Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
IIT Tanszék

Németh Botond

ORVOSI TÉRFOGATI ADATOK HATÉKONY MEGJELENÍTÉSE TÉRFOGAT VIZUALIZÁCIÓVAL

KONZULENS

Kárpáti Attila Ádám

BUDAPEST, 2022

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Németh Botond**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2022. 12. 09.

...…….
Németh Botond

Összefoglaló

A mai világban sokszor szükség lehet orvosi CT és MRI adatok megjelenítése mellett azok fizikai szimulációjára, amely a megszokott térfogati-adat-renderelési technikákkal nem, vagy csak nehezen megvalósíthatóak.

A dolgozat ismerteti a térfogati vizualizációra alkalmas algoritmusokat, majd egy szimulációra képes megjelenítést mutat be Metaballok segítségével., ezáltal segítve olyan alkalmazásokat, amelyekben kimagasló szerepet tölt be a szervek fizikai behatásra történő elmozdulás vizualizálása. Továbbá ismerteti a sugárkövetésen alapuló Ray Marching és Sphere Tracing technológiával megjelenített Metaball fontosabb lépéseit, bemutat több alkalmazható sűrűségfüggvényt.

Továbbá a nagyméretű Metaball-számból adódó performancia-problémára használt optimalizációs technológiák elemzése után részletesebben bemutat egy a sorrendfüggetlen átlátszóságban előszeretettel használt algoritmust, az A-Buffert.

A dolgozat befejezésül mérési adatokkal bemutatja az elkészített algoritmus előnyeit és hátrányait, majd továbbfejlesztési javaslatot tesz a negatívumok javítására.

Kulcsszavak:

Számítógépes grafika, sugárkövetés, OpenGL, optimalizálás, orvosi képfeldolgozás, C++, Metaball

Abstract

In today's world, in addition to the display of medical CT and MRI data, it is often necessary to physically simulate them, which is not possible, or only difficult, to implement with the usual volumetric data rendering techniques.

The thesis describes algorithms suitable for volumetric rendering and then presents a different approach, capable of simulation with Metaballs, thus helping applications in which the visualization of the movement of organs due to physical impact plays a prominent role. Furthermore, it describes the most important steps of the Metaball rendering with ray tracing based algorithms like Ray Marching and Sphere Tracing, and presents several density functions.

Moreover, after analysing the optimization technologies used for the performance problem, caused by the large Metaball number, it dives into more detail about an algorithm that is popular among the order-independent transparency techniques, the A-Buffer.

Finally, the thesis presents the advantages and disadvantages of the final algorithm with measurement data, and then makes a further development proposal to improve the negatives.

Keywords:

Computer graphics, Ray Tracing, OpenGL, optimalization, volume rendering, C++, Metaball

1 Bevezetés

Az orvosi képalkotás koncepciója 1895-ben alakult ki, amikor egy német fizikaprofesszor, Wilhelm Rontgen feltalálta a röntgensugárzást. A röntgen ionizált sugarakon alapul, melyet a testen átengedve, a mögötte elhelyezett fényérzékeny lemezre bocsájtanak, ezzel a testről készült képet kirazolva. Az megjelenített kép segítségével képesek vagyunk a szokásostól eltérő jelenségeket (például daganatot) észlelni.

Az 1970-es években fejlesztették ki a Computed Tomographyt (CT scan). A technológia lényege, hogy a test szerkezetéről szeletenként sorozatfelvételeket készítenek, majd az eredményt térfogat vizualizációval megjelenítik számítógépes környezetben.

Szintén az 1970-es években fejlesztették ki az MRI technológiát, amely a nuclear magnetic relaxation time elvén működik. A felhasznált mágneses erők segítségével megvizsgálják a sejtekben lévő protonok elrendezését, ezzel megállapítva, hogy a test szöveteiben tapasztalható e probléma [1].

Az orvosi képalkotás hatalmas fejlődésen ment keresztül az 1895-ös évek óta. Ennek köszönhetően sokkal pontosabb orvosi diagnózisokat tudunk megállapítani feltáró műtétek nélkül. Az elváltozások és problémák korai észlelését elősegítheti az egyes szervek fizikai szimulálása. Például egy autóbalesetet szenvedett beteg belső szerveinek feltételezhető elmozdulásait is meg tudnánk állapítani, ha azt modellezve szimulálnánk.

Az orvosi adatok alapján az adott szervnek szimulálásához viszont elő kell állítanunk egy erre alkalmas megjelenítési modellt. A térfogati adatok J. F. Blinn által publikált Metaballokkal [2] történő modellezése és megjelenítése megoldást tud nyújtani erre a problémára.

2 Irodalomkutatás

2.1 Volumetric rendering

2.1.1 Bevezetés

A Volumetrikus vizualizáció egy térfogati információkat tartalmazó adathalmaz lényeges információinak megjelenítésére és feldolgozására alkalmas technológia.

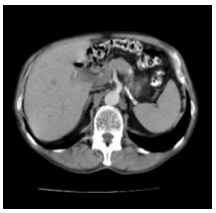
Az adatok ábrázolásához használt algoritmusok két csoportba bonthatók: Direkt Volume Rendering (DVR) és Surface-Fitting (SF). Amíg az első kategóriába tartozó technikák geometria primitívek (például pont) használata nélkül, direkt jeleníti meg a mintavételezett adatot, addig a másik csoportba tartozó algoritmusok az objektum tulajdonságai alapján renderelik ki annak izo-felületét. [3][6]

2.1.2 Térfogati adat

A Térfogati adatok általában egy *S* halmaza az *(x, y, z, v)* mintáinak, amelyek az *(x, y, z)* helyen található *v* értéket reprezentálják. Ha *v* 0 és 1 értékeit képes felvenni, akkor ebben az esetben az mintavételezett információ bináris adat, melynek 0 értéke a hátteret, 1 értéke pedig az objektumot jelenti. Az adat emellett felvehet akár több értéket is. Ebben az esetben *v* az objektum valamely mérhető tulajdonságát reprezentálja (például: sűrűség, nyomás, hő).

Mivel S egy szabványos rácson van definiálva, ezért az értékek tárolására tipikusan egy 3D tömböt szokás használni. Ha *(x, y, z)* folytonos értékeire van szükségünk, akkor ebben az esetben a *v* értéket valamilyen interpoláció segítségével határozhatjuk meg.

Ezen minta adatokat generálhatja például az orvosi képalkotás körében alkalmazott CT és MRI gépek, vagy akár a geológusok által használt Szeizmikus mérők.[5][4]



2.1 Térfogati adat egy szelete

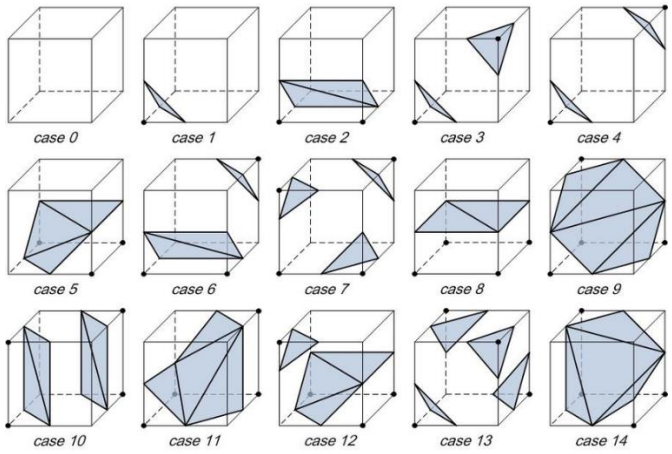
2.1.3 Technikák

2.1.3.1 Marching cubes

A Marching Cubes, vagy másnéven Masírozó Kockák, az egyik legszélesebb körben elterjedt, térfogati adatok megjelenítésére használt technika.

Az algoritmus lényege, hogy az adott orvosi adathalmaz miden voxelén végig haladunk, majd a mintavételezett kocka összes csúcspontjára megvizsgáljuk, hogy pozíciójában a térfogati adat milyen értéket vesz fel. Ha ez egy adott *T* küszöbérték felett van akkor belső, egyéb esetben külső elemnek jelöljük, majd ezen ismeretek alapján képezzük a izo-felületet közelítő háromszögeket.

Mivel minden vizsgált kockának 8 csúcspontja a korábban említett két állapot (külső vagy belső) valamelyikét veheti csak fel, így összesen $2^8 = 256$ esetet különböztethetünk meg. Mivel a 256 eset között vannak elemek, amelyek egymás szimmetrikus megfelelői, illetve egymás komplementerei. Ezzel összesen az egy voxelhez tartozó egyedi, lehetséges esetek számát redukálni tudjuk 15-re.



2.2 A kép a 15 lehetséges egyedi esetet mutatja.
A fekete pontok reprezentálják a belső pontokat.[9]

Ezt a 15 különböző esetet eltároljuk egy look-up táblában, majd minden eleméhez rendelünk egy egyedi azonosítót az alapján, hogy abban az előfordulásban a cella csúcspontjai a két lehetséges állapot melyikét vették fel. Ezáltal egyszerűen tudunk majd megfeleltetni háromszögeket a mintavételezett voxelhez.[6][8]

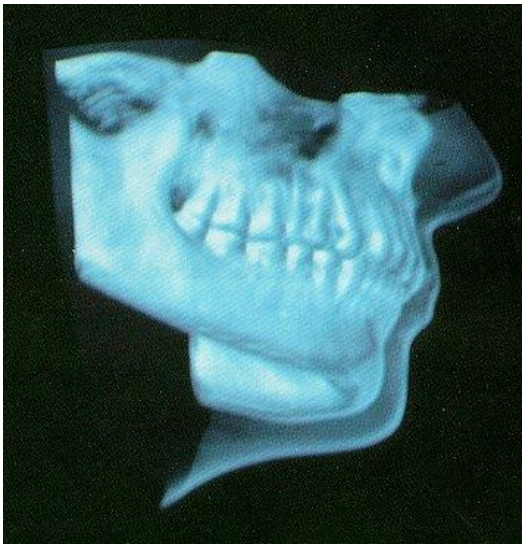
Az eredményt pontosíthatjuk, ha a generált háromszög csúcspontjait nem a cella éleinek középpontjába helyezzük el, hanem annak egy olyan pozíciójába amely kellően

közelíti a *T* küszöbértéket. Emellett az algoritmus kimenetelét a masírozó kocka méretének csökkentésével és a mintavételezés számának növelésével is javíthatjuk. [7]

Az eljárás alapját egymástól független elemek (kockák) adják, ennek köszönhetően könnyen párhuzamosítható amivel könnyedén segíthetjük az algoritmus teljesítményét, emellett az üres cellák kiszőrésével is sokat javíthatunk ezen. [8]

2.1.3.2 Ray Casting

A technika célja, hogy lehetőséget adjon 3 dimenziós adatok kirajzolására geometriai primitívek használata nélkül. Ezzel megoldást nyújthat az SF algoritmusok által nyújtott izo-felületek generálásának legnagyobb hátrányára, mégpedig, hogy nem csak a vékony felszíni rétegét jeleníti meg a vizsgált objektumnak. Így az anyag minőségét és átlátszóságát is képes egyszerűen szemlélteni. [10]



2.3 A kép egy Ray Castinggal kirajzolt állkapcsot ábrázol.
Az átlátszó rész mutatja a bőrt.[10]

Az algoritmushoz szükséges, hogy a viewport minden pixeléből képezzünk egy sugarat. Ezután meg kell vizsgálnunk, hogy az adott sugár esetében melyik az adathalmazba beérkező és melyik a távozó pontja.

A két pont előállítására több lehetőségünk van. Az egyik legegyszerűbb megközelítés a Two-Pass Ray Casting. A technika neve abból származik, hogy a vizualizálást két GPU draw hívásból valósítjuk meg:

1. Az első lépésben kiszámítjuk a sugár korábban említett két pozícióját, majd a bemeneti pontját egy *A* framebufferbe, a kimeneti pontját egy *B* framebufferbe tároljuk el.
2. A második lépésben egy a teljes viewportot lefedő vászon minden pixelében mintát veszünk az első lépésben létrehozott textúrákból. Ez fogja nekünk jelenteni a sugár kezdő- és végpontját. (Ha a pixelhez rendelt kezdő és végpont megegyezik, akkor az adott pixelhez generált sugár nem találta el az objektumot, tehát ebben az esetben a számításokat nem kell végrehajtanunk ebben a pixelben.) Ezután végig iterálunk kellően kicsi lépésekkel a két pont között, ezt a technikát hívják Ray Marching-nak. Ezután minden hozzáadott kis távolság után kialakuló pozícióban mintavételezzük a volumetrikus adathalmazunkat.

Bár ez a legegyszerűbb megközelítés, a kétszeres renderelés közel sem nyújt optimális megoldást. Egy hívásban sugár kezdő és végpontját egy, a térfogati adatot kellően közelítő geometriai test metszéspontjaival tudjuk megkapni. Ennek legegyszerűbb szcenáriója a kocka (kocka és a pixelből képzett sugár (félegyenes) két metszéspontja adja a sugár két pontját). [11]

$$RayDir = (E * V * P)^{-1}$$

2.1 Ray Direction mátrix képlete

- E a kamera pozíciójából képzett eltolás
- V a view mátrix
- P a projekciós mátrix

$$p = o + t * d$$

2.2 A sugár p pozíciójának kiszámítása.
O jelenti a sugár kezdőpontját és d a annak irányát.

- O a sugár középpontja
- d a sugár iránya

A viewport adott pixeléből indított sugár kiszámításához először a Ray Direction mátrixot kell kiszámolnunk a (2.1)-es képletnek megfelelően. Ezután a pixel pozícióját e mátrixxal beszorozva megkaphatjuk a pixelből indított sugárirány vektorát.

2.2 Ray Marching

2.2.1 Bevezetés

Egy jól ismert algoritmus komplex testek kirajzolására, melyhez elegendő információ egy pont és egy objektum felületének távolsága, a Ray Marching.

A Ray Marching Signed Distance Function-ökre (röviden: SDF), azaz előjeles távolságfüggvényekre épít. Ha például egy kört szeretnénk kirajzolni, akkor a következő képletet kell leimplementálnunk: [12]

$$\|\boldsymbol{p} - \boldsymbol{c}\| - \boldsymbol{r} < \boldsymbol{\varepsilon}$$

2.3 A kör távolság előjeles távolság-egyenlete

- p a vizsgált pont
- c a kör vagy gömb középpontja
- r a kör vagy gömb sugara
- ε egy kellően kicsi küszöbérték

A vizsgált pont sugárirányú elmozgatásához két lehetőségünk is van. Egyik a lineáris Ray Marching amely kellően kicsi lépésekkel lépteti tovább a pontot, a másik a Sphere Marching amit a későbbiekben részletesebben kifejtek.

$$\begin{cases} step = d * \min\left(\frac{t_{end} - t_{start}}{n_{step}}; \varepsilon\right) \\ p = p + step \end{cases}$$

2.4 Lineáris Ray Marching léptetésének képlete

- d a sugár iránya
- t_{end} a sugár kimeneti pontjának távolsága a kamerától
- t_{start} a sugár bemeneti pontjának távolsága a kamerától
- ε egy kellően kicsi küszöbérték
- p a vizsgált pont

A Ray Marching mindezek mellett lehetőséget ad számunkra, hogy szinte ingyen (nagy számítási igény nélkül) tudjunk akár lágy árnyékokat ábrázolni. Az ehhez használt legegyszerűbb eljárás, ha egy adott felületi metszéspontból a fényforrás felé mutató vektor irányába elindulunk. Ha a menetelés közben eltaláljuk a világ egy objektumát, akkor tudjuk, hogy a vizsgált pontot árnyékolnunk kell. [13]

```
float softshadow( in vec3 ro, in vec3 rd, float mint, float maxt, float k )
{
    float res = 1.0;
    for( float t=mint; t<maxt; )
    {
        float h = map(ro + rd*t);
        if( h<0.001 )
            return 0.0;
        res = min( res, k*h/t );
        t += h;
    }
    return res;
}
```

2.1 Az árnyékolás egy implementációja Sphere Tracing használatával[13]

2.2.2 Signed Distance Functions

Az egyenlet egyik hatása, ha az adott pont (melyre a képletet alkalmaztuk) az objektum belsejében található, a végeredmény negatív lesz. Ezert nevezik előjeles távolságfüggvénynek.

Signed Distance Function-ök sok geometriai testhez léteznek. Példaul Torus, Cylinder, Gömb[15]. Ilyen képletek segítségével különböző alakzatoknak akár vehetjük az unióját, metszetét, különbségét. Ezek segítségével igazán komplex objektumokat is létre tudunk hozni.[14]

2.2.3 Normálvektor

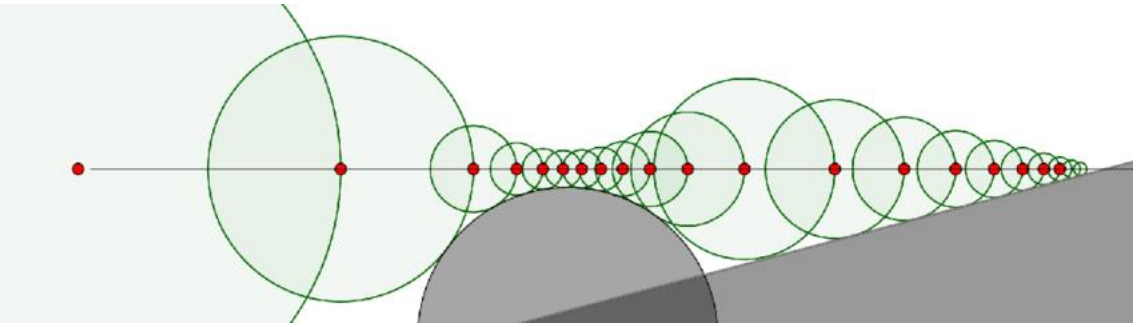
Ray Marching esetében a normál vektor a lágy árnyékoláshoz hasonlóan, szinte ingyen kiszámítható.

A normálvektorhoz a kiszámított felületi pontból minden koordinátatengely irányába kellően kicsi $\pm\varepsilon$-nal eltolt pontot mintavételezünk. A két vektor különbségének normalizált alakja meg is adja az objektum adott pontjának normálvektorát.[12]

2.2.4 Sphere Tracing

A Signed Distance Function segítségével a megjeleníteni kívánt világ bármely pontjából meg tudjuk határozni, hogy mekkora távolságra van a legközelebbi objektum. Ha ezt a kamera pozíciójából indított sugár egy tetszőleges pontjára alkalmazzuk, akkor ezáltal meg tudjuk állapítani, hogy abból a pozícióból mekkora távolságot haladhatunk előre a sugár irányába.

Ha a legközelebbi test felszínétől mért távolság elérte az előre definiált ε küszöbértéket, akkor elértük az objektumot. A Ray Marching ezen változatát Sphere Tracingnek nevezik. [12]



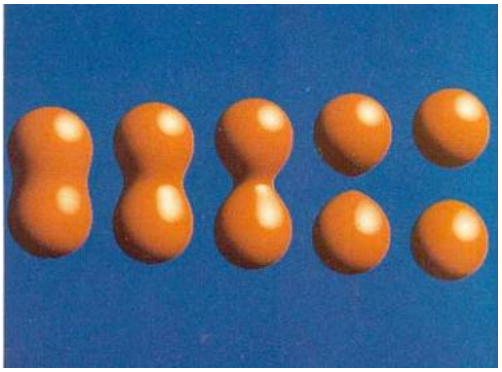
2.5 Sphere Tracing ábrázolása 2D-ben[16]

A Sphere Tracing sokkal gyorsabb lefutást és kisebb számítási igényt eredményez, hiszen nem csak kellően kicsi távolsággal lépünk előre ezáltal.

2.3 Metaballok

2.3.1 Bevezetés

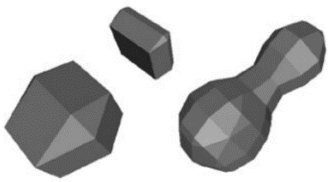
A Metaballok részecskeszimulációk, lágytestek és implicit felületek megjelenítésének széles körben elterjedt eszköze. A Metaballokat, mint technológiát J. F. Blinn amerikai tudós alkalmazta először molekuláris modellek megjelenítéséhez. (Blinn eredetileg a Metaballokra, a „blob” kifejezést használta.) [2]



2.3 Jim Blinn által "blob"-oknak nevezett metaballok[2]

A Metaballok megjelenítéséhez, a térfogati adatok vizualizációjához hasonlóan, két lehetőségünk van: izo-felületek generálásával a Metaballok felszínén, vagy Ray Marching és sűrűségfüggvények segítségével. [17]

A Metaballok felületének geometriai primitívekkel való leképzéséhez használt algoritmusok (például a korábban bemutatott Marching Cubes) voxelizációs folyamatának köszönhetően szögletesnek mondható jelenséget tapasztalhatunk nagy méretű mintavételezésnél. Nagy felbontás és sok objektum esetén jelentős memória és számítási kapacitás-igénye mellett az algoritmus által meghatározott rács rendszernél kisebb objektumok akár el is veszhetnek.. Ezzel ellentétben a sugárkövetés alapú megközelítések (például Ray Marching) sokkal szebb, simább felületet eredményezhetnek, viszont a hosszú számítási idővel ebben az esetben is találkozhatunk.[18]



2.4 Alacsony felbontású Marching Cube tecnológiával készített metaballok[19]

2.3.2 Sűrűségfüggvények

A Metaballokkal való modellezésnek és azok megjelenítésének egyik kulcsfontosságú eleme a sűrűségfüggvények. Az évek során különböző tudósok, mint például Nishimura, Wyvill vagy J. F. Blinn, különböző szemszögből próbálták meg definiálni a Metaballok térfüggvényit.[20]

J. F. Blinn ötletének alapja az volt, hogy a molekulák izo-felületét szerette volna megjeleníteni. A sűrűségfüggvény képletét végül a hidrogén atom sűrűségfüggvényéből származtatta. [2][17]

$$f_i(r) = e^{(-ar^2)}$$

2.5 J. F. Blinn által definiált density function[2]

- r a vizsgált pont és a Metaball középpontjának távolsága
- a egy tetszőlegesen választott konstans

A leggyakrabban használt sűrűségfüggvények közé tartozik Nishimura kvadratikus térfüggvénye, Wyvill eredetileg lágytestek szimulálására használt képlete [21] és Murakami egyszerű sűrűségfüggvénye.

$$f_i(r) = \begin{cases} 1 - 3 * \left(\frac{r}{R_i}\right)^2, & \frac{R_i}{3} \geq r \geq 0 \\ \frac{3}{2}\left(1 - \left(\frac{r}{R_i}\right)^2\right), & R_i \geq r \geq \frac{R_i}{3} \end{cases}$$

2.6 Nishimura által definiált density function

$$f_i(r) = -\frac{4}{9}\left(\frac{r}{R_i}\right)^6 + \frac{17}{9}\left(\frac{r}{R_i}\right)^4 - \frac{22}{9}\left(\frac{r}{R_i}\right)^2 + 1$$

2.7 Wyvill által definiált density function

$$f_i(r) = \left(1 - \left(\frac{r}{R_i}\right)^2\right)^2$$

2.8 Murakami által definiált density function

- r a vizsgált pont és a Metaball középpontjának távolsága
- R_i az i Metaball sugara

A Metaballok egy halmazát az egyes Metaballok sűrűségfüggvény-értékeinek összegzésével reprezentálhatjuk. A felület ezután abban a pontban jeleníthető meg, amely kielégíti a következő egyenletet.[2]

$$f(r) = -T + \sum_{i=0}^{n} f_i(r) = 0$$

2.9 A Metaballok implicit egyenlete

- T egy küszöbérték
- f_i az i Metaball sűrűségfüggvénye
- n a Metaballok száma

2.3.3 Optimalizálás

2.3.3.1 Bevezetés

A sugárkövetésen alapúló algoritmusok képesek sima felületeket nagy minőségben megjeleníteni akár kisebb méretű objektumokat is.

Azonban ezek a technikák általában nagy számítási költséggel járnak, melynek mértéke elsősorban a megjelenített kép felbontásától függ. Emiatt nagy számú Metaballok esetén valós időben, önmagában nem alkalmazhatóak. A valós idejű kép előállításának érdekében különböző gyorsítási technikákat szükséges alkalmaznunk. [22][23]

Az egyik lehetséges optimalizálási megközelítés, a pixelenkéni Metaballok számának redukálása. A csökkentést elérhetjük azáltal, ha a költséges izo-felület metszéspontjának tesztelésénél csak a kamerapozícióból indított sugár által elmetszhető objektumokat vesszük figyelembe.

Erre a problémára nyújtanak megoldást az elsősorban Order-Independent-Transparency-hez (röviden: OIT), azaz Sorrend Független Átlátszósághoz használt algoritmusok az A-Buffer és S-Buffer.

A két eljárás pixelenkénti, akár többszintes, információk tárolására alkalmas. Ennek köszönhetően lehetőségünk van ezeket az algoritmusokat, az egy pixelhez tartozó releváns Metaballok előszűrésére használni.[24][25]

2.3.3.2 A-Buffer

Az A-Buffer a számítógépes grafikában egy rejtett objektumok detektálási metódus, amely közepes méretű virtuális memóriával rendelkező környezetek estén alkalmazható. A módszert Anti-Aliased vagy Area-Averaged bufferként is ismerik

Az algoritmus 2 kulcsfontosságú strukturális elemre épít. Az egyik strukturális elem egy 2 dimenziós tömb, a másik pedig egy láncolt lista.

A 2 dimenziós tömb mérete a megjeleníteni kívánt kép pixeleinek számával egyenlő. Minden eleme (legegyszerűbb esetben) egy előjel nélküli egész számot tárol. Ez a szám jelenti az általa reprezentált pixelhez tartozó láncolt lista első elemének azonosítóját.

A láncolt lista egy eleme 2 releváns információt tárol. Egyrészt tartalmazza az általa reprezentált objektum tulajdonságait, például az Objektum színét, kamerától való távolságát vagy pozícióját, másrészt a láncolt lista következő elemét.

2.5 A láncolt lista egy reprezentációja[26]

Az algoritmus egyik legnagyobb hátránya, hogy nagy mennyiségű adat vagy nagyobb felbontás esetén sok memóriára van szükség. A lefoglalt memória mennyiségén javíthatunk ha megadjuk, a pixelenkénti maximális lefoglalható memória nagyságát. Ebben az esetben rendezést szükséges alkalmaznunk, hogy a maximális méretű elemek közé elsősorban a kamerához közelebb lévő objektumok kerüljenek be. [26] A rendezésből származó többletidőn, nagyméretű objektum-halmaz esetén javíthatunk, ha az A-Buffer láncolt lista struktúrája helyett Bináris fába építjük fel. Ebben az esetben célszerű a fa egyes elemeit mélység alapján rendezetten tárolni.[27]

Ámbár ez javít a memóriahasználaton, figyelembe kell vennünk, hogy ez a megközelítés nem minden esetben jelent számunkra megoldást. Lehetnek olyan szituációk, amikor értékes információt veszítünk ezáltal.

2.3.3.3 S-Buffer

Az A-Buffer memóriaigényes negatívumára megoldást nyújthat az S-Buffer. Az S-Buffer egy hatékony és memóriabarát algoritmus, amely az A-Buffer architektúrájára épül, anélkül hogy láncolt listákat és fix hosszúságú tömböket használna.

Ennek elérése érdekében az S-Buffer algoritmus az A-Bufferhez képest egyel több renderelési lépést hajt végre, amelyben megvizsgálja pontosan mekkora memóriára van szükség.

Megfigyelhetjük hogy ámbár memóriahasználatban sikerült javulást elérnünk, a plusz egy renderelési hívással akár nagy mértékű lassulást tapasztalhatunk. [28]

2.3.3.4 Említésre méltó algoritmusok:

Kanamori és társai az egyes pixelekhez tartozó objektumok számának csökkentése mellett a Metaballok felületének gyors tesztelésével közelítették meg a problémát. Ennek elérése érdekében a látósugarak és a Metaballok izo-felületeinek metszéspontjain végeztek Bezier Clippinget.[22]

A későbbiekben az algoritmuson L. Szécsi és D. Illés, az eljárás legköltségesebb aspektusának, a Bezier Clippingnek, köbös sűrűségfüggvény közelítéssel való lecserlésével gyorsítottak.[18]

3 Tervezés

3.1 Framework

3.1.1 Bevezetés

A projekthez a "boilerplate" kódok elrejtése (shaderek betöltése, erőforrások bindolása), a könnyebb bővíthetőség és átláthatóbb struktúra érdekében készítettem egy egyszerűbb keretrendszert, amely statikus könyvtárként (lib) kerül a projektbe.

A keretrendszer egyszerűbb geometriák kirajzolása mellett segíti a fejlesztőt a háttérben történő OpenGL hívások elrejtésével A vektorok és mátrixok struktúrált kezelése érdekében az OpenGL-hez készített GLM [29] könyvtárat használom.

3.1 Keretrendszerrel megjelenített egyszerűbb jelenet

3.1.2 Fontosabb elemek

3.1.2.1 Uniformok

OpenGL-ben a uniformok kezelése nagyobb szám esetén könnyen olvashatatlanná tudja alakítani a kódot. Ennek megelőzésére a Frameworkben létrehoztam egy Uniform osztályt, amelynek minden leszármazottja egy glsl-ben megtalálható változó típust reprezentál.

Egy Objektumhoz tartozó uniformokat az ahhoz tartozó Material osztály tárolja. Ezeket később a Program osztály segítségével állítja be. A uniformok beállításához szükségünk van arra, hogy a GPU-n hol található. A Program osztály az adott uniformhoz tartozó helyet egy rendezetlen map struktúrában cacheli ezzel gyorsítva annak elérését.

3.1.2.2 Textúrák

A keretrendszer 2 és 3 dimenziós textúrák létrehozását teszi lehetővé. A Textúra osztály példányosításakor egy lokális bufferbe tölti be a képet az stb_image [30] könyvtár segítségével, majd a továbbiakban a glTexImage metódus megfelelő változatával létrehozza az adott textúrát.

3D textúrák esetén az adatok betöltése után, azokat átalakítja olyan formába, hogy azt később az OpenGL megfelelő módon jelenítse meg.

Uniformokhoz hasonlóan a textúrákat is a Material osztály tárolja, és állítja be a Program segítségével.

3.1.2.3 Bufferek

Az OpenGL bufferei kezelésének összetett kódjára elkészített Buffer osztályok Victor Gordan struktúrája épülnek[31], ezzel megkönnyítve azok menedzselését. Ezen közül a két legfontosabb említésre méltó osztály a Shader Storage Buffer Object (SSBO) és az Atomic Counter Buffer (ACB).

A Shader Storage Buffer Object egy nagy, akár 128 MB nagyságú adathalmaz tárolására alkalmas Buffer Object. Az SSBO egyes elemein végrehajthatunk speciális atomikus metódusokat, mint például az atomicAdd vagy atomicExchange. Egyik legfontosabb tulajdonsága, hogy std140-es layout alkalmazásánál a feltöltött elemek struktúrájának mérete 16 byte többszörösének kell lennie, mivel a tömb típusok elemei nem feltétlenül vannak a C++-hoz hasonlóan összecsomagolva. [32][34]

Az Atomic Counter Buffer, egy számláló, amelyen atomikus műveleteket tudunk végrehajtani, mint ahogy a neve is arra enged következtetni.[33]

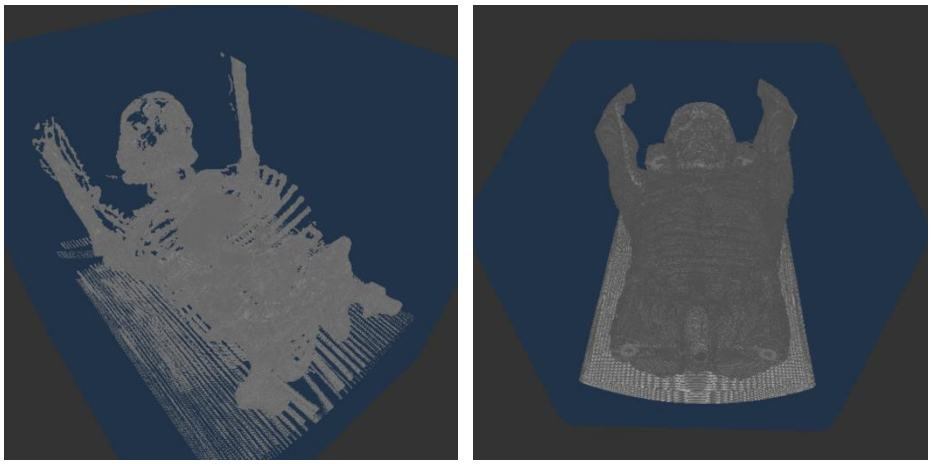
3.2 Volume rendering

3.2.1 Ray Marching a Volume renderingben

A feldolgozandó volumetrikus adatot egy kép file-ból olvasom be amely tartalmazza az objektum szeleteit. A szín intenzitása az adott pontban mintavételezett anyag sűrűségével egyenlő. A kép betöltése után a Texture3D osztály segítségével egy 3 dimenziós textúrát képzek az egyes szeletekből.

Eleinte a térfogati adathalmazt lineáris Ray Marching technológiával jelenítettem meg. Mivel a textúra egy [0; 1] intervallumon kívül eső pontban is mintavételezhető (ezzel megismételve a textúra [0; 1] intervallumon felvett értékeit) a vizualizált adat végtelen sokszor jelent meg. Ennek megelőzése céljából a releváns adatot csak egy őt körülvevő kockában értékeltem ki.

Az algoritmus előnye, hogy az egyszerű implementálás mellett részletgazdag képet kaphatunk. Emellett a technológiával lehetőségünk van csak megadott intenzitással rendelkező értékek megjelenítésére. Hátránya viszont hogy a szerv adott pontjának, fizikai behatásokból származó elmozdítását nem tudjuk megjeleníteni.



3.2 Azonos adatok különböző sűrűségekre való szűréssel

A későbbiekben a szokásostól eltérő megközelítéshez folyamodtam, a térfogati adatok, lágytestek szimulációjára használt, Metaball technológiával való vizualizációjához. Ez a technológia kellően nagy Metaballszám mellett lehetőséget ad viszonylag részletesen kirajzolható és fizikailag szimulálható eredmény modellezésére. Ennek oka, hogy a Metaballokat akár egymástól függetlenül is meg tudjuk jeleníteni vagy szimulálni.

3.3 Metaballok

3.3.1 Bevezetés

A Metaballok a számítógépes grafika, részecskeszimulációinak egyik legmeghatározóbb elemei. A Metaballokkal lehetőségünk van lágytestek megjelenítésére, mint ahogy azt Wyvill [21] publikációjában is olvashatjuk.

A Metaballok vizualizálásának 2 nagyobb csoportjából, az izo-felület generálásból és sugárkövetésből (melyeket részletesebben a 2.3-as pontban kifejtek), az utóbbit választottam, a nagy grid rendszerból eredő szögletes forma elkerülése és a kisebb objektumok esetében is pontos eredmény megjelenítése érdekében.

3.3.2 Sűrűségfüggvény

A sugárkövetés megközelítések a Metaballok density-function-jére építkeznek. Az évek során több ember különböző térfüggvényekkel próbálta meg vizualizálni a Metaballokat. A 2.3.2-es pontban bemutatott egyes sűrűségfüggvényei nagy objektumhalmaz esetén eltérő performanciát tudnak eredményezni.

Az egyik legszélesebb körben használt sűrűségfüggvény a Wyvill és társai által megalkotott hatod fokú egyenlet.[21] A képlet előnye, hogy véges R_j sugár esetén ha $r \geq R_j$, akkor $p_j(r) = 0$. Ezen kívül, ha $r < R_j$, akkor pedig egy polinomiális futásidejű egyenlet jellemzi, amely biztosítja az egyszerű, költséghatékony kiértékelést.

Ezeket az előnyöket Perlin ötödfokú változata is biztosítja, viszont észrevehetjük, hogy Wyvill és társai által definiált képlet minden vektorhosszúságot tartalmazó tagja páros hatványkitevőn szerepel. Ennek köszönhetően a képletet átalakítva elkerülhetjük a gyökvonásból eredő számítási időt.[18]

$$f_i(r) = \begin{cases} 0, & R_i \leq r \\ 1 - \left(\frac{r}{R_i}\right)^3 * \left(\frac{r}{R_i}\left(6 * \frac{r}{R_i} - 15\right) + 10\right), & R_i > r \end{cases}$$

3.3 Perlin ötödfokú sűrűségfüggvénye

- r a vizsgált pont és a Metaball középpontjának távolsága
- R_i az i Metaball sugara

```
float wyvillMetaball(float distance, float radius) {
     if (distance > pow(radius, 2)){
          return 0.0f;
     }
     float fi = 1.0f - 3 * distance / pow(radius, 2) + 3 * pow(distance, 2)
/ pow(radius, 4) - pow(distance, 3) / pow(radius, 6);
     return fi;
}

float wyvillMetaballTest(vec3 p, vec4 metaballCenter, out vec3 color) {
     float squaredDistance= dot(p - metaballCenter, p - metaballCenter);
     float f = wyvillMetaball(squaredDistance, 0.2f);
     color += f * metaball.color;
     return f;
}
```

3.4 Gyorsított Wyvill sűrűségfüggvény implementációja

Az eljárás implementálásánál a négyzetes távolságot a két pont (Metaball középpontja és a vizsgált pont) különbségéből származó irányvektorból számoltam. A vektorból az önmagával vett skalárszorzatával képeztem a négyzetes távolságot.

Ezután fontos, hogy a kritériumot is a módosított függvényhez igazítsuk, a sugár négyzetre való emelésével. Ezt azért tehetjük meg mivel a távolság és az adott Metaball sugara soha nem lehet negatív szám.

Mint látható a Metaball tesztelésére elkészített metódus az összesített sűrűség mellett az adott pont színét is kiszámítja. Ennek folyamata nagyon egyszerű, minden Metaball színét annak területfüggvényével súlyozza. Annak érdekében, hogy a szín kiszámításához ne kelljen még egyszer végig iterálnunk az alkotott világ minden Metaball objektumán glsl lehetőséget ad számunkra, hogy akár több értéket is visszaadjunk (nem visszatérési értékként). Ezt a paramétert megelőző out leíróval tudjuk elérni.

3.3.3 Ray Marching Metaballokhoz

Az egyes Metaballokat egy sugárkövetésen alapuló algoritmusra építettem, a Ray Marchingra. Kezdetben egy egyszerűbb, lineáris változatával rajzoltam ki az objektumokat.

Ebben az esetben a sugár 2 előre definiált (be- és kimeneti) pontja között egyenlő lépésekkel mozogtam előre. A 2 pontot előre definiált konstans távolságok alapján számoltam ki.

Ezt a későbbiekben egy sokkal elegánsabb megoldással, a Sphere Tracinggel váltottam le. A Sphere Tracinggel akkor detektálunk egy izo-felületet, ha az algoritmus

alapján, a legközelebbi objektumhoz kiszámított távolság elért egy konstans küszöbértéket.

Gömbök esetén a következő megoldással tudjuk meghatározni az adott pontból mért távolságot.

$$\|p - c_i\| - r_i$$

3.5 Gömbtől mért távolság képlete

- p a vizsgált pont
- c_i a kör vagy gömb középpontja
- r_i a kör vagy gömb sugara

Észrevehetjük, hogy ámbár az algoritmus egyszerű gömb objektumokra jól működik, Metaballok esetén a detektált pontban nem feltétlenül kapunk 0-nál nagyobb összesített sűrűség-értéket. Ennek elkerülése érdekében szükséges definiálnunk egy minimális konstans lépéstávolságot.

$$p = p + d * \max(s, s_{min})$$

- p a vizsgált pont
- d a sugár irányvektora
- s a legközelebbi gömbtől vett távolsága
- s_{min} a minimális konstans lépéstávolság

3.3.4 Modell Felépítése

3.3.4.1 CPU

A Volumetrikus adatok alapján felépített Metaball modellt először CPU-n készítettem el. A Metaballok középpontjának pozícióját egy std::vector tömbben tároltam, amit később egy Shader Storage Bufferbe betöltve, majd azt a megfelelő lokációra bindolva tudtam elérni az egyes Shaderekből. A pozíciókat egy 4 elemű vektorban (vec4) tároltam, az SSBO std140-es 16 bytos struktúra-megkötése miatt.

A későbbiekben felismertem, hogy a Metaballokat egymástól függetlenül is fel lehet dolgozni. Erre egy több szálú rendszer jelentené az ideális megoldást

3.3.4.2 Compute Shader

OpenGL-ben van lehetőségünk egy a szokásos GPU csővezetéktől függetlenül, többszálon futtatható shader létrehozására, amit Compute Shadernek neveznek.

Megszokott megjelenítő csővezetéktől függetlenül fut és nincsenek a felhasználó által definiált bemeneti, viszont beépített bemenetekkel rendelkezik. Ha a Shadernek mégis szüksége lenne bemeneti értékekre, akkor például textúrákkal és Shader Storage Bufferekkel megadhatjuk azokat. Ehhez hasonlóan egy Compute Shader számításainak kimeneteit szintén egy képpel vagy SSBO-val tudjuk elérni.

Típus	Név	Leírás
uvec3	gl_NumWorkGroups	Work Groupok száma
uvec3	gl_WorkGroupID	Az invokáció Work Groupjának ID-ja
uvec3	gl_LocalInvocationID	Invokáció egyedi ID-ja egy Work Goupon belül
uvec3	gl_GlobalInvocationID	Invokáció globálisan is egyedi ID-ja
uint	gl_LocalInvocationIndex	gl_GlobalInvocationID 1 dimenziós változata

3.6 Compute Shader beépített bemeneti változói

A Compute Shader Work Groupokból épül fel. A Felhasználó definiálhatja, hogy a hány Work Groupot szeretne futtatni. Ezek futásának sorrendje nem egyenletes, tehát például lehetséges, hogy a (2,4,5)-ös után az (1,1,1)-es Work Group kerül lefutásra.

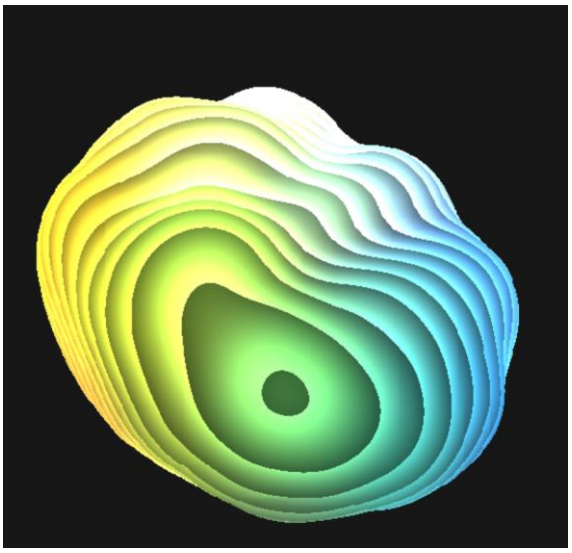
Egy Work Groupon belül akár több Shader Invokáció is lehet. A különböző invokációk egymástól eltérően és párhuzamosan futnak.[35]

3.3.4.3 GPU

A Compute Shader program létrehozást, a szükséges metódushívásokat és buffereket egy MetaballCreator nevű osztályba fogtam össze. Az osztály tárolja a létrehozni kívánt Metaballok számát, egy Shader Storage Buffert a pozíciók tárolására és egy Atomic Counter Buffert, melynek szerepét a továbbiakban részeletezem.

A Compute Shaderben egy uniformként feltöltött 3D textúrán haladok végig. A Compute Shader minden invokációja a textúra egy voxeléért felelős. Az invokáció-voxel hozzárendelést textúrakoordináta-generálás segítségével oldom meg. Minden invokációnak van egy egyedi uvec3-ban reprezentált azonosítója, melyet gl_GlobalInvocationID-nak neveznek. Ennek felhasználásával minden invokációhoz tudunk rendelni egy textúrakoordinátát, oly módon, hogy az invokáció gl_GlobalInvocationID-ját leosztjuk a workgroupok számával.

Ha az adott szálhoz tartozó textúra pozíciójában kiolvasott intenzitás elér egy megadott konstans értéket, akkor abban az esetben generál egy Metaballt, melynek pozíciója a textúrakoordinátával lesz egyenlő.



3.7 Egy agy formáját közelítő Metaballok

Mivel a Compute Shaderekben a konkurens környezetek egy szélsőséges helyzetével találkozunk, ezért fontos, hogy figyeljünk az erőforrások használatának párhuzamos hozzáférésből eredő nehézségeire.

Annak elkerülése érdekében, hogy 2 szál az SSBO azonos indexű eleméhez szúrja be az általa létrehozott objektum pozícióját, ezzel felülírva az előző értéket, létrehoztam egy Atomic Counter Buffert, amit minden szál objektum-létrehozáskor megnövel egyel. Ez azért vezet minket megoldásra, mert a rajta végzett atomikus műveletekkel annak értékét egyszerre csak 1 szál tudja módosítani. Ezután az Atomic Counter Buffer értéke fogja jelenteni az új metaball SSBO-indexét.

```
void main() {
    vec3 texCoord = gl_GlobalInvocationID/ volumeDimension;
    float h = 0.0f;
    h = texture(volumeData, texCoord).r;
    if (h > 0.3f) {
        vec4 metaball = vec4(texCoord, 1.0f);
        uint currentCounter = atomicCounterIncrement(counter);
        position[currentCounter] = metaball;
    }
}
```

3.8 Metaballokat generáló Compute Shader program

Bár a Metaballok megjelenítésén 2 pontban (Sphere Tracing és módosított Wyvill density function) is gyorsítottunk, korántsem értük el azok gyors, elfogadható számú vizualizálását. Ezért egyéb optimalizálásokat kell alkalmaznunk.

3.4 A-Buffer

3.4.1 Bevezetés

Metaballok esetén a legnagyobb eredményeket optimalizálás terén, a Metaball-tesztelések számának csökkentésével vagy annak valamilyen közelítésével tudjuk elérni. Eddig minden kamerapozícióból indított sugár minden lépésében megvizsgáltuk az elkészített világ összes objektumát, azokat is melyeknek feltehetően nincs vagy csak elenyésző hatása van kirajzolt objektum megjelenítésére.

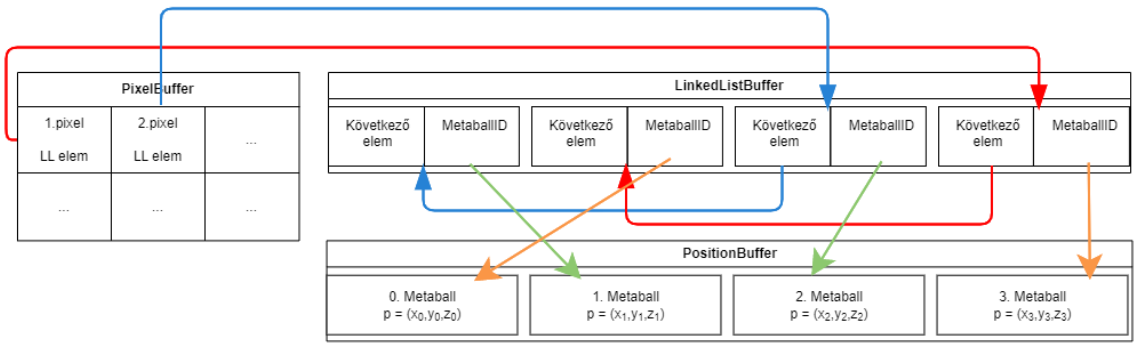
A problémára megoldást nyújthatna egy olyan algoritmus, amely pixelenként a releváns Metaballokat kiszűrve tudná csökkenteni azok teszteléseinek számát.

A 2.3.3.2 pontban részletesen bemutatott A-Buffer technika pixelenkénti információ tárolására alkalmas. Ennek köszönhetően meg tudnánk adni, hogy az egyes pixelekből indított sugár mely Metaballokat metszi el.

3.4.2 Felépítés

Az A-Buffer struktúrája három Shader Storage Bufferből és egy Atomic Counter Bufferből épül fel.

A három SSBO a ScreenBuffer, a LinkedListBuffer és PositionBuffer nevekre hallgatnak. A LinkedListBuffer elemei uvec2 típusúak, a és a láncolt lista egyes elemeit reprezentálják. Az elemek első adattagja tárolja a láncolt listában őt megelőző elem indexét, második eleme a láncolt listához tartozó Metaball indexe. A ScreenBuffer előjel nélküli egész számok segítségével tárolja a pixelhez tartozó láncolt lista utolsó elemét. PositionBuffer a Compute Shader által elkészített Metaballok pozícióit tárolja.



3.9 Bufferek kapcsolatának ábrája

- PixelBufferben az „LL elem” jelenti a pixelhez tartozó láncolt lista utolsó elemét
- LinkedListBufferben található „MetaballId” jelenti az adott elemhez tartozó Metaball azonosítóját
- A „Következő elem” a láncolt lista eleméhez viszonyított következő listaelem indexét tárolja
- A piros nyilak az első pixelhez tartozó láncolt listát jelentik, a narancssárga ennek elemeihez tartozó Metaballokat
- A kék nyilak a második pixelhez tartozó láncolt listát jelentik, a zöld ennek elemeihez tartozó Metaballokat

Az Atomic Counter Bufferre a Compute Shaderhez hasonlóan a láncolt lista bufferének megfelelő kezelése miatt van szükség.

Az A-Buffer algoritmushoz használt strukturális elemeket egy ABuffer osztályba gyűjtöttem. Az osztály a bufferek mellett azok elemeinek számát és az azokhoz tartozó lokációt is tárolja. Az A-Bufferrel optimalizált megjelenítésnek két külön szakaszát (struktúra és adatok felépítése, majd azok használata) két különböző GPU Program kezeli. A bufferek megfelelő használatához a shaderben megadott helyre kell őket feltöltenünk, amit megelőz a szakasz Sahdereiért felelős Program bindolása. Emiatt az Abuffer osztály a 2 szükséges Program mutatóját is eltárolja. A sok adattag inicializálásából kialakuló hosszú konstruktor-paraméterlista elkerülése érdekében az objektum példányosítását az osztály köré épített builderrel tudjuk megoldani.

Az SSBO elemeit egy paddinget is tartalmazó struktúrában volt szükség eltárolni, hogy elérjék a 16 bytos keretet. Ennek köszönthetően a struktúrák a következőképpen épülnek fel:

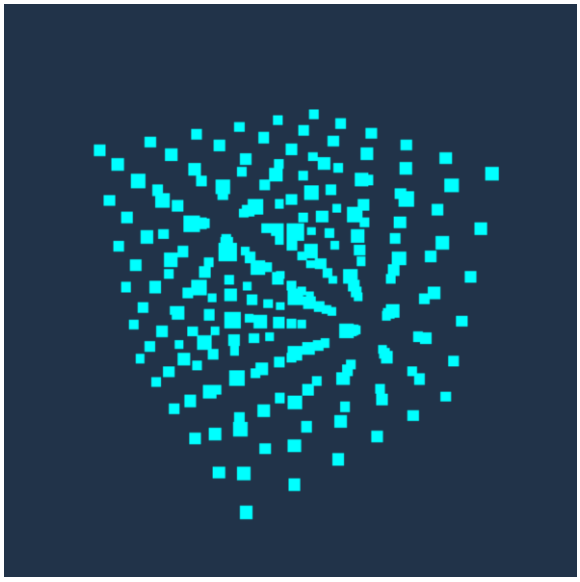
```
struct uint_element {
     unsigned int pixel = 0;
private:
     float __padding[3] = { 0.0f, 0.0f, 0.0f };
};
struct uvec2_element {
     glm::uvec2 element = glm::vec2(0);
private:
     float __padding[2] = { 0.0f, 0.0f };
};
```

3.10 SSBO uint és uvec2 elemeinek tárolására létrehozott struktúrák

3.4.3 Feldolgozás

Az adatok struktúra-felépítésének első lépésében a Vertex Shaderben kiolvasom a megfelelő Metaball középpontjának pozícióját. Ezt úgy valósítom meg, hogy a glDrawArrays metódusban megadtam a Metaballok számát, mint vertexek száma, így elérve, hogy a Vertex Shader pontosan ezzel megegyező számban fusson le. Ezután a gl_Position beépített változó értékének a PositionBuffer gl_VertexID-edik elemét állítottam be. Emellett definiálok egy uint típusú Vertex Shader kimenetet metaballId néven, amely a gl_VertexID-val lesz egyenlő.

Annak érdekében, hogy egy pixelből látható Metaballok esetén aktiválódjon a Fragment Shader, ezzel felépítve az előzőekben részletezett struktúrát, szükségünk van valamilyen geometria létrehozására a reszterizációs egység előtt. Ennek legegyszerűbb módja, ha egy Geometry Shaderben a középpontok körül Metaball-átmérőjű billboardokat hozunk létre. Emellett fontos, hogy még a render hívás előtt kikapcsoljuk a mélységtesztelést, hogy ezáltal megakadályozzuk, azt, hogy csak az első elemeket vegye figyelembe a Fragment Sahader.



3.11 Egy 6x6-os kockán felvett Metaballok billboardjai

A billboardok létrehozását úgy valósítom meg, hogy a Geometry Shaderben a Metaballok középpontjait normalizált eszközkoordinátarendszerbe transzformálom át. Ezután x és y tengely pozitív és negatív irányba eltolt pozíciójukon felvett pont primitívekből állítom őket össze.

Ezután a Fragment Shaderben először megnövelem az Atimoc Counterben tárolt számot, a Compute Shaderben is használt atomicCounterIncrement függvény segítségével. Az így kapott érték fogja jelenteni a láncolt lista új elemének azonosítóját. Majd ha a láncolt lista elemeit tároló buffer nem érte el annak maximális kapacitását, akkor ezt a számot egy atomicExchange metódus segítségével áthelyezem a PixelBuffer adott pixelt reprezentáló elemének helyére.

Ezt a tagot a pixelhez generált egyedi azonosítóval, pixelID változóval érem el. A pixelID-t a gl_FragCoord beépített uvec2 típusú attribútummal generálom a következő módon:

```
uint pixelId=uint(gl_FragCoord.y) * (windowSize.x-1) + uint(gl_FragCoord.x);
```

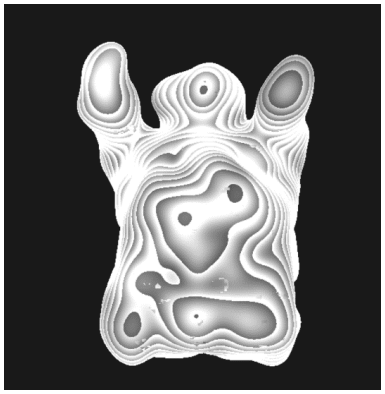
3.12 Pixel egyedi azonosítójának generálása

Ezt követően a láncolt listát egy új elemmel bővítem, melynek az előzőre mutató értéke az új index által lecserélt szám lesz.

3.4.4 Megjelenítés

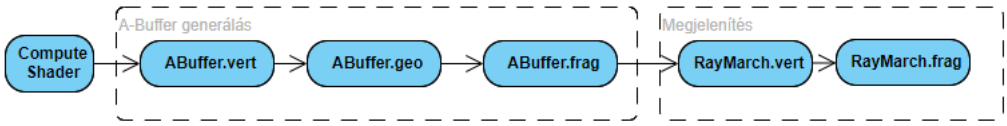
A render hívás előtt az ABuffer osztály changeProgram metódusa segítségével átállítom annak aktív GPU programját arra amelyik az algoritmus második (megjelenítés a felépített adatok alapján) szakaszáért felelős, majd a buffereket a Shaderben magadott helyükre feltöltöm.

A Shaderben ezután az, A-Buffer struktúrájának elkészítéséért felelős Fragment Shaderben generált pixelazonosítóhoz hasonlóan, itt is kiszámoljuk a pixelhez tartozó pixelID-t. Mivel a 2 renderfolyamat azonos viewportot használ, ezért a két Shaderben kiszámolt értéknek azonosnak kell lennie.



3.13 Végeredmény 8000 Metaballal

Ezután az azonosító segítségével elkérem a pixelhez tartozó elemet a PixelBufferből, majd a láncolt listán végig iterálva minden hivatkozott Metaballon végrehajtom a Wyvill density function tesztet. Az egyes Metaballok színe a 3D textúra, Metaball pozíciójában felvett intenzitásával egyenlő.



3.14 A Shaderek flowchart-ja

4 Értékelés

Az Orvosi térfogati adatok Metaballokkal történő kirajzolása fizikailag szimulálható állapotot eredményeznek. Kellően nagy Metaball-szám esetén az eredmény pontos és részletes képet tud adni. A továbbiakban megvizsgáljuk a Metaballok optimalizált mérési adatait.

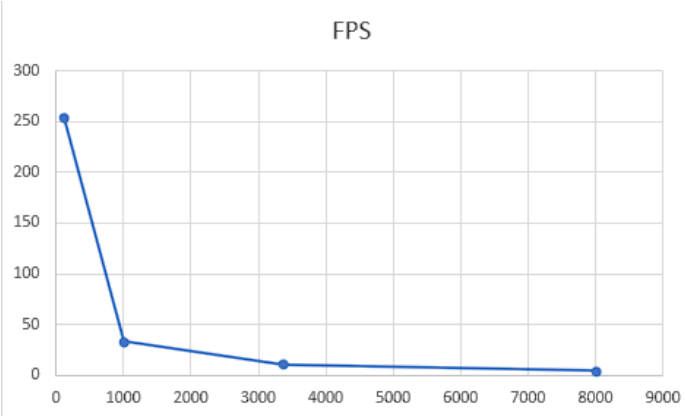
A megközelítés C++20 programnyelven, OpenGL és GLSL 4.3 shaderek használatával lett implementálva. A méréseket Windows 11-es környezetben egy laptopon futtattam a következő specifikációk mellett:

- Processzor: AMD Ryzen 9 5980HS 3.30 GHz
- Videókártya: AMD Radeon(Tm) Graphics; VRAM: 512 MB
- RAM: 32GB

A megjelenítéshez 600x600-as viewportot állítottam be.

Metaball	FPS	Felépítés	Memória
8000	4	0.45 ms	511.29 MB
3375	10	0.4 ms	334.79 MB
1000	33	0.38 ms	244.16 MB
125	254	0.3 ms	210.76 MB

4.1 Mérések adatai



4.2 Mérések adatai ábrázolva

A diagram bal oldali tengelye az FPS-t, az alsó pedig a Metaballok számát mutatja.

A mért értékek jól prezentálják az A-Buffer algoritmus legnagyobb hátrányát, a túlzott memóriahasználatot. Már 8000 Metaball mellett is elérte a teszteléshez használt laptop videokártya-memóriájának maximumát.

A mérési adatokból láthatjuk, hogy 8000 Metaballt interakcióra képes állapotban tudunk megjeleníteni. Az A-Buffer strukturált adatainak összeállítása a mérési adatok alapján eltérő Metaball-számokat vizsgálva elenyésző különbségeket eredményez.

Ezek alapján levonható a következtetés, hogy az algoritmus szűk keresztmetszetét a memóriából származó korlátok és a pixelenként nagyszámban eltárolt Metaballok megjelenítése adja.

5 Továbbfejlesztési lehetőségek

5.1 Bevezetés

Ebben a fejezetben az előzőekben felvezetett problémákra szeretnék megoldási javaslatokat tenni.

5.2 Ray Marching

Az algoritmus legnagyobb hátrányát az optimalizálásokkal nagyban gyorsított, de továbbra is költséges sugárkövetésen alapuló algoritmus adja.

Erre megoldást nyújthat egy polinom időben kiértékelhető közelítő algoritmus implementálása, melynek egy elegáns példáját olvashatjuk L. Szécsi publikációjában[18]. A jobb eredmények elérése érdekében a pixelenként eltárolt látható Metaballok számának további csökkentését is alkalmazhatjuk.

5.3 A-Buffer

Az implementált technológia másik legnagyobb szűk keresztmetszetét produkáló tényező a túlzott memóriahasználat. Ennek elkerülésére az A-Buffert kiegészíthetjük egy S-Bufferhez hasonló plusz renderelési hívással, amelyben az erőforrások lefoglalása előtt megvizsgáljuk az egy pixelből látható objektumok számát. Ebből kialakíthatunk egy pixelenként eltárolt maximális Metaball-számot. Ezzel csökkentve a memóriahasználatot és gyorsítva a Metaballok tesztelését.

A megközelítés implementálásánál viszont figyelnünk kell, hogy a struktúra felépítésénél a Metaballokat azok Z-Buffer értéke szerint rendezetten tároljuk el. Ezt a láncolt lista plusz „depth” adattagjának felvételével érhetjük el. A felvezetett padding értékek miatt a plusz információ valójában az egyik ilyen terület felhasználásával megoldható, így nem növelve az elemek szükséges méretét.

Az értékek ellenőrzéséhez használt keresés meggyorsításához az eddig használt láncolt lista struktúrát egy bináris fára érdemes lecserélni. Így az O(n) komplexitás helyett akár O(log n)-est is elérhetünk. [27]

5.4 Framework

A későbbiekben a saját készítésű keretrendszert egy környezetet ábrázoló textureCube-bal szeretném kiegészíteni. Valamint a megjelenített Metaballok árnyalását is el akarom érni, ezzel javítva az objektumok minőségén.

6 Köszönetnyilvánítás

Szeretnem megköszönni Kárpáti Attila Ádám konzulensemnek és Burkus Viktóriának, akik teljes egyetemi tanulmányom alatt készségesen segítettek, és, hogy dolgozatom készítése alatt szakmai tapasztalatukkal támogattak.

Irodalomjegyzék

[1] Flushing Hospital Medical Center: *History of Medical Imaging*, https://www.flushinghospital.org/newsletter/history-of-medical-imaging-a-brief-overview/#:~:text=The%20concept%20of%20medical%20imaging,photosensitive%20plate%20placed%20behind%20it.

[2] James F. Blinn: *A Generalization of Algebraic Surface Drawing*, ACM Transactions on Graphics, Vol. 1, No. 3, July 1982. 236-256

[3] Nvidia: GPU Gems, Chapter 39. *Volume Rendering Techniques*, https://developer.nvidia.com/gpugems/gpugems/part-vi-beyond-triangles/chapter-39-volume-rendering-techniques

[4] Arie Kaufman and Klaus Mueller: Overview of Volume Rendering, Chapter for The Visualization Handbook, 2005. 1-2

[5] Arie E. Kaufman: Volume Visualization, ACM Computing Surveys, Vol. 28, No. 1, March 1996. 165-167

[6] Marcos Vinicius Mussel Cirne, Hélio Pedrini: *Marching Cubes Technique for Volumetric Visualization Accelerated with Graphics Processing Units*, Article in Journal of the Brazilian Computer Society September 2013. 1-4

[7] Ben Anderson: *An Implementation of the Marching Cubes Algorithm*, https://www.cs.carleton.edu/cs_comps/0405/shape/marching_cubes.html

[8] Jonas Scholl: *Marching Cubes Algorithm*, https://code-specialist.com/python/marching-cubes-algorithm

[9] Zhongjie Long, Kouki Nagamune: *A Marching Cubes Algorithm: Application for Three-dimensional Surface Reconstruction Based on Endoscope and Optical Fiber*, Information, International Information Institute, 2015, 18 (4), pp.1425-1437. <hal-01205823>

[10] John Pawasauskas: *Volume Visualization With Ray Casting*, CS563 - Advanced Topics in Computer Graphics, February 18, 1997

[11] Martino Pilia: *GPU-accelerated single-pass volumetric raycasting in Qt and OpenGL*, September 17, 2018: https://martinopilia.com/posts/2018/09/17/volume-raycasting.html

[12] SimonDev: *Ray Marching, and making 3D Worlds with Math* https://www.youtube.com/watch?v=BNZtUB7yhX4&ab_channel=SimonDev

[13] Inigo Quilez: *Soft shadows in raymarched SDFs*, 2010, https://iquilezles.org/articles/rmshadows/

[14] Jamie Wong: *Ray Marching and Signed Distance Functions*, July 15, 2016, https://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/#signed-distance-functions

[15] Inigo Quilez: *Distance functions*, https://iquilezles.org/articles/distfunctions/

[16] Csaba Bálint, Gábor Valasek: *Interactive Rendering Framework for Distance Function Representations*, Annales Mathematicae et Informaticae 48 (2018) pp. 5–13

[17] C. Müller, S. Grottel, T. Ert: *Image-Space GPU Metaballs for Time-Dependent Particle Data Sets*, VMV January 2007

[18] L. Szécsi and D. Illés: *Real-Time Metaball Ray Casting with Fragment Lists*, The Eurographics Association 2012

[19] Patrik Andersson: *Voxelbaserad rendering med ”Marching Cubes”-algoritmen*, Kandidatarbete, 2009

[20] Cindy Müllenmeiste: *Simulation of Water Droplets using Metaballs on mobile phones*, University of Utrecht, August 2012

[21] Brian Wyvill, Craig McPheeters, Geoff Wyvill: *Animating soft objects*, The Visual Computer (1986) 2:235 242

[22] Yoshihiro Kanamori, Zoltan Szego, Tomoyuki Nishita: *GPU-based Fast Ray Casting for a Large Number of Metaballs*, The Eurographics Association, Volume 27 (2008), Number 2

[23] Kárpáti Attila, Burkus Viktória: *Karakteranimáció által vezérelt folyadékszimuláció*, Tudományos Diákköri Konferencia Budapest, 2020

[24] Joey de Vries: *Order-independent transparency*, https://learnopengl.com/Guest-Articles/2020/OIT/Introduction

[25] Loren Carpenter: *The A-buffer, an Antialiased Hidden Surface Method*, Computer Graphics Volume 18, Number 3 July 1984 103-107

[26] Geeks for geeks: *A-Buffer Method*, 21 Nov, 2021, https://www.geeksforgeeks.org/a-buffer-method/

[27] Andreas A. Vasilakis, Georgios Papaioannou, Ioannis Fudos: k^+-buffer: *An Efficient, Memory-Friendly and Dynamic k-buffer Framework*, IEEE Transactions On Visualization And Computer Graphics, Vol.21, No. x, xxxx 2015

[28] Andreas A. Vasilakis, Ioannis Fudos: *S-buffer: Sparsity-aware Multi-fragment Rendering*, The Eurographics Association 2012

[29] Christophe Lunarg: *GLM*, https://glm.g-truc.net/0.9.9/index.html

[30] Paul Silisteanu: *Reading and writing images with the stb_image libraries*, June 10, 2019, https://solarianprogrammer.com/2019/06/10/c-programming-reading-writing-images-stb_image-libraries/

[31] Victor Gordan:Organizing the OpenGL Buffers, https://www.youtube.com/watch?v=greXpRqCTKs&t=1s&ab_channel=VictorGordan

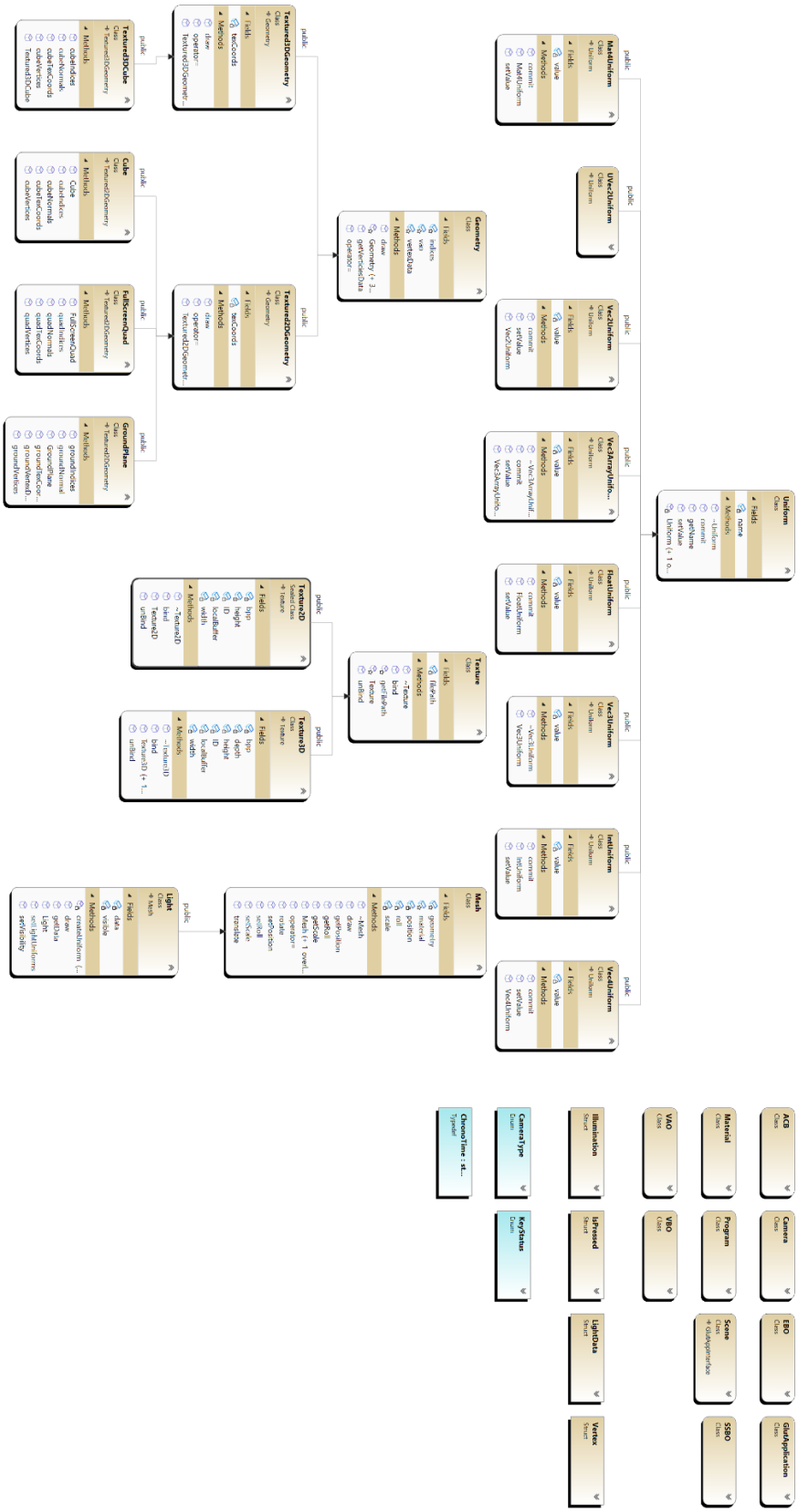
[32] Khronos: *Shader Storage Buffer Object*, https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object

[33] Khronos: *Atomic Counter*, https://www.khronos.org/opengl/wiki/Atomic_Counter

[34] Khronos: *Interface Block*
https://www.khronos.org/opengl/wiki/Interface_Block_(GLSL)#Memory_layout

[35] Khronos: *Compute Shader*, https://www.khronos.org/opengl/wiki/Compute_Shader

Függelék



F 1 A keretrendszer UML diagramja