Automatică și calculatoare

**Anul III**

**SIMULATOR FIZIC:**

**SIMULAREA ÎN TIMP REAL  
A NISIPLUI CĂZĂTOR**

**Candidat: Mironică Vasile**

**Coordonator științific: Lect. Norbert Gal-Nădășan**

Sesiunea: Iunie 2024

**Cuprins**

**1.0 Introducere**

**2.0 Prezentare generală**

**2.1 Piatră**

**2.2 Nisip**

**2.3 Apă**

**2.4 Fum**

**2.5.0 Motivația din spatele regulilor complexe**

**2.5.1 Ulei**

**2.5.2 Abur**

**2.5.3 Foc**

**2.5.4 Aer**

**2.5.5 Accelerația gravitațională**

**3.0 Structura proiectului**

**3.1 Configurarea folosind cmake**

**3.2 Structura directoarelor**

**3.3 Prelucrarea intrărilor**

**3.4 Un pas în simulare**

**3.5.0 Afișarea la ecran**

**3.5.1 CameraView**

**3.5.2 Modificarea pixelilor**

**3.5.3 Diversificarea culorilor**

**3.6 Forțe externe**

**4.0 Accelerarea simulării**

**4.1 Trierea Chunk-urilor**

**4.2 Afișarea activității**

**4.3 Paralelizarea actualizării chunk-urilor**

**4.4.0 OpenCL**

**4.4.1 clinfo**

**4.4.2 Rularea unui kernel**

**4.4.3 Memoria globală**

**4.4.4 Prezentarea kernel-ului final**

**4.4.5 Argumentul performanței kernel-ului final**

**1.0 Introducere**

Din cauza limitărilor tehnologice și a dorinței de a imita „graficele vieții reale”, jocurile video tind să arate o parte foarte mică a lumii în care jucatorul se află. Acest fapt conduce direct la lipsa dorinței de a juca un joc a doua oară – deoarece jucătorul poate explora, cu *relativ* puțin efort, multitudinea de acțiuni preprogramate, precum și multitudinea de obiecte create de artiști, lăsând puțin spațiu pentru imaginație.

Dacă se face un compromis între **realism** și **puterea de procesare**, se poate rezolva problema unei lumi statice, astfel aducându-se în joc o lume în care acțiunile jucătorului influențează direct întreaga lume în același timp, conducând la mecanici foarte interesante, consecințe neprevăzute și comportament imprevizibil. Ideile acestui proiect sunt foarte simple:

1. Tratează toate elementele din lume unul câte unul;
2. Elementele pot interacționa doar cu elementele din imediata lor împrejurime.

Prima idee conduce direct la ce vrem să obținem – o lume cât se poate de dinamică în care interacțiunile nu sunt „optimizate în neexistență”, iar a doua idee este prezentă ca o limitare a primei idei, ca să economisim putere de procesare, permițând jocului să ruleze în timp real. O a treia idee a proiectului, care nu întotdeauna se menționează, dar are afect *masiv* asupra experienței, este **performanța** – jocul trebuie să se simtă fluid în mișcare, jocul trebuie să răspundă „instant” la acțiunile jucătorului (se va reveni la această idee în capitolul Performanță).

Limbajul de programare predominant în lucrare este **C++** - un limbaj foarte puternic și bine cunoscut în comunitate pentru stabilitatea, flexibilitatea și performanța sa, dovedite pe parcursul întregii sale vieți de nenumărate ori. Partea grafică(afișarea la ecran) este adusă de către librăria open-source **SFML** – **S**afe and **F**ast **M**ultimedia **L**ibrary, care vine la pachet și cu funcții pentru citirea intrărilor de la mouse și tastatură. Afișarea la ecran este accelerată pe placa video cu ajutorul proiectului **OpenCL** de la organizația Khronos.

**2.0 Prezentare generală**

Ideea se comportă, la bază, ca un Automat Celular – se începe cu starea zero, iar starea următoare este determinată calculand rezultatul aplicării unor reguli asupra elementelor din starea precedentă. Simulatorul este capabil să simuleze 8 substanțe: aer, nisip, piatră, apă, ulei, abur, foc și fum. Regulile comportamentului elementelor se pot clasifica în două tipuri:

1. **Reguli simple** – schimbarea poziției elementului este singura consecință a trecerii de la o stare la alta.
2. **Reguli compuse** – în urma interacțiunilor cu un alt element, acesta își schimbă forma.

Un element poate avea multiple reguli simple, precum și multiple reguli compuse.

**2.1 Piatră**

Elementele de piatră servesc ca platformă pentru alte elemente. Acestea sunt complet staționare – nu sunt mișcate de nimic și ele însele nu sunt afectate de gravitație.

**2.2 Nisip**

Un element de nisip trebuie, în primul rând, să respecte gravitația.

* **Regula 1:** Dacă o celulă de nisip nu are nimic direct sub ea, aceasta ar trebui să înlocuiască elementul direct sub ea (fig. 2.1 ←).

|  |  |
| --- | --- |
|  |  |
| Figura 2.1: Nisip, regula 1 | Figura 2.2: Grămadă de nisip, folosind doar regula 1 |

Însă, doar această regulă generează comportament nenatural, nisipul formând „piloni”, când nisipul tinde să se „imprăștie” în viața reală. Această problemă poate fi ușor rezolvată introducând o regulă nouă:

* **Regula 2:** Daca o celulă de nisip nu are nimic direct în stânga-jos (sau în dreapta-jos), celula de nisip trebuie sa inlocuiască respectiva celulă (fig. 2.3 ↓).
  1. Dacă ambele poziții sunt valabile, se va selecta una aleatoriu cu șanse egală.
  2. Dacă nicio poziție nu este valabilă, celula de nisip stă pe loc.

|  |
| --- |
|  |
| Figura 2.3: Nisip, regula 2 |

|  |  |
| --- | --- |
|  |  |
| Figura 2.4: Grămadă de nisip în starea inițială | Figura 2.5: Grămadă de nisip simulată conform regulilor 1 și 2 |

**2.3 Apă**

Apa este un fluid, deci, în contextul simulării, are proprietatea de a umple spațiul în care se află. Deasemenea, apa este afectată de gravitație și are proprietatea de a se împrăștia – rezultă ca putem folosi aceleași reguli ca și la nisip, adaugând doar una nouă:

* **Regula 3:** Dacă o celula de apă nu are un alt element direct în stânga (sau în dreapta) ei, celula de apă trebuie sa inlocuiască respectiva celulă.
  1. Dacă ambele celule sunt valabile, se va selecta una aleatoriu cu șanse egale.
  2. Dacă regula a fost aplicată deja, trebuie să fie aplicată din nou în aceeași direcție.
  3. Dacă nicio poziție nu este valabilă, celula de apă stă pe loc.

**Observație:** Subregula **(b)** este adăugată, deoarece în lipsa ei celula de apă va ajunge mereu în aceeași poziție (asumând că durata simulării este infinită), din cauza faptului că șansa de a merge în dreapta sau în stânga este egală.

|  |
| --- |
|  |
| Figura 2.6: Apă, regula 3 |

|  |  |
| --- | --- |
|  |  |
| Figura 2.7: Apă în starea inițială | Figura 2.8: Apa după ce a umplut volumul dat |

**2.4 Fum**

Fumul este un gaz, acesta are aceleași proprietăți ca și un fluid – însă inversate pe axa verticală, rezultă că i se aplică aceleași reguli ca și apei, însă calculele coordonatelor sunt inversate pe axa verticală.

|  |  |
| --- | --- |
|  |  |
| Figura 2.9: Fum în starea inițială | Figura 2.10: Fum după simulare |

**2.5.0 Motivația din spatele regulilor complexe**

Cu toate că regulile curente au creat comportament destul de credibil pentru substanțele alese, aici este atins potențialul lor – nu putem introduce substanțe mai dinamice și imprevizibile, precum focul, doar cu acestea. Din acest motiv am introdus *regulile complexe*, care au scopul de a fi executate înaintea regulilor simple pentru a genera comportament mai dinamic!

Regulile simple depind numai de tipul substanței date și vecinii ei apropiați. Regulile complexe, pe de altă parte, sunt capabile să stocheze informație în celule (precum cronometre sau impuls) și să reacționeze în dependință de această informație în orice moment alt timpului.

**2.5.1 Ulei**

Amestecul uleiului cu apa este un experiment clasic pentru demonstrarea diferenței între densitățile diferitor substanțe. Comportamentul care dorim să-l replicăm în simulare este atunci când acestea două intră în contact direct, uleiul să rămână mereu deasupra apei.

Densitatea este o proprietate crucială în orice simulare, doarece e natural să ne gândim că nisipul ajunge la fundul unui lac, sau că uleiul „plutește” deasupra apei, cu toate că acesta este lichid la rândul său.

Pentru a obține acest comportament este suficient să stocăm într-un tabel densitățile tuturor substanțelor, și când aplicăm regulile simple să le aplicăm doar atunci când densitatea elementului curent este mai mare decât a elementului vecin.

* *Observația 1:* Nu este nevoie să stocăm valoarea exactă, reală a densității, deoarece în calcule contează doar ca densitatea uneia să fie mai mare ca a celeilalte.
* *Exemplu*: Introducerea valorilor numerice exact 1 și 2 ca densități pentru, respectiv, ulei și apă aduc același rezultat ca și densitățile reale exacte.
* *Observația 2:* Avem și opțiunea de a stoca densitatea elementului direct în celulă, însă am irosi memorie stocând aceeași valoare de mai multe ori, deoarece constanta fizică nu se poate schimba.

|  |  |
| --- | --- |
|  |  |
| Figura 2.11: Configurație aleatorie de ulei și apă | Figura 2.12: Aceeași configurație din fig 2.11 înainte să ajungă la echilibru complet |

**2.5.2 Abur**

Aburul este un gaz asemănător cu fumul, doar că cu densitate mai mare. În natură se observă cum aburul se poate condensa înapoi la apă dacă se află în anumite condiții – condiții care încalcă caracterul simplist al simulării (nu simulează presiunea, temperatura și alte fenomene naturale datorită cantității enorme de computații necesare). Însă putem replica transferul căldurii (în timp) cu un *generator de numere pseudo-aleatorii*!

Un **Generator de Numere Pseudo-Aleatorii** este un subprogram care generează o secvență finită de numere pseudo-aleatorii. Datorită faptului că calculatoarele moderne sunt Mașini Turing, acestea sunt deterministe și nu sunt în stare să genereze numere cu adevărat aleatorii, însă în contextul simulării nu avem nevoie de numere aleatorii, ci doar numere care „par suficient de aleatorii pentru ochiul uman”. O clasă foarte bine cunoscută de astfel de generatoare sunt **Generatoarele Liniar Congruențiale**.

Generatoarele Liniar Congruențiale sunt populare din cauza faptului că sunt foarte ușor de implementat și sunt extrem de ieftin de computat. Generatorul ales în proiect este generatorul din librăria standard a calculatorului **ZX81**.

|  |
| --- |
| **class FastRng {**  **public:**  **FastRng() {**  **std::srand(std::time(nullptr));**  **seed\_ = std::rand();**  **}**  **int32\_t NextValue() {**  **seed\_ = 75 \* seed\_ + 74;**  **seed\_ %= (1 << 16) + 1;**  **return seed\_;**  **}**  **private:**  **int32\_t seed\_;**  **};** |
| Figura 2.13: Implementarea unui generator asemănăptor cu cel din ***ZX81*** |

Constructorul generatorului inițializează generatorul la valoarea actuală a numărului de secunde în formatul *Posix*. Funcția **NextValue** computează următoarea valoare a generatorului, o stochează ca să fie folosită în apelurile următoare și o returnează.

Dacă simulăm numărul de încercări de care avem nevoie ca să atingem, spre exemplu, un multiplu al lui **5** de 1.000.000 de ori și le comparăm cu un standard – generatorul din limbajul Python, obținem aceste grafice:

|  |  |
| --- | --- |
|  |  |
| *Figura 2.14: 1.000.000 de simulări ale generatorului de numere pseudo-aleatorii* ***ZX81*** | *Figura 2.15: 1.000.000 de simulări ale funcției random.random() din* ***Python*** *cu condiția să valoarea să fie mai mică ca 0.2* |

Unde putem observa că distribuția simulărilor este foarte asemănătoare.

Cu un generator de numere pseudo-aleatorii putem efectiv să simulăm trecerea unei durate pseudo-aleatorii de timp, deoarece actualizarea lumii se face la intervale egale de timp, și la fiecare iterația putem resimula generarea unui numâr aleator până când condiția noastră este satisfacută.

Folosing generatorul de numere pseudo-aleatorii, putem adăuga o astfel de regulă:

* *Dacă un element de abur are minim un vecin direct care este aer, atunci are o șansă de 20% să se condenseze în apă.*

|  |
| --- |
|  |
| *Figura 2.16: Nor de abur condensându-se în apă într-un spațiu închis* |

**2.5.3 Foc**

În simulare, dorim ca *focul* să aibă proprietăți distructive: să consume *uleiul* și să evapore *apa*. Precum și să creeze *fum* când se află la sfârșitul vieții.

Regulile le putem imlementa astfel:

1. Dacă avem un vecin *apă*, convertim elementul curent în aer și vecinul în abur.
2. Dacă avem un vecin *ulei*, avem o șansă aleatorie sa convertim elementul curent în foc.
3. Dacă avem un vecin *aer*, avem o șansă aleatorie să distrugem elementul curent și să-l înlocuim cu fum.

Putem scuti elementul de reguli simple, ca să ne asigurăm că stă pe loc în lume, dându-i gravitație ca să poată restul elementelor să poată interacționa cu el.

|  |
| --- |
|  |
| *Figura 2.17: Minge de foc producând fum* |

|  |  |
| --- | --- |
|  |  |
| *Figura 2.18: Minge de foc pe cale să interacționeze cu apă* | *Figura 2.19: Minge de foc evaporând apa în cale* |

|  |  |
| --- | --- |
|  |  |
| *Figura 2.20: Minge de foc în mijlocul unei bălți de ulei* | *Figura 2.21: Răspândirea focului în ulei* |

**2.5.4 Aer**

Aerul este un concept foarte simplu în simulare, apărut din cauza nevoii unui element stabil, care nu face nimic singur – pasiv. Fără culoare, fără reguli, se află în simulare doar ca să fie înlocuit de alte elemente!

**2.5.5 Accelerația gravitațională**

Accelerația gravitațională se manifestă foarte simplu în realitate – obiectele în cădere tind să cadă din ce în ce mai repede, până la viteza terminală. Pentru a implementa asta în simulare, avem nevoie să stocăm în elemente impulsul vertical al acestora și să mărim valoarea la fiecare apel a **Regulii 1**, unde următoarea poziție elementului v-a fi calculată luând în calcul valoarea impulsului.

Singura problemă care apare cu această implementare, este șansa ca elementul să cadă „prin” alt element, ca și cum acesta nici nu ar fi fost acolo. Pentru a evita astfel de accidente, trebuie să verificăm fiecare element până la poziția dorită ca acesta să aibă densitatea mai mică ca elementul curent, iar dacă întâmpinăm un element cu densitate mai mare, oprim elementul curent din cădere înainte să ajungă la blocaj (fig. 2.22 ↓).

Pentru un comportament „mai natural”, putem să simulăm și impactul cu blocajul – în momentul impactului, după ce am mutat elementul în poziția de coliziune, alegem o direcție aleatorie (stânga sau dreapta), un impuls echivalent cu cel vertical și mutăm elementul orizontal (fig 2.23 ↓). ­*Impulsul orizontal nu este păstrat în simulare.*

|  |  |
| --- | --- |
|  |  |
| *Figura 2.22: Cădere accelerată* | *Figura 2.23: Căderea unui element (0 - 1), ciocnitura cu un alt element care îl oprește din drum (2) și aruncarea acestuia în dreapta (3)* |

**3.0 Structura proiectului**

Proiectul este construit cu ajutorul sistemului **Cmake**, care se ocupă de configurarea proiectului(opțiunile alese, transferatul asset-urilor etc.), configurația este apoi trimisă către generator (*make*, *ninja*, etc.).

Compilatorul pe care a fost făcută dezvoltarea este **clang++**, scris de echipa **LLVM**. Versiunea de C++ folosită este **c++-17**.

**3.1 Configurarea folosind cmake**

În timpul configurării se execută următorii pași:

1. Dacă variabila „*USE\_OPENCL\_FOR\_DRAW*” este setată la „*TRUE*”, se caută librăria **OpenCL** la nivel de sistem.
   * În cazul în care OpenCL nu este instalat se semnalează eroare pe loc și proiectul pică la configurare.
2. Dacă variabila „*USE\_CLANG\_TIDY*” este setată la „*TRUE*”, i se comunică generatorului să ruleze și analizorul static odată cu compilarea fișierelor sursă.
   * Analizorul static ales este „*clang-tidy*”, scris de echipa **LLVM**, versiunea folosită în timpul dezvoltării este 18.1.3.
3. Este librăria **SFML** instalată la nivel de sistem?
   * În cazul în care SFML nu este instalat se semnalează eroare pe loc și proiectul pică la configurare.
   * Componentele dorite să fie link-uite sunt: *graphics, window, system*. Acestea sunt componentele necesare din librăria *SFML* ca să putem crea o fereastră și să putem manipula pixelii din ea.
4. Se copie directorul **assets** în directorul executabilului.
   * Directorul **assets** conține fontul folosit pentru a scrie la ecran și kernel-ul folosit de *OpenCL*.
5. Se setează flagurile compilatorului la „*-Wall -Wextra -pedantic -Werror -O3*”
   * Primele patru flaguri se asigură că compilatorul ajută programatorul să facă mai puține greșeli „umane” în cod – marind numărul de warning-uri pe care compilatorul le raportează, dintre care flagul „*-Werror*” schimbă prioritatea warningu-urilor la erori de compilare, oprind compilarea când cu status eșuat dacă sunt întâmpinate.
   * Flagul „*-pedantic*” semnalează compilatorului să raporteze și warning-uri de paritate cu diferite compilatoare/arhitecturi – sporește șansa proiectului să poată fi construit pe alte mașini, cu compilatoare diferite, fără probleme.
   * Flagul „*-O3*” permite compilatorului să optimizeze agresiv codul **Assembly** generat.
6. Generatorului îi sunt comunicate toate fișierele sursă – luate recursiv din directorul „src”, precum și locația header-urilor folosite – directorul „include”.
7. Dacă variabila „*GENERATE\_DOCUMENTATION”* este setată la „*TRUE*”, generatorul va creea și documentația folosind pachetul **Doxygen**.
   * În cazul în care Doxygen nu este instalat se semnalează eroare pe loc și proiectul pică la configurare.

**3.2 Structura directoarelor**

Punctul de pornire al aplicației – funcția main() - se află în fișierul sursă „Main.cpp” (fig. 3.1 ↓).

|  |
| --- |
| **int32\_t main(const int32\_t /\*argc\*/, const char\* const\* const argv) {**  **const std::string\_view executable\_path(argv[0]);**  **std::filesystem::current\_path(**  **std::filesystem::path(executable\_path).parent\_path());**  **FallingSandEngine falling\_engine("Falling Sand Simulator",**  **{constants::kWorldWidth, constants::kWorldHeight});**  **falling\_engine.Run();**  **return 0;**  **}** |
| *Figura 3.1: Funcția main()* |

Funcția main are rolul de a creea motorul și de a porni motorul. Variabila standard C++ „*std::filesystem::curent\_path*” este modificată să reflecte locația exactă a executabilului, astfel putem citi asset-urile din același director cu executabilul, în cazul în care directorul din care se rulează aplicația nu conține și asset-urile.

Bucla de execuție a simulatorului arată astfel (fig 3.2 ↓):

|  |
| --- |
|  |
| *Figura 3.2: Bucla de execuție a simulării* |

Diferite calculatoare au diferite puteri de procesare, asta conduce la discrepanță între calculatoarele care sunt în stare să ruleze simularea de multiple ori pe secundă și cele care sunt în stare să ruleze de multiple ori pe *milisecundă*. Pentru a remedia această problemă și a face ca simularea să arate la fel indiferent de puterea de procesare (atâta timp cât aceasta este suficientă), se poate folosi o simplă buclă cu un cronometru, care permite continuarea simulării doar la intervale egale de timp (fig. 3.3 ↓). Durata intervalului aleasă în proiect este de 60 de cadre pe secundă, deoarece este valoarea cea mai des întâlnită ca rata de actualizare a monitoarelor.

|  |
| --- |
|  |
| *Figura 3.3: Bucla care asigură simularea la intervale egale de timp – de 1/60 secunde* |

**3.3 Prelucrarea intrărilor**

Pe decursul rulării SFML stochează toate evenimentele care au loc, acestea pot fi taste apăsate la tastatură, mișcarea mouse-ului, închiderea ferestrei etc. Pentru a prelucra aceste elemente se apelează repetat funcția „*sf::RenderWindow::pollEvent(Event &event)*” (fig. 3.4 ↓).

|  |
| --- |
| **for (sf::Event event; window.pollEvent(event);) {**  **switch (event.type) {**  **case sf::Event::Closed: {**  **// Fereastra a fost închisă**  **break;**  **}**  **case sf::Event::KeyPressed: {**  **if (event.key.code == sf::Keyboard::Key::A) {**  **// A fost apăsată tasta A**  **}**  **break;**  **}**  **...**  **}**  **}** |
| *Figura 3.4: Bucla care tratează evenimentele din fereastra „window”* |

Funcția „*pollEvent”* returnează în parametrul de tip „*sf::Event*” codul evenimentului și informațiile acestuia, iar valoarea returnată de funcție este un boolean care comunică dacă coada de evenimente este goală sau nu – așa știm să oprim bucla și nu mai avem evenimente de tratat. Tipul evenimentului poate fi identificat cu un switch, în ramurile căruia

sunt folosite informațiile evenimentului (implementate de SFML folosind o uniune de mai multe structuri reprezentând proprietățile fiecărui tip) pentru a modifica starea simulării.

**3.4 Un pas în simulare**

Structura de date în care sunt stocate elementele este un tablou bidimensional, unde elementele arată ca în (fig. 3.5 ↓).

|  |
| --- |
|  |
| *Figura 3.5: Structura Element* |

Având deja setate regulile pentru substanțele care le dorim simulate, a trece la următorul pas în simulare înseamnă să iterăm prin toate elementele din lumea bidimensională și să aplicăm regulile.

**Problema** în această abordare – de sus în jos și de la stânga la dreapta (fig. 3.6 ↓) – este că un element care este mutat în dreapta sau în jos va fi actualizat de mai multe ori pe iterație. Ca să rezolvăm această problemă se introduce un nou tablou bidimensional de valori boolean-e, de aceeași mărime ca și tabloul elementelor, unde o să stocăm dacă elementul de la un set de coordonate a fost deja actualizat sau nu, iar dacă se întâlnește un element deja actualizat, acesta se ignoră. În pseudo-funcția „UpdateElement” trebuie să fie setată valoarea vizitei la poziția elementului curent, precum și la poziția unde a ajuns în urma simulării acestuia.

*La fiecare pas al simulării tabloul de vizitați trebuie resetat la 0.*

|  |
| --- |
| **for (uint32\_t element\_y = 0; element\_y != kWorldHeight; ++element\_y) {**  **for (uint32\_t element\_x = 0; element\_x != kWorldWidth; ++element\_x) {**  **if (not visited[element\_y][element\_x]) {**  **world[element\_y][element\_x].UpdateElement();**  **// Se setează visited[element\_y][element\_x] = true**  **// Se setează visited[new\_element\_y][new\_element\_x] = true**  **}**  **}**  **}** |
| *Figura 3.6: Actualizarea elementelor unul câte unul* |

**3.5.0 Afișarea la ecran**

O tehnică de a afișa la ecran o lume simulată este de a atribui fiecărui pixel un element, astfel maparea de la lume la ecran este trivială – elementelor i se atribuie o culoare și cu ajutorul unei bucle bidimensionale se iterează prin toate elementele și la aceleași coordonate în lume se setează culoarea. Dezavantajul în această metodă este că lumea trebuie setată, obligatoriu, să aibă aceeași dimensiune ca și ecranul, dar dacă vrem să avem o lume cu dimensiuni variate și abilitatea de a facem zoom pe o anumită parte a lumii, trebuie să implementăm un obiect numit, în literatură, *CameraView*. Rolul camerei este să țină minte unde se află ochiul ecranului în lume.

**3.5.1 CameraView**

Obiectul CameraView arată astfel (fig. 3.7 ↓):

* Variabile membre:
  + **max\_size\_** : pereche de coordonate setate la mărimea lumii – au rolul de a limita poziția și mărimea camerei înauntrul lumii.
  + **view\_** : efectiv dreptunghiul vizibil.
  + **zoom\_level\_** : nivelul l-a care s-a făcut zoom.

|  |
| --- |
|  |
| *Figura 3.7: Obiectul CameraView* |

Deoarece dorim să facem diferența între planul lumii și planul camerei, trebuie să folosim sisteme diferite de coordonate (fig. 3.8 ↓), funcțiile *MapCoordsToPixel* și *MapPixelToCoords* fac tocmai asta:

* *MapCoordsToPixel* : conversia unui punct de la planul lumii la planul camerei.
  + Notăm:
* *MapPixelToCoords* : conversia unui punct de la planul camerei la planul lumii.

|  |
| --- |
|  |
| *Figura 3.8: Coordonatele unui punct în planul lumii și în planul camerei* |

Funcția MovePosition are rolul de a muta camera relativ cu poziția la care este deja:

* Notăm:
* ,

|  |
| --- |
|  |
| *Figura 3.9: Mutarea camerei la o poziție nouă – de la A1 la B1* |

Funcția Zoom v-a schimba mărimea camerei ca să aducă mai aproape (sau mai departe) un anumit punct (fig. 3.9 ↓). În proiect, punctul P este poziția mouse-ului, care se află în planul camerei. Dacă considerăm dreptunghiul lumea și dreptunghiul camera, dacă dorim să facem zoom 2x pe punctul P, atunci dreptunghiul reprezintă setarea nouă a camerei la care dorim să ajungem.

,

|  |
| --- |
|  |
| *Figura 3.10: Zoom pe un punct P* |

**3.5.2 Modificarea pixelilor**

Odată ce avem ce avem obiectul *CameraView*, afișarea la ecran are loc astfel: se iterează prin toți pixelii de pe ecran, se convertesc coordonatele pixelilor din planul camerei în planul lumii, se citește elementul de la poziția dată și se colorează pixelul în dependință de culoarea elementului.

SFML dispune de multiple tehnici de a modifica pixelii, însă o problemă clasică de performanță este atunci când se fac prea multe apeluri către API-ul care se ocupă cu comunicația cu GPU-ul, majoritatea copleșitoare a timpului este petrecută schimbând contextul între GPU și CPU, astfel avem nevoie de o tehnică în care se trasminte cât mai puțină informație câtre GPU. O astfel de tehnică este adusă de obiectul „*sf::Sprite*” în SFML. **sf::Sprite** permite utilizatorului să atașeze sprite-ului o textură, a cărei pixeli pot să fie modificați fără să se facă schimbarea contextului între dispozitive, iar când textura este finalizată ea este trimisă către GPU, care o afișează la ecran.

**3.5.3 Diversificarea culorilor**

Folosind câte o singură culoare pentru fiecare element devine foarte repede foarte plictisitor ochiului uman, pentru a remedia acest lucru putem aplica două tehnici:

1. Se setează mai multe posibile culori pentru elementele solide (nisip, piatră), iar la crearea lor (nu la desenarea lor!) li se atribuie una aleatorie, efectul produs este că elementele vor avea aceeași culoare (unică!) de fiecare dată când sunt afișate la ecran. Această valoare poate fi stocată în structura Element folosind o singură valoare întreagă.
2. Se setează mai multe posibile culori pentru celelalte elementele (gaze și fluide), iar la crearea acestora li se atribuie, din nou, o valoare întreagă aleatorie, însă de data asta culoarea elementului va fi setată ciclic în mulțimea de culori aleasă, valoarea întreagă reprezentând doar offset-ul culorii. Efectul produs este asemănător unei animații.

**3.6 Forțe externe**

Jucătorul are nevoie să interacționeze cu elementele din lume, altfel aplicația nu poate fi numită joc. Jucătorul are la dispoziție următoarele seturi de controale:

* *Cifrele 1-9* : setarea elementului.
* *Tasta F2* : pune pauză la simulare.
* *Tasta F3* : informația folositoare la debug.
  + Numărul de cadre pe secundă.
  + Procentul de timp ocupat de simulare dintr-un cadru. Dacă trece de 100% simularea nu mai este în timp real.
  + Poziția mouse-ului în planul lumi și în planul camerei.
  + Procentajul din cadru ocupat de tratarea evenimentelor.
  + Procentajul din cadru ocupat de actualizarea lumii.
  + Procentajul din cadru ocupat de afișarea la ecran.
* *Tasta F5* : afișarea delimitării chunk-urilor.
* *Tasta F7* : chunk-urile active sunt colorate cu verde, cele inactive cu roșu.
* *Mouse buton stâng apăsat* : așezarea elementului ales la poziția la care este mouse-ul.
  + Elementele de sub mouse sunt suprascrise la apăsare.
* *Mouse buton mijloc târât* : mutarea camerei în direcția opusă vectorului târârii.
* *Rotița la mouse* : mărirea/micșorarea pensulei cu care se așează elemente în lume.
* *Ctrl + Rotița la mouse* : zoom-in și zoom-out.

|  |
| --- |
|  |
| *Figura 3.11: Substanțele oferite și tastele necesare accesării* |

**4.0 Accelerarea simulării**

**4.1 Trierea Chunk-urilor**

Indiferent de puterea de procesare a unui calculator, cu o lume suficient de mare se va ajunge la rezultatul dezamăgitor că timpul în care se actualizează lumea este mult prea mare și se pierde aspectul de „în timp real” al jocului, deoarece numărul de secunde pe cadru depășește limita setată de . Prima tehnică aplicată pentru a repara situația este:

* **Trierea** elementelor care nu merită actualizate.

Ideea este simplă, nisipul după ce a căzut se stabilizează și nu se mai mișcă, apa după ce a umplut întregul spațiu nu mai are unde curge și deasemenea se oprește – *aceste părți ale lumii nici nu ar trebui atinse la actualizare*! Implementarea constă în:

* Se creează o nouă structură de date, a cărei rol este să împartă lumea în pătrate de mărime egală (fig. 4.1 ↓) – chunk-uri, unde o să stocăm activitatea. *Structura de date în care sunt stocate elementele nu este schimbată!*
* Actualizarea lumii nu se mai face pe fiecare element în parte, ci pe fiecare chunk în parte, înăuntrul căruia se actualizează fiecare element în parte – rezultatul final este același.
* Regulile de actualizare sunt modificate să returneze un valorea boolean-ă care reprezintă dacă elemenul dat „a făcut ceva” (s-a mișcat în lume, și-a schimbat forma etc.).
* Dacă, în actualizarea unui chunk, minim un element a schimbat lumea într-un fel valoarea activității chunk-ului este setata la TRUE, altfel la FALSE.
* Dacă un chunk a suferit modificare prin forțe externe, acesta are valoarea activității setată la TRUE.

|  |
| --- |
|  |
| *Figura 4.1: Împărțirea unei lumi 80x144 elemente în 5x9 chunk-uri de mărime 16x16.*  *(se observă liniile negre care delimitează chunk-urile)* |

Problema în această abordare este că unele elemente, după ce au fost actualizate, au șansa să intre în alt chunk, iar deoarece în această abordare activitatea chunk-urilor este setată doar din chunk-ul din care au început, nu în chunk-ul în care au ajuns, astfel celula (care spre exemplu, ar fi trebuit să-și continue căderea) stă pe loc necontenită (fig. 4.2↓ și fig. 4.3→).

|  |  |
| --- | --- |
|  |  |
| *Figura 4.2: Celulă de apă pe cale să intre în alt chunk* | *Figura 4.3: Chunk-ul neu nefiind actualizat, în pasul următor simulărea se va opri cu motivul că nu mai are nimic de făcut* |

Pentru a rezolva această problemă, se vor seta toate chunk-urile adiacente (nouă la număr) ca active.

|  |
| --- |
|  |
| *Figura 4.4: Toate chunk-urile adiacente fiind acum active, picătura de apă va continua să cadă în jos afectată de gravitație.* |

**4.2 Afișarea activității**

Prin convenție, am colorat chunk-urile inactive cu culoare **roșie** și cele active cu culoare **verde**, ca în (fig. 4.2 ↑). Pentru a le afișa ecran am folosit obiectul „sf::RectangleShape”, care arată astfel (fig. 4.5 ↓):

|  |
| --- |
|  |
| *Figura 4.5: sf::RectangleShape (doar atributele și metodele relevante)* |

Pentru a desena un dreptunghi:

1. Se creează un dreptunghi nou cu dimensiunile chunk-ului (care depind de zoom):
   * sf::RectangleShape rect\_shape(chunk\_size, chunk\_size);
2. Se setează poziția lui la poziția sus-stânga a chunk-ului:
   * rect\_shape.setPosition(chunk\_pos, chunk\_pos);
3. Se setează culoarea lui în dependință de activitatea chunk-ului. Deoarece dorim ca culoarea să fie transparentă, folosim a patra componentă a pixelului – valorea ALPHA, și o setăm la 51 (valorile admisibile sunt 0..255).
   * rect\_shape.setColor(0xFF000033); // pentru roșu
   * rect\_shape.setColor(0x00FF0033); // pentru verde
4. Se afișează la ecran:
   * rect\_shape.draw();

Problema în această implementare este că, în cazul în care afișăm mii de chunk-uri, desenarea activității pare să fie mai lentă decât simularea! Asta se întâmplă din cauza numărului foarte mare de schimbări de context care îl face API-ul care transmite informația GPU-ului. Pentru a depăși acest impas folosim tehnica **Batching** – o să grupăm mai multe dreptunghiuri de aceeași culoare, de pe aceeași linie, într-unul singur (fig. 3.6 ↓).

În exemplul din figura 3.6, numărul de apeluri „window.draw()” a scăzut de la 66 la numai 20! De pe urmele acestei tehnici avem numai de câstigat odată ce mărim numărul total de chunk-uri.

|  |
| --- |
|  |
| *Figura 4.6: Exemplu aleator de grupare a dreptunghiurilor* |

**4.3 Paralelizarea actualizării chunk-urilor**

Izolarea actualizării lumii în bucăți mici – chunk-uri, ne dezvăluie oportunitatea de a paraleliza actualizarea acestora pe CPU. CPU-urile moderne dispun de multiple nuclee, toate fiind capabile de a accesa memorie cache separată, de a rula cod complet separat, pe ramuri separate.

Motivul pentru care nici nu se încearcă paralelizarea pe GPU este că regulile de actualizare a elementelor prezintă mult prea multe ramuri de execuție, ceea ce nu sunt o problemă (exagerat de) mare pentru un CPU, însă GPU-urile tratează ramurile de execuție complet diferit. GPU-urile sunt sisteme care exploatează paralelismul la maxim, prin faptul că sute și mii de instanțe execută același cod în același timp (până la aceeași instrucțiune), eliminând foarte mult timp pierdut pe care l-ar face un CPU, care nu-și permite să ruleze atât de multe instanțe în același timp (însă instanțele individuale sunt mult mai rapide comparativ cu ale unui GPU!). GPU-urile execută diferitele ramuri de execuție în cel mai naiv mod posibil: din mulțimea de instanțe, cele care trebuie să execute o ramură o vor executa, iar celelalte vor *aștepta ca ele să termine*; procesul se repetă până când toate ramurile au fost executate – fenomen numit **Divergența ramurilor** (fig. 4.7 ↓). Divergenta ramurilor este un efect foarte dăunător performanței unui program executat pe GPU, deoarece, efectiv, o parte din GPU stă și așteaptă pentru cealaltă parte – din acest motiv, actualizarea lumii pe GPU este practic imposibilă.

|  |
| --- |
|  |
| *Figura 4.7: Divergența ramurilor pe GPU și reconvergența* |

C++ expune utilizatorului accesul la firele de execuție prin includerea fișierului din biblioteca standard „thread”. Prima funcție care trebuie explorată este

* **std::thread::hardware\_concurrency();**

Aceasta indică numărul de fire de execuție care hardware-ul le poate executa în paralel (fără a crea niciun fir virtual). Practica generală este să se scadă 1 din acest număr ca să păstrăm sistemului de operare un fir de execuție. Dacă se se folosește un număr mai mare decât acesta este garantat faptul că se vor folosi fire de execuție virtuale, care nu aduc performanță aplicației, deci (în acest caz) e important să nu le folosim.

Următorul punct de referință este obiectul

* **std::thread**

Constructorul obiectului „std::thread” primește funcția care să fie executată într-un alt fir de execuție, precum și o lista variadică de argumente care să fie transmiși funcției ca parametri.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17** | **auto update\_chunks\_by\_id = [&](const int32\_t thread\_id) {**  **// Actualizează chunk-urile care satisfac condiția**  **// (chunk\_idx % update\_threads\_) == thread\_id**  **};**  **std::vector<std::thread> worker\_threads;**  **worker\_threads.reserve(update\_threads\_ - 1);**  **for (int32\_t i = 1; i < update\_threads\_; ++i) {**  **worker\_threads.emplace\_back(update\_chunks\_by\_id, i);**  **}**  **update\_chunks\_by\_id(0);**  **for (auto& thr : worker\_threads) {**  **thr.join();**  **}** |
| *Figura 4.8: Paralelizarea actualizării lumii pe CPU* | |

* **Liniile 1-4** : Definesc funcția lambda care capturază *toate* variabilele din acest domeniu și primește ca parametru indexurile chunk-urilor care trebuie să le actualizeze.
* **Linille 6-11** : Rezervă spațiu pentru „*update\_threads\_ - 1*” fire de execuție și le creează, cu parametrul respectiv fiecăruia (fig 4.9 ↓). Se scade 1 din numărul de fire deoarece primul o să fie tratat de firul curent de execuție.
* **Linia 13** : Firul curent tratează chunk-urile care au rămas.
* **Liniile 15-17** : Se asigură că toate firele de execuție au terminat – rezultă că lumea a fost actualizată.

|  |
| --- |
|  |
| *Figura 4.9: Patru fire de execuție actualizând 25 de chunk-uri în paralel* |

**4.4.0 OpenCL**

**Open** **C**omputing **L**anguage este un limbaj care permite rularea de cod pe dispozitive tip acceleratoare – DSP, GPGPU, FPGA. Standardul OpenCL permite scrierea codului – în funcții numite **kernel** – o singură data și rularea acestuia pe un număr mare de platforme. OpenCL expune o bibliotecă scrisă în C pentru a accesa facilitățile standardului, dar expune și un header scris în C++ care înfășoară funcțiile de C, funcționalitate valabile în spațiul de nume „**cl**”, adesea făcând limbajul mai ușor de înțeles și mai ușor de folosit.

**4.4.1 clinfo**

Comanda „clinfo” expune absolut toată informația de care are nevoie utilizatorul ca să folosească un dispozitiv capabil de a rula OpenCL, informații precum extensiile disponibile, mărimea memoriei locale, a memoriei globale, numărul total de instanțe care pot fi rulate, numărul de instanțe care pot fi grupate pentru a diviza munca etc.

În figura 4.10 ↓ a fost folosit argumentul „-l” ca să printeze doar numele platformelor valabile, deoarece comanda tinde să scrie foarte multă informație altfel.

|  |
| --- |
| **user@user:~/data/programs/C++/Falling\_Sand/build(master)**  **$ clinfo -l**  **Platform #0: NVIDIA CUDA**  **`-- Device #0: NVIDIA GeForce RTX 4070** |
| *Figura 4.10: Rularea comenzii „clinfo -l” pe un sistem rulând Ubuntu cu un GPU montat* |

**4.4.2 Rularea unui kernel**

Funcțiile care sunt paralelizate de OpenCL se numesc, conform standardului, **kernel**deoarece acestea sunt, efectiv, nucleul programului – ele sunt funcțiile care consumă cele mai multe resurse și cel mai mult timp, motiv pentru care sunt scrise pe GPU. Pentru a rula un kernel utilizatorul trebuie să instanțieze obiectele cu care se comunică cu GPU-ul, precum să le elibereze la finalul muncii.

Pașii spre a rula un kernel OpenCL sunt următorii:

1. Selectarea unui dispozitiv și crearea contextului.
   * Pentru a selecta dispozitivul optim, în lucrare se iterează prin toate dispozitivele de tip GPU valabile și se selectează acela care maximizează formula

, cu toate că performanța unui dispozitiv depinde mult mai mulți factori, aceștia doi sunt adesea suficient pentru a le compara.

* + **cl::Context d\_context(GetFastestDevice());**

1. Crearea unei cozi de comenzi execuție folosind contextul dispozitivului.
   * Coada de execuție este structura de date menținută de driver-ul OpenCL care ține contul tuturor operațiilor care urmează să fie rulate și rulează pe dispozitiv, precum și statusul acestora, motivul pentru care primește dispozitivul la creare este că este, în întreaga ei viață, dependentă de el.
   * **cl::CommandQueue d\_queue(d\_context);**
2. Crearea programului care conține instrucțiunile ce trebuie rulate pe GPU.
   * Posibilitatea compilării programului în timpul rulării aplicației este motivul pentru care OpenCL este atât de flexibil, deoarece nu se livrează executabile clar configurate pentru o paltformă anume, ci surse care trebuie compilate cu driverul OpenCL.
   * În lucrare programul este citit ca cod sursă dintr-un fișier de pe disc predefinit și este compilat la începutul programului, deoarece compilarea de kernele durează foarte mult timp.
   * Constructorul programului primește contextul de dispozitive pentru care să se compileze kernel-ul și însuși codul sursă al kernel-ului.
   * **cl::Program d\_program(d\_context, kernel\_source);**
3. Crearea kernel-ului care va rula pe GPU.
   * Deoarece un program poate conține mai multe kernele, obiectul kernel primește ca argumente programul și numele funcției care să fie transformată în kernel.
   * În lucrare kernel-ul care populează textura cu pixeli este numit „PaintOn”.
   * **cl::Kernel d\_kernel(d\_program, ”PaintOn”);**
4. Rularea kernel-ului.
   * Pentru a rula kernel-ul, dispozitivul are nevoie să știe cum sa asigneze resursele între unitățile de computații și între instanțe, numărul total de instanțte care trebuie invocate și configurația instanțelor în grupuri.
   * **d\_queue.enqueueNDRangeKernel(d\_kernel, global\_range, local\_range);**
5. Eliberarea memoriei.

**4.4.3 Memoria globală**

Dispozitivele tip GPU dispun de propria memorie RAM, în standardul OpenCL aceasta este numită **memorie globală**. Programul mamă care rulează pe CPU nu poate accesa această memorie cum se accesează și RAM-ul, ci trebuie să folosească obiectele „**cl::Buffer**” expuse de OpenCL, iar pentru a transmite date la și de la dispozitiv trebuie să folosim coada de comenzi (fig. 4.11 ↓).

|  |
| --- |
|  |
| *Figura 4.11: Comenzile care transmit memorie între RAM și VRAM* |

Înaintea de rularea kernelului memoria din VRAM folosită trebuie inițializată cu input-ul care îl primește kernel-ul și după rularea acestuia datele trebuie aduse în RAM. Kernel-ul, ca orice funcție, poate primi parametri, însă setarea acestora este (marginal) mai complicată decât un apel de funcție în limbajul C, parametrii trebuie setați manual folosind funcția cu semnătura:

* **d\_kernel::setArg(size\_t arg\_index, type\_t\* arg\_value);**

Indexul este poziția arguementului în lista de argumente, începând de la 0. Pentru argumentele cu tip de date trivial este suficient sa trimită adresa către spațiul de memorie care are datele – funcția este templetizată pe tipul de date al argumentului, astfel compilatorul poate singur să deducă câți biți are tipul de date de la adresa dată, iar pentru spații de memorie globală se transmite direct buffer-ul de tip „*cl::Bufffer*”.

**4.4.4 Prezentarea kernel-ului final**

Kernel-ul naiv și cel mai ușor de implementat este un kernel care invocă o instanță pentru fiecare pixel de pe ecran, translatează punctul pe mapă și decide culoarea pixelului în funcție de elementul de la poziția dată. Această metodă este prezentată în (fig. 4.12 ↓).

|  |
| --- |
| **\_\_kernel void PaintOn(**  **\_\_global const struct Element elements[WORLD\_SIZE\_Y][WORLD\_SIZE\_X],**  **\_\_global unsigned pixels[SCREEN\_SIZE\_Y][SCREEN\_SIZE\_X],**  **const float view\_left, const float view\_top,**  **const float view\_step\_x, const float view\_step\_y**  **) {**  **const unsigned2 screen = {get\_global\_id(0), get\_global\_id(1)};**  **const unsigned2 world = {**  **view\_left + screen.x \* view\_step\_x,**  **view\_top + screen.y \* view\_step\_y**  **};**  **pixels[screen.y][screen.x] = ColorOf(elements[world.y][world.x]);**  **}** |
| *Figura 4.12: Kernel-ul care populează textura ecranului* |

* În variabila „screen” se stochează coordonata pixelului la care arată instanța curentă.
* În variabila „world” se stochează coordonata elementului respectiv în lume, calculată folosind poziția de sus-stânga a camerei și distanța între elemente pe ecran (care variază în funcție de zoom).
* Funcția „ColorOf” citește elementul de la poziția calculată și îi decide culoare, care este apoi scrisă în pixelul atribuit instanței. Funcția „ColorOf” este un simplu Lookup Table pe mulțimea predefinită de culori predefinită pentru fiecare element.
* Se observă ca mărimea lumii și a ecranului este dată la timpul compilării, asta are loc deoarece codul kernel-ului trebuie să fie cât mai eficient, astfel se pierde multe resurse, spre exemple, stocând valorile date în regiștri, apoi în decodificarea instrucțiunilor care fac operații pe acești regiștri. Variabilele transmise ca parametri sunt transmise astfel deoarece au șansa să se schimbe în fiecare cadru al jocului, spre deosebire de mărimea lumii și ecranului care mereu au aceeași valoare.

**4.4.5 Argumentul performanței kernel-ului final**

În ciuda implementării naive, aproape triviale, această o să atingă performanța maximă pe majoritatea GPU-urilor, deoarece GPU-urile moderne sunt limitate de viteza la care instanțele accesează memoria globală.

Dat fiind faptul că acest kernel are intensitatea aritmetică de doar aprox. 10/12 de **operații pe byte**, calculată ca numărul de operații pe cantitatea de memorie transmisă, care se întâmplă să fie 12 bytes, iar cu toate că numărul de operații nu este ușor de numărat (poate varia de la o platformă la alta), 10 este luat în acest caz ca limita superioară.

Dispozitivele moderne sunt capabile de o intensitate aritmetică de *zeci de operații pe byte* până ajung să fie limitate de puterea computațională, o intensitate aritmetică *subunitară g*arantează că cel mai mult timp se pierde accesând memoria globală, astfel orice imbunătățire la nivel de cod din acest punct va aduce adaosuri minimale în performanță.

**5.0 Asigurarea calității**

Proiectul a fost dezvoltat în concordanță cu **Google C++ Coding Style**. Acest standard expune o suită de reguli pentru a produce cod C++ care este ușor de întreținut, ușor de expandat și ușor de explicat. În această suită de reguli se află reguli de numire a indentificatorilor (variabile, funcții etc.), reguli de formatare a codului, interzice folosirea unor caracteristici C++ mai puțin ajutătoare – subiective la finalul zilei, însă libertatea în exprimare care o oferă limbajul este adesea compleșitoare și deciziile luate pe moment pot influența negativ capacitatea pe viitoare a înțelege codul scris azi, iar respectarea unui standard a ajutat mult a păstra un anumit de nivel de întelegere a codului în toate momentele de timp.

Stilul a fost forțat în cod folosind limbajul de server **clangd**. *clangd* rulează în timp real analizorul static, care la rândul său obligă folosirea regulilor precizate în fișierele „.clang-tidy” și „.clang-format” din rădăcina proiectului. Pe lângă asta *clangd* oferă colorizarea sintactică foarte profesională, care adesesa face mai ușoară la ochi citirea codului.

**6.0 Potențial**

Jocurile „sandbox” au devenit foarte populare în ultimii ani, jocuri precum Minecraft și Terraria au capturat inimile a milioane de jucători prin aparenta infinitate de interacțiuni cu mediul inconjurător, cu uneltele care jocul le pune la dispoziția jucătorului. Cu toate că această simulare este, deasemenea, un sandbox, nu poate fi ignorat faptul că pare incompletă – pur și simplu nu are suficiente substance și interacțiuni ca să țină un jucător interesant pentru o perioada mai lungă de timp.

**6.1 Potențiale interacțiuni și substanțe**

Din acest punct de vedere, cerul este limita! Substanțe precum sticla, care să fie produsul nisipului încălzit la o temperatură foarte mare, acidul – care ar dizolva toate substanțele și s-ar anula cu apa, sau lemnul – care ar absorbi apă ca să crească, sunt doar câteva exemple care ar mări *exponențial* numărul de moduri în care mediul creat de simulare poate interacționa cu sine însuși, făcându-l să pară cât mai natural.

Cu toate că o Simulare de Fluide Euleriene (*Euler Fluid Simulation*) este înafara limitărilor setate pentru simulare – această simulare are nevoie ca toate substanțele să interacționeze între ele indiferent de distanță, *Principul Vaselor Comunicante* este ceva ce pare natural, însă simularea eșuează lamentabil să pară naturală în acest caz (fig 5.1 ↓).

|  |
| --- |
|  |
| *Figura 5.1: Aceeași masă de apă, cu un perete în mijlocul acesteia, eșuând să se echilibreze conform principiului vaselor comunicante.* |

Implementarea *Principiului Vaselor Comunicante* este posibilă folosind o structură de date care stochează, pentru fiecare masă de apă, nivelele apei în diferinte coordonate și tinde în timp să le echilibreze.

**6.2 Corpuri rigide**

În realitate atomii unui obiect nu se dezintegrează instant, spre exemplu o scândură va rămâne intactă până i se aplică suficientă forță – în acest context, scândura este un corp rigid, simularea este complet incapabilă de a simula așa ceva cum este acum. Jocul Noita simulează folosind o combinație de algoritmi, enumerând câteva:

* *Marching Squares* este un algoritm folosit în simulările de pixeli/voxeli ca să micșoreze dramatic numărul de vârfuri ale unui corp geometric, astfel micșorând masiv valoarea computativă necesară pentru a simula acest obiect (fig. 5.2 ↓).

|  |
| --- |
|  |
| *Figura 5.2: Algoritmul Marching Squares micșorând numărul de vârfuri ale unui corp rigid. Se observă conturul roșu inițial al obiectului, apoi conturul nou verde al acestuia, care deține mult mai puține vârfuri deci este mai ușor de computat.* |

* Obiectul creat în urma alogoritmului poate fi trimis direct unui motor de simulare a coliziunilor corpurilor fizice, precum **Box2D.**

**6.3 Doar un motor pentru jocuri**

Cel mai mare potențial se ascunde în posibilitățile de a crea ceva deasupra acestei simulări – a crea un joc cu acest motor de jocuri. O lume dinamica complementează foarte bine un stil dinamic de joc, un exemplu poate să apară într-o lume post-apocaliptică generată procedural care se supune legilor simulării, jucătorul luând adăpost într-o vale, în timp ce adversarii acestuia încearcă să dărâme muntele, să provoace o avalanșă care să termine orice speranță a jucătorului – sentimentul că ceva se poate întâmpla înafara camerei, înafara domeniului care jucătorul îl stăpânește este foarte puternic când este integrat la scară mai largă.

**6.4 Lume generată procedural**

Lumile „infinite”, generate procedural, sunt foarte populare în jocurile sandbox, deoarece atunci când sunt executate bine, oferă jucătorului un grad made de *replayability* – numărul de interacțiuni este chiar mai mare ca înainte, jucătorul acum are o lume de explorat și sentimentul de a explora această lume diversă este saturat.

Tehnica general aprobată pentru a genera o lume procedural este de a lua multiple generatoare de numere pseudo-aleatoare, configurațe în așa fel încât la final să ofere teren care „pare natural”. Printre astfel de generatoare se află *Perlin Noise* – în general pentru teren sau *Voronoy Noise* – popular la randarea norilor. Multiple astfel de generatoare sunt folosite pentru a simula teren natural, luând în calcul relieful, temperatura, umiditatea, geologia, structuri de interes și multe altele.

Într-o lume infinită, care poate fi modificată, este foarte importantă stocarea acesteia. Stocarea me disc oferă posibilitatea jucătorului să oprească jocul și să revină la el mai târziu. Deasemenea deschide oportunitate de a juca multiplayer – multipli jucători jucând pe aceeași lume, observându-și schimbările altora și distrându-se împreună.

**7.0 Concluzii**

Am învățat foarte multe scriind acest proiect. A fost necesară rescrierea sa de multiple dăți, deoarece de fiecare dată învățam ceva nou ce trebuia neapărat aplicat – până am ajuns la structura curentă, care permite adăugarea (prin cod) relativ rapid de substanțe noi, localizarea corectă și eficientă a codului în proiect. Separarea motorului în obiecte separate numite „GameEngine” și „FallingSandEngine”, FallingSandEngine moștenind din GameEngine, s-a dovedit a fi o decizie înțeleaptă, deoarece izolația acestora a micșorat mărimea fișierelor proiectului (care în C++ tind să ajung exagerat de mari și contorsionate dacă nu se exercită precauție), micșorând timpul de compilare, cât și timpul irosit căutând unde este o anumită fărâmă de cod.

Sistemul de configurare **cmake** s-a dovedit a fi un aliat puternic al proiectului, flexibilitate oferită în configurația proiectului, viteza acestuia la execuție, cât și la adăugarea librăriilor noi, abilitatea de a rula analiză statică cu foarte puține bătăi de cap au contribuit enorm la a scădea timpul petrecut implementând idei și creșterea timpului de creat idei noi. Unealta de compilare **ccache** – provine de la „**c**ompiler **cache**”, oprește compilatorul din a compila un fișier a doua oară folosind filierul obiect generat de la o compilație anterioară, micșorând foarte mult timpul de compilare, a ajutat enorm în orele de debugging.

Multiplele dăți când observam că timpul de rulare dintre cadre creștea dramatic au dus la nenumărate ore în care exploram opțiuni de îmbunătățire, așa s-au născut soluția multi-core pentru actualizarea lumii, folosirea librăriei OpenCL pentru paralelizarea muncii, comunicarea către compilator a cât mai multă informație și timpul petrecut studiind librăria SFML ca să urmăresc practici bune.

Cu toate că nu are un scop bine definit, simularea reprezintă un pas important spre crearea unui motor de jocuri personal și deschide nenumărate uși spre a continua studiul optimizării codului folosind limbajul C++.