

TDK-dolgozat

Karikó Csongor Csanád

Lezárva: 2022 július 3-án

Távolságfüggvény és deriváltjainak generálása CSG gráfból

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

ALGORITMUSOK ÉS ALKALMAZÁSAIK TANSZÉK



Szerző:

Karikó Csongor Csanád
programtervező informatikus BSc
3. évfolyam

Témavezető:

Bán Róbert
doktorandusz

Budapest, 2022

Tartalomjegyzék

1. Bevezetés	3
2. Elméleti háttér	7
2.1. Fogalmak	7
2.2. Sphere tracing	8
2.2.1. Normálvektorok approximálása	9
2.3. Duális számok	11
2.3.1. Alapműveletek	11
2.3.2. Függvények	12
2.3.3. Automatikus differenciálás	13
2.3.4. Kiterjesztés magasabb deriváltakra	13
2.3.5. Kiterjesztés többváltozós függvényekre	15
3. Megvalósítás	20
3.1. Constructive Solid Geometry	20
3.1.1. CSG Gráf	20
3.1.2. Transzformációk részgráfokon	24
3.2. Megjelenítés	24
3.3. Kódgenerálás	25
3.3.1. Transzformációk kezelése - a transzformációs verem	27
3.3.2. Automatikus differenciálás	29
3.4. Problémák	35
3.4.1. Hardveres limitációk	35
3.4.2. Lassú shader fordítás	37
4. Eredmények	38
4.1. Deriváltak pontosabb meghatározása	38
4.2. Futásidő	41

4.2.1. Teljesítmény tesztek	41
4.2.2. Összegzés	45
5. Összefoglalás	46
5.1. További fejlesztési lehetőségek	47
5.2. Köszönetnyilvánítás	48
Irodalomjegyzék	49
A. Duális GLSL függvénykönyvtár	51
Ábrajegyzék	57
Algoritmusjegyzék	58

1. fejezet

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 háromszögrácssal 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. Dolgozatomban implicit módon megadott felületekkel, azon belül is előjeles távolságfüggvénnyel megadottakkal foglalkozom. Mindez lehetővé teszi a felületek által körülhatárolt térrészekben vett halmazműveletek viszonylag egyszerű implementációját.

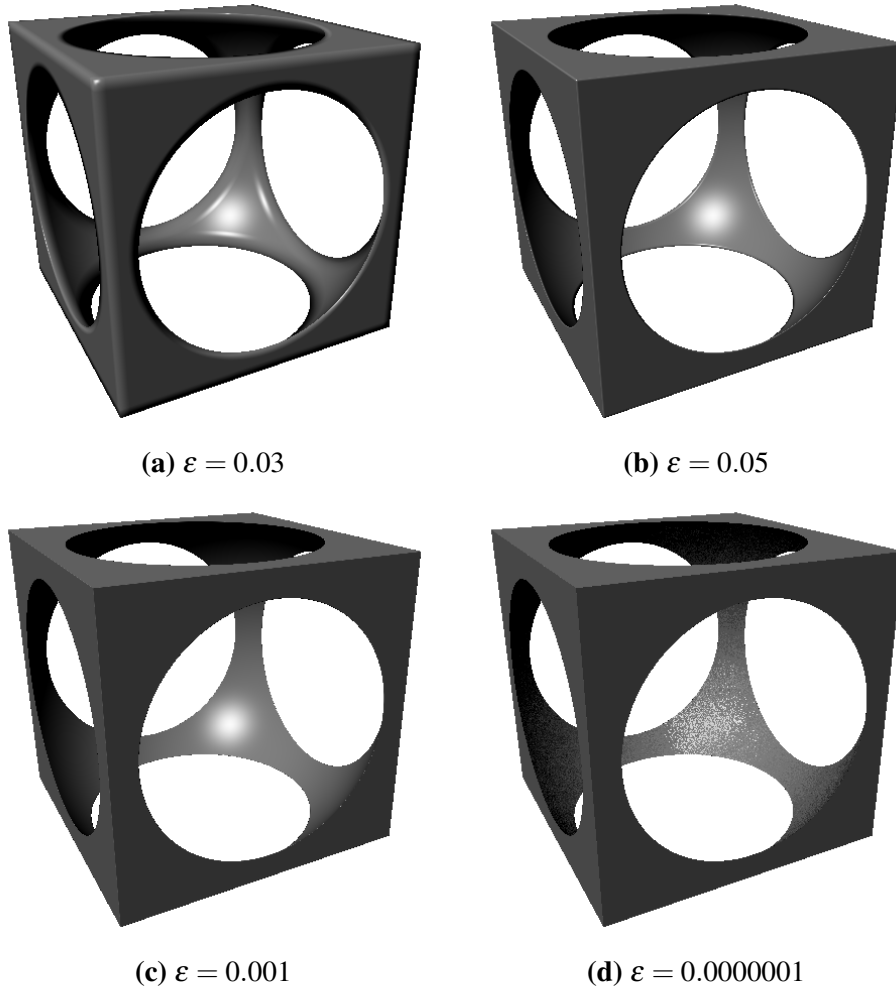
A számítógépes grafikában számos módszer ismert ilyen felületek megjelenítésére. Ezen megoldások alapvetően két kategóriába sorolhatóak. Vannak módszerek, melyek az ily módon megadott felületet először egy másik reprezentációba alakítják, de olyanok is, melyek közvetlenül képesek hasznosítani a távolságfüggvényből kinyerhető információt.

Előbbire példa a felület háromszöghálóval történő approximációja, például a marching cubes algoritmus [1] valamilyen variánsával. Az így kapott háromszögek már hatékonyan megjeleníthetőek hagyományos inkrementális képszintézis segítségével. Ezen vizualizációs módszerek legfontosabb előnye, hogy napjaink GPU-i rendkívül hatékonyan tudják gyorsítani háromszögek rajzolását. Ezzel a megközelítéssel azonban jelentős precizitást áldozunk fel a háromszöghálóvá alakítás során a megjelenítési sebesség érdekében.

A megjelenítési módszerek egy másik csoportját képezik azon sugárkövető eljárások, amelyek a felület kamerából induló félegyenesekkel történő metszésén alapulnak. Maga a metszéspont közelíthető sphere tracing-gel [2], illetve a megjelenítéskori árnyaláshoz

szükséges normálvektor pedig megkapható az előjeles távolságfüggvény környező pontokban történő kiértékelésével, majd az így kapott függvényértékekből számított differenciahányadosokkal. Ez az eljárás sem ad mindig jó megjelenítést, egyfelől a metszéspont is egy approximáció eredménye, valamint a normálvektor közelítésével is akár jelentős hiba kerül az árnyalásba. A kettő közül leginkább utóbbi szembeötlő, ugyanis amennyiben a távolságfüggvény megadása esetszétválasztásokat tartalmaz¹, vagy egyéb okokból a felület éles élekkel rendelkezik, az ott kapott normálvektorok hibája jól láthatóvá válik (1.1a ábra). Ezen bizonyos mértékben tudunk segíteni azzal, ha a differenciahányadosok számításánál minél kisebb ε távolságra mozdulunk el a sphere tracing eredményeként kapott közelített metszésponttól (1.1b ábra). Túl kicsi ε esetén viszont a lebegőpontos számábrázolásból eredő hibák nőnek meg, különösen a felület simább részein, ugyanis ilyenkor a szomszédos pontokban kapott függvényértékek nagyon közeliak, így különbségük hibája megnő. Valamint a numerikus hiba miatt is fellép, hogy a differenciahányadosokban kis ε -al osztunk.

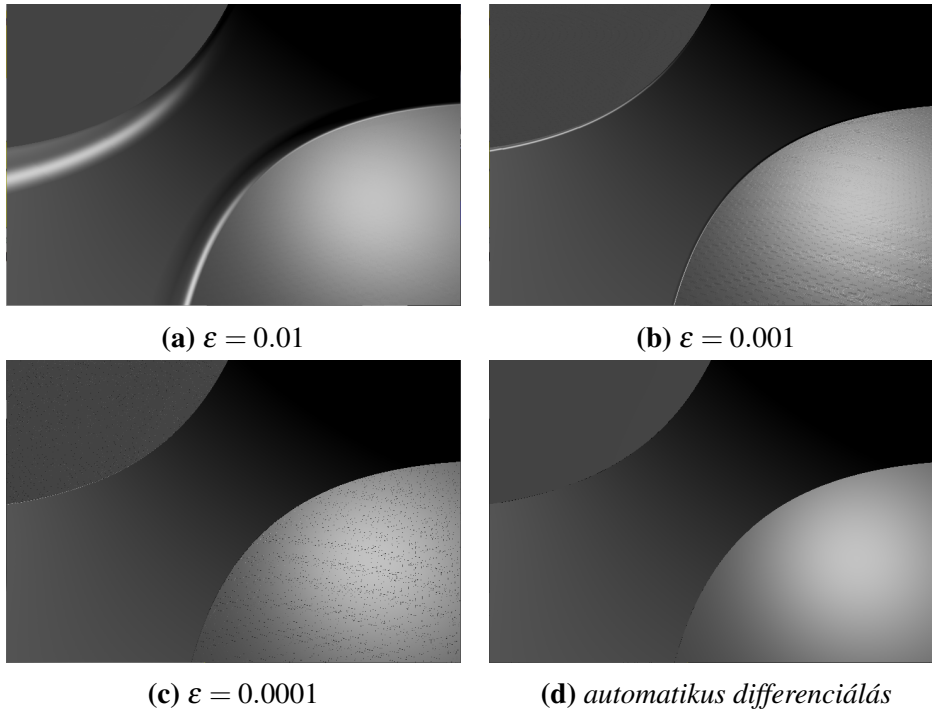
¹Az esetszétválasztások jellemzően elrontják a távolságfüggvény deriváltjának folytonosságát, az approximációs probléma innen származik. Jó péda erre például egy kocka távolságfüggvénye.



1.1. 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 ε mellett ezek a hibák kevésbé látszódnak, míg (c) esetén egyáltalán nem. Ha ε -t túlságosan lecsökkentjük, akkor egy más jellegű hiba lép fel, amely szemcsés árnyaláshoz vezet (d ábra).

Egyes modellek esetében ε megfelelő megválasztásával kiküszöbölhetőek a szemmel észrevehető hibák. Ugyanis bizonyos határokig ε csökkentése javít a pontosságon. Azonban ha a távolságfüggvény kiszámítása során nagy konstansok is szerepelnek², akkor az így megjelenő nagy számok miatt romlik a pontosság, azaz a az 1.1d ábrán látható szemcsés zaj az árnyalásban már nagyobb ε mellett is megjelenik. Tehát ha egy szintér egyszerre tartalmaz nagy objektumokat és apró részleteket, akkor pusztán ε beállításával nem kerülhető el minden normálvektor pontatlanságából következő árnyalási hiba. Egy ezt szemléltető példa látható az 1.2 ábrán.

²Például, ha a megjelenítendő objektum nagyméretű.



1.2. ábra. Példa olyan színtérre, ahol tetszőleges ε esetén láthatóak az approximációs hibák. Kiemelném a **(b)** ábrát, ahol mind a túl kicsi ε -ból fakadó, mind a túl nagy ε esetén jellemző hiba megjelenik. (1.1 ábrán látható lyukas kocka belseje, egy 1000 egység sugarú gömbön)

Dolgozatom célja távolságfüggvények deriváltjainak automatikus differenciálással történő meghatározása. Mindezt egy olyan háromdimenziós modellezőszoftver megvalósításán keresztül teszem, amely az előjeles távolságfüggvény mellett annak gradiensét³ is képes előállítani, majd azt a felület megjelenítéséhez felhasználni. Ezen kívül új eredményként megvizsgáltam a módszer magasabb rendű deriváltakra való kiterjesztését is.

A következő fejezetekben bemutatom az automatikus differenciálás elméleti hátterét, valamint a módszer háromváltozós függvények magasabb rendű deriváltjainak meghatározásához szükséges kiterjesztéseit. Ezt követően a megjelenítést és a hozzá szükséges távolságfüggvény kódjának előállítását részletezem. Végezetül kitérek a módszer korlátaira, a tapasztalt problémákra és a kapott eredményekre.

³Az előjeles távolságfüggvény gradiense a felület pontjaiban megegyezik a felület normálvektorával.

2. 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 bevezetek néhány jelölést az itt tárgyalt fogalmakra, melyeket a későbbiekben is használni fogok.

2.1. 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 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 Ω 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 Ω -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.

2.2. 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 Ω határfelülete ($\partial\Omega$).

A sugárkövetés elvéhez hasonlóan, a virtuális kamera minden képpontjából a nézet irányának megfelelően¹ 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**²), 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 lépünk, vagy nem érjük el a felületet. Tehát lépünk az aktuális pozícióból a sugár irányába d távolságot. Legyen az így kapott pont az új aktuális pozíció.

¹Azaz a fókuszpont és a képpont közeli vágósíkon lévő pozícióját összekötő szakasz meghosszabbításának irányába.

²Signed Distance Function

- 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 := from                                ▷ a jelenlegi pozíció
    d := sdf(p)                              ▷ távolság a felülettől
    i := 0
    while d > dmin and d < dmax and i < step count do
        p := p + d · dir                      ▷ lépés előre
        d := sdf(p)                          ▷ távolság frissítése
        i := i + 1
    end while
    return p

```

2.2.1. Normálvektorok approximálása

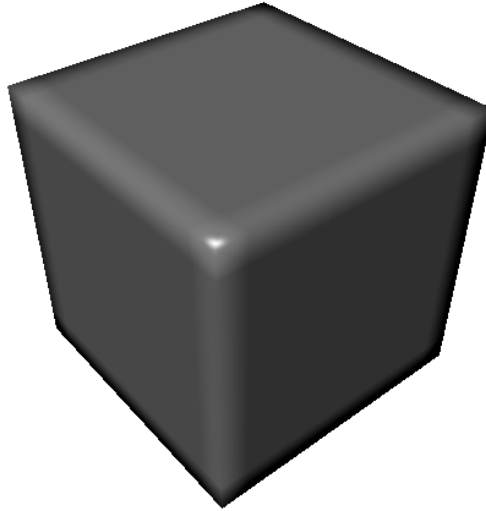
A felület árnyalásához szükséges meghatározoznunk a felület normálvektorát minden olyan pontban, ahol a sphere tracing eljárás során indított sugarak azt eltalálják. Mivel az előjeles távolságfüggvény gradiense ezen pontokban a felület normálvektora, ezért ezek meghatározásának legegyszerűbb módja a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

Legyen ε egy választott konstans érték, melyet a gradiens szimmetrikus differenciahányadosokkal történő közelítéséhez használunk³. Ekkor egy $\vec{p} \in \mathbb{R}^3$, $\vec{p} = [x, y, z]^T$ esetén:

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

ha a megjelenítendő felületünk éles éleket, illetve apró részleteket is tartalmaz, akkor előfordulhat, hogy a differenciahányadosok számításakor az ε 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 2.1 ábrán szemléltetem ezt a jelenséget.

³A közelítést akár több mintavételi ponttal is el lehet végezni [3], így pontosabb eredményt kapva. Ugyanakkor az így kapott formulák esetén is a nevező ε . A továbbiakban tárgyalt hibák ekkor is jelentkeznek.



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

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

- A fenti képletben 2ε -al osztunk, amely a lebegőpontos számábrázolás miatt jelentős pontatlanságot okozhat, amikor ε nagyon kicsi. Bár a gyakorlatban az ε -al történő osztást nem szükséges végrehajtani, a vektor későbbi normalizálása során a probléma ugyanúgy fellép.
- 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 1.1 számú ábrán szemléltetem a különböző ε értékek mellett kapott árnyalást, illetve az 1.2 ábrán azt is, hogy bizonyos esetekben nincs olyan megfelelő ε érték, amivel minden jellegű hiba eltűnne.

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

2.3. Duális számok

Az alábbiakban a duális számok rövid bemutatása található, melyet első sorban Szirmay-Kalos 2021-es cikkére [4] 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:

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

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

2.3.1. Alapműveletek

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

Az alapműveleteket a komplex számokkal analóg módon definiálhatjuk:

$$\overline{\mathbf{\ddot{x}}} = x_0 - x_1 i = D(x_0, -x_1) \quad (2.1)$$

$$\mathbf{\ddot{x}} \pm \mathbf{\ddot{y}} = D(x_0 \pm y_0, x_1 \pm y_1) \quad (2.2)$$

$$\begin{aligned} \mathbf{\ddot{x}} \cdot \mathbf{\ddot{y}} &= (x_0 + x_1 i) \cdot (y_0 + y_1 i) \\ \mathbf{\ddot{x}} \cdot \mathbf{\ddot{y}} &= x_0 y_0 + x_0 y_1 i + x_1 y_0 i + x_1 y_1 i^2 \\ \mathbf{\ddot{x}} \cdot \mathbf{\ddot{y}} &= x_0 y_0 + (x_0 y_1 + x_1 y_0) i = D(x_0 y_0, x_0 y_1 + x_1 y_0) \end{aligned} \quad (2.3)$$

$$\begin{aligned} \frac{\mathbf{\ddot{x}}}{\mathbf{\ddot{y}}} &= \frac{x_0 + x_1 i}{y_0 + y_1 i} \\ \frac{\mathbf{\ddot{x}}}{\mathbf{\ddot{y}}} &= \frac{\mathbf{\ddot{x}} \cdot \overline{\mathbf{\ddot{y}}}}{\mathbf{\ddot{y}} \cdot \overline{\mathbf{\ddot{y}}}} \\ \frac{\mathbf{\ddot{x}}}{\mathbf{\ddot{y}}} &= \frac{x_0 y_0 + (x_1 y_0 - x_0 y_1) i}{y_0^2 + (y_0 y_1 - y_1 y_0) i} \\ \frac{\mathbf{\ddot{x}}}{\mathbf{\ddot{y}}} &= \frac{x_0 y_0 + (x_1 y_0 - x_0 y_1) i}{y_0^2} = D\left(\frac{x_0}{y_0}, \frac{x_1 y_0 - x_0 y_1}{y_0^2}\right) \end{aligned} \quad (2.4)$$

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: $\underline{d} \in \mathcal{D}, \underline{d} = D(f(x), f'(x))$

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.

Például legyen $\underline{x}, \underline{y} \in \mathcal{D}$ és $\underline{x} = D(f(a), f'(a)), \underline{y} = D(g(a), g'(a))$ valamely $a \in \mathbb{R}$ és $f, g \in \mathbb{R} \rightarrow \mathbb{R}, f, g \in D\{a\}$ függvények esetén. Ekkor $\underline{x} \cdot \underline{y}$ műveletet 2.3 szerint elvégezve a kapott eredmény:

$$\underline{x} \cdot \underline{y} = D(f(a)g(a), f(a)g'(a) + f'(a)g(a)) = D(f(a)g(a), (f \cdot g)'(a))$$

2.3.2. 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:

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

Hiszen x_1 a belső függvény deriváltjának helyettesítési értéke.

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

2.1. táblázat. Néhány példa valós függvények duális megfelelőire.

A fentiek segítségével már bevezethető az automatikus differenciálás módszere.

2.3.3. 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.

Megjegyzendő, hogy a szimbolikus differenciáláshoz hasonlóan a deriválási szabályok szerint számolunk minden műveletnél, viszont itt ez a számítás nem különül el a deriváltképlet meghatározásától. Továbbá mivel az automatikus differenciálás szimbólumok helyett értékekkel dolgozik, ezért alkalmasabb a különböző vezérlő utasításokban (elágazások, ciklusok) gazdag függvények deriváltjainak meghatározására.

2.3.4. 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:

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

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

A magasabb rendű duális számok közötti műveleteket az elsőrendű esethez analóg módon kapjuk:

$$\bar{\mathbf{x}} = x_0 - \sum_{k=1}^N x_k i_k = D^{(N)}(x_0, -x_1, \dots, -x_N) \quad (2.5)$$

$$\bar{\mathbf{x}} \pm \bar{\mathbf{y}} = \sum_{k=0}^N (x_k \pm y_k) i_k = D^{(N)}(x_0 \pm y_0, x_1 \pm y_1, \dots, x_N \pm y_N) \quad (2.6)$$

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 [4] a kommutativitás és asszociativitás. A kapott szorzat képlet a következő:

$$\bar{\mathbf{x}} \cdot \bar{\mathbf{y}} = \sum_{j=0}^N \sum_{k=0}^{N-j} x_j y_k \binom{j+k}{k} i_{j+k} \quad (2.7)$$

Osztás Az osztás elvégzését szintén az elsőrendű esethez hasonlóan tesszük. Bővítünk a nevező konjugáltjával (2.5). 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 bővítéseket⁴ 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 [4].

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.

⁴Emiatt a duális osztás jelentősen költségesebb művelet a duális szorzásnál.

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

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

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:

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

Felhasználva a **Faà di Bruno formula** kombinatorikus alakját [5] a kapott duális szám komponenseit explicit módon is megadhatjuk:

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

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

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

Ahol:

- π_k az $\{1, 2, \dots, k\}$ halmaz összes osztályfelbontása.
- $B \in \pi$ azt jelenti, hogy B a π osztályfelbontásban szereplő egyik osztály. ("blokk")

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 $\{6\}$.

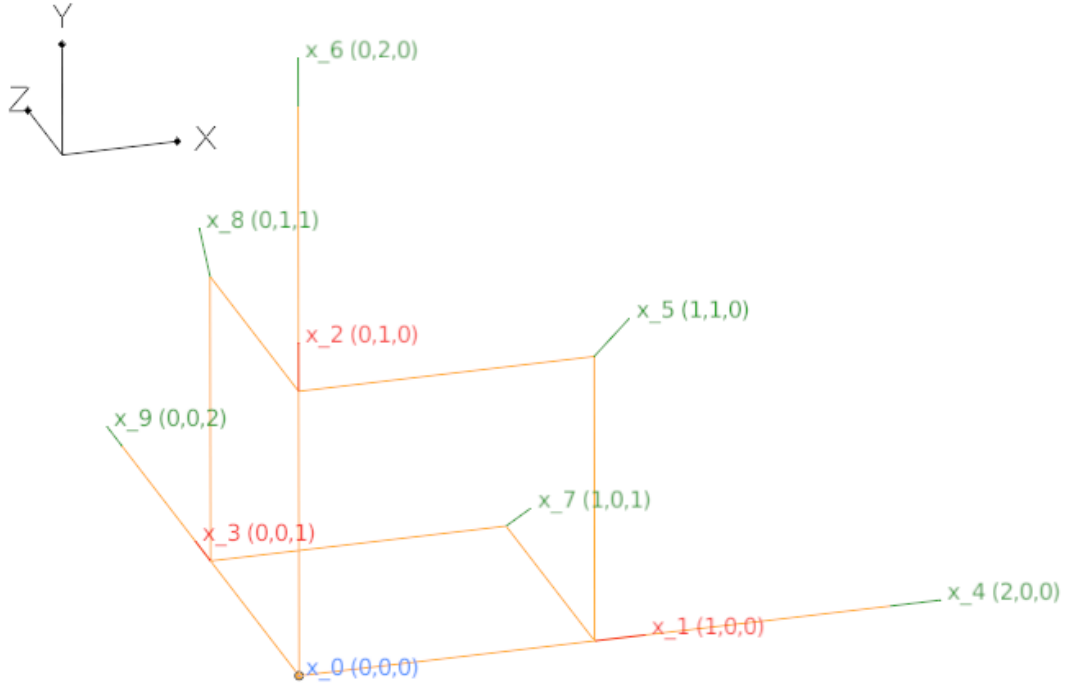
2.3.5. Kiterjesztés többváltozós függvényekre

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 2.2 ábra mutat be.



2.2. 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) -szeres⁵ 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: $\triangle(1) = 1, \triangle(2) = 3, \triangle(3) = 6$

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

12. Definíció. Adott parciális derivált tetraéder indexe.

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

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

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 együtthatója 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:

$$\overline{\ddot{\mathbf{x}}} = D_3^{(N)}(x_0, -x_1, \dots, -x_{\triangleleft(N)}) \quad (2.8)$$

$$\ddot{\mathbf{x}} \pm \ddot{\mathbf{y}} = D_3^{(N)}\left(x_0 \pm y_0, x_1 \pm y_1, \dots, x_{\triangleleft(N)} \pm y_{\triangleleft(N)}\right) \quad (2.9)$$

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 ([4])

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

$$\begin{aligned} \mathbf{\ddot{x}} \cdot \mathbf{\ddot{y}} &= \left(\sum_{a=0}^N \sum_{b=0}^{(N-a)} \sum_{c=0}^{(N-a-b)} x_{\text{idx}(a,b,c)} i_{a,b,c} \right) \cdot \left(\sum_{j=0}^N \sum_{k=0}^{(N-j)} \sum_{l=0}^{(N-j-k)} y_{\text{idx}(j,k,l)} i_{j,k,l} \right) \\ &= \sum_{a=0}^N \sum_{b=0}^{(N-a)} \sum_{c=0}^{(N-a-b)} \sum_{j=0}^N \sum_{k=0}^{(N-j)} \sum_{l=0}^{(N-j-k)} x_{\text{idx}(a,b,c)} y_{\text{idx}(j,k,l)} \cdot i_{a,b,c} i_{j,k,l} \\ &= \sum_{a=0}^N \sum_{b=0}^{(N-a)} \sum_{c=0}^{(N-a-b)} \sum_{j=0}^N \sum_{k=0}^{(N-j)} \sum_{l=0}^{(N-j-k)} x_{\text{idx}(a,b,c)} y_{\text{idx}(j,k,l)} \cdot \binom{a+j}{j} \binom{b+k}{k} \binom{c+l}{l} i_{a+j,b+k,c+l} \\ &\quad \text{Felhasználva, hogy } i_{a,b,c} \cdot i_{j,k,l} = 0 \text{ ha } a+b+c+j+k+l > N : \\ \mathbf{\ddot{x}} \cdot \mathbf{\ddot{y}} &= \sum_{a=0}^N \sum_{b=0}^{(N-a)} \sum_{c=0}^{(N-a-b)} \sum_{j=0}^{(N-a-b-c)} \sum_{k=0}^{(N-a-b-c-j)} \sum_{l=0}^{(N-a-b-c-j-k)} x_{\text{idx}(a,b,c)} y_{\text{idx}(j,k,l)} \cdot \binom{a+j}{j} \binom{b+k}{k} \binom{c+l}{l} i_{a+j,b+k,c+l} \end{aligned} \quad (2.10)$$

Osztás Háromdimenziós esetben is alkalmazható az osztás elvégzésére a 2.3.4 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:

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

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 [5] bonthatjuk ki, melyben megjelennek a belső függvény deriváltjai.

Ezeket az $\mathbf{\ddot{x}}$ 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 π osztályfelbontásban szereplő egyik osztály. ("blokk")⁶
- $B_x = |B \cap \mathfrak{A}|, B_y = |B \cap \mathfrak{B}|, B_z = |B \cap \mathfrak{C}|$ értékek adják meg, hogy a szorzaton belül az $\mathbf{\ddot{x}}$ 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. $\pi_{i,j,k}$ elemeit 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 annyiszoros 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 $\mathbf{\ddot{x}}$ -nek. Azt hogy egy csoporthoz melyik komponens tartozik az határozza meg, hogy benne hányszor szerepel x, y vagy z szerinti deriválást jelképező elem.

Ez a szabály a láncszabály és a szorzás deriválási szabályának ismételt alkalmazásából keletkezik, bizonyítása megtalálható Hardy "Combinatorics of Partial Derivatives" című cikkében. [5]

⁶Tehát 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. 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.

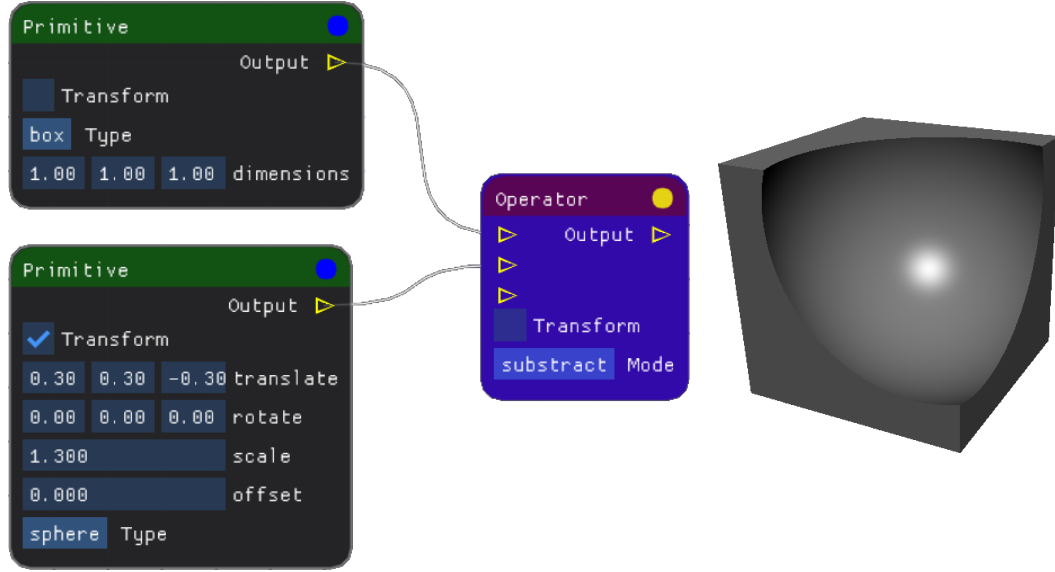
3.1. Constructive Solid Geometry

Constructive Solid Geometry (CSG) [6] 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 gráfjával, melyet CSG gráfnak nevezünk.

3.1.1. CSG Gráf

A CSG gráf egy olyan fagráf, melynek levelei primitív testeket reprezentálnak, míg belső csúcsai ezeken értelmezett halmazműveleteket. A csúcsok közötti élek¹ 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 3.1 ábrán látható egy példa az implementált modellezőszoftverből.

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



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

Műveletek távolságfüggvényeken

Implicit felületek CSG gráffal történő modellezése az előjeles távolságfüggvény, vagy annak egy alsó becslését² 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:

17. 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.

$$sd^*(\vec{x}, A \cup B) := \min(\alpha, \beta) \quad (3.1)$$

$$sd^*(\vec{x}, A \cap B) := \max(\alpha, \beta) \quad (3.2)$$

$$sd^*(\vec{x}, A \setminus B) := \max(\alpha, -\beta) \quad (3.3)$$

Megjegyzés. A fenti képletek közül egyik sem ad a tér minden pontjában helyes távolságfüggvényt a két test uniójára, metszetére vagy különbségére. a 3.1 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 3.2 metszet és a 3.3 különbség esetében a testen kívüli térrészben is csak alsó becslést kapunk a távolságra. Ugyanakkor ez a felület megjelenítésén nem változtat, ugyanis

²Itt 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).

az 1. algoritmussal megadott sphere tracing eljárás a távolságok alsó becslése mellett is jól működik, bár ilyenkor műveletigénye nő. Részletek Quilez cikkében [7].

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.

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

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

19. Definíció. Néhány további művelet távolságokon. [8] Az eredményként kapott felületeket a 3.2 ábra szemlélteti.

Legyen $k \in \mathbb{R}^+$ egy tetszőleges paraméter, melynek hatása a 3.3 ábrán látható.

Simított unió:

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

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

Simított metszet:

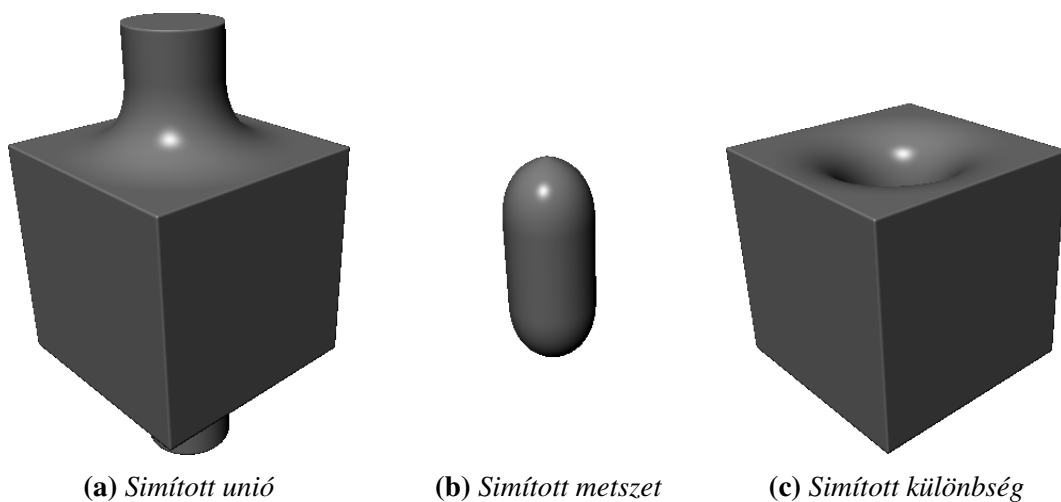
$$\text{sd}^*(\vec{x}, A \cap_k B) := -h\alpha + (1-h)\beta + kh(1-h)$$

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

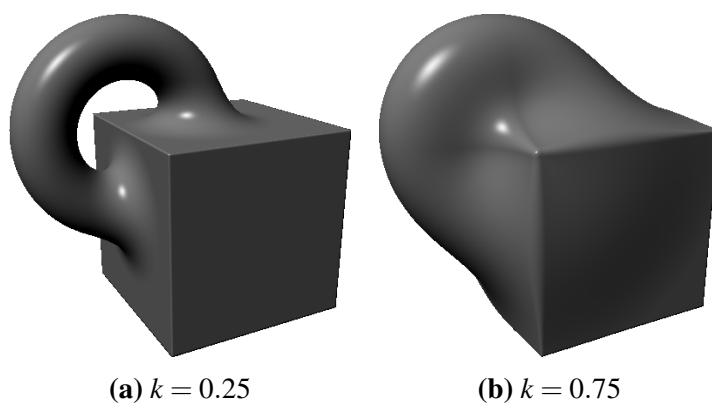
Simított különbség:

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

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

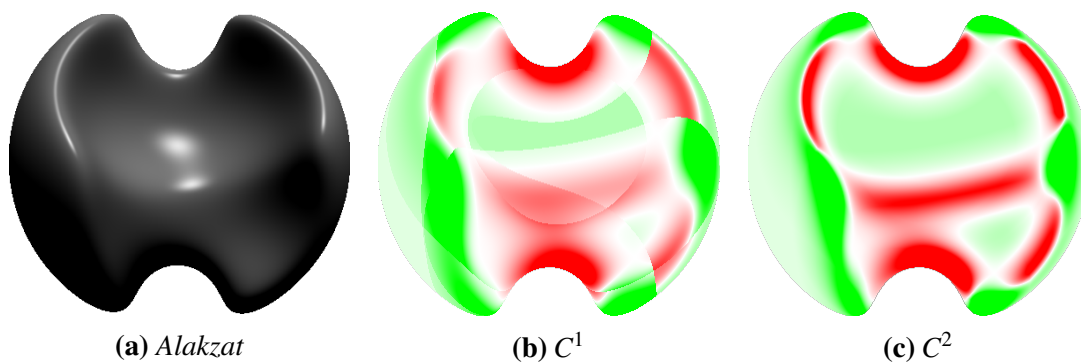


3.2. ábra. Simított műveletek egy kocka és egy henger között.



3.3. ábra. Simított unió különböző k paraméterek mellett.

Megjegyzés. A 19. 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 (3.4 ábra). A simított műveletek definiálhatóak C^n folytonosra is [9] alapján.



3.4. ábra. C^1 és C^2 folytonosság tartó műveletekkel kapott felület Gauss-görbülete [10].

3.1.2. 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 műveletekkel³ kapcsoltunk össze. Ezeken kívül még szükség van arra, hogy képesek legyünk az egyes részfa által reprezentált testeket transzformálni (eltolás, forgatás és uniform skálázás⁴). 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.

3.2. 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, így az egyes pixelek meghatározása párhuzamosan történik (a párhuzamosan számított pixelek száma az adott GPU-tól függ).

A **sphere tracing** eljárás kimeneteként megkapjuk, hogy az egyes képpontokon látható-e a felület. 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 3.5 ábrán). Ez sok olyan képpont esetén is fent áll, ahol a sugár végül nem találja el a felületet. Ezekben a képpontban amúgy

³Az 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.

⁴A 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.

sem számolnánk ki az árnylást, így méginkább felesleges a sugár léptetése során a duális távolságfüggvényt alkalmazni.



3.5. 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 GLSL⁵ kódjának CSG gráfból történő generálására. Az alábbiakban ennek a kódgenerálásnak a menetét mutatom be.

3.3. 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 háromdimenziós 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. Kihasználva, hogy a CSG gráf minden csúcsához tartozó részgráfja is rekurzívan CSG gráf, érdemes a mélységi bejárás stratégiáját követni. Mivel a gráfunk fagráf, ezért a gyökekből 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 kimenetként kapott távol-

⁵OpenGL Shading Language, az OpenGL-ben használatos nyelv a GPU programozására.

ságot. Ha az aktuális csúcs egy primitív, akkor ez rögtön megvalósítható⁶ és visszatérünk annak a változónak a nevével, amelybe a primitívtől 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 változókat⁷ 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

▷ visszatérés az eredmény változó nevével

else

▷ aktuális csúcs egy *művelet*

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)

▷ átmeneti változók felszabadítása későbbi

újrahasználás céljából

end for

return var

▷ visszatérés az eredményváltozó nevével

end if

Funct GenerateSdf(root, output <by reference>)

output += "float sdf(vec3 pos) {"

resultVarName := TraverseGraph(root, output <by reference>)

output += "return " + resultVarName + ";"

output += "}"

⁶Az implementáció során minden egyes támogatott primitívhez kézzel megadjuk a távolságfüggvényt kiszámító GLSL függvényt, itt csak meg kell hívni a megfelelő paraméterekkel.

⁷Azaz feljegyezzük a generátorban, hogy újrahasznosíthatóak.

3.3.1. Transzformációk kezelése - a transzformációs verem

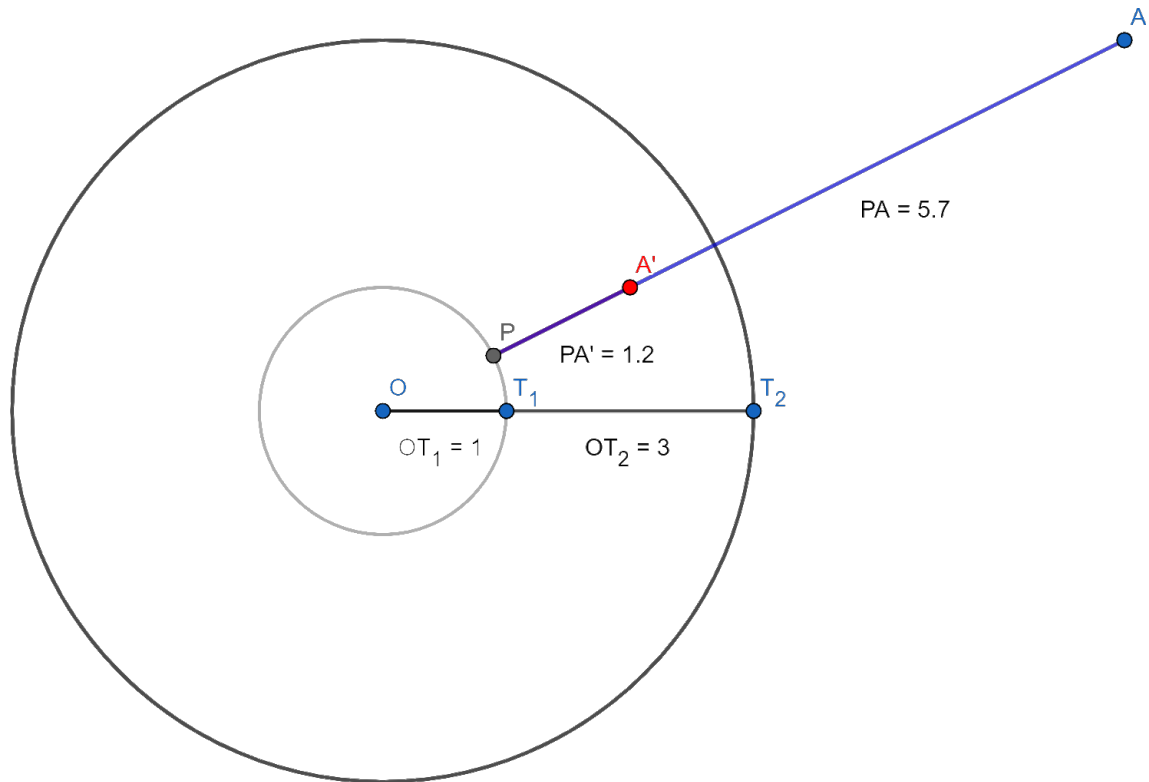
A három megvalósítandó transzformáció az eltolás, origó körüli forgatás és origótól való uniform skálázás. Ezek a műveletek nem értelmezhetők a csúcsok kimeneteként kapott távolságokon, hiszen azok csupán lokális információt hordoznak. Tehát nem lehet belőlük megállapítani, hogy milyen távolságot kaptunk 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 a hatékonyság érdekében 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 (3.6 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.



3.6. 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)

Func TraverseGraph(root, output <by reference>)

```

if root is primitive then
    invTransf := invert(transformStack.top * root.transform)
    output += "inv = " + invTransf + ";"
    output += "invPos = inv * pos;"           ▷ Sdf kiértékelés pontjának meghatározása
    var := CreateOrReuseVariable()
    output += var + " = "
    output += GenCodeForPrimitive(root as primitive, invPos) + ";"
    output += var + " *= " + root.scale + ";"   ▷ skálázási korrekció
    return var                                ▷ visszatérés az eredményváltozó nevével
else                                         ▷ aktuális csúcs egy művelet
    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 + ";"   ▷ skálázási korrekció
    for varName ∈ variables do
        FreeVariable(varName)                 ▷ átmeneti változók felszabadítása későbbi
        újrahasználás céljából
    end for
    return var                                ▷ visszatérés az eredményváltozó nevével
end if

```

Func GenerateSdf(root, output <by reference>)

```

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

```

3.3.2. Automatikus differenciálás

Az előbbieken ismertettem az előjeles távolságfüggvényt kiszámító GLSL kód generálását. A deriváltak meghatározásához a 2.3.3 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 2.2 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 };
```

3.1. forráskód. Duális típus GLSL-ben

A $SIZE$ konstans segítségével definiálhatóak a 2.3.5 részben meghatározott alapl műveletek. Ezek bátran implementálhatóak GLSL függvényekként, ugyanis a shader fordítás során inline-olva lesznek, tehát a függvényhívás nem fog extra költséggel járni. A 2.10 szorzás esetén hatékonysági okokból érdemes a képletben szereplő $\binom{n}{k}$ alakú tényezőket előre kiszámítani. Az így kapott Pascal háromszög eltárolható konstansként a shader elején⁸. A képletben szereplő egymásbaágyazott szummákat implementálhatjuk ciklusokkal, ugyanis mivel ezek iterációinak száma konstans ($SIZE$ -től függ), a fordító unroll-olni⁹ 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 alapl műveletek 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 elégséggé válik a deriváltak képleteinek felsorolása. A 4. algoritmus megvalósítja egy ilyen GLSL függvény legenerálását.

⁸OpenGL 4.3 verzió fölött támogatottak a többdimenziós tömbök is GLSL-ben, így még a tömb egydimenzióssá lapításával sem kell feltétlenül foglalkozni.

⁹Ezzel 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 [5]. Az algoritmus által generált duális gyökfüggvény megtekinthető A függelékben.

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

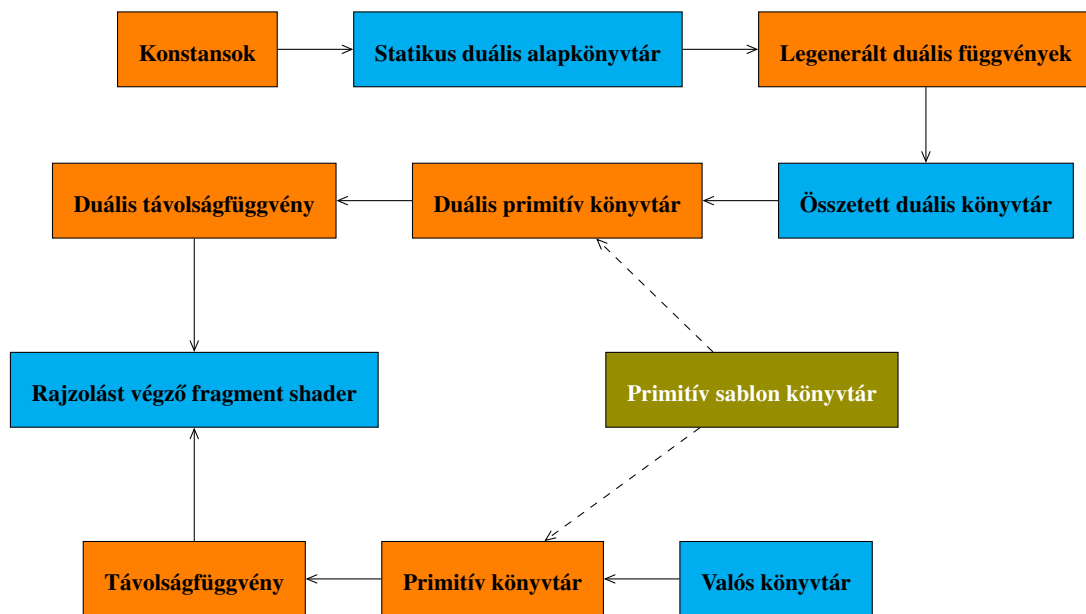
```

output += "dnum " + name + "(dnum x){ "
output += "result := zero();"           ▷ duális szám, melynek minden komponense nulla
for x = 0, ..., N do
    for y = 0, ..., N - x do
        for z = 0, ..., N - x - y do           ▷ Összes lehetséges vegyesderivált felsorolása
            Az (x, y, z) parciális vegyesderivált meghatározása:
            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 shadert 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 3.7 ábrán látható, hogy a teljes fragment shader amely a megjelenítést végzi, milyen részek összefűzésével kapható.



3.7. ábra. A megjelenítést végző fragment shader összeállításának lépései. Az ábrán szereplő folytonos nyilak összefűzést, a szaggatottak szöveghelyettesítéses 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 (példa az A függelékben).
- **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.

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

21. 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  }
```

3.2. 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  }
```

3.3. forráskód. Példa: gömb sablonból generált valós távolságfüggvénye

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

3.4. forráskód. Példa: gömb sablonból generált duális távolságfüggvénye

sablon	valós megfelelő	duális megfelelő
<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>

3.1. 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.

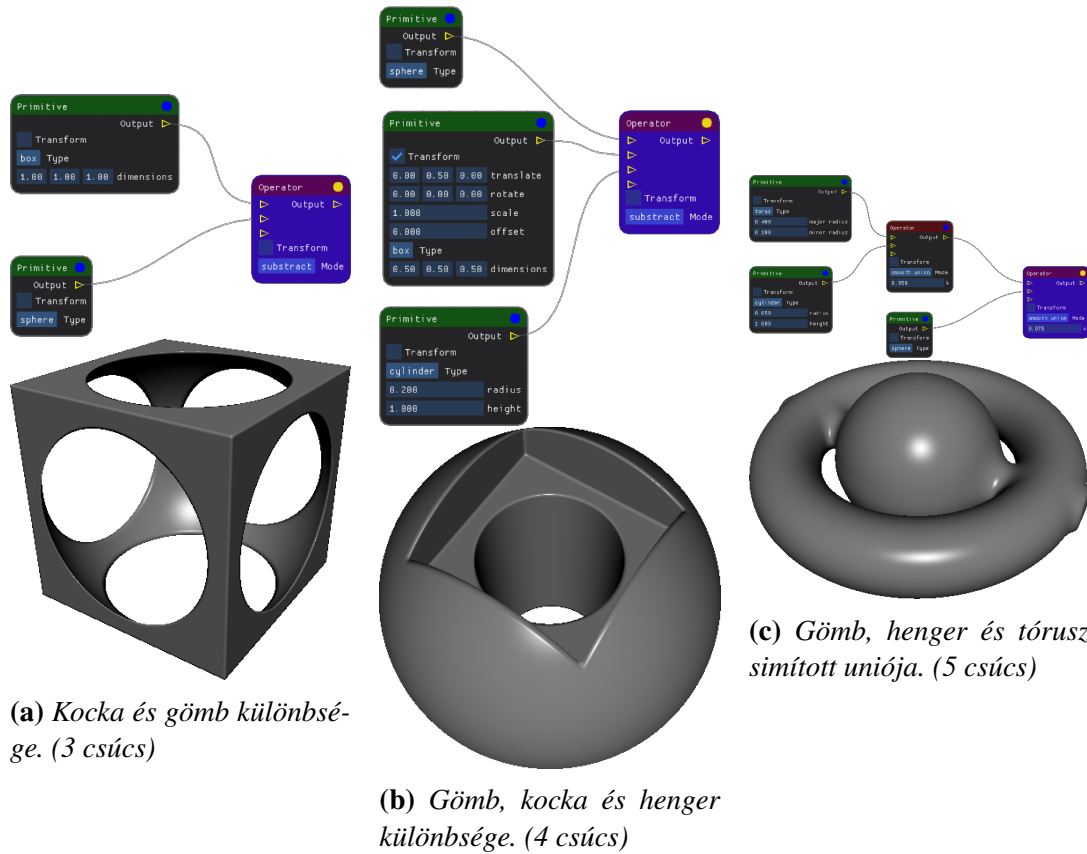
dnum2 és dnum3 rendre kettő és háromdimenziós duális vektorok.

3.4. Problémák

3.4.1. 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 kimenetként kapott GLSL kód esetén ez a fordítási hiba gyakran fellép.

Az alábbiakban néhány példa segítségével illusztrálom ezeken limitációkat a gyakorlatban (3.8 ábra). A tesztekhez egy **NVIDIA GeForce GTX 1660 Ti mobile (6GB)** dedikált, valamint egy **Intel UHD Graphics 630** integrált videokártyát használtam.



3.8. 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

3.2. 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 szerelősza-lagba szorosabban beépülő fragment shader esetén. Mindezt a több lépésre bontás-sal együtt is kipróbáltam. Az első pontban említetthez hasonló módon itt is össze-omlást tapasztaltam. ✗ *Sikertelen.*

3.4.2. Lassú shader fordítás

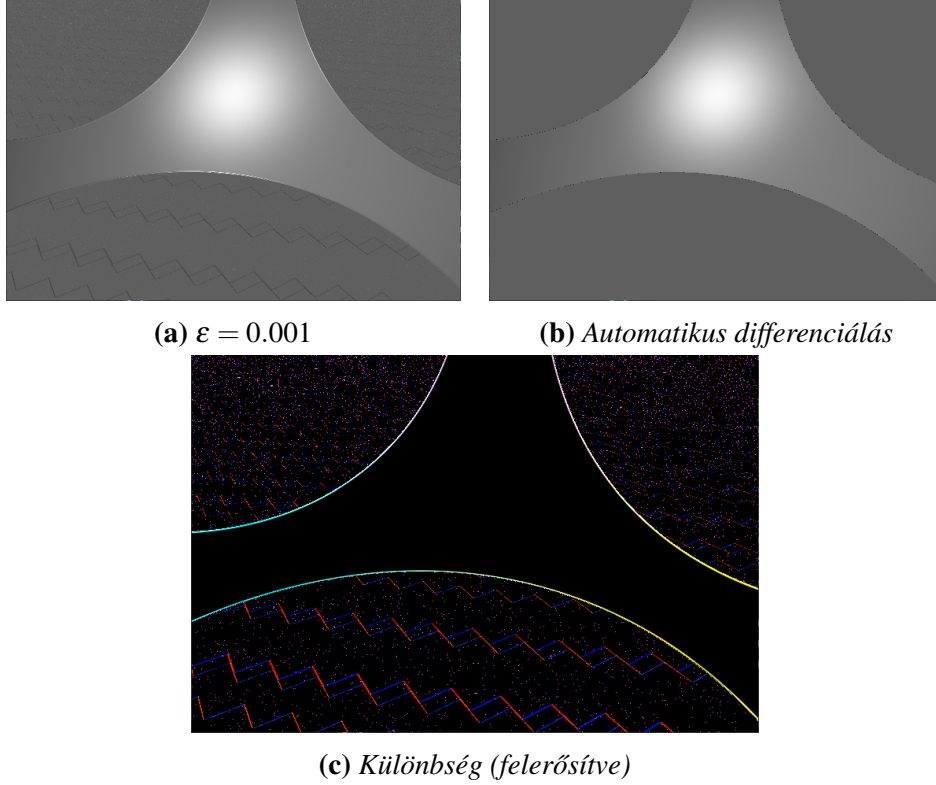
A GPU hardver sajátosságai miatt a GLSL fordító jellemzően agresszív optimalizációkat hajt végre. Ebbe beletartozik függvények inline-olása és gyakran bonyolult ciklusok kibontása is. Mindez jelentős futásidő igényt eredményez, különösen komplexebb shade-rek esetén. Jelen esetben ez úgy mutatkozik meg,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

4. fejezet

Eredmények

4.1. Deriváltak pontosabb meghatározása

A dolgozatban bemutatott módszerek és algoritmusok segítségével lehetséges az előjeles távolságfüggvény magasabb rendű deriváltjainak GPU-n történő pontos meghatározása (4.1 ábra), valamint az ehhez szükséges GPU kód CSG gráfból történő generálása. Az így előállított értékek felhasználhatóak a bevezetőben bemutatott árnyalási hibák kijavítására, vagy olyan felületi tulajdonságok vizualizálására, mint például a Gauss, vagy középgörbület, de akár lokális árnyékhatas megjelenítésére is alkalmasak [11].



4.1. ábra. *Approximált és automatikus differenciálással kapott normálvektorok közötti különbség. (2 élhosszúságú lyukas kocka belülről, egy 10000 sugarú gömb felszínén)*

Görbületek A differenciahányadossal történő approximáció numerikus hibái még inkább látványossá válnak, ha a megjelenítés során az ily módon kapott¹ Hesse-mátrixot is felhasználjuk. Ennek szemléltetésére vizsgáljuk meg az ebből számított Gauss- és középgörbületeket, melyek meghatározásához az alábbi képleteket használtam [10]:

22. Definíció. Gauss-görbület

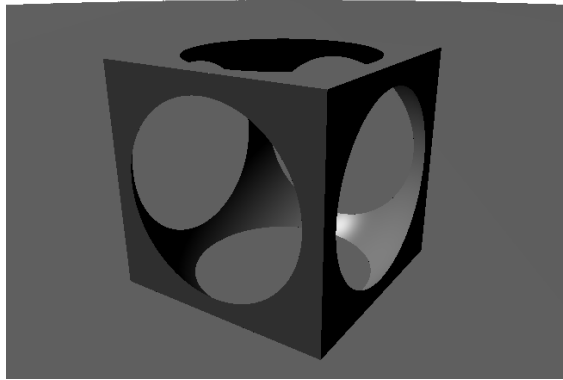
$$K_G = \frac{\nabla F \cdot H^*(F) \cdot \nabla F^T}{|\nabla F|^4}$$

23. Definíció. Középgörbület

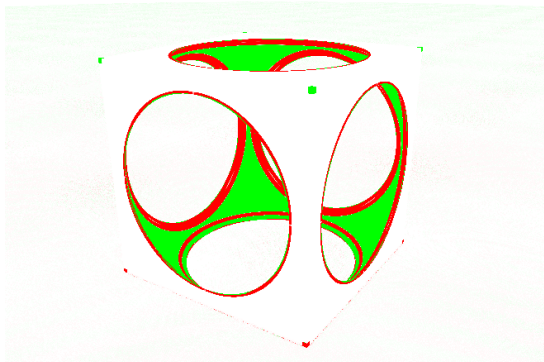
$$K_M = \frac{\nabla F \cdot H(F) \cdot \nabla F^T - |\nabla F|^2 \text{Trace}(H)}{2|\nabla F|^3}$$

Ahol F a távolságfüggvény, ∇F a gradiense, $H(F)$ a Hesse-mátrix, melynek $H^*(F)$ jelöli az adjungáltját. $|\bullet|$ a kettes normát jelenti. Az ezen formulákkal kapott görbületek és az approximált megfelelőjük közötti eltérések szemléltetésére szolgál a 4.2 ábra.

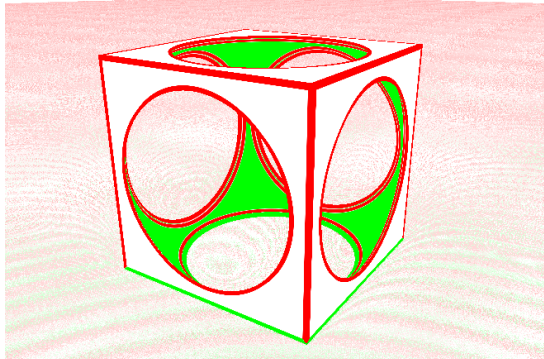
¹A differenciahányadossal approximált gradiens vektorokon vett differenciahányadosokkal kapott Hesse-mátrix.



(a) Phong modellel árnyalva.



(b) Gauss-görbölet



(c) Középgörbölet



4.2. ábra. *Approximációból eredő hibák görbület számításakor. Lyukas egységkocka egy 1000 sugarú gömb felszínén, $\varepsilon = 0.01$ 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 fehérnek látszik.*

4.2. Futásidő

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 színterek esetén megfelelő teljesítményű videokártya mellett képes volt valós idejű sebességet produkálni. Mindez azt jelenti, hogy minden bizonnyal lehetséges lenne további optimalizációkkal, esetleg a magasabbrendű deriváltakról lemondva 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 színterek shadereit elutasítja, sejthetően a 3.4.1 fejezetben leírt okokból.

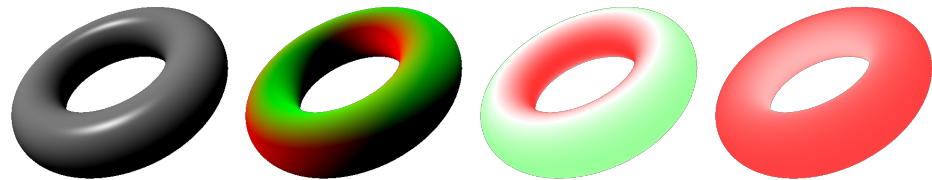
4.2.1. Teljesítmény tesztek

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 [12], 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 22. és a 23. 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.



(a) Phong (b) Gradiens (c) Gauss-görbület (d) Középgörbület

4.3. 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

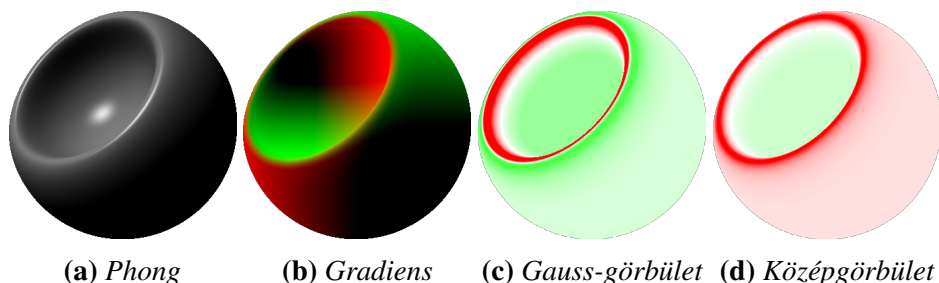
NVIDIA GeForce GTX 1660 Ti
Driver: 512.15
1920 × 1080

Intel UHD Graphics 630
Driver: 30.0.101.1340
640 × 480

4.1. táblázat. Tórusz rajzolására vonatkozó 3 perc futtatás alatt kapott átlagos FPS és képkocka idők. Az Intel GPU esetén szükséges alacsonyabb felbontás választása, ugyanis különben (különösen a későbbi teszteknel) egy képkocka kiszámítása olyan sok időbe telik, hogy az operációs rendszer állítja le a futást.

Simított különbség

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



(a) Phong

(b) Gradiens

(c) Gauss-gömbület

(d) Középgömbület

4.4. 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-gömbület	✗	✗	Gauss-gömbület	3.644	274.461
Középgömbület	✗	✗	Középgömbület	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

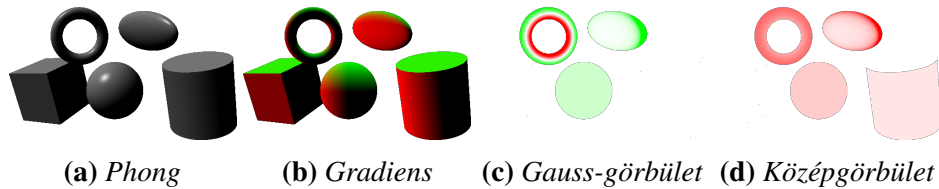
4.2. táblázat. Két gömb simított különbségének rajzolására vonatkozó 3 perc futtatás alatt kapott átlagos FPS és képkocka idők.

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 regiszterekben vagy a lokális memóriában, akkor az ilyen drasztikus mértékű teljesítményvesztéssel eredményez.



4.5. 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-görbület	✗	✗	Gauss-görbület	1.19	840.017
Középgörbület	✗	✗	Középgörbület	1.167	856.969

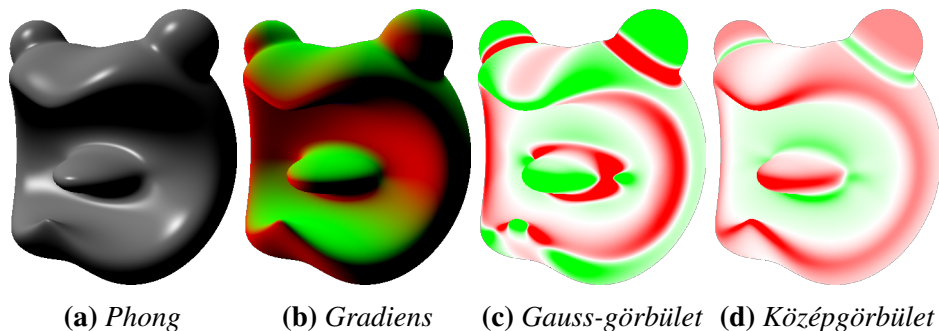
NVIDIA GeForce GTX 1660 Ti
Driver: 512.15
1920 × 1080

Intel UHD Graphics 630
Driver: 30.0.101.1340
640 × 480

4.3. 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.



4.6. 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

4.4. 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.

4.2.2. 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 középkeletkeletű 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.

5. fejezet

Összefoglalás

A dolgozat bemutatja egy olyan háromdimenziós modellezőszoftver megvalósítását, amelyben egy gráfszerkesztőben összeállított CSG¹ 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.

Új eredményként 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 ε 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 megnő², így különösen összetettebb szinterek esetén alkalmatlanná válik a valós idejű

¹Constructive Solid Geometry (3.1. fejezet)

²Hiszen az eredeti számítás során használt minden egyes lebegőpontos számot 4, 10 vagy akár még több lebegőpontos változó által lesz helyettesítve.

megjelenítésre. Ugyanakkor alkalmazható ha nincs szükség valósidejű futásidőre, viszont nagy pontosságú eredményekre igen. Egy ilyen alkalmazás például approximációs módszerek hibájának mérése, amikor az automatikus differenciálással kapott eredmény szolgálhat az összehasonlítás alapjaként. Egy másik példa ha a távolságfüggvény értékeit a tér diszkrétizált pontjaiban számítjuk előre és a későbbi interpoláció pontosságát a deriváltak segítségével kívánjuk növelni.

Az általam implementált megvalósítás csak az egyszerű színterek 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.

5.1. 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 platformspecifikus 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 skálázás, akkor az egy egyszerű összeadással, vagy vektor-skalár szorzattal megvalósítható.

5.2. Köszönetnyilvánítás

Végezetül szeretném megköszönni témavezetőmnek, Bán Róbertnek azt a rengeteg segítséget, támogatást és tanácsot, amely nélkül ez a dolgozat nem készülhetett volna el. További köszönettel tartozom Tóth Teklának dologozatom leadás előtti átolvasásáért és a tőle kapott visszajelzésekért.

Irodalomjegyzék

- [1] William E Lorensen és Harvey E Cline. “Marching cubes: A high resolution 3D surface construction algorithm”. *ACM siggraph computer graphics* 21.4 (1987), 163–169. old.
- [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] Cameron R. Taylor. *Finite Difference Coefficients Calculator*. [https : / / web . media.mit.edu/~crtaylor/calculator.html](https://web.media.mit.edu/~crtaylor/calculator.html). 2016.
- [4] László Szirmay-Kalos. “Higher Order Automatic Differentiation with Dual Numbers”. *Periodica Polytechnica Electrical Engineering and Computer Science* 65.1 (2021), 1–10. old.
- [5] Michael Hardy. “Combinatorics of Partial Derivatives”. *Electronic Journal of Combinatorics* 13 (2006. febr.). DOI: 10.37236/1027.
- [6] Aristides AG Requicha és Herbert B Voelcker. “Constructive solid geometry”. (1977).
- [7] Inigo Quilez. *Interior SDFs*. [https : / / iquilezles . org / articles / interiordistance/](https://iquilezles.org/articles/interiordistance/). Accessed: 2022-05-03. 2020.
- [8] Inigo Quilez. *Distance functions*. [https : / / iquilezles . org / articles / distfunctions/](https://iquilezles.org/articles/distfunctions/). Accessed: 2022-05-03.
- [9] Inigo Quilez. *Smooth minimum*. [https : / / iquilezles . org / articles / smin/](https://iquilezles.org/articles/smin/). Accessed: 2022-05-09. 2013.
- [10] Ron Goldman. “Curvature formulas for implicit curves and surfaces”. *Computer Aided Geometric Design* 22.7 (2005), 632–658. old.

- [11] Tomohito Hattori, Hiroyuki Kubo és Shigeo Morishima. “Real Time Ambient Occlusion by Curvature Dependent Occlusion Function”. *SIGGRAPH Asia 2011 Posters*. SA '11. Hong Kong, China: Association for Computing Machinery, 2011. ISBN: 9781450311373. DOI: 10.1145/2073304.2073358. URL: <https://doi.org/10.1145/2073304.2073358>.
- [12] 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>.

A. függelék

Duális GLSL függvénykönyvtár

Ebben a függelékben a 3. 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 2.10 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 Faádi 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 }
```

Ábrák jegyzéke

1.1. Árnyalási hibák differenciahányadosokkal approximált normálvektorokból.	5
1.2. Példa olyan szintérre, ahol tetszőleges ε esetén láthatóak az approximációs hibák.	6
2.1. Approximált normálvektorral kapott hibás árnyalás az élek mentén.	10
2.2. Parciális deriváltak tetraéderes indexelése.	16
3.1. Egyszerű CSG gráf és az általa reprezentált test.	21
3.2. Simított műveletek egy kocka és egy henger között.	23
3.3. Simított unió különböző k paraméterek mellett.	23
3.4. C^1 és C^2 folytonosság tartó műveletekkel kapott felület Gauss-görbülete [10].	23
3.5. Sphere tracing során megtett lépések száma.	25
3.6. Skálázás hatása a felülettől vett távolságra.	28
3.7. A megjelenítést végző fragment shader összeállításának lépései.	32
3.8. Teszteléshez használt CSG gráfok és az általuk reprezentált 3D modellek.	36
4.1. Approximált és automatikus differenciálással kapott normálvektorok közötti különbség.	39
4.2. Approximációból eredő hibák görbület számításakor.	40
4.3. Tórusz (Teszt)	42
4.4. Gömbök simított különbsége (Teszt)	43
4.5. Primitívek (Teszt)	44
4.6. Figura (Teszt)	44

Algoritmusjegyzék

1.	Sphere tracing	9
2.	Rekurzív kódgenerálás váza (transzformációk nélkül)	26
3.	Rekurzív kódgenerálás (transzformációkkal)	29
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 [5]. Az algoritmus által generált duális gyökfüggvény megtekinthető A függelékben.	31