Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatika kar
Irányítástechnika és Informatika Tanszék

BSP-fák használata játék-motor fejlesztésében

dr. Szirmay-Kalos László
konzulens

Boromissza Gergely
szerző

2006

Feladatkiírás

Nyilatkozat

Alulírott Boromissza Gergely, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

……..

Tartalomjegyzék

Összefoglaló

Munkám célja a BSP[1]-struktúra és alkalmazási területeinek bemutatása, kiemelt figyelmet fordítva az általam és mások által elért eredményekre, annak játékfejlesztésben való szerepére.

A videokártyák rohamos fejlődése ellenére még napjainkban is nagy szükség van az előfeldolgozás során egy olyan komplex struktúra felépítésére, mely a megjelenítés és az ütközésdetektálás mellett számos egyéb nagy számításigényű feladat futási idejére hat kedvezőleg a vizsgálandó elemek számának drasztikus csökkentésével. Ezen struktúrák közül az egyik talán legismertebb és legelterjedtebb a BSP, melynek még ma is rengeteg változata megtalálható a legkülönbözőbb számítástechnikai területeken, de főleg az FPS[2] játékokban.

A dokumentumot egy rövid történeti áttekintéssel kezdem. Az elvi alapok ismertetése után részletezem, hogyan lehet egy virtuális világhoz automatikusan BSP-fát generálni, és hogyan lehet azt a játék során a megjelenítés és egyéb vizsgálatok gyorsítására felhasználni. Bemutatok néhány a gyakorlatban igen jól használható és éppen ezért elterjedt módszert az alap struktúra kiterjesztésével a futási idő csökkentésére.

Elemzem az általam készített játék-motorban a struktúra használatának eltéréseit az elvi alapoktól. Az alkalmazásokban használt egyéb módszerek és formátumok bemutatására is sor kerül. Részletezem a fejlesztés során felmerült nehézségeket és végezetül a továbbfejlesztési lehetőségekről is említést teszek.

---

[1] Binary Space Partitioning – bináris térfelosztás

[2] First Person Shooter – belső nézetes lövöldözős játék, melyben a főhős testébe bújhatunk bele

Abstract

The aim of my work is to introduce the BSP[1]-structure and its usage through my and others' results especially on the field of game-development.

Though nowaday's rapid improvement of videocards, it is still needed to build a complex structure in preprocess time, which can reduce the amount of run-time on different fields like visualisation and collision detection with reducing the amount of elements, which should be tested. The BSP is possibly the most common structure, which versions can be still found in different computer applications, especially in FPS[2] games.

I start the document with the historical introduction. After a short overview of the theoretical basics, I explain how to build the BSP-structure automatically for a virtual world, and how it can be used to improve the speed of visualisation and collision detection. I introduce some extension of the basic method, that can be easily used to reduce the runtime of the applications.

I analyze the difference of the methods in the game-engine made by myself. There's also an introduction of the methods and formats used in the applications. I give full details of the difficulties I met during the improvement and at last I mention the possibilities of the method's improvement.

[1] Binary Space Partitioning

[2] First Person Shooter – a gametype, in which the user can play from the main character's print of view

Zusammenfassung

Das Ziel meiner Diplomarbeit ist die Vorstellung der BSP-Struktur und ihrer Anwendungsbereiche, wobei der Schwerpunkt sowohl in den Ergebnissen als auch in derer Rolle bei der Spielentwicklung liegt.

Trotz der raschen Entwicklung der Videokarten braucht man in unseren Tagen noch immer in der Vorbearbeitung den Aufbau einer Komplexstruktur, die neben der Darstellung und dem Zusammnestoß-detektieren auch positiv auf die Laufzeit der rechnungsreichen Aufgaben mit dem drastischen Senken der zu untersuchenden Elementenzahl auswirkt. Eine dieser bekanntesten und gebrauchtesten Strukturen ist BSP[1], die auch noch heute zahlreiche Abarten in den verschiedensten Bereichen der Computertechnik besitzt, besonders aber bei den FPS[2] Spielen.

Ich beginne die Dokumentation mit einem kurzen historischen Überblick. Nach dem Bekanntmachen der prinzipiellen Grundlagen beschreibe ich detailliert, wie man zu einer virtuellen Welt automatisch einen BSP-Baum generieren kann, und wie er im Spiellauf bei der Beschleunigung der Darstellung und anderer Untersuchungen angewandt werden kann. Ich stelle einige in der Praxis gut verwendbare und eben deshalb verbreitete Methoden mit der Ausdehnung der Grundstruktur vor, die die Laufzeit sinken.

Ich analysiere in dem von mir hergestellten Spielmotor die Abweichungen des Strukturgebrauchs von den prinzipiellen Grundlagen. Andere Anwendungsmethoden und Vorstellung der Formaten kommen in meiner Arbeit auch an die Reihe. Ich gehe auf die Details hinein, die bei

---

[1] Binary Space Partitioning – Binarisch Raumaufteilung

[2] First Person Shooter – Schießspiel vom inneren Aspekt, wir können in die Haut des Helden schlüpfen

der Entwicklung aufgetaucht sind, und zum Schluss erwähne ich die Möglichkeiten der Weiterentwicklung.

Bevezetés

A módszer létjogosultsága

A videokártyák rohamos ütemben fejlődnek és manapság már több millió háromszöget képesek megjeleníteni másodpercenként, de ezzel párhuzamosan nő a felhasználók igénye is. Elvárás a minél inkább élethű környezet megjelenítése dinamikus elemek használatával, mindemellett egy elfogadható FPS[1] produkálása és a minél rövidebb és ritkábban előforduló betöltési idő a különböző pályarészek között.

Ezen feltételek mellett nincs idő az egyre részletesebben kidolgozott teljes terep megjelenítésére, az összes háromszög, illetve poligon kirajzolására. Bár a mai modern videokártyák többsége a felesleges – az avatar[2] szemszögéből nem látható – elemeket automatikusan eldobja, ezen adatok átküldése és kiszűrése mégis értékes időt vesz igénybe, ezzel csökkentve a megjelenítés sebességét.

A pályák minél kisebbre való felszabdalása nem jelent megoldást a problémára, ilyenkor ugyanis gondoskodnunk kell a pályarészek közötti átjárásról, a dinamikus átváltásról és a dinamikus elemek áthelyezéséről. Ha ezeket a részeket nélkülöznénk, egysíkúvá tennénk a játékot, megszabnánk a felhasználónak, hogy mikor merre mehet, és hogy mit csinálhat. Ezzel együtt a pályarészek közötti területek is unalmassá válnának.

A megjelenítendő háromszögek szűrését végezhetnénk a gép processzorának segítségével real-time[3]-ban, így átvéve a feladat végrehajtását a

---

[1] Frames Per Second – másodpercenként megjelenített képkockák száma; a program sebességének egyik legfontosabb mérőszáma

[2] A felhasználó, játékos személye a virtuális világban

[3] Valós idejű – a számítások és kiértékelések a program futása során hajtódnak végre

videokártyától. A CPU[1] műveletei egyrészt nem erre vannak kiélezve, nem úgy mint a GPU[2]-é, másrészt a processzorra manapság rengeteg más feladat súlya nehezedik, úgy mint a játéklogika, a mesterséges intelligencia és a fizikai motor számítási igényeinek elvégzése (ez utóbbi a fizikai kártyák megjelenésével és elterjedésével bizonyos mértékig csökkenthető lesz).

Így lehetséges, hogy a játékfejlesztés témakörében még jelenleg, és valószínűleg a jövőben is az egyik legfontosabb problémakör a nem látható felületek kiszűrése (HSR[3]) előfeldolgozási időben.

Erre a feladatra ad elfogadható, bővíthető és továbbfejleszthető megoldást a BSP módszer, mely a program futása előtt automatikusan épít fel egy olyan egyéni adatstruktúrát, ami jelentős plusz információval szolgál a terep megjelenítésére vonatkozóan, ezzel csökkentve a kirajzolandó poligonok számát.

A módszer a későbbiekben részletezett struktúrájából kifolyólag nem csak a megjelenítés gyorsítására alkalmas, a nem látható felületek átküldésének megspórolásával. Lehetőség nyílik a számítások mennyiségének csökkentésére a sugárkövetés, ütközésdetektálás, CSG[4] logika és egyéb területeken is.

Habár a klasszikus módszert önmagában már egyre kevesebb helyen alkalmazzák, jelenleg a BSP-struktúrák használatának rengeteg változata megtalálható a legkülönbözőbb számítástechnikai területeken. Ezért is választottam munkám témájának ezen kutatási terület körüljárását, és különböző szemléltető programok írását a módszer demonstrálására.

---

[1] Central Processing Unit – központi feldolgozó egység, vagyis a gép processzora

[2] Graphical Processing Unit – grafikus feldolgozó egység, vagyis a videokártya processzora

[3] Hidden Surface Removal – nem látható felületek kiszűrése

[4] Constructive Solid Geometry – konstruktív tömörtest geometria

A dokumentum felépítése

Munkámban bemutatom a BSP módszer kialakulását és annak felhasználási területeit, érdemlegességét, kifejezett hangsúlyt fektetve annak játékokban való felhasználásáról (1. fejezet).

A munkám és a módszer megértéséhez szükséges grafikai és matematikai alapfogalmakat a 2. fejezetben taglalom.

A 3. fejezetben bemutatom a klasszikus BSP-struktúrát és példával szemléltetve megyek végig a fa felépítésének folyamatán.

A 4. fejezetben a felépített struktúra használatával mutatom be a játéktér különböző megjelenítésének és az ütközésdetektálásnak a módját a 3. fejezet példáját használva fel.

Az 5. fejezetben a BSP-struktúra befoglaló dobozos kiterjesztéséről és a megjelenítés sebességére tett pozitív hatásáról ejtek pár szót.

A 6. fejezet célja az olvasó megismertetése egy továbbfejlesztett, ún. PVS BSP-struktúrával, melyet a mai napig nagy előszeretettel alkalmaznak.

A programjaimban használt módszerek, illetve fájl-formátumok ismertetése a 7. fejezetben történik.

A 8. fejezetben mutatom be a módszer felhasználásával általam készített szemléltető programok felépítését és működési elvét.

A bemutatásra kerülő módszerek tanulmányozásából levont következtetésekről, valamint a továbbfejlesztési, továbblépési lehetőségekről a 9. fejezetben teszek említést.

1. Történeti áttekintés[1]

A BSP-struktúrát eredetileg Schumaker, Brand, Gilland és Sharp fejlesztette ki 1969-ben a poligonok gyors mélységi sorrendezésére. Ekkor ezt a módszert „prioritásos listá”-nak nevezték. Lényege, hogy kirajzoláskor egy kisebb prioritású elem, sosem írhat felül egy magasabbat.

Sutherland, Sproull és Schumaker 1974-ben továbbfejlesztette az ötletet és az elemeket prioritási egységekbe rendezte, melyekből bináris fát épített. A csomópontokban vágósíkok találhatóak, míg a levelekben a prioritás szerint sorrendezett szektorok.

Fuchs, Kedem és Naylor 1980-ra fektette le a ma ismert BSP algoritmus alapköveit.

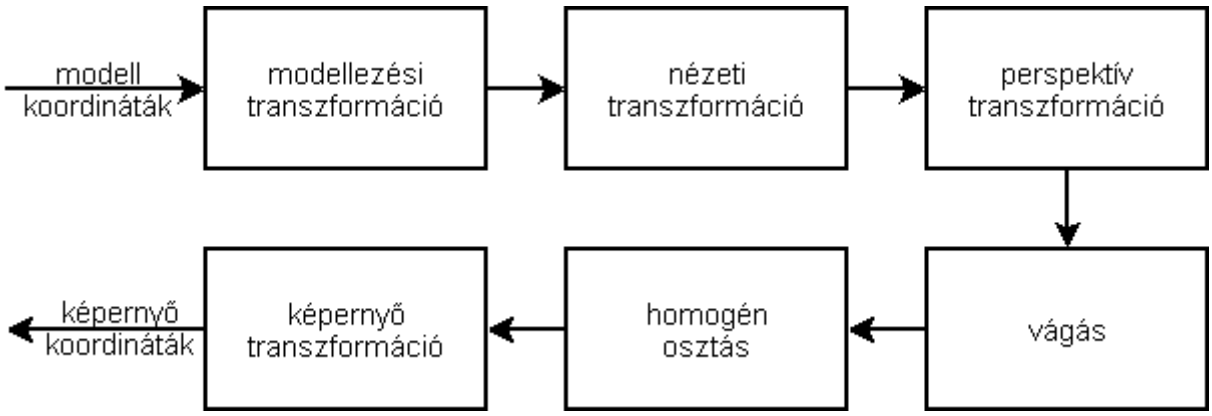
Az 1991-ben megjelenő Wolfenstein 3D sikerén felbuzdulva, az ID™ elsőként fejlesztette ki a BSP módszeren alapuló HSR módszert, melyet először a Doom-ban (1993) és a Quake 1-ben (1996) alkalmaztak. A cég által megjelentetett játékok motorjait más programfejlesztők is felhasználták alkalmazásaikban. A módszert azóta több helyen is továbbfejlesztették, de az alapjai még sok mai játékban is megtalálhatóak úgy, mint a Quake 2, Quake 3, Half-life, Jedi Knight, stb.

[1] [2] 1-2. oldal

2. Grafikai és matematikai alapfogalmak

2.1 A grafikai megjelenítés alapjai

A számítógépes grafikában háromdimenziós modellekkel dolgozunk. A megjelenítés során a teret a felhasználó szemszögéből kell a képernyőre vetítenünk úgy, hogy a térhatás élménye megmaradhasson. A feladat elvégzését a sebesség érdekében inkrementális képszintézis segítségével végezzük el. A világ képernyőre való leképezéséhez az 1. ábrán látható nézeti csővezetéken keresztül kell átküldeni a virtuális tér minden egyes elemét.



1. ábra[1]: Az inkrementális képszintézis nézeti csővezetéke

A módszer csak sokszögekre (vagyis háromszögekre) alkalmazható, így a csővezetékbe küldés előtt szükséges az összes modell tesszellációja[2].

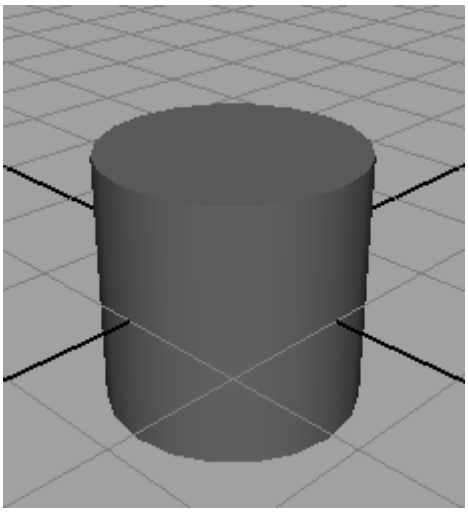
A háromszögesítés után a modelljeinket (2. ábra) azok modellezési transzformációinak segítségével (eltolás, forgatás, skálázás) egységesen a világ koordináta rendszerébe képezzük le (3. ábra), így kerül minden elem a virtuális világ megfelelő helyére.

---

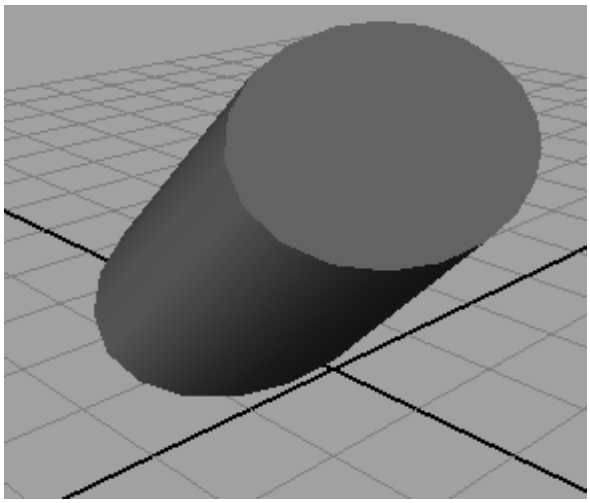
[1] [4] 191. oldal

[2] Sokszögekre (általában háromszögekre) bontás, a modellek alakjainak (pl. gömb) közelítésével

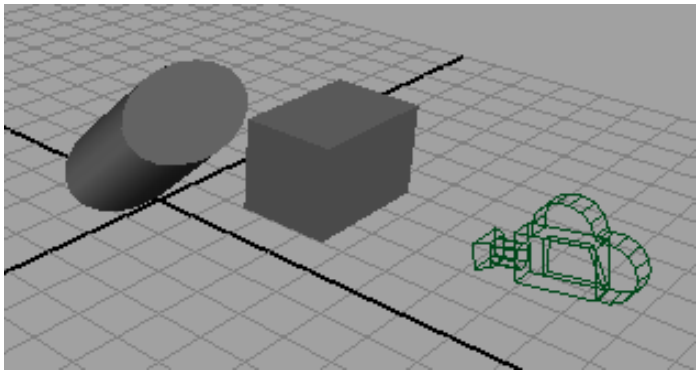
A nézeti transzformációt felhasználva a még mindig háromdimenziós terünket (4. ábra) beforgatjuk a kamera pozíciójának, illetve nézeti irányának megfelelően (5. ábra).



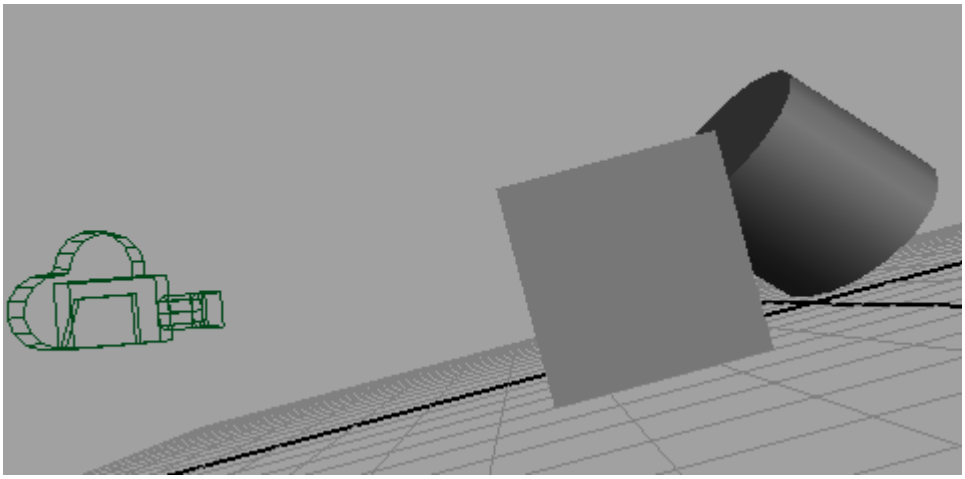
2. ábra: Háromszögesített modell



3. ábra: Modellezési transzformáció után

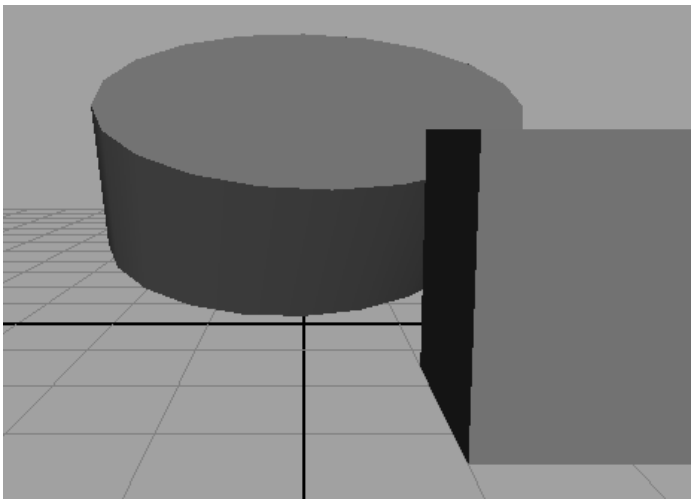


4. ábra: Nézeti transzformáció előtt



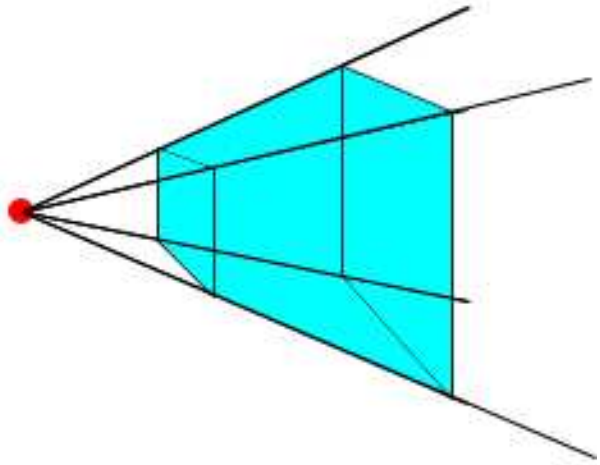
5. ábra: Nézeti transzformáció után

Ezek után következik a perspektív transzformáció, melynek végrehajtása után a felépített világunkat immáron az elhelyezett kamera pozíciójából szemlélhetjük meg (6. ábra).



6. ábra: Perspektív vetítés után

A transzformált világ jelenlegi állapotában – a perspektív vetítés után, de még a homogén osztás előtt – hajtjuk végre az egyik legfontosabb szűrési feladatot, a vágást. Ennek segítségével a kamera nyílásszöge (FOV[1]) által meghatározott négy, illetve az első és a hátsó vágósík által meghatározott két, vagyis összesen hat sík által határolt téglalap alapú csonka gúlára vágunk (7. ábra).



7. ábra: Látószög a hat vágósíkkal

[1] Field of View – látószög

Mivel a szűrésre a perspektív vetítés után kerül sor, a jelenlegi világunk vágását csonka gúla helyett téglatestre kell végeznünk. Ennek lényege, hogy az aktuális pozícióból nem látszódó és a túl közel vagy túl távol lévő elemeket eldobhassuk. Ezen kívül a látószögből részben kilógó elemeket teljesen a látómezőben lévőkkel helyettesítjük. A feladat elvégzésére a legoptimálisabb Cohen-Sutherland[1] vágást alkalmazni.

A feladat teljesítése után elvégezhetjük a homogén osztást és következhet a képernyőre való transzformálás is, mely a virtuális modellünket immáron képernyő koordinátákban tartalmazza kiegészítve egy Z mélységi értékkel.

Ebben a végső stádiumban hajtunk végre még néhány szűrési feladatot, melyek közül a legfontosabb a hátsó lapok eldobása (BFC[2]).

A számítógépes grafikában a háromdimenziós térben egy poligonnak „két oldala van”, egy előlapja és egy hátlapja. Az előlap a felhasználó számára látható, míg a hátlap nem kerül megjelenítésre. Ennek jelentősége, hogy a pályatervezés során egy poligont mindig csak egy irányból nézhetünk. Nem valószínű például, hogy egy szoba tapétáját, vagy egy focilabdát megnézünk majd a fal, illetve a labda belsejéből is. A háttal lévő poligonok kiszűrése kis számításigényű feladat, és átlagban a megjelenítendő háromszögek 50%-a megspórolható vele.

A poligon előlapja az, amerre a normál vektora mutat. Tehát egy elem leírásában nagy szereppel bír a csúcsok megadásának sorrendje, hiszen ez határozza meg az eltérő oldalak orientációját (lásd következő alfejezet).

---

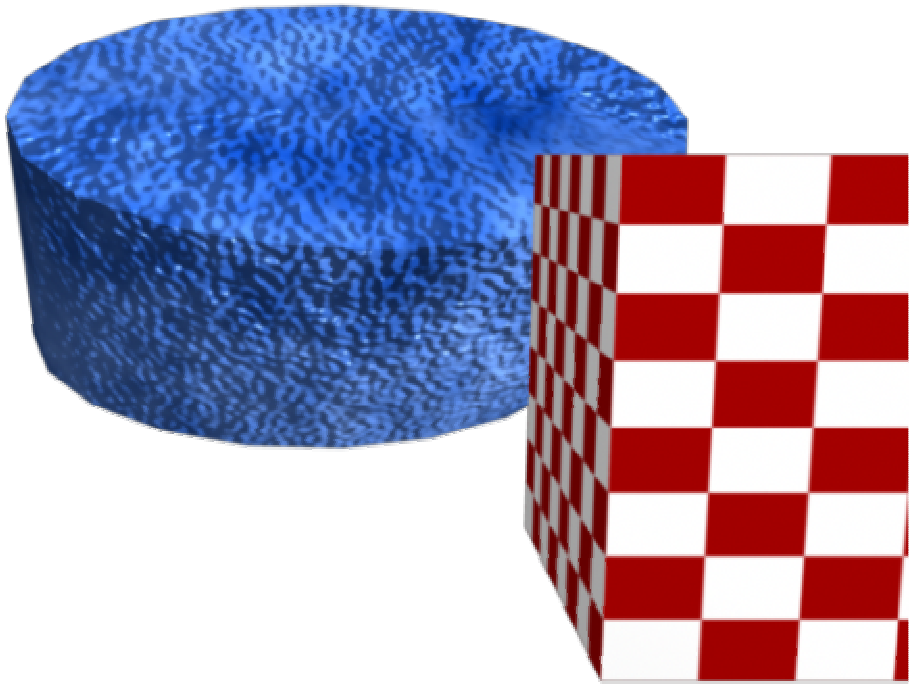
[1] Részletesen: [4] 198.oldal

[2] Back Face Culling – triviális hátsólap eldobás

Az egyszerűség kedvéért a későbbiekben mi mindig jobb sodrású koordináta-rendszerekkel (CCW[1]) számolunk, ahol nem, ott erre külön felhívjuk a figyelmet.

A soron következő szűrési feladat a takarási probléma feloldása. Ehhez használjuk fel a képernyő transzformáció után megtartott Z mélységi koordinátát. A feladatra a Z-buffer algoritmus[2] ad optimális megoldást. Ez mindenegyes pixelre eldönti, hogy az arra a pontra vetülő elemek közül melyik van a legközelebb a kamerához és csak annak a színét tartja meg.

A szűrések sora itt véget ér, a megmaradt pixelekre történik ezek után az árnyalási feladatok elvégzése, valamint a fényforrások, az anyagok, textúrák, stb. kezelése, míg végül a 8. ábrán látható eredményt nem kapjuk.



8. ábra: Megjelenített kép

[1] Counter Clock Wise – órajárás irányával ellentétes körüljárás

[2] Részletesen: [4] 202. oldal

Az itt leírásra került megjelenítési módszert használja mind az OpenGL, mind a DirectX. Jól látható, hogy a szűrési feladatok mindegyike csak a perspektív vetítés után kerül végrehajtásra, így a különböző transzformációk a virtuális világ minden egyes elemére végrehajtódnak. Ez egy részletesen kidolgozott pálya esetében (néhány millió háromszög) rengeteg számítási kapacitást vesz igénybe, ami a megjelenítés jelentős lassulásához vezet.

Ezért van szükség egy előre felépített struktúra használatára, mint amilyen a BSP-fa is, ami a nézeti csővezetékbe való bekerülése előtt szűri ki a virtuális világ amúgy sem látszódó háromszögeit.

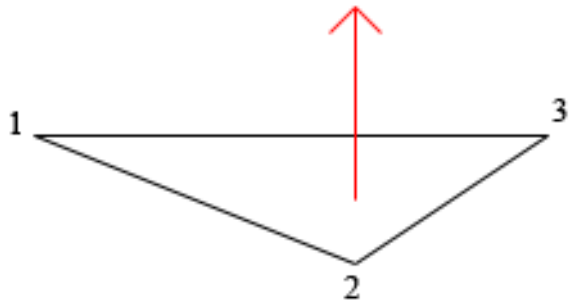
2.2 Matematikai primitívek

Ebben a szakaszban a matematikai alapfogalmak háromdimenziós térre koncentrálva kerülnek bemutatásra. Ahol számottevő különbség mutatkozik a kétdimenziós térrel szemben, arra külön kitérek. A fogalmak, műveletek, egyenletek és számítások könnyűszerrel átültethetők bármely más dimenziójú térbe is azok általánosságából kifolyólag.

Mivel az előző fejezetben már említettük, hogy a nézeti csővezetékbe vezetés előtt tesszellációra van szükség, az alábbiakban a fogalmakat háromszögekre mutatjuk be. A számítások többsége általában tetszőleges poligonra is alkalmazható.

Megemlítendő, hogy egy háromdimenziós pálya kétdimenziós térbe leképezhető, ha a terünk minden egyes háromszöge párhuzamos valamely tengellyel (magassági dimenzió). Ilyenkor az adott koordináta eldobásával egy alacsonyabb dimenziójú és könnyebben kezelhető térbe léphetünk át. Ha játéktérben gondolkozunk, egyszerűen csak nézzünk rá felülről. Ekkor a háromszögek helyett dolgozhatunk szakaszokkal.

A 9. ábrán egy jobbsodrású rendszerben megadott csúcspontokból álló háromszög normál vektorát figyelhetjük meg:

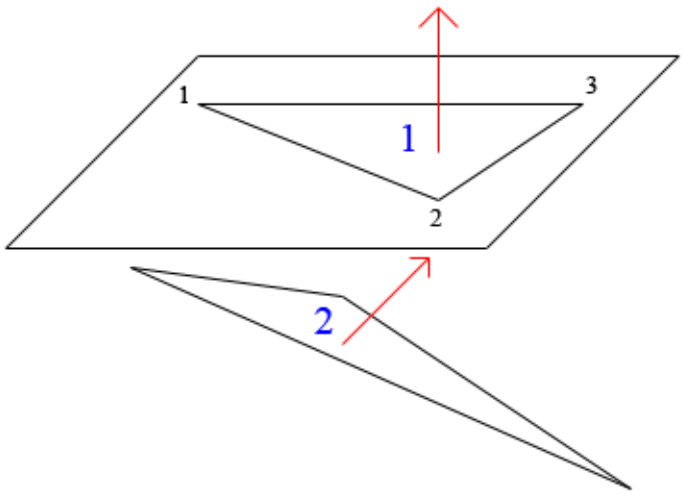


9. ábra: Háromszög normáljával

A háromszög szemből nézve látható, míg hátulról nem kerül megjelenítésre a triviális hátsólap eldobása miatt (mint azt az előző szakaszban már láthattuk).

Kétdimenziós térben a szakasz normál vektora a szakasz csúcsainak megadási sorrendjéhez képest előre meghatározott oldalán (bal vagy jobb) helyezkedik el.

Vezessünk be újabb fogalmakat, mint a szembe vele (front) és a mögötte (behind), melyet a háromszögek egymáshoz képest vett viszonyának jelölésére használunk. Ehhez vegyük a következő ábrán látható szemléletes példát.



10. ábra: Két háromszög viszonya

Az ábrán adott két poligon közül 2-essel szemben van az 1-es, hiszen az a normál vektora irányában helyezkedik el, míg 1-esnek mögötte van a 2-es, mert normál vektorával ellentétes irányban található. Jól látható, hogy ez a viszony nem kölcsönös.

Annak eldöntésére, hogy egy tetszőleges poligon a másik melyik oldalán helyezkedik el, a másik összes csúcspontjának vizsgálata után tudunk egyértelmű választ adni. A viszony eldöntésére álljon itt most egy pseudokód:

```
whichSide(Polygon poly1, Polygon poly2)
  behindNum←0, frontNum←0
  for each point of poly2
    side=whichSide(poly1, poly2.point)
    if (side=FRONT) frontNum←frontNum+1
    if (side=BEHIND) behindNum←behindNum+1
  if (frontNum>0 AND behindNum=0) return FRONT
  if (frontNum=0 AND behindNum>0) return BEHIND
  if (frontNum>0 AND behindNum>0) return INTERSECT
  return ON
```

Tehát ha a másik poligon minden pontja a vizsgált poligonnal szemben helyezkedik el, akkor a poligon is vele szemben, ha minden pontja mögötte, akkor mögötte helyezkedik el, ha van pontja vele szemben és mögötte is, akkor metszi, különben rajta van azon.

A pont poligonhoz vett viszonyának eldöntésére pár egyszerű fogalom ismeretére van még szükségünk.

A háromszög normálját az alábbi képlet szerint számolhatjuk pontjainak segítségével:

$$\overrightarrow{normal} = \begin{pmatrix} A & B & C \end{pmatrix}$$
$$\overrightarrow{normal} = \left(\vec{v}2 - \vec{v}1\right) \times \left(\vec{v}3 - \vec{v}1\right)$$

vagyis két oldalának meghatározott sorrendben vett keresztszorzataként.

Síkja egyenletének felírásához tudnunk kell annak általános alakját:

$$\overrightarrow{plane} = \begin{pmatrix} A & B & C & D \end{pmatrix}$$
$$Ax + By + Cz + D = 0 \quad ,$$

amiben A B C éppen a sík normál vektora, D pedig az origótól vett távolságának az ellentettje, ami az alábbi képlet szerint egyszerűen számolható a poligon egy tetszőleges pontjának segítségével:

$$D = -\overrightarrow{normal} \cdot \vec{v}1$$

Ezek után egy pont síkhoz vett viszonyának az eldöntéséhez csak az alábbi egyszerű képletet kell kiértékelnünk:

$$\vec{v} = \begin{pmatrix} x & y & z & 1 \end{pmatrix}$$
$$scalar = \overrightarrow{plane} \cdot \vec{v}$$

ahol v, vagyis a pontunk, homogén koordinátákkal adott.

Ezek alapján a pont poligonhoz vett viszonyának eldöntésére az alábbi pseudo kód használható:

```
whichSide(Polygon poly, Point point)
  side←poly.plane*point
  if (side>0) return FRONT
  if (side<0) return BEHIND
  return ON
```

A sík és a pont szorzata éppen a pontnak a síktól a normál vektorának irányában mért előjeles távolságát adja. Ha ez a szorzat pozitív, akkor a pont a síkkal szemben található, ha pedig negatív, akkor mögötte (0 esetén éppen rajta). Példaként vehetjük a már bemutatott 10. ábrát, ahol jól látható, hogy a 2. poligon az 1. mögött helyezkedik el, mivel annak minden csúcspontja az 1. síkja mögött található.

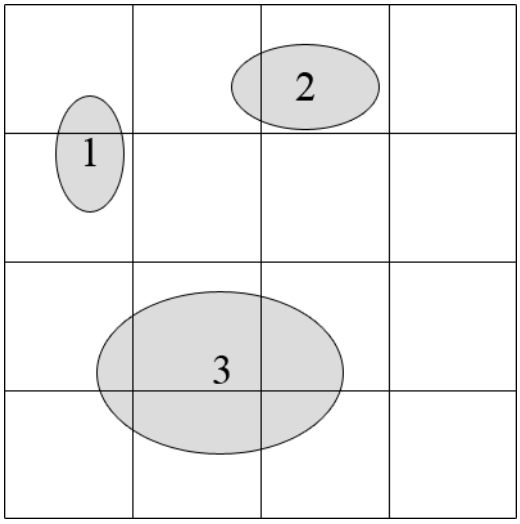
Ezen alapismeretek birtokában már sikerrel abszolválhatjuk a következő, BSP struktúra felépítéséről szóló fejezetet.

3. BSP-struktúra és felépítése

3.1 Térfelosztó módszerek[1]

A legegyszerűbb térfelosztó algoritmus a szabályos térfelosztás. Ekkor az objektumteret minden tengely mentén egyenlő részre osztjuk, és minden egyes térrészre meghatározzuk, mely objektumok tartoznak bele (11. ábra). Ez a módszer pazarló, a k dimenziós teret n felosztás esetén n^k részre osztja. A térrészek többsége, az objektumok nem egyenletes eloszlása miatt, üres lesz, így a kihasználtság alacsony szintű.

Az előbbi és a később bemutatásra kerülő módszerek esetén is előfordulhat, hogy egy objektum több térrészhez is tartozik, de ez egyik algoritmus esetén sem jelent problémát.

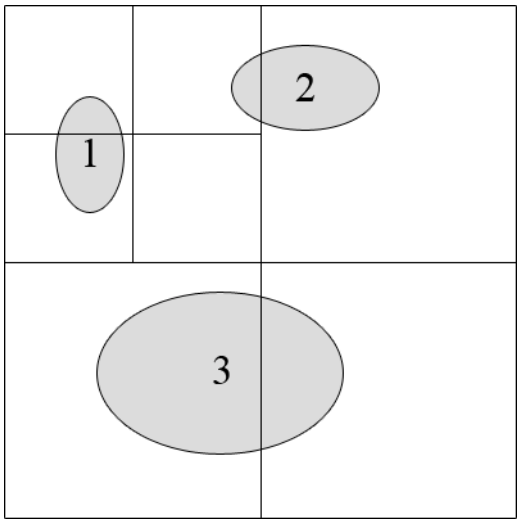


11. ábra: Egyenletes térfelosztás

A módszer továbbfejlesztése az oktális térfelosztás, melynél a teret 2^k egyenlő részre vágjuk (k továbbra is a tér dimenzió száma), majd minden egyes térrészt addig osztunk tovább, míg az objektumok száma az adott részben egy megfelelő érték alá nem csökken (12. ábra). A módszer nagy előnye, hogy a már eleve üres, vagy csak pár objektumot tartalmazó térrészek felosztása hamar abbamarad, mert független a többi résztől. A

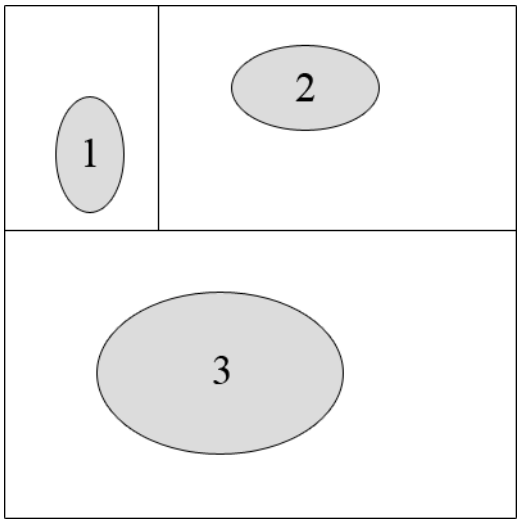
[1] [4] 174. oldal alapján

felesleges felosztások és terek száma így nagymértékben csökkenthető, ezzel növelve a kihasználtságot.



12. ábra: Oktális térfelosztás

A kd térfelosztó algoritmus módszer lényege, hogy a tereket tetszőleges, a tengelyekkel párhuzamos síkkal osztjuk két részre, ügyelve arra, hogy az objektumokat lehetőleg ne vágjuk el, és az eloszlásuk is minél egyenletesebb legyen (13. ábra).



13. ábra: kd térfelosztás

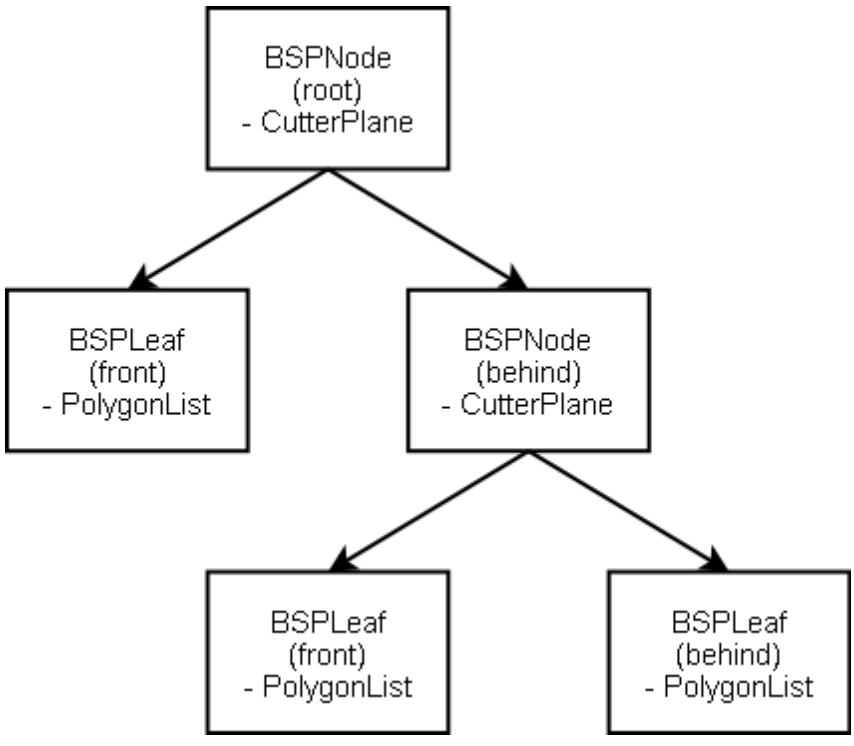
A módszer neve onnan ered, hogy a k dimenziós teret mindig k-1 dimenziós hipersíkokkal osztjuk két részre. A folyamatot a térrészekben található objektumok számára adott korlátozás elérésekor hagyjuk abba. Az algoritmus nagy előnye, hogy a teret egy lépésben nem 2^k, hanem csupán 2 részre vágja, így csökkentve és optimalizálva a keletkező térrészek számát.

A vágósíkot is mi választjuk, az esetek többségében nem a felezőt használjuk. Ezzel csökkenthetjük az elvágott poligonok számát és azok eloszlását is egyenletesebbé tehetjük a részek között.

A felosztásra a továbbiakban mi a tér egy poligonjának síkját fogjuk használni. A vágósíkot lehetőség szerint úgy szemeljük ki, hogy a felosztás után keletkező két térrészben a poligonok száma közel azonos eloszlású legyen, és hogy a vágással minél kevesebb háromszöget szabdaljunk fel. A felosztást addig folytassuk, amíg a poligonok által alkotott alakzat minden térrészben konvex nem lesz, vagyis amíg mindegyik mindegyikkel szembe nem néz.

3.2 A BSP-struktúra

A BSP módszer alkalmazása során a felépített struktúrát bináris fás ábrázolással szemléltethetjük a legjobban (14. ábra). Ebben a csomópontok reprezentálják az algoritmus során választott vágósíkokat, illetve fogják össze a sík által felosztott két részteret. A fa levelei a felosztás végén keletkező konvex térrészek.



14. ábra: BSP-fa struktúrája

Ha az adott elem egy levél, akkor tartalmazza az adott térrészhez tartozó poligonokat. Ha csomópont, akkor tartalmazza a vágósíkot, amivel két részre osztottuk a teret, illetve a 2 részteret. Az egyik, ami a poligonnal szemben helyezkedik el, a másik, ami mögötte. A struktúra pseudo-kódja az alábbi:

```
struct BSPNode : BSPElem
  Plane cutter
  BSPElem behind
  BSPElem front

struct BSPLeaf : BSPElem
  PolygonList polygons
```

3.3 Vágósík választása

Ez a pont a faépítés talán egyik legkényesebb része, mert nem áll rendelkezésünkre nem-exponenciális költségű, minden esetben a legoptimálisabb vágó síkot kiválasztó algoritmus. Két megközelítés létezik.

Az egyik, hogy válasszunk egy olyan síkot, ami a legjobb közelítéssel vágja két egyenlő számú elemet tartalmazó részre a teret. Ennek hátránya, hogy a választott sík akár az összes többi elemet is két részre vághatja, így megduplázva poligonjaink számát.

A másik lehetőség, hogy mindig a legkevesebb új poligont létrehozó vágó síkot válasszuk. Ennek hátránya viszont, hogy a fánk nem lesz elegendően kiegyensúlyozott, így annak használatával nem érünk el jelentős gyorsulást a megjelenítés terén.

Az optimális választás valahol a két megközelítés között van.

```
ChooseCutter(PolygonList polygons)
  bestrate←0.0
  bestintersect←MaxValue
  minrate←0.8
  while no cutter
    for each cutterPlane in polygonsPlanes
      frontNum←0, behindNum←0, intersectNum←0
      for each polygon is polygons
        side←whichSide(cutterPlane, polygon)
        if side=INTERSECT
          intersectNum←intersectNum+1
        if side=FRONT OR side=ON
          frontNum←frontNum+1
        if side=BEHIND
          behindNum←behindNum+1
      if frontNum>behindNum
        rate←behindNum/frontNum
      else
        rate←frontNum/behindNum
      if rate>=minrate AND
      (intersect<bestintersect OR
      (intersect=bestintersect AND rate>bestrate))
        bestintersect←intersect
        bestrate←rate
        cutter←cutterPlane
    minrate←minrate/2
    if minrate<0.001
      minrate←0
  return cutter
```

Mi most az előbbi pseudo-kódunkban az első megközelítést használjuk, és az azonos méretű térrészekre vágó poligonok közül a kevesebb új poligont létrehozót választjuk ki.

Az algoritmusban minden egyes iteráció elején meghatározzuk a minimum elvárásunkat a keletkezendő két térrész poligonszám arányára vonatkozóan. Minden egyes iterációban végigmegyünk az összes poligon síkon és azzal két részre vágjuk a teret. Kiválasztjuk a síkot, ami a legkevesebb vágást eredményezi, azon belül pedig azt, amelyik a két legegyformább méretű részre vágja a teret, feltéve ha megfelel a minimum elvárásunknak. Ha nem találunk megfelelő vágósíkot, a következő iterációban felezzük a minimum kritériumot. Adott elvárás elérése után nem foglalkozunk tovább a kritériummal, a legkevesebb vágást eredményező síkot választjuk, biztosítva az algoritmus véges idejű futását.

3.4 A tér felosztása

Ha már kiválasztottuk a vágósíkot, nincs más hátra, mint ezzel két részre osztanunk a teret, olyan poligonokra, amelyek a vágóval szemben, illetve mögötte helyezkednek el. Ha a vágónk metszene egy másik poligont, akkor a metszett poligont két részre vágjuk a metszésvonal mentén, így már be tudjuk sorolni az egyik felét az egyik, a másik felét a másik térrészbe (ez utóbbi kódja nem kötődik szorosan a BSP-fákhoz, így azt nem részletezem).

Az algoritmus pseudo-kódja:

```
cutDimension(Plane cutter, PolygonList polygons,
   PolygonList behindList, PolygonList frontList)
  for each polygon of polygons
    side←whichSide(cutter, polygon)
    if side=ON OR side=FRONT
      frontList←frontList+polygon
    else if side=BEHIND
      behindList←behindList+polygon
    else
      intersect(cutter,polygon)
      behindList←behindList+behindPolygon
      frontList←frontList+frontPolygon
```

3.5 Az építés vége

Mikor is hagyjuk abba a tér további részekre osztását? Erre egy nagyon egyszerű feltételt adhatunk: konvex-e a kapott térrészben található poligonok által alkotott alakzat? Ez egy logikus döntésnek tűnik, mivel konvex térrészben nem tudjuk további részekre bontani a teret, nincs olyan poligon sík, ami vágást eredményezne. Továbbá konvex alakzatban nem fordulhat elő takarás, és a megjelenítés szempontjából az egyik legfontosabb célkitűzés az volt, hogy takart poligonokat ne rajzoljuk ki feleslegesen. Egy háromszögekből álló alakzat konvexitásának eldöntésére ad algoritmust az alábbi pseudo-kód, melynek szerves részeit már az előbbiekben tárgyaltuk:

```
isConvex(PolygonList polygons)
  convex←true
  for each plane in polygonsPlane
    for each polygon in polygons
      side←whichSide(plane, polygon)
      if side=INTERSECT OR side=BEHIND
        convex=false
  return convex
```

Tehát a konvexitás feltétele, hogy minden háromszög minden másikkal szemben legyen. A kódból kitűnik, hogy minden poligont kétszer hasonlítottunk össze egymással. Ez nem meglepő, hiszen mint már az alapoknál is szó volt róla, a poligonok viszonya nem kölcsönös.

3.6 BSP-fa felépítése

Most már csak az maradt hátra, hogy a BSP-fánkat felépítsük az eddig leírt részek segítségével:

```
BuildNode(PolygonList polygons)
  if isConvex(polygons)
    return new BSPLeaf(polygons)
  else
    cutter←ChooseCutter(polygons)
    cutDimension(cutter, polygons, frontList,
      behindList)
    front←BuildNode(frontList)
    behind←BuildNode(behindList)
    return new BSPNode(cutter, front, behind)
```

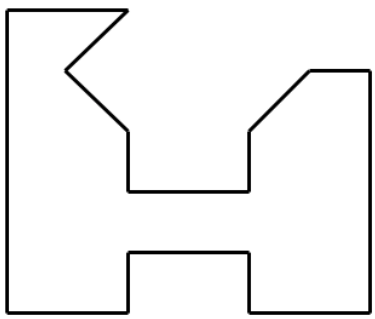
A fenti függvény bemenetként kapja a terünk minden egyes poligonját, visszatérésül pedig a felépített fa gyökerét szolgáltatja. Az építés során, amennyiben konvex térrészhez érkezünk, a megmaradt poligonokkal létrehozunk egy levelet. Ha konkáv a poligonok által alkotott alakzat, akkor kiválasztjuk a megfelelő vágót közülük, ezzel két részre vágjuk a teret, majd ezekkel rekurzívan tovább építjük az adott résztereket. A visszakapott két részteret egy csomópontban fogjuk össze. Így a 14. ábrán vázolt egyszerű fa-szerkezethez jutunk.

A fejezetben leírt módszerek alapján átfogó képet kaphattunk a struktúra teljes felépítésének folyamatáról és a pseudo-kódok alapján lehetőségünk nyílik az algoritmus egyszerű implementálására is.

3.7 Példa

Most egy egyszerű kétdimenziós példán nézzük végig az építés folyamatát.

A 15. ábrán látható játékterünk adott. A szakaszok normáljai természetesen a pálya közepe fele néznek, az átláthatóság kedvéért ezeket mégsem jelöltük.

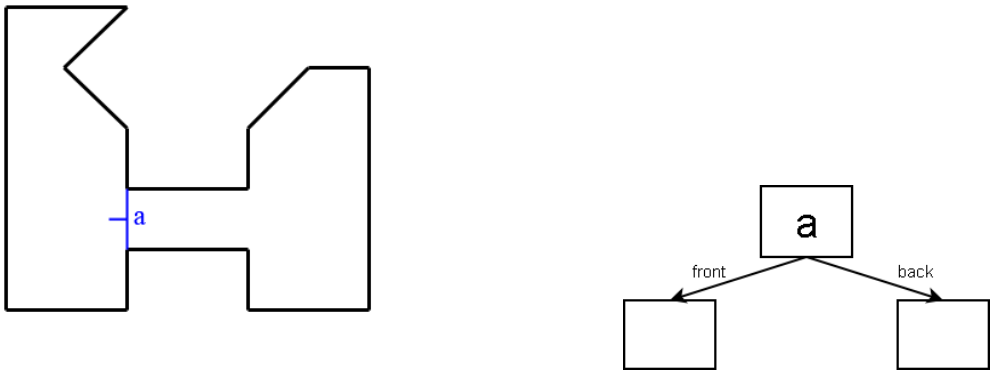


15. ábra: Példa virtuális világra

Mivel ránézésre látszik, hogy a pálya alakja konkáv, próbáljuk most megkeresni ebben a térben az ideális vágót. Az előző szakaszban leírt módszert alkalmazva a 16. ábrán is látható „a” szakasz tűnik a legésszerűbb

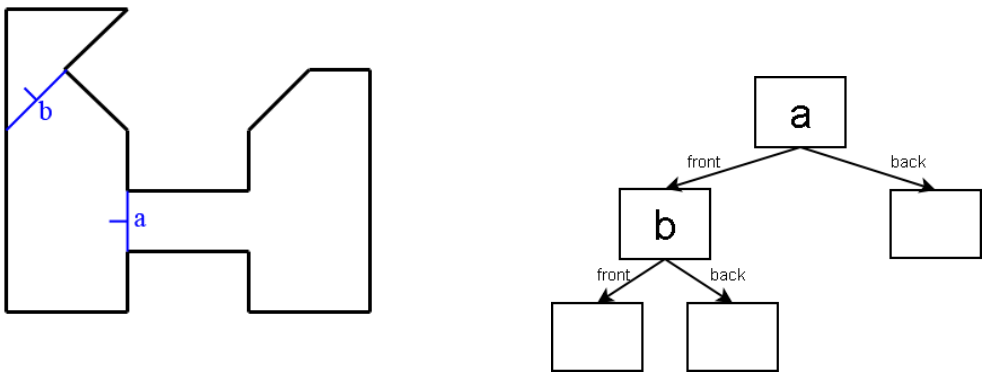
választásnak, hiszen ez egyetlen másik élt sem metsz el, és két közel azonos méretű részre osztja a teret.

Az ábra jobb oldalán láthatjuk a felosztást reprezentáló fát, melybe a választott él és a két résztér leendő helye már be is került.



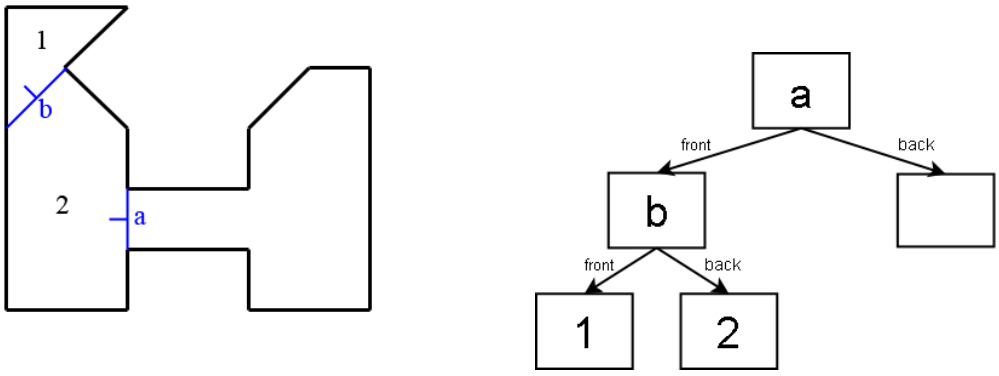
16. ábra: Példa vágósík választásra 1.

Foglalkozzunk most a vágóval szemközti, vagyis a bal oldali térrel. Mivel ez a rész még mindig nem konvex, szükség van újabb vágó keresésére. Most válasszuk a 17. ábrán is feltüntetett „b” élt. Látható, hogy ez a pálya bal oldali falát ugyan két részre osztja, mégis ez a választás eredményezi a két rész legkiegyensúlyozottabb eloszlását (5, illetve 3 fal).



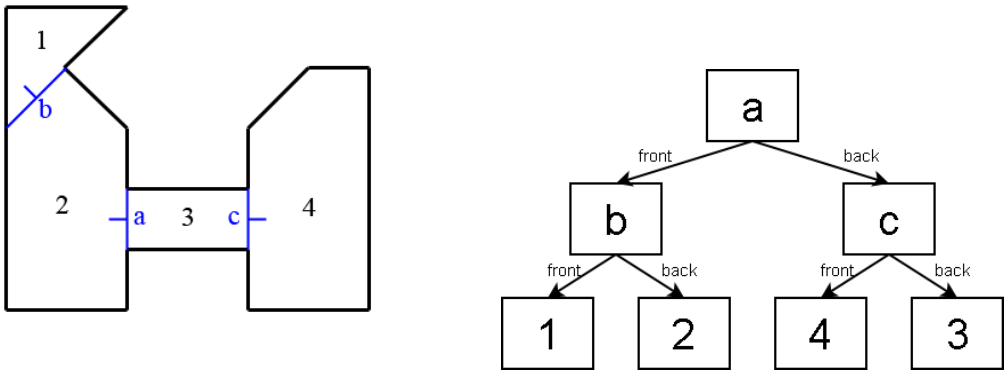
17. ábra: Példa vágósík választásra 2.

Menjünk tovább és vizsgáljuk meg a „b” él két oldalán lévő részt. Az ábrán ránézésre is látszik, hogy ezek már konvex alakzatot alkotnak, így nincs szükség azok további felbontására. Számozzuk meg ezeket a tereket és helyezzük el a fánk megfelelő leveleibe (18. ábra).



18. ábra: Példa térrész létrehozásra

Visszamenve újra a gyökérbe és az „a” él mögötti részre is végrehajtva az előbb leírt lépéseket, eredményül a 19. ábrán látható térfelosztáshoz, illetve felépített struktúrához jutunk.



19. ábra: Példa felépített BSP-struktúrára

Most hogy már fel tudjuk építeni a BSP-struktúránkat, nézzük, hogy mire és hogyan is tudjuk használni.

4. BSP struktúra használata

4.1 Aktuális térrész megkeresése

Szinte minden vizsgálathoz szükségünk van az aktuális térrész megkeresésére, ahol mi, vagyis az avatar tartózkodik a virtuális világban. Mivel a felépített struktúra többek között pont ennek gyors megtalálására lett kiélezve, ezen kérdés eldöntésére az alábbi egyszerű pseudo-kód szolgál:

```
actElem←root
while (actElem IS A BSPNode)
  if WhichSide(actElem.cutter, actPos)=Behind
    actElem←actElem.behind
  else
    actElem←actElem.front
```

A ciklust a fánk gyökerére indítjuk. Az eljárás megvizsgálja, hogy levélben vagyunk-e, ha igen, visszaadjuk a hivatkozást, megtaláltuk a térrészt, amiben aktuálisan tartózkodunk. Amennyiben csomópontban vagyunk, ha a vágó felől vagyunk, akkor a vágó felőli, ha a vágó mögött vagyunk, akkor a vágó mögötti térrészre folytatjuk a keresést.

Megemlítendő még, hogy ezen ciklus lefuttatása – bár nem nagy erőforrás igényű – mégis felesleges, amennyiben az avatar nem mozdult el előző helyéről. Ez esetben a térrész, melyben tartózkodik, az előző frame-belivel megegyező.

4.2 Ütközésdetektálás

A CD[1] elvégzése a megjelenítés után az egyik legnagyobb számításigényű feladat. A BSP-struktúra használatával ennek mértéke jelentősen csökkenthető.

Az ütközésvizsgálat folyamatát nem részletezném, csak pár szóban összefoglalnám. Ütközés akkor lép fel, ha az avatar előző és aktuális pozícióját összekötő szakasznak van metszéspontja valamely felülettel, és ez a metszéspont a poligonon belül helyezkedik el. Ezen vizsgálat elvégzését nyilván elegendő az aktuális térrészben végezni, hiszen más pályarészek építőelemeivel lehetetlen az ütközés fellépése.

Amennyiben az avatar nem mozog, pontosabban fogalmazva, amennyiben nem változik meg a helyzete, természetesen az egész vizsgálat elvégzésére nincs szükség. Ha viszont elmozdul, akkor szükség lenne a pálya minden egyes elemével való ütközésének a vizsgálatára. A BSP-fa használatának segítségével ez a vizsgálati kör leszűkíthető az aktuális térrészre.

Ha más szektorból érkezett az avatar, akkor az előző pályarész ellenőrzésére is szükség van. Feltételezzük, hogy a játékos egy frame alatt az előző és a jelenlegi pályarészen kívül más szektorokat nem érint. Ezzel a feltételezéssel élhetünk, ha a server frame[2] nem túl alacsony (általában 20 körüli érték megfelelő), vagyis az avatar pozícióját megfelelő időközönként frissítjük, és a mozgási sebesség sem túl nagy. Ez esetben a frame-enként megtehető út hossza nem fogja túllépni a felépített BSP-struktúrában egymáshoz esetlegesen közel lévő térrészek távolságát.

---

[1] Collision Detection – ütközésdetektálás

[2] Előre definiált érték, mely a megjelenítéstől független számítások elvégzésének (mint a fizikai motor és a mesterséges intelligencia) másodpercenkénti számát határozza meg

A módszerrel a vizsgálandó háromszögek számát, bármely más ismert gyorsítási eljáráshoz képest, jelentős mértékben redukáltuk.

A vizsgálat elvégzésének más módjáról még szó esik az utolsó fejezetben.

4.3 Back-to-front[1] megjelenítés

A hardveres Z-buffer megjelenése és elterjedése előtt szükség volt a poligonok sorrendezésének szoftveres úton való megvalósítására. Ennek lényege, hogy a képernyőn az egymást takaró felületek közül mindig a felhasználóhoz legközelebbi kerüljön megjelenítésre, megoldható legyen a hátsó részek kitakarása. A szoftveres Z-buffer teszt elvégzése rengeteg időt vesz igénybe, így annak implementálása nem jelent megoldást a problémára.

A felépített BSP-struktúra segítségével ezt a tesztet teljes egészében elhagyhatjuk. A fa hátulról előre történő bejárásával mindig a felhasználótól legtávolabb eső felületektől haladunk a közelebbiek felé. Így a közeli felületek kirajzolásakor a memóriában felülírjuk az adott pixeleken található távolabbi felületekből származó értékeket, ezzel helyesen megoldva a takarási problémát.

[1] Hátulról előre történő

A hátulról előre történő bejárás, illetve kirajzolás pseudo-kódja:

```
Draw (BSPElem elem)
  if elem IS A BSPLeaf
    DrawPolygons (elem.polygons)
  else if WhichSide(elem.cutter, actPos)=Front
    Draw (elem.behind)
    Draw (elem.front)
  else
    Draw (elem.front)
    Draw (elem.behind)
```

A függvényt megjelenítéskor a fánk gyökerére hívjuk meg. Amennyiben levélhez érkezünk, kirajzoljuk az adott levél összes poligonját. Ha csomópontnál vagyunk, akkor amennyiben a vágósíkkal szemben vagyunk, a vágósík mögötti térrészt jelenítjük meg elsőként, majd a vele szemben lévőt, ellenkező esetben pont fordított sorrendben. Röviden fogalmazva: mindig a távolabbi rész megjelenítésével kezdünk.

Megjegyzendő, hogy a függvény mindazonáltal alkalmas az adott térrész megkeresésére is, mivel futásának legvégén a legközelebbi, tehát az aktuális térrészt jeleníti meg.

Mint azt az aktuális térrész megkeresésénél már leírtuk, amennyiben az avatar helye nem módosul a virtuális világban, nem csak az aktuális térrész, de a többi térrész viszonya sem változik. Így a térrészek sorrendezési vizsgálatát sem kell minden egyes frame-ben elvégezni, ha egy listában letároljuk azt. Ezek után a vektort elegendő csupán térrész változtatáskor frissíteni, illetve megjelenítéskor a listában szereplő megfelelő sorrendben kirajzolni azt.

4.4 Front-to-back[1] megjelenítés

A hardveres Z-bufferek megjelenésével a back-to-front bejárás jelentőségét vesztette, a módszer előnye hátrányára fordult, mivel a Z-bufferbe és a szín-bufferbe történő írások száma e sorrendezés mellett a legmagasabb.

A fa elölről hátulra történő bejárása során a közelebbi térrészek megjelenítése felől haladunk a távolabbiak felé. A Z-buffer vizsgálat miatt a távolabbi felületek pontjai nem írják felül a közelebbiekét, így csökkenthető a memóriaírások száma.

A program pseudo-kódja a back-to-front bejárásával teljesen megegyező, ezért azt nem részletezem, egyedül a térrészek megjelenítésének sorrendjét kell minden esetben felcserélnünk.

A front-to-back bejárás egyetlen hátránya, ami miatt azt manapság egyáltalán nem használják, hogy az alfás felületeket[2] hibásan jeleníti meg. Ennek oka, hogy a Z-buffer szűrés miatt a távolabbi pontok színe automatikusan felülíródik, nem mosódik össze a már memóriában található közelebb lévő felületével. Ezen felületek helyes kirajzolásához a back-to-front módszer alkalmazása szükséges, mely a közelebbi felületekből származó színnel nem felülírja, hanem összemossa az előző, távolabbi poligonból származó színértékkel.

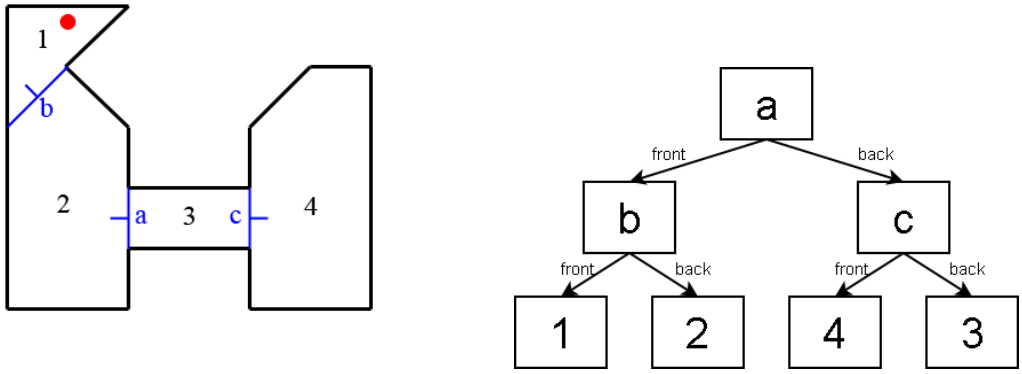
---

[1] Elölről hátulra történő

[2] Áttetsző, illetve átlátszó

4.5 Példa

Emlékezzünk vissza az előző fejezetben bemutatott példára, és tegyük fel, hogy az avatar a 20. ábra bal oldalán feltüntetett pontban tartózkodik.



20. ábra: Példa aktuális térrész keresésére, illetve front-to-back megjelenítésre

Keressük meg első lépésként az aktuális térrészt. Az avatar az „a” síkkal szemben helyezkedik el, tehát lépjünk az ábra jobb oldalán is látható fában „front” irányba. A következő vizsgálat eredményeképp azt kapjuk, hogy a játékos a „b” síkkal is szemben helyezkedik el, lépjünk tehát ismét „front” irányba. Levélhez érkeztünk, így sikeresen megkaptuk, hogy a felhasználó aktuálisan az 1-es térrészben tartózkodik.

Az avatar szektoron belüli elmozdulásakor nyilvánvalóan elegendő az ütközés vizsgálata az aktuális, esetünkben az 1-es, térrészben, hiszen például a 3-as szektor falaival nem léphetünk interakcióba. Példánkban ez a vizsgálatok számát a negyed, általános esetben n-ed (ahol n a térrészek száma), illetve szektorhatár átlépéskor n/2-ed részére csökkenti. Ez a számítások elvégzésének jelentős gyorsulásához vezet.

A továbbiakban nézzük a front-to-back megjelenítést. A módszer kísértetiesen hasonlít az aktuális térrész megkereséséhez. Az avatar az „a” síkkal szemben helyezkedik el, ezért először az azzal szembeni, majd az a mögötti térrészt rajzoljuk ki. Az „a”-val szemközti térrészben az avatar a „b” síkkal szintén szemben helyezkedik el, ezért ismét a szemközti, majd utána a hátsó térrészt rajzoljuk ki. A vele szemközti térrész egy levél, így

az 1-es térrész kirajzolásra kerül. Ezután vizsgáljuk a „b” mögötti térrészt. Ez szintén a fa egy levele, így azt is megjelenítjük. Lépjünk egy szinttel feljebb, az „a” sík mögötti térrészre. Ezt a teret a „c” sík osztja újabb 2 részre, vizsgáljuk meg, ennek mely oldalán tartózkodunk. Azt találjuk, hogy mögötte, így először a mögötte lévő részt rajzoljuk ki, ami nem más, mint a 3-as térrész, majd a szemköztit, ami pedig a 4-es szektor.

Így a megjelenítés sorrendje: 1-2-3-4.

A pályát szemügyre véve, valóban ez a pályaelemeknek a távolság függvényében vett sorrendje az avatar pillanatnyi pozíciójából nézve.

Végiggondolhatjuk a back-to-front bejárást is, aminek eredményeképp nem túl meglepő módon az alábbi sorrendet fogjuk kapni: 4-3-2-1.

5. BB BSP-struktúra

Az előző fejezetben bemutattuk, hogyan tudjuk a BSP-fák segítségével egyszerűsíteni az ütközésdetektálás folyamatát, illetve milyen módon tudjuk a megjelenítést segíteni a felületek sorrendezésének szoftveres megvalósításával. A bevezetésben elhangzott HSR feladat megoldásáról még nem esett szó. Ennek megvalósításához szükségünk van a BSP-fában további plusz információk tárolására, melyek ismertetése ebben és a következő fejezetben történik.

5.1 Befoglaló testek

A befoglaló testek egyszerű geometriája miatt azok láthatósága és az azokkal való ütközés könnyűszerrel eldönthető, ezért alkalmazzák őket nagy előszeretettel. Lényege, hogy a modellben, illetve a pályán logikusan kiválasztott elemeket összefogjuk, és köréjük egy egyszerűsítő testet húzunk, amelyen az összes elemünk belül helyezkedik el. Ezek után a vizsgálatokat végezhetjük az egyszerűsítő testre. Ha a vizsgálat nem lép interakcióba a testtel, akkor egyik általa tartalmazott elemmel sem fog, hiszen azok a testen belül helyezkednek el. Ha van interakció a befoglaló testtel, akkor a vizsgálatok elvégzése minden egyes befoglalt elemre is szükséges.

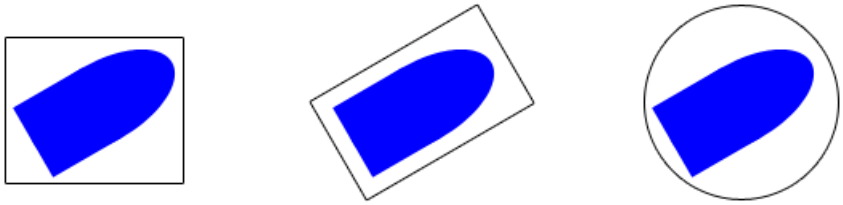
A befoglaló testeknek rengeteg típusa létezik, én most csak néhányat említek.

A befoglaló doboznak (BB[1]) két típusa van. Az egyik a tengelyekkel párhuzamos (AABB[2]), egyszerűen alkalmazható, minimum és maximum

[1] Bounding Box – befoglaló doboz

[2] Axis Aligned Bounding Box – tengelyekkel párhuzamos befoglaló doboz

csúcspontjával definiálható (21. ábra bal oldala). Egyszerűségéből kifolyólag nem mindig a legoptimálisabb.



21. ábra: Befoglaló testek típusai

Az előző módszernél biztosabb vizsgálati eredményeket produkál az OOBB[1] módszere (21. ábra közepe), mely nem a tengelyekkel párhuzamosan, hanem az objektum orientációjának megfelelően határozza meg annak befoglaló dobozát. Ennek használata annak körülményes megadása és nehéz alkalmazása miatt nem igazán népszerű.

Sokan szívesen alkalmazzák a 21. ábra jobb oldalán látható BS[2] módszerét. Ennek megadása egy pont és egy sugár segítségével történik. A számítások elvégzése mind közül ezzel a legegyszerűbb. Hátránya egyrészt, hogy felépítése során a gyökvonásból adódhatnak pontatlanságok, másrészt, hogy nem elég pontosan illeszkedik az elnyújtott modellekhez.

Az 1. táblázatban a különböző módszereket sorrendeztem. Jól látható, hogy nincs egyértelműen legjobb, az alkalmazástól függ, hogy az adott területen melyik a legmegfelelőbb.

	AABB	OOBB	BS
Kezelhetőség	2	3	1
Illeszkedés	2	1	3

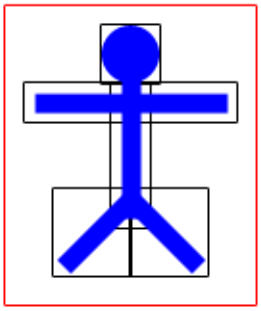
1. táblázat: Befoglaló testek rangsorolása

[1] Object Oriented Bounding Box – objektum orientált befoglaló doboz

[2] Bounding Sphere – befoglaló gömb

Gyakran alkalmazzák az itt felsorolt és egyéb módszereket együttesen is. Nagyon jól használható például az AABB és a BS módszer közösen.

A leírt befoglaló testekből gyakran építenek hierarchikus modellt. Ezen modellekre tett egyetlen megkötés, hogy a hierarchiában alul álló testnek a fölötte lévőn teljes egészében belül kell lennie (22. ábra). Ha a hierarchiában minden szinten ugyanolyan testekkel dolgozunk, akkor ez magától adódik. Ekkor egy vizsgálat elvégzésekor először a hierarchiában legfelül lévővel történő elemzés szükséges, majd ennek kimenetelétől függően haladunk tovább lefele vagy adunk egyértelmű választ.



22. ábra: Hierarchikus AABB

A továbbiakban mi hierarchikus AABB-t fogunk használni, de a módszerek bármilyen más befoglaló testre is kiterjeszthetőek.

5.2 A fa kiegészítése

A fa felépítése után határozzuk meg minden egyes térrész, vagyis levél, illetve minden egyes csomópont befoglaló dobozát. A levelekre ez az adott pseudo-kód implementálását jelenti:

```
min←max←polygons.first.apoint
for each polygon of polygons
  for each point of polygon
    for each axis
      if point.axis<min.axis
        min.axis←point.axis
      if point.axis>max.axis
        max.axis←point.axis
```

Mint az előző részben már említettük, egy AABB definiálható két ellentétes csúcsának koordinátáival, a minimummal és a maximummal. Ezt kezdetben inicializáljuk a térrész egy poligonjának tetszőleges pontjával. Ezek után menjünk végig a térrész összes sokszögén, annak összes csúcsán, és ha valamely tengely mentén kisebb, illetve nagyobb értéket találunk, mint a jelenlegi, akkor a minimum, illetve a maximum megfelelő koordinátáját az adottéra cseréljük. A vizsgálat elvégzése után a minimum és maximum koordináták határozzák meg a befoglaló dobozunk két ellentétes csúcspontját, amin belül helyezkedik el az adott szektor összes eleme.

A csomópontokra is hasonlóan tudjuk meghatározni a befoglaló dobozt, a két gyerek AABB-jének maximuma, illetve minimuma segítségével. Jól látszik, hogy a testek meghatározása egy rekurzív algoritmus, hiszen egy csomópont dobozának a meghatározásához mindkét gyerekére szüksége van. Az algoritmus lefutásának a végén a gyökérre kapott befoglaló

doboz az egész virtuális világ befoglaló doboza, amin jó esetben mindig belül tartózkodunk, így annak számítását és használatát akár mellőzhetnénk is a továbbiak folyamán. Jól látható, hogy ezzel a módszerrel egy hierarchikus AABB modellt építettünk fel a BSP-fánk mellé.

5.3 A megjelenítés

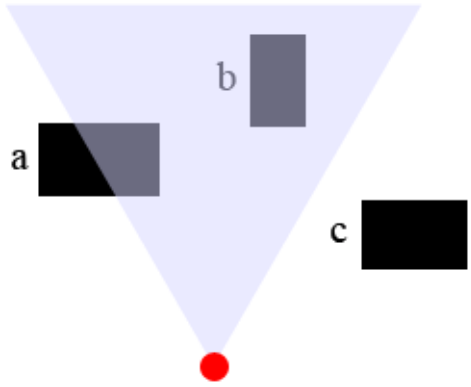
A megjelenítés az előző pontban leírtakhoz teljesen hasonló, a fán alkalmazhatunk front-to-back vagy back-to-front bejárást is. A különbség mindössze annyi, hogy az adott térrész kirajzolása előtt a befoglaló dobozos vizsgálat segítségével eldöntjük az adott térrész láthatóságát. Az alábbi megjelenítő algoritmust szintén a fánk gyökerére indítjuk hamis, vagyis nem teljesen látható értékkel inicializálva.

```
Draw (BSPElem elem, bool fullyVisible)
  if NOT fullyVisible
    visible←GetVisibility(elem.bbox)
    if visible=NOT_VISIBLE
      return
    if visible=FULLY_VISIBLE
      fullyVisible=true
  if elem IS A BSPLeaf
    DrawPolygons (elem.polygons)
  else if WhichSide(elem.cutter, actPos)=Front
    Draw (elem.behind, fullyVisible)
    Draw (elem.front, fullyVisible)
  else
    Draw (elem.front, fullyVisible)
    Draw (elem.behind, fullyVisible)
```

Adott csomópont vizsgálatakor, ha paraméterként nem teljes egészében látható értéket kapunk, lekérjük az adott elem, vagyis annak befoglaló dobozának láthatóságát az adott kamerapozíció és nézeti irány mellett. Ez a befoglaló doboz 8 csúcspontjának a látómezőt határoló 6 síkhoz képest vett viszonyából adódik.

A 23. ábra szemlélteti a leírt vizsgálatot két dimenzióban. Látható, hogy az „a” befoglaló doboz részben, a „b” teljesen belül, a „c” pedig teljesen kívül helyezkedik el a nézeti gúlán.

Ha azt találjuk, hogy az adott csomópont egyáltalán nem látható, akkor visszatérünk bármiféle rajzolás nélkül, hiszen ekkor annak egyik gyerekét, illetve saját magát sem kell megjelenítenünk, azok nem látszódhatnak (mint az ábrán a „c” dobozzal reprezentált térrész).



23. ábra: Befoglaló dobozos láthatósági vizsgálat

Ha azt találjuk, hogy az adott csomópont teljes egészében látható, akkor a „teljesen látható” értéket igazra állítjuk és a gyerekekre való továbbhíváskor is ezt az értéket fogjuk továbbadni. Hiszen ha egy csomópont, vagyis befoglaló doboz teljes egészében látható, akkor a hierarchiában alatta található összes is látszódni fog, ezért azok további vizsgálata felesleges (mint az ábrán is a „b” dobozzal reprezentált térrészé).

Ha az adott befoglaló test csak részben látható, semmilyen következtetést nem tudunk levonni belőle, további vizsgálatok szükségesek a rekurzió további szintjein („a” doboz az ábrán).

Ezek után a már megszokott módon továbbhívjuk a függvényt a beállított „teljesen látható” értékkel, illetve ha levélben tartózkodunk, kirajzoljuk az összes poligont.

Az algoritmus gyorsítására nyújt lehetőséget, ha a „teljesen látható” paraméter helyett egy 6 bites értéket adunk mindig tovább, mely a befoglaló doboz helyzetét a 6 látómezőt határoló síkra külön-külön tárolja. Ennek előnye, hogy ha egy befoglaló doboz ezen síkok valamelyikével teljes egészében szemben helyezkedik el, akkor a hierarchikus felépítés miatt annak gyerekei is szembe lesznek vele.

Az ábrán ezt az „a” jelű doboznál figyelhetjük meg, mely a látómezőt határoló jobb oldali egyenes bal oldalán helyezkedik el, így annak minden gyerekének azonos oldalon kell lennie, felesleges azt újra vizsgálni.

A fent leírt módszer előnye, hogy egy egyszerű vizsgálattal rengeteg háromszögtől szabadul meg pillantok alatt. Hátránya, hogy a takarásban lévő poligonokat ugyanúgy megjeleníti, ezek kiszűrésére egyéb megoldásokhoz kell folyamodnunk.

A legújabb OpenGL hardveresen támogat egy olyan funkciót, mely adott poligonra eldönti annak takartságát a jelenleg kirajzolt elemekhez képest (HOC[1]). Ez valójában a Z-buffer felülírását vizsgálja meg. Ennek a függvénynek a segítségével, amennyiben front-to-back bejárást alkalmazunk, a befoglaló doboz látómezőbe való esésvizsgálatán kívül ellenőrizhetjük annak takartságát is. Amennyiben az egyáltalán nem kerülne megjelenítésre, mert teljes egészében az eddig megjelenített terep mögött helyez-

[1] Hardware Occlusion – hardveres takarás vizsgálat

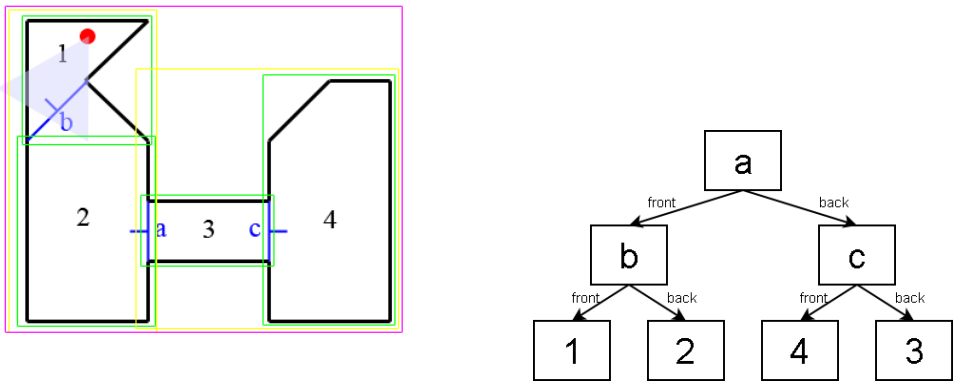
kedik el, a hierarchikus felépítés miatt a csomópont alatti összes levél vizsgálatát, illetve megjelenítését megspórolhatjuk.

A takarási probléma feloldására egy másik megoldást mutat be a következő fejezetben bemutatásra kerülő PVS BSP-struktúra.

5.4 Példa

Vegyük elő ismét az előző fejezetekben bemutatott példánkat (24. ábra).

Tegyük fel, hogy az avatar az ábrán látható pozícióban tartózkodik és az adott irányba néz. Kezdjük el az előző pontban ismertetett kirajzolás folyamatát. Először nézzük meg mi a viszonyunk a teljes világ befoglaló dobozával (lila keret). Látjuk, hogy az csak részben látszik, mivel azon belül helyezkedünk el. Az „a” vágóval szemben helyezkedünk el, így folytassuk a kirajzolás folyamatát a hátsó részre. A hátsó rész befoglaló doboza (jobb oldali sárga keret) egyáltalán nem látszik, így nem rajzolunk ki semmit, és visszamegyünk „a”-ba. Az 1-es és a 2-es térrész befoglaló doboza részben látszik (zöld keret), így azok a megfelelő sorrendben megjelenítésre kerülnek (2-1). Az algoritmus során tehát a 4 térrészből 2 kerül kirajzolásra, vagyis nagyjából 50%-ot nyerünk az adott esetben.



24. ábra: Példa BB BSP-s térmegjelenítésre

Az algoritmus nem tökéletes, de gyors, egyszerű és bizonyos esetekben nagyon jól alkalmazható.

Vegyük azonban észre, hogy amennyiben az avatarunk a pálya másik irányába fordulna, a 3-as és 4-es térrész befoglaló doboza is a látómezőbe kerülne, így azok is kirajzolódnának, holott mégsem láthatóak. Pontosan ezen takarásban lévő térrészek kiszűrésére alkalmazható az OpenGL takarási vizsgálata, vagy a következő fejezetben ismertetésre kerülő PVS BSP módszer.

6. PVS BSP-fák

Az előző fejezet példájából jól látható, hogy egy térrész láthatóságának az eldöntésére nem elegendő azok látómezőbe esésének vizsgálata. Lehetséges ugyanis, hogy egy másik térrész azt teljes mértékben kitakarja. Ezen probléma megoldására született a gondolat, hogy vizsgáljuk valamilyen módon a térrészek közötti átjárhatóságot, illetve átláthatóságot.

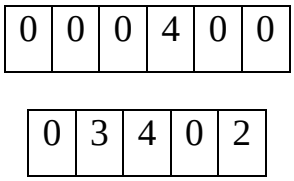
6.1 Átjárás a térrészek között

A térrészek között átjárást kell biztosítani, összeköttetést kell teremtenünk közöttük, mely azt a célt szolgálja, hogy megtudjuk, egy adott térrészből mely más térrészek láthatóak, akár köztes szektorok érintésével is. Ezt a kapcsolatot magában a struktúrában tároljuk, hogy ne kelljen sok mindent valós időben (frame-enként) számolnunk. A vizsgálat eldöntésére tetszőleges nem exponenciális komplexitású módszert választhatunk, mivel annak kiértékelése az előfeldolgozás során történik, így nincs hatással a program futási sebességére.

A térrészek közötti átjárhatóság tárolása általában flag-ekben (egy biten) történik, ez az ún. PVS[1] struktúra. Minden egyes térrészre eltároljuk, hogy onnan a többi szektor látható-e vagy nem. Ez a tárolási módszer négyzetes tárigényű. Egy pár millió poligonból álló tér BSP-struktúrájának felépítése után néhány 10.000 térrész keletkezik, aminek láthatósági flagjeinek tárolására nagyjából 100 Mb memória szükségeltetik. Ez az érték még a mai memóriák nagy mérete mellett is túl sok. A pályán egy szektorból általában csak néhány másik pályarész elérhető, így a láthatósági struktúrában rengeteg 0 byte keletkezik a nem átláthatóság miatt. A látható-

[1] Potentially Visible Sets – potenciálisan látható részek

sági struktúra tárolására ezért ZRLE[1] kódolást használnak. Ennek a módszernek a lényege, hogy a 0-s byte csoportok helyett csupán egy darab 0-t írnak, az utána következő érték pedig a 0-s byte-ok száma. A kódolás a 0-tól különböző byte-okat nem érinti (példa: 25. ábra). Ez a tárolási módszer kis helyigényű és a futás során két térrész láthatósága gyorsan eldönthető vele az adott térrész PVS-ének gyors dekódolása után.



25. ábra: ZRLE kódolás

A láthatóság előfeldolgozási időben történő eldöntésére számtalan egyszerűbb, bonyolultabb módszer létezik, én most ezek közül egyet emelek ki.

6.2 Portálok létrehozása

A legelterjedtebb módszer portálok használata. Ez esetben a BSP-struktúra felépítése után végigmegyünk a fánk csomópontjain és az adott vágósík felhasználásával portálokat hozunk létre. Ennek módja, hogy veszünk egy az adott vágó síkjára illeszkedő a térrész méreténél nem kisebb (például a befoglaló dobozából kilógó) primitívet. Ebből a primitívből CSG logika használatával kimetsszük az adott pályarészből ráeső elemeket. Az így keletkezett portálokat az adott csomópont mindkét gyerekében letoljuk egészen a levelekig az alattuk található vágósíkokhoz képest vett orientációjuknak megfelelően. Ha egy vágósík metszene egy portált, azt mindkét a fában elfoglalt helyén szétvágjuk és a továbbiakban két külön portálként kezeljük. Egy portál így két levél, vagyis konvex térrész között teremt összeköttetést. Ezek után a szomszédosság kritériuma közös portál létezése.

[1] Zero Run Length Encoding – Nulla sorozathossz kódolás

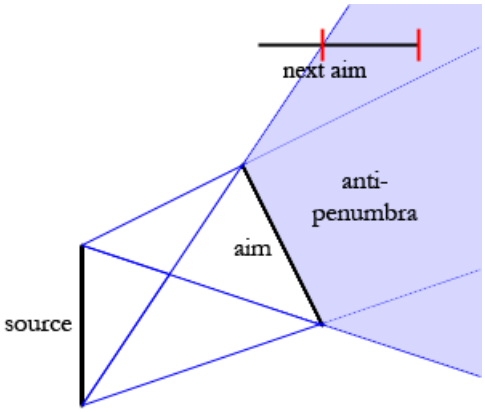
A módszer kritériuma, mivel CSG-logikával dolgozik, a pálya solid, tömör léte. Így a portál létrehozásának elkezdése előtt minden esetben szükséges a világ zártságának ellenőrzése! A zártságot nyílt terepek esetén akár láthatatlan poligonok alkalmazásával is biztosíthatjuk.

Keressük meg egy szektorhoz az összes onnan látható másik térrészt, ne csak a szomszédosakat, ezzel csökkentve a futásidejű vizsgálatok mértékét. A feladat megoldásához az avatar összes lehetséges pozícióját és nézeti irányát meg kéne vizsgálnunk és eldönteni mely térrészek láthatóak onnan. Ez azonban lehetetlen, az állapotok elvileg végtelen (gyakorlatilag mondjuk double pontosságú) száma miatt.

A feladat elvégzésére használjuk fel a már létrehozott portálokat és az úgy nevezett anti-penumbra módszerét. Vezessük be a látómező (viewing frustum) fogalmát. A látómezőnk minden irányban 360 fokos, a felhasználó bármerre nézhet. Ennek mértékét azonban befolyásolják a virtuális világ elemei, melyek részeket takarnak ki belőle. A felépített BSP-struktúrában két térrész közötti átláthatósági lehetőségek az azokat összekötő portálok, melyeket épp az előbbiek során határoztunk meg. Ezen portálok szűkítik le a látómezőnket a térrészek közötti áthaladás során. Az alábbiakban egy olyan algoritmust mutatunk be, ami minden egyes szektorra képes meghatározni az onnan látható összes másik térrészt.

Vegyük az éppen vizsgált térrészünk egy portálját, ez lesz a „forrás” (source), majd vegyük az átjáró túloldalán található szektort. Vizsgáljuk meg ezen szektor összes portálját, ezek lesznek a „célpont”-ok (aim). Nézzük meg hogyan módosul a látómezőnk, ha a forrás portálból nézünk a célportál irányába. Ezen mező meghatározására az alábbi módszert alkalmazzuk: vegyük a forrás egy pontja és a célpont két szomszédos pontja által meghatározott összes lehetséges síkot (két dimenzióban két pont által meghatározott egyenest). Válasszuk ki ezek közül azokat, melynek a forrás és a cél portál teljes egészében két ellentétes oldalán helyezkedik el. Ezek a hipersíkok határolják a látómezőnket (26. ábra). Ez

hipersíkok határolják a látómezőnket (26. ábra). Ez az a rész, ami az aktuális térrészből a forrás portálon keresztül látható lehet.



26. ábra: Anti-penumbra módszer használata

Ezek után az előbb említett műveletet rekurzívan ismételjük a célpont portál túloldalán található térrész összes olyan portáljára, ami az adott frustum-on belül helyezkedik el. Amennyiben az átjáró metszené az aktuális látómezőnket, a portált ideiglenes mi is elmetsszük és csak a ténylegesen látható részére végezzük el a vizsgálatot (az ábrán a két piros vonal közötti szakasz), az eddigi célportált forrássá „léptetve elő”.

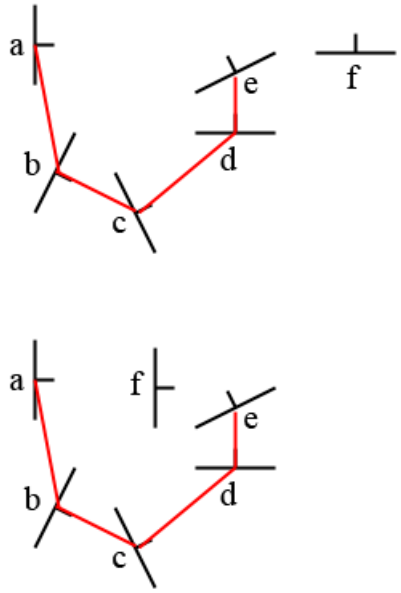
A vizsgálat során érintett térrészek természetesen az aktuális térrészből láthatóak közé sorolódnak, ezen portálok láthatóságát állítjuk be a PVS struktúránkban.

A vizsgálandó portálok számának csökkentésére 3 előfeldolgozási módszert alkalmazhatunk. Ezekhez előbb szükséges a portálok irányítottságának a bevezetése. A fában egy portálra két hivatkozás található, ez köt össze két térrészt. Válasszuk szét ezen portálokat úgy, hogy minden portál normál vektora az aktuális térrészből kifelé mutasson, a szomszéd térrész irányába. Vagyis duplikáljunk minden egyes átjárót egy ellentétes irányzékkal.

Mindegyik módszer lényege, hogy az átjárókhoz egy úgynevezett infront listában letároljuk mely más portálok látszódhatnak. Ezek után az

anti-penumbra módszer alkalmazása csak a listában szereplő átjárókra szükséges.

Az első módszer arra alapoz, hogy egy portálon keresztül egy másik átjáróba átlátni csak akkor lehetséges, amennyiben a második portál az első átjáróval szemben, míg az első a második átjáró mögött helyezkedik el. Ellenkező esetben nincs rá esély, hogy az adott portálokon átláthassunk A 27. ábra felső része azt az esetet mutatja, amikor az „f” átjáró az előzőleg vizsgált „e” portál mögött helyezkedik el, az alsó pedig azt, amikor az előzőleg vizsgált „e” a jelenlegi „f”-fel szemben található, így egyik esetben sem láthatunk át rajta „e”-ből érkezve. Az átjárási lehetőségeket az abc-ben sorba haladva piros vonallal jelöltem.

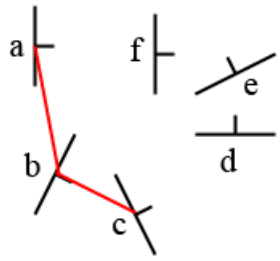


27. ábra: Az első módszer használata a vizsgálandó portálszám csökkentésére

A szűrés elvégzése az anti-frustum vizsgálat előtt szükséges. Minden portálra egy ún. prepare listában összegyűjtjük mely más portálok felelnek meg a fenti kritériumnak. Az ábra fenti példájára például „a”-nak „b” és „c”, „b”-nek „c”, „c”-nek „d”, „d”-nek „e”. Ezek után minden portálra megnézzük hogy az összegyűjtött portálok közül melyek találhatóak vele egy térrészben, ezeket rakjuk bele az infront listába. Kezdetben mondjuk „a”-nak „b”-t. Az így összegyűjtött elemekre rekurzívan újra elvégezzük a

vizsgálatot, és a portál infront listájába bekerülő elemeket az előző átjáró elemei közé is besoroljuk. Például „b” egy térrészben található „c”-vel, így az bekerül „b” infront listájába is, de mivel „a”-ból hívtuk, „a”-jéba is. A vizsgálat végeztével az anti-frustum vizsgálat minden portálnál csak az infront listájában összegyűjtött átjárókra szükséges.

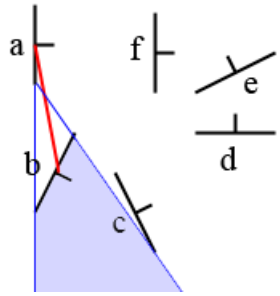
Egy szigorúbb feltétel is adható a vizsgálatok számának csökkentésére. Foglalkozzunk csak azon új portálokkal, melyek az előzőleg vizsgált összes átjáróval szemben, míg az előzőleg vizsgált összes az újonnan vizsgált mögött helyezkedik el. A 28. ábrán jól látható, hogy az előző példánál a portálok szűrése már a „d” vizsgálatánál abbamarad, mivel azzal szemben helyezkedik el az „a” átjáró. Így a vizsgálatok száma 3 portálra korlátozódik. Az algoritmus az előzőtől csak annyiban tér el, hogy a rekurzió visszatérésénél a vizsgált elem infront listájában lévő elemeket csak akkor rakjuk be a saját infront listánkba, ha az a prepare listánkban megtalálható, vagyis átláthatunk rajta. Így például „d” bekerül „b” infront listájába, de „a”-jéba már nem.



28. ábra: A második módszer használata a vizsgálandó portálszám csökkentésére

A harmadik módszer az infront lista méretének csökkentésére az anti-penumbra módszer alkalmazása. Ennek lényege, hogy a vizsgált portál infront listájába rekurzívan csak azon elemek kerülhetnek bele, melyek az adott átjárón keresztüli látómezőbe beleesnek. A 29. ábrán például jól látszik, hogy „c” nem fog „a” listájába sorolódni, mivel az a „b” által alkotott látómezején kívül esik. Ez a módszer a tényleges anti-penumbra

vizsgálattól annyiban különbözik, hogy a látómezőt nem szűkítjük a portálon való áthaladáskor, azt mindig csak az első elemre számoljuk.



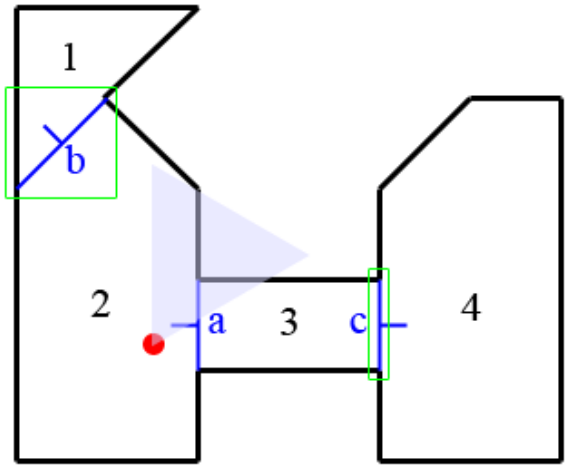
29. ábra: A harmadik módszer használata a vizsgálandó portálszám csökkentésére

6.3 A BB szűkítése

A portálok használata további egyszerűsítésre ad lehetőséget. Eddig egy térrész BB-os láthatósági vizsgálatánál a befoglaló doboz az adott szektor összes elemét körülvevő test volt. Vagyis a láthatósági vizsgálatnál azt vizsgáltuk látható-e valamely poligon az adott térrészből. A BB láthatósága viszont nem vonja maga után a térrész láthatóságát is. A portálok bevezetésével lehetőség nyílik a befoglaló doboz méretének csökkentésére. Ennek alapja, hogy egy térrész csak úgy lehet látható, ha annak valamely portálja látható számunkra és azon keresztül belelátunk a szektorba. Ellenkező esetben a térrész biztosan takarásban van.

Határozzuk meg egy szektor BB-át annak portáljai alapján! Ez minden esetben egy a teljes térrészt körülfogó doboznál nem kisebb testet eredményez. A térrész láthatóságának eldöntéséhez ekkor minden esetben elegendő a portálok által meghatározott befoglaló doboz látómezőbe esési vizsgálata.

A 30. ábrán a két zöld keret az 1-es, illetve a 4-es szektor módosított befoglaló dobozai, amik egyébként feleslegesen kerülnének megjelenítésre, de a portálok által módosított befoglaló doboznak köszönhetően még időben kiszűrésre kerülnek.



30. ábra: Befoglaló doboz szűkítése

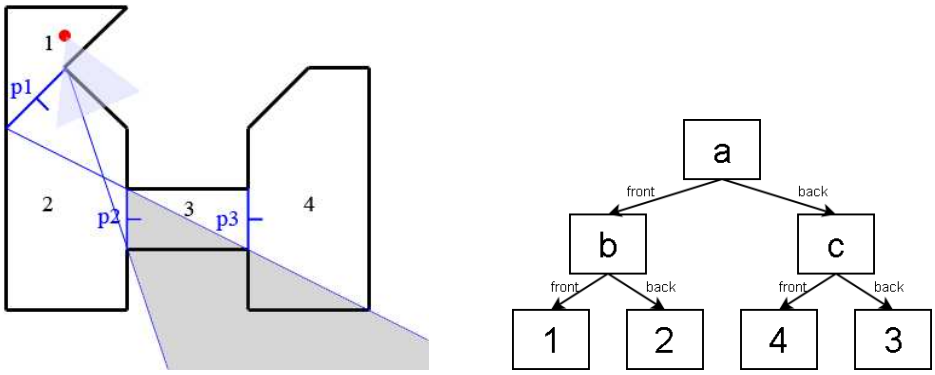
6.4 A tér megjelenítése

Az így felépített struktúra segítségével a tér megjelenítése során nagyon könnyű dolgunk van. A 4.1-es fejezetben bemutatott módszer segítségével megkeressük az aktuális térrészt, amiben tartózkodunk. Dekódoljuk a PVS-t, vagyis visszafejtjük a ZRLE algoritmust és összegyűjtjük mely térrészek látszódhatnak az avatar jelenlegi pozíciójából. Ezután az előző fejezetben bemutatott BB szűrést kell elvégeznünk az így megmaradt terekre, hogy ténylegesen csak a kamera aktuális orientációja mellett látszódó térrészek kerüljenek megjelenítésre. Természetesen a potenciálisan látható térrészek újra összegyűjtése, vagyis a PVS dekódolása csak akkor szükséges, ha az avatar mozgása során más térrészbe kerül át, míg a BB szűrés végrehajtása minden egyes frame-ben indokolt.

Ezen módszer használatával a virtuális világot felépítő poligonok 99%-a kiszűrhető, vagyis megspórolható azok videokártyára való átküldése és a nézeti transzformációk végigszámolása.

6.5 Példa

Nézzük a 31. ábra előző fejezetekből már jól ismert példáját.



31. ábra: Példa PVS BSP-s térmegjelenítésre

Valójában a példán eddig feltüntetett vágósíkok, már maguk a portálok voltak, de remélem, ez nem okoz félreértéseket. Az ábrán a portálok irányítottsága az egyszerűség kedvéért az aktuális, vagyis az 1-es térrészhez viszonyítva lett feltüntetve. A p1-es portál az 1-es és a 2-es, a p2-es a 2-es és a 3-as, míg a p3-as a 3-as és a 4-es térrész között teremt összeköttetést. Építsük most fel az 1-es térrész PVS-ét. A p1-es portálon keresztül a p2-es látható, mivel az szemben helyezkedik el vele, de a p3-as kívül esik a kettejük által meghatározott látómezőn. Így az 1-es térrészből a 2-es és a 3-as térrészek láthatóak. A megjelenítésnél amennyiben az ábrán megadott helyen tartózkodunk, az 1-es, 2-es és 3-as térrészekre végezzük el a BB-s vizsgálatot. Az ábrán feltüntetett irányba nézve ez mindhárom térrész kirajzolását eredményezi. Habár a 3-as térrész valójában nem látható, de az önmagában vett BB-s megjelenítéshez képest a 4-es térrész teljes kirajzolását megspóroltuk.

7. Felhasznált formátumok és módszerek

Az alábbi fejezetben az általam készített alkalmazásokban felhasznált formátumok és módszerek átfogó bemutatásával foglalkozom.

7.1 Az OBJ formátum[1]

Bevezetés

Az OBJ fájlok háromdimenziós modellek tárolására alkalmas szöveges formátumok. Többek között a Maya-ban szerkesztett világok is exportálhatóak erre a típusra, megfelelő beállítások mellett.

Ez a formátum rengeteg lehetőséget kínál fel számunkra, nemcsak poligonok, de egyéb bonyolultabb, úgynevezett szabad-formájú alakzatok (pl. spline-ok) tárolására is alkalmas, akár külső textúra fájlok használatával.

Az általam készített alkalmazások csak a poligonokat, azon belül is csak a háromszögeket kezelik, ezért exportálás előtt szükséges alkotásunk tesszellálása. A textúrázással nem foglalkozunk.

Csúcspont (vertex) adatok

Minden egyes csúcspontot 3 adattal jellemezhetünk: annak koordinátájával, normál vektorával és textúra koordinátájával (ez utóbbira esetünkben nem lesz szükség).

A pont koordinátájának leírására az alábbi szintaxis szolgál:

```
v x y z (w)
```

A `v` jelzi, hogy most egy csúcspont koordinátái következnek. Az `x`, `y` és `z` a pont koordinátái float típusban megadva. A `w` az opcionális negye-

[1] [7] specifikáció alapján

dik, homogén (súly) koordináta, melyre poligonoknál nincs szükség, de alapértelmezésként 1-nek vehető.

A pont normál vektorának leírására szolgál az alábbi frázis:

```
vn i j k
```

A `vn` jelzi, hogy egy normál vektor koordinátái következnek. Az `i`, `j`, `k` a normálvektor koordinátái float típusban megadva.

A pont textúra koordinátáinak leírására szolgál az alábbi kifejezés:

```
vt u v (w)
```

A `vt` jelzi, hogy egy csúcs textúra koordinátái következnek. Az `u`, `v` a textúra koordinátái float típusban megadva. A `w` az opcionális harmadik, mélységi koordináta, mely alapértelmezésként 0-nak vehető.

Az adatok természetesen derékszögű, jobb sodrású koordináta-rendszerben értendőek.

Az elemek a fájl elejétől kezdődően típusonként külön sorszámozódnak az 1-es indextől indulva.

Felület (face) megadása

Én most speciálisan a háromszögek megadásával foglalkozom, de egyéb poligonok megadását sem nehéz elképzelni ezek alapján. Mint már említettem, egy csúcsponthoz 3 fontos adat tartozik (pozíció, normál és textúra koordináta), egy háromszöghöz pedig 3 csúcspont. Így a megadás szintaxisa az alábbi:

```
f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3      vagy
f v1//vn1 v2//vn2 v3//vn3
```

A v-k a csúcspont, a vt-k a textúra, míg a vn-ek a normál koordináták sorszámai. A megadás sorrendje jobb oldali körüljárás szerint értendő.

Megjegyzés

A megjegyzés sorok kinézete az alábbi:

```
# szöveg
```

A # jelenti, hogy megjegyzés sor következik, az utána lévő szöveg pedig maga a megjegyzés. Ezek a sorok általában automatikusan eldobhatóak, a virtuális térrel kapcsolatosan semmilyen plusz információval nem szolgálnak.

7.2 Az indexelt világ

Az indexelés áttekinthetőbbé, egyszerűbbé és egyszerre helytakarékosabbá is teszi a világ tárolására szolgáló adatstruktúránkat.

Az alapelve az OBJ-hoz hasonló. Számozzuk be a koordinátákat, a normálokat, ill. a háromszög síkjait is (én most az egyszerűség kedvéért ezt 0-tól teszem). Amiben kicsit eltérek az OBJ-os megvalósítástól, hogy egy koordinátát, egy normált, ill. egy síkot (lehetőség szerint) csak egyszer veszek fel. Egy háromszöghöz csak a 3 csúcspont-, a 3 normál- és az egyetlen síkindexet kell eltárolni, és a világtól, ami számon tartja a koordinátákat, a normálokat és a síkokat, kérem le azok konkrét értékeit, amit így csak egyszer kell eltárolni a sok háromszöghöz.

Sokat nyerhetünk ezzel a módszerrel azon esetekben, amikor háromszögek egy csoportján végzünk vizsgálatot (mint pl. esetünkben, a BSP-fa építésénél). Ilyenkor ugyanis általában a csúcsokon és a síkokon végzünk műveleteket. 1 csúcshoz legalább 3 háromszög kapcsolódik (amennyiben szabályos a testünk), és általában több háromszög is egy síkba esik (gondoljunk csak pl. egy kocka oldalaira). Így ezzel a módszerrel nagyjából a

számítások és a hely 2/3-át spórolhatjuk meg, ami nagy mennyiségű adat esetén meglehetősen sokat számít. Továbbá nem utolsó szempont, hogy az építés folyamán egy poligon szétvágása során létrejött új poligonok felesleges sík, illetve normál számítását is megspórolhatjuk, valamint az azok újraszámolásából adódó számítási pontatlanságokat is kiküszöbölhetjük.

Ezért is választottam ezt a fajta tárolási módszert programom alapjául.

Az indexeléses módszert ezen kívül felhasználtam még a BSP-struktúránál is, ahol a fa tárolja a csomópontokat, a leveleket és a láthatóságot. A csomópont így csak 2 indexet tartalmaz a 2 térrészre és egy indexet a vágó síkjára (amit a világban tárolunk), valamint egy befoglaló dobozt, amit a láthatósági vizsgálatoknál használunk majd fel a későbbiek folyamán. A levél csak egy háromszögindex listát tartalmaz (a háromszögeket a világ tárolja) és egy befoglaló dobozt a láthatósági vizsgálatok elvégzéséhez.

7.3 A saját BSP formátum

Részletes magyarázat helyett lássuk inkább a fájl szerkezetét, amit az eddig bemutatott alapfogalmak segítségével könnyedén értelmezni tudunk. A módszer alapötletéül a Quake 3 BSP formátuma szolgált ([6]).

Adat típusok:

`Index`	size_t (4-byte unsigned integer, little-endian)
`float`	4-byte IEEE float, little-endian
`string[n]`	n byte-os ASCII string (nem feltétlen null-terminált)
`uchar`	1 byte-os nem előjeles ASCII karakter

Header

`string[3] magic` mágikus szöveg, mindig „BSP”

`uchar version` verziószám (jelenleg 5)

Vertexes

`Index vNum` vertexszám

`float[3][vNum] vertexes` vertexek (3 koordináta float-ban, vertexszám koordináta)

Normals

`Index nNum` normálszám

`float[3][nNum] normals` normálok (3 koordináta float-ban, normálszám normál)

Planes

`Index pNum` síkszám

`float[4][pNum] planes` síkok (4 koordináta float-ban, síkszám koordináta)

Triangles

`Index tNum` háromszögszám

`Triangle[tNum] tris` háromszögek

Triangle

`Index[3] verts` vertexindexek

`Index[3] norms` normálindexek

`Index plane` síkindex

Nodes

`Index nNum` csomópontszám

`Node[nNum] nodes` csomópontok

Node

`Index plane` vágósík indexe

`Index front` síkkal szembeni térrész indexe

`Index behind` sík mögötti térrész indexe

Leaves

`Index lNum` levélszám

`Leaf[lNum] leaves` levelek

Leaf

`float[3][2] bBox` befoglaló doboz koordinátái (min és max koordináta)

`Index tNum` térrész háromszögeinek a száma

`Index[tNum] tris` térrész háromszögeinek az indexei

Visibility

`Index vNum` láthatóság vektor mérete

`Index ovNum` egy láthatósági elem vektorának a hossza

`uchar[ovNum][vNum] visibility` láthatósági vektor

A fájl szerkezete pedig sorrendben:

Header

Vertexes

Normals

Planes

Triangles

Nodes

Leaves

Visibility

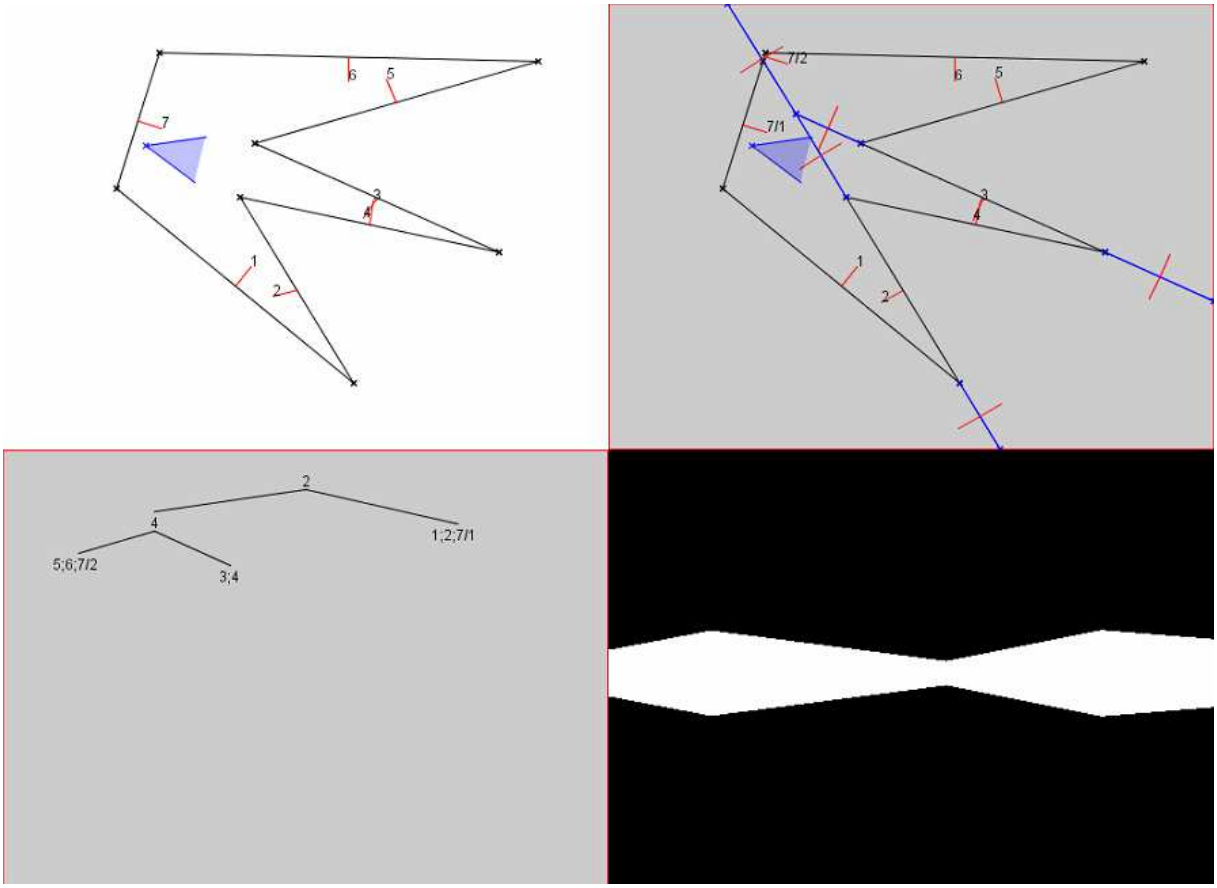
8. Saját alkalmazások és elért eredmények

8.1 Kétdimenziós PVS BSP alkalmazás[1]

Első munkám során Java3D környezetben fejlesztettem egy BSP-fa struktúrát használó 2D-s szemléltető programot.

A program ablak felülete 4 részre osztható.

A bal felső részbe rajzolhatunk falakat és helyezhetjük el a virtuális kamerát is. A képernyő jobb felső részében látható a felszabdalt tér a felszabdalt élekkel és a kékkel jelölt portálokkal együtt. A bal alsó sarokban vehetjük szemügyre a felépített BSP-fa szerkezetét az élek számozásaival (portálok nélkül). A jobb alsó sarokban pedig az adott kameraállásból látható kép jelenik meg számunkra (32. ábra).



32. ábra: Screenshot a program felületéről

[1] Részletesen: Első félévi önálló labor dokumentációm

A programban felhasznált PVS BSP módszer felépíti a BSP struktúrát és a befoglaló keretek segítségével létrehozza a portálokat. A portálok alapján a PVS struktúra felépítése nem hajtódik végre. Az anti-penumbra módszerrel történő látható szektorok összegyűjtése futási időben történik. A kamera aktuális pozíciójától és orientációjától függően a portálokon való áthaladáskor kerül sor a látótér csökkentésére. Ez a futási idejű számítások mértékének növekedését a megjelenítendő poligonok számának csökkentésével ellensúlyozza.

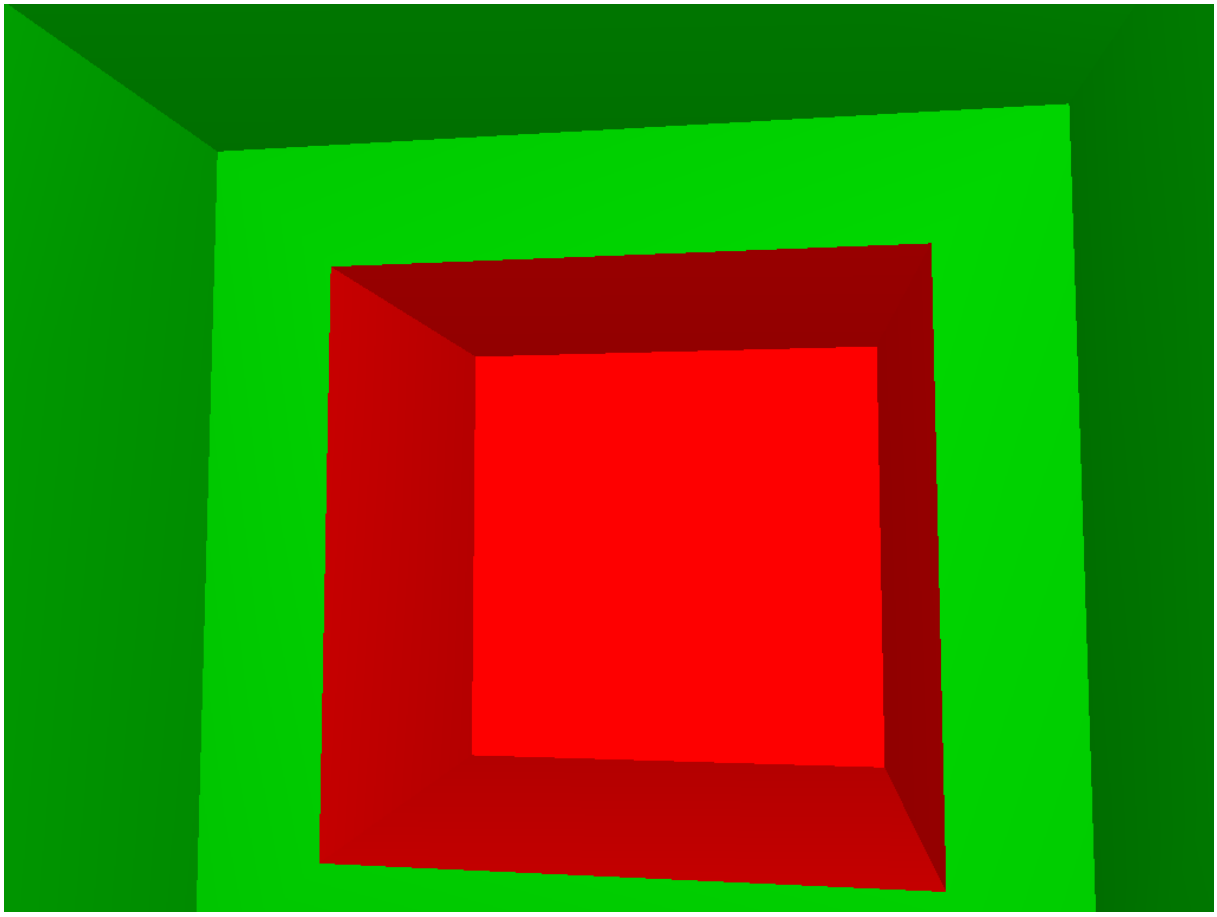
8.2 Módosított PVS BSP alkalmazás[1]

A második program C++ nyelven íródott OpenGL csomag felhasználásával, immáron valódi háromdimenziós terek kezelésével. A program nagyjából a dokumentációban is leírt pseudo-kódokat valósítja meg, objektumorientált környezetben. A pálya betöltése háromszögesített OBJ fájlból történik (ezt a kimenetet képes generálni pl. a Maya is), melyet parancssori paraméterezéssel tudunk megadni.

Indításakor a paraméterként kapott virtuális világ töltődik be. Felépül a BSP-fa struktúra a portálokkal együtt, ami kis időt vesz igénybe. Miután láthatóvá válik számunkra a kép, szabadon bepásztázhatjuk a teljes teret, szemügyre vehetjük az objektumokat. Az irányítás az egér és az iránybillentyűk segítségével történik. A térrész, amiben aktuálisan tartózkodunk zöld, míg a többi az aktuális pontból látható térrész piros színnel jelenítődik meg.

Az ESC billentyű megnyomásával léphetünk ki a programból és az F1 gomb segítségével váltogathatunk a teljes képernyős és az ablakos megjelenítés között (33. ábra).

[1] Részletesen: Második félévi önálló labor dokumentációm



33. ábra: Screenshot a program futásáról

A megjelenítés módja az előző alkalmazásával teljesen megegyező, a portálok és az azokon túl található térrészek láthatósága futási időben értékelődik ki.

8.3 Módosított BB/PVS BSP alkalmazás

Az ezen módszer bemutatására készült két program C++ nyelven íródott MFC és DirectX csomag felhasználásával. A programok a dokumentációban leírt pseudo-kódokat valósítják meg objektumorientált környezetben.

Az első alkalmazás OBJ fájlból generálja le a saját készítésű BSP fájlt. A második a BSP fájlok megjelenítésére, a pályákon való pásztázásra szolgál.

A BSP generáló program

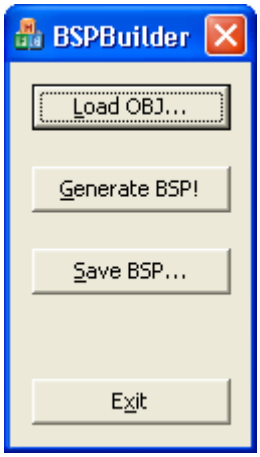
A program indításakor a 34. ábrán látható dialógus ablak jelenik meg.

A Load OBJ… funkcióval tölthetjük be a kívánt előre elkészített pályánkat OBJ fájlból.

A Generate BSP! funkció szolgál a BSP struktúra generálására.

A Save BSP… gomb segítségével menthetjük el BSP fájlba a generált BSP struktúránkat.

Az Exit segítségével pedig kiléphetünk a programból.



34. ábra: BSP generáló alkalmazás

A kiválasztott fájl betöltésére szolgál az OBJHandler osztály, ami OBJ fájlok parse-olására alkalmas. A beolvasott adatokkal a WorldHandler osztály tölti fel a világunkat, ami segédobjektumok felhasználásával biztosítja a koordináták, normálok és síkok egyediségét, hogy ne tartozzon különböző index azonos adatokhoz.

Megjegyzés: OBJ fájl készítésénél ügyeljünk a normálok irányaira, azok mindig a pálya belseje felé mutassanak! Megjegyzés 2: A pályát OBJ exportálás előtt ne felejtsük el háromszögesíteni!

A BSPHandler osztály feladata a BSP fa felépítése az eddig felépített világ alapján. Az osztály használja a WorldHandler-t is új vertexek,

normálok és háromszögek hozzáadására, valamint háromszögek lecserélésére.

Az építés a pseudo-kódokban leírtak szerint zajlik. A teret addig vágjuk szét a tér síkjaival, amíg az csak konvex térrészekből fog állni. A csomópontok gyerek indexeinek beállítása csak a csomópont létrehozása után lehetséges. Ennek oka, hogy a levelek indexe -1-től kezdődik és tart -∞-hez, míg a csomópontoké 0-tól kezdve tart +∞-hez. Egy csomópont létrehozásakor így még nem tudhatjuk, hogy azok gyerekei milyen típusúak lesznek.

A számítások gyorsítása érdekében a már egyszer kiszámításra került értékek segédobjektumokban tárolásra kerülnek. A vágó síkok által szétvágott háromszögek természetesen szintén szétvágásra kerülnek és a csomópont két gyerekébe kerülnek tovább, interpolálva a normál vektorukat a keletkezett új csúcspont(ok)ra nézve. A síkjuk természetesen ezzel nem változik meg.

A programban alkalmazott módszer a befoglaló dobozok szomszédossága alapján tölti fel a PVS struktúrát. A térrészek befoglaló testeire amúgy is szükség van a felesleges térrészek futásidőben történő kiszűrésének érdekében. A láthatóság beállítása ugyanúgy a teljes fa elkészülte után történik meg. Két térrész akkor látja egymást, amennyiben dobozaiknak van metszése (vagy befoglalják egymást). A módszer meglehetősen pontatlan eredményt szolgáltat, de könnyen implementálható és kezelhető, ezért demonstrációs célokra teljes mértékig megfelelő. A térrész láthatóságának eldöntésénél nincs szükség portálok képernyő-koordinátákra való transzformálására, elegendő az előre tárolt flag-ek vizsgálata, ami után csak a befoglaló dobozok láthatóságát kell számolnunk a tényleges láthatóság eldöntéséhez.

A láthatóság tárolására karakter tömböt használunk, amiben egy bit egy láthatóságot jelöl, az alábbiak alapján:

```
visibility[i][j/8]&(1<<(j%8))
```

1, ha az i. térrészből látható a j. térrész, és 0, amennyiben nem.

A BSP struktúra lementéséért a BSPWriter osztály a felelős. Ez exportálja a teljes világot és a BSP-fát is az előző fejezetben bemutatásra került formátumba.

A BSP megjelenítő program

A program indításakor a 35. ábrán látható dialógus ablak jelenik meg.

A Load BSP… funkcióval tölthetjük be az előző program segítségével OBJ fájlból generált BSP fájlunkat.

A Play! funkció szolgál a világ BSP struktúrás megjelenítésére, vagyis a játékra.

Az Exit segítségével léphetünk ki a programból.

35. ábra: BSP megjelenítő alkalmazás

A kiválasztott fájl betöltésére szolgál a BSPLoader osztály, ami BSP fájlok parse-olását végzi a BSPWriter osztály által létrehozott fájlokból, az ott leírtak alapján. A betöltés folyamán figyeli a fájl esetleges sérültségét vagy eltérő verzióját is.

A megjelenítésért a Game osztályunk a felelős. Ez először beállítja a lokális változókat, létrehozza a játék ablakot, beállítja rá a DirectX felületet

(jelenleg szoftveres renderelés 800x600-as méretű ablakban) a fényekkel és a Z-bufferrel, és feltölti a világgal a vertex buffert. Ezek után az ablak bezárásáig egy végtelen ciklusba kerül, amiben, ha esemény érkezett, lekezeli azt, elmozgatja a mozgatandó dolgokat (jelen esetben az avatar-t), majd lerendereli az új megjelenítendő képet.

Az üzenetkezelő ciklus kezeli az ablakhoz érkező eseményeket, úgymint annak bezárását, átméretezését, újrarajzolási kérését, illetve a fókusz elvesztését és újra visszanyerését. Az egérmozgatás mértékével arányosan forgatja el az avatar nézeti irányát, majd visszaállítja az egeret az ablak közepére, hogy ne tudjuk azt az ablakról elhúzni. A billentyűlenyomásokat egy tömbben tárolja, amit az avatar kérdez le mozgása során.

A mozgás megvalósítása az avatar feladata. Hogy a mozgás milyensége ne függjön az aktuális gép teljesítményétől, erre a célra egy külön nagy felbontású órát használunk, ami a processzor órajelét veszi alapul. A mozgatást fix időközönként végezzük (10 ms-onként), ezt nevezzük léptetésnek. Az avatart a jelenlegi időpont (az aktuális renderelés időpontja) után mozgatjuk, vagyis mindaddig léptetjük, amíg túl nem lépi azt. Az utolsó két léptetési pozíciót interpoláljuk a renderelési időre. Így jó közelítéssel kapjuk meg az avatar tényleges pozícióját az adott pillanatra. A léptetések során minden platformon ugyanazokba a pozíciókba jutunk.

Léptetéskor először kiszámoljuk az avatar új pozícióját, az aktuális nézeti irányából és az iránybillentyűk lenyomottsági állapotától függően, természetesen csak vízszintes mozgást engedélyezve. Ezek után hozzávesszük az avatar szabadesését is. Ezt kétféle módon számoljuk, egy konstans esési sebességgel és egy Verlet-integrálos gyorsulásos módszerrel. A két érték közül vegyük most a nagyobbat. Az előző időpontbeli és az ilyen módon számolt új értékkel meghívjuk a BSPHandler osztály segédfüggvényét, ami visszaadja a két ponttal meghatározott szakasz által elmetszett

síkot, vagy ha ilyen nincs egy null-vektort. A segédfüggvény az ütközést csak a két pont térrészében vizsgálja.

Amennyiben null értéket kapunk vissza, a Verlet-integrálos képlettel kiszámított szabadesést hajtjuk végre, természetesen a lenyomott iránybillentyűk figyelembe vétele nélkül (hiszen levegőben nem sok ember tud mozogni).

Ha egy valódi síkot kaptunk vissza, újraszámoljuk az avatar elmozdulását, természetesen az adott sík mentén, immáron az ugrás lehetőségét is beleszámítva. Ezt korrigáljuk a gyorsulásos fix sebességnek a lejtés irányába mutató komponensével, illetve azért, hogy minimális lejtésű talajon ne csúszkáljunk, ennek egy fix súrlódási tényezővel csökkentett értékével. Így nagyjából 20 fokos lejtőnél kezdünk el csúszni, és 45 foktól már nem tudjuk megakadályozni a csúszást.

Az így kapott új pozíciót szintén a BSPHandler-nek átadjuk az előzővel egyetemben, hiszen lehetséges, hogy újabb ütközésre került sor az új irány miatt. Amennyiben ez tényleg így van, akkor csak a sík menti elmozdulással rakjuk arrébb előző pozíciónkat. Mivel többszörös eltolásokra is szükség lehet pl. sarkoknál, a végtelen ciklus elkerülése végett korlátozzuk eme vizsgálatok számát.

A legelső ütközésvizsgálat szükségességét az támasztja alá, hogy az elmozdulás kívánt irányában lévő síkot kapjuk vissza, így nem tudunk beragadni a sarkokba. Ezen vizsgálat során a gravitáció jelentősége, hogy lejtés irányába való elmozdulásnál is létező síkot kapjunk vissza, ne szakadjunk el a talajtól.

Rendereléskor a játék fő osztálya beállítja a megfelelő nézeti és projekciós mátrixot, majd letörli a teljes képernyőt. A BSPHandler osztály segítségével pedig összegyűjti az adott pozícióból az adott nézeti irányban látható térrészeket. Ez a pseudo-kódban is leírt módon történik először a

jelenlegi térrész megkeresésével, majd a szomszéd térrészek összegyűjtésével, azok befoglaló dobozának láthatósági vizsgálatával.

Ezután a képernyő bal felső sarkába kiírunk pár fontos adatot, úgy mint a látható térrészek és az összes térrész aránya (SPs), az FPS értékét, amit egy kis felbontású órával mérünk és 100 frame-enként frissítünk, valamint az aktuális szemmagasságot, ami a tesztelés folyamán nagy jelentőséggel bírt.

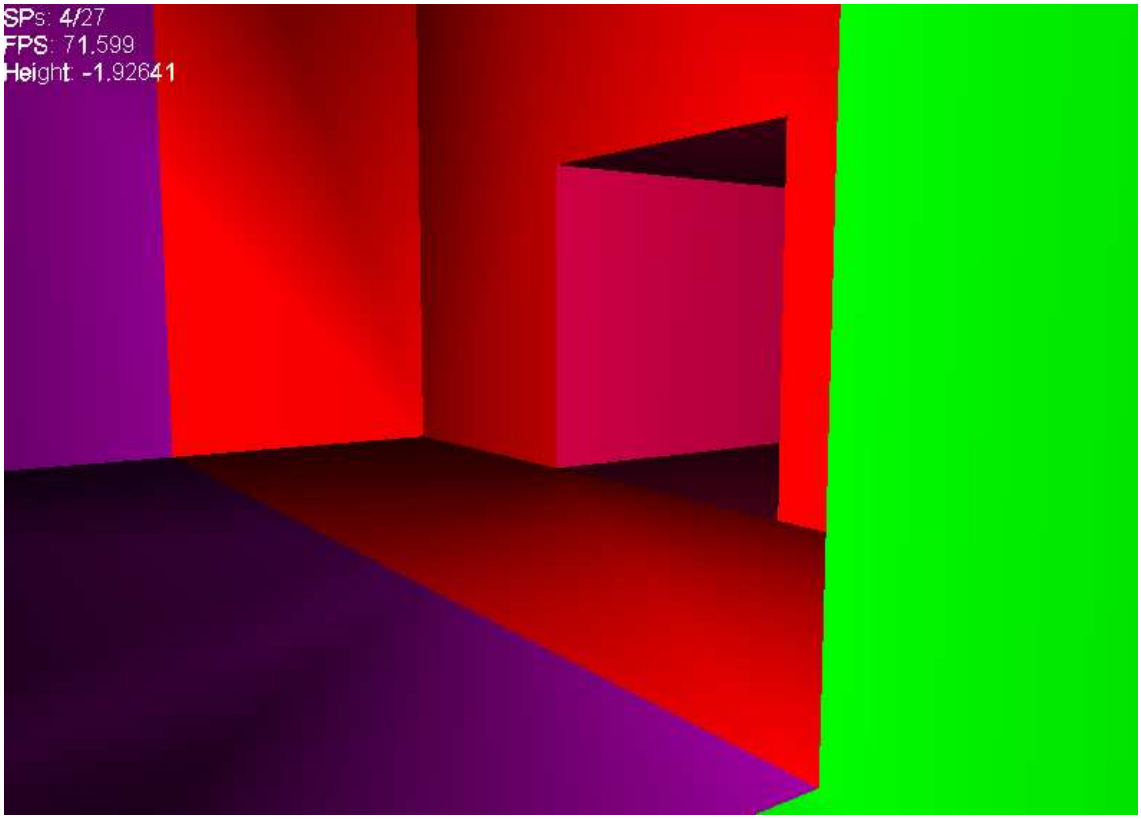
Ezt követően történik a fény(ek) beállítása, ami egy, az avatar aktuális pozíciójából az aktuális nézeti irányába mutató, spot lámpa.

Ha ezekkel megvagyunk, kirakjuk a képernyőre az összegyűjtött térrészek háromszögeit. Zölddel színezve az aktuálisat, és pirosból kékbe átmenő árnyalattal a többi láthatót.

Az egér segítségével nézelődhetünk, mint bármely szokványos FPS játékban (nincs invert mouse), az iránybillentyűk segítségével mozoghatunk előre-hátra, illetve oldalazhatunk, a SPACE-szel ugorhatunk és a C billentyűvel guggolhatunk. Kilépés az ESC billentyű leütésével.

A következő oldalon található két ábra a virtuális teret megjelenítő alkalmazás futása közbeni képet mutatja.

36. ábra: Screenshot a virtuális világ megjelenítéséről 1.



37. ábra: Screenshot a virtuális világ megjelenítéséről 2.

9. Tapasztalatok, egyéb lehetőségek

9.1 Poligonok a csomópontokban

Sok irodalomban és alkalmazásban a BSP-struktúra építése során a csomópontokban tárolják azon poligonokat, melyek a vágó síkjára esnek. Munkám során én ezzel a lehetőséggel nem éltem. Alkalmazásaimban a levelek, így velük együtt a háromszögek is egy helyen vannak tárolva, ezáltal önmagukban teljesen lefedve a virtuális világot. Így például a BSP-struktúra nélküli terepmegjelenítéshez elegendő a levelekben található sokszögek kirajzolása. A levelekben ténylegesen egy konvex térrész található, mely bármely más elemmel együtt már nem lenne az. Ha azonban a csomópontokban is lennének sokszögek, azokat a „front” gyerekhez még hozzá tudnánk venni. A struktúrában ezáltal szétszóródnának az egyszerre megjeleníthető konvex terek. Ezen okokból kifolyólag döntöttem a módszer mellőzése mellett.

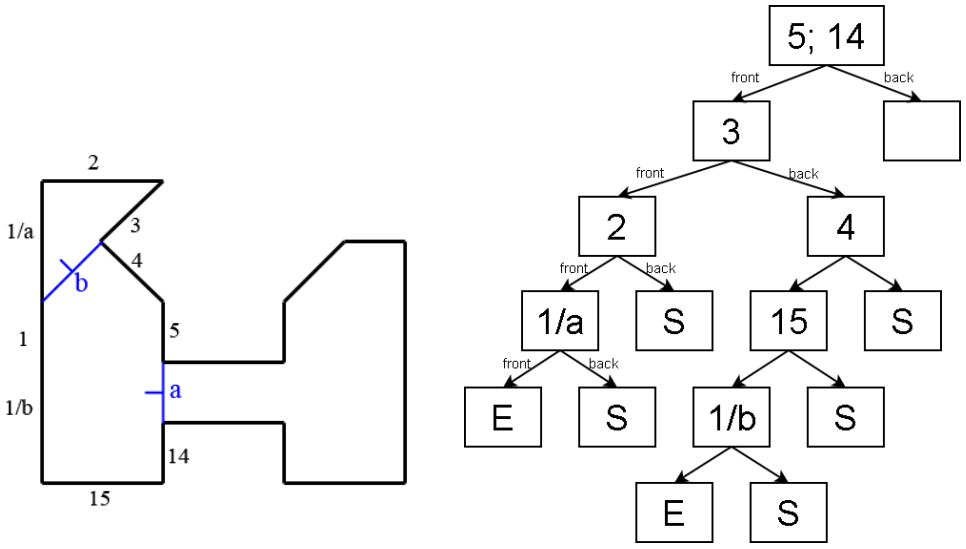
Mindazonáltal néhány módszerhez, mint például a következő pontban bemutatásra kerülő egyszerű portál-létrehozáshoz, szükség van a technika alkalmazására.

9.2 Egyszerűbb portál-létrehozás

A portálok létrehozása, mint említettük CSG logika segítségével történik, a vágósíkra a térrészből ráeső elemek körvonalainak kimetszésével. Az így keletkezett átjárók ebből kifolyólag tetszőleges alakúak lehetnek, a PVS meghatározásának egyszerűsítése érdekében szükség lehet ezen átjárók háromszögesítésére. Egy kevésbé körülményes portál alkotásra adunk módszert az alábbiakban.

Használjuk az előző pontban bemutatott módszert, vagyis a vágó síkjával egybeeső elemeket tároljuk a fa csomópontjaiban! Ezen kívül a BSP-fa építését ne hagyjuk abba konvex térrészhez való érkezésnél, hanem

addig folytassuk, amíg minden egyes poligon síkját nem használtuk vágóként. Amikor a térrész utolsó elemét használjuk vágásra, az bekerül a csomópontunkba, és két térrész keletkezik, egy a vágóval szemközti üres („empty”) és egy vágó mögötti érvénytelen („solid”). Szokásos példánkon a művelet egy részét a 38. ábrán vehetjük szemügyre. Jól látható, hogy a módszer még ilyen kis játéktér esetén is mennyivel megnöveli a fa méretét.



38. ábra: Példa egyszerűbb portál létrehozásra

Ezek után portál létrehozásához vegyük minden egyes vágó síkját, a gyökértől indítva vágjuk azt el minden egyes csomóponti síkkal, amíg a vágósík csomópontjához nem érkezünk. A megmaradt részt toljuk le mindkét gyerekbe és folytassuk a műveletet. Ha egy vágósík a keletkezendő portálunkat elvágja, azt mindkét fában elfoglalt helyén vágjuk el, és orientációjuknak megfelelően toljuk tovább a fában. Ha egy portállal „solid” részhez érkezünk, dobjuk el az adott átjárót. Ezen térrészek között nem jöhet létre átjárás, hiszen azok a tér pályán kívül eső részét jelképezik. Ha a fában a portál két üres részhez kerül, tartsuk meg. Ezek jelentik az átjárást a két térrész között.

Vegyük az előző példát és nézzük végig a „b” portál születését. Ez a 3-as elem síkjával kezdődik. A fa gyökerében található 5-ös, illetve 14-es elem síkjával vágjuk el. A jobb oldali részt eldobjuk, a bal oldalit letoljuk a

3-as csomópontba. Itt mindkét gyereknek továbbadjuk. Nézzük először a front-ot. A 2-es síkjával szemben van a portál, így front-ba továbbadjuk 1/a-nak. Ez az elem két részre vágja a keletkezendő portált, a bal oldali a back-be megy, ami „solid” térrész, így eldobhatjuk, a jobb oldalit megtartjuk, ez front-ba megy, ami „empty”, azaz üres, így megállunk. A fa 3-as mögötti térrészére megvizsgálva a megmaradt átjáró minden elemmel szemben van, így ott is üres térrészbe kerül. Tehát megtartjuk a „b” portált, ami a két térrész között fog átjárást biztosítani.

Az így felépített struktúrát az ütközés felismerésére is nagyon könynyen használhatjuk. A játék közben a pályán belül tartózkodás feltétele, hogy üres jelzésű térrészben tartózkodjunk. Ha az avatar mozgása során „solid” levélbe, vagyis a pályán kívülre kerülne, ütközés lép fel, ami minden esetben a szülő csomópont valamely poligonjával történik. A feloldáshoz az avatart minden esetben valamely üres levélbe kell visszahelyeznünk.

A módszer egyszerű, könnyen implementálható megoldást ad mind az átjárók megtalálására, mind az ütközések felismerésére. A konvex térrészek azonban teljesen elvesznek, megtalálásuk körülményes lesz. Mindazonáltal a fa mérete is kezelhetetlenül naggyá válik, még az egyszerű példánk esetén is.

9.3 Pályaelemek használata

Az eddigiekben a virtuális világ felszabdalását mutattam be. De mi van akkor, ha az egyik „szoba” közepén a grafikus egy labdát helyez el? A labda konkáv alakzat, így ha a struktúra építése közben ezt is figyelembe vennénk, annak minden lapjára el kellene vágnunk a teljes világunkat. Ez nyilván értelmetlen dolog lenne.

Az ilyen elemeket már a pályaépítés során el kell különíteni a pálya részeitől, hogy a struktúra építés során azok ne legyenek rá hatással, és

hogy egy díszítő elem módosításakor ne legyen szükséges a teljes fa újragenerálása.

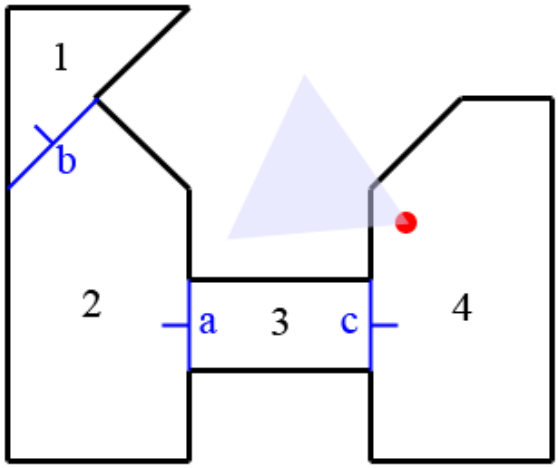
Ezen elemeket az építés végeztével mint különálló objektumokat helyezzük el a fánk megfelelő levelében, saját befoglaló dobozaikkal együtt. Mivel az objektumok akár több térrészen keresztül is átnyúlhatnak, a legegyszerűbb módszer, ha csak hivatkozásokat tárolunk el rájuk. Így egy objektum minden frame-ben csak egyszer kerül kirajzolásra, amennyiben befoglaló doboza és a tartalmazó térrészek közül legalább egy látható.

Lehetőség nyílik dinamikus elemek elhelyezésére is a pályán. Ilyenkor a feladatunk az objektumok fában történő mozgatásának megoldása. A pálya szerkezetének dinamikus módosítására ezen struktúra nem alkalmas, kivéve ha a módosítások helye előre ismert, és azt már az építés közben figyelembe vesszük.

9.4 Konklúzió

A dokumentációban bemutatott módszerek igen jól használhatóak a kitűzött HSR feladatra. A portálok létrehozására imént bemutatott példa kombinálható az alap BSP struktúrával. Az átjárók létrehozására használjuk ez előbbi módszert, míg a tárolás és megjelenítés folyamán folyamodjunk a PVS BSP struktúrához.

A PVS struktúra egyik hátránya, hogy egy térrészhez ugyan az összes onnan látható térrészt előre tárolja, ám azok hierarchiája nem kerül lementésre. Így, ha egy térrész az aktuális helyről csak egy köztes szektor érintésével látszódik, az akkor is kirajzolásra kerül, ha a köztes éppen nem látható. Erre példa a 39. ábra, ahol az aktuális 4-es mellett a 2-es pályarész is kirajzolásra kerül, holott az csak a 3-ason keresztül látszódna, amit a BB szűrésnél kiejtünk. A térrészek nagy száma miatt ilyen hierarchia tárolására nincs kapacitás, de a jövőben talán majd erre is születik használható megoldás.



39. ábra: Példa PVS BSP struktúra hiányosságára

A struktúra további hátránya, hogy a dinamikus pályaelemek használata nehezen, a dinamikus pálya megvalósítása szinte egyáltalán nem megoldható.

Úgy érzem, hogy munkám során a BSP-struktúra játékokban való felhasználhatóságára elegendő módszert és példát mutattam.

A BSP-struktúra, ami a ’90-es években forradalmasította a számítógépes játékipart, a benne rejlő lehetőségek határaihoz érkezett. A XXI. század megnövekedett felhasználói igényeinek (dinamikus pályaelemek és pályák) kielégítésére alkalmatlan. A mai több millió poligonból álló virtuális világokra az előfeldolgozás lefutása hatalmas erőforrás- és időigényű, melyet a hardverek rohamos fejlődése és árának csökkenése ellenére is egyre nehezebb kielégíteni. A BSP-struktúra jelenlegi formájának bővítése nem vezethet számottevő gyorsuláshoz és eredményhez a játékfejlesztésben, de a sugárkövetés, a CSG-logika és egyéb területeken a mai napig igen jó eredményeket produkál. Mindazonáltal a módszer megismerése során szerzett tapasztalatok felhasználhatók egy új, gyorsabb, dinamikus elemeket is támogató rendszer kialakításában.

Köszönetnyilvánítás

Köszönöm egész családom és barátaim támogatását és lelkesítését, mellyel nem csak ezen dokumentum és program megírásához, de az ötéves tanulmányaim sikeres teljesítéséhez is hozzájárultak a tőlük telhető legnagyobb mértékben.

Köszönöm konzulensemnek, dr. Szirmay-Kalos Lászlónak az elmúlt két évben nyújtott segítségét mind az önálló labor, mind a diplomamunkám elkészítésében és hogy felkeltette érdeklődésemet a számítógépes grafika iránt.

Mindazonáltal külön köszönöm évfolyamtársamnak és barátomnak Horváth Jánosnak kitartását, mellyel egyetemi éveim elejétől bátorított és segített a programozás rejtelmeinek megismerésében, mindemellett készséggel válaszolt értelmetlennek tűnő, avagy éppen értelmetlen kérdéseimre.

Irodalomjegyzék

[1] Samuel Ranta-Eskola:
Binary Space Partitioning Trees and Polygon Removal in Real Time 3D Rendering
Uppsala Master's Theses in Computing Science, 2001-01-19
http://www.gamedev.net/reference/programming/features/bsptree/bsp.pdf

[2] Joel Anderson:
View Space Linking, Solid Node Compression and Binary Space Partitioning for Visibility Determination in 3D walk-throughs
School of Computer and Information Science (SCIS),
20 January 2005, Last Updated 5 March 2006
http://www.gamasutra.com/features/20060417/Anderson_Thesis.pdf

[3] Kumm:
BSP fák használata
http://www.prog.hu/cikkek/?aid=61

[4] Dr. Szirmay Kalos Lászó, Antal György, Csonka Ferenc:
Háromdimenziós grafika, animáció és játékfejlesztés
ComputerBooks Kiadó Kft., Budapest, 2003

[5] NeHe Productions
http://nehe.gamedev.net

[6] Unofficial Quake 3 Map Specs
Kekoa Proudfoot, 2000
http://graphics.stanford.edu/~kekoa/q3/

[7] OBJ spec
http://www.dcs.ed.ac.uk/home/mxr/gfx/3d/OBJ.spec

Ábrajegyzék