

Bevezetés

Napjainkban a számítógépes térgeometria és grafika számos, folyamatosan fejlődő és terjedő területen, 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 legtöbbször 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 particionáló fák felépítése, majd az ezen fák levelein fekvő testeknek (továbbiakban atomok) felhasználása. A módszer egy speciális megközelítést, egy térfogat alapú metrikát használó, bizonyítottan konvergens, tér particionáló, iteratív algoritmust részletesen vázolnak Fábián Gábor és Gergő Lajos cikkei[1][2].

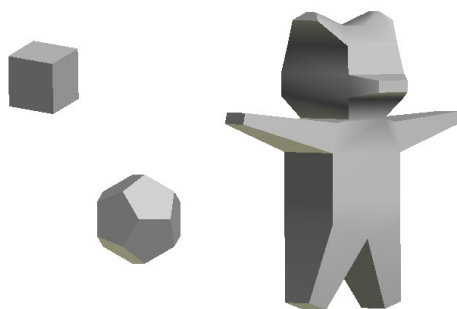
Ezen szakdolgozatban kitűzött feladat, olyan program elkészítése, mely a fent említett cikkek által felvázolt elméleti eredményeket, módszereket és adatszerkezetet képes a gyakorlatban bemutatni, ezzel segítve a további kutatásokat, illetve szemléletesebbé, közhírosabbá tenni azokat. A programnak képesnek 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.

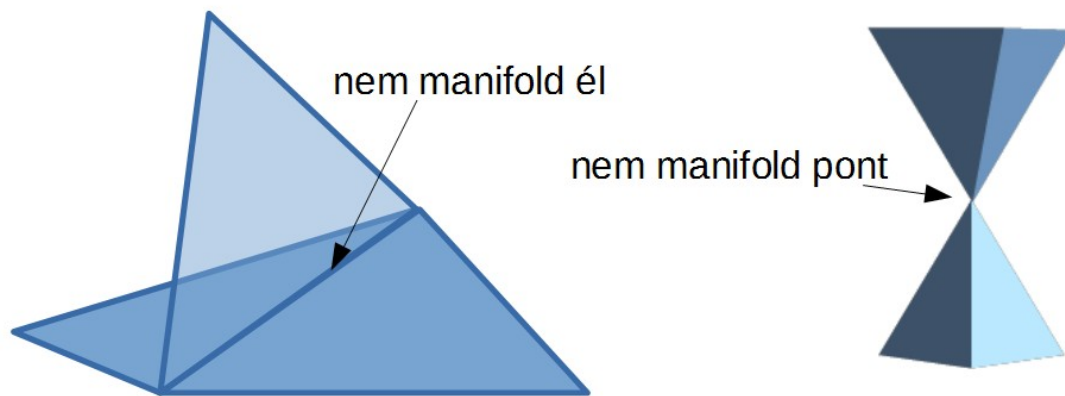
Fejlesztési terv

Definíciók, megkötések

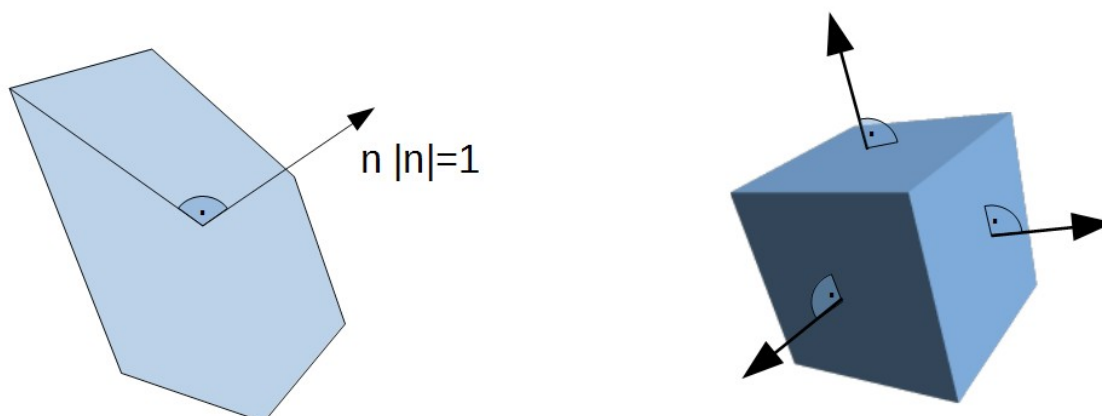
A konvex poliéderek a háromdimenziós tér olyan részhalmazai, melyek véges sok, síkkal határolt féltér metszeteként állnak elő, a poliéderek pedig véges sok konvex poliéder uniójaként állnak elő. A programban korlátos poliéderekkel foglalkozunk, melyeknek létezik köré írható gömbje, továbbá megköveteljük, hogy a határoló lapok konvex sokszögekké legyenek tagolva. A konvex poliédereket a forráscikkek[1][2] alapján a továbbiakban atomoknak is nevezhetjük, a poliéderekre hivatkozunk testekként, a határoló lapokat lapokként, borító lapokként, vagy fedlapokként emlegetjük. Ha a fedlapok mind háromszögek, a testre háromszögelt testként hivatkozunk.



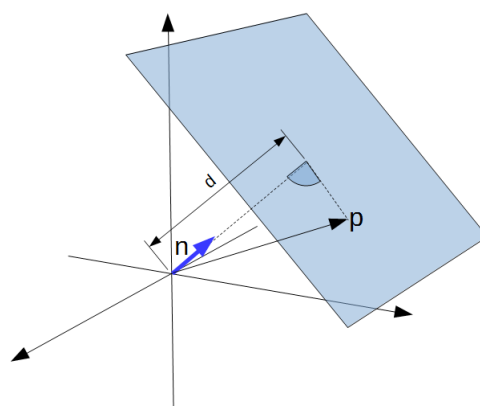
A bemeneti testektől megköveteljük a manifold tulajdonságot. (Minden fent látható test rendelkezik vele) Egy test pontosan akkor manifold tulajdonságú, ha minden csúcspontja és éle manifold tulajdonságú. Egy él pontosan akkor manifold tulajdonságú, ha pontosan két lap kapcsolódik hozzá. Egy csúcspont pontosan akkor manifold tulajdonságú, ha egy vele szomszédos lapot kiválasztva azt elhagyjuk, majd lapszomszédosság mentén tovább folytatva elhagyjuk az összes elérhető lapot, a csúcspont már nem érintkezik egy lappal sem.



Minden laphoz tartozik egy normálvektor, azaz olyan vektor, mely merőleges a lap síkjára. Ezen vektorokról a továbbiakban feltesszük, hogy egységnyi hosszúak az Euklideszi norma szerint, valamint testek felszínén olyan irányúak, hogy a kívülág felé mutassanak. A normálvektorokra normálisként, felületi normálisként is hivatkozhatunk.

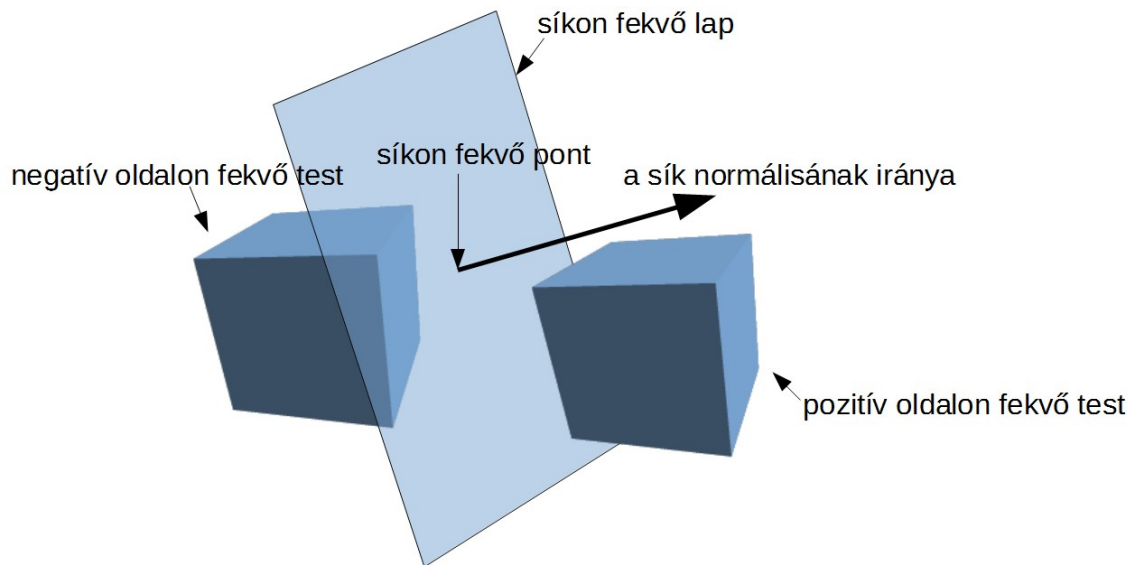


A síkokat többnyire vágásoknál használjuk, többször hivatkozunk rájuk vágósíkként is. A síkot leírhatjuk egy normálisával (n), valamint egy pontjával (p) vagy egy normálisával és az origóból induló adott normálisa mentén vett előjeles távolságával (d). A program során explicit irányítású síkokkal dolgozunk, azaz egy sík matematikailag ekvivalens, de ellenkező irányba mutató normálisú



leírásait különbözőnek tekintjük. A sík a teret két féltérre osztja, ezekre az egyértelműség miatt pozitív és negatív féltérként, esetleg oldalként hivatkozunk. A

pozitív féltér a sík azon oldalán helyezkedik el, amerre a normálisa mutat, a negatív az ellenkezőn. A kétdimenziós feladatoknál a síkok az egyenesek veszik át. Az egyenes analóg módon pozitív és negatív félsíkra bontja a síkot.



Skalárszorzat (vagy skaláris szorzat)[5] alatt a továbbiakban a következőt értjük:

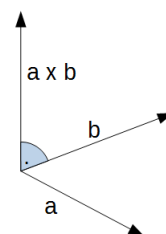
$$\langle a, b \rangle = \sum_{i=1}^n a_i \cdot b_i \quad (a, b \in \mathbb{R}^n)$$

A vektorokon értelmezett norma a skalárszorzatból származtatott kettes, vagy más néven Euklideszi[5] norma, ellenkező utalás nélkül a továbbiakban önállóan szereplő norma kifejezés is erre utal.

Ezután elmondhatjuk, hogy a fent említett n normálisú és p ponton átmenő síkra teljesül, hogy minden q pozitív féltérbeli pontra $\langle q-p, n \rangle > 0$ és minden r negatív féltérbeli pontra $\langle r-p, n \rangle < 0$

A keresztszorzat definíciója a következő:

$$a \times b = \begin{pmatrix} a_2 \cdot b_3 - a_3 \cdot b_2 \\ a_3 \cdot b_1 - a_1 \cdot b_3 \\ a_1 \cdot b_2 - a_2 \cdot b_1 \end{pmatrix} \quad (a, b \in \mathbb{R}^3)$$



Az a és b vektorok keresztszorzatára teljesül, hogy merőleges mindkét vektorra, hossza pedig a vektorok hosszának és bezárt szögük szinuszána szorzata.

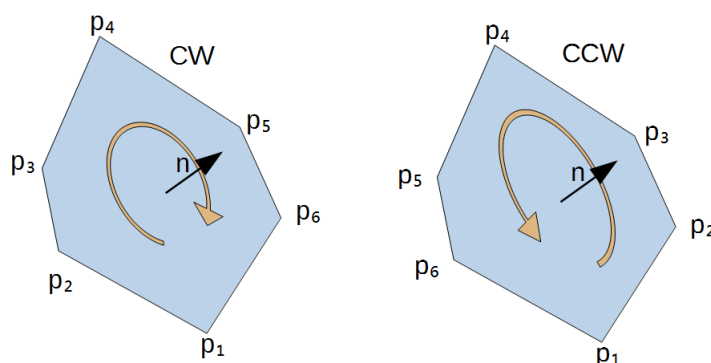
A keresztszorzat segítségével definiálhatjuk a jobb-rendszert, az $a, b, c \in \mathbb{R}^3$ bázis

ebben a sorrendben megadva jobb-rendszert alkot, ha $a \times b = c$. Amennyiben $a \times b = -c$, a három vektor bal-rendszert alkot.

A sokszögeket megadhatjuk csúcspontjaikkal, azokat bizonyos körüljárási irányban felsorolva. A felsoroláskor, a forráscikkekkel[1][2] ellentétben a sorban utolsó pont nem egyezik meg a legelsővel, minden pont pontosan egyszer szerepel, az első és az utolsó pontok implicit módon szomszédoknak tekintettek. A körüljárási irány lehet óramutató járásával megegyező (röviden CW - **C**lock**W**ise) vagy ellenkező irányú (CCW - **C**ounter**C**lock**W**ise), mostantól általában a rövidítésekkel hivatkozunk rájuk. A jobb-rendszerben, CCW körüljárással, p_1, p_2, \dots, p_m csúcspontjaival megadott, egyenesszög nem tartalmazó, n normálisú konvex sokszögek csúcspontjaira teljesül a következő.

$$\text{sgn}(\langle (p_k - p_i) \times (p_j - p_i), n \rangle) = 1 \quad (i, j, k \in \{1, \dots, m\}, i < j < k)$$

Azaz a keresztszorzat iránya a normálvektorral megegyező. Példa a felsorolásokra:



A testek egyik leglényegesebb tulajdonsága a térfogatuk[5][6]. A B test térfogatát matematikailag a következő képpen értelmezzük: $\int_B 1$ (A továbbiakban jelölése V .)

A Gauss–Osztrogradskij-féle divergenciatételre alapozott módszerrel[1][2], a testek térfogatának kiszámításához elegendő ismernünk lapjaik területeit és normálisait, valamint egy síkjukra eső tetszőleges pontot, ezekből a következő egyszerű képlettel határozhatjuk meg térfogatot:

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

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 normálisa

Egy lap általunk választott pontja és normálisa a lapok reprezentációjából kinyerhető, a sokszög területe pedig a sík kettő dimenziós koordináta rendszerébe leképezve a következő képlettel határozható meg:

$$A = \frac{1}{2} \left| \sum_{i=1}^n x_{(i+1) \bmod n} y_i - x_i y_{(i+1) \bmod n} \right|$$

A térfogat fogalmának segítségével definiálhatjuk a súlypont fogalmát.[5][6] A súlypont fogalma az általános használatban függ a test sűrűségétől, így tömegétől is, ám problémánk homogén testekkel foglalkozik, így a sűrűségfüggvényt konstans 1-nek vesszük, tehát a térfogat és tömeg megegyezik. A V térfogatú B test (x_s, y_s, z_s) súlypontja a következő:

$$(x_s, y_s, z_s) = \frac{1}{V} \left(\int_B x \, dx dy dz, \int_B y \, dx dy dz, \int_B z \, dx dy dz \right)$$

Testek körében értelmezhetjük az átmérő fogalmát, amely a test két tetszőleges pontját összekötő szakaszok közül olyan, melynek maximális a hossza. Az átmérő hossza egyértelmű, de az iránya nem, például egy kockának bármely testátlója az átmérője. A továbbiakban, ha átmérőről beszélünk, a test egy tetszőleges átmérőjét értjük alatta.

A továbbiakban többször felmerülnek a Fourier-együtthatók. Esetünkben egy atom Fourier együtthatója a cél test atommal vett metszetének és az atomnak a térfogat hányadosa. Egy atomot relevánsnak nevezünk, ha Fourier-együtthatója egy megadott határ fölött van. Az algoritmus konvergenciája abban az esetben bizonyított, ha ez a határ pontosan 0.5, tehát a releváns atomok a gyakorlatban azok, melyeknek legalább a fele a céltestbe esik.

Probléma és megközelítése

Célunk az absztrakt approximációs algoritmus[1][2] gyakorlatba ültetése. Az input egy

poliéder, melyet atomok halmazával közelítünk. A kiinduló halmaz egyetlen atom, amely magába foglalja a testet, esetünkben egy téglatest. Minden lépésben választunk egy atomot az előző halmazból, amelyet egy síkkal két diszjunkt részre vágunk, majd elhagyjuk az eredeti atomot és beszúrjuk a két keletkezett felet. A folyamatot addig folytatjuk, amíg a releváns atomok uniója az alkalmazott metrika szerint elég közel nem lesz a céltesthez. Az output igény szerint lehet a releváns atomok uniója, illetve az összes ilyen atomot tartalmazó halmaz.

Ez a leírás több kérdést hagy hátra. Hogyan válasszuk ki a vágandó atomot? Milyen síkkal vágjuk el, hogy az eredmény előnyös legyen? Ezek a kérdések jelenleg is aktív kutatás tárgyát képezik. Az algoritmus működhet automatikusan, előre megadott stratégiák szerint, vagy a felhasználó által felügyelve, manuálisan, esetleg a kettő kombinációjaként, azaz az egyes stratégiák felhasználó által felügyelt váltogatásával.

A következőkben bemutatott könyvtár feladata, hogy képes legyen kezelni a testeket, atomokat, valamint megfelelően felparaméterezve az iteráció egy fázisának végrehajtására, ezzel lehetővé téve az approximációt egy külső alkalmazás számára. A könyvtár feladata továbbá, hogy lehetőséget adjon az aktuális állapot megjelenítésére, fájlból való betöltésre és fájlba írásra. A könyvtár adatszerkezetei nagy részben a forráscikkek[1][2] által bemutatott indexelt adatszerkezetekre épülnek.

A probléma a következő részekre bontható:

- Az atomok halmazának kezelése.
 - Nyilvántartás, halmaznövelés vágási művelettel
 - Unió számítás
- Egy atom elvágása
 - Az atom lapjainak elvágása
 - A vágás síkján keletkező új lap megalkotása.
- Statisztikák készítése egyes atomokról a stratégiák számára
 - Térfogat számítás
 - A céltesttel vett metszet térfogat számítás

- Beeső lapdarabok meghatározása
- Az atom lapjaira eső metszetképek meghatározása
- Fourier-együttható meghatározás
- Súlypont számítás
- Átmérő számítás
- Atomba eső lapok meghatározása

A problémák egyszerűen bonthatóak részekre, az atomokon és testeken végzett számítások visszavezethetőek a lapjaikon végzett részműveletekre, ezek pedig pontjaikon végzett részműveletekre. Ez a tény lehetőséget ad a nagy részint objektum orientált megközelítésre és a probléma „függőleges” tagolására. Az implementáció alulról felfelé haladva egyszerűn elkülöníthető és tesztelhető részekre bontva történik.

A cikkekben az iteráció a halmazon csak cserét és hozzáadást végez, a nem elvágott atomok érintetlenül maradnak a halmaz részei, tehát nem történik törlés. Az implementációban a memória és műveletigény figyelembevételével, a későbbiekben biztosan érdektelen, céltesttel diszjunkt atomok eldobására is fontos lehetőséget adni. Ezen atomok nulla Fourier-együtthatóval rendelkeznek, így a választó függvények figyelmen kívül hagyják őket, valamint az iteráció megállásánál a végeredménybe sem számítanak bele.

Tervezési megfontolások

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

Fejlesztés időrendben

1. Vektorok, síkok, lapok típusainak felvétele
2. Kétdimenziós sokszögek típusainak felvétele
3. Testek, atomok típusának felépítése
4. Lapok, atomok elvágási algoritmusa
5. Működtető tárolók megalkotása
6. I/O műveletek megalapozása (összekapcsolás a másik féllel)
7. Metszetkép alkotó algoritmus
8. Metszet térfogat számítás
9. További statisztikák felvétele a testekre és atomokra
 1. Beeső lapok
 2. Átmérő
 3. Súlypont
10. Atomok uniójának megvalósítása
 1. Szomszédsági struktúra éppen tartása
 2. Atomvágás környezetre terjedése
 3. Egyesítő algoritmus

A könyvtár alkalmazása egy program részeként

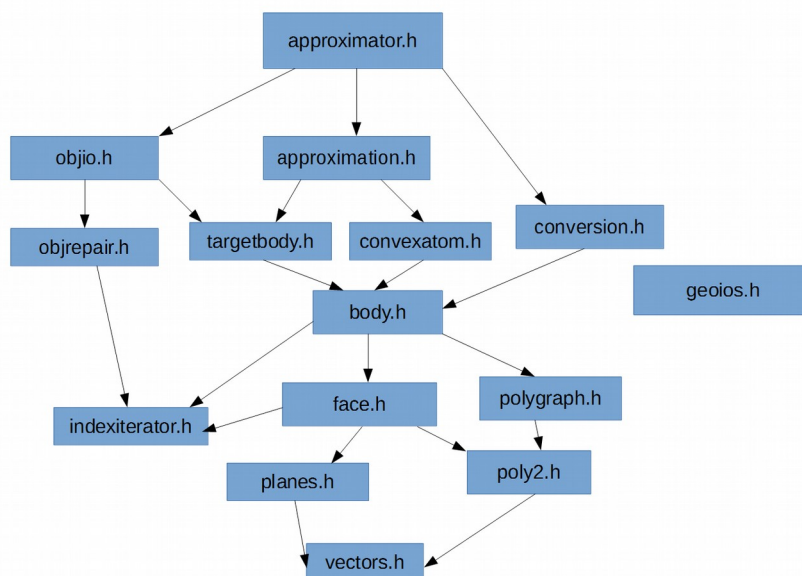
A következőkben könyvtár kívülről szemlélve lényeges felépítése, az egyes komponensek publikus tulajdonságai, a számítások műveletigényei, elő- és utófeltételei olvashatóak, a háttérben működő algoritmusokat részletesebben a következő főfejezet tárgyalja.

A könyvtár bármely C++11 vagy későbbi szabványt támogató fordítóval felhasználható. (A könyvtárat Microsoft Visual C++ 2015, valamint GNU g++ 5.2.0 alatt teszteltem, az utóbbit `-std=c++11` és `-std=c++14` fordítási direktívákkal használva.)

Mivel minden típus és függvény sablonok formájában van megvalósítva, a központi forráskód kizárólag header fájllokból áll, emellett elkülönítve megtalálhatóak a tesztelésre és példák kipróbálására használható példaprogramok forrásai. Az egyetlen külső függőség a szabadon felhasználható GLM csomag, melyet a <http://glm.g-truc.net> címről tölthetünk le, de kizárólag a megjelenítéshez szükséges.

A könyvtárt felhasználó programozók számára fontos kiemelni, hogy a C++ standard könyvtárához hasonlóan az osztályok metódusai kerülnek 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.



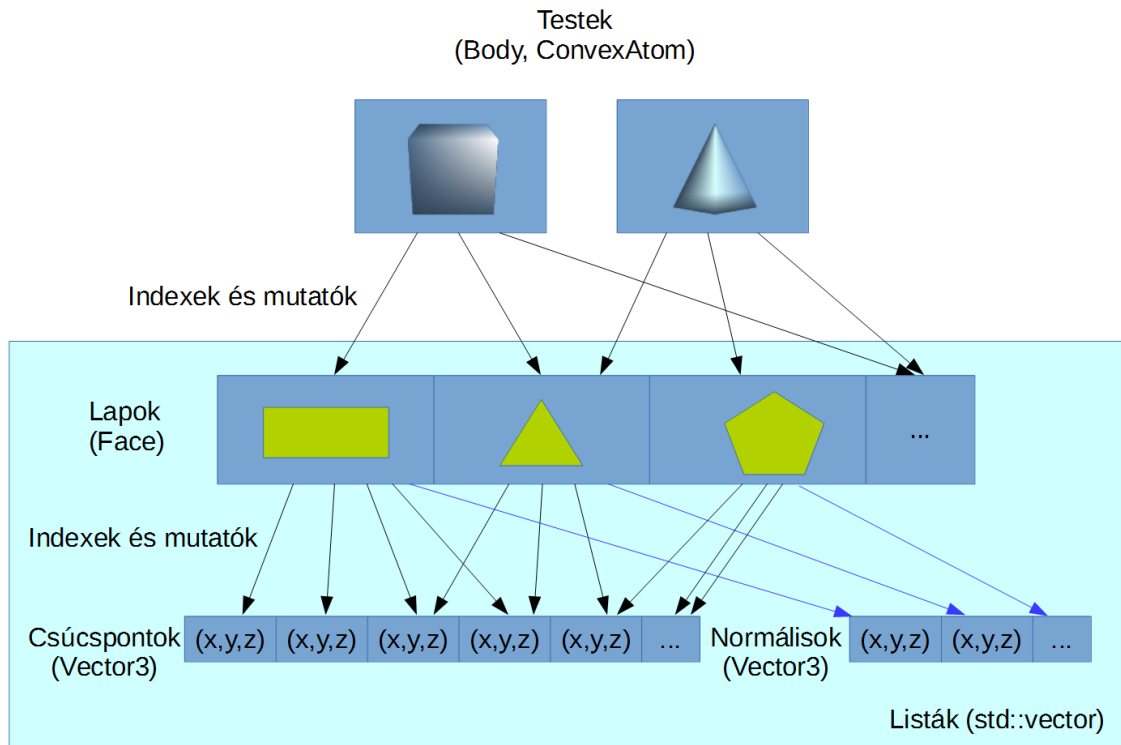
A GLM csomag a `conversion.h`-ban kerül felhasználásra, ha esetleg nincs szükség grafikus megjelenítésre a program működése során, az `approximator.h` és `conversion.h` kihagyásával, illetve az `approximator.h` használata esetén fordításkor az `APPROX_NO_CONVERSION` makró definiálásával mellőzhetjük a függőséget, lemondva a `conversion.h` és azt használó metódusok használatáról.

```
classDiagram
    class Approximator
    class Approximation
    class TargetBody
    class Body
    class Face
    class ConvexAtom
    class Polygon2
    class Vector3

    Approximator "0..1" o-- "0..1" Approximation
    Approximator "0..1" o-- "0..1" TargetBody
    TargetBody "1" -- "0..*" ConvexAtom
    TargetBody "0..1" o-- "1" Body
    Body "0..*" o-- "0..*" Face
    ConvexAtom "1..*" o-- "0..1" Polygon2
    ConvexAtom --|> Body
    ConvexAtom "3..*" o-- "0..1" Vector3
```

The diagram illustrates the relationships between the **Approximator** class and other classes in the system. The **Approximator** class is associated with **Approximation** (0..1 to 0..1), **TargetBody** (0..1 to 0..1), and **ConvexAtom** (0..1 to 0..*). The **TargetBody** class is associated with **ConvexAtom** (1 to 0..*), **Body** (0..1 to 1), and **ConvexAtom** (0..* to 0..*). The **Body** class is associated with **Face** (0..* to 0..*) and **ConvexAtom** (0..* to 0..*). The **ConvexAtom** class is associated with **Polygon2** (1..* to 0..1) and **Vector3** (3..* to 0..1). The **ConvexAtom** class also inherits from **Body**.

Az testek szemléletes felépítése:



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 feltüntettem. 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. Használatukról bővebb információ a tesztelés fejezetben található.

A továbbiakban az egyes algoritmusok műveletigényét az $O(f)$ jelöléssel adom meg, mely az f valós értékű függvény aszimptotikus felső becslését jelöli.

Vektorok

A számítások során két- illetve háromdimenziós valós vektorterekben dolgozunk. A vektorokat megvalósító sablonok a `vectors.h` fejlécfájlban találhatóak `Vector2` és `Vector3` néven. A vektorokat koordinátáik írják le, ezeket név szerint érhetjük el. Mindkét sablon a használt skalár típuskal 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, összeadási ellentett képzés műveletekkel, egyenlőség és különbözőség vizsgálattal, valamint skalárszorítás külső művelettel. Három dimenzióban a keresztszorítás 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); //keresztszorítás
v3+=v5; v4-=v3; v1*=2.0f;
float skalarszorítás = dot(v1,v2); //skaláris szorítás
```

A skalárszorítás és keresztszorítás segítségével a közbezárt szögek szinusza és koszinusza szögfüggvényeit is kiszámolhatjuk. Ennek jelentősége például a sokszögeken végzett műveletek pontosságánál merülhet fel, ahol a numerikus precizitás érdekében ezen értékek közül alkalmasat szeretnénk választani. A `length` módszer segítségével Euklideszi normát számolhatunk, a `normalize` módszer helyben normál, a

normalized metódus pedig kiszámítja az egység hosszú 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.

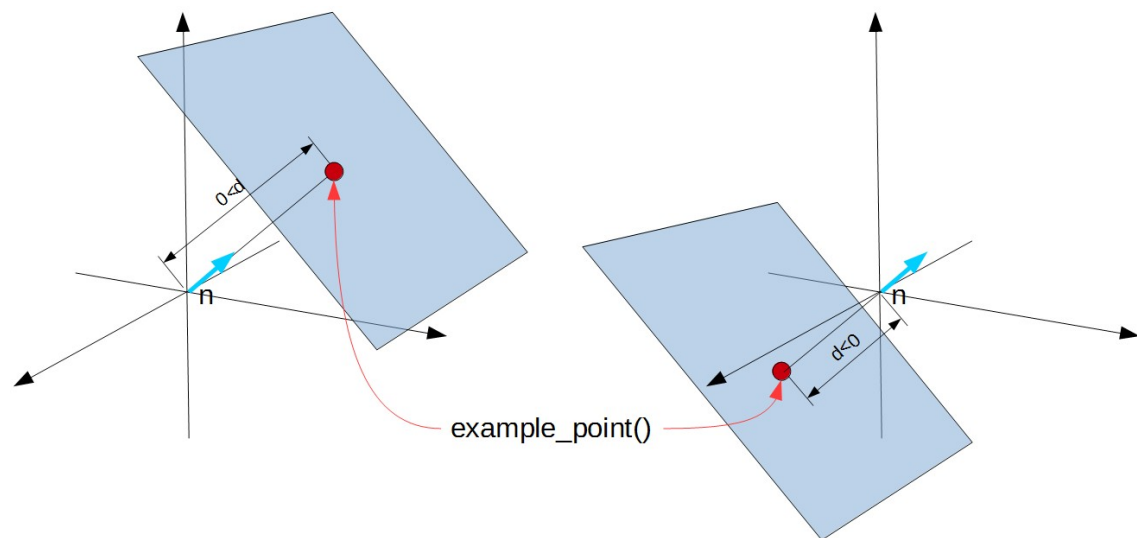
Síkok

A síkokkal kapcsolatos sablonok a planes.h fejlécfájlban 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, az érvénytelen sík normálisa a nullvektor, origótól mért távolsága a skalártípus default értéke.
HyperPlane(const Hyperplane&)	
	Másoló konstruktor.
T signed_distance() const	
	Az origótól mért előjeles távolság.
Vector normal() const	
	Egységnyi hosszú normálvektor.
Vector example_point() const	
	Az origóból indított normális meghosszabbításának metszéspontja a síkkal.
ScalarType classify_point(Vector point) const	
	Előjeles távolság a ponttól, az előjel a félterekbe sorolja a pontokat.

<code>ScalarType distance(Vector point) const</code>	
	A megadott ponttól vett távolság, kettes normában véve.
<code>bool valid() const</code>	
	Érvényes-e a sík.

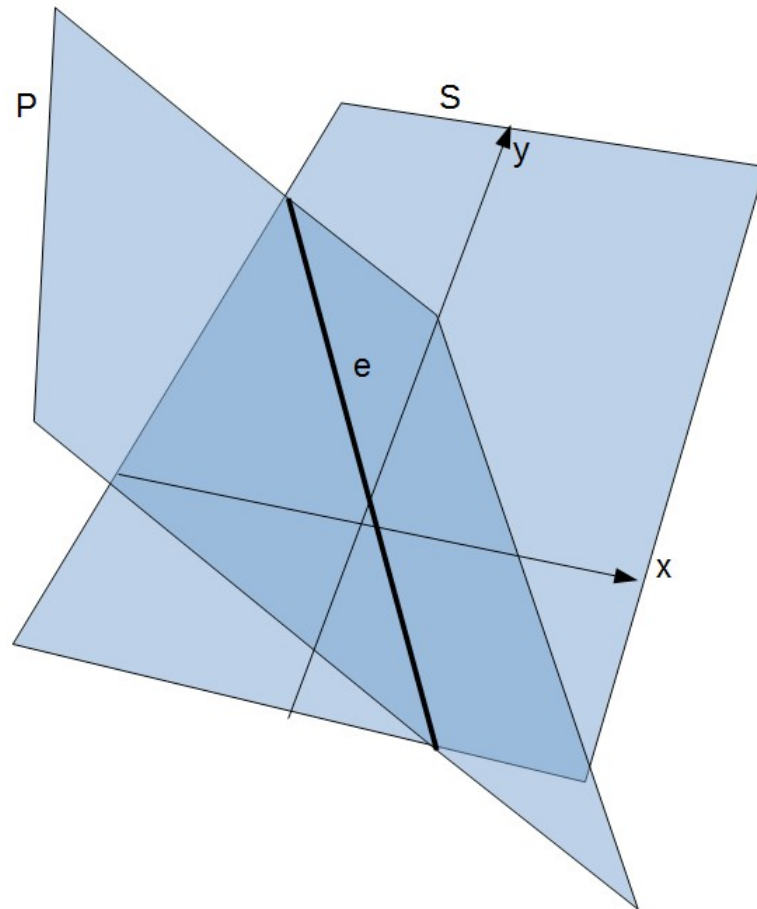
Háromdimenziós szemléltetés:



A kétdimenziós 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 érvénytelen létrehozni. A háromdimenziós síkot képviselő Plane sablon további metódusai:

<code>std::pair<Vector3<T>,Vector3<T>> ortho2d() const</code>	
	Vektorpár, melyek tengelyként használhatóak fel a saját koordináta-rendszerhez. A vektorok a numerikus hiba csökkentése érdekében Fábián Gábor koordináta kiválasztó módszerével[1][2] készülnek, elkerülve a túl kicsi számokkal való osztást.
<code>Line<T> intersection_line(plane) const</code>	
	Egy másik síkkal való metszés során kapott egyenes, a sík saját koordináta-rendszerében felírva.

Az alábbi ábra az S és P síkok metszésénél keletkező e egyenes esetével mutatja be az $S.intersection_line(P)$ működését. Az e egyenest a hívás az S sík x,y koordináta rendszerébe viszi.



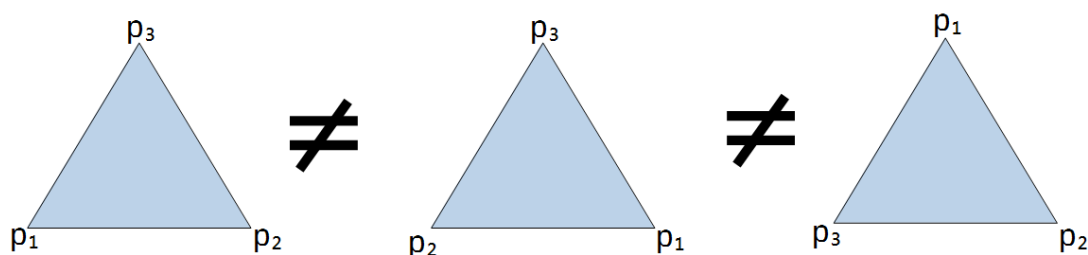
A metódus felhasználása a későbbiekben tárgyalt kétdimenziós metszetképek elvágására használható egyenesek megteremtése.

Sokszögek

A két- és háromdimenziós sokszögek egymástól eltérő elvárásokkal lettek megtervezve. A háromdimenziós sokszögeket reprezentáló **Face** sablon a Fábián Gábor és Gergő Lajos cikkeiben[1][2] 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. Két-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 kétdimenziós **Polygon2** sablon önmagában alkot egy egészt, nem igényel külső tárolót. Mindkét esetben egyezik azonban, hogy a sokszögeket pontjaik felsorolásával adjuk meg, tetszőleges körüljárási iránnyal, a reprezentálható sokszögek pedig egyetlen törött vonallal határolt, nem önmetsző sokszögek.

Polygon2

A `poly2.h`-ban található **Polygon2** sablon paramétere pontjainak skalártípusa. A skalárra a **Vector2** feltételei mellett megkötés, hogy értelmezhetőnek kell lennie az `std::abs(x)` és `x > 0` műveleteknek. Az alábbi metódusokon kívül, az osztály 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.



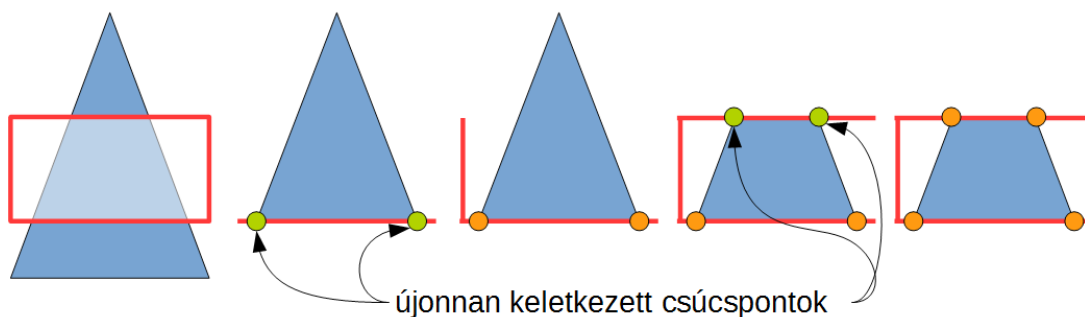
Bár pontjai külön-külön módosíthatóak, az osztály metódusai konstansként való felhasználásra készültek, mellékhatások nélkül végeznek számításokat, így függvény orientált megközelítésű feladatmegoldásra is használhatóak.

A Polygon2 metódusai:

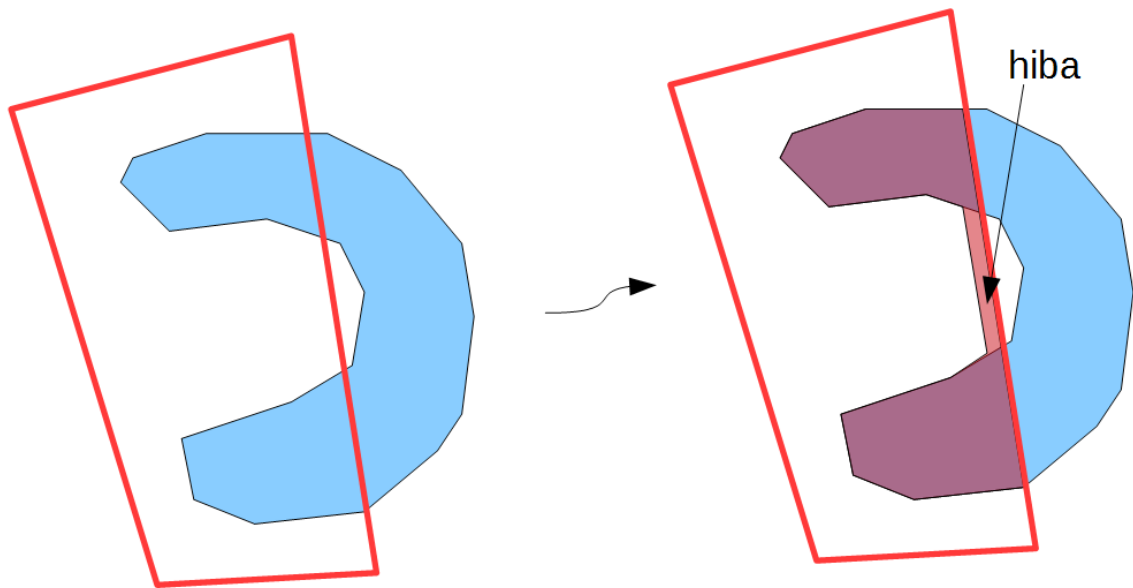
Polygon2(const std::vector<Vector2<T>>&)	
Polygon2(std::vector<Vector2<T>>&&)	
	Konstruktor a pontok vektorban megadott sorozatával. A körüljárási irány tetszőleges, az algoritmusok igazodnak hozzá.
Polygon2(const Polygon2&)	
Polygon2(Polygon2&&)	
	Másoló konstruktor és mozgató konstruktor.
template <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, ellenkező esetben pozitív. $O(size())$ 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, az előjeles terület előjelét használja fel.

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 félsíkba tartozás szerint vannak elnevezve.
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.

A Sutherland-Hodgman algoritmus[7] kétdimenziós sokszögek konvex sokszögekkel való körülvágására alkalmas. Mivel a konvex sokszögek megadhatóak véges sok félsík metszeteként, az algoritmus végigiterál a vágó sokszög oldalain, minden egyes oldalszakaszt egy egyenes irányvektoraként felfogva elfelezi a vágandó sokszöget és eldobja a nem kellő csúcsokat, ha két szomszédos csúcs az egyenes két oldalára esik, akkor közöttük létrehoz egy új csúcsot.



Az algoritmus előnye gyorsasága, egyszerűsége, hátránya pedig, hogy konkáv sokszögek vágásánál lényegében nulla területű hibák is keletkezhettek, ezt a következő, szemléltetés érdekében elnagyolt ábra szemlélteti.



A fent látható hiba eltűzött, a valóságban a keletkező rész térfogata nulla közeli, szemléletesen pedig egy vonalként jelennek meg, döntés alapján elfogadhatónak tekintjük létezésük.

Face

A face.h-ban található Face sablon sablonparamétere a pontvektoraiban felhasznált skalár típus. A **Vector3** sablon feltételei mellett, értelmezhetőnek kell lennie az $\mathbf{x} > \mathbf{0}$ és $\mathbf{x} < \mathbf{0}$ műveleteknek. A lapok egy test határoló lapjait írják le, így fontos szerephez jut normál vektoruk iránya, melynek a testből kifelé kell mutatnia. A szakirodalom[3][4] hangsúlyozza annak fontosságát, hogy a normálisokat ne a pontokból visszanyerve számítsuk, 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 követeljük meg a lapok között is állandó körüljárási irány használatát, 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 önmagukban képesek konkáv alakzatokat is leírni, sőt több számítás azokra is megfelelően működik, a program bevezetésben ismertetett megkötései szerint csak konvex alakzatokat használunk.

A vágások megvalósítása a felszínen funkcionális szemlélettel 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 változatlanosságát 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,normalcointainer, normalid)	
Face(container,ids,normalcointainer, ccw)	
Face(container,idbeg,idend,normalcointainer, normalid)	
	<ul style="list-style-type: none"> • 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>& indices() const	
std::vector<int>& indices()	
	Az pont indexvektor kinyerése.
int indices(ind) const	
	Az adott sorszámú index kinyerése.
int normal_index() const	
int& normal_index()	
	A normálvektor tárolóbeli indexe.

<code>size_t size() const</code>	
	A csúcspontok száma.
<code>std::vector<Vector3<T>>* vertex_container() const</code>	
	Mutató a ponttárolóra.
<code>std::vector<Vector3<T>>* normal_container() const</code>	
	Mutató a normálistárolóra.
<code>const Vector3<T>& normal()const</code>	
	Normálvektor.
<code>const Vector3<T>& points(int) const</code>	
	Az adott indexű csúcspont.
<code>Plane<T> to_plane() const</code>	
	A sík megadása, melyen a lap fekszik, a normálisuk egy irányba mutat.
<code>Face<T>::VertexIterator begin() const</code>	
	Konstans elérést biztosító iterátor az első csúcspontra
<code>Face<T>::VertexIterator end() const</code>	
	Konstans elérést biztosító iterátor az utolsó utáni csúcspontra.
<code>Face<T> migrate_to(std::vector<Vector3<T>>* target_vecs,</code> <code>std::vector<Vector3<T>>* target_normals) const</code> <code>Face<T> migrate_to(std::vector<Vector3<T>>* target_vecs,</code> <code>std::vector<Vector3<T>>* target_normals)</code>	
	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.
<code>void reverse_order()</code>	
	Helyben megfordítja a körüljárási irányt. $O(size())$ műveletigény.
<code>Face<T> reversed() const</code>	
	Létrehozza a lap ellenkező irányba álló másolatát, az ellentétes normálvektort beszúrja a tárolóba. $O(size())$ műveletigény.
<code>Vector3<T> centroid() const</code>	
	Súlypont számítása. $O(size())$ műveletigény.
<code>Polygon2<T> to_2d() const</code> <code>Polygon2<T> to_2d(const Vector3<T>& x, const Vector3<T>& y)</code> <code>const</code>	

	A lap csúcsait leképezve a sík koordináta-rendszerébe 2 dimenziós sokszöget kapunk. $O(size())$ műveletigény. Amennyiben adunk paramétereket, azokat tekinti koordináta tengelynek.
<code>Face<T>::CutResult cut_by(const Plane<T>& p)</code> <code>Face<T>::CutResult cut_by(const Plane<T>&p, Map map)</code> <code>Face<T>::CutResult cut_by(const Plane<T>&p,</code> <code>std::vector<Vector3<T>>* tv, std::vector<Vector3<T>>* tn)</code>	
	<p>Vágás adott síkkal.</p> <ul style="list-style-type: none"> • p – vágást végző sík • map – korábban beszűrt pontok indexeit kezelő asszociatív tároló • tv, tn – új tároló melybe az eredmény lapok kerülnek. <p>Az első és utolsó esetben $O(size())$ műveletigény, a map használata mellett $O(size()*K)$, ahol K a tároló beszúrás és keresési költsége.</p> <p>Ha egy konkáv alakzattal akarnánk alkalmazni, közel 0 területű hibák keletkezhetnek. A Sutherland-Hodgman algoritmus ciklusmagjának ötletét általánosítja.</p>
<code>bool is_ccw() const</code>	
	Pontosan akkor igaz, ha CCW körüljárás irányban megadott lapjaink vannak. $O(size())$ műveletigény, kétdimenzióba képzés, terület előjel vizsgálat.
<code>bool insert_index(int i1, int i2, int ind)</code>	
	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.
<code>std::pair<int,int> neighbours_of(ind) const</code>	
	Ha megtalálja az adott tárolóbeli indexű csúcspontot, 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 a bevezetésben tárgyalt tulajdonságokat. A `body.h`-ban található `Body`

sablon osztály indexhivatkozásokkal felépítve képes leírni ilyen testeket, valamint számításokat végezni rajtuk. 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 atomok származtatással keletkeztek.

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:

<code>Body(std::vector<Face<T>>* faces, const std::vector<int>& inds)</code> <code>Body(std::vector<Face<T>>* faces, std::vector<int>&& inds)</code>	
	<ul style="list-style-type: none"> • faces – mutató a laptárolóra • inds – indexsorozat mely a lapokra hivatkozik Rendelkezésre állnak másoló és mozgató konstruktorok is.
<code>bool valid() const</code>	
	Pontosan akkor igaz, ha a mutatója nem nullpointer és több mint egy darab lapja van. $O(1)$ műveletigény. A <code>bool</code> konverziós operátor is ezzel egyenlő.
<code>Body<T> migrate_to(std::vector<Face<T>>*) const</code> <code>Body<T> migrate_to(std::vector<Face<T>>*)</code>	
	Konstans esetben lemásolja a testet csak a mutatóját állítja át a megadottra, nem konstans esetben mozgatót használ. Konstans esetben $O(size())$, nem konstans esetben $O(1)$ műveletigény. Befoglaló tárolók költöztetésénél használandó, mikor a tárvektorok címei változnak.
<code>int size() const</code>	
	A test lapjainak száma. $O(1)$ műveletigény.
<code>const std::vector<int>& indices() const</code> <code>std::vector<int>& indices()</code>	
	Az indexvektor mely a lapokra hivatkozik.
<code>int indices(i) const</code>	

	Az i . lap indexe a tárolóban.
<code>const Face<T>& faces(i) const</code> <code>Face<T>& faces(i)</code>	
	Az testben i . sorszámú lap.
<code>T volume() const</code>	
	A test térfogatának kiszámítása. Elsőre $O(\text{size()}*K)$ műveletigény, ahol K a lapok átlagos csúcsszáma, $O(1)$ műveletigény amennyiben korábban kiszámoltuk és nem ürítettük a cache-t.
<code>Vector3<T> centroid() const</code>	
	A súlypont kiszámítása. Elsőre $O(\text{size()}*K)$ műveletigény, ahol K a lapok átlagos csúcsszáma, $O(1)$ műveletigény amennyiben korábban kiszámoltuk és nem ürítettük a cache-t.
<code>bool intersects_plane(const Plane<T>& p) const</code> <code>bool intersects_plane(const Plane<T>& p, T min) const</code>	
	Pontosan akkor igaz, ha az adott sík átmegy a testen és mindkét oldalán találhatóak csúcspontok. A széleken fellépő numerikus hiba ellen minimális távolságot is szolgáltatathatunk, amin belül egy pontot a síkon fekvőnek tekintünk.
<code>std::vector<std::pair<Polygon2<T>, bool>> cut_surface(const Plane<T>&) const</code>	
	Eredménye az a sokszögekből álló kétdimenziós kép ami az adott síkon keletkezik, ha elvágjuk vele a testet, a sík koordináta-rendszerében megadva. $O(\text{size}())$ műveletigény.
<code>Vector3<T> diameter() const</code>	
	Eredménye egy maximális hosszú test- illetve lapátló vektorként megadva. Elsőre $O(\text{size()}*K)$ műveletigény ahol K a fedlapok átlagos csúcsszáma, $O(1)$ műveletigény amennyiben korábban kiszámoltuk és nem ürítettük a cache-t.
<code>void clear_cache() const</code>	
	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 új atom keletkezik. 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 képesek meghatározni a saját Fourier-együtthatójukat.

A ConvexAtom metódusai:

<pre>ConvexAtom(std::vector<Face<T>>* faces, const std::vector<int>& indices, const Body<T>* target) ConvexAtom(std::vector<Face<T>>* faces, std::vector<int>&& indices, 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)</pre>	
	<ul style="list-style-type: none"> • faces – mutató a laptároló vektorra • indices – lapokra mutató indexvektor • target – mutató a közelítendő testre • plist – lapvetület lista
<pre>const Body<T>* target_body() const</pre>	
	A céltestre mutató mutató.
<pre>ConvexAtom<T>::CutResult cut_by(plane) const</pre>	
	Elvágás az adott síkkal. Hozzávetőlegesen $O(size()*K)$ műveletigény, ahol K a fedőlapok átlagos csúcsszáma.
<pre>bool point_inside(const Vector3<T>&) const</pre>	
	Pontosan akkor igaz ha a megadott pont az atomba esik. $O(size())$ műveletigény.
<pre>int target_vertex_count_inside() const</pre>	
	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 koordinátáinként 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. <ul style="list-style-type: none"> • 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. $O(\text{size()} * \text{target().size()})$ 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 bizzuk. 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.

Az 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)	
	<p>Megadott atom elvágása az adott síkkal. Ha van elfogadatlan vágásunk, az automatikusan visszavonódik ez előtt.</p> <ul style="list-style-type: none"> • 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)	

	<p>Az approximáció eredmény testének kiszámítása. Automatikusan végrehajt egy szemétgyűjtést. A kapott test érvénytelenné válhat későbbi szemétgyűjtés után.</p> <ul style="list-style-type: none"> • 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, alapértelmezett értéként -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 vektor 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 háromszögelést.
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)	
	<ul style="list-style-type: none"> • 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 <p>Az approximáció felkészítése az adott célra.</p>
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.

<code>BodyList target_drawinfo() const</code>	
	A céltest rajzolási adatai.
<code>BodyList approx_drawinfo() const</code>	
	A számított test rajzolási adatai.
<code>BodyList cut_drawinfo() const</code>	
	El nem fogadott vágásnál az atomok rajzadatai, ha nincs elfogadatlan vágásunk akkor nem definiált működés.
<code>std::vector<PolyFace2D> atom2dfaces(int ind) const</code>	
	Az adott indexű atom lap metszetképei rajzoláshoz.

Input, output

A valós felhasználásnál a közelítendő testet fájlból nyerjük, az eredményt pedig grafikusán 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 lapsorozatok 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 következők. Bármely más sorfajta figyelmen kívül

hagyunk, ezen sorok helyességét azonban megköveteljük.

„v x,y,z”	Csúcspont koordinátaival 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 fentiekén 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, T scale, const Vector3<T>& m)

void ObjectWriter<T>::save_obj(const std::string& filename,
const Approximation<T>& app, T scale, const Vector3<T>& m)

void ObjectWriter<T>::save_obj(const std::string& filename,
const Body<T>& b, T scale, const Vector3<T>& m)
```

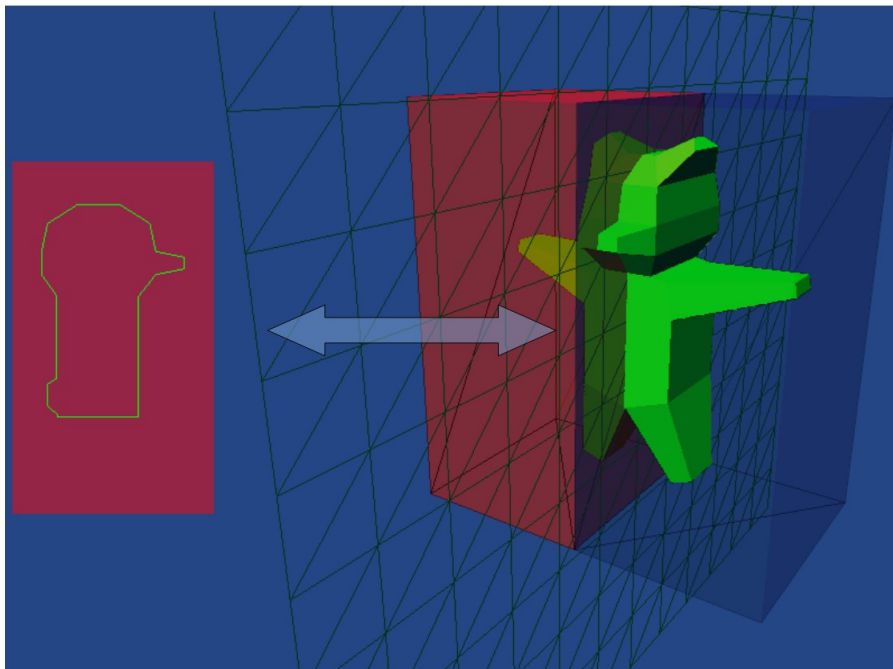
Amennyiben csak egy testet akarunk elmenteni, a csúcs és normális tárolókon szelekció hajtódik végre, hogy csak a szükséges pontok kerüljenek a fájlba. Az approximáció vagy iterátor tartomány fájlba írásánál ez nem történik meg. Az iterátor tartományról feltételezett, hogy egy tárolóra hivatkoznak, így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

amennyiben szükségtelenek alapértelmezett értékeik változatlanul hagyják a testeket.

Megjelenítés

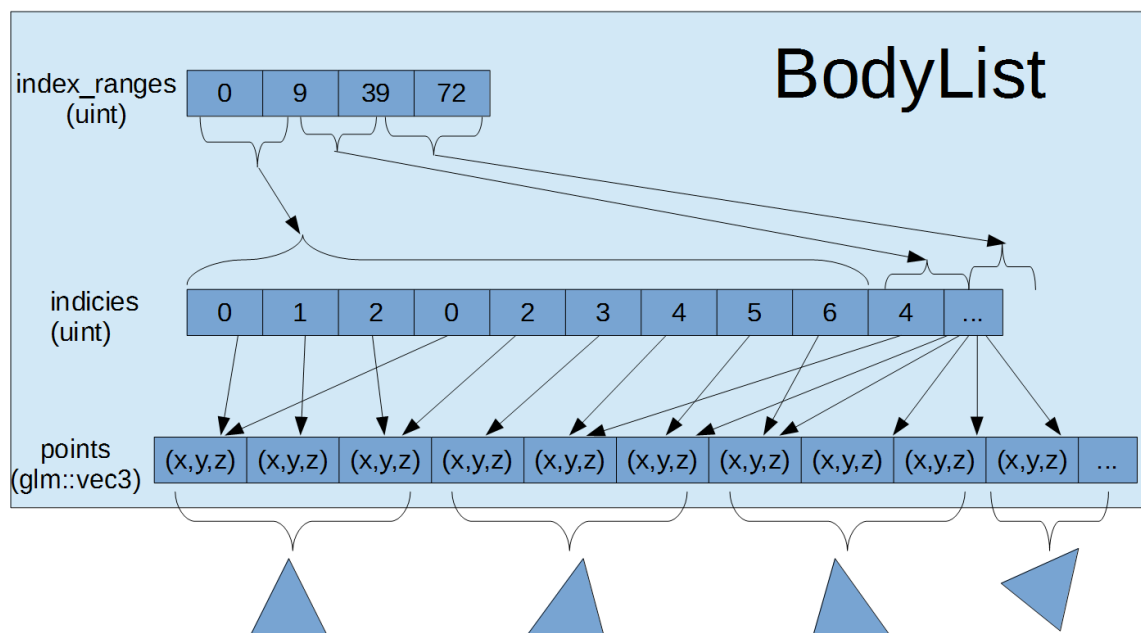
A sebességnövelés és némi helytakarékoság érdekében a számítások nem háromszögelt testeken történnek, azonban a modern grafikus keretrendszereknek háromszögelt felületekre van szükségük. Emellett elterjedt technika a hátlapeldobás, mely az irányítás alapján választja ki a megjelenítendő lapokat. Az indexelt szerkezet öröklődik a számításra használt szerkezetből, így a csúcsok ismétlődése csak index ismétléssel jár, így kevesebb a helyigény.

A kézi vezérlésű algoritmusok során felmerült az atom oldalain levő metszetképek külön megjelenítésének igénye. Ez kétdimenzióban 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. Témavezetőnkkel való egyeztetés után a képeket körvonalakként való megjelenítésre készítettem 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.

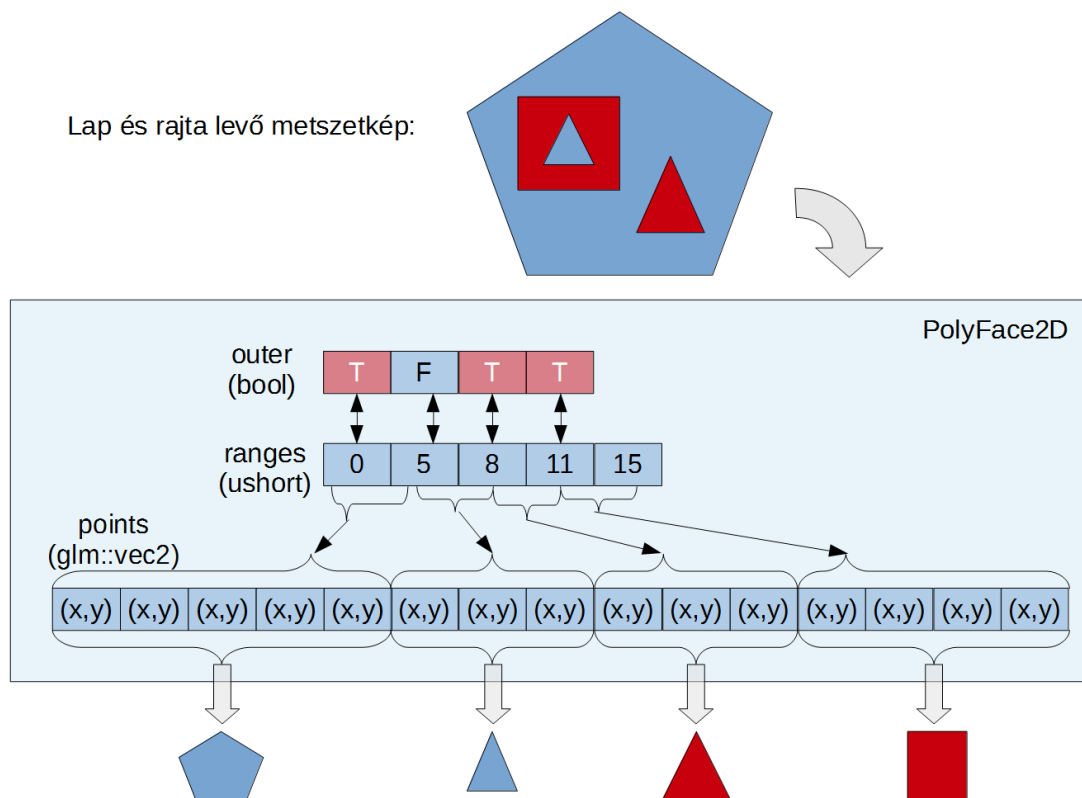


A megjelenítendő adatokat leíró típusok a `BodyList` és `PolyFace2D` típusok úgy készültek, hogy a videokártyákra való gyors és egyszerű feltöltést lehetővé tegyék.

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 keletkezett, így az indexek `unsigned int`-re cseréltük. Szemléletes struktúrája a következő ábrán látható. Az `index_ranges` különböző testekre osztja a listát és szeleteket jelöl ki az indexlistából, melynek egymás utáni elem hármasai hivatkoznak egy-egy háromszögre. A normálvektorok számolását a felhasználó kódra bízta, garantálva a CCW sorrendet.



A Polyface2D típus struktúrája az alábbi ábrán látható. Az érthetőség kedvéért itt a síkon keletkező metszetkép piros kitöltést kapott. A T és F rövidítések sorra az igaz és hamis értékeket jelölik. Az első elem maga a lap melyen a kép van, a hozzá tartozó logikai érték irreleváns, a könyvtár jelenleg igaz értéket rendel hozzá.



Háttér felépítés

Közbenső komponensek

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

Index iterátorok

A feladatok során többször előforduló igény, 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 könnyed 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 ezen absztrakt típusok konkretizált változataival járhatóak be.

Javítók

Az algoritmusokban, ahol vektor folyamatot 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 $N \cdot \log(N)$

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 előállítás. Ezen lapok azon pontokból állnak melyek a vágásnál keletkeznek, ám a pontok felsorolási sorrendjében nincs semmilyen szabályosság, sőt egyes pontok legalább kétször, de speciális csúcsok esetében többször is előfordulhatnak a felsorolásban. A megoldást a Fábián-Gergő-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 Sutherland-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.

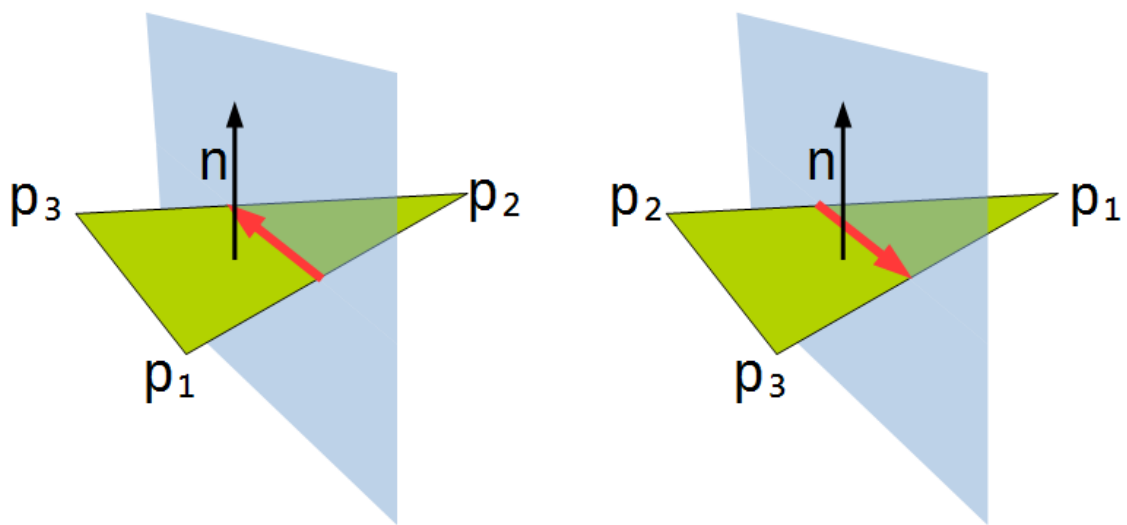
Térfogat számítás

A térfogatot a bevezetőben ismertett képletek segítségével számolhatjuk ki. A testek lapjain végig iterálva összegezzük területük és síkjuk origótól mért előjeles távolságának szorzatát. A lapok nem tárolják saját területüket, azonban ezt pontjaik számában lineáris műveletigénnyel képesek meghatározni, saját maguk a sík koordináta rendszerébe való leképezésével két dimenzióba lépve már alkalmazható a bevezetőben

található terület képlet.

Metszet kép építés

A `polygraph.h`-ban található függvények a testvágás során keletkező kétdimenziós 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, az irányított szakaszok irányítására még rögzített körüljárási irány mellett sem támaszkodhatunk a csúcssorrend meghatározásához, ennek szemléltetése az alábbi ábrán látható.

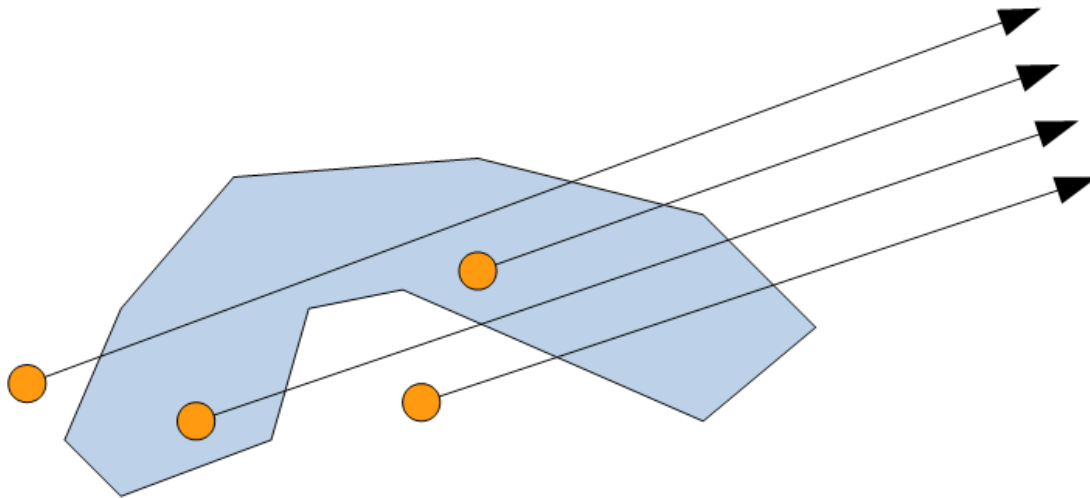


A probléma megoldását a gráfokra veztettem vissza. A szakaszok végpontjait egy irányítatlan gráf csúcspontjainak tekintem, melyek között pontosan akkor vezet él, valamelyikből indul irányított szakasz a másikba. Ezzel az átalakítással a sokszög keresést a gráfban való körök keresésére vezetem vissza, a 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.

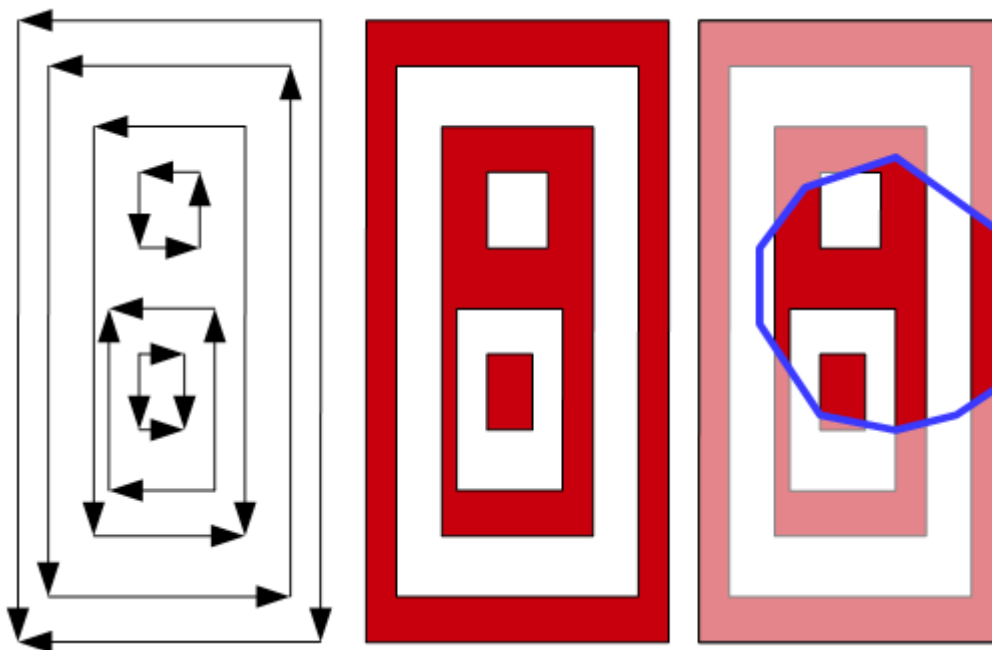
A kimenetként keletkező listát a `Body` megfelelő metódusában dolgozom fel. Az egyes

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[8] használom a kisebb sokszög csúcspontjain. Az algoritmus lényege, hogy a sokszög belsejéből kiinduló tetszőleges, rögzített irányú félegyenes páratlan, míg a rajta kívül indított páros számú, (esetleg nulla) helyen metszi a határoló törött-vonalat.



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. A listában szereplő sokszögeket az atom síkra eső lapjával körbevágom, így a kapott kép csak a lapra eső részt tartalmazza.



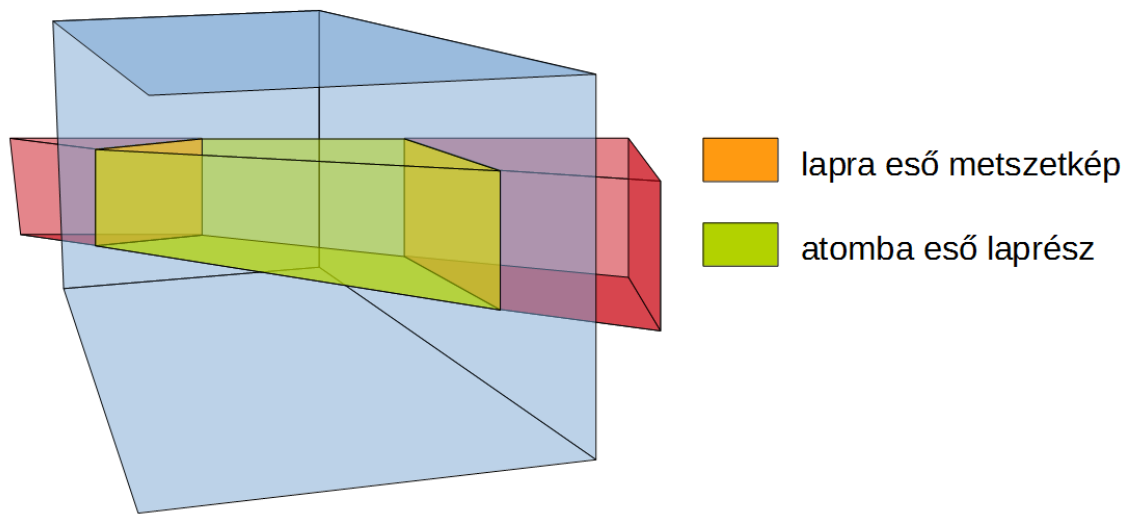
Metszet térfogat számítás

Az atomok tulajdonságai között fontos szerepet kap a céltesttel vett metszetének térfogata. Ez a metszet maga is egy poliéder, 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, ezzel helytakarékos és hatékony működést érünk el.

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. (Ez lényegében a Sutherland-Hodgman algoritmus háromdimenzióba ültetett változata). 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 elraktározzuk a számértéket, lényegi memóriaigénybeli növekedés nélkül gyorsíthatjuk a futást. Ezeket figyelembe véve az atomok nem tartják nyilván a beléjük eső lapokat, ellenben a térfogat és metszettérfogat számértékét kiszámítás után elraktározzák.

Súlypont számítás

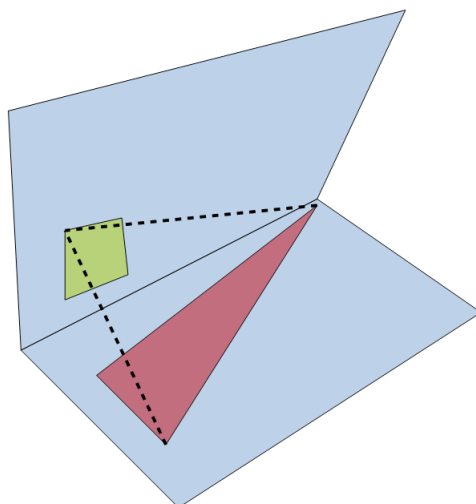
A súlypont meghatározására használt módszert a [6] forrás tárgyalja. Az eljárásnak egy háromszögelt poliéderre van szüksége. A cikk továbbá megköveteli a CCW orientálású lapokat, de az explicit módon lapokhoz rendelt normálvektorok segítségével, az eredeti képleten apróbb módosítást eszközölve tetszőleges felsorolási orientálást képes kezelni, a program megkötései szerint konvex fedlapok pedig triviálisan, akár a ciklus közben is háromszögelhetők. Ekkor adott V térfogatú test c súlypontjának d . koordinátája a következő alakban írható fel.

$$c e_d = \frac{1}{2V} \sum_1^N \frac{|(b_i - a_i) \times (c_i - a_i)|}{24} n_i e_d (((a_i + b_i) e_d)^2 + ((a_i + c_i) e_d)^2 + ((b_i + c_i) e_d)^2) \\ (d=1,2,3)$$

Ahol a_i, b_i, c_i az egyes háromszögek csúcspontjai, n_i pedig a hozzájuk tartozó normálvektor. A képlet használható konvex és nem konvex testeken is, tehát a közönséges testekben megvalósítva, örökléssel az atomokra ruházva, a program minden poliéderen ezt a módszert alkalmazza. Az tehát eljárás lineáris műveletigényű a háromszögek számában, azonban a testek elraktározzák az egyszer kiszámolt eredményt, így másodjára már konstans idejű a lekérdezés.

Átmérő számítás

A poliéder átmérőjének meghatározásánál fontos tulajdonság, hogy az egymástól legtávolabb eső pontok biztosan csúcspontok. Ezt kihasználva, elég minden p_i, p_j csúcspár távolságát megvizsgálni, majd venni közülük a legnagyobbat, az átmérő szakaszt pedig megfeleltethetjük a $p_i - p_j$ irányvektornak. Az eljárás így négyzetes műveletigényű a csúcsok számában. Felmerült az optimalizálás kérdése, annak kihasználása, hogy egy adott lap minden pontja egy síkon helyezkedik el, így a lapok szintjéről vizsgálva elég lenne egy lapon meghatározni a másik lap síkjától legtávolabbi pontot. A következő ábra igyekszik egy ellenpéldát érzékeltetni, ahol a piros háromszög a zöld négyzet síkjához legközelebbi pontjából húzható a legnagyobb távolságú szakasz a négyzet csúcsaihoz.



Atomok uniója

A közelítő algoritmus egyik végcélja a kellő atomok egyesítéseként kapott test előállítása. Ez művelet testek uniójának előállítását követeli meg. A test azokból a releváns atomokból egyesül, művelet eredménye olyan manifold tulajdonságú test mely nem tartalmaz felesleges, nem a határán levő lapokat. Az előkészületek már az atomok vágásai során 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.

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

```

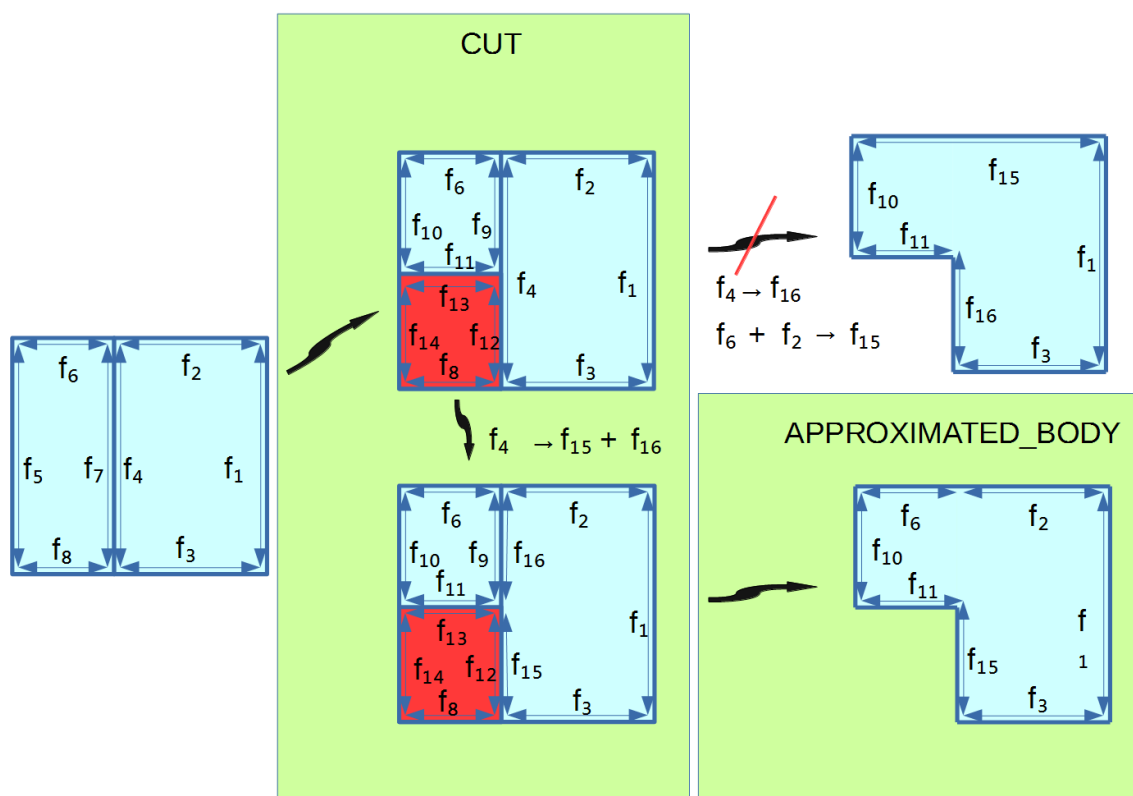
PROCEDURE APPROXIMATED_BODY(mode,fouriermin)

    garbage_collection();
    fouriers[0..n-1] := fourier(atoms[0..n-1])
    inds:=[]
    FOR i:=0..n-1 LOOP
        IF fouriers[i] >= fouriermin 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 eredetie



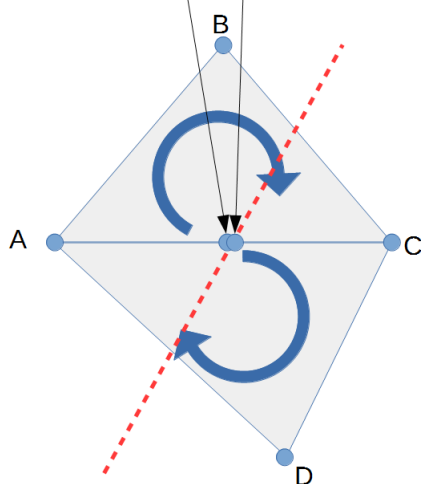
A test előállításának teljes folyamatát a fenti ábra szemlélteti. Ahogy az ábrán is megfigyelhető, az algoritmus során keletkezhetnek olyan felületek, melyeket a szükségesnél több lappal írunk le. Ezek száma azonban egy gyakorlati alkalmazás során elhanyagolhatónak tekinthető, valamint ha két lap összevonása 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 megköötést 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.

Általában $(1-t)A+tC = E1 \neq E2 = (1-t')C+t'A$



A hiba megoldását a csúcspontok állandó sorrendben való kezelése adta, kezdetben a központi tárolóbeli index alapján rendezést alkalmaztam, majd a vágósík negatív-pozitív oldalára esés szerint állításra cseréltem.

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.

A funkciók jól elkülönülnek az egyes osztályok között, így a tesztelést lényegében típusonként külön lehetett bontani. A tesztelés a fejlesztéssel párhuzamosan, szintenként történt, a könyvtárhoz mellékelt teszt kódok a régi tesztek közérthetőbbé tett változatai. A cpp fájlokon kívül a mappában található a gummybear.obj és a test.obj melyek a teszteléshez felhasználható fájlok.

Ha rendelkezünk g++ fordítóval, a példák fordítása az examples mappában a következő paranccsal történhet:

```
g++ -I "../include" -std=c++11 <test>.cpp
```

Az alábbiakban az examples mappa fájljainak tömör ismertetése olvasható.

vectortest.cpp	
	A vektorokon értelmezett műveleteket mutatja be egy előre megadott a és b vektor páron.
facetest.cpp	
	Lapokkal végezhető műveletekre ad példát. Bemutat vágásokat és súlypont számítást.
planetest.cpp	
	Síkok és vonalak alkalmazására, pontok velük történő osztályozására, metszési egyenes készítésre mutat példákat.
centroidtest.cpp	
	Egy dőlt piramis súlypontját határozza meg.
polygontest.cpp	
	Kétdimenziós sokszögek használatát bemutató program. Vágásokat, körbevágást, statisztikák számítását mutatja be.
polygraphtest.cpp	
	A metszetkép építéshez használható sokszögepítő algoritmusok működését mutatja be. Feltölti a gráfot, majd felépíti belőle a sokszögeket.
targettest.cpp	
	Betölti a gummybear.obj-t, meghatározza a súlypontját és térfogatát.
approxtest.cpp	
	Bemutatja, hogy a legmagasabb szintről nézve milyen lehetőségeink vannak az adatszerkezetek használatára. A test.objt felhasználva megmutatja a vágások lebonyolítását, statisztikák ellenőrzését.

Irodalomjegyzék

TODO Simon jegyzetre pontosan?

- [1] Gábor Fábián , Lajos Gergó: Adaptive algorithm for polyhedral approximation of 3D solids. Stud. Univ. Babes-Bolyai Math. Vol. 60, No. 2, 2015. pp 27591.
- [2] Gábor Fábián , Lajos Gergó: Fast Algorithm to Split and Reconstruct Triangular Meshes, Studia Universitatis Babes-Bolyai Series Informatica, Special Issue 1, pp. 90-102 (2014)
- [3] Christer Ericson: Real Time Collision Detection, Elsevier Inc., 2005, 1-55860-732-3
- [4] Sherif Ghali: Introduction to Geometric Computing, Springer-Verlag London Limited, 2008, 978-1-84800-114-5
- [5] Simon Péter: Bevezetés az analízisbe II. egyetemi jegyzet
- [6] <http://wwwf.imperial.ac.uk/~rn/centroid.pdf> Elérés dátuma: 2016.03.10
- [7] Ivan Sutherland, Gary W. Hodgman: Reentrant Polygon Clipping. Communications of the ACM, vol. 17, pp. 32–42, 1974
- [8] Ivan Sutherland et al., "A Characterization of Ten Hidden-Surface Algorithms" 1974, *ACM Computing Surveys* vol. 6 no. 1.