NYILATKOZAT

Név: Molnár Dóra Viktória

ELTE Természettudományi Kar, szak: Matematika BSc

NEPTUN azonosító: IZLPI6

Szakdolgozat címe:
Az octree adatstruktúra alkalmazásai a számítógépes grafikában

A **szakdolgozat** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló szellemi alkotásom, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2023. 06. 05.

a hallgató aláírása

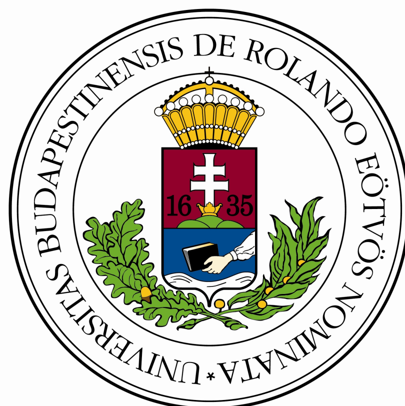
EÖTVÖS LORÁND TUDOMÁNYEGYETEM
TERMÉSZETTUDOMÁNYI KAR

Molnár Dóra Viktória

Az octree adatstruktúra alkalmazásai a számítógépes grafikában

Szakdolgozat
Matematika BSc
Alkalmazott Matematikus Szakirány

Témavezető:
Dr. Jüttner Alpár
Operációkutatási Tanszék



Budapest, 2023

Tartalomjegyzék

1. Bevezetés **3**

2. Az octree **4**
2.1. Története 4
2.2. Felépítése, jellemzői 6
2.3. Előnyei 7
2.4. Octree kódolás 10
2.4.1. Szomszédos csúcsok 12

3. Színkvantálás **15**
3.1. Korábbi algoritmusok 16
3.1.1. Egyenletes felosztás 16
3.1.2. Népszerűségi felosztás 16
3.1.3. Medián vágás 17
3.2. Octree alapú algoritmusok 17
3.2.1. Gervautz és Purgathofer módszere színkvantálásra 17
3.2.2. Bloomberg módszere 20

4. Ray tracing adaptív octree használatával **22**
4.1. Az octree-R 23

5. Ütközésérzékelés **26**
5.1. Video avatarok 26
5.2. AVMIX rövid áttkintése 28
5.3. Ütközésérzékelés octreevel 28

6. Boole-operációk **31**
6.1. Általános áttekintés 31
6.2. Az algoritmus 33
6.3. Implementáció, példa objektumok 43

7. Szeletelés **45**
7.1. Octree alkalmazása szeleteléshez 45
7.1.1. Memóriaigény 45
7.1.2. Octree elkódolása söprés mentén 46
7.1.3. Octree felépítése 48
7.1.4. Octree alapú szeletelés 49

8. Összefoglalás **51**

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőmnek, Dr. Jüttner Alpárnak, aki a rendszeres konzultációk során ötleteivel és meglátásaival segítette szakdolgozatom létrejöttét. Hálás vagyok családomnak és barátaimnak a tanulmányaimat végigkísérő támogatásukért és belém vetett bizalmukért.

1. Bevezetés

A nagy számítási kapacitású eszközök megjelenése olyan alkalmazási területek kifejlődését vagy előtérbe kerülését hozta magával, mint a CAD, a számítógépes grafika vagy a vizualizáció. Az általuk meghatározott problémák megoldása rendkívül erőforrásigényes, emellett speciális algoritmusok kidolgozását is szükségessé tette. Ezek jelentősége nem elhanyagolható az orvosi képalkotás, a robotika vagy - napjainkból példát keresve - a videojátékok területén sem.

Dolgozatomban egy, a számítógépes grafika világában megkerülhetetlen adatstruktúrával, az octreevel foglalkozom. Célom rávilágítani arra, hogy bár "feltalálója" 1980-ban jelentette meg az első erről szóló cikket [24] (és 1992-ben sikeresen szabadalmaztatott is egy octree-kódolású objektumok megjelenítésére szolgáló módszert), ez a típusú reprezentáció mind a mai napig megjelenik a számítógépes grafika egy-egy gyakorlati alkalmazásában.

A 2. fejezet teljes egészében az octreeről szól, bemutatom a történetét, a pontos leírását és felépítését, a hozzá tartozó kódolási lehetőségeket, illetve elemzem a hatékonyságát. Ezt az áttekintést öt, lényegében különböző alkalmazási mód követi. Ezeket úgy választottam ki, hogy a lehető legtöbb szempontból bemutassák ezt az adatstruktúrát, az egyes példákon keresztül pedig egyre jobban megértsük annak működését, és észrevegyük a benne rejlő lehetőségeket. Az alkalmazások hozzávetőlegesen megjelenésük szerinti időbeli sorrendben kerülnek megvizsgálásra, ami segít megérteni, hogyan vált egyre szélesebb körben alkalmazottá ez az adatstruktúra.
A 3. fejezet az octree egyik viszonylag korai felhasználási módjával, a színkvantálással foglalkozik. Ebben a részben áttekintem a feladat lényegét és fontosságát, a megoldásra adott korai algoritmusokat, és megmutatom, milyen szempontokból jelentett áttörést az octree alkalmazása.
A 4. fejezet középpontjában a ray tracing (sugárkövetés) hatékonyságát növelő adaptív octree ötlete áll. A háromdimenziós virtuális környezet - a megfigyelő szemszögének megfelelő - kétdimenziós képpé alakításában az octree már a kezdetektől fogva jelen van. Ebben a részben az adatstruktúra egy speciális, a feladat megoldását hatásosabban gyorsító verzióját elemzem.
Az 5. fejezetben áttérek egy modernebb, napjainkban is fontos problémára, az ütközésérzékelésre. Ez a feladat a videojátékok működésének szerves részét képezi, máig izgalmas kérdés annak hatékonyabbá és gyorsabbá tétele.
A 6. fejezetben egy háromdimenziós háromszögelt felületek közötti Boole-operációk végrehajtására szolgáló algoritmust mutatok be, amelyet C++ nyelven meg is valósítottam, így a módszer lépéseinek részletes leírása után saját eredményeimet is megosztom. A dolgozatban található 3D-s ábrák többsége is az említett programmal készült el.
A 7. fejezetben egy rendkívül aktuális problémát vizsgálok meg, az elemzés alapjául szolgáló cikk alig egy éve jelent meg. A "szeletelés" a 3D nyomtatás alapvető részét képező matematikai probléma, amely egy speciális octree-kódolással került optimalizálásra.

Ezzel a felépítéssel remélem, hogy sikerül bemutatni az olvasó számára az adatstruktúra széles körű alkalmazási lehetőségeit.

2. Az octree

2.1. Története

Az octree történetének kezdete 1980-ra tehető, ekkor jelent meg ugyanis Donald J. R. Meagher első ezzel foglalkozó cikke "Octree Encoding: A New Techinque For The Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer" [24] címmel. Ekkoriban kezdett egyre fontosabb kérdéssé válni a háromdimenziós objektumok digitális reprezentálása, Requicha [30] 6 kategóriába osztotta az ismert koncepciókat:

1. Primitív példányok módszere (Pure Primitive Instancing Schemes) - Előre megadott objektumcsaládokkal dolgozik, egy bizonyos objektumot a családjának neve és véges sok paraméter határoz meg.
2. Térbeli felsorolás (Spatial Occupancy Enumerating) - Az objektum által elfoglalt voxelek (minimális térbeli egységek, eredete: volume és pixel szavak összevonása) felsorolása meghatározott sorrendben.
3. Részekre bontás (Cell Decomposition) - A térbeli felsorolás általánosítása, ahol nem voxelekre, hanem más, tetszőleges számú oldallal rendelkező (és nem feltétlenül identikus) objektumokra bontjuk a testet.
4. Konstruktív térgeometria (Constructive Solid Geometry - CSG) - Az objektumok primitív testeken (hengerek, téglatestek, stb.) végrehajtott Boole-operációkkal (metszet, unió, különbségképzés) kerülnek reprezentálásra.
5. Söprés útvonal mentén (Sweep Representation) - A testet egy két- vagy háromdimenziós objektum és egy útvonal határoz meg: az a térfogat, amit előbbi végigsöpör miközben az útvonalat bejárja.
6. Határolás (Boundary Representation) - Az objektumokat határoló felületeik reprezentálják (oldalak, patchek, stb.) amiket éleik és csúcsaik határoznak meg.

A legtöbb technika egyértelmű hátránya volt, hogy a térbeli testek csak egy korlátozott körének reprezentálására voltak alkalmasak, hiszen előre meghatározott primitív testekből vagy felületekből kerültek előállításra. Néhány rendszer alkalmas volt bonyolultabb objektumok kezelésére is, viszont ennek ára a módosításkor vagy megjelenítéskor fellépő lényegesen megnövekedett számítási igény volt. Hasonló gondok merültek fel, ha a primitívekkel dolgozó módszerek alap objektumainak bővítése vagy általánosítása volt a cél. Ilyen előzményekkel természetes módon merült fel az igény egy olyan térbeli modellezésre alkalmas adatstruktúra létrehozására, amely lehetővé teszi a tér tetszőleges bonyolultságú elemeinek elkódolását, tárolását, módosítását, elemzését és megjelenítését. Mindez pedig lehetőleg működjön hatékonyan, az akkori igény szerint tehát valós időben vagy azt megközelítve. Meagher öt pontban fogalmazta meg az octreevel kapcsolatos alapvető elvárásokat [25], ezek a következők:

- Egy olyan kódolási formátum, amely képes teszőleges 3-(vagy N-)dimenziós objektum reprezentálására, tetszőleges felbontással.
- Bármilyen objektumon vagy objektumok halmazán végrehajthatóak legyenek a Boole-operációk (unió, metszet, különbség) illetve geometriai transzformációk (eltolás, átméretezés, forgatás).
- Hatékonyan (lineáris időben) megoldható legyen a két N-dimenziós objektum közötti ütközés észlelésének problémája.
- Lineáris időben megjeleníthető legyen tetszőleges számú objektum, tetszőleges nézőpontból színezéssel, árnyalással és árnyékolással, több fényforrás és átlátszó testek esetében, ortográfiai vagy perspektivikus nézetben, élsimítással.
- A módszer párhuzamosítható legyen, elkerülve a lebegőpontos műveleteket, egész számok szorzását és osztását.

Ezen célok elérése érdekében készült el az octree (vagy oktális fa) névre hallgató adatstruktúra, amellyel kapcsolatban létrehozása után két további fontos tulajdonságot is megfigyeltek: alkalmazás során a rendszer hatékonysága a feladat komplexitásának növekedéséhez képest lassú ütemben csökken, illetve a felhasználó szabadon egyensúlyozhat a számítási költség és az elvárt precizitás között. Például egy képalkotási eljárás során a módszer kifejezetten gyorsan adhat egy durva közelítést az eredményre, amely az idő előrehaladtával részletesebbé, kidolgozottabbá válik a több számítási művelet végrehajtásának köszönhetően.

Az octree tulajdonképpen több, külön-külön már korábban felmerült vagy esetleg alkalmazott ötlet fúziója. A hierarchikus geometriai struktúra elképzelése Clarkhoz [7] köthető, aki ezt javasolta a felületek láthatóságával foglalkozó algoritmusok alapjának. A többdimenziós bináris keresőfa (k-d(imenziós) fa) és az oszd meg és uralkodj elv alkalmazásának bevezetése Bentley nevéhez fűződik [4, 5], aki számos N-dimenziós geometriai probléma megoldásán dolgozott. Nem sokkal később pedig Franklin kifejlesztett egy rácsokon alapuló technikát a takarásban lévő vonalak és felületek meghatározásához [11]. Talán az egyik legegyértelműbb előzménye az octree létrejöttének egy kétdimenziós fa adatstruktúra, a quadtree. Ezt korábban már többen tanulmányozták, legfőképpen képfeldolgozási célokra használták. Hunter és Steigliz részletesen vizsgálták a quadtree tulajdonságait, algoritmusait és komplexitását [17]. A quadtree térbeli általánosításának ötlete számos szerzőnél felmerült, végül Meagher írta le először mélységében az octree adatsruktúra felépítését, alkalmazásait, tulajdonságait és algoritmusait cikkeiben [24, 25]. Utóbbiak közül az eltolás, a 90 fokkal történő forgatás és a Boole-operációk már a quadtreevel kapcsolatban megoldottak voltak, ezeket a módszereket sikerült is átemelni magasabb dimenzióba. Az átméretezésre, tetszőleges szöggel való forgatásra és a rejtett felületek meghatározására alkalmas algoritmusokat Meagher már N dimenzióban alkotta meg, így ezek kétdimenzióban (quadtreek esetén) is használhatók.

2.2. Felépítése, jellemzői

Meagher [24, 25] leírása szerint az octree kódolási módszere a korábban felsoroltak közül a térbeli felsorolás és a részekre bontás ötletéhez hasonlít legjobban. Lényegében az ebben az adatstruktúrában eltárolható információ ugyanúgy az objektum által elfoglalt voxelek felsorolása, mint az említetteknél, azonban az adatokat az octree esetében egy hierarchikus fa struktúrában tároljuk, ahol a fa csúcsai a tér egy-egy diszjunkt téglatestét reprezentálják, melyek mérete a fa mélységével exponenciálisan csökken. Az octree alapvetően egy 3-dimenziós adatstruktúra (tekinthető a quadtree 3-dimenziós kiterjesztésének is), azonban megépítésének és használatának elve N dimenzióban is érvényes. A magasabb dimenziókban használt általánosított verziót N-dimenziós octreenek hívjuk, így könnyen zavar támadhat afelől, hogy éppen mire gondolunk octree alatt. Ebben a dolgozatban az általános leírás folyamán együtt tárgyalom a 3- és az N-dimenziós struktúrákat, a későbbiekben viszont mindig az előbbiről lesz szó, ha azt külön nem jelzem. Itt szeretnék kitérni arra is, hogy octreevel lényegében bármilyen térbeli konstrukciót reprezentálhatunk, alkalmazhatjuk csak pontok vagy esetleg szakaszok eltárolására, én azonban a 3-dimenziós objektumokra fókuszálva mutatom be a felépítését, tulajdonságait. Általánosságban pedig feltesszük, hogy ezek valódi térfogattal rendelkező elemek, és nincsen 2-, 1- vagy 0-dimenziós, nem a térfogatot határoló részük.

Definíció. Az octree tehát egy gyökeres fa adatstruktúra, amely N-dimenziós objektumok tárolására alkalmas, és a következő tulajdonságok teljesülnek rá:

1. Legyen V a reprezentálandó objektum térfogata, B pedig egy körülzáró doboz, amely teljes egészében tartalmazza V-t. B lehet véges vagy végtelen.
2. Az octree minden csúcsa egy N-dimenziós téglatest alakú térrésznek felel meg B-ben, amellyel gyakran azonosítjuk is a csúcsot.
3. A fa gyökere B-t (és ezáltal V-t) reprezentálja, ezt a csúcsot nevezzük univerzumnak is.
4. A fában kétféle csúcs létezik:
   - Egy v csúcsot köztes csúcsnak nevezünk, ha pontosan 2^N gyereke van a fában. Ezek a csúcsok a v-hez tartozó térrész egy partícióját adják, jelölésük: $C(v)$.
   - Egy v csúcs levél, ha nincsen gyereke.
5. Egy csúcs szintje megegyezik az egyetlen, a csúcsot a gyökérrel összekötő út hosszával. (Így a gyökér a 0. szinten helyezkedik el.)
6. A fa csúcsai között a következő további relációk állhatnak fenn:
   - Egy v csúcs szülője az a w csúcs, melynek v gyereke.

- v leszármazottainak halmazát képezik azok a csúcsok, amelyek a v mint gyökér által meghatározott részfát alkotják.
- v ősei azok a csúcsok, melyeknek v leszármazottja.
- v csúcs testvére w, ha azonos szintűek és közös a szülőjük.

7. Egy köztes csúcsot az osztópontján (centrum) keresztülmenő, tengelyekkel párhuzamos (hiper)síkok 2^N diszjunkt részre (3 dimenzióban ezek az oktánsok) osztanak, ezeket reprezentálják a fában a szülő csúcs gyerekei.

További megjegyzések három dimenzióban:

- Az octree gyökerének leggyakrabban a V-hez tartozó minimális AABB-t (axis aligned bounding box) használják, mely két átellenes csúcsának koordinátái 3 dimenzió esetén $(x_{min}, y_{min}, z_{min})$ és $(x_{max}, y_{max}, z_{max})$, ahol x_{min}, y_{min} és z_{min} az objektum minimális x, y és z koordinátái, hasonlóan x_{max}, y_{max} és z_{max}.

- Egy csúcs osztópontjának meghatározása kétféle módon történhet (quadtreeket részletesen osztályozott Aref és Ilyas [2]): PR (point region) típusú octreeben minden csúcshoz eltárolunk egy explicit pontot a 3-dimenziós térben, ez tölti be a centrum szerepét, ami a csúcs továbbosztása során minden gyerek térrész egyik sarkát adja. MX (matrix-based) típusú octree esetén az osztópont nem más, mint a csúcs által reprezentált tér/térrész közepe, e mentén osztjuk a csúcsot. Természetesen utóbbi esetben a gyökér csak korlátos halmaz lehet, hogy a centrum jóldefiniált legyen. A fa csúcsai tehát mind az univerzum egy-egy területét reprezentálják, illetve jellemzik azt néhány hozzájuk rendelt értékkel. Amennyiben ezek az értékek egyértelműen képesek leírni az adott térrészt, levél csúcsokról beszélünk, melynek nincsenek további gyerekei. Ha ez nem áll fenn, akkor a csúcs pointere a nyolc gyerekére mutat, amelyek a szülő térrészének nyolc oktánsával azonosak [25].

- Három dimenzió esetén sokszor a gyökér csúcsot egy kockának választjuk, majd a csúcsok továbbosztása mindig a térbeli centrum mentén történik. Így a fa minden csúcsa egy kockának felel meg a térben, gyakran a csúcsokat azonosítjuk is a kockákkal.

2.3. Előnyei

Az octree számos előnye megfogalmazható, Meagher [24, 25] többek között az alábbiakat emeli ki ezek közül:

- Az octree lehetővé teszi, hogy tetszőleges objektumot elkódoljunk az általunk választott precizitással. Nem szükségesek a korábban használt adatstruktúrák feltételei (konvexitás/primitív testekből való előállíthatóság).

- A hierarchikus struktúra miatt a gyökér az egész objektumot tartalmazza. Minden szinten a csúcsok és azok ősei együtt leírják az egész objektumot az annak a szintnek megfelelő pontossággal, így a különböző algoritmusok dolgozhatnak a fa csak egy bizonyos szintje feletti csúcsokkal, amennyiben nem szükséges nagyobb precízió. Egy alkalmazás során például lehetőség van rá, hogy az objektumok nagy része csak durvább felbontásban szerepeljen az elsődleges memóriában, a magasabb felbontású részleteket pedig másodlagos memóriában tároljuk. Ez a tulajdonság különösen hasznossá válik, ha interferencia észlelés a feladat.

- Ha egy másik módon reprezentált objektumot Octree-kódolási formátumba alakítunk, szükség van valamilyen térbeli rendezésre (tipikusan az egyik koordináta szerint). Ez után a kezdeti rendezés után azonban ezek az elemek vagy a belőlük Boole- vagy geometriai transzformációkkal képzett másik objektumok már nem igényelnek térbeli rendezést, akkor sem, ha megváltoztatjuk a nézőpontot.

- Mivel az objektumokat folyamatosan térben rendezve tároljuk, a takarásban lévő felületek keresésére vagy az interferencia észlelésére szolgáló algoritmusok lineáris időben megvalósíthatók. Az előbbi esetében például csak a fa csúcsainak egy speciális sorrendben történő feldolgozására van szükség, amelyet kizárólag a nézőpont határoz meg.

- Az octreevel történő kódolás jelentősen megkönnyíti az objektum számos jellemzőjének meghatározását is. Például a tömeg kiszámításához a gyökérből indulva minden szinten könnyen megadhatunk egy alsó és egy felső korlátot. Amikor a két érték közötti különbség már kellően kicsi, megállhatunk, ehhez sokszor a csúcsok csak csekély hányadát kell átvizsgálnunk.

- Az octreevel dolgozó alapvető algoritmusok (unió, metszet, kivonás, eltolás, átméretezés, forgatás és különböző megjelenítési módszerek) egyike sem igényel lebegőpontos műveleteket, egész számok szorzását vagy osztását. Emiatt ezek könnyen implementálhatók olcsó hardvereken is. Emelett egy csúcs feldolgozása 0-8 újabb alszámítást generál, így hasznos lehet nagy mennyiségű, párhuzamosan dolgozó processzort használni a műveletek elvégzéséhez.

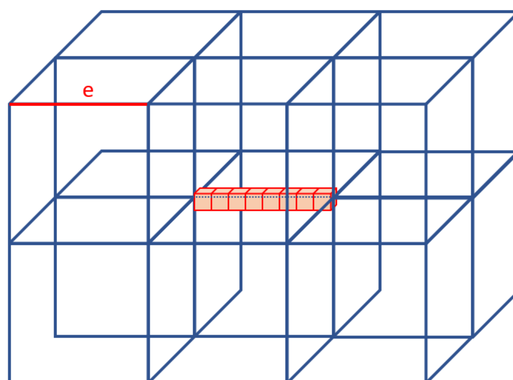
- Az adatstruktúra alkalmas az objektumok felületéhez tartozó színek eltárolására, amely számos további alkalmazást elősegít.

Mindezek mellett az alkalmazások során figyelembe kell vennünk az octree adatstruktúra memóriaigényét. Hunter és Steigliz megmutatták [17], hogy egy síkbeli objektum quadtreevel történő eltárolásához (és a fa felépítéséhez) szükséges memória az objektum kerületének nagyságrendjével arányos. Meagher pedig leírta ennek az általánosítását [24]:

Tétel. *Egy 3-dimenziós összefüggő objektum elkódolásához szükséges octreenek $\mathcal{O}(S)$ darab csúcsa van, ahol S az objektum felszíne.*

Bizonyítás. Feltehetjük, hogy minden objektum minimális, n. szintű kockákból, azaz voxelekből épül fel, amelyek teljes lapjaikkal szomszédosak, így egy objektum felszíne azon voxel-lapjainak területének összege, amelyek nem szomszédosak más voxelekkel. Legyen továbbá egy voxel éleinek hossza 1 (ez feltehető, hiszen csak konstans szorzóval változtatja meg az eredményt, ami a nagyságrendet nem befolyásolja).

- *Állítás:* Az octree egy k. szintű csúcsának éle legyen e. Ekkor egy $4e + 2$ felszínű objektum legfeljebb 12 darab k. szintű csúcsot érinthet.
  Bizonyítás: Indirekten tegyük fel, hogy P egy olyan $4e + 2$ felszínű test, amely legalább 13 k. szintű kockát érint. Ekkor biztosan igaz, hogy valamelyik tengelyirányban legalább e kiterjedéssel rendelkezik. Amennyiben ez nem teljesül, minden irányban legfeljebb két k. szintű kockát érinthet, tehát összesen legfeljebb nyolcat. A legkisebb felszínű olyan test, amely valamilyen irányban e kiterjedéssel rendelkezik, egy e hosszúságú voxelekből álló sorozat, ennek felszíne pontosan $4e + 2$. Tehát P nem más, mint az előbb leírt és az alább ábrázolt test. P azonban nem metszhet 13 csúcsot, hiszen két tengely irányában egy voxel élének kiterjedésével rendelkezik, a harmadik tengely mentén pedig e szélességű. Így P legfeljebb $2 * 2 * 3 = 12$ csúcsot metszhet.



1. ábra: A voxelekből álló sorozat 12 csúcsot metszhet

- Jelölje S egy objektum felszínét, m_k pedig azt, hogy hány k. szintű csúcsra van szükség az objektum reprezentálásához (vagyis hány ilyen kockát metsz a felszíne). Az előző gondolat alapján $4e + 2$ felület ábrázolásához legfeljebb 12 k. szintű csúcs szükséges, így $m_k < 12(\frac{S}{4e+2})$

- Így mivel $e = 2^{n-k}$

$$m_k < \frac{3S}{2^{n-k}+0.5} < \frac{3S}{2^{n-k}}$$

- Legyen L a az összes szükséges csúcs száma. Ekkor tehát:

$$L = \sum_{k=0}^{n} m_k < \sum_{k=0}^{n} \frac{3S}{2^{n-k}}$$

- Jelölje $r = n - k$ és fordítsuk meg az összegzés sorrendjét:

$$L < 3S * \sum_{r=0}^{n} \frac{1}{2^r} < 6S$$

- Vagyis $L = \mathcal{O}(S)$.

$\square$

Tekintsünk egy 3-dimenziós univerzumot, amelyhez tartozó octree n+1 szintes, 0. szintjén található a gyökér. A továbbiakban azonosítjuk a csúcsokat és az általuk reprezentált kockákat a térben. Egy adott szinten minden kocka éle ugyanolyan hosszú, tegyük fel, hogy az n. szinten ez 1, így a k. szinten egy él 2^{n-k} hosszú.
Fontos megjegyezni, hogy az octree csúcsainak száma nem csupán magától az objektumól függhet, figyelembe kell venni annak helyzetét is az univerzumban. A test mindig eltolható úgy, hogy a gyökér csúcs három oldalát érintse, azonban forgatással még mindig különböző eredményeket kaphatunk. Így sokszor egyenletes eloszlásból vett szögekkel történő forgatások után szükséges csúcsok számának átlagát veszik, vagy használhatják a minimum/maximum szükséges csúcsszámot is mérőszámként.
Bebizonyítottuk tehát, hogy az octree kódoláshoz szükséges memóriaigény az objektum felszínének nagyságrendjével arányos, ami a test komplexitásától és az elvárt precizitástól függően sokszor túl nagynak bizonyulhat. Egyszerű objektumok esetében, ahol például a primitív testek alkalmazása elengendő, előfordul, hogy nem az octree az ideális választás. Azonban bonyolultabb elemeknél ez a reprezentáció nagy előnyökkel szolgálhat.

2.4. Octree kódolás

Az octree alapötlete, felépítése világos, azonban egy objektum ezzel történő elkódolásának gyakorlati megvalósítása további kérdéseket vet fel. Meagher [24, 25] javaslata alapján az octree tekinthető rendezett párok halmazának is, ahol minden csúcshoz tartozik egy P, amely a csúcs tulajdonságainak véges listája. Ez utóbbi legalább egy tulajdonságot tartalmaz, ez három különböző értéket vehet fel:

- ÜRES: Az objektum egyáltalán nem metszi ezt a csúcsot, ezért nem igényel további felosztást, nincsenek gyerekei.

- TELJES: Az objektum teljes egészében kitölti ezt a csúcsot, ezért nem igényel további felosztást, nincsenek gyerekei.

- RÉSZBEN KITÖLTÖTT: nem homogén csúcs, a V objektum metszi a csúcsot, de nem tartalmazza magában. Nem határozza meg egyértelműen az objektum helyzetét az általa reprezentált térrészben, így szükség van a továbbosztására.

Az ÜRES és TELJES csúcsok egyértelműen leírják az általuk reprezentált térrészt, uniójuk a levél csúcsok halmaza. A RÉSZBEN KITÖLTÖTT csúcsok azok, amelyeknek további gyerekei vannak a fában, így ezek a fa köztes csúcsai. (Az irodalomban sokszor az üres csúcsokat fehérrel, a teljeseket feketével, a részben kitöltötteket szürkével jelölik.) Ennek a tulajdonságnak az eltárolása mindenképpen szükséges (és 2 biten meg is valósítható), de természetesen bonyolultabb alkalmazás során számos egyéb jellemző rendelhető a csúcsokhoz: szín, sűrűség, anyag, hővezető képesség, stb.

Sokszor felmerül a kérdés, hogy milyen mélységig építsük meg az octreet egy adott objektumhoz. A választ erre szinte kivétel nélkül a feladat maga határozza meg, függhet az objektum bonyolultságától, attól, hogy mi milyen precizitást várunk el, illetve mire van szükségünk, de a későbbi példáinkban előfordul az is, hogy szeretnénk egy memóriaigény és számítási idő közötti egyensúlyt fenntartani. A fa ezen tulajdonságát levél feltételekkel befolyásolhatjuk, amelyeket két csoportra oszthatunk:

1. Vonatkozhatnak explicit a fára: annak mélységére (pl. a 15. szintű csúcsok már mindenképpen levelek legyenek) vagy a csúcsok számára.
2. Függhetnek az objektum bizonyos tulajdonságaitól (pl. ha annak már csak elenyésző része esik a csúcsba, akkor tekintsük levélnek és ne osszuk tovább, vagy egy másik alkalmazás során, ha már csak azonos színű pontok esnek egy csúcsba, akkor legyen levél).

Mindezek mellett még egy kérdés biztosan felmerül egy octree implementálása során: hogyan kódoljuk a reprezentáló fát? Knuth adatstruktúrákról szóló művében [21] három különböző megközelítést emel ki fák kódolására, amelyeket Samet és Webber [31] részletesen elemez quadtreek és octreek esetében. Ezek az alábbiak:

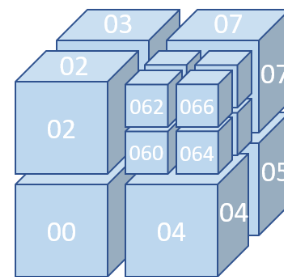
1. Pointerekkel/mutatókkal történő kódolás. Minden köztes csúcsnak 8 pointere van, amik az egyes gyerekeire mutatnak (vagyis az általuk reprezentált részfákra), a levél csúcsokhoz nem tartozik pointer. Sok algoritmusnál hasznos lehet egy szülő pointer fenntartása is a nem gyökér csúcsokhoz, de a legtöbb alkalmazás a gyökérből indulva felfedezi a fát, és az átvizsgálás során is megjegyezhetők a szülő-gyerek kapcsolatok. Ez a legegyszerűbb, legkorábban használt kódolási mód, de a tárhelyfelhasználás alacsony hatékonysága miatt legtöbbször nem ezt alkalmazzák.
2. Lineáris kódolás. A fa csúcsainak egy meghatározott sorrendben történő felsorolása pointerek nélkül. Egyetlen információt szükséges még minden csúcshoz eltárolni, amivel meg tudjuk különböztetni a leveleket és a köztes csúcsokat. A konkrét sorrend meghatározása az alkalmazástól függ, gyakran használatos a szélességi és mélységi keresésen alapuló sorrend. Az előbbinél tetszőleges N_1 és N_2 csúcsok esetén, ha N_1 mélysége kisebb, mint N_2 mélysége, akkor N_1 N_2 előtt van a felsorolásban, míg utóbbinál minden csúcs után közvetlenül a gyerekei részfái következnek. Mindkét

esetben szükséges még egy gyerekeket rendező függvény is a sorrend egyértelműségéhez. Ebbe a kategóriába tartozik a 7. fejezetben tárgyalt kódolási forma is.

3. Helyzeti kódolás. Nevezik Morton-kódnak vagy Z-sorrendnek is. Számos verziója ismert, de ezek közös tulajdonsága, hogy minden csúcsot egy egyedi számkód jelöl az alábbi változat alapvető struktúráját betartva:
   - Legyen a gyökér sorszáma 0.
   - Legyenek az 1. szint 8 csúcsának azonosítói az ábra szerint 00, 01, 02, 03, 04, 05, 06, 07.
   - Ezek után minden csúcs számkódja úgy keletkezik, hogy a szülő azonosítóját és az ő pozícióját (ami az első szintnek megfelelő szám 0 és 7 között) konkatenáljuk.

   Ez a módszer egy sorrendet is meghatároz a csúcsok között, így sokszor a lineáris kódolással együtt használják. (Ez a sorrend lényegében a mélységi keresés eredménye egy speciális gyerekeket rendező függvényt alkalmazva.)

A Morton-kód nagyon gyakran használatos, mivel számos algoritmus alapozható az ismeretére. A továbbiakban ezek közül egyet ismertetek részleteiben Hasbestan és Senocak [19] cikke alapján, amelynek célja egy tetszőlegesen kiválasztott octree csúcs összes szomszédjának felsorolása. A szomszédsági kapcsolatok egy octreeben - alkalmazástól függően - szintén fontosak lehetnek, ahogy azt a 4. fejezetben meg is említem.



Morton-kód

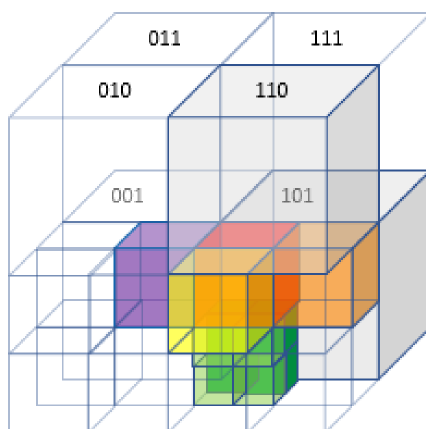
2.4.1. Szomszédos csúcsok

Feladat: Adott v (piros) csúcs az octreeben. Soroljuk fel az összes olyan w csúcsot, amelynek van közös lapja v-vel.

Megoldás: Először is gondoljuk meg, hogy két szomszédos octree-csúcsnak nem feltétlenül azonos a szintje, sőt bármekkora különbség lehet közöttük, így a feladat kezdetekor nem tudjuk, hány csúcsot kell majd felsorolnunk. (Azt viszont igen, hogy 6 irányban kell keresnünk a szomszédos csúcsokat, a v csúcs 6 lapjának megfelelően.) Másodsorban pedig az általánosság megszorítása nélkül feltehető, hogy az octree minden csúcsa szabályos kocka.

Tekintsük a Morton-kódolás bináris formáját, a könnyebb átláthatóság kedvéért az egyes szintekhez tartozó koordinátákat vesszővel el is választjuk (ebben az ábrázolásban a gyökérnek nincsen kódja). Minden szinten egy koordináta 3 bitből áll, melyre a felosztás által úgy is tekinthetünk, mint amelyek az x, y, és z irányokhoz tartoznak. Ha a kocka osztópontja

x tengely irányában bal és jobb oldali, y tengely irányában alsó és felső, míg z tengely irányában elülső és hátulsó gyerekekre osztja a szülő csúcsot, akkor egy koordináta első bitje 0, ha bal oldali és 1, ha jobb oldali gyereke a szülőjének, a második és harmadik bit pedig ugyanígy meghatározza, hogy alsó/felső, illetve elülső/hátulsó gyerek-e a csúcs.



2. ábra: A piros csúcs és szomszédai

Tekintsük a 2. ábrát, ahol a piros kocka kódja az általunk bevezetett kódolás szerint $(100, 011)$, ezen csúcs szomszédait kell meghatároznunk. A Morton-kód egyértelmű, így számunkra elegendő, ha a szomszédos csúcsok kódját meg tudjuk adni. Az algoritmus 2 részből fog állni:

1. Szomszédos testvérek megtalálása.

   - Nyilvánvalóan egy tetszőleges szintű octree csúcsnak pontosan három szomszédos testvére van, x, y és z irányban 1-1-1.
   - Ezek Morton-kódját úgy kapjuk, hogy a piros kocka kódjának utolsó koordinátájában rendre ellentétessé változtatjuk az egyes biteket, hiszen ez jelenti azt, hogy azonos szinten egy olyan testvér csúcsot találunk, amely x, y vagy z irányban szomszédos a piros csúccsal.
   - Ez a három testvér vagy levél csúcs, vagy további gyerekeik vannak. Ha levél csúcs, akkor felvehetjük a szomszédok listájába, ha nem, akkor viszont a gyerekeit kell vizsgálnunk.
   - Testvér szomszéd gyerekeinek vizsgálata: Tekintsük az y irányú testvért (zöld kocka), amelynek további gyerekei vannak, és tegyük fel, hogy a testvér utolsó koordinátájának y szerinti bitje 0. Ekkor ő az alsó testvére a piros csúcsnak, így az ő összes felső gyerekét kell vegyük, tehát azt a 4 Morton-kódot, amely úgy áll elő, hogy a zöld csúcs kódjához hozzáveszünk még egy olyan koordinátát, amelynek második bitje 1. Rekurzívan ugyanezt megcsináljuk a 4 gyerekre, ha

ők levelek, akkor bevesszük a szomszédsági listába, ha nem, tovább vizsgáljuk a gyerekeiket. Természetesen ez szimmetrikusan elvégezhető x és z irányokban is az 1. illetve 3. biteket használva.

2. Nem testvér szomszédok megtalálása. Ez a lépés újabb 2 részből áll:

   (a) Az első rész leírása algoritmussal:

   1. Algorithm Bit-csere szintjének meghatározása

   ```
   1: T := összes csúcs Morton-kódja
   2: Adott M ∈ T
   3: k := 1,2,3: (x,y vagy z koordináta irányában keresünk)
   4: f := 0 (bit-csere szintje)
   5: M_l := M szintje
   6: bit := M(d * (M_l − 1) + k)
   7: for i = M_l − 2 : 0 do
   8:     if bit ≠ M(d * i + k) then
   9:         f = i + 1
   10:        break
   11:    end if
   12: end for
   13: return f
   ```

   Kövessük ezt végig az ábrán is bemutatott példán. A piros kocka Morton-kódja $(100, 011)$, 3 testvér szomszédjából $((100, 010), (100, 111), (100, 001))$ kettő levél (narancssárga és citromsárga), egynek (zöld) további gyerekei vannak, ezeket megtaláljuk az 1. pontban leírtak szerint. Most célunk az ábrán lilára színezett szomszédos csúcs Morton-kódjának meghatározása. Ez a piros csúcs x tengely szerinti szomszédja, így az algoritmusban $k = 1$. A piros kocka utolsó koordinátája 011, tehát az x-nek megfelelő bitje 0. Egy szinttel feljebb lépve a koordináta már 100, vagyis megtaláltuk a szintet, ahol a kérdéses bit megcserélődik, ez jelen esetben az 1. szint.

   (b) A második lépés során a bit-csere szintjétől kezdve lefele minden szinten meg kell változtatni az egyes koordináták x tengelyhez tartozó bitjét 1-ről 0-ra illetve fordítva. Így megkapjuk a lila csúcs Morton-kódját: $(000, 111)$.
   Az eddig leírtak tökéletesen működnek, amennyiben a keresett szomszéd szintje megegyezik a piros csúcséval. Ha ez nem teljesül, újabb 2 esetet kell szétválasztanunk:

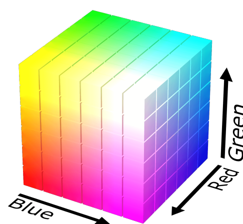
   - Ha a szomszéd csúcs alacsonyabb szintű, mint a piros, akkor a generált Morton-kód utolsó néhány (amennyi a különbség a szintek között) koordinátáját elhagyjuk (így találjuk meg az ábrán szürkére színezett szomszéd csúcsokat).

- Ha a szomszéd csúcs magasabb szintű, mint a piros csúcs, akkor a testvér szomszéd eseténél látottak alapján járunk el. Az első két lépés szerint legeneráljuk az azonos szintű szomszéd csúcsot, majd tovább vizsgáljuk a gyerekeit. Ezt úgy is megtehetjük, hogy az azonos szintű szomszéd csúcs utolsó koordinátáját megduplázzuk, ez lesz az egyik szomszédos gyerek. A másik 3 szomszédos gyerek pedig az y és z tengelyhez tartozó bitek megváltoztatásával nyerhető. Ha a gyerekek nem levél csúcsok, az utolsó eljárást rekurzívan ismételjük.

Külön ki kell térni a határoló csúcsokra, melyeknek van olyan lapja, ami a gyökér csúcs lapjának része. Ezeknek nincs mind a 6 irányban szomszédjuk, attól függően, hogy 1, 2 vagy 3 határoló lapjuk van, ennyi szomszédjuk hiányzik. Ez pontosan az az eset, amikor az 1. rész algoritmusa $f = 0$ értékkel tér vissza, ekkor a 0. szinttől kezdve kéne megváltoztatnunk a biteket, ez jelzi, hogy ebben az irányban nincsen (nem testvér) szomszéd.

3. Színkvantálás

A színkvantálás célja, hogy egy kép ábrázolásához szükséges színek számát jelentősen lecsökkentsük úgy, hogy közben a kép minőségét, vizuális élvezhetőségét minél kevésbé rontjuk el. A fejezet első része Gervautz és Purgathofer összefoglaló munkáján [13] alapszik, az ő nevükhöz köthető az octree felhasználása is ezzel a problémával kapcsolatban. Az emberi szem körülbelül kétszázféle erősségű piros, zöld és kék színt tud érzékelni, így összesen nagyjából 8-10 millió szín megkülönböztetésére vagyunk képesek. Az RGB-skálával reprezentált képekben minden pixelt három 8 bites kód ír le, egyenként reprezentálva 0 és 255 között az elsődleges színeket. Ez a rendszer 256^3, azaz nagyjából 16 millió szín elkülönítésére alkalmas, ami jóval meghaladja az emberi képességeket, így az egyik legelterjedtebb színábrázolási technikává vált a számítógépes grafikában. Ezen színek mindegyikét ábrázolni azonban nehézkes és nagy memóriaigényű lehet, így sokszor éredemesebb a teljes színpaletta csak egy részét használni, vagyis K (pl. $K = 256$) színnel ábrázolni a képet (K a színtábla mérete).



Az RGB-kocka [34]

Amikor egy K-nál több eredeti színből álló képet szeretnénk ilyen módon megjeleníteni, két fontos kérdés merül fel:

1. Hogyan válasszuk ki a K db reprezentáló színt?
2. Hogyan rendeljük az egyes pixelekhez a színeket?

Természetesen ezek után rögtön következik a rendszeresen előforduló dilemma is, vagyis hogy mennyi időt és számítási kapacitást kell - illetve érdemes - erre a feladatra fordítani. A továbbiakban bemutatok néhány korábbi színkvantálásra adott algoritmust, majd két octree alapú megoldást. Mindegyikükben közös, hogy a színteret 3-dimenziós kockaként képzeli el, ahol az egyes koordinátatengelyek a piros, kék illetve zöld komponensnek felelnek meg.

3.1. Korábbi algoritmusok

3.1.1. Egyenletes felosztás

A legegyszerűbb módszer, ami nem függ a tömöríteni kívánt kép színeinek mennyiségétől és minőségétől. Osszuk fel az RGB-kockát mindhárom dimenziója mentén egyenlő részekre, ami $K = 256$ esetén legyen 8-8 rész a piros és zöld tengelyen, 4 rész pedig a kék tengely mentén (ugyanis a kék színre a legkevésbé érzékeny az emberi szem). Minden, a felosztás után kapott kisebb téglatest középső pontjának megfelelő szín kerüljön fel a színtáblára. Ezek után egy eredeti pixel színének helyét keressük meg a felosztott RGB-kockában, és helyettesítsük a megtalált kis téglatesthez tartozó színnel a színtáblából.
A téglatest középső pontja helyett választhatjuk az ebbe a térrészbe eső pixelek színeinek átlagát is, de Bloomberg [6] úgy találta, hogy ez nem változtat észrevehető mértékben a képminőségen.

3.1.2. Népszerűségi felosztás

Ez az algoritmus az eredeti képen legtöbbször előforduló K darab színt helyezi a színtáblába, amihez először szükséges a kép teljes átvizsgálása. Ennek során minden előforduló színnek meghatározzuk a relatív gyakoriságát, majd kiválasztjuk ezek közül a K legnagyobbal rendelkezőt. Ez után még hozzá kell rendelnünk a pixelekhez a színtáblabeli színeket. Természetes elképzelés, hogy a valamilyen szempontból "legközelebbi" reprezentást választjuk minden színhez, ez sokszor az euklideszi távolságot jelenti az RGB-kockában. Erről Heckbert [15] részletesebben ír cikkében, de térbeli legközelebbi szomszéd keresésére azóta már számos más algoritmus is született.
A módszernek három fő hátránya van:

- Ha az eredeti kép sokkal több színnel rendelkezik mint K, akkor kifejezetten nagy hibával rendel hozzá reprezentáns színeket a színtérben azoktól távol elhelyezkedő színekhez.

- A legközelebbi szomszéd keresés relatív nagy időigényű, függetlenül a használt algoritmustól.
- Dithering/zajmoduláció során szintén rosszul teljesít, mivel ilyenkor csak a reprezentáló színek konvex burkában lévő színeket alkalmazzuk. A dithering egy olyan folyamat, amely során a kvantált értékekhez kis méretű, véletlenszerű zajt adunk, így téve a hiba mértékét is véletlenszerűvé, ezzel sokszor szubjektíven javítva a vizuális élményt.

3.1.3. Medián vágás

A medián vágás algoritmus célja, hogy úgy válasszon ki K reprezentáló színt, hogy mindegyikük nagyjából azonos mennyiségű pixelt képviseljen a kvantálni kívánt képen. A folyamat során K téglatestre bontjuk az RGB-kockát, amelyhez szintén szükséges a teljes kép átvizsgálása és a pixelek elhelyezése a színtérben. Mind a $K-1$ felosztási lépésben azt a téglatestet osztjuk tovább, amely a legtöbb színt tartalmazza. Mindig a leghosszabb dimenziójára merőleges síkkal vágjuk két részre úgy, hogy azok körülbelül ugyanannyi pontot tartalmazzanak. Végül a színtáblába az egyes téglatestekben található színek átlagát tesszük. Amennyiben a felosztást k-d faként tároljuk, a pixelekhez történő színrendelés már gyorsan működik. Ez a módszer láthatóan jól teljesít a színtér azon részein, ahol nagy a kép színsűrűsége, viszont a ritkább területeknek megfelelő téglatestek nagy térfogatúak lesznek, relatív nagy hibát generálva ezzel. Ezt Bloomberg [6] a következőképpen javítja: A $K-1$ felosztás fele a legtöbb színnel rendelkező téglatestet, másik fele viszont a legnagyobb térfogatút osztja ketté. Ez a módszer egyrészt a téglatestek térfogatának csökkentésével a legnagyobb színhiba méretét is korlátozza, másrészt mivel a reprezentáló színek egyenletesebben kerülnek elosztásra a színtérben, a dithering során is jobb eredményeket kapunk.

3.2. Octree alapú algoritmusok

3.2.1. Gervautz és Purgathofer módszere színkvantálásra

Az algoritmus alapötlete a következő: a kép pixeleit tetszőleges sorrendben dolgozzuk fel, az első K előforduló színt ideiglenesen elhelyezzük a színtáblában. Amint egy újabb színre is szükségünk lenne, a $K+1$ színből két elég közelit összeolvasztunk, vagyis helyettesítjük az átlagukkal. Ezt a lépést minden újonnan előkerülő szín esetén ismételjük, így a színtáblában az algoritmus folyamán (így a végén is) minden pillanatban pontosan K szín található.
Vegyük a teljes RGB-kocka egy olyan octree-reprezentációját, melynek maximum mélysége 8. Így a zöld, a piros és a kék tengely mentén is 256 részre osztottuk a színteret, vagyis az octree minden levele pontosan egy színt képvisel. A köztes csúcsok reprezentálnak minden olyan színt, melyhez tartozó levél a csúcs leszármazottja, és minél kisebb egy köztes csúcs szintje, annál nagyobb részkocka feleltethető meg neki a színtérből. Így egy köztes csúcs szintje egy természetes korlátot ad az általa reprezentált színek távolságára.

Az algoritmus a medián vágáshoz hasonlóan három részből áll:

1. A reprezentáló színek meghatározása.
2. A színtábla kitöltése.
3. Színek rendelése a színtáblából a kép eredeti pixeleihez.

Reprezentáló színek meghatározása
Az algoritmus folyamán a teljes 8 szintű octreenek csak azon részét konstruáljuk meg, amely a képben előforduló színeket tartalmazza. Kezdetben az octree üres. A kép pixelein végighaladva, amennyiben olyan színt találunk, amely még nem szerepel az octreeben, vegyük fel a neki megfelelő 8. szintű levelet a fába. Ezzel a módszerrel egy hiányos octreet készítünk.
Valójában nincs is szükség minden szín felvételére, hiszen egyszerre mindig legfeljebb K színt tárolunk a színtáblában. Vagyis az octree építése folyamán, ha már K darab levele van a fának, és egy $K + 1$. levelet szeretnénk hozzáadni, akkor néhány közeli színt egybeolvasztunk, és a továbbiakban egy színnel reprezentáljuk őket. Ezt a lépést nevezzük az octree redukálásának.
Redukálás tehát akkor történik, amikor az octreenek $K + 1$ levele van. A folyamathoz a következő lépéseket hajtjuk végre:

1. Megkeressük a legnagyobb szintű olyan köztes csúcsot, amelynek egynél több gyereke van, ezt nevezzük v-nek.
2. A v-ből induló részfa v-n kívüli csúcsait töröljük az octreeből, így v levél lesz.
3. v-hez az előző lépésben törölt levelek által reprezentált színek átlagát rendeljük.

Természetesen előfordulhat, hogy az 1. pontban több azonos szintű, és egynél több gyerekkel rendelkező csúcs közül kell kiválasztanunk v-t. Ekkor az optimális eredmény érdekében többféle kritérium alapján választhatunk, például:

- Legyen az a csúcs v, amely eddig a pontig a legkevesebb pixelt reprezentálja. Így az algoritmus végén kapott hibaösszeget minimalizáljuk.
- Legyen az a csúcs v, amely eddig a pontig a legtöbb pixelt reprezentálja. Így a kvantált képen nagy kiterjedésű, eredetileg hasonló színű területek egyszínű (az eredetitől kis mértékben eltérő) színezést kapnak, de a nagyobb részletességű árnyékolás megmarad.

Az octree megépítéséhez tehát az összes pixel egyszeri átvizsgálása szükséges.

Színtábla kitöltése
Az 1. pontban felépített octree minden levele egy színtáblabeli színt reprezentál, ezeket kell feljegyeznünk. Ehhez a legmélyebb szintről indulva vizsgáljuk át az octree csúcsait, és amennyiben levél csúcsot találtunk, az általa reprezentált színt vegyük fel a színtáblába. A 3. lépés előkészítése érdekében az octree leveleihez rendeljük hozzá a hozzájuk tartozó színek színtáblabeli indexét is.

Színek rendelése a pixelekhez
Az eredeti kép pixelein újra végighaladva minden képponthoz hozzárendeljük a szükséges kvantált színt. Ehhez a pixel eredeti színét próbáljuk megkeresni az octreeben: a teljes színteret ábrázoló octreeben a gyökér és a keresett szín között egyértelmű út vezet. Ezen az úton elindulva a gyökérből lépjünk addig, ameddig tudunk. A keresés egy levél csúcsnál fog megállni valamilyen mélységben: ha ez a mélység 8, akkor az eredeti színét kapja a képpont. Mivel az előző lépésben minden levél csúcshoz eltároltuk a hozzá tartozó szín színtáblabeli indexét is, így a pixelhez egyből hozzárendelhető a kvantált színe.

Összességében, ha az eredeti kép nem tartalmazott K-nál több színt, akkor semmilyen változtatást nem végeztünk, ha viszont igen, akkor egy általunk használt c szín azon színeknek az átlaga, amelyekhez vezető út a gyökérből áthaladt azon a csúcson, amely végül c-t reprezentálja. Az octreevel való kvantálás a szerzők szerint vizuális minőségben a medián vágás által kapott eredményhez hasonlítható.

Számítási és memóriaigény
A kvantálás futásidejének szignifikáns részét egy megfelelő redukálható csúcs megkeresése teszi ki. Éppen ezért ezeket a csúcsokat az octree felépítése közben meg lehet találni, és egy megfelelő adatstruktúrában tárolni. Erre a célra nyolc lista tökéletesen megfelel, mindegyikbe egy-egy szint redukálható csúcsait tesszük. Így a legnagyobb szintű redukálható csúcs megkeresése konstans időben működik, a redukció szintén.

Vezessük be az alábbi jelöléseket:
N: Az eredeti kép pixeleinek száma.
K: A színtábla mérete.
D: Az eredeti kép különböző színeinek száma.
Általában ezen értékek között az $N > D > K$ és $N >> K$ relációk állnak fenn.
Az octree memóriaigényének felső korlátja $8K$ csúcs, ugyanis minden pillanatban legfeljebb K levele és $7K$ köztes csúcsa van (utóbbi abban az esetben, ha mind a K db 8. szintű levélhez egy-egy éldiszjunkt út vezet a gyökérből). Tehát a memóriaigény független N és D méretétől, csak a kvantáláshoz használt színek száma befolyásolja.
A lépésszám az algoritmus különböző szakaszaiban az alábbi:

- Octree felépítése: $\mathcal{O}(N * 8)$
  N-szer próbálunk meg új színt a fába illeszteni, ezt vagy 8 mélységben tesszük, vagy legfeljebb 8 mélységben már megtaláljuk a keresett színt, így nem szúrunk be újat. A redukálható csúcs megkeresése és a redukálás konstans idejű.

- Színtábla létrehozása: $\mathcal{O}(K * 2)$
  A megépített octreet egyszer végig kell járni, a levél csúcsok színeit felvenni a színtáblába.

- Színek rendelése a pixelekhez: $\mathcal{O}(N * 8)$
  N db pixel mindegyikéhez megkeressük az octreeben a reprezentáló színt, legfeljebb 8 mélységben meg is találjuk.

Tehát ez az octree alapú kvantálási módszer összességében $\mathcal{O}(N)$ idejű.

3.2.2. Bloomberg módszere

Bloomberg octree alapú színkvantálási algoritmusa [6] nagyban alapszik Gervautz és Purgathofer munkáján, lényegében az általuk adott módszer javításának tekinthető. A fa reprezentálásához lineáris kódolást használnak, így az $n-1$. szintű i indexű csúcs gyerekei a $8i+j, j = 0..7$ indexű n. szintű csúcsok. Így egy színhez tartozó csúcs indexét megkapjuk, ha az RGB kódjának bitjeit a legjelentősebbtől kezdve az alábbi sorrendben soroljuk fel (a szükséges szintig):

$$r1, g1, b1, r2, g2, b2, r3, g3, b3, r4, g4, b4, ...$$

Példa: Ha a $(123, 88, 204)$ RGB-kódú pixel 4. szintű csúcsát szeretnénk megtalálni, akkor először a kód kettes számrendszerbeli alakját írjuk fel: $(01111011, 01011000, 11001100)$. Ekkor $r1 = 0$, $g1 = 0$, $b1 = 1$. A biteket megfelelően sorbarendezve a 4. szintig az eredmény 001111100110, ami tízes számrendszerben a 998 indexű csúcsot jelöli.

Reprezentáló színek meghatározása
A reprezentáló színek kiválasztásához egy teljes octree kerül felépítésre, mely $MaxDepth$ mélységű (Bloomberg munkájában a $MaxDepth = 4, 5, 6$ lehetőségeket próbálta ki). Az első lépésében a kép összes pixelét átvizsgáljuk, és a fent leírt módszer szerint konstans időben meghatározzuk a hozzá tartozó $MaxDepth$ szintű csúcsot. Így minden levélhez eltároljuk, hogy az eredeti képen hány pixel színe esik az általa reprezentált színek közé. A szerző azt is kiemeli, hogy a futásidő csökkentése érdekében érdemes lehet az összes pixel csak egy bizonyos hányadát átvizsgálni, mivel a véletlen minta már egészen jól meghatározza a színek eloszlását. Ezután a korábbiakhoz hasonlóan a fa redukálása következik. A $MaxDepth$. szinttől kezdve minden szinten átvizsgáljuk a csúcsokat, egy lépésben egyszerre 8 testvért tekintve. Ezekre mindig az alábbi tulajdonságok közül egy vagy kettő teljesül, ahol CTE-nek azokat a csúcsokat nevezzük, amelyek egy-egy színtáblabeli színt képviselnek:

1. A csúcsok közül legalább 1 már CTE.

2. A nem CTE csúcsok között van olyan, amely elegendő pixellel rendelkezik ahhoz, hogy CTE legyen, így mostantól CTE.

3. Egyik csúcs sem CTE és nincs elég pixele a CTE-vé váláshoz.

Az első és második esetben a szülő csúcs automatikusan CTE-vé válik, azokat a gyerekeket tartalmazva, amelyek még nem voltak CTE-k. (Kivétel az az eset, amikor mind a 8 gyerek

CTE, ekkor a szülő csúcsot megjelöljük, de nem tartozik hozzá külön reprezentáló szín.) A harmadik esetben semmit nem változtatunk az adott szinten.

Azt, hogy mennyi pixel szükséges a CTE-hez váláshoz egy küszöbszám határozza meg. Ha az adott csúcs által reprezentált pixelek és a még nem reprezentált pixelek aránya meghaladja a küszöböt, akkor felvehetjük a csúcsot a CTE-k közé. A redukálást valójában csak az 2. szintig csináljuk, ugyanis mind a 64 második szintű csúcs eleve CTE. Így biztosítjuk, hogy az egész színtér le legyen fedve a reprezentáló színekkel és korlátozzuk a maximális hibát is.

Észrevehető, hogy ebben az algoritmusban nincs korlátozva a színtáblabeli színek száma. Ha az algoritmus K-nál több reprezentáló színt választ ki, újrafuttathatjuk, miközben módosítjuk $MaxDepth$ vagy a CTE-hez váláshoz szükséges küszöb értékét. Egy CTE által meghatározott reprezentáló szín legegyszerűbben itt is a csúcs középső pontja lehet.

Reprezentáló színek rendelése a pixelekhez
Két megoldás merül fel:

1. Minden levélhez explicit eltároljuk a hozzá tartozó reprezentáló szín indexét a színtáblából. Majd minden pixel RGB-kódjából kiszámoljuk a levél indexét és hozzárendeljük a levélnél tárolt reprezentáló színt.

2. Gervautz és Purgathofer algoritmusához hasonlóan minden pixel helyét a gyökérből indulva kezdjük el megkeresni az octreeben. Ha CTE-hez jutunk, hozzárendeljük a CTE által képviselt színt.

Olyan képeknél, ahol a pixelek száma nagyon meghaladja a legnagyobb szintű octree csúcsok számát, az első módszer hatékonyabb lesz, ellenkező esetben viszont érdemes a második módszert használni.

Számítási igény

- Octree felépítése: $\mathcal{O}(8^{MaxDepth} + N)$
  Az octreenek $8^{MaxDepth}$ db csúcsa van, a pixelek összeszámolása $\mathcal{O}(N)$ időt vesz igénybe.

- Redukálás: $\mathcal{O}(8^{MaxDepth-1})$
  A $MaxDepth - 1$. szinttől kezdve minden csúcsról konstans időben eldöntjük, hogy CTE-e, majd hozzárendeljük a középpontját, mint reprezentáló színt.

- Színek rendelése a pixelekhez: $\mathcal{O}(N * MaxDepth)$
  N db pixel mindegyikéhez megkeressük az octreeben a megfelelő CTE-t, legfeljebb $MaxDepth$ mélységben meg is találjuk.

Tehát ez az octree alapú kvantálási módszer is $\mathcal{O}(N)$ idejű, amennyiben a $MaxDepth$ paramétert előre meghatározott konstansnak tekintjük.

4. Ray tracing adaptív octree használatával

A ray tracing egy számítógépes grafikában használt eljárás, amely a fény terjedését, tárgyakkal való interakcióját szimulálja, így alakítja a virtuális háromdimenziós környezetet kétdimenziós képpé. A módszer során egy pontszerű kamerából (ezt nevezhetjük nézőpontnak, esetleg szemnek is) indított nagy számú fénysugár útját követjük. Minden sugár a kamera előtt elhelyezkedő kép egy pixelén halad át, amelyhez a sugár útja által meghatározott szín fog tartozni. A legegyszerűbb esetben (ray shooting) a virtuális térben csak homogén felületű objektumok találhatók, melyek minden irányból ugyanúgy vannak megvilágítva. Ekkor a feladat a következő [3]: Adott egy korlátos térrész, amely tartalmazza az összes objektumot, ezt nevezzük színtérnek. Adott továbbbá ezen n objektumból álló S halmaz a színtéren. Minden keresősugár esetében határozzuk meg a legelső olyan S-beli objektumot (amennyiben van ilyen), amellyel az útja során találkozik a keresősugár. Ennek eredményeképpen egy olyan kétdimenziós képet konstruálhatunk, melyben eltekintünk mindenféle fényvisszaverődéstől, árnyéktól illetve fényforrástól. A ray tracing tehát a ray shooting feladaton alapszik, annak általánosítása, melyben már figyelmbe veszünk tükröződő illetve áttetsző felületeket, lámpákat, ablakokat (melyek fényforrásként szolgálnak) és árnyékokat.

A ray tracing feladat legnagyobb kihívását a számítási idő csökkentése nyújtja, ugyanis bármiféle optimalizálás nélkül minden sugár-objektum párra ellenőriznünk kell, hogy metszik-e egymást. Erre a problémára számos megoldás született, ezek közül most Whang et al. [33] ötletét részletezem, amely egy adaptív octree felépítésén alapszik.

Látható, hogy ray tracing során nagyszámú sugár-objektum metszési tesztet kell végrehajtanunk. A számítási idő csökkentésének egyik módja a tér felosztása kisebb diszjunkt térrészekre, amelynek ábrázolásához kiválóan alkalmas az octree adatstruktúra. Az ilyen típusú algoritmusok fő tulajdonsága, hogy az egyes objektumokat a sugár haladási irányának megfelelő sorrendben vizsgáljuk meg, így az első sugarat metsző test megtalálásához nincs szükség elemek közötti rendezésre. Másrészről mivel csak azokat az objektumokat vizsgáljuk meg, amelyek a sugár útjában vagy ahhoz közel helyezkednek el, jelentősen csökkenthetjük a számítási időt.
A tér felosztásán alapuló módszereket két osztályba sorolhatjuk.

- Uniform felosztás. Előnye, hogy hatékonyan kiszámíthatók a sugár által metszett térrészek az uniformitást kihasználva [12]. Ezzel szemben az egyenletes felosztás nem veszi figyelembe a színtéren elhelyezkedő objektumok eloszlását, így például nagy üres térrészeket sok kisebb üres kockára bont. Egy ebbe az irányba kilőtt sugarat így mindegyik üres kockával is tesztelni kell, ami felesleges számításokat igényel.

- Nem uniform felosztás. Jellemzője, hogy a sűrű területeket (ahol több objektum helyezkedik el) több térrészre osztja, míg a ritkábbakat kevesebbre, így sokkal jobban alkalmazkodik a testek térbeli eloszlásához, mint az uniform változat.

Whang et al. [33] egy adaptív octree adatstruktúrát javasol a hatékony ray tracinghez, melynek az octree-R nevet adták, a fejezet hátralevő részében az ő munkájukat mutatom be. Először tekintsük át röviden az egyszerű octree alkalmazását a ray tracing feladatban: A gyökér csúcs a színteret reprezentálja. Amennyiben egy előre megadott limitnél több objektum található benne, a térbeli centrumot véve nyolc részre osztjuk. Minden új csúcsra ellenőrizzük, hogy eléri-e a limitet az általa metszett objektumok száma, és amennyiben így van, újraosztjuk. A sugárkövetés során mindig két lépésre lesz szükségünk: meghatározzuk, hogy azt a csúcsot, ahol éppen tart a sugár, melyik irányban hagyja el, illetve meghatározzuk az ezirányú szomszédot. Ezen lépésekkel cikkükben részletesen foglalkozik Endl és Sommer [9], de felhasználható a 2.4.1 fejezetben bemutatott szomszédos csúcsokat kereső algoritmus is.

4.1. Az octree-R

Az octree-R alapvetően ugyanazzal a szerkezettel rendelkezik, mint a hagyományos octree, a fő különbség az, hogy az adaptív fában a csúcsok centruma nem (feltétlenül) a térbeli középpont. Ahhoz, hogy az osztópontot megfelelően tudjuk kiválasztani, be kell vezetnünk egy költségmodellt, ahol a költség az egy sugár végigkövetéséhez szükséges időt jelenti. Legyen:

n_v: a sugár által metszett csúcsok száma
n_t: a sugár-objektum metszési tesztek száma
T_v: amennyi idő alatt tud a sugár a következő csúcsba jutni
T_t: egy sugár-objektum metszési tesz ideje

Ezekkel a jelölésekkel a teljes költség:

$$C = n_v * T_v + n_t * T_t$$

Megjegyzés: Tekintsünk két azonos színtérre épített octreet, melyek közül az egyiknek N csúcsa, a másiknak kevesebb, mint N csúcsa van. Ekkor a több csúccsal rendelkező octree felépítéséhez a teret többször osztottuk fel, így egy csúcsot átlagosan kevesebb objekum metsz, mint abban az octreeben, aminek N-nél kevesebb csúcsa van. Vagyis a csúcsok számának növelésével az n_t paraméter csökken. Azonban több csúcs létrehozásával egy sugár átlagosan többet is metsz közülük, így az n_v paraméter nő. Vagyis az n_v és n_t paraméterek mindig egymással ellentétes irányban változnak, az n_v paraméter pedig az octree csúcsszámával arányos. Így ha egy adott N-re N csúcsú octreek közül szeretnénk minimális költségűt találni, akkor elég az n_t paramétert minimalizálni.

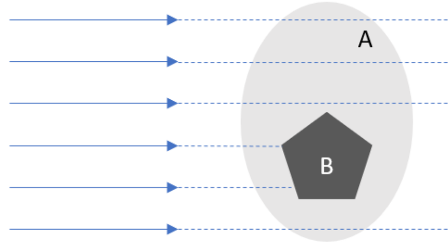
A következőkben leírásra kerül egy minimális költségű octree-R felépítésére szolgáló algoritmus. Ehhez először figyeljük meg, hogy két azonos színtérre épített, azonos csúcsszámmal rendelkező octreehez tartozó n_t paraméterek várható értékei különbözők lehetnek attól függően, hogy az egyes csúcsokat milyen síkkal osztjuk tovább. Következésképpen az algoritmus célja egy olyan octree-R fa építése, amely ezen várható értéket minimalizálja.

Az Octree-R felépítése:

Tegyük fel, hogy A térfogat tartalmazza B-t, ahogyan az a 3. ábrán látszik. Annak valószínűségét, hogy a sugár metszi B-t, feltéve, hogy metszi A-t, jelöljük $Pr(B|A)$-val. Feltéve, hogy a vizsgált sugarak egyenletesen oszlanak el, $Pr(B|A)$ megyegyezik a B és A merőleges vetületeinek átlagos nagyságának arányával [14]. Ez jól közelíthető a két objektum felszínének arányával, H.C. van de Hulst azt is belátta [32], hogy konvex objektumok esetében az átlagos vetített terület a felszín negyedével egyenlő. Így feltéve, hogy A és B konvex objektumok:

$$Pr(B|A) = \frac{< P(B,d) >}{< P(A,d) >} = \frac{S_B}{S_A}$$

Itt $P(V,d)$ a V merőleges vetülete d irányra, $<>$ a minden lehetséges d-re vett átlag, S_V a V felszíne.



3. ábra: $Pr(B|A)$ - Whang et al. alapján [33]

Célunk a sugár-objektum metszési tesztek várható értékének meghatározása. Feltesszük a sugarak egyenletes eloszlását a térben. Mivel egy csúcs mindig konvex, egy síkkal való felosztásakor pedig két konvex téglatestre bomlik, alkalmazhatjuk az előbbi egyenlőséget. Amennyiben egy R csúcsot osztunk fel a 4. ábra szerint, annak valószínűsége, hogy egy sugár metszi A-t (a felosztás által kapott egyik új csúcsot), feltéve, hogy metszi R-t a következő:

$$Pr(A|R) = \frac{S_A(t)}{S_R} = \frac{2(tb + tc + bc)}{2(ab + ac + bc)} \tag{1}$$

Itt $S_A(t)$ a felülete annak az új csúcsnak, amelyet akkor kapunk, ha az osztó sík az a hosszú élre merőleges, és azt t és $a - t$ hosszú élekre bontja. Hasonlóan annak valószínűsége, hogy egy sugár metszi B-t, feltéve, hogy metszi R-t:

$$Pr(B|R) = \frac{S_B(t)}{S_R} = \frac{(a - t)(b + c) + bc}{a(b + c) + bc} \tag{2}$$

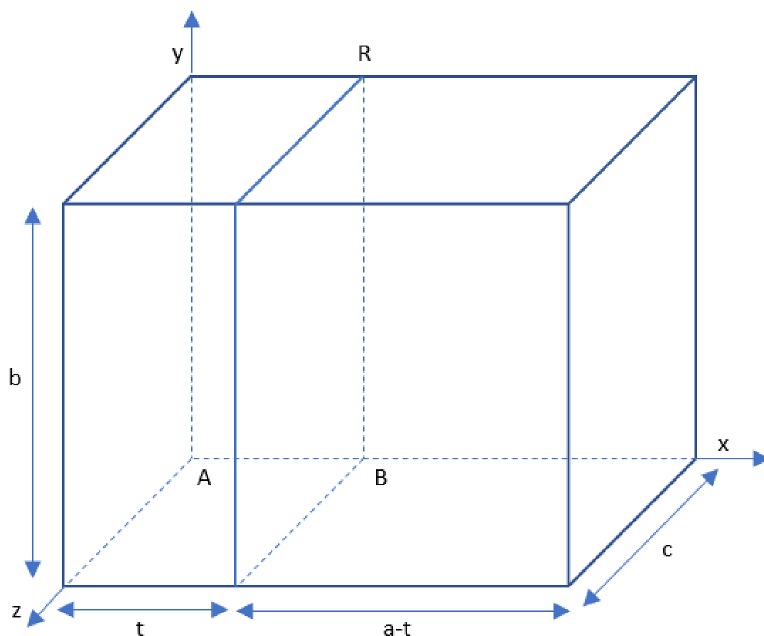
Először tegyük fel, hogy egyik színtéren lévő objektum sem metszi az osztó síkot, illetve $n(t)$ és $m(t)$ jelölje az A-ba és B-be belemetsző objektumok számát. Ekkor a sugár-objektum

metszési tesztek számának várható értéke, amikor a sugár az R csúcson halad át:

$$E(t) = \frac{S_A(t)}{S_R} n(t) + \frac{S_B(t)}{S_R} m(t) = \frac{t(b+c)+bc}{a(b+c)+bc} n(t) + \frac{(a-t)(b+c)+bc}{a(b+c)+bc} m(t) \quad (3)$$

Most tekintsük azt az esetet, ahol az objektumok metszhetik az osztó síkot is. Ekkor jelölje $n(t)$ és $m(t)$ azon objektumok számát, amelyek csak A-t illetve B-t metszik, $s(t)$ pedig az osztó síkot metszőkét. Ekkor a tesztek számának várható értéke a következőképpen módosul:

$$E(t) = \frac{S_A(t)}{S_R} n(t) + \frac{S_B(t)}{S_R} m(t) + \left[\frac{S_A(t)}{S_R} + \frac{S_B(t)}{S_R} s(t) \right] = \frac{t(b+c)+bc}{a(b+c)+bc} n(t) + \frac{(a-t)(b+c)+bc}{a(b+c)+bc} m(t) + \frac{a(b+c)+2bc}{a(b+c)+bc} s(t) \quad (4)$$



4. ábra: A legjobb vágósík megtalálása - Whang et al. alapján [33]

Megmutattuk tehát, hogy a sugár-objektum metszési tesztek számának várható értéke az osztó sík elhelyezkedésétől függ. Az octree-R felépítése során a tesztek számának csökkentése érdekében azon osztó síkot válasszuk, amelyikhez tartozó t érték minimalizálja az előző képletet. Ezen t megtalálásához azonban felhasználhatjuk McDonald és

Booth eredményét [23], amely szerint az optimális vágósík a térbeli közép (az a vágósík, amely a csúcs adott élét elfelezi) és az objektum közép (olyan vágósík, amelyre igaz, hogy az objektumok fele az egyik, másik fele a másik oldalán helyezkedik el) között fekszik. Az algoritmus futása során tehát, amikor egy csúcs teljesíti a kritériumokat és így nyolc oktánsra bontjuk, a következő történik:

- Megkeressük az X, Y és Z tengelyhez tartozó, (4)-et minimalizáló t_x, t_y, t_z értékeket.
- Ehhez a térbeli és az objektum közép közötti intervallumot $k + 1$ egyenlő részre osztjuk, ezek közül választjuk ki az optimális t-t.
- A (t_x, t_y, t_z) pontot centrumként használva felosztjuk a csúcsot 8 gyerekre, rájuk újra ellenőrizzük a felosztási kritériumot.

Whang et al. [33] kísérleteiben az octree-R használatával 4% és 47% közötti javulást sikerült elérni az egyszerű octreehez képest. Az octree-R az objektumok nemuniform eloszlása esetén mutatta a legnagyobb különbséget, ez magyarázható azzal, hogy uniform eloszlás esetén az azonos méretű csúcsok közel azonos számú objektumot tartalmaznak, így a (4)-et minimalizáló t közel esik a térbeli középhez. Szükséges még megemlíteni, hogy a 3 osztósíkot egymástól függetlenül választjuk, így elképzelhető, hogy együtt nem adnak optimális felosztást, de ennek vizsgálata már későbbi tanulmányok alapját képezi.

5. Ütközésérzékelés

Ericson [10] szerint az ütközésérzékelés egy rendkívül széles körű és sokak érdeklődését felkeltő, látszólag egyszerű probléma: döntsük el, hogy két (vagy több) objektum metszi-e egymást. Pontosabban három fő kérdésre keres választ: *metszik-e* egymást a testek, és amennyiben igen, *mikor* és *hol*. A válaszok megtalálásának nehézsége pedig durván ebben a sorrendben emelkedik. Az ütközésérzékelés azonban a számítógépes grafika megkerülhetetlen kihívása, alapját képezi a szimulációknak (animációk), robotikának, számítógépes prototípusok készítésének, mérnöki teszteléseknek és végül, de nem utolsó sorban a számítógépes játékoknak. Ebben a fejezetben egy octree alapú valós idejű ütközésérzékelési technikát mutatok be, amely olcsó hardver eszközökön is lehetővé teszi egy video avatar navigációját és interakcióit egy háromdimenziós virtuális környezetben. A következőkben Nakamura és Tori [29] leírása alapján mutatom be a módszer részleteit.

5.1. Video avatarok

A kiterjesztett valóság (AR - Augmented Reality) kifejezést a virtuális és valós világ egybeolvasztására, közös reprezentációjára használjuk. Ennek során például egy kamerával érzékelt valós környezetbe virtuális elemek integrálhatók. Az AR világának egyik legfigyelemfelkeltőbb alkalmazása a video avatarok használata. A video avatar tekinthető a felhasználó virtuális reprezentációjának, amely egy (vagy több) videokamera képén alapszik, és folyamatosan frissülve követi a valós személy mozgását, mozdulatait, gesztusait és ezáltal

interakcióit a virtuális környezettel. A video avatarok használata nem csak szórakoztatási célokat szolgálhat, önmagunk megfigyelése egy virtuális környezetben egy edzés részét is képezheti (Hämäläinen et al. tanulmányában [18] harcművészetek oktatására használták). Mindemellett egy avatar anélkül is hasznos lehet, hogy interakcióba tudna lépni a környezetével: egy videokonferencia során például növelheti az elmélyülést, személyességet és a tényleges jelenlét érzetét.

A video avatarok interakcióinak legelső lépcsőfoka annak érzékelése, hogy az hozzáér-e egy másik elemhez a környezetében. Ehhez elengedhetetlen az avatar két- vagy háromdimenziós modellje, amit használhatunk az ütközésérzékelés folyamán. Az elvárás, miszerint az ütközésérzékelés valós időben (pl. a videó fps értékének megfelelően) történjen jelentősen korlátozza a probléma megközelítési lehetőségeit. Sok ütközésérzékelési technika merev testekre (vagy azok valamilyen egységére) korlátozódik, azonban a video avatar geometriája előre nem ismert. Erre a problémára egy lehetséges megoldás, ha a videokamera képét a felhasználóról minden pillanatban egy-egy előre meghatározott emberszerű modellhez párosítjuk. Ennek a megközelítésnek két fő előnye van:

- Eredményeképpen minden pillanatban egy teljes geometriai modell áll rendelkezésünkre, amellyel akár nagyon precíz ütközésérzékelés is végezhető.
- Lehetőségünk van azt is meghatározni, hogy az avatar melyik része érint egy másik elemet.

Ez a módszer azonban nagy számítási igénnyel rendelkezik, akár több számítógép egyidejű futására is szükség lehet. Emellett több különböző nézőpontú kamera képére is szükség van, hogy a felhasználó valós helyzetéhez a legmegfelelőbb geometriai modellt rendeljük. Mivel a szerzők kifejezett célja volt, hogy akár egy egyszerű hétköznapi számítógép is elegendő legyen a valós idejű ütközésérzékeléshez, ezt a modellt elvetették. Ehelyett végül térfogati megvalósítás mellett döntöttek, amely a video avatar depth mapjén (szürkeárnyalatos mélységi kép) alapszik. Általában a depth mapet alkalmazó technikák létrehoznak egy poligon hálót a mélységi képből, Nakamura és Tori azonban egy octree felépítéséhez használják. A továbbiakban a video avatar és a virtuális elemek közötti érintkezések detektálása a feladat. Néhány video avatart használó alkalmazásban kétdimenziós ütközésérzékelést alkalmaznak. Ehhez összesen egy kamera képe szükséges, ez alapján feltérképezésre kerül a felhasználó sziluettje. Ezután az avatar és az elemek sziluettjének síkbeli metszése könnyen számítható. Ennek a módszernek két előnye van: egyrészt mivel csak egy kamerára van szükség, elkerülhetők a kameraképek szinkronizációjával járó nehézségek, másrészt a megvalósítás kisebb számítási igénnyel bír. Ezek a faktorok lehetővé teszik az olcsó hardvereken történő felhasználást, azonban a technika a mélység reprezentációjának hiányával jelentősen limitálja a felhasználó interakcióit a környezetben. Utóbbi következtében a szerzők módszerükben inkább két kamera képéből meghatározzák a virtuális kép mélységi információit is.

5.2. AVMIX rövid áttkintése

Az AVMIX az a video avatarokat kezelő rendszer, amelyhez Nakamura és Tori az ütközésérzékelés technikájukat javasolták. A rendszer részletes leírása megtalálható a szerzők korábbi cikkében [28]. Az AVMIX első számú célkitűzése, hogy otthoni és oktatási célokra alkalmazható legyen, így nem igényelhet magasabb szintű hardvereket, mint egy átlagos egyéni felhasználásra készült számítógép vagy videokamera. A video avatar rendszer 5 fő alegységből áll:

1. *Image Acquisition* - a videokamerákból érkező képkockák lekérdezéséért, valamint a képek torzításkorrekciójáért és rektifikációjáért felelős.
2. *Segmentation* - eltávolítja a hátteret a képekből, így a felhasználó képe homogén háttér előtt jelenik meg.
3. *Depth Mapping* - A szegmentált képkockákból szürkeárnyalatos mélységi képet készít.
4. *Rendering* - a video avatar vizuális reprezentációját készíti el.
5. *Collision detection* - Ütközésérzékelésért felelős alrendszer, amellyel részletesen foglalkozunk ebben a fejezetben.

A *collision detection* alegység a depth mapping outputjától függ. Ez az output egy szürkeárnyalatos mélységi kép a video avatart éppen tartalmazó jelenetről. A mélység - avagy a távolság az objektum és a kamera között - a következő egyenlettel számítható minden pixel esetében:

$$D = \frac{f * B}{w} \tag{1}$$

ahol f a kameralencsék fókusztávolsága, B a két kamera közötti távolság, w a disparity (egy pont x-koordinátájának a két kamera által látott két képen tapasztalható különbsége).

5.3. Ütközésérzékelés octreevel

A szerzők által javasolt módszer alapja, hogy a video avatar egy octreevel kerül reprezentálásra. Az octree gyökere az egész tér, amiben a video avatar mozogni tud, a csúcs gyerekei az adatstruktúrának megfelelően a tér 8 diszjunkt részre való osztását jelképezik. A mozgókép minden képkockájához tehát először kiszámítjuk az octreet, majd végrehajtjuk az ütközésérzékelést.

Octree felépítése
A octree felépítéséhez két lépésre van szükség: meghatározni a video avatart felépítő háromdimenziós pontok koordinátáit és meghatározni a helyüket az adatstruktúrában. Ha ez megtörtént, minden olyan csúcsot a fában, amely legalább egy pontját tartalmazza az

avatarnak megjelölünk "kitöltöttnek". Chen et al. [22] alapján a depth map p_{ij} pixelének térbeli koordinátái a következő módon számíthatók ki:

$$x_{ij} = D * tan(\alpha) * \frac{i - \frac{W}{2}}{\frac{W}{2}} \tag{2.1}$$

$$y_{ij} = D * tan(\beta) * \frac{\frac{H}{2} - j}{\frac{H}{2}} \tag{2.2}$$

$$z_{ij} = D \tag{2.3}$$

Az egyenletekben α és β a kamerák vízszintes és függőleges látószögének fele, W és H a mozgókép egy képkocájának dimenziói (szélesség és magasság), D pedig az (1)-ből eredményül kapott mélységi érték.
Miután egy pont térbeli koordinátái kiszámításra kerültek, a pontot elhelyezzük az octree megfelelő levelében, amelyet így "kitöltöttnek" is jelölünk.

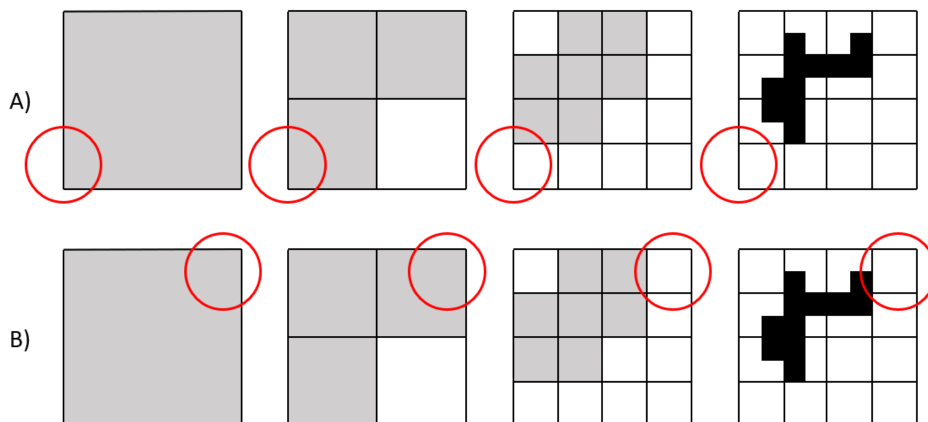
Ütközésérzékelés
Az AVMIX rendszerben minden olyan környezetbeli elem, amellyel a video avatar kapcsolatba léphet kétféleképpen kerül reprezentálásra: az objektum körülhatároló dobozával vagy gömbjével. Az ütközésérzékelés tehát az octree és a doboz/gömb között játszódik le, rekurzív módon. Pontos implementálását részletesen leírja Ericson [10]. Első lépésben az octree gyökere és az objektum között végzünk el metszési tesztet. Amennyiben ezek metszik egymást, a gyökér csúcs minden "kitöltöttnek" jelölt gyerekével folytatjuk a tesztek elvégzését egészen addig, amíg az objektum

- egy TELJES (levél) csúcsot metsz, vagy
- az octree egy szintjén egyetlen "kitöltött" csúcsot sem metsz.

Az első esetben valós ütközést észleltünk, egyébként nem történt ilyen. Az 5. ábra [29] 2 dimenzióban mutat meg egy 3 szintű quadtreevel történő ütközésérzékelést. Az A) esetben a 3. szinten egy kitöltött csúcsot sem metsz az objektum, így nem történt ütközés, a B) esetben viszont igen.

Megvalósítás
Az octreeket általában dinamikus adatstruktúraként szokták tárolni, ami tárkihasználás szempontjából előnyös. Ezen alkalmazás közben azonban minden képkockához fel kell építeni egy octreet, így a teljesítmény növelése érdekében a szerzők az octree statikus (lista alapú) tárolását választották. Egy K mélységű teljes octree

$$N = \sum_{i=0}^{K} 8^i$$



5. ábra: Ütközésérzékelés rekurzívan

csúccsal rendelkezik. Az octree listaként történő tárolásának fő hátránya, hogy a csúcsok száma - és így a szükséges memória mérete - a fa mélységének növelésével exponenciálisan nő. $K = 5$ esetben $N = 37449$, míg $K = 6$ esetben a szükséges csúcsok száma már $N = 299593$, tehát a gyors ütközésérzékelés érdekében érdemes az octree mélységét legfeljebb 5-re korlátozni. Ekkor minden tengely mentén 32 részre oszthatjuk a teret, amivel egy 2 m oldalhosszúságú gyökér esetén az 5. szinten levő csúcsok 62,5 mm oldalhosszúságúak lehetnek. Ez a precizitás ugyan nem teszi lehetővé a kisebb részletek (pl. ujjak) precíz ábrázolását, de elegendő lehet egy egészalakos video avatar interakcióinak reprezentálásához.

A térbeli pontokat a háromdimenziós koordinátájuk alapján a Morton-kód használatával egyből a megfelelő levél csúcshoz rendelhetjük, ahogyan arról részletesen írtam a 2.4 fejezetben. A Morton-kód szintén korlátozhatja a fa mélységét, ugyanis 32 bites előjel nélküli egész számokat használva minden koordináta maximum 10 bites lehet. Ez limitálná a mélységet 10-re, azonban ez gyengébb korlátozás az előzőleg praktikai okok miatt meghatározott 5 mélységnél. Az általános megközelítés a következő lenne: az avatar egy pontját a gyökértől indulva rekurzívan teszteljük az octree csúcsaival, így keressük meg a megfelelő levelet. Ez egy K szintű octree esetén pontonként $K + 1$ művelet, mindegyikben 3 lebegőpontos összehasonlítással. Ezzel szemben a Morton-kód használatával közvetlenül meghatározható a megfelelő levél. Ekkor viszont alulról felfelé haladva még meg kell jelöljük a csúcsokat, amelyek ”kitöltöttek”. Az alábbi táblázat azt mutatja, hogy hány művelet (pont klasszifikáció, csúcsok bejárása) szükséges az egyes megközelítéseket használva, amennyiben az avatar 10000 pontból áll (a tesztek során átlagosan ekkora volt a ponthalmaz mérete).
Megfigyelhető, hogy a Morton-kód használata előnyösebb $1 \leq 5 \leq 6$ esetén, míg nagyobb mélységű fa használatával a pontok számának exponenciális növekedése miatt a

	Műveletek száma	
Octree mélysége	Rekurzív kereséssel	Morton-kód használatával
0	10000	10001
1	20000	10009
2	30000	10073
3	40000	10585
4	50000	14681
5	60000	47449
6	70000	309593
7	80000	2406745

Műveletek számának összehasonlítása
Nakamura és Tori [29] munkája alapján

Morton-kód kevésbé hatékony. Megjegyzendő még, hogy a pontok klasszifikációja nem függ egymástól, így párhuzamosan végezhető.

6. Boole-operációk

Boole-operációk geometriai modelleken történő végrehajtása a számítógépes grafika egyik megkerülhetetlen feladata. A modellezés leggyakrabban B-Rep módszerrel történik, amely során a szilárd testeket határoló felületek halmazaként adjuk meg, ezek választják el a belső és külső pontokat. Számos reprezentációban a felületek csak síklapok lehetnek. A 3D nyomtatás és a reverse engineering (fordított tervezés) elterjedése teret adott a diszkrét felületek széles körű alkalmazásának, így váltak leggyakrabban használttá a háromszögelt objektumok (meshek). Ezen modellezés során egy test B-Rep ábrázolása csak háromszögeket alkalmaz határoló felületekként. A következőkben egy zárt háromszögelt felületek közötti Boole-operációkat (unió, metszet, különbségképzés) végrehajtó algoritmus részleteit mutatom be, amelyet C++ nyelven meg is valósítottam. A módszer és a fejezet alapjául Mei és Tipper [26] cikke szolgált, azonban azt számos alkalommal kiegészítettem, bizonyos részalgoritmusokat megváltoztattam, egyszerűsítettem vagy gyorsítottam.

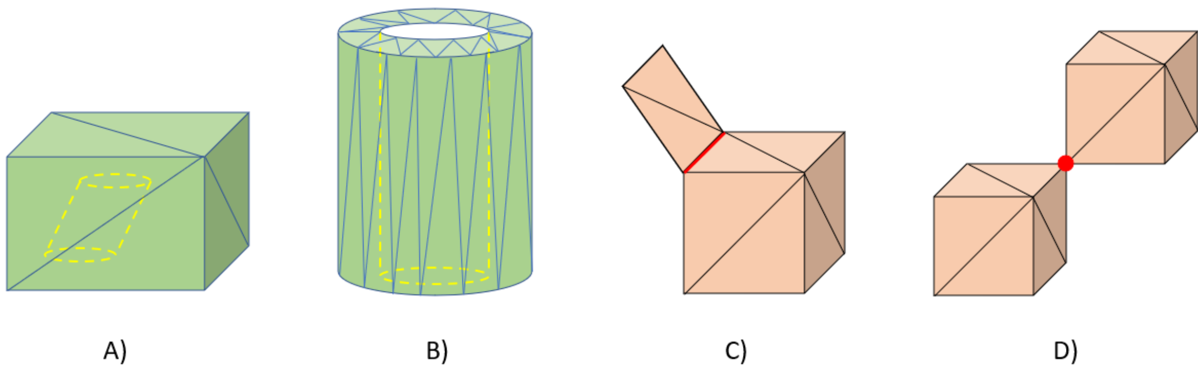
6.1. Általános áttekintés

A Boole-operációkat elvégző algoritmusok általában két fő fázisra oszthatók:

1. *Háromszögek által meghatározott metszésvonalak kiszámítása.* Az algoritmus kezdetén meghatározzuk a lehetséges metsző háromszögpárokat, mindezt egy octree felépítésével és átvizsgálásával tesszük hatékonnyá. Kiszámítjuk a metszésvonalakat, eltároljuk azokat, majd az elmetszett háromszögeket újraosztjuk.

2. *Blokkok előállítása, majd az eredményül kapott objektumok meghatározása.* Ebben a fázisban a metszethurkokból patcheket (háromszögelt részfelületeket), azokból pedig részblokkokat (zárt háromszögelt felület által határolt térfogat) állítunk elő. Végül kategorizáljuk a részblokkokat, miszerint azok az unió, a metszet vagy valamelyik különbség részét képezik.

A legtöbb alkalmazás során elvárható, hogy a háromszögelt objektumok rendelkezzenek a topológiai sokaság fogalmával, így ezt az algoritmus inputjára is feltételezzük. Továbbá nem foglalkozunk az önmagukat metsző meshekkel sem. A 6. ábrán tehát az algoritmus az A) és B) eseteketre alkalmazható, a C) és D) esetekre azonban nem. Továbbá különítsük el az A $\subseteq$ B és B $\subseteq$ A eseteket, ezek egyszerűen meggondolható eredményre vezetnek.



6. ábra: A) és B) teljesíti a sokaság feltevést, C) és D) azonban nem

Mesh modellek Boole-operációira számtalan algoritmus született az évek során, hamar kiderült azonban, hogy hatékonyság szempontjából két kulcsfolyamatot találunk:

1. Hogyan találjuk meg az összes metszésvonalat, illetve rendezzük őket hurkokba? E feladat hatékony megoldásának alapja a lehetséges metsző háromszögek pontos és gyors meghatározása, melyhez az octree alkalmazása nagy segítséget nyújt.

2. Hogyan gyűjtsük össze és kategorizáljuk megfelelően a részblokkokat unió, metszet és kétféle különbség csoportokba? Leggyakrabban a pontok belső/külső klasszifikálását választják, amely az egyes pontokhoz ellenőrzi, hogy a megfelelő blokkon belül vagy kívül helyezkedik el. Ennek részletes leírása megtalálható Jiang et al. [20] cikkében. Az itt leírt algoritmusban azonban egy teljesen más csoportosítási módszert alkalmazunk, amelyet a 6.2.6 fejezetben mutatok be.

Az újabban megjelenő Boole-operációt végrehajtó algoritmusok tehát többségében ezt a két folyamatot igyekeznek hatékonyabbá tenni.

6.2. Az algoritmus

1. lépés: Metsző háromszögek megkeresése

Mielőtt bármely két háromszöget metszési teszt alá vetnénk, fontos meghatározni azokat a háromszögpárokat, amelyeket egyáltalán érdemes tesztelni, tehát lehet, hogy metszik egymást. A legegyszerűbb (de annál lassabb) módszer az egyik felület minden háromszögének határoló dobozát hasonlítaná össze a másik felület ugyanilyenjeivel. Ezt a folyamatot tudjuk gyorsabbá tenni az octree használatával.
Legyen a két (A és B) mesh együttes körülzáró (vagy határoló) doboza az octree gyökere. Jelölje N_A, illetve N_B azon A és B-beli háromszögek számát, amelyek metszik az octree N csúcsát. Az octree felépítésének szabályai szerint a térbeli centrumot használva osszuk mindig 8 gyerekre a csúcsot, ha az egyik megállási feltételt sem teljesíti. Ezek a következők:

- A csúcs mélysége eléri az előre meghatározott max_depth értéket.
- N_a és N_b értéke is kisebb, mint az előre megadott $min_triangles$ érték.
- N_a vagy N_b nulla.

Az octree méretének csökkentése érdekében gyökérnek választhatjuk a két mesh "közös részét" is [20], amelyet a következőképpen kapunk meg: számítsuk ki A és B meshek határoló dobozait, melyek legyenek Box_A és Box_B. Legyen $Box_{AB} = Box_A \cap Box_B$. Növeljük meg Box_{AB} éleit minden irányban l értékkel, amely A és B leghosszabb élének hossza. Az így kapott téglatestet válasszuk az octree gyökerének.

Annak ellenőrzése, hogy egy háromszög metsz-e egy octree csúcsot, történhet egy egyszerű metszési teszttel a háromszög határoló doboza és a csúcs között. Ha azok metszik egymást, a háromszög tekinthető a csúcsba tartozónak, a hatékonyság érdekében pedig előre kiszámítható és eltárolható minden háromszög határoló doboza. Az algoritmus megvalósítása során én azonban a pontos meszési teszt mellett döntöttem, melyet a következőképpen implementáltam:

- Egy octree csúcsba csak azok a háromszögek tartozhatnak, amelyek a csúcs szülőjéhez is tartoztak, így mindig csak ezeket a háromszögeket teszteljük.
- Legyen a centrum (amely szerint felosztottuk a szülő csúcsot) az (x_c, y_c, z_c) koordinátájú pont, a T háromszög csúcsai pedig legyenek v_1, v_2, v_3. Tekintsük a háromszög és a jobb-felső-hátsó gyerek (melynek x, y és z koordinátái a legnagyobbak) közti metszési tesztet, a többi hasonlóan működik.
- 1. lépés: Hasonlítsuk össze T három csúcsának x-koordinátáit x_c-vel:
  - Mindhárom kisebb, mint x_c: T nem tartozik a csúcshoz.
  - 1 nagyobb, 2 kisebb x_c-nél: Legyen v_1 x-koordinátája nagyobb x_c-nél. Ekkor a (v_1, m_1, m_2) háromszöggel folytatjuk a következő lépést, itt m_1 és m_2 azok a pontok, ahol a (v_1, v_2) és a (v_1, v_3) élek metszik az $x = x_c$ síkot.

  - 2 nagyobb, 1 kisebb x_c-nél: Legyen v_1 és v_2 x-koordinátája nagyobb x_c-nél. Ekkor a (v_1, m_1, v_2) és (v_2, m_1, m_2) háromszögekkel folytatjuk a következő lépést, itt m_1 és m_2 azok a pontok, ahol a (v_1, v_3) és a (v_2, v_3) élek metszik az $x = x_c$ síkot.
  - Mindhárom nagyobb x_c-nél: T-vel folytatjuk a következő lépést.

- 2. lépés: Hasonlítsuk össze az előző lépésben meghatározott háromszög(ek) három csúcsának y-koordinátáit y_c-vel, a lehetséges esetek az 1. lépéshez hasonlóak.
- 3. lépés: Ismételjük a 2. lépést z-koordinátákkal, ha bármelyik csúcs z-koordinátája nagyobb z_c-nél, akkor T a csúcshoz tartozik, ellenkező esetben nem.

Az octree felépítése során csak a levél csúcsoknál tároltuk el ténylegesen a hozzá tartozó háromszögeket, a köztes csúcsoknál csak átmenetileg tartottuk fenn azokat. Észrevehetjük továbbá, hogy egy háromszög több levél csúcshoz is tartozhat.

2. lépés: Háromszögek metszése és metszethurkok konstruálása

Két háromszög metszésére Möller [27] egy egyszerű és hatékony algoritmust ír le, én is ezt implementáltam, a továbbiakban pedig röviden összefoglalom.

Háromszögek metszése Möller alapján

Jelölje a két háromszöget T_1 és T_2, a nekik megfelelő csúcsokat V_0^1, V_1^1, V_2^1, illetve V_0^2, V_1^2, V_2^2, a két háromszög síkját pedig jelölje π_1 és π_2. A π_2 sík egyenlete: $N_2 \cdot x + d_2 = 0$, ahol:

$$\begin{aligned} N_2 &= (V_1^2 - V_0^2) \times (V_2^2 - V_0^2) \\ d_2 &= -N_2 \cdot V_0^2 \end{aligned} \tag{1}$$

Ekkor T_1 csúcsainak előjeles távolsága π_2 síktól (egy konstanssal megszorozva) kiszámítható a csúcs behelyettesítésével a sík egyenletébe:

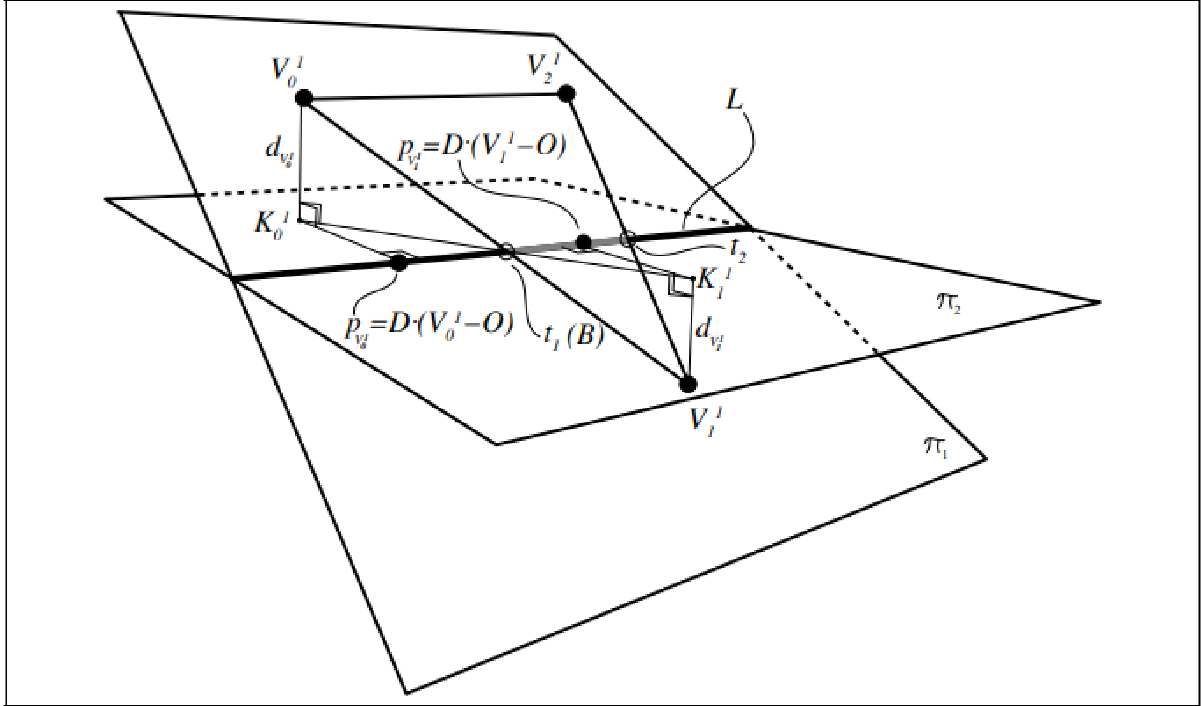
$$d_{V_i^1} = N_2 \cdot V_i^1 + d_2, \quad i = 0, 1, 2 \tag{2}$$

Ha minden $i = 0, 1, 2$-re $d_{V_i^1} \neq 0$ (vagyis nem esnek T_1 csúcsai a π_2 síkra), és előjeleik megegyeznek, akkor T_1 egészében a π_2 sík egyik oldalán helyezkedik el, így nem metszheti egymást a két háromszög. Ugyanezt az ellenőrzést megtehetjük T_2 és π_1 esetén. Ha minden $i = 0, 1, 2$-re $d_{V_i^1} = 0$, akkor a háromszögek egy síkba esnek, ezt az esetet később külön kezeljük. Ha ez nem áll fenn, akkor π_1 és π_2 metszete egy egyenes, melynek egyenlete: $L = O + tD$, ahol $D = N_1 \times N_2$ az egyenes iránya, O pedig egy tetszőleges pontja. A két háromszög a korábbi esetszétválasztások miatt már biztosan metszi L-t, a két metszés pedig két (esetleg elfajuló) intervallumot határoz meg az egyenesen. Ha az intervallumok átfedésben vannak, akkor a két háromszög metszi egymást.

Tegyük fel, hogy V_0^1 és V_2^1 π_2 egyik oldalán helyezkedik el, V_1^1 pedig a másikon, és számítsuk ki az intervallumot, ami T_1 és L metszetének felel meg. Ehhez a (V_0^1, V_1^1) és (V_1^1, V_2^1) élek L egyenessel vett metszéspontját határozzuk meg. Először vetítsük T_1 csúcsait L-re:

$$p_{V_i^1} = D \cdot (V_i^1 - O) \tag{3}$$

Ezután számítsuk ki t_1-et, amelyre igaz, hogy B pont, ami (V_0^1, V_1^1) él és L metszete megegyezik $O + t_1 D$-vel. Jelölje K_i^1 a V_i^1 csúcs π_2-re vett vetületét. Az eredeti cikk [27] alábbi ábrája mutatja a pontok helyzetét.



7. ábra: Háromszögek metszési tesztje

Megfigyelhető, hogy a (V_0^1, B, K_0^1) és (V_1^1, B, K_1^1) háromszögek hasonlóak, így fennáll a következő egyenlőség:

$$t_1 = p_{V_0^1} + (p_{V_1^1} - p_{V_0^1}) \frac{d_{V_0^1}}{d_{V_0^1} - d_{V_1^1}} \tag{4}$$

Hasonló módon kiszámítható t_2 és a T_2-höz tartozó intervallum is. Ha tehát ezek az intervallumok metszik egymást, akkor a háromszögek is, és az $O + t_i D$ képlet felhasználásával a metszetszakasz végpontjait is megkapjuk.

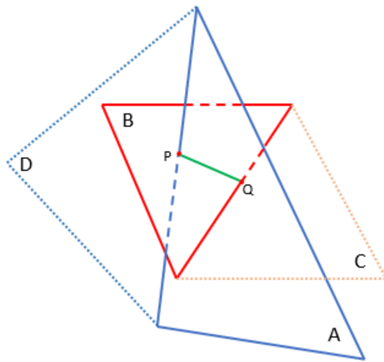
Ha a két háromszög egy síkban fekszik, vetítsük azokat arra a két tengely által meghatározott síkra, ahol maximális a területük. Itt már egyszerű síkbeli háromszögek közötti

metszési tesztet kell végrehajtani. Teszteljük T_1 és T_2 minden élét, hogy metszik-e egymást. Ha igen, akkor a háromszögek metszők. Ha nem, még ellenőriznünk kell, hogy az egyik háromszög tartalmazza-e a másikat, ezt pedig megtehetjük egy egyszerű pont-a-háromszögben teszttel.

Metszethurkok konstruálása

A háromszögek metszési szakaszait érdemes csoportosítanunk. Zárt meshek Boole-operációi esetén ezek mindig gráfelméleti értelemben vett köröket alkotnak, azaz minden metszetszakasz egyik végpontja egy másik szakasz egyik végpontjával egyezik meg. Ezeket a köröket nevezzük hurkoknak (vagy metszethurkoknak), amelyeket eltárolhatunk csupán a szakaszok végpontjainak felsorolásával. Mei és Tipper [26] munkájában a hurkok konstruálása az összes háromszög elmetszése után történik, ezután gyűjtik össze minden szakaszhoz a hurokban utána következőt (az összes eddig nem hurokba sorolt szakasz közül olyat keresnek, amely végpontja megegyezik azzal a ponttal, amihez szomszédot keresünk). Ez a módszer egyáltalán nem használja ki azt, hogy a metszetszakaszokat mely háromszögek adták és azok a térben hol helyezkednek el, így én egy másik módszer mellett döntöttem.

Az STL formátumú fájl (háromszögelt meshek gyakori formátuma) beolvasása során külön vektorokban eltároltam a háromszögeket, a csúcsokat és az éleket is, utóbbiak pedig tartalmazzák, hogy mely két háromszöghöz tartoznak. Ez segített abban, hogy a háromszögek metszését és a metszetszakaszok hurkokba rendezését egyszerre végezzem. Ha a program talál két háromszöget, melyek metszik egymást, létrehoz egy új hurkot, melynek első szakasza a most megtalált metszet. Ebben a metszetben eltároljuk azt az információt is, hogy a két végpont a két háromszöghöz (T_1 és T_2) hogyan viszonyul: területükön belül helyezkedik el, vagy valamelyik élükön. Fontos megjegyezni, hogy egy végpont sosem lehet mindkét háromszög belsejében. Ezek után vesszük az egyik végpontot, és megnézzük, hogy melyik háromszög az, amelyiknek az élén helyezkedik el. Tegyük fel, hogy az A meshhez tartozó T_1 háromszög e élére esik a végpont, a B meshhez tartozó T_2 háromszögnek viszont belső pontja. A többi esetet a 8. ábra mutatja. Ekkor a következő lépésben az e-hez tartozó T_1-től különböző és T_2 háromszög metszését vizsgáljuk, a metszet pedig pont az elindított hurok következő szakasza lesz. Ezt a folyamatot folytatjuk addig, amíg egy olyan szakaszhoz érünk, amelynek mindkét végpontja már a hurok eleme. Ezzel lezártunk egy hurkot, újabb két, eddig egymással szemben nem vizsgált háromszög metszése esetén újabb hurkot konstruálunk.



B mesh \ A mesh	Élen	Belül
Élen	A: szomszéd B: szomszéd	A: marad B: szomszéd
Belül	B: marad A: szomszéd	-

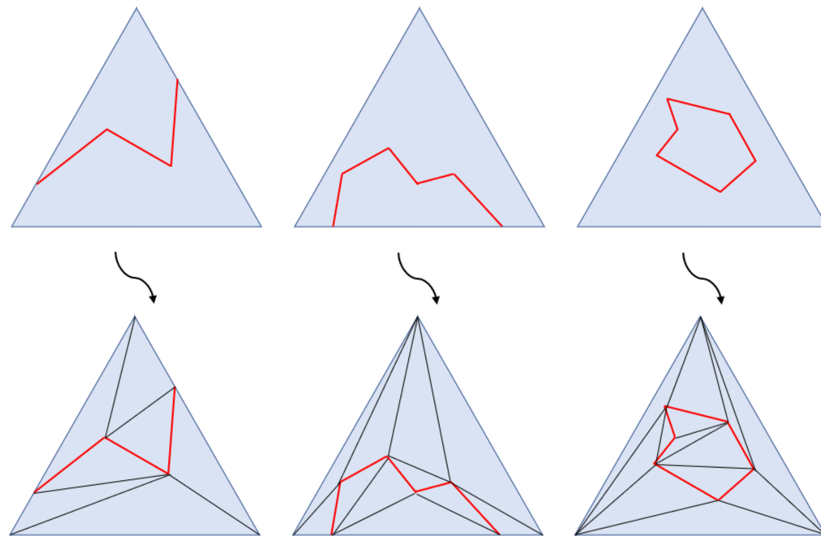
8. ábra: P az A háromszög élén helyezkedik el és a B háromszög belső pontja, Q fordítva. Így a következő metszés D és B vagy A és C háromszög között történik.

3. lépés: Újraháromszögelés

Minden lehetséges háromszög-pár elmetszése után egy háromszögön belül általában több metszetszakasz található, melyek több poligonra osztják azt. A szerzők az összes metszetszakasz kiszámítása után gyűjtik össze azokat a szakaszokat, amelyek egy adott háromszöghöz tartoznak, ezeket a metszethurkokhoz hasonló körökbe rendezik (melyek csúcsai a háromszög csúcsai is lehetnek), majd ezután újraháromszögelik a kapott poligonokat. Az implementálás során én az újraháromszögelést is a metszethurkok konstruálása közben láttam észszerűnek elvégezni. Egy háromszög "kész" állapotba kerül, ha egy adott hurok generálása során már az alábbi 3 lehetőség közül legalább az egyik teljesül:

- A hurok elért a háromszög egyik élétől a másikig, vagyis van a hurokban egy csúcs, amely a háromszög egyik élén fekszik, az összes következő csúcs a háromszögön belül van, a hurok utolsó csúcsa pedig a háromszög egy másik élén helyezkedik el.
- Az előbbi esethez hasonló, de a két élen elhelyezkedő csúcs ugyanazon élen fekszik.
- A hurok elkészült, és mindegyik csúcs a háromszögön belül van.

Amint egy háromszög "kész", újraosztjuk azt háromszögekre. Ennek előnye, hogy a háromszög mindig pontosan 2 poligonra bomlik, amelyeket egy ear-clipping algoritmussal [16] háromszögelünk újra. Ennek alapja, hogy egy poligonnak mindig létezik füle: olyan v_i csúcs, amelyre a v_{i-1}, v_i, v_{i+1} szög konvex, és az általuk alkotott háromszög nem tartalmaz a poligon többi csúcsa közül egyet sem. Ez a fül "levágható", azaz a (v_{i-1}, v_i, v_{i+1}) háromszöget bevesszük a háromszögelésbe, a poligont pedig frissítjük: töröljük a v_i csúcsot. A folyamatot rekurzívan folytatva fel tudjuk osztani a (konvex vagy konkáv) poligont háromszögekre. Kis változtatással az algoritmus alkalmazható olyan poligonokra is, amelyeket nem egy töröttvonal határol, azaz "lyukak" vagy "szigetek" találhatók benne, ami a mi esetünkben előfordulhat, ha egy metszethurok teljes egészében egy háromszögön belül helyezkedik el. Azonban mivel rögtön egy háromszög "kész" állapotba kerülése után



9. ábra: Újraháromszögelés attól függően, hogy a háromszög melyik feltétel miatt került "kész" állapotba

újraosztjuk azt, mindig legfeljebb egy szigettel kell foglalkoznunk. Fontos még megjegyezni, hogy az ear-clipping algoritmus alkalmazásához ellenőriznünk kell, hogy a poligon csúcsai pozitív irányban legyenek sorszámozva, illetve az elfajuló háromszögek megjelenésének elkerülése céljából érdemes mindig a legnagyobb területű háromszöget adó fület levágni.

4. lépés: Frissítés

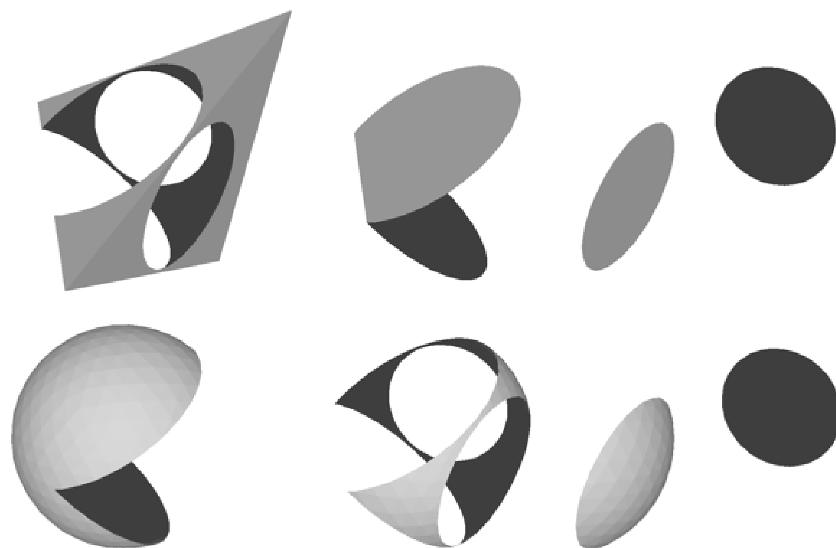
Az előző két lépés folyamán rendre új csúcsokat, éleket és háromszögeket hoztunk létre, amellyel kibővítjük az eredeti meshelemek sorát. A következő lépéseknél fontos lesz az eredeti meshek néhány tulajdonsága, ezért át kell vizsgálnunk minden újonnan létrehozott elemet, hogy azok teljesítik-e a követelményeket, és ha nem, ki kell javítsuk őket. Így tehát a következőket kell végrehajtsuk:

- Az újonnan keletkező csúcsok, élek és háromszögek eltárolása. Azon élek és háromszögek, melyek a felosztás után már nem elemei a meshnek, törlésre kerülnek.
- A negatív irányú háromszögeket megfordítjuk.
- A háromszögek és hurkok csúcsainak indexei megváltozhattak, így azokat is frissítjük.

Ezekkel a lépésekkel elérjük, hogy a fázis után ne tároljunk azonos csúcsokat és éleket, minden háromszög pozitív irányítású legyen, illetve ne szerepeljenek elfajuló háromszögek. Ezek az ellenőrzések végezhetők az első 3 lépés után, de azokkal egyszerre is. Az én megvalósításomban minden újraháromszögelésnél az újonnan keletkező háromszögeket az eredetiekhez adom úgy, hogy azok mindig teljesítsék a fenti elvárásokat, így a 3. lépés után már nem szükséges újraellenőrizni minden elemet.

5. lépés: Alfelületek (patchek) létrehozása

A 2. lépésben kapott metszethurkokat felhasználva alfelületeket hozunk létre. Ezek 1-1 eredeti mesh háromszögeiből állnak, összefüggőek és metszethurkok határolják.



10. ábra: Egy tetraéder és egy gömb mesheinek metszéséből kapott patchek

Egy patch háromszögeinek összegyűjtéséhez szélességi keresést használunk, ezt mutatja be a 2-es algoritmus. Minden metszethurok pontosan négy patchnek lesz határolója, 2-2 patch tartozik A és B meshhez. Ha már mind a négy patchet előállítottuk, akkor a hurok "feldolgozott" lesz. Egy pach létrehozásakor választunk egy hurkot, amely még nem feldolgozott, és az egyik éléhez tartozó olyan háromszöggel indítjuk a keresést, amely még nem szerepel patchben. Ennek a háromszögnek (hacsak nem önmagában egy hurok) van olyan éle, amely nem egy hurokba tartozó él, ennek a túloldalán lévő háromszöget bevesszük a patchbe. Az új háromszögnek is lehetnek ilyen élei, rajtuk keresztül folytatjuk a keresést. Így valójában egy BFS algoritmust futtatunk, melyben a háromszögek a csúcsok, két háromszög között pedig akkor vezet él, ha nem hurok-élen keresztül szomszédosak. Végezetül egy-egy patchhez a keresés során eltároljuk, hogy mely hurkok határolják.

Az algoritmus lefuttatása után megkapjuk a várt patcheket, amelyek mindegyike egy vagy több metszethurok által határolt, ezen tulajdonságuk szerint nevezhetők privát vagy közös patcheknek. Megjegyzendő, hogy egy meshhez mindig tartozik legalább egy privát patch és legfeljebb egy közös patch.

2. Algorithm Patchek létrehozása metszethurkokból

```
 1: Input:  Metszethurkok vektora (Hurkok) és egy háromszögelt felület (S)
 2: Output:  Patchek vektora (Patchek)
 3: while ∃ hurok L_i ∈ Hurkok: L_i.használt < 2 do
 4:     L_i.használt += 1
 5:     új üres patch létrehozása: újPatch
 6:     újPatch.határoló_hurkok.push(L_i)
 7:     V ← L_i egyik éléhez tartozó S-beli háromszög,
 8: amely még nem szerepelt patchben
 9:     Q ← sor.push(V)
10:     while Q.NemÜres() do
11:         T ← Q.FrontAndPop()
12:         újPatch.háromszögek.push(T)
13:         for e_j éle T-nek do
14:             if e_j nem metszethurok-él then
15:                 K ← e_j-hez tartozó másik háromszög
16:                 if K ∉ újPatch.háromszögek then
17:                     újPatch.háromszögek.push(K)
18:                     Q.push(K)
19:                 end if
20:             else if e_j-hez tartozó hurok L_k ∉ újPatch.határoló_hurkok then
21:                 újPatch.határoló_hurkok.push(L_k)
22:                 L_k.használt += 1
23:             end if
24:         end for
25:     end while
26:     Patchek.push(újPatch)
27: end while
```

6. lépés: Alblokkok kialakítása patchekből, majd kategorizálás

Ebben a lépésben a patchekből alblokkokat alakítunk ki, amelyek önmagukban egy-egy meshként kezelhetők. Minden patch pontosan 2 alblokkban szerepel. Ezen algoritmus alapja a következő: vegyünk egy patchet, amely még nem szerepel 2 blokkban, gyűjtsük össze az őt határoló metszethurkokat (a közös patch esetében lehet egynél több, privát patchek esetében egy). Találjunk ezekhez a hurkokhoz ”párokat”. Az első páratlan hurokhoz (L_1) keressünk olyan patchet a másik meshből, amelyet L_1 metszethurok határol. Ha ehhez a patchhez más határoló hurkok is tartoznak, vegyük fel azokat is a páratlanok közé. Folytassuk ezt addig, amíg minden huroknak párt nem találunk, közben ügyelve a következőkre:

- Egy blokkot ne hozzunk többször létre. Ez megoldható, ha eltároljuk minden patchhez azokat a másik meshbeli patcheket, amelyekkel már egy blokkot alkotnak.

- Csak olyan patchet vegyünk a blokkba, amelyhez tartozó határoló hurkok vagy páratlanok (és van legalább egy ilyen), vagy nem szerepelnek még a blokkban. Ha olyan metszethurok határolja a patchet, amelyet már párosítottunk a blokk előállítása során, akkor nem vehetjük a blokkhoz.

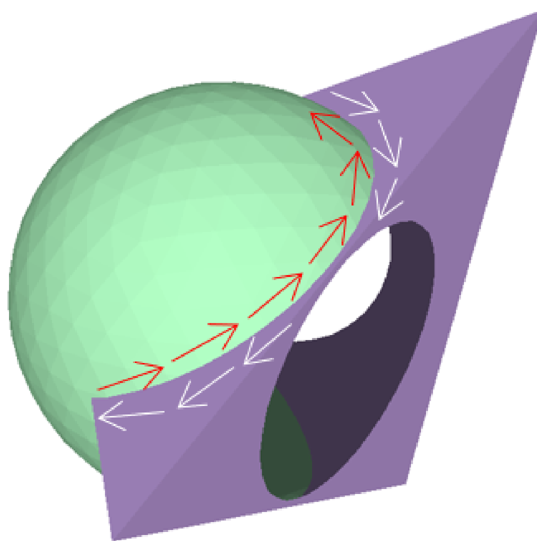
Minden új bevehető patch keresése során megfelelőnek nevezzük azokat a patcheket, amelyek ezeket a feltételeket teljesítik. Az algoritmus tehát a következő:

3. Algorithm Alblokkok létrehozása patchekből

1: *Input:* Patchek vektora (Patchek)
2: *Output:* Alblokkok vektora (Blokkok)
3: **while** $\exists$ patch $P_i \in$ Patchek: P_i.használt < 2 **do**
4: P_i.használt $+= 1$
5: új üres blokk létrehozása: újBlokk
6: újBlokk.patchek.push(P_i)
7: Páratlanok $\leftarrow$ sor
8: **for** $L_i \in P_i$.határoló_hurkok **do**
9: Páratlanok.push(pár(L_i, P_i.mesh))
10: **end for**
11: **while** Páratlanok.NemÜres() **do**
12: $(L_i, M) \leftarrow$ Páratlanok.FrontAndPop()
13: **if** $L_i \in$ újBlokk.párosított_hurkok **then**
14: **continue**
15: **end if**
16: **for** $P_j \in$ Patchek **do**
17: **if** megfelelo(P_j) and P_j.mesh $\neq M$ **then**
18: újBlokk.patchek.push(P_j)
19: P_j.használt $+= 1$
20: újBlokk.párosított_hurkok.add(L_i)
21: **for** $L_k \in P_j$.határoló_hurkok, $L_k \neq L_i$ **do**
22: **if** $L_k \in$ Páratlanok **then**
23: újBlokk.párosított_hurkok.add(L_k)
24: **else**
25: Páratlanok.push(pár(L_k, P_j.mesh))
26: **end if**
27: **end for**
28: **end if**
29: **end for**
30: **end while**
31: Blokkok.push(újBlokk)
32: **end while**

Kategorizálás

1. lépésben csak azt döntjük el, hogy egy adott blokk a két mesh valamelyik irányú különbségéhez tartozik-e vagy sem. Vegyünk egy a blokkhoz tartozó hurkot, és a blokkban szereplő két olyan patchet, amelyeket ez a hurok határol. Ha a két patchben a hurok csúcsai ugyanabban a sorrendben szerepelnek, akkor a blokk a különbségképzések eredményeihez tartozik, ellenkező esetben az unió vagy a metszet meshhez.



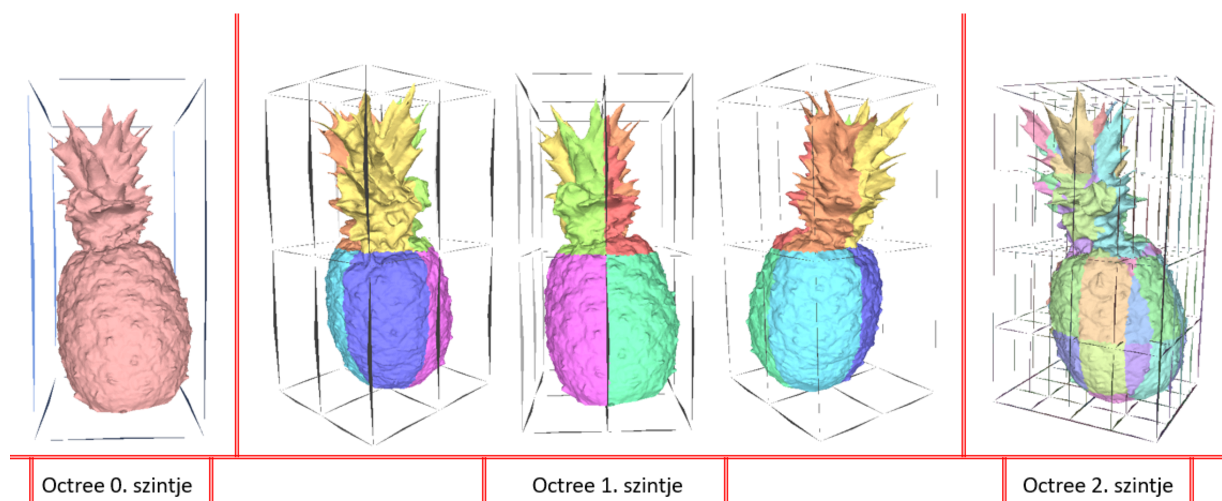
11. ábra: A gömbhöz és a tetraéderhez tartozó patchekben a hurkok ellentétes irányúak, így a hozzájuk tartozó blokk unió vagy metszet

2. lépésben elkülönítjük, hogy az unió/metszet kategóriába sorolt blokkok közül pontosan melyik tartozik melyik eredménybe. Nyilvánvalóan $(A \cap B) \subseteq (A \cup B)$, egyenlőség pedig csak $A = B$ esetben lehetséges, amelyet az algoritmus legelején elkülönítettünk. Tehát $(A \cap B) \subset (A \cup B)$ áll fenn. Vagyis a metszet összes csúcsának minimum koordinátái sosem kisebbek az unió összes csúcsának minimum koordinátáinál, a maximumok pedig sosem nagyobbak. Következésképpen a metszet/unió kategóriába sorolt blokkok összes csúcsának minimum és maximum koordinátái megegyeznek az unió minimum és maximum koordinátáival. Azon blokkok, amelyek minimum és maximum koordinátái nem egyeznek meg az előbb kiszámolt értékekkel, a metszet részét képezik.

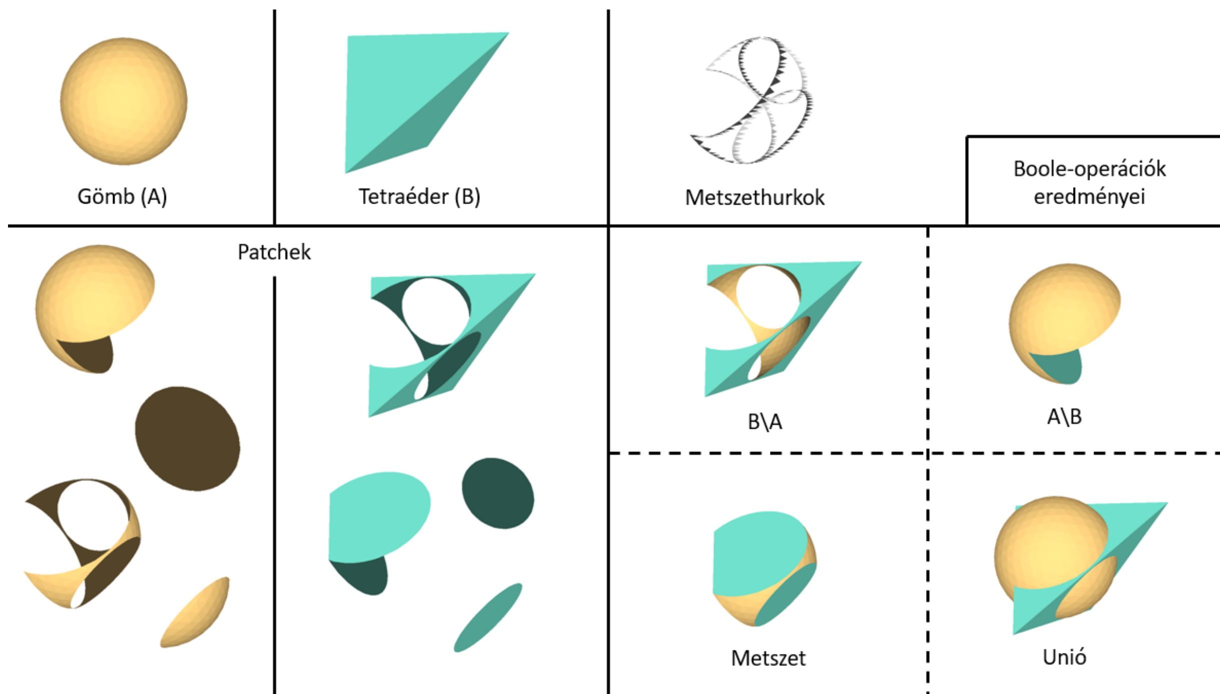
3. lépésben válasszuk külön a különbség kategóriába sorolt blokkokat aszerint, hogy azok A\B vagy B\A részét képezik-e. Ehhez a 2. lépésben meghatározott metszet és unió eredményeket használjuk. Minden patchet, ami a metszet eredményhez tartozik nevezzünk belsőnek, míg az unióhoz tartozókat külsőnek. Ekkor tekintsük egy különbség kategóriabeli blokk egy patchét. Ha ez külső patch, és A meshhez tartozik, akkor a blokk az A\B mesh része, ha viszont B meshből keletkezett, akkor B\A része. Hasonlóan, ha ez belső patch, és A meshhez tartozik, akkor a blokk a B\A mesh része, ha viszont B meshből keletkezett, akkor A\B része. Kategorizálás után a különbségek részét képező belső patchek minden háromszögét fordítsuk meg, hogy a keletkezett mesh háromszögeinek normálvektorai mind kifelé mutassanak.

6.3. Implementáció, példa objektumok

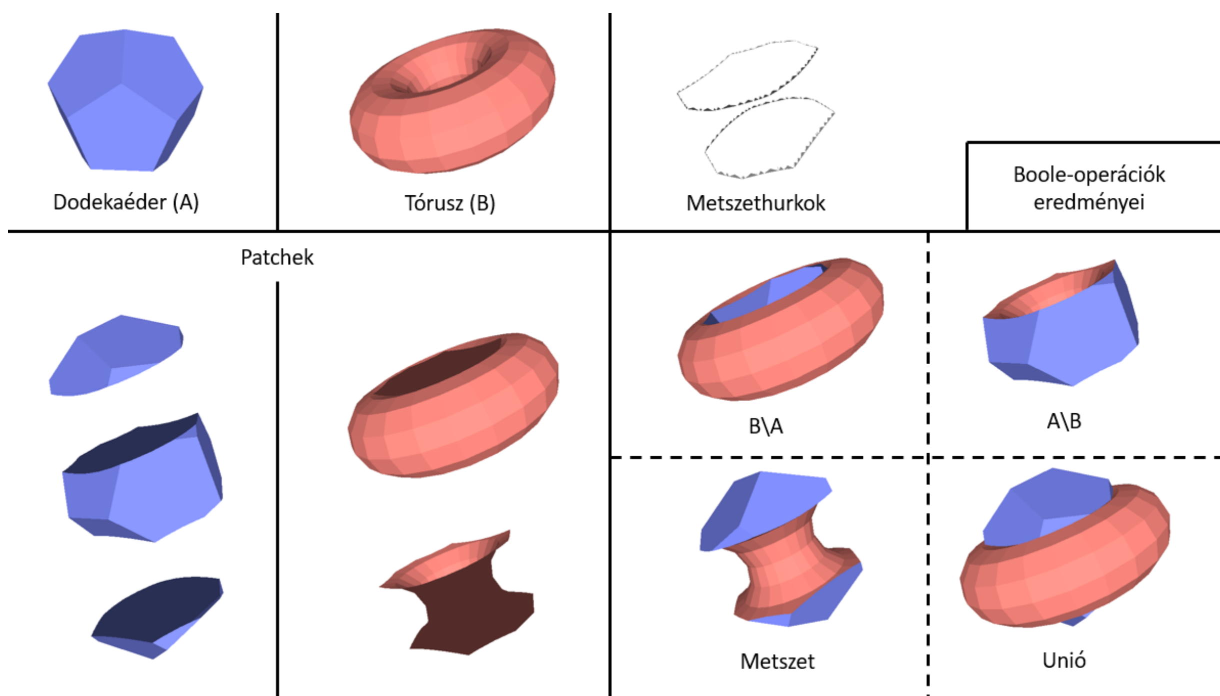
Ahogy a fejezet elején említettem, a Boole-operációkat végrehajtó program C++ nyelven implementálásra is került, a 10. és 11. ábra is a program eredményeként készült el. A megvalósítás alapvető céljául az eredeti algoritmus részletesebb megismerése, a felmerülő problémák kezelése és az általam megfogalmazott módosítások gyakorlatba ültetése szolgált. A program alapja egy általam létrehozott Stl nevű osztály, amely lehetővé teszi egy input mesh beolvasását, csúcsainak, éleinek, háromszögeinek eltárolását és egy objektum STL fájlformátumú mentését. Az algoritmus 4 további nagy feladatáért (octree felépítése, metszethurkok, patchek, majd alblokkok előállítása) külön függvények felelősek. Ezek eredményeit az alábbi ábrák szemléltetik, melyeket egy háromdimenziós mesheket feldolgozó szoftver (MeshLab) segítségével jelenítettem meg.



12. ábra: Az octree szintjei egy példa objektumon



13. ábra: Gömb és tetraéder Boole-operációi



14. ábra: Dodekaéder és tórusz Boole-operációi

7. Szeletelés

Szeletelésnek (slicing) nevezzük azt a folyamatot, amely során egy objektum és egy vágósík sorozatos metszeteit számítjuk ki, miközben a sík egy útvonal mentén végighalad. Számos felhasználási módja közül a két leggyakoribb a 3D nyomtatás és az orvosi képalkotás tematikájához kapcsolódik. Ebben a fejezetben a szeletelés egy rendkívül hatékony, az octree adatstruktúrán alapuló módszerével foglalkozom, amelyet Comino Trinidad et al. 2022-ben megjelent cikkükben [8] mutattak be, a továbbikban az ő munkájukat veszem alapul.

A szeletelési algoritmusok két fő kategóriába oszthatók:

- Kétdimenziós szeletelés. Minden szelet egy síkbeli metszet az objektum és a vágósík között.
- Térfogati szeletelés. Az algoritmus a vágósík által generált, 1 voxel vastagságú térbeli szeleteket számítja ki.

A két megközelítés eredménye alapvetően különbözhet egymástól, látványos példa erre egy objektum, amelynek a vágósíkkal párhuzamos lapja van, ezt kétdimenziós szeleteléssel nehézkes ábrázolni. Számos olyan alkalmazás is létezik, amelyben elengedhetetlen a szeletek háromdimenziós kiterjedése a valódi szeletek vastagságának precíz modellezéséhez, ilyen például a 3D nyomtatás. Utóbbi folyamán sokszor a kinyomtatott objektum felületét alkotó voxelek egyéb tulajdonságokkal is bírhatnak: beállíthatjuk például a színüket vagy anyagukat. Ehhez feltétlenül szükséges minden határoló felülethez tartozó voxel meghatározása (ezeket nevezzük szürke voxeleknek), ez azonban kétdimenziós szeletelés során sokszor nem pontos (ld. vágósíkkal párhuzamos lapok). Többek között az itt megemlített okok miatt foglalkozunk ebben a fejezetben a térfogati szeletelés módszerével.

7.1. Octree alkalmazása szeleteléshez

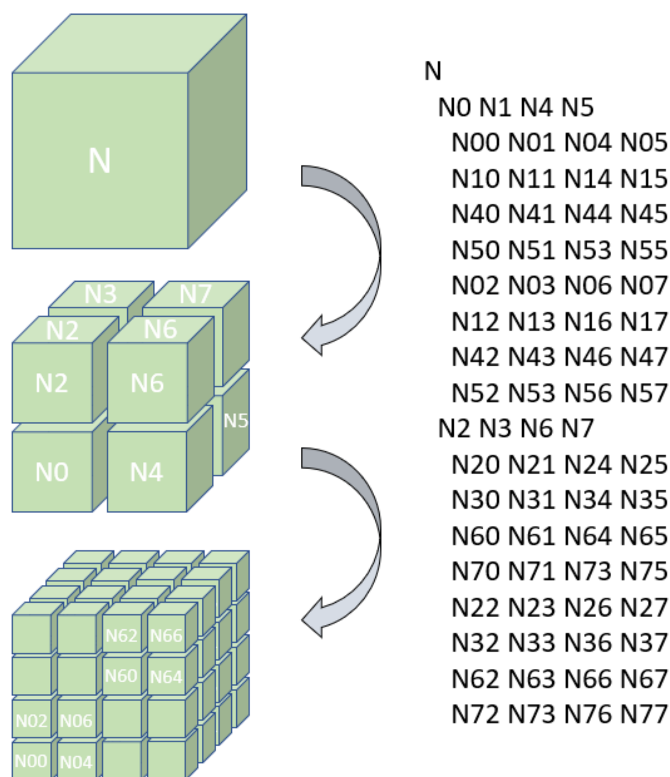
7.1.1. Memóriaigény

Minden octree maximális mélysége meghatározza a maximális precizitást, amivel az adatstruktúra az adott objektumot reprezentálni képes. Az alkalmazások során napról napra nő az elvárt pontosság, amely hatalmas méretű octreeket eredményez, így szükségessé válik az adatstruktúra külső memóriában való tárolása és megfelelő időben részleteinek a gyorsítótárba való importálása. Azonban a külső memóriában való tárolás alapvetően lassabb, így fontos feladat a benne való tárolás optimalizálása is. Tekintsük a 3D nyomtatás példáját: egy mai forgalomban kapható átlagos 3D nyomtató 12x16x16 inches területen dolgozik, a vízszintes síkon 1200vpi (voxel per inch), függőlegesen 300 szelet/inch felbontással. Folyamatos gyártás esetén mind a 4800 szelet nagyjából $2,8*10^8$ voxelt tartalmaz, amelyek mindegyikének tulajdonságait meg kell határozni a nyomtató haladásával párhuzamosan. Így eszköztől és nyomtatási módtól függően kevesebb, mint 7 másodperc áll rendelkezésre egy szelet legenerálásához. Ebben az időben természetesen más, nagy számítási idejű algoritmusokat is le kell futtatnunk a jó minőségű nyomtatás eléréséhez, így a szerzők

által használt nyomtató esetében 3 másodpercnél is kevesebb idő jutott az objektum szeletelésére.

7.1.2. Octree elkódolása söprés mentén

A szeletelés optimalizálásához Comino Trinidad et al. [8] által javasolt octree-kódolási forma a Sweep traversal encoding, amelyet söprés menti kódolásnak nevezünk. Ez egy lineáris kódolási forma, ahogyan arról a 2.4 fejezetben részletesen írtam. Lényege az octree csúcsok egy speciális sorrendje: legyen adott egy **d** vektor, és egy rá merőleges P_d sík. A csúcsokat olyan sorrendben soroljuk fel, amilyen sorrendben a P_d sík metszi őket, ahogyan a **d** iránynak megfelelően halad. Az általánosság megszorítása nélkül feltehetjük, hogy a sík az univerzum valamelyik koordinátatengelyére merőleges, legyen tehát **d** a z tengely pozitív irányával párhuzamos. Legyen ekkor N_a és N_b két octree csúcs, jelölje N_a^z az N_a csúcs minimális z koordinátáját. Ekkor a lineáris tárolás során N_a csúcs megelőzi N_b-t, ha:



15. ábra: Csúcsok sorrendje söprés menti elkódolás esetén (Morton-kóddal számozva)

- $N_a^z < N_b^z$, vagy
- $N_a^z = N_b^z$ és N_a mélysége kisebb, mint N_b mélysége, vagy
- $N_a^z = N_b^z$ és N_a mélysége megegyezik N_b mélységével, de N_a Morton-kódja kisebb N_b Morton-kódjánál.

A csúcsok sorrendje a söprés menti kódolásnál megegyezik azzal a sorrenddel, amit akkor kapnánk, ha a fát z tengely irányban mélységi, x és y tengely irányban szélességi kereséssel járnánk be. A fenti, Comino Trinidad et al. [8] cikkében szereplő illusztráción alapuló 15. ábra egy 2 mélységű teljes octree söprés menti kódolását mutatja be.

A helyzeti kódolás számos alkalmazásban hasznos, azonban nagyobb tárhely igényű, mint a lineáris kódolás BFS, DFS vagy söprés menti változata. Ezen három módszer hatékonysága már a konkrét feladattól függ, azonban később látni fogjuk, hogy a söprés menti kódolás 3D modellek szeletelése esetén felülmúlja mindkét másik bejárást, mivel a külső memóriáról minden csúcsot csak egyszer kell beolvasnunk.

Ahogy korábban említettük, folyamatos gyártás esetén a legfontosabb korlátozás az egy szelet előállítására rendelkezésre álló idő. 3D nyomtatás esetén minden szelet t idő alatt kerül kinyomtatásra, de ennek folyamatosnak kell lennie, nem késhetünk vele és nem is szakíthatjuk meg azt. Egy esetleges késés deformációt okozhat a már kinyomtatott részben, ami súlyos hibákhoz vezethet, így nem megengedett. Tehát minden szeletnek rendelkezésre kell állnia abban a pillanatban, amikor az előző nyomtatása befejeződött, vagyis egy szelet generálása egy t_{max} paramétert (mely a pontos eszköztől és technikától függ) nem haladhat meg. Egy átlagos nyomtató nyomtatási képe rasztergrafikus formában tárolva, ha pixelenként csak 1 bit méretű is, nagyjából fél terabyte helyet foglal, de különböző tömörítési eljárásokat alkalmazva is sok gigabyte méretű lenne. Ugyanez a voxelizált elem ábrázolható egy nagyjából 15 szintű octreevel is, minden csúcsához 2 bitben meghatározva, hogy az fekete/szürke/fehér. Az itt leírt octreeben minden csúcshoz 16 biten a 8 gyerekének tulajdonságát tároljuk el. A valós alkalmazások folyamán gyakran az octree által reprezentált térfogat jelentős része homogén (tömör vagy üres), ezeken a területeken a csúcsokat nem osztjuk tovább, így összességében jelentősen csökkenthetjük a felhasznált memóriát. Itt is használhatunk tömörítési eljárásokat, de ekkor a tömörítetlen adat visszanyerése is t_{max} időn belül kell történjen.

Miért optimális a söprés menti kódolás szeleteléshez?

Egy külső memórián tárolt objektum szeletelése során 1 voxel vastagságú metszetek halmazát gyűjtjük össze, ahogyan a vágósík az adott $\mathbf{d}$ irányban végighalad. Továbbra is feltesszük, hogy $\mathbf{d}$ megegyezik a z tengely pozitív irányával. Jelöljön S_z egy z értékhez tartozó vágósíkot, N_{minZ} és N_{maxZ} pedig egy N csúcs minimum és maximum z koordinátáit. Ebben az esetben a söprés menti elkódolást három fő tulajdonsága teszi hatékonnyá:

- A külső memórián tárolt adatokat az ottani sorrendjük szerint, minden csúcsot egyetlen egyszer kiolvasva használjuk.
- Minden S_z által meghatározott szelet kiszámításakor csak az $N_{minZ} \leq z \leq N_{maxZ}$ feltételt teljesítő csúcsokat tároljuk a központi memóriában, ezeket nevezzük a z-hez tartozó aktív csúcsoknak. Ahogy z értéke növekszik, újabb - eddig a külső tárhelyen tárolt - csúcsok válnak aktívvá, míg azok, amelyek már nem teljesítik a $N_{minZ} \leq z \leq N_{maxZ}$ egyenlőtlenséget, törlésre kerülnek a fő memóriából (és soha nem lesznek újra aktívak).
- Az aktív csúcsok által felhasznált tárhely a fő memóriában az objektum adott szelethez tartozó kerületével arányos, amely következménye a 2.3 fejezetben bemutatott komplexitási tételnek quadtreek esetében.

7.1.3. Octree felépítése

Egy objektum bemeneti formátuma tipikusan háromszögelt mesh vagy síkmetszetek halmaza. A nyomtatáshoz és későbbi felhasználásokhoz az objektumhoz tartozó octree felépítése szükséges, ami a modelltől függően top-down vagy bottom-up módszerrel történhet.

1. **Top-down felépítés**
   Egy octree szokásos felépítési módja, háromszögelt mesheknél alkalmazzuk. A gyökértől kezdve rekurzívan teszteljük a csúcsokat, hogy a mesh mely háromszögei metszik azt. Minden csúcsot (a gyökeret kivéve) a szülőjét metsző háromszögekkel szemben tesztelünk, ehhez egy gyors, Akenine-Möller által leírt módszert használhatunk [1]. Amennyiben van a csúcsot metsző háromszög, szürkének jelöljük, és 8 részre osztjuk. Metsző háromszög hiányában fekete vagy fehér jelölést kap, annak megfelelően, hogy az objektum belsejét képezi, vagy azon kívül helyezkedik el. A fekete és fehér csúcsok különválasztásához elegendő egy a csúcs közepéből indított tetszőleges irányú sugár vizsgálata. Ha a sugárnak páratlan számú metszéspontja van a meshsel, akkor a fekete, ellenkező esetben a fehér osztályba kerül.
   A csúcsok a generálásukkal egy időben kerülnek eltárolásra. Ahhoz, hogy a söprés menti sorrendben tároljuk őket, egy vermet használunk. Egy csúcs feldolgozása során teszteljük, hogy a gyerekei mely háromszögeket metszik. Azok a gyerekek, amelyek legalább egy háromszöget metszenek, 2 sorba kerülnek: az egyikbe a nagyobb z koordinátával rendelkező gyerekek, a másikba a kisebbel. Ezután mindkét sort (az előbbi sorrendben) a verembe rakjuk. A következő lépésben a verem tetején levő csúcsokat dolgozzuk fel, így végül a kívánt sorrendben vizsgáljuk meg és tároljuk el az octree csúcsait.

2. **Bottom-up felépítés**
   Amennyiben az input síkmetszetek halmazaként adott, az octree top-down felépítése túlságosan költséges, mivel az alacsony szintű csúcsoknál nagy mennyiségű poligonnal kell dolgoznunk. Helyette az egyes szeletekhez felépítünk egy megfelelő quadtreet, majd a quadtreek összevonásával konstruáljuk az octreet. Minden szelethez egy ugyanolyan mélységű quadtreet generálunk, mint a kívánt octree, függetlenül a poligonoktól. Ezután minden quadtree négy testvér leveléhez található (a megelőző vagy a következő szeletnek megfelelő) másik négy levél, amelyek együtt egy octree 8 testvér levelét adják. Az így keletkezett részoctreek közül nyolc a következő lépésben eggyel nagyobb mélységű részoctreekké vonható össze. Ezt a lépést rekurzívan ismételjük addig, amíg megkapjuk a gyökér csúcsot.
   Legyen E az input összes élének száma, d a quadtreek és octreek mélysége, N az octree leveleinek száma. Ekkor $d = \mathcal{O}(\log N)$. A quadtreek létrehozása során minden élet legfeljebb annyiszor vizsgálunk meg, amennyi a quadtree mélysége, vagyis a quadtreek generálása $\mathcal{O}(E * \log N)$ időt vesz igénybe. A quadtreek összevonásakor minden levelüket egyszer megvizsgálva létrehozzuk az octree leveleit, majd ezeket fokozatosan egybeolvasztva generáljuk az octree összes csúcsát. Tehát az algoritmus második fázisa az octree összes csúcsának számával arányos, ami $K = \sum_{i=0}^{d} \frac{N}{8^i} < \frac{8}{7}N$, így az algoritmus teljes futásideje $\mathcal{O}(N + E * \log N)$.

A szerzők által javasolt octree felépítésével és kódolásával számos előnnyel szolgál a szeletés folyamán.

- Az objektum voxelizált változata könnyen tárolható a külső memóriában.
- A külső memóriából szeletenként olvashatók ki a voxelek, amely lehetővé teszi egy adott szelet időkorláton belüli legenerálását.
- Minden szürke (határoló) voxelt hatékonyan meghatároz és eltárol. A voxelek az objektumhoz tartozásuk szerint (belső/külső/határoló) feketék, fehérek vagy szürkék, utóbbiak egyéb, az objektum felületére vonatkozó információt is hordozhatnak (szín, anyag).

7.1.4. Octree alapú szeletelés

Az eddig bemutatott söprés menti elkódolás lehetővé teszi, hogy az egymás után követező szeleteket gyorsan kiszámítsuk, miközben az algoritmus egy (a központi memóriában kis helyet elfoglaló) állapotát folyamatosan fenntartjuk. Lentebb látható a módszer pszeudokóddal történő leírása.
A módszer lépései tehát a következők:

1. Legyen **StQ** egy verem, amely octree csúcsok sorait tartalmazza, és minden ilyen **Q** sor csak egyazon szintű csúcsokból áll. Továbbá jelölje **AN** azon octree csúcsok egy listáját, melyeket metsz egy adott z koordináta.
2. Az algoritmus inicializálásakor **StQ** egyetlen sort tartalmaz, amelynek egyetlen eleme a gyökér csúcs.
3. Ezután minden lépésben rekurzívan feldolgozzuk a verem legfelső sorának elemeit, egészen addig, amíg **StQ** ki nem ürül.
4. Egy **Q** sor feldolgozásakor az összes benne szereplő **N** csúcshoz tartozó, a gyerekeik tulajdonságait leíró 16 bitet elolvassuk. Ha **N** csúcsot metszi z, akkor ugyanez pontosan **N** 4 gyerekére igaz, ezeket hozzáadjuk **AN**-hez.
5. Ezen felül a 4 kisebb z koordinátával rendelkező gyerek közül a szürkéket hozzáadjuk a **kisebbZ** sorhoz, míg a nagyobb z koordinátájú szürke gyerekek a **nagyobbZ** sorba kerülnek.
6. A **Q** sorban található összes **N** csúcs feldolgozása után **StQ**-hoz adjuk először a **nagyobbZ**, majd a **kisebbZ** sort. Így az algoritmus mindig először az alacsonyabb z koordinátájú csúcsokat dolgozza majd fel.
7. Amint az első olyan sor feldolgozásra kerül, amely leveleket tartalmaz, **AN** az összes z-t metsző csúcsot tartalmazni fogja. Innen a csúcsokat a quadtree síkra vetítve könnyen megkapjuk a kívánt szelet quadtreejét.

8. A következő, $z+1$ koordinátához tartozó szelet generálásához az **AN**-ben található csúcsokat átvizsgáljuk, amelyet $z+1$ már nem metsz, azt töröljük. Ezután folytatjuk az algoritmust a 4. ponttól. A verem legfelső sorától kezdve addig dolgozzuk fel a sorokat, amíg egy olyat nem találunk, amely leveleket tartalmaz. Ekkor a $z+1$ koordinátához tartozó szelet az időközben frissített **AN** lista csúcsainak quadtree síkra vetített képe.

4. Algorithm Söprés mentén elkódolt octree szeletelése

```
 1: Eljárás szeletel(gyökér, z)
 2: if NemInicializált() then
 3:     Q ← sor.push(gyökér)
 4:     StQ ← verem<sor>.push(Q)
 5:     AN ← [ ]
 6: else
 7:     AN ← [N_i|N_i ∈ AN, metsz(N_i, z)]
 8: end if
 9: while StQ.NemÜres() do
10:     nagyobbZ ← sor
11:     kisebbZ ← sor
12:     Q ← StQ.TopAndPop()
13:     while Q.NemÜres() do
14:         N ← Q.FrontAndPop()
15:         if Metsz(N,z) then
16:             C ← MetszettGyerekek(N,z)
17:             AN.add(C)
18:         end if
19:         nagyobbZ.push(N.nagyobbZSzürkeGyerekek())
20:         kisebbZ.push(N.kisebbZSzürkeGyerekek())
21:     end while
22:     if nagyobbZ.NemÜres() then
23:         StQ.push(nagyobbZ)
24:     end if
25:     if kisebbZ.NemÜres() then
26:         StQ.push(kisebbZ)
27:     end if
28: end while
```

A szerzők számos különböző komplexitású objektumon tesztelték a söprés menti elkódoláson alapuló szeletelési eljárást. Azonban az algoritmus futásideje sokkal érzékenyebb az alkalmazni kívánt felbontásra (az octree mélysége), mint a háromszögelt felület csúcsainak és lapjainak számára. Így a tesztek során 10, 13 és 15 mélységű octreek használatát vizsgálták. Az előzetes elvárásnak megfelelően az eredményül kapott futásidők a mélységhez képest

exponenciálisan nőnek, így 15 szintű octree esetén a fa felépítése már gyakran a 10 percet is meghaladta. Mindezek mellett a szerzők által javasolt szeletelési eljárás számos szempontból felülmúlja az irodalomban eddig megjelent algoritmusokat a teszt elemeken, különösen nagy méretű, komplex objektumok esetében.

8. Összefoglalás

Dolgozatomban először bemutattam az octree adatstruktúra felépítését, elkódolási lehetőségeit és jellemzőit. Ezután öt fejezetben, öt eltérő területen történő alkalmazásával világítottam rá az octree különböző tulajdonságaira, felhasználási módjaira. Megmutattam, hogy egyszerűsége tökéletesen alkalmassá teszi széles körű használatra, különösen a számítógépes grafika világában.

Habár célom az adatstruktúra többrétű bemutatása és a benne rejlő lehetőségek összefoglalása volt, az itt leírtak az octreevel kapcsolatos problémák, megközelítések vagy alkalmazások csupán egy szeletét fedik le. Az octree felhasználási módjainak tárháza szinte végtelen, a dolgozat kiegészítéseként érdemes lenne megvizsgálni például a pontfelhőből történő felszínrekonstruálás, a végeselem módszer vagy a legközelebbi szomszédkeresés feladatban betöltött szerepét, velük való kapcsolatát.

Az általam implementálásra került mesheket kezelő program is számtalan további lehetőséget rejt magában. Ezek nem csak a memóriafelhasználás és futási sebesség terén történő fejlesztéseket foglalják magukban, a program a meshek kezelésének több algoritmusával is kiegészíthető lenne: például az objektumok szeleteléséhez egy a szakdolgozatban precízen leírt algoritmus kapcsolódik, melynek alapjaihoz a programban már minden lépés implementálásra került. További példa a meshek átméretezésének problémája, ennek megvalósítása során már újabb kihívások, nehézségek merülhetnének fel.

Végezetül kiemelném, hogy a 3D nyomtatás vagy a videojátékok példájához hasonlóan még számtalan olyan technikai újítás várható, amelyben a tér rekurzív felosztása központi szerepet játszik. Ilyen problémák során pedig az octree adatstruktúra egy biztos és részleteiben ismert alapot nyújt, amelyre megéri építeni.

Hivatkozások

[1] Tomas Akenine-Möller. Fast 3d triangle-box overlap testing. *SIGGRAPH Comput. Graph.*, 2005.

[2] Walid Aref and Ihab Ilyas. A framework for supporting the class of space partitioning trees. 2008.

[3] Boris Aronov, Hervé Brönnimann, Allen Y. Chang, and Yi-Jen Chiang. Cost prediction for ray shooting in octrees. *Computational Geometry*, 34(3):159–181, 2006.

[4] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[5] Jon Louis Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4):214–229, 1980.

[6] Dan S. Bloomberg and Leptonica. Color quantization using octrees. 2003.

[7] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, 1976.

[8] M. Comino Trinidad, A. Vinacua, A. Carruesco, A. Chica, and P. Brunet. Sweep encoding: Serializing space subdivision schemes for optimal slicing. *Computer-Aided Design*, 146, 2022.

[9] Robert Endl and Manfred Sommer. Classification of Ray-Generators in Uniform Subdivisions and Octrees for Ray Tracing. *Computer Graphics Forum*, 1994.

[10] Christer Ericson. Real-time collision detection. The Morgan Kaufmann Series in Interactive 3D Technology, pages 1–6. Morgan Kaufmann, San Francisco, 2005.

[11] Wm. Randolph Franklin. A linear time exact hidden surface algorithm. *SIGGRAPH Comput. Graph.*, 14(3):117–123, 1980.

[12] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986.

[13] M. Gervautz and W. Purgathofer. A simple method for color quantization: Octree quantization. pages 219–231, 1988.

[14] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, 1987.

[15] Paul Heckbert. Color image quantization for frame buffer display. *SIGGRAPH Comput. Graph.*, page 297–307, 1982.

[16] Martin Held. Fist: Fast industrial-strength triangulation of polygons. *Algorithmica*, 30:563–596, 2001.

[17] Gregory M. Hunter and Kenneth Steiglitz. Operations on images using quad trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):145–153, 1979.

[18] Perttu Hämäläinen, Tommi Ilmonen, Johanna Höysniemi, Mikko Lindholm, and Ari Nykänen. Martial arts in artificial reality. pages 781–790, 2005.

[19] Jaber J. Hasbestan and Inanc Senocak. Binarized octree generation for cartesian adaptive mesh refinement around immersed geometries. *Journal of Computational Physics*, 368, 2017.

[20] Xiaotong Jiang, Qingjin Peng, Xiaosheng Cheng, Ning Dai, Cheng Cheng, and Dawei Li. Efficient booleans algorithms for triangulated meshes of geometric modeling. *Computer-Aided Design and Applications*, 13:1–12, 2016.

[21] Donald E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, third edition, 1997.

[22] Chen Ling, Tian Mei-Hong, Chen Gen-cai, and Chen Chun. Using stereo camera system to realize realistic video avatar in virtual environment. *6th International Conference on Signal Processing, 2002.*, 1:707–710, 2002.

[23] J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6:153–166, 1990.

[24] Donald Meagher. Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-D objects by computer. 1980.

[25] Donald Meagher. Geometric modeling using octree-encoding. *Computer Graphics and Image Processing*, 19:129–147, 1982.

[26] Gang Mei and John C. Tipper. Simple and robust boolean operations for triangulated surfaces. *ArXiv*, 2013.

[27] Tomas Möller. A fast triangle-triangle intersection test. *Journal of Graphic Tools*, 2, 2004.

[28] Ricardo Nakamura and Romero Tori. A technique for collision detection and real-time video avatar interaction in mixed reality environments. 2007.

[29] Ricardo Nakamura and Romero Tori. Improving collision detection for real-time video avatar interaction. 2012.

[30] Aristides G. Requicha. Representations for rigid solids: Theory, methods, and systems. *ACM Comput. Surv.*, 12(4):437–464, 1980.

[31] H. Samet and R.E. Webber. Hierarchical data structures and algorithms for computer graphics. I. Fundamentals. *IEEE Computer Graphics and Applications*, 8(3):48–68, 1988.

[32] H.C. van de Hulst. *Light Scattering by Small Particles*. Dover Books on Physics. Dover Publications, 1981.

[33] Kyu-Young Whang, Ju-Won Song, Ji-Woong Chang, Ji-Yun Kim, Chong-Mok Park, and Il-Yeol Song. Octree-r: An adaptive octree for efficient ray tracing. *Visualization and Computer Graphics, IEEE Transactions on*, 1:343 – 349, 1996.

[34] Wikibooks. Rgb color solid cube.png, 2020. [Online; accessed 10-May-2023].