EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

NUMERIKUS ANALÍZIS TANSZÉK

VÍZFELÜLET 3D MODELLEZÉSE

Témavezető:

Bognár Gergő

egyetemi tanársegéd

Készítette:

Mészáros Gergely

programtervező informatikus BSc

Budapest, 2019.

TARTALOMJEGYZÉK

I. BEVEZETÉS

Választott témám a Vízfelületek 3D modellezése, ami a számítógépes grafika komoly feladatának tekinthető. Ezen problémára számos módszer létezik, köztük valós idejű szimulációra illetve megjelenítésre alkalmasak is. Jelen szakdolgozat egy részecskemodell alapján közelíti meg a problémát ahol a részecskék közti fizikai erőhatások modellezésével kapott erőhatások által meghatározott differenciálegyenlet diszkrét numerikus megoldásának tekinthető. Ezzel léptethető a részecskék pozíciója illetve sebessége a lépésköz alapján. Az így kapott részecskéket vízfelszínként megjelenítésére megvalósítottam egy módszert ami a videokártyán hajt végre sugárvetést közvetlen a részecskepozíciók alapján, elkerülve a felület explicit meghatározását, és látványos megjelenítést lehetővé téve.

További motivációm a témaválasztáskor, hogy valós példán keresztül ismerkedhettem meg az általános célú grafikus programozással (General Purpose GPU Programming GPGPU) és annak a sajátos szemléletmódjával.

Leggyakrabban a GPGPU programozás kapcsán az implementációk az NVIDIA által fejlesztett és csak ezen cég videokártyáival elérhető CUDA programozási nyelven készülnek. Én ezzel szemben a dolgozatomban is támogatom a platformfüggetlenséget illetve a nyílt forráskódú megoldásokat. Emiatt a programom az OpenGL API-n keresztül valósítja meg a grafikus programozást illetve megjelenítést.

II. Felhasználói dokumentáció

A program egy interaktív 3D grafikus felülettel rendelkező alkalmazás melyben betöltődéskor egy függőleges vízoszlopot láthatunk, ahogy egy zárt dobozban szétfolyik. A program későbbi futása során a víztömeg hullámzásának animációját követhetjük nyomon, illetve szabályozhatjuk billentyűkombinációkkal.

2.1 Rendszerkövetelmények

A program futtatásához egy OpenGl 4.5 illesztővel rendelkező modern videokártya szükséges, illetve a következő könyvtárakra is szükség van:

- Simple Directmedia Layer (SDL)
- SDL_Image
- SDL_TTF
- GL extension wrangler (GLEW)
- Open Asset Import Library (assimp)

2.2 A program használata

1. Ábra

2. Ábra

3. Ábra

4. Ábra

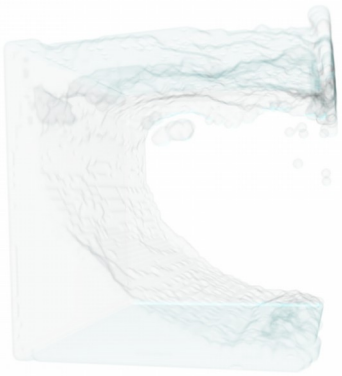
A program elindításakor azonnal megkezdődik a víztömeg szimulációja egy függőleges vízoszlop becsapódásával. (1. - 4. ábra).

5. Ábra

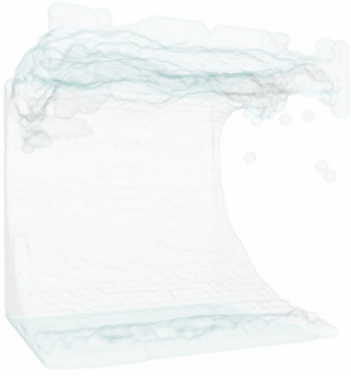
6. Ábra

2.2.1 Felhasználói Interakciók

A nézeti pozíciót megváltoztathatjuk a ***W A S D*** billentyűkkel a szimulációban a gravitáció iránya követi a kamera függőleges tengelyét, vagyis a ***W*** és ***S*** billentyűkkel tulajdonképpen a dobozt tudjuk megdönteni (5. 6. ábra). A szimuláció bármikor megállítható és folytatható a ***K*** billentyű megnyomásával. Megállított szimuláció mellett a kamera ugyanúgy szabadon mozgatható (7. 8. ábra).



7. Ábra



8. Ábra

A programot az ablak bezárásával vagy az ***ESC*** billentyű megnyomásával zárhatjuk be.

2.3 A FELADAT LEÍRÁSA

A megvalósított feladat egy víztömeg diszkrét részecskealapú modellezése, szimulációja egy véges térfogatban, és az így kapott pontfelhő megjelenítése, időbeli animációja.

2.3.1 SPH MÓDSZER

A szimuláció az úgynevezett *Smoothed Particle Hydrodynamics* módszer alapján működik. Ebben a módszerben a víztömeget diszkrét részecskékkel modellezzük. Minden részecske képviseli a tömeg egy részét, tároljuk a sebességét és pozícióját, ezen felül a víz tulajdonságait (sűrűség, nyomás) értékeit is a részecskék pozíciójában adjuk meg.

A víztömeg animációja az egyes részeire ható erőhatások által meghatározott differenciálegyenlet, aminek megoldása térben (részecskék) és időben (képkockák) diszkrét numerikus módszerrel történik.

A következő részben ismertetem az eljárás fontosabb eredményeit Müller et al. alapján. [1]

A módszer lényege, hogy a diszkrét részecskepozíciókban definiált mezőértékeket interpolálhatjuk a tér bármely pontjában a környező részecskék értékei és pozíciója szerint. Az interpolációt általánosan tárgyalva, tekintsük a következő $A:\mathbb{R}^3\to\mathbb{R}$ skaláris mezőtulajdonságot, melynek értéke adott $r\in\mathbb{R}^3$ pontban:

$$A(r)=\sum_j m_j\frac{A_j}{\rho_j}W(r-r_j,h) \quad ,$$

ahol *j* futóindex az összes részecskén, m_j a *j* részecske tömege, r_j a pozíciója, ρ_j a sűrűsége, és A_j az *A* mezőtulajdonság értéke a *j* részecskében.

A $W:\mathbb{R}^3\times\mathbb{R}\to\mathbb{R}$ függvény az úgynevezett simító magfüggvény *h* hatókörrel. A jobb numerikus stabilitás érdekében olyan magfüggvényeket választunk, amelyek párosak és normalizáltak, vagyis $W(r,h)=W(-r,h)$ és $\int W(r,h)dr=1\;\forall\, h$, továbbá

megköveteljük, hogy $\forall r,h: \|r\|>h \Rightarrow W(r,h)=0$. Ezeket a megszorításokat betartva tetszőleges függvényeket választhatunk. A szakdolgozatban felhasznált magfüggvényeket később a Magfüggvények alcím alatt mutatom be.

A továbbiakban r_j mindig a *j* részecske pozícióját, m_j a tömegét, ρ_j a sűrűségét és v_j a sebességét jelöli.

Minden részecske esetében tároljuk annak pozícióját és sebességét. Ezen felül minden képkockában kiszámoljuk a sűrűségét (vagyis a részecske térfogata változik a szimuláció közben). Az interpoláció általános egyenletébe behelyettesítve:

$$\rho_i \stackrel{\text{def}}{=} \rho(r_i) = \sum_j m_j \frac{\rho_j}{\rho_j} W(r_i - r_j, h) = \sum_j m_j W(r_i - r_j, h)$$

Ennek segítségével már megkaphatjuk az *i részecske gyorsulását* a_i *-t.*

$$a_i = \frac{\partial v_i}{\partial t} = \frac{f_i}{\rho_i} \quad ,$$

ahol ∂v_i az *i* részecske *sebességváltozása és* f_i *a ráható erők eredője, mely három tényezőből áll:*

$$f_i = f_i^{nyomás} + f_i^{viszkozítás} + f_i^{külső}$$

nyomási erőből, viszkozitási erőből és külső erőből, ami nem a részecsketulajdonságokból számolható, hanem előre adott minden részecskére (pl. gravitáció).

NYOMÁS

A nyomásból következő erő a környező részecskék nyomása alapján kapható meg és iránya a magfüggvény deriváltja felé mutat.

$$f_i^{nyomás} = -\sum_j \frac{p_i + p_j}{2\rho_j} \nabla W(r_i - r_j, h)$$

Ehhez *j* részecske nyomását p_j-t az ideális gáztörvényt alapul véve kaphatjuk meg:

$$p_j = k(\rho_j - \rho_0) \quad ,$$

ahol *k* az „összenyomhatóságot” jellemző állandó. A víz a valóságban nem összenyomható, de ebben a modellben az egyszerűség érdekében megengedjük az összenyomódást. Ahhoz, hogy ennek ellenére élethű eredményeket kapjunk *k* értékét nagyra választjuk meg (a program esetében az értéke 100). Ekkor kicsi sűrűségingadozások is nagy taszító erőket képeznek, vagyis hamar visszaáll az összenyomás előtti helyzet. ρ_0 a nyugalmi sűrűség erre az értékre tér vissza a szimuláció sűrűsége idővel. A programban az értéke 250. Ez az eltolás nem befolyásolja a nyomásból eredő erő értékét, hiszen az a deriváltjától függ, de numerikusan stabilabbá teszi a számolást.

VISZKOZITÁS

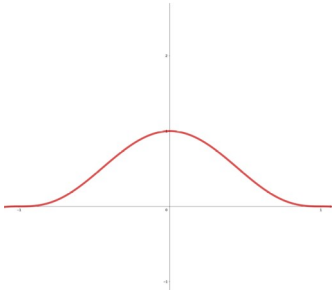
A viszkozitási erőt a következő módon kapjuk meg:

$$f_i^{viszkozitás} = \mu \sum_j m_j \frac{v_j - v_i}{\rho_j} \nabla^2 W(r_i - r_j, h) \quad ,$$

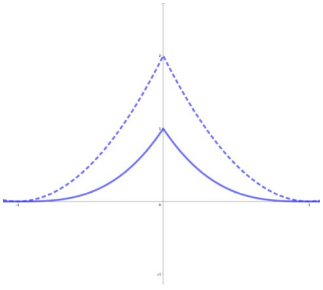
ahol $\nabla^2 W$ a Laplace operátor, azaz $\sum_{i=1}^{3} \frac{\partial^2 W}{\partial r_i}$ és μ a viszkozitást jellemző állandó (értéke ezen szakdolgozat esetében 5).

MAGFÜGGVÉNYEK

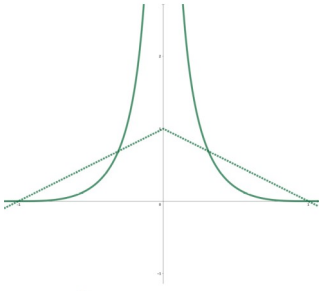
Müller et al.[1] alapján a következő magfüggvényeket használtam fel a számolásokhoz. Balról a poly6 magfüggvény, a hegyes magfüggvény és gradiense, és a viszkozitási magfüggvény és Laplace-operátorának értéke.



9. Ábra: poly6 magfügvény



10. Ábra: hegyes magfüggvény



11. Ábra: viszkozitási magfüggyvény

$$W_{poly6}(r,h)=\frac{315}{64\pi h^9}\begin{cases}(h^2-r^2)^3 & 0\le r\le h\\ 0 & \textit{egyébként}\end{cases}$$

A poly6 magfüggvény a sűrűségszámításhoz. Mivel az *r* távolság csak négyzetben szerepel benne, így gyökvonás nélkül kiszámítható.

$$W_{hegyes}(r,h)=\frac{15}{h^6}\begin{cases}(h-r)^3 & 0\le r\le h\\ 0 & \textit{egyébként}\end{cases}$$

$$\Delta W_{hegyes}(r,h)=\frac{45}{h^6}\frac{r}{\|r\|}\begin{cases}2(h-\|r\|)^2 & 0<\|r\|\le h\\ 0 & \|r\|>h\end{cases}$$

A hegyes magfüggvény és gradiense, amit a nyomásból eredő erők kiszámításához használok fel. Mivel a poly6 függvény a 0 közelében lapos (gradiense közel 0) ezt használva a részecskék hajlamosak lennének magas nyomás alatt összetorlódni. A hegyes magfüggvénynek ezzel szemben mindenhol pozitív a deriváltja így minden esetben taszító erőhatások keletkeznek.

$$W_{viszkozitás}(r,h)=\frac{15}{2\pi h^3}\begin{cases}-\frac{r^3}{2h^3}+\frac{r^2}{h^2}+\frac{h}{2r}-1 & 0\le r\le h\\ 0 & \textit{egyébként}\end{cases}$$

$$\nabla^2 W_{spiky}(r,h)=\frac{45}{\pi h^6}(h-r)$$

A viszkozitásszámításra használt függvény és Lagrange függvénye. Előnye, hogy mindig negatív irányban jelentkezik, így a részecskéket csak lassítani tudja egymáshoz képest.

IDŐINTEGRÁLÁS

A numerikus integrálás az ún. Leapfrog séma alapján történik, melynek előnye az alacsony memóriaigény és jó teljesítmény, hiszen csak egyszer kell az erőket meghatározni képkockánként.

$$a_t=F(r_t)$$
$$v_{t+1/2}=v_{t-1/2}+a_t\Delta t$$
$$r_{t+1}=r_t+v_{t+1/2}\Delta t$$

2.3.2 SZOMSZÉD KIVÁLASZTÁS

Ahogyan korábban már említettem a magfüggvények a *h* hatókörön kívül mindig 0 értéket vesznek fel, azaz minden részecske tulajdonságainak kiszámításához csak a *h* sugarú környezetének ismerete szükséges. Így a korábbi egyenletekhez képest az összegzést elég a közeli pontokon elvégezni.

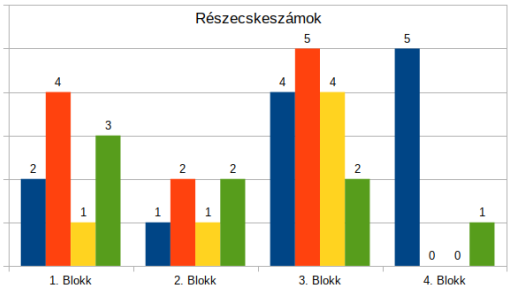
Ezeknek a közeli pontoknak a kiválasztását a szimuláció terének egy egyenlő rácsra való felosztásával érjük el, ahol minden cella oldalának a hossza pontosan *h*.

Ezzel az optimalizációval a számítási komplexitás $\Theta(n^2)$ -ről $\Theta(nm)$ -re csökkenthető, ahol *n* a részecskék száma, *m* pedig az egyes cellába jutó átlagos részecskeszám.

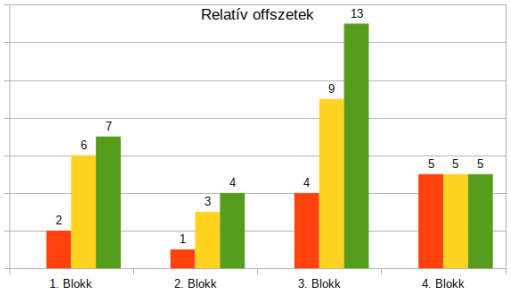
Ehhez a részecskéket egy GPU-ra optimalizált párhuzamosított counting sort-hoz hasonló algoritmussal rendeljük a cellákhoz. Az algoritmust részben [2] inspirálta.

Ebben a darabszámok szukcesszív összegzése nem a teljes tömb méretén, hanem párhuzamosan, annak kisebb, fix méretű blokkjain megy végbe, ugyanis a videokártyán az egyszálú végrehajtás nagyon lassú. Ezeket a blokkeltolásokat aztán egy közös szülőblokk alapján összegezzük. Az algoritmus lépései:

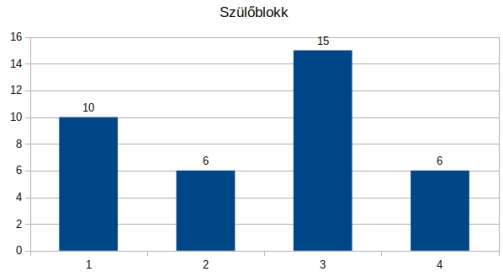
1. Minden cellában megszámoljuk a beleeső részecskéket, és egyúttal feljegyezzük minden részecske esetében, hogy melyik cellába tartozik.
2. A cellákban található részecskeszámokat blokkon belül összegezzük. Így a blokkhoz relatív eltolásértékeket kapunk. A blokk utolsó cellájának eltolását (azaz a blokkban szereplő részecskék számát) a szülőblokk a blokk sorszámának megfelelő helyére írjuk.
3. Összegezzük a szülőblokk celláit, hogy megkapjuk a blokkok eltolását.
4. A szülőblokk alapján hozzáadjuk a blokk eltolását benne lévő cellák eltolásához, így minden cellára megkaptuk a globális eltolást.
5. A részecskéket az új helyükre mozgatjuk a cellák eltolás értékei alapján.



12. Ábra

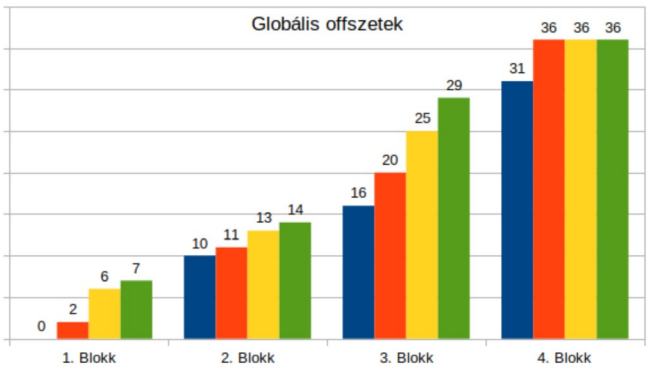


13. Ábra



14. Ábra

15. Ábra



16. Ábra

A cellákba soroló algoritmus egy kisebb példán keresztül. Az 12. és 13. ábrán az oszlopok a cellákat jelölik, magasságuk pedig a bennük található részecskék számát mutatja. A cellák aszerint vannak csoportosítva, hogy melyik blokkba tartoznak. A 14. és 15. ábrán a szülő blokk látható az offszetelés előtt és utána. A 16. ábrán a cellák végső eltolásértékei vannak.

2.3.3 MEGJELENÍTÉS

Mivel a szimuláció eredménye egy diszkrét pontfelhő, ennek egybefüggő víztömegként megjelenítésére implementáltam Goswarmi et al.[3] alapján a következő módszert.

Először meghatározzuk a felületi pontokat a következő képlettel:

$$A=\{i:\ \|r_i-r_i^M\|>a\} \qquad r_i^M=\frac{\sum_j m_j r_j}{\sum_j m_j} \quad .$$

Itt az *A* halmaz a felületi pontok halmaza és *j* befutja az összes olyan részecskét melyre $\|r_i-r_j\|<h$. Minden részecskére meghatározzuk a helyi tömegközéppontot r_i^M -t. Ez alapján akkor felületi pont egy pont, ha a távolsága a tömegközéppontól nagyobb mint az előre meghatározott *a* határ. A program esetében ez az érték 0,0001.

17. Ábra: A távolságtextúra egy szelete

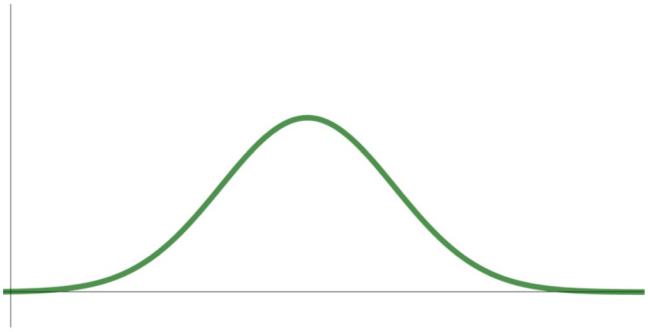
A felületi pontokból meghatározzuk a távolságfüggvényt, ami minden ponthoz a felülettől vett távolságát rendeli. Ennek a függvénynek az értékeit diszkrét pontokban kiszámoljuk és egy 3 dimenziós textúrában tároljuk. A textúra egy metszete a 17. ábrán látható (a részecskéktől vett távolságok alapján sötéttől a világosig).

$$D:\ \mathbb{R}^3 \to \mathbb{R},\ \ D(r)=\min_{i\in A}\|r-r_i\|$$

A textúrát a megjelenítés előtt egy maximális értékkel feltöltjük, majd az összes felületi pont körül egy *r* élhosszú kockát rajzolunk. Minden cellában meghatározzuk a részecskétől vett távolságot *d*-t és ez alapján frissítjük a korábbi értéket.

$$d_v' = min(d, d_v) \quad ,$$

ahol d_v' a cella új távolsága, d_v a korábbi távolsága, és *d* a most számolt érték.



18. Ábra: az átmenetfüggvény alakja

A textúrát közvetlen sugárvetéssel jelenítjük meg. A képernyő összes pontján keresztül fénysugarakat küldünk át és ezen sugarak mentén mintákat veszünk a távolságfüggvényből. A minta értékeihez egy *átmenetfüggvény* alapján átlátszóság értékeket rendelünk. A programban található átmenet függvény a következő alakú:

$$f(x) = c \cdot e^{-b(x-a)^2} \quad ,$$

ahol *a*, *b*, *c* a megjelenést szabályozó konstansok. Szemléletesen *c* a víz áttetszőségét, *a* a felület részecskéktől vett távolságát és *b* a felület élességet szabályozza. A függvény grafikonja a 18. ábrán látható.

A program esetében $a = 0.02$, itt olyan értéket kellett választani, ami lehető legkisebb, hiszen ekkor van legközelebb a megjelenített felület az azt meghatározó pontokhoz, de elég nagy ahhoz, hogy egybefüggő felület keletkezzen. Túl kicsi *a* értékekre a részecskék körüli felületek nem érnek össze, és lényegében pontonkénti megjelenítést kapunk.

A *b* konstans értéke a programban 160000, ezzel azon távolságértékek, amelyek érdemi átlátszóságot eredményeznek a [0.01, 0.03] tartományba esnek (az intervallum szélein az átmenetfüggvény értéke 0,000000225).

C megválasztásakor ügyelni kell arra, hogy milyen lépésközzel dolgozunk a sugárvetésben, hiszen ettől függ, hogy hány minta komponálásából alkotjuk meg a végleges színt. A programban az értéke a lépéshossz kétszerese, vagyis $c = 2 \cdot 0{,}005 = 0.01$.

A megvilágítást Blinn-Phong modell szerint végezzük, amihez a központi különbségek alapján számítjuk ki a felületi normálist:

$$\nabla d_{i,j,k} = \begin{pmatrix} \frac{\partial d}{\partial x} \\ \frac{\partial d}{\partial y} \\ \frac{\partial d}{\partial z} \end{pmatrix} \approx \begin{pmatrix} d_{i+1,j,k} - d_{i-1,j,k} \\ d_{i,j+1,k} - d_{i,j-1,k} \\ d_{i,j,k+1} - d_{i,j,k-1} \end{pmatrix} ,$$

ahol *d* a felülettől vett távolság az adott pontban, $d_{i,j,k}$ pedig az (*i, j, k)* koordinátájú cella értéke.

A pixel végső színét a mintákból kapott színek és átlátszóságok keveréséből kapjuk meg.

III. Fejlesztői dokumentáció

3.1 Fordítás

Az alkalmazás C++ nyelven íródott, fordításához egy c++17 képes fordító szükséges. A fejlesztés közben használt fordító a clang volt Linux környezetben, de másik fordító is használható, ehhez csak a makefile legelső sorában a *CXX* változó értékét kell megváltoztatni.

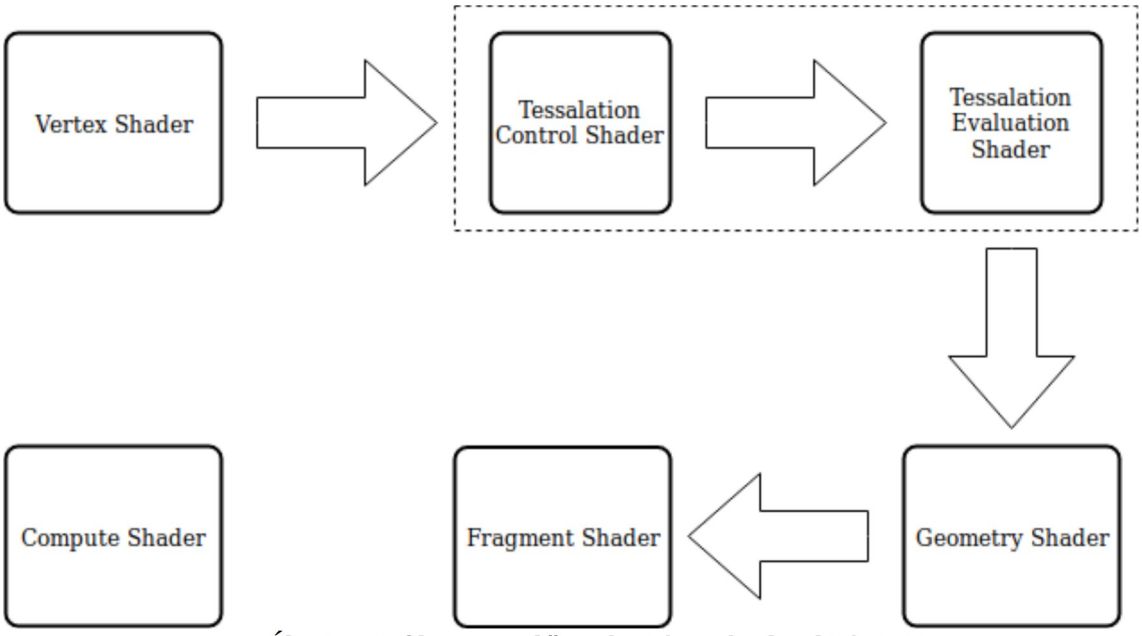
A fordítás *GNU make*-el történik és a futtatáshoz szükséges csomagok fejlesztői környezete is szükséges hozzá.

A fordítás és futtatás is működik Windowson, illetve Linuxon is, de a fejlesztés és tesztelés Linuxon történt.

3.2 OpenGL Áttekintés

Az OpenGL egy multiplatform szabvány API a 3D megjelenítés eszköztől független megvalósítására. Később általános célú videokártya programozási (GPGPU) lehetőségekkel is kibővült. A programom OpenGL-en keresztül az összes számítást a videokártyán végzi, ezért a következőkben bemutatom az ebből a szempontból legfontosabb részeit.

3.2.1 Shaderek



19. Ábra: Grafikus szerelő szalag és a shaderek típusai

Shaderek olyan programok melyeket eredetileg a világításmodell megadásához használtak, mostanra viszont több 3D megjelenítési feladat, speciális effekt vagy akár általános célú felhasználásra is alkalmasak. Ezeket a videokártyán található számítási egységek nagyjából egymástól függetlenül párhuzamosan hajtják végre.

Az inkrementális képszintézisben a színteret háromszöghálókból álló modellekre bontjuk, ezeknek a csúcsaihoz rendelünk információkat. Ezeket és a felülethez rendelt információkat (textúrák) jelenítjük meg adott nézeti pozícióból. Az inkrementális képszintézist a grafikus szerelőszalag valósítja meg.

A grafikus szerelőszalag első lépése a Vertex Shader, bemenete a rajzoló hívásban megadott vertexek, vagyis a háromszögháló csúcsaihoz rendelt értékek, amikre egy transzformációt hajt végre. Minden vertexre pontosan egy shader példány jut. A Tessalation Control és Evaluation Shaderekkel egy felület részletessége növelhető, vagyis újabb vertexek adhatóak hozzá. A Geometry Shader bemenetei primitívek (pl. háromszög, négyszög, szakasz) ezeket módosíthatja, és akár a primitív típusát is megváltoztathatja, pl. szakaszokból háromszögeket csinálhat. Ezután a videokártya raszterizálja a primitíveket, és a kapott összes pixelre egy-egy Fragment Shader példány jut, ami a pixel színét határozza meg. Amennyiben nincs szükség tesszelációra vagy a primitívek átalakítására, úgy ezeket a shadereket nem kell megadni.

A szerelőszalag adatai mellett olyan adatokra is szükség van, amik ugyanazok a shader példányok között. Ezek lehetnek uniformok, textúrák és bufferek.

A uniformok kis méretű, csak olvasható adatok, amelyek értéke ugyanaz minden shader példányra. Értéküket kívülről a CPU oldali programból kapják, nem a vertex adatokból vagy a szerelőszalagból származtatják. Lehetnek például színek, anyagtulajdonságok, vagy ezen szakdolgozat esetében egyes szimulációs állandók értéke is uniformmal van megadva.

A shaderek nem csak a saját kimenetükre írhatnak, elérhetőek írható és olvasható változók Shader Storage Bufferek, továbbiakban gyakran csak bufferek, is. A bufferek a videomemóriában foglalnak helyet és az első feltöltésükön kívül egyáltalán nem tároljuk őket a rendszermemóriában. A programban a részecskék tulajdonságai és lényegében az összes szimulációs adat ezekkel van megvalósítva.

A shaderek programkódját OpenGL esetében egy magas szintű programozási nyelvben glsl-ben adhatjuk meg, amit az illesztőprogram futási időben fordít a videokártyának megfelelő gépi kóddá.

Compute Shader

A compute shader egy a shaderek típusai közül, és többivel ellentétben nem része a grafikus szerelőszalagnak, hanem a megjelenítéstől függetlenül általános célú számításra használható.

Mivel nem része a grafikus szerelőszalagnak, így az adatokat csak uniformokból, textúrákból és bufferekből olvashatja, és mivel rögzített kimenete sincs, így csak bufferekbe és texturákba tudja az eredményeit rögzíteni.

A compute shaderek különlegessége, hogy az egyes példányok futása nem teljesen független, hanem munkacsoportokba szerveződnek. Ezeken a munkacsoportokon belül az egyes példányok közti koordinált számításra, adatok megosztására, és szinkronizációra is lehetőség van.

A munkacsoportba tartozó példányok száma a shader fordítási idejében adott. A számítás indításakor adható meg a munkacsoportok száma.

3.2.2 SUGÁRVETÉS

A videokártyák alapvető felépítése az inkrementális képszintézis (vagyis az elöző fejezetben részben bemutatott háromszögeken alapuló megjelenítési módszer) gyorsítására lett kialakítva. Ez a módszer nagyon részletes modellek és jelenetek gyors megjelenítésére alkalmas, hiszen minden egyes háromszög megjelenítése szinte független az összes többitől. Ebből eredően viszont, sok a valóságban megfigyelhető jelenség megjelenítésére nem alkalmas. Ilyenek például a valódi tükröződések, és a fénytörés.

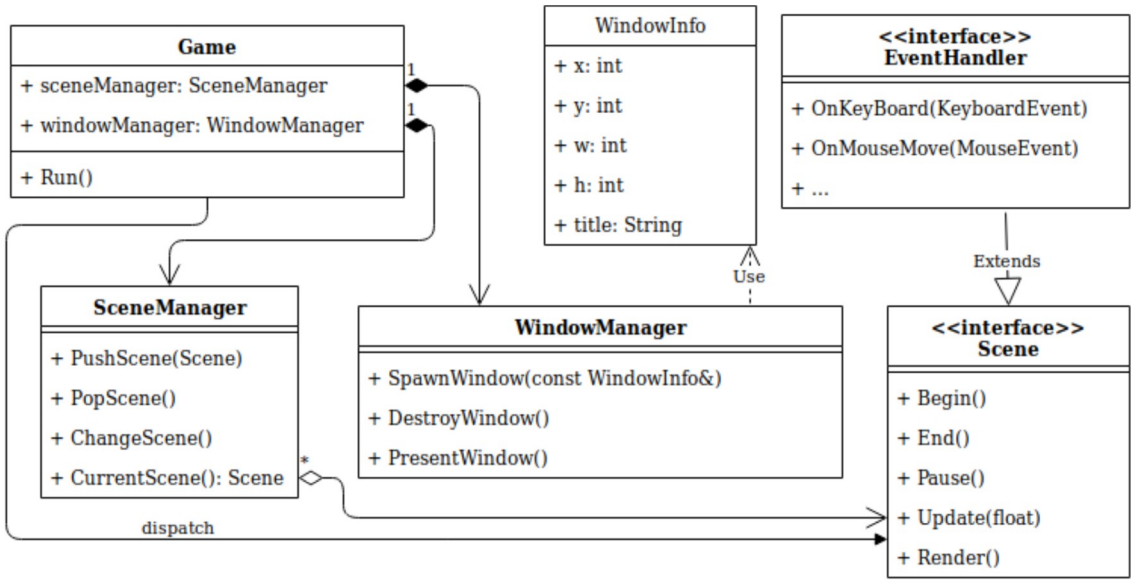
A valósághoz egy jobban igazodó, de jelentősen költségesebb, megjelenítési mód, amelyben ezek a jelenségek is kezelhetőek, illetve más hatások is egyszerűbben és természetesebben adhatóak vissza (pl. az árnyékok), a sugárkövetés. Ez a módszer a fényforrásból érkező sugarak útvonalán halad végig, ahogyan az tárgyról tárgyra tükröződik, és végül eléri a képernyő pixeleinek világon belüli pozícióját, megadva a pixel színét. A gyorsabb számítás érdekében ahelyett, hogy a fényforrásokból indítanánk a sugarakat, fordított irányban haladunk a képernyő pixeleitől kezdve. Ekkor ugyanis nem számítjuk ki feleslegesen azoknak a sugaraknak az útvonalát melyek végül soha nem érik el a képernyőt.

A programban a megjelenítés sugárvetéssel van megvalósítva, ami a sugárkövetés egy egyszerűbb formája, ahogyan az a felhasználói dokumentációban olvasható. Azért esett erre a módszere a választás, mert ezzel sokkal látványosabb megjelenítés érhető el (pl. valóban jól kezelhető az átlátszóság), illetve mert egy részecskefelhő megjelenítésére sokkal egyszerűbben alkalmazható. Az inkrementális képszintézishez ugyanis meg kellene határozni a részecskefelhő felületét, majd háromszögekre kellene bontani azt. Ezzel szemben a sugárvetéskor nem kell a felületet explicit meghatározni.

3.3 RENDSZERTERV

A program három elkülönülő részre tagolható, egy általános eseményvezérlő és ablakkezelő rendszerre, az OpenGL erőforrások objektum orientált kezelését biztosító modulra, és a konkrét szimulációt megvalósító osztályokra és adatszerkezetekre.

3.3.1 MODULOK

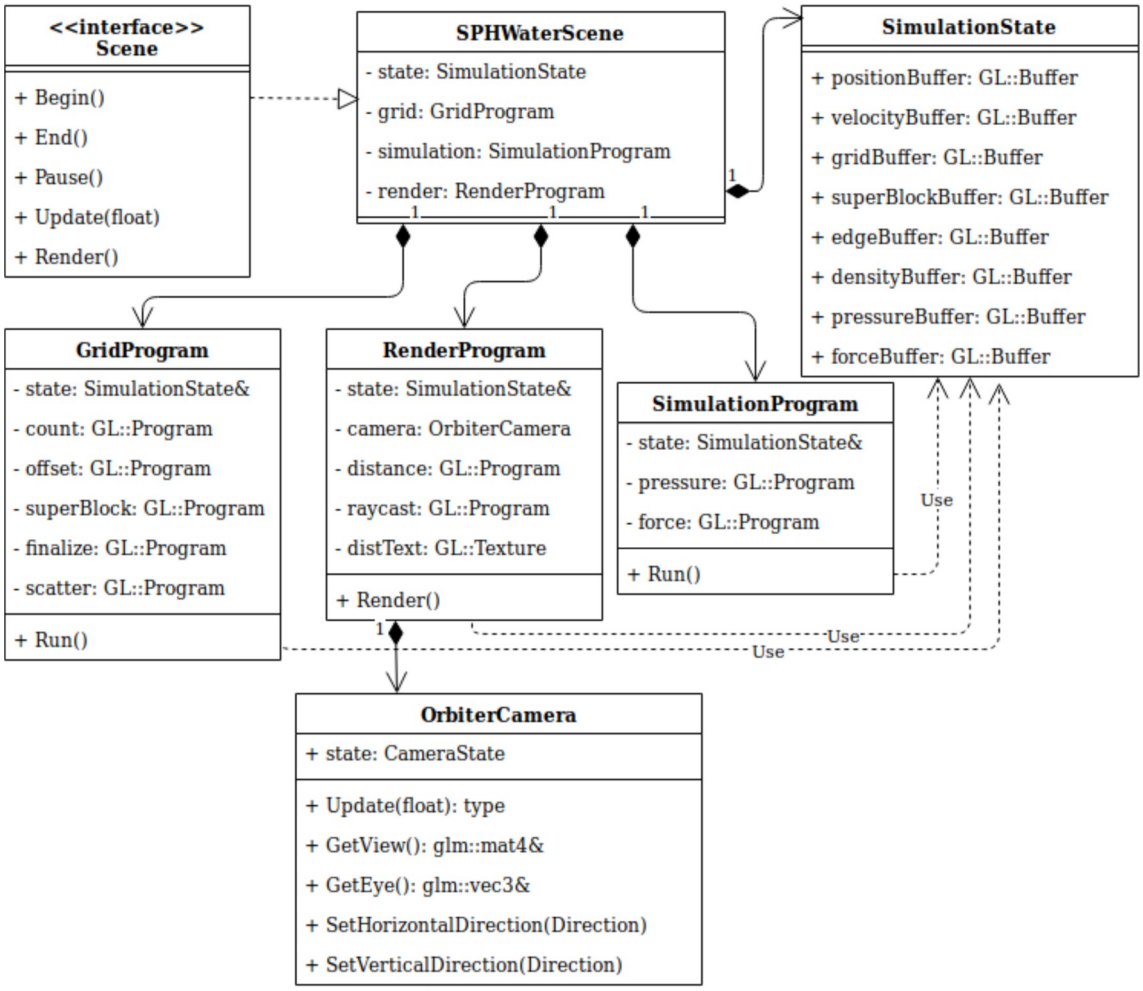


20. Ábra: Az ablakkezelő modul osztályai

Az ablakkezelő és jelenet vezérlő modul legfontosabb osztályai és köztük lévő kapcsolatok a 20. ábrán láthatóak. A modul feladata az ablak, a hozzátartozó 3D OpenGL környezet létrehozása és a jelenetek kezelése.

Egy jelenet meghatározza az összes esemény kezelését továbbá a rajzolást is maga valósítja meg. A jelenetek átválthatnak egy másik jelenetre, ettől kezdve az új jelenet a felelős az események kezeléséért és a rajzolásért. A teljes váltáson kívül megnyithatnak egy jelenetet maguk felett, ekkor ugyanúgy az új jelenet veszi át az események kezelését és a rajzolást, de annak a befejeződésekor újra az előző jelenet lesz aktív.

A platformfüggetlenség érdekében az ablak és eseménykezelés az *SDL* nevű platform absztrakciós rétegen keresztül történik.

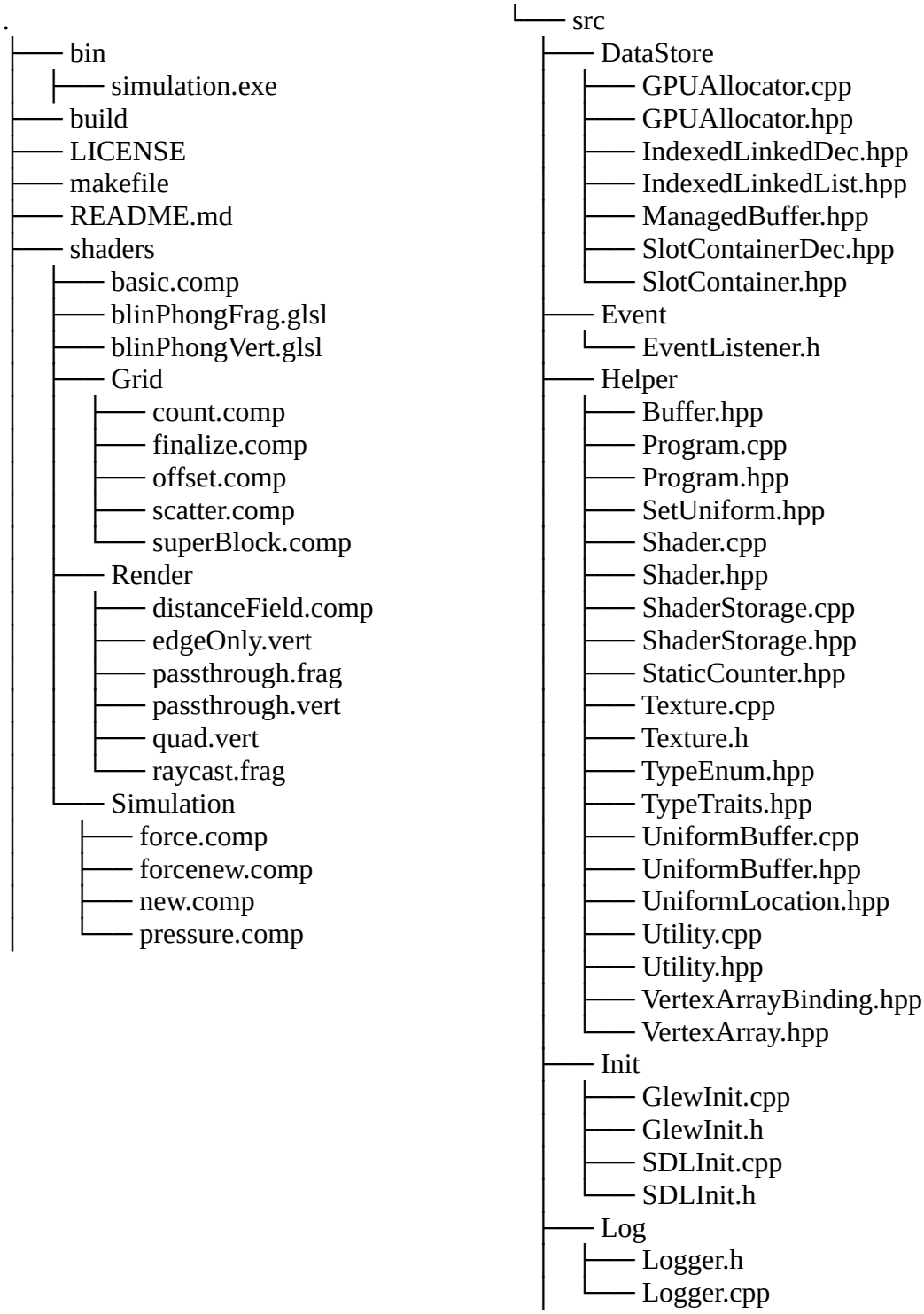


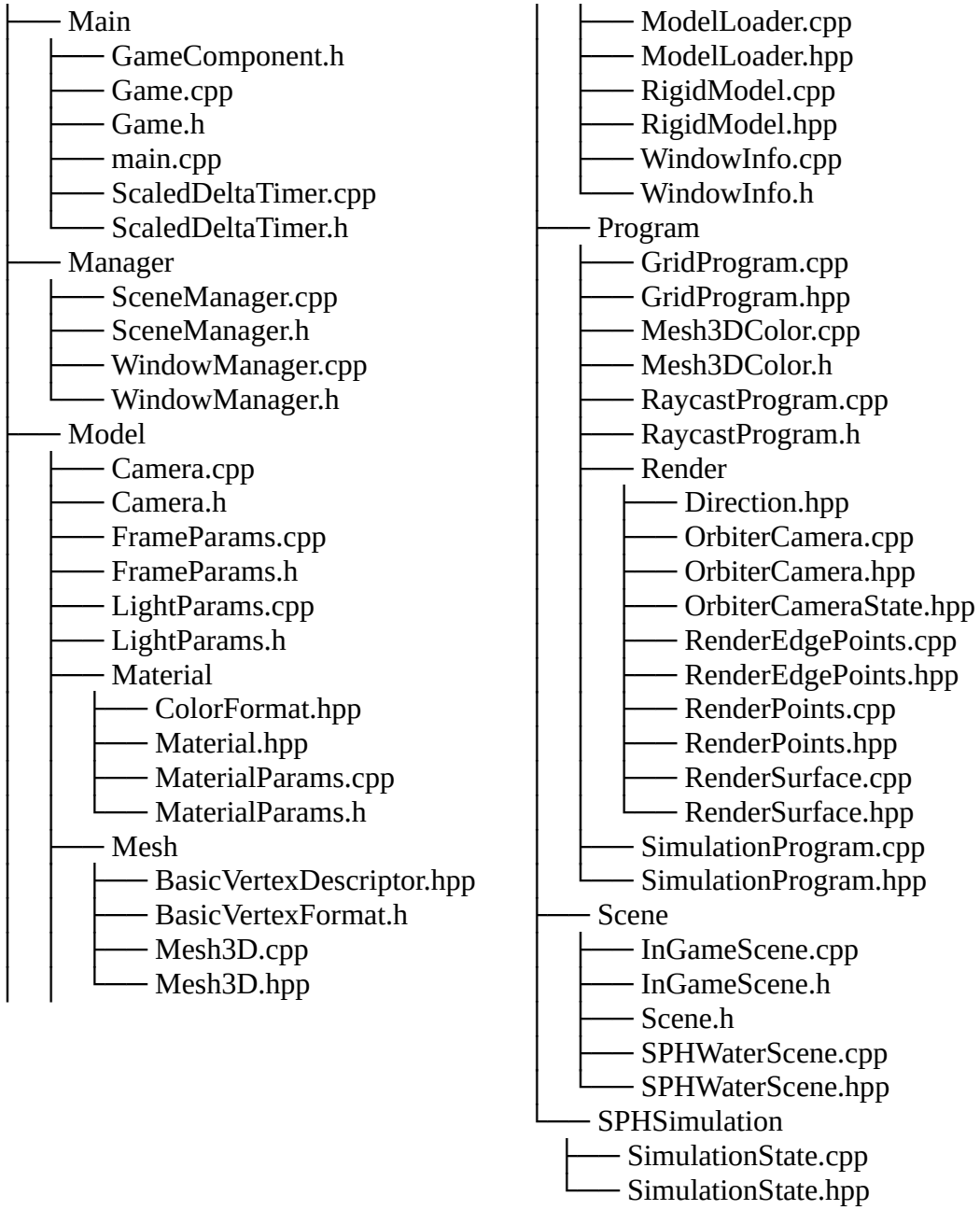
21. Ábra: A szimuláció osztályai

A konkrét szimulációt megvalósító modul szerkezete. A szimulációt vezérlő jelenetet az SPHWaterScene osztály valósítja meg, amely az ablakkezelő modulban lévő Scene interfészt implementálja (21. ábra)

Ez a modul tartalmazza a szimuláció egyes lépéseit megvalósító osztályokat is. GridProgram a közelségi háló kiszámítására, SimulationProgram a részecskék szimulálására és RenderProgram a megjelenítésre. Az egyes programok a SimulationState-n keresztül érik el a szimuláció állapotát, vagyis az egyes OpenGL buffereket. A RenderProgram a kamera megvalósítására tartalmazza az OrbiterCamera osztályt, ami nézeti mátrix kiszámításáért és a kamera animációjáért felelős.

3.3.2 MAPPASZERKEZET

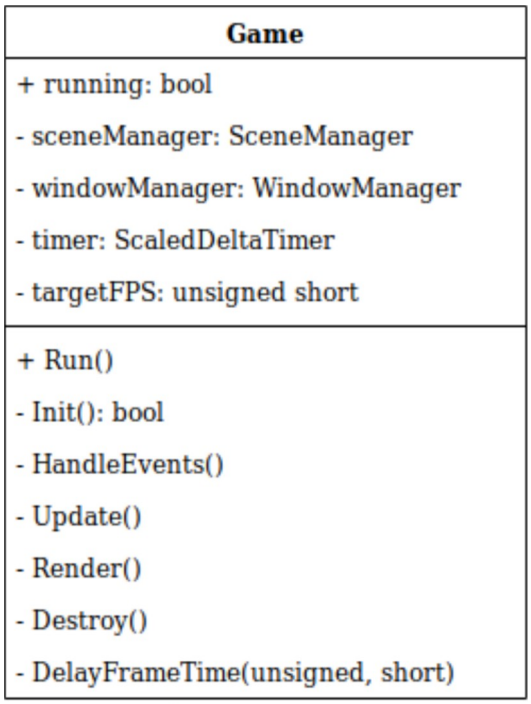




A szakdolgozat fájljainak struktúrája. A *bin* mappában a futtatható állomány és a szükséges dinamikus könyvtárak találhatóak. A *shaders* mappában a glsl shaderek forráskódja található. A *build* mappába az inkrementális fordításhoz a tárgyfájlok kerülnek. A CPU oldali forráskódok az *src* mappa alatt találhatóak.

3.4 Implementáció

3.4.1 Ablakkezelő Modul



22. Ábra: Game osztály

A Game osztály (22. Ábra) a belépési pont a program futásában. Eljuttatja az eseményeket az aktív jelenetnek.

`void Run()`

Belépési pont az alkalmazásba, itt fut az eseményvezérlő ciklus. A program a teljes élettartama alatt ebben a függvényben van.

`bool Init()`

Felállítja az alkalmazás futásához szükséges erőforrásokat, létrehozza az ablakot és elindítja az első jelenetet.

`void HandleEvents()`

Lekérdezi az előző képkocka alatt érkezett eseményeket az SDL-től és eljuttatja a jelenleg aktív jelenetnek.

`void Update()`

Frissíti az időzítőt és az aktív jelenetet.

`void Render()`

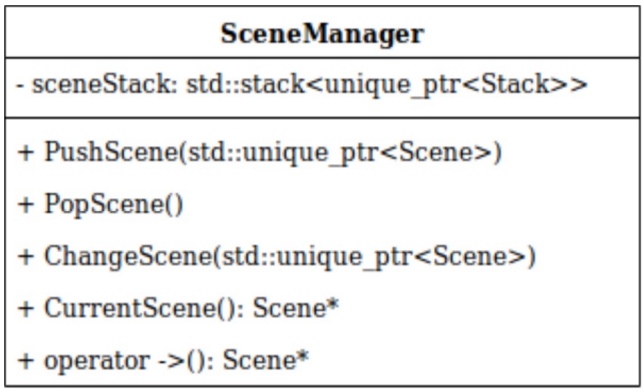
Meghívja az aktív jelenet rajzolóját.

`void Destroy()`

Felszabadítja az alkalmazás által használt erőforrásokat.

`void DelayFrameTime(unsigned frameStart, short targetFPS)`

Limitálja az alkalmazás képfrissítéseinek számát. A képkocka számításának kezdete és a jelenlegi idő alapján annyit vár, hogy a képfrissítésnek megfelelő idő teljen el.



23. Ábra: SceneManager osztály

SceneManager a jeleneteket kezelő osztály, ami egy verem adatszerkezetben tárolja a jeleneteket, hogy később folytatni tudja azokat amikor a „gyerekeik” befejeződtek, a 23. ábrán látható.

`void PushScene(std::unique_ptr<Scene> scene)`

Megállítja a jelenleg aktív jelenetet, felteszi *scene* jelenetet a verem tetejére és elindítja azt.

`void PopScene()`

Befejezi a jelenleg aktív jelenetet, leveszi a verem tetejéről és folytatja az előzőt.

`void ChangeScene(std::unique_ptr<Scene> scene)`

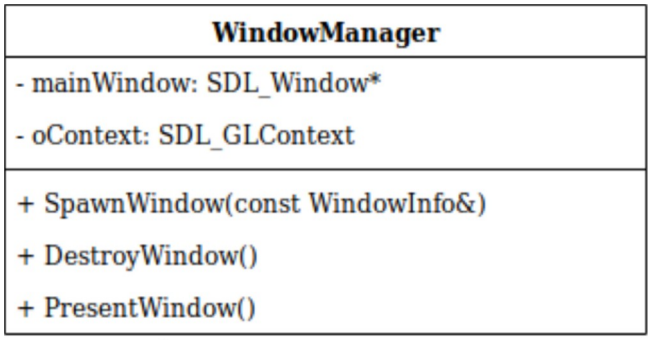
Befejezi a jelenleg aktív jelenetet és leveszi a verem tetejéről, helyére pedig a paraméterben megadott *scene* jelenetet teszi és indítja el.

`Scene* CurrentScene()`

Visszaadja a verem tetején található jelenet mutatóját.

`Scene* operator->()`

Megvalósítja, hogy a SceneManager példányain keresztül operator→-val elérhetjük az aktív jelenetet.



24. Ábra: WindowManager osztály

A WindowManager osztály a főablak létrehozásáért törlésért és az OpenGL környezet létrehozásáért felelős (24. ábra).

`void SpawnWindow(const WindowInfo& wnd)`

Létrehozza a főablakot a *wnd* struktúrában kitöltött adatok alapján és felállítja az OpenGL környezetet is. A *WindowInfo* struktúrával az ablak címe, mérete, képernyőn vett pozíciója és láthatósága szabályozható. Emellett megadhatjuk, hogy az ablak átméretezhető legyen-e.

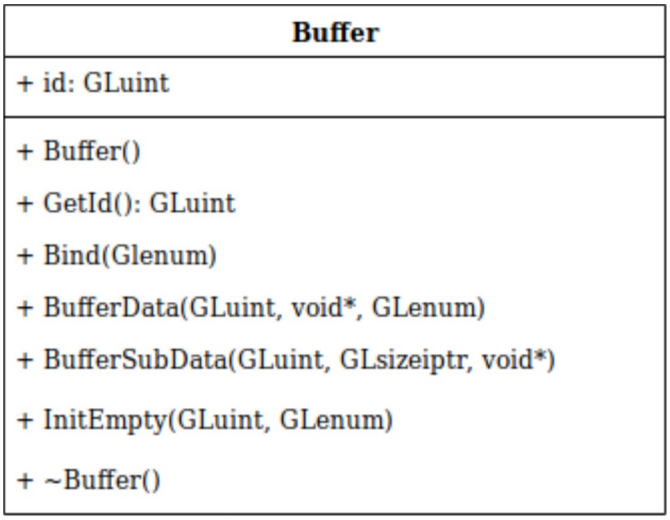
`void DestroyWindow()`

Felszabadítja az ablakot és a kapcsolódó OpenGL környezetet.

`void PresentWindow()`

Megjeleníti a képkocka alatt történt változásokat az ablakon. (Kicseréli a hátsó és első buffereket).

3.4.2 OpenGL támogató modul



25. Ábra: Buffer osztály

Buffer osztály (25. ábra) egy OpenGL buffer CPU oldali elérését biztosítja objektum orientált formában. A függvényei csak az OpenGL hívások részleteit rejtik el.

`Buffer()`

A buffer konstruktora. Létrehozza az OpenGL buffer azonosítóját.

`Gluint GetId()`

Visszatér a buffer OpenGL azonosítójával.

`void Bind(GLenum target)`

Az adott használati célhoz köti a buffert. Az OpenGL API-nak önmagában önmagában is van állapota (ún. stateful API). A 4.5-ös verzió előtt a műveletek előtt az egyes objektumokat (bufferek, texturák, vbo-k, vao-k stb.) egy általános célhoz kellett kötni, majd a műveletnél a célra hivatkozni. Pl. glBindBuffer(GL_UNIFORM_BUFFER, buffer) majd glBufferData(GL_UNIFORM_BUFFER, …) hívásokkal lehetet a buffert új értékekkel feltölteni.

`void BufferData(GLuint size, void* data, Glenum usage)`

Feltölti a buffert tartalommal. A *data* címen kezdedő adat mérete *size*. A feltöltött adat használatát (az optimális tárterület kiválasztása érdekében) usage adja meg.

`void BufferSubData(GLuint offset, GLsizeiptr size, void* d)`

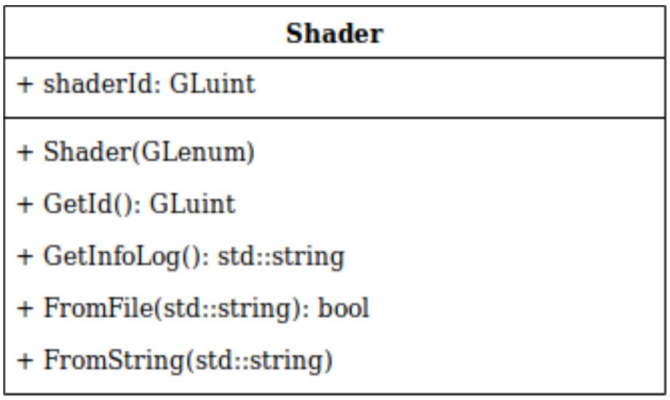
Feltölti a buffer egy részét adattal. *Offset* az adat bufferen belüli kezdőpozíciója, *size* a mérete, *d* az adat mutatója.

`void InitEmpty(GLuint size, GLenum usage)`

Létrehozza a buffer mögötti tárhelyet adat nélkül. *Size* a mérete, *usage* a használata.

`~Buffer()`

A buffer destruktora felszabadítja a buffer azonosítóját és törli a buffert.



26. Ábra: Shader osztály

A Shader osztály egy OpenGL shader létrehozását és fordítását biztosítja (26. ábra).

`Shader(GLenum type)`

Létrehoz egy *type* típusú üres shadert.

`GLuint GetId()`

Visszaadja a Shader azonosítóját.

`std::string GetInfoLog()`

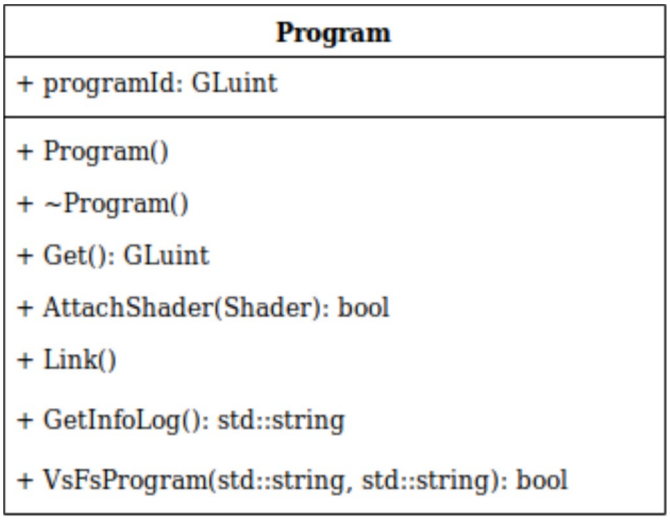
Visszatér a fordítás közben keletkezett szöveges napló tartalmával. Pl. a fordítás hibaüzeneteit tartalmazza

`bool FromFile(const std::string& fileName)`

Lefordítja a *fileName* fájlnevű fájl tartalmát. A fájlt a standard c++ könyvtár segítségével a memóriába olvassa és áthív a kapott stringgel a FromString metódusba.

`bool FromString(const std::string& string)`

Lefordítja a kapott stringet shaderként.



27. Ábra: Program osztály

A Program osztály egy OpenGL programhoz biztosít hozzáférést. Mivel egy feladat (pl. a megjelenítés esetében) több shadert is alkalmazhat, így ezek együttesét alkotja a Program, amellyel végrehajthatóak a rajzoló hívások (27. ábra).

`Program()`

A konstruktor létrehoz egy üres programot és elmenti az azonosítóját *programId*-ba.

`GLuint Get()`

Visszatér a program azonosítójával.

`bool AttachShader(const Shader& shader)`

Hozzácsatol a programhoz egy előzetesen már lefordított shadert.

`void Link()`

Linkeli az OpenGL program egyes shadereit. Ekkor kapcsolódnak össze a megjelenítési szerelőszalag lépései egymással. A linkelt program már használható rajzolásra vagy Compute Shader esetében számításra.

`std::string GetInfoLog()`

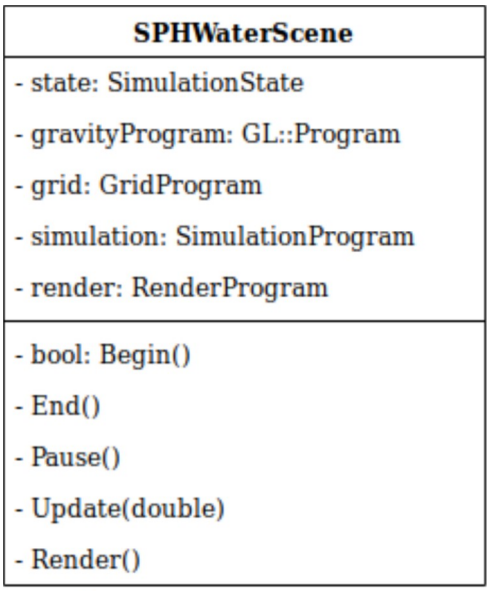
Visszaadja a program linkelése közben kapott esetleges OpenGL hibákat szöveges formában.

```
bool VsFsProgram(const std::string& vertexFileName,
                 const std::string& fragmentFileName)
```

Segéd függvény, ami létrehoz egy vertex és fragment shaderből álló programot (ez a kettő a kötelező a megjelenítéshez). A paraméterek a shaderek forráskódjainak fájlnevei.

Igazzal tér vissza, ha sikeres a fordítás és linkellés, hamissal, ha nem.

3.4.3 Szimulációs Modul



28. Ábra: SPHWaterScene osztály

A szimuláció jelenetét megvalósító osztály (28. ábra), tartalmazza a részfeladatokat megoldó osztályok példányait és a szimuláció állapotát leíró osztályt is. Itt valósítottam meg az integrálást is.

`bool Begin()`

Lefordítja az időintegrációs shadert és OpenGL konstansokat állít be. (Pl. a háttérszínt)

`void Update(double delta)`

A képkockák közt eltelt idő (delta) alapján lefuttatja a szimuláció lépéseit, ha az nincs megállítva. Csak akkor lépteti a szimulációt, ha az előző lépés óta egy lépésköznyi idő már eltelt. Meghívja a GridProgram-ot a közelségi háló kiszámítására, a SimulationProgramot a szimuláció erőinek meghatározására és elvégzi az integrációt a gravityProgram shaderrel.

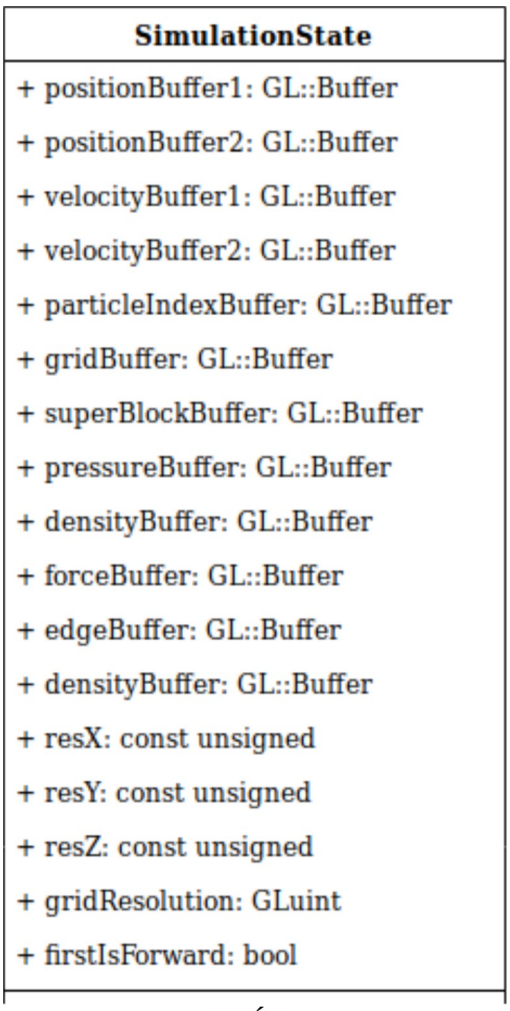
Az integrálás, ahogyan minden más számítás a videokártyán történik egy Compute Shaderrel, amelynek a forráskódja a shaders/basic.comp fájlban található.

Ez a shader minden részecskére egyszer fut, felhasználja a részecskék pozícióját, sebességét, sűrűségét és a rájuk ható erőket, hogy ezekből frissítse a pozíciót és

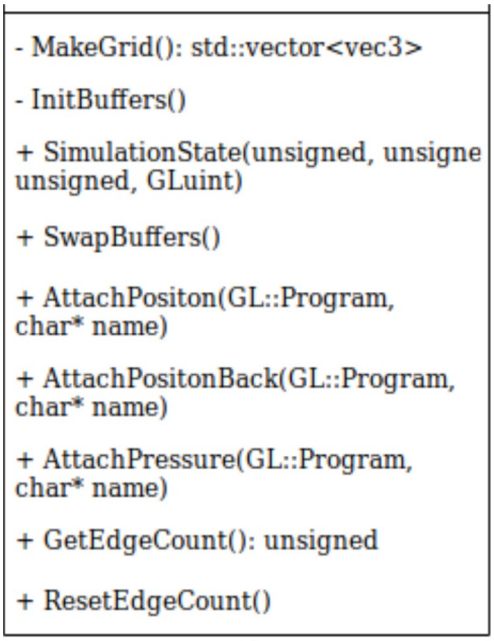
sebességet. A részecskére ható belső erőkhöz egy uniformban megadott gravitációs gyorsulást is hozzáad és integrálja a pozíciókat az eltelt idő alapján (ami szintén egy uniformban adott). Továbbá a szimuláció terének kijelölt kocka belsejében tartja a részecskéket, azzal, hogy ha ennek a falaiba ütköznek, akkor visszapattannak onnan (a sebességük fal irányába mutató komponensét tükrözi).

`void Render()`

Kirajzolja a képkockát, vagyis meghívja a RenderProgram.Render() függvényt.



29. Ábra



30. Ábra

A szimuláció állapotát tartalmazó osztály (29. 30. ábra), feladata a kezdeti feltételek felállítása, és az egyes részfeladatok (tehát a cellák kiszámítása, a szimuláció az integrálás és a megjelenítés) ezen az osztályon keresztül osztják meg az adatokat egymással.

Ez az osztály birtokolja a szimuláció összes bufferét (a „buffer” végződésű változók), és itt kapcsolódnak hozzá a shaderekhez az Attach* függvényhívásokon keresztül.

A pozíciókat és a sebességeket kétszer is eltároljuk, és ezek között minden képkockában váltunk. Ez alapján van minden képkockában egy elülső és egy hátsó változatuk. Erre a hálószámításkor van szükség, ugyanis az az utolsó lépésben párhuzamosan helyezi át az összes részecskéket az új pozíciójára. Ha csak egy buffer lenne, akkor a részecskék áthelyezéskor felülírhatnák egymást.

`std::vector<glm::vec3> MakeGrid()`

Elkészíti a szimuláció kezdeti állapotát. Egy stl vektorba kiszámítja a részecskék pozícióját ez fog később feltöltődni a videókártyára.

`void InitBuffers()`

Létrehozza az összes OpenGL buffert, illetve amelyiket szükséges feltölti tartalommal.

`void SwapBuffers()`

Kicseréli az első és hátsó pozíció és sebesség buffereket.

`void Attach*(const GL::Program& program, const char* name)`

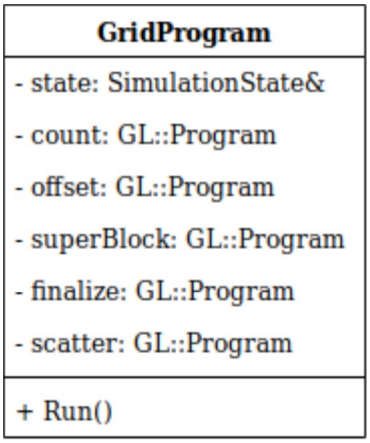
Hozzákapcsolja a *program* programhoz a glsl kódban *name* nevű kapcsolási ponthoz a * buffert. Minden bufferre egyszer szerepel egy ilyen függvény, a pozíció és sebességnél mind az első és a hátsó bufferre is egy-egy.

`unsigned GetEdgeCount()`

A kirajzoláshoz a felület közelében lévő részecskék pozícióját egy másik bufferbe is másoljuk. Ez a függvény visszaadja, hogy hány ilyen részecske volt.

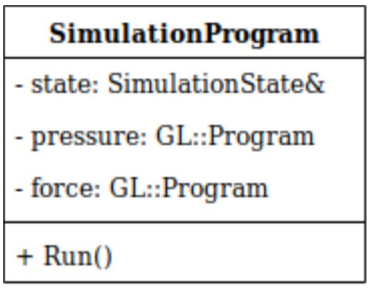
`void ResetEdgeCount()`

Visszaállítja nullára a felület közelében lévő részecskék számát.



31. Ábra: GridProgram osztály

A háló kiszámítását végző osztály (31. ábra). A fejlesztői dokumentációban leírt algoritmus öt lépését öt Compute Shaderrel a videokártyán valósítja meg, a forráskódjuk a shaders/Grid mappában található.



32. Ábra: SimulationProgram osztály

A sűrűséget, nyomást és ebből eredő erők számolásáért felelős osztály. (32. ábra)

A nyomás számítását a shaders/new.comp forrású shader végzi el. A shader minden munkacsoportja egy cellában található részecskék sűrűségét és nyomását számolja ki a következő lépések szerint.

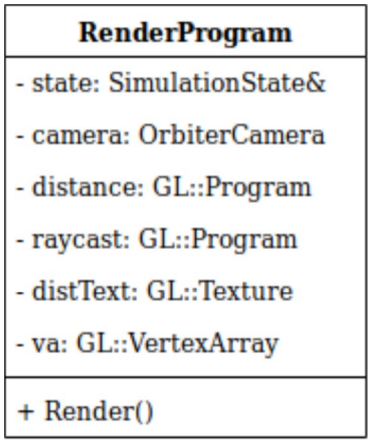
1. Az első 27 példány meghatározza a szomszédos cellák részecskéinek számát illetve azt, hogy hol kezdődnek a részecskék a bufferekben, hiszen a sorba rendezés után részecskék a cellák szerint folytonosan szerepelnek a bufferekben.
2. Minden példány betölti egy részecske pozícióját a cellából egy lokális és egy megosztott változóba is (maximum 128 részecske lehet a cellában, ennyi példány jut egy munkacsoportra).
3. Minden példány kiszámolja a sűrűségét és a helyi tömegközéppontot a többi részecske alapján.
4. Sorban betöltik a szomszédos további 26 cella részecskéit és ezekre is elvégzik az összegzést a 2. és 3. pont szerint.
5. A kapott sűrűségek alapján kiszámolják a nyomást és a részecske tömegközéppontól való távolsága alapján a felülethez közeli részecskéket (ezek esetében ez a távolság nagyobb egy küszöbnél) egy másik bufferbe másolják.

A számításban minden részecskére több shader példány is juthat, az erőforrások jobb kihasználása érdekében. Ekkor a feladatot a szomszédos részecskék szerint osztják meg. Például, ha a cellában 64 részecske van, akkor minden részecskére 2 shader példány jut így mindegyik példány csak a szomszédok felét dolgozza fel. Képlettel kifejezve:

$$\sum_{j=1}^{n} A(x_i, x_j) = \sum_{j=1}^{n/2} A(x_i, x_j) + \sum_{j=n/2+1}^{n} A(x_i, x_j)$$

Ebben a példában a teljes feladat (első szumma) helyett minden példány csak a számítás felét végzi el (második és harmadik szumma).

Az erőket kiszámító shader forráskódja a shaders/forcenew.comp fájlban található. Felépítése lényegében megegyezik a sűrűségszámító shaderével, csak itt a nyomási és viszkozitási tényezőkből számítja ki a részecskére ható erőt és nem csak a részecskék pozícióját, hanem sebességüket, nyomásukat és sűrűségüket is egy megosztott változóba másolja a számításhoz.



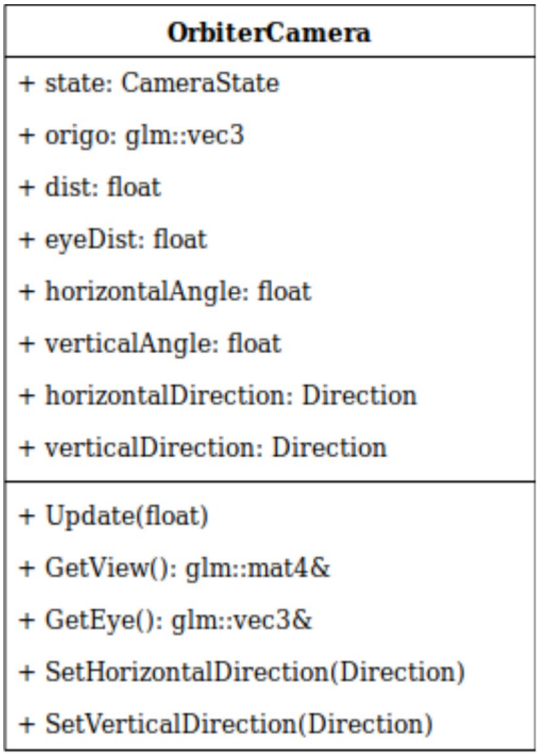
33. Ábra: RenderProgram osztály

A RenderProgram osztálya változói és metódusai a 33. ábrán láthatóak. Az SPHWaterScene a megjelenítéshez ennek az osztálynak a Render függvényét hívja meg.

A megjelenítéshez először a távolságtextúrát egy shader rajzolja ki a felületi részecskékből, amelyeket a sűrűségszámításért felelős shader határozott meg. A shadernek egy példánya jut minden részecskére és köré egy kockát raszterizál, majd a kocka celláitól vett távolság alapján frissíti a távolságtexturát. Ennek a shadernek a forráskódja a shaders/render/distancefield.comp fájlban található.

A készített távolságtextúrát egy sugárvető programmal jelenítem meg. Forráskódja a shaders/render/quad.vert és shaders/render/raycast.frag -ban található. A vertex shader egy négyzetet rak pontosan a képernyő síkjába, a fragment shadernek pedig ennek a világon belüli koordinátáit adja át. A fragment shader így a képernyő összes pixelére fut le és bemenete a pixel koordinátája a szimuláció koordináta rendszerében.

A fragment shader először a fénysugárral a szimuláció terét jelentő kockának a lapjaira számolja ki a metszéspontokat. A kocka belsejében adott távolságokat lép a fénysugár egyenese mentén. Minden ilyen ponton mintát vesz a távolságtextúrából, ebből kiszámolja az átlátszóságot az átmenetfüggvény szerint és meghatározza a felületi normált, majd megvilágítja a pontot a Blinn-Phong világításmodell szerint. Az egyenes mentén vett szín és átlátszóság értékek adják a pixel végső színét.



34. Ábra OrbiterCamera osztály

A kamera osztálya a 34. ábrán látható. Egy szatellitszerű kameramozgást valósít meg. Az *origo* középponttól *dist* távolságra kering. *HorizontalAngle* adja meg a hosszúsági szöget radiánban, *verticalAngle* pedig az egyenlítő síkjával bezárt szöget (pozitív felette negatív alatta). A vertikális szög korlátozva van ±75º-ra.

A mozgásának irányát a *horizontalDirection* és *verticalDirection* adja meg (előre, állj, hátra)

`void Update(float delta)`

A mozgás irányának és az eltelt időnek *delta* megfelelően növeli a *horizontalAngle*-t és *verticalAngle*-t és kiszámítja az ezekhez tartozó nézeti mátrixot és szempozíciót.

`glm::mat4& GetView()`

Visszaadja a kiszámolt nézeti mátrixot.

`glm::vec3& GetEye()`

Visszaadja a szem pozícióját.

`void SetHorizontalDirection(Direction dir)`

Beállítja a vízszintes mozgás irányát *dir*-re.

`void SetVerticalDirection(Direction dir)`

Beállítja a függőleges mozgás irányát *dir*-re.

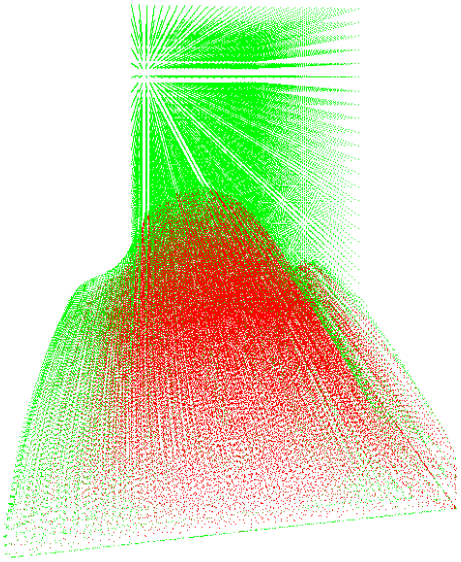
3.5 TESZTELÉS

A programot a fejlesztés közben folyamatosan teszteltem és a jelentkező hibákat javítottam. Funkcionális, modul és integrációs teszteket is végrehajtottam, fekete és fehér doboz elven is.

A shaderek működését speciális módon fehér doboz alapon teszteltem, emellett mértem a program teljesítményét a részecskeszám függvényében.

3.5.1 A SHADEREK TESZTELÉSE

A program kódjának jelentős része a videokártyán végrehajtott shader forráskódból áll. Ezeknek a tesztelése és a hibák felderítése speciális módszereket igényel.



35. Ábra

A végleges megjelenítés mellett a fejlesztés közben egy másik megjelenítési módot is alkalmaztam a tesztelésre és hibajavításra. A részecskéket pontokként jelenítettem meg

és a színüket a vizsgált tulajdonság értékei szerint állítottam be. A 35. ábrán például zöld színnel az alacsony nyomású, pirossal a magas nyomású részecskék láthatóak.

Egyes esetekben az animáció megállítása mellett vizsgáltam meg a részecskék tulajdonságait. Ekkor mivel minden pillanatban ugyanazok voltak a részecskepozíciók és a tulajdonságok, a kiszámolt értékek is ugyanazok kellett, hogy legyenek, vagyis ezzel a módszerrel azt ellenőriztem, valóban determinisztikus-e a számítás. Ez a megközelítés segített a szinkronizációs hibák felderítésében, mert ezek általában nemdeterminisztikus viselkedést eredményeztek.

Ezt a módszert alkalmazva fel tudtam fedezni a sűrűség és erőszámító shaderek hibáit, de a közelségi hálót kiszámító algoritmus és megvalósítása ezzel nem tesztelhető.

Erre a célra és a shaderek hibajavításokhoz a *RenderDoc*[4] nevű önálló grafikus debuggert használtam. Ezzel az eszközzel futás közben vizsgálható a különböző bufferek és textúrák tartalma, nyomon követhetőek a rajzoló és számítási hívások és az OpenGL API állapota is.

Ezzel a shaderek működését és az alkalmazott képleteket teszteltem fekete doboz elven vizuálisan kiértékelve és fehér doboz elven a számítások kimenetének elemzésével.

3.5.2 A KERETRENDSZER TESZTELÉSE

A CPU oldali keretrendszer tesztelésére, ami bufferek létrehozásáért, a shaderek elindításáért felelős, implementáltam egy kapcsolót (-d) amely paraméterrel indítva a programot részletes hibakeresési kimenetet kaphatunk.

Vizsgáltam a funkciók működését egymástól függetlenül és az egész rendszert együttesen is.

A hibakeresési üzenetekben megjelenhetnek például a shaderek fordítási hibái, és az OpenGL objektumok azonosítói a létrehozásukkor. Például egy buffer létrehozásakor a következő üzenetet kapjuk: „Created shader storage buffer with binding index: 1”

Így nyomon követhető, hogy a megfelelő számú buffer jön-e létre, illetve azt is, hogy ezek hogyan kapcsolódnak a shaderekhez.

3.5.3 A MEGJELENÍTÉS TESZTELÉSE

A pontfelhő megjelenítését vizuálisan ellenőriztem, illetve a távolságtextúra tartalmát *RenderDoc[4]* segítségével vizsgáltam.

3.5.4 TELJESÍTMÉNY

A szimuláció teljesítményét elsősorban a szimulált részecskék száma befolyásolja, emellett a közelségi háló felbontása is lényeges szempont. Minél több cellából áll, annál kevesebb részecske jut egy cellába, ezzel pedig kevesebb szomszéddal kell számolni. Tehát a cellák méretének megválasztásával a pontosság kárára javítható a teljesítmény.

A megjelenítés teljesítményét a képernyő felbontásán kívül távolságtextúra felbontása is befolyásolja és a sugárvetéskor a minták közötti távolság (vagyis a minták száma).

A KÖRNYEZET

A méréseket egy AMD RX580 videokártyával végeztem Ubuntu 19.04-en, Mesa 19.02 illesztőprogrammal.

A méréseket 20x20x20-as ráccsal és a megjelenítéssel mért esetben 64x64x64-es méretű távolságtextúrával illetve 0.005-ös lépésközzel végeztem.

1. Táblázat: fps a részecskeszám szerint, megjelenítéssel és anélkül

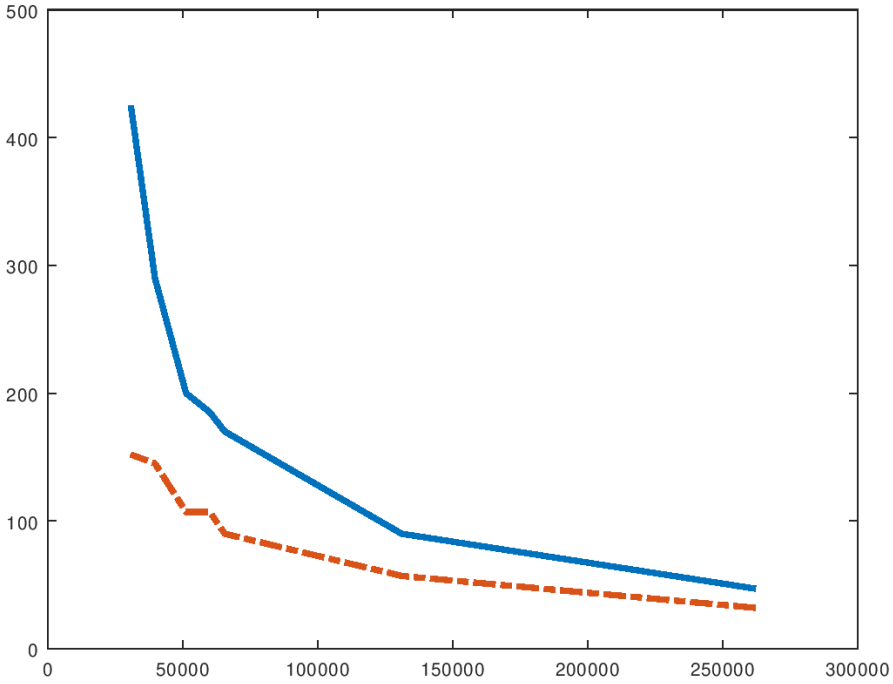
262144	**131074**	**65536**	**60000**	**51200**	**39680**	**30752**
32	57	90	107	107	145	152
47	90	170	185	200	290	425

A MÉRÉS

Minden részecskeszám esetében két mérést végeztem, az elsőt bekapcsolt megjelenítéssel a másodikat a megjelenítés nélkül. A teszt alatt kapott képfrissítések száma másodpercenként az 1. táblázatban látható.

A táblázaton kívül a mért értékeket grafikonon is ábrázoltam, ami a 36. ábrán látható. Felül a szimuláció lépéseinek száma másodpercenként megjelenítés nélkül, alatta szaggatottal a képfrissítési szám a megjelenítéssel együtt. Ezen jól látható, hogy a

teljesítmény a részecskék számának nagyjából négyzetével arányos ugyanakkora hálófelbontás esetében. Ezen túl az is jól látszik, hogy a megjelenítés költsége ugyan viszonylag magas, de a részecskeszámtól nagyjából független.



36. Ábra: képfrissítések száma a részecskék számától függően

3.5.5 TESZTELÉSI TAPASZTALATOK

A folyamatos teszteléssel a fejlesztés közben sok hibát tártam fel és javítottam. Ilyen volt például, hogy a felületi részecskéket meghatározó módszer hibája miatt a részecskék eltűntek, majd újra megjelentek. A program vizuális értékelése közben talált másik hiba volt, hogy a felület felszínén egyértelműen látszódtak a részecskék gömbök formájában. A javítását az átmenetfüggvény konstansainak megváltoztatásával értem el, illetve a világításmodellt is módosítottam emiatt. A tesztelés közben megállított szimuláció mellett a megjelenítés vibrálni látszott, aminek az oka az volt, hogy megállított szimuláció mellett is minden képkockában újra elkészült a távolságtextúra.

IRODALOMJEGYZÉK

[1] Matthias Müller, David Charypar, Markus Gross: Particle-Based Fluid Simulation for Interactive Applications, Department of Computer Science, Federal Institute of Technology Zürich (ETHZ), Svájc 2003 http://matthias-mueller-fischer.ch/publications/sca03.pdf (2019. 05. 11.)

[2] Erik Sintirn, Ulf Assarsson:, Department of Computer Science and Engineering Chalmers Univeristy of Technology Gothenburg, Svédország 2008 http://www.cse.chalmers.se/~uffe/hybridsort.pdf (2019. 05. 11)

[3] Prashant Goswarmi, Philipp Schlegel, Brabara Solenthaler, Renato Pajarola: Interactive SPH Simulation and Rendering on the GPU, Visualization and MultiMedia Lab, Department of Informatics, Univeristy of Zürich 2010 https://www.ifi.uzh.ch/vmml/publications/interactive-sph/InteractiveSPH.pdf (2019. 05. 11.)

[4] RenderDoc https://renderdoc.org/ (2019.05.11)