Budapesti Műszaki és Gazdaságtudományi Egyetem

Falling Sand szimuláció C++-ban

Önálló laboratórium beszámoló

Bertalan Áron

Bevezetés

A falling sand automaták egyszerű cella alapú autómaták (cellular automaton), melyek különböző részecskék egymással való interakcióját modellezik.

Ez a feladat több szempontból is nagyon tetszett nekem Egyrészt látványos vizuális eredményekkel jár, ami különösen motiváló volt. Másrészt érdekes kihívást ad a szimuláció optimalizálása. Tudomásom szerint az ilyen típusú szimulációkat leginkább csak játékokban használják, de az optimalizálási módszerek, amelyeket itt alkalmaztam, számos más területen is hasznosak lehetnek. Különösen a folyadékszimulációk mutatnak sok hasonlóságot, ahol a hatékony algoritmusok és adatstruktúrák alkalmazása kulcsfontosságú a valós idejű teljesítmény biztosításához.

Project terv

Tervezési Szempontok

A projekt tervezése során a legfőbb szempont a teljesítmény volt. Mivel rengeteg cellát kell a szimuláció minden egyes lépésében feldolgozni, az alapvető műveletek (pl.: `getCell`, `setCell`) nem igényelhetnek néhány utasításnál többet. Egy másik szükséges módja az optimalizációnak a feldolgozandó cellák számának csökkentése, melyet chunkok bevezetésével oldok meg.

Egy másik fontos része a tervezésnek a felelősségek jó elkülönítése volt (SRP). Ugyan a szimuláció egy részét teljesítmény miatt nem lenne előnyös OOP elveinek megfelelően implementálni, viszont a többi feladat megoldása során nagy segítséget nyújt.

Verziókezelés és feljesztési környezet

A projektet GitHub-on tárolom a https://github.com/Ayayron1234/cellular-automata repository-ban és git-et haszálok verziókezeléshez. Ezekre az eszközökre azért van egyedül is szükségem, mert lehetővé teszik, hogy különböző részfeladatokon dolgozzak párhuzamosan (ha egy feature éppen nem is működik, korábbi verzióját használhatom amíg egy újat implementálok).

Fejlesztői környezetként a Visual Studio 2022-t használom. Azért választottam, mert korábbi projektjeim során hasznosnak találtam a debuggerét és egyéb kényelmi funkcióit.

Kódoláshoz a C++20 stanndardot használom és az MSVC-t használom compiler-ként.

Megvalósítás

Cellák chunkokba szervezése

A Chunkokba szervezett adatnak sok előnye van. Egyrész így véges memóriával akár látszólag végtelen lehet a szimuláció tere (természetesen ez nem igaz, hiszen még ha nem is nagy a chunkok száma, akkor is a véges méretű számok miatt nem lehetne az origin-től túl messze lévő chunkokat kezelni), másrészt lehetőséget ad arra, hogy egyszerűen csökkentsük a szimuláció léptetése során frissített cellák számát.

Az én megoldásomban a szimuláció celláinak nyilvántartásához 2 osztályt használok: `Chunk` és `World`.

A `Chunk` feladata, hogy egy előre meghatározott számú cellát tároljon, egy statikus méretű array-ben. Ennek az array-nek az élettartama megegyezik a chunk élettartamával.

A `World` feladata pedig, a szimuláció során létrehozott chunkok kezelése, mely a projektemben a következő módon működik:

- A world egy hash map-ben tárolja a chunkokat ahol a kulcs a chunk koordinátája.
- Amikor egy cellát akarunk módosítani a World osztályon keresztül, a world először megnézi, hogy a cella koordinátájánál van-e már chunk, és ha nincs létrehoz egyet, majd meghivja a chunk `getCell` vagy `setCell` metódusát.

Az eddig leírt megoldás csak a szükséges memóriát csökkenti, az szimuláció teljesítményét viszont igazából csak csökkenti. Optimalizációra az ad lehetőséget, hogy minden szimulációs lépésnél el tudja dönteni a chunk, hogy őt a következő lépésben kell e frissíteni. Amennyiben úgy döntött, hogy nem, a cellákat frissítő algoritmus nem megy végig a chunkon így rengeteg időt spórol meg. A chunk frissítésének szükségességének eldöntése, a `ChunkState` osztály feladata.

Adatorientált struktúra

Már a projekt kezdetén tapasztaltam, hogy memóriában kevesebb helyet foglaló cellákkal sokkal gyorsabb tud lenni a szimuláció. Ennek leginkább az az oka szerintem, hogy kisebb méret mellett több cella tud egyszerre a gyorsítótárba kerülni. Kezdeti kísérleteim során ez akár 20-30%-os teljesítménynövekedést is eredményezett (igaz, a projekt ezen korai szakaszában a szimuláció algoritmusa még nagyon egyszerű volt, így valószínű, hogy a későbbiekben ez az előny valamelyest csökkent).

A lehető legkisebb cella méret érdekében, az alábbi módon tárolom a cellákhoz tartozó adatot. A különböző cella fajtákhoz tartozó tulajdonságokat egy union-ba szerveztem és minden értéket a számára szükséges minimális méretű bitfieldben tárolom.

```
union {
   struct {  // Properties common for each cell type
        Type     type : 5;
        unsigned shade : 6;
        ...
   }

   TypeASpecificProps a;
   TypeBSpecificProps b;
}
```

Ez a megoldás sajnos egyáltalán nem felel meg az OOP elveinek, azonban lényegesen csökkenti cella méretét.

Az implementáció során egyébként ez nem nagyon okoz extra kellemetlenséget, hiszen új cella típus felvétele igazából leszármazás mellett is komplikált lenne. Az OOP elvek követése mellett az osztálynak polymorph típusnak kellene lennie, hogy a különböző típusok update() metódusát meg lehessen hívni a cella típusának compile idejű ismerete nélkül, azonban így egy cella mérete a jelenlegi 4 helyett 16 byte lenne. Virtuális metódusok nélkül viszont nem ismerek olyan megoldást amivel ne kellett volna a cella típusa alapján még kódolás során különböző függvényeket meghívni, ismétlődő kóddal.

Párhuzamosság és Optimalizálás

A világ chunkokba szervezésének előnye, hogy így a cellák feldolgozását nagyon egyszerű megosztani különböző szálak között. Minden szál egy közös pool-ból kivesz egy chunkot majd frissíti a chunkban lévő összes cella állapotát. Ezt addig folytatják amíg el nem fogynak a chunkok. Ilyenkor jelzik a fő szálnak, hogy haladhat tovább a végrehajtás és várják, hogy újra legyenek frissítendő chunkok.

A szálak létrehozása, ütemezése és lezárása a `ChunkWorkerPool` osztály feladata. Ez az osztály egy thread pool implementáció kifejezetten a chunkok frissítésére tervezve. A chunkok iterálása a `ChunkCollection` template osztályon keresztül történik ami egy tetszőleges iterálható tároló osztályon tud végighaladni. A `ChunkWorkerTask` feladata pedig a chunk-okon végrehajtandó függvény és a hozzá tartozó paraméterek tárolása.

A fent leírt osztályok használatával, egyszerűen kezelhető a chunk-ok párhuzamos frissítése:

```
auto updateTask = new ChunkWorkerTask(&Chunk::process);
updateTask->setArgs(options, updateFunction);
m_workers->execute(updateTask);
```

A másik előnye a chunk-okba szervezésnek, hogy egy chunk jó egységet ad annak ellenőrzésére, hogy a benne lévő cellákat szükséges-e frissíteni. Egy cella frissítése akkor szükséges, ha megfelelő körülmények vannak a környezetében arra, hogy mozogjon, vagy ha valamilyen belső állapotváltozást hajt végre. Ezek eldöntéséért a `ChunkState` osztály felelős.

A `ChunkState` osztály mindig értesül a hozzá tartozó chunk celláinak írásáról és ilyenkor egyrészt megjegyzi, hogy a következő szimulációs lépésben is frissíteni kell, másrészt, ha az írt cella a chunk szélén van, akkor értesíti a szomszédos chunk-okat. A másik feladata az osztálynak, hogy figyeljen a chunk telítettségére. Ez azt jelenti a gyakorlatban, hogy ha a chunk-ban már csak üres cellák vannak, akkor a `World` felszabadíthatja a chunk-hoz tartozó memóriát.

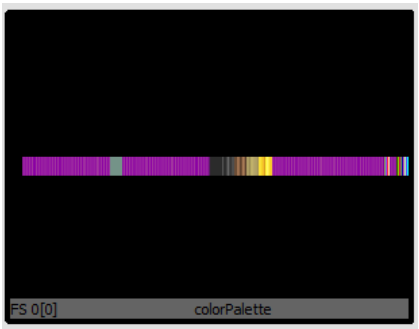
Világ renderelése

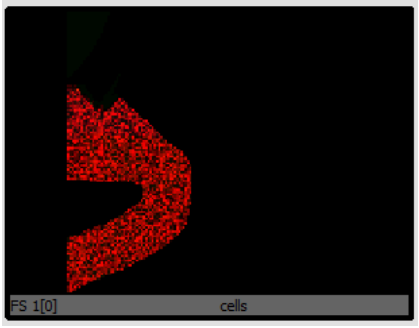
A renderelést harmadik nekifutásra (erről még a kihívások című szakaszban írok) végül OpenGL-el oldottam meg.

A renderelés CPU oldalon való kezeléséért a `ChunkView` és a `WorldView` osztályok felelősek. Ezek előkészítik a szükséges uniform-okat és létrehozzák az OpenGL draw hívásokat.

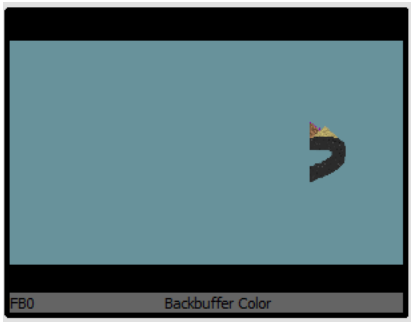
A chunkok kirajzolásának menete a következő:

- A `WorldView` frissíti a kamera uniform-ját és feltölti a szín palettát a GPU-ra egy 1 dimenziós textúraként.
- A `WorldView` rendre meghívja a chunk-ok ChunkView-jának a draw metódusát.
- A `ChunkView` eldönti hogy látszik-e a képernyőn. Ha nem akkor nem halad tovább a rajzolás műveletével.
- A `ChunkView` feltölti a GPU-ra a benne lévő cellákat tartalmazó tömböt.
- Meghívódik a `glDrawArrays` függvény
- Amennyiben engedélyezve van, kirajzolja a `ChunkView` a chunk határait is.





A fragment shader ezek után eldönti hogy a jelenlegi pixel melyik cellába esik, megnézi a cella első néhány bitjének az értékét és ezzel az értékkel indexeli a szín palettát.

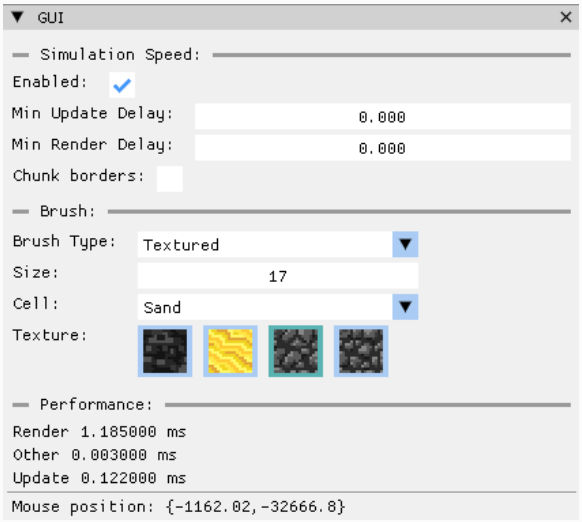



```
vec4 cellData
     = texture(cells, texcoord * (chunkSize-1) / chunkSize);

int byte  = int(cellData.r * 255.0);
int bytes = int(cellData.r * 255.0 + cellData.g * 255.0 * 255.0);

unsigned int type = (byte & 0x001F);
unsigned int shade = (bytes & 0x07E0) / 32;

vec4 color = texture(colorPalette, type / 32.0 + shade / 2048.0);
```

Felhasználói felület

A UI megvalósításához két könyvtárat használtam: SDL és ImGUI. SDL az ablak, egér és billentyűzet kezeléséhez kellett, ImGUI (Immediate mode Graphical User Interface) feladata pedig a szimulációval való interakció segítése, valamint a teljesítménnyel kapcsolatos metrikák mutatása.

Az SDL inicializálásával és eseményeinek kezelésével foglalkozó kódot egy korábbi projektemből vettem át (`IO.h`, `IO.cpp`), ez egy nagyon egyszerű implementáció, amiben csak a feltétlen szükséges funkcionalitást kezelem.



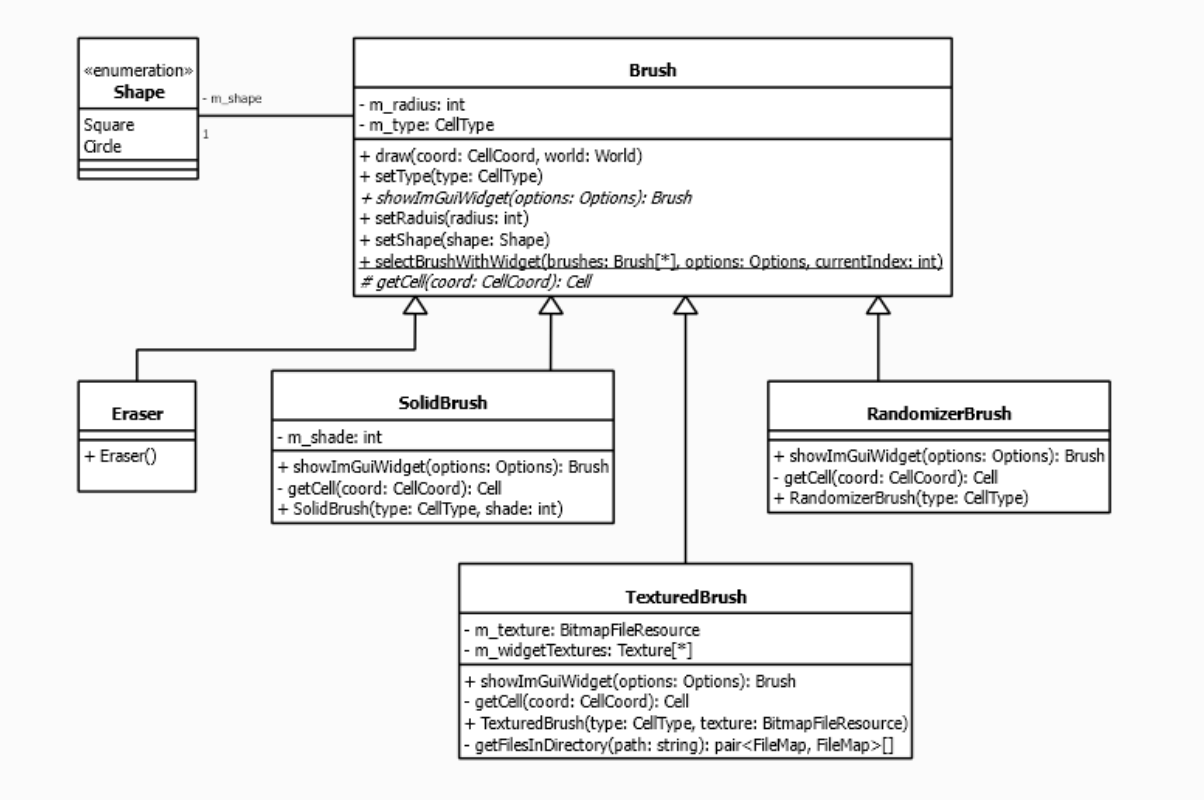
Az ImGUI ablak segítségével a következő lehetőségei vannak a felhasználónak:

- Szimuláció indítása és megállítása az `Enabled` checkbox-on keresztül.
- Szimuláció lassítása a `Min Update Delay` állításával. Ennek értéke megadja, hogy a szimuláció legalább mennyi időt várjon két szimuláció lépés között.
- Renderelés lassítása a `Min Render Delay` állításával. Ennek értéke a két frame között eltelt minimális időt adja meg.
- Chunkok határainak megjelenítésének ki- és bekapcsolása a `Chunk borders` checkbox-on keresztül.
- A szimulációba cellák elhelyezésének módja a `Brush:` szekcióban állítható, erről még később.
- A `Render`, `Update` és `Other` metrikák rendre a rendereléssel, szimuláció léptetésével és egyéb feladatokkal töltött időt mutatják

ezredmásodpercben. Ezen metrikák megszerzése a `PerformanceMonitor` singleton osztály feladata.
- A `Mouse position` a kurzor poziciójának x és y koordinátáját mutatja a világban.

Cellák elhelyezése a szimulációban

A szimulációba különböző cellák elhelyezése az implementációmban a `Brush` osztály és leszármazottai által lehetséges. Ezeknek az osztályoknak a feladata a `World` osztállyal való interakció (cellák elhelyezése), valamint a különböző ecsetek kiválasztására és beállítására való ImGUI widgetek megjelenítése és kezelése.

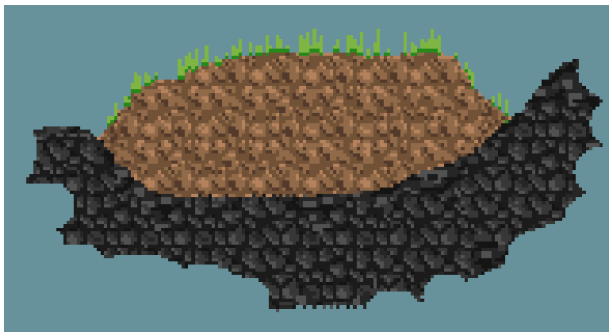


Az *Eraser* Air típusú cellákat helyez el a világban, ennek az ecsetnek nincs semmi speciális tulajdonsága.

A *SolidBrush* a konstruktorában paraméterként kapott `shade` és `type` paraméterekkel hoz létre azonos típusú és árnyalatú cellákat.

A *RandomizerBrush* a konstruktorban paraméterként kapott `type` típusú cellát hoz létre random `shade` értékkel.

A *TexturedBrush* is egy paraméterként kapott típusú cellát hoz létre, azonban a cella shade értékét egy kapott bitmap-ből tölti be. A textúrák közül az ImGUI widgeten lehet választani a `Texture:` felirat megjelenő képek közül. Új textúrákat a `data/textures/` mappába kell beilleszteni a következő képpen:



TexturedBrush-al rajzolt alakzat

- Cella shade-t megadó textúra: `<texture_name>.bmp`
- Textúra preview képe: `<texture_name>_preview.bmp`

A bitmap fájlok kezelése a `Bitmap` osztály feladata (ez egy gyors implementáció bitmap fájlok beolvasására).

Erőforrások kezelése

A program működéséhez sok külső erőforrás elérése szükséges. Ezek közül a főbbek:

- szimuláció, kamera és ablak általános beállításai (`data/options.json`),
- szín paletta, mely az cellák típusához és árnyalatukhoz egy színt rendel (`data/color_palette.bmp`),
- a világ általános adatait tartalmazó json file (`data/worlds/<world-name>/world.json`),
- a chunk-ok celláinak bináris adata (`data/worlds/<world-name>/chunks/<chunk-name>.bin`),
- a `TexturedBrush` által használt textúrák (`data/textures/<texture-name>.bmp`)
- és a shaderek (`data/shaders/<shader-name>.glsl`).

Így tehát 4 típusú erőforrás létezik a projekten belül: bináris, json, bitmap és shader.

A bináris fájlok kezelésével a `Chunk` osztály foglalkozik, és a `World` osztály `save` és `load` metódusán keresztül lehet kezelni. Ennek az implementációja egyszerű bináris írás és olvasás.

A json fájlok kezelésére már korábban készítettem egy kezdetleges parsert (`Json.h`). Ennek elkészítése nem volt a legegyszerűbb, hiszen c++-ban reflexió hiányában nem lehet hozzáférni az osztályok deklarációjához futtatás idejében. Egy példa a használatára az Options osztály utáni makró:

```
JSON_C(Options,
    JSON_M(windowWidth), JSON_M(windowHeight),
    JSON_M(camera), JSON_M(stateTransitionTickDelay),
    JSON_M(brushSize), JSON_M(simulationEnabled),
    JSON_M(brushCellType), JSON_M(updateWaitTimeMs),
    JSON_M(renderWaitTimeMs)
)
```

Ez ugyan nem a legelegánsabb, de összetett típusokkal is működik és miután a fenti makróval megadtam a szükséges adatokat az osztály szerializációja nagyon egyszerű lesz:

```
Options options{};
Json optionsJson = Json(options);
Options copy = optionsJson;
```

A bitmap fájlok olvasására létrehoztam a Bitmap osztályt. Ennek csak a fájl header és a pixel array beolvasása a feldatata.

A shaderek kezelésére az OpenGL függvényeit használom.

Az egyszerűbb és gyorsabb fejlesztés érdekében (ugyan erre valószínüleg ilyen rövid futamidejű projektben nem feltétlenül volt szükség) a json, bitmap és shader fájlokra implementáltam hot-reloadingot, így a program újraindítása nélkül is megváltoztathatóak (ez leginkább a shader-ek javításánál jött jól). A hot-reloading kezelése ezen típusokra rendre a `JsonFileResource`, `BitmapFileResource` és a `TextFileResource` plusz `Shader` osztályok felelősek.

Kihívások

Mint szinte minden projektben, itt is a legtöbb kihívást magamnak okoztam, néha jól elrejtett hibákkal, máskor pedig rosszul megtervezett részekkel, amelyeket később újra kellett dolgoznom.

A legtöbb időmet a renderelés kétszeres újraírása vette el. A projekt kezdetén a Témalaboratórium tárgy keretében megoldott feladatot vettem alapul, amely egy CUDA-val megvalósított fraktálrajzoló volt. Ezt azért tettem, mert megfelelő tapasztalatom hiányában féltem OpenGL-el implementálni a chunkok rajzolását. A CUDA azonban, mivel elsősorban nem erre a célra szolgál, sok felesleges fejfájást okozott az adatstruktúrám tervezésénél és implementációjánál. Amikor végül sikerült működésre bírnom, a teljesítmény elmaradt az általam elvárttól és nem is tudtam rá büszkén nézni. Második próbálkozásom pusztán SDL renderer-t használta, azonban ez amellett, hogy lényegesen át kellett terveznem a párhuzamos részét a kódomnak, valahogy még kevésbé volt hatékony. Végül kénytelen voltam feladni a küzdelmet és mégis átismételni az OpenGL működését, hogy implementálni tudjak egy olyan renderelőt amire büszek tudok lenni.

Másik nagy kihívást az időm nem éppen optimális beosztása okozta. Arra számítottam, hogy a félév vége fele lényegesen kevesebbet fogok tudni foglalkozni a projektel, viszont az is sok időt elvett, hogy a feladat szempontjából kevésbé fontos részek túlságosan lekötötték a figyelmemet. Ilyen volt például a TexturedBrush amivel szerintem messze túl sokat foglalkoztam és elég lett volna egy kevésbé kidolgozott verzió, vagy a bitmap fájlok és shaderek hez írt hot-reloading, ami ugyan talán nem vett el annyi időt, viszont ez is jól mutatja, hogy kevéssé fontos részeit előrébb vettem a projektnek.

Egy kihívás ami nem hibás döntésből származott az a multi-threading megoldása volt. Számomra ez viszonylab új feladat volt, korábban csak kevésbé részletesen, nem annyira kidolgozva foglalkoztam ilyennel, szóval gyakran kellett hosszú ideig keresnem egy-egy jól megbúvó hibát.

Lehetséges Fejlesztések

Szimuláció irányában a fejlesztési lehetőségek gyakorlatilag határtalanok, hiszen számtalan valóságon vagy akár fantázia szülte részecskék közötti interakciót lehetne modellezni, ezért erre itt nem is nagyon térnék ki.

Optimalizáció irányában egy lehetséges továbbfejlesztés a dirty rectangle lenne, amit eredetileg terveztem is implementálni, viszont végül sajnos kifutottam az időből. Ez úgy működik, hogy minden chunk-on belül számontartja hogy mely koordináták között történhet változás és a chunk frissítése során csak ebbe a területbe eső cellákkal foglalkozik a szimuláció.

Egy másik nagy előrelépés a szilárd testek bevezetése lenne. Erre nagyon jó példa a Noita nevű játék fizikája. A szilárd testeket külön fizikai motor kezeli, mely kölcsönhatásban van a falling sand automatával ezen objektumok részecskéi egy testként mozognak és képesek eltörni, leesni, stb.

Vizuális téren hasonló alkalmazásokban egy gyakori feature a bloom. Ennek bevezetése nem igényelne túl sok változtatást a renderelésen, de nagy előrelépés lenne a látvány érdekességében.

Összefoglalás

A végeredményként létrejött alkalmazás véleményem szerint szórakoztató, élveztem bele új részecsketípusok felvételét és az interakciójuk figyelését.

Összességében meg vagyok elégedve a létrehozott programmal. Nagyrészt szerintem sikerült optimális, kiterjeszthető és átlátható kódot írnom, ami a tervem volt. Sok tapasztalatot szereztem, leginkább a párhuzamos tervezés és debugolás és az OpenGL használatában.

Hivatkozások

Függőségek

- SDL - https://github.com/libsdl-org/SDL
- ImGUI - https://github.com/ocornut/imgui
- OpenGL - https://www.opengl.org/
- glew - https://glew.sourceforge.net/

Inspiráció és segítség

- https://w-shadow.com/blog/2009/09/29/falling-sand-style-water-simulation/
- https://pvigier.github.io/2020/12/12/procedural-death-animation-with-falling-sand-automata.html
- https://www.youtube.com/watch?v=5Ka3tbbT-9E&t=1084s
- https://nollagames.com/fallingeverything/

Napló

A napló elejéről hiányzik jópár bejegyzés. Ennek oka, hogy utólagos tárgyfelvétel miatt csak akkor értesültem a napló írásának szükségességéről amikor már a projekt része készen volt. Sajos ezen kívül se sikerült mindig időben lejegyeznem a feldattal töltött időmet, de a lényeges részének implementációja azért szerepel köztük.

2024.03.23	3 óra	fixed bugs with chunk updating
2024.03.30	2 óra	Multithreading hibák javítása. Valamivel stabilabb a rendszer.
2024.04.02	3 óra	Thread pool rework. Ez az első thread pool amit csináltam, szóval csak idő közben értettem meg a c++ multithreadinghez tartozó osztályok működését. Sikerült megelőzni a random lefagyásokat, de rájöttem hogy vannak hibák amik a jelenlegi végrehajtással nem javítható. (a chunkok feldolgozása 3 szinkronizált fázisban kell történjen.)
2024.04.04	2 óra	Új chunk feldolgozási rendszer tervezése. (vázlatos szekvencia diagram)
2024.04.05	5 óra	Thread pool rework. Implementáltam a korábbi tervet, így a multi threading már jól működik, viszont nem vagyok elégedett a jelenlegi rendereléssel ami CUDA-t használ, mert feleslegesen sokszor passzolok adatot videó memóriába és vissza.
2024.04.08	3 óra	Rendering rework to only use SDL_Renderer. A chunkokat SDL_Texture textúrába rajzoltam cpu-ról multithreading-el. Hiba: SDL nem tudja kezelni a több szálról érkező hívásokat.
2024.04.18	12 óra	Rendering reworked to opengl and reorganized Chunk class to have less responsibility (Created: ChunkView, ChunkState, WorldView). A chunkok celláinak renderelését megvalósítottam opengl-ben. (Chunk határokat még nem tud rajzolni, és minden chunk egy külön draw call amit később lehet még javítani.)
2024.04.19	3	Chunk border drawing implemented with opengl, minor

	óra	bug fixes. (ideiglenes megoldás, később majd chunkoktól független, optimalizált négyzetrajzolással kéne megoldani.)
2024.04.20	5 óra	Hot shader reloading. Shaderek autómatikusan frissülnek, ha az őket tartalmazó file-ba írok. Letisztítottam kicsit a Shader osztály implementációját.
2024.04.23	6 óra	Bug hunting. Javítottam egy hibát a szomszédos chunkok frissítésével. Javítás egyszerű volt, de nehezen találtam meg a hiba okát.
2024.04.24	3 óra	Fixed occasional freezing on chunk creation. Fixed some memory leaks. Started removing CUDA integration which was needed for earlyer rendering implementation.