Óbudai Egyetem
Neumann János Informatikai Kar
Alkalmazott Informatikai Intézet

TUDOMÁNYOS DIÁKKÖRI DOLGOZAT

Fák növekedésének szimulálása környezeti hatások fényében

Szerző: **Tóth Bence Tamás**
mérnökinformatikus BSc. szak, V. évf.

Konzulens: **Dr. Szénási Sándor**
egyetemi docens

Budapest, 2018.

Tartalomjegyzék

Összefoglalás

A fák növekedése bonyolult és rengeteg tényező által befolyásolt folyamat. E dolgozat célja ennek a folyamatnak a vizsgálata, és minél valósághűbb szimulálása.

A szimuláció során a program figyelembe veszi a fafajra jellemző alaki tulajdonságokat, az őt körülvevő fizikai környezetet, illetve a bejövő fény irányát, amely meghatározására grafikus kártyával gyorsított sugárkövető algoritmus szolgál (a fát befolyásoló tényezők listája könnyedén bővíthető további környezeti hatásokkal). A szimuláció egy időben több fát, akár kisebb facsoportot is képes szimulálni, ahol az egyes fák növekedésük során hatással vannak mind saját magukra, mind a környező fák fejlődésére.

A szimuláció eredményeként a fa több életszakaszában, akár évente is megjeleníthető. Az így létrejövő háromdimenziós famodellek jól tükrözik az őket meghatározó faji jellegzetességeket. Ezek a faji jellegek viszont a környezeti hatások által torzítva jelennek meg, így valóságosabb képet adva a fa adott környezetbe való beilleszkedésének menetéről.

Abstract

The growth of a tree is a complicated process affected by a lot of external factors. The goal of my thesis is to study and simulate this behaviour.

During the simulation the program takes count of the properties of the species and the surrounding environment for the particular specimen, including the direction of the incoming light. These directions are computed with a GPU accelerated path tracing algorithm. The list of environmental modifiers can be easily changed or even updated with new factors. The simulation can handle multiple or even a smaller group of trees at once. During their growth, the trees affect both their own and their neighbours' development.

The state of the simulation can be displayed multiple times during the process, meaning that it is able to show the steps of the growth by the trees' lifecycle or even yearly. The resulting three dimensional tree models reflect the traits of their species. These traits however, are distorted by environmental effects, therefore giving a more realistic model of the trees' integration to their environments.

1. Bevezetés

1.1. A probléma

Nap mint nap körbevesznek bennünket a fák, a legtöbb ember viszont nagyon keveset tud róluk, és még a mindennapi munkájuk során velük kapcsolatba kerülő szakemberek se tudnak mindent. Ez persze érthető, mivel ha mélyebben megvizsgáljuk, akkor kiderül, hogy a fák növekedése bonyolult és rengeteg tényező által befolyásolt folyamat. Ezért napjainkban még egy megoldásra váró problémának tekinthető az, amikor azt kellene meghatározni, hogy mi lesz egy fával évekkel azután, hogy elültették egy megadott helyre. A valóságban pedig ez egy gyakori kérdés, legyen szó akár egy kertes ház udvarának befásításáról, egy park tervezéséről, vagy akár egész erdők telepítéséről.

Kevés szakember van, aki meg tudja mondani, hogy hová milyen fát érdemes ültetni. És ez a tudás is gyakran csak tapasztalatokon alapuló, nagyvonalú becslést tesz lehetővé. A felelősség viszont nagy, hiszen egy esetleges tévedés amellett, hogy jelentős vagyoni károkat okoz, csak nagyon nehezen és időigényesen korrigálható. Ezért szükség lenne egy olyan eszközre, amely pontosan meg tudja határozni, hogy egy megadott helyen, a környezet ismeretében (környező objektumok, további fák, fényviszonyok, stb.), egy megadott fa fajta milyen növekedést fog produkálni a következő években, évtizedekben.

A napi gyakorlati alkalmazás mellett a tudományos életben is szükség lenne minél pontosabb növekedési szimulációkra. A napjainkban lejátszódó éghajlati változások hatása is nehezen jósolható a fákra nézve. Ezekre a problémákra megoldást jelenthet egy szimuláció, amely a fák növekedését modellezi, figyelembe véve az őket körülvevő környezetet, és természeti hatásokat.

Természetesen számos alkalmazás érhető már el ezen a területen, ezek azonban két, egymástól elkülönülő kategóriába sorolhatók:

- Modellező eszközök: ezek általában egyszerű fa generátorok, amelyek megadott paraméterek alapján emberi szemnek nagyon kellemes fa modelleket tudnak generálni. Azonban ezek nem szimulációk, nem veszik figyelembe a környezeti hatásokat, pusztán az esztétika a cél.

- Matematikai modellek: összetett matematikai modellek, amelyek a növekedés

egy-egy (rész)folyamatát modellezik (energiaellátás, stb.), esetleg célorientált erdészeti programok (fahozam tervezés). Ezek viszont nem adnak egy kézzelfogható és látható 3D modellt, illetve a környezeti hatásoknak csak egy szűk (az adott részfolyamat szempontjából releváns) körét veszik figyelembe.

Szükség lenne tehát egy olyan megoldásra, amely ötvözi a két kategória funkcionalitását, tehát egy jól definiált matematikai modell alapján (a fa fajta és a környezeti hatások figyelembevételével) egy 3D modellt épít fel. Ilyen alkalmazást azonban az irodalomkutatás során nem találtam.

Ennek talán az is lehet az oka, hogy a szükséges folyamat nagyon számításigényes. Már néhány éves fák esetén is több órás futásidőkre lehet szükség, és ez az idővel exponenciálisan növekszik (miként maga a fa is). Ezen viszont segíthetnek a napjainkban már elérhető párhuzamosítási technikák, ezek közül is érdemes kiemelni a grafikus kártyák által elérhető adatpárhuzamos gyorsítási lehetőségeket.

1.2. A kutatás célja

A kutatás célja egy olyan szimuláció megalkotása, amely képes a fák növekedésének lépéseit nyomon követve, egy valósághű 3D-s modellt alkotni a fa életének különböző szakaszaiban. Ehhez szükség van egy minden fafajra alkalmazható növekedési szabályrendszer megállapítására, valamint a növekedést befolyásoló tényezők bevonására a szimulációba. A cél egy olyan pontos szimuláció megalkotása, amivel előre meghatározható, hogy egy fa megadott környezetben milyen módon viselkedhet. A program képes egyszerre több fa szimulálására is, ilyen esetben a fák hatással lehetnek egymás növekedésére (fizikai akadályként, illetve árnyékot vetnek egymásra). A végeredményként kapott modellt felhasználhatják erdészek, kertészek, park tervezők arra, hogy megállapítsák, hogy adott fafaj megfelelően illeszkedik-e az adott környezetbe, alkalmazkodik-e az éghajlati tényezőkhöz. A modellek továbbá felhasználhatók animációk és játékok készítése során is, javítva ezzel azok látványvilágát és valósághűségét. A program tervezése során az elsődleges szempontok közé tartozik az, hogy könnyen használható legyen bármilyen 3D-s modellező alkalmazás részeként.

1.3. Jelenleg létező megoldások

A fentiekben említetthez hasonló célokat fogalmaz meg a 2003-as Structural simulation of tree growth and response című kutatás [4]. Ebben a szimulációban a fa strukturális stabilitására, energia befektetés és visszanyerés arányára fektetik a fő hangsúlyt. Ennek megoldására matematikai modellt hoztak létre, ami minden egyes ág tömegét és teherbírását figyelembe veszi. Emellett számol a növekedésbe fektetett és a fotoszintézisből visszanyert energiával is. A fák növekedésük során minél nagyobb fotoszintetizáló felület megteremtésére törekszenek a saját fizikai korlátaikon belül. A szimuláció így viszonylag kevés környezeti hatás figyelembe vételével is valósághű képet ad a szimulált fa alakjáról.

A fák növekedésének szimulálásával már korábban is foglalkoztak. Catherine Jirasek és Przemyslaw Prusinkiewicz 1999-ben [16] az ágak alakjával és az elágazásokkal kapcsolatos kutatást végzett. Ezt egy évvel később Catherine Alena Jirasek folytatta és kiegészítette a kutatást egy L-Systemen alapuló grafikus alkalmazással is. Ebben a kutatásban olyan modellt alkotott, ami a környezetre érzékeny módon határozta meg a fák alakját. Az általa figyelembe vett tényezők a következők [17]:

- Az elágazások szöge
- Az elágazások elfordulása
- Gravitáció
- Heliotropizmus, és geotropizmus
- Hossz- és keresztirányú növekedés
- Ütközés

A már említett L-System (Lindenmayer systems [19]) a növények növekedésének matematikai modelljeként jött létre. A kezdeti verzió csak egy irányba növekvő egyszerű szálak modellezésére volt alkalmas, a későbbiekben továbbfejlesztett modell viszont akár magasabb szintű növények háromdimenziós szimulálására is használható. A rekurzív megvalósítás miatt tökéletesen használható önhasonló formák modellezésére. Működésének alapja, hogy adva van egy kezdeti állapot és szabályrendszer,

erre mutat példát az (1) egyenlet. Ezután az algoritmus meghatározott mélységű rekurzióig alkalmazza a szabályrendszerben meghatározott helyettesítéseket.

$$\begin{aligned} &S\{A,B\} \\ &\alpha\{A\} \rightarrow \{B,B\} \\ &\beta\{B\} \rightarrow \{A\} \end{aligned} \tag{1}$$

Majd a kapott karaktersorozat elemeihez geometriai jelentést rendelünk. Ezzel a módszerrel természetesnek ható növényi modellt kaphatunk.

Jason Weber és Joseph Penn 1995-ben a fák realisztikus megjelenítésével foglalkozott [25]. A megközelítés a fák növekedését tekintve nem szimuláció jellegű. A fák megjelenése egy egyszerű szabályrendszerrel alakítható, ahol többek között megadható a fa alakja, elágazások száma, felbontása, elágazások szöge, a törzs elágazásainak száma. A megadott adatokból készül el a gúlákból álló ág rendszer, ami viszont nem veszi figyelembe a környezet semmilyen hatását, csupán a szabályokat. Mivel a program főleg a megjelenésre koncentrál, ezért a rekurzív elágazások utolsó szintje ágak helyett leveleket generál. Véletlen faktorok bevezetésének köszönhetően egy seed érték módosításával több variáns is elkészíthető egy fa fajtából.

Azóta rengeteget fejlődtek a renderelési és szimulációs technikák, és ezzel együtt a növényzetet modellező alkalmazások sokasága is megszületett. Ezek nagyrésze viszont a legutóbb említett technikát használja a fák térhálójának előállítására. Ezek közül a legnépszerűbb a SpeedTree [22], további hasonló alkalmazások: Tree It, Forester Pro, Arboro és a Blender addon Sappling. Ezen alkalmazások használatakor a felhasználónak kell tisztában lennie a fa jellegével és környezetével, ha abba illeszkedő alakot szeretne kapni végeredményként, mivel nem szimulációként, hanem modellező eszközként viselkednek. Szükséges az ágak számának, eloszlásának, hosszának megadása, amiből egy véletlenszerű modell készül, amit a program a kiválasztott fa típus alakjára vág. Ezután a felhasználó kézzel igazíthat az ágakon, ha pontosabb elképzelése van, a véletlen seed módosításával itt is több variáció kapható egy adott fából. A profibb programok akár textúrát, leveleket és szélmozgás animációt is készítenek a fákhoz. Az ilyen módszerrel előállított fákat animációk, játékok építőelemeiként használják.

A fák növekedését szimuláló másik programcsoport az erdészeti, faipari felhasz-

nálásra készült, fahozamot számító szimuláció. Ezek az alkalmazások általában speciálisan egy-egy terület erdőinek növekedési ütemét és fahozamát számítja előre. Ezek a programok nem rendelkeznek vizuális megjelenítéssel.

1.4. Előrelépés

Az általam készített program a következő előrelépéseket mutatná az előzőekben bemutatottakhoz képest:

1. A fa növekedése a fajtájának megfelelő.
2. A fa növekedése a környezeti hatásokhoz alkalmazkodik.
3. A növekedési szimuláció eredménye egy tényleges 3D modell.
4. Több fa szimulálása egy időben, amelyek egymásra hatással vannak.
5. Könnyű bővíthetőség: egyszerűen adható hozzá bármilyen módosító hatás az ágak növekedéséhez.
6. Egyszerű interfészen keresztül elérhető C++ dll, ami bármilyen 3D-s modellező alkalmazásba beépíthető.
7. Az általános célú GPU programozás segítségével gyorsabb szimulációk.
8. Könnyen kezelhető, felhasználóbarát alkalmazás.

2. Probléma analízise

2.1. Szimuláció

A számítógépes szimulációk sok problémára jelentenek megoldást. A valóságban nem, vagy csak nehezen és drágán megvalósítható eseményekre adnak olcsóbb és egyszerűbben használható végeredményt. Kipróbálható velük olyan dolog is, ami a valóságban nem lenne lehetséges, illetve a jelen helyzetben rengeteg időt venne igénybe (például egy fa felnövéséhez akár több száz évre is szükség lehet). Tesztelhetünk veszélyes dolgokat is, amik a valóságban kipróbálva súlyos következményekkel járhatnak, ha nincsenek előre szimulálva.

A szimuláció minden esetben a probléma analízisével kezdődik. A probléma lehet esemény, folyamat vagy egy fizikai rendszer viselkedése. A következő lépés a szimuláció tárgyának lehető legpontosabb tanulmányozása, ezután a probléma részekre bontása és egy matematikai modell megalkotása a feltárt értékek alapján. Ezután következik a modell feltöltése a kutatás során feljegyzett adatokkal, végül a szimuláció futtatása és a végeredmény kiértékelése.

A szimulációk egy valószínű megoldást adnak a szimulált problémára, ami annyit jelent, hogy nem biztos, hogy teljesen az fog történni, amit a szimuláció jósolt, de minél pontosabb a modellünk, annál közelebb lesz a végeredmény a valósághoz. A szimulációk két csoportba sorolhatók. Lehetnek determinisztikusak és sztochasztikusak. Determinisztikus egy modell, ha nem tartalmaz valószínűségeket és véletlen számokat. Sztochasztikus, ha ezek előfordulnak benne. A jelen dolgozat tárgyát képző szimuláció az utóbbihoz tartozik, ezért fontos megemlíteni a pszeudo random fogalmát. A pszeudo random generáló egy számsorozatot hoz létre, ami az adott tartományban egyenletesen oszlik el. Random szám alatt a továbbiakban ennek a sorozatnak egy elemét értem [21].

Az általam megvalósított szimuláció nagyrészt determinisztikus, de a fák véletlen természetét követve néhány sztochasztikus elemet is tartalmaz. A természet szimulálására a legalkalmasabb módszer az, hogyha ugyanazt az utat követjük, amit a természet is bejár. Ez egy fa növesztésénél azt jelenti, hogy például egy öt éves fa esetén nem csak a végső állapotot határozza meg a szimuláció, hanem az oda vezető lépéseket évenként egyesével hajtja végre. Ezt a természet-motivált szimulációs

technikát sok algoritmus jó eredményekkel alkalmazza. Ilyenek például a genetikus, evolúciós modellt követő optimalizáló algoritmusok, a képalkotásra használt sugárkövető algoritmusok, vagy akár a mostanában nagy népszerűségnek örvendő neurális hálózatokat alapul vevő gépi tanulás is.

Ha pedig a természet viselkedését szeretnénk szimulálni, akkor különösen érdemes a nyomában járni, mivel az evolúciónak több ideje volt kitapasztalni, hogy mi az ami működik és mi az ami nem. Éppen ezért az általam megvalósított szimuláció évenkénti iterációkban számítja ki az ágak állását, és közben igyekszik azokat a lépéseket tenni, amit egy fa is tenne.

2.2. Fák

2.2.1. Fák a valóságban

Mivel a fák helyváltoztatásra nem képesek, ezért másképp kellett, hogy alkalmazkodjanak a környezetükhöz. Evolúciójuk során rengeteg különböző módszert fejlesztettek ki, amelyek segítik az életben maradásukat. Ha egy fára gondolunk, az első dolgok között jut eszünkbe a magassága. A fák magassága nem lenne fontos szempont, ha minden fa egyformán alacsonyra nőne, viszont ha egy mégis valamilyen mutáció miatt kiemelkedne a többi közül, akkor fölénybe kerülne és az ő leszármazottai szaporodnának el. Ez a folyamat egészen addig folytatódna, amíg nem kerül több energiába a plusz növekedés, mint amennyivel többet nyer az így szerzett fénnyel.

Ebből is látszik, hogy a fák növekedését vizsgálva a legfontosabb tényező a fény. A növekedés irányát befolyásolja a gravitáció is, de ez jóval kisebb mértékben. Akkor játszik főbb szerepet, hogyha teljes sötétségben van a fa, mert ilyenkor minél gyorsabban próbál minél magasabbra nőni a fény elérésének reményében. A növekedés irányának harmadik fontos meghatározói a környezet fizikai objektumai. Hiszen a fának szükséges ezeket érzékelni és kikerülni [7].

Az utolsó jelentős növekedési irányt befolyásoló tényező a faji tulajdonság. Ez mindent magába foglal, ami egyes fa fajok jellegzetes megjelenését adja. A fontosabb megjelenést leíró tényezőket a „2.2.3 A fák tulajdonságai” fejezet írja le. Ezek közül a dolgozat csak az ágak megjelenését közvetlenül befolyásolókkal foglalkozik, ezek a „2.2.4 Fák a szimulációban” fejezetben találhatók.

A fák növekedését egyéb környezeti tényezők is befolyásolják: víz, talajtípus, tápanyagok, hőmérséklet, szél. A fa típusától, életszakaszától és az évszaktól függ, hogy miből milyen mennyiségre van szüksége a normális növekedéséhez, szaporodásához. Ha kevesebb van valamelyikből, akkor a fa nem tud megfelelően növekedni, ha sokkal kevesebb, akkor pedig elpusztul. Ha több, akkor nagyobb mértékű a növekedés, de a túl sok víz, túl erős napsütés a növény pusztulásához is vezethet.

A fák és más növények ezen kívül képesek reagálni a környezetük hatásaira sok más módon is. Például ha egy hernyó elkezdi rágni a leveleiket, akkor különleges kémiai vegyületeket bocsátanak ki, amelyek odavonzzák a hernyó ragadozóit, emellett a fa többi levele és a szomszédos fák levelei is megérzik a vegyület jelenlétét és elkezdik termelni azt, így megelőzve a fertőzés terjedését. Bizonyos fa fajoknál előfordulnak egészen extrém esetek is, amikre a kutatók a mai napig nem szolgáltak biztos magyarázattal. Ilyen például a *Crown shyness* ami azt jelenti, hogy a felnőtt fák lombja nem ér össze, hanem mindenhol egyenletes távolságot tart egymástól. Ezzel és más hasonlóan bonyolult és különleges esetekkel viszont a szimuláció nem foglalkozik.

2.2.2. Érzékelés

A növények a fényt a hajtásrügy legvégében érzékelik. Innen kémiai vegyületeket továbbítanak visszafelé a száron, ahol serkentik a sejtek szaporodását és hosszúkás alakú elnyúlt sejteket képeznek, ez okozza az ágak fény felé hajlását. Ezt a viselkedést heliotropizmusnak hívják.

Az előző években fejlődött ágrészek már nem hajlanak tovább, mivel az ott található sejtek már megszilárdultak, befásodtak. A gravitációt itt is érzékelik, a fák igyekszenek a gravitációtól elfelé növekedni, mivel a fény többnyire abból az irányból jön, ezt a hatást geotropizmusnak nevezik és sokkal gyengébben hat az ágakra, mint a heliotropizmus. A geotropizmus jelentős szerepet játszik a gyökerek növekedési irányának meghatározásában.

A fák rengeteg különféle dolgot érzékelnek ezeken felül a környezetükből, például képesek néhány kémiai vegyületet felismerni és reagálni azokra, érzékelik ha valami hozzájuk ér (ezt bizonyítja az is, hogy ha egy ágat minden nap megfogunk, az kevésbé fog fejlődni; ennek is az az oka, hogy a fa nem költ felesleges energiát egy ág

növesztésére, ami nagy valószínűséggel letörhet). Ezekkel viszont a szimuláció nem foglalkozik mivel nagyon minimális látható végeredménye lenne, viszont rendkívüli módon lelassítaná a számítást.

2.2.3. A fák tulajdonságai

Ez a fejezet a fák megjelenését befolyásoló tulajdonságokat írja le. A dendrológia (a növény rendszertannak a fás növényekkel foglalkozó ága) szerint a következőket szükséges tudni egy fáról:

- Alaki tulajdonságok: termet, törzs, kéreg, rügy, vessző hajtás, levél, virág, virágzat, termés, mag, fa szöveti jellemzők.
- Elterjedés: vertikális megjelenés (regionalitás), előfordulás.
- Ökológiai igények: fény, hő, víz, talaj kémhatás, nitrogén, só.

2.2.4. Fák a szimulációban

Ezek közül csak a szimuláció számára fontosakkal foglalkoztam, viszont ki is egészítettem egy pár tulajdonsággal, amiket az erdészet nem tart fontosnak, viszont a számítások elvégzéséhez elengedhetetlen. Alaki tulajdonságok: termet, elágazás típusa. Ökológiai igények: fény, víz és a talaj összes tulajdonsága összefoglalva, talajminőség néven.

A fák növekedését a szimulációban egy szabályrendszer határozza meg, amit a program egy XML fájlból olvas be. A szabályrendszerben a fa élete 5 különböző szakaszra van bontva: magonc, csemete, suháng, kifejlett fa, magtermő fa (seedling, plant, sapling, grown, old) egy-egy szakasz egy-egy állapotot reprezentál. Egy szakasznak van neve, ami csak az azonosításban segít hibakereséskor, illetve egy kor értéke. Ez a kor érték adja meg, hogy a fa hány éves koráig követi az adott állapot növekedési szabályait. Az állapotokon belül három szint lett megkülönböztetve, ahol az egyes szintek a fa ágainak szintjét jelölik. A 0. szint a törzs, az első szint a törzsről közvetlenül leágazó ágak, a 2. szint az első szint ágairól leágazó ágakat jelöli. A további szintek mind a második szint szabályait követik.

Ezek a szintek a következő értékeket határozzák meg:

- Elágazás típus, ez 4 különböző érték lehet: örvös, ellentétes, felváltott, spirális (whorled, opposite, alternate, spiral).
- Elágazás elfordulása: két ág egymáshoz képesti elfordulási szögét adja meg.
- Elágazási szög: a szülőágtól számított dőlésszöget adja meg.
- Elágazás darabszáma (min/max): egy évben egy ágról létrejövő elágazások száma, tól-ig értékként megadva.
- Hosszanti növekedés: ideális tápanyag és vízfelvétel esetén vett hosszanti növekedés mértéke.
- Szélességi növekedés: ideális tápanyag és vízfelvétel esetén vett keresztirányú növekedés mértéke.

Az állapotok ezenkívül tartalmazzák a fa ideális növekedéséhez szükséges mennyiségeket is.

- ideális víz mennyiség, a fa növekedéséhez szükséges víz mennyisége `[mm]`

2.2.5. Az ágak szimulálásának lépései

Egy iteráció során az ágak egyesével növekednek a fa mélységi bejárása során. A növekedés a következő lépésekből áll:

1. Az ág helyének meghatározása a szülő ágon (csak az új ágaknál).
2. A fa stílusjegyeinek megfelelő, alap növekedési irány kiszámítása.
3. Az előzőekben számított irány enyhe módosítása a legerősebb látható fényforrás irányába.
4. Az ág adott évi hosszirányú növekedésének kiszámítása.
5. Ütközés vizsgálat a környező objektumokkal, és az irány módosítása az esetleges ütközésnek megfelelően.
6. A vezérlő (kontroll) pontok alapján egy természetes ív számítása.
7. Leszármazott ágak növesztésének rekurzív hívása.
8. Új leszármazott ágak létrehozása.

2.3. Környezet

A szimuláció során kétféle környezetet különböztetek meg: a szimulációs teret és a környezeti hatásokat. A szimulációs térhez tartoznak a fényforrások, a többi fa, illetve egyéb árnyékot adó és fizikailag blokkoló objektum. A környezeti hatások a talaj minősége, a csapadék mennyiség és változásai.

2.3.1. Szimulációs tér

A szimulációs teret a szimuláció a futtató környezetből kapja, a fényforrások listája pozícióval és energiával együtt. Minden térben található objektumot háromszögekre bontva kap, mivel a háromszögek könnyebben kezelhetőek, mint a teljes objektumok, és a fa szempontjából nem számít, hogy minek a részét képezi az adott háromszög. A szimuláció ezeket az adatokat és a szabályrendszert felhasználva határozza meg a fa növekedési irányát.

2.3.2. Környezeti hatások

A környezeti hatások külön konfigurációs fájlból töltődnek be a szimuláció kezdetekor. Ez tartalmazza a csapadék mennyiséget éves, azon belül pedig havi bontásban. Ezzel szimulálható egy aszályosabb év, esetleg mesterséges locsolás is. A talaj jellemzése már jóval komplexebb kérdés, mivel sok minden van rá hatással. Számít a víztartása, ásványianyag tartalma (ami változhat is az évek során, ahogy a fa ásványi anyagokat vesz fel, viszont lombhullással vissza is adja egy részét, vagy épp többet is). Ezt a komplexitást bővítési lehetőségnek szeretném megtartani, a talajt csak a minőségével jellemzem 0-10 között, és ez a szimuláció során nem változik. A környezeti hatások listáját a jövőben tervezem bővíteni tengerszint feletti magassággal, és uralkodó széliránnyal is. Viszont ezek pontos hatása és annak mértéke még rengeteg utánaolvasást igényel.

2.4. Sugárkövetés

A sugárkövetés egy képalkotási technika, ami a fény útjának visszakövetésével és a különböző felületekkel való interakciójának szimulálásával határozza meg a pixelek színét. Jelen helyzetben azonban az ágat érő legerősebb fény irányának megha-

tározására lesz alkalmazva. Így az alap sugárkövetés sok tényezőben módosul és egyszerűsödik is.

2.4.1. Monte Carlo módszer

A Monte Carlo módszer magába foglal egy nagy csoport algoritmust, amelyek mind ismételt véletlenszerű mintavételezésen alapulnak. A véletlenszerű mintavételezés megoldja azokat a problémákat, amik előkerülhetnek egy determinisztikus megoldás alkalmazásakor. A sugárkövetés is pont egy ilyen probléma, mivel, ha nem véletlenszerűen, hanem egy rács mentén futtatnánk a sugárkövetést, akkor nagyon pici eséllyel találná el a fényforrást, főleg ha az pontszerűen van modellezve. A sugarak véletlenszerű visszaverődése megfelelően szimulálja a valóságban mikroszkopikus egyenetlenségekkel rendelkező felületeken szóródó fényt is [2][18].

Az általam módosított sugárkövetés nem képet ad, hanem a tér egy pontjában a legerősebb fény érkezési irányát mutatja meg. Először is megvizsgálom, hogy az adott pontból látszik e fényforrás. Ha látszik, akkor nincs árnyékban, tehát vége is az algoritmusnak, nincs szükség további műveletekre. Ha nem látszik, akkor a pontból indul el a sugárkövetés teljes gömbfelületet vizsgálva Monte Carlo módszerrel, de itt is minden egyes új találati pontból megvizsgálom, hogy a fényforrás látható vagy sem. Ha látható, akkor visszatér az algoritmus az értékkel, és az első sugár iránya lesz a keresett irány. Ha nem, indul az újabb sugárkövetés. Ez az algoritmus alapesetben rekurzív, viszont a GPU kódban nem optimális a rekurzió, illetve a rekurzió okozta plusz hívások lassítják is a programot, ezért egy iteratív megoldásra lesz szükség.

2.5. Adatszerkezet

Mint az a sugárkövetésnél is látható, a természet szimulálására a legalkalmasabb módszer, hogyha ugyanazt az utat követjük, mint a természet. Éppen ezért a fa eltárolására egy fa adatszerkezet a legalkalmasabb. A törzs a gyökérelem és minden elemnek lehet N darab gyerekeleme. Az N értéke áganként változó, és a futás során változhat, ha egy új ág nő, vagy épp elszárad egy régi.

3. Eszközök kiválasztása

3.1. Programnyelv

A programozási nyelv kiválasztásakor több szempont is döntő szerepet játszott. Mint a szimulációk általában, ez a program is nagy számítási igénnyel rendelkezik. Ezt alátámasztják korábbi sugárkövetéssel kapcsolatos projektjeim is. Emiatt fontos olyan programozási nyelv kiválasztása, ami a lehető legjobb teljesítmény elérésére ad lehetőséget. Ezen felül fontos az általános célú GPU programozás támogatása is.

3.1.1. Python

A Python egy magas szintű programozási nyelv, ami a programozás sebességére helyezi a hangsúlyt a futás sebességével szemben. Emiatt gyakran használják prototípus készítésre, és ha már végleges az algoritmus, akkor átírják egy gyorsabb futást lehetővé tevő nyelvre. Rengeteg harmadik féltől származó támogató könyvtára van, és támogatja az általános célú GPU programozást is.

3.1.2. C#

Magas szintű programozási nyelv, a .NET keretrendszerrel kiegészítve rengeteg hasznos funkciót foglal magába. Az általános célú GPU programozás csak harmadik fél által írt könyvtárakon keresztül érhető el. A fejlesztés megkezdésekor ezek a könyvtárak több éve nem voltak frissítve. Azóta megjelent egy Hybridizer nevű fordító, ami lehetőséget ad több szálas processzorokra vagy GPU-kra optimalizált bináris vagy forrás kód generálására [15].

3.1.3. C++

A C++ egy általános célú objektumorientált programozási nyelv. Magas szintű funkciói mellett lehetőséget ad alacsony szintű memóriakezelésre is. A tervezési szempontjai között elsődleges a gyorsaság, a hatékonyság és a rugalmasság. Támogatja az általános célú GPU programozást is [23].

A fent felsorolt nyelvek csak egy kis részét fedik le a létező rengeteg programozási nyelvnek, mivel igyekeztem olyanokat választani, amiknek relevanciája van a probléma terében. Közülük a C++ bizonyult a legalkalmasabbnak a feladat elvégzésére

a gyorsasága és a közvetlen GPU támogatása miatt.

3.2. Fájl leíró nyelv

A szabályrendszerek, és a környezetek leírására szükségtelen egy teljes relációs adatbázis használata, mivel kevés adatot kell tárolni, a módosítások száma pedig nem jelentős. Jobb megoldás valamilyen jelölő nyelv használata az alábbiak közül.

- XML: Extensible Markup Language, egy általános célú hierarchikus adatleíró formátum. Szigorú előírásoknak kell megfelelnie, emiatt könnyen feldolgozható programok segítségével. Emellett ember által is könnyen értelmezhető szintaktikával rendelkezik.
- JSON: JavaScript Object Notation, egy kisméretű, szöveges adatleíró nyelv. Ember által olvasható formátumban határozza meg az adatszerkezeteket és értékeket. Általában szerver és kliens közötti adatátvitelre használják.
- YAML: Ember által olvasható szerializáló nyelv, általában konfigurációs fájlnak használják.

A fentiek közül az XML-re esett a választás a következő miatt: ember által a legkönnyebben olvasható, a kötelező nyitó és záró tag elnevezéseknek köszönhetően. Emellett támogatja a az automatikus validálást is.

3.2.1. XML kezelő könyvtár

Mivel a szabályrendszereket XML fájlokban tárolom, ezért szükség van egy egyszerű XML kezelő könyvtárra.

Népszerű XML kezelő C++ könyvtárak a LibXML2, Xeres, RapidXML és a TinyXML. Korábbi projektjeimnél a TinyXML2-re került a választás kis mérete és egyszerű használata miatt, ezért itt is ez került a projekt függőségei közé.

3.3. Általános célú GPU programozás

A videokártya általában csak a számítógépes grafikák megjelenítéséhez szükséges számításokat végzi, ezek a műveletek nagyon sokszálas párhuzamosítást igényelnek.

Ezt a tulajdonságot használják ki a videokártyát általános célokra használó alkalmazások. Ezek esetében nagyon sok, viszonylag egyszerű műveletet kell elvégezni minél rövidebb idő alatt. Ha ezek a műveletek egymástól függetlenek, akkor a probléma jól párhuzamosítható. A szimulációkban általában, és a jelen szakdolgozatban is előfordulnak ilyen számítások, így feltétlenül szükséges a technológia alkalmazása.

Jelenleg két fejlesztői platform áll rendelkezésre az általános célú GPU programozásra. Az egyik az Nvidia által fejlesztett CUDA fejlesztői csomag, ami C és C++ alkalmazások gyorsítására használható. Elérhető Windowsra, Linuxra és Macre is, viszont csak az Nvidia videokártyák támogatják (bár itt akár több videokártyát is kezelhetünk egy időben). Viszonylag széles eszköztára van mind a fejlesztés, mind a hibakezelés szempontjából. Több, a gyártó által támogatott könyvtár is létezik, amik megkönnyítik a használatot. Emellett részletes és könnyen követhető dokumentáció is tartozik hozzá.

A másik lehetőség az OpenCL használata, amely platformfüggetlen lehetőséget ad a grafikus kártyák (és általában a sokprocesszoros adatpárhuzamos rendszerek) kezelésére. Ennek használata jóval nehézkesebb, a hozzá adott fejlesztői eszközök is kiforratlanabbak.

Bár a platformfüggőség kizáró ok lehet ipari környezetben, egy kutatási projekt esetén ez nem feltétlenül okoz gondot. Jelen esetben is így van, ennek megfelelően az egyszerűbben használható, jobban támogatott CUDA környezetet választottam.

3.4. Build rendszer

A build rendszerek lehetővé teszik a szoftverek fordításának, tesztelésének és csomagolásának automatizálását. Ezeknél van egy számomra hasznosabb funkciójuk is, képesek a projektek függőségeinek megkeresésére, letöltésére és beállítására, ez alapján pedig különböző fejlesztő környezetek projektfájljainak generálására. Ennél a projektnél is különböző eszközökön folyt a fejlesztés, amiken mind más elérési utakon találhatók a függőségek, és a fejlesztő környezetek is különbözőek. Emiatt előnyös az Out-of-source build rutin használata, ami azt jelenti, hogy a forrásfájlok és a bináris fájlok egymástól elválasztva, külön mappában találhatók. Ilyen esetben a verziókövetésben is csak a forrásfájlok és a build rendszer fájljai szerepelnek, ezzel csökkentve a repository méretét. Ennek a gyakorlatnak az egyetlen hátrá-

nya az, hogy a fejlesztőkörnyezetből nem adható hozzá új forrásfájl, mivel az IDE nem a megfelelő mappába teszi, ezért minden új forrásfájl hozzáadás után újra kell generálni a projektfájlokat.

Több szoftver is rendelkezésre áll erre a feladatra, többek között a Ninja, Meson, Sharpmake, Maven, Ant, Gradle, CMake. Ezek közül a legkiemelkedőbb és C++ környezetben a legtöbbet használt a CMake. Ez egy nyílt forráskódú platformfüggetlen eszköz, ami szoftver projektek buildelésére, tesztelésére és csomagolására szolgál. CMakeLists fájlokban határozható meg a program viselkedése, ezek segítségével felépíthetünk bonyolult, több projektet magukba foglaló forrásfákat, beállíthatjuk ezek függőségeit. Támogat több fejlesztő környezetet is, ezek között a Visual Studiot, aminek a 2017-es verziójában már integrálva is megtalálható.

Mivel a fentiek közül ez az egyetlen, ami támogatja a CUDA nyelvet is, így a választásom értelemszerűen erre esett.

3.5. Verziókövetés

A verziókövetés a szoftver rendszerek készítésénél fontos fejlesztést segítő eszköz. A fejlesztés során lehetőséget ad a változások időbeli követésére, valamint jegyzi, hogy az egyes változásokat ki hajtotta végre. Emellett könnyen kezelhető módon létezhet több verziója is a fejlesztett projektnek. Több fejlesztő esetén szinkronizálja az emberek munkáját. Valamint hasznos, hogy a forráskód nem csak a fejlesztők gépén tárolódik, hanem egy központi szerveren is, így egyrészt biztonságosabb a több másolat miatt, másrészt bárhonnan elérhető. A leggyakrabban használt alternatívák az alábbiak:

- Subversion: a CollabNet által fejlesztett verziókövető rendszer az Apache Software Foundation projekt része. Egyszerűen megtanulható és kezelhető. Hátránya, hogy a helyi repository nem teljes értékű, így folyamatos kapcsolat szükséges a szerverrel (például a commitoláshoz).

- Git: nyílt forráskódú verziókövető rendszer, eredetileg a Linux kernel fejlesztéséhez készült, napjainkban a legelterjedtebb. A megtanulása kicsit több időt igényel, mint az SVN, de az alapvető műveletek itt is könnyen megérthetők. A helyi repository teljes értékű, történettel, és minden fájllal. Lehetséges bár-

milyen művelet végrehajtása offline állapotban is, majd a változások későbbi pusholása a szerverre. Egy hátránya, hogy a teljes történet jelenléte miatt a helyi másolat nagy méretet érhet el.

- Team Foundation Version Control: a Microsoft által fejlesztett verziókövető rendszer. Az SVN-hez hasonlóan központi szerveren tárolja a kódot, innen a fejlesztők lefoglalják a számukra szükséges fájlokat, amikről egy pillanatnyi másolatot kapnak. Ez ismételten felveti az SVN-nél már említett problémát.

A nagy népszerűségének köszönhetően, és annak, hogy gyakran végzek fejlesztést buszon és más helyeken, ahol nincs internet hozzáférés, a Gitre esett a választásom.

3.5.1. Verziókövető szerver

A verziókövetéshez szükséges egy szerver, ami a repositoryt tárolja, enélkül a legtöbb előnyt elveszítjük. A következőkben a legnépszerűbb gitet támogató verziókövető szerverek előnyeit és hátrányait ismertetem.

- GitHub: A githubon diákként lehetőség van ingyenes privát repository létrehozására. Nincs se fő, se maximum repository méret limitáció. Viszont a maximum fájlméret 80Mb.
- Bibucket: A bitbucketen lehetőség van privát repository létrehozására. Az ingyenes verzióban korlátozva van a maximum résztvevők száma a projektben, és a maximum projekt méret is 2Gb. Egy fájlra vonatkozó megszorítás nincs.
- Azure DevOps: Maximum 5 fős csapatok használhatják az ingyenes verziót, nincs megszorítás se a fájl, se a teljes repository méretére.

A fent említett verziókövetést támogató oldalak mindegyike megfelel a projekt tárolására, mivel egyedül végzem a fejlesztést, és a projekt csak forráskódot tartalmaz, ami inkább több kis méretű fájlt jelent, amiknek az összmérete is kicsi. Mivel a projektjeim nagy része már ott található, és azzal dolgoztam a legtöbbet, ezért a GitHubot választottam

3.5.2. Verziókövetési modell

A verziókövetésnél a lehető legegyszerűbb modellt követtem. Mivel egyedül végzem a fejlesztést, és általában a programnak csak egy részén dolgozom, ezért nem szükséges az egyes projektek számára külön ág létrehozása. A fejlesztés folyamata végig a fő (master) ágon folyik. Néhány jelentős módosítást kivéve (amiknek véglegességében nem voltam biztos), amelyek külön ágra kerültek, és csak a módosítások végrehajtása és ellenőrzése után lettek visszaolvasztva a fő ágba. Korábbi projektjeimnél használtam bonyolultabb modelleket, de ezek az egyszemélyes fejlesztésnél több nehézséget okoznak, mint amennyi hasznuk van.

3.6. Keretprogram

A fejlesztési célok között szerepelt, hogy az általam készített rendszer rugalmasan felhasználható legyen különféle környezetekben. Ennek megfelelően fejlesztésem eredménye egy osztálykönyvtár, ami csatolható más projektekhez. Egy másik felhasználási mód, ha az általam készített könyvtárat pluginként működtetjük egy már létező (tipikusan 3D modellező) rendszer részeként.

A keretprogram az a harmadik fél által gyártott program, aminek keretein belül az általam írt dll fut. Ez az alkalmazás a szimuláció indításáért és a végeredmény megjelenítéséért felelős. Az az előnye annak, hogy egy már létező modellező programba épülő modulként lesz használható a szimulációm, hogy így már van egy 3D-s megjelenítő környezet, kezelőfelülettel együtt. A szimuláció bemeneteként használható a már létező modellezett környezet, így az ahhoz illeszkedő fát generál. A végeredmény ezután renderelhető képpé, vagy elmenthető az általános 3D-s fájlformátumokba, hogy más programokban is használható legyen. A szimuláció végeredménye pedig akár utólag módosítható is.

Az alábbi modellező eszközök közül kellett választanom:

- Blender 3D: a Blender3D egy ingyenes nyílt forráskódú 3D modellező program rendkívül széles eszközpalettával. A programot ugyan 1996 óta fejlesztik, de csak az utóbbi években vált egyre népszerűbbé, mivel a korábbi verziói még meglehetősen instabilak voltak. A jelenlegi legfrissebb verzió száma 2.79. 2018 augusztusában várható a 2.80-as megjelenés, ami jelentős módosításokat hoz

majd mind a kezelőfelület, mind a modellezési technikák tekintetében. A fő funkciói a modellezés, animáció, szimuláció, renderelés. Emellett lehetőséget ad mozgáskövetésre, videó szerkesztésre, vagy akár játékmotorként is használható. Éppen emiatt a széles eszközkészlet miatt megfelelő választás, hogy kezelő és megjelenítő felületet adjon a szimulációhoz. A programhoz lehetőség van Python nyelven scriptet és add-ont írni.

- Autodesk 3ds Max: a 3ds Max szintén egy rendkívül széles eszközpalettával rendelkező program, ami képes szinte mindenre, amire a Blender3D. Ami funkció hiányzik, az pedig megtalálható a szintén az Autodesk által fejlesztett Maya nevű alkalmazásban. A két szoftver nagyon hasonló, csak a felhasználási céljukban különböznek egymástól: a Maya inkább animációk karaktereinek készítésére ajánlott, míg a 3ds Max modellezésre, látványtervezésre. Az én céljaimnak ez utóbbi inkább megfelelő, ezért a továbbiakban a Mayáról nem esik szó. A 3ds Max hoz C++ és C# nyelven is készíthető plugin, ami könnyebbé tenné a fejlesztést, mivel kihagyható lenne a Python.

A választásom mégis a Blender3D-re esett. A programok mérete ugyanis rendkívül különböző, a Blender3D hordozható verziója 280MB míg a 3ds csak telepítve érhető el és 6Gb a tárhely követelménye. További érv a Blender mellett az, hogy nyílt forráskódú, így a forráskódból könnyen lehet példákat találni. Az utolsó érv pedig az, hogy 6 éve használom a Blendert modellezésre, ezalatt néhány scriptet is írtam már hozzá, így a program kezelését (ami nem kis feladat egy ilyen bonyolultságú eszköznél) már nem kell elsajátítanom.

3.6.1. Blender Python

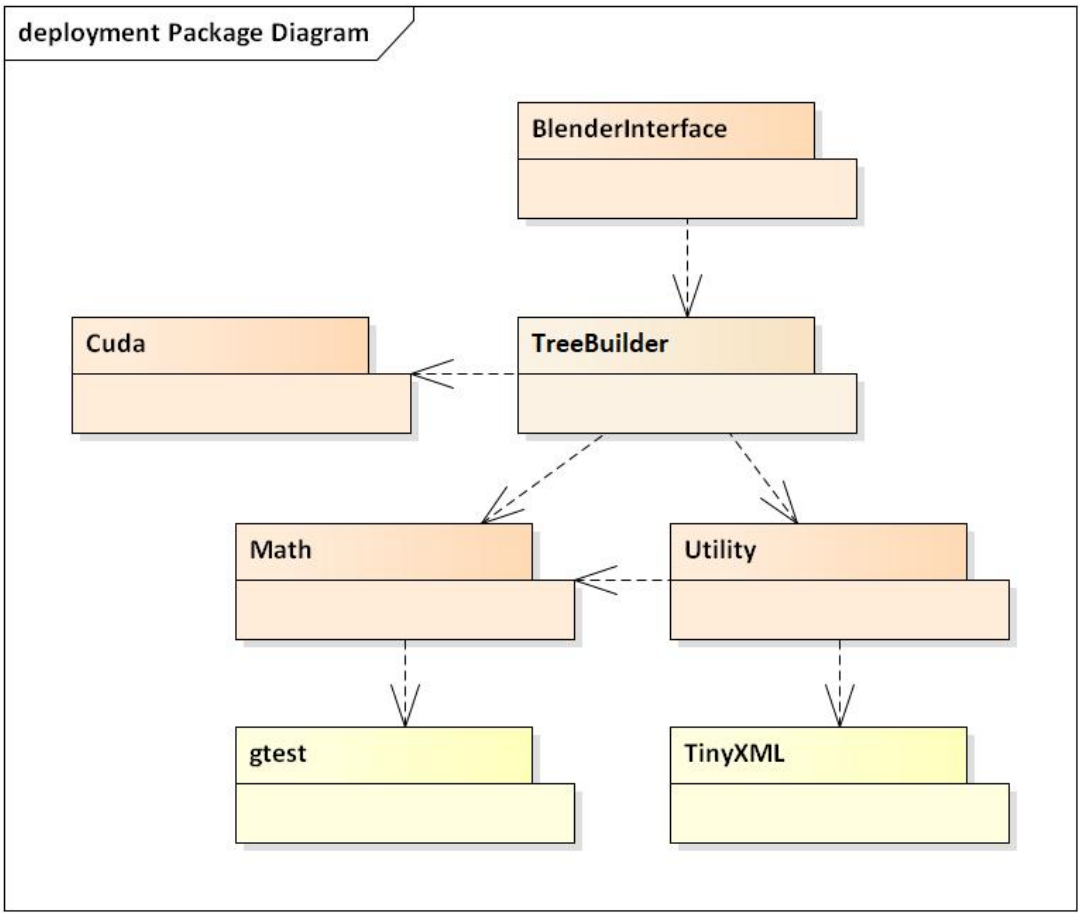
A Python egy objektumorientált script nyelv, ami tartalmaz modulokat, hibakezelést és magas szintű dinamikus adattípusokat, illetve osztályokat. Éppen ezért ad jó alapot a blender script és addon rendszerének, kiegészítve Blender specifikus osztályokkal.

A script segítségével lekérdezhető a scene minden eleme, létrehozható, módosítható, tehát minden opció elérhető, ami a kezelőfelületen vagy gyorsgombokkal alkalmazható lenne. Ezen túl hívható külső könyvtár is, ezt a lehetőséget haszná-

lom arra, hogy összekössem a Blendert a bpy-n (Blender Python) keresztül a C++ kóddal ami módot ad a CUDA használatára.

4. Tervezés

4.1. Csomag diagram



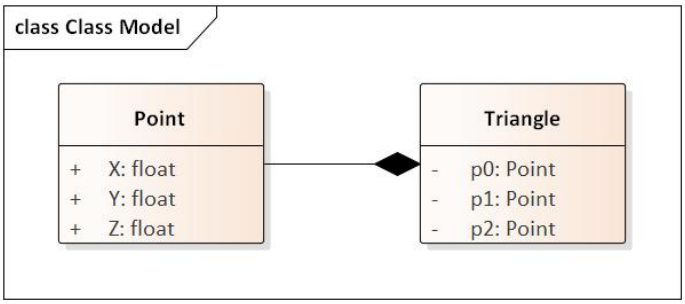
4.1. ábra. Csomag diagram.

A 4.1. ábra mutatja a rendszer alapvető felépítését. A program fő, összefoglaló részét a Tree Builder komponens képzi, ez felelős a fa növekedésének szimulálásáért. Ezt segíti a Math komponens, ami a matematikai műveleteket megvalósító osztályokat tartalmazza, valamint a Utility csomag, ami az I/O műveletekért felelős, az XML feldolgozástól a mappakezelésen át a naplózásig. A CUDA komponens a számításigényes műveletek felgyorsításáért felel. A BlenderInterface a pythonnal való kommunikációban segít, ezzel interfészt képezve a Blender irányába.

A gtest egy google által fejlesztett C++ teszt könyvtár [12].

A TinyXML egy kisméretű, könnyen használható XML kezelő könyvtár [24].

4.2. Math



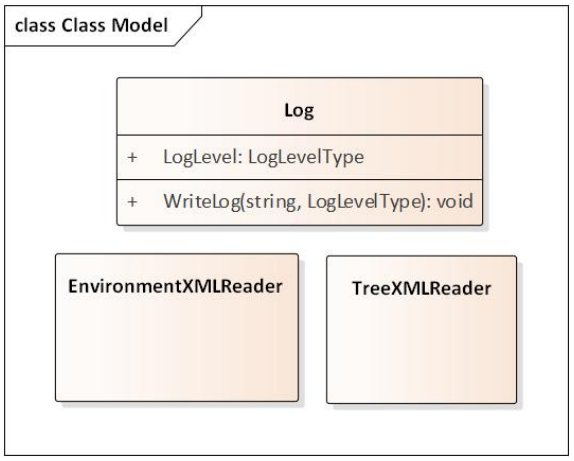
4.2. ábra. Geometriai műveleteket megvalósító osztályok.

A Math csomag (4.2. ábra) tartalmazza a programhoz szükséges geometriai műveletek megvalósításait.

A `Point` osztály egy 3 dimenziós térben értelmezett `float` pontosságú vektor. Ez a vektor pontként és vektorként is értelmezhető a problémától függően. A három komponens `x, y, z` néven vagy `Points[0-2]` módon érhető el. Az osztály a vektorokon végezhető alapvető műveleteket tartalmazza.

A `Triangle` osztály a pont osztály három példányát használva valósít meg egy háromszöget, melynek csúcsai `p0, p1, p2` néven érhetők el. Emellett a háromszögnek van egy `normal` mezője is, ami a háromszög által meghatározott sík normálvektora.

4.3. Utility



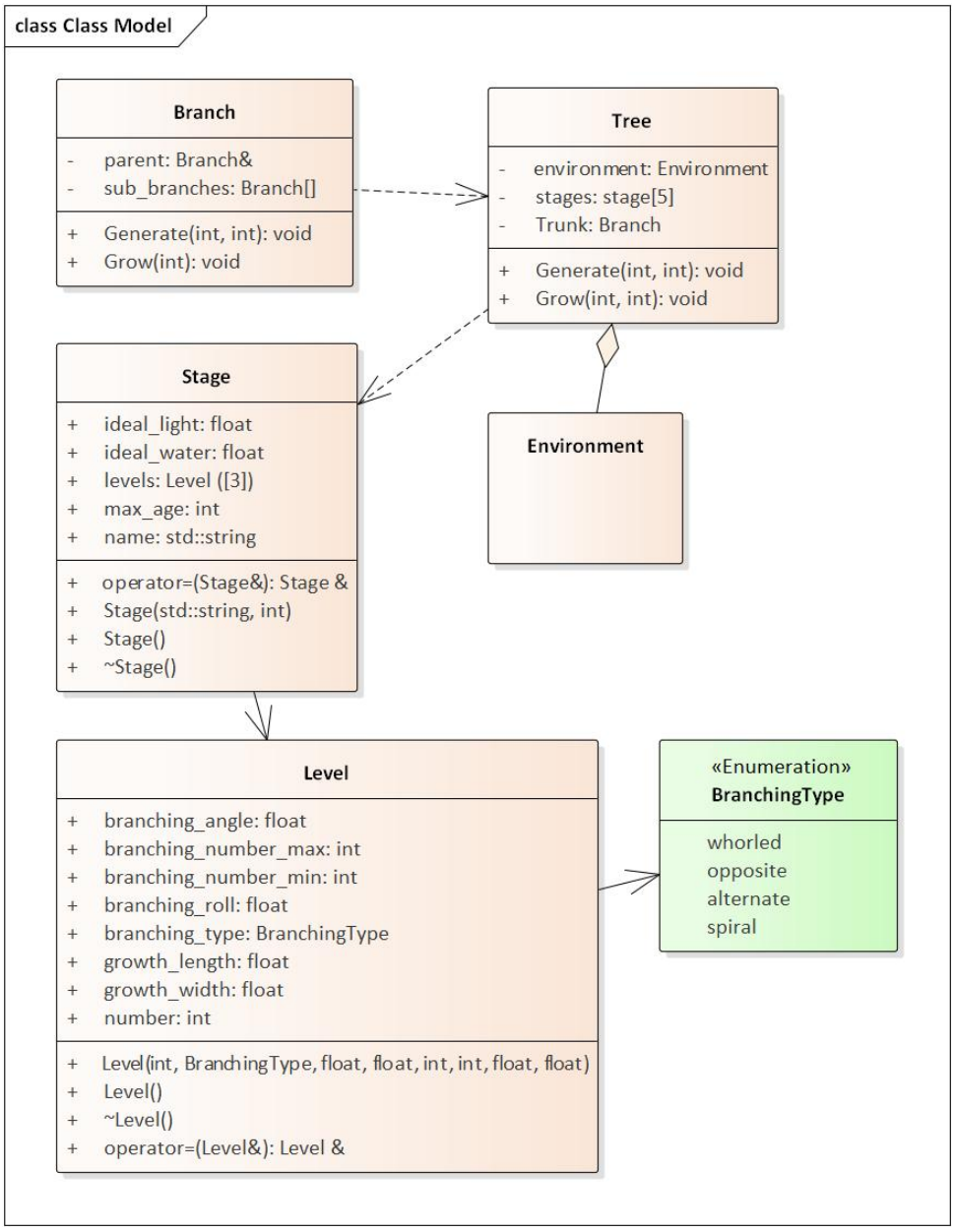
4.3. ábra. Kisegítő osztályok.

Az Utility csomag (4.3. ábra) a program I/O részeinek kezeléséért és kisebb kisegítő osztályokért felelős. A legtöbbet használt osztály a Log osztály, ami többszintű naplózást tesz lehetővé (`Exception, Trace, Debug, Message, Warning, Cuda, Error`). A különböző log szintek más színnel jelennek meg, és beállítható a minimum szint is, amit szeretnénk kiíratni.

A Random osztály egyszerűbb elérést biztosít a `std` által adott `mt19937` osztályhoz.

A Tree és Environment XMLReader osztályok felelnek az XML fájlban megadott szabályrendszerek feldolgozásáért.

4.4. Tree Builder



4.4. ábra. A Tree Builder tervezett osztályai.

A 4.4. ábrán a fa ábrázolásáért és növekedéséért felelős osztályok láthatók. Ennek a csomagnak volt egy előző verziója, ami a `Stage, Level` osztályok helyett egy *life stage enum, tree type* és egy *branchin rule* osztályt tartalmazott, viszont ez nem adott elég szabadságot, és nehezen kezelhető volt a különálló elágazás osztály és enum miatt.

A fa egy tényleges fa adatszerkezetként épül fel, aminek egy nodeját a `Branch` osztály egy eleme jelöli. Innen egy tömb mutat a gyerek elemekre, és egy referen-

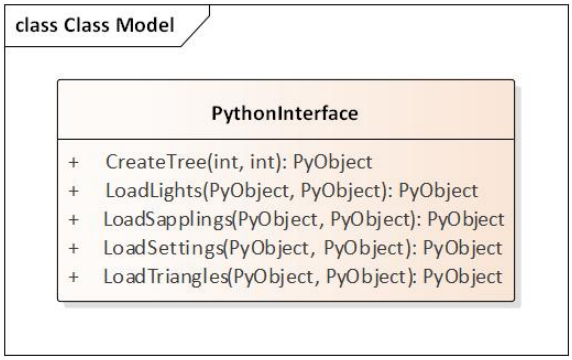
cia a szülőre. A `Stage, Level` osztályok a „2.2.4 Fák a szimulációban” fejezetben részletezett struktúrát tárolják. A generate metódussal legenerálható a fa adatstruktúra, a grow metódus hajtja végre a tényleges szimulációt. Ez a megközelítés a későbbiekben dinamikus adatszerkezetekre módosult (5.3 Adatszerkezetek).

4.5. CUDA

A CUDA projekt a futtatások alapján leglassabbnak ítélt metódus általános célú GPU programozással gyorsított verzióját tartalmazza, illetve a metódushoz szükséges egyéb metódusok CUDA nyelvű implementációját. Ilyen a fény keresésére szolgáló `Trace` metódus, illetve az általa használt metódusok, mint a vektor műveletek, illetve a háromszög metszés metódusa.

A projekt metódusai az általam készített Pont osztály helyett a CUDA-ba épített float3 struktúrát használják. A háromszögek tárolására nincs beépített adatstruktúra, erre a `cu_triangle` szolgál.

4.6. Blender Interface



4.5. ábra. Csomagoló osztály a TreeBuilder fő metódusaihoz.

A Blender Interfész (4.5. ábra) csomag nem tartalmaz logikát, csupán elfedi a Tree Builder csomag fő metódusait egy pythonból hívható változattal. Ezek a metódusok amellett, hogy elfedik a metódusokat, adatkonverziót is végeznek a C++ és a Pythonos típusok között. Az adatok mindkét irányú átadásakor szigorú szabályoknak kell pontosan megfelelniük, különben hibát okoznak a feldolgozáskor. Az átadott adatszerkezetek pontosabb leírására a „Python - C++ kommunikáció 5.1” fejezetben kerül sor. A Load előtagú metódusok a Python→C++ irányú adatáramlást

jelölik, a többi metódusnál az adatátvitel iránya fordított.

Ez a csomag tartalmazza a program buildelésében részt nem vevő python scriptet is, amit a blender addonként felismerve használ. Ez dolgozza fel a blenderben található szín elemeit, és küldi őket tovább a szimulációt végző dll-nek.

A script továbbá képes a kapott ponthalmaz alapján kirajzoltatni a fa ágait is. Egy egyszerű néhány gombos felhasználói felülettel is bővíti a blender kezelőfelületét, amiről a szimuláció beállításai érhetők el.

5. Fejlesztés

A fejlesztés során a következő lépéseket hajtottam végre. Először, a Blenderhez szükséges Python nyelvű addon és a C++ kommunikációját oldottam meg. Ezután a program egyéb fájlfeldolgozási műveleteit, amelyek a fa és a környezet szabályainak beolvasását takarják. Ezzel párhuzamosan fejlesztettem a fa és a környezet tárolásához szükséges adatszerkezeteket is. Ezt követően a fa alap megjelenését megvalósító algoritmus, majd a környezeti hatások módosításai következtek. Az összes folyamatot folyamatos javítás és refaktorálás jellemezte. Az utolsó lépés pedig a program gyorsítása volt, általános célú GPU programozással.

5.1. Python - C++ kommunikáció

A fejlesztés első kihívása a több különböző technológia együttműködésének megvalósítása volt. Mivel egyszerűen használható alkalmazást szerettem volna létrehozni, ezért az addon készítést választottam a blender forráskódjába írás, és saját blender verzió fordítása helyett. Ilyen módon egy mappa bemásolásával és az addon hozzáadásával már használható is a kutatás eredménye.

Viszont a blender Python nyelvű scripteket vár addonként, de én az újrahasznosíthatóság és sebesség szükségessége miatt, valamint az egyszerű CUDA kompatibilitás elérése érdekében C++ nyelvű könyvtárat szerettem volna. Erre a problémára adott megoldást a Python API, ami lehetőséget teremt olyan C++ nyelvű könyvtár írására, ami Python kódból használható. Az egyetlen hátrányt a típusok közötti nehézkes átalakítás jelenti, de mivel csak a szimuláció elején és végén történik a két alkalmazás között adatcsere, ezért ez a nehézség elfogadható.

A szimuláció megkezdése előtt az addon átadja a szükséges környezeti elemeket. Ezek a környezeti elemek a fényforrások, a fák helyei és a szín objektumai. Mindegyik elemhez tartozik egy metódus, ami az adatok átalakítását és feldolgozását végzi. Emellett átadódnak a szimuláció alapvető beállításai is. Ezek az alapvető beállítások a szimulálni kívánt fa indexe (-1 átadása esetén az összes fa), a szimulálni kívánt évek száma, és a szimuláció felbontásának mélysége. Ami annyit jelent, hogy hányadik alágig jöjjenek létre leszármazott ágak.

A szimuláció elindításához két metódus hívása szükséges, először az `InitTree`,

ami a fák adatainak betöltését végzi, illetve a környezet és fa párosítását. Ezután a `GenerateTree` metódussal elindíthatjuk a szimulációt a beállításokban megadott évre. Ez a metódus többször is hívható, ebben az esetben a szimuláció folytatódik onnan, ahol abbamaradt az előző hívás végén. Végül a `GetTree` metódus hívásával megkapjuk a fák ágainak adatait. Az adatok egy Python listában kerülnek átadásra, ami a szimulációban szereplő összes fát tartalmazza.

5.2. Fájl és mappa műveletek

A szimulációhoz szükséges növekedési szabályokat és környezeti adatokat két XML fájl tartalmazza. Ezek az XML-ek a dll melletti trees és environments mappákban találhatók. Amennyiben több is található a fájlokból, abban az esetben kiválasztható, hogy melyik töltődjön be a szimuláció futása során. Ennek a működésnek a megvalósításához először is szükség van a fájlok megkeresésére, erre a `Directory` osztály `LoadTypes` metódusa szolgál. Az első verzióban két különböző metódus létezett a fák és a környezet keresésére, de mivel a kettő keresése azonos, így egy plusz paraméter bevezetésével és általánosabb működéssel egy metódusban megvalósíthatóvá vált. A metódus egy mappa nevet vár a dll mellett, amiben a keresést végzi, és vesszővel elválasztva visszaadja fájlok elérési útját.

Két másik fontos osztály az XML-ek tényleges feldolgozásáért felelős osztályok a `TreeXMLReader` és a `EnvironemntXMLReader`. Ezek a TinyXML2 XML feldolgozó könyvtár segítségével elvégzik az XML-ek feldolgozását, és a szükséges objektumok létrehozását. Az XML-séma a fejlesztés során egyszer változott jelentősen, amikor néhány paraméterrel bővült a fák elágazási szabályrendszere. Ekkor módosítani kellett vele együtt a tároló és feldolgozó osztályokat is.

A fejlesztés elején sor került egy `Log` osztály létrehozására, ami egyszerű és jól átlátható naplózást biztosít mindössze egy header fájl includeolása után. A naplózásnál megadható egy üzenet, az üzenet szintje, ami lehet `Exception, Trace, Debug, Message, Warning, Cuda, Error`, valamint egy igaz-hamis érték, amivel kikapcsolható az adott log kiírása. A log kiírás szintje beállítható a statikus Log osztályon, ezzel meghatározva a minimum kiírandó szintet. A log szintje továbbá befolyásolja a konzolra írt üzenet színét is a könnyebb átláthatóság érdekében.

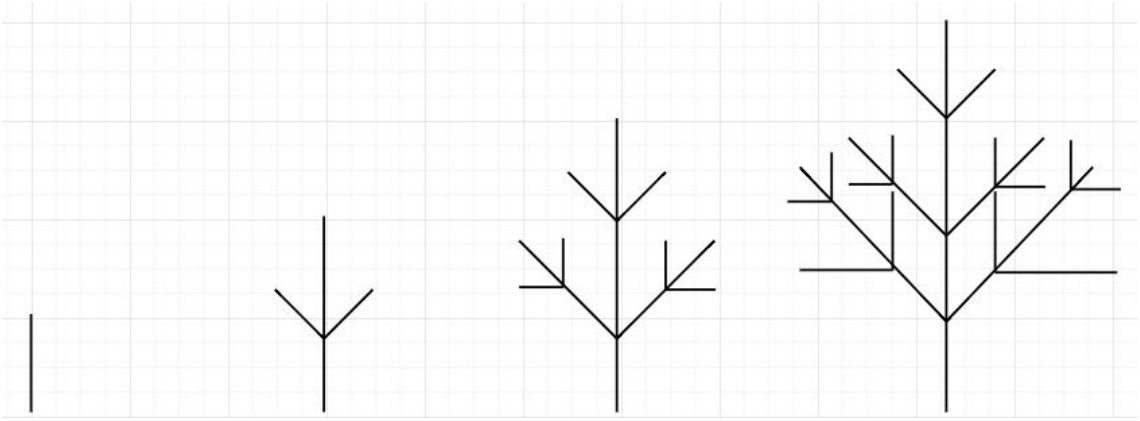
5.3. Adatszerkezetek

A fa tárolására egy fa adatszerkezet szolgál, aminek egy elemét a `Branch` osztály valósítja meg. Ez az osztály ment át a legtöbb változáson a fejlesztés során, és a jövőben is rengeteg módosítást tervezek rajta végrehajtani. A fa gyökerét egy, a `Tree` osztályon lévő ág példány alkotja, innentől ugyanolyan módon épül és növekszik a fa adatszerkezet, mint ahogy a valóságban is. Minden ágon található egy `std::vector<>` a leszármazott ágakkal, valamint két mutató, egy a fára (ami a gyökér elemet tartalmazza), egy pedig a saját szülő ágára.

A kezdeti verzióban a standard vectorok helyett listákat alkalmaztam, mivel még nem tudtam biztosan, hogy mennyire lesz lassú a szimuláció folyamata, és melyik részeket kell majd átírni később CUDA-ra. Viszont amikor elkészült egy alap verzió és látszódott, hogy a sebességgel itt még nem lesz probléma, refaktoráltam az egészet dinamikusan működő `std::vector<>` használatára. Ez szükségessé tette a korábbi, memória mennyiségek számítása, foglalása, majd szimuláció futtatása megközelítés megszüntetését, és lehetőséget adott a szimuláció közbeni memória foglalásra, ami lehetővé tette a szimuláció futtatási idejének megadását, a korábbi, előre adott maximum hossz helyett.

A fa osztályban a gyökér elemen kívül, ami a fa törzsét képezi, található még egy referencia a környezetre, valamint egy a növekedési szabályzatot meghatározó `Stage` osztályból képzett 5 elemű vektorra, ami a fa különböző életszakaszaihoz tartozó szabályokat tartalmazza. A szabályrendszer részletezése megtalálható a „2.2.4 Fák a szimulációban” fejezetben.

Az ágakon még fontos szerepe van egy szám vektornak, ami azt tartalmazza, hogy adott évben hány új ág jött létre. Ennek a vektornak a segítségével számolhatók az elágazás irányok, illetve az ágak törlésénél is fontos szerepe van. Minden ágon található még egy referencia a szabályrendszerre is, ezt alap esetben a szülő ágtól örökli, de lehetőség van a módosításra esélyt hagyva oltások szimulálására.



5.1. ábra. A növekedés 2D-s ábrázolása 0, 1, 2, 3 évben.

5.2. ábra. A szimuláció eredménye környezeti hatások nélkül 1-8 évre.

5.4. Fa növesztése

5.4.1. Módosítók hatása

A fa növesztésére az ágakon lévő `Grow` metódus szolgál, ami a következőképp működik. Elsőként a szabályrendszer kerül feldolgozásra, ami meghatározza az ág kezdeti növekedésének irányát. Ezután pedig a különböző `BranchModifier` leszármazottak fejtik ki hatásukat az ágra a következő sorrendben

1. `GrowDirectionModifier` Létrehoz egy egységvektort az ág korábbi növekedésének irányával.

2. `GrowLocationModifier` Transzformálja az egységvektort az ág korábbi végpontjába.

3. `LightModifier` Megállapítja a legerősebb látható fényforrás irányát és elforgatja az ág véget annak irányába (a forgatás mértéke súlyozással állítható).

4. `GrowLengthModifier` Meghosszabbítja az ág végi újonnan létrehozott egységvektort a szabályrendszer és a környezet figyelembe vételével.

5. `CollisionModifier` Amennyiben az ág valamilyen objektumnak ütközne, abban az esetben a növekedés az objektum síkjára vetítve folytatódik tovább.

6. `CurveModifier` Az ágat alkotó vezérlő pontokra egy ív számolódik (Centripetal Catmull–Rom spline algoritmus segítségével).

7. `SubBranchModifier` Meghívja az ág leszármazott ágainak a növekedés metódusát, ennek eredményeként mélységi bejárással növekszik a fa. Ez a modifier két lépést hajt végre, első lépésben növeszti a már létező leszármazott ágakat. Második lépésként pedig létrehozza az új adott évi ág hajtásokat.

8. `CalculateShadeModifier` Az ágakhoz tartozik egy leveleket reprezentáló háromszög lista, ami a saját magára, illetve környező fákra vetett árnyék kiszámítására szolgál. A leveleket egyenlő szárú háromszögek képzik, amiknek egyik csúcsa az n-1. kontroll pont, az alap felező pontja pedig az n. kontroll pont. A háromszögek mindig az x,y síkkal párhuzamosan állnak, a magasságukat az n-1. pont adja meg.

Ezek pontos működésének részletezése a „5.5 Módosítók hatásai” fejezetben található.

A módosítók alkalmazásakor fontos azok sorrendjének betartása, mivel azok sokszor számítanak az előttük végrehajtottak módosításaira. Ezek a módosítók eredetileg az ág osztály részét képezték, privát metódusok formájában, azonban a kód átláthatóságának megőrzése érdekében, szükségessé vált a külön osztályokba való kirendezésük (a „parancs” tervezési mintának megfelelően). Az ág osztályon csak egy vektor maradt a `BranchModifier` absztrakt osztály leszármazottaival, aminek feltöltése a konstruktorban történik meg, ilyen módon könnyen hozzáadható vagy elvehető egy-egy módosító.

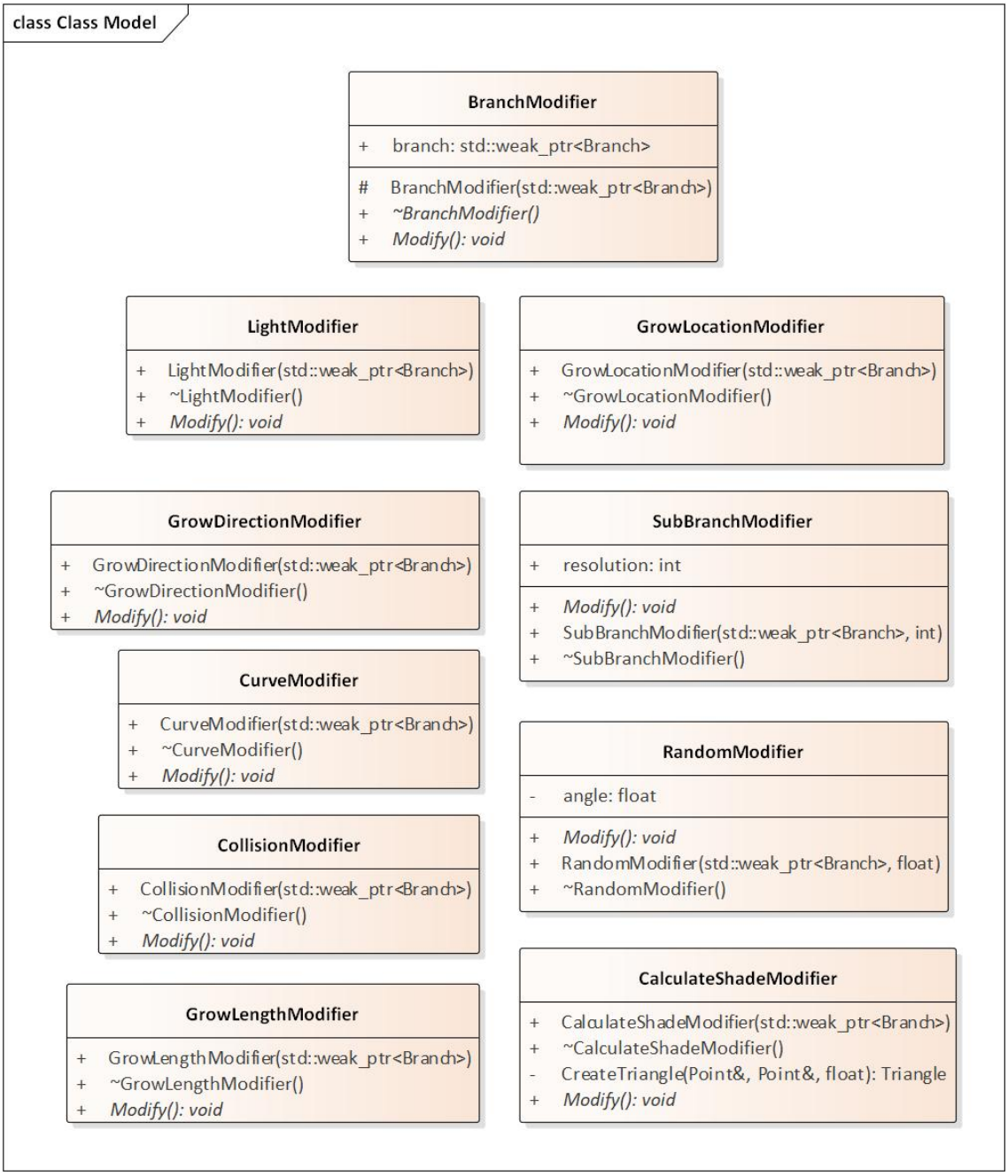
Az újonnan létrejövő ágak létrehozásáért a `BranchBuilder` osztály felelős. Az ág kezdeti elágazási helyét és irányát a `Branching` absztrakt osztály megfelelő leszármazottja végzi. Az osztály előír egy `SetDirection` és egy `SetLocation` metódust, amik a szülő osztály ismeretében adják meg az új ág helyét és irányát. Ezeknek az

elágazás típusoknak a részletezése a „Fák a szimulációban 2.2.4” fejezetben található. Mivel az elágazási szabályokat megvalósító algoritmusok egy `std::map`-ben találhatók, aminek a kulcsa az elágazás típusa, az értéke pedig egy `Branching` leszármazott példány, így könnyen hozzáadható új elágazás típus a rendszerhez. Ez korábban if-es és swich case-es megoldással működött, ami kevésbé átlátható és nehezebben bővíthető volt.

5.4.2. Centripetal Catmull–Rom spline

Ez az algoritmus képes négy kontroll pont közé olyan módon újabb pontokat beszúrni, hogy azok ívet alkossanak. A sok ívszámító algoritmus közül több okból is erre esett a választás, amik közül a legfontosabbak: a számított ív minden esetben érinti a kontrollként szolgáló pontokat, egyszerű megvalósítás és viszonylag kis számításigény mellett. Az algoritmus mindenképpen négy kontroll pontot vár egy ív előállításához, és csak a 2. és az n-1. pont közötti részre számol új pontokat. Ez a probléma könnyen megoldható az első és az utolsó pont kétszeri megadásával. Továbbá szabályozható az ív felbontása is, vagyis hogy két kontroll pont közé hány plusz pont kerüljön.

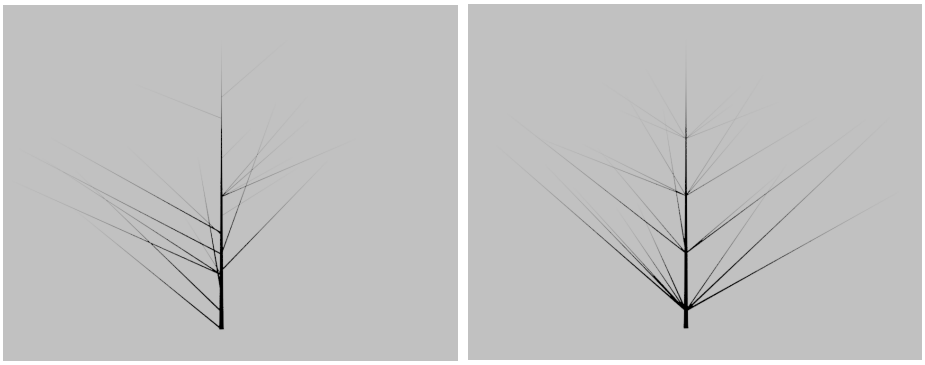
5.5. Módosítók hatásai



5.3. ábra. A módosítók diagramja.

5.5.1. Elágazás

Az elágazások iránya a szabályrendszernek megfelelően négyféle lehet: örvös, ellentétes, felváltott, spirális. Emellett az elágazás szöge és elfordulása is felelős az ágak alapirányának megadásáért. Az ágak szöge a szülő ágtól számított elfordulás. Az elfordulás pedig a szülő ág körüli elfordulást jelenti. Az ág helye az előző évben nőtt ívrész pontjai közül kerül kiválasztásra.

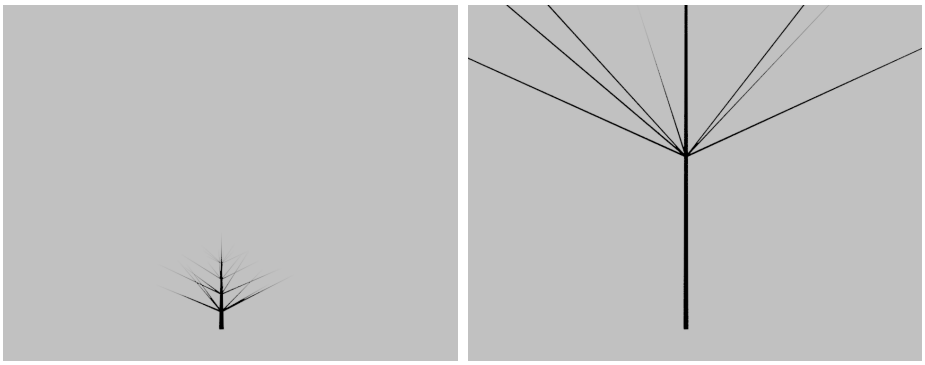


(a) Spirál elágazás (b) Örvös elágazás

5.4. ábra. Az alap növekedési iránya.

5.5.2. Növekedés hossza

A növekedés hosszát a szabályrendszerben meghatározott növekedési hossz adja meg.



(a) Kis növekedés (b) Nagy növekedés

5.5. ábra. Különböző növekedési mértékek.

- Örvös *(Whorled)* elágazás esetén az elágazás helye mindig az előző éves növekedés végét jelző ponttól kezdődik. Ezzel szintek alakulnak ki az ágak rendszerében. Ezek a szintek jól megfigyelhetők 5.4a ábrán. Az ágak elfordulása egyenletesen oszlik el az ágak között, ±5% véletlen bevezetésével.

- Ellentétes *(Opposite)* elágazás esetén minden **n**. ág helyzete véletlenszerűen kerül meghatározásra az előző évben nőtt ív szakasz pontjai közül, minden **n+1**. ág pedig ugyanarra a helyre kerül, mint az előző. Az ágak elfordulása ilyen esetben 180°az előző ághoz képest, így az ágak szembe kerülnek egymással. Az első ág elfordulását a szabályrendszerben megadott elfordulás adja.

- Felváltott *(Alternate)* elágazásnál a hely véletlenszerűen kerül kiválasztásra az előző évi növekedés ív részéről. Az elágazás elfordulása pedig a szabályrendszer szerint történik, de általában 180°.

- Spirál *(Spiral)* elágazás esetén (5.4b. ábra) az ág kezdőpontja véletlenszerűen kerül kiválasztásra az előző évben nőtt ív szakaszról. Az elfordulás pedig az előzőághoz képest a szabályrendszerben meghatározott szöggel több. Az első ág esetén pedig 0°.

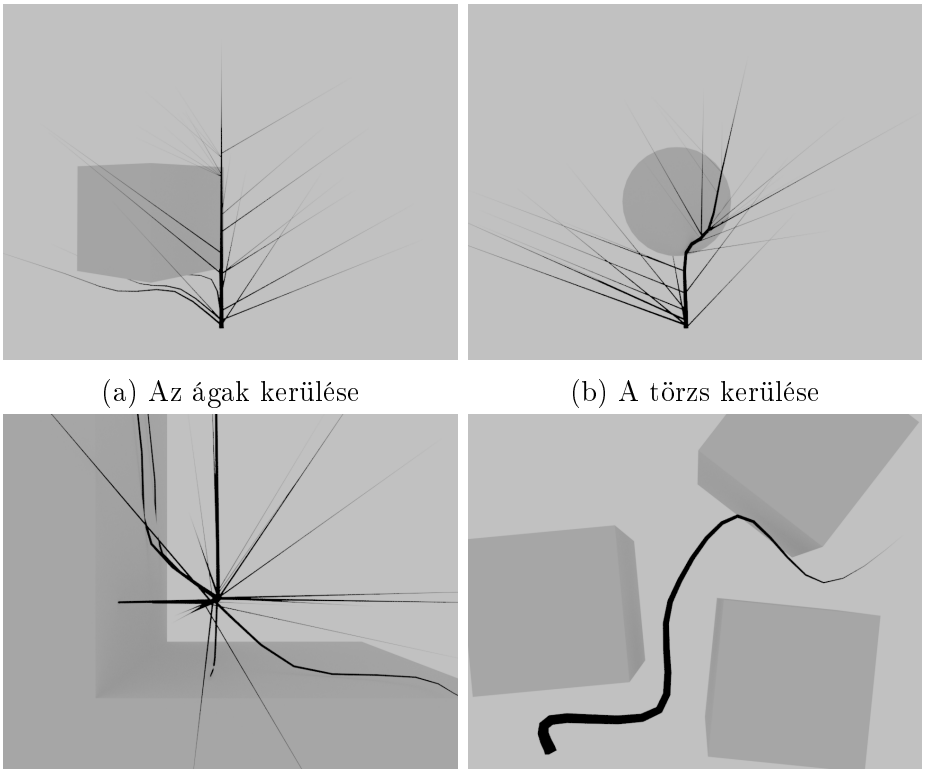
Az összes érték ±5% véletlen bevezetésével kerül felhasználásra a természetesebb hatás elérésének érdekében.

5.5.3. Ütközés detektálás

Az ütközés detektálás először megvizsgálja, hogy az ág találkozik-e háromszöggel a modelltérben. Ha nem, akkor a növekedés iránya nem módosul. Ha igen, akkor az ág növekedése módosul, a háromszög síkjára vetített irányban (5.6. ábra). Ez a megoldás két problémát hordoz magával, az egyik egy folyamatosan jelen lévő, a másik pedig egy ritkán előforduló. Az előbbi a futási idő jelentős megnövekedése az összes háromszög állandó ellenőrzése miatt. Az utóbbi pedig a vetítés természetéből következik. Ha az ág vektora pontosan merőlegesen érkezik a háromszög síkjára, akkor a vetített vektor a metszéspont lesz, ami az ág növekedésének megállását jelenti.

A fa fény felé növekedését a Sugárkövetés 2.4 fejezetben részletezett algoritmus végzi. A fény felé hajlás mértéke egy szorzóval állítható. Lehetőség van a sugárkövetést néhány egyszerű paraméterrel módosítani, amikkel a pontosság rovására futásidő nyerhető. Ezek a paraméterek a rekurzió mélysége és a fotonok száma. Amennyiben a rekurzió mértéke túl kicsi, akkor lehetséges, hogy az ágra közvetve esne ugyan fény, de a szimuláció nem találja meg azt. Ebben az esetben a fának látványosan sok ága nem kap fényt, és a végeredményen jól látható a hiba. Amennyiben túl nagy a rekurzió mértéke, az pedig a futásidő jelentős növekedésével jár.

A túl kevés mintavétel is hasonló tüneteket okoz, mint a túl kis rekurzió mélység. A túl nagy mintavétel nem okoz a valóságtól eltérő viselkedést, viszont nagyban növeli a futásidőt, sokszor feleslegesen.



(a) Az ágak kerülése (b) A törzs kerülése

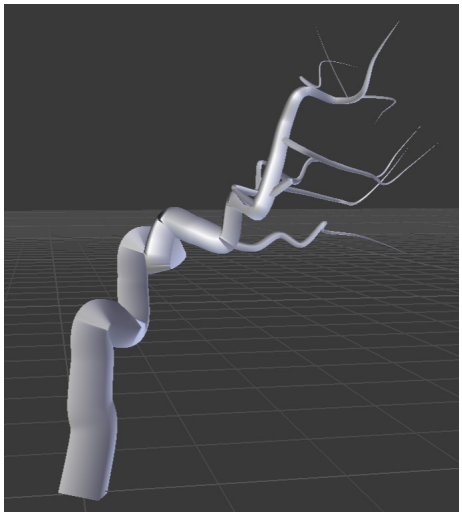
(c) Kerülés, felülnézetből (d) Egy ág bonyolultabb kerülése

5.6. ábra. Az ág viselkedése ütközés esetén.

A két érték optimumának megtalálásánál korábbi sugárkövetéssel kapcsolatos munkáim során szerzett tapasztalataimra hagyatkoztam. Ezek a programok a szimuláció előkészítéseként születtek, a fény keresési algoritmusom koncepciójának igazolására. Tapasztalataim alapján a minimum egy maximum három mélységű rekurzió, 512 és 1024 közötti mintavételi érték mellett a legmegfelelőbb a futási idő és a pontosság szempontjából. A pontosság kedvéért ellenőriztem különböző esetekre az értékeket a tényleges fa szimulációkon is, itt a fákon könnyedén látható, ha a megadott értékek túl kicsik, ez látható az 5.7. ábrán.

Megfelelő értékek esetén a a fa a fény felé növekszik, és megvan az elvárt mennyiségű ága is (5.8. ábra).

Ha **B** ágat nem éri fény akkor a `no light from` értéke 1-ről 0-ra vált, ezzel jelezve, hogy az ág nem kapott fényt. Ha **B**-ről leágazó **C** ágat éri fény, akkor **B** ág `no light from` értéke a **C** ág keletkezésének helyére módosul. A növekedési módosítók futása után egy újabb `Clear` metódus járja végig az ágakat, és törli azokat, amiknek a `no light from` értéke 0, és a megfelelő helyen levágja azokat, amiknek az értéke nagyobb mint nulla (5.8. ábra).



5.7. ábra. A fa növekedése nem megfelelő fény keresési beállítások mellett.

5.8. ábra. A fa növekedése a fény felé.

5.6. CUDA

Teljesítmény tesztek alapján kiderült, hogy a szimuláció futási idejének legnagyobb részét a fény keresésével tölti, mivel rengeteg ütközés vizsgálatára van szükség.

A sugárkövetéses fénykeresés lassú folyamat, ezért szükség volt egy gyorsabb algoritmus elkészítésére. Mivel az algoritmus további optimalizálása már nem gyorsított a folyamaton, ezért általános célú GPU programozással, és a videokártya minél nagyobb szintű kihasználásával kezdtem a további gyorsítást.

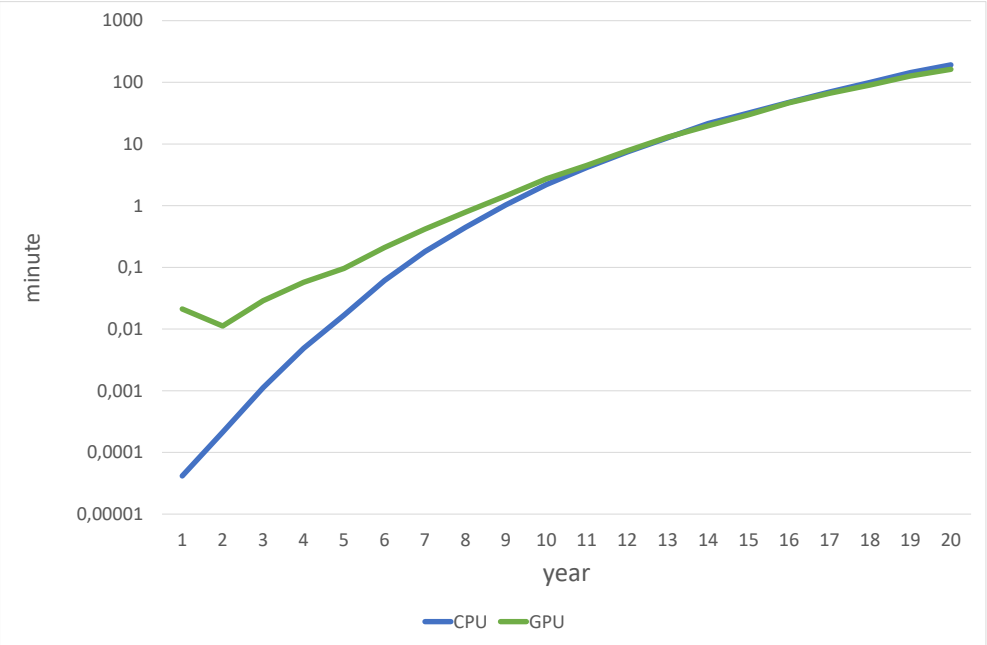
A saját pont osztályom helyett itt a CUDA nyelvbe épített float3 struktúrát használtam. Ehhez szükség volt az általános vektor műveletek implementálására a float3 típusú tagokra is. A háromszögek reprezentálására is új CUDA kompatibilis struktúrát és hozzá tartozó metódusokat kellett implementálni.

A GPU gyorsított Trace algoritmus ugyanazon az elven működik, mint a CPU változata, csupán annyi különbséggel, hogy a különböző utakat nem szekvenciálisan, hanem párhuzamosan számolja.

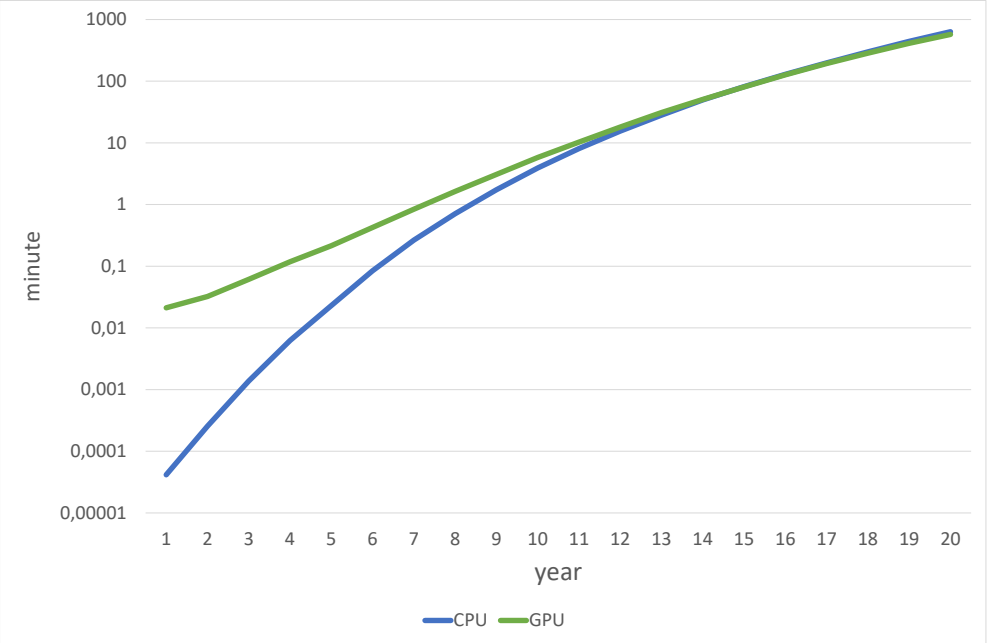
A grafikus kártyán történő számítás esetén mindig szükség van az adatok oda-vissza másolására, mivel az a processzortól különálló memória területen dolgozik. Ezek a memória műveletek költségesek, ezáltal lassúak is.

Az első tervezett GPU gyorsított verzió a CPU verziót leváltva áganként hívódott meg. Az egyszerű felépítésnek köszönhetően, miszerint a GPU verzió adatok másolása, számítás és adatok visszamásolása lépésekből állt, melyek külön metódusként valósultak meg, nem kellett a korábbi kódot módosítani, csupán a `CPUTrace` osztályt lecserélni a `GPUTrace` osztályra. Ez a verzió (bár jó ötletnek tűnt és a tényleges fénykeresést gyorsabbá tette), a korábbiakban említett rengeteg adatmozgatás miatt valójában nem gyorsított a szimuláción. Ezt ábrázolja az 5.9a és az 5.9b ábra.

Ennek kiküszöbölésére kellett találni egy megoldást. Az adatmozgatások és memóriaműveletek mennyiségét csökkenteni az első ötlet mellett nem lehetett, mivel minden ág számításával frissültek az adatok, és minden ágnak más adatra volt szüksége. Ezért egy egész más megoldáshoz folyamodtam. Az ágakat reprezentáló osztályokon bevezetésre került egy fény irányát jelölő változó, ami a megfelelő módosító meghívásakor kerül alkalmazásra. A fény iránya viszont nem a módosító meghívásakor számolódik áganként, hanem a szimuláció adott évének elején, az összes ágra egyszerre. Ezzel a korábban az ágak mennyiségétől függő, évente akár ezres nagyság-



(a) Futási idő évről évre (log skála)



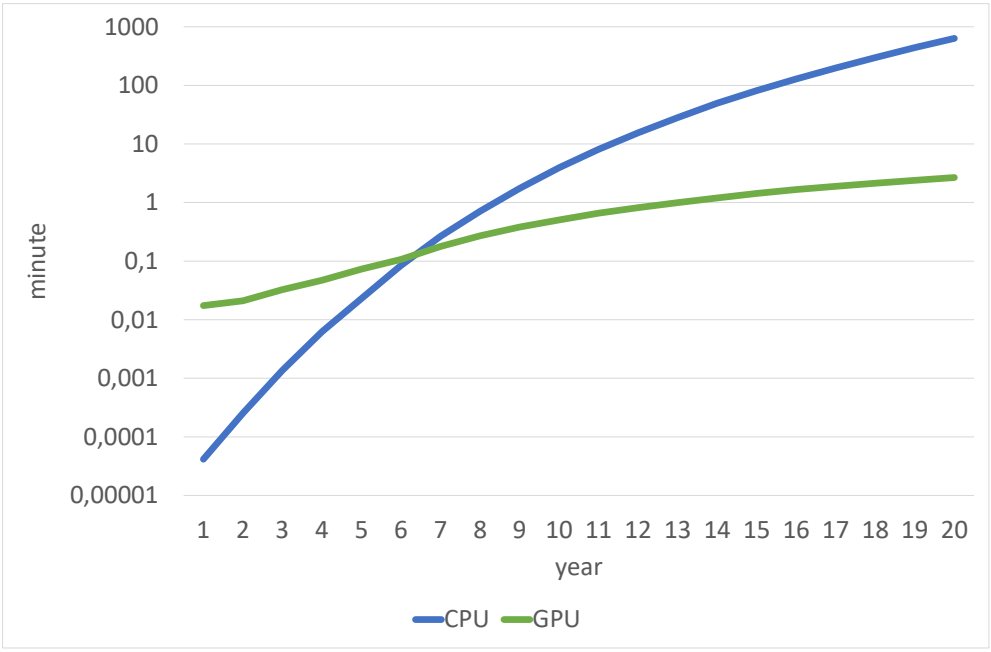
(b) Összegzett futási idő

5.9. ábra. CPU-GPU első verzió futási ideje (log skála)

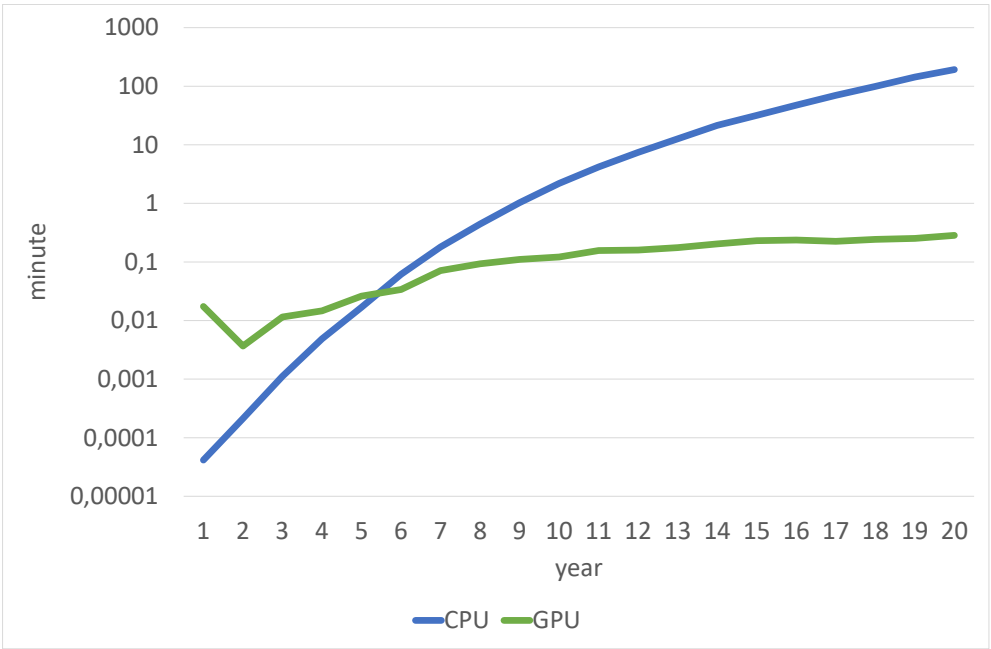
rendű redundáns adatmozgatás, évi egyre csökkent. Mivel az összes ág ugyanabban a környezetben van, és a fényszámítás során nem módosítják azt, így csupán egyszer szükséges a fényforrások helyzetének és az árnyékoló objektumoknak az átmásolása.

Ezzel a megoldással már jelentős sebességnövekedés volt elérhető változatlan pontosság mellett. A következő diagramok a program futásának sebességét mutatják kétféle bontásban. Az 5.10a. ábra évről évre mutatja a számítási időt, az 5.10b. ábra pedig az adott évig eltelt összes időt mutatja. Mivel nagyobb elemszám esetén a GPU futásidő szinte elhanyagolható a CPU-hoz képest, ezért logaritmikus skálán ábrázoltam az eredményeket.

A futásidők vizsgálata során az alábbi konfigurációt használtam: CPU: `Intel Core i5-2500K`, GPU: `NVIDIA GeForce GTX 1070`, RAM: `8GB DDR3`, HDD: `KINGSTONE SV300S37A120G`.



(a) Futási idő évről évre (log skála)



(b) Futási idő

5.10. ábra. CPU-GPU második verzió futási ideje (log skála)

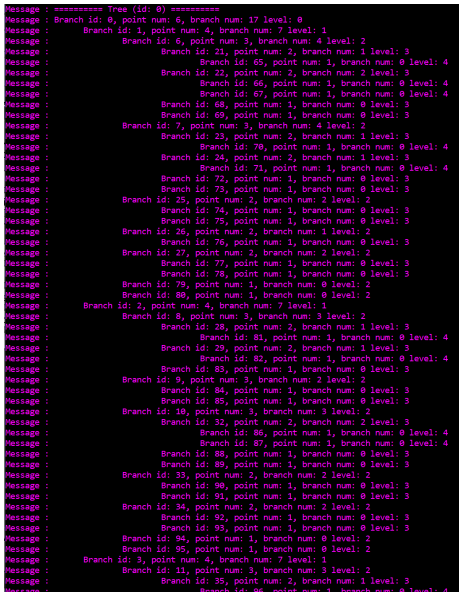
6. Tesztelés

Egy szimulációs módszer tesztelése mindig esetleges, hiszen maga a modell mindig csak egy leegyszerűsítése a valós folyamatoknak. Az összehasonlítást pedig tovább nehezíti a véletlen események szerepe (arról már nem is beszélve, hogy egy fa növesztése évtizedeket igényelne). Ezért a matematikai modellt tudtam csak egzakt módon ellenőrizni, hogy az tényleg az elvártaknak megfelelően működik.

Magát a tényleges szimulációt pedig egyrészt részlépésein keresztül (elágazás, fény felé fordulás, stb.) vizsgáltam, illetve szakértők (erdészek) segítségét kértem. Matematikailag kielégítő validálást természetesen ők se tudtak adni, de tapasztalataik alapján elmondható, hogy a modell által növesztett fák megfelelnek a valóságban várhatónak.

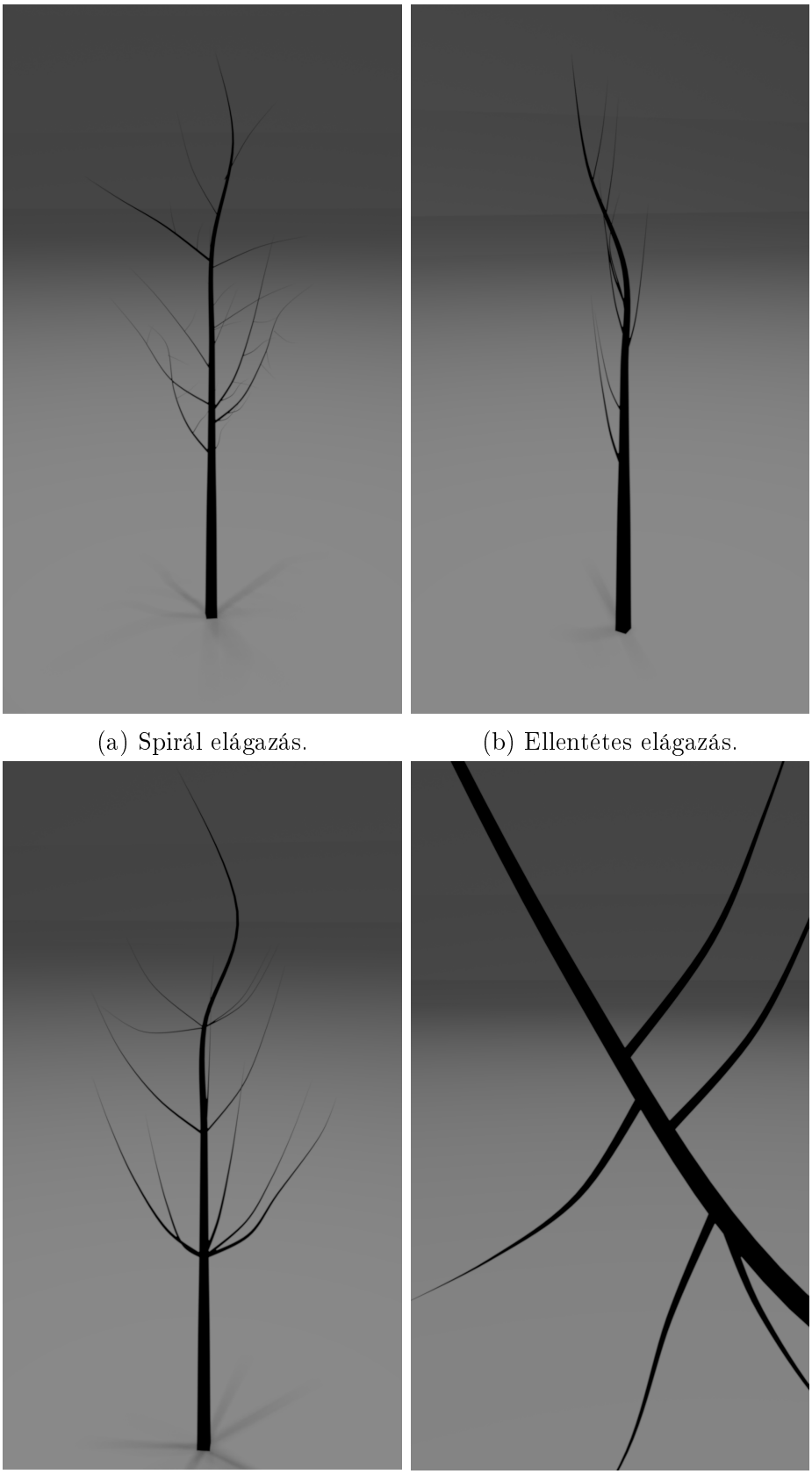
6.1. Elágazás tesztek

Az elágazások megfelelő helyének és mennyiségének ellenőrzésére a 6.1 ábrán látható log kiírás használható.



6.1. ábra. Fa szerkezete a logban.

Az elágazások vizsgálatára a törzsről leágazó ágak megfigyelése adja a legjobb lehetőséget. Ezért ezek megtekintéséhez csak kettő szintet alkalmazó szimulációt használtam. Az elágazások teljesen megfelelnek a szabályok által megadottnak, és láthatóan hasonlítanak a természetben láthatóhoz (6.2. ábra).



(a) Spirál elágazás. (b) Ellentétes elágazás.

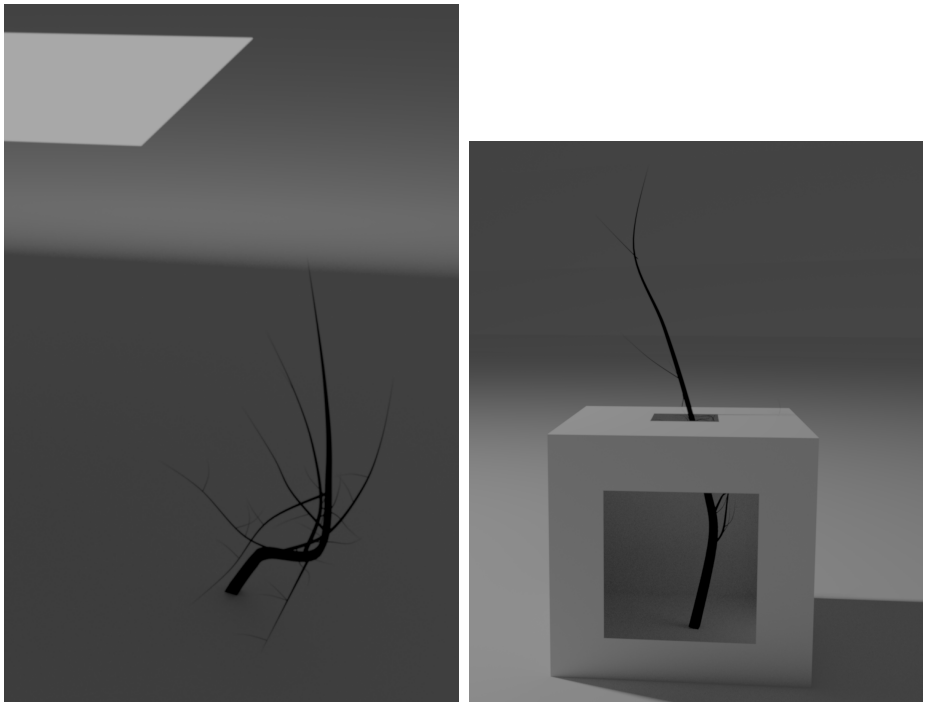
(c) Örvös elágazás. (d) Felváltott elágazás.

6.2. ábra. Elágazások vizuális ellenőrzése.

6.3. ábra. Változó fényirány.

6.2. Fénykövetés tesztek

A 6.3 ábrán látható, hogy ha a fa növekedése során minden évben módosul a fényforrás helyzete, és ezzel a fény érkezésének iránya, akkor a fa követi az új irányt. Az ábrán látható fa szimulációja során a fényforrás minden évben átkerült a fa ellentétes oldalára. Jól látható, hogy a növekedés iránya is ennek megfelelően dinamikusan változott.



(a) Fénykeresés csak tükröződő felülettel

(b) Fénykeresés környezet dinamikus változásával

6.4. ábra. Fénykeresés vizuális tesztelése.

Ha teljes árnyékban van a fa, akkor is képes a tükröződő fények alapján meghatározni azt a növekedési irányt, amivel fényre kerülhet (6.4a. ábra).

A 6.4b. ábrán a kamera felé néző nyílást csak a fa növesztése után vágtam ki, hogy látható legyen a belső viselkedés is. A törzs először a bal oldal irányába hajlik, mivel onnan tükröződik a felülről beszűrődő fény. Viszont amikor a fényforrás látótérbe került, onnantól módosul a növekedési irány. A lentebb látható ágak nem találták meg a fényforrást, így bent ragadtak a dobozban. Ez a teszt részben az ütközéseket is ellenőrzi.

6.3. Ütközés tesztek

Az ütközések működésére néhány további példa látható a „5.5.3 Ütközés detektálás” fejezet 5.6 ábráján. Mindegyiken jól látható, hogy a fa növekedése során észlelte ha nekiütközött valaminek, és az előre megadott szabályok alapján automatikusan irányt változtatott. Ez az eredmény teljesen megegyezik a természetben tapasztalhatóval, a növény ágai kikerülik az akadályokat.

6.4. Több fa növesztése egy időben

A 6.5 ábrán látható több fa egy időbeni szimulálása. A hátsó fa 3 évvel idősebb a társainál, a többi fa csak annak a növesztése után került a szimulációs térbe.

6.5. ábra. Több fa párhuzamos növesztése.

7. Eredmények értékelése

7.1. Elért eredmények

A kutatás célja egy olyan szimulációs modell létrehozása és implementálása volt, amellyel egy vagy akár több fa növekedését lehet vizsgálni az adott fajta jellemzőinek, illetve a környezeti hatásoknak megfelelően. A kitűzött célokat maradéktalanul sikerült elérnem, a tesztelés alapján elmondhatom, hogy sikerült megvalósítani a szükséges funkciókat.

Mint minden szimulációnak, ennek is egy részletesen kidolgozott matematikai modell az alapja. Az irodalomfeldolgozás során megvizsgáltam a már meglévő eredményeket, majd ezek alapján a saját ötleteimmel kiegészítve elkészítettem a szükséges modellt. Ez tartalmazza a fák növekedésével kapcsolatos biológiai folyamatokat (növekedés, elágazások, stb.), amelyek részben az adott fa fajtától is függnek, ezért számos tényező paraméterként jelenik meg. A fák faji tulajdonságait leíró szabályrendszer rengeteg különböző fa előállítására ad lehetőséget. Kicsit szabadabb értelmezésével akár cserjék és lágyszárúak is előállíthatóak.

A modell alapján már el tudtam végezni a szükséges szimulációkat, amelyek megadják a fák növekedési folyamatát. Munkámban egyedinek tekintem, hogy az eredmények nem csak egyszerű számokként jelennek meg, hanem ezek alapján egy tényleges háromdimenziós fa modellt növesztek, amely így vizuálisan is jól ellenőrizhető, és a gyakorlatban is sokrétűen használható (park rendezés, erdőgazdálkodás, animációk, játékfejlesztés, stb.).

A modell részeit képezik a különféle fizikai lépések, mint például az ütközések kezelése, vagy a gravitáció hatása. Ezeknek köszönhetően a fa reagál a környezetére, a növekedése során az objektumokat igyekszik elkerülni (a természetben is tapasztalható módon), stb.

Szintén a modell fizikai részéhez tartozik, de kiemelném a kulcsfontosságú fény vizsgálatot. A fa növekedése szempontjából kritikus megállapítani a domináns megvilágítás irányát, amely egy meglehetősen komplex sugárkövetési feladat (közvetlen megvilágítás, tükröződő fény, árnyékoló objektumok, a fa önmagára vetett árnyéka, levelek szerepe, stb.). Erre kidolgoztam egy Monte Carlo módszeren alapuló megoldást, amivel jól lehet közelíteni a természetben is meglévő állapotokat.

Mivel még ez a módszer is meglehetősen számításigényes, ezért grafikus gyorsítók alkalmazásával implementáltam a sugárkövetést végző alrendszert. Mivel a futásidő szempontjából ez a program legkritikusabb része, így ezzel a teljes futásidőt is jelentősen sikerült csökkenteni. Részletes futásidőre vonatkozó eredmények megtalálhatók az 5.9a–5.10b. ábrákon.

Szintén egyedinek tekinthető, hogy az általam kidolgozott modell nem tartalmaz korlátot az egy időben növelni kívánt fák számára, így egyszerre akár egy kisebb fa csoport vizsgálata is megoldható. Az egyes fák hatással vannak egymásra, ennek megfelelően összetett esetek vizsgálatára is lehetőség nyílik. A szimulációban pedig akár minden fa lehet különböző fajú.

Rendszeremet alaposan teszteltem. Szimuláció lévén a tesztelés meglehetősen nehézkes, de számos vizuális teszteset alapján elmondható, hogy a modell jól működik. Az általa szolgáltatott eredmények a szakértők szerint is közel állnak a természetben tapasztalhatóhoz (7.1. ábra).

(a) Élő fa (b) Szimulált fa

7.1. ábra. 5 éves tölgyfa és szimulált fa.

Sikerült megvalósítani azt a követelményt is, hogy az elkészült modell rugalmasan beépíthető legyen más rendszerekbe. Az általam készített alkalmazás alapvetően egy dll, amit tetszőleges programba be lehet ágyazni.

Ennek legjobb demonstrációja az elkészült Blender kapcsolat. Ennek segítségével a modell beépíthető a Blender pluginjei közé, ezzel annak funkciói a nagy rendszer részeként használhatók. Ez egy kimondottan felhasználóbarát kezelést tesz lehetővé. A modellező rendszer saját funkcióival van lehetőség létrehozni a modellteret (fényforrások, akadályokat képző objektumok), majd abban egy gombnyomással el lehet indítani a szimulációt, aminek végeredménye egy 3D fa lesz a modelltérben, ami növekedése során alkalmazkodott a környezetéhez, majd ezt követően szabadon alakítható tovább a Blender alapvető funkcióival.

Kiemelném azt, a szintén egyedinek tekinthető jellemzőt, hogy a szimuláció jellegéből adódóan a növekedés bármelyik fázisában meg lehet azt állítani, át lehet alakítani körülötte a modellteret (új akadályok, fényforrások, stb.) és így lehet folytatni a műveletet. Ezzel lehetőség nyílik számos összetettebb probléma vizsgálatára.

7.2. Kutatás hatása

Eredményeim a gyakorlatban közvetlenül is hasznosíthatók. Egyik közvetlen felhasználási terület a 3D modellezés, amit már most is ki tud szolgálni, hiszen az előre megadott paraméterek alapján a környezet objektumait figyelembe vevő, egészen élethű modelleket tud generálni.

A másik felhasználási terület a tényleges parkrendezési és faipari környezetben való elemzés. Családi ismerettség révén kapcsolatban állok erdészekkel (akik a fejlesztés során is nagyon sokat segítettek a fák viselkedésének megismerésében), akik nagyon türelmetlenül várták az első stabil verziók elkészültét. Elmondásuk szerint a generált modellek tényleg megfelelnek az elvárásoknak, és közvetlenül jól tudják a rendszert használni a napi munkájuk során.

És természetesen lehetőség van a különféle környezeti viszonyokkal kapcsolatos összehasonlító vizsgálatokra, pl. az éghajlatváltozás ügyében. A szép 3D modelleken túl a szimuláció képes egzakt, kvantitatív adatok szolgáltatására is.

Eredményeimet konzulensemmel együtt publikálni szeretnénk a „IEEE 13th International Symposium on Applied Computational Intelligence and Informatics" konferencián. Az ehhez szükséges cikket beadtuk, jelen dokumentum lezárásakor sajnos még nem érkezett meg a visszaigazolás.

7.3. Továbbfejlesztési lehetőségek

A fejlesztés során a rendszer minél összetettebb lett, annál több ötlet merült fel a továbbfejlesztési irányokkal kapcsolatban. Ezeket alapvetően az alábbi csoportokba lehet rendezni:

- Fizikai környezet finomítása: pl. uralkodó erős szélirány figyelembevétele, stb.
- Biológiai modell finomítása: pl. talaj minőségének figyelembevétele, fa visszahatása a talaj minőségére, stb.
- Megjelenítés fejlesztése: pl. levelek megjelenítése, textúrázás, stb.
- Integrálás más környezetekbe: pl. Unreal Engine 4, Unity, 3ds Max, Maya

Irodalomjegyzék

[1] FU Tian shuanga et al. “Tree Modeling and Dynamics Simulation”. *Physics Procedia* 33 (2012), 1710–1716. old.

[2] James Arvo et al. *State of the Art in Monte Carlo Ray Tracing*. Stanford University, 2001.

[3] Jean-François et al. “A Structural Whole-plant Simulator Based on Botanical Knowledge and Designed to Host External Functional Models”. *Annals of Botany* 101 (8 2008), 1125–1138. old.

[4] John Hart et al. “Structural simulation of tree growth and response”. *The Visual Computer* 19.2 (2003), 151–163. old.

[5] Katsuhiko Onishi et al. “Interactive modeling of trees by using growth simulation”. *Proceedings of the ACM symposium on Virtual reality software and technology*. ACM. 2003, 66–72. old.

[6] *Blender Documentation*. 2018 (letöltve: 2018.11.08). URL: `https://docs.blender.org/api/current`.

[7] Daniel Chamovitz. *What a Plant Knows: A Field Guide to the Senses*. Scientific American, 2012. ISBN: 978-0-374-28873-0.

[8] *CUDA Toolkit Documentation*. 2018 (letöltve: 2018.11.08). URL: `https://docs.nvidia.com/cuda`.

[9] *Extending Python with C or C++*. 2011 (letöltve: 2018.11.08). URL: `http://download.autodesk.com/global/docs/3dsmaxsdk2012/en_us/index.html`.

[10] *Extending Python with C or C++*. 2018 (letöltve: 2018.11.08). URL: `https://devblogs.nvidia.com/building-cuda-applications-cmake/`.

[11] Schmidt Gábor és Tóth Imre. “Kertészeti dendrológia”. (2006).

[12] *gtest*. 2018 (letöltve: 2018.11.08). URL: `https://github.com/google/googletest/tree/master/googlemock`.

[13] Bill Hoffman és Ken Martin. *Mastering CMake*. Kitware, 2015. ISBN: 978-1-930934-31-3.

[14] Petr Horácek. "Introduction to tree statics & static assessment". *Tree statics and dynamics seminar, interpreting the significance of factors affecting tree structure & health, Westonbirt, UK*. 2003.

[15] *Hybridizer: High-Performance C# on GPUs*. 2018 (letöltve: 2018.11.08). URL: `https://devblogs.nvidia.com/hybridizer-csharp`.

[16] Catherine Jirasek és Przemyslaw Prusinkiewicz. "A biomechanical model of branch shape in plants". *Proceedings of the western computer graphics symposium, Whistler, Canada*. Citeseer. 1998, 23–26. old.

[17] Catherine Alena Jirasek. "A Biomechanical Model of Branch Shape in Plants Expressed Using L-Systems". Dissz. University of Calgary, 2000.

[18] Demeter Deli Kristóf. "Fizikai alapú képszintézis a GPU-n". Budapesti Műszaki és Gazdaságtudományi Egyetem, 2016.

[19] Aristid Lindenmayer. *Mathematical Models for Cellular Interactions in Development*. II. Simple and Branching Filaments with Two-sided Inputs köt. City University of New York, New York, 1968.

[20] Robert Maynard. *Building Cross-Platform CUDA Applications with CMake*. 2017 (letöltve: 2018.11.08). URL: `https://devblogs.nvidia.com/building-cuda-applications-cmake/`.

[21] Harry Perros. *Computer Simulation Techniques*. NC State University, 2009.

[22] *speedtree main page*. 2018 (letöltve: 2018.11.08). URL: `https://store.speedtree.com`.

[23] Bjarne Stroustrup. *A C++ programozási nyelv I, II.* 2001. ISBN: 978-9-639301-18-4.

[24] *TinyXML-2*. 2018 (letöltve: 2018.11.08). URL: `http://leethomason.github.io/tinyxml2`.

[25] Jason Weber és Joseph Penn. "Creation and rendering of realistic trees". *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. ACM. 1995, 119–128. old.

Ábrák jegyzéke