Bevezetés

Napjainkban a számítógépes térgeometria és grafika számos, folyamatosan fejlődő és terjedő terülten, többek között tervezőprogramokban, videojátékokban, számítógépes animációkban, virtuális valóság szimulációkban is felbukkan. Általánosan elmondható, hogy a megkövetelt minőséggel arányosan a feladatok számítási igénye gyors iramban növekedhet, ezért kulcsfontosságú a megfelelő reprezentációs struktúrák, algoritmusok és feladatspecifikus kerülőutak használata.

Ezen alkalmazások egyik legtöbbet használt absztrakciós fogalma a poliéder, azaz olyan test, melyet minden oldalról sokszögek határolnak. Ezen testek ábrázolása triviális módon történhet borítólapjaik halmazaival, a legnépszerűbbek és legáltalánosabbak a háromszögekkel leírt poliéderek. A használt testek származhatnak tervezőprogramokból, keletkezhetnek egy tárgy térbeli letapogatásával, generálódhatnak különböző algoritmusok segítségével, méretük, részletességük, a kívánt formához való közelségük változó lehet.

A műszerek által létrehozott testek felesleges zajt, hibákat tartalmazhatnak, melyek nem csak bonyolítják, de hibássá is tehetik a számításokat. Egyetlen programon belül gyakori, hogy egyetlen testnek több, különböző részletességű, reprezentációjú változata is létezik párhuzamosan, például a képernyőn megjelenítendő részletes testet használni a fizikai ütközés érzékeléshez túl költséges, egy egyszerűsített, kevesebb lapból álló test alkalmazása nagyságrendekkel gyorsabb működés mellett adhat elfogadható eredményt. Alkalmazástól függően szükséges lehet előállítani testek darabokra tört, deformálódott, felszeletelt mását. Számos területen jelent előnyt, ha feltehető, hogy kizárólag konvex testekkel kell dolgoznunk, mivel így hatékonyabb algoritmusokat használhatunk fel, ekkor a konkáv testeket valamilyen módszerrel konvex darabokra kell osztanunk. Néha van lehetőségünk az eredeti testtel együtt beszerezni ezen testeket is, esetleg manuálisan előállítani őket, azonban egyes területeken ez kizárt vagy nehézkes, ekkor nyújthatnak segítséget a testapproximációs algoritmusok.

A testapproximációs algoritmusok során a bemenet egy célpoliéder, a kimenet pedig egy olyan poliéder vagy poliéder halmaz, mely a feladatspecifikus kritériumok szerint az eredetihez hasonló tulajdonságokkal rendelkezik, azonban nekünk kedvezőbb felépítésű, esetleg további tulajdonságai is vannak. Az eredmény előállítására számos,

módszer létezik, a terület jelenleg is aktív kutatás tárgyát képezi. Egy lehetséges módszer a tér síkokkal való particionálásával bináris tér partícionáló fák felépítése, majd az ezen fák levelein fekvő testeknek (továbbiakban atomok) felhasználása. A módszer egy speciális megközelítését, egy térfogat alapú metrikát használó, bizonyítottan konvergens, tér particionáló, iteratív algoritmust részletesen vázolnak Fábián Gábor és Gergó Lajos cikkei.

Ezen szakdolgozatban kitűzött feladat, olyan program elkészítése, mely a fent említett cikkek által felvázolt elméleti eredményeket, módszereket és adatszerkezetet képes a gyakorlatban bemutatni, ezzel segítve a további kutatásokat, illetve szemléletesebbé, közérthetőbbé tenni. A programnak képesnek kell lennie a megadott céltest kezelésére, atomok létrehozására, síkkal való elvágására, az atomokról való fontosabb adatok, többek között térfogat, céltesttel vett metszet térfogat, átmérő és súlypont számítására, majd az adatok alapján manuálisan, vagy a felhasználó által választott stratégia szerint kért számú lépésben végrehajtani az approximációs algoritmust.

A feladat összetettsége indokolta annak felosztását, a felhasználói felületet, grafikus megjelenítést, vágósík meghatározási és kiválasztási stratégiákat Lukács Péter készítette, az alacsony szintű működést, a geometriai adatszerkezeteket, vágási és test statisztikai metódusokat Tóth Máté implementálta.

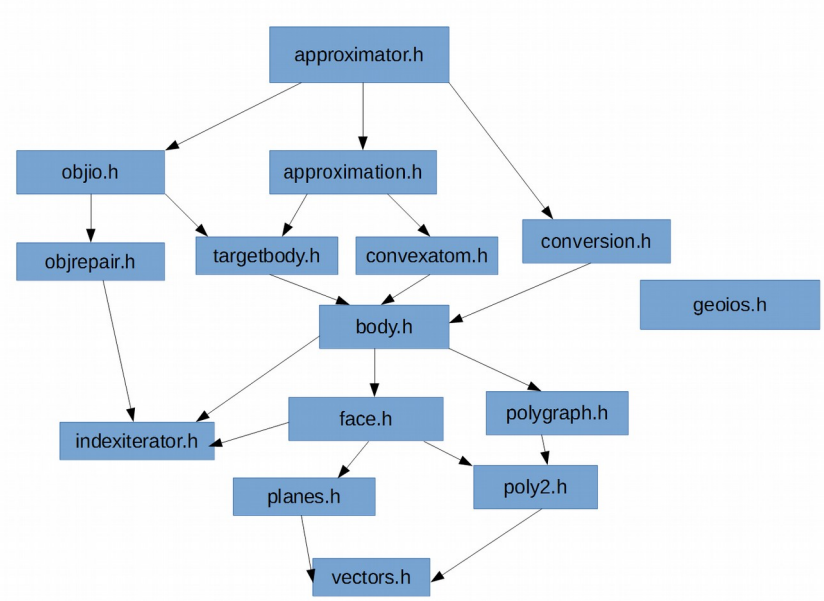
Felhasználói dokumentáció

A működő program felhasználói dokumentációja Lukács Péter szakdolgozatában olvasható, én a saját könyvtáram kódba való beépítését mutatom be egy másik programozó számára. A könyvtár bármely C++11 vagy későbbi szabványt támogató fordítóval felhasználható. Mivel minden típus és függvény sablonok formájában van megvalósítva, a központi forráskód kizárólag header fájlokból áll, emellett elkülönítve megtalálhatóak a tesztelésre és kódpéldák kipróbálására használható példaprogramok forrásai. Az egyetlen külső függőség a GLM csomag, de ez kizárólag a megjelenítéshez szükséges.

A könyvtárat Microsoft Visual C++ 2015, valamint GNU g++ 5.2.0 fordítókkal teszteltem, az utóbbit -std=c++11 és -std=c++14 fordítási direktívákkal használva.

A könyvtárt felhasználó programozók számára fontos kiemelni, hogy a C++ standard könyvtárához hasonlóan az osztályok metódusai kerülik a kivételkezelést, ennek fő oka a sebesség. A típusok és függvények mind az `approx` névtérben találhatóak, ezzel támogatva a C++ nyelv kód elkülönítési irányelveit.

A könyvtár fejállományainak egymásra épülése az alábbi diagramon látható. Alapvető felhasználásnál elég az approximator.h hozzáadása az includeokhoz.

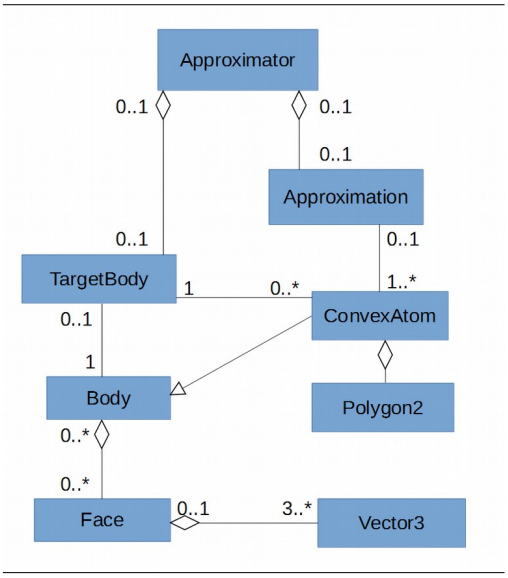


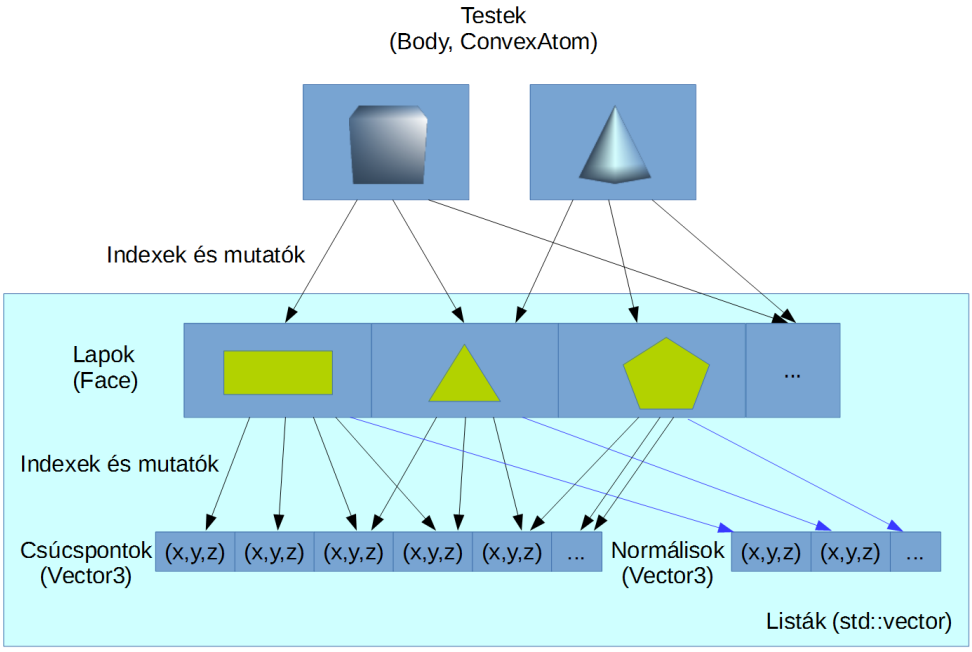
A geoios.h elkülönül a többi fájltól. Létezésének fő célja a fejlesztés közbeni kijelzés streamek segítségével. A használt leírási formátum az elkülönülést és a könnyű beazonosíthatóságot tartja szem előtt, végfelhasználóknak szóló üzenetek vagy fájlok

létrehozására nem ajánlott felhasználni.

A GLM csomag a conversion.h-ban kerül felhasználásra, ha esetleg nincs szükség grafikus megjelenítésre a program működése során, az approximator.h és conversion.h kihagyásával a függőség megszüntethető, illetve az approximator.h használata az APPROX_NO_CONVERSION definiálásával mellőzi a conversion.h és azt használó metódusok használatát.

A könyvtár központi típusainak kapcsolatrendszere az alábbi UML diagramon látható:





Mivel az egyes felhasználási területek különböző numerikus pontosságot követelhetnek meg, a skalár típusok mindenhol sablonparaméter formájában jelennek meg. Az algoritmusok elkészítésénél törekedtem minél kevesebb feltételt szabni ezen típusokra, általában az alapműveletek megléte elégséges, ám néhol ennél többre is szükség van, az ilyen követelményeket a dokumentációban külön jelzem. A sablonok a C++11-ben bevezetett mozgató konstruktor és értékadás felhasználásával készültek, így a függvények eredményeként kapott értékek is hatékonyan használhatóak.

A következőkben rövid példákkal, ábrákkal, metódus leírásokkal együtt szerepelnek az egyes típusok tulajdonságai és felhasználási területei. A könyvtár bemutatása alulról felfelé történik, mivel a típusok legtöbbje matematikai fogalmakat képvisel melynek definíciója, felhasználása szintén egymásra épít, ezért a dokumentációban is ezt az irányt követem.

A fontosabb típusok használatát bemutató példa-teszt forráskódok megtalálhatóak a könyvtár examples mappájában. Ezek olyan rövid, parancssori programok, melyek előre megadott adatokon végeznek valamilyen számítást, azt pedig a standard outputra írják. Használatukhoz nem szükséges a GLM csomag, csupán a könyvtár include mappájának elérési útját kell megadni fordításukhoz.

Jelölések, rövidítések

A leírások során többször élek rövidítésekkel, jelölésekkel, ezek definíciói alább olvashatóak.

CW	Clockwise, óramutató járásával megegyező körüljárású felsorolás.
CCW	Counter Clockwise, óramutató járásával ellentétes körüljárású felsorolás.
O(f)	Az f valós-valós függvény aszimptotikus felső becslése, műveletigény megadására.
manifold	TODO

Vektorok

A számítások során két- illetve háromdimenziós valós vektorterekben dolgozunk. A vektorokat megvalósító sablonok a `vectors.h` fejállományban találhatóak Vector2 és Vector3 néven. A vektorokat koordinátáik írják le, ezeket név szerint érhetjük el. Mindkét sablon a használt skalár típussal paraméterezendő. Mivel a program során az eltérő dimenzióbeli vektorok eltérő feladatot látnak el, teljesen elkülönülnek egymástól. A vektorok rendelkeznek összeadás, kivonás, skalárral szorzás és osztás, ellentett képzés műveletekkel, egyenlőségvizsgálattal, valamint skalárszorzás külső művelettel. Három dimenzióban a keresztszorzat is értelmezett. A `float` és `double` típusokkal példányosított sablonok `Vector2f, Vector2d, Vector3f, Vector3d` néven megtalálhatóak.

```
approx::Vector3<float> v1(1.0f,2.0f,3.0f); //az (1,2,3) vektor
approx::Vector3<float> nullvect; //default inicializációval
nullvektor
approx::Vector3<float> v2(-0.7f,2.43f,3.3f);
approx::Vector3<float> v3 = v1 + v2, v4=v1 - v2, v5 = -5.0f *
v2;
approx::Vector3<float> v6 = cross(v1, v2); //keresztszorzat
v3+=v5; v4-=v3; v1*=2.0f;
float skalarszorzat = dot(v1,v2);
```

A skalárszorzás és keresztszorzás segítségével a közbezárt szögek szinusz és koszinusz szögfüggvényeit is kiszámolhatjuk. A `length` metódus segítségével Euklideszi normát számolhatunk, a `normalize` metódus helyben normál, a `normalized` metódus pedig kiszámítja az egységhosszú azonos irányú vektort.

```
float hossz = v1.length();
v1.normalize(); v6 = v2.normalized();
float sin1 = sin(v1,v2), cos1 = cos(v1,v2); //bezárt szögekre
értelmezve
float sin2 = sin(v1,v2,v3), cos(v1,v2,v3); //a különbségekre
nézve
```

A fenti példák Vector2 sablonokra analóg módon átültetve működnek, kivéve a sin

függvényeket, mivel a keresztszorzat nincs értelmezve két dimenzióban.

Síkok

A síkokkal kapcsolatos sablonok a planes.h fejállományban találhatóak. A háromdimenziós síkok és két dimenziós egyenesek ugyanazt az alap feladatkört látják el, így közös szülőosztályuk a HyperPlane sablon, melynek metódusait az alábbiakban részletezem.

`HyperPlane(Vector normal, ScalarType distance)`	
	Megadható normálissal és előjeles távolsággal az origótól.
`HyperPlane(Vector normal, Vector point)`	
	Megadható normálissal és ráeső ponttal
`HyperPlane()`	
	Érvénytelen síkot létrehozó default konstruktor
`HyperPlane(const Hyperplane&)`	
	Másoló konstruktor.
`T signed_distance() const`	
	Az origótól mért előjeles távolság.
`Vector normal() const`	
	Egységnyi hosszú normálvektor.
`Vector example_point() const`	
	A normális meghosszabbításának metszéspontja a síkkal.
`ScalarType classify_point(Vector point) const`	
	Előjeles távolság a ponttól, az előjel használható a tér pontjainak csoportosítására.
`ScalarType distance(Vector point) const`	
	A megadott ponttól vett távolság, kettes normában véve.
`bool valid() const`	
	Érvényes-e a sík, az érvénytelen sík normálisa a nullvektor, origótól mért távolsága a skalártípus default értéke.

A két dimenziós egyenesek ezekkel a metódusokkal eleget is tettek a követelményeknek, így a Line valójában egy template megfeleltetés, azonban egy háromdimenziós síknak további feladatokat is el kell látnia, ezen kívül bizonyos előfeltétel egyszerűsítések végett a háromdimenziós síkokból nem lehet default konstruktorral érvénytelent létrehozni. A Plane sablon további metódusai:

- `std::pair<Vector3<T>,Vector3<T>> ortho2d() const`
  Vektorpár, melyek tengelyként használhatóak fel a saját koordináta-rendszerhez. A vektorok a numerikus hiba csökkentése érdekében Fábián Gábor koordináta kiválasztó módszerével készülnek, elkerülve a túl kicsi számokkal való osztást.
- `Line<T> intersection_line(plane) const`
  Egy másik síkkal való metszés során kapott egyenes, a sík saját koordináta-rendszerében felírva.

Sokszögek

A két- és háromdimenziós sokszögek egymástól eltérő elvárásokkal lettek megtervezve. A háromdimenziós sokszögeket reprezentáló `Face` sablon a Fábián Gábor és Gergó Lajos cikkeiben felvázolt approximációs adatstruktúrában előforduló lapokat hivatott reprezentálni, csúcspontjaira mutatóval és indexekkel hivatkozik egy külső tárolóba, így a csúcsmegosztás explicit módon történik. Ezen kívül a háromdimenziós számítások során a lapoknak irányítása is létezik, normálisuk irányába néznek, így többek egyszerű sokszögnél. Kettő dimenzióban a sokszögek sokszor ideiglenesen vagy egyedi úton, eltérő koordináta rendszerekben jönnek létre, a csúcsmegosztás nem szerepel a követelmények között, gyakorlatban csak csökkentené a hatékonyságot. A két dimenziós `Polygon2` sablon önmagában alkot egy egészt, nem igényel külső tárolót. Mindkét esetben egyezik azonban, hogy a sokszögeket pontjaik felsorolásával adjuk meg, tetszőleges körüljárási iránnyal, egyetlen törött vonallal határolt, nem önmetsző sokszögeket támogatva.

Polygon2

A poly2.h-ban található `Polygon2` sablon paramétere pontjainak skalártípusa. A skalárra a `Vector2` feltételei mellett megkötés, hogy értelmezhetőnek kell lennie az `abs(x)` és `x > 0` műveleteknek. Az alábbi metódusokon kívül támogatja az egyenlőségvizsgálatot és értékadást is, két sokszöget akkor tartok egyenlőnek, ha pontjaik sorra megegyeznek, így ugyanazok a pontok más sorrendben megadva nem lesznek egyenlőek. Bár pontjai külön-külön módosíthatóak, az osztály metódusai konstansként való felhasználásra készültek, mellékhatások nélkül végeznek számításokat, így függvény orientált megközelítésű feladatmegoldásra is használhatóak.

A Polygon2 metódusai:

`Polygon2(const std::vector<Vector2<T>>&)` `Polygon2(std::vector<Vector2<T>>&&)`	
	Konstruktor a pontok vektorban megadott sorozatával. A körüljárási irány tetszőleges, az algoritmusok igazodnak hozzá.
`Polygon2(const Polygon2&)` `Polygon2(Polygon2&&)`	
	Másoló konstruktor és mozgató konstruktor.
`template <class Iter> Polygon2(Iter beg, Iter end)`	
	Pontokra mutató iterátor tartománnyal is megadhatjuk a sokszöget. A C++ hagyományok szerint balról zárt, jobbról nyitott tartományt várunk.
`Iterator begin()` `ConstIterator begin() const`	
	Konstans vagy nem konstans elérésű, első pontra mutató iterátor.
`Iterator end()` `ConstIterator end() const`	
	Konstans vagy nem konstans elérésű, utolsó utáni pontra mutató iterátor.
`int size() const`	
	A sokszög csúcspontjainak száma.
`const std::vector<Vector2<T>>& points() const`	
	A pontok egy felsorolva egy vektorban.
`const Vector2<T>& points(size_t i) const` `Vector2<T>& points(size_t i)`	
	Az i. csúcspont, konstans vagy módosítható eléréssel.
`T signed_area() const`	
	Előjeles terület, CCW felsorolásban negatív. *O(size())* műveletigény.
`T area() const`	
	Terület. O(size()) műveletigény.
`bool is_ccw() const`	
	Eldönti, hogy a pontok CCW felsorolásban vannak-e megadva. O(size()) műveletigény. Működik konvex és konkáv esetekben is.
`Polygon2<T>::CutResult cut_by(const Line<T>&) const`	
	A sokszög kettévágása a megadott vonallal, eredménye a két fél. *O(size())* műveletigény. Az eredmény negative és positive adattagjai a vágó egyenes szempontjából vannak besorolva.
`bool contains(const Vector2<T>& point) const`	

	Eldönti, hogy a pont benne van-e a sokszögben. *O(size())* műveletigény.
`bool contains(const Polygon2<T>& poly) const`	
	Eldönti, hogy a másik sokszög benne van-e ebben a sokszögben. *O(size()*poly.size())* műveletigény.
`bool contains(const Vector2<T>& poly) const`	
	Eldönti, hogy a pont benne van-e ebben a sokszögben. *O(size()*poly.size())* műveletigény.
`Polygon2<T> convex_clip(const Polygon2<T>&) const`	
	A másik sokszög körbevágása ezzel, a Sutherland-Hodgman algoritmus szerint. Előfeltétel hogy ez a sokszög konvex legyen. O(size()*poly.size()) műveletigény.
`Vector2<T> centroid() const`	
	Kiszámolja a sokszög súlypontját. *O(size())* műveletigény.

Face

A face.h-ban található Face sablon sablonparamétere a pontvektoraiban felhasznált skalár típus. A `Vector3` sablon feltételei mellett, értelmezhetőnek kell lennie az `x > 0` és `x < 0` műveleteknek. A lapok egy tömör test határoló lapjait írják le, így fontos szerephez jut normál vektoruk iránya, melynek a testből kifelé kell mutatnia. A szakirodalom hangsúlyozza annak fontosságát, hogy a normálisokat ne a pontokból visszanyerve kapjuk, hanem a bemenetként kapott értékeket használjuk később is, ezért a lapok explicit megadott normálisokkal dolgoznak, ha arra van igény képesek a pontjaikból is meghatározni azt konstruktorukban. Nem megkövetelt a lapok között is állandó körüljárási irány használata, a metódusok működnek CW és CCW irányítású felsorolásban is, grafikai felhasználásra a későbbiekben látható konverziós függvények elvégzik a háromszögelést és irányba forgatást, így lapeldobást alkalmazó rendszerekben is helyesen ábrázolódnak lapjaink. A lapok alkalmazhatóak tényleges lapként és indexsorozatként is, ahogy a feladat megköveteli. Az algoritmusokban fontos szerephez jutó vágásoknál a numerikus pontosság nagy jelentőséggel bír. Az osztály garantálja, hogy a szomszédos lapok oldalain ejtett vágások által keletkezett csúcs pontok egyezni fognak. Bár a lapok képesek konkáv alakzatokat is leírni, egyes algoritmusoknál, például a vágásnál előfeltétel a konvexitás, így nem ajánlott konkáv lapok létrehozása.

A vágások megvalósítása a felszínen funkcionális megközelítésűen zajlik, azaz a vágás

kiszámolja a keletkező feleket az eredeti sokszöget módosítatlanul hagyva, azonban a felszín alatt a tároló vektorokba bekerülnek az új pontok is, így valójában itt mellékhatásokkal dolgozunk, a lapok szempontjából azonban ez nem jelent változást, mivel indexekkel hivatkoznak pontjaikra. A vágási eredmény tartalmaz minden olyan adatot is, mely szükséges a tároló állapotának visszaállítására, amennyiben a vágásokat visszavonnánk. Amennyiben a pontokat tartalmazó tárolók változatlanságát is megköveteljük, a vágást végrehajthatjuk független tárolóba beszúrva is, az összefüggő sokszögek vágásánál pedig a pontismétlődést elkerülendő használhatunk buffert, mely a korábban beszúrt pontokat hatékonyan képes előkeresni. A keletkezett lapok besorolása a vágósík szempontjából negatív és pozitív féltérbe esőként van megadva, ahol a pozitív térfél az melybe a síkról a normálvektor irányába elmozdulva jutunk.

A Face metódusai:

`Face(container,ids,normalcontainter, normalid)`
`Face(container,ids,normalcontainter, ccw)`
`Face(container,idbeg,idend,normalcontainter, normalid)`	

	• container – pontokat tároló vektor • ids – int vektor mely a pontokra hivatkozik • normalcontainer – a normálvektorokat tároló vektor • ccw – ha igaz akkor a számol normálvektor jobbkézrendszerben, ha hamis akkor balkézrendszerben számolódik (a számolt normális csak konvex, egyenesszög nélküli) • idbeg,idend – iterátorokkal megadott indextartomány • normalid – az előre megadott normális indexe
`const std::vector<int>& indicies() const` `std::vector<int>& indicies()`	
	Az pont indexvektor kinyerése.
`int indicies(ind) const`	
	Az adott sorszámú index kinyerése.
`int normal_index() const` `int& normal_index()`	
	A normálvektor tárolóbeli indexe.
`size_t size() const`	
	A csúcspontok száma.

`std::vector<Vector3<T>>* vertex_container() const`	
	Mutató a ponttárolóra.
`std::vector<Vector3<T>>* normal_container() const`	
	Mutató a normálistárolóra.
`const Vector3<T>& normal()const`	
	Normálvektor.
`const Vector3<T>& points(int) const`	
	Az adott indexű csúcspont.
`Plane<T> to_plane() const`	
	A sík megadása, melyen a lap fekszik, a normálisuk egy irányba mutat.
`Face<T>::VertexIterator begin() const`	
	Konstans elérést biztosító iterátor az első csúcspontra
`Face<T>::VertexIterator end() const`	
	Konstans elérést biztosító iterátor az utolsó utáni csúcspontra.
`Face<T> migrate_to(std::vector<Vector3<T>>* target_vecs, std::vector<Vector3<T>>* target_normals) const` `Face<T> migrate_to(std::vector<Vector3<T>>* target_vecs, std::vector<Vector3<T>>* target_normals)`	
	Konstans esetben lemásolja a lapot, de indexei a megadott tárolókra mutatnak, míg nem konstans esetben a másolatot mozgatással hozza létre. Konstans esetben *O(size())*, nem konstans esetben *O(1)* műveletigény.
`void reverse_order()`	
	Helyben megfordítja a körüljárási irányt. *O(size())* műveletigény.
`Face<T> reversed() const`	
	Létrehozza a lap ellenkező irányba álló másolatát, az ellentétes normálvektort beszúrja a tárolóba. *O(size())* műveletigény.
`Vector3<T> centroid() const`	
	Súlypont számítása. *O(size())* műveletigény.
`Polygon2<T> to_2d() const` `Polygon2<T> to_2d(const Vector3<T>& x, const Vector3<T>& y) const`	
	A lap csúcsait leképezve a sík koordináta-rendszerébe 2 dimenziós sokszöget kapunk. *O(size())* műveletigény. Amennyiben adunk paramétereket, azokat tekinti koordináta tengelynek.
`Face<T>::CutResult cut_by(const Plane<T>& p)`	

`Face<T>::CutResult cut_by(const Plane<T>&p, Map map)`
`Face<T>::CutResult cut_by(const Plane<T>&p,`
`std::vector<Vector3<T>>* tv, std::vector<Vector3<T>>* tn)`	

	Vágás adott síkkal. Feltételezi, hogy a lap konvex, konkáv esetben közel 0 területű hibák keletkezhetnek. • p – vágást végző sík • map – korábban beszúrt pontok indexeit kezelő asszociatív tároló • tv, tn – új tároló melybe az eredmény lapok kerülnek. Az első és utolsó esetben *O(size())* műveletigény, a map használata mellett *O(size()*K)*, ahol *K* a tároló beszúrási és keresési költsége.
`bool is_ccw() const`	
	Pontosan akkor igaz, ha CCW körüljárási irányban megadott lapjaink vannak. *O(size())* műveletigény.
`bool insert_index(int i1, int i2, int ind)`	
	Amennyiben az i1 és i2 szomszédos csúcsok indexei a lapon, közéjük illeszti az ind indexűt, különben nem tesz semmit. Eredménye pontosan akkor igaz, ha történt beszúrás. *O(size())* műveletigény. Létezésének oka a szemközti atomok vágásánál keletkező lapcserék mellett a test tulajdonságainak megtartása.
`std::pair<int,int> neighbours_of(ind) const`	
	Ha megtalálja az adott tároló beli indexű pontot, megadja szomszédai indexeit, egyébként (-1,-1)-et. *O(size())* műveletigény.

Testek

A testek szintje az, ahol az approximációs algoritmusok nagy része dolgozik. A testekről feltesszük, hogy homogén, konvex sokszög lapokkal burkolt, melyek lapnormálisai kifelé mutatnak, valamint pontjaikra teljesül a manifold tulajdonság. A `body.h-ban` található `Body` sablon osztály indexhivatkozásokkal felépítve képes leírni ilyen testeket, valamint számításokat végezni rajtuk. Az approximáció során konvex testekkel dolgozunk, melyeket továbbiakban atomnak nevezek. Az atomokat reprezentáló `ConvexAtom` sablon osztály a `convexatom.h`-ban található. Az atomokon konvexitásuk miatt több műveletet definiálhatunk mint az átlagos testeken, de a testekre értelmes műveletek rájuk is értelmesek, így az objektum orientált megközelítés szerint az atomokat a testekből származtattam.

Body

A `Body` osztály indexhivatkozásokkal éri el lapjait egy külső laptárolóból, önmagában pusztán egy burkoló, mely képes testként kezelni a lapok sorozatát, valamint testeken definiált műveleteket elegánsan ellátni.

A Body metódusai:

`Body(std::vector<Face<T>>* faces,const std::vector<int>& inds)` `Body(std::vector<Face<T>>* faces, std::vector<int>&& inds)`	
	• faces – mutató a laptárolóra • inds – indexsorozat mely a lapokra hivatkozik Rendelkezésre állnak másoló és mozgató konstruktorok is.
`bool valid() const`	
	Pontosan akkor igaz, ha a mutatója nem nullpointer és több mint egy darab lapja van. *O(1)* műveletigény. A bool konverziós operátor is ezzel egyenlő.
`Body<T> migrate_to(std::vector<Face<T>>*) const` `Body<T> migrate_to(std::vector<Face<T>>*)`	
	Konstans esetben lemásolja a testet csak a mutatóját állítja át a megadottra, nem konstans esetben mozgatást használ. Konstans esetben *O(size())*, nem konstans esetben *O(1)* műveletigény. Befoglaló tárolók költöztetésénél használandó, mikor a tárvektorok címei változnak.
`int size() const`	
	A test lapjainak száma. O(1) műveletigény.
`const std::vector<int>& indicies() const` `std::vector<int>& indicies()`	
	Az indexvektor mely a lapokra hivatkozik.
`int indicies(i) const`	
	Az i. lap indexe a tárolóban.
`const Face<T>& faces(i) const` `Face<T>& faces(i)`	
	Az testben i. sorszámú lap.
`T volume() const`	
	A test térfogatának kiszámítása. Elsőre *O(size()*K)* műveletigény, ahol *K* a lapok átlagos csúcsszáma, *O(1)* műveletigény amennyiben korábban kiszámoltuk és nem ürítettük a cache-t .

`Vector3<T> centroid() const`	
	A súlypont kiszámítása. Elsőre *O(size()*K)* műveletigény, ahol *K* a lapok átlagos csúcsszáma, *O(1)* műveletigény amennyiben korábban kiszámoltuk és nem ürítettük a cache-t .
`bool intersects_plane(const Plane<T>& p) const` `bool intersects_plane(const Plane<T>& p, T min) const`	
	Pontosan akkor igaz, ha az adott sík átmegy a testen és mindkét oldalán találhatóak csúcspontok. A széleken fellépő numerikus hiba ellen minimális távolságot is szolgáltathatunk, amin belül egy pontot a síkon fekvőnek tekintünk.
`std::vector<std::pair<Polygon2<T>, bool>> cut_surface(const Plane<T>&) const`	
	Eredménye az a sokszögekből álló kétdimenziós kép ami az adott síkon keletkezik, ha elvágjuk vele a testet, a sík koordináta-rendszerében megadva. *O(size())* műveletigény.
`Vector3<T> diameter() const`	
	Eredménye egy maximális hosszú test- illetve lapátló vektorként megadva. Elsőre *O(size()*K)* műveletigény ahol *K* a fedlapok átlagos csúcsszáma, *O(1)* műveletigény amennyiben korábban kiszámoltuk és nem ürítettük a cache-t.
`void clear_cache() const`	
	Kiüríti a cachet, így a benne tárolt számításokat ismét el fogjuk végezni szükség esetén. *O(1)* műveletigény

ConvexAtom

Az atomok konvexitásából eredően ha síkokkal szeleteljük őket, a kapott darabok is konvexek lesznek, valamint ha metsző síkkal vágjuk el az atomot, pontosan két darab fog keletkezni. Ez a meglátás lehetővé teszi a vágás definiálását, mely az egyik kulcsfontosságú lépés az algoritmusban. Az atomok ismerik a testet, melynek közelítésére használjuk őket, így lehetőségünk van olyan számítások elvégzésére is, melyek a céltesttel való hasonlóságot mutatják meg.

A ConvexAtom metódusai:

`ConvexAtom(std::vector<Face<T>>* faces, const std::vector<int>&`

`indicies, const Body<T>* target)`
`ConvexAtom(std::vector<Face<T>>* faces, std::vector<int>&&`
`indicies, const Body<T>* target)`
`ConvexAtom(std::vector<Face<T>>* f, const std::vector<int>& i,`
`const Body<T>* target, const`
`std::vector<std::shared_ptr<SurfacePoly>>& plist)`
`ConvexAtom(std::vector<Face<T>>* f, std::vector<int>&& i, const`
`Body<T>* targ, std::vector<std::shared_ptr<SurfacePoly>>&&`
`plist)`	

	• faces – mutató a laptároló vektorra • indicies – lapokr mutó indexvektor • target – mutató a közelítendő testre • plist – lapvetület lista
`const Body<T>* target_body() const`	
	A céltestre mutató mutató.
`ConvexAtom<T>::CutResult cut_by(plane) const`	
	Elvágás az adott síkkal. Hozzávetőlegesen *O(size()*K)* műveletigény, ahol *K* a fedőlapok átlagos csúcsszáma.
`bool point_inside(const Vector3<T>&) const`	
	Pontosan akkor igaz ha a megadott pont az atomba esik. *O(size())* műveletigény.
`int target_vertex_count_inside() const`	
	A céltest atomba eső csúcspontjainak száma, minden pont pontosan egyszer van felszámolva, függetlenül attól, hány lapra csúcspontja. *O(size()*K)* műveletigény ahol *K* a fedlapok átlagos csúcsszáma a céltestben.
`T intersection_volume() const`	
	Kiszámolja a céltesttel vett metszetének térfogatát. *O((size()+size()*target_body().size()*L)*K)* műveletigény ahol *K* az atom, *L* pedig a céltest átlagos csúcsszáma.
`T fourier() const`	
	Fourier-együttható számítása, a metszet- és saját térfogatának hányadosa. Műveletigénye a `volume` és `intersection_volume` összege.
`ConvexAtom<T>::PolyPtr surf_imprints(int i) const`	

	Az i. lapra eső metszetképre mutató osztott mutató.
`Vector3<T> avg_point() const`	
	A fedlapok súlypontjainak koordinátánként vett számtani átlaga. *O(size()*K)* műveletigény ahol *K* a fedlapok átlagos csúcsszáma.
`bool replace_face_with(int ind,int ind1, int ind2, const PolyPtr& print1, const PolyPtr& print2, int pt1, int pt2)`	
	Laptörés megvalósítása, egy lapot kettő másikra cserélünk. A szomszédos lapokat is javítja. Műveletigénye hozzávetőlegesen lineáris az atom lapjainak számában. • ind – a lecserélendő lap tárolóbeli indexe • ind1,ind2 – a két új lap tárolóbeli indexe • print1,print2 – a laplenyomatok osztott mutatói • pt1,pt2 – a lapvágásnál keletkezett két csúcs tárolóbeli indexe
`int bad_normal_ind() const`	
	Ha egy művelet rossz lapot hozna létre és az egyik fedlap normálisa fordított irányú, ez a metódus visszatér az első megtalált lap atombeli indexével, különben pedig -1-el. *O(size())* műveletigény. Fejlesztőknek öntesztelő funkció.
`std::vector<int> face_indicies_inside() const`	
	A céltest atomba eső vagy belenyúló lapjainak tárolóbeli indexeit sorolja fel. *O(size()*target().size())* műveletigény.
`std::vector<Face<T>> faces_inside() const`	
	A céltest atomba eső vagy belenyúló lapjainak listája. *O(size()*target().size())* műveletigény.

Működtető tárolók

Az algoritmusok szempontjából érdekes, matematikai fogalmakat leíró típusok működéséhez háttérben több tárolóra is szükség van, amelyek tárolják a tényleges listákat melyre a lapok és testek hivatkoznak. Fontos szempont, hogy az számításokat végző típusoknak nem feladatuk a tárolóik rendben tartása, ezért az algoritmusok haladásával a régi eldobott elemek szemétként felgyűlhetnek. Az approximációt felügyelő Approximation és a céltestet tároló Targetbody sablonok ezekkel a megfontolásokkal születtek. A gördülékeny használat érdekében az Approximator osztály egy sor kényelmi függvényt biztosít, így a fejlesztők tényleg a feladat szintjén

gondolkozhatnak.

Approximation

Ez a tároló felügyeli az atomokat. Képes szemétgyűjtést szolgáltatni, azonban ezt csak explicit utasításra teszi meg, mivel minden vágás után elvégezni a műveletet nagyban lassítja a működést, így a sűrűséget az üzemeltetőre bízzuk. Ha elvégeztük egy atom vágását, az eredményt kezdetben tanulmányozhatjuk, majd döntésünktől függően elfogadhatjuk, vagy semmissé tehetjük, ezeket a lentebb ismertetett CutResult osztály segítségével hajthatjuk végre.

A üreget tartalmazó test közelítésénél, illetve speciális törlési kérések után keletkezhet olyan üres régió, ahol a külvilággal nem érintkező fedlapok vannak. Ezeket a lapokat nevezzük belső lapoknak. A belső lapok lehetnek lényegtelenek, szükség lehet megőrzésükre, illetve fordított irányú megfelelőik képezhetik a test miniatűr mását is. Ezeket az opciókat sorra az InsideHandling::Leaveout, InsideHandling::Addinside, InsideHandling::FlipInside enumok jelzik a számítás számára.

Az Approximation metódusai:

`Approximation(const TargetBody<T>* target, T border)`	
	target – mutató a céltestre border – a kezdőatom ekkora kerettel veszi körbe a testet

`Approximation<T>::Iterator begin()`
`Approximation<T>::Iterator end()`
`Approximation<T>::ConstIterator begin() const`
`Approximation<T>::ConstIterator end() const`	

	Iterátorok az atomok bejárásához.
`const std::vector<Vector3<T>>& vertex_container() const`	
	Hivatkozás a csúcspont tárolóra.
`const std::vector<Vector3<T>>& normal_container() const`	
	Hivatkozás a normálvektor tárolóra.
`const std::vector<Face<T>>& face_container() const`	
	Hivatkozás a lap tárolóra.
`const TargetBody<T>& target_body() const`	
	Hivatkozás a céltestre.
`int size() const`	

	Az atomok száma a tárolóban.
`Approximation<T>::CutResult cut(int ind,const Plane<T>& p)` `Approximation<T>::CutResult cut(Iter iter, const Plane<T>& p)`	
	Megadott atom elvágása az adott síkkal. Ha van elfogadatlan vágásunk, az automatikusan visszavonódik ez előtt. • p – vágósík • ind – a választott atom indexe • iter – a választott atomra mutató iterátor
`bool pending() const`	
	Pontosan akkor igaz, ha van elfogadatlan vágásunk.
`void garbage_collection()`	
	Szemétgyűjtés végrehajtása, eltünteti az olyan pontokat, lapokat, és normálisokat, melyekre nem hivatkozik atom. Elfogadatlan vágás esetén nem csinál semmit.
`Approximation<T>::CutResult last_cut_result()`	
	CutResult mely az utolsó elfogadatlan vágásra hivatkozik, ezzel az objektummal ellenőrizhetjük, fogadhatjuk el, vagy explicit visszavonhatjuk a vágást.
`Body<T> approximated_body(InsideHandling mode, T fmin)`	
	Az approximáció eredmény testének kiszámítása. Automatikusan végrehajt egy szemétgyűjtést. A kapott test érvénytelenné válhat későbbi szemétgyűjtés után. • mode – az esetleges belső oldalak kezelési módja, alapértelmezetten kihagyás. • fmin – a minimális, bevételhez szükséges fourier együttható, alapértelmezetten 0,5.
`void final_transform()`	
	Amennyiben a céltesten transzformációkat hajtottunk végre a betöltésnél, ez a metódus végrehajtja az ellenkezőjét az atomokon.

A vágási eredményeket kezelő `CutResult` osztály végzi a műveletek ellenőrzését és véglegesítését. A kapott atomok közül kiválaszthatjuk melyiket tartjuk meg, ezzel eltávolítva a felesleges atomokat. Az osztály igyekszik megvédeni tárolót a hibás atomok keletkezésétől, így ha hibás atomot próbálunk meg elfogadni, visszavonja a műveletet és hamis visszatérési értékkel jelzi a sikertelenséget. Ha a vágást elfogadjuk,

az eredeti atom törlődik a tárolóból.

A `CutResult` metódusai:

`const ConvexAtom<T>* positive() const`	
	A pozitív oldali eredmény atom.
`const ConvexAtom<T>* negative() const`	
	A negatív oldali eredmény atom.
`bool choose_both()`	
	Mindkét eredmény atom beszúrása a tárolóba, igaz ha sikeres, hamis különben.
`bool choose_negative()`	
	Csak a negatív rész beszúrása, igaz ha sikeres, hamis különben.
`bool choose_positive()`	
	Csak a pozitív rész beszúrása, igaz ha sikeres, hamis különben.
`void undo()`	
	Művelet visszavonása.

TargetBody

A közelítendő testet tároló sablon osztály a targetbody.h-ban található TargetBody. Paramétere a használandó skalártípus. Külön kérésre, a test megadott méretűre alakítható és az origóba transzformálható, ezzel kezelhetőbbé téve a testet.

`Const std::vector<Vector3<T>>& vertex_container() const`	
	Hivatkozás a csúcspont tárolóra.
`Const std::vector<Vector3<T>>& normal_container() const`	
	Hivatkozás a normálvektor tárolóra.
`Const std::vector<Face<T>>& face_container() const`	
	Hivatkozás a laptárolóra.
`const Body<T> body() const`	
	Hivatkozás a testre.
`void transform_to_origo(T size)`	
	A súlypont eltolása az origóba és átméretezés úgy, hogy a test befoglaló dobozának legnagyobb élhossza a megadott legyen. Ha nem pozitív méretet adunk meg, méretezés nem történik, alapértelemzett értékként -1 lesz megadva.
`void transform_back()`	

	Az átalakítást semmissé tesszük.
`T inverse_scale() const`	
	Az átméretezés arányának reciproka.
`Vector3<T> inverse_transform() const`	
	Az origóba toló fektor ellentettje.

Approximator

Az Approximation és TargetBody gördülékeny használatához, valamint ki- és bemeneti műveletekkel való könnyű kombinálásához áll rendelkezésre az Approximator osztály, mely képes felügyelni az approximáció minden kellékét, valamint elvégezni a kellő lépéseket. Alapértelmezésben a típus kényelmi függvényeivel lehetőséget ad a grafikus megjelenítéshez konvertálásra, azonban, ha a `APPROX_NO_CONVERSION` makró definiált a fejállomány fordításakor, a feltételes fordítással kizáródnak ezek a metódusok, ezzel a GLM csomag használatát is elkerülhetjük, lemondva a nyújtott megjelenítési típusokról.

Az Approximator metódusai:

`Approximator()`
`Approximator(std::string targetfile, T border, T cube, bool triangulate)`
`Approximator(const TargetBody<T>* target, T border, T cube)`	

	A konstruktor paramétere a céltest, vagy azt tartalmazó fájl, a keretvastagság, illetve a fájlból való betöltésnél kényszeríthetjük a háromszögeltetést.
`bool valid() const`	
	Pontosan akkor igaz, ha kész ellátni a feladatot.
`Approximation<T>& container()`	
	Hivatkozás a belső Approximation objektumra.
`Const TargetBody<T>& target() const`	
	Hivatkozás a céltestre.
`void restart()`	
	Az approximáció kezdőállapotba hozása egyező kezdő feltételekkel.
`bool set_target(const std::string& file, T border, T epsilon, T cube, bool triangulate)` `bool set_target(std::unique_ptr<TargetBody<T>>&&`	

`target,border,cube)`	
	• file – a céltestet tartalmazó obj fájl • target – előre elkészített TargetBody • border – a kezdőatom keretvastagsága • epsilon – pontkorrekciós sugár beolvasásnál • cube – ha megadjuk a targetbodyt ekkora kockába mozgatjuk az origóba • triangulate – a nem háromszögelt test háromszögelése Az approximáció felkészítése az adott célra.
`void save_atoms(const std::string& fájl) const`	
	Az összes atom kimentése egy fájlba.
`void save_approximated_body(const std::string& fájl) const`	
	A céltest kiszámolása és fájlba mentése.
`BodyList atom_drawinfo() const`	
	Az összes atom rajzolási adatai közös bufferbe való feltöltéshez.
`BodyList target_drawinfo() const`	
	A céltest rajzolási adatai.
`BodyList approx_drawinfo() const`	
	A számított test rajzolási adatai.
`BodyList cut_drawinfo() const`	
	El nem fogadott vágásnál az atomok rajzadatai, ha nincs elfogadatlan vágásunk akkor nem definiált működés.
`std::vector<PolyFace2D> atom2dfaces(int ind) const`	
	Az adott indexű atom lap metszetképei rajzoláshoz.

Input, output

A valós felhasználásnál a közelítendő testet fájlból nyerjük, az eredményt pedig grafikusan megjelenítve vagy fájlba exportálva szeretnénk látni. A könyvtár lehetőséget ad a Wavefront Technologies által készített „.obj” kiterjesztésű nyílt fájlformátum kezelésére, mind kimenet mind bemenet során. A grafikai megjelenítéshez a GLM könyvtár típusai segítségével konverziós függvényekkel képes háromszögelt, CCW irányítású, indexelt lapsorozatokat előállítani.

Fájlkezelés

Az obj fájlformátumot rugalmassága, könnyű kezelhetőség és elterjedtsége miatt

választottam támogatott fájlformátumnak. A formátum számos lehetősége közül a könyvtár csak az alapvetőeket használja fel. A fájlkezelés megvalósítása az objio.h-ban található. Az olvasást az ObjectLoader az írást az ObjectWriter végzi. Ezek az osztályok elrejtik a segédfüggvényeket és csak a szükségeseket mutatják publikusan. A csúcspontokban külön irányba mutató normálvektorok megengedettek a fájlformátumban ám approximációs szempontból egy laphoz pontosan egy normálvektor tartozik, így csak az irányítás biztosítására használom fel a megadott normálisokat, a használtakat úgy számítom, hogy biztosan megfelelőek legyenek. Normálisok jelenlétének hiányában a fájlformátum alapértelmezetten CCW felsorolást. Az ObjectLoader osztály a TargetBody barát osztálya, így hatékonyan képes feltölteni azt, amennyiben más fájlformátum használatát is be szeretnénk építeni a könyvtárba, ennek az osztálynak a bővítése javasolt.

Beolvasásnál kiértékelt sorfajták az alábbiak. Bármely más sorfajta figyelmen kívül hagyódik.

„v x,y,z”	Csúcspont koordinátáival megadva.
„vn x,y,z”	Normálvektor koordinátákkal megadva.
„f v/[vt]/[vn] v/[vt]/[vn] v/[vt]/[vn] ...”	Lap a pontjai indexeivel megadva. A textúra koordinátákat nem használom fel semmire, a normálvektorok állása eltérhet a lap geometriai normálisától, ezért csupán a számolt normális irányának ellenőrzésére használom fel őket.

A kimenetnél a fentieken kívül az objektumot kijelölő „o” kezdetű sor fordulhat elő.

A betöltés a következő statikus metódussal történik:

```
bool ObjectLoader<T>::load_obj(const std::string& file,
TargetBody<T>& target, T epsilon, bool triangulate)
```

A paraméterek sorra a betöltendő fájl elérése, a céltest ahova betöltjük, a pontösszevonási távolság és a háromszögelés kényszerítése. Az utolsó két paraméter elhagyható, alapértelmezetten nem történik összevonás, de a háromszögelés alapértelmezett, mivel több alkalmazás számára előfeltétel.

A fájlba mentés a következő statikus metódusokkal történik:

```
void ObjectWriter<T>::save_obj(const std::string& filename,
BodyIter first, BodyIter last)

void ObjectWriter<T>::save_obj(const std::string& filename,
const Approximation<T>& app)

void ObjectWriter<T>::save_obj(const std::string& filename,
const Body<T>& b)
```

Amennyiben csak egy testet akarunk elmenteni, a csúcs és normális tárolókon szelekció hajtódik végre, hogy csak a szükséges pontok kerüljenek a fájlba. Az approximáció vagy iterátor tartomány fájlba írásánál ez nem történik meg. Az iterátor tartományról feltételezett, hogy egy tárolóra hivatkoznak, így az első test tárolóinak pontjai kerülnek a fájlba.

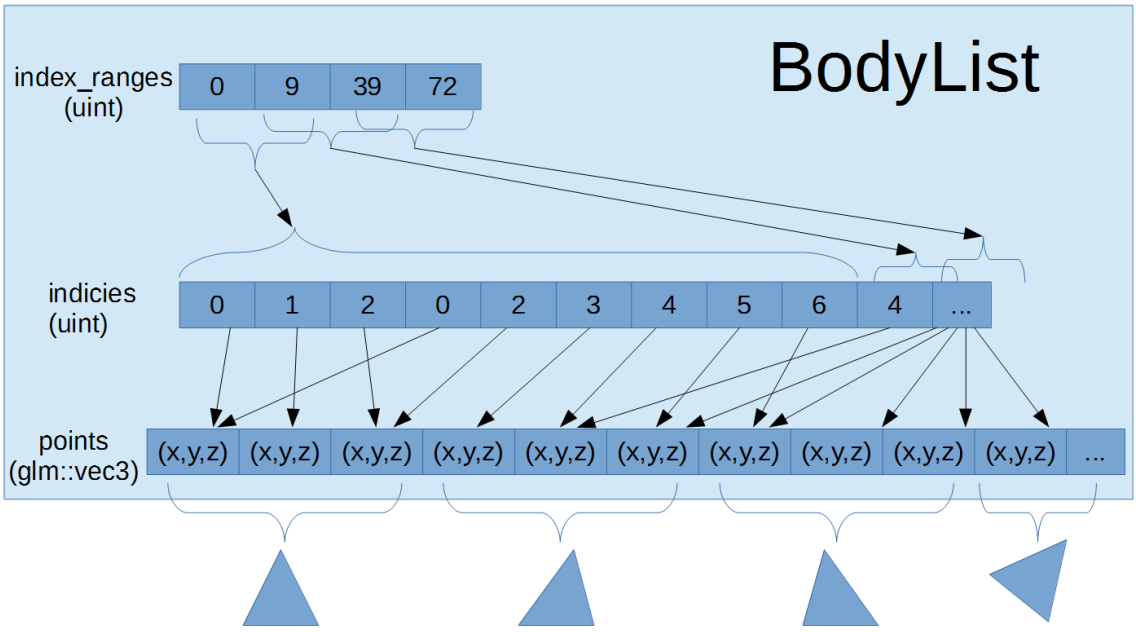
Megjelenítés

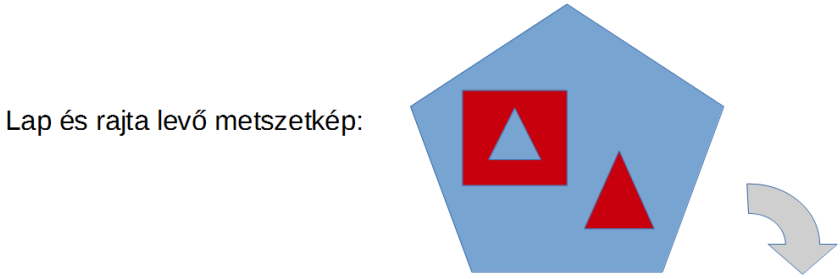
A sebességnövelés és némi helytakarékosság érdekében a számítások nem háromszögelt testeken történnek, azonban a modern grafikus könyvtáraknak háromszögelt felületekre van szükségük. Emellett elterjedt technika a hátlapeldobás, mely az irányítás alapján választja ki a megjelenítendő lapokat. Az indexelt szerkezet öröklődik a háttérszerkezetből, így a pontismétlődés kevesebb helyigénnyel jár.

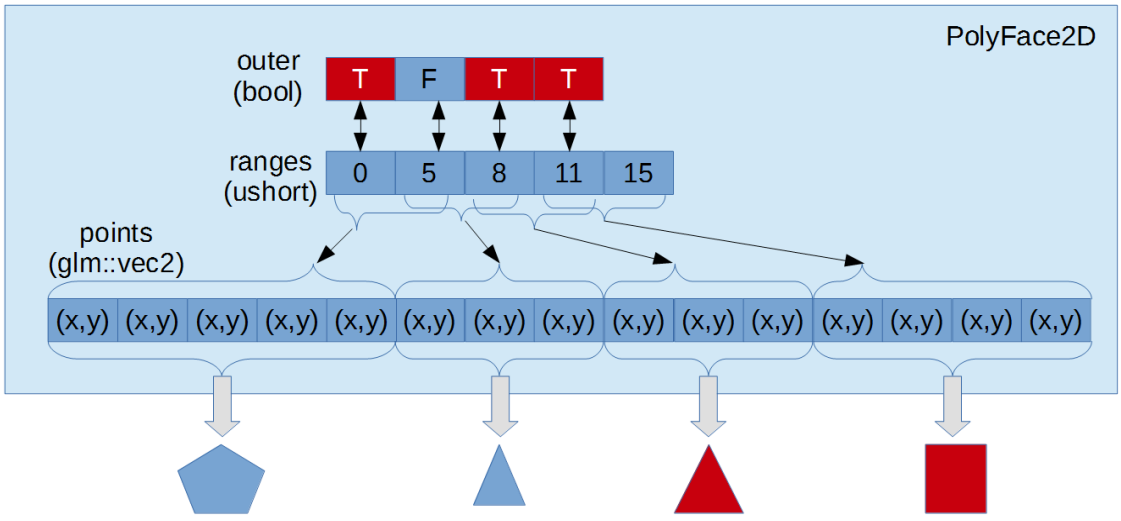
A kézi vezérlésű algoritmusok során felmerült az atom oldalain levő metszetképek külön megjelenítésének igénye. Ez kettő dimenzióban történik. Az atom lapja konvex, így háromszögelése triviális, ám a vetület, tetszőleges számú konvex, konkáv esetleg lyukas síkidomot is tartalmazhat, így háromszögelése erőforrás igényesebb, bonyolultabb módszereket igényelt volna. Témavezetőmmel való egyeztetés után a képeket körvonalakként való megjelenítésre készítem fel, így nincs szükség bonyolultabb, esetleg időigényes háromszögelő algoritmusok végrehajtására. A vonalak megkapják a jelölést miszerint egy valódi sokszöget, vagy egy kivágott lyukat írnak-e le, így eltérő színnel jelezhető szerepük.

A megjelenítendő adatokat leíró típusok a BodyList és PolyFace2D típusok úgy készültek, hogy a videokártyákra való gyors és egyszerű feltöltést lehetővé tegyék.

A fejlesztés első szakaszában, a `BodyList` `unsigned short` indextípust használt, ám ez (a használt C++ implementációkban) 16 bites, így csak 65536 indexet volt képes ábrázolni, tesztelés során pedig 2000 vágással nagyjából 137 ezer index is keletezett, így az indexek `unsigned int`-re cserélődtek.







Fejlesztői dokumentáció

Tervezési megfontolások

A könyvtár tervezése során fontos szerepet játszott a forráscikkekhez való hűség és az absztrakció, valamint feladat elkülönítés ötvözése, így az eredetileg helyenként egyszerű index listákként definiált elemek önellátó, feladatukat magukba burkoló osztályokká fejlődtek. Az objektumorientált szervezés természetesen illeszkedett a problémához, ám ahol felesleges terhet jelentett volna, ott elrugaszkodtam tőle.

Közbenső komponensek

Az alábbiakban részletezem a könyvtár összetettebb, de a korábban nem említett, fontosabbnak tekintett sablonjainak működését, felépítését, a fejlesztés során felmerülő észrevételeket.

Index iterátorok

A feladatok során többször előforduló feladat, hogy egy listában tárolt adatokra indexlistával hivatkozva leírt adatokat kell bejárni. Mivel a listákat az std::vector sablon különböző példányosításaival, az indexeket pedig std::vector<int> osztállyal reprezentáljuk, a feladatok megkönnyítésére és a C++ standard könyvtárával való felhasználásra születtek meg a saját iterátor típusok, az IndexIterator és ConstIndexIterator. Az osztály sablonok az indexiterator.h-ban találhatóak. Mindkét sablon teljesíti a C++ RandomAccessIterator feltételeit, belőlük választva konstans és nem konstans tárolók bejárására is lehetőségünk van. A könyvtár több típusa, mint a lapok és testek is ezekkel a típusokkal járhatóak be.

Javítók

Az algoritmusokban, ahol vektor folyamokat várunk bemenetként, előfordulhat felesleges ismétlődés, vagy hibás adatok. A javító típusok, az objrepair.h-ban megtalálható RepairVector és NullRepair erre a problémára nyújt megoldást.

A RepairVector a megadott metrika szerint, a megadott távolságnál közelebbi pontokat összevonja, mindig a régebbi pontot megtartva. A beérkezett vektorsorozat javított változatát megkaphatjuk egy újabb vektorként, indexhivatkozások javítását kérhetjük. A

típus minden beszúrásnál végigvizsgálja az egész tároló tartalmát, így feltöltésének műveletigénye négyzetes a pontok számában.

Amennyiben nem szükséges a hasonló pontokat összevonni, de az esetleges ismétlődéseket szeretnénk elkerülni, a NullRepair osztály gyorsabb működést garantál társánál. Belül egy másodlagos gyorsítótárat is alkalmaz, melynek segítségével a beillesztések műveletigénye logaritmikus, így a feltöltés összességében N*log(N) műveletigényű ahol N a bemenő vektorok száma.

Algoritmusok

Vágások

A testek elvágását lapok elvágására vezetem vissza. Az atomok esetében a konvexség adott, tehát borító lapjaik is konvexek. Ezen lapok elvágását a Sutherland-Hodgman algoritmus ciklusmagjának egyszeri alkalmazásával vágom el, azaz minden pontot csoportosítok aszerint, hogy a vágósík melyik oldalára esik, valamint ha a felsorolásban következő az ellenkező oldalra esik, köztük új pont keletkezik mely mindkét lapban szerepel. A felek normálisai megegyeznek az eredeti lap normálisával, így ezekre csak hivatkozási indexet kell másolni.

A lapokat szétválogatom a sík két oldalára, ám még hátravan a vágással keletkezett két szemközti lap előállítása. Ezen lapok azon pontokból állnak melyek a vágásnál keletkeznek, ám a pontok felsorolási sorrendjében nincs semmilyen szabályosság, sőt egyes pontok legalább kettőször, de speciális csúcsok esetében többször is előfordulhatnak a felsorolásban. A megoldást a Fábián-Gergó-féle cikkekben szereplő polárkoordinátás rendezés, majd pontelhagyás adja.

Az atomok egyik feladata lehet a későbbiekben részletezett metszettérfogat számítás, melyhez szükség van az atom lapjaira eső metszetképek meghatározására. Ezeket a képeket a hatékonyság érdekében a vágások alatt határozom meg. A vágás síkjára eső képet megkapva (ennek módszerét a következőkben részletezem), a Sutherand-Hodgman algoritmus segítségével kivágom a lapra eső részt. Később ezeket a képeket ugyanúgy elvágom mint a lapokat. A képek megegyeznek a szemben álló atomok egymásra néző lapjain, így osztott mutatóval tárolom őket.

Térfogat számítás

A Gauss–Osztrogradszkij-féle divergenciatételre alapozott módszerrel, a testek térfogatának kiszámításához elegendő lapjaik területeinek és normálisainak ismerete, ezekből a következő egyszerű képlettel határozhatjuk meg térfogatot:

$$V = \frac{1}{3} \sum_{1}^{n} A_i \langle a_i, n_i \rangle$$

Ahol a jelölések jelentése:

- n a test lapjainak száma
- A_i az i. lap területe
- a_i az i. lap egy tetszőleges pontja
- n_i az i. lap normál vektora

A könyvtár ezt a képletet felhasználva számol, az objektumok minden, a képlethez szükséges adatot képesek megadni.

A lapok nem tárolják saját területüket, azonban ezt pontjaik számában lineáris műveletigénnyel képesek meghatározni.

Metszet kép építés

A polygraph.h-ban található függvények a testvágás során keletkező két dimenziós kép felépítésére szolgálnak. A megoldandó probléma, hogy a vágás során keletkező irányított szakaszokból előállítsuk a sokszögeket. A keletkező sokszögek lehetnek konvexek, konkávok, sőt lyukak is lehetnek bennük, a félszakaszok irányítása nem meghatározott. A probléma megoldását a gráfelméletre vezettem vissza. A szakaszok végpontjait egy irányítatlan gráf csúcspontjainak tekintem, melyek között pontosan akkor vezet él, valamelyikből indul irányított szakasz a másikba. Ezzel az átalakítással a sokszög keresést a gráfban való körök keresésére vezetem vissza. A program által elfogadott testek tulajdonságai miatt minden pont pontosan egy körbe tartozhat bele, így a feladatot a mélységi bejárás alkalmazásával oldom meg. A kimenet sokszögek sorozata, melyek nem tartalmaznak lyukakat, az esetleges lyukak maguk is önálló sokszögekként jelennek meg, a lista feldolgozását a magasabb szintre hagyom. Az eljárás műveletigénye lineáris a csúcsok számában.

A kimenetként keletkező listát a Body megfelelő metódusában dolgozom fel. Az egyes sokszögek jelölhetik a test belsejét, vagy a benne levő lyukakat. Előfordulhat, például fodrozódó vízfelszín elvágásánál, hogy a lyukakon belül is helyezkedik el tömör terület, viszont a testek tulajdonságaiból adódóan, a határ sokszögek vagy egymástól függetlenek, vagy az egyik teljes egészében a másikon belül helyezkedik el. Ezekkel a feltételekkel a megoldási ötlet, hogy rendezzük a sokszögeket terület szerint növekvő sorrendbe, majd vizsgáljuk meg, hogy az egyes sokszögek hány náluk nagyobb belsejében találhatóak meg. Ha egy sokszög páros számú (esetleg nulla) másik belsejében található meg, akkor egy tömör terület külső határát képviseli, ha páratlan számúban, akkor egy üres területet jelöl. A sokszögek tartalmazásvizsgálatának működnie kell konvex és konkáv esetekben is, ezért a sugárkövetéses algoritmust használom a kisebb sokszög csúcspontjain.

A végleges algoritmus kimenete egy olyan címkézett sokszög lista, melynek címkézése megadja, hogy külső vagy belső határvonalat jelölünk. A kép területének kiszámítása egy egyszerű összegzés, ahol a belső felületeket leíró sokszögek területe negatívan számolódik fel.

Metszet térfogat számítás

Az atomok tulajdonságai között fontos szerepet kap a céltesttel vett metszetének térfogata. Ez a metszet maga is egy határoló lapjaival leírható test, így elvben ugyanaz az algoritmus használható rá, mint a már ismertetett térfogat számítás, azonban gyakorlatban némileg eltérő megközelítés szükséges. Mivel magára a testre nincs szükségünk, csupán lapjaira, ezért magát a testet nem állítjuk elő, lapjait egyenként határozzuk meg és használjuk fel.

A metszet lapjait két forrásból nyerjük:

- Az atom lapjaira eső metszetképek. Ezek a képek két dimenziósak, így meghatározhatjuk területüket, normálisuk pedig pontosan egyezik a lapéval amelyen fekszenek.
- A céltest atomba eső laprészei. Ezeket nem tároljuk sehol, ám meghatározásuk egyszerű. A test minden egyes lapját sorra elvágjuk az atom minden lapjával, az egyes lépések után mindig a negatív oldalt megtartva. Amennyiben megmaradt valami a lapból, annak területe és normálisa felhasználható a számításhoz.

Ezek után a metszet térfogat kiszámítása a fenti két forrás elemeinek végig iterálásából és összegzéséből áll össze.

Fejlesztés során felmerült, hogy az egyes atomokhoz hozzárendelhetnénk a céltest beléjük eső lapjait indexhivatkozásokkal, így valamivel gyorsítható lenne a számítás, ám ezek meghatározása nagyságrendileg egyező műveletigényű a metszetszámítással, a memóriaigény pedig egy lineáris szorzóval növekedne. Ezen kívül, előfordulhat, hogy egy lap csak részlegesen lóg bele az atomba, így a tárolt listán még mindig végig kellene haladni a vágó ciklussal is, nem elég csak összegezni. Egy atom metszet térfogata az algoritmus során keletkezésétől megszűnéséig állandó, így ha egyszer meghatározzuk, majd a hívási helyen elraktározzuk a számértéket, lényegi memóriaigénybeli növekedés nélkül gyorsíthatjuk a futást. Ezeket figyelembe véve az atomok nem tartják nyilván a beléjük eső lapokat.

Atomok uniója

A közelítő algoritmus egyik végcélja a kellő atomok egyesítéseként kapott test előállítása. Ez művelet testek uniójának előállítását követeli meg. A test azokból az atomokból egyesül, melyek Fourier együtthatója a megadott minimum (gyakorlatban 0,5) felett van. A művelet eredménye olyan manifold tulajdonságú test mely nem tartalmaz felesleges, nem a határán levő lapokat. Az előkészületek már az atomok vágásai során a elkezdődnek. Az atomok lapjaik mentén szomszédosak más atomokkal, ezeket a szomszédságokat a vágások során megfelelően beállítom. Az atomvágás után kapott lapok három csoportra bonthatóak.

- A vágás által érintett lapok. A lap el fog tűnni a nyilvántartásból, a szemközti lappal szemben két lap áll majd, ez felborítaná a lapszomszédsági rendszert. A megoldás a szemközti lap kettébontása a féllapokkal szemközt, így minden lappal szemben legfeljebb egy másik lap fog elhelyezkedni. Ez két új csúcsot fog eredményezni a szemközti atomban, a manifold tulajdonság megtartására a lap szomszédjaiba egyenesszögű csúcsként fel kell venni. Ezek a módosítások a környező atomok felépítését megváltoztatják, ám logikailag ugyanaz marad a leírt test.
- A vágás által érintetlen lapok. Ezek a lapok változatlanok maradnak de egy másik atomhoz fognak tartozni. A lap és szemközti párjának szomszédsági adatait felül kell írni.

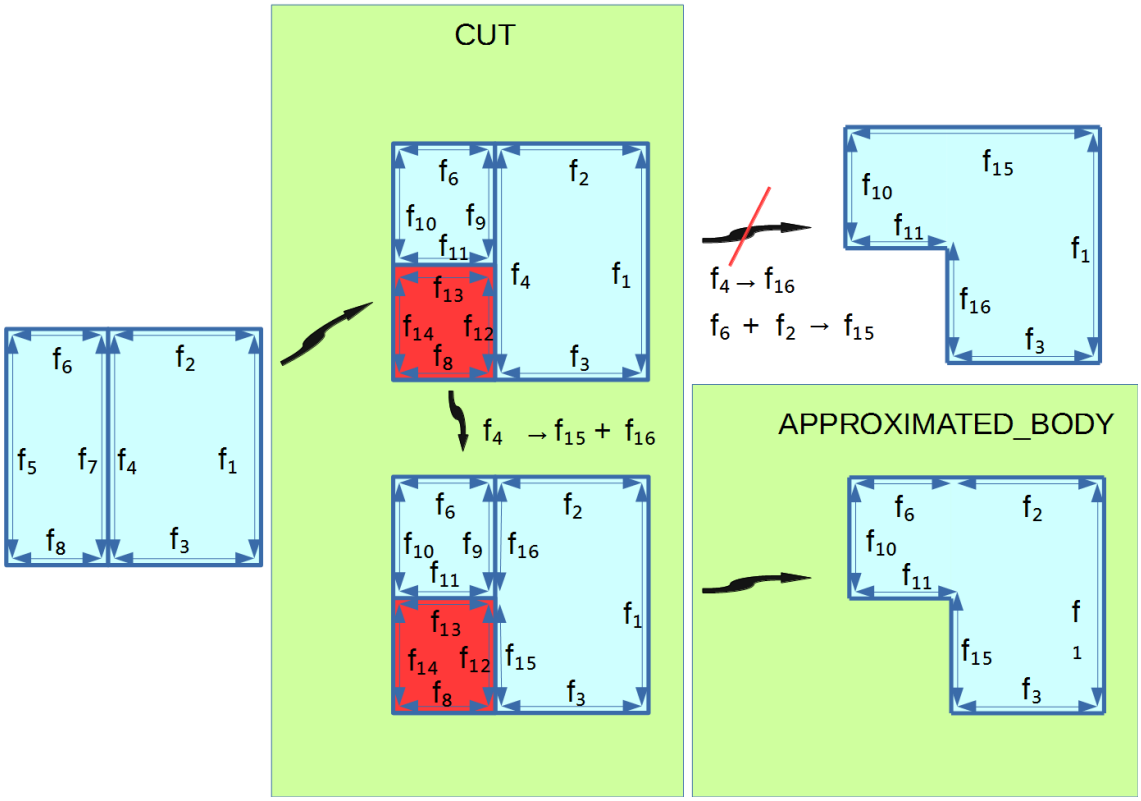
- Vágás által keletkezett lapok. A vágósíkon fekvő két egymással szembenéző lap. Ezek a keletkezett feleket kötik össze és egymás párjai lesznek.

Ezeken a csoportokon végigjárva minden vágás után megjavítom a szomszédsági viszonyokat, így a kimeneti test kiszámításához elegendő a következő algoritmust alkalmazni.

```
PROCEDURE APPROXIMATED_BODY(mode,fouriermin)
        garbage_collection();
        fouriers[0..n-1] := fourier(atoms[0..n-1])
        inds:=[]
        FOR i:=0..n-1 LOOP
                IF fouriers[i] >= fourier THEN
                        FOR idx in indicies(atoms[i]) LOOP
                                IF add(idx, mode, fouriermin) THEN
                                        inds:=[inds, idx]
                                ELIF  addflipped(idx, mode. fouriermin) THEN
                                        inds:=[inds, reversedcopy(idx)]
                                END IF
                        END LOOP
                END IF
        END LOOP
        RETURN Body(inds)
END PROCEDURE
```

Ahol a jelölések a következőket jelentik:

- add(index, mode, fouriermin): Pontosan akkor igaz, ha az adott indexű lappal szemben nincsen atom, vagy nem éri el a megadott Fourier-együttható minimumot, valamint a lap vagy külső lap, vagy belső, de a módszer AddInside.
- addflipped(idx, mode, fouriermin): Pontosan akkor igaz, ha az adott indexű lappal szemben nincsen atom, vagy nem éri el a megadott Fourier-együttható minimumot, valamint a lap vagy külső lap, vagy belső, de a módszer FlipInside.
- reversedcopy(idx): Az adott indexű lap megfordított másolata, a normálvektora ellenkező irányba áll, mint az eredetié

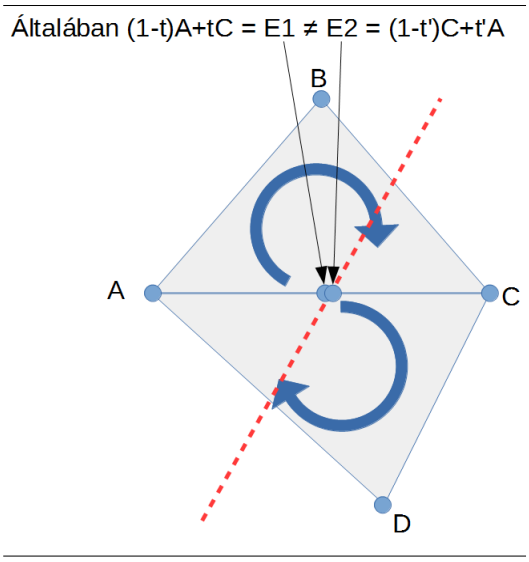


A test előállításának teljes folyamatát a fenti ábra szemlélteti. Ahogy az ábrán is megfigyelhető, az algoritmus során keletkezhetnek olyan felületek, melyeket a szükségesnél több lappal írunk le. Ezek száma azonban egy gyakorlati alkalmazás során elhanyagolhatónak tekinthető, valamint ha két lap összevonása nem szükségszerűen konvex, a konkáv lapok pedig ellenkeznek az alkalmazás megkötéseivel. Figyelembe véve továbbá, hogy a kimenő fájl valószínűsíthetően háromszögelt lesz, az algoritmus eltekint ezen felületek felkutatásától és összevonásától.

Tesztelés

A tesztelés során észlelt hibák legtöbbje a numerikus hibákból, valamint az indexelési szintek közötti zavarokból eredt. A hibakeresést nehezítette, hogy egyes hibák csak vizuálisan megjelenítve, mások pedig csak statisztikákat készítve kerültek elő.

Az egyik megközelítésbeli ötlet az volt, hogy a numerikus probléma elkerülésére a program lehetőleg az egyezőnek tekintett adatokat egy helyről kapja minden alkalommal, így a hiba is mindig egyezni fog. Ezt a megkötést sikerült tartani, ám először hibaforrás volt, hogy két szomszédos lapon azonos csúcsok közötti él más irányítással fordul elő, így esetleg ugyanazzal a síkkal vágva nem ugyanazt a pontot adja. Az alábbi ábra szemlélteti.



Több algoritmus során merült fel a sokszögek csúcspontjainak speciális esetként kezelésének szüksége. Ha egy sokszöget pontosan a csúcsponton keresztülmenő síkkal vágunk el, ne szúrjunk be új pontot. Ha pont tartalmazást vizsgálunk sugárkövetéses algoritmussal, a csúcsponton átmenő sugár esetét külön kell kezelni, különben kétszer számolhatunk fel egy határvonalat. A tesztelés során többek között az említett esetekben is, az eredeti algoritmusokat kisebb módosításoknak kellett alávetni a megfelelő működés érdekében.

Irodalomjegyzék

TODO (pl meshapproxextended cikk hogy és mint)

http://wwwf.imperial.ac.uk/~rn/centroid.pdf Elérés dátuma: 2016.03.10

Fábián Gábor, Gergó Lajos, Adaptive algorithm for polyhedral approximation of 3D solids. Stud. Univ. Babes-Bolyai Math. Vol. 60, No. 2, 2015. pp 27591.

Fábián Gábor, Gergó Lajos, Fast Algorithm to Split and Reconstruct Triangular Meshes, Studia Universitatis Babes-Bolyai Series Informatica, Special Issue 1, pp. 90-102 (2014)

Christer Ericson: Real Time Collision Detection, Elsevier Inc., 2005, 1-55860-732-3

Sherif Ghali: Introduction to Geometric Computing, Springer-Verlag London Limited, 2008, 978-1-84800-114-5