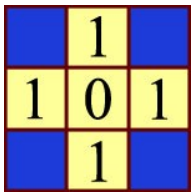
SZÉCHENYI ISTVÁN EGYETEM
MŰSZAKI TUDOMÁNYI KAR

INFORMATIKA TANSZÉK



BSC FOKOZATÚ INFORMATIKUS MÉRNÖK SZAK

DIPLOMAMUNKA

– kivonat –

Nagyméretű tetraéderhálózatok hatékony kezelési módszereinek vizsgálata

Martin József
mérnök informatikus

Győr 2011.

MARTIN JÓZSEF

NAGYMÉRETŰ TETRAÉDERHÁLÓZATOK HATÉKONY KEZELÉSI MÓDSZEREINEK VIZSGÁLATA

ÖSSZEFOGLALÁS

Fizikai szimulációkban gyakori megoldás, hogy egy testet tetraéderek hálózatára bontunk, és ezeket homogénnek tekintjük (pl. nyomás, sűrűség tekintetében). Ezeken a cellákon hajtunk végre olyan szimulációs számításokat, ahol minden cella csak a szomszédaitól függ.

Az, hogy hogyan ábrázoljuk a tetraéderhálózatot számítógépen, egyáltalán nem triviális. Nyilván kell tartanunk a tetraéder négy csúcsát, néhány statikus adatát (pl. térfogat) és a szimuláció során változó adatokat. Mindezek tárolására a következő lehetőségek kínálkoznak:

- a tetraéder összes adata egy struktúrát alkot, és ezeket láncolt listába rendezzük
- a tetraéder összes adata egy struktúrát alkot, és ezeket egy tömbben tároljuk
- minden adatot külön tömbben tárolunk; az egy tetraéderhez tartozó adatok azonos index alatt érhetőek el

Az elvégzett tesztek azt mutatták, hogy a leghatékonyabb a harmadik megoldás, feltehetően azért, mert ügyesen kihasználja a processzor belső cache-ét. Itt (és a másik tömbös megoldásnál is) gondot okoz, ha a hálózat maga is változik, azaz tetraédereket törlünk belőle és adunk hozzá. A törölt helyek nyilvántartásának egy kombinált módszerével azonban itt is elérhető a láncolt listák törlési-beszúrási sebessége.

A dolgozat bemutat néhány olyan megoldást is, amik egy-egy részterületet gyorsíthatnak:

- az egy csúcshoz tartozó tetraéderek tárolása tárolása láncolt listák tömbjében
- az oldalszomszédos tetraéderek megkeresése rendezés segítségével
- több dimenziós keresőfák alkalmazása a legközelebbi pont megkeresésére
- egy pont bennfoglaló tetraéderének megtalálása a legközelebbi pont segítségével

A dolgozathoz tartozó kód csak tesztcélokat szolgál, de könnyen kialakítható lenne belőle egy olyan függvénykönyvtár, amely nagyméretű tetraéderhálózatokon végzett szimulációk bázisául szolgálhatna.

JÓZSEF MARTIN

EFFECTIVE MANAGEMENT OF LARGE TETRAHEDRON NETWORKS

SUMMARY

In case of simulating physical laws inside of 3 dimensional objects, we often use large networks of small tetrahedrons to model our object. We consider all the tetrahedrons as homogeneous in self (by the temperature, pressure etc.) With this cells we can make the simulation calculations easily, supposed, that each cell depends only on its neighbours.

But how to represent this network in a software? The coordinates of the 4 vertices is enough to define a tetrahedron, but we need store some static data (e.g. volume) and also the dynamical properties. We can use one of the following solutions:

- we push all this data in a structure; and make a linked list of this structures
- we push all data in a structure and store this structures in an array
- we define one array per property; and all properties of a tetrahedron belong to the same index

Our test shows that the most effective method is the third one. Presumably because the CPU internal cache can clever pre-read the data in case of an iteration trough the network. But the array solutions are really ineffective when we need modify the network self by adding and deleting tetrahedrons. We show a combination of two methods for managing the free array elements; with this we can achieve the speed of linked lists by delete and insert elements.

We also show other processes to speed up some subcomponents of the software:

- using an array of linked list to store tetrahedrons sharing common vertex
- searching all the neighbourhoods in the network using quick sort
- using k-dimensional trees for searching the nearest neighbour of a vertex
- find out which tetrahedron includes a given point – speeding up this method with nearest neighbour search

The included software was written for test purposes only. But it could easily be developed into a library which can be used as base of simulation software working with large tetrahedron networks.

TARTALOMJEGYZÉK

1. BEVEZETÉS

Fizikai szimulációkban gyakori megoldás az, hogy egy testet tetraéder alakú cellákra bontunk, és ezen cellákat homogénnek tekintjük, azaz feltételezzük, hogy például a hőmérséklet, a nyomás, a sűrűség stb. egy ilyen cellán belül mindenhol ugyanakkora. Az egymással szomszédos cellák hatással vannak egymásra, kiszámítható például, hogy időegység alatt mennyi hő áramlik egy cellából a szomszédos cellába. Ha ezeket a számításokat egy alapállapotból kiindulva az összes cellára elvégezzük, megkapjuk a test következő időegységbeli állapotát. Egy kellő finomsággal felbontott testmodell 10^5 - 10^7 cellát is tartalmazhat.

A dolgozatban arra keresünk választ, hogy hogyan lehet hatékonyan kezelni ilyen hálózatokat, milyen adatstruktúrákban ábrázolhatjuk őket és milyen algoritmusokkal célszerű dolgozni ezeken a struktúrákon. Megvizsgálunk több lehetséges adatszerkezetet is, végrehajtunk mindegyiken egy tesztsorozatot, és összehasonlítjuk erőforrásigényeiket (idő, memória). Bemutatunk néhány olyan megoldást is, melyek tovább gyorsíthatják a tetraéderhálózatokon végzett műveleteket. A cél az, hogy – esetleg alkalmazási iránytól függően – javaslatot tudjunk tenni adatstruktúrák és algoritmusok egy olyan csoportjára, melyekkel a nagyméretű tetraéderhálózatok gyorsan és könnyen kezelhetőek.

A fejlesztett szoftver csak tesztelési célokra készült, ezért jelenlegi formájában nem alkalmas végfelhasználói használatra, de könnyen kialakítható lenne belőle egy olyan függvénykönyvtár, amely nagyméretű tetraéderhálózatokon végzett szimulációk bázisául szolgálhatna.

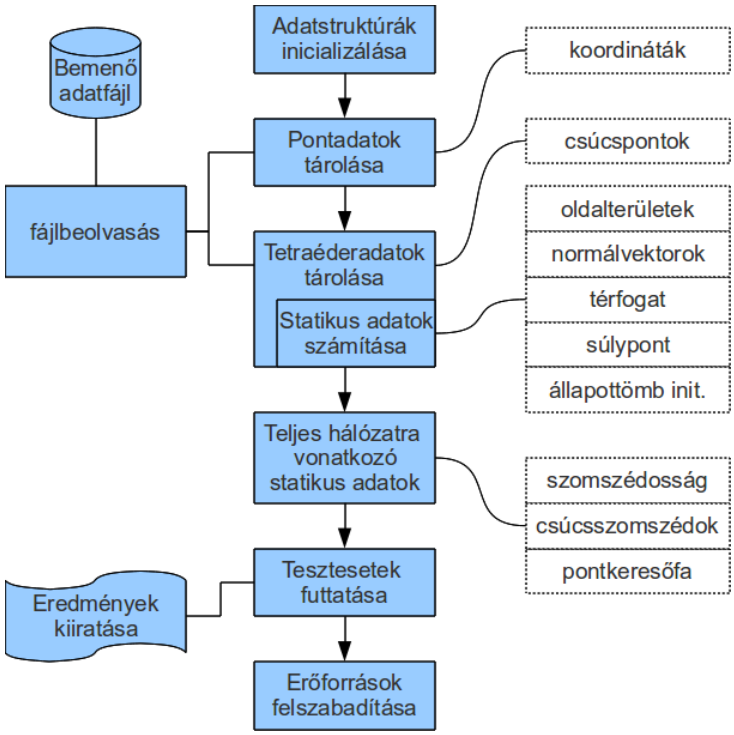
Az elvárások a szoftverrel szemben a következők voltak:

- tudjon beolvasni nagy méretű tetraéderhálózatokat leíró szöveges fájlokat
- a kezelt tetraéderek számának nagyságrendje 10^5 - 10^6 legyen
- tárolja el a rácspontok koordinátáit illetve a tetraéderek adatait (csúcspontok és előre kiszámolt statikus tulajdonságok, pl. térfogat, súlypont) a memóriába
- lehessen gyorsan lekérdezni a következő információkat: egy tetraéder csúcsai, térfogata, súlypontja, oldalainak felszíne és normál vektora, egy tetraéder melyik tetraéderekkel szomszédos, egy rácsponthoz melyik tetraéderek tartoznak, egy koordinátáival megadott pont melyik tetraéderben található
- a hálózat legyen dinamikus, tudjunk törölni és hozzáadni tetraédereket
- a szoftver futtasson komplex teszteseteket a hálózaton, mérje az ezekhez szükséges erőforrásokat (idő, memória)

2. A PROGRAM MŰKÖDÉSE

2.1. Általános ismertetés

A program alapvető működését az alábbi folyamatábra szemlélteti.



1. ábra: A program alapvető működése

2.2. Pontok és tetraéderek tárolása

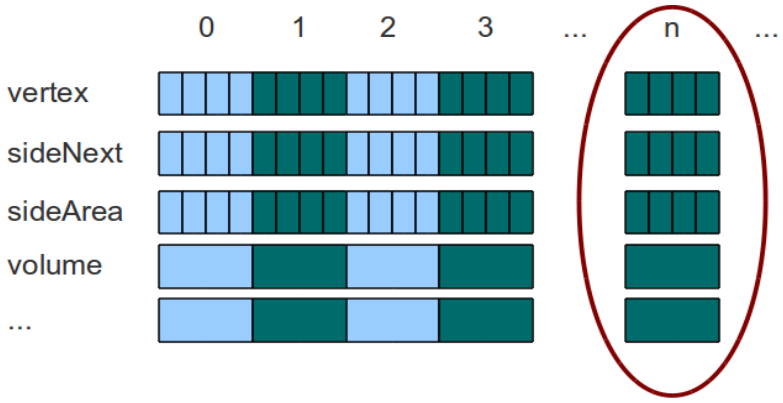
A pontok koordinátái egy dinamikus tömbbe kerülnek; a pontok azonosítására a tömbbeli index szolgál.

A tetraéderek tárolásakor azok csúcspontjaként már csak a pontok referenciáit használjuk. Tároljuk még néhány statikus adatok, ezek: az egyes oldalak területe és normál vektora illetve a tetraéder térfogata és súlypontja. A tetraéderadatok tárolása változatonként eltérő adatszerkezetben történik.

2.2.1. V3, V3X, V3Y – tulajdonságok tömbjei

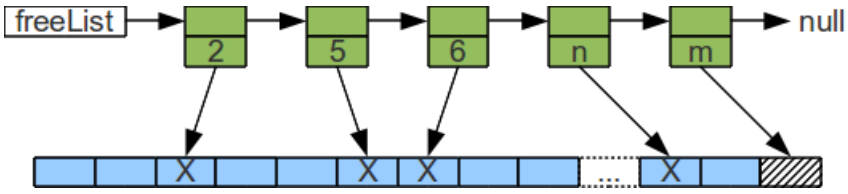
A tetraéderek egyes adatai külön tömbökben tárolódnak. Azaz egy nagy tömbben szerepel az összes tetraéder térfogata, egy másikban az oldalszomszédok referenciái stb.

Egy tetraéder azonosítására az indexe szolgál (*tTetraRef*), ez alatt az index alatt található meg a tetraéder valamennyi adata az egyes tömbökben.



2. ábra: Tulajdonságok tömbjei

Ha törlünk egy tetraédert a hálózatból, valamilyen módon nyilván kell tartani, hogy az adott index üres. Ehhez a V3 egy láncolt listát használ, amibe növekvő sorrendben szúrja be a felszabadult helyek indexeit. A lista rendezettsége szükséges ahhoz, hogy a tömb bejárásakor gyorsan megtaláljuk és átléphessük az üres elemeket. A lista rendezetten tartása azonban időigényes feladat.



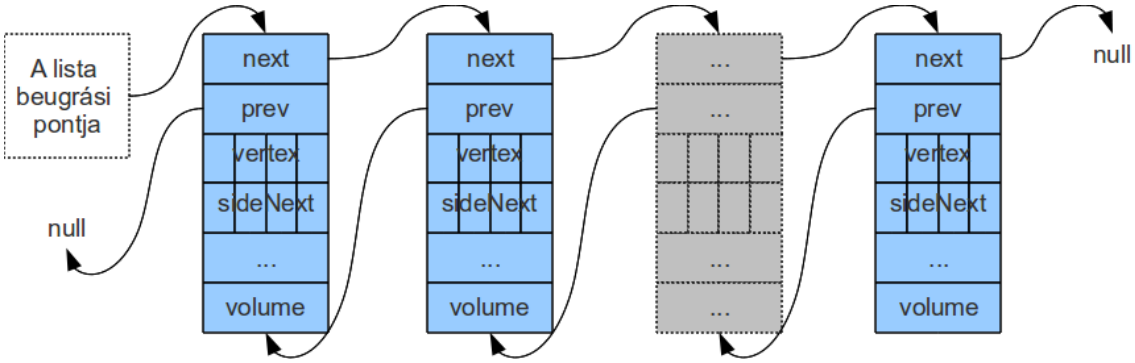
3. ábra: Üres helyek nyilvántartása láncolt listában

A V3X változat esetén egy index érvénytelenségét csak azzal jelezzük, hogy a hozzá tartozó térfogatot negatívra állítjuk. Az első ilyen szabad indexet egy külön változóban tároljuk. Új tetraéder beszúrásakor ezt az első szabad indexet használjuk, majd a megkeressük a következő szabad helyet, és eltároljuk.

A V3Y egyszerre állítja be a térfogatot negatívra (lásd V3X) és fűz hozzá egy elemet az üres helyek listájához (lásd V3). A lista azonban nem rendezett, az újonnan felszabaduló helyeket a lánc elején tároljuk, új tetraédert pedig az első elem által mutatott helyre szúrunk be. A negatív térfogatokkal való megjelölést csak a bejáráskor használjuk, pontosan úgy, ahogy a V3X esetén tettük.

2.2.2. V4 – láncolt lista

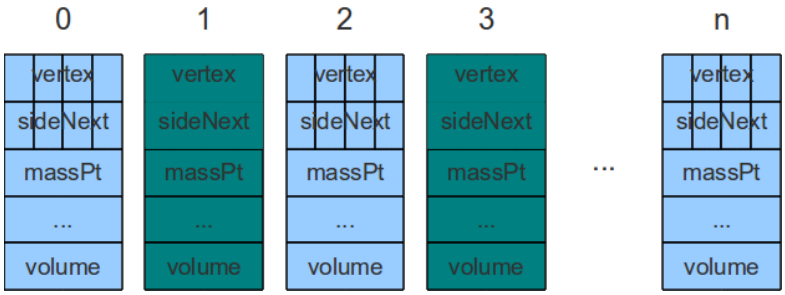
A tetraéderhálózat elemei ebben a verzióban egy duplán láncolt listát alkotnak. (Lásd 4. ábra.) Minden egyes tetraéderhez egy struktúrát (*struct*) rendelünk. A tetraéderre vonatkozó összes adatot ebben a struktúrában tároljuk.



4. ábra: Tetraéderstruktúrák láncolt listája

2.2.3. V5, V5X, V5Y – tetraéderek tömbje

A V5 változat ötvözi a tömbös és a láncolt listás megoldások sajátosságait. Az elemek megegyeznek a V4 változat láncolt listájának elemeivel, de azokat ezúttal egy tömbben tároljuk. Ezzel megspóroljuk az előre és vissza mutató pointerek tárolását, és persze a mutatóműveleteket is.



5. ábra: Struktúrák tömbje

A tömbös változat újra előhívja az üres helyek nyilvántartásának problémáját is. A V5 változat ugyanazt a nyilvántartást alkalmazza, mint a V3; a törölt elemek indexeit egy láncolt listában tárolja, és a bejáráskor is ezt a listát figyeli. A V5X a felszabaduló tetraéder térfogatát negatív értékűre állítja, a V5Y pedig ugyanúgy kombinálva használja a megjelölésen alapuló és a láncolt listás üreshely-nyilvántartást, mint a V3Y.

2.3. Oldalszomszédossági adatok

Célszerű minden tetraéderre egyszer megkeresni és tárolni, hogy melyek az oldalszomszédos tetraéderei. Kínálkozik egy egyszerű, trükkös algoritmus, ami egyszerre térképezi fel a teljes hálózatot.

Készítsünk egy $4n$ (n a tetraéderek száma a hálózaton) elemű tömböt, ami tárolja az összes tetraéder minden oldalát, csúcspontreferenciák szerint. Ebben a tömbben a találkozó oldalak ponthármasai kétszer fognak szerepelni, hiszen ugyanazok a csúcsok határozzák meg őket. Rendezzük a tömböt (pl. quicksort-tal) a csúcspont-referenciák szerint. Ekkor a kétszer előforduló ponthármasok egymás mellé kerülnek. Nézzük át a tömböt elejétől a végéig. Ahol azonos ponthármasokat találunk egymás után, ott állítsuk be a talált oldalakat kölcsönösen szomszédosnak.

Ha egy újonnan beszúrt tetraéder szomszédait akarjuk megkeresni, a fenti algoritmus alkalmazása aránytalan erőkifejtés lenne. Szerencsére kihasználhatjuk ekkor a csúcsszomszédossági adatok ismeretét.

2.4. Csúcsszomszédossági adatok

A csúcsszomszédossági adatok megmondják egy rácspontról, hogy melyik tetraéderek kapcsolódnak hozzá. Ennek az információnak jó hasznát vesszük például akkor, ha egy új tetraédernek keressük a szomszédait, de hasznos lehet akkor is, ha egy új pontról szeretnénk kideríteni, hogy melyik tetraéderben található.

Ezen adatok nyilvántartására láncolt listák tömbjét használjuk. Egészen pontosan egy m elemű tömböt (m a rácspontok száma), aminek minden eleme egy-egy láncolt lista kezdőcímét tartalmazza, a listák pedig az adott indexű ponthoz tartozó tetraéderek referenciáit. Egy tetraéder adatait rendkívül egyszerű ide felvinni: egyszerűen csak minden csúcspontra beszúrjuk a ponthoz tartozó lánc elejére a tetraéder referenciáját.

2.5. Legközelebbi pont megkeresése

Praktikus, ha gyorsan meg tudjuk mondani egy új pontról, hogy melyik rácspont van hozzá a legközelebb. Ezt az információt használjuk abban az esetben, ha egy új pontot viszünk fel a rácspontok tömbjébe, és ellenőrizzük, hogy szerepel-e már a hálózatban. De hasznos akkor is, ha egy új pontról el kell döntenünk, hogy melyik tetraéder belsejében van.

A legközelebbi pont gyors megkeresésére egy úgy nevezett több dimenziós keresőfát (*kd-tree*) alkalmazunk. Konkrétan egy kiegyensúlyozott, 3 dimenziós fát valósítunk meg, és egy legközelebbi-pont-keresést (*nearest neighbour search*) hajtunk végre azon.

A kiegyensúlyozott fán megvalósított keresés *O(log n)* komplexitású művelet.

3. TESZTEK ÉS TESZTEREDMÉNYEK

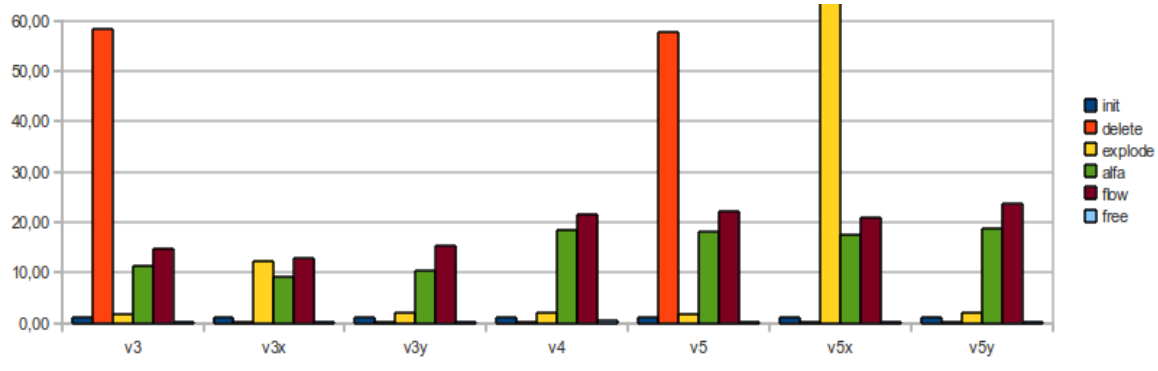
A teszteket két eltérő méretű hálózaton végeztük el. A fúvóka 642 700 tetraédert tartalmaz, a szívócső 155 325-öt. Mind a hét szoftverváltozattal elvégeztük ugyanazokat a teszteket.

3.1. Memóriaigény

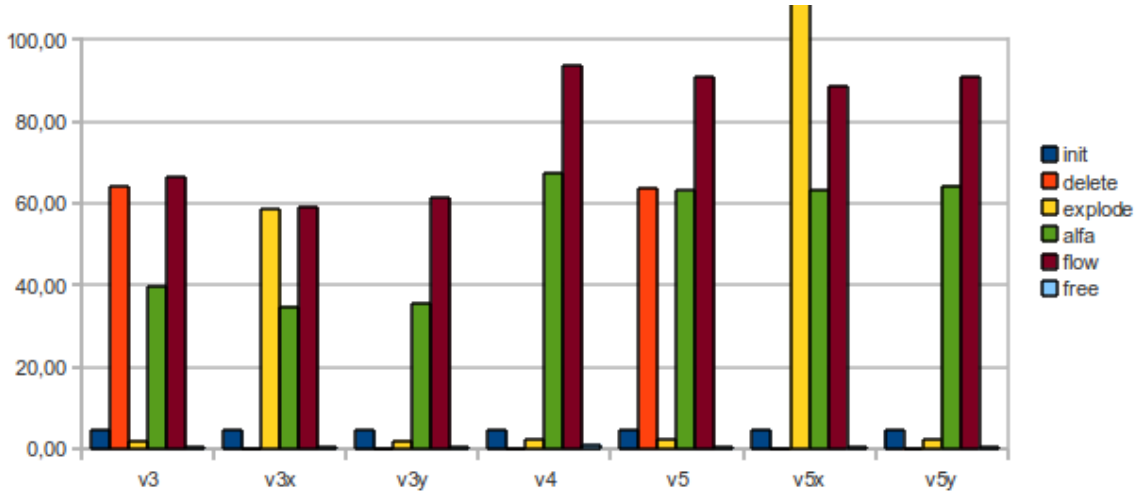
változat	fúvóka	szívócső
V3	249 800 kB	78 852 kB
V3X	249 800 kB	78 692 kB
V3Y	249 800 kB	78 852 kB
V4	259 824 kB	82 416 kB
V5	249 784 kB	78 836 kB
V5X	249 780 kB	78 672 kB
V5Y	249 784 kB	78 832 kB

A táblázat az egyes változatok maximális memóriaigényét mutatja. Jól látszik, hogy a tömbös változatok (V3, V5 és ezek alváltozatai) egyforma memóriaigénnyel dolgoznak, a láncolt lista pedig pedig ezeknél körülbelül 4%-kal több helyet foglal a tárban, nyilván a láncszemek közti kapcsolatok mutatói miatt.

3.2. Futási idők



6. ábra: Tesztek futási ideje a szívócsőmodellen (másodpercben)



7. ábra: Tesztek futási ideje a fúvókamodellen (másodpercben)

3.2.1. Explode

Ez a teszt tetraédereket bont fel a súlypont felhasználásával 4 kisebb tetraéderre. Az eredeti tetraéder törlésre kerül. A teszt 50 000 tetraédert bont fel.

A teszt a fúvóka modellen 421 másodpercig futott, azaz egy nagyságrenddel több időt vett igénybe, mint a V3X *explode* tesztje (58 mp), és két nagyságrenddel többet, mint az összes többi változaté (2 mp körüli értékek). Utólag be kell látnunk, hogy az *explode* teszt kimondottan kedvezőtlenül lett tervezve a V3X és a V5X változatok szempontjából.

Figyelemre méltó, hogy noha a V3X ugyanezt az algoritmust használja beszúráshoz, mégis sokkal gyorsabban képes megvalósítani azt. Az adatstruktúra előnye itt nyilvánul meg igazán. Míg a V5X a teljes tetraéderstruktúrákat éri el, addig a V3X csak a térfogatok tömbjében vizsgálódik. (142 MB helyett csak 5 MB-s tömbbel dolgozik.) A CPU-cache-be előre beolvasott adatok hasznosak, míg a V5X -nél jórészt feleslegesek.

3.2.2. Delete

Ez a teszt 20 000 tetraédert töröl a hálózatból. A hálózat végéről indulva szomszédos tetraéderek láncán fut végig.

A V3 és a V5 is 60 másodperc fölött teljesítette csak a 20 000 elem törlését. Az összes többi változat 0,1 másodperc körüli. A nagy eltérést csak az indokolja, hogy ezekben a változatokban az üres pozíciók listáját rendezetten tartjuk. Ez a megoldás nagyon lassú, cserébe viszont nagyon gyorsan lehet új elemet beszúrni. A V3 és a V3X (illetve a V5 és a

V5X) változatok komplementerei egymásnak: a törlés vagy a beszúrás művelet egyikét nagyon gyorsan el tudják végezni, a másikat viszont csak nagyon lassan.

A V4 láncolt listában tárolja a tetraédereket, abból törölni és abba beilleszteni nagyon gyorsan lehet elemet. Ezen nem csodálkozunk, a láncolt listát szinte erre találták ki.

Meglepő eredmény viszont, hogy a V3 és a V3X megoldásinak kombinálásával egy olyan változatot kaptunk (V3Y), ami ugyanolyan gyors törlést és beszúrást valósít meg, mint a láncolt lista, de tömbös adatstruktúrákat használ. Figyeljük meg azt is, hogy a V3Y nem csak sebességben veszi fel a versenyt a V4-gyel, hanem memóriaigényben is.

3.2.3. Alfa, flow

Ezekben a tesztekben bejárjuk a teljes hálózatot, és minden cellának kiszámítjuk az új állapotértékét (pontosabban az állapottömb egy elemét) a saját és szomszédos cellák aktuális állapotából. Az *alfa* csak az állapotok súlyozott átlagát számolja, a *flow* viszont használja a térfogat és az oldalterület adatokat is.

Itt egyértelműen látszik a V3 csoport előnye. Az az adatstruktúra tehát, amelyben a tetraéderek egyes tulajdonságai különálló tömbökben tárolódnak, gyorsabban járható be, mint az, ahol a tetraédereket egységes egészként kezeltük. A külön tömbös megoldásnál a processzor belső cache-ébe előre beolvasásra kerülő adatok éppen a soron következő tetraéderek megfelelő adatai, és jól kihasználhatóak a következő lépésben. Az egész struktúrákat kezelő változatok esetén viszont ugyanannak a tetraédernek más adatai kerülnek a cache-be, olyanok, melyekre jó eséllyel semmi szükségünk nem lesz a későbbiekben.

Minél több adatát használjuk fel egy tetraédernek egy számítás során, annál nagyobb eséllyel találunk hasznos információt a cache-ben a struktúra alapú megközelítések (V4 és V5) esetén is. Az *alfa* tesztnél a V4-nek még 93%-kal több időre volt szüksége, mint a V3Y-nak, míg a *flow* esetében már csak 58%-os időtöbblettel dolgozik a V4.

3.2.4. pointLocation

A hálózat minden tetraéderében kijelölünk egy pontot, majd megkerestetjük, hogy melyik tetraéderben van az adott pont. A teszt a találat helyességét is ellenőrzi.

Ezt a tesztet nem futtattuk le minden változaton, csak arra szolgált, hogy igazolja a pont bennfoglaló tetraéderének keresésénél használt keresőfás segédalgoritmus hatékonyságát. Csak a V3Y változaton és csak a szívócsőmodellen futtattuk a tesztet. A segédalgoritmus nélkül 1443 másodpercre, annak alkalmazásával pedig 2,3 másodpercre volt szükség. Annak ellenére, hogy a keresőpontokat az oldallapok középpontjához közel választottuk, az 155 325 vizsgált pontból mindössze 34 esetben nem volt találata a segédalgoritmusnak,

azaz csak ennyiszer (0,02%) kellett brute-force módszerrel nekiesni az összes tetraéder vizsgálatának.

3.3. Végkövetkeztetés

Meglepő módon a hét tesztelt szoftverváltozat közül a vizsgált szempontok szerint egyértelműen kiválasztható egy legjobb változat, kompromisszumok nélkül; ez pedig a V3Y. Ha tehát egyéb korlátozó feltételek, követelmények nem állnak fenn, akkor érdemes a tetraéderhálózat elemeit tömbökben tárolni láncolt lista helyett, mégpedig úgy, hogy a tetraéderek egyes adatait különálló tömbökbe helyezzük el, azonos index alatt. A tömbös megoldás ugyan önmagában nem támogatja az elemek könnyű törlését és beszúrását, de az üres helyek kettős nyilvántartásával (megjelölés és láncolt lista) ez a probléma teljesen megszüntethető.

4. IRODALOMJEGYZÉK

[1] Aftosmis, Michael: *Intersection of Generally Positioned Polygons in R3*
http://people.nas.nasa.gov/~aftosmis/cart3d/bool_intersection.html

[2] Bauer Péter: *Programozás I-II. C programnyelv*
Universitas-Győr Kht., 2005

[3] Kernighan, B. W., Ritchie, D. M.: *A C programozási nyelv, Az ANSI szerint szabványosított változat.* Műszaki Könyvkiadó Kft., 2008

[4] Skiena, S. S.: *The Algorithm Design Manual,*
Springer, Berlin, 2010

[5] Wikipedia – kd-tree szócikk
http://en.wikipedia.org/wiki/Kd-tree

[6] Wirth, N.: *Algoritmusok + Adatstruktúrák = Programok*
Műszaki Könyvkiadó, 1982