Lattice Boltzmann Method

Bartłomiej Szwarga

Grudzień 2024

1 Zastosowane narzędzia

Do realizacji projektu LBM (Lattice Boltzmann Method) wykorzystano język programowania C++, który ze względu na swoją wydajność i szerokie zastosowanie w symulacjach numerycznych stanowi idealny wybór do implementacji algorytmów wymagających dużej mocy obliczeniowej.

Podobnie jak w projekcie LGA, do wizualizacji wyników symulacji użyto biblioteki SFML (Simple and Fast Multimedia Library), która umożliwia łatwe tworzenie interfejsów graficznych, obsługę zdarzeń oraz zarządzanie multimediami. Dodatkowo, w celu renderowania grafiki wykorzystano OpenGL, który zapewnia wsparcie dla zaawansowanych technik graficznych w czasie rzeczywistym. Dzięki OpenGL możliwe było efektywne przedstawienie dynamiki płynów modelowanych metodą LBM.

Rozwiązania te okazały się niezwykle użyteczne w realizacji projektu, pozwalając na płynne przeniesienie doświadczeń z projektu LGA na nową implementację. C++ zapewnił solidną platformę do przeprowadzania intensywnych obliczeń numerycznych, podczas gdy SFML i OpenGL umożliwiły stworzenie atrakcyjnych wizualizacji, pozwalających na analizę wyników symulacji w czasie rzeczywistym.

2 Opis i realizacja modelu

W projekcie zaimplementowano model przepływu oparty na metodzie Lattice Boltzmann (LBM), który jest rozwinięciem klasycznego modelu Lattice Gas Automaton (LGA). Model ten wykorzystuje dyskretne stany komórek oraz dwuwymiarową macierz *Matrix*, której elementami są obiekty klasy *Cell*. Każda komórka posiada funkcje rozkładu, które opisują zachowanie cząstek, oraz dodatkowe informacje dotyczące stanu komórki, takie jak gęstość cząstek czy waga jako parametr. Klasa *Simulation* jest odpowiedzialna za obliczenia i aktualizację stanów komórek, a także za synchronizowanie obydwu operacji w każdej iteracji symulacji.

2.1 Klasa Cell

Klasa *Cell* reprezentuje podstawową jednostkę przestrzeni symulacji w modelu Lattice Boltzmann. Każda komórka przechowuje kluczowe informacje umożliwiające opis przepływu cząstek.

Dane przechowywane w komórce obejmują trzy wektory funkcji rozkładu:

- **fun-in**: funkcje wejściową, opisującą transport masy do wewnątrz zdyskretyzowanej komórki z czterech kierunków w danym kroku czasowym.
- **fun-ex**: funkcje wyjściową z komórki, opisującą transport masy na zewnątrz zdyskretyzowanej komórki w czterech kierunkach w danym kroku czasowym.
- **fun-eq**: funkcje rozkładu w stanie równowagi, czyli stanu, który będzie przy niezmiennych warunkach przepływu

Ponadto każda komórka przechowuje:

- **density**: lokalną gęstość cząstek, obliczaną jako suma wartości w wektorze **fun-in**.
- **weight**: dodatkowy parametr używany w obliczeniach.

2.2 Klasa Matrix

Przestrzeń symulacji jest reprezentowana w postaci macierzy obiektów klasy *Cell*. Domyślnie wszystkie komórki w macierzy są inicjalizowane jako **EMPTY**, z zerowymi wartościami funkcji rozkładu. Przygotowanie środowiska symulacji odbywa się za pomocą metody *prepare-environment*, która pozwala na modyfikację wybranych komórek w celu odwzorowania warunków brzegowych, takich jak ściany (**WALL**) lub komórki początkowe (**STARTING-STATE**).

Na rysunku poniżej przedstawiono przykładowe środowisko symulacji z układem ścian ograniczających obszar przepływu.

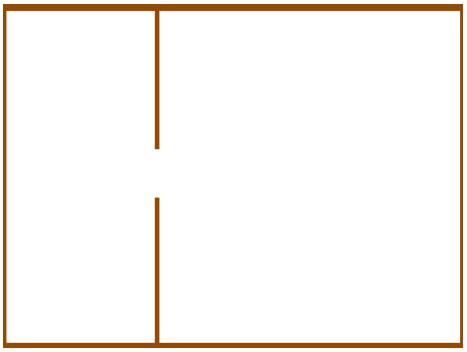

Figure 1: Środowisko symulacji

W stanie początkowym funkcje wejściowe każdej z komórek w których znajdują się cząsteczki inicjalizowane są równomiernie we wszystkich kierunkach wartością 0.25, co odzwierciedla stan równowagi w układzie. Dzięki temu możliwe jest zachowanie stabilności przepływu w początkowej fazie symulacji.

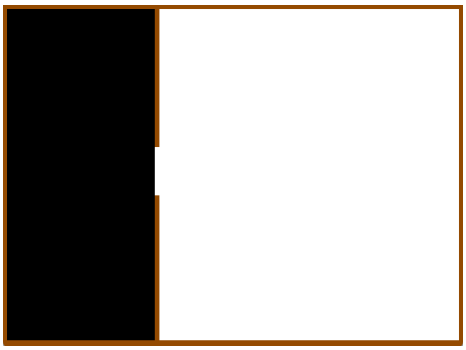

Figure 2: Stan początkowy

2.3 Klasa Simulation

Klasa *Simulation* jest odpowiedzialna za przeprowadzanie symulacji przepływu metodą Lattice Boltzmann (LBM). Proces symulacji składa się z dwóch głównych etapów: streamingu i kolizji. Oba etapy są wykonywane na nowej macierzy (kopia oryginalnej macierzy), co zapewnia spójność wyników w obrębie każdej iteracji. W poniższych podsekcjach przedstawiono szczegóły implementacji obu operacji.

2.3.1 Obsługa streamingu

Streaming odpowiada za przesunięcie cząsteczek między komórkami zgodnie z ich bieżącymi kierunkami ruchu. Każda cząsteczka przemieszcza się do sąsiedniej komórki w wyznaczonym kierunku, pod warunkiem, że nie jest to komórka typu **WALL**. Jeśli cząsteczka napotka ścianę, zostaje odbita, a jej kierunek ruchu ulega zmianie na przeciwny. Dzięki temu operacja streamingu odzwierciedla realistyczne zjawiska przemieszczania się cząsteczek w ograniczonej przestrzeni.

Metoda `streaming` aktualizuje funkcje wejściową komórki na podstawie kierunków funkcji wyjściowych komórek otaczających, przesuwając wartości do odpowiednich sąsiednich komórek. W poniższym fragmencie kodu przedstawiono realizację tej operacji:

```
void Simulation::streaming() {
    Matrix next_matrix = s;
    int rows = s.get_rows_num();
    int columns = s.get_columns_num();
    for (size_t i = 1; i < rows - 1; i++) {
        for (size_t j = 1; j < columns - 1; j++) {
            Cell& current_cell = s.get_element(i, j);
            array<double, 4> fun_ex = current_cell.get_fun(FUN_EX);

            if (fun_ex[0]) {
                Cell& right_cell = s.get_element(i, j + 1);
                if (right_cell.get_fun(FUN_IN) == WALL) {
                    next_matrix.get_element(i, j).set_direct_fun(FUN_IN, 2, fun_ex[0]);
                }
                else
                    next_matrix.get_element(i, j + 1).set_direct_fun(FUN_IN,0, fun_ex[0]);
            }
            if (fun_ex[1]) {
                Cell& down_cell = s.get_element(i + 1, j);
                if (down_cell.get_fun(FUN_IN) == WALL) {
                    next_matrix.get_element(i, j).set_direct_fun(FUN_IN, 3, fun_ex[1]);
                }
                else
                    next_matrix.get_element(i + 1, j).set_direct_fun(FUN_IN, 1, fun_ex[1]);
            }
            if (fun_ex[2]) {
                Cell& left_cell = s.get_element(i, j - 1);
                if (left_cell.get_fun(FUN_IN) == WALL) {
                    next_matrix.get_element(i, j).set_direct_fun(FUN_IN, 0, fun_ex[2]);
                }
                else
                    next_matrix.get_element(i, j - 1).set_direct_fun(FUN_IN, 2, fun_ex[2]);
            }
            if (fun_ex[3]) {
                Cell& up_cell = s.get_element(i - 1, j);
                if (up_cell.get_fun(FUN_IN) == WALL) {
                    next_matrix.get_element(i, j).set_direct_fun(FUN_IN, 1, fun_ex[3]);
                }
                else
                    next_matrix.get_element(i - 1, j).set_direct_fun(FUN_IN, 3, fun_ex[3]);
            }
        }
    }
    s = next_matrix;
}
```

2.3.2 Obsługa kolizji

Kolizja w metodzie LBM polega na modyfikacji funkcji rozkładu cząsteczek w komórkach w wyniku ich interakcji. W implementacji założono, że kolizja prowadzi do przekształcenia funkcji rozkładu, co jest realizowane przez relaksację do funkcji równowagi. Każda komórka w obrębie symulacji oblicza nową gęstość, a także wartości funkcji rozkładu w stanie równowagi i w wyniku kolizji.

W poniższym fragmencie kodu przedstawiono metodę `collision`, która dokonuje obliczeń kolizyjnych w każdej komórce:

```
void Simulation::collision() {
    Matrix next_matrix = s;
    int rows = s.get_rows_num();
    int columns = s.get_columns_num();
    for (size_t i = 1; i < rows - 1; i++) {
        for (size_t j = 1; j < columns - 1; j++) {
            Cell& next_cell = next_matrix.get_element(i, j);
            if (s.get_element(i, j).get_fun(FUN_IN) != WALL) {
                next_cell.calculate_density();
                next_cell.calculate_fun_eq();
                next_cell.calculate_fun_ex();
            }
        }
    }
    s = next_matrix;
}
```

3 Wyniki symulacji

W pierwszej fazie symulacji, jak przedstawiono na **Rysunku 3**, system początkowo znajduje się w stanie równowagi, gdzie cząsteczki rozłożone są równomiernie we wszystkich kierunkach. Jest to stan wyjściowy, w którym wszystkie funkcje rozkładu w komórkach są zainicjowane do wartości 0.25, 0.25, 0.25, 0.25.

Po przeprowadzeniu kilku iteracji symulacji, cząstki zaczynają się rozprzestrzeniać w kierunkach wyznaczonych przez funkcje rozkładu, co można zaobserwować na **Rysunku 4** i **Rysunku 5**. Widać, jak cząsteczki płynu przemieszczają się przez przestrzeń, rozprzestrzeniając się od obszaru początkowego.

Na **Rysunku 6** widzimy, że cząstki osiągnęły dalszy etap rozprzestrzeniania się, gdzie widać znaczące zmiany w rozkładzie cząsteczek oraz dalszą ich propagacje.

Na **Rysunku 7** przedstawiono stan końcowy, w którym cząstki osiągnęły stabilny rozkład w przestrzeni i są równomiernie rozproszone w kierunkach zgodnych z symulowanym przepływem.

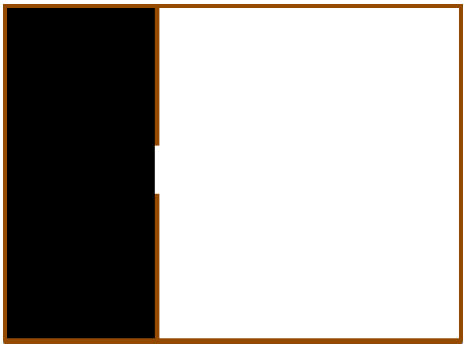

Figure 3: Stan początkowy

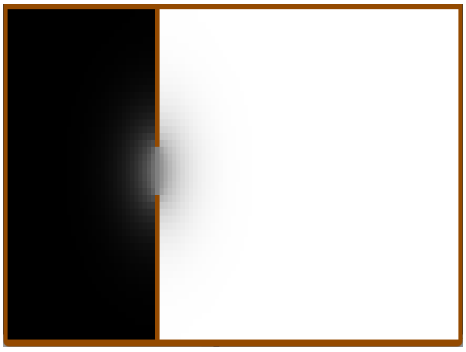

Figure 4: Rozprzestrzenianie się cząstek

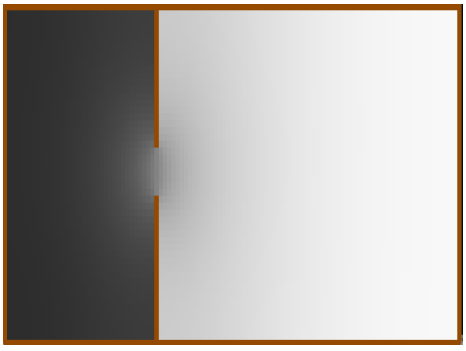

Figure 5: Dalsza propagacja

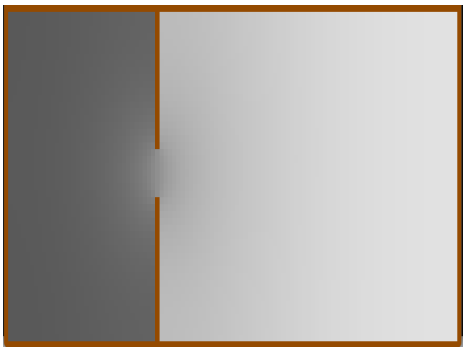

Figure 6: Końcowe rozprzestrzenianie się cząstek

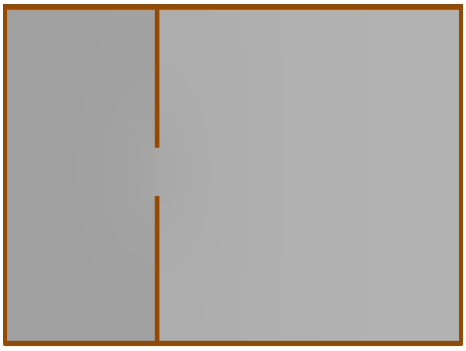

Figure 7: Stan końcowy