Eötvös Loránd Tudományegyetem

Informatikai Kar

Informatikatudományi Intézet

Numerikus Analízis Tanszék

Folyadékok szimulálása

Szerző:

Ézsől András Zalán

Programtervező informatikus BSc.

Témavezető:

Dr. Fábián Gábor

egyetemi adjunktus, informatika PhD

Budapest, 2024

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

SZAKDOLGOZAT TÉMABEJELENTŐ

Hallgató adatai:

Név: Ézsől András Zalán
Neptun kód: U0815G

Képzési adatok:

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

Belső témavezetővel rendelkezem

Témavezető neve: *Dr Fábián Gábor*

munkahelyének neve, tanszéke: **ELTE IK, Numerikus Analízis Tanszék**
munkahelyének címe: **1117, Budapest, Pázmány Péter sétány 1/C.**
beosztás és iskolai végzettsége: **egyetemi adjunktus, informatika PhD**

A szakdolgozat címe: Folyadékok szimulálása

A szakdolgozat témája:

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

Feladatom folyadékok dinamikai szimulációja 2 dimenzióban, amelyet C# nyelven Unity környezetben valósítok meg. A szimuláció alapja a Navier-Stokes egyenletek diszkretizációja, ezért a probléma egy parciális differenciálegyenlet-rendszer numerikus megoldásának előállítására vezethető vissza. A szimuláció során egy tároló edényben megjelenik majd a folyadék, melynek a felhasználói felületen beállítható a nyomása és a viszkozitása, a tároló edény alakja pedig tetszőlegesen változtatható. A folyadék az edényben alapvetően mozgatás nélkül is áramlik, de lesz lehetőség a tároló edény elmozdítására, fix falak és felhasználó által mozgatható testek hozzáadására.

Budapest, 2023. 11. 27

Tartalomjegyzék

Bevezetés

Folyadékok, illetve gázok szimulálására rengeteg esetben lehet szükség. Egy játék esetében fontos lehet a különböző vízfelületek mozgásának az animálása, vagy egy folyó áramlásának a szimulálása. Gázok mozgásának a szimulálása pedig felmerülhet abban az esetben, ha például egy füstfelhőből előkerülő objektum körül szeretnénk a füst mozgását meganimálni, vagy esetleg gőz mozgását szimulálnánk egy tűzhely körül. Játékokon kívül pedig számos mérnöki alkalmazásban hasznát vehetjük a szimulációnak. Ez lehet egy szélcsatorna szimulálása, ahol egy test körül áramlik a levegő, vagy akár egy folyadék csövekben való áramlásának a szimulálása is.

Ezen kívül az említett két felhasználás között nagy különbségek vannak. Egy mérnöki alkalmazásban például a szimuláció pontossága nagyon fontos tényező. Éppen ezért ezek a programok számolás igényesek, emiatt pedig nagyobb a hardver igényük. Ezzel szemben egy játékban működő szimulációnál a pontosság csak másodlagos szempont, itt a közel pontos megoldás is elegendő tud lenni, feltéve, ha a látvány kellő mértékben hasonlít az anyag valódi viselkedésére.

Ezen szimulációk megjelenésben nem feltétlenül különböznek sokban, ellenben a szimuláció mögötti modellek sokfélék lehetnek. Egyes modellek pontszerű testként modellezik az anyag részecskéinek a mozgását, és a megjelenítésben elmossák a részecskék közti teret, ezzel megteremtve az anyag egyben mozgásának a látszatát, más modellek pedig differenciálegyenletek segítségével szimulálják az anyag mozgását. Szakdolgozatomban az utóbbi differenciálegyenletes megközelítést fogom megvalósítani kétdimenzióban. Az elkészült program felhasználható lesz egyaránt folyadékok, illetve gázok mozgásának a szimulálására is.

Matematikai bevezetés

A feladat megoldásához a Navier-Stokes egyenletek diszkretizálásával jutunk el. Ezek parciális differenciálegyenletek, melyek egy folyadéknak vagy gáznak a mozgását írják le:

$$\frac{\partial \mathrm{u}}{\partial \mathrm{t}} = -(\mathrm{u} \cdot \nabla)\mathrm{u} - \frac{1}{\rho}\nabla \mathrm{p} + \nu \nabla^2 \mathrm{u} + \mathrm{F}$$

$$\nabla \cdot \mathrm{u} = 0$$

$$\frac{\partial \rho}{\partial \mathrm{t}} = -(\mathrm{u} \cdot \nabla)\rho - \frac{1}{\rho}\nabla \mathrm{p} + \kappa \nabla^2 \rho + \mathrm{S}$$

Ahol (a feladat 2 dimenzióra korlátozása miatt):

- **u** az anyag adott helyen és időpillanatban lévő sebességét adja meg. Tehát **u** egy vektor-vektor függvény, jelen esetben $\mathbf{u} \in \mathbb{R}^3 \to \mathbb{R}^2$.
- **ρ** az anyag adott helyen és időpillanatban lévő sűrűségét adja meg. Tehát **ρ** egy nemnegatív értékű vektor-skalár függvény, jelen esetben $\boldsymbol{\rho} \in \mathbb{R}^3 \to \mathbb{R}$
- **p** az anyag adott helyen és időpillanatban lévő nyomását adja meg. Tehát **p** egy vektor skalár függvény, jelen esetben $\mathbf{p} \in \mathbb{R}^3 \to \mathbb{R}$
- **ν** az anyag viszkozitását leíró skalár. A viszkozitás jellemzi, hogy egy adott anyagban a diffúzió milyen gyorsan megy végbe.
- **F** az anyagra ható külső erőt adja meg. Tehát **F** egy kétdimenziós vektor.
- **S** egy külső tényezőből származó sűrűséget ad meg. Tehát **S** skalár.

A vektoranalízis operátorai a következő alpontban lesznek definiálva:

Legyen $\mathrm{f} \in \mathbb{R}^3 \to \mathbb{R}^2$, ekkor:

- $\mathrm{grad}(\mathrm{f}) = \nabla \mathrm{f} = \left(\frac{\partial \mathrm{f}}{\partial \mathrm{x}}, \frac{\partial \mathrm{f}}{\partial \mathrm{y}}\right)$ az f gradiense, vagyis a parciális deriváltakból készített vektor
- $\nabla \cdot \mathrm{f} = \frac{\partial \mathrm{f}}{\partial \mathrm{x}} + \frac{\partial \mathrm{f}}{\partial \mathrm{x}}$ az f vektormező divergenciája/forrássűrűsége, ami az elsőrendű parciális deriváltak összege, ami gyakorlati értelemben azt méri, hogy az f mennyire viselkedik forrásként vagy nyelőként egy adott pontban
- $\nabla^2 \mathrm{f} = \nabla \cdot \nabla \mathrm{f} = \frac{\partial^2 \mathrm{f}}{\partial \mathrm{x}^2} + \frac{\partial^2 \mathrm{f}}{\partial \mathrm{y}^2}$ az f Laplace operátora, ami a gradiens divergenciája, vagy a másodrendű parciális deriváltak összege

- $f \cdot \nabla f = f_x \frac{\partial f}{\partial x} + f_y \frac{\partial f}{\partial y}$ az f advekciós operátora, ami az elsőrendű parciális deriváltak és a hozzá tartozó koordináta függvények szorzatának az összege, ami gyakorlati értelemben azt írja le, hogy az anyag a sebesség mező alapján mozog

Ezen operátorok ismeretében részletesen tárgyalhatjuk az egyenleteket:

A sebességre vonatkozó első egyenlet jobb oldala négy részre bontható fel (balról jobbra haladva):

- Az első az advekciós lépés, ami leírja, hogy az anyag mozgása a sebességmező alapján fog történni.
- A második a nyomásra vonatkozó rész, ennek a kiszámítása később lesz részletesen leírva.
- A harmadik a diffúziós lépés, ami leírja, hogy az anyagra hat a diffúzió. A diffúzió értelmében a nagyobb sebesség vektorok eloszlanak a környező vektorokba. Ez később lesz részletesen tárgyalva.
- A negyedik lépés a külső erők hozzáadása. Ezek lehetnek olyan erők, amik folyamatosan hatnak az egész anyagra, mint például a gravitációs erő, vagy lehetnek csak az anyag egy részére ható erők, mint például hullámok indítása egy adott pontból.

A sebességre vonatkozó második egyenlet:

- A sebesség mező divergenciája 0 az egész tartományon. Ez fontos, mivel garantálja a peremfeltételekkel együtt (később lesz tárgyalva), hogy az anyagnak állandó a tömege (jelen esetben sűrűsége), tehát nem folyik el anyag a semmibe, illetve nem is termelődik anyag a semmiből.

A sűrűségre vonatkozó egyenlet egy az egyben megegyezik a sebességre vonatkozó első egyenlettel, azzal a különbséggel, hogy a sebesség vektor értéke helyett a sűrűség skalár értékére vonatkozik.

Ezek ismeretében tárgyalhatjuk az egyenletek megoldását. A Navier-Stokes egyenletek folytonos értelmezési tartományon nem megoldhatóak, és számítógépen egyébként is diszkrét tartományok adhatók csak meg. Ennek értelmében az egyenletek diszkretizálására van szükség. Itt több lehetőség is fennáll (végeselem módszer, véges térfogat módszer). A továbbiakban a végeselem módszer lesz alkalmazva.

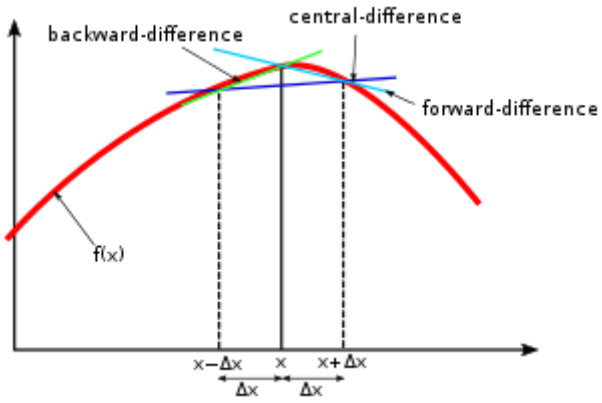
Első lépésben a függvények értelmezési tartományát diszkrét részekre osztjuk fel, ami gyakorlati értelemben azt jelenti, hogy a szimuláció egy diszkét rácson fog történni. A rács reprezentációja egy kétdimenziós tömb lesz, a tömb elemei pedig a rácspontoknak feleltethetők meg.

Következő lépés, hogy az időt is felosztjuk diszkrét időintervallumokra, és ez alapján végezzük a keresett függvény interpolálását.

Ezzel megadtuk a függvények értelmezési tartományának a diszkrét verzióját.

Szükség van még a deriváltak és az előbb bevezetett vektoranalízis operátorok diszkrét verziójának megadására. Ezek kiszámítására a véges differencia módszer lesz alkalmazva.

Véges differencia módszert alkalmazva három lehetőség van a derivált kiszámítására: előre mutató, hátra mutató és központi differencia számítás.



1. ábra: Véges differencia számítás

A továbbiakban az előre mutató lesz használva. Ezek alapján a differencia hányados a következőképpen módosul a folytonos esethez képest:

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h} \approx \frac{f(x_0 + h) - f(x_0)}{h} + O(h)$$

Az előbb felírt eredmények felhasználva, és az előző pontban bevezetett értelmezési tartomány diszkretizálási módszert alkalmazva az operátorok kiszámítása a következőképpen egyszerűsödik (az előző f jelölést használva, és $f_{i,j} = f(i, j)$, ahol $(i,j) \in D_f$):

$$\nabla f = \left(\frac{f_{i+1,j} - f_{i-1,j}}{2\delta x}, \frac{f_{i,j+1} - f_{i,j-1}}{2\delta y} \right)$$

$$\nabla \cdot f = \frac{f_{i+1,j} - f_{i-1,j}}{2\delta x} + \frac{f_{i,j+1} - f_{i,j-1}}{2\delta y}$$

$$\nabla^2 f = \frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{(\delta x)^2} + \frac{f_{i,j+1} - 2f_{i,j} + f_{i,j-1}}{(\delta y)^2}$$

Ahol δx és δy a cellák szélessége, illetve magassága. A továbbiakban a $\delta x = \delta y = 1$ feltételezéssel fogunk élni. Ezek alapján az operátorok az alábbiak szerint egyszerűsödnek:

$$\nabla f = \left(\frac{f_{i+1,j} - f_{i-1,j}}{2}, \frac{f_{i,j+1} - f_{i,j-1}}{2}\right)$$

$$\nabla \cdot f = \frac{f_{i+1,j} - f_{i-1,j} + f_{i,j+1} - f_{i,j-1}}{2}$$

$$\nabla^2 f = f_{i+1,j} - 2f_{i,j} + f_{i-1,j} + f_{i,j+1} - 2f_{i,j} + f_{i,j-1} = f_{i+1,j} + f_{i-1,j} + f_{i,j+1} + f_{i,j-1} - 4f_{i,j}$$

Ezzel minden készen áll a differenciálegyenlet megoldó algoritmus felépítéséhez. Az algoritmus egy iterációja egy lépésben ki fogja számolni a sebesség mezőt majd ezen a sebességmezőn fogja a sűrűség mezőt mozgatni.

Mivel az összes lépés egy metódusban való számítása nem lenne effektív, ezért több lépésre bontjuk az egyenlet megoldását. A sebességre vonatkozó első egyenlet és a sűrűségre vonatkozó egyenlet hasonló felépítésű, tehát ezen túl a sebesség mező számítása lesz leírva, ebből a sűrűség mező számítása analóg módon történik.

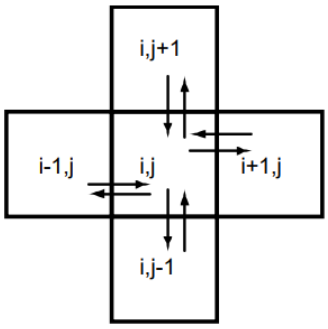
Fontos még megjegyezni, hogy a korábban említett kétdimenziós tömbös reprezentáció úgy értendő, hogy a tartomány négy szélén lévő egy cella széles sávok értékeit nem az egyenlet megoldó alapján számítjuk, hanem a peremértékek alapján adjuk meg. Ez szintén később lesz kifejtve.

Diffúziós lépés:

A diszkretizáció miatt minden cella a négy szomszédjával cserél értékeket az alábbi módon:

Jelölje uprev az előző időegységben az **u** cella értékét, **a** pedig a diffúziós állandóval kapcsolatos arányossági tényezőt

$$u_{i,j} = uprev_{i,j} + a(uprev_{i+1,j} + uprev_{i-1,j} + uprev_{i,j+1} + uprev_{i,j-1} - 4uprev_{i,j})$$



2. ábra: Diffúzió számítás

Ezt a képletet egy az egyben implementálni is lehetne, azonban így az algoritmus nem lenne stabil tetszőleges tartomány méretre, időegységre és diffúziós állandóra. Ennek érdekében az egyenletet átrendezve a következő összefüggés adódik:

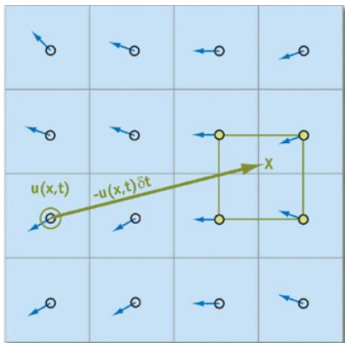
$$\mathrm{uprev_{i,j}} = \mathrm{u_{i,j}} - \mathrm{a}(\mathrm{uprev_{i+1,j}} + \mathrm{uprev_{i-1,j}} + \mathrm{uprev_{i,j+1}} + \mathrm{uprev_{i,j-1}} - 4\mathrm{uprev_{i,j}})$$

Ez pedig egy lineáris egyenletrendszer $u_{i,j}$-re vonatkozóan, ami egy iterációs módszerrel, jelen esetben Gauss-Seidel iterációval megoldható.

Advekciós lépés:

Az advekciós lépésben minden cella értékre meg kell tudnunk mondani, hogy a sebességmező melyik cellába viszi az adott cellát. Ezt lehetne olyan módon implementálni, hogy időben előre lépünk és kiszámoljuk a cella helyét. De ez a megoldás nem vezetne eredményre, mivel a tömbös reprezentáció miatt elég kicsi a valószínűsége, hogy a sebességmező eleme egy cellát pontosan egész koordinátájú cellába vinne. Ebben az esetben pedig nem tudnánk megindexelni a tömböt valós számmal.

Ennek kiküszöbölése érdekében nem előre lépünk az időben, hanem azt számoljuk ki, hogy azok a cellaértékek, amelyeknek most egész a koordinátája hogyan jutottak abba az állapotba az előző időpillanatban. Ezután pedig a kiszámolt cellapozíciót bilineárisan interpoláljuk a négy szomszédja alapján és ezt az értéket adjuk az adott cellának.



3. ábra: Advekció számítás bilineáris interpolációval

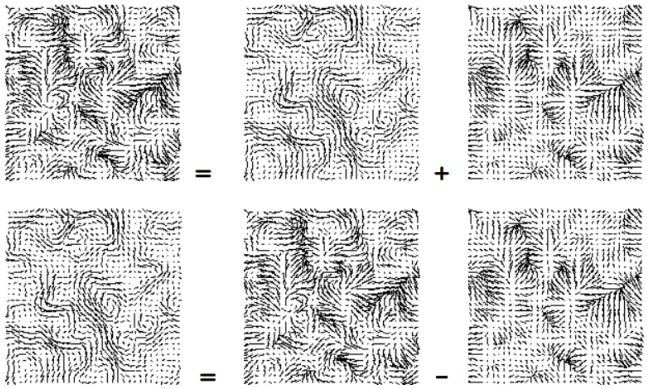
Külső erős lépés:

A gravitációt nem a megszokott $\mathbf{F} = \mathbf{mg}$ képlet alapján számoljuk, mivel a tömeg nincs értelmezve a modellben. Helyette sűrűségekkel számolunk, így $\mathbf{F} = \boldsymbol{\rho}\mathbf{g}$. Ezt az időegységgel korrigálva minden időegységben hozzá kell adni minden cella értékéhez.

Az anyag egy részére ható erők esetében pedig szintén az időegységgel korrigálva kell hozzáadni az értéket az adott cella értékéhez.

Projekciós lépés:

A sebességre vonatkozó második egyenlet megköveteli, hogy a sebességmezőnek 0 legyen a divergenciája. Ennek a 0 divergenciájú mezőnek az előállítása lesz az algoritmus utolsó lépése. A mező előállításához szükségünk van a Helmholtz-Hogde dekompozíciós tételre.



4. ábra: Helmholtz-Hodge mező dekompozíció

A tétel kimondja, hogy minden vektormező előáll egy 0 divergenciájú vektormező és egy skalármező gradiensének az összegeként. Ez a két mező pedig rendelkezésre is áll, mivel a sebességmező, amit az algoritmus kiszámol egy nem 0 divergenciájú vektormező lesz, a skalármező gradiense pedig később részletezett módon előállítható a nyomás mező

gradienseként. Innen pedig már csak ki kell vonnunk a sebességmezőből a nyomás mező gradiensét és megkapjuk a kívánt 0 gradienssel rendelkező sebességmezőt.

Mindez egzakt jelölésekkel:

Helmholtz-Hodge dekompozíciós tétel (a tétel bizonyítása az irodalomjegyzékben található linken elérhető):

Legyen w és u két vektormező, $\nabla \cdot \mathbf{u} = 0$, p egy skalár mező, ezen kívül legyen **D** egy tartomány, amin az előbbi függvények értelmezve vannak. Legyen ezen kívül $\partial\mathbf{D}$ (a tartomány határa) sima, tehát egyszeresen differenciálható. Legyen **n** a **D** tartomány felületi normálisa, $\partial\mathbf{D}$ pedig legyen párhuzamos **u**-val. Ez azt jelenti, hogy $\boldsymbol{n} = 0$ $\partial\mathbf{D}$-n, vagyis **D** határán. Ekkor:

$$\mathrm{w} = \mathrm{u} + \nabla\mathrm{p}$$

Ezek alapján, a tételben szereplő egyenletre a divergencia operátort alkalmazva a következő adódik:

$$\nabla \cdot \mathrm{w} = \nabla \cdot \mathrm{u} + \nabla \cdot \nabla\mathrm{p}$$

$$\nabla \cdot \mathrm{w} = \nabla \cdot \mathrm{u} + \nabla^2\mathrm{p}$$

$$\nabla^2\mathrm{p} = \nabla \cdot \mathrm{w}$$

Ez pedig egy Poisson egyenlet. Ehhez definiáljuk a Poisson egyenlet általános alakját.

$$\nabla^2\mathrm{f} = \mathrm{g}$$

Ahol f és g valós, -vagy komplex értékű függvények, amelyek egy tartományon vannak értelmezve. Az előbbi feltételek pedig teljesülnek a modellünkre.

Ezek után a Poisson egyenlet megoldását kell megadnunk, ami a már korábban említett Gauss-Seidel iterációval megoldható.

A projekciós lépés következő része, hogy definiálunk egy projekciós operátort. Legyen ez az operátor $\mathbb{P}$, ami a dekompozíciós tétel alapján egy vektormezőt (a tételben **w**) projektál a divergencia mentes komponensére (a tételben **u**). $\mathbb{P}$-t alkalmazva a dekompozíciós egyenletre a következő adódik:

$$\mathbb{P}\mathrm{w} = \mathbb{P}\mathrm{u} + \mathbb{P}(\nabla\mathrm{p})$$

Viszont az előzőek alapján $\mathbb{P}w = u$, a projekció tulajdonságai alapján pedig $\mathbb{P}u = u$. Mindezekből pedig következik, hogy $\mathbb{P}(\nabla p) = 0$. Ezek után alkalmazzuk $\mathbb{P}$-t az első egyenletre:

$$\mathbb{P}\frac{\partial u}{\partial t} = \mathbb{P}\left(-(u \cdot \nabla)u - \frac{1}{\rho}\nabla p + \nu\nabla^2 u + F\right)$$

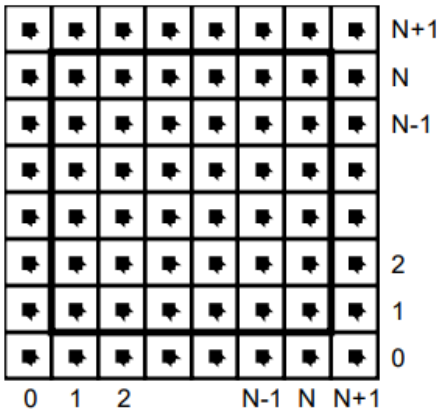
$$\frac{\partial u}{\partial t} = \mathbb{P}(-(u \cdot \nabla)u + \nu\nabla^2 u + F)$$

Tehát a projekciós operátor bevezetésével sikeresen eliminálunk egy változót az egyenletből, ezzel pedig el is készült a projekciós lépésünk.

A lépéseken kívül egy dolog maradt még ki az algoritmusból, ez pedig a peremfeltételek tárgyalása. Peremfeltételből külső és belső peremfeltételeket különböztetünk meg.

Matematikai értelemben nem teszünk a kettő között különbséget, azonban értelmezésben eltérnek egymástól. Számolás szempontjából a peremfeltételeknek több fajta megvalósítása is létezik, ezek közül a Neumann, és a No-slip peremfeltételek lesznek használva.

A külső peremfeltételek a tartomány szélein lesznek értelmezve, jelen esetben a tároló edény négy falán.



5. ábra: Tartomány külső pereme

Ezzel szemben a belső peremfeltételek a tartomány belső részén lesznek értelmezve, tehát az esetleges belső falakon. Ez a tömbös reprezentációban azt fogja jelenteni, hogy bizonyos belső cellákat peremként fogunk kezelni, amelyeknek pedig az elhelyezkedésük szempontjából több fajtája lehet: bal felső sarok, felső, jobb felső sarok, jobb, jobb alsó sarok, alsó, bal alsó sarok, bal.

Mindez azt jelenti, hogy az adott cella megnevezett szomszéd cellái közül melyik cella perem cella, és melyik nem. Tehát a bal felső esetében a bal és a felső szomszéd cella nem perem cella, a többi szomszéd cella pedig perem cella.

Az előbbi kategorizálásra egyrészt a peremfeltételek számítása miatt, másrészt pedig a peremfeltételek egyértelműsége miatt van szükség. Ugyanis abban az esetben, ha egy cellának kettőnél több nem perem cella szomszédja lenne, abból nem lehetne egyértelműen megmondani, hogy az adott sűrűség melyik cellából, illetve milyen irányból érkezett a peremre. Ezen megfontolásból a továbbiakban minden olyan cellát, amely az előbbi feltételnek nem tesz eleget nem valós peremnek fogunk tekinteni. Ez esetben pedig a peremfeltételek számítása nem definiált.

Külön megjegyezhető az előbbiek alapján, hogy a külső peremek megfeleltethetők a megfelelő belső perem fajtáknak: bal perem – jobb belső perem, alsó perem – felső belső perem stb. Ezt a megfeleltetést kihasználva nézzük meg a peremfeltételek kiszámítását, kezdve az egy darab nem perem cella szomszéddal rendelkező cellákkal.

Elsőként a Neumann peremfeltétel kimondja, hogy a peremeken az első derivált 0 kell, hogy legyen. A korábban tárgyalt előre mutató véges differencia számolással ez a következőt fogja jelenteni:

$$\frac{f_{i,j} - f_{0,j}}{\delta x} = 0$$

$$f_{0,j} = f_{i,j}$$

Ahol $f_{0,j}$ a számítani kívánt perem cella, $f_{i,j}$ pedig a nem perem szomszéd cellát jelöli. A képlet csak a bal oldali perem számítását írja le, a többi perem számítása ezzel analóg módon történik. A Neumann peremfeltétel lesz alkalmazva a nyomás, divergencia, illetve a sűrűség esetében is.

A következő peremfeltétel a No-slip peremfeltétel. Ez kimondja, hogy az anyag nem folyhat át a peremeken, tehát a sebesség megfelelő komponense 0 lesz a peremen. Vagyis a függőleges peremeken a sebesség vízszintes komponense, a vízszintes peremeken pedig a sebesség függőleges komponense legyen 0. Ez a tömbös reprezentációnál azt fogja jelenteni, hogy a két

cella átlaga lesz 0, mivel alapvetően a két cella között kellene a sebességnek 0-nak lennie, viszont az értékeket a cellák középpontjában tároljuk:

$$\frac{u_{0,j} + u_{i,j}}{2} = 0$$

$$u_{0,j} = -u_{i,j}$$

Ahol $u_{0,j}$ a számítani kívánt perem cella, $u_{i,j}$ pedig a nem perem szomszéd cellát jelöli. A képlet csak a bal oldali perem számítását írja le, a többi perem számítása ezzel analóg módon történik. A No-slip peremfeltétel lesz alkalmazva a sebesség esetében.

Ezt követően pedig következhetnek a két nem perem cella szomszéddal rendelkező perem cellák. Itt minden cella a két nem perem cella értékének az átlagát kapja majd értékül. Az előbbi képlet lesz alkalmazva minden korábbi peremfeltétel (Neumann, No-slip) esetén.

Külön megjegyezhető még, hogy elméletileg megengedett az olyan belső perem cellák használata, amelyeknek két szomszédos nem perem cella szomszédjuk van, de ezen két cellával szomszédos cella szintén perem cella. Ellenben a gyakorlatban látható, hogy egyrészt ez a szimulált anyag peremeken való átszivárgásához vezetne, másrészt pedig a belső peremek osztályozásának egyértelműségét elrontaná. Ezen megfontolásból az előbb leírt belső peremekre a peremfeltétel számítása szintén nem definiált. Mindezekről részletes leírás a felhasználói dokumentációban található.

Mindezek után, pedig minden készen áll arra, hogy definiáljuk az egyenlet megoldó algoritmusunk egy lépését:

Vezessünk be minden lépésnek egy külön operátort:

- külső erős lépés: $\mathbb{F}$
- diffúziós lépés: $\mathbb{D}$
- advekciós lépés: $\mathbb{A}$
- projekciós lépés: $\mathbb{P}$
- teljes algoritmus lépése: $\mathbb{S}$

Így tehát:

$$\mathbb{S} = \mathbb{P} \circ \mathbb{A} \circ \mathbb{D} \circ \mathbb{F}$$

Felhasználói dokumentáció

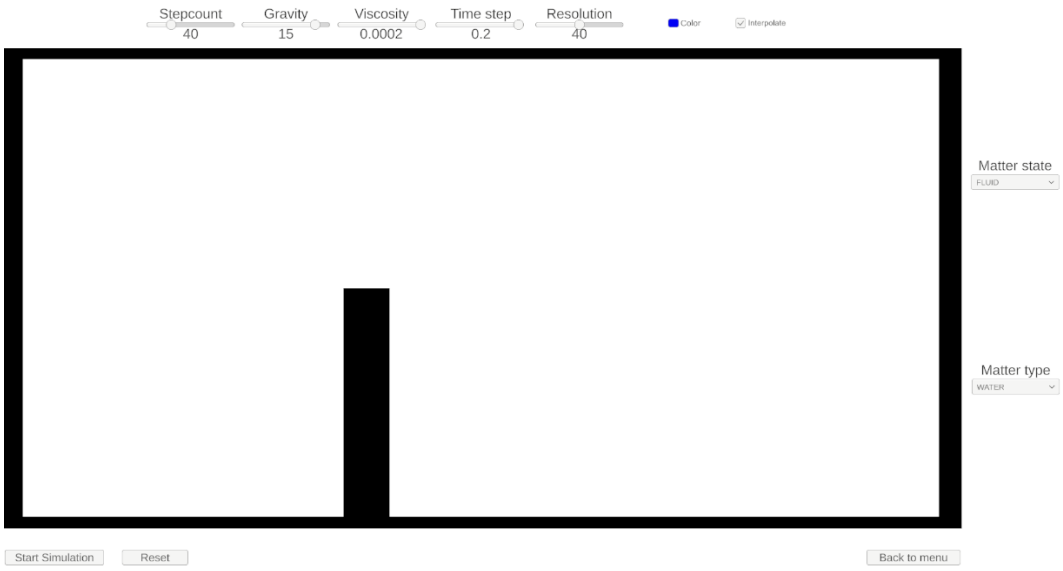
Első lépésben töltsük le a szakdolgozatot és csomagoljuk ki a fájlokat. A kicsomagolás utáni mappaszerkezet a következő lesz: a „Fluid Simulation” mappában találhatóak az alkalmazás fejlesztéséhez szükséges forrásfájlok, az „Application” mappában pedig az alkalmazás futtatásához szükséges fájlok. Maga a program az „Application” mappában található „Fluid Simulation.exe” alkalmazás futtatásával indítható el.

Az alkalmazás alapvetően három fő ablakból áll: menü, szerkesztő és szimulációs ablakból. A következőkben ezek bemutatása fog megtörténni.

A program elindítása után megnyílik a menü ablak. Itt két gomb található: az „Editor” gomb megnyomásával tovább lehet lépni a szerkesztő ablakba, az „Exit” gomb megnyomásával pedig a felhasználó bezárhatja az alkalmazást.

6. ábra: Menü ablak

A szerkesztő ablakot megnyitva a felhasználónak lehetősége van személyre szabni a szimulációs felületet, beállítani a szimulálni kívánt anyag tulajdonságait, illetve elindítani a szimulációt.



7. ábra: Szerkesztői ablak

A szerkesztő ablak közepén található egy fekete keretű, fehér, négyzet alakú ablak. Ez az ablak lesz az a hely, ahol a szimuláció ténylegesen végbe fog menni. A felhasználó az ablakra bal egérgombbal kattintva feketére színezheti az ablak egyes pixeleit, illetve jobb egérgombbal kattintva a fekete pixeleket vissza színezheti fehérre. Mindkét esetben az egérgombok lenyomva tartásával folyamatosan lehet színezni, illetve törölni. A program ezen funkciója arra szolgál, hogy majd a későbbiekben tárgyalt szimulációs ablakban a szerkesztő felületen beszínezett fekete pixelek fognak a belső falaknak megfelelni. Tehát a beszínezett pixeleken a szimuláció során nem lehet szimulált anyag, és ezen pixelekkel az anyag megfelelően reagál majd.

A színezésnél és a törlésnél fontos megjegyezni, hogy csak 2x2-es négyzetek színezése és törlése engedélyezett, olyan értelemben, hogy mind színezésél, mind pedig törlésnél az egér pozíciója az adott négyzet bal felső sarkát fogja jelenteni. Továbbá a törlés csak azokban az esetekben lehetséges, ha a törlés után esetlegesen megmaradt fekete falakra teljesülnek a matematikai bevezetésben tárgyalt feltételek.

Ezen kívül a színezés feltétele még, hogy a színezni kívánt négyzet minden pixele a külső falakon belül legyen. Tehát nem lehet például a falakra és a jobb, illetve alsó fal melletti egy pixeles sávra se színezni. Az utolsó megkötés a színezéssel kapcsolatban az, hogy színezni nem lehet abban az esetben, ha a kirajzolt minta sakktábla-szerű lenne. Ez azt jelenti, hogy a beszínezett

négyzetek vagy nem érintkeznek egymással, vagy pedig a pixeleik lapszomszédok kell, hogy legyenek, azaz, ha a pixelek csak sarokszomszédok az nem elegendő a helyes működéshez.

Mindezen megszorításokra azért van szükség, mert a fekete pixelek a matematika bevezetés részben tárgyalt belső peremfeltételeknek feleltethetők meg. Ezek egyértelműségét biztosítják a megszorítások, amik pedig elengedhetetlenek a pontos szimulációhoz. Fontos még megjegyezni, hogy felhasználói szempontból, ha nem elvégezhető a színezés vagy a törlés, akkor egyszerűen nem történik meg a művelet.

A szerkesztő ablak felső sávjában találhatóak az anyag és a szimuláció tulajdonságait beállító vezérlők:

- A „Start simulation” gombra megnyílik a szimulációs ablak és elindul a szimuláció a beállított paraméterekkel, és kirajzolt szimulációs térrel.
- A „Reset” gombra kattintva az összes felhasználói vezérlő, és a szimulációs felület is visszaállítható az alapértelmezett állapotába.
- A „Stepcount” csúszka állításával szabályozható a matematikai bevezetés részben tárgyalt iterációs algoritmus lépésszáma. A lépésszám értéke a csúszka felett látható egész szám. Ezen értékkel szabályozható az integrálás pontossága, ami magával vonja a szimuláció pontosságát is. Itt fontos megjegyezni, hogy kis lépésszám esetén elég pontatlan eredmény érhető el, ellenben a program hatékonysága jobb lesz a kisebb hardver igény miatt. Nagyobb lépésszám esetén pedig javítható a szimuláció pontossága, azzal a hátránnyal, hogy így jóval lassabb és hardver igényesebb lesz a szimuláció.
- A „Gravity” csúszka állításával szabályozható a gravitáció mértéke. A gravitáció értéke a csúszka felett látható valós szám. Ezen érték felvehet pozitív, illetve negatív értékeket is egyarán. A pozitív érték a szimuláció szempontjából azt fogja jelenteni, hogy a szimulált anyag milyen gyorsan esik lefelé, a negatív érték pedig, hogy milyen gyorsan száll az anyag felfelé a szimulációs térben.
- A „Viscosity” csúszka állításával szabályozható az anyag viszkozitása. A viszkozitás értéke a csúszka felett látható valós szám. Ezen érték megadja, hogy az anyag milyen gyorsan oszlik el a szimulációs térben. A gyorsulás mértéke egyenesen arányos a

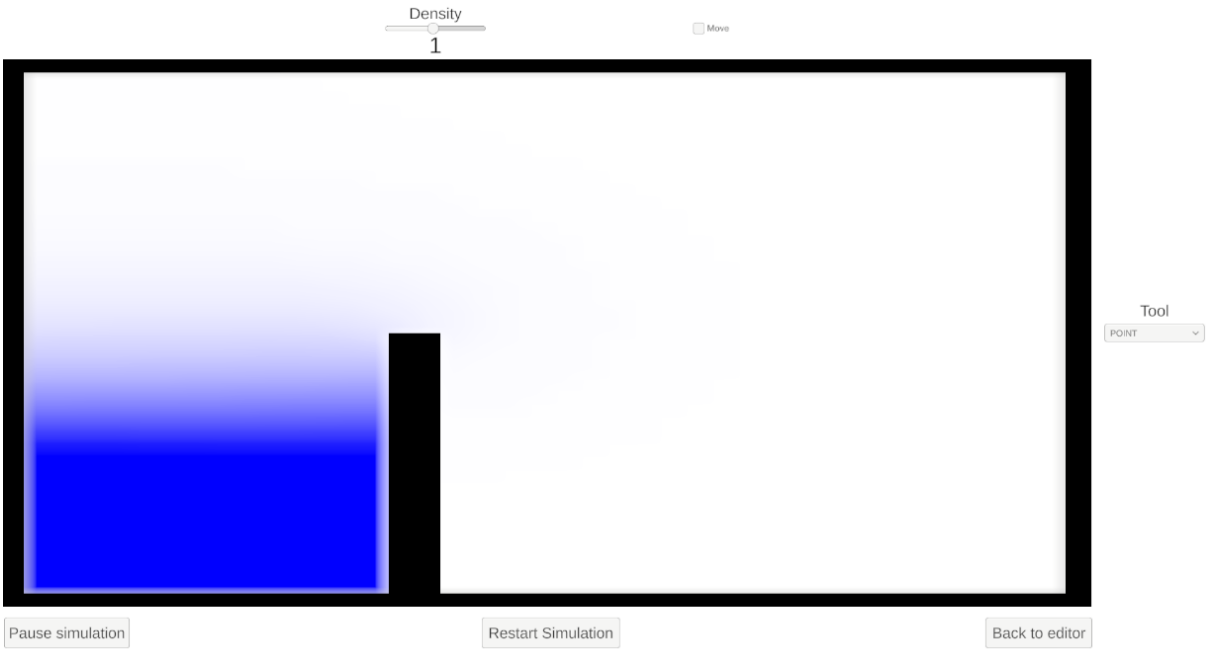
viszkozitás értékével. Megjegyezhető, hogy kis érték megadása gázok szimulálására megfelelő, nagyobb érték pedig folyadékok szimulálására.

- A „Time step” csúszka állításával szabályozható a szimuláció pontossága, illetve gyorsasága. Az időbeli megváltozás értéke a csúszka felett látható valós szám. Ezen érték megadja, a matematikai bevezetés részben tárgyalt integrálás pontosságát. Tehát egyfelől szabályozható vele a szimuláció pontossága. A pontosság fordítottan arányos az időbeli megváltozás értékével. Másfelől pedig az időbeli megváltozás egyenesen arányos a szimuláció gyorsaságával, mivel azt is jelenti, hogy milyen gyorsan telik az idő a szimuláció közben.
- A „Resolution” csúszka állításával szabályozható a szimulációs felület mérete, tehát a felbontása. A felbontás értéke a csúszka felett látható egész szám. Ezen érték megadja, hogy mennyi pixel legyen az oldalhossza a szimulációs felületnek. A csúszka állításával dinamikusan változik a mérettel együtt a pixelek nagysága. Állításkor újra generálódik az egész felület, tehát az eddig rajzolt falak állításkor eltűnnek, és az egész felület fehér lesz. Fontos megjegyezni, hogy a felbontás nagy értékre állítása drasztikusan növeli a műveletigényt, tehát csak erős hardverrel ajánlott.
- A „Matter type” legördülő listával kiválasztható egy előre beépített anyag minden tulajdonságával együtt. Az anyag kiválasztása beállítja a megfelelő értékre a vezérlőket, és a szimuláció elindításával az adott anyag szimulálása fog megtörténni. Például méz kiválasztása esetén a szimulált anyag lassan folyó sárga színű és nagy viszkozitású méz szerű anyagot eredményez. A listából egy anyag kiválasztása után a vezérlők értékei továbbra is módosíthatók maradnak, viszont egy érték átállítása esetén a lista értéke átvált „CUSTOM” értékre. Hasonló meggondolásból, ha a vezérlők értékei pont egy előre megadott anyag értékeinek egyeznek meg, akkor a lista felveszi az adott anyag értékét. Ez lehetővé teszi, hogy egy adott anyaggal kísérletezzen a felhasználó, például a méz valamely tulajdonságán kicsit változtatva milyen módon változik a szimuláció.
- A „Matter state” legördülő listával kiválasztható a szimulálni kívánt anyag halmazállapota. Ez két értéket vehet fel: folyékony, illetve gáz. A két érték között a szimuláció során a megjelenítésben van különbség. Folyékony halmazállapot esetén a matematikai bevezetésben tárgyalt diffúzió is hat az anyagra, gáz esetén pedig nem.
- A „Color” gombra kattintva felugrik egy színes ablak a szimulációs felület előtt. Ez után a színes ablak egy pixelére kattintva állítható be a szimulált anyag színe. A szín jelen

esetben az ablak egy pixelének a színét jelenti. Folyamatos kattintás esetén a szín dinamikusan változik, követve a kurzort. Addig, amíg a szín állító ablak látható, addig nem lehet a szimulációt elindítani, illetve nem lehet a szimulációs felületre rajzolni, vagy onnan törölni. Látható színválasztó ablak esetén a „Color” gombra kattintva a színválasztó ablak eltűnik, és az addig blokkolt vezérlők újra állíthatóvá válnak.

- Az „Interpolate” kapcsolóval állítható be az anyag szimuláció közbeni megjelenése. Bekapcsolt állapotban a szimuláció során az anyag felülete sima lesz, ezzel szemben kikapcsolt állapotban pixeles képet kap a felhasználó.
- A „Back to menu” gombra kattintva a felhasználó visszakerül a menü ablakba és az eddigi beállításai elvesznek.

A szimulációs ablakban középen a szerkesztő ablaknak megfelelően megjelenik a szerkesztőben kirajzolt pálya. Itt fog majd a szimuláció zajlani.



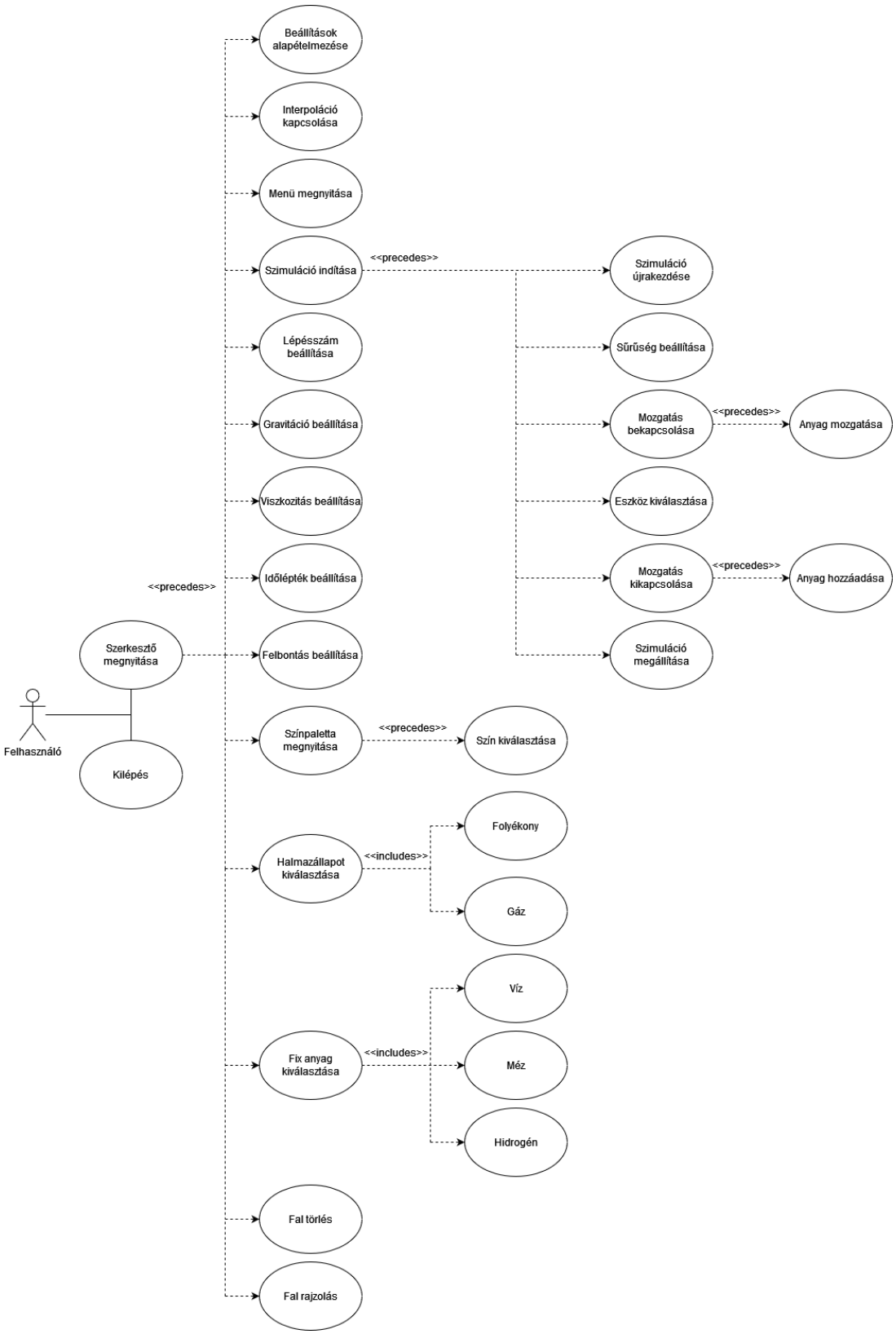
8. ábra: Szimulációs ablak

A szimulációs ablak alsó, illetve felső sávjában találhatóak a szimuláció menetével kapcsolatos, és az anyaggal való interakciót lehetővé tevő vezérlők:

- A „Pause simulation” gombra kattintva a szimuláció megáll, illetve az anyaggal való interakció nem lesz lehetséges. Ezek után, ha a felhasználó még egyszer lenyomja a gombot, akkor a szimuláció folytatódik onnan, ahol meg lett állítva.

- A „Restart simulation” gombra kattintva a felhasználó újrakezdheti a szimulációt. Ez azt jelenti, hogy a megrajzolt falak a helyükön maradnak, viszont az eddig a szimulációs térbe helyezett anyag eltűnik.
- A „Density” csúszka állításával szabályozható az, hogy egy kattintással mennyi anyagot adagol a felhasználó a szimulációba. A sűrűség értéke a csúszka felett látható egész szám. A felhasználó minél nagyobb sűrűség értéket juttat a szimulációba, annál hamarabb fog megtelni a szimulációs tér.
- A „Move” kapcsoló ki/be kapcsolásával a felhasználó válthat a szimuláció interakcióinak két fajtája között:
    - Kikapcsolt állapotban, ha a felhasználó a szimulációs felületre kattint, akkor a szerkesztő ablakban beállított anyagból abban a pontban anyagot juttat a szimulációs térbe. Folyamatos kattintás mellett a felhasználó folyamatosan adagolhatja az anyagot, arra a pontra, ahol a kurzor van.
    - Bekapcsolt állapotban ellenben a felhasználó a szimulációs felület egy pontjára kattintva a szimulációs térben lévő anyagot mozgatni tudja. Ez úgy valósul meg, hogy az egeret lenyomva tartva az egér mozgásának az irányába fog elmozdulni az anyag. Ezen kívül bekapcsolt állapotban piros színnel megjelenik az egér pozíciója a felhasználói felületen. Ez segíti a felhasználót abban, hogy láthassa egyrészt az egér pozícióját, másrészt a kiválasztott eszközt.
- A „Tool” legördülő listából a felhasználó kiválaszthatja, hogy milyen eszközzel akar a szimulációba belenyúlni. Fontos, hogy a kiválasztott eszköz csak akkor használható, ha a „Move” kapcsoló bekapcsolt állapotban van. A legördülő listából egy eszközt kiválasztva az adott eszköz fog pirosan megjelenni a kurzoron feltéve, hogy a kurzor a felhasználói felület felett van. Ez azt jelenti, hogy például a „POINT”-ot kiválasztva egy darab piros pixel fog megjelenni, a téglalapot kiválasztva pedig egy 2x3-as piros téglalap. Ezen kívül az eszközzel való anyag mozgatás a szimulációban azt fogja jelenteni, hogy minél nagyobb eszközzel mozgatjuk az anyagot, annál nagyobb mozgást tudunk benne kiváltani. Az eszköz kiválasztásához tartozik még, hogy minden eszköz úgy jelenik meg, hogy az adott alakzat bal felső sarka van közvetlenül a kurzoron.
- A „Back to editor” gomb lenyomásával a felhasználó visszakerül a szerkesztői ablakba, amivel együtt az addigi szimuláció is elveszik.

Végül mindezen funkciók leírása után következhet az alkalmazás használati eset diagramja:

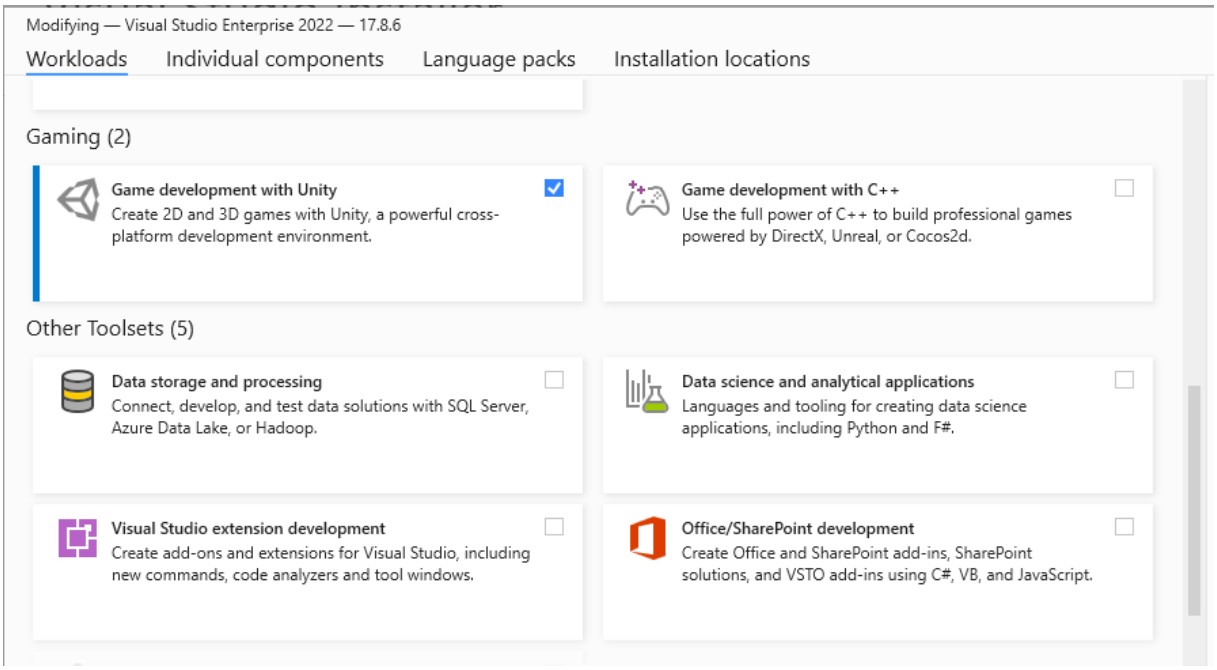


9. ábra: Használati eset diagram

Fejlesztői dokumentáció

A szoftver Microsoft Visual Studio 2022, Unity Hub és Unity 2022.3.10f1 programok segítségével készült. Ezen programok telepítése szükséges lesz a fejlesztéshez. Először telepítsük a Microsoft Visual Studio 2022-t az irodalomjegyzékben található link segítségével.

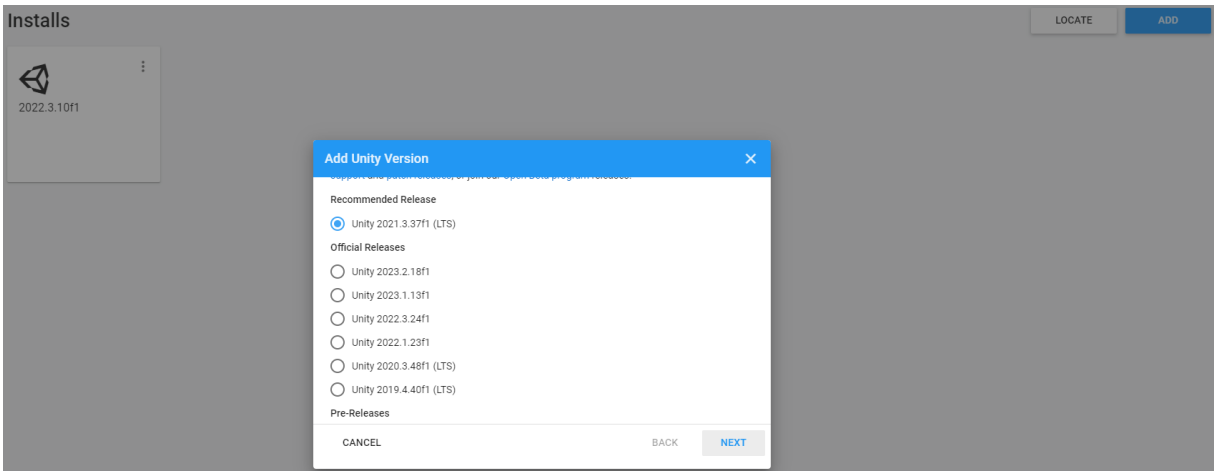
Telepítés után adjuk hozzá a „Game development with Unity” bővítményt a Visual Studio 2022-höz. Ez a bővítmény fogja lehetővé tenni a Unity-ben létrehozott C# scriptek fejlesztését.



10. ábra: Visual Studio Unity bővítmény

Ezzel a Visual Studio 2022 készen áll a fejlesztésre. A következő lépés a Unity verziók telepítése, és konfigurálása. Ehhez először a Unity hub telepítésére lesz szükség, amely a Visual Studio-hoz hasonlóan az irodalomjegyzékben megtalálható link segítségével megtehető.

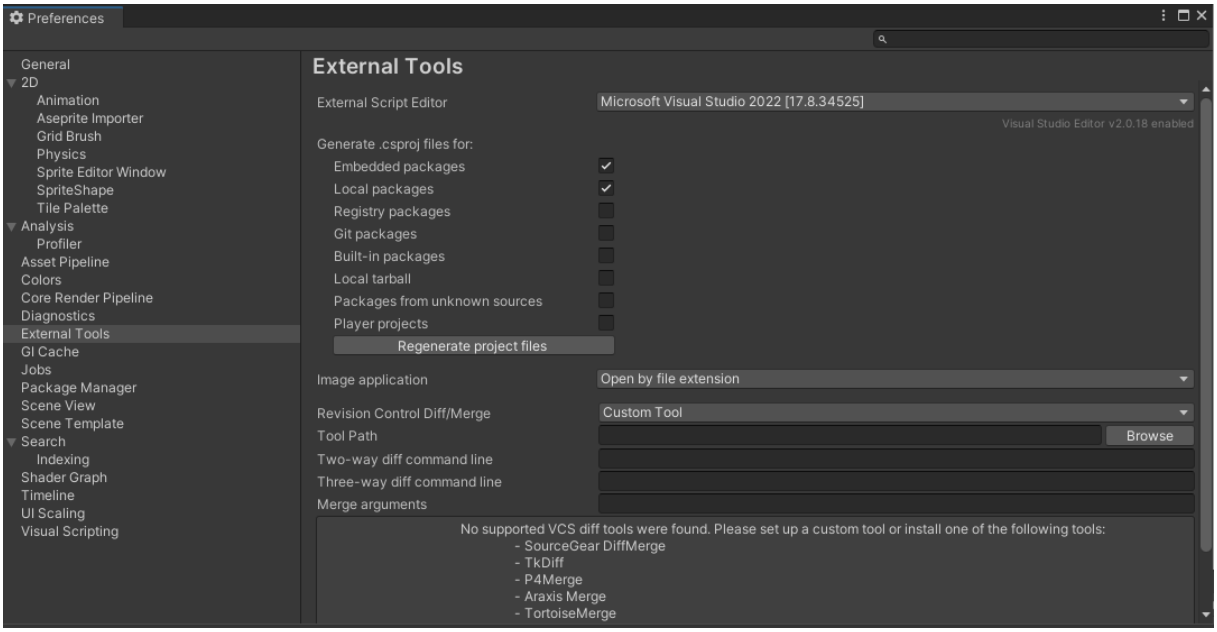
A Unity hub felületéről érhetőek majd el a Unity projektek és a verzió kezelést is itt lehet majd intézni. Telepítés után a Unity hub alkalmazáson belül az „Installs” menüpontra navigálva és ott az „Add” gombra kattintva kiválasztható a telepíteni kívánt Unity verzió. Ez jelen esetben a 2022.3.10f1 verziót fogja jelenteni.



11. ábra: Unity telepítés

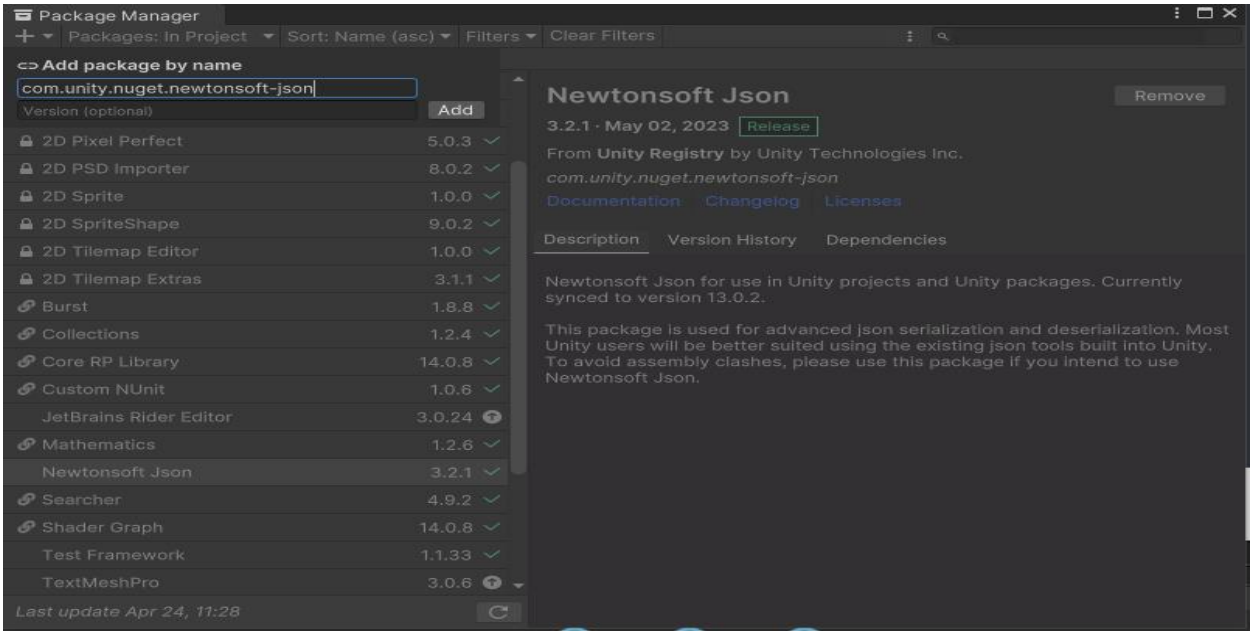
A megfelelő Unity verzió telepítése után pedig a Unity hub alkalmazásban a „Projects” menüpont alatt az „Add” gombra kattintva kiválasztható a szakdolgozat projektje. Ehhez a letöltött és kicsomagolt szakdolgozatból a „Fluid simulation” mappát kell kiválasztani.

Következő lépésben össze kell kötni a Unity-t a Visual Studio 2022-vel, és telepíteni kell a megfelelő Unity bővítményeket. Ehhez a Unity Hub alkalmazásban el kell indítani a projektet, és a felső sorban az „Edit” menüpont „Preferences” alpontját kiválasztani. Ezen belül pedig az „External tools” alpont „External Script Editor” listájából kell kiválasztani az előbbiekben telepített Visual Studio 2022-t.



12. ábra: Unity Visual Studio összekapcsolás

A bővítmények telepítéséhez a Unity alkalmazás felső sorában a „Window” menüpont „Package manager” alpontját kell kiválasztani. A package managerben pedig a bal felső plusz ikonra kattintva az „Add package by name” opciót választva és az alábbi linket bemásolva tudjuk telepíteni a szükséges bővítményt: com.unity.nuget.newtonsoft-json



13. ábra: Unity bővítmény

A bővítményre a json formátumú konfigurációs fájlok kezelése miatt van szükség. Ez részletesseben le lesz írva a későbbi pontokban. Ezzel a lépéssel pedig minden készen áll a szoftver fejlesztésére.

Az alkalmazás C# Unity keretrendszerben lett megvalósítva, amelynek több oka is van. Egyrészt a C# programozási nyelv a szemétgyűjtéses memória kezeléssel egyszerűbbé teszi az alkalmazásban használt bonyolult numerikus algoritmusok implementálását, mivel ennek köszönhetően nem kell a manuális memória kezeléssel foglalkozni. Másrészt pedig a Unity keretrendszerben egészen egyszerű módon lehet fejlesztői felületet létrehozni és szerkeszteni. Egyszerűen „drag and drop” módszerrel el lehet helyezni a felhasználói felületen az elemeket, majd a felülethez egy C# scriptet csatolva az elemek viselkedését is be lehet egyszerűen állítani. Mindezek segítettek abban, hogy a szimulációt minél pontosabban leíró numerikus differenciálegyenlet megoldó algoritmusok implementálására lehessen a fejlesztés során helyezni a hangsúlyt.

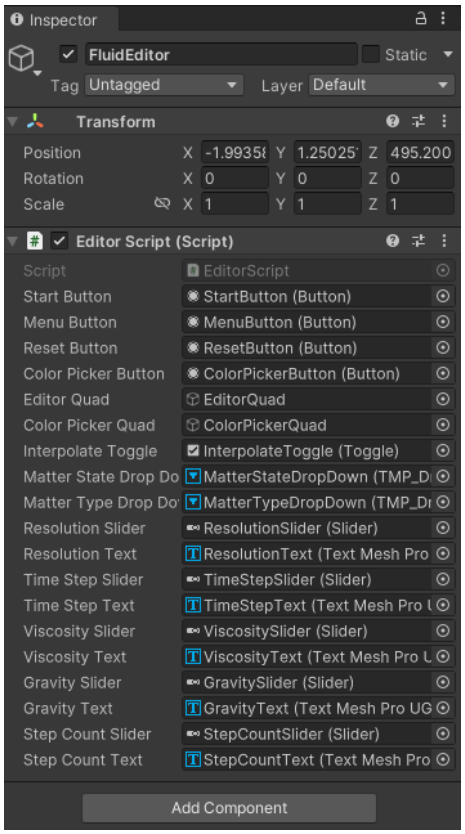
A fejlesztéssel szempontjából mindenképp kell egy pár szót ejteni magáról a Unity keretrendszerről, a keretrendszer működéséről, illetve a benne való fejlesztésről. A Unity keretrendszer alapvetően egy játékok fejlesztésére szánt keretrendszer, illetve játékmotor. Ettől eltekintve természetesen más egyéb alkalmazások fejlesztésére is alkalmas, ilyenek például a szimulációs alkalmazások.

A Unity alkalmazást megnyitva középen megjelenik a jelenet, illetve a játék nézet, amelyek között váltani lehet. A jelenet nézetben lehet összerakni magát a felhasználói felületet. Tehát például a felhasználói vezérlők, hátterek, primitívek elhelyezkedését és kinézetét lehet a jelenet nézetben beállítani. A beállítások elvégzése után pedig a játék nézetre átváltva lehet látni a kész felhasználói felületet.

A bal oldalon elhelyezkedő hierarchia ablakban találhatóak a jelenethez tartozó játékobjektumok. A játékobjektum a Unity keretrendszer alaposztálya, tehát minden a keretrendszerbe tartozó osztály ezen osztálynak a leszármazottja. A hierarchia ablakban egy játékobjektumra rákattintva a jobb oldalon található „inspector” ablakban megjelennek az adott játékobjektum tulajdonságai, illetve komponensei.

Az „inspector” ablak a fejlesztés szempontjából talán a legfontosabb, mivel itt lehet egy játékobjektumhoz logikát, illetve komponenseket kapcsolni. Egy játékobjektumnak több komponense lehet. Ezeket hozzá lehet adni manuálisan a jelenet nézetben, de akár az objektum script komponensén keresztül is. Ezen kívül a script komponensen keresztül lehet a felhasználói vezérlőket összekötni a nézeti réteggel, olyan módon, hogy a script mezőben megjelenő megfelelő változóra „drag and drop” módszerrel ráhúzzuk a megfelelő vezérlőt.

14. ábra: Unity hierarchia ablak



15. ábra: Unity inspector ablak

Az előbbihez hozzá tartozik még, hogy ahhoz, hogy egy adattag a script komponensnél megjelenjen a hierarchia ablakban, ahhoz magában a c# scriptben az adott adattagot „SerializeField” attribútummal kell ellátni. Ez azt fogja jelenteni, hogy az adott privát adattagot is szerializálni fogja a Unity, mivel alap esetben ez csak a publikus adattagokra vonatkozik.

A Unity C# scriptekhez tartozik még , hogy ahhoz, hogy egy scriptet hozzá lehessen egy játékobjektumhoz adni komponensként, ahhoz az adott scriptnek a „MonoBehaviour” osztály leszármazottjának kell lennie. Ez a szülő osztály biztosít olyan metódusokat, amelyek lehetővé teszik a fejlesztőnek, hogy képkockáról képkockára irányítsa a szoftver viselkedését. Az alkalmazásban leggyakrabban használt két ilyen metódus a „Start” és az „Update”.

A „Start” metódus a script elindítása után egyszer fut le, mégpedig az első képkocka előtt. Emiatt ez a metódus használható az osztály objektumainak a példányosítására, és a vezérlők kezdeti értékeinek a beállítására, mivel egyfajta konstruktorként funkcionál. Az „Update” metódus ezzel szemben minden egyes képkockában lefut, ezzel lehetővé téve a felhasználói interakciók detektálását. Mindezek után következhet a szoftver szerkezetének a leírása.

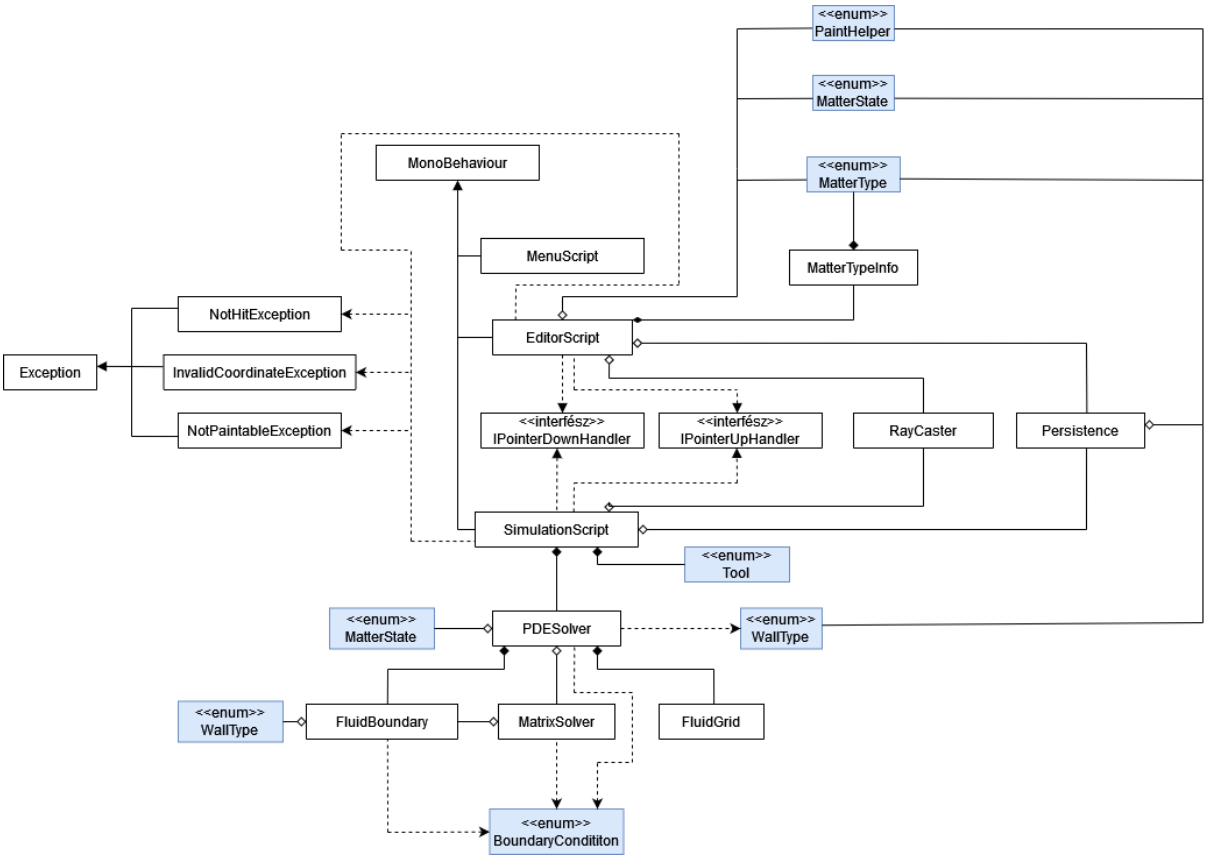
Az alkalmazás MV (model-view) architechtúrát valósít meg. Ennek értelmében a felhasználó a felhasználói felületen a vezérlőket módosítja, a nézeti réteg eseménykezelőkön keresztül érzékeli a változást, és ennek megfelelően végrehajtja a modellben a kellő módosításokat.

Az alkalmazás fejlesztéséhez a forrásfájlok letöltés, majd kicsomagolás után a „Fluid Simulation" mappában találhatóak meg. A „Fluid Simulation" mappán belül az „Assets" mappában a „Model" mappán belül találhatóak a modell réteg megvalósításához tartozó c# fájlok, a „View" mappán belül pedig a megjelenítési réteg megvalósításához tartozó c# fájlok.

Ezeken kívül az „Assets" mappán belül található még az összes a projekt szempontjából fontos mappa. A mappák tartalmának részletes tárgyalása a későbbiekben fog megtörténni.

- Az „Enums" mappában találhatóak a tervezés során használt enum adatszerkezetek egy c# fájlba összegyűjtve.
- A „JsonValues" mappában találhatóak a projekthez tartozó json (JavaScript Object Notation) formátumú fájlok, kezelését végző c# fájlok.
- A „Persistence" mappában találhatóak a perzisztencia réteg megvalósításához kapcsolódó c# fájlok.
- A „Resources" mappában találhatóak az alkalmazáshoz szükséges erőforrások. Ezek jelen esetben a json fájlok, a textúrák anyagai, illetve a hátterek lesznek.
- A „Scenes" mappában találhatóak az alkalmazásban használt jelenetek. Ezek Unity specifikus fájlok, amik az alkalmazás szempontjából egy az egyben megfeleltethetők a felhasználói dokumentációban részletezett három ablaknak. A megfelelő jelenetek ezen kívül kiválaszthatók a Unity alkalmazáson belülről is: a „File" menüpont „Open scene" alpontját megnyitva, és a fájlok közül kiválasztva a megfelelő jelenetet. Ezt követően a betöltött jelenet fog a Unity főfelületén megjelenni.
- A „Tests" mappában találhatóak az alkalmazás tesztelését végző tesztfájlok.

A mappaszerkezet leírása után következhet az alkalmazás osztálydiagramja:



16. ábra: Osztálydiagram

Az itt megjelenő osztályok részletezését jelenetenként fogom leírni az adott jelenet megfelelő funkciójához kapcsolva. Az osztályok részletezéséhez kerülnek majd beszúrásra a részletes osztálydiagramok is.

Az első jelenet legyen a szerkesztői jelenet, amely megjelenítéséért és logikájáért az „EditorScript” osztály felelős. Az osztály a „MonoBehaviour” osztály leszármazottja, illetve megvalósítja az „IPointerDownHandler”, és „IPointerUpHandler” interfészeket. A leszármazás korábban részletezve volt, azonban az interfészek megvalósítására szükség van, mivel így nyílik lehetőség arra, hogy a program érzékelje a folyamatos kattintást az interfészek eseménykezelőinek a megvalósításával. Az „IPointerDownHandler” interfész „OnPointerDown” metódusa érzékeli az egérgombok lenyomását, és az „IPointerUpHandler” interfész „OnPointerup” metódusa pedig az egérgombok felengedését.

Ezek után következzen a szerkesztői jelenet leírása. A szerkesztői jelenet egy „Canvas” objektumon jelenik meg. Ezen az objektumon található az összes többi felhasználói vezérlő, beleértve a gombokat, csúszkákat, legördülő listákat, és a hátteret.

17. ábra: EditorScript osztály

A felhasználói felületen található még ezen kívül két „Quad” objektum. A „Quad” a Unity primitívjei közé tartozik, és azért esett a fejlesztés során erre az objektumra a választás, mivel egyszerű a textúrázása, ami pedig elengedhetetlen a fal rajzolás és törlés miatt. A két „Quad” közül az egyik felel a szimulációs ablak megrajzolásáért, a másik pedig az anyag színének a kiválasztásáért.

A rajzolás és a színválasztás is egyaránt olyan módon van megoldva, hogy a „Quad” textúrájára sugarakat bocsájtunk ki a kamera pozíciójából a kurzor pozíciójába. Ezen funkciók megvalósításáért felelős a „RayCaster” osztály. Az osztály egyke tervezési mintával lett megvalósítva, mivel a program futása során felesleges lenne egynél több példányt eltárolni az osztályból. Ezen kívül az osztály a szerkesztői jeleneten kívül a szimulációs jelenetben is használva van, de mivel a felhasználás alig tér el az itt tárgyaltaktól, ezért a funkciók csak itt lesznek kifejtve.

RayCaster
- _instance: RayCaster
<<property>> + Instance: Raycaster <<metódus>> - RayCaster() - ValidCoordinate(pixelCoordinate: (int,int), gridSize: int): bool - ValidCoordinate(pixelCoordinate: (int,int), gridSize: int, wallTypes: WallTypes[,]): bool - Paintable(pixelCoordinate: (int,int), gridSize: int, paintHelper: PaintHelper[,], leftDown: bool): bool + CalculatePixelCoordinates(quad: GameObject, texture: Texture2D): (int,int) + CalculatePixelCoordinates(quad: GameObject, texture: Texture2D, wallTypes: WallType[,]): (int,int) + CalculatePixelCoordinates(quad: GameObject, texture: Texture2D, paintHelper: PaintHelper[,], leftDown: bool)

18. ábra: RayCaster osztály

Az előbb tárgyalt sugárkibocsátás módszere teszi lehetővé, hogy érzékeljük azt, hogy az adott „Quad” textúrájának melyik pixelére mutat a felhasználó a kurzorral. Ezen túl mindkét „Quad” textúrája egy-egy „Texture2D” textúra objektumban van eltárolva.

Ez a formátum pixeltömbként tárolja le a textúrát, lehetővé téve a textúra méretének, illetve megfelelő pixeleinek a futási időben történő megváltoztatását. Erre pedig szükség is van, mivel a sugárkibocsátás által meghatározott pixelt fogjuk a programban manipulálni. Fal rajzolás esetében az adott pixelek színét fehérről feketére, fal törlés esetében pedig feketéről fehérre állítjuk be. Színválasztás esetében pedig az adott pixel színét mintavételezzük, és az anyag színét, illetve a „Color” gomb színét is a megfelelő színre állítjuk be.

A sugárkibocsátáshoz és a textúra manipulálásához hozzá tartozik még, hogy a kivétel kezelés saját kivétel osztályok megvalósításával történik meg. Az említett osztályok az „Exception” fájlban találhatóak meg:

- A „NotHitException” kivétel akkor kerül kiváltásra, ha nem volt sikeres a sugárkibocsátás, tehát ha a felhasználó a kurzorával nem valamelyik „Quad”-ra kattint.
- Az „InvalidCoordinateException” kivétel abban az esetben kerül kiváltásra, ha a felhasználó ugyan a „Quad”-ra kattintott, de mégsem a megfelelő pixelekre kattintott. Ezek lehetnek azok az esetek amikor a felhasználó a felület keretére kattint.
- A „NotPaintableException” kivétel akkor kerül kiváltásra, ha az adott pixelt nem lehet falnak festeni. Ez abban az esetben lehetséges, ha az adott pixel már eleve fekete, vagy a felhasználói dokumentációban már részletezett lerakási feltételek valamelyike nem teljesül.

Az előbbi kivételek kiváltási feltételei csak az adott jelenethez (jelen esetben a szerkesztői) tartoznak. A további felhasználásuk megfelelő jeleneteknél lesz részletezve.

A textúrákhoz hozzá tartozik még, hogy az előbb említett két „Quad” olyan módon helyezkedik el, hogy a szimulációs felületért felelős alapvetően eltakarja a színválasztásért felelőst. Ez a kölcsönös takarás úgy lett megvalósítva, hogy alap esetben a szimulációs felület van előrébb, azonban a színválasztó gomb lenyomásával megcserélődik a két „Quad” render sorrendje, és a „z” koordinátája szerint előrébb lesz transzformálva a színválasztó „Quad”, ezzel előtérbe hozva azt. Ezek után a színválasztó gomb ismételt lenyomásával visszacserélődik a render sorrend és a transzformáció is semmissé válik, ezzel az eredeti takarást előteremtve.

A szimulációs felületen való fal rajzolási feltételek ellenőrzése miatt szükséges volt egy rajzolást segítő „PaintHelper” típusú enum tömb bevezetése. Ez olyan módon valósul meg, hogy a lerakott fekete pixelek állásának megfelelő „PaintHelper” enum értékei adják az enum tömb megfelelő elemeinek az értékét.

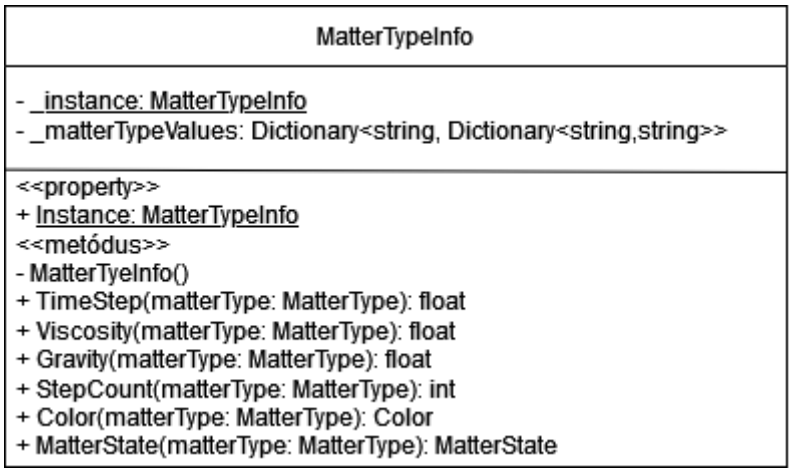
Ez azt jelenti, hogy a kurzor helyén létrejövő fekete pixel lesz a „TOPLEFT”, a tőle jobbra lévő a „TOPRIGHT”, az alatta lévő a „BOTTOMLEFT”, a jobbra átlósan alatta lévő a „BOTTOMRIGHT”, a keret pixelek a „BOUNDARY”, a fehér pixelek pedig a „NONE” értékek. Így a pixelek besorolása után a lerakási feltételek ellenőrzése leegyszerűsödik az enum tömb megfelelő értékeinek az ellenőrzésére. A rajzolási, illetve törlési feltételek a következőek lesznek:

- Rajzolni csak és kizárólag olyan pixelre lehet, amelynek a lerakást megelőzően mind a három szomszédos pixele, illetve önmaga is „NONE” értékű. Ez pontosan azt jelenti, hogy a lerakni kívánt 2x2-es négyzet minden pixele „NONE”.
- Rajzoláshoz figyelembe kell venni a lerakni kívánt 2x2-es pixel négyzetet körülvevő 4x4-es pixel négyzetet. Ez olyan módon történik, hogy ha a belső négyzet egy adott pixelének két külső négyzethez tartozó élszomszédos pixele „NONE” értékű, akkor a lerakás csak abban az esetben érvényes, ha az adott pixel külső négyzethez tartozó sarokszomszédos pixele is „NONE” értékű. Az előző eset érvényes marad abban az esetben is, ha a két élszomszédos cella nem „NONE” értékű, és ekkor a sarokszomszédos cella értéke nem releváns.
- Törlés csak és kizárólag olyan pixelre vonatkozhat, amelynek az értéke „TOPLEFT”. Ez azt jelenti, hogy csak egybefüggő pixel négyzetek törlése lehetséges.

A szerkesztő felületen található két legördülő lista enumokkal lett megvalósítva. Ez azt jelenti, hogy a legördülő listák az értékeiket a megfelelő enumok értékeiből kapják. A szerkesztő felülethez tartozó enumok az „Enum” fájlban találhatók, amik a következők: „MatterType” és „MatterState”. Emiatt fejlesztés szempontjából elég csak a megfelelő enumokat bővíteni, a legördülő listákon változtatásokat végezni nem szükséges.

Fontos még megjegyezni, hogy a „MatterType” legördülő listából egy elemet kiválasztva a paraméterek automatikusan megváltoznak a megfelelő konfigurációra. Ezen konfigurációk beállításai megtalálhatók a MatterTypeValues.json konfigurációs fájlban. A json formátum használata megkönnyíti a fejlesztés menetét, mivel az adott anyag paramétereinek a megváltoztatásához elég csak a json fájl egy-egy sorát megváltoztatni.

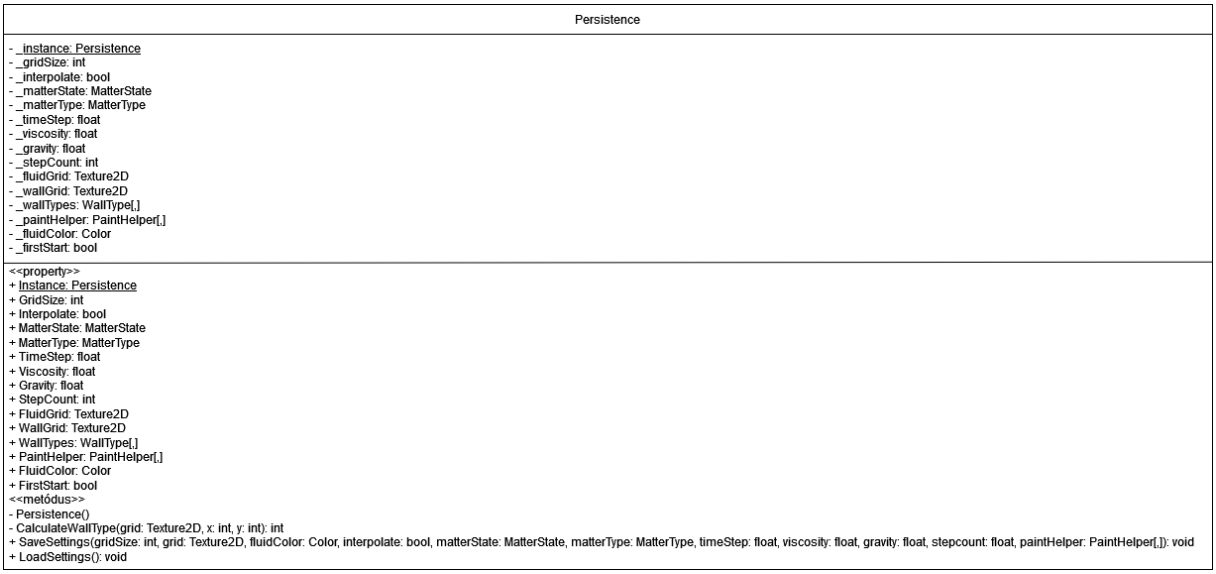
Ezen json fájl szerializációját végzi el a „MatterTypeInfo” osztály. Az osztály egyke tervezési mintát használva lett megvalósítva, mivel a program futása során felesleges lenne egynél több példányt eltárolni az osztályból.



19. ábra: MatterTypeInfo osztály

Az objektum szerializációt a „Newtonsoft” csomag teszi lehetővé, amelynek a megfelelő telepítése a fejlesztői dokumentáció elején részletezve volt.

A következő fontos funkció a szerkesztő felületen a „Start Simulation” gomb megnyomását követő lépések. Ekkor ugyanis a szerkesztőhöz kapcsolt „EditorScript” elmenti a szerkesztő felület adatait a perzisztencia réteg segítségével. A perzisztencia réteget megvalósító osztály a „Persistence”.



20. ábra: Persistence osztály

Az adatok elmentése olyan módon valósul meg, hogy a „Persistence” osztály mentés metódusa átveszi a nézeti rétegtől a szimuláció szempontjából fontos adattagokat, ezt követően pedig a „Persistence” osztály egy szövegfájlba menti az adatokat. Az elmentett adatokat tartalmazó

szövegfájl a gyökér mappában található settings.txt néven. A fájl a következő adatokat tartalmazza:

- felbontás, anyag színe, interpolálás megléte, halmazállapot, anyag típus, időlépték, viszkozitás, gravitáció, iterációs lépésszám, szimulációs felület textúrája, rajzolás segédtömb
- Ezen adatok közül az egész, illetve valós típusúak egy az egyben egy-egy sorát alkotják a fájlnak.
- Az enum típusúak a nekik megfelelő egész számra konvertálást követően alkotnak egy sort.
- A szín típusúak az RGB színcsatornáiknak megfelelően alkotnak három-három sort.
- A textúra, illetve a tömb típusúak balról jobbra, és fentről lefelé haladva alkotnak a méretükkel megegyező számú sort, illetve oszlopot a fájlban. Ehhez fontos megjegyezni, hogy alapvetően a tömbök és textúrák indexelése is balról jobbra, és lentről felfelé történik. Tehát az indexelés konzisztenciájának fenntartása érdekében szükséges a két koordináta rendszer közti konverzió. Ez úgy lehetséges, hogy az írás és olvasás során is megcseréljük az x és y koordinátákat, illetve az x koordinátát a ciklus során a nagyobb értéktől csökkentjük a kisebbig.

Ezen funkciókon kívül a „Persistence” osztály feladata még, hogy a szerkesztői felület textúrájára rajzolt falakról meghatározza, hogy azok a peremfeltelek szempontjából milyen kategóriába tartoznak. Ez olyan módon fog megtörténni, hogy a szerkesztői felület átadja a „Persistence” osztálynak az adott texúrát, majd a perzisztencia ezen textúrából generál egy „WallType” enum típusú kétdimenziós tömböt.

Ezen tömb fogja a faltípusokat eltárolni, annak érdekében, hogy a szimulációért felelős osztályok ehhez majd hozzá tudjanak férni. A falak kategóriába sorolása úgy történik meg, hogy minden pixelre megvizsgáljuk egyrészt azt, hogy az adott pixelnek milyen a színe. A fehér színű pixelek fognak a „NONE” típusnak megfelelni, a fekete pixeleknek pedig a négy lapszomszéd pixelének a színe fogja a típusát meghatározni. Tehát például „TOPLEFT” típusú lesz az a pixel, amelynek felső és bal szomszédja is fehér színű, a többi pedig fekete, és „BOTTOMLEFT” típusú lesz az a pixel, amelynek az alsó és bal szomszédja fehér színű, a többi pedig fekete.

Megjegyezhető, hogy a típusok meghatározásánál már nincsen szükség hibakezelésre, mivel a szerkesztői felületen a színezési, illetve törlési feltételek ellenőrzése miatt, csak és kizárólag érvényes peremnek számító fekete pixeleket tartalmaz a textúra.

Az előbb leírt „Persistence” osztályra, és fájlba írásra azért van szükség, mivel a Unity nem tárol el adatokat az adott jelenetek között. Tehát ha egy adott jelenetről átváltunk egy másikra akkor a jelenet során példányosított objektumok felszabadulnak és a jelenet legközelebbi betöltésekor újra példányosulnak. Ezért szükséges a „Persistence” osztály a jelenetek közötti adatátvitel megvalósításához.

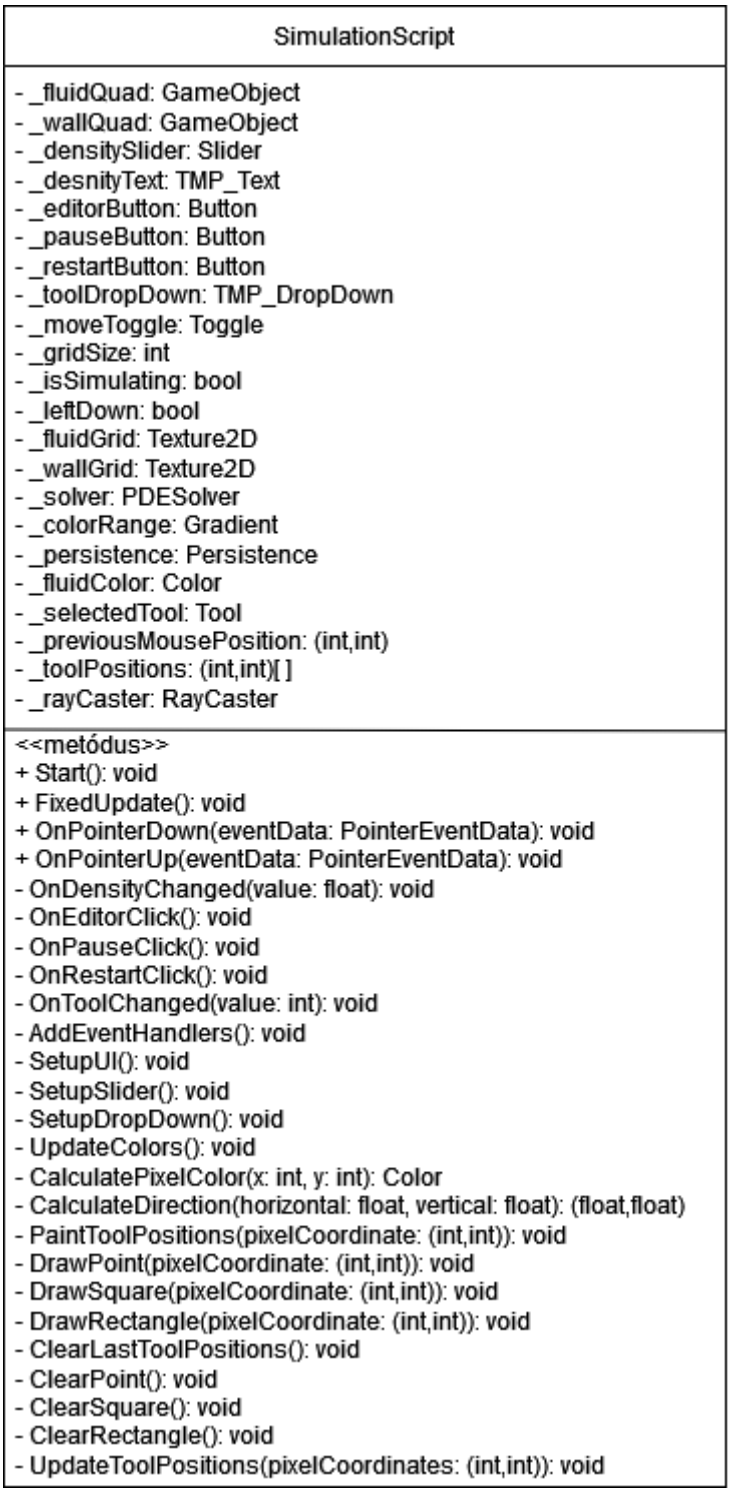
Ehhez tartozik még, hogy a „Persistence” osztály egyke tervezési mintával lett megvalósítva, mivel a program futása során felesleges lenne egynél több példányt tárolni az osztályból.

A szerkesztői felület következő eleme az interpolációs kapcsoló. Ez a vezérlő határozza meg a szimulációs felület textúrájának az interpolációját. Ez azt jelenti, hogy abban az esetben, ha a kapcsoló bekapcsolt állapotban van a textúra bilineáris interpolációval lesz kirajzolva, kikapcsolt állapotban pedig nem lesz interpoláció alkalmazva a textúrára. Az interpoláció nélküli textúra ideális lehet adott paraméter beállítások tesztelésére, a bilineáris interpoláció használata pedig a szebb megjelenítést segíti elő.

A szerkesztői felület „Reset” gombja újra pédányosítja és alap állapotba állítja a szerkesztői jelenet minden objektumát beleértve a textúrákat, és a felhasználói vezérlőket.

A szerkesztői felületre vonatkozó utolsó fontos dolog a szimulációs felület textúrájának a felbontását szabályozó csúszka működése. A csúszka állításával megváltozik a textúra mérete, ezáltal pedig megváltozik a textúra pixeleinek a mérete is. Fontos megjegyezni, hogy minden egyes változtatásnál új értéket kap az egész textúra, tehát a változtatás pillanatáig a felületre rajzolt falak elvesznek. Ezzel pedig végére értünk a szerkesztői jelenet tárgyalásának.

A következő jelenet legyen a szimulációs jelenet. A jelenet megjelenéséért és logikájáért a „SimulationScript” osztály felelős. Az osztály az „EditorScript” osztályhoz hasonlóan a „MonoBehaviour” osztály leszármazottja, és megvalósítja az „IPointerDownHandler”, és „IPointerUpHandler” interfészeket. Mind a leszármazás, mind a megvalósítások egy az egyben megegyeznek az „EditorScript” osztálynál leírtakkal, ezért ezeket tovább nem részletezném ebben a pontban.

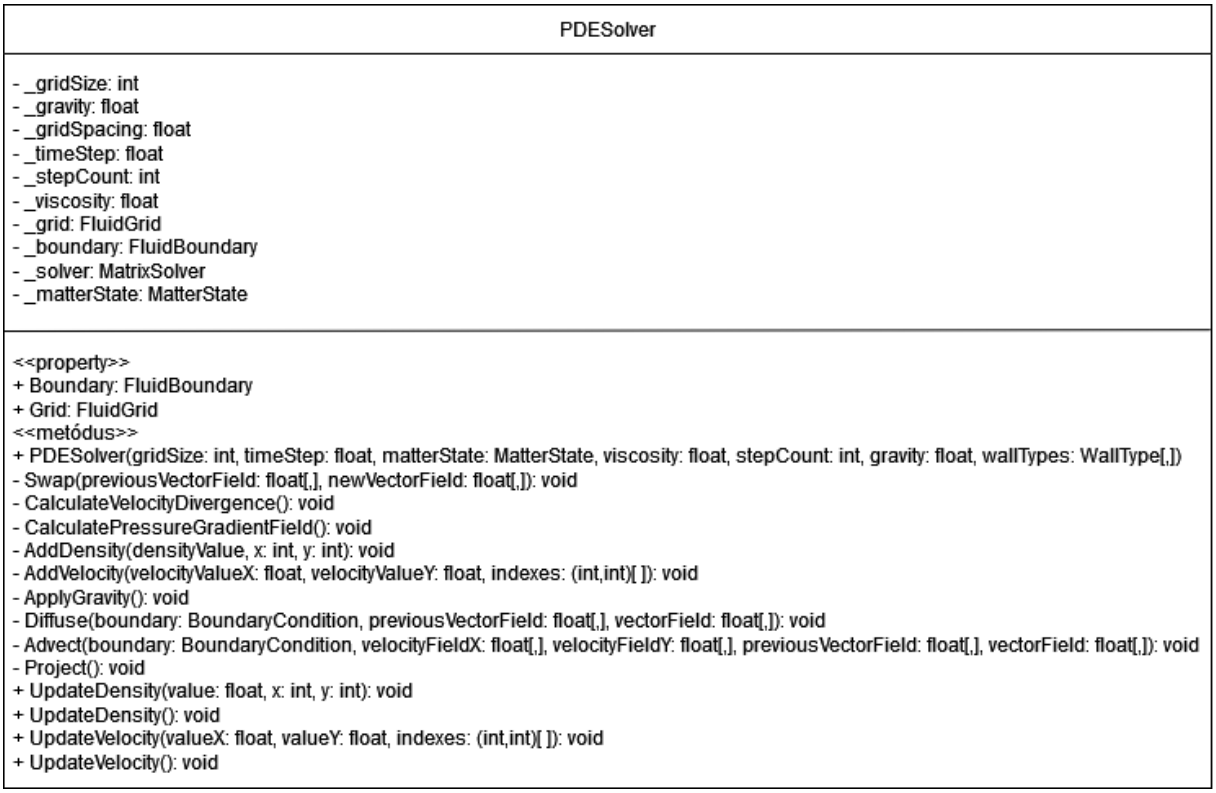


21. ábra: SimulationScript osztály

A szimulációs jelenet a szerkesztői jelenethez hasonlóan egy „Canvas” objektumon jelenik meg. Ezen az objektumon találhatóak a felhasználói vezérlők, a textúrák, illetve a háttér is. A jelenet betöltése rögtön azzal kezdődik, hogy a szimulációs jelenet a szerkesztői jelenttől a perzisztencia réteg betöltés metódusán keresztül átveszi a szimulációhoz szükséges adatokat.

Ezek után példányosulnak a megkapott adatok segítségével a szimuláció megjelenítéséért és a szimuláció végrehajtásáért felelős objektumok.

A szimuláció menetéért felelős osztály a „PDESolver” osztály.



22. ábra: PDESolver osztály

A „SimulacionScript” osztály ettől az osztálytól kapja meg a modell réteg algoritmusai által előállított nyers adatokat. Ezen adatoknak egy az egyben textúra pixelekké való konvertálása nem lehetséges, mivel a modell skalár sűrűség értékeket ad vissza, a pixelek színe viszont egy háromdimenziós RGB értékekből álló vektor kell legyen. Ennek érdekében meg kell határoznunk egy megfelelő konverziót a két érték között.

Ez a konverzió a Unity beépített „Gradient” osztályának a segítségével fog megtörténni. A szimuláció kezdetekor eltárolódik egy minimum, illetve egy maximum érték is, amelyek a sűrűség értékek szélsőértékeire fognak vonatkozni. Ezek után a „Gradient” objektum két megadott színérték szerint fogja minden egyes pixel színét kiszámolni olyan módon, hogy a pixel sűrűség értéke alapján interpolálja a pixel színét a megadott két érték közé. Ezen a ponton megjegyezhető, hogy az alsó értékhez tartozó szín minden esetben a fehér lesz, a felső értékhez tartozó szín pedig a szerkesztői felületen beállított érték, és a nagyobb sűrűség

értékekhez nagyobb intenzitású színek fognak tartozni. Ezeket a színértékeket minden egyes képkockában újra kell számítani a modell változásai miatt.

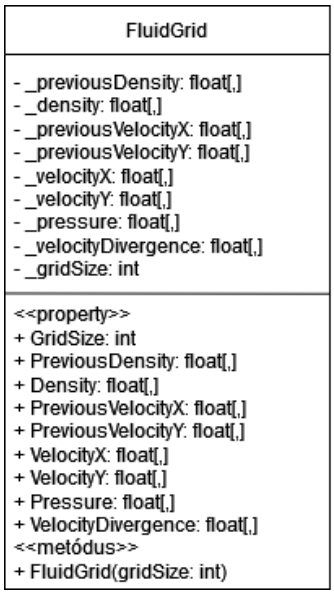
Mindezek mellet a „PDESolver” osztály valósítja meg a matematikai bevezetésben részletezett parciális differenciálegyenletrendszer megoldását, ami a szimuláció lelkét adja. A szimuláció a gyakorlatban úgy valósul meg, hogy a „SimulationScript” osztály a „FixedUpdate” metódusában meghívja a „PDESolver” osztály „UpdateVelocity”, illetve „UpdateDensity” metódusait a megfelelő paraméterezéssel.

Ezek a metódusok felelnek meg a differenciálegyenlet megoldó algoritmus egy-egy lépésének. Fontos, hogy a sebességmező frissítése megelőzze a sűrűség mező frissítését, mivel a szimuláció pontossága csak ebben az esetben garantált. Megjegyezhető még ezen kívül, hogy a „SimulationScript” osztályban a „MonoBehaviour” osztály „FixedUpdate” metódusa lett használva az „Update” metódus helyett, mivel ez a metódus képkockánként egy előre meghatározott fix értékszer hívódik meg, ezzel garantálva a szimuláció konzisztenciáját.

A megoldó lépésekhez hozzá tartozik még, hogy a tömbök megfelelő frissítése érdekében, minden korábban leírt tömbnek el van még tárolva az előző időpillanatban felvett értéke is. Erre azért van szükség, mivel a számítások során sok esetben az előző értékből számítódik ki a következő érték.

Mindezek miatt egy megoldó lépés utolsó lépése az kell, hogy legyen, hogy a korábbi értékek megcserélődnek az aktuális értékekkel. Ez olyan módon történik meg, hogy a tömbelemeket referenciaként veszi át egy „Swap” függvény, és megcseréli a két értéket.

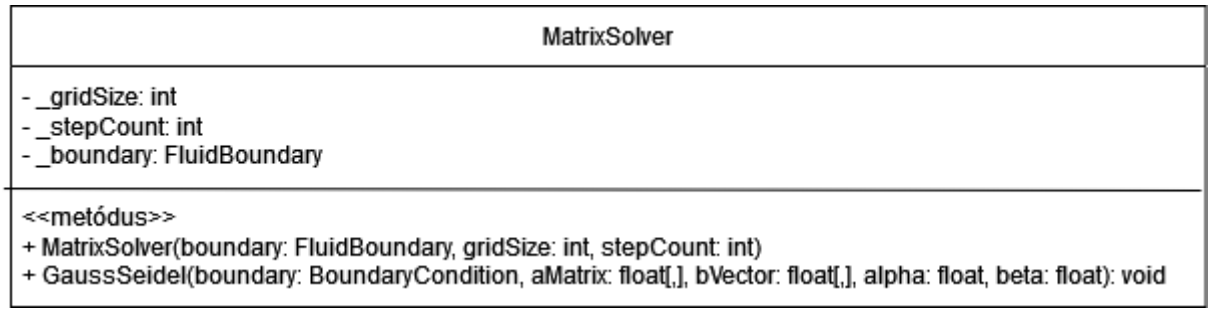
A „PDESolver” osztály a szimulációs lépések kiszámításán kívül eltárol még minden, a szimuláció szempontjából fontos adatot. Ezek az adatok nagyrészt külön osztályokban vannak eltárolva. Az egyik ilyen osztály a „FluidGrid” osztály, amely eltárolja a szimulációs tér diszkrét felosztását kétdimenziós tömbök formájában.



23. ábra: FluidGrid osztály

Minden tömb egy-egy értéket tárol el, például a sűrűséget, a sebességet vagy a nyomást. Fejlesztés szempontjából, ha további mezők hozzáadására lenne szükség, akkor ezen az osztályon kellene a megfelelő változtatásokat elvégezni. Tehát ha a szimulációt bővíteni szeretnénk például hőmérséklet értékekkel, akkor azt ebbe az osztályba kéne megtenni.

A következő segédosztály a „MatrixSolver” osztály, amely az iterációs algoritmusokért felelős.

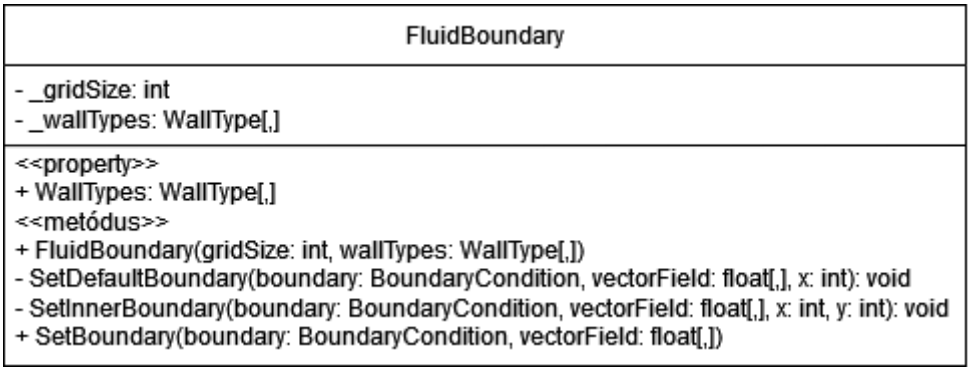


24. ábra: MatrixSolver osztály

A matematikai bevezetésben részletezett Poisson egyenlet megoldó Gauss-Seidel iteráció ebben az osztályban lett implementálva. Fejlesztés szempontjából ez azt jelenti, hogy az további iterációs módszerek implementációi ezen osztályba kell, hogy kerüljenek. Ez lehet például egy párhuzamosított Jakobi iteráció, vagy akár egy konjugált gradiens módszer is. Ezen a ponton megjegyezhető, hogy azért esett a Gauss-Seidel iterációra, mint iterációs egyenlet megoldó algoritmusra a választás, mivel viszonylag egyszerű az implementációja.

Ezen kívül az alkalmazás CPU-ra való implementációja miatt gyorsabb, mint például a Jakobi iteráció, illetve kevesebb memóriát használ, mint például a konjugált gradiens módszer. Ugyanakkor az alkalmazás esetleges GPU implementációra való átírása esetén a párhuzamosíthatóság miatt a Jakobi iteráció lenne az ideális megoldás.

Mindezek után következzen a „PDESolver” osztály utolsó segédosztálya, amely a „FluidBoundary”.



25. ábra: FluidBoundary osztály

Ezen osztály felelős a peremértékek számolására implementált algoritmusokért. Két fő metódusa van, ezek közül az egyik számolja a külső falakra vonatkozó peremfeltételeket, a másik pedig a belső falakra vonatkozókat.

A külső peremfeltételek számítása viszonylag egyszerű, mivel minden olyan tömbelemre kell ezt kiszámolni, amelynek az egyik indexe vagy nulla, vagy a tartomány hossza plusz egy. A belső peremfeltételek számításához azonban szükség van egy „WallType” enum típusú segédtömbre, amely eltárolja a belső falaknak egyrészt a pozícióját, másrészt pedig a fajtáját.

A belső fal fajtájának és pozíciójának a kiszámítása a „Persistence” osztályban történik meg, korábban részletezett módon. Ezek után pedig végig kell iterálnunk a segédtömbön és abban az esetben, ha a tömbelem nem „NONE” értékű, akkor ki kell a peremértéket számolnunk. A peremérték számítások a „PDESolver”, illetve a „MatrixSolver” osztály metódusaiban is felhasználásra kerülnek.

A szimulációs felületen található „Move” kapcsoló állása határozza meg, hogy a korábban említett mező frissítésért felelős függvények milyen paraméterezéssel hívódnak meg.

Abban az esetben, ha a kapcsoló kikapcsolt állapotban van, akkor a sűrűség frissítő függvény a sűrűségre vonatkozó csúszka értékével, mint paraméterrel hívódik meg. Ez azt fogja

eredményezni a szimuláció szempontjából, hogy a sűrűségeket tároló tömb megfelelő elemét megnöveljük a paraméterként kapott értékkel.

Jelen esetben a tömb megfelelő eleme a tömb azon indexen lévő elemét fogja jelenteni, amelynek megfelelő textúra pixelére kattint a felhasználó. Megjegyezhető még, hogy a tömbök és a hozzá tartozó textúrák indexelése megegyezik.

A kikapcsolt esetben ezen kívül a sebesség frissítő függvény paraméterek nélkül hívódik meg. Mindezek összefoglalva azt jelentik, hogy kikapcsolt állapotban lévő kapcsoló mellett és kattintást érzékelve a sűrűségeket tartalmazó tömb először megváltozik, majd frissül, a sebességeket tartalmazó tömb pedig csak frissül.

Ehhez képest abban az esetben, ha a kapcsoló bekapcsolt állapotban van, akkor a sebességet frissítő függvény egy előre meghatározott értékkel, mint paraméterrel hívódik meg. Ez azt fogja eredményezni a szimuláció szempontjából, hogy a sebességeket tároló tömb megfelelő elemét megnöveljük, vagy lecsökkentjük az adott értékkel.

Mivel a sebesség tömbünk a kétdimenziós szimuláció miatt két komponensből áll, ezért meg kell tudnunk határozni azt, hogy az adott tömb elemének „x” vagy „y” komponensét változtatjuk meg, illetve azt is el kell döntenünk, hogy növelést, vagy csökkentést hajtunk-e végre. Ennek meghatározásához a kurzor „x”, illetve „y” tengely menti elmozdulását kell figyelembe vennünk.

Ez olyan módon fog megtörténni, hogy ha a kurzor elmozdulása az x tengely irányában pozitív, akkor a tömbelem „x” komponenséhez egy pozitív értéket adunk hozzá, ha pedig negatív, akkor egy negatív értéket adunk hozzá. Az „y” tengely menti elmozdulásnál ugyanezen módon járunk el.

Mindezek összefoglalva azt jelentik, hogy bekapcsolt állapotban lévő kapcsoló mellett, és kattintást érzékelve a sűrűségeket tartalmazó tömb csak frissül, a sebességeket tartalmazó tömb pedig először megváltozik, majd frissül.

A bekapcsolt állapotú kapcsolóhoz hozzá tartozik még az, hogy a kurzor, illetve a kiválasztott eszköz alakja piros pixelekkel legyen jelölve a szimulációs felületen. Ezen funkció megvalósításához szükség van egy tömbre, amelyben el lesznek tárolva az eszköz jelenlegi

koordinátái. Ennek a tömbnek a mérete megváltozik majd abban az esetben, ha a felhasználó másik ezközt választ majd ki az eszközválasztó legördülő listából.

Erre azért van szükség, mivel minden eszköz eltérő számú pixelből áll. Ezek után a tömb elemei olyan módon számítódnak ki, hogy a korábban ismertetett sugárkibocsátás használatával kiszámoljuk a kurzor pozícióját, majd ennek függvényében kiszámoljuk az adott eszköz összes többi koordinátáját is. Tehát pont esetében csak a kurzor pozíciója számít, azonban négyzet esetén a kurzor lesz a négyzet bal felső sarka a többi három pixel pedig innentől egyértelmű.

Ezen kívül a folyamatos kurzor követés miatt szükséges eltárolni a kurzor megelőző pozícióját is. Erre azért van szükség, mivel nem elég az alakzat kirajzolása minden egyes képkockában, hanem az előző kirajzolt piros alakzat eredeti színre visszarajzolása is fontos. Tehát ezek után az a funkció, hogy a felhasználó egy kiválasztott eszközzel a kurzorát a szimulációs felületre viszi úgy néz ki, hogy először megtörténik a kurzor jelenlegi pozíciója és az eszköz pozíciók segítségével az eszköz helyének a kirajzolása, majd ezt követően az előző kurzorpozíció segítségével törlésre kerül az eszköz előző pozíciója.

Fontos megjegyezni, hogy jelen esetben a pixelek pirosra színezése nem változtat a modell állapotán. Ez a tény fogja ugyanis lehetővé tenni, hogy törlés után a modell alapján újra meghatározzuk az adott pixel színét. Hozzá tartozik még a kurzor követéshez, hogy a hibakezelés a már korábban említett saját kivétel kiváltásával történik meg.

Jelen esetben „NotHitException”-t a sugárkibocsátás sikertelensége esetén vált ki a program, „InvalidCoordinateException”-t abban az esetben vált ki ha a szerkesztői jelenetben felrajzolt falak valamelyikére kattint a felhasználó, „NotPaintableException”-t pedig akkor, ha a kirajzolni próbált eszköz bármely pixele a külső keretre, vagy azon kívülre esne.

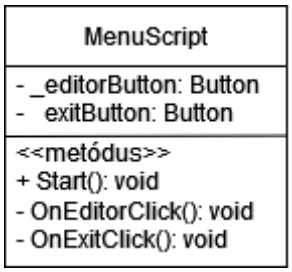
Az eszköz kirajzolási funkciók leírása után következzen a szimulációs felület megjelenítésének a leírása. A megjelenítés olyan módon lett megvalósítva, hogy a „Canvas” objektumon két darab „Quad” helyezkedik el egymás előtt, teljes fedésben. A két „Quad”-ra azért van szükség, mivel az egyik textúrája fogja megjeleníteni azt a teret, ahol az anyag a szimuláció során ténylegesen áramolhat, a másik pedig a felhasználó által a szerkesztői jelenetben megrajzolt falak megjelenítéséért felel.

Ezen két textúra megjelenítése pedig nem minden esetben fog megegyezni, ugyanis a falak megjelenítése minden esetben interpoláció nélkül fog megtörténni, mivel így garantálható az,

hogy a falaknak sima marad a széle, nem pedig elmosódik. A falak nélküli szimulációs tér megjelenítése pedig a felhasználó döntésén múlik, mivel ő dönthet az interpoláció használatáról. Ebből következik, hogy abban az esetben, ha a felhasználó bilineáris interpolációt választ ki a szimulációhoz akkor, ha egy textúraként kezelnénk az egész szimulációs teret, akkor nem lehetne megvalósítani azt, hogy a falakon ne legyen interpoláció.

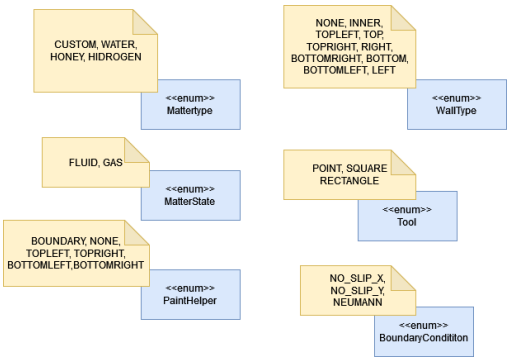
Ennek érdekében történik meg a textúra két textúrává való szétválasztása. Ezek után a falakat tartalmazó textúra fog az előtérbe kerülni, és ezen textúra fekete pixeleinek, illetve a másik textúra nem fekete pixeleinek az átlátszósága teljesen átlátszóra lesz állítva, így előteremtve a kívánt megjelenítést. A textúrák tárgyalásával pedig végére értünk a szimulációs rész tárgyalásának.

A program utolsó jelenete, ami kimaradt a leírásból az a menü jelenet. A jelenetért a „MenuScript” osztály felel. Az osztály tartalmazza a jelenet két gombjának a működési logikáját.



26. ábra: MenuScript osztály

Mindezek után az egyetlen dolog, amely az alkalmazás leírásából kimaradt, az szoftver fejlesztésében használt enum osztályok leírása. Ezek a program során a fontosabb osztályok nagyrészében megjelennek, illetve ezek használata a korábbiakban már leírásra került, ezért az enumokat tartalmazó diagram most kerül beszúrásra.



27. ábra: Enum diagramok

Tesztelés

Az alkalmazás működése szempontjából elengedhetetlen, hogy a szerkesztői, illetve a szimulációs jelenetben beállítható felhasználói vezérlők bármilyen kombinációjára a szimuláció úgy történjen meg, hogy az megjelenítésben minden szempontból tükrözze a valóságot. Éppen ezért szükséges lesz a differenciálegyenlet megoldást végző osztály, azaz a „PDESolver” osztály tesztelése minden lehetséges paraméter kombinációra.

Jelen esetben a minden lehetséges paraméter kombináció azt fogja jelenteni, hogy a felhasználó vezérlők két szélsőértékére lesznek tesztesetek. Tehát minden egyes csúszka legkisebb, és legnagyobb értéke, illetve a halmazállapot legördülő lista két értéke fogják a lehetséges paraméterértékek halmazát alkotni. Ezen a ponton megjegyezhető, hogy feltételezhető az algoritmus stabilitása miatt, hogy ha a differenciálegyenlet megoldó algoritmus helyesen működik a felhasználói vezérlők szélsőértékeire, akkor helyesen fog működni a köztes értékekre is.

A teszteléshez használt keretrendszer a „Unity Test Framework”. Ez a keretrendszer hozzá tartozik a „Unity” keretrendszerhez, tehát manuális telepítésére nincs szükség. A keretrendszer alapvetően háromfajta teszt futtatását támogatja: szerkesztői mód, játékmód, játékos. A játékmód és játékos tesztek lehetőséget nyújtanak arra, hogy a fejlesztő létrehozzon saját teszt jeleneteket, majd a jelenetet futtatva tesztelje az alkalmazás megjelenését. Ezen tesztfajták nem lettek használva az alkalmazás fejlesztése során. Ezzel szemben a szerkesztői mód tesztek lehetőséget adnak arra, hogy a jelenetekhez írt c# scripteket tesztelje a fejlesztő az adott jelenet futtatása nélkül.

A fejlesztés során megírt szerkesztői mód tesztek a „Unity” alkalmazásban a „Window” fül „General” alpontjában a „Test Runner” alpont megnyitásával tekinthetők meg. Az ablak megnyitása után az „EditMode” pontot kiválasztva fognak a differenciálegyenlet megoldó algoritmus tesztjei megjelenni, és a futtatásuk a „Run All” gombra kattintva indítható el. Ehhez hozzá tartozik, hogy a tesztek futtatása számítógéptől függően akár jó pár percet is igénybe tud venni, a tesztesetek száma miatt.

Mindezek után térjünk rá a tesztek tartalmára. A modell réteghez tartozó egységtesztek a „PDESolverTest” osztályban találhatók. Alapvetően az osztály nyolc fajta tesztet tartalmaz, de ezek sokfajta paraméterezéssel hívódnak majd meg. A tesztek működése a következő:

- A „DensityAdditionTest” metódus ellenőrzi, hogy az üres sűrűség tömb egy eleméhez egy sűrűség értéket adva az adott érték egy algoritmus lépés után is benne marad-e a tömbben.
- A „DensityConservationTest” metódus ellenőrzi, hogy a sebesség mező valóban sűrűség tartó-e. Tehát ha egy adott sűrűség értéket hozzáadunk a sűrűség tömb egy eleméhez, akkor az több iteráció elteltével is teljes egészében megmarad-e a tömbben. Vagyis ellenőrzi, hogy a tömbelemek sűrűségeinek az összege az iterációk után megegyezik-e a hozzáadott értékkel.
- A „LongTermSystemDensityChange” metódus ellenőrzi, hogy ha a sűrűség tömb elemeit több iteráción keresztül növeljük egy adott értékkel, akkor az az érték megmarad-e a tömbben. Ez azt jelenti, hogy minden iterációban ellenőrizzük, hogy a tömbelemek sűrűségeinek az összege az előző iterációban megnőtt-e a hozzáadott értékkel.
- A „SystemDensityTestFromInitialValue” metódus ellenőrzi, hogy az üres sűrűség tömb egy elemét megnövelve és az algoritmust léptetve a tömb elemeinek az összege megegyezik-e a hozzáadott értékkel.
- A „SystemDensityTestFromPreviousValue” metódus ellenőrzi, hogy egy nemüres sűrűség tömb egy elemét megnövelve és az algoritmust léptetve a tömb elemeinek az összege megegyezik-e a hozzáadott értékkel.
- A „VelocityAdditionTestWithPoint” metódus ellenőrzi, hogy egy pont típusú eszközzel egy adott értéket hozzáadva a sebesség tömbök megfelelő elemeihez a sebesség tömb megfelelő értékei egy algoritmus lépés után valóban megváltoznak-e.
- A „VelocityAdditionTestWithRectangle” metódus ellenőrzi, hogy egy téglalap típusú eszközzel egy adott értéket hozzáadva a sebesség tömbök megfelelő elemeihez a sebesség tömb megfelelő értékei egy algoritmus lépés után valóban megváltoznak-e.
- A „VelocityAdditionTestWithSquare” metódus ellenőrzi, hogy egy négyzet típusú eszközzel egy adott értéket hozzáadva a sebesség tömbök megfelelő elemeihez a sebesség tömb megfelelő értékei egy algoritmus lépés után valóban megváltoznak-e.

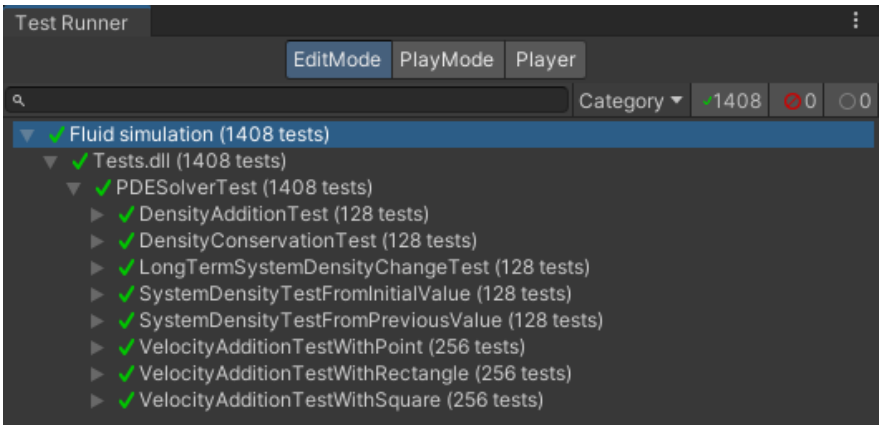
Az előbb részletezett metódusok mind megkapják paraméterként a felhasználói vezérlők értékeit, és a paraméterek minden lehetséges kombinációiban lefutnak. Az összes tesztben előforduló paraméterek a következők lesznek: felbontás, időegység, halmazállapot, viszkozitás,

lépésszám, gravitáció. Ezen kívül a sűrűség tesztek utolsó paramétere a hozzáadott sűrűség érték, a sebesség tesztek utolsó két paramétere pedig a hozzáadott két sebesség érték. Figyelembe véve azt, hogy minden paraméter két értéket vehet fel és egy metódusnak a megfelelő számú paramétere lehet, kiszámolható a tesztesetek száma. Ez a sűrűség teszt esetében 128 esetet, a sebesség teszt esetében pedig 256 esetet jelent.

A fenti sűrűségre vonatkozó teszt metódusok mindegyike olyan módon lett megvalósítva, hogy a sűrűség tömb egy véletlenszerű elemét növeljük meg egy adott értékkel. Ez a véletlen elem választás azonban a sebességre vonatkozó tesztek esetében nem lehetséges, mivel a kiválasztott alakzat miatt ez a tömbből való kiindexeléshez vezetne. Ennek érdekében ezen teszteknél minden esetben egy fix tömbelemnek a növelése, illetve csökkentése történik majd meg.

Ezen kívül a tesztek ellenőrzése egy hibakorlát bevezetése mellett történik meg. Erre azért van szükség, mivel az algoritmus csak közelíteni tudja az adott értékeket, tehát a numerikus hibás értékekkel kell tovább számolnunk. Továbbá a numerikus értékekkel való számítás miatt az egyenlőség vizsgálatra sincs lehetőség. Mindezek miatt az összehasonlítás olyan módon fog megtörténni, hogy a kiszámolt értékről döntjük el, hogy kisebb vagy nagyobb-e a hibával korrigált elvárt értéktől. Továbbá azt, hogy két értékre kisebb vagy nagyobb relációt alkalmazzunk az alapján lehet eldönteni, hogy az elvárt érték pozitív vagy negatív-e.

Mindezek után következzen a tesztek futtatása:



28. ábra: Tesztelés

A tesztek lefutásából következik, hogy a differenciálegyenlet megoldó algoritmus helyesen számítja ki a szimuláció lépéseit feltéve, hogy azok a megadott szélsőértékek között vannak.

Továbbá következik, hogy az egyenlet megoldó algoritmus helyessége miatt az esetlegesen a valósággal nem összeegyeztethető szimulációs kimenetek a megjelenítés hibájából adódhatnak.

További fejlesztési lehetőségek

Az alkalmazás implementációja két dimenzióban készült el, azonban a matematikai bevezetés részben tárgyalt megoldó algoritmus függvényei közül egyik sem korlátozódott két dimenzióra. Ennek következményeként az alkalmazás különösebb technikai nehézségek nélkül átalakítható háromdimenziós szimulációvá. Fontos azonban megjegyezni, hogy habár az algoritmus elméletileg általánosítható háromdimenzióba, azonban a gyakorlatban ez jelentős bonyolultság növekedéssel fog járni, amely miatt a hardver igény is jelentős lesz.

Mindezeket szem előtt tartva egy következő fejlesztési lehetőség egyrészt párhuzamos algoritmusok használata a program során, ezzel csökkentve a futási időt. Egy másik ehhez kapcsolódó lehetőség pedig az alkalmazás CPU implementációjának átírása GPU implementációra, ezzel is növelve a teljesítményt, és csökkentve a futási időt.

A szimuláció kibővítése is egy hasznos fejlesztési lehetőség lehet. A szimulációs paraméterek kibővítése a hőmérséklet értékkel például növelheti a gázok szimulálásának a pontosságát. Ezen kívül a folyadékokba helyezett testek szimulálása is hasznos lehet. Ehhez számításba kell azonban venni a felhajtó erőt, és az egyéb ehhez tartozó törvényeket is.

A szoftverben a Neumann, illetve a No-Slip peremfeltételek lettek használva, azonban lehetséges más peremfeltételek használata is. Free-Slip peremfeltételt használva például a falakhoz nem fog olyan mértékben hozzátapadni a szimulált anyag, periodikus peremfeltételt használva pedig elérhető, hogy a szimulációs tér egyik oldalán távozó anyag a vele átellenben lévő oldalon visszajöjjön. Ezen kívül gázok esetében a külső erőket ki lehet egészíteni egy örvényességet okozó erővel, amely garantálja, hogy a gáz folyamatos mozgásban maradjon.

Irodalomjegyzék

[1] Visual Studio telepítés: [Online]. Elérhető: https://learn.microsoft.com/en-us/visualstudio/install/install-visual-studio?view=vs-2022
[Hozzáférés dátuma: 2024.05.10].

[2] Unity Hub telepítés: [Online]. Elérhető: https://unity.com/unity-hub
[Hozzáférés dátuma: 2024.05.10].

[3] Unity: [Online]. Elérhető: https://unity.com/.
[Hozzáférés dátuma: 2024.05.10].

[4] Unity learn: [Online]. Elérhető: https://learn.unity.com/.
[Hozzáférés dátuma 2024.05.10].

[5] Unity Test Framework: [Online]. Elérhető: https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/index.html
[Hozzáférés dátuma: 2024.05.10].

[6] Folyadék szimuláció játékokhoz: Jos Stam 2003 „Real-Time Fluid Dynamics for Games”.

[7] Gáz szimuláció elmélet: Fedkiw, R., J. Stam, and H.W. Jensen. 2001. "Visual Simulation of Smoke."

[8] Folyadék szimuláció elmélet: Griebel, M., T. Dornseifer, and T. Neunhoeffer. 1998. „Numerical Simulation in Fluid Dynamics: A Practical Introduction”.

[9] GPU folyadék szimuláció: [Online]. Elérhető: https://developer.nvidia.com/gpugems/gpugems/part-vi-beyond-triangles/chapter-38-fast-fluid-dynamics-simulation-gpu
[Hozzáférés dátuma: 2024.05.10].

[10] Vektor analízis elmélet: [Online]. Elérhető: https://www.feynmanlectures.caltech.edu/II_02.html
[Hozzáférés dátuma: 2024.05.10].

[11] Véges differencia módszer elmélet: Christian Grossmann; Hans-G. Roos; Martin Stynes (2007). „Numerical Treatment of Partial Differential Equations”

[12] Végeselem módszer elmélet: Reddy, J. N. (2006). „An Introduction to the Finite Element Method” (Third ed.). McGraw-Hill.

[13] Leray projekciós operátor elmélet: Temam, Roger (2001). „Navier-Stokes equations: theory and numerical analysis”.

[14] Explicit-implicit numerikus módszerek elmélet: U.M. Ascher, S.J. Ruuth, R.J. Spiteri: „Implicit-Explicit Runge-Kutta Methods for Time-Dependent Partial Differential Equations”

[15] Dekompozíciós tétel bizonyítás: [Online]. Elérhető: https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://www.wtamu.edu/~cbaird/SuppHelmholtzDecomposition.pdf&ved=2ahUKEwichp2mjIuGAxXahv0HHdVLB3EQFnoECBIQAQ&usg=AOvVaw2A-xsGrF9t1NBUp1q8f6tB [Hozzáférés dátuma: 2024.05.13].