A látótérbe nem eső portal-okat figyelmen kívül hagyjuk. A látótérbe eső portal segítségével a látótér vágósíkjait elvágjuk úgy, hogy csak akkora teret foglaljanak magukba, amekkorát a portal megenged. Belépünk a portal másik oldalán található szobába. Az itt található sokszögeket és portal-okat szintén elvágjuk az előbb kialakított vágósíkokkal. Az algoritmust rekurzív módon addig folytatjuk, amíg van olyan szoba, amely valamelyik portal-on keresztül látható.

Megfigyelhetjük, hogy a vágások száma igen magas. Minden olyan szoba összes sokszögét, amelybe belátunk, el kell vágni az éppen aktuális vágósíkok mindegyikével. Ez időigényes feladat. Sokkal időigényesebb, mintha egy megfelelően gyors grafikus kártyának átadnánk az összes sokszöget, és ő végezné a vágásokat és a takarásvizsgálatot megjelenítés közben. Ezt természetesen megtehetjük, azonban mi most az alapalgoritmust tárgyaljuk, amely a vágást a fent említett módon implementálja. A portal technika hátrányai közé sorolható még, hogy

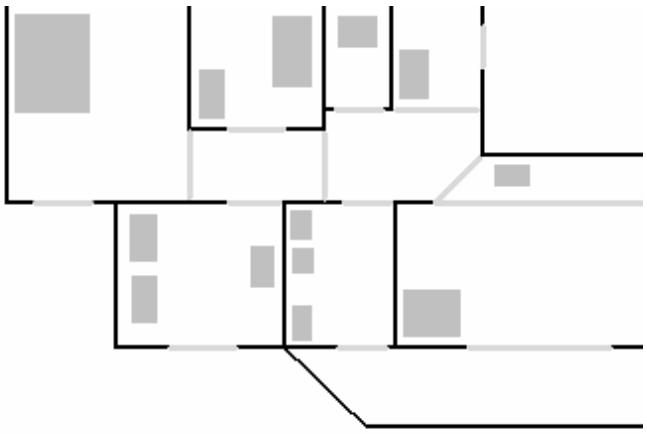
csak belső színtereken működik, valamint célszerű konvex térrészeket biztosítanunk számára. A konkáv térrészek esetén elvesztjük azt a nagy előnyét, hogy egyetlen pixelt sem rajzolunk felül.

Az előnyök közé tartozik még, hogy gyors és egyszerű rekurzív algoritmust alkalmazhatunk. Dinamikusan változó világokat definiálhatunk segítségével oly módon, hogy portal-okat vehetünk el, illetve adhatunk hozzá a térhez, illetve a portal-okat kapcsolgathatjuk ki, illetve be.

5.2 A portal-ok manuális behelyezése

Ha olyan megjelenítő motort szeretnénk használni, amely a portal technikát alkalmazza a láthatóság-vizsgálat, valamint a megjelenítés feladatának megoldására, akkor kézenfekvő megoldás az, hogy szobákat helyezzünk el a térképen, majd ezeket kössük össze átjárókkal portal-okkal. Ha csak buta módon sokszögeket helyeznénk el, amelyek között nem határoznánk meg átjárókat, akkor igen nehéz feladat lenne ezt a logikátlan adathalmazt a portal technika által alkalmazható formára hozni.

Mint említettük portal lehet bármilyen átlátható rész, amely szektorokat köt össze. Általában ajtó, ablak. Ezek manuális behelyezésével nincs is gond, mivel amikor leteszünk egy ajtót, akkor automatikusan hozzáadja a szerkesztő programunk a megfelelő portal-t. A portal így „tudhatja”, hogy melyik két szobát köti össze. Minden szoba rendelkezik egy egyedi azonosítóval, amelyeket a portal-ban is eltárolunk. Mindenképpen el kell kerülnünk, hogy a szoba az átjáróin kívül máshol is lyukas legyen. Mivel az ilyen lyukakat nem tekintjük portal-nak, és ezek a lyukak teljesen fekete pixeleket eredményezhetnek. Akkor is gond adódik, ha valamelyik



Konkáv térrészek konvex felbontása

szobánk konkáv. Ekkor a szobába olyan portal-okat kell elhelyeznünk, amelyek megszüntetik a szoba konkávitását, és konvex részekre bontja.

Vessük össze az előző részben található, hasonló felépítésű ábrát a fentivel! Megfigyelhetjük, hogy az előző rész ábráján konkáv szobákat is láthattunk, a fentiben már csak konvexeket. Ha az első ábrán futtatnánk a portal algoritmust, akkor lennének olyan pixelek, amelyeket felülrajzolnánk, valamint olyan portal-okat találhatna felhasználhatónak a rekurzió alkalmazására az algoritmus, amelyek nem láthatóak, így alkalmatlanok a látósíkok számítására, mivel eltakarja őket egy fal.

Ezek az esetek nézőpontfüggők. A felülrajzolt pixeleket természetesen kiküszöbölhetjük a Z-buffer algoritmus alkalmazásával, a felesleges portal-okat máskülönben csak nagy számítás árán tudnánk meghatározni.

A következő fejezetben, ahol a PVS technikát tárgyaljuk bemutatunk egy megoldást arra, hogy miként tudunk portal-okat algoritmikus eszközökkel behelyezni. Ott azzal az előnnyel rendelkezünk majd, hogy a színterünk egy BSP-fában található, és nem szobákra van osztva, tehát most más megoldást kell választanunk, mivel az a megoldás itt nem alkalmazható. Most két lehetőségünk van:

Az első megoldás az, amikor csak konvex testekből (kocka, piramis, téglatest, stb.) rakjuk össze a konkáv szobáinkat. E konvex testek illesztési síkjainál automatikusan elhelyezhetünk 1-1 portal-t. Ennek az a hátránya, hogy komplex szobák felépítése nehézkes lehet. Az érintkező felületek pontos meghatározása és a határoló sokszög, azaz portal kiszámítása sok vágást igénylő geometriai feladat. Az előbb említett testek zártak, tekinthetjük őket homogén, tömör szilárd testeknek. Azt a technikát, amikor ilyen testekkel halmazműveleteket végzünk CSG-nek (Constructive Solid Geometry) nevezzük. A fenti megoldásban a halmazműveletek közül az uniót – amikor egymás mellé illesztettünk két konvex testet -, valamint a metszetképzést - amikor meghatároztuk, hogy az illeszkedő felületek közötti sokszöget – alkalmaztuk. Ezt a technikát leginkább tudományos geometriai számítások elvégzésére alkalmazzák.

A másik megoldás az, hogy konkáv szobákat hozunk létre, és addig helyezünk be az ilyen konkáv szobákba portal-okat, amíg konvex szobákhoz nem jutunk. A szoba konvexitását azzal az algoritmikussal is ellenőrízhetjük, amelyet a BSP-fa generálásánál bemutattunk.

Tehát a pályaszerkesztő programunkba beépíthetünk egy olyan szolgáltatást, amely megvizsgálja, hogy minden szoba konvex-e. Ha nem, akkor jelzi, hogy melyik szobába helyezzünk be portal-okat. A portal-ok manuális behelyezése egyáltalán nem nehéz feladat, tehát a szerkesztőprogram felhasználójára is bízhatjuk.

Azonban ezt a procedúrát automatizálhatjuk is egy egyszerű algoritmus megírásával: Vegyük minden egyes fal által meghatározott síkot. Ezt a síkot vágjuk el a szoba összes fala által meghatározott síkkal. Ha találtunk olyan síkot, ahol a szoba belsejében keletkezik sokszög az előbb elvágott síkból, akkor e sík mentén vágjuk ketté a konkáv szobát. Ide helyezzünk be egy portal-t. Ha valamelyik így keletkezett szoba még mindig konkáv, akkor erre is alkalmazzuk rekurzívan ezt az algoritmust.

5.3 Bejárasa – kirajzolása

A bejáró algoritmus abból a szobából indul, ahol a nézőpont elhelyezkedik. A látóteret határoló síkokkal elvágja a szoba összes sokszögét, majd a látható sokszögeket, illetve a részlegesen látható sokszögek látótéren belül maradt darabjait kirajzolja. Ezekkel a látósíkokkal elvágja a szoba összes portal-ját is. A portal-ok is sokszögek, tehát a vágó algoritmus teljesen megegyezik a szoba sokszögeinél alkalmazottal. A portal-ok élei mentén újraszámítjuk a látóteret, azaz meghatározzuk azokat az új látósíkokat, amelyek azokra a háromszögekre illeszkednek, amelyek egyik csúcsa a nézőpont, a másik kettő a portal által meghatározott sokszög 2-2 csúcsa. Ezután a portal másik oldalán található szobába lépünk át és rekurzívan alkalmazzuk tovább az algoritmust megismert lépéseit. Addig lépünk be új szobákba, amíg találunk olyan portal-t, amely a látótérben található.

Tekintsük a bejáró algoritmust:

```
DrawRoomRecursively (Room actualroom, Frustum viewfrustum)
begin
      list<Polygon> clippedpolygons;

      for each polygon p in actualroom.GetPoligons() do
      begin
            if viewfrustum.IsPolygonVisible(p) then
            begin
                  clippedpoly = p.ClipBy(viewfrustum)

                  clippedpolygons.add(clippedpoly)
            end
      end

      cippedpolygons.DrawAllPolygon();

      for each portal p in actualroom.GetPortals() do
      begin
            if viewfrustum.IsPortalPolyVisible(p) then
            begin
                  clippedportal = p.ClipBy(viewfrustum)

                  newviewfrustum =
                        CreateNewFrustumOnSidesOf(clippedportal)

                  otherroom = clippedportal.GetOtherRoom(actualroom)

                  DrawRoomRecursively(otherroom, newviewfrustum)
            end
      end
end
```

Ebben az implementációban minden egyes háromszöget elvágunk a látósíkokkal, valamint betesszük egy listába, és ezt adjuk át a megjelenítő motornak. A sok vágás nem előnyös, ha 3D gyorsító kártyát használunk a megjelenítéshez, mivel a vágásokat a processzor végzi, ami nem tartalmaz erre alkalmas speciális utasításokat, így sokkal lassabban oldja meg a feladatot, mintha az egészet a direkt erre a célra specializált grafikus processzorra bíznánk. Ebben az esetben mindenképpen alkalmaznunk kell a Z-buffer algoritmust.

5.4 Alkalmazásának történeti áttekintése

Az első portal technológiát használó grafikus motorral rendelkező PC-n is elérhető alkalmazás a Duke Nukem 3D volt. Kiadási idejét tekintve a Doom és a Quake közé tehető. A Doom-tól eltérően itt már egymás felett is elhelyezkedhettek a szobák, valamint a kamerát is eldönthettük.

Duke Nukem 3D

Ha a kamera egyenesen állt, akkor a falak kirajzolása itt is függőleges oszloponként történt és nem vízszintesen. Ha eldöntöttük a kamerát, akkor már vízszintes scanline-okra tért át a motor. Az első esetben ezt azért tehette meg, mivel függőlegesen a Z érték egy falon belül konstans. A perspektíva korrekt textúrázásnál [Abr97, Szir03] alkalmazott osztási műveletet így sokkal kevesebbszer kellett végrehajtania. Az ebből a megoldásból adódó különbség a másodpercenként kirajzolt képkockák között a nem döntött és döntött kamerát tekintve szemmel látható volt az FPS-mérő bekapcsolásával.

A másik játék (amelyről sajnos nem áll rendelkezésünkre kép) a Descent volt. Ezután a portal-motorokat kiszorította a BSP+PVS párost használó motorok tömkelege.

6 A PVS

Mint az előző fejezetekben rávilágítottunk, a BSP-fa algoritmus kitűnően megoldja a sokszögek, illetve objektumok sorba rendezésének problémáját, valamint megoldja a látótéren kívül eső térrészek hierarchikus eltávolítását is. Egy igen fontos problémát, az eltakart testek és sokszögek eltávolítását azonban önmagukban nem képesek megoldani. Ezért ötvözték a BSP-fákat a PVS-el (Potentially Visible Set). Nem oldja meg egzaktul a feladatot, hanem egy felső becslést ad a kirajzolandó konvex térrészekre, ami nekünk tökéletesen elég. A „konvex térrész” elnevezés helyett (BSP terminológia) alkalmazhattuk volna a szoba, illetve szektor (portal terminológia) fogalmat is, mivel ez a két fogalom szoros kapcsolatban áll egymással. A következő részekben rávilágítunk, hogy miért. A két fogalom ugyanazt jelenti a következőkben.

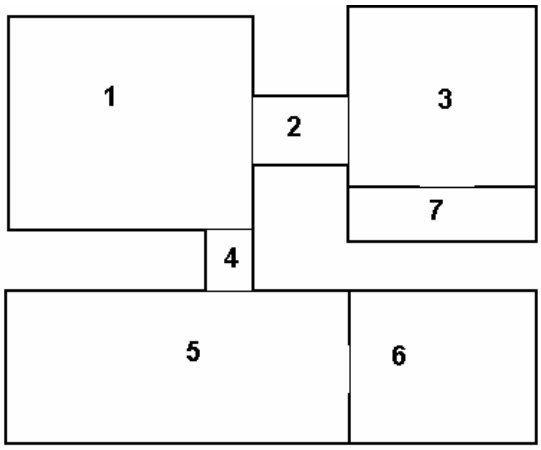
6.1 Működési elve

A PVS [MikL03, NathW] a BSP-fa és a Portal-ok tulajdonságait kihasználva határozza meg a láthatósági relációkat. A BSP konvex térrészekre osztja a teret. Ezek a konvex térrészek természetesen nem teljesen zártak. Ha azok lennének, akkor nem láthatnánk át egyik térrészből a másikba, nem lenne érdekes a feladat.

A PVS ezek között adja meg a láthatósági relációt, amely azt határozza meg, hogy van-e olyan összekötő szakasz a két térrész között, amely nem metsz el egyetlen sokszöget sem és a végpontjai a vizsgált két térrész egy-egy pontja.

Nyilvánvaló, hogy ha van olyan szakasz, amely ennek a tulajdonságnak megfelel, az nem jelenti azt, hogy a kirajzolási fázisban a nézőpontot tartalmazó térrész bármely pontjából látható a másik térrész.

A következő ábrán egy egyszerű színteret láthatunk, amelyet akár még BSP



Egyszerű belső színtér

fával is generálhattunk volna, mivel konvex térrészekből áll és gondolatban behelyezhetők az osztósíkok. Az első osztósík a 4-es és 5-ös szoba határán található fal lenne. Az 5-ös és a 6-os közé osztósík helyezésével mind a kettő konvex lesz. A következő az 1-es és a 2-es határánál lenne. A következő az 1-est és a 4-est választaná szét, így mindkettő konvexszé válna, majd hasonlóan a 2-est és a 3-ast. Így a 2-es konvex, már csak a 3-as és a 7-es közé kell egy osztósíkot helyezni. Ezzel készen vagyunk. A BSP-fa generáló algoritmus is a fentihez hasonló módon járt volna el. A portal-okat az említett osztósíkokra helyezzük el, vékonyabb vonallal jelöltük őket. Az, hogy ezeket a portal-okat hogyan kell algoritmikus módszerekkel létrehozni a következő rész témája lesz.

A másik fontos dolog, amit meg kell határoznunk az a láthatósági relációk mátrixsza. Ez a mátrix vízszintesen és függőlegesen is annyi oszlopot illetve sort

tartalmaz, mint amennyi konvex térrész, azaz szoba keletkezett a felosztás során. A mátrix elemei bitek. A bit értéke 1 lesz, ha van olyan összekötő szakasz a két térrész között, amely nem metsz el semmilyen sokszöget, 0, ha nincs. Ez a mátrix nagy terek esetében nagy lehet. Pl. egy 5000 térrész tartalmazó térben 5000x5000 bitet, azaz 3.125.000 bájtot foglal a táblázat. Mivel az ekkora terek elég bonyolultak, ezért a távol eső térrészek kis eséllyel láthatóak, így a mátrix meglehetősen ritka lesz. Ebből következik, hogy akár a RLE (Run Length Encoding) tömörítő eljárással is lényegesen csökkenthetjük a mátrixunk méretet. A gyakorlatban is ezt az eljárást alkalmazzák.

A fenti tér láthatósági relációit ábrázoló mátrix:

	1	2	3	4	5	6	7
1	1	1	1	1	1	0	1
2	1	1	1	1	1	0	1
3	1	1	1	0	0	0	1
4	1	1	0	1	1	1	0
5	1	1	0	1	1	1	0
6	0	0	0	1	1	1	0
7	1	1	1	0	0	0	1

Egy szimmetrikus mátrixot kaptunk, amelynek elég csak az alsó vagy felső háromszögmátrixát eltárolni és nagy tér esetén tömöríteni.

Amikor kirajzoljuk a teret, akkor egyszerűen a bejárjuk a BSP-fát [MHAV]. Először meghatározzuk, hogy melyik térrészből indult a kirajzolás. Majd egyszerűen, amikor bejárjuk a BSP-fát, akkor a térrész oszlopából illetve sorából egyszerűen kiolvassuk, hogy ki kell-e rajzolni a térrészt (ha az még a látótérben található).

6.2 Generálása

Az előzőekben láthattuk a PVS működési elvét. Ebben a fejezetben bemutatjuk, hogy ezeket a láthatósági relációkat miként határozhatjuk meg.

A generáló algoritmusunk kiindulása az a BSP-fa, amelynek generálási algoritmusát az előzőekben már ismertettük. Emlékeztetőül megemlítjük, hogy ennek a fának csak a leveleiben vannak sokszögek, a fa többi csúcsában csak az osztósíkra vonatkozó információt tároljuk. Az ilyen típusú BSP-fákat „leafy", azaz „leveles" fának nevezzük. Feladatunk a levél-levél láthatósági viszonyok meghatározása. Mivel a többi csúcs több levelet „fog" össze, ezért ezek láthatósági relációi érdektelenek, ez a levél-levél viszonyokból általánosítható.

A generáló algoritmus két részre bontható. Az első részben a BSP-fa osztósíkjai mentén el kell helyeznünk a portal-okat. A második részben ezeknek a portal-oknak a segítségével kell meghatároznunk a levél-levél láthatósági viszonyokat.

A feladat első részét Nathan Whitaker [NathW] által is ajánlott algoritmus alapján oldjuk meg, amely három részből áll: Az első lépésben létrehozunk egy befoglaló téglatestet az egész színtér köré, majd az osztósíkokon sokszögeket hozunk létre, amelyeket elmetszünk a befoglaló objektum oldalaival. Így nagy méretű négyszögek keletkeznek. Majd pedig minden egyes így keletkezett sokszöget elmetsszük a BSP-fa összes osztósíkjával. Így nagyon sok lehetséges portal-t kapunk, mindegyiket elhelyezzük egy listában, és egy egyedi azonosítót rendelünk hozzájuk. A második lépésben a lista összes sokszögét „letuszkoljuk" a BSP-fába, amíg levelet nem találunk. Ha a sokszög az osztósík előtt helyezkedik el, akkor a fa elülső része felé küldjük le a sokszöget, ha mögötte helyezkedik el, akkor a hátsó része felé, ha a síkon fekszik, akkor mindkét irányba „letuszkoljuk". A harmadik lépésben megvizsgáljuk, hogy melyek azok a sokszögek, amelyek két levélbe is lekerültek, ezekkel foglalkozunk a továbbiakban. Majd mindkét levél összes sokszöge mentén elvágjuk a potenciális portal-unkat. Ami kiesik, az nem lehet portal. A vágás végeredménye egy konvex sokszög, amely a két levél határán helyezkedik el.

A feladat második részét a „félárnyék" algoritmus segítségével oldjuk meg. Minden levélből kiindulva rekurzívan, az összekötő portal-ok mentén további leveleket érintve bejárást hajtunk végre. Akkor látható a levél, ha a felhasznált portal látszódik a kiindulási portal árnyékából és fordítva. A másik megoldási lehetőség a Plucker koordináták használata lenne, ezt nem tárgyaljuk.

A feladat megoldó algoritmusának első és második részét a következőkben ismertetjük részletesen, pszeudó-programok bemutatásával.

A generáló algoritmusokhoz részletesebb pszeudó kódot adunk, mivel így jobban megérthető az algoritmusok valódi működése.

6.2.1 Portal-ok algoritmikus behelyezése

Az első lépés legelső mozzanata, hogy egy befoglaló dobozt hozunk létre az egész színtér köré. Rekurzívan végighaladunk a BSP-fa összes csúcsán, valamint levelén, és az összes itt található sokszög összes csúcspontja által meghatározott minimum és maximum koordinátákat keressük meg. Miután megvannak ezek a koordináták, ezekre könnyen illeszthetünk hat síkot, amely a befoglaló dobozt fogja reprezentálni.

A következő mozzanat az osztósíkok mentén nagy méretű négyszögek létrehozása majd minden egyes négyszög elmetszése a befoglaló doboz lapjaival.

```
void CBSPandPortalGenerator::PrecalculateLargePortals()
begin
      if m_frontside <> NULL and m_backside <> NULL then
      begin
            CPortal tmp, p;

            tmp.CreateLargeFromPlane(m_divisorplane, SCENESIZE_MAX);
            p = BoundingBox.CreateBBoxClippedPortal(tmp);

            AddToList(LargePortalList, p);
      end

      if m_frontside <> NULL then
            m_frontside.PrecalculateLargePortals();

      if m_backside <> NULL then
            m_backside.PrecalculateLargePortals();
end
```

Tekintsük a PrecalculateLargePortals metódust! Ez a metódus rekurzívan belelép az összes csúcsba, de mindenek előtt minden csúcsra ellenőrizi hogy levél-e, ahol éppen állunk. Ha nem levél, akkor az osztósíkon létrehozzunk egy olyan nagy méretű sokszöget, amelyet a színtér befoglaló téglateste határol. Ezt két lépésben tesszük meg.

Először egy akkora négyszöget kell létrehoznunk az osztósík felületén, amely biztosan minden irányban túllóg a befoglaló dobozon (CreateLargeFromPlane metódus). Majd pedig ezt el kell vágnunk a befoglaló doboz minden lapjával (CreateBBoxClippedPortal metódus). Majd pedig hozzáadhatjuk a portal-t a LargePortalList listához. A fenti eljárások közül a CreateLargeFromPlane az, amelyik bonyolultabb és szólnunk kell róla részletesebben.

A CreateLargeFromPlane tehát egy nagyméretű portal-t hoz létre egy síkon. Bemenő paramétere az osztósík, valamint egy „nagy” szám, amelyet próbálgatással nagyon könnyen előállíthatunk. Először lekérdezzük a sík paramétereit. Majd

keresünk egy pontot a síkon, amit később a center vektorban tárolunk. A center vektort először egy nagyon távoli pontra állítjuk, amely biztosan nincs a sík, valamint semelyik koordinátatengely közelében sem. A pont kereséséhez először megpróbáljuk a síkot átszúrni a X tengellyel. Ezt akkor tehetjük meg, ha a normálvektor „x” koordinátája nem nulla.

```
CPortal::CreateLargeFromPlane(CPlane pln, float ext)
begin
      Vector norm = pln.GetNormal;
      float D = pln.GetDistance;
      Vector center(BIGNUM, BIGNUM, BIGNUM);

      if norm.x <> 0 then
            center = Vector(D/norm.x, 0, 0);
      else
      if norm.y <> 0 then
      begin
            Vector tmp(0, D/norm.y, 0);

            if length(tmp) < length(center) then
                  center = tmp
      end else
      if norm.z <> 0 then
      begin
            Vector tmp(0, 0, D/norm.z);
            if length(tmp) < length(center) then
                  center = tmp
      end

      Vector up(0, 1, 0), right(1, 0, 0), ahead(0, 0, 1);

      Vector move = right X norm;
      float act = abs(right*norm);

      if abs(up*norm) < act then
      begin
            act = abs(up*norm);
            move = up X norm;
      end

      if abs(ahead*norm) < act then
      begin
            act = fabs(ahead*norm);
            move = ahead X norm;
      end

      Vector p1 = center + move* ext / length(move);
      Vector p2 = center + (p1-center) X norm;
      Vector p3 = center + (p2-center) X norm;
      Vector p4 = center + (p3-center) X norm;

      v=(p4, p3, p2, p1);

      frontleaf = NULL; backleaf = NULL;
      partitionednodeid = -1; portalid = -1;
end
```

Ezután próbálkozunk az „y” tengellyel. Ha sikerül átszúrnunk az „y” tengellyel, akkor megnézzük, hogy az új pont közelebb van-e az origóhoz, mint az „x” tengellyel átszúrt (illetve, ha nem sikerült ezzel átszúrni, akkor az előredefiniált távoli pontunk). Ha közelebb van, akkor a center értéke ez a pont lesz. Ugyanez az eljárás a „z”

tengely esetében is. A legközelebbi metszéspont keresésének célja a numerikus stabilitás biztosítása volt.

A következő lépés egy olyan vektor meghatározása, amely eltolható a síkra. Ez a vektor nyilvánvalóan merőleges a sík normálvektorára, mivel a sík normálvektora minden síkbeli vektorra merőleges. A merőleges kiszámításához segédvektorként felhasználjuk az egységvektorokat. A minimalizálási feltétele (a következő két if) egyrészt a numerikus stabilitás, másrészt, de leginkább a vektoriális szorzás párhuzamos vektorokkal való elvégzésének elkerülése miatt kerültek az algoritmusba.

Majd egyszerű vektorösszeadások és vektoriális szorzások segítségével meghatározzuk a szükséges négy darab pontot, valamint inicializáljuk a portal attribútumait.

Ezt a nagy portal-t, esetünkben négyszöget, már elvághatjuk a befoglaló doboz lapjaival a CreateBBoxClippedPortal metódus segítségével.

Az első lépés utolsó mozzanata az, hogy az így keletkezett portal-okat elvágjuk a BSP-fa összes osztósíkjával. Ezt a következő algoritmus végzi:

```
CBSPandPortalGenerator::SplitLargePortalsByDivisorPlanes()
begin
      AllPortalList = LargePortalList;

      foreach CPortal lp in LargePortalList do
      begin
            CPlane splitter = lp.GetPlane;

            int maxcnt = AllPortalList.Size;
            port = AllPortalList.First;

            while maxcnt > 0 do
            begin
                  CPortal frontportal, backportal;

                  Result = port.SplitPortal(splitter,
                                    frontportal, backportal);

                  if result = PORTALWASSPLIT then
                  begin
                        AllPortalList.Remove(port)

                        AllPortalList.AddPortalToListTail(frontportal);
                        AllPortalList.AddPortalToListTail(backportal);
                  end

                  port = AllPortalList.Next

                  maxcnt = maxcnt - 1;
            end
      end
end
```

Az eredmény az AllPortalList listában fog megjelenni, ami kezdetben a nagy portal-okat fogja tartalmazni, mivel ezeket kell feldarabolni. Legegyszerűbben úgy tudjuk megszerezni az osztósíkokat, hogy a LargePortalList-ben található

sokszögekre illeszkedő síkokat határozzuk meg. Másik lehetőség a BSP-fa rekurzív bejárása lenne, de ez nem annyira hatékony. Tehát mi most végighaladunk az összes portal-on és meghatározzuk a rá illeszkedő síkokat, amelyeket a splitter változóban tárolunk el. Minden lépésben meghatározzuk a portal-ok számát, mivel ez növekedni fog, ha vágás történt. Indítunk egy ciklust, amiben annyi lépést hajtunk végre ahány portal van az aktuális lépés elején (maxcnt) Az aktuálisan vágni kívánt portal a port nevű változóban foglal majd helyet. Elvégezzük a vágást. Ha nem sikerült elmetszeni a portal-t, akkor a következőre lépünk át. Ha sikerült, akkor kitöröljük az elmetszett portal-t, és betesszük a vágás után keletkezett két portal-t szigorúan az AllPortalList végére, hogy ebben a lépésben még egyszer ne dolgozzuk fel (úgysem történne vágás rajta).

Az első lépés ezzel lezárult. Eredmény: nagyon sok kisebb portal az AllPortalList listában. A második lépés bemenete ez a lista. Először minden portal-nak adunk egy egyedi azonosítót, majd minden portal-t letuszkolunk a BSP-fába:
Az AddPortal metódus rekurzívan bejárja a BSP-fát. Ha levélhez ért, akkor

```
uid = 0

foreach CPortal p in AllPortalList do
begin
      p.portalid = uid;
      uid = uid + 1

      p.frontleaf = NULL;
      p.backleaf = NULL;
end

foreach CPortal p in AllPortalList do
begin
      BSPTree.AddPortal(p);

      AllPortalList.RemovePortal(p);
end

void CBSPandPortalGenerator::AddPortal(CPortal p)
begin
      if IsLeaf() then
      begin
            m_portals.push_back(p);
            return;
      end

      side = m_divisorplane.ClassifyPortal(p);

      if side == SIDE_FRONT then
            m_frontside.AddPortal(p);
      else
      if side == SIDE_BACK then
            m_backside.AddPortal(p);
      else
      if side == SIDE_ONPLANE then
      begin
            m_frontside.AddPortal(p);
            m_backside.AddPortal(p);
      end
end
```

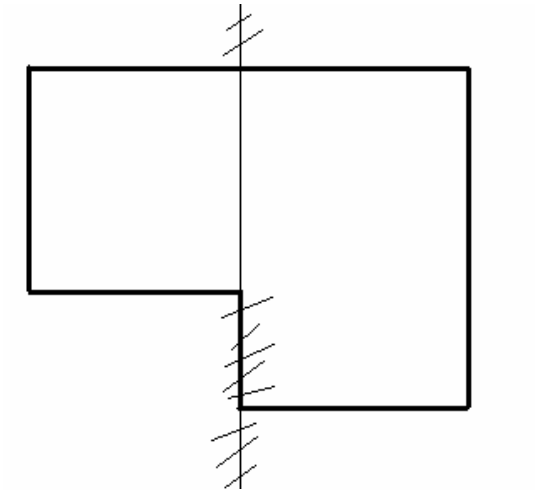
hozzáadja a portal-t és kilép a metódusból. Ha nem, akkor megvizsgálja, hogy a portal az osztósík előtt, mögött illetve rajta helyezkedik-e el. Ettől függően tuszkolja le a portal-t a megfelelő ágon. Ha az osztósíkon van a portal, akkor mindkét ágba letuszkolja. A későbbiek szempontjából csak azok a portal-ok lesznek fontosak, amelyek valamelyik lépés mindkét ágba is belekerültek. Látható, hogy most még nagyon sok olyan portal található a levelekben, amelyeket letuszkoltunk ugyan egy illetve két levélig, de nem lehet valódi portal, mert olyan helyre került, amely nem köthet össze két levelet.

A harmadik lépés ezeket a hibás portal-okat távolítja el a levelekből. Rekurzívan megkeresünk minden egyes levelet, majd minden egyes levélben minden egyes portal-on végighaladunk. Az első lépés az, hogy megnézzük, hogy

```
CBSPandPortalGenerator::FindTruePortals(CBSPandPortalGenerator root)
begin
      if IsLeaf() then
      begin
            foreach CPortal p in m_portal do
            begin
                  foundpair = root.CheckForSinglePortals(this, p);

                  if not foundpair or p.GetArea() < EPSILON then
                        m_portal.DeletePortal(p)
                  else
                  begin
                        CPortal tmp, res;

                        p.frontleaf.ClipByAllPolygons(p, tmp);
                        p.backleaf.ClipByAllPolygons(tmp, res);

                        if res.numvertices = 0 then
                              m_portal.DeletePortal(p)
                  end
            end
      end else
      begin
            m_frontside.FindTruePortals(root);
            m_backside.FindTruePortals(root);
      end
end
```

annak a portal-nak az azonosítójával, amelyet éppen vizsgálunk rendelkezik-e másik portal is. Ha nem rendelkezik, az azt jelenti, hogy egy példány van belőle, tehát el kell távolítanunk a listából. Akkor is el kell távolítanunk a listából, ha a területe közel nulla, azaz, a numerikus hibahatáron belül van, tehát tekinthető nullának is. Ha megfelelően nagy területű párral rendelkező portal-t találunk, akkor mindkét levél (amelyekben a portal-ok találhatók) összes sokszögével el kell vágnunk a portal-t.
Az ábrán felülnézetből látható két egymás mellett található levél köztük egy portal-al. Minden sokszög a hozzá tartozó térrész belseje felé mutat, a térrészek konvexek, mint ahogy ezt elvárhatjuk. Ha a portal-al párhuzamos távoli sokszögek által meghatározott síkkal vágunk, azok nem befolyásolják a portal-t. Ha a portal-on fekvő sokszög által meghatározott síkkal vágunk, akkor azok szintén nem befolyásolják a

portal-t. A maradék sokszögek, amelyek esetünkben merőlegesek a portal-ra, kivágják a portal-ból azokat a részeket, amelyek kívül esnek a két levél közötti átjárható részen, valamint azokat a részeket is, amelyek rajta fekszenek a portal-on.



Portal vágása sokszögekkel

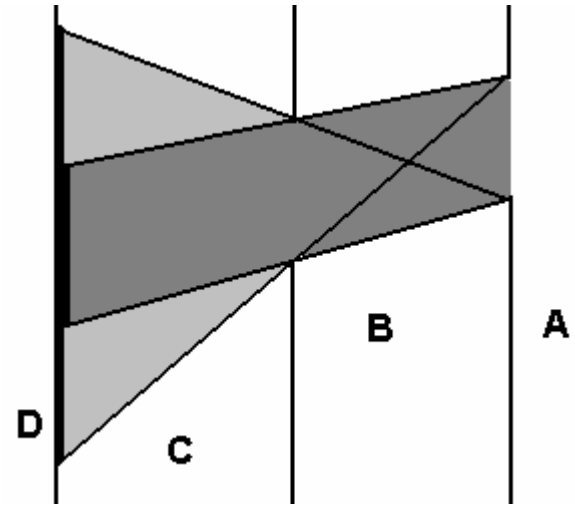
Megkaptuk a portal-okat. Egy utolsó, de szintén eléggé számításigényes feladatunk van még hátra. Meg kell határoznunk a portal-ok közötti láthatósági relációkat.

6.2.2 Láthatósági relációk meghatározása

A láthatósági relációk megkeresése az a folyamat, amikor azt határozzuk meg, hogy létezik-e az egyes térrészekben olyan pont, amelyből egy másik térrész valamely pontja látható. Azaz tudunk-e tetszőleges két ilyen pontot találni a térrész-párok között, hogy a közöttük húzott szakasz csak portal-on megy keresztül, és nem döf át sokszöget. Látható, hogy ez a láthatósági reláció reflexív, azaz ha az egyik térrészből átlátunk a másikba, akkor a másikból is az elsőbe.

Ilyen szakaszok meghatározása nem véges feladat, mivel a szakaszok száma végtelen lehet, mivel a végpontjaik száma is végtelen sok lehet. Ezért más megoldást választunk.

Továbbiakban a Penumbra jelenti az árnyékot, amely akkor keletkezik, ha egy fényforrással megvilágítunk egy tárgyat. Az Antipenumbra épp az ellentettje. Vegyünk egy papírlapot, és készítsünk rá egy lyukat. Az Antipenumbra a lyukon keresztüláramló fényt jelenti. Pontosabban az általa meghatározott teret. A lyuk tekinthető a portal analógiájának, a papírlap az a rész, amely körülveszi a portal-t. A



Antipenumbra

fényforrásunk nem pontszerű, azaz ha egy ilyen fényforrással megvilágítjuk a papírlapot, akkor az Antipenumbra rész nem lesz homogén. Az ábrán ezt szemléltetjük. A jobb oldalon látható a fényforrás. A középső, lyukas felület a papírlap. A bal oldalon egy olyan felület látható, ahol az árnyékot, és a megvilágított részt vizsgáljuk. A megvilágított rész érdekes számunkra. A megvilágított rész kétfelé bontható. Az egyik rész, amelyet vastagabb vonallal jelöltünk a bal oldali síkon valamint sötétszürkével jelöljük a fény által bejárt teret, az a teljesen

megvilágított terület, azaz a fényforrás bármely pontja összeköthető ezekbe a pontokba vezető szakasszal. A vékonyabb vonallal illetve világosszürkével jelölt rész olyan pontokat jelöl, amelyek a fényforrás nem minden pontjával köthetők össze, de ettől függetlenül megvilágított részben találhatók.

Mit is jelent ez a mi esetünkben? Az „A” szobában állunk. A fényforrásnak tekintjük azt a portal-t, amelyen keresztül az „A”-ból a „B” szobába nézhetünk át. A „B” szoba természetesen látható az „A” szobából. A „C” szoba látható, a konvexitási feltételek miatt. A meghatározás módját a „D” szoba esetén vizsgáljuk. A „D” szoba akkor és csak akkor látható ha van olyan összekötő portal a „C” és a „D” térrész között, amely közöttük található vastag vonalon van (lásd ábra). Az algoritmus tovább folytatható egy „E” jelű térrésszel, a fenti módon.

Tekintsük az alábbi algoritmust, amely rekurzívan bejárva a térrészeket, meghatározza a láthatósági relációkat egy adott térrészből:

```
CPVSCompiler::ProcessPVSFrom(Leaf SourceLeaf)
begin
      SourceLeaf.PVSMarkVisible(SourceLeaf)

      foreach Portal SourcePortal in SourceLeaf.AllPortals do
      begin
            TargetLeaf = SourcePortal.GetOtherLeaf(SourceLeaf);

            SourceLeaf.PVSMarkVisible( TargetLeaf );

            foreach Portal TargetPortal in TargetLeaf.AllPortals do
            begin
                  if TargetPortal = SourcePortal
                        continue;

                  if TargetPortal.Coplanar(SourcePortal)
                        continue;

                  RecursePVS(TargetLeaf, TargetPortal, SourcePortal);
            end
      end
end
```

A ProcessPVSFrom meghívása egy levél paraméterrel történik (SourceLeaf). Ez lesz az a levél, ahonnan a többi levél láthatóságát meg kívánjuk határozni. A SourceLeaf önmagából természetesen látható, amit be is állítunk. Meghatározzuk az összes olyan portal-t, amely SourceLeaf levélhez kapcsolódik. Minden egyes portal-hoz meghatározzuk a másik oldalán található olyan levelet (TargetLeaf), amely a SourceLeaf-ből szintén látható. A TargetLeaf összes portal-ja látható a SourceLeaf-ből, nem foglalkozunk a párhuzamos portal-okkal. Mivel a portal-ok láthatók, ezért minden olyan levél, amely a másik oldalukon található, felhasználható a rekurzív bejárás során. Ezt a következőkben ismertetjük.

Meghívjuk a rekurzív bejárást, amelynek paraméterként megadjuk a TargetLeaf-et (a kiindulási levéllel szomszédos levél), a TargetPortal-t (a kiindulási

```
CPVSCompiler::RecursePVS(TargetLeaf, TargetPortal, SourcePortal)
begin
      GeneratorLeaf = TargetPortal.GetOtherLeaf(TargetLeaf).

      SourceLeaf.PVSMarkVisible( GeneratorLeaf );

      foreach Portal GeneratorPortal GeneratorNode.AllPortals do
      begin

            if GeneratorPortal = TargetPortal continue;

            if SourcePortal.IsOnSameSide(GeneratorPortal, SourceLeaf)
                  continue;

            if TargetPortal.IsOnSameSide(GeneratorPortal, TargetLeaf)
                  continue;

            NewGenPortal = AntiPenumbraClip(GeneratorPortal,
                                    TargetPortal, SourcePortal);
            NewSrcPortal = AntiPenumbraClip(SourcePortal,
                                    TargetPortal, GeneratorPortal);

            if newSrcPortal.NumV = 0 or newGenPortal.NumV = 0
                  continue;

            RecursePVS(GeneratorLeaf, NewGenPortal, NewSrcPortal);
      end
end
```

levelet és a szomszédos levelet összekötő portal) és a SourcePortal-t (kiindulási levél). Tehát a TargetPortal másik oldalán található levél (GeneratorLeaf) látható a kiindulási levélből, a BSP-fa leveleinek konvexitása miatt. Vizsgáljuk meg a GeneratorLeaf összes portal-ját (GeneratorPortal). Vannak olyan portal-ok, amelyek biztosan nem láthatók illetve nem használhatók fel a rekurzió továbbfolytatására. Ezek a következők:

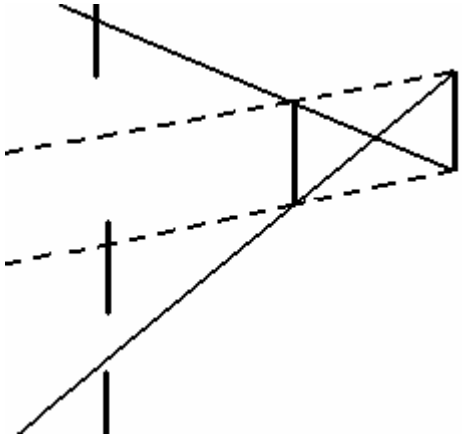
1. Ha a GeneratorPortal és a TargetPortal megegyezik
2. Ha a kiindulási portal ugyanazon oldalán található a kiindulási levél és a GeneratorPortal is
3. Ha a TargetPortal ugyanazon oldalán található a TargetLeaf és a GeneratorPortal is

Az első kizárás triviális, a második és a harmadik akkor következik be, ha a portal-ok és a levelek sorrendje nem megfelelő.

A következőkben kiszámíthatjuk, hogy valójában melyik GeneratorPortal használható fel a rekurzió továbbfolytatására. A fenti feltételeket teljesítő portal-ok már biztosan nem. Térjünk vissza a lyukas papírlap analógiájára. Először vizsgáljuk meg, hogy ha fényforrásnak tekintjük a SourcePortal-t, a lyuknak a TargetPortal-t, akkor az Antipenumbra tartományba esik-e a GeneratorPortal. Ha igen, akkor csak azt a részét tartsuk meg, amely ebbe a tartományba esik (NewGenPortal). Majd végezzük el ennek a műveletnek az inverzét is. Tekintsük fényforrásnak az eredeti GeneratorPortal-t, lyuknak most is a TargetPortal-t. A SourcePortal vizsgálata és vágása az ezek által meghatározott Antipenumbra tartományhoz képest történik

(NewSrcPortal). Az új kiindulási portal Antipenumbra -vágására azért van szükség, mert a TargetPortal-on keresztül a GeneratorPortal-ból nézve elképzelhető, hogy kisebb rész látszik. Ha az így keletkezett portal-ok nem tűnnek el, akkor ismételjük meg rekurzívan a fenti eljárást, azonban most a TargetLeaf szerepét a GeneratorLeaf fogja betölteni, a TargetPortal szerepét a NewGenPortal, a SourcePortal szerepét a NewSrcPortal.

Az AntiPenumbraClip metódusnak tehát három paramétere van. Az első, amelyet el akarunk vágni, a második jelzi a „papírlapon található lyukat”, a harmadik



Antipenumbra-vágás

a fényforrást, visszatérési értéke az elvágott portal. Működése: vesszük a fényforrás összes csúcspontját. Minden csúcsponthoz vesszük a lyuk egymás mellett levő összes csúcspont-párját. Minden esetben így három csúcspontot kapunk, amely meghatároz egy síkot. Csak azokat az így keletkezett síkokat tartjuk meg, amelyeknek a fényforrás és a lyuk ellentétes oldalán van. Az ábrán látható szaggatott vonallal jelzett síkok nem megfelelőek, csak a folytonos vonallal jelöltek. Az ábra bal oldalán látható három portal közül csak a felső kettő esik bele abba a tartományba, ami látható (ezek közül az egyik teljesen).

Mivel a láthatósági reláció reflexív, azért egy szimmetrikus mátrixot kapunk. A most tárgyalt algoritmus futási ideje felére csökkenthető, ha a mátrixnak csak az alsó vagy a felső háromszög mátrixszát számítjuk ki.

Most már csak el kell tárolnunk az adatokat, amit kiszámítottunk. Mint az előzőekben említettük egy nxn-es szimmetrikus mátrixot kapunk, ahol n a BSP-fa leveleinek száma. Ennek a mátrixnak elég csak az alsó vagy a felső háromszög mátrixszát eltárolni, sőt még le is tömöríthetjük az adatokat. Mivel jó esetben ez a mátrix viszonylag ritka, ezért RLE (Run Length Encoding) eljárással is hatékonyan tömöríthetjük az adatokat.

6.3 Bejárása – kirajzolása

A bejárás teljesen hasonló módon történik, mint a BSP-fa esetében. A BSP-fa bejárását ismertető fejezetben egy hasonló algoritmust közöltünk, mint ami itt

```
BSPTree::WalkTree()
Begin
      if not Frustum.IsBBoxInFrustum
            continue;

      if viewpoint*hyperplane >= 0 then
      begin
            if frontnode!= NULL then
                  frontnode.WalkTree()

            if PVS.IsLeafVisible(ViewersLeaf, this)
                  DrawPolygons()

            if backnode!= NULL then
                  backnode.WalkTree()
      else
      begin
            if backnode!= NULL then
                  backnode.WalkTree()

            if PVS.IsLeafVisible(ViewersLeaf, this)
                  DrawPolygons()

            if frontnode!= NULL then
                  frontnode.WalkTree()
      end
end
```

látható. Az egyik különbség a BSP-fákhoz kapcsolódik: megvizsgáljuk, hogy az aktuális csúcs befoglaló doboza a látótérbe esik-e. Ha nem esik a látótérbe, akkor a csúccsal és a csúcs alatt található részfával nem kell foglalkoznunk. A másik különbség, hogy a kirajzolás előtt megvizsgáljuk, hogy ki kell-e rajzolnunk a térrészt. Ezt a PVS által meghatározott mátrix egy elemének kiolvasásával tehetjük meg. Pontosabban megnézzük, hogy a nézőpontot tartalmazó levél oszlopában illetve ebben az oszlopban található, az aktuális levél sorához tartozó pozíción 1 vagy 0 elem áll-e. Ha nem levélben vagyunk, akkor a vizsgálat kimenetele mindenképpen hamis, mivel csak levél esetében érdekes ennek vizsgálata. Azt vizsgáljuk, hogy ahhoz a levélhez képest, ahol a nézőpont található az aktuális levél látható-e, és csak igaz válasz esetén rajzolunk.

A nézőpont helyét, mint a BSP-fák esetében mindig, a fenti függvény minden egyes meghívása előtt előre meg kell határoznunk, azaz el kell döntenünk, hogy melyik levélben található. Ezt egy $\log N$ -es rekurzív algoritmussal meghatározhatjuk.

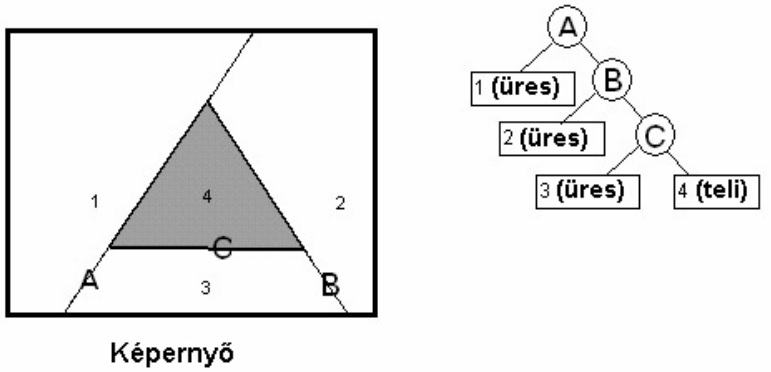
A kirajzoló algoritmus komplexitása logaritmikus, hasonlóan a BSP-fáknál alkalmazottal.

6.4 Alkalmazásának történeti áttekintése

A PVS valós idejű alkalmazása a Quake nevű játékkal jelent meg. Azóta a legtöbb háromdimenziós belső tereken alkalmazott láthatóság-vizsgáló algoritmus ezt a megközelítést használja.

De hogy is jutottak el a Quake fejlesztői arra a következtetésre, hogy ezt a megoldást használják? A Quake elődje a Doom 2D-s BSP-fával és nem forgatható kamerával dolgozott, ezért felülrajzolás nélkül megoldhatták az eltakart sokszögek eltávolítását, mint erről már szóltunk korábban. A Quake 3D-s BSP-fát használ és a kamera bármilyen szögben és irányba nézhet, sőt fénytérképek és kezdetleges valós idejű árnyékokat is megjelenített. Tehát a Quake legnagyobb problémája a felülrajzolás volt. A látótérből kilógó térrészek eltávolítása nem probléma, mint ahogy ezt már a BSP-fa tárgyalásánál említettük.

A Quake-et először úgy tervezték, hogy közelről távolra rajzolja ki a BSP-fa leveleit és tartsa számon, hogy a képernyő mely részeire rajzoltunk eddig, ezt Beam-tree (Sugár-fa) segítségével tette meg. A Beam-tree végső soron a BSP-fa által meghatározott, sokszöglapokkal határolt térrészek (Beam-ek) képernyőre vetítésével (a nézőpontba) áll elő.



Beam-tree

Először az egész képernyő szabad, majd ahogy bejárjuk a BSP-fa leveleit és sokszögeit, úgy adjuk hozzá a Beam-tree-hez is a sokszögeket. Ha minden sokszög elfogyott vagy az egész képernyő foglalt (a Beam-tree homogén), akkor terminál az algoritmus. A sokszögek Beam-tree-hez való hozzáadása nagyon időigényes feladat, valamint sokszor nem vált homogénné a Beam-tree soha sem.

Kiderült, hogy másik megoldást kell keresni, mert nem elég hatékony a megoldás. A következő megoldás az lehetett volna, hogy a képernyő 8x8-as régiói

felé sugarakat lövünk ki és megvizsgáljuk melyik leveleket érintette először a sugár. Ez a megoldás is időigényes volt, valamint kis levelek illetve sokszögek a sugarak közé eshettek, így nem kaphattuk volna egzakt képet.

Egy következő megoldás lehetett volna egy olyan puffer használata, amely a képernyő pixeleinek foglaltságát jelzi 1 biten, azonban ezzel az a probléma, hogy minden sokszöget végig kellett volna futtatni ezen a pufferen, ami szintén nem hatékony.

Szóba került még az a megoldás, hogy a sokszögeket vetítsük le a képernyő síkjára, majd az így keletkezett vízszintes szakaszokat (amelyek távolságértéke a lényeges paraméter) metsszük el egymással, és csak a fennmaradó szakaszdarabokat rajzoljuk ki. De sajnos ez is időigényes feladat.

Tehát maradt az a megoldás, hogy valahogyan a portal-okat alkalmazzuk a feladat megoldására. A tradicionális portal-motorokkal az a probléma, hogy időigényes feladat a portal-ok és sokszögek vágása, mint ahogy ezt az ezzel a témakörrel foglalkozó fejezetben említettük.

A nagy áttörés a PVS alkalmazása volt, amely ugyan nem ad egzakt megoldást, de ez elhanyagolható a láthatóság gyors meghatározásához képest. A felülrajzolás tehát minimális és egy levél láthatósága egy feltételvizsgálattal meghatározható. Ennek az ára természetesen a bonyolult előfeldolgozó eljárás. Ezt azonban csak egyszer kell elvégeznünk, utána már csak élveznünk kell a magas képfrissítés nyújtotta előnyöket.

Irodalomjegyzék

[Abr97] Michael Abrash, *Graphics Programming Black Book*, Special Edition, Coriolis Group Books, 1997

[AWPH97] Andrew J. Willmott és Paul S. Heckbert, An Empirical Comparison of Radiosity Algorithms, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1997

[CRL99] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest: *Introduction to Algorithms*, magyar nyelven: Algoritmusok, Műszaki Könyvkiadó, alkotószerkesztõ: Iványi Antal, 1999

[DPLCG] David P. Luebke and Chris Georges, *Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets*, University of North Carolina at Chapel Hill, 1995, ISBN: 0-89791-736-7

[Elm99] Niklas Elmqvist, *Introduction to Portal Rendering*, Chalmers Medialab, 1999

[MDB00] Mark. de Berg, *Linear Size Binary Space Partitions for Uncluttered Scenes*, Algorithmica, 2000

[MHAV] Markus Hadwiger, Andreas Varga, *Visibility Culling*, Technical University of Wien

[MikL03] Mikko Laakso, *Potentially Visible Sets (PVS)*, Helsinki University of Technology, 2003

[NathW] Nathan Whitaker, *Extracting Connectivity Information From A BSP Tree, http://www.exaflop.org/docs/naifgfx/naifebsp.html*

[SRE01] Samuel Ranta-Eskola, *Binary Space Partition Trees and Polygon Removal in Real Time 3D Rendering*, Uppsala University, 2001

[STCQ01] Seth J. Teller, Carlo H. Séquin, *Visibility Preprocessing For Interactive Walkthroughs*, University of California at Berkeley, 2001

[Szir03] Dr. Szirmay-Kalos László, Antal György, Csonka Ferenc, *Háromdimenziós grafika, animáció és játékfejlesztés*, ComputerBooks, Budapest, 2003, ISBN: 963 618 303 1

[HZh98] Hansong Zhang, *Effective Occlusion Culling for the Interactive Display of Arbitrary Models*. The University of North Carolina at Chapel Hill, 1998