
# Virtuális fák számítógépes grafikai rendszerekben

Prim András
Konzulens: Dr. Szirmay-Kalos László

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék
2002

Alulírott Prim András, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen a forrás megadásával megjelöltem.

# Tartalomjegyzék

# Kivonat

Ez a diplomadolgozat az általam írt famodellező programról szól. Leírom, hogy milyen tudás és eszközkészlet állt rendelkezésemre a program elkészítéséhez, és hogy hogyan működik a program. Nagy vonalakban: a felhasználó faleírásából — mely lényegében egy formális nyelvtan — készít egy egyszerű hasábokból álló vázat. Ám ez a váz túl durva: még ha hengerekből állna is, a csatlakozási vonalaknál akkor is szögletes lenne. Finomítsuk hát ezt a vázat úgynevezett felületfelosztásos technikával. Ehhez viszont egy egybefüggő felület kell, viszont a váz egymást metsző hasábokból áll. Erre megoldás, hogy a csatlakozó ágak unióját képezzük, ezzel levágva a hasábok egymás belsejébe eső felületrészét. Mindemellett leveleket is elhelyezhetünk a fán, melyekkel nincs túl sok teendő. Hogy aztán lássuk is a fát, a modell előzetes képét OpenGL segítségével megjeleníti: ezzel megfelelő pozicióba állíthatjuk a fát, és „lefényképezhetjük”, azaz a beállítást elmenti RenderMan formátumban, amiből aztán egy megfelelő renderelő előállítja a kész képet.

# Abstract

This thesis is about the tree modeller program I wrote. I put down what knowledge and toolkit was available to me for making the program, and how the program itself works. In outlines: according to the tree description of the user — which is essentially a formal language — it produces a frame of prisms. But this frame is very coarse: even if it consisted of cylinders, the joints would be too cornered. So let us smooth this with the so colled sub-division technique. But this requires a contiguos surface, however the frame consists of intersecting prisms. The solution is that the program produces the union of the branches that have common base: this eliminates the part of the surface of the prisms which is inside an other prism. Besides, we can place leaves on the tree, but there is not much to do with them. In order we see the tree, the program displays it using OpenGL: with the help of this, we can position the tree as we like, and then make a "photograph" of it. This means that the program saves the model in RenderMan format, from which a proper renderer can make the final picture.

# Bevezető

Korunkban egyre inkább elterjednek a három dimenziós, virtuális világok. Ezek nagy része a valós világ másolata igyekszik lenni, vagy legalábbis az alapján épül fel. Egyfelől az ember alapvetően még mindig természeti lény, így hát a virtuális világokban is otthonosabban érzi magát, ha élőlényekkel népesítjük azt be; másrészt ha a valós világ egy darabját egy az egyben próbáljuk modellezni, akkor gyakran ki sem kerülhetjük a növényeket. Hogy a dolog elterjedtségét érzékeljük, gondoljunk csak bele, hogy mára már a lakáshirdetések képeiről sem hiányozhatnak a modellezett fák, bokrok.

Célom olyan program megírása volt, amely fák három dimenziós modelljét állítja elő, a felhasználónak lehetőleg minél nagyobb beleszólási lehetőséget biztosítva a kinézetbe. Ez jelenleg úgy valósult meg, hogy a felhasználó készíti el a fa generálását leíró fájlt. Nem volt cél a fa képének saját előállítása (csupán az előkép szintjén), hanem a modellt létező szabványnak megfelelő formára kellett hozni, és azt átadni a szabványt ismerő renderernek. Továbbá szívügyem volt, hogy a program Linux alatt (is) működjön, kizárólag ingyenes eszközöket felhasználva.

Jóllehet a famodellezés nem egy forradalmian új terület, de két okom is volt, amiért mégis ezt választottam. Elsőként az, hogy nem találtam könnyen elérhető ingyenes fa modellezőt Linux alá. A másik, hogy gyakorlatilag semmi tapasztalatom nem volt három dimenziós programozásban — ugyanis közvetlen a diplomamunka előtt jöttem rá, hogy ez az a terület ami érdekel — ezért nem mertem járatlan ösvényre térni. Ez azt is eredményezte, hogy a diplomamunkát az alapok elsajátítására is használtam, így szinte mindent magam írtam meg.

Jelölések és értelmezések: az előforduló vertexeket vastag, kis betűvel ($\mathbf{p}$), a vektorokat vastag, kis betűvel, felette nyíllal ($\vec{\mathbf{v}}$) jelölöm. A vertex egy három dimenziós pontja a térnek. A vektor szintén három dimenziós, de az értéke a végpontja és a kezdőpontja közötti különbség, bárhol is helyezkedjen el a térben. A mátrixokat vastag nagy betűvel ($\mathbf{M}$) jelölöm. A skalárokat kisbetűvel, az egyéb elemeket (háromszög, test, vonal) nagy betűvel írom.

Az első részben írom le mindazt, amit a diploma elkészítéséhez kívűlről igénybe vettem. A második rész a tervezésről és a program működéséről szól. A harmadik részben az elért eredményt értékelem. Végül a függelékben található két példa.

A program elérhető a http://impulzus.sch.bme.hu/~pbandi/tree címen.

# 1. fejezet

# Előzmények

## 1.1. A felhasznált tudás és eszközök

Ebben a fejezetben azt ismertetem, hogy a diplomamunka elkészítéséhez milyen tudás illetve eszköztár állt már rendelkezésemre. Ez öt elkülöníthető részre osztható, nevezetesen a Lindenmayer formális nyelv rendszerekre, koordinátageometriai ismereteimre, a felosztott felületekre (sub-division surface), az OpenGL-re, illetve a RenderMan interfészre. Ebben a fejezetben főként csak azt írom le, hogy mit tudtam ezekről, azt, hogy pontosan mire használtam őket, majd később, a program leírásában ismertetem.

### 1.1.1. Lindenmayer rendszerek

Az L-rendszerek gyakorlatilag formális nyelvek [1]. 1968-ban vezette be őket egy biológus: Aristid Lindenmayer, biológiai indíttatásból. Fő eltérésük a Chomsky féle nyelvtanoktól, hogy a produkciós szabályok nem egymás után hajtandók végre, hanem az egy mondatra alkalmazható összes szabályt egyszerre kell végrehajtani — hasonlóan ahhoz, ahogy egy élőlény sejtjei is párhuzamosan osztódnak. Eme tulajdonságukból adódik, hogy az általuk generálható nyelvek halmaza is eltér: például léteznek olyan nyelvek, melyeket környezetfüggetlen L-rendszerrel elő lehet állítani, viszont környezetfüggetlen Chomsky nyelvtannal nem.

Ezen kívűl a L-rendszerek nem különböztetnek meg terminálisokat és nem-terminálisokat. Minden jelre van produkciós szabály, ha valamely jelre ezt külön nem is adjuk meg, akkor alapértelmezésként olyan szabály érvényes rá, amely önmagára cseréli le. Ezért egy mondat generálása nem fejeződik be akkor, mikor már nem lehet további szabályt alkalmazni — ez ugyanis, az előzőek értelmében, csak az üres mondat esetére volna igaz — hanem végtelenségig folytatható.

Egy L-rendszert egy hármas határoz meg: az $ABC$-je; a kiindulási mondat, melyet *axiómának* is hívunk; valamint a *produkciós szabályok* halmaza. Mint említettem, az ABC azon elemeire, melyre nem adtunk meg explicite produkciós szabályt, az identitás szabályt értelmezzük.

Ezen túlmenően lehetőség van az L-rendszer elemeinek paraméterezésére, ha ezzel élünk, akkor nem egyes betűkön értelmezettek a produkciós szabályok, hanem a betűn és a paraméterén: ezeket a paramétereket pedig fel lehet használni a produkciós szabály törzsében az új paraméterek kiszámolásához, illetve a kész mondat értelmezéséhez. Globális paramétereket is megadhatunk, ez a lehetőség a C nyelv `#define` utasításához hasonlít.

Sztochasztikussá is tehetjük az L-rendszert, mégpedig úgy, hogy minden produkciós szabályhoz egy valószínűséget rendelünk, és minden esetben, amikor egy betűn szabályt kell végrehajtani, eme valószínűségek alapján sorsoljuk ki az adott esetben alkalmazandót.

Az előállított mondat értelmezésének egyik lehetősége a teknőc-reprezentáció. Ez esetben a mondat betűinek sorozata egy rajzoló teknőcnek szóló parancssorozat. Hagyományosan az egyes betűk értelmezése:

- `'F'`: a teknőc előrehalad, miközben húz egy vonalat.
- `'+' '-'`: a teknőc balra, illetve jobbra fordul.
- `'/' '\'`: a teknőc balra, illetve jobbra forog hossztengelye körül.
- `'&' '^'`: a teknőc fel, illetve le fordul.
- `!`: a vonalvastagságot állítja.

Nem paraméterezett esetben az utasításokhoz alapértelmezett egység rendelhető, paraméterezett esetben pedig a teknőc mozgásának mértékét az aktuális paraméter határozza meg. Értelmezhető ezen kívűl két karakter: a `'['` hatására a teknőc elmenti az aktuális állapotát, a `']'` hatására pedig visszatér hozzá. Ezek a karakter párok természetesen egymásba is ágyazhatók, és tulajdonképpen verem működést biztosítanak.

Jóllehet az L-rendszerek lehetőségei ezzel még nem értek véget, de programomban csak ezeket használtam fel.

### 1.1.2. Koordinátageometria

A program megírásakor csak három dimenziós tér koordinátageometriájával foglalkoztam, és az alábbi igazságokat használtam fel:

- Két egymásra merőleges vektor vektorszorzata 0.
- Minden $\vec{\mathbf{n}}$ vektor és *const* skalár pár meghatároz egy $P$, síkot, melyre igaz, hogy

$$\mathbf{v} \in P \quad \Leftrightarrow \quad \vec{\mathbf{n}}^T \cdot \mathbf{v} \; = \; const \tag{1.1}$$

Illetve, ha $\vec{\mathbf{n}}^T \cdot \mathbf{v} > const$, akkor $\mathbf{v}$ a sík által meghatározott két térfél közül abban van, melybe a $\vec{\mathbf{n}}$ vektor mutat — amennyiben a talppontja a $P$ síkon van.

- A tér két, nem egybeeső $\mathbf{a}$ és $\mathbf{b}$ pontja meghatároz egy $L$ egyenest, melyre a következő igaz:

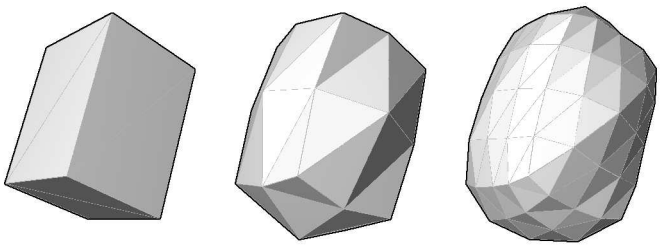
$$\mathbf{v} \in L \quad \Leftrightarrow \quad \mathbf{v} = \alpha * \mathbf{a} + (1 - \alpha) * \mathbf{b} \tag{1.2}$$

Ha ezen kívűl $0 \leq \alpha \leq 1$, akkor $\mathbf{v}$ az $\mathbf{ab}$ szakaszon helyezkedik el. Ha $\alpha = 0$, akkor $\mathbf{v} = \mathbf{b}$, ha $\alpha = 1$, akkor $\mathbf{v} = \mathbf{a}$.

Talán az Olvasó tisztában van a fenti összefüggésekkel, ám mindenképpen fel akartam sorolni, hogy milyen matematikára építettem.

### 1.1.3. Felosztott felületek

Egy poligonokból álló felületet az úgynevezett felosztással finomítani lehet — ahogy ez az 1.1. ábrán látható — hasonlóképpen, mint ahogy egy törtvonalat b-spline-nal, vagy egyebekkel lehet közelíteni [3]. Felosztásra is több technika létezik, én a butterfly módszert [2], illetve annak általam kicsit módosított változatát használtam.



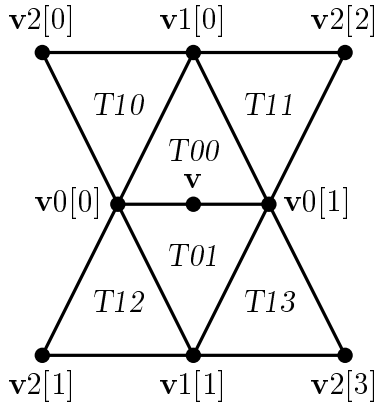
1.1. ábra. A kiindulási test és a butterfly felosztás első és második iterációjának eredménye.

Az alap butterfly felosztási módszer feltételezi, hogy a finomítandó felület háromszögekből áll, és az alábbi módon működik: minden iterációban a

felület minden élére kiszámolunk egy osztópontot, majd minden háromszögéből négy háromszöget csinálunk az új osztópontok felhasználásával. Egy **v** új osztópont poziciójának kiszámításához az 1.2. ábrán látható környező pontokat vesszük figyelembe, az alábbi súlyozással:

- **v**0[] elemeinek súlya $\frac{1}{2}$
- **v**1[] elemeinek súlya $\frac{1}{8} + 2w$
- **v**2[] elemeinek súlya $-\frac{1}{16} + w$

Ebben szerepel egy $w$ súly is. Ha ez –1/16 értékű, akkor az új pontok a régi élek szakaszfelezőire kerülnek, így a felület alakja nem változik; míg 0 felé haladva egyre nagyobb hatással lesz. (Az 1.1. ábrán w értéke 0.)




1.2. ábra. Új osztó vertexet befolyásoló pontok.

### 1.1.4. OpenGL

Az OpenGL a Silicon Graphics, Inc. bejegyzett védjegye, ezen kívűl ez egy eszközfüggetlen interfész a grafikai hardver és a programozó közt. Tulajdonképpen egy állapotgépről van szó, melynek az állapotát a programozó megfelelő utasításokkal változtathatja, illetve más utasításokkal grafikai primitíveket (vonalak, poligonok) rajzoltathat ki a grafikai eszközre — a ki-

rajzolás pedig a gép állapotától (aktuális szín, megvilágítási mód, transzformációk, stb. ) függ.

Azon kívűl, hogy a modell előzetes képét OpenGL segítségével jelenítettem meg, a dokumentációja [5] alapján saját koordináta-transzformációs függvényeket írtam, melyeket a modell építéséhez használtam. (Erről bővebben a 2.2.2. és a 2.3.2. fejezetekben.)

A megjelenítéshez egyébként használtam a GLUT eszköztárát is. Ez egy szintén platformfüggetlen csomag, és egy könnyen kezelhető utat biztosít OpenGL használatára egy ablakban [6]. (t.i. csupán meg kell vele nyitni egy ablakot, megadni az eseménykezelő függvényeket, utána elindítani a GLUT eseménykezelő ciklusát.)

### 1.1.5. RenderMan interfész

A RenderMan interfész a Pixar által definiált interfész modellező programok és fotorealisztikus renderelő programok közt [7]. Egyrészt egy C API-t határoz meg, másrészt egy bájtfolyam formátumot — ez utóbbi lehetőséget teremt a modell leírásának file-ban való tárolására. Mindkét megközelítés úgy épül fel, hogy egy grafikai primítív említésekor rendelkezésre áll minden információ annak megjelenítéséhez. A bájtfolyam formátum egyrészt lehet ember által olvasható is, de hatékonysági szempontból minden elem kódolható — akár egy bájtfolyamon belül váltogatva is.

## 1.2. Környezet

A fejlesztést az alábbi hardver- és szoftverkörnyezetben végeztem:

**Architektúra** : i686

**Operációs rendszer** : RedHat Linux 7.2 (Enigma), 2.4.7-10 kernelverzió

**Fordító eszközök** : gcc 3.0; GNU Make 3.79.1

**OpenGL meghajtó** : Mesa 3.4.2

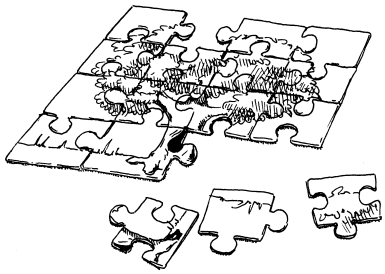
**RenderMan renderelő** : Blue Moon Rendering Tool 2.6

A program C nyelven íródott, az egyetlen felhasznált C++ elem a // komment.

A diplomadolgozat elkészítéséhez a LaTeX rendszert használtam.

# 2. fejezet

# A program



## 2.1. Tervezés

Mikor konzulensemnek megemlítettem, hogy fát szeretnék modellezni, rögtön kezembe nyomta a Lindenmayer rendszerekről [1] és a felület felosztásról [2] szóló irodalmat: ez nagyrészt meg is határozta, hogy miként fog működni a program. A Lindenmayer nyelvtan alapján felépül a fa, majd a felületét finomítással tesszük természetesebbé.

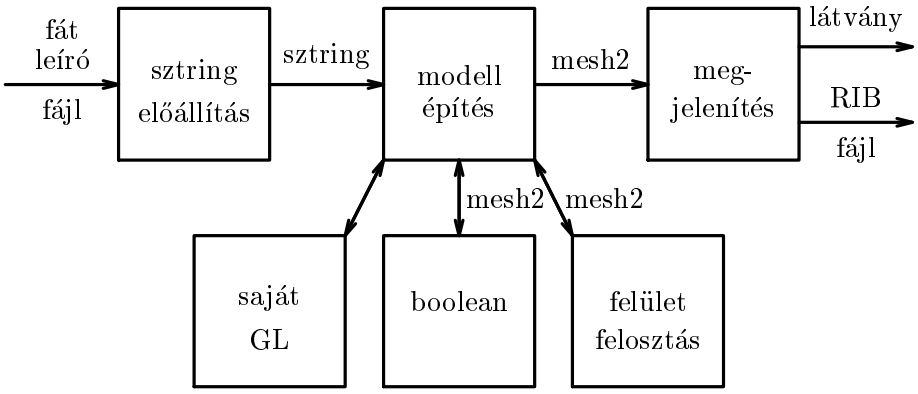
A tervezéskor a moduláris programozás elvét alkalmaztam, amit az tett könnyűvé, hogy a program a bemenettől a kimenetig egy lineáris lépéssorozatot hajt végre. A problémát az alábbi fő modulokra bontottam, melyek közötti interfészt megpróbáltam minél egyszerűbb szinten tartani:

1. Egy L-rendszer leírását beolvasva előállítja a rendszer valamely mondatát, ami egy sztring. Megfelelő leírás esetén ez a fát leíró sztring

teknőc reprezentációja, de akár egész más is lehet.

2. A teknőc reprezentáció sztringjét értelmezve előállítja a fa modelljét. Ez a modell egy általános test lesz, csak éppen fára hasonlít.

3. A test megjelenítése: egyrészt egy interaktív előképet állít elő, másrészt a végleges képet eljuttatja a szabványos interfészig.




2.1. ábra. A program moduljai.

Az első gondolatom ezután az volt, hogy a megjelenítést OpenGL-lel és GLUT-tal végezzem. Ez akkor azért volt vonzó, mert egyszerű felületen keresztül lehet velük három dimenziós ábrázolást programozni, és használatukkal hordozható marad a forrás. A GLUT viszont nem biztosít túl kifinomult felhasználói felületet, de ezt nem is bántam: inkább egy jobb modellt akartam előállítani esetleg spártai felület mellett, minthogy sok időt fordítsak ez utóbbi programozására, a modellezéstől véve el az értékes időt.

A fentiek közül a második modulba lehet beilleszteni a felület felosztást. A felület felosztás összefüggő felületet vár el, jobban mondva összefüggő felület esetén ad olyan eredményt, amilyet várunk, tehát a fa modelljét ennek megfelelően kell alakítani, mielőtt a finomításra térhetnénk — kérdés, hogy hogyan. A valós fa csomópontjaiban nem igazán lehet eldönteni, hogy melyik

ág meddig tart, csak kijjebb mondhatjuk rá, hogy ez már egy önálló ág. Hasonló probléma merül fel a virtuális fák csomópontjaiban: nem lehet tudni, meddig tartson a felülete, hogy az pont illeszkedjen a többi ág felületéhez. Ekkor jön a képbe az unió boolean művelet: alkossuk úgy az ágakat, hogy az ős ághoz illeszkedjenek, majd képezzük a közös őshöz csatlakozó ágak unióját: ez ugyanis pont arról gondoskodik, hogy a felület összefüggő legyen. Mind a felület felosztás, mind az unió művelet elég általános, ezért úgy gondoltam, hogy a későbbi újrafelhasználásuk hasznos lehet, tehát érdemes őket külön modulban megvalósítani. Ezeket nem almodulnak hívnám, mivel nem a három főmodul stafétabot átadás rendszerében működnek, inkább nevezném őket bedolgozó modulnak, vagy egyszerűen segédeszköznek, hiszen az egyik főmodul keze alá dolgoznak.

Ezen a ponton látszik, hogy érdemes készíteni egy általános célú test struktúrát, mely támogatja az unió és felületfelosztás műveleteket és a megjelenítést, és interfészként szolgál a két utolsó főmodul, illetve a felület felosztás és az unió művelet felé. A leírások alapján tudtam a következő elvárásokat: a megjelenítéshez szükség van a test háromszögeinek normálisaira, melyek kifelé kell hogy mutassanak; a felület felosztáshoz pedig a szomszédos háromszögek közt könnyű átjárást kell biztosítani. Ezen kívűl logikus volt a vertexeket és a háromszögeket külön tárolni, úgy hogy a háromszögek mutassanak a vertexeikre, mivel így ha a vertex változik, nem kell megkeresni a hozzá tartozó összes háromszögben az előfordulását. (Megjegyzem, hogy ezt a lehetőséget eddig még nem használtam ki.) Aztán a felület felosztáson és a megjelenítésen gondolkodva érdemesnek tűnt a test vonalait is külön tárolni, mert ekkor nem kell mindkét határolt háromszög esetén külön kiszámolni az adott oldalon beszúrandó osztópontot, illetve a drótvázas megjelenítés is egyszerűbb így. Ezeket figyelembe véve úgy döntöttem, hogy a kis helyfoglalás rovására a sebességet fogom növelni, és az egyes elemek közti átjárást mutatókkal segítem, mivel a memória egyre olcsóbb, egy napban viszont továbbra is csak 24 óra van. (A másik lehetőség az volna, hogy kimerítő kereséssel kutatjuk a megfelelő elemet. Például vagy mutatót

tartok fenn a szomszédos háromszögre, vagy minden alkalommal az összes háromszögek közül megkeresem azt, amelyik használja ugyanazt a két vertexet.) Visszatérve a háromszögek szomszédosságára: mikor felveszünk egy háromszöget, akkor a szomszédaira hivatkozó mutatóit ki kell tölteni. Ez gyorsabb, ha csak az új háromszög pontjait szintén használó háromszögeket nézzük, tehát jó, ha a vertexekből el lehet érni a háromszögeit. Mindezeket figyelembe véve alakítottam ki a 2.2.4. fejezetben ismertetett struktúrát.

A felület felosztás algoritmust [2] jól leírja, ennek tervezésével nem kellett foglalkoznom. Viszont az unió műveletet magam akartam megtervezni, bár tudom, hogy már számtalan helyen megvalósították. Elgondolkoztam, hogy geometriailag hogyan lehetne megfogalmazni, hogy mit várunk ettől a művelettől, ami nem más, mint hogy törölje ki azokat a felületrészeket, melyek a másik test belsejébe esnek. Legegyszerűbben algoritmikusan tudtam ezt elképzelni: keressük meg a metsző háromszögeket, majd a másik test normálisával ellentétes irányban induljunk el a másik test belsejébe, és onnan töröljük a felületet, amíg a másik test falába (azaz más metsző háromszögekbe) nem ütközünk. Ez ugyan nem kezeli azt, mikor az egyik test teljesen a másikon kívűl vagy belül van, de ilyen rendes fákban nem is fordul elő, ezért most ezzel a bonyolító tényezővel nem foglalkoztam. Mivel a test struktúra támogatja a szomszédos háromszögeken keresztül haladást, csak két problémát kellett megoldani: a metsző háromszögek, illetve metszetük megkeresését; valamint a metsző háromszögekből a szükségtelen darabok törlését. Illetve fogalmazzuk át: mivel az unió művelet után is szükség lehet a két eredeti testre, ezért a törlés helyett inkább építsünk egy új testet. Ekkor tehát a metsző háromszögek másik testen kívüli részét kell keresni, és a testből kifelé indulva felvenni a másik test szükséges háromszögeit. Az előbb említett két problémára a megoldást inkább a boolean művelet implementálásáról szóló fejezetben (2.2.5.) ismertetem, remélve, hogy így könnyebben érthető lesz.

A tervezés idejének végére maradt a program kiindulópontja, mégpedig az, hogy a fát leíró nyelvet a felhasználótól milyen szabványos, elemezhető

formában várjuk el. Ezt a formátumot is inkább az implementációról szóló fejezetben (2.3.1.) írom le, mivel ott gyakran hivatkozom majd a leírás elemeire, és nem szeretném, hogy sokat kelljen lapozgatni.

A továbbiakban a modulokat alulról felfelé építkezve ismertetem: kezdve a legegyszerűbb segédeszközöktől, haladva a három fő modul felé úgy, hogy lehetőleg a hivatkozások a már leírt részekre mutassanak.

## 2.2. Segédeszközök

### 2.2.1. Egyszerű dinamikus adatszerkezetek

A programhoz három, többé-kevésbé hasonló adatszerkezetet hoztam létre: *dynastring*,*stack*, *store*.

A *dynastring*, mint neve is sejteti, egy dinamikus sztring. Mivel számomra nem kellett több funkcionalitás, az alábbi dolgokat tudja: elem hozzáadása (ez lehet karakter, sztring, float) illetve az egész sztring egy bufferbe másolása. Ezt a fát leíró sztring előállításakor használom.

A *stack*-en természetesen végrehajthatók a *push* és a *pop* műveletek. Az adatokat folytonos memóriaterületen tárolja, így dinamikus tömb létrehozására is használható (és használom is), ám ez a tömb a következő *push* vagy *pop* művelettel elmozdulhat.

A *store* tulajdonképpen egy dinamikus tömb, ami garantálja, hogy az elemeinek címe nem változik építés közben, viszont csak a saját függvényeit lehet használni még nem hivatkozott elem címének megismeréséhez.

### 2.2.2. Saját OpenGL-szerű függvények

A fa modelljének építéséhez jól jön egy transzformációs mátrix. Az OpenGL több mátrixot is kezel, és megvalósítja rájuk a következő funkciókat:

- Külső mátrix betöltése. (Speciális esete az identitás mátrix betöltése.)
- A mátrix megszorzása külső mátrix-szal.

- A mátrixhoz egy eltolás hozzáadása. (Tulajdonképpen előállít egy eltolás mátrixot, és azzal megszorozza az eredeti mátrixot. Így ha egy vertexet megszorzunk a keletkezett mátrix-szal, akkor azon elvileg először végrehajtódik ez az eltolás, majd utána az eredeti mátrix transzformációi [4].)
- A mátrixhoz egy forgatás hozzáadása. (Ugyanaz elmondható, mint az eltolásnál)
- Veremműveletek: push, pop.

A mátrixok $4 \times 4$ méretűek, hogy az eltolás is egy mátrix – vertex szorzással elvégezhető legyen, de ennek érdekében a vertexeknek is kell lennie egy negyedik koordinátájuknak, ami alapértelmezés szerint 1. Hogy ezt gyorsan megértsük, álljon itt a szorzat vertex x koordinátájának kiszámolási módja:

$$v_1' = M_{11} \cdot v_1 + M_{12} \cdot v_2 + M_{13} \cdot v_3 + M_{14} \cdot v_4$$

Amennyiben $v_4 = 1$, akkor $v_1'$-hez egyszerűen hozzáadódik a $M_{14}$ értékű, x tengely irányú eltolás. Mivel a vertexek negyedik koordinátáját csak mátrix – vertex szorzás esetén használtam, és minden esetben megfelelt az alapértelmezés szerinti 1 érték, ezért nem is tároltam ezt a koordinátát.

A rajzoló teknőc (ld. az 1.1.1. fejezetben) állapotát pont ezekkel a műveletekkel változtatjuk. Ám az OpenGL a transzformált vertexeket nem adja vissza, csak a megjelenítés folyamatában használja őket, a fa modelljéhez viszont szükség van ezekre a vertexekre, ezért a szükséges függvényeket magam is megvalósítottam.

A felsorolt függvények megírásában segítségemre volt az OpenGL specifikáció [5], mely — mivel a meghajtó programok íróinak is útmutatásul szolgál — megadja az eltolás és forgatás mátrixok kiszámolásának módját. A veremműveletekhez természetesen a 2.2.1. fejezetben említett *stack*-et használtam.

Ezen kívűl szükség volt egy mátrix – vertex és egy mátrix – vektor szorzás függvényre. Bár mind a vertex, mind a vektor egy három dimenziós tömb,

közöttük az alábbi különbség van: a vertex a tér egy pontja, tehát rajta az eltolást el kell végezni; míg a vektornak valójában két végpontja van, csakhogy a talppontja az egyszerűség kedvéért mindig az origó, ezért a tárolt végponton csak a forgatást kell elvégezni. Ezt úgy oldottam meg, hogy a vektor szorzásakor csak a mátrix felső $3 \times 3$ elemét használtam.

### 2.2.3. Koordinátageometria

*Háromszög normálisa*: az **abc** háromszög és $\vec{\mathbf{n}}$ normálisa közt igaz, hogy

$$\overrightarrow{\mathbf{ab}} \cdot \vec{\mathbf{n}} = \mathbf{0} \ , \ \overrightarrow{\mathbf{ac}} \cdot \vec{\mathbf{n}} = \mathbf{0}$$

A harmadik hasonlóan felállítható egyenlet lineárisan függ ezektől, tehát vele nem sokra megyünk. Legyen $\overrightarrow{\mathbf{ab}} = (a, b, c)$ , $\overrightarrow{\mathbf{ac}} = (d, e, f)$. Ekkor az egyenletrendszer egy megoldása a következő:

$$\vec{\mathbf{n}}_x = b \cdot f - c \cdot e \ , \ \vec{\mathbf{n}}_y = c \cdot d - a \cdot f \ , \ \vec{\mathbf{n}}_z = a \cdot e - d \cdot b$$

A függvény ezt a vektort normalizálja. Így számolva a normális akkor fog felénk mutatni, ha az (**a b c**) pontokat az óramutató járásával megegyező irányban látjuk sorban. Ha ez nem felel meg, akkor például a (**b a c**) pontsorrendű háromszögre számolt normális pont ellentétes irányú lesz.

*Háromszög oldalának normálisa*: többször szükség lesz olyan vektorra, mely merőleges a háromszög egyik oldalára, és párhuzamos a háromszög síkjával. Könnyen belátható, hogy ha az $\vec{\mathbf{n}}$ normálisú **abc** háromszög **ab** oldalának normálisát keressük, akkor az megegyezik az $\mathbf{a}(\mathbf{a}+\vec{\mathbf{n}})\mathbf{b}$ háromszög normálisával, amit az előző bekezdés szerint ki tudunk számolni. Ilyen pont sorrend esetén egyébként egy, a háromszögből kifelé mutató vektort kapunk.

### 2.2.4. A mesh2

A kifinomultabb test struktúrájának a *mesh2* név jutott, mivel a *mesh* nevet egy jóval egyszerűbb struktúra foglalta. A *mesh2* fő elemei a következőek: három *store*, mely a vertexeket, az éleket illetve a háromszögeket

tárolja, indexelő szerkezet a vertexekre, valamint a testen végrehajtandó transzformációs mátrix.

A könnyebb programozás érdekében rengeteg mutatót tartalmaznak az elemek:

- Egy vertex mutat a hozzá csatlakozó első vonalra.
- Egy vonal mutat a két oldalán található háromszögekre, ezen kívűl
- Egy vonal mutat mindkét végpontjának következő vonalára. Ezek szerint egy vertex mutat az első hozzá csatlakozó vonalra, ez utóbbi a következő, a vertexhez csatlakozó vonalra, és így tovább, így egy láncolt listát képezve, mely elemei a keletkezésük sorrendjében kerülnek becsatolásra.
- Egy háromszög mutat a három csúcs vertexére, valamint
- Egy háromszög mutat a három határoló vonalára, végül
- Egy háromszög mutat a három szomszédos háromszögre. Ez utolsó csak kis megtakarítást eredményez, hiszen a szomszédját úgy is megkaphatnánk, hogy a határoló vonalnak vesszük azt a határolt háromszögét, mely nem a vizsgált háromszögünk.

Mivel a *store*-ba helyezett elemek memóriahelye nem változik, ezeket a mutatókat később nem kell frissíteni.

Ezeken kívűl mind a három elemfajta tartalmaz nyolc bitnyi flag-et is, melyet a függvények kedvükre használhatnak, illetve a háromszög tárolja a felületi normálisát is. A felületi normális mindig a testből kifelő mutat — ez egyrészt az OpenGL-es megjelenítéshez, másrészt a boolean művelethez (2.2.5) kell.

A vertex indexelő szerkezet egy csomópontonként nyolc felé ágazó fa: az adott csomópont által mutatott vertex a koordináta tengelyekkel nyolc térrészre osztja a koordinátateret, az egyes ágak ezekbe mutatnak.

Talán a mutatók felsorolásánál már feltűnt, hogy *feltételezem, hogy egy vonal mentén nem találkozik kettőnél több háromszög.* Ez a fa építésénél nem is szabad, hogy előforduljon.

A test vertexeinek illetve felületi normálisainak használatakor a tisztességes eljárás azokat a külön erre való függvényekkel lekérdezni, hiszen a transzformációs mátrix tartalmától függően a tárolt poziciójuktól eltérő helyen lehetnek. Ez főként két test együttes használatakor fontos, például a boolean művelet esetén.

A *mesh2*-höz tartozik egy OpenGL kirajzoló függvény is. Ez vagy a test drótvázát, vagy az egész testet rajzolja ki.

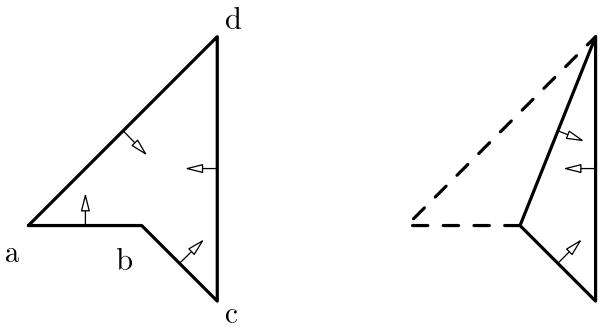
### A mesh2 építése

A felmerülő igényeknek megfelelően *vertex* hozzáadására egy olyan függvény van, mely csak akkor szúr be új vertexet, ha azon a helyen nincs már eleve egy vertex, egyébként a már létező vertex címével tér vissza; *vonal* és *háromszög* beszúró függvénye viszont olyan, mely nem ellenőrzi, hogy van-e már ott elem. A vonal létezését a háromszög beszúró függvény ellenőrzi (különálló vonal beszúrására nem merült fel igény), és azt feltételezem, hogy a testet építő figyel arra, hogy ne adja hozzá ugyanazt a háromszöget kétszer.

Egy *háromszög* felvétele tehát a következőképp történik: megkeressük a három csúcs vertexét a vertextárban (ezt gyorsítja az indexelő szerkezet), illetve beszúrunk újat, ha nem találtunk megfelelő vertexet. Ezután azokra a határoló vonalakra, melyek mindkét végpontja benne volt a vertextárban, megkeressük valamelyik végpont vertexének láncolt listájában az említett vonalat. Ha megtaláltuk, akkor a vonal egyik oldalán már korábban is volt háromszög, tehát a régi és az új háromszög, valamint a vonal szomszédos háromszög mutatóit kitöltjük. Ha nincs még meg az adott vonal, akkor természetesen bevesszük a testbe, és eme új vonal csak az új háromszögre fog mutatni. Ezzel be is került az új háromszög.

Ezen kívül létezik egy függvény, mely határoló vonalaival megadott, akár konkáv, de nem lukas *poligon* hozzáadását végzi. A határoló vonalaknak

rendelkezniük kell egy-egy vektorral is, melyek a poligon belsejébe mutatnak. A függvény sorra kiválaszt két szomszédos vonalat, melyek vektorai az általuk meghatározott háromszögbe mutatnak — ezzel elkerüljük a konkáv oldalpárokat — és az általuk meghatározott háromszögbe nem esik másik vonal — ez is konkáv poligonnál fordulhat elő. (Valójában ama párok közül, melyekre ez a kettő teljesül, azt választja, mely vonalai a legkisebb szöget zárják be: ettől azt várom, hogy „arányosabbak" lesznek a keletkező háromszögek, ami később, a sub-divisionnál jól jöhet.) Ha megvan a vonalpár, akkor az általuk közrezárt háromszöget a fentebbi fejezetben ismertetett háromszög hozzáadó függvénnyel felveszi a testbe, majd a vonalpárt eltávolítja a listából, és a távolabbi pontjaikat összekötő vonalat beveszi. Így folytatja, iterációnként egyel kevesebb vonallal, míg kettőnél több vonal van a listában. Feltételezem, hogy a megadott poligon zárt, ekkor pedig a végén maradó két vonal megegyezik, csak a normálisuk mutat ellenkező irányba.




2.2. ábra. Háromszög kiválasztása poligonból.

A 2.2. ábra szemlélteti a megfelelő vonalpár kiválasztását. Az ab–bc pár nem jó, mert konkáv, a cd–da pár azért nem jó, mert az általuk határolt háromszögbe két vonal is beleesik. A fenmaradó két vonalpárból véletlenszerűen a bc–cd párt választottam. A két nemkívánatos tulajdonságot a következőképpen ellenőrzöm:

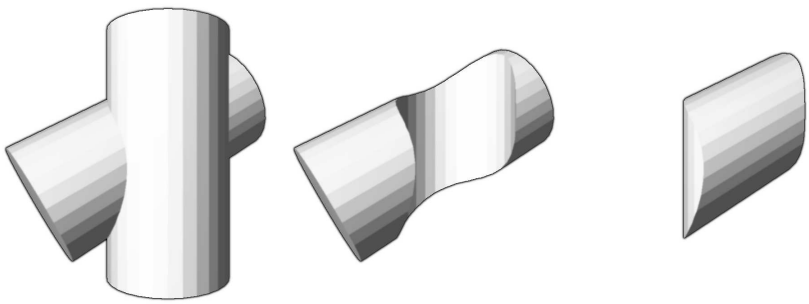
**Konkávitás :** Konvex vonalpár esetén mindkét vonal, és vektora, mint normális vektor által meghatározott sík „fölött" kell lennie a másik vonal

távolabbi pontjának. Ez, az 1.1.2. fejezetben írtak alapján, egyszerűen ellenőrizhető.

**Beleeső vonalak :** A vonalpárhoz generálható a harmadik oldal is, vektorral együtt (a vektor számításáról ld. 2.2.3.). Egy pont beleesik a háromszögbe, ha mindhárom vonal, és vektora, mint normális vektor által meghatározott sík „fölött" van.

### 2.2.5. Boolean művelet

Programom legnehezebb része a boolean művelet volt, mivel kötöttem magam ahhoz, hogy egészen magam találjam ki és valósítsam meg. A művelet lényege, hogy két test unióját, metszetét, vagy különbségét képezze (Mint az a 2.3. ábrán látható).



2.3. ábra. Boolean műveletek: unió, különbség, metszet.

Először az unió műveletet ismertetem, majd leírom, hogy milyen egyszerű ezt átalakítani metszet és különbség műveletté.

Az unió művelet az alábbi lépéseket hajtja végre:

1. Megkeresi a két test háromszögeinek metszet szakaszait.
2. Eme szakaszok alapján a metszett háromszögeket darabolja.
3. Végül megkeresi az unióba szükséges, de nem metszett háromszögeket.

### Metszet szakaszok keresése

Az unió művelet az egyik test minden egyes háromszögét párba állítja a másik háromszög összes háromszögével, és ezeket a párokat vizsgálja, hogy metszik-e egymást, illetve megadja a metszés szakaszát. A metszetet mindkét testnek külön tárolja, mivel, bár a metsző szakasz természetesen megegyezik a két háromszögre, a kisérőinformációk eltérnek: ezekről eme alfejezet végén szólok. A metszet keresését a következőképpen teszi: mindkét háromszög összes oldalára megkeresi, hogy az adott oldal által meghatározott egyenes hol metszi a másik háromszög síkját. Nevezzük a két háromszöget A-nak és B-nek, az A szögeit **a**, **b**, **c**, nek, az A normálisát $\vec{\mathbf{n}}_A$-nak, a B-ét $\vec{\mathbf{n}}_B$-nek. A B háromszög $P_B$ síkjának egyenlete ekkor, az 1.1. egyenlet alapján:

$$\mathbf{p} \in P_B \quad \Leftrightarrow \quad \vec{\mathbf{n}}_B^T \cdot \mathbf{p} \; = \; const,$$

ahol *const* megkapható, ha **p**-be a B háromszög valamely szögét behelyettesítjük. Az A háromszög **ab** oldala által meghatározott $L_{\mathbf{ab}}$ egyenes egyenlete pedig az 1.2. egyenlet alapján

$$\mathbf{p} \in L_{\mathbf{ab}} \quad \Leftrightarrow \quad \mathbf{p} = \alpha * \mathbf{a} \; + \; (1 - \alpha) * \mathbf{b} \tag{2.1}$$

Mi pedig azt a **p** pontot keressük, mely mindkét egyenletnek megfelel. Ha a második egyenlet jobboldalát behelyettesítjük az első egyenletben **p** helyére, akkor $\alpha$-ra a következő adódik:

$$\alpha = \frac{const - \vec{\mathbf{n}}_B^T \cdot \mathbf{b}}{\vec{\mathbf{n}}_B^T \cdot \mathbf{a} - \vec{\mathbf{n}}_B^T \cdot \mathbf{b}}$$

Ha eme tört nevezője 0, az azt jelenti, hogy az **ab** szakasz párhuzamos a B háromszög síkjával. Amennyiben $\alpha$ 1 és 0 közötti, akkor a metszéspont a háromszög oldalán van — nekünk erre van szükségünk — , egyébként azon kívűl. Innét már a 2.1. egyenletbe helyettesítve $\alpha$-t megkapjuk a metszéspontot.

Ha mindkét háromszögnek két-két oldalát metszi a másik háromszög síkja, akkor továbbmehetünk, keresve a két metszés szakasz leghosszabb

közös szakaszát, egyébként biztos, hogy a két háromszög nem metszi egymást. A közös szakasz kereséséhez kiválasztom az egyik szakaszt — ezt nevezzük $L_A$-nak — , és ismét az 1.2. egyenlethez nyúlva megvizsgálom, hogy a másik szakasz — ennek a neve legyen $L_B$ — végpontjai hogyan viszonyulnak hozzá, az $\alpha$ értéke alapján. Ez a viszony a négyféle lehet:

- Az $L_B$ szakasz mindkét végpontjára $\alpha$ kisebb 0-nál vagy mindkettőre nagyobb 1-nél. Ez azt jelenti, hogy a két szakasznak nincs közös pontja, tehát a háromszögek nem metszik egymást.

- Az $L_B$ szakasz egyik végpontjára $\alpha$ kisebb 0-nál, a másikra $\alpha$ nagyobb 1-nél. Ekkor az $L_A$ a két szakasz leghosszabb közös része. (Mint a 2.4. ábrán.)

- Az $L_B$ szakasz egyik végpontjára $\alpha$ 0 és 1 közé esik. Ekkor az $L_B$ a két szakasz leghosszabb közös része.

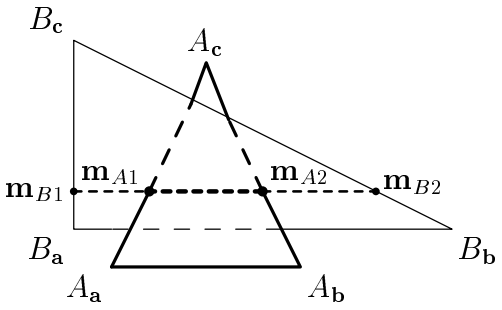
- Az $L_B$ szakasz egyik végpontjára $\alpha$ 0 és 1 közé esik, a másikra azon kívűlre. Ekkor a leghoszabb közös szakasz egyik végpontja $L_B$ azon végpontja, melyre $\alpha$ 0 és 1 közé esik. A másik végpontja az $L_A$ szakasz 1.2. egyenlet szerinti **a** végpontja, ha a másik $\alpha$ nagyobb 1 nél; illetve az egyenlet szerinti **b** végpontja ha a másik $\alpha$ kisebb 0-nál.

Azon kívűl, hogy megvan a két háromszög metszet szakasza, még két információra szükség lesz a későbbiekben: a közös szakasz végpontjai hol helyezkednek el az A illetve a B háromszögön, illetve, hogy merre mutat a másik háromszög normálisa. Tehát az A háromszöget tartalmazó test az alábbi adatokat kapja a metszet szakaszról:

- A szakasz két végpontjának abszolút koordinátája.

- Az egyes végpontok viszonya az A háromszöghöz.

- A B háromszög normálisának az A háromszög síkjára eső vetülete (illetve ennek normalizáltja).



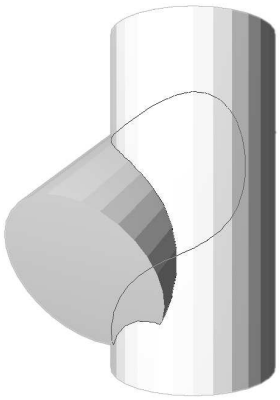

2.4. ábra. A és B háromszögek metszete.

Az elhelyekedés a következő hét érték valamelyikét veheti fel: „nincs" (ez hibát jelez), „belső", „**a** csúcs", „**b** csúcs", „**c** csúcs", „**ab** oldal", „**ac** oldal", „**bc** oldal". A 2.4. ábrán például, mikor keressük az A háromszög oldala – B háromszög síkja metszéspontokat ($\mathbf{m}_{A1}, \mathbf{m}_{A2}$), akkor tudjuk, melyik metszéspont melyik oldalon vagy csúcson van (esetünkben **ac** oldal, **bc** oldal). Ha a leghoszabb közös szakaszba beválasztunk egy ilyen végpontot, akkor arról tudjuk, hogy az A háromszöghöz hogy viszonyul, a B háromszöghöz való viszonyát viszont ki kell találnunk. Ez nem nehéz, mivel az A és a B háromszög metszet szakasza ugyanarra az egyenesre esik. Így, ha a B háromszög metszet szakasza végpontjainak ($\mathbf{m}_{B1}, \mathbf{m}_{B2}$) típusa — mint az ábrán is — „**ac** oldal" és „**bc** oldal", akkor az összes köztük lévő pont, tehát a leghoszabb közös metszet szakasz előbb említett végpontja is „belső" típusú. Ezzel szemben, például, ha a B háromszög metszet szakasza végpontjainak típusa „**a** csúcs" és „**b** csúcs", akkor a közöttük lévő pontok típusa „**ab** oldal", de minden más típuspárra is megadható a köztes pontok típusa.

A 2.2.3. fejezetben leírtak szerint számolom a B háromszög normálisának az A háromszög síkjára eső vetületét : veszem az ($\mathbf{m}_{A1}, \mathbf{m}_{A2}, \mathbf{m}_{A1} + \vec{\mathbf{n}}_A$) háromszög normálisát. Mivel ez a pontok sorrendjétől függően két irányba is mutathat, ezért az 1.1. egyenlet és a hozzá fűzött megjegyzés alapján ellenőrzöm, hogy az új vektor és az $\vec{\mathbf{n}}_B$ vektor szorzata nagyobb-e nullánál. Ha nem, akkor az új vektor nem egy irányba mutat $\vec{\mathbf{n}}_B$-nel, ezért fordított pontsorrenddel újraszámolom a normálist. (A normális számoló függvény

egységnyi hosszú vektort ad vissza, tehát a normalizálással nem kell foglalkozni a továbbiakban.)

Ez a normális segít eldönteni, hogy a metszet szakasz melyik oldala kerül az unióba. Mint írtam, a *mesh2* háromszögeinek normálisai a testből kifelé mutatnak. Az unióba a két test azon felületét kell bevenni, mely nem a másik test belsejében van, azaz egy metszet háromszög azon felét, melybe a másik háromszög normálisa (illetve annak vetülete) mutat.



2.5. ábra. Metszet szakaszokból álló áthatás görbe.

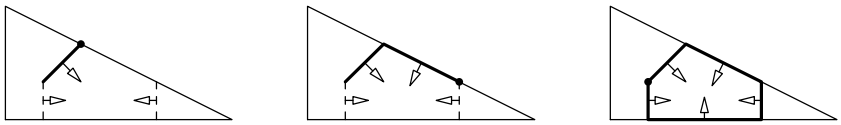
### Háromszögek darabolása

A következő lépés, hogy minden olyan háromszöget, melyet metsz a másik test legalább egy háromszöge, a metszés szakaszok alapján darabolom, és a keletkező kis háromszögeket hozzáadom az építendő unió testhez. Jelenleg nem kezelem helyesen az olyan metszés szakaszokból álló hurkot, mely teljes egészében egy háromszögön belül van, azaz ami egy lukat képez a háromszög belsejében — ilyen alkalmasint az építendő fában nem is fordulhat elő. Lássuk viszont azt, hogy mit csinál a program!

Minden háromszöggel külön foglalkozik: addig keres egybefüggő poligonokat, míg az összes, az adott háromszögre eső metszet szakaszt fel nem használta. Ezeket a poligonokat a 2.2.4. fejezetben ismertetett függvénnyel

adja hozzá az új testhez. Egy poligon kiválasztása a következőképpen megy végbe:

Választ egy még nem használt metszet szakaszt. Ezután keres ehhez kapcsolódó metszet szakaszokat, vagy azok hiányában oldal darabokat, míg az elsőnek választott metszet szakaszhoz vissza nem ér. A kapcsolódó metszet szakasz keresése triviális: végignézi a háromszöghöz tartozó, még fel nem használt szakaszokat, és ha valamelyikük egyik végpontja egybeesik az előző szakasz szabad végével, akkor meg is van a folytató szakasz. Valamivel komplikáltabb a megfelelő oldaldarab kiválasztása, ha nem találtunk folytatólagos metszet szakaszt.



2.6. ábra. Poligon építése metszet szakaszok alapján.

Itt lép képbe a metszet szakaszzal járó információ, hogy melyik vége hol helyezkedik el a háromszögben — ugyanis ez alapján keressük azt az oldalt, amely darabjával folytathatjuk az előző szakaszt, feltéve hogy ez a típus nem „nincs" vagy „belső". Más esetekben kétfelé indulhatunk: pl. egy „**a** csúcs" vagy egy „**bc** oldal" típusú végből a **b** és **c** csúcsok felé. Hogy melyik felé, azt viszont a metszet szakaszhoz kapott normálisból dönthetjük el, ez a normális ugyanis mindig az új testbe felveendő poligon belsejébe mutat. Ezért afelé a pont felé kell indulni, mely az előző szakasz normálisa által meghatározott sík fölött helyezkedik el. Ezt az 1.1. egyenlet és megjegyzése alapján úgy dönthetjük el, hogy a normálist (mely tárolt formájában origó kezdőpontú, tehát egy origón átmenő síkot határoz meg) megszorozzuk a folytatandó szakaszvégpontból a csúcsokba mutató vektorral (mely szintén origó kezdőpontú): amelyiknél ez nagyobb nullánál, azt kell választani.

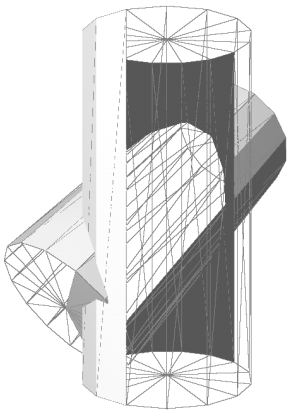
Ezzel még nem fejeztük be, mert ellenőrizni kell, hogy a folytatandó

szakaszvégpont és a választott csúcs között van-e metszet szakasz végpont, ugyanis ez esetben csak a legközelebbi ilyenig tarthat az új szakasz. Tehát sorra kell venni a még nem érintett szakaszokat és a poligont kezdő szakaszt (ha ez utóbbit nem tennénk, akkor esetleg körbe-körbe szaladgálhatnánk a háromszögön, mivel a poligon keresésének leállófeltétele, hogy a legutoljára választott szakasz végpontja egybeesik a poligont kezdő szakasz kezdőpontjával), és az 1.2. egyenletet a 2.2.5. fejezetben is említett módon felhasználva megkeressük az adott oldalon fekvő (ezt a metszet szakasz tárolt típusából tudjuk) legközelebbi pontot tartalmazó szakaszt. Ha ez még nem használt szakasz, akkor a következő szakasz keresésénél automatikusan ezt a szakaszt fogjuk választani, ha ez a kezdőszakasz, akkor el is értük a poligonkeresés leálló feltételét. Az oldaldarabot ezután az eme algoritmusban használt szakasz formátumúra kell hozni, azaz meg kell adni a végpontja típusát és generálni kell a normálisát, amit a 2.2.3. fejezetben írt módon teszünk.

### Nem darabolt háromszögek keresése

Miközben a háromszöget daraboljuk, a program figyeli a kapott szakasz-végek tulajdonsága alapján, hogy bevettünk-e egy teljes oldalt csúcstól csúcsig. Ekkor ugyanis az azon az oldalon szomszédos háromszöget is be kell venni az unióba. Ha ezt nem tennénk, csak a metsző háromszögek darabjait tartalmazná az unió, és úgy nézne ki, mint az a 2.7. ábrán látható.

Tehát ha csúcstól csúcsig szakaszt talál, akkor azt az oldalát megjelöli, és a darabolás után a darabolt háromszöget is megjelöli, hogy már érintette (minderre a háromszög flag-jét használja). Mikor végzett a darabolással, sorra veszi a darabolt háromszögeket, és a megjelölt oldalakon elindulva elkezdi elárasztásos algoritmussal bevenni a test háromszögeit. Eközben megjelöli az érintett háromszögeket, és arra nem megy tovább, ahol már érintett háromszöget talál. Így minden háromszöget legfeljebb egyszer vesz az unióba, és mivel a darabolt háromszögek is érintettként vannak jelölve, nem fog az elárasztás átfolyni az áthatás görbe túloldalára.



2.7. ábra. Két henger drótváza és metsző háromszögeik unióba tartozó darabjai.

### Unió művelet átalakítása metszetté és különbséggé

Ez, mint említettem, igen egyszerű manőver. A metszet művelet esetében a metszet szakaszok normálisát kell csak megfordítani, hiszen annál pont a másik test belsejébe eső háromszögekre van szükség. Ezután a háromszögek darabolása automatikusan a háromszögek megfelelő felét veszi be az unióba, ennek következtében az elárasztás is jó irányba kezdődik el.

A különbség művelet csak egy picit bonyolultabb. Vegyük az A test minusz B test műveletet. Ebben az esetben az A test B testen kívüli és a B test A testen belüli felületére van szükségünk, tehát csak a B test esetében kell a metszet szakaszok normálisát megfordítani. Ezen kívül a B testből vett háromszögek normálisait is meg kell fordítani a különbség testben, mivel azok alapesetben az A test, egyúttal a különbség test belsejébe mutatnak.

Ezek alapján a négy műveletet (unió, metszet A minusz B, B minusz A) egyetlen függvény is végre tudja hajtani, a paraméterként kapott művelet típus alapján fordítgatva a megfelelő normálisokat.

### 2.2.6. Felület felosztás

A felületfelosztás teljes módszere le van írva [2]-ben, az általam használt részt az 1.1.3. fejezetben ismertettem, azt csupán a saját adatszerkezetre alkalmazni kellett. Ez nem volt olyan nehéz feladat, mivel az adatszerkezetet direkt ennek a támogatására hoztam létre. Ezek szerint az algoritmus a következő lépéseket teszi (az elnevezések értelmezése az 1.2. ábrán látható) :

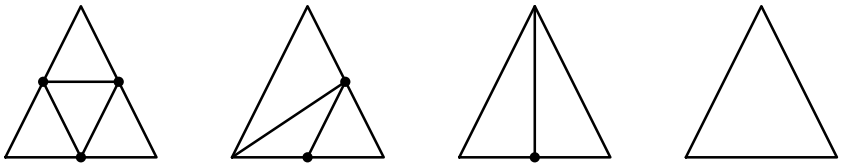
1. A test minden háromszögének minden oldalához foglal helyet egy-egy osztóvertexnek.

2. A test vonalain végigmenve mindegyikhez kiszámolja az osztópontot. A $\mathbf{v}0[]$ pontokra a vonal mutat, a $\mathbf{v}1[]$ pontokat a vonal által mutatott két *T00* és *T01* háromszögeken keresztül, a $\mathbf{v}2[]$ pontokat pedig eme háromszögek által mutatott szomszédokon keresztül éri el. Az osztópontot az előző pontban említett pufferbe, mindkét háromszöghöz elmenti.

3. A test háromszögein és az új pufferen végigmenve létrehozza az új osztott háromszögeket, és hozzáadja a finomított testhez.

Az osztópont számolásnál, ha nem létezik a *T00* vagy *T01* háromszög (mert esetleg nem folytonos a test felülete), akkor az osztópont a $\mathbf{v}0[0]$ $\mathbf{v}0[1]$ szakasz felezőpontja lesz. Ha a *T1x* háromszögek valamelyike nem létezik, akkor a megfelelő $\mathbf{v}2[]$ pontként a feltételezett háromszöggel szomszédos oldal (pl. *T10* esetén a $\mathbf{v}0[0]$ – $\mathbf{v}1[0]$ oldal) felezőpontját veszi föl.

Paraméterként megadható a w súly (1.1.3. fejezet) és egy határérték, melynél rövidebb oldalakon nem hoz létre osztópontot. Ezek szerint lehet olyan háromszög, melynek nem mind a három oldalán van osztópont. Ezt a 2.8 ábra szemlélteti.



2.8. ábra. Három, kettő, egy és nulla osztópont esetén képzett háromszögek.

## 2.3. A fa építése

### 2.3.1. A fát leíró sztring előállítása

Itt következik a fát leíró fájl szerkezetének leírása. A <x*> alakú elemek jelentése: semmi, vagy <x> elemek sorozata. A <x+> alakú elemek jelentése: egy, vagy több <x> elem sorozata.

```
<file>       ::= <leafdef> <paramlist> <axiomdef> <rulelist>

<NL>         ::= a soremelés karakter (0x0a vagy 0x0d)
<blank>      ::= space vagy vízszintes tab karakter
<white>      ::= <blank> | <NL>
<alnum>      ::= egy betű vagy szám
<digit>      ::= egy szám
<float>      ::= <digit+> | <digit+> '.'  <digit+>
<anybutNL>   ::= bármilyen karakter, kivéve <NL>
<anybutNLq>  ::= bármilyen karakter, kivéve <NL> és '"'
<anybutq>    ::= bármilyen karakter, kivéve '"'
<comment>    ::= '#' <anybutNL*> <NL>
<NLorc>      ::= <blank*> <NL> | <blank*> <comment>

<leafdef>    ::= <NLorc*> <triangle*>
<triangle>   ::= <blank*> '[' <blank*> <vertex> <blank+>
                 <vertex> <blank+> <vertex> <blank*> ']'
<vertex>     ::= <coordx> <blank+> <coordy> <blank+> <coordz>
<coordx>     ::= <float>
<coordy>     ::= <float>
<coordz>     ::= <float>

<paramlist>  ::= <NLorc*> <parameter*> <NL>
<parameter>  ::= <blank*> <pname> <blank+> <pval> <NLorc> |
                 <blank*> <pname> <blank+>
```

```
                <pval> <blank+> <pval2> <NLorc>
<pname>         ::= <alnum+>
<pval>          ::= <float>
<pval2>         ::= <pval>

<axiomdef>      ::= <NLorc*> <blank*> '"' <axiom> '"' <NLorc>
<axiom>         ::= <anybutNLq*>

<rulelist>      ::= <NLorc*> <rule*> <NL>
<rule>          ::= <blank*> <token> <blank*> ':'
                <blank*> <head> <blank*> '->' <bodydef> <NLorc>
<token>         ::= nyomtatható karakter
<head>          ::= '*' | 'p(' <prob> ')'
<prob>          ::= <float>
<bodydef>       ::= '"' <body> '"'
<body>          ::= <anybutq*>
```

Ennek megértésében valószínűleg sokat segít az A. függelékben leírt példa.

Maga a beolvasás menete nem túl elegáns, és nem is foglalkozik a Lindenmayer nyelven belüli szintaktikai hibákkal: az ilyenek inkább csak abból derülnek ki, hogy az előállított fa egyáltalán nem úgy néz ki, ahogy terveztük. A beolvasás során az axiómából (`<axiom>`), a szabályok fejéből (`<head>`) és törzséből (`<body>`) már most eltávolítja a whitespace karaktreket a további feldolgozás előtt: tehát eme elemeket kedvünkre tagolhatjuk ilyen karakterekkel.

A program jelenleg környezetfüggetlen, paraméteres L-rendszereket kezel, kétféle sztochasztikus tulajdonsággal: egyrészt, mint az 1.1.1. fejezetben írtam, az egyes szabályoknál meg lehet adni, hogy milyen valószínűséggel hajtódjanak végre; másrészt ha a rendszer valamely paraméterét (`<pname>`) két számmal adjuk meg (`<pval>, <pval2>`), akkor annak minden kiértékelésekor egy véletlenszámot kapunk, mely egyenletes eloszlású a két megadott

szélsőérték között.

Mint az a `<token>` fenti definiciójából sejthető, az L-rendszer egyes tokenjei egyetlen karakter hosszúságúak lehetnek, a paraméterlistájuk — ha van ilyen — zárójelek közé zárt, vesszővel elválasztott paraméterek listája. Jelenleg csak az első nyolc paramétert kezeli a program, a többit figyelmen kívül hagyja. Egy paraméter a négy alapművelettel kiszámolható kifejezés, melynek tagjai `<float>`-ok, `<paramlist>`-ben deklarált `<pname>`-k, vagy a token, melyen a szabályt végrehajtjuk, paraméterére hivatkozás lehet (`$1`,`$2`, ... , `$8`). Természetesen az utóbbinak nincs értelme az axióma leírásában. Nem létező paraméterre hivatkozás értéke mindig 0. Jelenleg a műveletek balról jobbra, precedencia és zárójelezés nélkül értékelődnek ki.

A fát leíró sztringet tehát a következőképp állítja elő a program: Kiértékeli az axióma paramétereit, az axiómát átmásolja, hogy a fa aktuális mondata legyen, majd a kért számú származtatási lépést végrehajtja. Egy származtatási lépésben végigmegy a fa aktuális mondatán, sorra véve a mondat tokenjeit, és paramétereiket. (A paraméterek az aktuális mondatban mindig kiértékelt formában, azaz egyszerű lebegőpontos számokként vannak jelen.) Ha talál szabályt, mely illik a soron következő tokenre és paraméterlistájára, akkor a szabály törzsében a paramétereket kiértékeli, és az új mondathoz fűzi a kiértékelt törzset. Ha nem talál megfelelő szabályt, akkor alapértelmezés szerint a tokent paraméterlistástul (ha van neki) hozzáfűzi az új mondathoz.

Az alkalmazandó szabály kiválasztására az alábbi technika működik egyelőre: minden szabályhoz tartozik egy valószínűség, mely vagy a fejében megadott `<prob>`, vagy ennek hiányában 1. (Valójában a kerekítés hibájának kiküszöbölésére ez utóbbi érték 2.0, ennek használhatósága hamarosan kiderül.) Mikor egy származtatási lépésben egy új token-hez ér a program, akkor 0-ra állít egy számlálót. A számláló értékéhez sorra hozzáadja ama szabályok valószínűség értékét, melyek `<token>`-je illik a tokenre. Ha egy szabálynál a számláló eléri az 1-et, akkor azt a szabályt fogja végrehajtani. Láthatóan a felhasználóra van bízva, hogy megfelelően adja meg a valószínűség értékeket.

Ez akkor lehet jó, ha a gyorsan meg akarja nézni, milyen volna a fa egy sztochasztikus szabály nélkül: ekkor egyszerűen 1 fölé állítja az előtte álló azonos `<token>`-ű szabály valószínűségét.

### 2.3.2. Fa modelljének előállítása a leíró sztring alapján

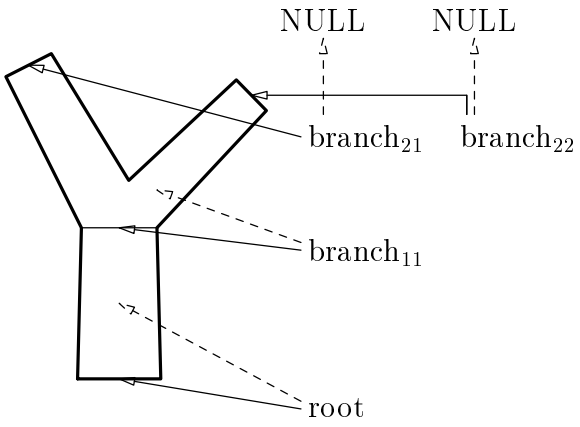
Mindenekelőtt szeretném megemlíteni, hogy — mint a fát leíró fájl szerkezetében látszik is — leveleket is kezelek, ezt tehát be kellett venni a teknőc reprezentációba (1.1.1). Erre egy újabb tokent vezettem be, ez az `'L'` betű, két paraméterrel: a levél pozíciója a legutóbbi `'F'` által megadott ágszakasz hossza mentén, illetve az ugyane ágszakasz tengelye körüli elforgatás szöge. A pozició relatív adandó meg: 0 értéknél van az ágszakasz töve, 1-nél a vége, de ezen kívüli pozició is megadható. Az elforgatás szöge fokban adandó meg. A levél elforgatása 0 szögének helye a felhasználó számára véletlenszerűnek tűnhet, de egy ágszakaszon belül változatlan. A levél koordinátáit a `<leafdef>`-ben egy olyan koordinátarendszerrel kell megadni, melynek origója az aktuális ágszakasz felületén helyezkedik el, x tengelye az ágszakasz keresztmetszetét érinti, y tengelye párhozamos az ágszakasz palástjával, z tengelye pedig átmegy az ágszakasz középpontján. Eme koordinátarendszer egysége megegyezik az ágszakaszt definiáló koordinátarendszer egységével. A levél pontos elhelyezéséről hamarosan lesz még szó.

A (1.1.1). fejezetben ismertetett tokeneknek egy-egy paraméterük van, ez forgatás esetében a forgatás szöge fokban, `'F'` esetén az ágszakasz hossza, `'!'` esetén pedig az ezután következő ágszakaszok sugara. Ez utóbbi két méret megadásához segítséget nyújthat, hogy a megjelenítő ablak egy $2 \times 2$ méretű négyzet ebben a koordinátarendszerben, a rajta át látható látószög pedig 90 fok.

A fát leíró sztringet olvasó teknőc egy elágazó struktúrát épít, melynek az elemei az ágszakaszok. Minden ágszakasz mutat a közvetlen ősére és a belőle közvetlen származó ágszakaszokra. Ezen kívül egy ágszakasz külön tárolja a lezáró sokszögének pontjait, és a belőle kiinduló ágszakaszok alkotta testet. A kezdőelem a gyökér, ennek a lezáró sokszöge egyben a törzs alap sokszöge,

a gyökérelemben tárolt test pedig maga a törzs első szakasza. Ez a kissé furcsa megoldás, hogy az ág teste az ősében tárolódik, azért született, hogy az építés közben a közös pontból származó ágak unióját lehessen tárolni: ezt egy helyen érdemes tenni, méghozzá az ágak közös pontjában, azaz az ősben. Hogy az uniókat csomópontonként külön építem, annak az az oka, hogy az unió művelet lépésszáma a bemeneti testek háromszögszámával exponenciálisan növekszik, az pedig, hogy így a fa ágai nem nőhetnek össze, nem nagy veszteség.

Ezzel párhuzamosan megépíti a levelek testét: ez nem lesz egy összefüggő felület, csak háromszögek halmaza. Ezen később már nem is hajt végre műveletet.

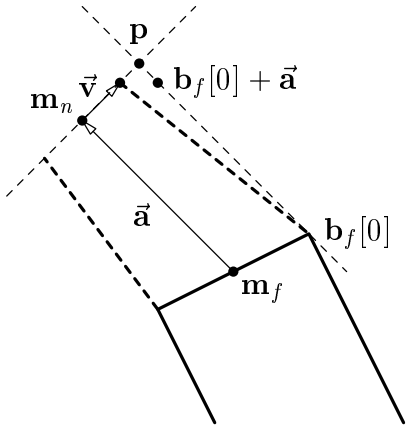



2.9. ábra. Az ágszakaszokhoz tartozó lezáró sokszögek és testek.

A teknőc tehát az olvasott tokeneknek megfelelően a pozicióját és irányát az OpenGL-szerű függvényekkel (2.2.2. fejezet) tárolja. Először az új ág lezáró sokszögét úgy állítottam elő, hogy a megfelelő sugarú, az xy tengelyek síkján, középen elhelyezkedő sokszöget a teknőc állapotát képviselő transzformációs mátrixszal vittem a helyére. Ám mivel a teknőc saját hossztengelye körül is elfordulhat, ezzel a módszerrel az ágakban esetleg csavarodás léphet fel. Ezért olyan eljárást alkalmaztam, mely ezt kivédi, és a következőképp működik: Nevezzük az ős és az új ágszakasz lezáró sokszögének közepét $\mathbf{m}_f$-

nek és $\mathbf{m}_n$-nek; pontjaik számát n-nek; sugaraikat $r_f$-nek és $r_n$-nek; pontjaikat $\mathbf{b}_f[n]$-nek és $\mathbf{b}_n[n]$-nek; a teknőc állapotát hordozó mátrixot abban a pillanatban, mikor az új ág végén áll $\mathbf{M}_n$-nek. Az új ág lezáró sokszögének képzésekor ismertek $\mathbf{m}_f$, $r_n$ és $\mathbf{M}_n$.

$$\mathbf{m}_n = \mathbf{M}_n \cdot (0,0,0,1)^T \vec{\mathbf{a}} = \mathbf{m}_n - \mathbf{m}_f$$




2.10. ábra. Új lezáró sokszög szerkesztése.

Itt $\vec{\mathbf{a}}$ az új ágszakasz tengely-vektora: az kell, hogy az új sokszög síkja erre merőleges legyen. Eme vektorra alkalmazva az 1.1. egyenletet, és a $(\mathbf{b}_f[0], \mathbf{b}_f[0] + \vec{\mathbf{a}})$ pontpárra alkalmazva az 1.2. egyenletet egy olyan egyenletrendszert kapunk, mely megoldása egy olyan $\mathbf{p}$ pont, mely az előállítandó lezáró sokszög síkján, az $\mathbf{m}_n$ ponttól $r_f$ távolságra van, ezen kívűl a $\mathbf{pb}_f[0]$ szakasz párhuzamos az ős és az új ág középpontját összekötő szakasszal: tehát nem okozna ágcsavarodást. Legyen:

$$\vec{\mathbf{v}} = (\mathbf{p} - \mathbf{m}_n) * r_n / r_f$$

Ez egy olyan vektor, mely párhuzamos az előállítandó lezáró sokszög síkjával, és $r_n$ hosszú. Ha ezt elforgatjuk az $\vec{\mathbf{a}}$ vektor körül 360/n fokonként,

majd $\mathbf{m}_n$-nel eltoljuk, akkor megkaptuk a kívánt sokszög pontjait. Eme szerkesztés talán túl bonyolultnak tűnik, de egyszerűbben nem tudtam garantálni a csavarodásmentességet.

Ha már megvan az ős és az új ág lezáró sokszöge is, akkor közöttük létrehozza a program a 2n háromszögből álló testet, és az ős eddigi ágaival uniót képez. Az ős ágait lecseréli erre az új unió testre.

Ha `'L'` tokennel találkozik, akkor leveleket fog elhelyezni az aktuális ágszakaszon, a token paramétereinek megfelelően. Ehhez használja a nemrég említett $\vec{\mathbf{a}}$ vektort, valamint egy $\vec{\mathbf{l}}$ vektort, mely az $\mathbf{m}_n$ pontból a $\mathbf{b}_f[0]\mathbf{b}_n[0]$ pontok által meghatározott egyenes, az `'L'` token paramétere által meghatározott pontjába mutat. Konkrétan egy olyan mátrixot állít elő, mely a pontokat $\vec{\mathbf{l}}$-el eltolja, $\vec{\mathbf{a}}$ körül a paraméterben adott szöggel elforgatja, végül $\mathbf{m}_n$-nel eltolja.

Mikor a leíró sztring végére ér, az épített elágazó struktúra ágszakaszait összemásolja egyetlen testbe. Ezen a testen lehet alkalmazni a felület felosztást.

### 2.3.3. Felhasználói felület

Mikor elkezdtem a diplomamunkát, még semmilyen tapasztalatom nem volt Linux-os felhasználói felületek programozásában, így nem lett túl kidolgozott és barátságos a program ezen része. Csupán a GLUT alapvető funkcióit használja, és a következőképp működik:

A fát leíró fájlt parancssorban kell megadni. A használható billentyűk :

**w s :** feljebb / lejjebb mozgatja a fát

**a d :** balra / jobbra mozgatja a fát

**r f :** távolabb / közelebb mozgatja a fát

**' / :** feljebb / lejjebb mozgatja a fényforrást

**[ ] :** balra / jobbra mozgatja a fényforrást

**; . :** távolabb / közelebb mozgatja a fényforrást

**+ - :** növeli / csökkenti a származtatási lépésszámot

**0 1 2 :** a felület felosztás lépésszámát állítja

**? :** a látható beállítást a 'tree.rib' fájlba menti

A származtatást minden állításkor az axiómától kezdi, tehát ha a leírásunk véletlent is használ, akkor visszatérve ugyanarra a lépésszámra, már valószínűleg más fát kapunk eredményül. Az egér bal gombját lenyomva és az egeret húzva forgatni lehet a fát.

# 3. fejezet

# Értékelés

Figyelembe véve, hogy a 3D grafika területén teljesen kezdő vagyok, nem vagyok elégedetlen a programommal. Nem tűnik hiábavalónak a sebesség előnyben részesítése sem az OpenGL használata, sem a *mesh2* struktúra esetén. Bonyolultabb fák esetén (amilyen a függelékben is látható) hardveres gyorsítással is elmaradt a megjelenítés sebessége a 24 FPS-től; másrészt az ekkora fák építése, de főleg a felület felosztás másodpercekben mérhető időt vesz igénybe.

Ám a program nem mentes hibáktól. A legszembetűnőbb a boolean műveletben van: mivel a koordinátákat lebegőpontosan kezelem, nem lehet olyat kérdezni, hogy két koordináta megegyezik-e, csak olyat, hogy egymás epszilon sugarú környezetében vannak-e. Ezt sajnos nem sikerült tökéletesen kézben tartani, ezért a művelet egyes háromszögeket egyszerűen kihagy az

eredményből. Ez ott szokott néha előfordulni, ahol a két test egy-egy vonala metszi egymást — ezt megpróbálhatjuk elkerülni, ha a fa leírásban nem szabályos szögeket használunk.

Másik két szembetűnő probléma a felületfelosztás kapcsán merül fel — jóllehet ezek nem programozási hibák, hanem a butterfly módszer tulajdonságai — nevezetesen:

- Ha a felosztandó felület háromszögei nagyon aránytalanok, akkor az eredmény elég csúnya lesz. Ezt úgy védhetjük ki, hogy hasonló széles és magas ágszakaszokat írunk elő — ha hosszabb ágat szeretnénk, akkor építsük fel több elégazás nélküli ágszakaszból.

- A butterfly módszer interpoláló, azaz a régi test vertexei azonos pozícióban kerülnek az új testbe: ez a csatlakozások ívességet rontja.

A jövőben a következőkkel volna hasznos foglalkozni:

- A boolean művelet kijavítása.

- A felület felosztás approximációssá tétele.

- A felhasználói felület fejlesztése. (Menün keresztüli fájl beolvasás, az L-rendszer paramétereinek futás közbeni állítása, stb. )

- Folytató ágak mellett kinövő ágak kezelése. (Jelenleg az új ág és az ős ág lezáró sokszögét köti össze a program, ami derékszög környéki elfordulás esetén már elég torz ágakat eredményez.)

- A tokenek paramétereinek kiértékelése legalább zárójelezést felismerjen.

- Más modellezők felé felületet biztosítani szabványos modell formátumba mentéssel, ugyanis a RenderMan interfészt gyakorlatilag csak a renderelők olvassák, a modellezők csupán írják.

# A. Függelék

# Példa fák

## Első fa

```
# definition of leaf : one triangle per line
[-0.4 0.0 0.2  0.4 0.0 0.2  0.1 -0.2 0.6]
[-0.6 0.1 0.6  0.2 0.0 0.7  0.0 0.2 0.12]
[0.6 0.1 1.3  0.5 0.0 1.4  0.3 0.2 1.7]
[-0.6 0.4 1.7  0.2 0.4 1.8  0.2 0.4 1.13]
[0.0 0.4 1.3  0.5 0.4 1.4  0.3 0.5 1.7]
[-0.2 0.3 1.7  0.4 0.0 1.8  0.2 0.2 1.13]


# parameters
width  0.1 0.15  # initial radius of branches
length 0.8 1     # initial length of a branch segment
rot    0 40      # variance of rotation
sh     20 40     # pitch shift
wrate  1.1 1.2   # rate of width growth in one derivation
lrate  1.1 1.2   # rate of elongation in one derivation
rd     0 20      # bend of a branch
lp     0 1       # leaf position
```

```
la     0 360     # leaf angle

# axiom
"B(rd,width*wrate,width,0)A"

# B parameters:
# $1: bend of branch       $2: width at beginning
# $3: initial pitch shift
# rules
A : p(0.7) -> "\(70+rot) [ B(rd,width,sh) A ]
     \(70+rot) [ B(rd,width,sh) A ]

     \(70+rot) [ B(rd,width,sh) A ]
     \(70+rot) [B (rd,width,sh) A ] "
A : p(0.4) -> "\(100+rot) [ B(rd,width,sh) A ]
     \(100+rot) [ B(rd,width,sh) A ]
     \(100+rot) [ B(rd,width,sh) A ]"
B : * -> "!($2) +($3) F(length) L(lp, la) L(lp, la)
          !($2 * wrate + $2 / 2) +($1) F(length)
          L(lp, la) L(lp, la) L(lp, la) L(lp, la)
          !($2 * wrate) +($1) F(length) L(lp, la)
          L(lp, la) L(lp, la) L(lp, la) L(lp, la)"
L : p(0.3) -> ""
F : * -> "F($1*lrate+0.1)"
! : * -> "!($1*wrate+0.1)"
```

Látható, hogy az egyes szekciókat (levél definíció, paraméterek, axióma, szabályok) egy-egy üres sor választja el egymástól, ezért viszont szekción belül nem lehet üres sor, ám kommentek bárhol elhelyezhetők. Szintén jól látszik, hogy a szabályok törzse whitespace karakterekkel tetszőlegesen tördelhető, sztringen belül akár üres sorok is lehetnek.

A levelek egymástól független háromszögek. Z koordinátájuk jelenti a

A.1. ábra. Az első fa renderelt képe.

fa felületől való távolságukat: ez némelyiknek nullánál jóval nagyobb, ezzel olyan hatást érek el, mintha egy nem látható vékony ág végén ülne. Ugyanis minden ilyen kis ágat valóban ágként modellezni túl sok poligont igényelne, és amúgysem igen látszana. Azért van több levél háromszög, hogy több rétegben helyezkedhessenek el az ágtól kifelé, és hogy ne kelljen nagyon sok `'L'` tokent írni.

A `'B'` tokenből egy három szegmensből álló görbe ág lesz. Több szegmensből áll azért is, hogy az egyes szegmensek közel ugyanolyan szélesek legyenek, mint hosszúak: ez szebbé teszi a felület felosztást. Az első és második paraméterre azért is szükség van, hogy az ágon belül ugyanazt az értéket használhassuk: Ha a `$1` helyére rendre `rd`-t írnánk, akkor azt minden esetben újrasorsolná, és egy ágon belül változna a görbület mértéke.

Az öregebb ágak kopaszítását úgy végzem, hogy minden lépésben az előző lépésben már meglévő leveleket véletlenszerűen kitöröltetem.

## Második fa

```
# definition of leaf : one triangle per line
[-0.4 0.0 0.2  0.4 0.0 0.2  0.1 -0.2 0.6]
[-0.6 0.1 0.6  0.2 0.0 0.7  0.0 0.2 0.12]
[0.6 0.1 1.3  0.5 0.0 1.4  0.3 0.2 1.7]
[-0.6 0.4 1.7  0.2 0.4 1.8  0.2 0.4 1.13]
[0.0 0.4 1.3  0.5 0.4 1.4  0.3 0.5 1.7]
[-0.2 0.3 1.7  0.4 0.0 1.8  0.2 0.2 1.13]


# parameters
width  0.1 0.15  # initial radius of branches
length 0.8 1     # initial length of a branch segment
rot    0 80      # variance of rotation
sh     10 20     # pitch shift
div    30 60     # divergence of B
wrate  1.1 1.2   # rate of width growth in one derivation
lrate  1.1 1.2   # rate of elongation in one derivation
rd     0 5       # bend of a middle branch
lp     0 1       # leaf position
la     0 360     # leaf angle

# axiom
"A"

# V parameters
# $1 width at beginning     $2 bend of branch
# rules
A : * -> "V(width,rd)[
           /(50+rot)[+(sh)V(width/4*3,sh)B]
           /(50+rot)[+(sh)V(width/4*3,sh)B]
```

```
           /(50+rot)[+(sh)V(width/4*3,sh)B]
           /(50+rot)[+(sh)V(width/4*3,sh)B]]A"
B : * -> "[+(sh)&(div)V(width/4*3,sh)B]
          [+(sh)^(div)V(width/4*3,sh)B]
          [+(sh)&(div-55/2)V(width/4*3,sh)B]"
V : * -> "!($1)F(length)L(lp,la)L(lp,la)
          !($1/wrate+$1/2)+($2)F(length)L(lp,la)
          !($1/wrate)+($2)F(length)L(lp,la)L(lp,la)"
L : p(0.3) -> ""
F : * -> "F(lrate*$1+0.1)"
! : * -> "!(wrate*$1+0.1)"
```

A.2. ábra. A második fa renderelt képe.

# Irodalomjegyzék


[1] *The Algorithmic Beauty of Plants* — Przemyslaw Prusinkiewicz; Aristid Lindenmayer
Springer-Verlag

[2] *3D Games* — Alan Watt — 2000

[3] *Számítógépes Grafika* — Dr. Szirmay-Kalos László — 2001
ComputerBooks

[4] *A számítógépes grafika alapjai IBM PC-n* — Székely Vladimír; Poppe András — 1997
ComputerBooks

[5] *The OpenGL Graphics System: A Specification (Version 1.2.1)* — Mark Segal; Kurt Akeley — 1999
ftp://ftp.sgi.com/opengl/doc/opengl1.2/opengl1.2.1.ps

[6] *GLUT manual pages*
http://reality.sgi.com/opengl/spec3/spec3.html

[7] *The RenderMan Interface (version 3.2)* — Pixar — 2000
http://www.pixar.com/renderman/developers_corner/rispec/
/rispec_pdf/RISpec3_2.ps

[8] *Blue Moon Rendering Tools User Manual* — Nikos Dragos — 2000
http://www.exluna.com/products/bmrt/bmrtdoc/2.6/index.html