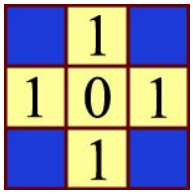
SZÉCHENYI ISTVÁN EGYETEM
MŰSZAKI TUDOMÁNYI KAR

INFORMATIKA TANSZÉK



BSC FOKOZATÚ MÉRNÖK INFORMATIKUS SZAK

DIPLOMAMUNKA

Nagyméretű tetraéderhálózatok hatékony kezelési módszereinek vizsgálata

Martin József
mérnök informatikus

Győr, 2011.

Feladat:

TODO – KELL EZ A LAP IDE???

Részfeladatok:

a) **A szakdolgozat elkészítéséhez szükséges résztevékenységek**:

xxx

b) **A szakdolgozat főbb részei**:

xxx

Belső konzulens: xxx

A diplomamunka benyújtásának határideje: 2010. december 15.

Győr, 2010. február 20.

....

tanszékvezető szakfelelős

A diplomamunkát ellenőriztem: ***beadható*** – ***nem adható be***

dátum belső konzulens aláírása

A diplomamunkát ellenőriztem: ***beadható*** – ***nem adható be***

dátum külső konzulens aláírása

Bíráló adatai: .....
.....
.....

A diplomamunka minősítése

A belső konzulens javaslata:

érdemjegy dátum konzulens aláírása

A Tanszék javaslata:

érdemjegy dátum tanszékvezető aláírása

Az ÁVB határozata:

érdemjegy dátum ÁVB elnök aláírása

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 a 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 apró részekre, cellákra bontanak, és ezen cellákat homogénnek tekintik, azaz feltételezik, 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.

Minél finomabb felbontást választunk, azaz minél több cellára bontjuk fel a vizsgált testet, annál pontosabb eredményt kapunk. A finom felbontásnak azonban ára van: mivel egy elemi lépés (a következő időegységbeli állapot kiszámítása a teljes testen) csak a cellák közvetlen szomszédaira van hatással, ezért az időegységet úgy kell megválasztanunk, hogy egy változás tényleg ne hathasson ez idő alatt messzebb a cellák méreténél. Minél több cellára bontjuk fel tehát a modellezett testet, annál kisebbre kell választunk a szimulációs időegységet, így viszont többször kell elvégeznünk a egymás után a következő állapot kiszámítását. Kétszeresen is meg kell fizetnünk tehát a pontos szimulációért: a cellák számának növelésével az elemi lépések száma is növekszik.

A leggyakrabban használt felbontási egység a tetraéder alakú cella. Egyszerű alakzat, a 4 csúcspontjával leírható, mindössze 4 szomszédja van, és gyorsan kiszámíthatóak olyan alaptulajdonsági, mint például a térfogat és a súlypont. Ebben a dolgozatban olyan testmodellekkel fogunk foglalkozni, melyeknek minden cellája tetraéder. Azokat a tetraéderekből felépülő adathalmazokat, ahol a tetraéderek (szomszédossági) kapcsolatban állnak egymással, a továbbiakban tetraéderhálózatoknak nevezzük. Egy kellő finomsággal felbontott testmodell 10^5 - 10^6 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 – a javaslatot tudjunk tenni adatstruktúrák és algoritmusok egy olyan csoportjára, melyekkel a nagyméretű tetraéderhálózatok gyorsan és könnyen kezelhetőek.

2. A FEJLESZTŐKÖRNYEZET

2.1. A programnyelv

A szoftver fejlesztése C nyelven történt. A nyelv kiválasztása nem is volt kérdéses; a C napjainkban is általánosan használt nyelv fizikai szimulációk terén, illetve más olyan szoftverekben, melyek nagy adathalmazokon végeznek számításigényes műveleteket. Mivel ebben a dolgozatban a tárolási struktúrát minél mélyebb szinten szerettük volna kontrollálni (illetve saját magunk megtervezni), ezért a pl. a C++ kész tárolási megoldásinak (vektorok, listák) nem sok hasznát vettük volna. A dolgozatban levont következtetések viszont nagy valószínűséggel jól alkalmazhatóak más programnyelvi megoldásokban is.

2.2. Operációs rendszer

A fejlesztés és tesztelés teljes folyamatát Ubuntu Linux 10.04 operációs rendszer alatt végeztem. A kernel verziószáma: 2.6.32-27-generic.

2.3. Fejlesztőkörnyezet

A forráskódot a Code::Blocks integrált fejlesztőkörnyezet 10.05-ös verziójában szerkesztettem, ugyanezzel a keretszoftverrel végeztem a fordítást, linkelést és hibakeresést is. Ez egy kis erőforrás-igényű, épp ezért gyors IDE, ami csak a C és C++ nyelveket támogatja, azokat viszont kiválóan. Használata gyorsan elsajátítható, és nagyon kényelmes tud lenni. Támogatja a projekteket, így szépen önálló projektekként lehetett vele kezelni az egyes szoftverváltozatokat.

2.4. Segédprogramok

A diplomadolgozat elkészítéséhez a fentieken kívül még az alábbi szoftvereket használtam:

- **OpenOffice.org** – szövegszerkesztés, táblázatkezelés, ábrák rajzolása
- **gvim** – egyszerű szövegfájlok szerkesztése
- **git** – verziókövető rendszer
- **gimp** - képszerkesztés

A felhasznált szoftverek mindegyike nyílt forráskódú.

3. A PROGRAM FELÉPÍTÉSE

3.1. A program feladata

A fejlesztett szoftver tesztelési célokra született, jelenlegi formájában nem alkalmas végfelhasználói használatra, de a szoftver magjából igény esetén könnyen kialakítható lenne egy olyan függvénykönyvtár, amely nagyméretű tetraéderhálózatokon végzett szimulációk bázisául szolgálhatna.

Az elkészített szoftver nem nyújt felhasználói interfészt, nincs grafikus felülete, de még futási időben változtatható paraméterei sincsenek. Egyetlen megadható parancssori paramétere csak azt mutatja meg, melyik fájlt kell beolvasnia. Indítás után a program végrehajtja az előre meghatározott tesztfeladatokat, majd kiírja az egyes tesztek futtatásához szükséges időt és a maximálisan használt memóriamennyiséget a standard kimenetre.

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
  - egy tetraéder térfogata
  - egy tetraéder súlypontja
  - egy tetraéder egy oldalának felszíne
  - egy tetraéder egy oldalának kifelé mutató normál vektora
  - egy tetraéder egy oldalával melyik tetraéder szomszédos
  - egy rácsponthoz melyik tetraéderek tartoznak
  - egy koordinátáival megadott pont melyik tetraéderben található
- a szoftver futtasson komplex teszteseteket a hálózaton, mérje az ezekhez szükséges erőforrásokat (idő, memória)

3.2. Az egyes modulok feladata

A teljes programot modulokra tagoltuk, melyek önálló, jól körülhatárolható feladatokat látnak el, és egymással csak interfész függvényeiken keresztül érintkeznek.

Mivel a dolgozat célja összehasonlításokon alapuló optimalizálás volt, a tetraéderek tárolását többféleképpen is megvalósítottuk. A szoftver fejlesztése alatt V1-től kezdődő neveket használtunk az egyes változatok megkülönböztetésére. A munka során egyes változatok (V1 és V2) kiestek, mert nem valósítottak meg minden szükséges funkciót, másoknak pedig alesetei születtek, így a végső tesztekben a következő változatok vettek részt: V3, V3X, V3Y, V4, V5, V5X.

A változatok csak a **tetranet** modul kódjában térnek el egymástól, azon belül is a csak tetraéderek tárolásának struktúrájában és az így módosult adatszerkezeteket kezelő algoritmusokban. Minden változatnak egy önálló projekt felel meg a fejlesztői keretrendszerben. Ezek a projektek nagyobb részben közös forráskódot használnak (ezek a fájlok a *base* könyvtárban találhatók), és csak a tetranet modul került projektenként külön tárolásra (a projekt nevével egyező mappában).

Az egyes modulok feladatát ismertetik a következő fejezetek.

3.2.1. main

Ez a modul tartalmazza a program belépési pontját, a main() függvényt. Ebben a modulban csak a **tTetranet** típushoz rendelt interfész függvényeket használjuk, nem olvassuk vagy módosítjuk közvetlenül a hálózatot. Itt kerülnek meghívásra a hálózatot létrehozó és inicializáló rutinok, és itt hívjuk meg a tényleges tesztelési lépéseket.

A modulnak nincs header-fájlja, mert nem kell senki felé interfészt biztosítania.

3.2.2. tetranet

A header-fájl (*tetranet.h*) tartalmazza a tetraéderhálózat definícióját, azaz a **tTetranetDesc** struktúra leírását. A **tTetranet** típust egy erre a struktúrára mutató mutatóként határoztuk meg.

A header ezen kívül tartalmazza azon függvények definícióját, melyek a **tTetranet** struktúrán végrehajthatóak (létrehozás, inicializálás, lekérdezés, bővítés stb.) A különböző megvalósításokban a struktúra definíciója módosulhat, de a függvények definíciója változatlan marad. A tetraéderhálózathoz a tesztelés során csak a függvényeken keresztül férünk hozzá, ezzel biztosíthatjuk azt, hogy a tesztelő rutinok kódja a különböző megvalósítások mellett is változatlan maradhat.

A source-fájl (*tetranet.c*) a függvények megvalósítását tartalmazza. A különböző változatok kódjai az alapstruktúra függvényében jelentősen különböznek egymástól.

A szoftver felépítésének alapkoncepciója az volt, hogy a felhasználónak (szoftver) csak a *tetranet* modul interfészét kell és szabad használnia, minden más modult elfed az itt definiált interfész. Az így elérhető függvények a következők:

A hálózat létrehozása és módosítása

`tTetranet tetranet_new();`

Létrehoz egy új tetraéderhálózatot, és visszaad egy arra mutató pointert.

`void tetranet_free( tTetranet tn );`

Felszabadítja a *tn* hálózat által által foglalt teljes memóriát, törli a hálózatot.

`void tetranet_init( tTetranet tn, char *filename );`

Inicializálja a *tn* hálózatot a *filename*-ben megadott nevű fájlból származó adatokkal.

`tPointRef tetranet_insertPoint( tTetranet tn, tPoint p );`

A *p*-ben megadott pontot felviszi a *tn* hálózat rácspontjai közé és visszaadja annak referenciáját. Ha már szerepelt a hálózatban, akkor visszaadja a már meglévő referenciát.

`tTetraRef tetranet_insertTetra( tTetranet tn, tPointRef pr0, tPointRef pr1, tPointRef pr2, tPointRef pr3 );`

A *p0-p3* pontok által meghatározott tetraédert felviszi a *tn* hálózatba és visszaadja az új tetraéder referenciáját.

`void tetranet_delPoint( tTetranet tn, tPointRef pr );`

Törli a *pr* referenciájú pontot a *tn* hálózatból. Ez a függvény nincs megvalósítva; a pontok ténylegesen nem törlődnek.

`void tetranet_delTetra( tTetranet tn, tTetraRef tr );`

Törli a *tr* referenciájú pontot a *tn* hálózatból.

Adatok lekérdezése a hálózatból

`tPoint tetranet_getPoint( tTetranet tn, tPointRef pr );`

A *tn* hálózat *pr* referenciájú rácspontjának lekérdezése: a visszaadott struktúra a pont koordinátáit tartalmazza.

`tPointRef tetranet_getVertex( tTetranet tn, tTetraRef tr, unsigned vi );`

A *tn* hálózat *tr* referenciájú tetraéderének *vi* indexű csúcspontját adja vissza. A *vi* értéke 0, 1, 2 vagy 3 lehet.

`double tetranet_getTetraVolume( tTetranet tn, tTetraRef tr );`

Megadja a *tn* hálózat *tr* referenciájú tetraéderének térfogatát.

`tPoint tetranet_getTetraMassPoint( tTetranet tn, tTetraRef tr );`

Megadja a *tn* hálózat *tr* referenciájú tetraéderének súlypontját.

`double tetranet_getState( tTetranet tn, tTetraRef tr, unsigned int sti );`

Megadja a *tn* hálózat *tr* referenciájú tetraéderének állapotvektorának *sti*-edik elemét.

`tTetraRef tetranet_getSideNext( tTetranet tn, tTetraRef tr, tSideIndex si );`

Megadja a *tn* hálózat *tr* referenciájú tetraéderének *si*-edik oldalával (*si* = 0 .. 3) szomszédos tetraéder referenciáját.

`double tetranet_getSideArea( tTetranet tn, tTetraRef tr, tSideIndex si );`

Megadja a *tn* hálózat *tr* referenciájú tetraéderének *si*-edik oldalának területét.

`vector tetranet_getSideNormalVector( tTetranet tn, tTetraRef tr, tSideIndex si );`

Megadja a *tn* hálózat *tr* referenciájú tetraéderének *si*-edik oldalának kifelé mutató normál vektorát.

`tTetraRef tetranet_getPointLocation( tTetranet tn, tPoint p );`

Megadja, hogy a (koordinátáival megadott) *p* pont a *tn* hálózat melyik tetraéderében van. A bennfoglaló tetraéder referenciáját adja vissza, vagy *NULL_TETRA*-t (nem létező vagy érvénytelen tetraéderek jelzésére használt referencia).

`tTetraRef tetranet_getLastTetraRef( tTetranet tn );`

Visszaadja a *tn* hálózat utolsó érvényes tetraéderének referenciáját. (A tömb vagy lánc utolsó (érvényes) eleme.)

`tPointRef tetranet_getLastPointRef( tTetranet tn );`

Visszaadja a *tn* hálózat utolsó érvényes pontjának referenciáját. (A tömb utolsó használt eleme.)

`unsigned long tetranet_getNumberOfTetras( tTetranet tn );`

Megadja a *tn* hálózat (érvényes) tetraédereinek számát.

`unsigned long tetranet_getNumberOfPoints( tTetranet tn );`

Megadja a *tn* hálózat rácspontjainak számát.

Beállítófüggvények

`void tetranet_setState( tTetranet tn, tTetraRef tr, unsigned int sti, double value );`

Beállítja a *tn* hálózat *tr* referenciájú tetraéderének állapotvektorának *sti*-edik elemét value értékre.

Bejárófüggvények

`void tetranet_iteratorInit( tTetranet tn );`

Inicializálófüggvény a *tn* hálózat bejárásához. A függvény meghívása után a bejárás (újra) az első tetraédernél fog kezdődni.

`tTetraRef tetranet_iteratorNext( tTetranet tn );`

Léptetőfüggvény a *tn* hálózat bejárásához. Visszaadja a soron következő tetraéder referenciáját, illetve NULL_TETRA-t, ha nincs több elem.

`bool ( *tetranet_atVertexInit )( tTetranet tn, tPointRef pr );`

Inicializálófüggvény a *tn* hálózat *pr* referenciájú rácspontjához kapcsolódó tetraéderek sorozatának lekérdezéséhez.

`tTetraRef( *tetranet_atVertexNext )( tTetranet tn );`

Léptetőfüggvény a *tn* hálózat egy adott rácspontjához tartozó tetraéderek lekérdezéséhez. A soron következő csúcsszomszéd tetraéder referenciáját adja vissza, illetve NULL_TETRA-t, ha nincs több kapcsolódó tetraéder. A rácspontot a *tetranet_atVertexInit* függvény paramétereként kell megadni, még e függvény meghívása előtt.

3.2.3. atvertex

A modul feladata az egy ponthoz kapcsolódó tetraéderek lekérdezése. Ahhoz, hogy ez hatékony lehessen, valamilyen indexelési eljárást kell alkalmaznunk. Az indextáblák struktúrája, az ezeket kezelő műveletek és a kapcsolódó memóriakezelési eljárások mind a forrásfájlban (*atvertex.c*) kerülnek megvalósításra; a fejlécfájlban (*atvertex.h*) csak a publikus függvények definíciója található. Az atvertex modul független a tetraéderhálózat struktúrájától, megvalósításától abban az értelemben, hogy a saját függvényeinek forrásában csak a tetraéderhálózat publikus függvényeit használja. Így a modul kódja változatlan maradhat a hálózat különböző megvalósításaival is.

3.2.4. neighbours

A modul feladata a hálózat szomszédossági viszonyainak feltérképezése. Az egyes tetraéderek szomszédainak lekérdezésében viszont nincs a modulnak feladata. Maguk a szomszédossági kapcsolatok ugyanis a tetraéderhálózat struktúrájában vannak eltárolva. A a neighbours modulnak csupán ezt a kapcsolatrendszert kell felépítenie illetve tetraéderek törlésekor vagy hozzáadásakor módosítania. A modul kódja csak a tetraéderhálózat publikus függvényeit használja, így független a hálózat megvalósításától.

3.2.5. nearest

A modul feladata egy adott ponthoz legközelebb eső rácspont megkeresése. A hatékony keresés érdekében a modul felépít egy speciális keresőfát a rácspontokból. Erről bővebben lásd a 4.5.3. fejezetet. Az interfész függvényekkel inicializálhatjuk a keresőfát, kereshetünk benne illetve bővíthetjük azt.

A legközelebbi rácspont ismeretének két esetben is hasznát vesszük. Egyrészt akkor, mikor a rácsot bővíteni akarjuk, és ellenőrizni kell, hogy a beszúrandó pont nem eleme-e már a rácsnak. Másrészt akkor, mikor egy tetszőleges nem rácsponthoz keressük az azt tartalmazó tetraédert.

3.2.6. testcase

Ez a modul tartalmazza az egyes teszteseteket. Ezek a tetraéderhálózaton megvalósított valamely funkciót, esetleg funkciókat tesztelik. A tesztesetek mérik a futtatáshoz szükséges időt és a lefoglalt maximális memória mennyiségét. A program kimenete is itt íródik a képernyőre.

3.2.7. vector

Háromdimenziós vektorok típusát és az ezeken végezhető műveleteket definiálja illetve valósítja meg.

3.2.8. errors

Néhány egyszerű segédfüggvényt tartalmaz a hibák kiíratásának megkönnyítésére. Főleg a fejlesztés során volt szerepe, a végső mérések, tesztek futtatásakor már nem szabad meghívódnia egyetlen itt implementált hibajelző függvénynek sem.

3.2.9. nasreader

A modul a *.nas* kierjesztésű fájlokból való olvasást támogatja. A .nas fájlok tetraéderhálózatok definícióját tartalmazzák. A **nasreader** modul az egyes pontok és tetraéderek adatait szolgáltatja szekvenciális, előfeldolgozott (azaz helyes formátumra alakított) formában.

3.2.10. common

Általános, közös használatra szánt konstansokat és változókat tartalmazó modul. A **bool** típust definiáljuk benne, és azt a határszámot, aminél kisebb lebegőpontos számokat már nullának tekintünk (**eps**). Itt tartunk fenn egy-egy változót a program saját nevének és a generálás dátumának tárolására.

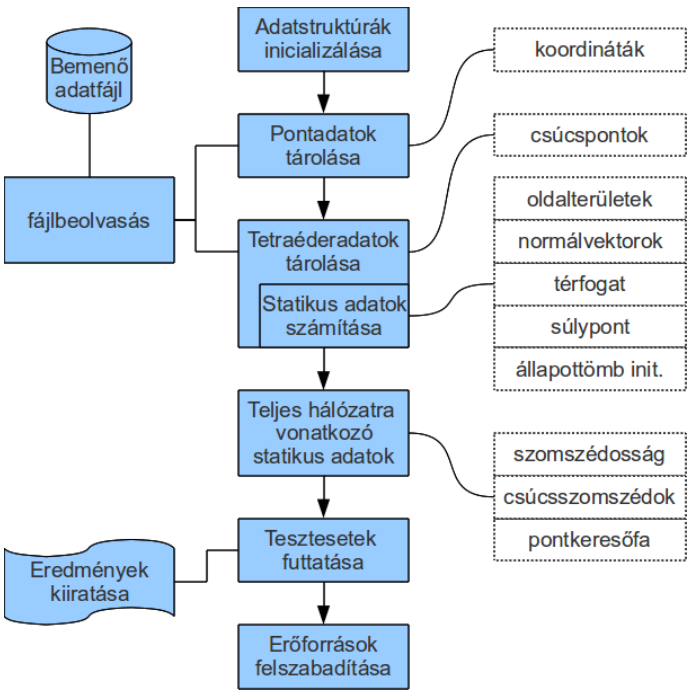
4. A PROGRAM MŰKÖDÉSE

4.1. Általános ismertetés

A diplomamunka keretében fejlesztett program célja, hogy nagyméretű tetraéderstruktúrák tárolási és kezelési módjainak hatékonyságát vizsgálja. Ezért a szoftver nem egy tényleges, konkrét számítási feladatot végez el, hanem tesztesetek sorát futtatja le, és kiírja a mérési eredményeket a képernyőre.

Mivel a cél összehasonlításokon alapuló optimalizálás volt, a tetraéderek tárolását többféleképpen is megvalósítottuk. A szoftver fejlesztése alatt V1-től kezdődő neveket használtunk az egyes változatok megkülönböztetésére. A munka során egyes változatok (V1 és V2) kiestek, mert nem valósítottak meg minden szükséges funkciót, másoknak pedig alesetei születtek, így a végső tesztekben a következő változatok vettek részt: V3, V3X, V3Y, V4, V5, V5X. A változatok csak a **tetranet** modul kódjában térnek el egymástól, azon belül is a csak tetraéderek tárolásának struktúrájában és az így módosult adatszerkezeteket kezelő algoritmusokban.

A program alapvető működését az alábbi folyamatábra szemlélteti.



1. ábra: A program alapvető működése

A következő fejezetek az 1. ábrán feltüntetett lépéseket mutatják be részletesebben.

4.2. Fájlok beolvasása

A program bemenetéül *.NAS kiterjesztésű fájlok szolgálnak, amik a NASA által fejlesztett Nastran nevű végeselemű-analízis szoftver fájlformátuma. A fájlformátum egyszerű szöveges fájl; egy tetraéderhálózat teljes felépítését tárolja. A tartalmazott adatokból a mi programunk csak a rácspontok adatait (GRID sorok, 1 azonosítóból és 3 koordinátából állnak) és a tetraéderek adatait (CTETRA sorok, 1 azonosítóból és 4 pontazonosítóból állnak) használjuk fel. A bemeneti fájlok olvasását és a kinyert adatok konvertálását (pl. koordinátát *double* típusú változóra) a **nasreader** modul végzi. Lásd bővebben a 3.2.9 fejezetet.

4.3. Pontok tárolása

A pontokat minden minden programváltozatban azonos módon tároljuk. A pontok koordinátái egy dinamikus tömbbe kerülnek; a pontok azonosítására a tömbbeli index szolgál (**tPointRef**). A tömb elemei **tPoint** típusúak, ez egy x, y és z nevű, *double* típusú mezőket tartalmazó *struct*. Új elem felvitelére csak a tömb végén van lehetőség. Pontot törölni nem lehet. Ez gyors és kényelmes megoldás, mert a pontok referenciája mindvégig állandó marad.

A tetraéderek tárolásakor csak a csúcspontok referenciáit tároljuk el, a tényleges koordináták csak a pontok tömbjében találhatóak meg. Ezekre egyébként csak ritkán van szükség; a legtöbb belőlük származó statikus adatot előre kiszámoljuk és a tetraéderek tulajdonságaként tároljuk el (így a térfogatot, a súlypont koordinátáit, az oldallapok területét és kifelé mutató normál vektorát).

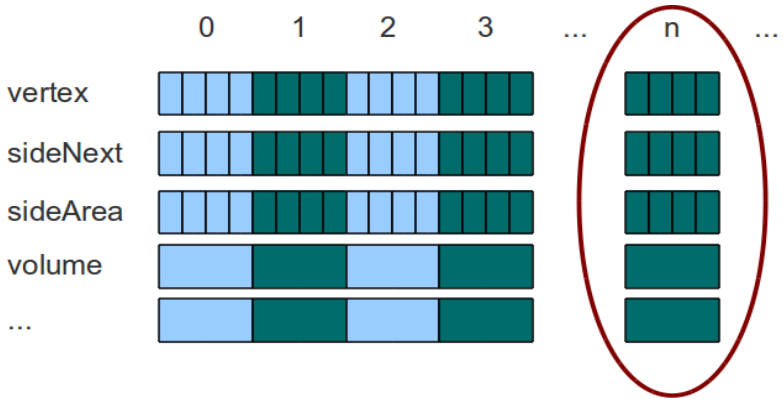
4.4. Tetraéderek tárolása

Egy tetraédernek nem csak a Nastran-fájlból kiolvasott csúcspontjait tároljuk el, hanem már itt kiszámítunk néhány olyan adatot, ami a futtatások során sosem fog változni. Ezek a statikus adatok: az egyes oldalak területe és normál vektora illetve a tetraéder térfogata. Az adatok tárolása változatonként eltérő adatszerkezetben történik.

4.4.1. V3 – 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. Az 1. ábra mutatja az n-edik tetraéderhez tartozó adatokat. Látható, hogy az egyes tulajdonságok más-más tömbökből

származnak. Az egyes elemek hossza nem feltétlenül azonos, csak az áttekinthetőség miatt készült így a rajz. Némely elem nem egyetlen adatot tartalmaz, hanem egy tömböt, pl. a vertex tömb egy eleme a 4 csúcspont referenciáinak tömbje.



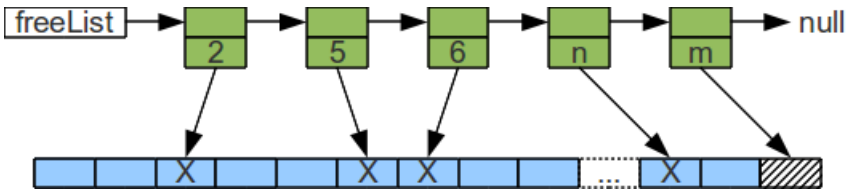
2. ábra: Tulajdonságok tömbjei

A struktúra előnye, hogy bejárása gyors, és ha lineárisan haladunk végig a hálózaton, akkor jól kihasználhatóak a processzor belső cache-ébe előre betöltött adatok. Ha pedig nincs szükségünk minden adatra egy bejárás során, akkor ezek nem is kerülnek beolvasásra a memóriából. Ezen kívül a tömbök létrehozása és bővítése nagyméretű allokációs lépésekkel történik, ami várhatóan gyorsabb, mint ha minden tetraéderhez külön foglalnánk memóriát.

Nem triviális, hogy mi történik az egyes tetraéderek törlésekor. Hatékony megoldás lehetett volna az, hogy a törölt tetraéder helyére mozgatjuk a hálózat utolsó elemét. Ekkor a tömbök mindig folytonosak maradnak, bővítéskor pedig az új tetraéder csak a hálózat végére kerülhetne. Ha nem ezt az utat választjuk, a tömbökben lyukak keletkeznek, amiket valahogy nyilván kell tartanunk. Figyelnünk kell ezekre ugyanis a bejárás során, nehogy az itt lévő – érvénytelen – adatokkal is számoljunk. Ha pedig nem akarunk pazarlóan bánni a memóriával, akkor bővítés esetén az új tetraédereket ezekre az üres helyekre kell beszúrnunk.

Hogy miért vetettük mégis el ezt a rendkívül jónak ígérkező megoldást? Azért, mert az átrendezésnek van egy kellemetlen hozadéka is: az utolsó helyről átmozgatott tetraéder referenciája megváltozik. Ha valahol valamelyik felhasználó ezt tárolta, kellemetlen meglepetésben lesz része, ha újra hivatkozni akar rá. (Feloldható lenne ez azzal, hogy a referencia is legyen tárolt adat, legyen egy saját tömbje. Ekkor azonban elveszítenénk azt hatalmas előnyt, hogy a referencia – mint index – ismeretében egyetlen lépéssel elérhetőek a tetraéder adatai.)

A V3 változat tehát nyilvántartja az üres helyeket. Ehhez egy láncolt listát használ, amibe növekvő sorrendben szúrja be a felszabadult helyek indexeit. A listának mindig van legalább egy eleme, ami az utolsó érvényes tetraéder utáni indexet tárolja. (Lásd a 2. ábrát.) Új tetraéder beszúrásakor a lánc elő eleme által tárolt helyet használjuk. A hálózat bejárásakor a tömbindexeket és a listaelemeket párhuzamosan léptetve ellenőrizzük, hogy az aktuális index nem egy üres hely-e éppen. Ezért kell a listának rendezettnek lennie, különben minden egyes lépésnél végig kellene nézni a teljes listát, hogy szerepel-e benne valahol az indexünk.



3. ábra: Üres helyek nyilvántartása láncolt listában

A módszer legnagyobb hátránya, hogy a tetraéderek törlése lassú, ugyanis minden egyes felszabadított indexnek meg kell keresnünk a helyét a listában, hogy megtarthassuk a rendezettséget.

4.4.2. V3X

Ez a változat a V3 gyenge módosítása, csupán az üres helyek nyilvántartásában különbözik attól. Egy tetraéder törlésekor a pozíciót megjelöljük (flag), érvénytelenségét azzal jelezzük, hogy a térfogatát 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.

Ez a megoldás nagyon gyors törlést fog eredményezni, hosszabb viszont a beszúrás ideje, mert meg kell találnunk a következő szabad helyet, és ehhez rossz esetben akár a teljes tömböt (a jelzőként szolgáló térfogattömböt) végig kell olvasnunk.

Bejáráskor előny a V3 megoldásához képest, hogy nem kell a láncolt listát figyelni és kezelni, hátrány azonban, hogy minden lépéskor be kell olvasni a memóriából a térfogattömb aktuális elemét, hogy ellenőrizzük, nem negatív-e. Így akkor is végigolvassuk a térfogattömböt egy teljes bejárás során, ha annak adatira nincs is szükségünk. A mérések azt mutatták, hogy nincs jelentős különbség a V3 és V3X változat bejárási hatékonyságában.

4.4.3. V3Y

A V3 változattal gyorsan lehet új elemet beszúrni, a V3X pedig gyors törlést tesz lehetővé. A V3Y változat ötvözi a fenti két módszert úgy, hogy azok előnyös tulajdonságait tartja csak meg.

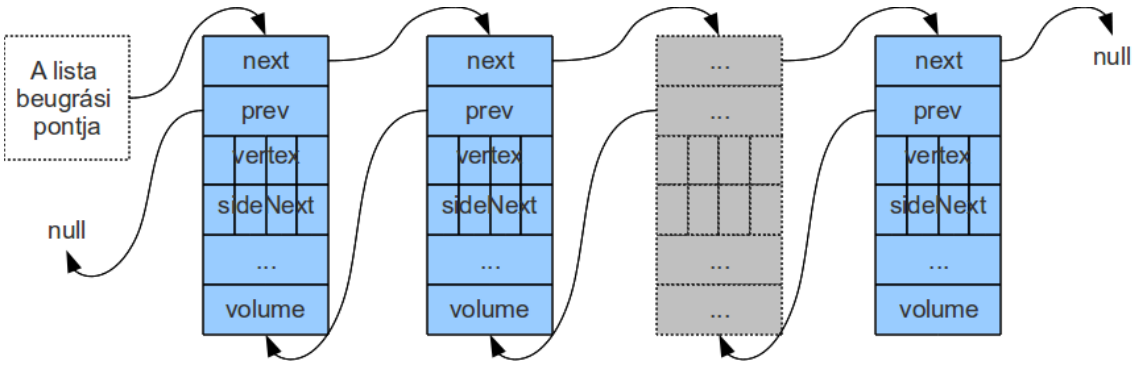
A láncolt listás nyilvántartásnak az volt gondja, hogy sok időbe került fenntartani a lista rendezettségét. Ez a rendezettség csak a bejárás gyorsításához kellett. A bejárás viszont – láttuk a V3X esetén – nem kevésbé hatékonyan elvégezhető az üres helyek megjelölésével.

A V3Y egyszerre állítja be a térfogatot negatívra és fűz hozzá egy elemet az üres helyek listájához. A lista azonban nem rendezett, az újonnan felszabaduló helyeket a lánc elején tároljuk. Ez nagyon gyors lesz, mert nem kell keresnünk a láncban. Új elem beszúrásakor pedig mindig a lánc első eleme által mutatott helyre szúrunk be. Ez is gyors művelet.

A negatív térfogatokkal való megjelölést csak a bejáráskor használjuk, pontosan úgy, ahogy a V3X esetén tettük.

4.4.4. V4 – láncolt lista

A tetraéderhálózat elemei ebben a verzióban egy duplán láncolt listát alkotnak. (Lásd 1. ábra.) Minden egyes tetraéderhez egy struktúrát (*struct*) rendelünk. A tetraéderre vonatkozó összes adatot ebben a struktúrában tároljuk. Ez a megközelítés közelebb áll a valósághoz, mint a V3 modell; itt egy tetraéder egy egységes egész, és ilyen önmagukban teljes egységek kapcsolatai alkotják a hálózatot. (Ez olyannyira így van, hogy szomszédossági kapcsolatok leírásánál egyszerűen pointereket használunk az oldalszomszédos tetraéderekre.)



4. ábra: Tetraéderstruktúrák láncolt listája

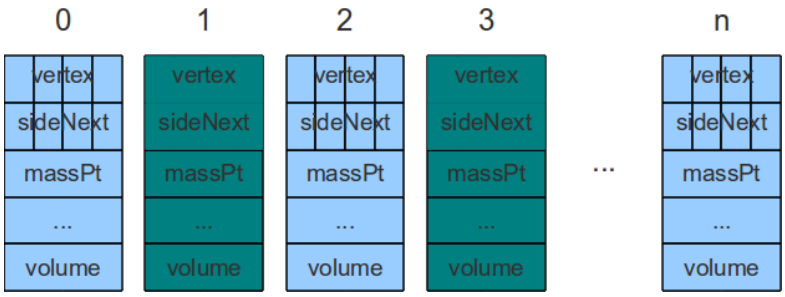
A láncolt listás adatszerkezet mentesít mindazon problémák alól, amiket a tömbös megoldásban az üres helyek nyilvántartása jelentett. Egyszerű átláncolással (és a

memóriaterület felszabadításával) megoldható egy tetraéder törlése. A hálózat bejárásakor sem kell ügyelni a törölt elemekre. Új elemek beszúrása is egyszerűen megoldható a lánc végéhez való kapcsolással.

Ahol hátrányba kerülhet a láncolt lista a tömbös megoldással szemben, az a bejárás sebessége. Itt ugyanis valószínűleg nem számolhatunk olyan gyors memória-hozzáféréssel (vagy a processzor cache olyan szintű kihasználásával), mint a tömbös esetben. A szükséges adatokkal együtt mindig beolvasásra kerülnek a memóriából a tetraéderstruktúrában tárolt közeli, de nem használt adatok is. A mérési eredmények igazolják ezt az előzetes várakozást. Lásd az 5. fejezet.

4.4.5. V5 – 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. Számíthatunk továbbá a tömbök előolvasásából (memóriából a processzorcache-be) származó sebességnövekedésre is, ugyanakkor nem kerülhetjük el a felesleges adatok beolvasását, mivel egységként kezelt struktúrákat használunk.



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.

4.4.6. V5X

A V5X a V5 változat módosítása, ami a V3X estében ismertetett eljárást használja az üres helyek nyilvántartására, azaz a felszabaduló helyeket megjelöljük, mégpedig úgy, hogy az ott található tetraéder térfogatát negatív értékűre állítjuk.

4.4.7. V5Y

A V5Y változat a V3Y megfelelője, ugyanúgy kombinálva használja a megjelölésen alapuló és a láncolt listás üreshely-nyilvántartást, de a V5-nél ismertetett struktúratömbös adatszerkezettel dolgozik.

4.5. A teljes hálózatot jellemző statikus adatok

A program három olyan kisegítő adathalmazzal dolgozik, melyek a hálózatban való gyorsabb eligazodást teszik lehetővé. (Az ezekhez kapcsolódó algoritmusokat egy-egy különálló modul valósítja meg.) Ezek a következők:

- oldalszomszédossági adatok (neighbour modul)
- csúcsszomszédossági adatok (atvertex modul)
- pontkeresőfa (nearestp modul)

Egyik adathalmaz előkészítése sem triviális. Az alkalmazott megoldásokat a következő fejezetek ismertetik.

4.5.1. Oldalszomszédossági adatok

Fizikai szimulációk esetén az egyes tetraédercellák valamely tulajdonsága (nyomás, hőmérséklet stb.) függ a szomszédos cellák állapotától. Szükség van tehát arra, hogy gyorsan le tudjuk kérdezni az oldalszomszédok állapotait. Egy tetraédernek négy oldala van, az ezek mindegyikéhez legfeljebb egy tetraéder kapcsolódik. (A modellezett test felületét alkotó oldalaknak nincs szomszédjuk.) A szomszédsági adatok tárolásához a tetraéderek tárolásánál használt struktúra egy mezőjét használhatjuk tehát, ami 4 referenciát tartalmaz a szomszédos tetraéderekre.

Hogyan kereshetjük meg az oldalszomszédokat? Ha egyesével minden oldalhoz átnéznénk a teljes hálózatot szomszédot keresve, az nagyon időigényes lenne. Kínálkozik viszont egy egyszerűbb, trükkös algoritmus is, ami egyszerre térképezi fel a teljes hálózatot. A programunk ezt használja:

1. Készítsünk egy $4n$ (n a tetraéderek száma a hálózaton) mezőből álló ideiglenes tömböt, ahol minden mező a következő adatokat tartalmazza: 1 tetraéder 1 oldalának 3 csúcspontjának referenciái (ezek egész számok) növekvő sorrendben és a tetraéder referenciája és az oldal indexe.
2. Töltsük fel a tömböt minden tetraéder minden oldalának adatával. Ebben a tömbben a találkozó oldalak ponthármasai kétszer fognak szerepelni, hiszen ugyanazok a csúcsok határozzák meg őket.

3. Rendezzük a tömböt (pl. quicksort-tal) a csúcspont-referenciák szerint. Ekkor a kétszer előforduló ponthármasok egymás mellé kerülnek.
4. Nézzük át a tömböt elejétől a végéig. Ahol azonos ponthármasokat találunk egymás után, ott (a tárolt tetraéder-referenciák és oldalindexek segítségével) állítsuk be a talált oldalakat kölcsönösen szomszédosnak.
5. Kész. Ha végeztünk, felszabadíthatjuk az ideiglenes tömböt.

Ez a módszer lehetőséget nyújt arra is, hogy bizonyos szempontból ellenőrizzük a hálózat konzisztenciáját: ha ugyanazt a ponthármast kettőnél többször látjuk ismétlődni, akkor hálózat szabálytalan, három vagy több oldal ugyanis nem illeszkedhet egymásra, ez fizikai képtelenség.

A fent leírt algoritmus nagyon hatékony a tetraéderhálózat inicializálásakor, nem tudjuk használni azonban akkor, mikor a hálózatot átalakítjuk, beillesztünk vagy eltávolítunk tetraédereket.

Tetraéder törlésekor nincs is gond. Vesszük a négy szomszédos tetraédert, és mindegyikből töröljük az eltávolítandó tetraéderre mutató referenciákat.

Nagyobb gondot okoz, ha egy újonnan beszúrt tetraéder szomszédait akarjuk megkeresni. Egy új elem miatt elvégezni a fenti algoritmust aránytalan erőkifejtés lenne. Szerencsére nem kell ezt tennünk, sőt nem is kell végigellenőriznünk az összes tetraédert, hogy nem szomszéd-e véletlenül. Kihasználhatjuk a csúcsszomszédossági adatok (lásd 4.5.2. fejezet) ismeretét, mindössze arra kell figyelnünk, hogy ezeket már beállítsuk az új elemre, mielőtt az oldalszomszédosságot beállítanánk.

Az új tetraéder egy oldalához a következőképpen keresünk szomszédot:

1. Válasszuk ki az oldal egyik csúcspontját
2. Nézzük meg, hogy mely tetraéderek csatlakoznak ehhez a csúcshoz
3. Ha van a tetraédernek szomszédja, akkor ezek között kell legyen, hiszen annak is tartalmaznia kell az adott csúcsot. Ellenőrizzük le nyugodtan az összeset. Csúcsszomszédból viszonylag kevés van, legalábbis a hálózat méretéhez képest.
4. Ha nem találtunk ezek között szomszédot, akkor az adott oldal a modellezett test felszínén van.

4.5.2. Csúcsszomszédossági adatok

A csúcsszomszédossági adatok megmondják egy rácspontról, hogy melyik tetraéderekben szerepel, azaz másként fogalmazva, hogy melyik tetraéderek kapcsolódnak hozzá. Ennek információnak jó hasznát vesszük például akkor, ha egy új tetraédernek keressük a szomszédait, lásd a 4.5.1. fejezetet. De hasznos lehet akkor is, ha egy új pontról szeretnénk kideríteni, hogy melyik tetraéderben található.

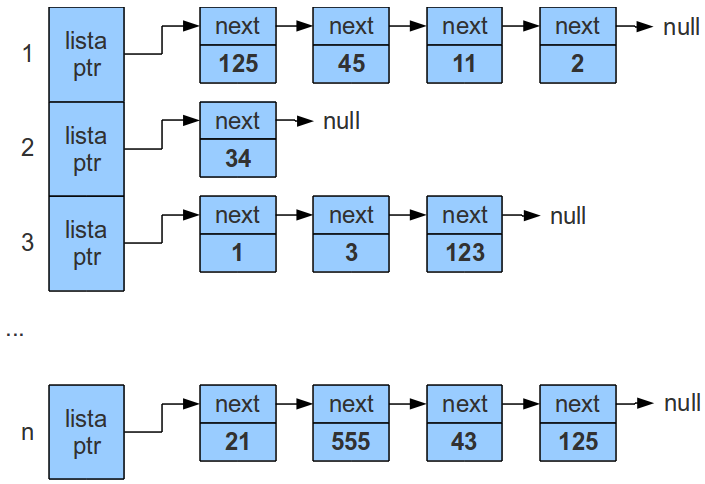
Mivel a tetraéderek adataiból ez az információ nem deríthető ki gyorsan (minden egyes tetraédert meg kellene vizsgálni, hogy tartalmazza-e az adott pontot), ezért célszerű előre feltérképezni és indexelni a teljes hálózatot.

Az, hogy hány tetraéder kapcsolódik egy rácsponthoz, egyáltalán nem egyértelmű. Egy egytetraéderes hálózatban ez a szám minden pontra 1, de egy bonyolult hálózatban gyakran 10 fölötti, szélsőséges esetben pedig elérheti akár a százas nagyságrendet is. A kapcsolatok számának összege a teljes hálózaton azonban nagyon is meghatározott: mivel minden egyes tetraéderre pontosan 4 hivatkozás történik (a 4 csúcspontból), ezért az összes tárolandó kapcsolatok száma $4n$, ahol n a tetraéderek száma.

Ilyen szép elemszámra adja magát a megoldás, hogy tároljuk a tetraéderreferenciákat rendezve egy nagy tömbben, és használjuk egy kis segédtömböt annak indexelésére, hogy hol kezdődnek az egyes pontokhoz tartozó tetraéderreferenciák. A teljes hálózat feltérképezése gyorsan megoldható lenne egy hasonló rendezéses algoritmussal, mint amilyet az oldalszomszédosság esetén használunk.

Az ötlet használható mindaddig, amíg nem akarunk dinamikusan változtatni a hálózat felépítésén. Ebből a kéttömbös struktúrából azonban eltávolítani vagy – különösen – ahhoz hozzáadni egy tetraéder adatait rendkívül körülményes dolog.

Ezért a programban egy kevésbé helytakarékos, talán kevésbé gyors elérésű, ugyanakkor könnyen kezelhető adatszerkezetet használunk: láncolt listák tömbjét. Egészen pontosan egy m elemű tömböt (m a rácspontok száma), aminek minden eleme egy-egy láncolt lista kezdőcímét tartalmazza (pointerként), a listák pedig az adott indexű ponthoz tartozó tetraéderek referenciáit.



6. ábra: Egy ponthoz tartozó tetraéderek

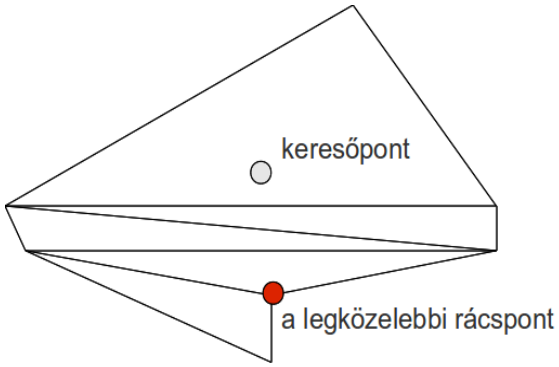
Egy tetraéder adatainak felvitele rendkívül egyszerű: egyszerűen csak minden csúcspontra beszúrjuk a ponthoz tartozó lánc elejére a tetraéder referenciáját. Törölni sem sokkal nehezebb, igaz, itt meg kell keresni, hogy hol van az adott ponthoz tartozó láncban a törölni kívánt tetraéder referenciája, így ez valamivel lassúbb művelet.

4.5.3. Legközelebbi pont megkeresése

Legyen adott 3 koordinátájával egy pont, ami nem szükségszerűen már létező rácspont. A feladat az, hogy találjuk meg a rácspontok közül azt a pontot, amelyik a legközelebb esik hozzá. Ezt a feladatot oldja megy hatékonyan a **nearestp** modul.

Mi szükség lehet erre? A szoftver két helyen is használja saját működésének egyszerűsítésére a legközelebbi pontot.

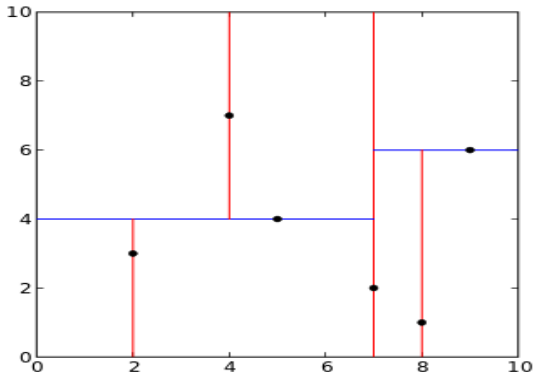
1. Abban az esetben, ha egy új pontot viszünk fel a rácspontok tömbjébe, szeretnénk elkerülni, hogy az egybeessen valamelyik már létező ponttal, azaz a pontok duplikációját. Ezért egy pont beillesztése előtt mindig ellenőrizzük, hogy szerepel-e már a hálózatban. Hogy ne kelljen minden pontot megvizsgálni, megkeressük a ponthoz legközelebb eső rácspontot, és csak azt vizsgáljuk. Ha ez egybeesik az új ponttal (vagyis ha egy elhanyagolhatóan kicsinek választott konstans távolságnál közelebb esik hozzá), akkor a pontot nem visszük fel újra, csak jelezzük a hívónak, hogy a pont már létezik, és visszaadjuk a referenciáját. Ha nem esik egybe vele, akkor semelyik másik ponttal sem eshet egybe, tehát felvisszük új rácspontként.
2. Abban az esetben, ha egy új pontról el kell döntenünk, hogy melyik tetraéder belsejében van, nem nézzük végig rögtön az összes tetraédert. Először megkeressük a hozzá legközelebb eső rácspontot, és megvizsgáljuk az ahhoz kapcsolódó (csúcsszomszédos) tetraédereket, esetleg azok szomszédait. (A szoftver elvégzi a vizsgálatot a szomszédokra is.) Ezzel megvizsgáltuk a pont közelébe eső tetraéderek jó részét. Ha így nem találtuk meg a pontot tartalmazó tetraédert, akkor a szoftver megvizsgálja a hálózat összes tetraéderét, és csak ha így sem találtunk bennfoglaló tetraédert, akkor jelentjük ki, hogy a pont a hálózaton kívül van. Az összes tetraéder megvizsgálása rendkívül időigényes feladat, néha azonban nem kínálkozik más, egyszerűbb megoldás. A 7. ábra egy olyan síkbeli hálózatot mutat be, ahol nincs a legközelebbi pont környezetében a bennfoglaló alakzat. Látható, hogy nem is olyan nehéz ilyet konstruálni. Az esetek nagy részében azonban megtaláljuk a keresett pontot legközelebbi pont szomszédai között. Így ha nem is tökéletes megoldást, de hatékony gyorsítást jelent módszerünk.



7. ábra: Túl távoli a legközelebbi pont

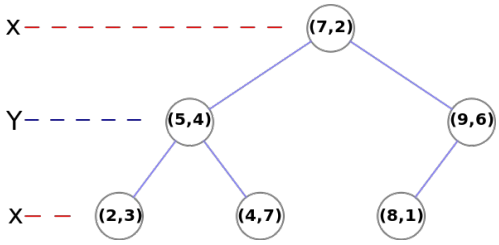
A legközelebbi pont gyors megkeresésére egy úgy nevezett 3 dimenziós keresőfát alkalmazunk. A több dimenziós keresőfák (*k-d tree*) elméletének részletes ismertetésére nem térünk ki. A programunk egy egyszerű, kiegyensúlyozott, 3 dimenziós fát implementál, és egy legközelebbi-pont-keresést (*nearest neighbour search*) tud végrehajtani azon.

A fa minden csomópontjában egy pont (referenciája) áll. A fa minden szintje valamely koordináta szerinti rendezést tükröz, az alapján ágazik el. A fa gyökerét például úgy kapjuk, hogy x koordináta szerint rendezzük az összes elemet, és kiválasztjuk a mediánt. A bal oldali ágban a nála kisebb x koordinátájú, jobbra pedig a nagyobb x koordinátájú elemek kerülnek. A következő szinten mindkét ágat (külön) y koordináta szerint rendezem. A csomópontba ismét a rendezett részhalmaz mediánja kerül, a többiek pedig 2 csoportra osztva kerülnek az alsóbb ágakba. Egy szinttel lejjebb z koordináta szerint rendezünk, aztán megint x szerint, és így tovább, amíg el nem fogynak az elemek. Vizuálisan ez azt jelenti, hogy minden térrészt kettéosztunk úgy, hogy az osztósík átmenjen egy ponton, és mindkét térrészben ugyanannyi elem legyen. A térrészeket ezután újra kettéosztjuk egy, az előzőre merőleges síkkal, majd megint ketté, míg el nem fogynak a pontjaink. Ezzel a teret diszjunkt cellákra daraboltuk fel (lásd a 8. ábrát).



8. ábra: Egy kétdimenziós példa kd-tree-re

Egy pont legközelebbi szomszédjának keresésekor a gyökértől indulunk. Minden szinten megnézzük, merre esik a keresőpont a csomópont választó koordinátájától (pl. ha az adott csomópont egy y szerinti medián volt a lista kialakításakor, akkor csak az y-t vizsgáljuk), és arra haladunk tovább. Minden érintett csomópontra kiszámoljuk a tőle való távolságot, és a legközelebbit feljegyezzük. Ha elértünk egy levélhez, azaz egy olyan cellába, ami már nincs továbbosztva, akkor meg kell néznünk, hogy cella utolsóként átlépett fala közelebb van-e hozzánk (ez az utolsó pont választó koordinátája szerinti távolsága), mint az eddig megtalált, feljegyzett legközelebbi elem. Ha közelebb van, folytatnunk kell ugyanezt a vizsgálatot a fal túloldalán, azaz be kell járunk a szomszédos ágat is. Ha azonban a fal távolabb van, akkor már nem lehet máshol közelebbi elem, és végeztünk, a feljegyzett pont a legközelebbi.



9. ábra: Az előző képen látható fa szerkezete

Mind a fa felépítése, mind a benne való keresés rekurziós lépésekkel egyszerűen elvégezhető. A felépítés a sokszoros rendezés miatt (minden elágazásnál újra kell rendezni az egyre csökkenő méretű halmazt) viszonylag lassú folyamat. Érdemes viszont ezt a felépítési módot használni, mert így a fa kiegyensúlyozott lesz, azaz minden levele közel egyforma hosszú úton lesz elérhető. A tesztek során használt nagyobb hálózat kb. 120 000 rácspontot tartalmaz. A fa levelei így legfeljebb 17 ($log_2 n$) lépésből érhetőek el. A

legközelebbi pont keresésének lépésszáma is $log_2 n$ nagyságrendű, azaz bármely elemhez nagyon gyorsan megtalálható a keresőfa segítségével a hozzá legközelebb eső rácspont.

A keresőfába új elemet beszúrni nagyon egyszerű, csak végig kell lépegetni a gyökértől a keresés szabályai szerint egy levélig, és annak megfelelő oldalára beszúrni az új pontot. Sok elem beszúrása azonban előbb-utóbb elrontja a keresőfa kiegyensúlyozottságát. Ezért érdemes lehet figyelni a leghosszabb út hosszát, és szükség esetén újra elölről felépíteni a keresőfát.

4.6. Tesztesetek

A szoftver a következő teszteket végzi el a felépített hálózaton. A tesztek fordítási időben paraméterezhetők.

4.6.1. Delete

Tetraéderek törlése. A törlő rutin 20 000 tetraédert távolít el a hálózatból, egészen pontosan a hálózat végéről. Első körben letörli az utolsó tetraédert (a tömb vagy lánc utolsó elemét), azután annak valamelyik oldalszomszédját, majd annak is egy szomszédját, egészen addig, míg olyan elemet nem talál, aminek nincs szomszédja. Ezt is letörli, azután veszi újra az utolsó elemet, és onnét folytatja a sort a kívánt elemszámig. Mivel a törlés sorrendjét oldalszomszédsági viszonyok határozzák meg, az keletkező üres helyek (tömbös adatstruktúra esetén) nagy valószínűséggel szórtan lesznek a hálózatban.

4.6.2. Explode

Ez a teszt leginkább az új tetraéderek beszúrását hivatott tesztelni, de tartalmaz törlő lépéseket is. Tetraédereket bont fel 4 kisebb tetraéderre, úgy, hogy ezek mindegyiké a súlypont és az eredeti tetraéder 3 csúcspontja csúcspontja határozza meg. Az eredeti tetraéder törlésre kerül. A teszt 50 000 tetraédert bont fel, a hálózat első elemével kezdve, a bejárás sorrendjét lineárisan követve. A tesztet egy delete teszt után hívjuk meg, így az első 20 000 beszúrás művelet (tömbös tárolás esetén) a hálózat szórtan elhelyezkedő üres helyeire történik.

4.6.3. Alfa

A hálózat minden tetraéderének van egy **states** tömbje, amelynek elemei a tetraéder valamely állapotát írja le. (Pl. fizikai szimulációkban nyomást, hőmérsékletet stb.) Ez a teszt ezeken az állapottömbökön dolgozik. Egyetlen állapot új értékét számítja ki a saját és a szomszédok állapotának súlyozott átlagából, a következő képlet szerint:

$$a_{új} = k a_{régi} + (1-k) \sum_{i=0}^{3} n_{i_{régi}}$$

ahol a az aktuális n_i pedig a i-edik szomszédos tetraéder állapotát jelöli, k pedig egy 0 és 1 között szabadon választható konstans.

A számítást egy ciklusban a hálózat minden tetraéderére végrehajtjuk, az új értékeket ideiglenesen tároljuk (jelen esetben a **states** tömb egy másik indexe alatt), és ha a teljes hálózattal végeztünk, akkor visszamásoljuk ezeket a régi értékek helyére. Ez a teszt 200 ilyen, teljes hálózaton végzett számítást hajt végre egymás után. Azt tudjuk tesztelni ezzel, hogy mennyire gyors a hálózat bejárása, és mennyire hatékonyan tudunk írni és olvasni adatokat az egyes tetraéderekből.

4.6.4. Flow

Hasonlóan az alfa teszthez itt is a szomszédos tetraéderek állapotainak segítségével határozzuk meg egy tetraéder új állapotát. Fontos azonban, hogy itt több adatra is szükségünk van a tetraéderről, mégpedig a térfogatára és az oldalak felszínére. Az elvégzett számítás is bonyolultabb: egy egyszerű áramlási modell:

$$a_{új} = a_{régi} + t \sum_{i=0}^{3} -k \frac{a_{régi} - n_{i_{régi}}}{m_a - m_{n_i}} S$$

ahol a az aktuális n_i pedig a i-edik szomszédos tetraéder állapotát jelöli; k egy szabadon választható konstans; t az időtényező (mértéke meghatározza, hogy egy ciklus alatt mekkora változás mehet végbe egy cella állapotában); S_i az aktuális és az i-edik szomszédos tetraéder közös oldalának területe, az m_a és m_{ni} pedig az aktuális illetve a szomszédos tetraéderben a súlypont és a közös oldal távolsága. Ez a távolság pontosan megegyezik a tetraéder térfogatának (V_a és V_{ni}) és a közös oldalfelszínnek a hányadosával. Így a képlet a következő formára egyszerűsödik:

$$a_{új} = a_{régi} + t \sum_{i=0}^{3} -k \frac{a_{régi} - n_{i_{régi}}}{V_a - V_{n_i}} S^2$$

Ez a teszt is 200 ciklust hajt végre a fent leírt számításból a teljes hálózaton.

5. TESZTEREDMÉNYEK

Az egyes szoftververziók által kilistázott futási eredményeket egy szöveges fájlba irányítottuk, ott gyűjtöttük össze. Ez a fájl kerül most elemzésre, kiértékelésre. A fájl teljes, nyers tartalma a mellékletben megtalálható.

A teszteket két eltérő méretű hálózaton végeztük el. A fúvóka modellje 640 000 tetraédert tartalmaz, a szívócsőé pedig 150 000-t. Mind a 7 szoftverváltozatot lefuttattuk, minden egyes változattal pontosan ugyanazokat a teszteket végeztettük el, és minden egyes teszt futási idejét feljegyeztük. Figyeltük ezen kívül minden változat esetén a futás közben foglalt maximális memóriamennyiséget.

A fúvóka és a szívócső modellen kapott grafikonok hasonlóak lettek, az értékek többnyire az elvárt módon aránylanak egymáshoz. Ahol komolyabb eltérést tapasztalható, ott arra külön kitérünk, egyébként csak a fúvóka modell eredményeire fogunk hivatkozni a továbbiakban.

5.1. Memóriaigény

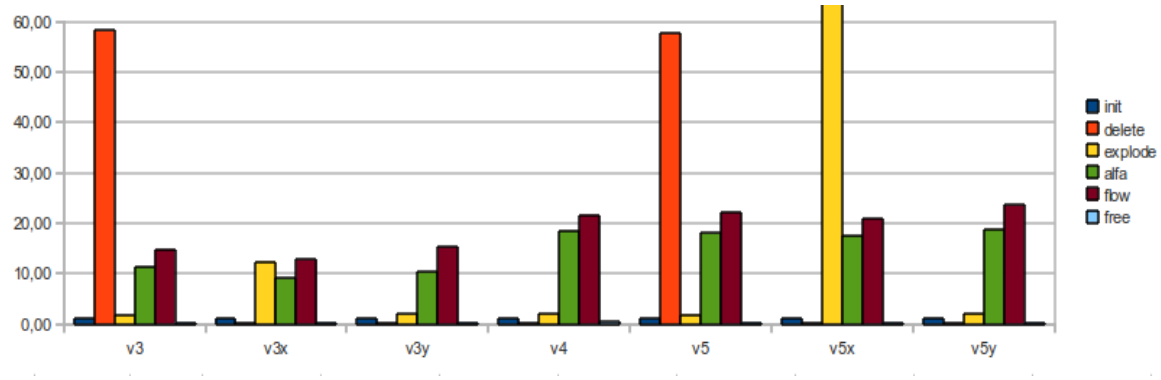
Az alábbi táblázat az egyes szoftverváltozatok által igényelt memóriamennyiséget mutatja:

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

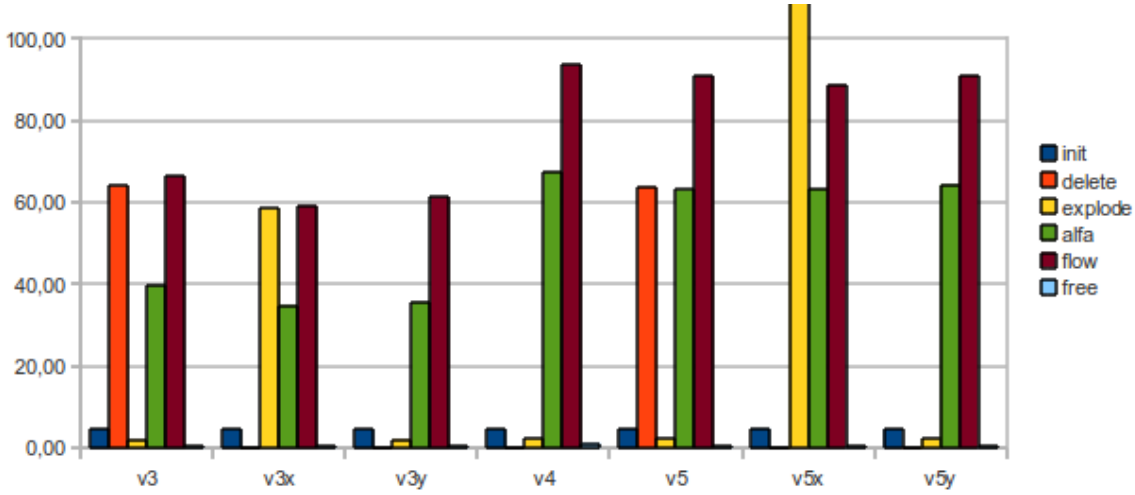
Jól látszik, hogy a tömbös változatok (V3, V5 és ezek alváltozatai), egyforma memóriaigénnyel dolgoznak, a láncolt lista pedig pedig ezeknél körülbelül 4%-kal több helyet foglal a tárban. A különbség egyértelműen azzal indokolható, hogy a láncolt listának tárolnia kell a láncszemek közti előre és vissza mutató kapcsolatokat. Az eltérés nem olyan mértékű, hogy csak ezért le kellene mondanunk a láncolt listák használatáról. Csak ha valóban szűkös memóriakeret áll a rendelkezésünkre, akkor lenne ez megfontolandó, de a memóriaárak alakulását figyelve ez a veszély nem fenyeget.

5.2. Futási idők

Az alábbi grafikonok a 4.6. fejezetben ismertetett tesztesetekhez tartozó futási időket ábrázolják.



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



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

5.2.1. Init, free

Az adatstruktúra generálásához szükséges idő minden változat esetén közel azonos, 4 másodperc körüli. Ha jobban belegondolunk, ez meglepő adat. Igaz, a fájlolvasást és a statikus adatok kiszámítását minden esetben ugyanúgy el kell végeznünk. De míg a V5-ös változat egyetlen lépésben foglalja le magának a teljes tetraédertömböt, addig a V4-es egyesével foglal memóriát minden tetraédernek. Logikus lenne azt gondolni, hogy a

V4-nek ezért sokkal lassúbbnak kell lennie. A tesztek azt mutatják, hogy ez nem így van. Úgy tűnik, hogy a memóriafoglalás művelete igen szépen optimalizált az operációs rendszer szintjén. (Legalábbis van annyira gyors, hogy a fájlolvasás meglehetősen időigényes művelete lesz a szűk keresztmetszet.)

A program lezárásakor a lefoglalt memóriaterületek felszabadítása egyformán rövid ideig tart minden esetben. Ez az idő (2-3 tizedmásodperc) elhanyagolhatóan kevés, a továbbiakban nem is foglalkozunk vele.

5.2.2. Explode

A V5X *exlpode* oszlopának tetejét le kellett vágnunk, hogy a többi adat is látható maradjon. 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. Lássuk, mi történik egy lépésben ezeken a struktúrákon:

1. Letöröljük a hálózat első elemét, az üres helyet megjelöljük – ezzel ez lesz az első üres hely.
2. Beszúrunk egy – a felbontásból származó – tetraédert az első üres helyre, azaz a hálózat elejére.
3. Mikor be akarjuk szúrni a következő tetraédert is, meg kell keresni a következő szabad helyet. Ha a hálózatban nincsenek lyukak, akkor ez a hely a hálózat utolsó eleme utáni pozíció lesz. Ahhoz azonban, hogy ezt megtaláljuk, végig kell nézzük a hálózat minden egyes tetraéderét. Itt veszítünk el rengeteg időt.
4. A maradék két elemet szépen beillesztjük a hálózat végére, már tudjuk, hogy csak ott van szabad hely.
5. A következő felbontáskor viszont újra valahol a hálózat elején szabadul fel hely, ide állítjuk be az első szabad elemet jelző mutatót.
6. Ide viszont újra csak 1 új tetraéder fér el, az után megint végig kell nézni a teljes hálózatot, hogy rájöjjünk, hogy csak a végére tudunk új elemet fűzni. És ezt minden egyes elem felbontásakor megcsináljuk.

Azzal, hogy a felbontandó tetraédereket a hálózat elejéről választjuk, erősen ellene dolgozunk az X-es változatok beszúró rutinjának. Ugyanakkor beláthatjuk, hogy hasonlóan kedvezőtlen konstrukciót egyetlen másik adatszerkezet beszúrási algoritmusához sem tudnánk kitalálni. A V5X-es változat tehát egyetlen olyan alkalmazáshoz sem ajánlható, ahol sok új tetraédert kell felvinni a hálózatba (például gyakran kell finomítani a hálózatot). A grafikonról kitűnik, hogy ez a változat a bejárásos tesztek során sem teljesít különösen jól. Egyetlen erénye, hogy nagyon gyorsan lehet

tetraédereket törölni belőle. Ennek a jó tulajdonságnak azonban önmagában nemigen vesszük hasznát.

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. A tetraéderstruktúra mérete 232 byte, míg a térfogatot egy 8 byte-os *double* írja le; az eltérés 29-szeres. Másként megfogalmazva: a fúvóka modell estében V3X-nek csak egy 5 MB nagyságú tömböt kell kezelnie egy 142 MB-os helyett. Természetesen a V5X esetében sem olvassa be a processzor a teljes a tetraéderstruktúrát a memóriából. De amit a keresett adattal együtt beolvas a belső cache-be (a térfogat mellett álló adatok a struktúrában), az nem hasznos adat; míg a V3X esetében a cache-elt adatok a következő lépésben szükséges térfogatokat tartalmazzák.

5.2.3. Delete

A törlési tesztben két változat ért el kirívóan gyenge eredményt: a V3 és a V5 is 60 másodperc fölött teljesítette csak a 20 000 elem törlését.

Ne csodálkozzunk: a fúvókamodellen és a szívócsőmodellen közel azonosak a futási idők. Ez azért van, mert ebben a tesztben a törölt elemek száma nem függ a tetraéderhálózat elemszámától: minden esetben ugyanannyi tetraédert törlünk, ezért a segédlista, ahol a törölt indexeket tároljuk, is ugyanakkora lesz. Az legtöbb időt pedig ennek a listának a rendezetten tartásával vesztegetjük el.

Az összes többi megoldás 0,1 másodperc körül teljesítette ezt a tesztet. A nagy eltérést csak az indokolja, hogy ezekben a változatokban az üres pozíciók listáját rendezetten tartjuk. Minden egyes elem törlésekor meg kell ezért keresnünk annak a helyét a az üres helyek láncolt listájában. Ehhez jobb híján végig kell néznünk az elejéről a teljes listát. Minél nagyobb a törölt elemek száma, annál lassabban lehet tehát törölni egy újabb elemet. 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 ilyen szempontból komplementerei egymásnak: az 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, ideális választás lehet, ha gyakran vagy nagy arányban változik a hálózat. 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. (A mért eredmények csak 1-2 századmásodperces elérést mutatnak – és azt is V3Y javára.) A trükk nyilván ott van, hogy a V3Y is használ láncolt listát, mégpedig az üres helyek tárolására. Ezt a listát

azonban – ellentétben a V3 megoldásával – nem kell rendezetten tartanunk. A rendezésre csak azért volt szükség, hogy lehetségessé váljon a hálózat gyors bejárása az üres helyek átugrásával; ezt azonban sokkal könnyebben is megtehetjük, V3X megjelöléses módszerével. Az elképzelés tehát pofonegyszerű: mindent használjuk arra, amire legjobban alkalmas.

Figyeljük meg azt is, hogy a V3Y nem csak sebességben veszi fel a versenyt a V4-gyel, hanem memóriaigényben is. A V3Y egy törölt tetraéder nyilvántartására kétszer 4 byte-ot használ: a tetraéder indexét és a láncolt lista következő elemére mutató pointert. A V4 láncolt listája viszont tetraéderenként tartalmaz kétszer 4 byte plusz adatot (előre és hátra mutató pointerek). A V3Y memóriafogyasztása tehát akkor érné el csak V4-ét, ha a hálózat minden egyes elemét letörölnénk. Ennek viszont – lássuk be – nincs sok értelme.

5.2.4. Alfa, flow

Azoknál a teszteknél, amelyek a tetraéderhálózat teljes bejárására épülnek, 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. Ez azzal indokolható, hogy a szükségtelen adatok tömbjeivel a V3-mas esetek egyáltalán nem foglalkoznak, míg ugyanezek a felesleges információk kénytelen-kelletlen beolvasásra kerülnek a memóriából, ha a teljes struktúra részeiként tárolódnak. Ugyanez a magyarázat egy másik megközelítésben is értelmezhető. A processzor belső cache-ébe beolvasásra kerül az éppen lekérdezett adat környezete is, az utána álló byte-ok a memóriából, azért, hogy azt a következő lépésben a processzor gyorsabban elérhesse, ha esetleg szükség lenne rá. A külön tömbös megoldásnál ezek a járulékosan beolvasott 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. Ennek a hatása jól látszik az *alfa* és a *flow* tesztek arányain. Az *alfa* tesztnél csak az állapotinformációkkal dolgoztunk; itt a V4-nek még 93%-kal több időre volt szüksége, mint a V3Y-nak, azaz csaknem kétszer olyan hosszan tartott a teszt. A *flow* esetében használjuk a térfogatot és az oldalterületeket is az állapotinformációk mellett. Itt már csak 58%-os időtöbblettel dolgozik a V4. Ha még több tárolt információra lenne szükségünk, akkor a különbség még tovább csökkenne, de azt sejtjük, hogy a tömbös változatok előnye megmaradna.

6. LEZÁRÁS

6.1. Végkövetkeztetés, tanulságok

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.

Le kell vonnunk még néhány általános tanulságot is a dolgozat végén. Ezek mindegyike szerepel a dolgozat korábbi fejezeteiben részletesebben kifejtve, itt csak megemlítjük őket:

- A láncolt listás megoldás nem volt annyira lassú, mint amennyire előítéletesek voltunk vele. Nem volt érzékelhetően lassúbb egyesével lefoglalni a 640 000 tetraédernek a memóriát, mint egyetlen tömbként.
- A processzor belső cache-e nagy segítségünkre lehet, de csak ha ügyesen a keze alá dolgozunk.
- Egy tömb üres helyeinek – körülményesnek tűnő – nyilvántartása nem feltétlenül vezet lassúbb vagy memóriapazarlóbb megoldáshoz, mint egy lista a maga egyszerű átláncolásaival.

6.2. Amire nem tér ki a dolgozat

Egy hosszabb munkát csak ritkán zár le úgy az ember, hogy ne maradna benne valami kétely: lehetett volna ezt még jobban is csinálni, azt még ki sem próbáltuk, hogy... Vagy, mi lett volna, ha... A következő vázlatpontokban néhány olyan gondolatot villantunk fel, amelyekkel ez dolgozat nem foglalkozott, de ott rejlik bennük a további javítás, fejlesztés lehetősége.

1. Érdemes lenne a szoftvert lefordítani másik C fordítókkal és más operációs rendszer alatt is. Vajon megmaradnak legalább nagyjából a mérési eredmények arányai?
2. A szoftver végleges változatát a következő C fordítási opciók bekapcsolásával fordítottuk: *-O3 -march=native -fprefetch-loop-arrays -funroll-loops -ffast-math*.

Érdemes lenne kipróbálni további fordítási opciókat is. Mi az ami elhagyható? Mi az, ami tényleges javulást okoz?

3. A szoftverben implementált pontkereső fa megoldás tovább optimalizálható, például a rekurziós függvény paramétereinek csökkentésével. Jelenleg egy igazán nyers változat van csak megvalósítva.
4. Az implementált pontkereső fa sok pont beszúrása után teljesen elveszítheti kiegyensúlyozott jellegét. Elegendő lenne egy egyszerű megoldást beépíteni, ami újra építi elölről a teljes fát, ha a leghosszabb lánc benne meghaladja a $log_2 n$ értékét egy előre meghatározott mértékben.
5. A láncolt listás megoldás egyesével foglalja a memóriát a tetraédereknek. Már ez a megoldás is meglepően jónak bizonyult, de érdemes lenne kipróbálni, hogy javít-e ezen még valamit a klasszikusan ajánlott csoportos allokáció. (Azaz: egy allokációval több ezer tetraéder helyét foglalnánk le előre, és azt utólag osztanánk ki.)
6. A tömbös megoldásoknál a tetraéder térfogatában, azaz 8 byte-on jelezzük, hogy az adott tetraéder érvényes-e, nem lett-e törölve. Érdemes lenne ennek az információnak egy új mezőt szentelni egyetlen byte-tal: így egy memóriaolvasással 8-szor annyi tetraéder foglaltsági adatát húzhatnánk be a CPU-cache-be, mint a jelenlegi legjobb változattal.

Sorolhatnánk még további bővítési, javítási ötleteket is. Nem szabad megfeledkeznünk azonban arról sem, hogy az optimalizálási lehetőségek végleges tárházát az fogja igazán korlátozni, hogy milyen is lesz az a végleges struktúra, amin az algoritmusainkat futtatni akarjuk, mely függvények lesznek gyakran hívva, melyekre nem lesz esetleg egyáltalán szükség, röviden: mire is akarjuk valójában használni majd az itt megtervezett megoldásinkat.

7. 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