



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

ALGORITMUSOK ÉS ALKALMAZÁSAIK

TANSZÉK

## Távolságfüggvények generálása gráf alapú objektummodellből

*Témavezető:*

Bán Róbert

MSc, doktorandusz

*Szerző:*

Karikó Csongor Csanád

programtervező informatikus BSc

*Budapest, 2022*

## SZAKDOLGOZAT TÉMABEJELENTŐ

**Hallgató adatai:**

Név: Karikó Csongor Csanád

Neptun kód: JPOIWR

**Képzési adatok:**

Szak: programtervező informatikus, alapképzés (BA/BSc/BProf)

Tagozat : Nappali

Belső témavezetővel rendelkezem

*Témavezető neve: Bán Róbert*

*munkahelyének neve, tanszéke: ELTE-IK, Algoritmusok és Alkalmazásai Tanszék*

*munkahelyének címe: 1117, Budapest, Pázmány Péter sétány 1/C.*

*beosztás és iskolai végzettsége: Doktorandusz, MSc*

**A szakdolgozat címe:** Távolságfüggvények generálása gráf alapú objektummodellből

**A szakdolgozat témája:**

*(A témavezetővel konzultálva adja meg 1/2 - 1 oldal terjedelemben szakdolgozat témájának leírását)*

Előjeles távolságfüggvénynek nevezünk egy, a háromdimenziós tér tetszőleges pontjához, a pont egy felülettől vett előjeles távolságát megadó egyértelmű hozzárendelést. Egyszerű felületek távolságfüggvénye könnyen megadható, összetettebb esetben primitívek távolságfüggvényeinek kombinálásával konstruálható. Ez a modellezési eljárás Constructive Solid Geometry (CSG) néven is ismert, melynek során primitív objektumokból (például gömb, henger, téglatest) halmazműveletekkel és geometriai transzformációkkal állítjuk elő a kívánt objektumot.

A szakdolgozat célja egy gráf alapú szerkesztőprogram megvalósítása, melyben CSG technikával összeállítható egy háromdimenziós modell. A programban egy fagráfot építünk, melynek levelei primitívek, belső csúcsai pedig halmazműveletek vagy transzformációk. A fa gyökere reprezentálja a kapott végeredményt. A program lehetőséget biztosít az elkészített modell mentésére és betöltésére, valamint az egyes csúcsokban a modell állapotának 3 dimenziós valós idejű megjelenítésére is. Utóbbi a felhasználó számára lényegesen leegyszerűsíti a szerkesztési folyamatot és az esetleges szerkesztési hibák megkeresését, javítását. Ezen kívül a program automatikusan generálja a reprezentálni kívánt felület távolságfüggvényét, illetve egyéb árnyalási technikák GPU kódját.

A megvalósítás C++ programozási nyelvet használ, OpenGL grafikus könyvtárral. A felhasználói felület Dear ImGui segítségével kerül megvalósításra, míg a generált GPU kód GLSL nyelvű (OpenGL Shading Language).

Budapest, 2021. 11. 30.

# Tartalomjegyzék

<b>I.</b>	<b>Bevezetés</b>	<b>4</b>
<b>II.</b>	<b>Felhasználói dokumentáció</b>	<b>10</b>
1.	A feladat rövid ismertetése . . . . .	11
2.	Hardveres és szoftveres követelmények . . . . .	11
3.	Telepítés és futtatás . . . . .	11
4.	Az alkalmazás használata . . . . .	11
4.1.	Szerkesztő ablak . . . . .	11
4.2.	Menü . . . . .	12
4.3.	Megjelenítő ablak . . . . .	14
5.	Vizualizációs módok . . . . .	14
6.	Primitívek és műveletek . . . . .	14
6.1.	Közös tulajdonságok . . . . .	15
6.2.	Primitívek . . . . .	16
6.3.	Műveletek . . . . .	16
<b>III.</b>	<b>Fejlesztői dokumentáció</b>	<b>19</b>
<b>1.</b>	<b>Fejlesztői környezet</b>	<b>20</b>
1.1.	Felhasznált könyvtárak . . . . .	20
1.2.	Fordítás . . . . .	21
1.3.	Futtatási környezet . . . . .	21
<b>2.</b>	<b>Specifikáció</b>	<b>22</b>
2.1.	Funkcionális követelmények . . . . .	22
2.1.1.	Felhasználói történetek . . . . .	22
2.1.2.	Felhasználói eset diagramm . . . . .	26

<b>3. Elméleti háttér</b>	<b>28</b>
3.1. Fogalmak . . . . .	28
3.2. Sphere tracing . . . . .	29
3.2.1. Normálvektorok approximálása . . . . .	30
3.3. Duális számok . . . . .	32
3.3.1. Alapműveletek . . . . .	32
3.3.2. Függvények . . . . .	33
3.3.3. Automatikus differenciálás . . . . .	34
3.3.4. Kiterjesztés magasabb deriváltakra . . . . .	34
3.3.5. Kiterjesztés többváltozós függvényekre . . . . .	36
3.4. Görbületek . . . . .	41
<b>4. Megvalósítás</b>	<b>42</b>
4.1. Constructive Solid Geometry . . . . .	42
4.1.1. CSG Gráf . . . . .	42
4.1.2. Transzformációk részgráfokon . . . . .	46
4.2. Megjelenítés . . . . .	46
4.3. Kódgenerálás . . . . .	47
4.3.1. Transzformációk kezelése - a transzformációs verem . . . . .	49
4.3.2. Automatikus differenciálás . . . . .	51
<b>5. Architektúra</b>	<b>58</b>
5.1. main függvény . . . . .	58
5.2. App osztály – megjelenítés és vezérlés . . . . .	60
5.3. Editor osztály . . . . .	61
5.4. NodeVisitor osztály – Gráfbejárások . . . . .	64
5.5. Persistence osztály – mentés és betöltés . . . . .	65
5.5.1. Szerializálás és deszerializálás . . . . .	65
5.5.2. NodeJsonSerializer osztály . . . . .	65
5.6. Shader generálás . . . . .	66
5.6.1. SDFGenerator és DifferentiatedSDFGenerator . . . . .	66
5.6.2. ShaderLibManager osztály . . . . .	66
5.7. GuiNode és Node típusok . . . . .	67
5.7.1. „Típusleíró” osztályok: Primitive és Operator . . . . .	67
5.8. Bővítés új primitívekkel . . . . .	68

<b>6. Tesztelés</b>	<b>71</b>
6.1. Tesztelési jegyzőkönyv . . . . .	72
6.2. Problémák . . . . .	79
6.2.1. Hardveres limitációk . . . . .	79
6.2.2. Lassú shader fordítás . . . . .	81
6.2.3. Driver összeomlások (Nvidia) . . . . .	82
6.3. Teljesítmény tesztek . . . . .	82
6.3.1. Összegzés . . . . .	86
 <b>IV. Összegzés</b>	 <b>87</b>
 <b>7. További fejlesztési lehetőségek</b>	 <b>89</b>
 <b>Irodalomjegyzék</b>	 <b>91</b>
 <b>Algoritmusjegyzék</b>	 <b>92</b>
 <b>A. Duális GLSL függvénykönyvtár</b>	 <b>93</b>

## I. rész

### Bevezetés

---

Az informatikán belül a számítógépes grafika területének központi eleme a különböző háromdimenziós modellek reprezentációjának és megjelenítésének vizsgálata. A grafika eddigi fejlődése során számos megközelítés látott napvilágot. Ezek közül az egyik leginkább meghatározó a felület háromszögráccsal történő, közelítő megadása. Ezen kívül felületeket reprezentálhatunk matematikai módon, azok implicit, explicit vagy parametrikus képletének megadásával. Ezen megoldások leginkább a geometriai modellezésben használatosak, például mérnöki tervezőszoftverekben. Ebben a dolgozatban implicit felületekkel foglalkozom.

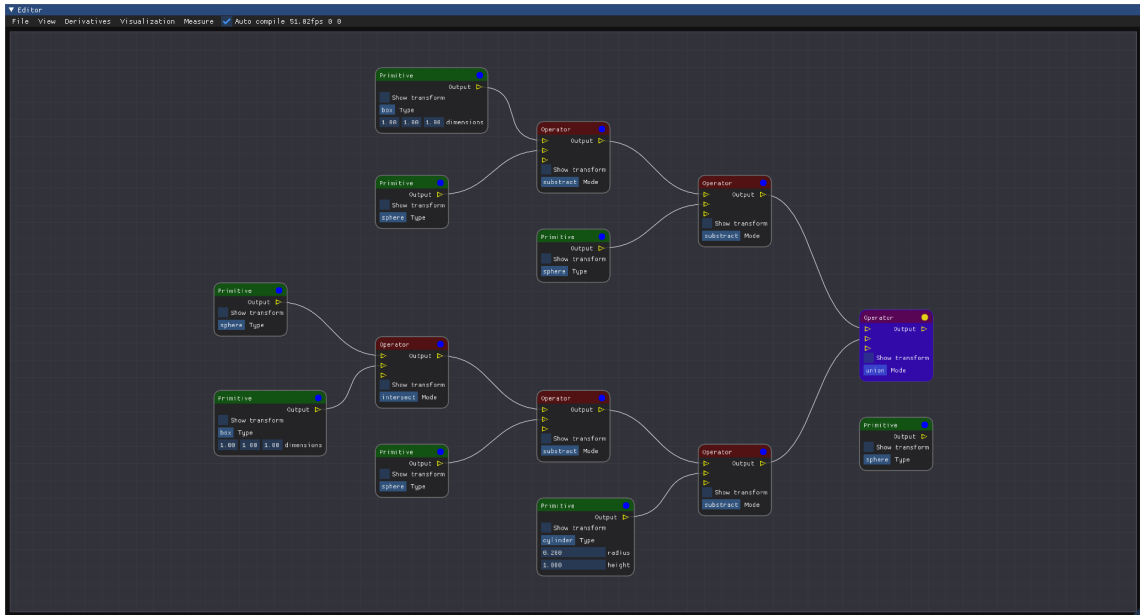
Szakdolgozatom célja egy egyszerű háromdimenziós modellezőszoftver megvalósítása, amely a szilárdtestmodellezésben használt *Constructive Solid Geometry* (**CSG**) technikán alapul. Ezen eljárás során a modellt primitívekből, például gömbökből, téglatestekből és egyéb matematikailag jól megadható testekből építjük fel, halmazműveletek segítségével. Ezen műveletek megadnak egy gráfot, melyet **CSG gráfnak** szokás nevezni.

A program a felhasználó szempontjából két részből áll: egy gráfszerkesztőből, valamint egy a kapott modellt megjelenítő ablakból. A kapott felület kirajzolása a gráfrepresentációból generált előjeles távolságfüggvény (**SDF**<sup>1</sup>) segítségével történik.

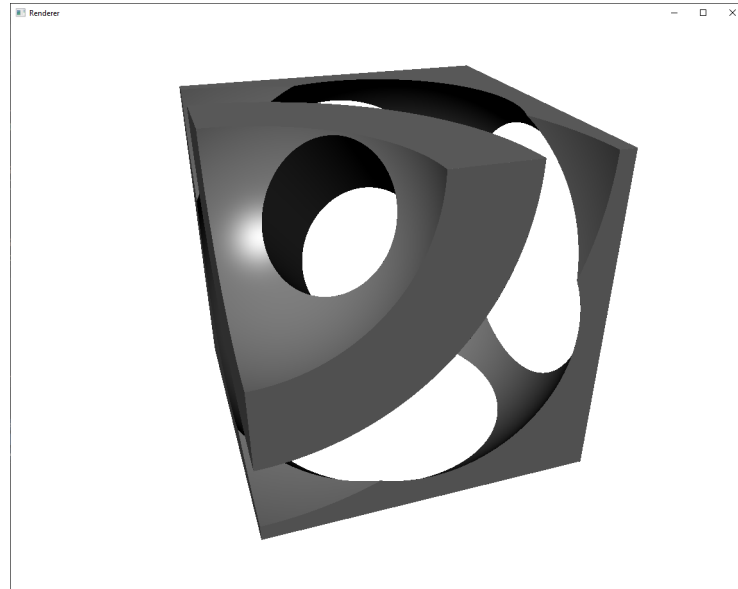
Mindemellett a dolgozatban a **távolságfüggvény deriváltjainak automatikus differenciálással történő meghatározásával** is foglalkozom. A program képes a deriváltak kiszámítására, valamint néhány belőlük meghatározott felületi tulajdonság szemléltetésére is. Többek között a deriváltak megadják a normálvektorokat a felület pontjaiban, melyek szükségesek a megjelenítés során alkalmazott térhatású árnyaláshoz. Az így kapott normálvektorok pontosabbak mintha azokat differenciahányadosos approximációval határoznánk meg, a program segítségével a két módszerből kapott árnyalás összehasonlítható.

---

<sup>1</sup>Signed Distance Function



(a) Gráfszerkesztő



(b) Megjelenítő

1. ábra. Kép a kész programról.

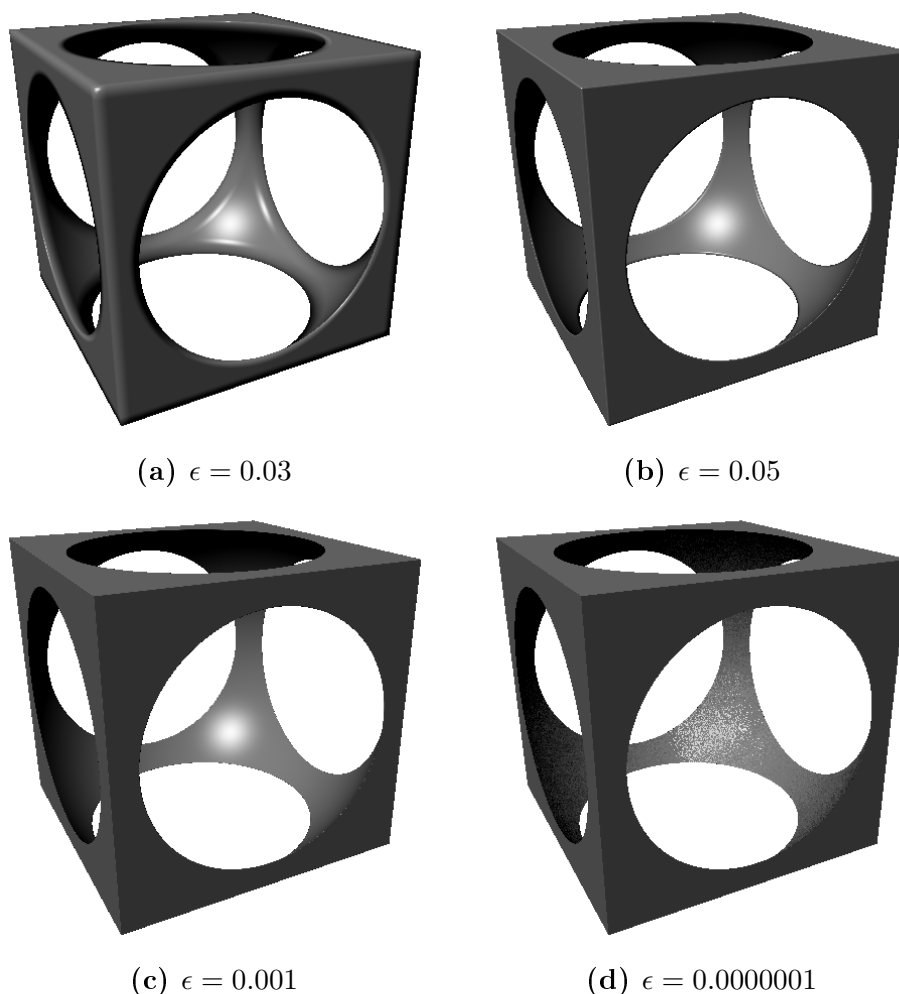
## Motiváció az automatikus differenciáláshoz

Az árnyaláshoz szükséges normálvektorok kiszámításának leginkább kézenfekvő módja a távolságfüggvényből történő differenciahánadosos approximáció. Ez az eljárás nem ad mindig jó megjelenítést, ugyanis amennyiben a távolságfüggvény megadása esetszétválasztásokat tartalmaz<sup>2</sup>, vagy egyéb okokból a felület éles élek-

<sup>2</sup>Az esetszétválasztások jellemzően elrontják a távolságfüggvény deriváltjának folytonosságát, az approximációs probléma innen származik. Jó példa erre például egy kocka távolságfüggvénye.



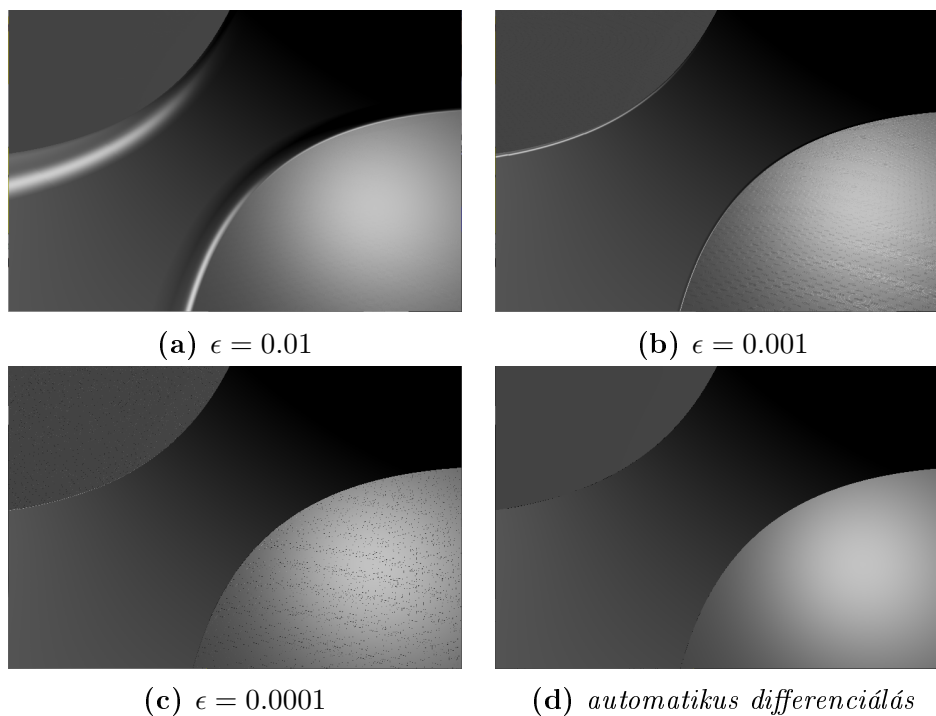
kel rendelkezik, az ott kapott normálvektorok hibája jól láthatóvá válik (2a ábra). Ezen bizonyos mértékben tudunk segíteni azzal, ha a differenciahányadosok számításánál minél kisebb  $\epsilon$  távolságra mozdulunk el a sphere tracing eredményeként kapott közelített metszésponttól (2b ábra). Túl kicsi  $\epsilon$  esetén viszont a lebegőpontos számábrázolásból eredő hibák nőnek meg, különösen a felület simább részein, ugyanis ilyenkor a szomszédos pontokban kapott függvényértékek nagyon közeliek, így különbségük hibája megnő. Valamint a numerikus hiba amiatt is fellép, hogy a differenciahányadosokban kis  $\epsilon$ -al osztunk.



**2. ábra.** Árnyalási hibák differenciahányadosokkal approximált normálvektorokból. (2 élhosszúságú kocka és 0.65 sugarú gömb különbsége) Az (a) ábrán az élek mentén látványos árnyalási hibák vannak. A (b) ábrán kisebb  $\epsilon$  mellett ezek a hibák kevésbé látszódnak, míg (c) esetén egyáltalán nem. Ha  $\epsilon$ -t túlságosan lecsökkentjük, akkor egy más jellegű hiba lép fel, amely szemcsés árnyaláshoz vezet (d ábra).

Egyes modellek esetében  $\epsilon$  megfelelő megválasztásával kiküszöbölhetőek a szemmel észrevehető hibák. Ugyanis bizonyos határokig  $\epsilon$  csökkentése javít a pontosságon.

Azonban ha a távolságfüggvény kiszámítása során nagy konstansok is szerepelnek<sup>3</sup>, akkor az így megjelenő nagy számok miatt romlik a pontosság, azaz a 2d ábrán látható szemcsés zaj az árnyalásban már nagyobb  $\epsilon$  mellett is megjelenik. Tehát ha egy szintér egyszerre tartalmaz nagy objektumokat és apró részleteket, akkor pusztán  $\epsilon$  beállításával nem kerülhető el minden normálvektor pontatlanságából következő árnyalási hiba. Egy ezt szemléltető példa látható az 3 ábrán.

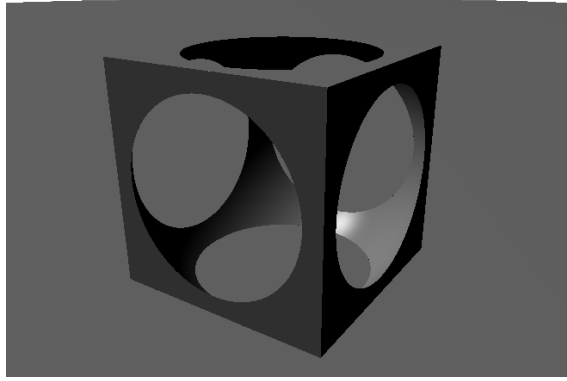


**3. ábra.** Példa olyan szintérre, ahol tetszőleges  $\epsilon$  esetén láthatóak az approximációs hibák. Kiemelném a **(b)** ábrát, ahol mind a túl kicsi  $\epsilon$ -ből fakadó, mind a túl nagy  $\epsilon$  esetén jellemző hiba megjelenik. (2 ábrán látható lyukas kocka belseje, egy 1000 egység sugarú gömbön)

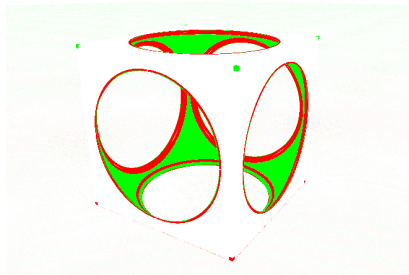
A differenciahányadossal történő approximáció numerikus hibái méginkább látványossá válnak, ha a megjelenítés során az ily módon kapott<sup>4</sup> Hesse-mátrixot is felhasználjuk. Ennek szemléltetésére vizsgáljuk meg az ebből számított Gauss- és középgörbületeket.

<sup>3</sup>Például, ha a megjelenítendő objektum nagyméretű.

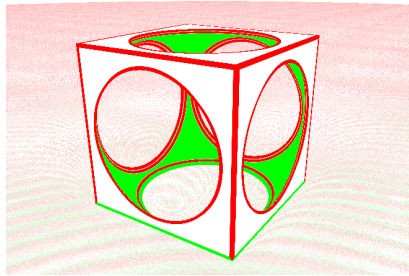
<sup>4</sup>A differenciahányadossal approximált gradiens vektorokon vett differenciahányadosokkal kapott Hesse-mátrix.



(a) Phong [1] modellel árnyalva.



(b) Gauss-gömbölet



(c) Középgömbölet

**4. ábra.** Aprroximációból eredő hibák görbület számításakor. Lyukas egységkocka egy 1000 sugarú gömb felszínén,  $\epsilon = 0.01$  melletti differenciahányadosokkal számolt approximációval (**bal**), valamint automatikus differenciálással (**jobb**). Hibátlan számítás esetén egyedül a lyukas kocka belső görbe felületei látszódnának zöldes árnyalással, ugyanakkor még az automatikus differenciálással kapott képeken is látható néhány elszórt színes pixel. A nagy gömb görbülete annyira kis mértékű, hogy a képeken fehéreknek látszik.

## II. rész

# Felhasználói dokumentáció

---

## 1. A feladat rövid ismertetése

Az alkalmazás segítségével lehetséges háromdimenziós modellek összeállítása és rajtuk egyes felületi tulajdonságok vizualizációja. Ezen tulajdonságok megjelenítéséhez szükség van a távolságfüggvény deriváltjainak meghatározására, amelyhez a program két különböző kiszámítási módot képes alkalmazni. Ez a két módszer nem más, mint a numerikus approximáció differenciahányadosokkal és az automatikus differenciálás. A felhasználónak lehetősége van a két módszerből kapott képek hibáinak összevetésére.

## 2. Hardveres és szoftveres követelmények

### Hardver

- 64 bites, x86 utasításkészletet támogató CPU
- OpenGL 4.6 képes GPU (A 2017 óta megjelent GPU-k többsége megfelelő)
- 16 GB RAM<sup>5</sup>
- billentyűzet, egér

### Szoftver

- Operációs rendszer: Windows 10
- OpenGL 4.6-ot implementáló GPU driver

## 3. Telepítés és futtatás

A szoftver telepítése nem igényel különleges műveleteket. A mellékelt tömörített állomány kicsomagolását követően az "Editor.exe" futtatásával használható.

## 4. Az alkalmazás használata

### 4.1. Szerkesztő ablak

Az elkészült program segítségével a felhasználó egy CSG gráf szerkesztésével állíthat össze különböző háromdimenziós modelleket. Magát a gráfot egy külön ablak-

---

<sup>5</sup>Maga a szerkesztőprogram jellemzően nem használ 100-200Mb-nál többet, viszont összetettebb modellek esetén bekapcsolt magasabb deriváltak mellett komplex shaderek keletkeznek, melyek lefordításakor GPU driver a tapasztalat szerint átmenetileg akár 15GB memóriát is lefoglalt.

---

ban adhatja meg. A gráf csúcsai két csoportra oszthatóak. Vannak primitív csúcsok, melyek bemenettel nem, csak kimenettel rendelkeznek. A másik csoportot a műveleti csúcsok adják. Ezek rendelkeznek több bemenettel is, valamint egy kimenettel, amely a művelet eredményét adja.

Az új csúcs felvételét lehetővé tévő menü a szerkesztő ablak üres felületére jobb egérgombbal kattintva hozható elő. Itt lehetőség van a két csúcstípus közül választani. A létrehozott csúcs megjelenik a gráfszerkesztőben, annak szerkeszthető tulajdonságaival együtt.

Az, hogy egy primitív vagy műveleti csúcs melyik konkrét primitívet vagy műveletet reprezentálja, a *Type* (primitíveknél) és a *Mode* (műveleteknél) mező segítségével választható ki. Ez alatt szerkeszthetők az adott primitívre vagy műveletre jellemző paraméterek. Minden csúcs rendelkezik egy transzformációs taggal is, melynek szerkesztéséhez be kell kapcsolni annak megjelenítését a csúcson lévő *Show transform* kapcsoló segítségével. Itt lehetőség van a csúcs kimenetén egyszerűbb transzformációk végrehajtására, mint például az eredményként kapott alakzat eltolására, forgatására, uniform skálázására vagy a későbbiekben bemutatott *offset* műveletre.

Összetettebb modelleket a primitív csúcsok műveleti csúcsok segítségével való kombinálásával kaphatóak. Ehhez mindössze annyit kell tenni, hogy a bal egérgombot lenyomva tartva összekötjük az egyik csúcs kimenetét a másik csúcs egyik bemenetével. Az összekötés során nem alakulhat ki körkörös hivatkozás, az ilyen élek behúzását a program nem engedélyezi.

## 4.2. Menü

A gráfszerkesztő ablak felső részében található menü a következő lehetőségeket biztosítja:

- *File* menü – fájlműveletek
  - Új, üres szintér létrehozása (*New*)
  - Szintér betöltése fájlból (*Open*)
  - Aktuális szintér mentése és mentése másként (*Save*, *Save as*)
- *View* menü – gráfszerkesztő nézetét manipuláló műveletek
  - Ugrás a kijelölt csúcsokra (*Jump to selection*)

- 
- Nézet beállítása úgy, hogy a teljes gráf látszódjon (*Fit entire graph*)
  - A tengelyek mutatásának ki és bekapcsolása (*Axes*)
  - A sarokban a koordináta rendszer kamerához képesti irányait mutató ábra ki és bekapcsolása (*Directions*)
  - *Derivatives* menü – A távolságfüggvény automatikus differenciálásával kapcsolatos beállítások. Részletek későbbi fejezetben.
    - Az automatikus differenciálás be és kikapcsolását szolgáló kapcsoló. (*Enable*)
    - Hanyadrendű deriváltak legyenek automatikus differenciálással meghatározva. (*Order*)
  - *Visualization* – A háromdimenziós megjelenítéssel kapcsolatos beállítások.
    - Valósídejű megjelenítés be és kikapcsolása. Ha ki van kapcsolva akkor csak akkor történik újrarajzolás ha valamilyen változás volt a színtérben. (*Realtime*)
    - A felület színezési módjának kiválasztását lehetővé tévő választó gombok.
      - \* Színezés fénnnyel való árnyalással. (*Shaded*)
      - \* Színezés a felületi normálvektoroknak megfelelően. XYZ koordináták  $\sim$  RGB színtkomponensek. (*Normals*)
      - \* Színezés a felület Gauss-görbülete szerint. (*Gaussian curvature*)
      - \* Színezés a felület Középgörbülete szerint. (*Mean curvature*)
      - \* A megjelenítést végző *sphere tracing* algoritmus lépésszámának vizualizálása. (*Steps*)
      - \* Az approximált és automatikus differenciálással kapott normálvektorok eltérésének vizualizációja. Csak bekapcsolt első deriváltak esetén elérhető. (*Normal error*)
    - Váltás a numerikusan approximált és az automatikus deriválással kapott deriváltak között. Ez a gomb csak akkor jelenik meg, ha a generált shader rendelkezik a kiválasztott megjelenítési módhoz szükséges deriváltak automatikus deriválásához szükséges kóddal. Ez a *Derivatives* menüben állítható be. Normálvektorokhoz legalább első, görbületekhez legalább második deriváltak szükségesek. (*Use automatic differentiation*)

- *Auto compile* kapcsoló – Bekapcsolja, hogy a CSG gráf vagy a deriválási beállítások módosulása esetén automatikusan legenerálódjon és lefordítódjon a megjelenítéshez szükséges shader. Minderre azért van szükség mert bonyolult gráfok és magasabb deriváltak esetén a shader fordítás lassúvá válik. Ha ki van kapcsolva, akkor megjelenik egy *Compile* gomb is mellette, amely segítségével manuálisan indítható el a shader fordítás.

### 4.3. Megjelenítő ablak

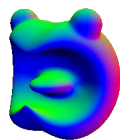
A megjelenítő ablakban háromdimenzióban megtekinthetőek a gráf egyes részei. Azt, hogy melyik csúc kimenete kerüljön kirajzolásra a csúcsok jobb felső sarkában lévő pötty jelzi, amely az éppen kiválasztott csúc esetén sárga. A kiválasztáshoz elég a megjeleníteni kívánt csúc pöttyére kattintani.

A háromdimenziós színtérben a **WASD** gombok segítségével lehetséges a mozgás. Nézelődni a bal egérgombot lenyomva tartva az egeret mozgatva lehet. A fénnnyel történő árnyalás esetén a **space** gomb lenyomásával állítható a fény iránya a kamera aktuális nézési irányára.

## 5. Vizualizációs módok



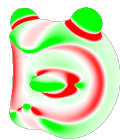
(a) *Fénnnyel árnyalt*



(b) *Normálvektorok*



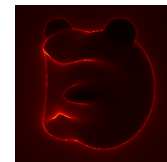
(c) *Approximáció és auto-diff különbsége*



(d) *Gauss-görbület*



(e) *Középgörbület*



(f) *Lépésszám*

5. ábra. *Vizualizációs módok*

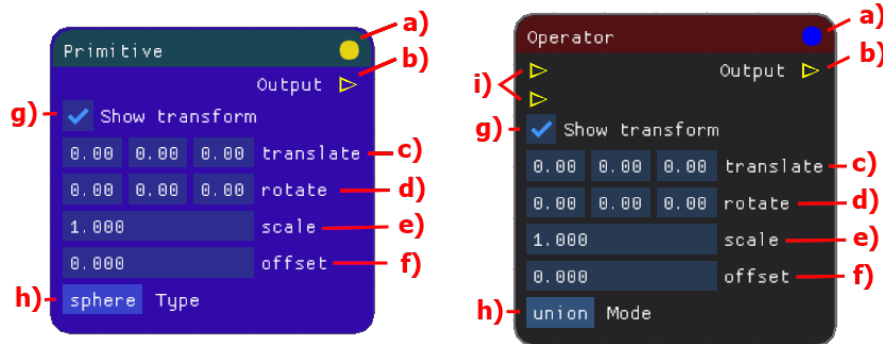
## 6. Primitívek és műveletek

Ebben az alfejezetben az alkalmazás által támogatott primitíveket és tulajdonságait mutatom be.



## 6.1. Közös tulajdonságok

Minden primitív és művelet gráfbeli csúcsa rendelkezik egy kimenettel, amely primitívek esetén önmagukat, műveletek esetén az eredményt jelképezi.



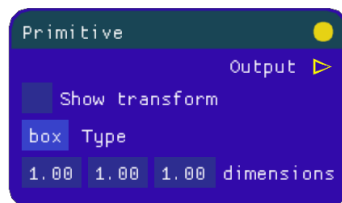
6. ábra. Primitív és operátor csúcs a szerkesztőben

- a) **Gyökér jelölő.** Jelöl, hogy mely csúcs a megjelenített részgráf gyökere. Az aktuálisan kijelölt gyökércsúcs esetén sárga, egyébként kék színű. Egy éppen nem gyökércsúcs jelölőjére kattintva lehet azt gyökérré tenni. A gyökércsúcs háttérszíne élénkebb kék, ezzel elősegítve a kiemelését.
- b) **Kimenet.** Csúcsok összekötésekor innen kell indítani a bal egérgomb lenyomása közbeni húzó mozgulatot.
- c) **Eltolás vektor.** Megadja, hogy az adott csúcs kimenetén milyen eltolást alkalmazzunk.
- d) **Forgatás.** Megadja, hogy rendre az  $x, y$  és  $z$  tengelyek mentén hány fokok forgatást alkalmazzunk a kimeneten.
- e) **Skálázás.** Megadja, hogy a csúcs eredményének méretét hányszorosára növeljük vagy csökkentjük. A skálázás az origótól történik.
- f) **Offset.** Hozzáadott érték a kimenet távolságfüggvényéhez. Ez jellemzően (de nem mindig) lekerekítő hatást eredményez. Amikor a felhasznált távolságfüggvény csak alsó becslés (például ellipszoid), akkor negatív értékek esetén furcsán működhet.
- g) **Show transform.** Az eltolás, forgatás és offset műveletek elrejtését szabályozó kapcsoló.
- h) **A csúcs altípusa.** Megadja, hogy a primitívek vagy műveletek közül melyiket jelentse a csúcs.
- i) **Bemenetek.** Az ide bekötött csúcsok által megadott alakzatokon lesz a művelet elvégezve. Minden műveleti csúcs esetén legalább kettő van, egyes esetekben több is lehet.

---

## 6.2. Primitívek

Egyes primitívek rendelkeznek külön szerkeszthető attribútumokkal, melyek a gráfszerkesztőben a primitív csúcsán állíthatóak be, a primitív típusa alatt.



7. ábra. Téglatest primitív szerkeszthető attribútumai.

Gömb – *Sphere*

Téglatest – *Box*

Attribútumai: a három kiterjedése.

Henger – *Cylinder*

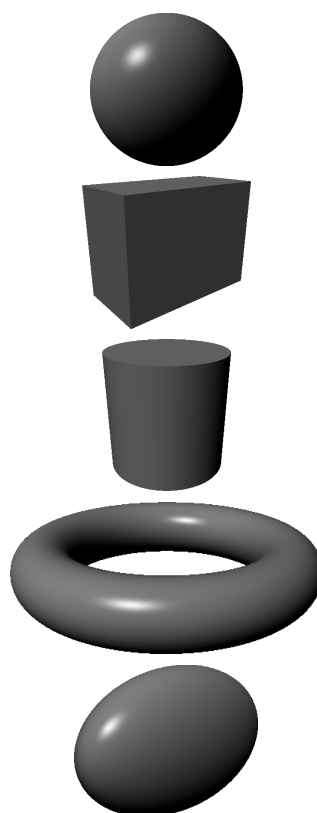
Attribútumai: sugár és magasság.

Tórusz – *Torus*

Attribútumai: a két sugara.

Ellipszoid – *Ellipsoid*

Attribútumai: a három féltengelye.



1. táblázat. Támogatott primitívek és szerkeszthető tulajdonságaik.

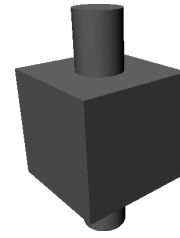
## 6.3. Műveletek

A program által támogatott halmazműveletek két csoportba oszthatóak. Egyrészt használhatóak az unió, metszet és különbség műveletek. Ezen kívül mindhárom művelet rendelkezik egy paraméterezhető „simított” megfelelővel is. A támogatott műveletek és eredményeik az alábbi ábrán láthatóak.

---

**Unió – *Union***

Tetszőleges számú bemenetet elfogad.



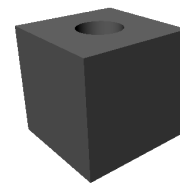
**Metszet – *Intersect***

Tetszőleges számú bemenetet elfogad.



**Különbség – *Subtract***

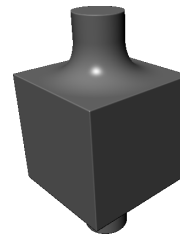
Tetszőleges számú bemenetet elfogad.



**Simított unió – *Smooth union***

Pontosan két bemenetet fogad el.

A simítás mértéke paraméterezhető. ( $k$ )



**Simított metszet – *Smooth intersect***

Pontosan két bemenetet fogad el.

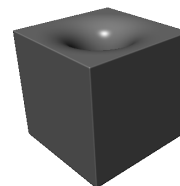
A simítás mértéke paraméterezhető. ( $k$ )



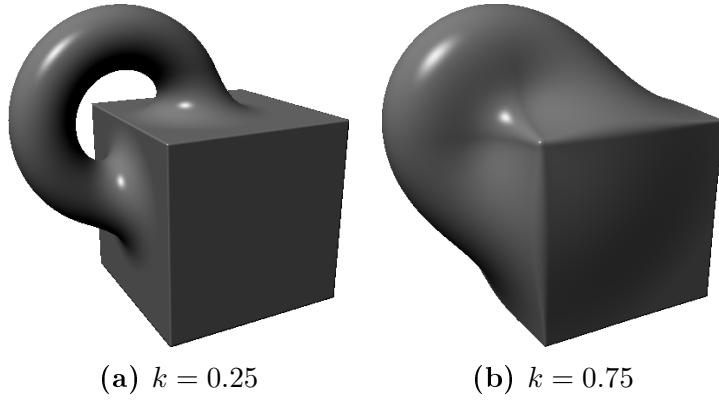
**Simított különbség – *Smooth subtract***

Pontosan két bemenetet fogad el.

A simítás mértéke paraméterezhető. ( $k$ )



**2. táblázat.** *Támogatott műveletek és szerkeszthető tulajdonságaik.*



**8. ábra.** *Simított unió különböző  $k$  paraméterek mellett.*

### III. rész

## Fejlesztői dokumentáció

# 1. fejezet

## Fejlesztői környezet

A program fejlesztése során **Visual Studio 2022** fejlesztői környezetet használtam, **Windows 10** operációs rendszeren. Szükség van a Visual Studio C++ komponenseire is, melyeknek tartalmazniuk kell egy **C++17** képes MSVC fordítót.

### 1.1. Felhasznált könyvtárak

- Dragonfly (MIT licenz) – Grafikus megjelenítéshez. A szükséges fájlok részben a *DragonflyPack*-ben, részben a projekt *Dragonfly* almappájában. A DragonFly-on keresztül közvetetten az alábbi könyvtárak is felhasználásra kerülnek:
  - Simple DirectMedia Layer – Ablakkezeléshez.
  - OpenGL Extension Wrangler Library (GLEW) – OpenGL-hez.
  - OpenGL Mathematics – matematikai vektor és mátrixműveletekhez műveletekhez.
- Nlohmann Json (MIT licenz) – Json szerializáláshoz és deszerializáláshoz. Egy headerből áll, ez a *Libraries/include/json.hpp*.
- Native file dialog (Szabad felhasználású licenz) – A mentés és betöltés során használt dialógus ablakokhoz. Fájlok a projektmappán belül a *nativefiledialog* mappában.

## 1.2. Fordítás

A projekt lefordításához szükség van a **DragonflyPack** megfelelő konfigurációjára. Ehhez futtassuk *DragonflyPack* mappa mellett található **mount\_t.bat** fájlt. Ez létrehoz egy virtuális **T:** meghajtót, melynek gyökere az a mappa lesz, ahol a script is található. Erre azért van szükség, mert a projekt oly módon van felkonfigurálva, hogy a DragonFly-hoz szükséges könyvtárakat *T:/DragonflyPack/* elérési úton keresi. A T: meghajtó felcsatolására minden egyes rendszerindítás után szükség van. Ha szeretnénk a meghajtót lecsatolni, azt az **unmount\_t.bat** script-tel tehetjük meg.

Előfordulhat, hogy a fordítás során C++ template hibákat kapunk, melyek a Dragonfly könyvtárból származnak. Ennek pontos oka nem ismert, lehet egy MSVC bug, vagy egy Dragonfly hiba is. A fejlesztés során a használt **MSVC Toolset** verziójának **14.30.30705**-re történő csökkentése megoldotta ezt a problémát.

## 1.3. Futtatási környezet

Fordítsuk le a programot **Release** módban. A kapott futtatható állomány az *x64/Release/CSGEditor.exe* lesz, azonban idegen környezetben ennek futtatásához szükség van a *DragonflyPack/LibBin*-en belül található *glew32.dll* és *SDL2.dll* fájlokra is. Ezen kívül szükség van a *CSGEditor/Shaders/* mappában található *trace.vert*, *trace.frag*, *gizmo.vert*, *gizmo.frag*, *number.frag* és *primitives.frag* shaderfájlokra is. A futtatható állomány közzétételekor vele egy mappába helyezzük a két említett dll-t és a Shaders mappát, benne a felsorolt shaderfájlokkal.

## 2. fejezet

# Specifikáció

### 2.1. Funkcionális követelmények

A programtól egy egyszerű gráfszerkesztő funkcionalitását, valamint az ebben összeállított modell háromdimenziós megjelenítésének képességét várjuk el. Ennek megfelelően a funkcionális követelmények a reprezentáció szerkesztését, mentését és betöltését, valamint a megjelenítést fedik le.

#### 2.1.1. Felhasználói történetek

<i>Színtér gráf mentése és betöltése</i>		
A felhasználónak lehetősége van a színtér reprezentálására használt gráfot - és ezáltal a színteret - fájlba menteni, illetve onnan betölteni.		
AS A		Felhasználó
I WANT TO		Folytatni egy megkezdett gráfot a program bezárása majd elindítása után.
1	GIVEN	Mentés menüpontot választjuk.
	WHEN	Kitallózzuk a mentendő fájl helyét és megadjuk annak nevét.
	THEN	A program elmenti a gráfot (erdőt) a megadott helyre.
2	GIVEN	Betöltés menüpontot választjuk.
	WHEN	Kitallózzuk a betöltendő fájl helyét és megadjuk annak nevét.



	THEN	A program betölti a fájlba elmentett a gráfot (erdőt), illetve jelez ha rossz fájlt választottunk.
<i>Szintér gráf szerkesztése</i>		
A szintért reprezentáló gráf szerkeszthető, egy ezt a célt szolgáló gráfszerkesztő ablakban. Itt lehetőség van:		
<ul style="list-style-type: none"> <li>• Új primitív vagy művelet csúcsok felvételére.</li> <li>• Csúcsok törlésére.</li> <li>• Csúcsok összekapcsolására.</li> <li>• Csúcsok szerkesztésére.</li> <li>• Másolásra és beillesztésre.</li> <li>• A megjelenítendő gyökércsúcs kijelölésére.</li> <li>• A megjelenített csúcsok elrendezésének módosítása (csúcsok mozgatása).</li> </ul>		
AS A		Felhasználó
I WANT TO		A szintér gráfot szerkeszteni.
1	GIVEN	A gráfszerkesztő egy üres pontjára a jobb egérgombbal kattintottunk.
	WHEN	Kiválasztottuk, hogy milyen csúcsot szeretnénk beszúrni (primitív vagy művelet).
	THEN	A kiválasztott típusnak megfelelő csúcs beszúrára kerül arra pontra ahova eredetileg a jobb egérgombbal kattintottunk.
2	GIVEN	Egy vagy több csúcs ki van jelölve.
	WHEN	Megnyomjuk a <i>Delete</i> gombot.
	THEN	Törlődnek a kijelölt csúcsok és azok az élek melyeknek legalább egyik végén kijelölt csúcs van.
3	GIVEN	Egy él van kijelölve.
	WHEN	Megnyomjuk a <i>Delete</i> gombot.
	THEN	Az él törlődik.
4	GIVEN	Vonalat húzunk egy csúcs egyik pinjéből <sup>1</sup> , a bal egérgombot lenyomva tartva.
	WHEN	A vonal húzását egy másik csúcs pinje fölött fejezzük be.

<sup>1</sup>csatlakozási pont

	THEN	Ha a behúzott él szabályos, a két csúc között létrejön a kapcsolat, ha nem, akkor azt piros színnel jelzi a program még a vonal húzása közben. Amennyiben a vonalat egy már foglalt bemeneti pinbe húztuk, akkor a korábban oda csatlakozó él megszűnik.
5	GIVEN	Egy csúc ki van jelölve a szerkesztőben.
	WHEN	A csúcson megjelenő tulajdonságok egyikének értékét megváltoztatjuk.
	THEN	Az érték megváltozik és a reprezentáció frissítésre kerül, a shaderprogramok újragenerálódnak.
6	GIVEN	Automata shader generálás be van kapcsolva.
	WHEN	A gráfrepresentációt tetszőleges módon módosítjuk, vagy egy az automatikus differenciálást érintő beállításon változtatunk.
	THEN	A shaderprogramok újragenerálódnak és fordításra kerülnek. Ezt követően a megjelenítő már az új állapotnak megfelelő modellt rajzolja ki.
7	GIVEN	Egy vagy több csúc ki van jelölve.
	WHEN	<i>Ctrl+C</i> billentyűkombinációval másolást kezdeményezünk.
	THEN	A kijelölt csúcsok, valamint az ezek között futó élek a vágólapra kerülnek. <sup>2</sup>
8	GIVEN	A vágólapon vannak lemásolt csúcsok, esetleg élek is.
	WHEN	<i>Ctrl+V</i> billentyűkombinációval beillesztést kezdeményezünk.
	THEN	A lemásolt csúcsok és élek beillesztésre kerülnek, a gráfszerkesztő jelenlegi nézetének megfelelő helyre. <sup>3</sup>
9	GIVEN	Van olyan megjeleníthető csúc a gráfszerkesztőben, amely nem gyökércsúc <sup>4</sup> .
	WHEN	Egy ilyen csúc jobb felső sarkában található színes gombra kattintunk.

<sup>2</sup>A megvalósítás során igazából egy hivatkozás kerül a vágólapra, amely szerint a beillesztéskor fog elkészülni a másolat.

<sup>3</sup>A lemásoláskori nézetbeli relatív pozíciójukat megőrizve.

<sup>4</sup>A gyökércsúc eltérő háttérszínnel van kiemelve.

	THEN	A csúcs gyökércsúccsá válik és az új gyökérnek megfelelő részgráf kerül megjelenítésre a térbeli nézetben. Az előző gyökércsúcs megszűnik annak lenni, hiszen csak pontosan egy (megjelenítendő) gyökércsúcs létezhet.
10	GIVEN	Vannak kijelölt csúcsok a gráfszerkesztőben.
	WHEN	A csúcsok egyikét a bal egérgombot lenyomva tartva egy húzó mozdulattal mozgatjuk.
	THEN	A kijelölt csúcsok gráfszerkesztőbeli pozíciója a mozgatásnak megfelelően változik.

*Automatikus differenciálást érintő beállítások módosítása*

A programban lehetőség van a távolságfüggvény deriváltjainak approximálás helyett automatikus differenciálással történő meghatározására. Ez két beállítást jelent:

- Az automatikus differenciálás be/ki kapcsolása.
- Az automatikus differenciálással meghatározott legmagasabb rendű derivált rendjének megadása.

1	GIVEN	A gráf nem üres.
	WHEN	A fenti két beállítás egyikét módosítjuk.
	THEN	A beállítások módosulnak. A következő shader generálást és fordítást követően az új deriváltak elérhetővé válnak a megjelenítés során, vagy megszűnnek elérhetőek lenni.

*Felület vizualizációs beállítások módosítása*

A felhasználónak lehetősége van több felületi tulajdonság megjelenítése közül választani. Továbbá lehetőség van a számítások approximálással vagy automatikus differenciálással történő elvégzése közül is választani.

1	GIVEN	A gráf nem üres.
	WHEN	Vizualizációs módot váltunk.
	THEN	A megjelenítőben a modell az új beállításnak megfelelően kerül színezésre. Ha elérhetőek autodiff-el számolt deriváltak, akkor lehetőség van választani, hogy a két módszer közül melyiket alkalmazzuk.
	GIVEN	Valósídejű megjelenítés ki van kapcsolva.

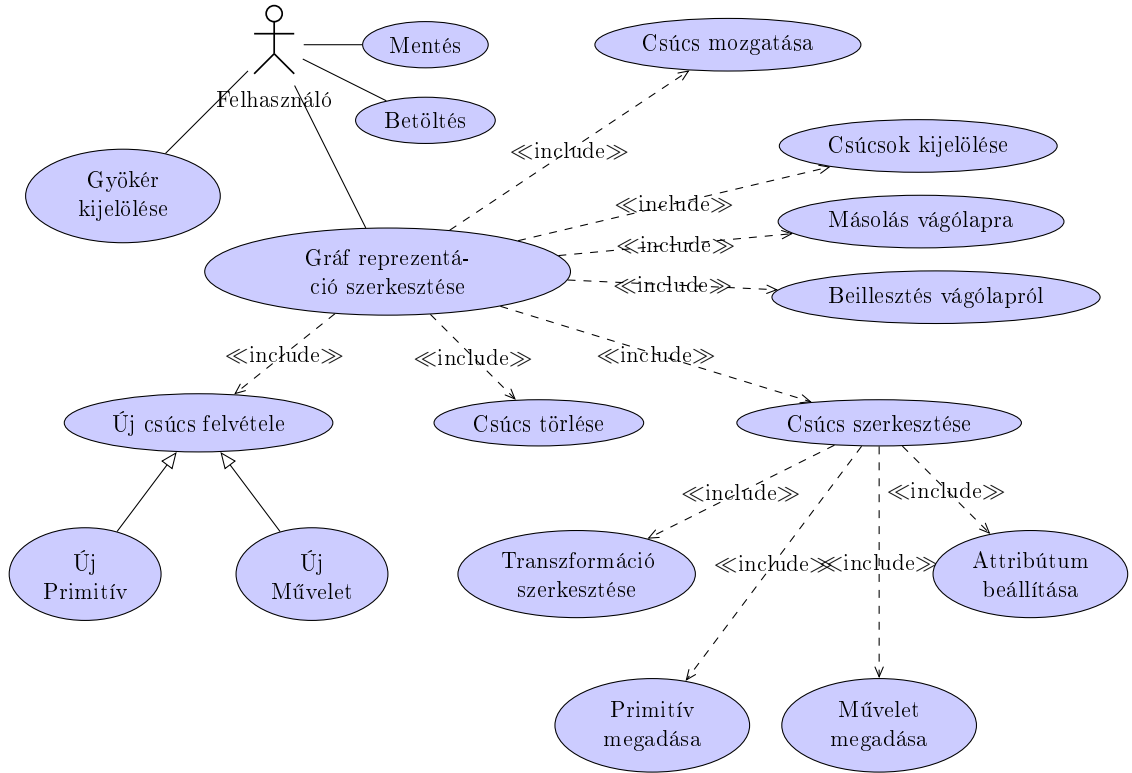
	WHEN	A kamera elmozdul vagy elfordul a színtérben, vagy egy a megjelenített eredményre hatással lévő paraméter módosul.
	THEN	A megjelenített kép újrarajzolódik.
3	GIVEN	Valósídejű megjelenítés ki van kapcsolva.
	WHEN	Nem történik a megjelenítést befolyásoló változás.
	THEN	A megjelenített kép nem rajzolódik újra.
<i>Navigáció a térbeli nézetben</i>		
A programnak része egy térbeli nézet, amelyben megtekinthető a gráfszerkesztőben elkészített modell. A térbeli nézetben lehetőség van a modell tetszőleges pozícióból való megtekintésére, valamint egyszerűbb szerkesztési műveletek (transzformációk) végrehajtására.		
1	GIVEN	A térbeli nézet az aktív ablak.
	WHEN	A bal egérgombot lenyomva tartva húzzuk az egeret.
	THEN	Az egérmozgás irányának megfelelően a kamera <sup>5</sup> elfordul.
2	GIVEN	A térbeli nézet az aktív ablak.
	WHEN	A <i>WASD</i> billentyűk egyikét lenyomva tartjuk.
	THEN	A billentyűnek megfelelő kamerához képest relatív irányba <sup>6</sup> mozog a kamera.

2.1. táblázat. *Funkcionális követelmények felhasználói történetekkel megadva.*

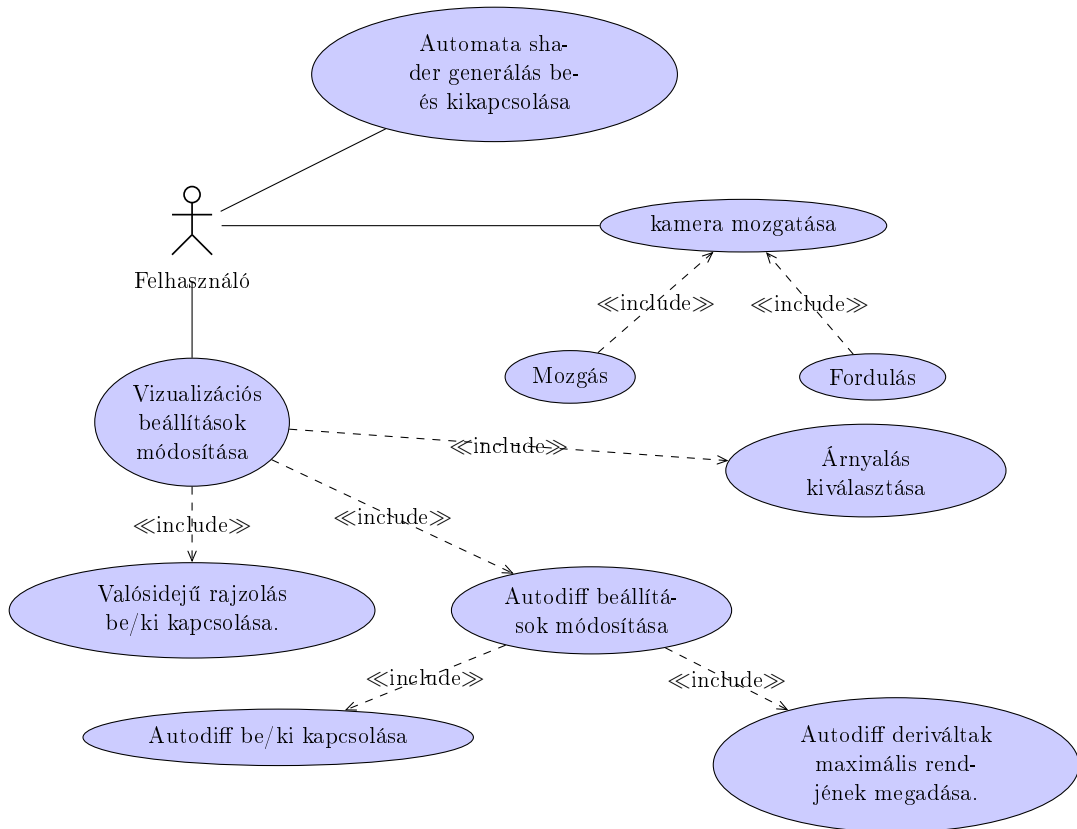
### 2.1.2. Felhasználói eset diagramm

<sup>5</sup>A térbeli nézetben annak a pozíciónak, elfordulásnak és látószögnek az együttese, ahonnan a modellt nézzük.

<sup>6</sup>*W* - előre, *A* - balra, *S* - hátra, *D* - jobbra



2.1. ábra. Gráfszerkesztéssel kapcsolatos felhasználói esetek.



2.2. ábra. Megjelenítéssel kapcsolatos felhasználói esetek.

## 3. fejezet

# Elméleti háttér

Ebben a fejezetben a ismertetem felület megjelenítéséhez felhasznált sphere tracing algoritmus alapjait, valamint a deriváltak pontosabb meghatározásához felhasznált automatikus differenciálás módszerének matematikai háttérét. Továbbá bevezetünk néhány jelölést az itt tárgyalt fogalmakra, melyeket a későbbiekben is használni fogok.

### 3.1. Fogalmak

#### 1. Definíció. Implicit felület

Legyen  $f : \mathbb{R}^3 \rightarrow \mathbb{R}$  a felületet megadó függvény.

Ekkor a tér azon  $\vec{x} \in \mathbb{R}^3$  pontjai, ahol  $f(\vec{x}) = 0$  alkotják az  $f$  által megadott implicit felületet.

#### 2. Definíció. Euklideszi távolság

Legyenek  $\vec{x}, \vec{y} \in \mathbb{R}^3$  pontok a háromdimenziós euklideszi térben.

Ekkor  $\vec{x} = [x_1, x_2, x_3]^T$  és  $\vec{y} = [y_1, y_2, y_3]^T$  euklideszi távolsága:

$$d(\vec{x}, \vec{y}) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + (x_3 - y_3)^2}$$

#### 3. Definíció. Pont és halmaz távolsága

Legyen  $\vec{x} \in \mathbb{R}^3$  és  $\Omega \subset \mathbb{R}^3$ .

Ekkor  $\vec{x}$  és  $\Omega$  távolsága:

$$d(\vec{x}, \Omega) = \inf_{\vec{a} \in \Omega} d(\vec{a}, \vec{x})$$

#### 4. Definíció. Előjeles távolságfüggvény

Legyen  $\Omega \subset \mathbb{R}^3$  nyílt halmaz. Ekkor az  $\Omega$ -hoz tartozó  $f$  előjeles távolságfüggvény:

$$f : \mathbb{R}^3 \rightarrow \mathbb{R}, f(\vec{x}) := \begin{cases} d(\vec{x}, \partial\Omega) & \vec{x} \notin \Omega \\ -d(\vec{x}, \partial\Omega) & \vec{x} \in \Omega \end{cases}$$

ahol  $\partial\Omega = \overline{\Omega} \setminus \Omega$  a halmaz határa.

### 3.2. Sphere tracing

Hart 1996-ban jelentette meg implicit felületek valósidejű megjelenítésére kidolgozott eljárását, **sphere tracing** néven [2]. A módszer arra épül, hogy ha megadjuk egy  $\Omega \subset \mathbb{R}^3$  nyílt halmaz  $f$  előjeles távolságfüggvényét a 4. definícióban leírtak szerint, akkor az így kapott  $f$  függvény által meghatározott implicit felület nem más, mint  $\Omega$  határfelülete ( $\partial\Omega$ ).

A sugárkövetés elvéhez hasonlóan, a virtuális kamera minden képpontjából a nézet irányának megfelelően<sup>1</sup> sugarakat indítunk. Ellentétben viszont a háromszöghálókön végzett sugárkövető eljárásokkal, itt nem pontos metszéspontokat számítunk ki, hanem a sugár mentén haladva közelítünk a felülethez.

Az algoritmus bemenete az előjeles távolságfüggvény (**SDF**<sup>2</sup>), a kiindulási pozíció és a sugár iránya. Legyen egy változónk, amellyel a sugár és a felület metszéspontját közelítjük. Kezdetben ezt inicializáljuk a kiindulási pozícióval. Ezen kívül szükségünk van egy változóra, amely a felülettől való aktuális távolságunkat fogja tárolni. Inicializáláskor ezt állítsuk az SDF kiindulási pozícióban felvett függvényértékére.

Ezt követően az alábbi lépéseket ismételjük egészen addig, amíg el nem érünk egy maximális lépésszámot, egy maximális távolságot, vagy elég közel kerülünk a felülethez:

1. Mivel ismerjük a felület  $d$  távolságát az aktuális pozíciótól, ezért egy  $d$  sugarú gömbben biztosan nem található a felületnek egy pontja sem. Ez azt jelenti, hogy a sugár irányába  $d$  távolságot haladva vagy pontosan a felület egy pontjára

<sup>1</sup>Azaz a fókuszpont és a képpont közeli vágósíkon lévő pozícióját összekötő szakasz meghosszabbításának irányába.

<sup>2</sup>Signed Distance Function

ra lépünk, vagy nem érjük el a felületet. Tehát lépjünk az aktuális pozícióból a sugár irányába  $d$  távolságot. Legyen az így kapott pont az új aktuális pozíció.

2. Frissítsük a felülettől való távolságunkat az SDF új pozícióban történő kiértékelésével.

A fent leírtak megvalósítását az 1. algoritmus mutatja be.

---

#### 1. Algoritmus Sphere tracing

---

**Funct** ST(from, dir, sdf)

```

     $p := \text{from}$  ▷ the current position
     $d := \text{sdf}(p)$  ▷ calculate current distance from surface
    while  $d < \text{minimum distance to surface}$  and not out of steps do
         $p := p + d \cdot \text{dir}$  ▷ move along ray
         $d := \text{sdf}(p)$  ▷ update distance from surface
    end while
    return  $p$ 

```

---

### 3.2.1. Normálvektorok approximálása

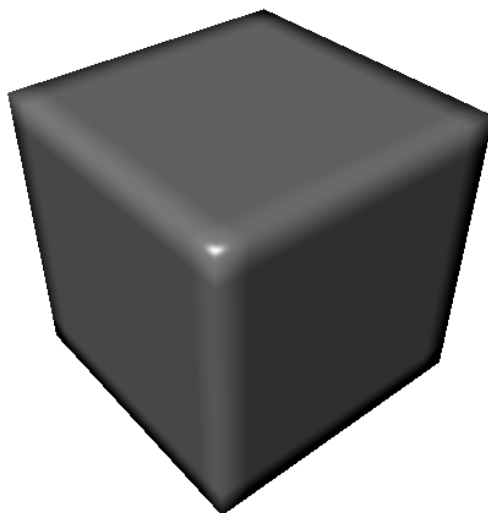
A felület árnyalásához szükséges meghatároznunk a felület normálvektorát minden olyan pontban, ahol a sphere tracing eljárás során indított sugarak azt eltalálják. Mivel az előjeles távolságfüggvény gradiense ezen pontokban a felület normálvektora, ezért ezek meghatározásának legegyszerűbb módja a gradiens numerikus approximálása.

Legyen  $\epsilon$  egy választott konstans érték, melyet a gradiens szimmetrikus differenciához használunk. Ekkor egy  $\vec{p} \in \mathbb{R}^3$ ,  $\vec{p} = [x, y, z]^T$  esetén:

$$\nabla f(\vec{p}) \approx \begin{bmatrix} \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon} \\ \frac{f(y+\epsilon) - f(y-\epsilon)}{2\epsilon} \\ \frac{f(z+\epsilon) - f(z-\epsilon)}{2\epsilon} \end{bmatrix}$$

ha a megjelenítendő felületünk éles éleket, illetve apró részleteket is tartalmaz, akkor előfordulhat, hogy a differenciához számításakor az  $\epsilon$  méretű lépéssel fontos részleteket lépünk át a távolságfüggvényen. Például egy kocka éléhez közeli felületi pontokban előfordulhat, hogy a számlálóban található különbség két tagját a kocka különböző oldalainak közeléből vesszük. Ilyenkor a kapott normálvektor nem az adott oldalra merőleges vektor lesz, hanem egy a szomszédos oldal normálvektora által torzított vektor. A 3.1 ábrán szemléltetem ezt a jelenséget.





**3.1. ábra.** *Differenciahányadosokkal approximált normálvektorral kapott hibás árnyalás az élek mentén. Az oldalak határa nem különül el élesen. (1 élhosszúságú kocka,  $\epsilon = 0.1$ )*

Ennek a problémának a mérséklésére célszerű  $\epsilon$ -t csökkentenünk. Ilyenkor azonban több numerikus probléma is fellép:

- A fenti képletben  $2\epsilon$ -al osztunk, amely a lebegőpontos számábrázolás miatt jelentős pontatlanságot okozhat, amikor  $\epsilon$  nagyon kicsi.
- Amikor a normálvektort olyan pontban számítjuk ki, ahol a felület görbülete kicsi, akkor a számlálókban szereplő különbségek kiszámításakor két egymáshoz nagyon közeli szám különbségét határozzuk meg. Ez lebegőpontos számábrázolásnál szintén jelentősen növeli az eredmény hibáját.

A bevezetésben található 2 számú ábrán szemléltetem a különböző  $\epsilon$  értékek mellett kapott árnyalást, illetve a 3 ábrán azt is, hogy bizonyos esetekben nincs olyan megfelelő  $\epsilon$  érték, amivel minden jellegű hiba eltűnne.

Mindez motivációt ad arra, hogy más módot keressünk a normálvektorok meghatározására.

### 3.3. Duális számok

Az alábbiakban a duális számok rövid bemutatása található, melyet első sorban Szirmay-Kalos 2021-es cikkére [3] alapozok.

**5. Definíció.** Egyváltozós elsőrendű duális szám

Legyen  $r, d \in \mathbb{R}$  és  $i$  egy képzetes egység, melyet a komplex számoknál használt imaginárius egységhez hasonlóan definiálunk, azzal a különbséggel, hogy  $i^2 := 0$  legyen. Legyen  $\mathcal{D}$  az így kapott  $r + di$  duális számok halmaza.

**Jelölés:**

$$r + di = D(r, d)$$

$$\ddot{\mathbf{x}} \in \mathcal{D}, \ddot{\mathbf{x}} = D(x_0, x_1)$$

#### 3.3.1. Alapműveletek

Legyen  $\ddot{\mathbf{x}}, \ddot{\mathbf{y}} \in \mathcal{D}$ .

Az alapműveleteket a komplex számokkal analóg módon definiálhatjuk:

$$\overline{\ddot{\mathbf{x}}} = x_0 - x_1 i = D(x_0, -x_1) \quad (3.1)$$

$$\ddot{\mathbf{x}} \pm \ddot{\mathbf{y}} = D(x_0 \pm y_0, x_1 \pm y_1) \quad (3.2)$$

$$\begin{aligned} \ddot{\mathbf{x}} \cdot \ddot{\mathbf{y}} &= (x_0 + x_1 i) \cdot (y_0 + y_1 i) \\ \ddot{\mathbf{x}} \cdot \ddot{\mathbf{y}} &= x_0 y_0 + x_0 y_1 i + x_1 y_0 i + x_1 y_1 i^2 \\ \ddot{\mathbf{x}} \cdot \ddot{\mathbf{y}} &= x_0 y_0 + (x_0 y_1 + x_1 y_0) i = D(x_0 y_0, x_0 y_1 + x_1 y_0) \end{aligned} \quad (3.3)$$

$$\begin{aligned} \frac{\ddot{\mathbf{x}}}{\ddot{\mathbf{y}}} &= \frac{x_0 + x_1 i}{y_0 + y_1 i} \\ \frac{\ddot{\mathbf{x}}}{\ddot{\mathbf{y}}} &= \frac{\ddot{\mathbf{x}} \cdot \overline{\ddot{\mathbf{y}}}}{\ddot{\mathbf{y}} \cdot \overline{\ddot{\mathbf{y}}}} \\ \frac{\ddot{\mathbf{x}}}{\ddot{\mathbf{y}}} &= \frac{x_0 y_0 + (x_1 y_0 - x_0 y_1) i}{y_0^2 + (y_0 y_1 - y_1 y_0) i} \\ \frac{\ddot{\mathbf{x}}}{\ddot{\mathbf{y}}} &= \frac{x_0 y_0 + (x_1 y_0 - x_0 y_1) i}{y_0^2} = D\left(\frac{x_0}{y_0}, \frac{x_1 y_0 - x_0 y_1}{y_0^2}\right) \end{aligned} \quad (3.4)$$

Most tekintsünk a duális szám első (valós) komponensére úgy, mint egy  $f : \mathbb{R} \rightarrow \mathbb{R}$  függvény helyettesítési értékére. A második komponenset pedig fogjuk fel úgy, mint a függvény  $f'$  deriváltjában azonos helyen vett helyettesítési értékét. Azaz

legyen  $x \in \mathbb{R}$  és  $f : \mathbb{R} \rightarrow \mathbb{R}, f \in D$ , ahol  $f \in D$  azt jelöli, hogy  $f$  deriválható a teljes értelmezési tartományán. A duális számunk pedig legyen:  $\ddot{\mathbf{d}} \in \mathcal{D}, \ddot{\mathbf{d}} = D(f(x), f'(x))$

Most ha visszatekintünk az előbbieken bevezetett műveletekre, akkor azt látjuk a jobb oldalon, hogy a négy alpművelet mindegyikénél az eredményként kapott duális szám valós része egyezik a két input duális szám valós részein, mint valósokon elvégzett művelet eredményével. Hasonlóan, a kapott duális szám képzetes része az adott művelet esetén alkalmazandó deriválási szabály eredményével egyezik.

### 3.3.2. Függvények

**6. Definíció.**  $\mathbb{R} \rightarrow \mathbb{R}$  függvény  $\mathbb{R} \rightarrow \mathcal{D}$  megfelelője

Legyen  $f : \mathbb{R} \rightarrow \mathbb{R}, f \in D$  függvény.

Ekkor  $\underline{f} : \mathbb{R} \rightarrow \mathcal{D}$  legyen a következő módon képezett függvény:

$$\underline{f}(x) = D(f(x), f'(x))$$

**7. Definíció.**  $\mathbb{R} \rightarrow \mathbb{R}$  függvény  $\mathcal{D} \rightarrow \mathcal{D}$  megfelelője

Legyen  $f : \mathbb{R} \rightarrow \mathbb{R}, f \in D$  függvény.

Ekkor  $\overline{f} : \mathcal{D} \rightarrow \mathcal{D}$  megfelelőjét a láncszabály segítségével képezzük:

$$\overline{f}(\ddot{\mathbf{x}}) = D(f(x_0), f'(x_0) \cdot x_1)$$

Hiszen  $x_1$  a belső függvény deriváltjának helyettesítési értéke.

$f(x)$	$\underline{f}(x)$	$\overline{f}(\ddot{\mathbf{x}})$
$c \in \mathbb{R}$	$D(c, 0)$	$D(c, 0)$
$\sin(x)$	$D(\sin(x), \cos(x))$	$D(\sin(x_0), \cos(x_0)x_1)$
$x^n, n \in \mathbb{N}$	$D(x^n, nx^{n-1})$	$D(x_0^n, nx_0^{n-1}x_1)$

**3.1. táblázat.** Néhány példa valós függvények duális megfelelőire.

A fentiek segítségével már bevezethető az automatikus differenciálás módszere.

### 3.3.3. Automatikus differenciálás

Az automatikus differenciálás célja nem más, mint hogy egy függvényérték kiszámítása közben a valós számok aritmetikáját duális számokkal és műveletekkel helyettesítve a függvényérték mellett a derivált értékét is meghatározzuk. Ehhez nincs másra szükség, mint hogy a függvényérték kiszámításának műveletsorát duális számokkal számoljuk végig. Így az eredményként kapott duális szám valós részében a függvényértéket, képzetes részében a deriváltat kapjuk meg. Az eljárás legnagyobb előnye, hogy implementálható egy könyvtárba oly módon, hogy a típusok kicserélésén kívül csak minimális változtatás kelljen egy bonyolultabb függvény deriváltjának meghatározásához is. Ezen kívül ez az eljárás mentes a differenciahányadossal való közelítés pontatlansági problémáitól.

### 3.3.4. Kiterjesztés magasabb deriváltakra

Magasabb rendű deriváltak esetén a kiterjesztés során azt a tulajdonságot akarjuk megőrizni, hogy a duális szám képzetes részeiben tartalmazza a deriváltakat. Ez vezet el az alábbi kiterjesztett duális számfogalomhoz:

**8. Definíció.**  $N$ -ed rendű duális szám

Legyen  $1 \leq N \in \mathbb{N}$ .

Legyen  $\vec{x} \in \mathbb{R}^N$ .

Ekkor az  $N$ -ed rendű duális szám alakja a következő:

$$\mathcal{D}^{(N)} \ni \ddot{\mathbf{d}} = \sum_{k=0}^N x_k i_k = D^{(N)}(x_0, x_1, \dots, x_N)$$

Ahol az egyszerűbb jelölés érdekében  $i_0 := 1$  és  $i_1, \dots, i_N$  egymástól különböző képzetes egységek, valamint  $\forall j \in \{1, 2, \dots, N\} : i_j^2 = 0$ .

#### Műveletek

Legyen  $\ddot{\mathbf{x}}, \ddot{\mathbf{y}} \in \mathcal{D}^{(N)}$ .

A magasabb rendű duális számok közötti műveleteket az elsőrendű esethez analóg

módon kapjuk:

$$\bar{\bar{\mathbf{x}}} = x_0 - \sum_{k=1}^N x_k i_k = D^{(N)}(x_0, -x_1, \dots, -x_N) \quad (3.5)$$

$$\ddot{\mathbf{x}} \pm \ddot{\mathbf{y}} = \sum_{k=0}^N (x_k \pm y_k) i_k = D^{(N)}(x_0 \pm y_0, x_1 \pm y_1, \dots, x_N \pm y_N) \quad (3.6)$$

**Szorzás** Ha kibontjuk két  $N$ -ed rendű duális szám szorzatát, akkor a kapott összegben megjelennek  $i_j i_k (j, k \in \mathbb{N})$  együtthatóval rendelkező tagok. Először definiáljuk két különböző képzetes egység szorzatát:

**9. Definíció.** Különböző képzetes egységek szorzata

Legyen  $j, k \in \mathbb{N}, j \neq k$ .

$$i_j i_k := \binom{j+k}{k} i_{j+k}, \text{ ha } j+k \leq N, \text{ különben } 0$$

Ezzel a definícióval elvégezve a szorzást bizonyítható [3] a kommutativitás és asszociativitás. A kapott szorzat képlet a következő:

$$\ddot{\mathbf{x}} \cdot \ddot{\mathbf{y}} = \sum_{j=0}^N \sum_{k=0}^{N-j} x_j y_k \binom{j+k}{k} i_{j+k} \quad (3.7)$$

**Osztás** Az osztás elvégzését szintén az elsőrendű esethez hasonlóan tesszük. Bővítünk a nevező konjugáltjával (3.5). Azonban  $N$ -ed rendű esetben már nem garantált, hogy egy bővítés után a nevezőben szereplő duális szám minden képzetes komponense kinullázódik. Ezért az osztás során rekurzívan ismételni kell a bővítéseket<sup>3</sup> egészen addig, amíg a nevezőben csupán egy valós érték marad, amellyel már elvégezhető a tagonkénti osztás. Belátható, hogy a rekurzió  $O(\log_2(N) + 1)$  lépésben befejeződik [3].

## Függvények

**10. Definíció.**  $\mathbb{R} \rightarrow \mathbb{R}$  függvény  $\mathbb{R} \rightarrow \mathcal{D}^{(N)}$  megfelelője

Legyen  $f : \mathbb{R} \rightarrow \mathbb{R}, f \in D^N$  függvény.

Ekkor  $\underline{f} : \mathbb{R} \rightarrow \mathcal{D}^{(N)}$  megfelelője a következő függvény:

$$\underline{f}(x) = D(f(x), f'(x), \dots, f^{(N)}(x))$$

<sup>3</sup>Emiatt a duális osztás jelentősen költségesebb művelet a duális szorzásnál.

**11. Definíció.**  $\mathbb{R} \rightarrow \mathbb{R}$  függvény  $\mathcal{D}^{(N)} \rightarrow \mathcal{D}^{(N)}$  megfelelője

Legyen  $f : \mathbb{R} \rightarrow \mathbb{R}, f \in D^N$  függvény.

Ekkor  $\bar{f} : \mathcal{D}^{(N)} \rightarrow \mathcal{D}^{(N)}$  megfelelője a következő függvény:

$$\bar{f}(\mathbf{\ddot{x}}) = D(f(x_0), (f(x_0))', \dots, (f(x_0))^{(N)})$$

Felhasználva a **Faà di Bruno formula** kombinatorikus alakját [4] a kapott duális szám komponenseit explicit módon is megadhatjuk:

$$\bar{f}(\mathbf{\ddot{x}}) = D(f(x_0), d_1, d_2, \dots, d_N)$$

$\forall k \in \mathbb{N}, 1 \leq k \leq N$ :

$$d_k = \sum_{\pi \in \pi_k} f^{(|\pi|)}(x_0) \prod_{B \in \pi} x_{|B|}$$

Ahol:

- $\pi_k$  az  $\{1, 2, \dots, k\}$  halmaz összes osztályfelbontása.
- $B \in \pi$  azt jelenti, hogy  $B$  a  $\pi$  osztályfelbontásban szereplő egyik osztály. ("blokk")<sup>4</sup>

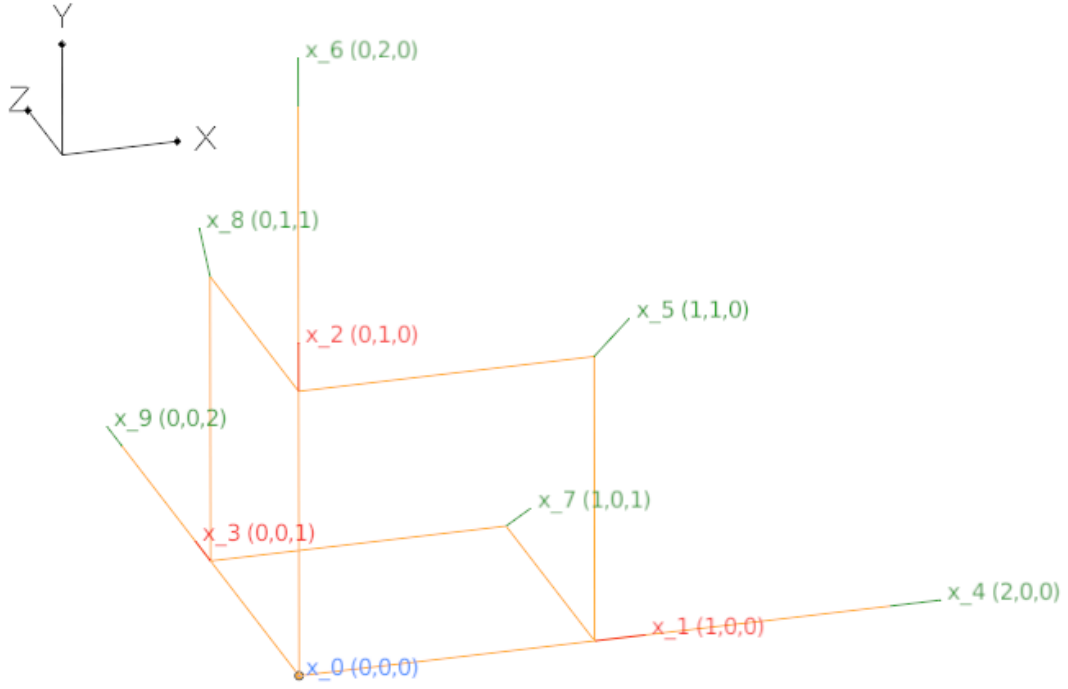
### 3.3.5. Kiterjesztés többváltozós függvényekre

Mivel maga az előjeles távolságfüggvény három változós, deriváltjainak meghatározására szükséges a duális számok elméletét többváltozós függvényekre is kiterjeszteni. Az egyszerűség kedvéért itt most csak a háromváltozós esetet tárgyaljuk. Ahogy egyváltozós esetben a duális szám a deriváltak helyettesítési értékeit tárolta, úgy többváltozós esetben a különböző parciális deriváltak értékeit fogjuk képzetes részként tárolni.

#### Tetraéderes indexelés

Implementációs szempontból érdemes meghatározzuk a különböző parciális deriváltak sorrendjét, indexelését. Előnyös, ha ez az indexelés magasabb rendre való bővítéskor az alacsonyabb rendű deriváltakhoz tartozó parciálisokat helyben hagyja, azaz indexük nem változik. Egy lehetséges ilyen indexelést a 3.2 ábra mutat be:

<sup>4</sup>Tehát ha például  $\pi = \{\{1, 2, 3\}, \{4, 5\}, \{6\}\}$  osztályozás, akkor  $B$  rendere a következő értékeket veszi fel:  $\{1, 2, 3\}$ , majd  $\{4, 5\}$  és végül  $\{6\}$ .



**3.2. ábra.** *Parciális deriváltak tetraéderes indexelése. Az ábrán látható  $x_k(a, b, c)$ ,  $0 \leq a + b + c \leq N$  jelölés a következő képpen értelmezendő: Az  $x, y, z$  szerint rendre  $a, b, c$ -szeres parciális derivált indexe  $k$ .*

*Kék:  $x_0(0, 0, 0)$  a függvényérték (0. derivált)*

*Piros: az első derivált parciálisai*

*Zöld: a második derivált parciálisai*

Megfigyelhető, hogy ezen sorrend mellett tetszőleges  $(a, b, c)$ -szeres<sup>5</sup> parciális derivált indexe mindig ugyan az, amennyiben  $N \geq a + b + c$ . Hiszen ha  $N$ -et növeljük magasabb rendű deriváltak meghatározása céljából, akkor az új parciális deriváltak egy új háromszöglapon fognak megjelenni.

**Jelölés:**

- $\triangleleft(n), n \in \mathbb{N}$ : Az  $n$ -edik tetraéder szám.

Például:  $\triangleleft(1) = 1, \triangleleft(2) = 4, \triangleleft(3) = 10$

- $\triangle(n), n \in \mathbb{N}$ : Az  $n$ -edik háromszög szám.

Például:  $\triangle(1) = 1, \triangle(2) = 3, \triangle(3) = 6$

**12. Definíció.** Adott parciális derivált tetraéder indexe.

Legyen  $x, y, z \in \mathbb{N}, x + y + z \leq N$ .

$$\text{idx}(x, y, z) = \triangleleft(x + y + z + 1) - \triangleleft(x + y + 1) + y$$

<sup>5</sup> $(a, b, c)$  a  $\partial^a x \partial^b y \partial^c z$  vegyes deriváltat jelenti.

**13. Definíció.** 3 változós  $N$ -ed rendű duális szám

Legyen  $N \in \mathbb{N}, N > 0$ .

Ekkor a 3 változós  $N$ -ed rendű duális szám alakja:

$$\mathcal{D}_3^{(N)} \ni \ddot{\mathbf{d}} = D_3^{(N)}(x_0, x_1, \dots, x_{\triangleleft(N)})$$

Itt  $\forall 0 < k \in \mathbb{N}$  esetén  $x_k$  azt az  $(a, b, c)$  parciális deriváltat tartalmazza, amelyre  $\text{idx}(a, b, c) = k$ .

Képzetes egységekkel összegként felírva:

$$\begin{aligned} \ddot{\mathbf{d}} &= x_0 i_{0,0,0} + x_1 i_{1,0,0} + x_2 i_{0,1,0} + x_3 i_{0,0,1} + x_4 i_{2,0,0} + \dots + x_{\triangleleft(N)} i_{0,0,N} \\ \ddot{\mathbf{d}} &= \sum_{a=0}^N \sum_{b=0}^{N-a} \sum_{c=0}^{N-a-b} x_{\text{idx}(a,b,c)} i_{a,b,c} \end{aligned}$$

Ahol a korábbiakhoz hasonlóan  $i_{0,0,0} = 1$  és a többi  $i_{a,b,c}$  képzetes tag együttthátója a rendre  $x, y, z$  szerint  $a, b, c$ -szeres parciális derivált, azaz  $x_{\text{idx}(a,b,c)}$ .

Továbbá  $\forall a, b, c \in \mathbb{N}, a + b + c \leq N : (i_{a,b,c})^2 = 0$ .

### Műveletek

Legyen  $\ddot{\mathbf{x}}, \ddot{\mathbf{y}} \in \mathcal{D}_3^{(N)}$ .

Az összeg és különbség, valamint a konjugált meghatározásának módja nem változik többváltozós esetben:

$$\overline{\ddot{\mathbf{x}}} = D_3^{(N)}(x_0, -x_1, \dots, -x_{\triangleleft(N)}) \quad (3.8)$$

$$\ddot{\mathbf{x}} \pm \ddot{\mathbf{y}} = D_3^{(N)}\left(x_0 \pm y_0, x_1 \pm y_1, \dots, x_{\triangleleft(N)} \pm y_{\triangleleft(N)}\right) \quad (3.9)$$

**Szorzás** A szorzás elvégzéséhez értelmeznünk kell az  $i_{a,b,c} \cdot i_{j,k,l}$  szorzatot:

**14. Definíció.** Különböző 3 változós képzetes egységek szorzata ([3])

Legyen  $a, b, c, j, k, l \in \mathbb{N}$ .

$$i_{a,b,c} \cdot i_{j,k,l} := \begin{cases} \binom{a+j}{j} \binom{b+k}{k} \binom{c+l}{l} i_{a+j,b+k,c+l} & \text{ha } a + b + c + j + k + l \leq N \\ 0 & \text{különben} \end{cases}$$



Legyen  $\ddot{\mathbf{x}}, \ddot{\mathbf{y}} \in \mathcal{D}_3^N$ . Ekkor:

$$\begin{aligned}
 \ddot{\mathbf{x}} \cdot \ddot{\mathbf{y}} &= \left( \sum_{a=0}^N \sum_{b=0}^{(N-a)} \sum_{c=0}^{(N-a-b)} x_{\text{idx}(a,b,c)} i_{a,b,c} \right) \cdot \left( \sum_{j=0}^N \sum_{k=0}^{(N-j)} \sum_{l=0}^{(N-j-k)} y_{\text{idx}(j,k,l)} i_{j,k,l} \right) \\
 &= \sum_{a=0}^N \sum_{b=0}^{(N-a)} \sum_{c=0}^{(N-a-b)} \sum_{j=0}^N \sum_{k=0}^{(N-j)} \sum_{l=0}^{(N-j-k)} x_{\text{idx}(a,b,c)} y_{\text{idx}(j,k,l)} \cdot i_{a,b,c} i_{j,k,l} \\
 &= \sum_{a=0}^N \sum_{b=0}^{(N-a)} \sum_{c=0}^{(N-a-b)} \sum_{j=0}^N \sum_{k=0}^{(N-j)} \sum_{l=0}^{(N-j-k)} x_{\text{idx}(a,b,c)} y_{\text{idx}(j,k,l)} \cdot \binom{a+j}{j} \binom{b+k}{k} \binom{c+l}{l} i_{a+j,b+k,c+l} \\
 &\quad \text{Felhasználva, hogy } i_{a,b,c} \cdot i_{j,k,l} = 0 \text{ ha } a+b+c+j+k+l > N : \\
 \ddot{\mathbf{x}} \cdot \ddot{\mathbf{y}} &= \sum_{a=0}^N \sum_{b=0}^{(N-a)} \sum_{c=0}^{(N-a-b)} \sum_{j=0}^{(N-a-b-c)} \sum_{k=0}^{(N-a-b-c-j)} \sum_{l=0}^{(N-a-b-c-j-k)} x_{\text{idx}(a,b,c)} y_{\text{idx}(j,k,l)} \cdot \binom{a+j}{j} \binom{b+k}{k} \binom{c+l}{l} i_{a+j,b+k,c+l} \quad (3.10)
 \end{aligned}$$

**Osztás** Háromdimenziós esetben is alkalmazható az osztás elvégzésére a 3.3.4 részben leírt rekurzív módszer.

### Függvények

Folytatjuk a kiterjesztést:

**15. Definíció.**  $\mathbb{R}^3 \rightarrow \mathbb{R}$  függvény  $\mathbb{R}^3 \rightarrow \mathcal{D}_3^{(N)}$  megfelelője

Legyen  $f : \mathbb{R}^3 \rightarrow \mathbb{R}, f \in D^N$  függvény.

Ekkor  $\underline{f} : \mathbb{R}^3 \rightarrow \mathcal{D}^{(N)}$  megfelelője a következő függvény:

$$\underline{f}(x, y, z) = \sum_{j=0}^N \sum_{k=0}^{(N-j)} \sum_{l=0}^{(N-j-k)} \partial_x^j \partial_y^k \partial_z^l f(x, y, z) \cdot i_{j,k,l}$$

**16. Definíció.**  $\mathbb{R} \rightarrow \mathbb{R}$  függvény  $\mathcal{D}_3^{(N)} \rightarrow \mathcal{D}_3^{(N)}$  megfelelője

Legyen  $f : \mathbb{R} \rightarrow \mathbb{R}, f \in D^N$  függvény.

Ekkor  $\bar{f} : \mathcal{D}_3^{(N)} \rightarrow \mathcal{D}_3^{(N)}$  megfelelője a következő függvény:

$$\bar{f}(\ddot{\mathbf{x}}) = \sum_{j=0}^N \sum_{k=0}^{(N-j)} \sum_{l=0}^{(N-j-k)} \partial_x^j \partial_y^k \partial_z^l f(x_0) \cdot i_{j,k,l}$$

A  $\partial_x^j \partial_y^k \partial_z^l f(x_0)$  tényezőt az előzőekhez hasonlóan a **Faà di Bruno féle formula** kombinatorikus alakjával [4] bonthatjuk ki, melyben megjelennek a belső függvény deriváltjai. Ezeket az  $\ddot{\mathbf{x}}$  duális paraméter tartalmazza:

$$\partial_x^j \partial_y^k \partial_z^l f(x_0) = \sum_{\pi \in \pi_{j,k,l}} f^{(|\pi|)}(x_0) \prod_{B \in \pi} x_{\text{idx}(B_x, B_y, B_z)}$$

Ahol:

- $\pi_{j,k,l}$  az  $\underbrace{\{\alpha_1, \dots, \alpha_j\}}_{\mathfrak{A}} \cup \underbrace{\{\beta_1, \dots, \beta_k\}}_{\mathfrak{B}} \cup \underbrace{\{\gamma_1, \dots, \gamma_l\}}_{\mathfrak{C}}$  halmaz összes osztályfelbontása.
- $B \in \pi$  azt jelenti, hogy  $B$  a  $\pi$  osztályfelbontásban szereplő egyik osztály. ("blokk")<sup>6</sup>
- $B_x = |B \cap \mathfrak{A}|$ ,  $B_y = |B \cap \mathfrak{B}|$ ,  $B_z = |B \cap \mathfrak{C}|$  értékek adják meg, hogy a szorzaton belül az  $\ddot{\mathbf{x}}$  duális paraméter  $(B_x, B_y, B_z)$  parciális derivált komponensét kell venni.

A fenti képlet mögötti kombinatorikai jelentés a következő:

$\pi_{i,j,k}$  halmaz jelképezi az összes  $f$ -en végrehajtandó parciális deriválást.  $\pi_{i,j,k}$  elemeket három csoportra osztjuk  $(\mathfrak{A}, \mathfrak{B}, \mathfrak{C})$  aszerint, hogy  $x, y$  vagy  $z$  szerintiek. Ezeket felosztjuk az összes lehetséges módon csoportokra. A képletben szereplő összegben belül  $f$  annyszoros deriváltja szerepel, ahány csoport van az aktuális felbontásban. A produktum pedig végighalad az összes csoporton a felbontáson belül és az egyes csoportoknak megfelelő parciális komponenseit szorozza össze  $\ddot{\mathbf{x}}$ -nek. Azt hogy egy csoporthoz melyik komponens tartozik az határozza meg, hogy benne hányszor szerepel  $x, y$  vagy  $z$  szerinti deriválást jelképező elem.

<sup>6</sup>Tehát ha például  $\pi = \{\{1, 2, 3\}, \{4, 5\}, \{6\}\}$  osztályozás, akkor  $B$  rendere a következő értékeket veszi fel:  $\{1, 2, 3\}$ , majd  $\{4, 5\}$  és végül  $\{6\}$ .

Ez a szabály a láncszabály és a szorzás deriválási szabályának ismételt alkalmazásából keletkezik, bizonyítása megtalálható Hardy “Combinatorics of Partial Derivatives” című cikkében. [4]

### 3.4. Görbületek

A második deriváltak szemléltetésére a dolgozatban Gauss- és középgörbületeket alkalmaztam. Ezek melyek meghatározásához az alábbi képleteket használtam [5]:

**17. Definíció.** Gauss-görbület

$$K_G = \frac{\nabla F \cdot H^*(F) \cdot \nabla F^T}{|\nabla F|^4}$$

**18. Definíció.** Középgörbület

$$K_M = \frac{\nabla F \cdot H(F) \cdot \nabla F^T - |\nabla F|^2 \text{Trace}(H)}{2|\nabla F|^3}$$

Ahol  $F$  a távolságfüggvény,  $\nabla F$  a gradiense,  $H(F)$  a Hesse-mátrix, melynek  $H^*(F)$  jelöli az adjungáltját.  $|\bullet|$  a kettes normát jelenti.

## 4. fejezet

# Megvalósítás

Ebben a fejezetben a távolságfüggvényt és annak deriváltját kiszámító GPU kód generálását mutatom be. Mindehhez a háromdimenziós modellt Constructive Solid Geometry megközelítéssel állítom elő, az ennek megfelelő műveleti gráffal reprezentálva. Mindezt követően kitérek az eredményként kapott GPU kóddal kapcsolatosan felmerülő problémákra is.

### 4.1. Constructive Solid Geometry

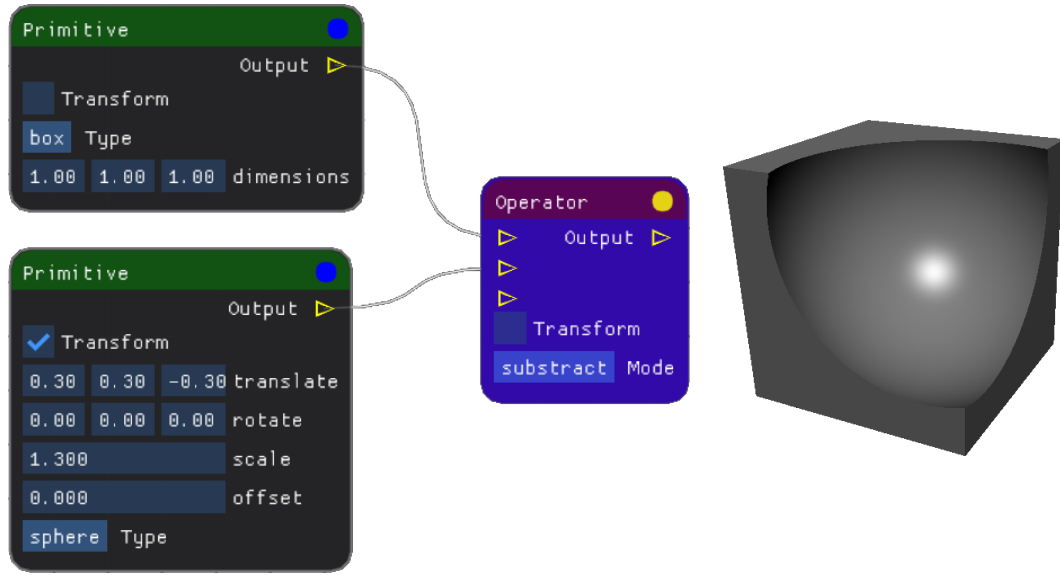
**Constructive Solid Geometry (CSG)** néven a szilárdtest modellezési eljárások azon csoportjára szokás hivatkozni, melyek primitív testek és halmazműveletek segítségével írják le összetettebb háromdimenziós modelleket. Minden CSG modell megadható annak műveleti grájával, melyet CSG gráfnak nevezünk.

#### 4.1.1. CSG Gráf

A CSG gráf egy olyan fagráf, melynek levelei primitív testeket reprezentálnak, míg belső csúcsai ezeken értelmezett halmazműveleteket. A csúcsok közötti élek<sup>1</sup> reprezentálhatnak a csúcsok kimenetén értelmezett unáris műveleteket: például eltolást, forgatást, skálázást vagy egyéb geometriai transzformációkat. Egy CSG gráf tetszőleges csúcsából mint gyökérből kiinduló részfája is értelmes CSG gráf, ami lehetővé teszi a rekurzív módon történő feldolgozást. A 4.1 ábrán látható egy példa az implementált modellezőszoftverből.

---

<sup>1</sup>Az általam implementált programban a transzformációkat az élek helyett ekvivalens módon a csúcsokhoz csatolva tárolom.



4.1. ábra. Egyszerű CSG gráf és az általa reprezentált test.

### Műveletek távolságfüggvényeken

Implicit felületek CSG gráffal történő modellezése az előjeles távolságfüggvény, vagy annak egy alsó becslését<sup>2</sup> megadó függvény gráfból történő generálásával történik. A továbbiakban az előjeles távolságot  $sd$ , annak egy az előbbi értelemben vett alsó becslését  $sd^*$ -gal fogom jelölni. Vezessünk be néhány modellezéshez használatos műveletet távolságokon:

**19. Definíció.** Legyen  $\vec{x} \in \mathbb{R}^3$  a tér egy pontja, valamint  $A$  és  $B$  két felület.  $\alpha := sd^*(\vec{x}, A)$  és  $\beta := sd^*(\vec{x}, B)$  a felületektől vett előjeles távolságok, vagy azok alsó becslése.

$$sd^*(\vec{x}, A \cup B) := \min(\alpha, \beta) \quad (4.1)$$

$$sd^*(\vec{x}, A \cap B) := \max(\alpha, \beta) \quad (4.2)$$

$$sd^*(\vec{x}, A \setminus B) := \max(\alpha, -\beta) \quad (4.3)$$

*Megjegyzés.* A fenti képletek közül egyik sem ad a tér minden pontjában helyes távolságfüggvényt a két test uniójára, metszetére vagy különbségére. a 4.1 unióra vonatkozó képlet esetén a testen kívüli térrészben helyes a két előjeles távolság minimumát venni, ugyanakkor a belső térrészben ez a (negatív előjeles) távolságnak felső becslését adja meg. a 4.2 metszet és a 4.3 különbség esetében a testen kívüli

<sup>2</sup>Itt az alsó becslés esetén szét kell választani a felületen belül és kívül lévő pontokat. Ha egy pont kívül van, akkor az alsó becslés a valós távolságnál kisebb. Ha a felületen belül van, akkor abszolút értékben kisebb az előjeles távolságnál (de szintén negatív).

térrészben is csak alsó becslést kapunk a távolságra. Ugyanakkor ez a felület megjelenítésén nem változtat, ugyanis az 1. algoritmussal megadott sphere tracing eljárás a távolságok alsó becslése mellett is jól működik, bár ilyenkor műveletigénye nő. Részletek Quilez cikkében [6].

A minimum és maximum (illetve a későbbiekben a *clamp* művelet és az abszolút érték) használata elrontja  $f$  deriválhatóságát. A megjelenítés során viszont ez nem okoz gondot, ugyanis a lebegőpontos számok pontatlansága, valamint a pixelekre osztottság miatt elhanyagolhatóan ritkán fordul elő az, hogy a kilőtt sugár pontosan az élen találja el a felületet.

**20. Definíció.**  $a, b, x \in \mathbb{R}, a < b$

$$\text{clamp}_{a,b}(x) := \begin{cases} a & \text{ha } x < a \\ x & \text{ha } a \leq x \leq b \\ b & \text{ha } b < x \end{cases}$$

**21. Definíció.** Néhány további művelet távolságokon. [7] Az eredményként kapott felületeket a 4.2 ábra szemlélteti.

Legyen  $k \in \mathbb{R}^+$  egy tetszőleges paraméter, melynek hatása a 4.3 ábrán látható.

**Simított unió:**

$$\text{sd}^*(\vec{x}, A \cup_k B) := h\alpha + (1-h)\beta - kh(1-h)$$

Ahol  $h = \text{clamp}_{0,1}\left(\frac{1}{2} + \frac{\beta-\alpha}{2k}\right)$ .

**Simított metszet:**

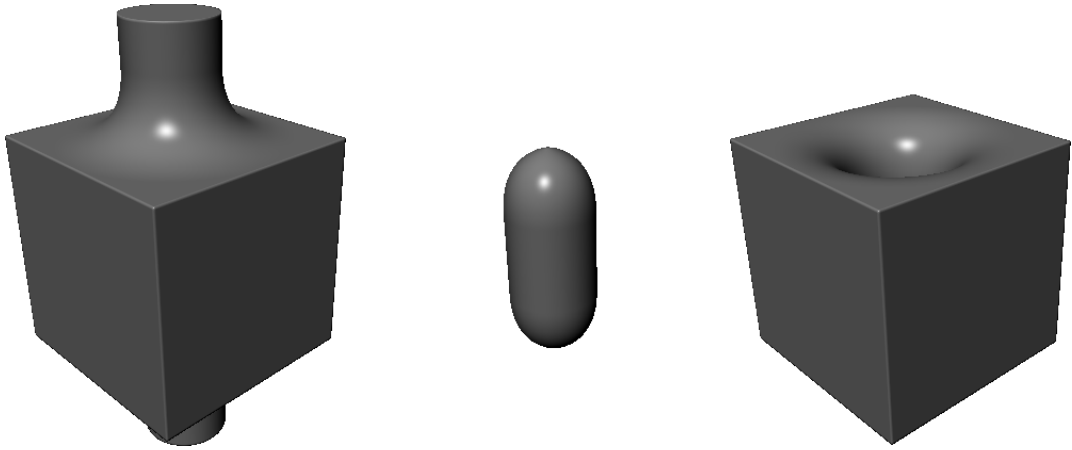
$$\text{sd}^*(\vec{x}, A \cap_k B) := -h\alpha + (1-h)\beta + kh(1-h)$$

Ahol  $h = \text{clamp}_{0,1}\left(\frac{1}{2} - \frac{\beta-\alpha}{2k}\right)$ .

**Simított különbség:**

$$\text{sd}^*(\vec{x}, A \setminus_k B) := h\alpha + (1-h)\beta + kh(1-h)$$

Ahol  $h = \text{clamp}_{0,1}\left(\frac{1}{2} - \frac{\beta-\alpha}{2k}\right)$ .

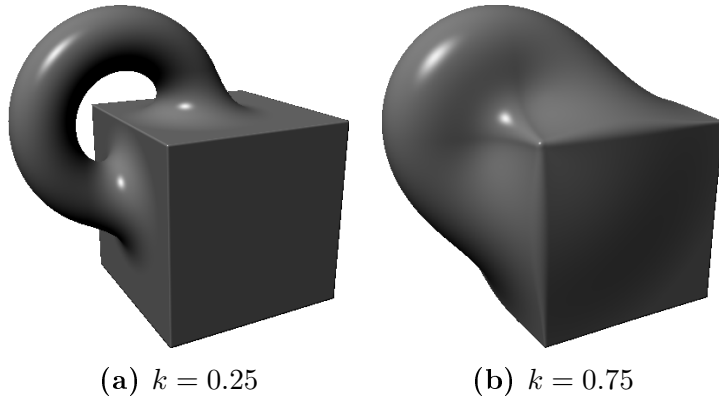


(a) *Simított unió*

(b) *Simított metszet*

(c) *Simított különbség*

**4.2. ábra.** *Simított műveletek egy kocka és egy henger között.*

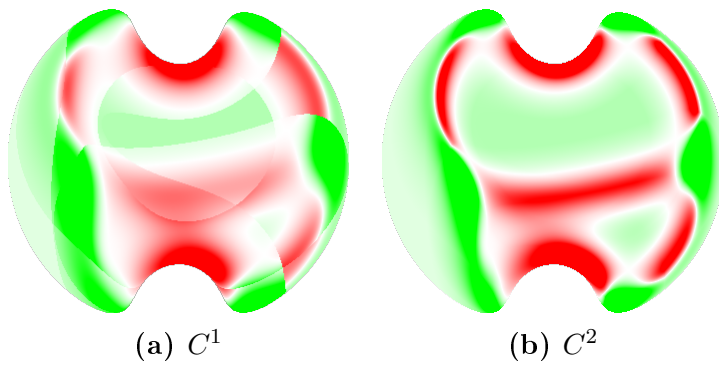


(a)  $k = 0.25$

(b)  $k = 0.75$

**4.3. ábra.** *Simított unió különböző  $k$  paraméterek mellett.*

*Megjegyzés.* A 21. definícióban megadott műveletek a folytonosságot csak  $C^1$ -ig tartják meg. Ez a legtöbb esetben nem okoz problémát, de másodrendű deriváltaknál viszont már éles határok jelennek meg (4.4 ábra). A simított műveletek definiálhatóak  $C^n$  folytonosra is [8] alapján.



(a)  $C^1$

(b)  $C^2$

**4.4. ábra.**  $C^1$  és  $C^2$  folytonosság tartó műveletekkel kapott felület Gauss-görbülete.

### 4.1.2. Transzformációk részgráfokon

Minden egyes CSG gráfbeli részfa önmagában is CSG gráf. Az általa reprezentált művelet sor eredménye egy előjeles távolság, amelyet a felület megjelenítéséhez használunk fel. Eddig a gráfunk primitívekből állt, melyeket bináris, vagy nagyobb aritású műveletekkel<sup>3</sup> kapcsoltunk össze. Ezeken kívül még szükség van arra, hogy képesek legyünk az egyes részfa által reprezentált testeket transzformálni (eltolás, forgatás és uniform skálázás<sup>4</sup>). A megvalósítás során ez azt jelenti, hogy a gráf minden csúcsát kiegészítjük az ezeket a transzformációkat leíró opcionális paraméterekkel, majd a kódgenerálásnál ezeket figyelembe véve készítjük el a távolságfüggvényt.

## 4.2. Megjelenítés

A CSG gráffal meghatározott modelleket háromdimenzióban, valós időben szeretnénk megjeleníteni. Ennek érdekében a megjelenítés OpenGL-ben lett implementálva. A vizualizáció a bevezetőben bemutatott **sphere tracing** algoritmust használja, amely a GPU-n egy fragment shaderben lett implementálva.

A **sphere tracing** eljárás kimeneteként megkapjuk, hogy az egyes képpontokon látható-e a felület<sup>5</sup>. Mindez még nem elég a megjelenítéshez, a felületet árnyalni is kell, melyhez szükséges a normálvektor. A bevezetésben bemutatott árnyalási hibák elkerülése érdekében ezt automatikus differenciálással állítjuk elő. Ennek azonban különösen magasabb deriváltak esetén jelentősen magasabb műveletigénye van, mint pusztán a távolság meghatározásának. Mivel maga a sphere tracing nem igényli a deriváltak ismeretét, ezért ehhez meghagyunk egy  $\mathbb{R}^3 \rightarrow \mathbb{R}$  távolságfüggvényt, míg az újonnan bevezetett  $\mathbb{R}^3 \rightarrow \mathcal{D}_3^{(N)}$  duális távolságfüggvényt csak az árnyalás során fogjuk felhasználni. Ez az optimalizáció azért is jelentős, mivel a sphere tracing során egy képpont esetén gyakran olyankor alakul ki magas lépésszám és így sok SDF kiértékelés, amikor a kilőtt sugár közel párhuzamosan halad a felülettel (Példa a 4.5 ábrán). Ez sok olyan képpont esetén is fent áll, ahol a sugár végül nem találja

---

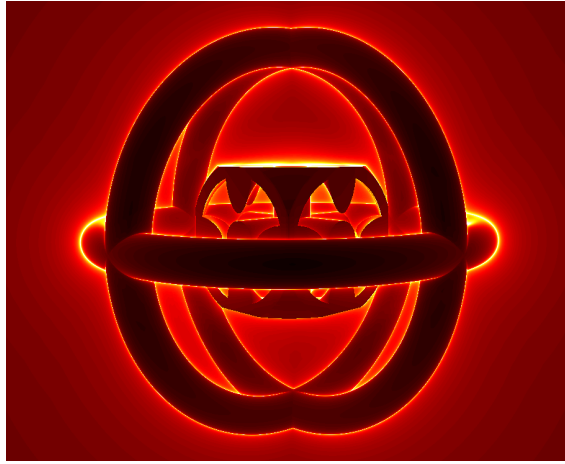
<sup>3</sup>Az implementációban az unió, metszet és különbség műveletek 2 vagy több inputot fogadnak el. Ez a kivonás esetében azt jelenti, hogy az első inputból vonjuk ki az összes többi input által reprezentált testet.

<sup>4</sup>A skálázások közül azért csak az uniform változatot valósítjuk meg, mert a nem uniform esetre csak becslés adható. A távolságot pusztán a transzformálatlan primitívtől tudjuk meghatározni, abból pedig nem mondható meg, hogy egy nem uniform skálázást követően milyen messze lesz a felület a ponttól.

<sup>5</sup>Akkor látható, ha a sphere tracing során az algoritmus azért állt le, mert a képpontból kilőtt sugár elég közel ért a felülethez.



el a felületet. Ezekben a képpontban amúgy sem számolnánk ki az árnylást, így méginkább felesleges a sugár léptetése során a duális távolságfüggvényt alkalmazni.



**4.5. ábra.** *Sphere tracing során megtett lépések száma. Egy pont minél világosabb, annál közelebb volt a lépésszám a maximumhoz. (Maximum: 100)*

Ahhoz, hogy mindez megvalósítható legyen, szükség van a két távolságfüggvény GLSL<sup>6</sup> kódjának CSG gráfból történő generálására. Az alábbiakban ennek a kódgenerálásnak a menetét mutatom be.

### 4.3. Kódgenerálás

Adott a gráfszerkesztő segítségével megadott CSG gráf, melynek levelei primitíveket, belső csúcsai műveleteket reprezentálnak. Minden csúcs rendelkezik továbbá a kimenetén elvégzendő geometriai transzformációkat megadó paraméterekkel. Célunk egy olyan GLSL függvény generálása, amely paraméterként egy három dimenziós vektort kap és ebből meghatároz egy előjeles távolságot, amely alulról becsüli a bemenetként megkapott pont távolságát a CSG gráf által megadott test felületétől.

A kódgeneráláshoz gráfbejárást alkalmazunk. Kihhasználva, hogy a CSG gráf minden csúcsához tartozó részgráfja is rekurzívan CSG gráf, érdemes a mélységi bejárás stratégiáját követni. Mivel a gráfunk fagráf, ezért a gyökből elindított bejárás minden csúcsot pontosan egyszer érint.

A bejárás során a csúcsokhoz létrehozunk (vagy újrahasznosítunk) egy GLSL változót, majd olyan kódot generálunk, ami ebbe letárolja a csúcs kimeneteként ka-

---

<sup>6</sup>OpenGL Shading Language, az OpenGL-ben használatos nyelv a GPU programozására.

pott távolságot. Ha az aktuális csúcs egy primitív, akkor ez rögtön megvalósítható<sup>7</sup> és visszatérünk annak a változónak a nevével, amelybe a primitívtól vett távolság kerül. Amennyiben jelenleg egy belső, műveleti csúcsot dolgozunk fel, akkor első lépésként rekurzívan meghatározzunk az input csúcsokból érkező távolságokat generáló kódot és eltároljuk, hogy ezek az értékek milyen GLSL változókba kerülnek elmentésre. Ezt követően második lépésként lefoglalunk egy új változót és legeneráljuk azt a kódot amely meghatározza a műveletünk eredményét a tárolt input változónevek segítségével, majd az eredményt az imént létrehozott változóba tárolja. Most felszabadíthatjuk az input változókat<sup>8</sup> és visszatérhetünk az új változó nevével. Az itt leírt eljárást a 2. algoritmus pszeudokódja mutatja be.

---

## 2. Algoritmus Rekurzív kódgenerálás váza (transzformációk nélkül)

---

**Funct** TraverseGraph(root, output *<by reference>*)

```

if root is primitive then
    var := CreateOrReuseVariable()
    output += var + " = "
    output += GenCodeForPrimitive(root as primitive) + ";"
    return var                                ▷ return name of variable holding the result
else                                          ▷ root is operator
    variables := EmptyList()
    for node ∈ root.inputs do
        variables.append(TraverseGraph(node, output))
    end for
    var = CreateOrReuseVariable()
    output += var + " = "
    output += GenCodeForOperator(root as operator, variables) + ";"
    for varName ∈ variables do
        FreeVariable(varName)                ▷ free temporary variables for future reuse
    end for
    return var                                ▷ return name of glsl variable containing result
end if

```

**Funct** GenerateSdf(root, output *<by reference>*)

```

output += "float sdf(vec3 pos) {"
resultVarName := TraverseGraph(root, output <by reference>)
output += "return " + resultVarName + ";"
output += "}"

```

---



---

<sup>7</sup>Az implementáció során minden egyes támogatott primitívhez kézzel megadjuk a távolságfüggvényt kiszámító GLSL függvényt, itt csak meg kell hívni a megfelelő paraméterekkel.

<sup>8</sup>Azaz feljegyezzük a generátorban, hogy újrahasznosíthatóak.

### 4.3.1. Transzformációk kezelése - a transzformációs verem

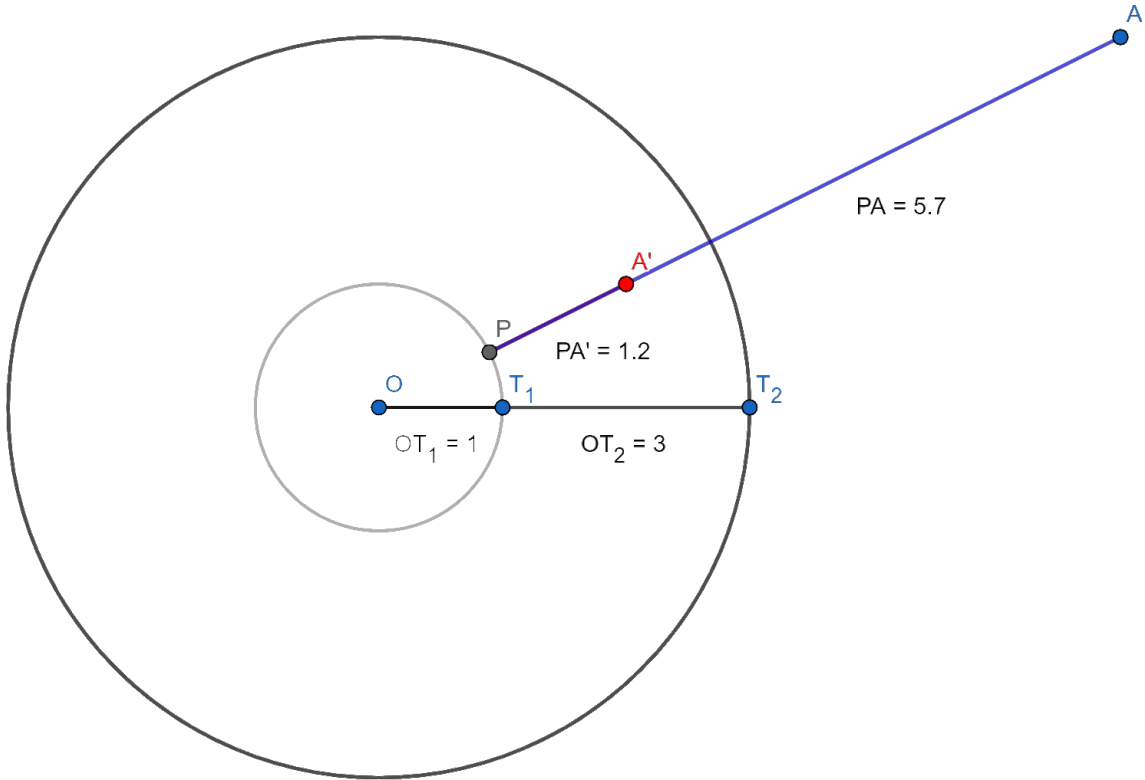
A három megvalósítandó transzformáció az eltolás, origó körüli forgatás és origótól való uniform skálázás. Ezek a műveletek nem értelmezhetők a csúcsok kimeneteként kapott távolságokon, hiszen azok csupán lokális információt hordoznak. Tehát nem lehet belőlük megállapítani, hogy milyen távolságot kaptunk volna, ha az adott csúcs által reprezentált test például el van forgatva.

Ha egy művelet eredményeként kapott összetett objektumot transzformálunk, azt úgy is elvégezhetjük, hogy az azt alkotó primitíveken már a művelet elvégzése előtt végrehajtjuk ugyanazon transzformációkat. Ezzel a problémát az egyes primitívek eltolására, forgatására vagy skálázására vezetjük vissza. Mindez pedig megoldható úgy is, hogy nem magát a primitívet transzformáljuk, hanem azt a pontot helyezzük át a transzformáció inverzének megfelelően, ahol a távolságfüggvényt kiértékeljük. Azaz nem a testet "mozgatjuk", hanem a mintavételezés pontját az elenkező irányba. Ez a megközelítés azért is hasznos, mert így új primitívek felvételekor elég azokat pusztán a legegyszerűbb formájukban az origóba helyezve bármilyen forgatás vagy skálázás nélkül megadni. Kérdés tehát, hogy a gráf feldolgozása közben hogyan határozzuk meg a végső transzformációt a primitívektől vett távolság kiszámításához attól függően, hogy az egyes primitívekre közvetlenül vagy a belső csúcsokban lévő műveletek során milyen transzformációkat alkalmaztunk.

Ennek megoldására vezessünk be egy **transzformációs vermet**. Ezt a struktúrát a bejárás közben minden egyes csúcs feldolgozásakor úgy tartjuk karban, hogy amikor egy új csúcsra lépünk, a verem legfelső eleme egy olyan  $\mathbb{R}^{3 \times 3}$ -beli mátrix legyen, amely megadja a csúcs kimenetén végrehajtandó összes transzformáció eredőjét. Ezt felhasználva tetszőleges csúcs esetén a rá vonatkozó lokálisan megadott transzformációs mátrixot az éppen a verem tetején lévő mátrixszal balról szorozva megkapjuk a csúcs által reprezentált testen végrehajtandó összes transzformációt reprezentáló mátrixot. Ha az aktuális csúcs egy primitív, akkor ennek a mátrixnak az inverzét még a kód generálásakor meghatározhatjuk. Az inverz mátrixot beleégetjük a generált GLSL kódba, ahol amikor a primitívtől vett távolságot számoljuk a csúcs kiértékelésekor, a kért pozíciót az inverz transzformációs mátrixszal szorozzuk, majd az eredményvektorban értékeljük ki a primitív távolságfüggvényét.

Skálázás esetén a transzformációs mátrix invertálása torzítást hoz be a kapott távolságban (4.6 ábra). Ha például egy testet háromszorosára méretezünk, az azt

jelenti, hogy a mintavételezés pontját fogjuk  $\frac{1}{3}$ -al skálázni. Ekkor viszont a felülettől való távolság is harmadára csökken. Ennek elkerülése érdekében minden csúcsban a visszakapott távolságot az adott csúcsban használt skálázási paraméter reciprokával szükséges szoroznunk.



**4.6. ábra.** Skálázásnál a távolságot is skálázzuk: Ha az ábrán szereplő egységsugarú kört 3 szorosára növeljük, akkor a kódgenerálás során a távolságot  $A'$  pontban fogjuk meghatározni, 1.2-t kapva. Az  $A$  pont skálázott körtől vett távolsága ezzel szemben ennek háromszorosa: 5.7.

Módosítsuk a transzformációs verem és a skálázási korrekció bevezetésével a 2. algoritmust. Az így kapott eljárás már alkalmas tetszőleges CSG gráffal megadott háromdimenziós modell GLSL távolságfüggvényének generálására. (3. algoritmus)

**3. Algoritmus** Rekurzív kódgenerálás (transzformációkkal)**Funct** TraverseGraph(root, output *<by reference>*)

```

if root is primitive then
    invTransf := invert(transformStack.top * root.transform)
    output += "inv = " + invTransf + ";"
    output += "invPos = inv * pos;"      ▷ Calculate point for sampling the sdf
    var := CreateOrReuseVariable()
    output += var + " = "
    output += GenCodeForPrimitive(root as primitive, invPos) + ";"
    output += var + " *= " + root.scale + ";"      ▷ scaling correction
    return var      ▷ return name of variable holding the result
else      ▷ root is operator
    transformStack.push(transformStack.top * root.transform)
    variables := EmptyList()
    for node ∈ root.inputs do
        variables.append(TraverseGraph(node, output))
    end for
    transformStack.pop()
    var = CreateOrReuseVariable()
    output += var + " = "
    output += GenCodeForOperator(root as operator, variables) + ";"
    output += var + " *= " + root.scale + ";"      ▷ scaling correction
    for varName ∈ variables do
        FreeVariable(varName)      ▷ free temporary variables for future reuse
    end for
    return var      ▷ return name of glsl variable containing result
end if

```

**Funct** GenerateSdf(root, output *<by reference>*)

```

output += "float sdf(vec3 pos) {"
resultVarName := TraverseGraph(root, output <by reference>)
output += "return " + resultVarName + ";"
output += "}"

```

**4.3.2. Automatikus differenciálás**

Az előbbiekben ismertettem az előjeles távolságfüggvényt kiszámító GLSL kód generálását. A deriváltak meghatározásához a 3.3.3 pontban bemutatott automatikus differenciálást használjuk,  $N$ -ed rendű háromváltozós duális számokkal. Ehhez először szükség van egy duális számokat és rajtuk értelmezett műveleteket megvalósító GLSL könyvtárra.

## Duális számok megvalósítása a GPU-n

A 3.2 ábrán bemutatott tetraéderes indexelés segítségével a duális szám komponensei egy tömbben tárolhatóak, melynek hossza az  $N$ -edik tetraéder szám (ha legfeljebb az  $N$ -edik deriváltakat szeretnénk meghatározni).

```
1 #define SIZE 10 // Nth tetrahedral number (here: 2nd)
2 struct dnum {
3     float d[SIZE];
4 };
```

### 4.1. forráskód. Duális típus GLSL-ben

A *SIZE* konstans segítségével definiálhatóak a 3.3.5 részben definiált alpműveletek. Ezek bátran implementálhatóak GLSL függvényekként, ugyanis a shader fordítás során inline-olva lesznek, tehát a függvényhívás nem fog extra költséggel járni. A 3.10 szorzás esetén hatékonysági okokból érdemes a képletben szereplő  $\binom{n}{k}$  alakú tényezőket előre kiszámítani. Az így kapott Pascal háromszög eltárolható konstansként a shader elején<sup>9</sup>. A képletben szereplő egymásbaágyazott szummákat implementálhatjuk ciklusokkal, ugyanis mivel ezek iterációinak száma konstans (*SIZE*-től függ), a fordító unroll-olni<sup>10</sup> fogja őket. Az esetszétválasztással járó műveleteket (min, max, abs, clamp) úgy valósítjuk meg duális számokon, hogy az összehasonlítást a valós részen végezzük el, az eredménybe pedig az a duális szám kerül, amelynek valós értékét választottuk. Az alpműveletek GLSL implementációja az A függelékben található.

Külön figyelmet érdemelnek a  $\mathbb{R} \rightarrow \mathbb{R}$  függvények  $\mathcal{D}_3^{(N)} \rightarrow \mathcal{D}_3^{(N)}$  megfelelői, ugyanis ezek implementációjához szükség van a megvalósítandó függvény deriváltjainak képleteire. Emiatt ezen függvényeket még CPU oldalon kell legenerálni a shader fordítás megkezdése előtt. Maga a generálási folyamat általánosítható, így új függvények bevezetéséhez elégségesse válik a deriváltak képleteinek felsorolása. A 4. algoritmus megvalósítja egy ilyen GLSL függvény legenerálását.

<sup>9</sup>OpenGL 4.3 verzió fölött támogatottak a többdimenziós tömbök is GLSL-ben, így még a tömb egydimenzióssá lapításával sem kell feltétlenül foglalkozni.

<sup>10</sup>Ezzel elkerüljük a ciklusfeltételek gyakori ellenőrzéseit, melyek egyébként a végrehajtandó utasítások nagyrészét tennék ki.

---

**4. Algoritmus**  $\mathbb{R} \rightarrow \mathbb{R}$  függvény  $\mathcal{D}_3^{(N)} \rightarrow \mathcal{D}_3^{(N)}$  megfelelőjét kiszámító GLSL kód generálása, a Faá di Bruno formula kombinatorikus alakjának alkalmazásával [4]. Az algoritmus által generált duális gyökfüggvény megtekinthető az A függelékben.

---

**Funct** GenerateChainRuleFunc(name, output *<by reference>*, derivativeFormulae)

```

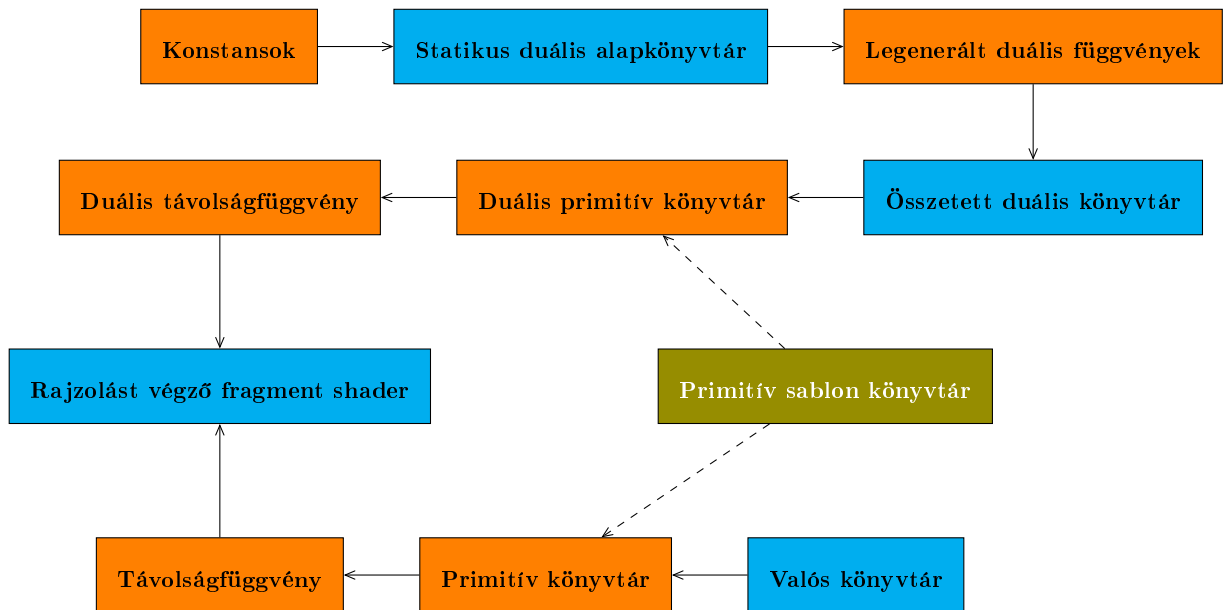
    output += "dnum " + name + "(dnum x){"
    output += "result := zero();"  ▷ a dual number with all components zeroed out
    for x = 0, ..., N do
        for y = 0, ..., N - x do
            for z = 0, ..., N - x - y do          ▷ Enumerate all possible partials
                Calculate the (x, y, z) partial derivative:
                s := {1, 2, ..., x + y + z}
                for  $\pi \in$  partitions of s do
                    output += "tmp := "
                    output += StringReplace(derivativeFormulae[| $\pi$ |], "x", "x.d[0]") +
                        ","
                    for B  $\in$  blocks of  $\pi$  do
                        dx := count numbers from range [1; x] in B
                        dy := count numbers from range [x + 1; x + y] in B
                        dz := count numbers from range [x + y + 1; z] in B
                        output += "tmp *= x.d[idx(dx, dy, dz)];"
                    end for
                    output += "result.d[idx(x, y, z)] += tmp;"
                end for
            end for
        end for
    end for
    output += "}"

```

---

## Deriváltak meghatározása

Az automatikus differenciálásnak köszönhetően a deriváltak meghatározásához elegendő a távolságfüggvényt kiszámító kódban használt lebegőpontos típusokat duális megfelelőikre, a rajtuk végzett műveleteket duális műveletekre cserélni. Mivel mind a távolságfüggvények, mind a duális könyvtár egyes részei is procedurálisan generáltak, ezért a megjelenítést végző fragment shader célszerű több fájl összefűzésével előállítani. Mindemellett a két különböző távolságfüggvényhez szükség van két külön primitív könyvtárra is. Az egyiket a sphere tracing implementáció használja és pusztán a távolság meghatározására képes, a másik emellett a deriváltakat is képes meghatározni. Utóbbit a megjelenítés során használjuk. Annak érdekében, hogy elkerüljük a primitívek távolságfüggvényeinek többszöri megadását, ezeket egy sablon könyvtárban definiáljuk, melyből külön generálható a típusok és függvénynevek lecserélésével egy duális és egy valós primitív könyvtár a két távolságfüggvény számára. A 4.7 ábrán látható, hogy a teljes fragment shader amely a megjelenítést végzi, milyen részek összefűzésével kapható.



**4.7. ábra.** A megjelenítést végző fragment shader összeállításának lépései. Az ábrán szereplő folytonos nyilak összefűzést, a szaggatottak szöveghelyettesítési generálást jelölnek.

*Kódgenerálással kapott fájl*

*Manuálisan írt fájl*

- **Konstansok** Első sorban itt állítjuk be egy makró segítségével  $N$ -et, azaz azt, hogy legfeljebb hanyad rendű deriváltakat szeretnénk meghatározni. A



duális típus definiálásakor felhasznált *SIZE* makrót is itt definiáljuk, ugyanis az  $N$ -től függ. Ezen kívül az előre generált háromszög és tetraéder számokat, valamint pascal háromszöget tartalmazza.

- **Statikus duális alapkönyvtár** Definiálja a duális típusokat, valamint a köztük értelmezett alpműveleteket.
- **Legenerált duális függvények** Vannak CPU oldalon generált függvényeink (4. algoritmus), ezeket a duális típusok és alpműveletek definiálása után fűzzük. Ilyen például a négyzetgyökvonást megvalósító *sqrt* függvény.
- **Összetett duális könyvtár** A primitívek távolságfüggvényeinek egyszerűbb megadása érdekében érdemes egyes összetettebb beépített GLSL függvények duális megfelelőit is létrehozni. Ezek megadásakor már szükség lehet CPU-n generált duális függvényekre is, ezért csak utóbbiak megadását követően definiáljuk őket. Ilyenek például a *length*, vagy *distance* GLSL függvények, melyekhez szükség van a legenerált négyzetgyökvonó függvényre.
- **Valós könyvtár** Az itt szereplő függvények egyetlen célja, hogy leegyszerűsítsék a primitív könyvtárak sablonból való szövegbehelyettesítéssel történő generálását. Ennek érdekében az operátorral elvégezhető alpműveleteket is függvényekkel helyettesítettem. Minderre azért van szükség az egyszerűsítéshez, mivel a GLSL nem támogat operátor túlterhelést, valamint a függvények túlterhelése se engedélyezett, ha a visszatérési típus különbözik.

## Primitív könyvtárak

A két primitív könyvtár tartalmazza a primitívek előjeles távolságfüggvényeit. Minden egyes primitívhez egy darab sablonfüggvényt kell megadni, melyből a duális és valós változat lesz legenerálva. A generálás a sablonfüggvényekből egyszerű szövegbehelyettesítéssel történik. Ennek egyszerűsítése érdekében a sablonokban olyan szintaxist alkalmazunk, ami könnyűvé teszi a behelyettesítést. Továbbá mivel egy változó és egy konstans máshogy adandó meg duális számként, illetve mivel a duális távolságfüggvény általánosabb, ezért a sablon könyvtárban duális sablontípusokkal dolgozunk.

**Konstansok és változók** A számítások során előfordulnak konstans és változó értékek. Ezek csak a duális esetben térnek el, hiszen az egyetlen különbséget közöttük a deriváltak jelentik.

**22. Definíció.** Legyen  $x \in \mathbb{R}$  a számításaink szempontjából konstans érték. Ekkor a duális műveletek során használt duális megfelelője  $\mathcal{D}_3^{(N)} \ni \ddot{\mathbf{x}} = D_3^{(N)}(x, 0, 0, \dots, 0)$ .

**23. Definíció.** Legyen  $\vec{\mathbf{p}} \in \mathbb{R}^3 = [x, y, z]^T$  a számításaink szempontjából változó érték, például az a térbeli pont, ahol a távolságfüggvényt és annak deriváltjait kiértékeljük. Ekkor duális megfelelője egy duális számokat tartalmazó háromdimenziós vektor:

$$\left(\mathcal{D}_3^{(N)}\right)^3 \ni \vec{\mathbf{p}}' = \begin{bmatrix} D_3^{(N)}(x, 1, 0, 0, 0, \dots, 0) \\ D_3^{(N)}(y, 0, 1, 0, 0, \dots, 0) \\ D_3^{(N)}(z, 0, 0, 1, 0, \dots, 0) \end{bmatrix}$$

```
1  _dnum_ _TEMPLATE_sphere(float radius, _dnum3_ point) {
2      return _sub_(_dlength_(point), _constant_(radius));
3  }
```

**4.2. forráskód.** *Példa: gömb sablon távolságfüggvényének megadása*

```
1  float r_sphere(float radius, vec3 point) {
2      return r_sub(r_dlength(point), r_constant(radius));
3  }
```

**4.3. forráskód.** *Példa: gömb sablonból generált valós távolságfüggvénye*

```
1  dnum d_sphere(float radius, dnum3 point) {
2      return sub(dlength(point), constant(radius));
3  }
```

**4.4. forráskód.** *Példa: gömb sablonból generált duális távolságfüggvénye*

sablon	valós megfele- lő	duális megfe- lelő
<code>_dnum_</code>	<code>float</code>	<code>dnum</code>
<code>_dnum2_</code>	<code>vec2</code>	<code>dnum2</code>
<code>_dnum3_</code>	<code>vec3</code>	<code>dnum3</code>
<code>_FUNCTION_</code>	<code>r_FUNCTION</code>	<code>FUNCTION</code>
<code>_TEMPLATE_PRIMITIVE</code>	<code>r_PRIMITIVE</code>	<code>d_PRIMITIVE</code>

**4.1. táblázat.** Sablonszövegek és behelyettesített megfelelőik.

*FUNCTION* egy tetszőleges függvény, melyre létezik valós és duális változat is a megfelelő nevekkkel.

*PRIMITIVE* egy primitív neve, melyből a két megfelelő primitív könyvtárbeli függvény generálódik.

*dnum2* és *dnum3* rendre kettő és háromdimenziós duális vektorok.

## 5. fejezet

# Architektúra

A program architektúrájának tervezésekor törekedtem a megjelenítés, a tárolás és a modell izolációjára. A program a bővíthetőség szempontjából aszimmetrikusan lett megvalósítva, azaz a hangsúly leginkább az új műveletek és primitívek egyszerű bevezethetőségére lett fektetve, szemben például újfajta bejárások megvalósításával. Várhatóan ugyanis utóbbira ritkábban kerülne sor.

Ennek megfelelően a program legfőbb architekturális egységei:

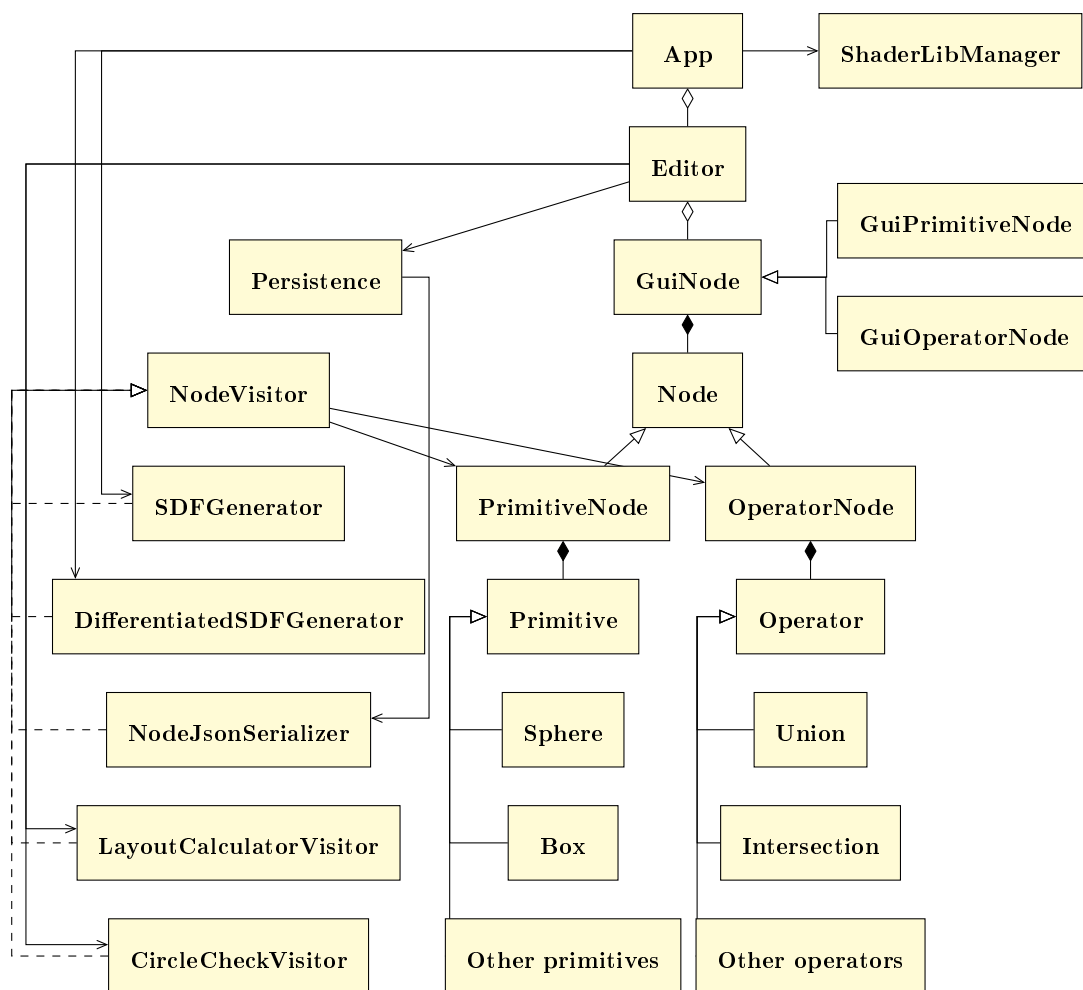
- 3D megjelenítő
- Gráfszerkesztő
- Háttér gráf reprezentáció
- Primitívek és műveletek
- Shader generátor(ok)
- JSON szerializáló és perzisztencia réteg

Ezen részek között egyes esetekben szorosabb kapcsolat van, máskor teljesen izoláltak egymástól.

A részegységek közötti kapcsolatokat az 5.1 ábra mutatja be.

### 5.1. main függvény

A DragonFly-on keresztül inicializálja az OpenGL-t és az SDL-t. Továbbá tartalmazza a *render loop*-ot, amely biztosítja a GUI rajzolás – állapot frissítés – kirajzolás ismétlődő hármását.



5.1. ábra. A program fontosabb osztályai és azok egymással való kapcsolatai.

## 5.2. App osztály – megjelenítés és vezérlés

Az **App** osztály felel a háromdimenziós megjelenítésért, melyet a DragonFly keretrendszeren keresztül OpenGL-t használva valósít meg. Ezen kívül a program különböző részeinek vezérlése is innen történik, a különböző funkciók itt érnek össze.

Az inicializálás, a shaderek betöltése, fordítás és linkelése valamint a vbo-k, vao-k feltöltése és felkonfigurálása a konstruktorban illetve az innen hívott függvényekben történik. A billentyűről érkező input kezelését a *HandleKeyDown* és *HandleKeyUp* függvények valósítják meg, kivéve a kamerakezelést, melyért a **Camera** osztály felelős.

### Fontosabb publikus metódusok

**void DrawUI()** – **ImGui** segítségével megjeleníti a szerkesztő felületet egy külön ablakban. Ebbe beletartozik a gráfszerkesztő (node editor) és a fenti menüsáv kirajzolása, továbbá az ezekben bekövetkező események kezelése. Mivel az ImGui egy *immediate mode* (azonnali módú) grafikus felhasználói felület könyvtár, ezért a bekövetkező eseményekre a GUI rajzolása közben reagálunk, a két logika nincs élesen elválasztva egymástól.

**void Update()** – Minden egyéb olyan mechanizmus megvalósítását tartalmazza, amik se a felhasználói felülettel, se az OpenGL rajzolási hívásokkal nem állnak közvetlen összefüggésben. Ide tartozik a kamera mozgatása és forgatása a billentyűzetről és egérről érkező inputok alapján, valamint a megjelenítésnél használt távolságfüggvény újragenerálása, amennyiben az ezt megelőző *DrawUI* hívás alatt a gráf módosult és az automatikus shaderfordítás be van kapcsolva.

**void Render()** – A DragonFly keretrendszer biztosít egy viszonylag magas absztrakciós szintű API-t az OpenGL fölött, melynek használatával a *Render* függvény megjeleníti a **sphere trace**-elt 3D modellt.

### Fontosabb privát metódusok

**bool isShaderGenerationPending()** – Eldönti, hogy a program jelenlegi állapotában szükséges-e a megjelenítő shader bármely részének újragenerálása. Ez egy

összetett feltétel amely függ attól, hogy be van-e kapcsolva az automatikus shader generálás és attól is, hogy milyen változások voltak. Mivel az ImGui működése miatt a shader generálást néhány képkockával el kell halasztani, ezért több helyen is hívódik.

**void GenerateShaders(std::shared\_ptr<Node> root)** – Elvégzi a shader részletek újragenerálását és az ezt követő shaderfordítást. A generálás végrehajtásához az **SDFGenerator** és **DifferentiatedSDFGenerator** osztályokat példányosítja. Az esetlegesen fellépő kivételeket kezeli és tárolja későbbi megjelenítésre a felhasználói felületen. Végezetül visszaállítja a megfelelő flag-eket, hogy ne kerüljön ismét meghívásra.

### 5.3. Editor osztály

Az **Editor** osztály a gráfszerkesztő megjelenítésért és az itt bekövetkező módosításoknak megfelelően a háttérben meghúzódó gráfrepresentáció frissítését végzi. Magának a gráfszerkesztőnek a megvalósításához az **ImGui** *node-editor* kiterjesztését használja. A gráfrepresentációt csúcsai (**Node**) a megjelenítéshez **GuiNode** példányokba vannak becsomagolva.

#### Fontosabb publikus metódusok

**void Draw()** – Az **Editor** osztály legfontosabb metódusa. Ez felel a gráfszerkesztő kirajzolásáért és manipulálásáért is. A gráfszerkesztővel való felhasználói interakció kezelése is ebben a metódusban történik. Egyes gráfszerkesztővel kapcsolatos műveleteket<sup>1</sup> más osztályból hívva szeretnénk végrehajtani, viszont a *node-editor* API-ja ezeket csak a gráfszerkesztő rajzolása közben tudja kezelni. Ezeket az **Editor** késleltetve hajtja végre a kérelem utáni első **Draw** hívásban.

**bool IsDirty()** – Ennek a függvénynek a visszatérési értéke jelzi, hogy a gráfban keletkezett-e változás. Ennek megvalósítása egy egyszerű flag-en keresztül történik.

**void ResetDirtyFlag()** – Ezt a függvényt kell meghívni a dirty flag visszaállításához minden egyes alkalommal amikor a gráfból a shadereket legeneráltuk.

---

<sup>1</sup>csúcsok kijelölése, egész gráfra fókuszálás, kijelölésre fókuszálás

## NodeHandle API

A belső reprezentációk kiszivárgásának elkerülése érdekében az **Editor**, **Persistence** és **NodeJsonSerializer** osztályok publikus függvényei nem Node referenciákat, hanem ezeket becsomagoló **NodeHandle** példányokat adnak vissza és várnak paraméterként. Az **Editor** osztály rendelkezik függvényekkel, melyek lehetővé teszik a gráfrepresentáció külső szerkesztését. Ezek a csúcsokra **NodeHandle** példányokon keresztül hivatkoznak.

**NodeHandle AddNode(...)** – Ezzel a két metódussal lehetséges külsőleg új csúcsokat hozzáadni a gráfhoz. A hozzáadandó csúcsok osztályát kívül kell megkonstruálni majd egy rá hivatkozott `shared_ptr`-t átadni a függvénynek. Ekkor létrehozásra kerül a csúcsot becsomagoló **GuiNode** osztály, majd a rá hivatkozó **NodeHandle**-el visszatér a metódus.

**void ConnectNodes(NodeHandle from, NodeHandle to)** – Összeköti a két megadott csúcsot. A „*from*” csúcs kimenetét a „*to*” csúcs első szabad bemenetével.

**void Clear()** – Letörli a gráfot.

**void SetNodePosition(NodeHandle node, ImVec2 pos)** – Ezzel a metódussal lehetséges egy csúcs gráfszerkesztőbeli pozícióját külsőleg módosítani.

**void SelectNode(s)(...)** – Ez a két metódus felel egy vagy több csúcs külsőleg történő kijelöléséért.

**void FocusContent()** és **void FocusSelection()** – Lehetővé teszik a gráfszerkesztő nézetének pozícionálását és méretezését oly módon, hogy azon az egész gráf, vagy az aktuálisan kijelölt csúcsok legyenek láthatóak.

**void AutoArrange()** – Automatikusan elrendezi a gráf csúcsait úgy, hogy azok ne legyenek átfedésben és a kapott elrendezés mellett a gráf olvasható és értelmezhető legyen. Első sorban betöltés után van használni.



**vector<NodeHandle> GetRootNodes()** – Visszaadja a gráf összes valódi gyökércsúcsát (nem a felhasználó által megjelenítésre kijelöltet). Első sorban a gráf elmentésekor jut szerephez.

**shared\_ptr<Node> GetCurrentRoot()** – Visszaadja a felhasználó által gyökérnek kijelölt csúcsot. A megjelenítés során csak az ehhez a csúcshoz tartozó részfa kerül kirajzolásra. Ez a függvény kivételesen nem **NodeHandle**-t ad vissza<sup>2</sup>, ugyanis a felhasználás során nincs többé szükség a **GuiNode**-ra.

## Fontosabb privát metódusok

**... CreatePrimitiveNode(...)** és **... CreateOperatorNode(...)** – Egy paraméterként kapott **PrimitiveNode**-ot vagy **OperatorNode**-ot csomagolnak be rendre **GuiPrimitiveNode**-ba és **GuiOperatorNode**-ba. Ez magába foglalja a szükséges csúcsok létrehozását az *imgui node editor* oldalán is.

**void SetRoot(shared\_ptr<GuiNode> newRoot)** – Új gyökércsúcsot állít be a megjelenítés szempontjából. A **GuiPrimitiveNode** és **GuiOperatorNode** példányok hívhatják egy konstruálásukkor átadott funktoron keresztül.

**... CreatePinForNode(...)** – Egy megadott **GuiNode**-hoz hoz létre egy új pint az *imgui node editor* oldalán, valamint a program oldalán is. Egy **GuiOperatorNode** példány is hívhatja egy átadott funktoron keresztül, ha elfogytak a szabad input pinjei.

**bool AllowConnection(Pin& from, Pin& to)** – Két pin összeköthetőségét dönti el. Figyelembe véve a kialakuló körök tiltását és a pinek típusait is.

**void CreateLink(shared\_ptr<Pin> from, shared\_ptr<Pin> to)** – Két pin összekötését végzi el, frissítve minden szükséges kapcsolódó változót. ha korábban bármelyik pin foglalt volt, akkor az oda csatlakozó él törlésre kerül.

**void DeleteLink(Link& link)** – Töröl egy linket a gráfszerkesztőben és a háttérreprezentációban is.

---

<sup>2</sup>ha **NodeHandle**-t adna akkor a **GuiNode** osztálynak meg kellene jelenjen az **App** vagy az **SDFGenerator** osztályok egyikében.

**void DeleteNode(GuiNode& node)** – Törli a megadott csúcsot a gui-ról és a háttérreprezentációból is. A hozzá kapcsolódó élek is törlésre kerülnek.

**void CopyToClipboard(ClipBoard& cb)** – A gráfszerkesztő aktuális kijelölését másolja le a megadott vágólapra, mély másolással.

**void PasteClipboard(ClipBoard& cb)** – Beilleszti a gráfszerkesztőbe a megadott vágólap tartalmát.

## 5.4. NodeVisitor osztály – Gráfbejárások

A program több részén is a CSG gráf feldolgozására van szükség, melyet mélységi bejárással érdemes megtenni. Ezek implementációjakor a látogató tervezési mintát alkalmaztam a csúcsok és a bejárások egymástól való elkülönítése érdekében. Minden ilyen bejárást egy külön osztály végez el, amely a **NodeVisitor** osztály leszármazottja. Maga a **NodeVisitor** osztály pusztán a bejárásokhoz szükséges két () operátor túlterhelést tartalmazza definíció nélkül. Ezek abban térnek el, hogy a paraméter primitív vagy műveleti csúcs.

### Bejárások általános működése

A program több mélységi bejárást is használ különböző célokra. Ezek implementációja mind oly módon történik, hogy a bejárást végző osztály implementálja a () operátor két változatát: *PrimitiveNode* és *OperatorNode* paraméterekre. Ezek elvégzik a paraméterként kapott csúcs feldolgozását. A továbbhaladáshoz a csúcs gyerekein meghívják a minden **Node**-on implementált **visit** metódust, paraméterként a saját magukra mutató pointert átadva. Ekkor az adott *PrimitiveNode* vagy *OperatorNode* a paraméterként kapott bejáróobjektumon meghívja a bejárást folytató () operátor, paraméterként saját magát átadva. Így biztosítva van a mélységi bejárás rekurziója. Ez az architektúra lehetővé teszi, hogy mind új bejárásokat mind új primitíveket vagy műveleteket felvehessünk anélkül, hogy a többi osztályon módosítani kellene. Ugyanakkor ha egy új csúcstípust akarunk bevezetni (ami nem primitív és nem is művelet), akkor arra minden bejárást fel kell készíteni.

## 5.5. Persistence osztály – mentés és betöltés

Ezen osztályon keresztül történik a mentés és betöltés. Az osztály két statikus függvénnyel rendelkezik, melyek a **NodeJsonSerializer** osztályra támaszkodva megvalósítják a szerializációt és deszerializációt.

### 5.5.1. Szerializálás és deszerializálás

A csúcsok szerializálásához és deszerializálásához az *nlohmann json* könyvtárat használtam fel. A művelet rekurzívan történik a **NodeJsonSerializer** osztályban implementált mélységi bejárás segítségével.

Az alkalmazott **json** formátumban a gráf csúcsai a gyökértől kezdve egymásba ágyazva jelennek meg. Minden csúcs első attribútuma megadja, hogy az operátor vagy primitív és ezeken belül melyik változat. Ezt követik a közös attribútumok és értékeik, mint az eltolás, forgatás, skálázás és az offset. Ezek után az adott csúcs konkrét típusának megfelelő extra attribútumok kerülnek felsorolásra értékeikkel együtt. Végezetül az operátorok esetén az „*inputs*” mező egy listát tartalmaz, melybe beágyazva kerülnek a csúcs bemeneteit képező csúcsok és adataik. Ha a gráf erdő, akkor több gyökércsúcs is van, ezért a gyökércsúcsok egy listában vannak megadva.

### 5.5.2. NodeJsonSerializer osztály

**static string Serialize(vector<NodeHandle> roots)** – Az 5.4 bekezdésben leírtak szerint valósít meg egy mélységi bejárást a gráfon minden gyökérből külön-külön, majd az eredményeket egy json tömbbe helyezi. A bejárás során fenntart egy vermet, melynek tetején mindig az adott szinten testvér csúcsokat tartalmazó json tömb van. Az egyes csúcsok attribútumainak szerializálásához a **PrimitiveNode** és **OperatorNode** osztályok **SerializeToJson** metódusát hívja, melyet minden primitív és művelet esetén implementálni kell.

**static void DeserializeInto(Editor& nodes, string jsonString)** – A kapott json parsolását követően végighalad az erdő fáin és mindegyik gyökerén a **LoadTree** privát metódust hívja a deszerializáláshoz.

**static NodeHandle LoadTree(Editor& nodes, ordered\_json& root)** – Rekurzívan feldolgozza a kapott json-t és az **Editor** osztály **Nodehandle API**

jának felhasználásával feltölti azt. Az egyes csúcsok példányosításához template-ek segítségével megadott típus listát használ, így új primitívek vagy műveletek bevezetésére is fel van készítve. A példányosításhoz csupán a primitív vagy művelet osztályának definiálásakor kell egy *ordered\_json* paraméterű konstruktort megadni, ami kiolvassa a json-ból az adott csúcs attribútumait, de nem kell a csúcs esetleges gyerekeivel foglalkozzon.

## 5.6. Shader generálás

### 5.6.1. SDFGenerator és DifferentiatedSDFGenerator

A két távolságfüggvény GLSL kódját előállító gráfbejárásokat az **SDFGenerator** és a **DifferentiatedSDFGenerator** osztályok valósítják meg. Előbbi a valós távolságfüggvényt, utóbbi a duálisat generálja. A két osztály nagyrészt egyezik, leginkább a generálás során alkalmazott függvény és típusnevekben van különbség. A későbbiekben érdemes lehet a két osztály összevonása. A generálásra alkalmazott algoritmus a 4.3 részben került bemutatásra.

A 4.7 ábrán látható módon a végső megjelenítést végző fragment shader ezen kívül még több generált összetevőből is áll. Az ezeket előállító kód a **ShaderLibManager** osztályban található. A végső shader összeállítása a részekből az **App** osztály **GenerateShaders** metódusában történik.

### 5.6.2. ShaderLibManager osztály

**void GeneratePrimitiveLibs()** – Beolvassa *primitives.frag* fájlt, amely tartalmazza a primitívek távolságfüggvényeinek sablon definícióit. Ezekből szöveg helyettesítéssel a 4.1 táblázat szerint legenerálja a valós és duális távolságfüggvényeket, melyeket elment két külön fájlba (*primitives\_real.frag* és *primitives\_dual.frag*).

**string GenerateFromTemplate(string str, bool dual)** – Elvégzi a szöveg helyettesítést. Ebben a függvényben van megadva a helyettesítési tábla is, melyet a későbbiekben célszerű lenne egy külön fájlba kiszervezni.

**string GenerateConstants(int derivativeOrder)** – Legenerálja az adott rendű deriváltakhoz használt duális műveletek szükséges konstansait tartalmazó shader részletet. Ezen kívül néhány beállítást megadó makrót is belegenerál a kapott kódba.

**string GenerateChainRuleFunc(string name, vector<string> func, int order)** – A 4. algoritmus szerint legenerálja egy  $\mathbb{R} \rightarrow \mathbb{R}$  függvény  $\mathcal{D}_3^{(N)} \rightarrow \mathcal{D}_3^{(N)}$  megfelelőjét. Paraméter a függvény neve, és a saját és deriváltjainak képleteit tartalmazó vektor. A képlet megadásakor valós GLSL függvények használhatóak, a változó  $x$ -szel jelölendő.

## 5.7. GuiNode és Node típusok

Az ImGui-t érintő funkcionalitás elszeparálása végett minden gráfszerkesztőbeli csúcs két részből áll: egy **PrimitiveNode** vagy **OperatorNode** típusúból és egy ezt becsomagoló **GuiPrimitiveNode** vagy **GuiOperatorNode** objektumból. Utóbbi tartalmazza a megjelenítéshez szükséges logikát, előbbi a gráfbejárások során szükséges információkat. Azaz lényegében a gráfot a nézet oldalon a **GuiNode**-ok, a modell oldalán a **Node**-ok reprezentálják.

### 5.7.1. „Típusleíró” osztályok: Primitive és Operator

A konkrét primitívra vagy műveletre vonatkozó információkat külön **Primitív** és **Operator** „típusleíró” osztályokba szerveztem ki. Ez a modularitás lehetővé teszi, hogy új primitív vagy művelet bevezetésekor pusztán a legszükségesebb információkat kelljen megadni, mindössze egy fájl módosításával. A „típusleíró” osztályoknak a következő információkat kell biztosítani:

- Egy üres konstruktort és egy olyat, ami egy **ordered\_json&** értéket kap paraméterül és az attribútumokat az ebben tárolt értékek szerint tölti fel.
- **std::string GetName()** – Megadja a primitív nevét. Ez fog megjeleníteni a típus választóban a primitív csúcsokon, valamint az elmentett gráf json-jában is.
- **bool NodeEditorDraw()** – Egy függvény ami kirajzolja a szerkeszthető mezőket (attribútumokat) a gráfszerkesztő csúcsán. A becsomagoló **GuiNode**

osztálpéldány fogja hívni amikor kirajzolódik. Akkor tér vissza igazzal, ha valamelyik érték módosult.

- **bool NodeEditorPopup()** – Hasonló célt szolgál, felugró ablakok rajzolására. (Ezeket a kirajzolás más pontján kell megjeleníteni.)
- **SaveToJson(ordered\_json& json)** – Egy függvény ami a megadott json objektumba elmenti az attribútumokat.
- **ostream& GenerateShader(ostream& code, string sampleCoordVarName)** – Egy függvény ami a megkapott output stream-be beleírja, hogy hogyan kell a primitív távolságfüggvényét meghatározni. Ennek megadásakor a primitív távolságfüggvényét hívó kódot érdemes generálni, a csúcs paramétereivel. A kiértékelés pontját a *sampleCoordVarName* változóban megadott változónevű GLSL változó tárolja. A hívott primitív függvény neve legyen a sablonban megadott, a `"_TEMPLATE_"` prefix nélkül. Azaz például `_TEMPLATE_cylinder` sablonfüggvény esetén a henger **GenerateShader** metódusában csak egyszerűen *cylinder*-t használunk. Fontos, hogy ez a függvényhívás a legelején legyen a kódnak, szóköz vagy whitespace nélkül.
- **unique\_ptr<Primitive> clone()** – Egy másoló függvény. (unique\_ptr miatt kell)

## 5.8. Bővítés új primitívekkel

A program architektúrája lehetővé teszi, hogy új primitíveket viszonylagos egyszerűséggel felvehessünk. Egy új primitív hozzáadásához például az alábbi művelet-sort kell végrehajtani:

### Normálvektorával adott sík (féltér) hozzáadása

- Először is szükség van a primitív távolságfüggvényére. Ez a jelenlegi példa esetén  $\text{sdf}(\vec{p}) = \vec{p} \cdot \vec{n} + h$  ha  $\vec{n}$  a normálvektor és  $h$  egy  $\vec{n}$  menti eltolás.
- Győződjünk meg róla, hogy a távolságfüggvényben használt összes függvényhez (például egy gyökfüggvény vagy szinusz) generálunk duális megfelelőt. Ez jelenleg az **App:GenerateShaders** függvényen belül történik. Ha valamelyik

hiányzik, akkor adjuk hozzá a deriváltjainak képleteit megadva. Ebben az esetben a **ShaderLibManager::GenerateFromTemplate** függvényen belül is egészítsük ki a név táblát. Az első oszlop a template név, a második a valós, a harmadik a duális megfelelő neve.

- Adjuk hozzá a primitív távolságfüggvényének sablonját a *primitives.frag* fájlhoz:

```
1 _dnum_ _TEMPLATE_plane(vec3 n, float h, _dnum3_ point) {  
2     return _add_(_ddot_(point, _constant3_(n)), _constant_(h));  
3 }
```

### 5.1. forráskód. Féltér távolságfüggvénye

- Hozzunk létre egy új osztályt **Plane** néven. Legyen a **Primitive** osztály lezármazottja. Privát adattagként vegyük fel a primitív attribútumait ( $\vec{n}, h$ ).
  - Definiáljuk felül **GetName()**-et, térjen vissza „plane”-vel. Ez a név fog megjeleníteni a gráfszerkesztő primitív választásakor, valamint ez a típus név lesz használva (de)szerializáláskor is.
  - Definiáljuk felül **GenerateShader**-t. Itt azt kell megadnunk, hogy a távolságfüggvény generálásakor hogyan hívódjon meg a fent definiált primitív távolságfüggvény.

```
1 std::ostream& Plane::GenerateShader(std::ostream& code,  
    std::string sampleCoordVarName)  
2 {  
3     return code << "plane(" << n << ", " << h << ", " <<  
        sampleCoordVarName << ')';  
4 }
```

- Definiáljuk felül **NodeEditorDraw**-t. Itt kell kirajzolni az ImGui-ra az osztály extra attribútumait ( $\vec{n}, h$ ). Ha nincs ilyen akkor nem szükséges felüldefiniálni. Amennyiben a csúcs szeretne valamilyen popup menüt is kirajzolni, akkor azt a **NodeEditorPopup** metódus felüldefiniálásával tegye meg.

```
1 bool Plane::NodeEditorDraw()  
2 {  
3     bool changed = false;
```

```
4   if (ImGui::InputFloat3("n", &n[0], 3,
    ImGuiInputTextFlags_EnterReturnsTrue)) {
5       changed = true;
6       n = glm::normalize(n);
7   }
8   return changed || ImGui::InputFloat("h", &h);
9 }
```

- A (de)szerializáláshoz adjuk meg **SaveToJson** metódust és azt a konstruktor, amelyik json-t vár paraméterül.

```
1 Plane::Plane(ordered_json& json)
2 {
3     json.at("n").get_to(n);
4     json.at("h").get_to(h);
5 }
6
7 void Plane::SaveToJson(ordered_json& json)
8 {
9     json["n"] = n;
10    json["h"] = h;
11 }
```

- Végül definiáljuk felül a **clone** függvényt is:

```
1 std::unique_ptr<Primitive> Plane::clone()
2 {
3     auto copy = std::make_unique<Plane>();
4     *copy = *this;
5     return copy;
6 }
```

- A *Primitive.h* elején található **PRIMITIVES** makróhoz adjuk hozzá az új primitív osztály nevét. (**Plane**)

**Új műveletek hozzáadása** Amennyiben a programot új műveletekkel szeretnénk bővíteni, azt a primitívekhez hasonlóan tehetjük meg. Ez úttal azonban az **Operator** osztályból kell leszármazni.



## 6. fejezet

# Tesztelés

Számítógépes grafikai alkalmazások esetén az automatizált egységtesztelés kevésbé megbízható és megvalósítható a gyakorlatban. Ennek egyik legfőbb oka, hogy a GPU-n végzett számítások során nincs lehetőség köztes eredmények egyszerű vizsgálatára, lényegében csak a végeredményként kapott pixelekhez van hozzáférésünk. Ezek viszont jellemzően valamilyen közelítő lebegőpontos számítás eredményei. Mindezt tovább nehezíti, hogy a számítógépes grafikában az elérni kívánt eredmény nem határozható meg olyan egyértelműen mint például ha a feladatban diszkrét mennyiségekkel dolgoznánk. Gyakran megelégszünk azzal, ha az eredmény kellően hihető és esztétikus. A dolgozatban árnyaláshoz használt Phong [1] modell sem fizikai alapokon nyugvó korrekt és pontos árnyalást ad, csak egy olyat ami a felhasználó számára a valósághoz eléggé hasonlóan néz ki ahhoz, hogy hihető legyen.

Automatikus egységtesztelésnél felvetődne a kérdés, hogy a kapott eredményt mivel hasonlítsuk össze? Az egyetlen rendelkezésre álló minta a program által kiszámított szemmel validált kimeneti ábra. Ugyanakkor ha ezt használjuk akkor a program eredményét a saját magával hasonlítjuk össze<sup>1</sup>.

Ezen okokból az elkészült szoftver tesztelését manuálisan végeztem, a teszteket és eredményüket tesztelési jegyzőkönyvben rögzítve.

---

<sup>1</sup>Mindez legfeljebb arra alkalmas, hogy a szoftver egyes verziói között vizsgáljuk, hogy fellépett-e valamilyen új hiba. Viszont itt is óvatosnak kell lenni, direkt egyezés helyett valamilyen metrikát kellene venni a két kép különbségén, hiszen néhány pixel minimális eltérése még nem tekintendő hibának.

## 6.1. Tesztelési jegyzőkönyv

Teszteset	Leírás	Eredmény
Program elindítása	A program megnyitásakor üres gráf fogad, a megjelenítőben csak a tengelyek és az irányok látszanak. A felhasználói felület helyesen jelenik meg.	✓
Új csúcs 1	A gráfszerkesztő üres részére jobb egérgombbal kattintva a menü megjelenik, ott mind a primitív mind az operátor csúcsot kiválasztva a csúcs beszúrásra kerül.	✓
Új csúcs 2	Ha egy meglévő csúcs egyik kimeneti pinjéből élet húzunk és azt a gráfszerkesztő háttere felett engedjük el akkor egy új műveleti csúcs kerül beszúrásra ebben a pontban és az él az első bemenetéhez kerül bekötésre. Ha bemeneti pinből húzzuk az élt akkor primitív csúcs kerül beszúrásra.	✓
Összekötés 1	Egy csúcs szabad kimenetéből élt húzva egy másik csúcs szabad bemenetébe a kapcsolat létrejön és ez a generált shaderben is megjelenik.	✓
Összekötés 2	Ha a kiindulási pin foglalt, akkor az onnan korábban kiinduló él törlődik, az új él létrejön.	✓

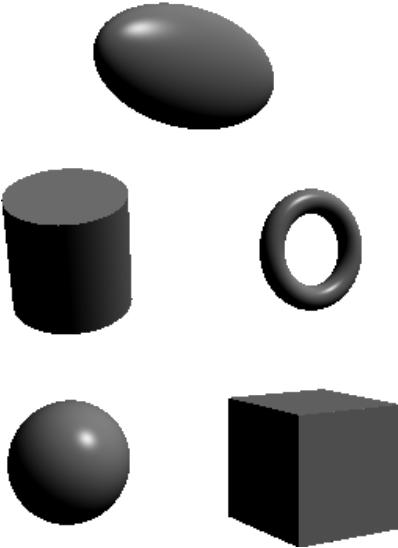
Összekötés 3	Ha a célpin foglalt, akkor az oda korábban befutó él megszűnik, az új él létrejön.	✓
Összekötés 4	Ha az új él kört okozna a gráfban akkor még az egérgomb felengedése előtt prior színt kap, a felengedés után nem kerül létrehozásra.	✓
Összekötés 5	Nem lehet két kimeneti pint vagy két bemeneti pint összekötni.	✓
Összekötés 6	Ha egy műveleti csúcs összes bemeneti pin-je foglalttá válik akkor egy új szabad bemeneti pin is létrejön az összekötéskor.	✓
Összekötés 7	Ha a különbség műveleti csúcs esetén az összekötések során felcseréljük két él sorrendjét akkor az eredményben is látszik a sorrend megváltozása.	✓
Összekötés 8	Az él helyesen jelenik meg, a két vége a két összekötött pin-nél van.	✓
Él törlése	Ha egy élre kattintva kijelöljük majd a delete gombot lenyomjuk akkor az él törlődik.	✓
Csúcs törlése 1	Ha egy csúcsot kijelölünk majd delete-el törlünk akkor törlődik és az összes hozzá kapcsolódó él is törlődik.	✓

Csúcs törlése 2	Ha az éppen kitörölt csúcs a gyökér akkor egy másik csúcs válik azzá.	✓
Csúcsok törlése 1	Ha egyszerre több csúcs is ki van jelölve a delete lenyomásakor akkor mindegyik törlődik és az összes olyan él is amelynek legalább egyik vége ezek csúcsok között volt.	✓
Csúcsok törlése 2	Ha a csúcsok között volt a gyökér is, akkor a megmaradt csúcsok egyike lesz az új gyökér.	✓
Gyökér kijelölése	Ha egy csúcs jobb felső sarkában a pöttyre kattintunk akkor az lesz az új gyökércsúcs, az eddigi gyökér megszűnik annak lenni. Az aktuális gyökér helyesen van jelölve.	✓
Csúcs mozgatása	Egy csúcsot a fejlécénél megfogva az egérrel lehet mozgatni.	✓
Csúcsok mozgatása	Ha több csúcs van kijelölve és az egyiket a fejlécénél fogva mozgatjuk akkor a többi is vele mozog.	✓
Gráfszerkesztő navigáció	A szerkesztőablakban a jobb egérgombot lenyomva tartva az egér mozgatásával lehet navigálni, a görgővel nagyítani és kicsinyíteni.	✓

Ugrás kijelölésre	Ha a <i>View</i> $\rightarrow$ <i>Jump to selection</i> gombra kattintunk akkor a szerkesztő nézete a kijelölt csúcsokra ugrik.	✓
Ugrás egész gráfra	Ha a <i>View</i> $\rightarrow$ <i>Fit entire graph</i> gombra kattintunk a szerkesztő nézete úgy módosul hogy ez egész gráf látható legyen.	✓
Másolás- Beillesztés 1	Ctrl+C lenyomásával az aktuálisan kijelölt csúcsok a köztük lévő élekkel együtt lemásolódnak a vágólapra. Ctrl+V hatására a vágólap tartalma beillesztődik.	✓
Másolás- Beillesztés 2	A beillesztett csúcsok pozíciója követi a nézetet.	✓
Másolás- Beillesztés 3	Csak a vágólapra került csúcsok közötti élek másolódnak le, de azok helyesen.	✓
Másolás- Beillesztés 4	Ha a vágólap üres, beillesztéskor nem történik semmi.	✓
Másolás- Beillesztés 5	A vágólap mély másolatot tartalmaz ( <i>deep copy</i> ). Ha a lemásolt csúcsokat a beillesztés előtt töröljük, akkor is beillesztésre kerülnek.	✓

Másolás- Beillesztés 6	A vágólap tartalma akkor is megmarad ha új gráfot kezdünk vagy betöltünk egy másikat.	✓
<p><b>Perzisztencia tesztek.</b> Mindegyik teszt során a gráfot elmentettem majd a program újraindítását követően betöltöttem és ellenőriztem, hogy ugyanazt kapjuk vissza. Ezen kívül az elmentett json fájlok validálását is elvégeztem<sup>2</sup>. A kapott fájlok a <i>tests</i> mappában találhatóak.</p>		
Mentés-Betöltés 1	Egy primitív csúcs elmentése, az alapértelmezettől különböző értékekkel.	✓ <i>single_box.json</i>
Mentés-Betöltés 2	Egy műveleti csúcs elmentése, az alapértelmezettől különböző értékekkel.	✓ <i>single_sunion.json</i>
Mentés-Betöltés 3	Egy primitív és egy műveleti csúcs összekötve.	✓ <i>cyl_union.json</i>
Mentés-Betöltés 4	Erdő (összes primitív).	✓ <i>forest.json</i>
Mentés-Betöltés 5	Közepesen összetett gráf (figura fej).	✓ <i>head.json</i>
Mentés-Betöltés 6	Összetett gráf.	✓ <i>logo.json</i>

<sup>2</sup><https://jsononline.net/json-validator>

Mentés-Betöltés 7	Üres gráf.	✓ <i>empty.json</i>
Mentés másként 1	A <i>Save as</i> gomb mindig rákérdez a fájl-névre, a <i>Save</i> gomb csak akkor ha a gráf még nem volt elmentve.	✓
Mentés másként 2	Ha a <i>New</i> gomb segítségével új gráfot kezdünk akkor a <i>Save</i> gomb rákérdez a fájl-névre.	✓
Mentés másként 3	Ha a <i>Open</i> gombbal betöltöttünk egy gráfot akkor a <i>Save</i> ebbe ment.	✓
<p><b>Megjelenítésre vonatkozó tesztek</b> A shader generálást és a modell megjelenítését manuálisan, vizuálisan ellenőriztem. A megjelenítésénél az approximált és az automatikus differenciálással kapott normálvektorokkal kapott eredményt is összehasonlítottam, a két eredmény hasonlósága mutatja, hogy a két eljárás implementációja nagy valószínűséggel helyes. Az ábrákon az automatikus differenciálással kapott kép látható.</p>		
Primitívek Eltolás Forgatás Unió		✓ <i>primitives.json</i>

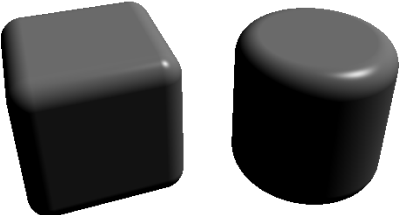

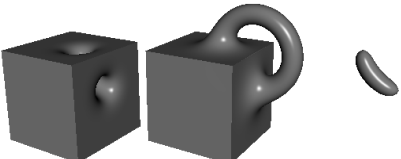
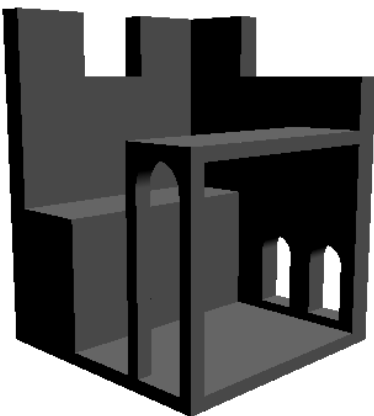
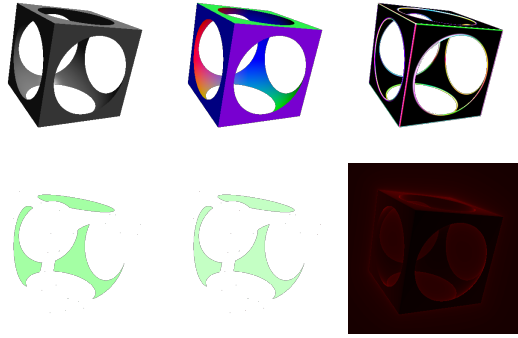
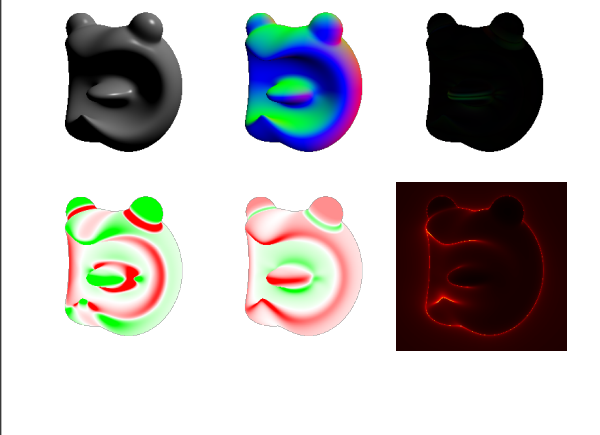
Offset		✓ <i>offset.json</i>
Unió Különbség Metszet		✓ <i>operators.json</i>
Simított: Különbség Unió Metszet		✓ <i>smoothops.json</i>
Összetett alakzat		✓ <i>logo.json</i>
Kocka és gömb különbsége: Phong árnyalás Normálvektorok Approx. hibák Gauss-görbület Középgörbület Lépésszám		✓ <i>cube_sphere.json</i>



Figura: Phong árnyalás Normálvektorok Approx. hibák Gauss-görbület Középgörbület Lépésszám		✓ <i>head.json</i> <sup>3</sup>
--	--	------------------------------------

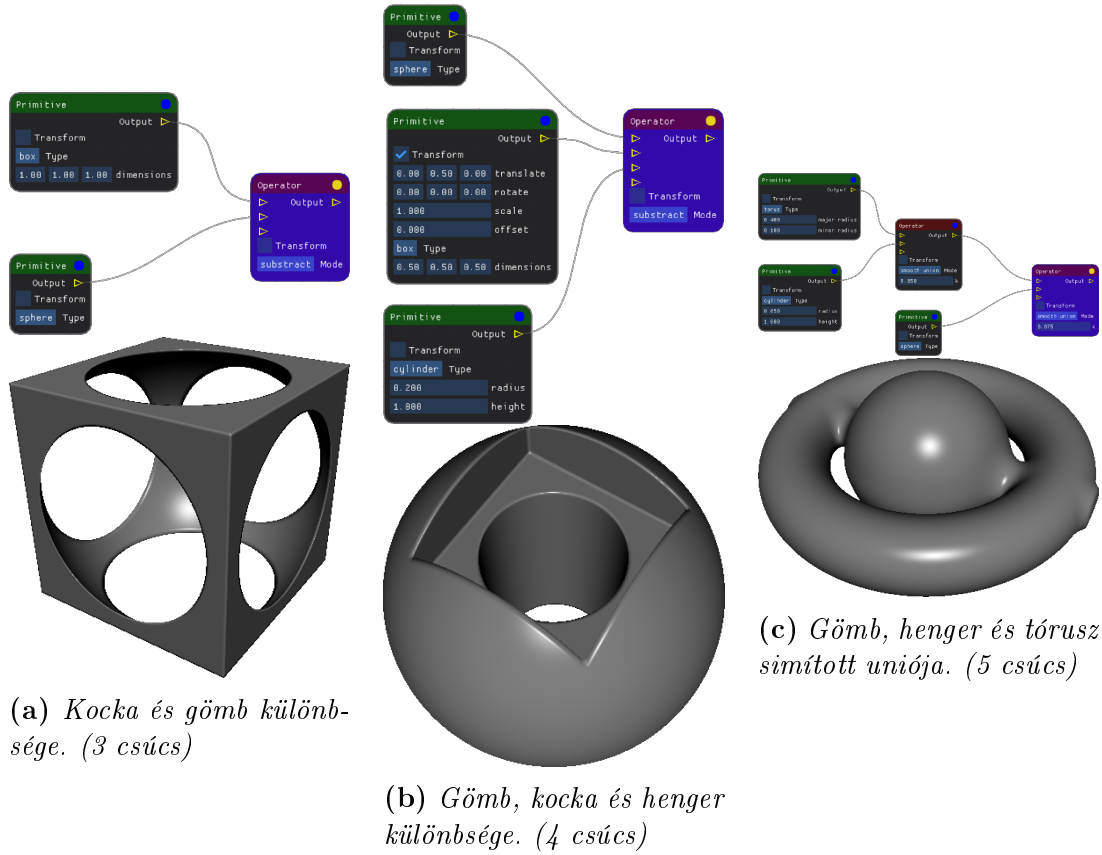
## 6.2. Problémák

### 6.2.1. Hardveres limitációk

A napjainkban használt GPU architektúrák mellett nem minden szintaktikailag helyes shaderprogram futtatható. Ennek oka abban rejlik, hogy a memória felépítése miatt az egy shader által használható változó és konstans memóriamennyiség limitált. Továbbá ha a program túl sok utasításból áll, az szintén problémákhoz vezethet. A driver részét képező GLSL fordító ezen eseteket felismeri és amennyiben nem tudja a memóriát megfelelően kiosztani az egyes változóknak, linkelési hibát jelez. Sajnos a duális műveletek bonyolultsága, valamint a magasabb rendű duális számok esetén a megnövekedett memóriaigény miatt az ezen dolgozatban bemutatott algoritmusok kimeneteként kapott GLSL kód esetén ez a fordítási hiba gyakran fellép.

Az alábbiakban néhány példa segítségével illusztrálom ezeken limitációkat a gyakorlatban (6.1 ábra). A tesztekhez egy **NVIDIA GeForce GTX 1660 Ti mobile (6GB)** dedikált, valamint egy **Intel UHD Graphics 630** integrált videokártyát használtam.

<sup>3</sup>Megjegyzés: Itt nvidia gpu-n hibás eredményt kapunk, vagy nem fordul a shader.



**6.1. ábra.** CSG gráfok és az általuk reprezentált 3D modell. Ezen modellekkel teszteltem a különböző GPU-kon való fordíthatóságot.

derivált	a (3)	b (4)	c (5)
0.	✓	✓	✓
1.	✓	✓	✓
2.	✓	✗	✗
3.	✗	✗	✗

**NVIDIA GeForce GTX 1660**

**Ti**

Driver: 512.15

derivált	a (3)	b (4)	c (5)
0.	✓	✓	✓
1.	✓	✓	✓
2.	✓	✓	✓
3.	✓	✓	✓

**Intel UHD Graphics 630**

Driver: 30.0.101.1340

**6.2. táblázat.** A kapott shader fordíthatósága. (Win 10 64 bit, 21H1)

### Kísérletek javításra

A kapott shaderek *NVIDIA GPU*-n való fordíthatóságának javítására az alábbi kísérleteket tettem:

- Megpróbálkoztam a duális szám reprezentációjának átalakításával, az értékek tömb helyett külön-külön mezőben való tárolásával. Az emögötti motiváció az volt, hogy ez a fordító számára nagyobb felxibilitást tesz lehetővé esetleges optimalizációk végrehajtására. A kapott shader ugyan lefordult, de néhány képkocka kirajzolása után összeomlott a driver. Sajnos ez egy előforduló jelenség amikor az operációs rendszer által is megjelenítésre használt GPU-n teljesítményigényes számításokat végzünk. ✗ *Sikertelen*.
- A shader komplexitásának csökkentése érdekében megpróbálkoztam a megjelenítés több lépésre bontásával, a duális számítások és a sphere tracing implementáció, valamint a végső árnyalás külön-külön shaderekbe mozgatásával. A tapasztalat azt mutatja, hogy a duális műveletek már önmagukban is túl sok temporális változót igényelnek. ✗ *Sikertelen*.
- Megpróbálkoztam a duális számítások compute shader-be történő áthelyezésével, annak reményében, hogy egy általános számításokra szolgáló compute shaderben kevesebb ismeretlen limitáció áll fent, mint egy grafikai célú, a grafikus szerelőszalagba szorosabban beépülő fragment shader esetén. Mindezt a több lépésre bontással együtt is kipróbáltam. Az első pontban említetthez hasonló módon itt is összeomlást tapasztaltam. ✗ *Sikertelen*.

#### 6.2.2. Lassú shader fordítás

A GPU hardver sajátosságai miatt a GLSL fordító jellemzően agresszív optimalizációkat hajt végre. Ebbe beletartozik függvények inline-olása és gyakran bonyolult ciklusok kibontása is. Mindez jelentős futásidő igényt eredményez, különösen komplexebb shaderek esetén. Jelen esetben ez úgy mutatkozik meg, hogy bekapcsolt duális könyvtár mellett, a shaderfordítás időtartama több másodperctől hosszú percekig terjedhet.

### 6.2.3. Driver összeomlások (Nvidia)

Elsősorban nvidia GPU-n figyeltem meg összeomlásokat, melyek az nvidia opengl driver-ből származnak (nvogl64.dll). Ezek okai valószínűleg a túl komplex shaderekre vezethetők vissza, kizárólag az erőforrásigényes automatikus differenciálás mellett fordultak elő.

## 6.3. Teljesítmény tesztek

A számítógépes grafikai eljárások egyik fontos jellemzője, hogy elvégezhetőek-e elég gyorsan ahhoz, hogy valós időben megjeleníthetők legyenek. Az eme dolgozatban tárgyalt módszer kétségkívül magas számításigénnyel bír, ahogy ez a mérési eredményeken is látszani fog. Ugyanakkor egyszerűbb szinterek esetén megfelelő teljesítményű videokártya mellett képes volt valós idejű sebességet produkálni. Mindez azt jelenti, hogy minden bizonnyal lehetséges lenne további optimalizációkkal, esetleg az általánosítás csökkentésével komplexebb szinterek esetén is valós idejű futásidőt elérni, ugyanakkor ennek megvalósítására már nem állt rendelkezésre elég idő a dolgozat leadása előtt. Komoly akadály volt még a mérések során, hogy a számomra elérhető dedikált Nvidia GPU driver GLSL fordítója a komplexebb szinterek shade-reit elutasítja, sejtetően a 6.2.1 fejezetben leírt okokból.

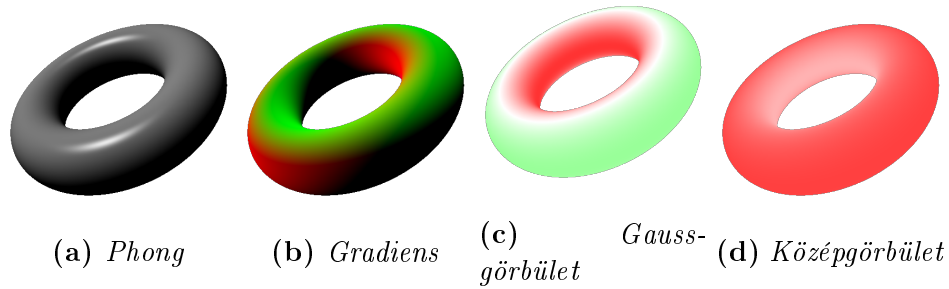
A tesztelés során négy árnyalást vizsgáltam meg több modell esetén.

- **Phong** Referenciának választottam egy a valós idejű grafikában gyakran használt árnyalást, amely egy egyszerű megvilágítást modellez. Az árnyalás kiszámításának módja a Phong árnyalásban használt modell [1], azzal az eltéréssel, hogy mivel itt nem testhálós a reprezentáció, a normálvektorok meghatározása máshogy történik. Utóbbit itt differenciahányadossal approximáltam, ugyanis ez az elterjedt és kevésbé számításigényes megoldás rá, így szolgálhat az összehasonlítás alapjául.
- **Gradiens** A távolságfüggvény gradiense, a felület pontjaiban, melyet normálvektorként használhatunk árnyaláskor. Az ábrákon a normalizált gradiens vektor  $(x, y, z)$  koordinátáit a megjelenített szín  $(r, g, b)$  csatornáiba kódoltam.
- **Gauss-görbület és Középgörbület** A 17. és a 18. definíciókban megadott képletekkel kiszámított értékek, a másodrendű deriváltak felhasználásával. Az

ábrákon a 0 fehérrel, a negatív értékek vörössel, míg a pozitív értékek zölddel vannak színezve.

### Tórusz

Első tesztnek egy egyszerű tórusz megjelenítését választottam, ugyanis ez még csupán egy darab primitív, de egyben nem is a legkevésbé számításigényes közülük.



6.2. ábra. A teszt során rajzolt ábrák.

Árnyalás	FPS (Hz)	Képkocka (ms)	Árnyalás	FPS (Hz)	Képkocka (ms)
Phong	376.614	2.656	Phong	129.491	7.723
Gradiens	367.635	2.721	Gradiens	68.039	14.697
Gauss-görbület	377.572	2.648	Gauss-görbület	10.449	95.704
Középgörbület	378.268	2.644	Középgörbület	10.425	95.924

**NVIDIA GeForce GTX 1660 Ti**

Driver: 512.15

1920 × 1080

**Intel UHD Graphics 630**

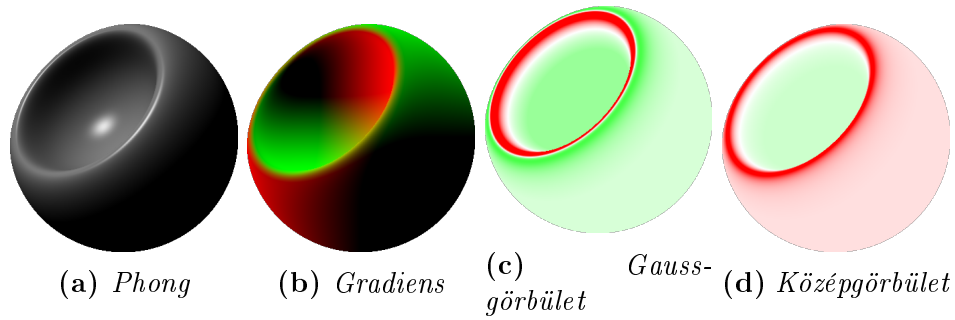
Driver: 30.0.101.1340

640 × 480

**6.3. táblázat.** Tórusz rajzolására vonatkozó 3 perc futtatás alatt kapott átlagos FPS és képkocka idők.

### Simított különbség

A második tesztben a megjelenítendő színtér egy nagyobb és egy kisebb gömb simított különbségéből áll.



6.3. ábra. A teszt során rajzolt ábrák.

Árnyalás	FPS (Hz)	Képkocka (ms)	Árnyalás	FPS (Hz)	Képkocka (ms)
Phong	365.115	2.739	Phong	133.96	7.465
Gradiens	367.824	2.719	Gradiens	28.382	35.233
Gauss-görbület	✗	✗	Gauss-görbület	3.644	274.461
Középgörbület	✗	✗	Középgörbület	3.645	274.425

**NVIDIA GeForce GTX 1660 Ti**  
 Driver: 512.15  
 1920 × 1080

**Intel UHD Graphics 630**  
 Driver: 30.0.101.1340  
 640 × 480

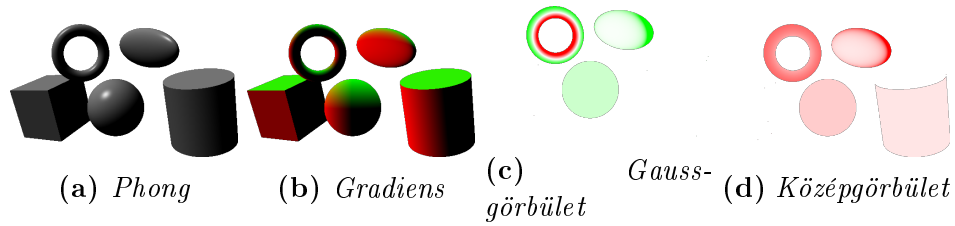
6.4. táblázat. Két gömb simított különbgésének rajzolására vonatkozó 3 perc futtatás alatt kapott átlagos FPS és képkocka idők.

## Primitívek

A harmadik tesztben egyszerre jelenítem meg az implementáció által ismert összes primitívet. Ezek közül mind a sphere tracing, mind az automatikus differenciálás számára az ellipszoid bír a legnagyobb műveletigénnyel, ugyanis ennek távolságfüggvénye sok szorzást és osztást, valamint több hossz számítást is tartalmaz, valamint magát a távolságot csak alulról becsüli.

A mellékelt táblázatban van egy a korábbiak fényében meglepő eredmény. A dedikált Nvidia GPU-ra az előző két tesztben az volt jellemző, hogy nem volt jelentős teljesítmény csökkenés a deriváltak számításának hatására, amikor azok shaderei lefordultak. Most viszont azt látjuk, hogy már a gradiensek automatikus differenciálással történő meghatározása is ahhoz vezet, hogy elveszik a valós idejű sebesség. Emögött a jelenség mögött azt valószínűsítem, hogy a komplexebb távolságfüggvényben végzett duális számítások sokkal több részeredmény tárolását követelik meg.

Amikor ezek már nem férnek el a GPU gyorsítótárában, akkor az ilyen drasztikus mértékű teljesítmény veszteséget eredményez.



6.4. ábra. A teszt során rajzolt ábrák.

Árnyalás	FPS (Hz)	Képkocka (ms)	Árnyalás	FPS (Hz)	Képkocka (ms)
Phong	303.174	3.299	Phong	118.8	8.417
Gradiens	12.708	78.694	Gradiens	12.767	78.326
Gauss-gömbület	✗	✗	Gauss-gömbület	1.19	840.017
Középgömbület	✗	✗	Középgömbület	1.167	856.969

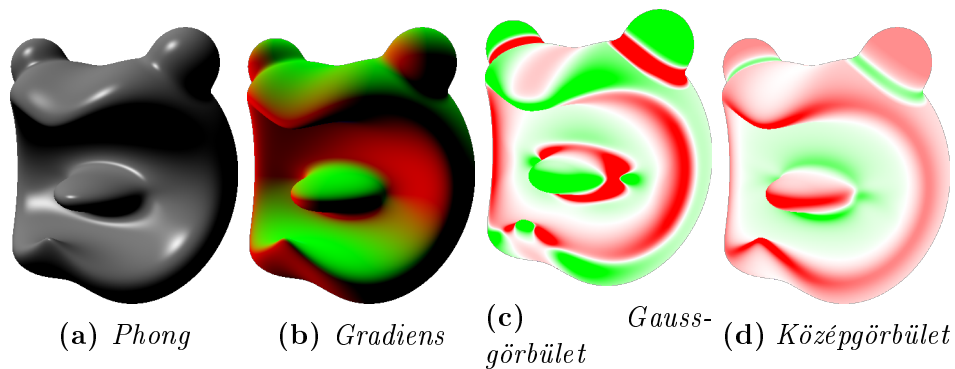
**NVIDIA GeForce GTX 1660 Ti**  
Driver: 512.15  
1920 × 1080

**Intel UHD Graphics 630**  
Driver: 30.0.101.1340  
640 × 480

6.5. táblázat. A primitívek rajzolására vonatkozó 3 perc futtatás alatt kapott átlagos FPS és képkocka idők.

## Figura

Negyedik és egyben utolsó tesztként egy kiáltó figurát készítettem különböző primitívekből simított műveletekkel.



6.5. ábra. A teszt során rajzolt ábrák.

Árnyalás	FPS (Hz)	Képkocka (ms)	Árnyalás	FPS (Hz)	Képkocka (ms)
Phong	37.591	26.603	Phong	91.238	10.96
Gradiens *	✗	✗	Gradiens	7.525	132.885
Gauss-görbület *	1.174	851.915	Gauss-görbület	0.368	2717.43
Középgörbület *	1.17	854.911	Középgörbület	0.368	2720.04

**NVIDIA GeForce GTX 1660 Ti**

Driver: 512.15  
1920 × 1080

**Intel UHD Graphics 630**

Driver: 30.0.101.1340  
640 × 480

**6.6. táblázat.** A figura rajzolására vonatkozó 3 perc futtatás alatt kapott átlagos FPS és képkocka idők.

*Megjegyzés.* Az Nvidia driver a \*-gal jelölt esetekben instabil GPU kódot generált, vagy még a fordítás során összeomlott.

**6.3.1. Összegzés**

A fenti tesztek során összefoglalva az látható, hogy a teljesítményre legnagyobb hatással a számítások során egyszerre használt ideiglenes változók száma van. Mindez akár a shader fordíthatóságára is hatással lehet. Ugyanakkor amennyiben egyszerűbb modellünk van viszonylag kevés művelettel, akkor egy megfelelő teljesítményű közepkategóriás dedikált GPU-n a valós idejű megjelenítés nem jár szignifikáns extra költséggel, azaz a gyakorlatban akár más eljárás mellett, a szintér egy részletének megjelenítésére is alkalmazható.



## IV. rész

### Összegzés

---

A dolgozat bemutatja egy olyan háromdimenziós modellezőszoftver megvalósítását, amelyben egy gráfszerkesztőben összeállított CSG<sup>4</sup> gráf segítségével adható meg a megjeleníteni kívánt modell. A jelenleg létező ilyen szoftverek a modell megjelenítését változatos eljárásokkal valósítják meg, melyek egyike a sphere tracing. Utóbbi alkalmazásához előállítják a CSG gráfból a modell előjeles távolságfüggvényét kiszámító GPU kódot. Kirajzoláskor az árnyalás kiszámításához szükséges normálvektorokat differenciahányadossal való approximációval határozzák meg a távolságfüggvényből.

A dolgozatban bemutatott módszer a távolságfüggvény mellett egy másik függvényt is legenerál, amely a numerikus approximáció helyett automatikus differenciálás segítségével számítja ki a távolságfüggvény gradiensét, mely nem más mint a felület normálisa. Kiterjesztésképp megvizsgáltam magasabb rendű deriváltak meghatározására alkalmas GPU kód generálását is.

Az így kapott eljárás pontossága meghaladja a numerikus approximációval elérhető, ugyanakkor magas költsége miatt csak kivételes esetekben érdemes előnyben részesíteni. Amennyiben egy szintér méretéhez képest apró részleteket is tartalmaz, akkor az automatikus differenciálás minden  $\epsilon$  lépéstávolság mellett felülmúlja a differenciahányadossal történő approximációt. Mindemellett minél magasabb deriváltakat kívánunk meghatározni, annál inkább halmozódnak a közelítő eljárásokból származó hibák, így egyre jelentősebb a különbség pontosság tekintetében az automatikus differenciálás javára. Ugyanakkor a magasabb rendű esetekben utóbbi memóriahasználata és műveletigénye is drasztikusan megnő<sup>5</sup>, így különösen összetettebb szinterek esetén alkalmatlanná válik a valós idejű megjelenítésre.

Az általam implementált megvalósítás csak az egyszerű szinterek esetén képes valósidejű megjelenítésre, ugyanakkor a kódgenerátor jelenlegi formájában csak minimálisan törekszik optimális glsl kód generálására, azaz vélhetően még további optimalizációkkal a futásidő érezhető mértékben javítható.

---

<sup>4</sup>Constructive Solid Geometry (4.1. fejezet)

<sup>5</sup>Hiszen az eredeti számítás során használt minden egyes lebegőpontos számot 4, 10 vagy akár még több lebegőpontos változó által lesz helyettesítve.

## 7. fejezet

# További fejlesztési lehetőségek

A gyakorlatban való alkalmazhatóság szempontjából elsődlegesen a shaderek komplexitását lenne érdemes minél inkább csökkenteni, hogy összetettebb modellek esetén is fordíthatóak legyenek Nvidia videokártyákon. Ennek megvalósítására több tervezett módosítás is van:

- Általános deriváltak helyett korlátozni lehetne ezek kiszámítását legfeljebb másodrendű esetre. Így sok ciklus helyettesíthető lenne utasítások sorozatával. Noha a ciklusok kibontását a glsl fordító is általában megteszi, ezt nem tudjuk közvetlenül befolyásolni.
- Az Nvidia platformon való fordíthatóság érdekében érdemes lenne platform-specifikus optimalizációkat eszközölni, felhasználva a gyártó által kínált GLSL kiterjesztéseket.
- Amd platformon nem volt lehetőség átfogó tesztelésre, a továbbiakban érdemes lenne itt is megvizsgálni a generált shaderek viselkedését, fordíthatóságát. Az Intel és Nvidia közötti váratlan és jelentős különbségek miatt ez minden bizonnyal hasznos információval szolgálna.
- A jelenlegi implementáció csak a legegyszerűbb optimalizációkat hajtja végre a kódgenerálás során. Célszerű lenne megvizsgálni és kipróbálni további módszereket a használt regiszterek számának csökkentése érdekében. Egy ilyen lehetőség például a transzformációs mátrixok felbontása eltolásra, forgatásra és skálázásra, ami lehetőséget adna arra, hogy egyes felesleges mátrixszorzásokat ne végezzünk el. Például ha az egyetlen transzformáció egy eltolás vagy

skalázás, akkor az egy egyszerű összeadással, vagy vektor-skalár szorzattal megvalósítható.

# Irodalomjegyzék

- [1] Bui Tuong Phong. “Illumination for Computer Generated Pictures”. *Commun. ACM* 18.6 (1975), 311–317. ISSN: 0001-0782. DOI: 10.1145/360825.360839. URL: <https://doi.org/10.1145/360825.360839>.
- [2] John C Hart. “Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces”. *The Visual Computer* 12.10 (1996), 527–545. old.
- [3] László Szirmay-Kalos. “Higher Order Automatic Differentiation with Dual Numbers”. *Periodica Polytechnica Electrical Engineering and Computer Science* 65.1 (2021), 1–10. old.
- [4] Michael Hardy. “Combinatorics of Partial Derivatives”. *Electronic Journal of Combinatorics* 13 (2006. febr.). DOI: 10.37236/1027.
- [5] Ron Goldman. “Curvature formulas for implicit curves and surfaces”. *Computer Aided Geometric Design* 22.7 (2005), 632–658. old.
- [6] Inigo Quilez. *Interior SDFs*. <https://iquilezles.org/articles/interiordistance/>. Accessed: 2022-05-03. 2020.
- [7] Inigo Quilez. *Distance functions*. <https://iquilezles.org/articles/distfunctions/>. Accessed: 2022-05-03.
- [8] Inigo Quilez. *Smooth minimum*. <https://iquilezles.org/articles/smin/>. Accessed: 2022-05-09. 2013.

# Algoritmusjegyzék

1.	Sphere tracing . . . . .	30
2.	Rekurzív kódgenerálás váza (transzformációk nélkül) . . . . .	48
3.	Rekurzív kódgenerálás (transzformációkkal) . . . . .	51
4.	$\mathbb{R} \rightarrow \mathbb{R}$ függvény $\mathcal{D}_3^{(N)} \rightarrow \mathcal{D}_3^{(N)}$ megfelelőjét kiszámító GLSL kód generálása, a Faá di Bruno formula kombinatorikus alakjának alkalmazásával [4]. Az algoritmus által generált duális gyökfüggvény megtekinthető az A függelékben. . . . .	53

## A. függelék

# Duális GLSL függvénykönyvtár

Ebben a függelékben a 4. fejezetben bemutatott kódgeneráláshoz szükséges duális könyvtár legfontosabb függvényeinek GLSL kódja olvasható.

```
1 #define DERIVATIVE_ORDER 2
2 #define SIZE 10 // Nth tetrahedral number (here: 2nd)
3 struct dnum { // dual number type
4     float d[SIZE];
5 };
6
7 dnum constant(float val) {
8     dnum res;
9     res.d[0] = val;
10    for(int i = 1; i < SIZE; ++i) {
11        res.d[i] = 0;
12    }
13    return res;
14 }
15
16 dnum variable(float val, float dx, float dy, float dz) {
17     dnum res;
18     res.d[0] = val;
19     res.d[1] = dx;
20     res.d[2] = dy;
21     res.d[3] = dz;
22     for(int i = 4; i < SIZE; ++i)
23         res.d[i] = 0;
24     return res;
25 }
```

```
1 dnum conj(dnum a) {
2     for(int i = 1; i < SIZE; ++i) {
3         a.d[i] *= -1;
4     }
5     return a;
6 }
7
8 dnum neg(dnum a) {
9     for(int i = 0; i < SIZE; ++i) {
10        a.d[i] *= -1;
11    }
12    return a;
13 }
14
15 #define PRECISION 0.000001
16 bool isReal(dnum a) {
17     bool res = true;
18     for(int i = 1; i < SIZE; ++i) {
19         res = res && (a.d[i] < PRECISION || a.d[i] > -PRECISION);
20     }
21     return res;
22 }
23
24 dnum add(dnum a, dnum b) {
25     dnum c;
26     for(int i = 0; i < SIZE; ++i) {
27         c.d[i] = a.d[i] + b.d[i];
28     }
29     return c;
30 }
31
32 dnum sub(dnum a, dnum b) {
33     dnum c;
34     for(int i = 0; i < SIZE; ++i) {
35         c.d[i] = a.d[i] - b.d[i];
36     }
37     return c;
38 }
```



```

1 // tetra and tri are precomputed arrays
2 // containing tetrahedral and triangular numbers
3 #define IDX(x,y,z) (tetra[(x)+(y)+(z)+1] - tri[(x)+(y)+1] + (y))
4
5 dnum mul(dnum a, dnum b) {
6     dnum c = zero(); // zero initialize all components
7     for(int x = 0; x <= DERIVATIVE_ORDER; ++x){
8         int yend = DERIVATIVE_ORDER - x;
9         for(int y = 0; y <= yend; ++y){
10             int zend = yend - y;
11             for(int z = 0; z <= zend; ++z){
12                 int iend = zend - z;
13                 for(int i = 0; i <= iend; ++i) {
14                     int jend = iend - i;
15                     for(int j = 0; j <= jend; ++j) {
16                         int kend = jend - j;
17                         for(int k = 0; k <= kend; ++k) {
18                             // x+y+z+a+b+c <= DERIVATIVE_ORDER
19                             c.d[IDX(x+i,y+j,z+k)] +=
20                                 a.d[IDX(x,y,z)] * b.d[IDX(i,j,k)]
21                                 * choose[x+i][x] * choose[y+j][y]
22                                 * choose[z+k][z];
23                             // choose is a precomputed const array
24                             // choose[i][j] = i nCr j
25                         }
26                     }
27                 }
28             }
29         }
30     }
31     return c;
32 }

```

A 3.10 képlet szerinti szorzás rendkívül költséges. Bár az egyes ciklusok külön külön vett iterációinak száma viszonylag kevés, egymásba ágyazottságuk miatt így is sok műveletet kell elvégezni még akkor is, ha a fordító a ciklusokat (nagy valószínűséggel) kibontja.

```
1 dnum div(dnum a, dnum b) {
2     while(!isReal(b)){
3         dnum c = conj(b);
4         a = mul(a, c);
5         b = mul(b, c);
6     }
7
8     for(int i = 0; i < SIZE; ++i) {
9         a.d[i] /= b.d[0];
10    }
11    return a;
12 }
13
14 dnum dabs(dnum a) {
15     if(a.d[0] < 0)
16         return mul(a, -1.0);
17     return a;
18 }
19
20 dnum dmax(dnum a, dnum b) {
21     if(a.d[0] >= b.d[0])
22         return a;
23     return b;
24 }
25
26 dnum dmin(dnum a, dnum b) {
27     if(a.d[0] <= b.d[0])
28         return a;
29     return b;
30 }
31
32 dnum dclamp(dnum val, dnum low, dnum high) {
33     if(realValue(val) < realValue(low)) {
34         return low;
35     } else if(realValue(val) > realValue(high)) {
36         return high;
37     }
38     return val;
39 }
```

```

1 dnum dmix(dnum a, dnum b, dnum t) {
2     return add(mul(a, sub(constant(1), t)), mul(b, t));
3 }

```

Az alábbi függvény a 4. algoritmus szerint lett legenerálva a láncszabály és a Faà di Bruno formula alkalmazásával, az egyváltozós  $\mathbb{R} \rightarrow \mathbb{R}$  gyökfüggvény deriváltjainak képleteiből.

```

1 dnum dsqrt(dnum d) { // generated
2     float tmp;
3     dnum result = zero();
4     result.d[0] = sqrt(d.d[0]);
5     tmp = 1/(2*sqrt(d.d[0]));
6     tmp *= d.d[IDX(0, 0, 1)];
7     result.d[IDX(0, 0, 1)] += tmp;
8     tmp = 1/(2*sqrt(d.d[0]));
9     tmp *= d.d[IDX(0, 0, 2)];
10    result.d[IDX(0, 0, 2)] += tmp;
11    tmp = -1.0/4 * 1/sqrt(d.d[0]*d.d[0]*d.d[0]);
12    tmp *= d.d[IDX(0, 0, 1)];
13    tmp *= d.d[IDX(0, 0, 1)];
14    result.d[IDX(0, 0, 2)] += tmp;
15    tmp = 1/(2*sqrt(d.d[0]));
16    tmp *= d.d[IDX(0, 1, 0)];
17    result.d[IDX(0, 1, 0)] += tmp;
18    tmp = 1/(2*sqrt(d.d[0]));
19    tmp *= d.d[IDX(0, 1, 1)];
20    result.d[IDX(0, 1, 1)] += tmp;
21    tmp = -1.0/4 * 1/sqrt(d.d[0]*d.d[0]*d.d[0]);
22    tmp *= d.d[IDX(0, 1, 0)];
23    tmp *= d.d[IDX(0, 0, 1)];
24    result.d[IDX(0, 1, 1)] += tmp;
25    tmp = 1/(2*sqrt(d.d[0]));
26    tmp *= d.d[IDX(0, 2, 0)];
27    result.d[IDX(0, 2, 0)] += tmp;
28    tmp = -1.0/4 * 1/sqrt(d.d[0]*d.d[0]*d.d[0]);
29    tmp *= d.d[IDX(0, 1, 0)];
30    tmp *= d.d[IDX(0, 1, 0)];
31    result.d[IDX(0, 2, 0)] += tmp;
32    tmp = 1/(2*sqrt(d.d[0]));
33    tmp *= d.d[IDX(1, 0, 0)];

```

```
34     result.d[IDX(1, 0, 0)] += tmp;
35     tmp = 1/(2*sqrt(d.d[0]));
36     tmp *= d.d[IDX(1, 0, 1)];
37     result.d[IDX(1, 0, 1)] += tmp;
38     tmp = -1.0/4 * 1/sqrt(d.d[0]*d.d[0]*d.d[0]);
39     tmp *= d.d[IDX(1, 0, 0)];
40     tmp *= d.d[IDX(0, 0, 1)];
41     result.d[IDX(1, 0, 1)] += tmp;
42     tmp = 1/(2*sqrt(d.d[0]));
43     tmp *= d.d[IDX(1, 1, 0)];
44     result.d[IDX(1, 1, 0)] += tmp;
45     tmp = -1.0/4 * 1/sqrt(d.d[0]*d.d[0]*d.d[0]);
46     tmp *= d.d[IDX(1, 0, 0)];
47     tmp *= d.d[IDX(0, 1, 0)];
48     result.d[IDX(1, 1, 0)] += tmp;
49     tmp = 1/(2*sqrt(d.d[0]));
50     tmp *= d.d[IDX(2, 0, 0)];
51     result.d[IDX(2, 0, 0)] += tmp;
52     tmp = -1.0/4 * 1/sqrt(d.d[0]*d.d[0]*d.d[0]);
53     tmp *= d.d[IDX(1, 0, 0)];
54     tmp *= d.d[IDX(1, 0, 0)];
55     result.d[IDX(2, 0, 0)] += tmp;
56     return result;
57 }
```