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**

**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.