POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Automatyka i Robotyka
SPECJALNOŚĆ: Technologie informacyjne w systemach automatyki

PRACA DYPLOMOWA INŻYNIERSKA

Rozwiązywanie równań różniczkowych z zastosowaniem programowania równoległego

Solving differential equations with parallel programing

AUTOR:
Damian Jarek

PROWADZĄCY PRACĘ:
dr inż. Łukasz Jeleń, W4/K9

OCENA PRACY:

WROCŁAW 2015

Spis treści

1. Wstęp

Znaczna część powszechnie stosowanych do opisu zjawisk fizycznych modeli matematycznych zawiera równania różniczkowe, lub też układy równań różniczkowych. Niestety, w znacznej większości praktycznych przypadków nie jest możliwe znalezienie ich rozwiązań przy pomocy metod analitycznych, trzeba zatem w tym celu stosować metody przybliżone, numeryczne. Ogólnie, polegają one na przybliżeniu infinitezymalnych różnic, różnicami skończonymi, zaś granic ilorazami różnicowymi. Ponadto konieczna jest także często dyskretyzacja samej przestrzeni, w której symulowane jest dane zjawisko. Wielkości, które są polami (skalarnymi lub wektorowymi) nie dają się reprezentować w sposób dokładny w pamięci komputera, co wymusza zastosowanie metod takich jak metoda elementów skończonych, która polega na dyskretyzacji przestrzeni symulacji (zazwyczaj na proste elementy geometryczne, takie jak trójkąty czy kwadraty w przypadku problemów 2-wymiarowych lub czworościany czy sześciany w 3 wymiarach). W tej pracy jako przykład zjawiska użyty zostanie przepływ nieściśliwego płynu (w dwóch wymiarach) symulowany za pomocą metody stabilnych płynów opracowanej przez J. Stama („Stable Fluids", J. Stam, 1999). Kilka czynników przyczyniło się do wybrania tego zjawiska fizycznego. Po pierwsze, łatwość wizualizacji – można w tym celu użyć jednego pola skalarnego, które można łatwo zareprezentować na ekranie komputera, poprzez uzależnienie koloru danego piksela od gęstości barwnika unoszonego przez symulowany płyn. Po drugie, przepływ płynów opisany jest układem nieliniowych równań różniczkowych (równaniami Naviera-Stokesa), co powoduje, że problem jest wystarczająco złożony obliczeniowo, by miało sens stworzenie implementacji z użyciem technologii OpenCL. Ponadto, metoda J. Stama jest stabilna (nie licząc oczywiście sytuacji, gdy wartość liczonej wielkości przekroczy w którymś miejscu zakres użytego typu liczbowego) oraz została ona zaimplementowana na CPU, zatem zaimplementowanie jej dla urządzeń wspierających OpenCL stanowi ciekawe zadanie zarówno w celu porównania wydajności, ale także stopnia złożoności samej implementacji. Dodatkowo, założono, że symulacja odbywać się będzie w czasie rzeczywistym (oczywiście, narzuca to górną granicę rozmiaru symulacji, zależną od sprzętu, na którym uruchamiana jest symulacja).

Wraz ze wzrostem dostępności mocy obliczeniowej i spadkiem jej ceny, nastąpiło powiększenie się złożoności zjawisk, które mogą być symulowane. Niestety, dawno minęła już era wykładniczych wzrostów wydajności pojedynczych jednostek obliczeniowych (mowa tu m.in. o rdzeniach CPU). Wzrost wydajności zachodzi teraz już prawie wyłącznie poprzez zwiększenie liczby jednostek obliczeniowych oraz, w mniejszym stopniu, usprawnienia architektur i zestawów instrukcji urządzeń. Stwarza to znaczący problem przy próbie efektywnego wykorzystania pełni możliwości nowoczesnego sprzętu. Ponadto sytuacja jest jeszcze bardziej skomplikowana przez powstanie i rozpowszechnienie się technologii takich jak GPGPU (ang. General Purpose Graphics Processing Unit) czy Xeon Phi®.
Sytuacja ta wymaga od programisty niestandardowego podejścia i porzucenia głęboko utartego paradygmatu programowania równoległego i współbieżnego przez niskopoziomowe struktury takie jak wątki, muteksy, semafory czy zmienne warunkowe i skupieniu się na efektywnym podziale problemu na podproblemy, które wymagają niewielkich zasobów do rozwiązania. Z pomocą przychodzi OpenCL

(z ang. Open Computation Library), który w znacznym stopniu ukrywa złożony zestaw operacji, związanych z wydajnym kolejkowaniem zadań, za warstwą abstrakcji, która jednocześnie nie jest ograniczająca i pozwala na łatwe i efektywne wykorzystanie szerokiej gamy urządzeń obliczeniowych. Kod typowego programu napisanego z użyciem tej technologii składa się z dwóch zasadniczych części – hosta, czyli programu uruchomionego na CPU, którego zadaniem jest inicjalizacja urządzeń, kolejkowanie i szeregowanie zadań oraz z kerneli OpenCL, które wykonują zasadniczą część obliczeń. Kernele są po prostu funkcjami, które mają pewne ograniczenia, np. przed wersją 2.0 OpenCLa nie było możliwe kolejkowanie kerneli z poziomu kernela. Funkcja mogą być wywoływane z poziomu kerneli, jednakże muszą być to funkcje, które mogą zostać poddane operacji inline'owania przez kompilator OpenCla. OpenCL jest podobny pod wieloma względami do innego, znanego frameworka obliczeniowego OpenMP. Najbardziej znaczącą różnicą jest docelowa architektura – OpenMP stworzony został z myślą o klastrach obliczeniowych na CPU, zaś OpenCL na każdym kroku zdradza, to że powstał głównie jako technologia obliczeniowa dla GPGPU (szczególnie widać to w przestrzeniach adresowych, które reprezentują rejestry, cache i VRAM tego typu urządzeń), co narzuca ograniczenia, niespotykane w programach tworzonych przy pomocy technologii „tradycyjnych”, np. wspomniane wyżej przestrzenie adresowe oraz to, że pamięć w nich zawarta jest w ogólności dostępna tylko przez wskaźnik o odpowiednim atrybucie. Dodatkowo należy liczyć się z mało intuicyjnymi faktami związanymi z wydajnością, np. to, że instrukcje warunkowe (np. ***if***) mają bardzo niską wydajność na urządzeniach typu GPGPU (dlaczego tak jest zostanie wspomnianie w rozdziale poświęconym opisie OpenCLa). Oczywiście, nic nie stoi na przeszkodzie używaniu CPU do wykonywania obliczeń w OpenCLu, jednakże należy liczyć się z zdecydowanie niższą wydajnością w porównaniu do wykonywania na urządzeniach typu GPGPU.

2. Metoda stabilnych płynów

Płynem, w fizyce, nazywamy każdą substancję, która wykazuje zdolność do przepływu, tzn. nawet niewielka siła może spowodować względne przemieszczenie się elementów na niego się składających. Warto tutaj zaznaczyć, że występuje znaczna różnica pomiędzy potocznym znaczeniem słowa „płyn", a jego ścisłą definicją. W fizyce płyny to nie tylko ciecze, ale także gazy, plazma, i mieszaniny takie jak zawiesiny czy emulsje. Dynamika płynów opisana jest przez równania Naviera-Stokesa, które dla przepływu nieściśliwego płynu wyglądają następująco:

$$\frac{\delta u}{\delta t} = -(u \cdot \nabla)u - \frac{1}{\rho}\nabla p + \upsilon \nabla^2 u + f, \tag{1}$$

$$\nabla \cdot u = 0, \tag{2}$$

gdzie użyto operatorów o następujących definicjach:

$$\nabla \cdot u = \frac{\delta u_x}{\delta x} + \frac{\delta u_y}{\delta y}(dywergencja), \tag{3}$$

$$\nabla p = (\frac{\delta p}{\delta x}, \frac{\delta p}{\delta y})(gradient), \tag{4}$$

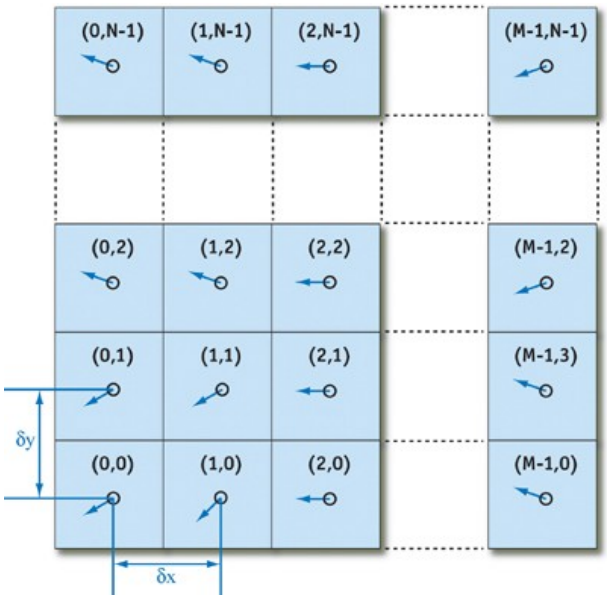
$$\nabla^2 u = \frac{\delta^2 u_x}{\delta x^2} + \frac{\delta^2 u_y}{\delta y^2}(laplasjan), \tag{5}$$

gdzie występują następujące wielkości:

u - pole wektorowe opisujące prędkość płynu,
p - pole skalarne opisujące ciśnienie płynu,
ρ - gęstość płynu,
ν - lepkość kinematyczna,
f - pole wektorowe opisujące przyspieszenia zewnętrzne.

Pierwsze równanie Naviera-Stokesa (1) opisuje zmianę pola wektorowego prędkości w czasie, zaś równanie (2) opisuje nieściśliwość płynu (innymi słowy, zachowanie masy). Pierwszym krokiem w stworzeniu modelu matematycznego zjawiska przepływu płynu, który da się zareprezentować w pamięci komputera, jest dyskretyzacja przestrzeni zjawiska. W tym wypadku przestrzeń jest 2-wymiarowa(aczkolwiek wprowadzenie 3. wymiaru nie jest z matematycznego punktu widzenia skomplikowane), zatem najwygodniej jest podzielić ją na prostokąty o szerokości δx i wysokości δy, zwane dalej komórkami. Wartość danej wielkości (np. prędkości przepływu płynu) jest określona w środku każdej komórki. Dla uproszczenia obliczeń, założyłem δx=δy. Całkowita liczba komórek symulacji to (M+2)x(N+2) (2 komórki w każdym z wymiarów to komórki brzegowe, dla których zamiast każdego z kroków symulacji, aplikowane są warunki brzegowe, które opisane

są niżej). Na rysunku 2.1 widoczna jest siatka komórek „wewnętrznych”symulacji, nie przedstawione są na nim komórki brzegowe.



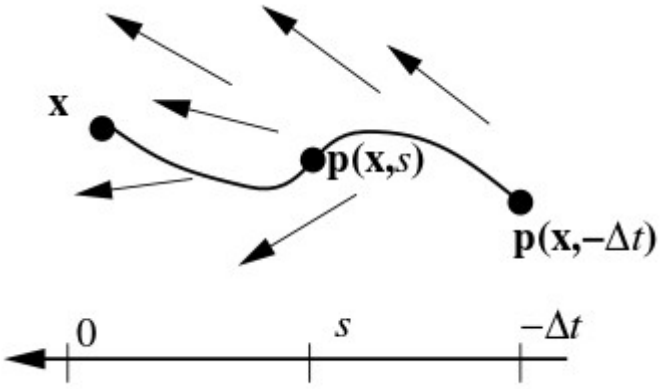
Rys. 2.1 Dyskretyzacja przestrzeni symulacji.[2]

W każdej chwili czasu δt aplikowane są następujące kroki symulacji:
1) dodanie przyspieszeń spowodowanych siłami zewnętrznymi,
2) adwekcja(unoszenie) prędkości,
3) dyfuzja prędkości,
4) zastosowanie operatora rzutowania,
5) aplikacja warunków brzegowych pola wektorowego u.

Oprócz tych podstawowych kroków, w mojej implementacji symulacji wykonywanych jest także kilka dodatkowych etapów. Ich opis znajduje się w rozdziale poświęconym implementacji. Najbardziej charakterystyczny dla metody J. Stama jest etap adwekcji, który wyróżnia ją wśród innych metod symulacji przepływu płynów. Jest on konieczny w celu poradzenia sobie z nieliniowością równania Naviera-Stokesa. Polega on na „śledzeniu” wstecz w czasie położenia cząstek płynu. Same cząstki nie są opisane explicite w żadnym z równań, ponieważ nie ma takiej konieczności – wynika to z zerowej dywergencji prędkości. Zapisanie równań Naviera-Stokesa w postaci opisującej unoszenie cząstek cieczy, prowadzi do wniosku, że równania opisujące unoszenie prędkości i cząstek są ekwiwalentne w przypadku płynu nieściśliwego. Adwekcja używana jest także do opisu zmian pola skalarnego reprezentującego barwnik unoszony przez przepływ płynu. Adwekcja wielkości q w punkcie (x, y) dla kroku czasowego δt opisana jest następującym równaniem:

$$q(x, y, t+\delta t) = q((x, y) - u(x, y, t)\delta t, t). \tag{6}$$

Wielkością poddawaną adwekcji przy zastosowywaniu tej operacji do cząstek płynu jest u (q=u). Rys. 2.2 ilustruje przebieg adwekcji w czasie.



Rys. 2.2 Ilustracja adwekcji.[1]

Należy też wspomnieć, że obliczony w adwekcji punkt z poprzedniej chwili czasowej nie koniecznie będzie mieć wartość całkowitoliczbową, zatem trzeba dokonać interpolacji unoszonej wielkości w wyznaczonym punkcie. W tej pracy wykorzystałem metodę interpolacji bilinearnej, ze względu na jej niskie wymagania obliczeniowe. Polega ona zastosowaniu interpolacji linearnej 3-krotnie – 2-krotnie w kierunku x oraz raz w kierunku y. Zakładając, że dane są punky (x_1, y_2) i (x_1, y_2) interpolacja linearna dana jest następującym wzorem:

$$y = lininterpol(y_{1,} y_{2,} t_x) = y_1 + (y_2 - y_1) t_x, \tag{7}$$

$$t_x = \frac{x - x_1}{x_2 - x_1}. \tag{8}$$

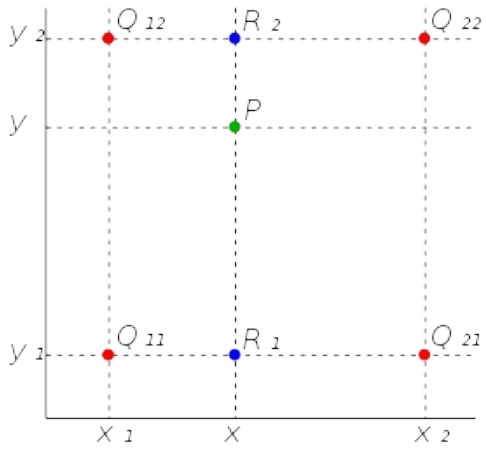
Interpolacja bilinearna jest, tak jak wspomniane to zostało wyżej, złożeniem interpolacji linearnej wzdłuż osi x i osi y, zatem jest ona określona w następujący sposób:

$$\begin{aligned} u_{11} &= u(x_{1,} y_1), \\ u_{12} &= u(x_{1,} y_2), \\ u_{21} &= u(x_{2,} y_1), \\ u_{22} &= u(x_{2,} y_2), \end{aligned} \tag{9}$$

$$t_y = \frac{y - y_1}{y_2 - y_1}, \tag{10}$$

$$bilinterpol(u_{11,}u_{12,}u_{21,}u_{22,}t_x,t_y)= \\ linterpol(lininterpol(u_{11,}u_{12,}t_x),lininterpol(u_{21,}u_{22,}t_x),t_y), \tag{11}$$

gdzie u to interpolowana wielkość, a u_{11}, u_{12}, u_{21}, u_{22} to znane wartości. W tej pracy są to wartości sąsiednich komórek symulacji. Znane wartości i punkt dla którego wykonywana jest interpolacja przedstawione zostały na rysunku 2.3.



Rys. 2.3 Interpolacja bilinearna.[8]

Ponadto należy pamiętać, że jest możliwe wystąpienie takiej wartości wektora prędkości, której prześledzenie w tył w czasie dałoby w rezultacie położenie poza obszarem symulacji, zatem konieczne jest, przed zastosowaniem interpolacji znormalizowanie otrzymanych współrzędnych tak, by mieściły się w obszarze symulacji. Zasadniczą zaletą tej metody modelowania adwekcji jest stabilność – wybranie zbyt dużej wartości kroku czasowego symulacji δt nie powoduje niestabilności numerycznej. Ponadto, ewolucja systemu zasadniczo opisana jest dzięki temu jednym równaniem różniczkowym, nie licząc równania opisującego zachowanie masy, gdyż stosowane jest ono implicite podczas kroku reprezentującego operator rzutowania, który jest drugą charakterystyczną operacją dla metody J. Stama. Jest on ważny, gdyż wynikiem adwekcji jest pole wektorowe w, które ma niezerową dywergencję (nie zachowuje masy). Aby zdefiniować ten operator, należy najpierw przywołać twierdzenie Helmholtza, znane z analizy wektorowej. Mówi ono, że każde dwukrotnie różniczkowalne i ograniczone pole wektorowe może zostać rozłożone na pole o zerowej dywergencji i gradient pola skalarnego o zerowej rotacji, co opisuje poniższe równanie:

$$w = u + \nabla p. \tag{12}$$

Dowód poprawności tego rozkładu pola wektorowego w znajduje się w „A Mathematical Introduction to Fluid Mechanics” Chorin, Marsden. Wyznaczenie u z tego równania opisuje sposób uzyskania ostatecznej wartości prędkości płynu po zastosowaniu operatorów sił,adwekcji i dyfuzji. Jednakże niezbędne jest w tym celu

wyznaczenie pola skalarnego p, opisującego ciśnienie, wywołane prędkością przepływu płynu w. Po obustronnym zastosowaniu operatora dywergencji do powyższego równania otrzymujemy:

$$\nabla \cdot w = \nabla \cdot u + \nabla \cdot \nabla p. \tag{13}$$

Następnie korzystamy z równania zachowania masy (2), by otrzymać:

$$\nabla \cdot w = \nabla^2 p. \tag{14}$$

Otrzymane równanie różniczkowe nazywane jest równaniem Poissona. Istnieje wiele metod numerycznego rozwiązywania tego typu równań. Zostanie to opisane w dalszej części pracy. Wspomniany wcześniej operator rzutowania P, rzutuje pole wektorowe w na jego składnik u, który ma zerową dywergencję. Operator ten definiują następujące równania:

$$P(w) = u, \tag{15}$$

$$P(u) = u. \tag{16}$$

Po obustronnym zastosowaniu tego operatora do 1. równania Naviera-Stokesa otrzymamy:

$$P\left(\frac{\delta u}{\delta t}\right) = P\left(-(u \cdot \nabla)u - \frac{1}{\rho}\nabla p + \upsilon \nabla^2 u + f\right), \tag{17}$$

$$P(\nabla p) = 0, \tag{18}$$

$$\frac{\delta u}{\delta t} = P\left(-(u \cdot \nabla)u + \upsilon \nabla^2 u + f\right), \tag{19}$$

$$\frac{\delta u}{\delta t} = P\left(\frac{\delta w_{adwekcji}}{\delta t} + \frac{\delta w_{dyfuzji}}{\delta t} + f\right). \tag{20}$$

Ostatnie równanie opisuje w jaki sposób można podzielić każdy krok czasowy symulacji, a dokładniej mówi nam, że możemy rozdzielić wyznaczanie wartości w i p. W przedstawionej metodzie symulacji przepływu płynu występują 2 równania Poissona – wspomniane wcześniej równanie służące do wyznaczenia ciśnienia, oraz równanie dyfuzji prędkości:

$$\frac{\delta w_{dyfuzji}}{\delta t} = \upsilon \nabla^2 u, \tag{21}$$

gdzie $w_{dyfuzji}$ to składnik pola wektorowego w opisujący dyfuzję prędkości, czyli efekt spowodowany zjawiskiem lepkości cieczy. Rozwiązywanie tego równania w tej

postaci powoduje dosyć dużą niestabilność, przy dużych wartościach lepkości. J. Stam zaproponował rozwiązywanie tego równania w następującej postaci, w celu zapewnienia stabilności:

$$(I-\upsilon\delta t\nabla^2)u=\delta w_{dyfuzji}. \tag{22}$$

Istnieje wiele metod numerycznego rozwiązywania równań Poissona. Ze względów praktycznych, opisanych w rozdziale dotyczącym implementacji, zdecydowałem się użyć w tej pracy metody Jacobiego. Jest to metoda iteracyjna o dosyć powolnej zbieżności. W celu zapisania równania opisującego tą metodę rozwiązywania równań Poissona, należy najpierw podać definicję zdyskretyzowanego laplasjanu (w postaci różnic skończonych). Wyprowadzenie dla punktu (i, j) wygląda następująco:

$$\nabla p=(\frac{p_{i+1,j}-p_{i-1,j}}{2\delta x},\frac{p_{i,j+1}-p_{i,j-1}}{2\delta y}), \tag{23}$$

$$\nabla\cdot w=\frac{w_{i+1,j}-w_{i-1,j}}{2\delta x}+\frac{w_{i,j+1}-w_{i,j-1}}{2\delta y}, \tag{24}$$

$$\nabla^2 p=\nabla\cdot\nabla, \tag{25}$$

$$\nabla^2 p=\frac{p_{i+1,j}+p_{i-1,j}-2p_{i,j}}{(\delta x)^2}+\frac{p_{i,j+1}+p_{i,j-1}-2p_{i,j}}{(\delta y)^2}, \tag{26}$$

$$\delta x=\delta y, \tag{27}$$

$$\nabla^2 p=\frac{p_{i+1,j}+p_{i-1,j}+p_{i,j+1}p_{i,j-1}-4p_{i,j}}{(\delta x)^2}, \tag{28}$$

Korzystając z równania (21), możemy sformułować ogólne równanie opisujące rozwiązywanie równań Poissona za pomocą metody Jacobiego:

$$x_{i,j}^{k+1}=\frac{x_{i+1,j}^{k}+x_{i-1,j}^{k}+x_{i,j+1}^{k}x_{i,j-1}^{k}-\alpha b_{i,j}}{\beta}, \tag{29}$$

gdzie k to numer iteracji metody, a x^k to wartość x w k-tej iteracji. Metody Jacobiego. Dla równania opisującego ciśnienie x, b, α, β mają następujące wartości:

$$\begin{gathered} x=p,\\ b=\nabla\cdot w,\\ \alpha=-(\delta x)^2,\\ \beta=4. \end{gathered} \tag{30}$$

Natomiast w równaniu opisującym dyfuzję prędkości dane są w następujący sposób:

$$\begin{aligned} x &= b = u, \\ \alpha &= \frac{(\delta x)^2}{\upsilon \delta t}, \\ \beta &= 4 + \alpha. \end{aligned} \tag{31}$$

Zdefiniowanie metody Jacobiego w ten sposób (tzn. z użyciem współczynników) ułatwia późniejsze zaimplementowanie jej dla obydwu równań Poissona. W zerowej iteracji wyznaczana wielkość inicjalizowana jest zerami, by zmniejszyć ilość iteracji niezbędnych do osiągnięcia zbieżności. Najprostszym etapem symulacji jest dodanie przyspieszeń spowodowanymi siłami. Wspomniane wcześniej pole wektorowe reprezentujące przyspieszenia zewnętrzne wygląda następująco:

$$f = F \frac{\delta t}{\delta m}. \tag{32}$$

Zakładamy, że siła zewnętrzna F nie zmienia się w sposób znaczący w ramach jednego kroku czasowego symulacji. δm reprezenuje masę elementu(cząstki) płynu.
Ostatnią niezbędną operacją symulacji jest aplikacja warunków brzegowych. Zakładamy, że płyn zawarty jest w prostokątnym pojemniku i nie może wypływać poza jego obszar. Ponadto, poprawne rozwiązanie równania opisującego ciśnienie wymaga zerowej pochodnej ciśnienia na granicy przestrzeni symulacji. Przykładowo, dla lewej granicy obszaru symulacji równania opisujące warunki brzegowe wyglądają następująco:

$$j \in [0, N], \tag{33}$$

$$\frac{u_{0,j} + u_{1,j}}{2} = 0, \tag{34}$$

$$\frac{p_{0,j} - p_{1,j}}{2} = 0, \tag{35}$$

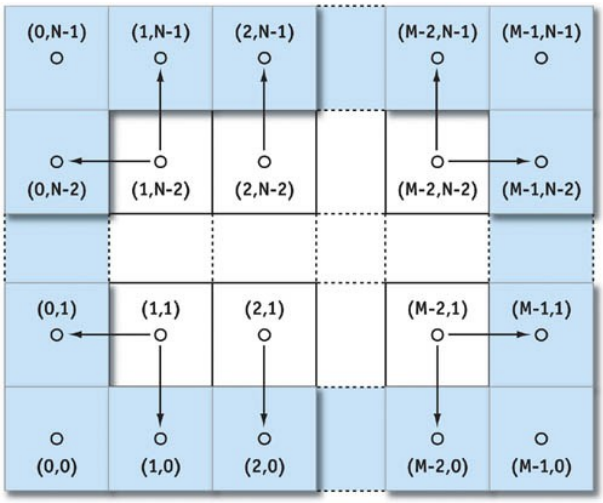
co można uprościć do postaci:

$$u_{0,j} = -u_{1,j}, \tag{36}$$

$$p_{0,j} = p_{1,j}. \tag{37}$$

Warto zauważyć, że warunek brzegowy dla prędkości pozwala nam otrzymać poprawne zachowanie symulowanej cieczy, gdy pozycja z interpolacji bilinearnej wykonywanej w kroku adwekcji znajduje się poza obszarem symulacji i musi być do niego znormalizowana. Innymi słowy mówiąc, dzięki warunkom brzegowym ciecz „odbija” się od granicy przestrzeni symulacji, co jest zgodne z założeniami

początkowymi co do zachowania się cieczy w „pojemniku”. Powyższe równanie zapisane zostały dla lewej granicy obszaru symulacji – używają one offsetu współrzędnych x o wartości 1 i dla współrzędnej y wartości 0. Offset ten opisuje, która komórka „wewnętrzna” symulacji jest źródłem wartości p i u dla danej komórki brzegowej. Najprościej wyjaśnić te offsety współrzędnych na schemacie, widocznym na rysunku 2.4. Na niebiesko zaznaczone zostały komórki brzegowe. Strzałki oznaczają relację komórka „źródłowa” i „docelowa”.



Rys. 2.4 Relacje między wartościami komórek wewnętrznych i brzegowych.[2]

3. Rozszerzenia metody stabilnych płynów.

Metodę symulacji przepływu płynu opisaną w poprzednim rozdziale można poszerzyć o kilka dodatkowych kroków umożliwiających wizualizację przepływu i obserwację zjawisk, których nie można zaobserwować w podstawowej wersji. Pierwszym takim rozszerzeniem jest adwekcja dodatkowego pola skalarnego d (z ang. *dye)*, które reprezentuje gęstość barwnika unoszonego przez symulowany płyn. Warto zaznaczyć, że zakładamy, że ów barwnik ma niewielką (pomijalną) masę i przez to zerowy wpływ na przepływ, tzn. jest jedynie przez niego bezwładnie unoszony. W ogólności, zachowanie takiego barwnika opisuje następujące równanie:

$$\frac{\delta d}{\delta t} = -(u \cdot \nabla) d + \gamma \nabla^2 d + S. \tag{38}$$

Jak widać, barwnik podlega zjawiskom adwekcji i dyfuzji. S oznacza rozproszenie (zanik) barwnika, zazwyczaj zakłada się, że rozproszenie jest stałe w czasie. Adwekcję barwnika można zdyskretyzować w sposób identyczny jak adwekcję prędkości w poprzednim rozdziale, co przedstawia poniższe równanie:

$$d(x, y, t + \delta t) = d((x, y) - u(x, y, t)\delta t, t). \tag{39}$$

Natomiast dyfuzję barwnika wyznaczamy analogicznie do dyfuzji prędkości, rozwiązując równanie Poissona metodą Jacobiego.
Kolejnym rozszerzeniem, które można dodać do symulacji jest impuls(siła) o rozkładzie gaussowskim. Dzięki niemu można osiągnąć efekt „przeciągnięcia" podłużnego obiektu w cieczy, co szczególnie przydaje się przy implementowaniu części interaktywnej symulacji. Impuls ten opisany jest następującym równaniem:

$$f = F \delta t \exp\left(\frac{(x - x_p)^2 + (y - y_p)^2}{r^2}\right), \tag{40}$$

gdzie F to stały współczynnik wektorowy określający kierunek i wartość siły w punkcie przyłożenia omawianego impulsu (x_p, y_p). r określa zasięg impulsu.
Dyskretyzacja przepływu płynu powoduje tłumienie (numeryczne) wielkości zwanej wirowością, określającą tendencję płynu do „skręcania". Utraconą wirowość można odzyskać, poprzez zastosowanie techniki zwanej „vorticity confinement". Polega ona na wstępnym obliczeniu wirowości płynu z jej definicji:

$$\omega = \nabla \times u. \tag{41}$$

Następnie obliczamy znormalizowane pole wektorowe Ψ na podstawie gradientu wirowości:

$$\eta = \nabla |\omega|. \tag{42}$$

$$\Psi = \frac{\eta}{|\eta|}, \tag{43}$$

Korzystając z równań (41) i (43) możemy wyznaczyć przyspieszenia, które należy dodać do f, aby odzyskać stłumione cechy wirowości płynu:

$$f_{vc} = \epsilon(\Psi \times \omega)\delta t\,. \tag{44}$$

Ostatnim rozszerzeniem, które zostanie opisane w tej pracy to dodanie wpływu temperatury i grawitacji. Zjawiska te należy uwzględnić przede wszystkim podczas symulacji dynamiki dymu w gazie. Siła wywołująca zjawisko konwekcji opisana jest następująco:

$$f_{konwekcji} = \sigma(T - T_o)\hat{j}, \tag{45}$$

gdzie T_o to dana temperatura otoczenia, T to temperatura elementu płynu, sigma to stała określająca wpływ temperatury na wartość temperatury a j to wersor w kiernku y. Zakładam tutaj, że różnica temperatur nie ma znaczącego wpływu na gęstość płynu, co dla cieczy i kilku kelwinów różnicy nie odbiega znacząco od rzeczywistości.
Natomiast siła wyporu, działająca na unoszoną substancję, określona jest w sposób następujący:

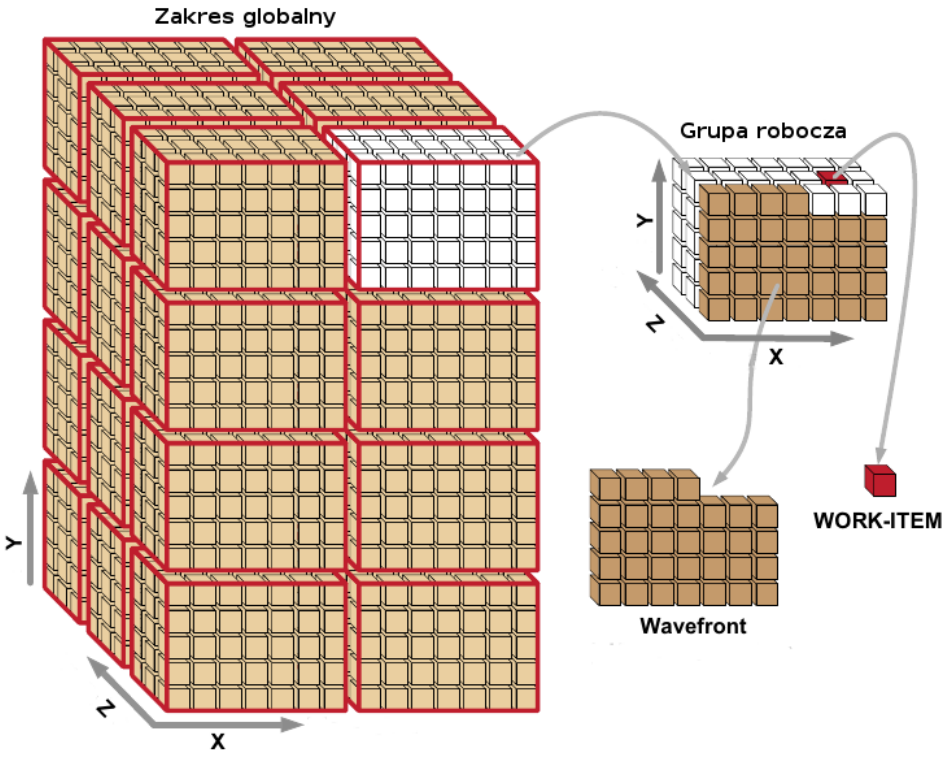
$$f_{wyporu} = -\kappa d\hat{j}, \tag{46}$$

gdzie d to gęstość unoszonej substancji, a κ współczynnik proporcjonalności uzależniający przyspieszenie od gęstości.

4. Open Computation Library (OpenCL) – model obliczeniowy i model pamięci

Przy pomocy aparatu matematycznego, zaprezentowanego w poprzednich rozdziałach, można dokonać implementacji interaktywnej symulacji przepływu płynu w dwóch wymiarach. Najpierw, chciałbym jednak pokrótce nieco dokładniej przedstawić wspomnianą na wstępie bibliotekę i technologię obliczeniową OpenCL. W wersji 2.0 standardu OpenCL część wykonywaną po stronie urządzeń obliczeniowych implementuje się w rozszerzeniu języka C99, natomiast kod programu gospodarza (ang. *host*) napisany jest w C bądź C++ (dla tego drugiego zaimplementowane zostały wrappery, znacząco ułatwiające ich użycie, opisane w pozycji [5]). Warto wspomnieć, że program-gospodarz może być napisany w dowolnym języku, który posiada interfejs umożliwiający wywoływanie kodu z zewnętrznych bibliotek stosujących ABI(ang. Application Binary Interface) C, czyli w praktyce dotyczy to większości powszechnie stosowanych języków programowania. Podstawową jednostką obliczeniową, którą posługuje się OpenCL przy kolejkowaniu i szeregowaniu zadań jest tzw. work-item. Work-item to instancja kernela obliczeniowego, dla danych jego argumentów. Kernele to funkcje, oznaczone atrybutem ***kernel*** lub ***__kernel,*** które mogą zostać zakolejkowane w kolejce poleceń (ang. *command queue*) i wykonane na danym urządzeniu. Kolejka poleceń przypisana jest do jednego z dostępnych urządzeń obliczeniowych. Ważną cechą kerneli jest to, że są „czystymi" funkcjami (ang. *pure functions*). Jest to pojęcie znane przede wszystkim z języków funkcyjnych, mówiące o tym, że funkcja nie modyfikuje żadnego współdzielonego, globalnego stanu, a jedyne miejsca w pamięci, które mogą być modyfikowane z widocznymi efektami ubocznymi są przekazane do niego wskaźniki przekazane przez argumenty. Ponadto, trzeba też wspomnieć, że dynamiczna alokacja pamięci poprzez funkcje analogiczne do ***malloc()***, czy operatora ***new*** w C++ nie jest możliwa z poziomu kernela. Oczywiście, jest możliwa alokacja zmiennych automatycznych, które automatycznie mają kwalifikator przestrzeni adresowej ***private***, czyli modyfikacja tych zmiennych jest widoczna jedynie w ramach work-itema, w którym zostały zadeklarowane. Co prawda standard tego nie określa, ale zazwyczaj zmienne te zapisane są w rejestrach dostępnych dla każdej jednostki obliczeniowej. W odróżnieniu od typowych programów w C, nie ma funkcji ***main()*** będącej punktem wejściowym. Warto też wspomnieć, że urządzenia, na których zazwyczaj wykonywane są kernele (GPGPU) mają nietypową architekturę wielowątkową, często nazywaną SIMD (ang. Single Instruction Multiple Data), tzn. w ramach jednej grupy roboczej, wszystkie jednostki obliczeniowe wykonują tę samą instrukcję programu w ramach jednego cyklu. Gdy napotkana jest instrukcja warunkowa (np. if), wszystkie możliwe gałęzie kodu w ramach wykonywanego bloku instrukcji są wykonywane, a potem, w zależności od prawdziwości lub nieprawdziwości warunku wyniki z „fałszywych" gałęzi są odrzucane w pojedynczych jednostkach obliczeniowych. Znacząco różni się to od architektur stosowanych w CPU, np. SMT (ang. Simultaneous Multi Threading), gdzie wiele wątków wykonywanych jest niezależnie od siebie na kilku jednostkach obliczeniowych. Na GPU jeden „wątek" wykonywany jest, dla różnych paremetrów wejściowych na wielu, jednostkach obliczeniowych. Dla przykładu, na urządzeniu AMD Radeon™ HD 290X możliwe jest jednoczesne wykonywanie 112 640 work-itemów (wątków). Na podstawie tego stwierdzenia łatwo można wyciągnąć wniosek, że kernele powinny być jak najmniejsze (pod względem ilości instrukcji) oraz zawierać możliwie jak najmniej instrukcji warunkowych. Stosowanie pętli w

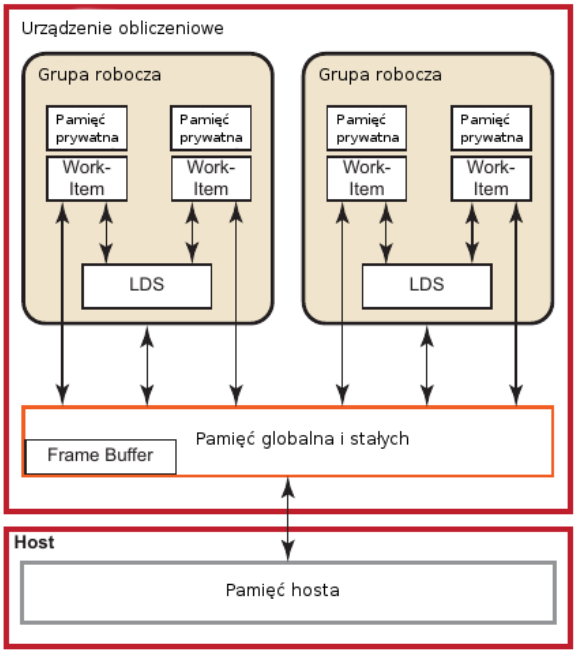
kernelach powszechnie uważane jest za złą praktykę, nie tylko dlatego, że pętle zawierają instrukcje warunkowe, ale także dlatego, że zbyt długi czas wykonywania pojedynczego work-itema może spowodować reset GPU, ponieważ GPU zawierają zazwyczaj sprzętowy watchdog, który resetuje urządzenie, jeśli nie odpowiada ono przez jakiś czas. Stosowany jest on przede wszystkim dlatego, że GPU nie posiadają mechanizmu wywłaszczeń, zatem nie jest możliwe przerwanie działania kernela i zakolejkowanie innego. Ponadto, trzeba pamiętać, że często GPU jest współdzielone przez programy OpenCL i aplikacje korzystające z GPU do renderowania grafiki i wyświetlania obrazu. Dlatego też największą wydajność można osiągnąć, jeśli zastosowany jest podział jeden work-item – jedna podstawowa operacja dla jednej komórki symulacji. Wspomniana wcześniej grupa robocza (ang. *work group*) jest podstawową jednostką, za pomocą której grupuje się work-itemy. Przy kolejkowaniu grupy roboczej stosuje się tzw. NDRange (od ang. N-Dimensional Range), które określają zakresy w N wymiarach, gdzie N to 1,2 lub 3. Rozmiar grupy roboczej w danym wymiarze musi być potęgą liczby 2 i zawiera się w przedziale od 1 do 256.
Podział całkowitej liczby work-itemów na grupy robocze może zostać wykonany automatycznie przez OpenCL lub „ręcznie” przez programistę. Automatyczny podział ma zasadniczą wadę, mianowicie używa największego rozmiaru grupy roboczej, który dzieli całkowitą liczbę work-itemów w danym wymiarze. Zatem, dla 512 work-itemów rozmiar grupy roboczej będzie wynosić 256, ale dla 511 będzie on wynosić 1! Kolejkowanie kerneli ma duży narzut, zatem jest znaczna różnica między zakolejkowaniem 2 grup roboczych o rozmiarze 256, a 511 grupy roboczych o rozmiarze 1 (dla tego drugiego przypadku czas kolejkowania zadań, najprawdopodobniej, znacząco przewyższy czas wykonywania work-itemów). Dlatego też dla większości zadań konieczny jest ręczny podział zadań i ich kolejkowanie by uniknąć dużego stosunku czasu bezczynności urządzenia obliczeniowego do czasu wykonywania użytecznych obliczeń. Warto wspomnieć, że maksymalny rozmiar grupy roboczej jest zależny od danego urządzenia. Dla większości GPU jest to wartość 256, jednakże poprawnie napisany program-gospodarz powinien odpytać urządzenie o wspierany maksymalny rozmiar grupy roboczej. Zależność między NDRange, a grupami roboczymi przedstawia rysunek 4.1. Warto tutaj zauważyć, że wymiary NDRange są od siebie niezależne, np. jeśli problem jest 2 wymiarowy, jeden z wymiarów będzie mieć rozmiar o wartości 1. Na schemacie występuje też jednostka zwana *wavefront*, która określa ile work-itemów jest jednocześnie przetwarzanych w ramach jednej grupy roboczej. Rozmiar tej jednostki jest oczywiście całkowicie zależny od sprzętu. Wspomniany został wcześniej fakt, że OpenCL nie pozwala na alokację pamięci wewnątrz kernela. Wszelkie alokacje pamięci muszą zostać wykonane po stronie gospodarza przed zakolejkowaniem kernela. Zasadniczo istnieją 2 typy obiektów, które mogą zostać zaalokowane na urządzeniu – bufory i obrazy N wymiarowe.



Rys. 4.1 Zależność między NDRange, a grupami roboczymi.[3]

Bufory to typowe znane z C tablice, jednakże warto wspomnieć, że nie jest możliwe zaalokowanie tablicy wielowymiarowej ze względu na sposób działania mechanizmów adresowania GPU. Ponadto tego typu obiekty wykorzystywałyby bardzo nieefektywnie mechanizmy cache'ujące GPU, gdyby możliwe było ich utworzenie. Dlatego zazwyczaj stosuje się obrazy, gdy potrzebne są odpowiedniki wielowymiarowych tablic. Oprócz uproszczonego dostępu do wielowymiarowych przestrzeni, obrazy oferują także adresowanie zmiennoprzecinkowe i automatyczną interpolację linearną adresu (indeksy w obrazach zazwyczaj mają wartości należące do przedziału [0, 1] (należy pamiętać przy ich użyciu o odpowiednim przeskalowywaniu i normalizacji). Można także zapisać wielowymiarową tablicę w płaskim buforze i odpowiednio liczyć indeksy, jednakże ta metoda jest potencjalnie mniej wydajna niż stosowanie obrazów. Niestety, nie wszystkie urządzenia obliczeniowe wspierają obrazy, zatem, jeśli wymagana jest przenośność, niezbędne jest użycie buforów. Bufory mogą być tylko do odczytu, tylko do zapisu, a także zarówno do zapisu jak i odczytu. Te pierwsze umieszcza się w przestrzeni adresowej dla stałych (jej kwalifikatorem w języku OpenCL C jest ***constant***), której na większości urządzeń typu GPU jest około 64 KiB, natomiast pozostałe w przestrzeni globalnej dostępnej dla każdego kernela wykonywanego w danym momencie na danym urządzeniu (ta przestrzeń adresowa oznaczona jest kwalifikatorem ***global***). Nie wspomniana została jeszcze przestrzeń adresowa dzielona między kernelami należącymi do tej samej grupy roboczej, nazywana przestrzenią lokalną i oznaczoną kwalifikatorem ***local***. Jej głównym zastosowaniem jest synchronizacja między kernelami w tej samej grupie roboczej przy pomocy operacji atomowych. Z technicznego punktu widzenia istnieje jeszcze jedna przestrzeń adresowa – przestrzeń należąca do hosta. Należy jednak pamiętać o tym, że przestrzenie adresowe gospodarza i urządzeń obliczeniowych są

rozdzielne – nie jest możliwe przekazanie wskaźnika do kernela w celu uniknięcia kopiowania, ani w ogóle bezpośredni dostęp do pamięci Kopiowanie jest zatem niezbędne, niestety, jest ono względnie powolne, gdyż odbywa się przez magistralę, do której podłączone jest urządzenie obliczeniowe, w przypadku kart graficznych jest to magistrala PCI-Express. Należy więc starać się zminimalizować ilość wykonywanych transferów. Co prawda jest możliwość użycia technologii zwanej DMA (ang. *Direct Memory Access* – Bezpośredni Dostęp do Pamięci), jednakże jej użycie narzuca na programistę pewne ograniczenia. Przede wszystkim, i host i urządzenie obliczeniowe muszą wspierać DMA (zarówno sprzętowo jak i muszą posiadać wsparcie w sterownikach). Po drugie, po stronie hosta pamięć użyta do DMA musi znajdować się w tzw. stronie przypiętej (ang. *locked page*), co powoduje, że napisanie przenośnego kodu z użyciem tej technologii jest niezwykle trudne. Jednakże nawet użycie DMA nie likwiduje problemu związanego z koniecznością transferu danych przez magistralę PCI-Express. Oczywiście trzeba wziąć pod uwagę ilość danych, które musi zostać przetransferowane przez magistralę PCI-E w danej jednostce czasu. Mimo, że ta magistrala jest wolna w porównaniu do zdolności nowoczesnych urządzeń typu GPGPU do generowania danych, to i tak oferuje ona przepustowość wynoszącą do 16 GiB/s dla PCI-E 3.0 i 16 linii. Przedstawione informacje na temat work-itemów, grup roboczych i podziału przestrzeni adresowych i pamięci bardzo dobrze podsumowuje schemat przedstawiony na rysunku 4.2.



Rys. 4.2 Uproszczony schemat urządzenia obliczeniowego typu GPGPU.[3]

Nie wspomniany został LDS widoczny na Rys. 4.2. LDS (ang. Local Data Store) to obszar pamięci dostępny tylko w ramach jednej jednostki obliczeniowej. Alokacja części LDS występuje dla grup roboczych. Jeśli wykonywany kernel dokona odczytu lub zapisu poza zaalokowanym obszarem LDS, nie jest określony wynik tej operacji (gdyż może nastąpić wyścig danych). Pamięć zaalokowana w LDS może służyć jako część mechanizmu synchronizacji między work-itemami lub jako lokalny, bardzo

szybki cache pamięci. Schemat widoczny na rysunku 4.2 wskazuje też na ważny aspekt kolejkowania zadań – dane urządzenie obliczeniowe może być w stanie jednocześnie przetwarzać work-itemy z różnych grup roboczych, co oznacza, że dla takich urządzeń kolejki poleceń są asynchroniczne – nie jest określona kolejność wykonania zakolejkowanych grup roboczych. Niezbędne są zatem mechanizm synchronizacji między grupami roboczymi. Niestety nie jest to możliwe z poziomu samych kerneli obliczeniowych(tak to zostało wspomniane wcześniej, możliwa jest synchronizacja między work-itemami w ramach jednej grupy roboczej). OpenCL udostepnia 2 mechanizmy synchronizacji wykonywania grup roboczych i innych operacji– bariery i listy wydarzeń (ang. *events lists*). Bariery działają intuicyjnie – powodują, że wszystkie zakolejkowane operacje, takie jak wykonanie kernela czy kopiowanie pamięci między przestrzenią adresową hosta i urządzenia, zostaną wykonane przed operacjami znajdującymi się za barierą w kolejce poleceń urządzenia. Natomiast listy wydarzeń pozwalają na bardziej precyzyjne określenie zależności pomiędzy poleceniami w kolejce urządzenia poprzez podanie uchwytów do zakolejkowanych już wydarzeń, od których zależne jest rozpoczęcie wykonywania kolejkowanego polecenia. Wydarzeniem w wypadku OpenCL jest zakończenie wykonywania polecenia. Wszystkie polecenia, które mogą zostać zakolejkowane w kolejce urządzenia OpenCL posiadają dodatkowy (opcjonalny) parametr umożliwiający przekazanie opisanej listy dependencji. Istnieje jeszcze jeden, niewspomniany mechanizm synchronizacji między grupami roboczymi – tzw. *„command queue flush”* - polegający na wstrzymaniu wykonywania obecnego wątku gospodarza dopóki nie zostaną zakończone wszystkie zadania w kolejce. Nie jest on zbyt często stosowany, gdyż bardzo łatwe jest „zagłodzenie” urządzenia obliczeniowego, czyli doprowadzenia do sytuacji, w której urządzenie to nie wykonuje żadnych poleceń i czeka na polecenia ze strony gospodarza.

5. Ogólna struktura zaimplementowanego programu-gospodarza

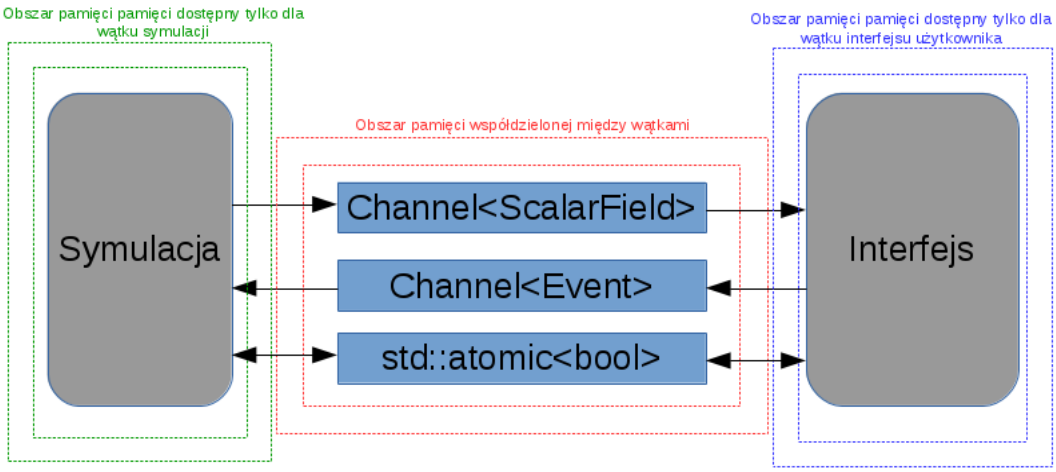
Program-gospodarz zaimplementowanej aplikacji napisany został w języku C++ w wersji 14. Wybór C++14 był kierowany przede wszystkim dwoma powodami. Po pierwsze, wysokiej jakości wsparcie dla OpenCLa w tym języku. Po drugie, gospodarz wymaga względnie niskich opóźnień w wykonywaniu, by możliwe było nasycenie urządzenia obliczeniowego w jak najwyższym stopniu. Osiągnięcie tego celu jest zdecydowanie trudniejsze w językach, których typowe środowiska uruchomieniowe używają GC (ang. G*arbage Collector)* w celu zarządzania pamięcią, gdyż większość mechanizmów GC powoduje znaczące pauzy w wykonywaniu programu w celu odnalezienia niedostępnych obszarów pamięci. Wydawać by się mogło, że nie używanie GC znacząco zwiększałoby złożoność kodu (nie mylić ze złożonością obliczeniową), co najprawdopodobniej byłoby stwierdzeniem prawdziwym w C. Jednakże, w C++ można tego uniknąć poprzez używanie idiomu RAII (ang. Resource Acquisition Is Initialization), który polega na uzależnieniu czasu życia zasobu (np. pamięci) od czasu życia obiektu umieszczonego na stosie, dzięki czemu programista nie musi pamiętać o zwolnieniu zasobu – kompilator wykona odpowiednią operację, gdy obiekt-właściciel przestanie istnieć. Mimo tego, że zaimplementowany gospodarz zawiera sporo operacji dynamicznej alokacji, nie posiada on żadnego wystąpienia tzw. „gołych" operatorów ***new*** i ***delete***. W połączeniu z użyciem standardowych uchwytów do zasobów takich jak ***std::vector<>***, ***std::unique_ptr<>*** czy ***std::shared_ptr<>*** daje to nam gwarancję braku błędów związanych z alokacją i dealokacją pamięci. Oprócz ułatwienia życia programiście, technika ta ma też kolejną ważną zaletę – zapewnia deterministyczne zwalnianie zasobów, co pozwala na ograniczenie opóźnień. Dzięki temu, zaimplementowana aplikacja była w stanie prawe w pełni(92%) nasycić obliczeniowo urządzenie GPGPU, na którym była testowana (AMD Radeon™ HD6870), mimo tego, że kod nie posiadał znaczących optymalizacji, ze względu na to, że najprawdopodobniej spowodowałyby znaczny spadek czytelności kodu. Program-gospodarz zasadniczo składa się z 5 części:

1) inicjalizacja urządzeń, kontekstów i kompilacja kerneli OpenCL,
2) załadowanie parametrów symulacji,
3) tworzenie kanałów komunikacyjnych,
4) tworzenie obiektów reprezentujących interfejs wizualizacyjny i symulację,
5) przetwarzanie wydarzeń w interfejsie i aktualizacja stanu symulacji.

Warto tutaj zauważyć, że nie zostało wspomniane zwalnianie zasobów. Dzięki zastosowaniu *RAII* krok ten generowany jest automatycznie przez kompilator. W punkcie 3) wspomniane zostały kanały komunikacyjne. Jest to pewnego rodzaju kalka językowa z języka programowania *Go* implementująca znany z tego języka paradygmat komunikacji między wątkami *„Don't communicate by sharing memory; share memory by communicating."* (Nie komunikuj się przez współdzielonie pamięci, współdziel pamięć przez komunikację). W „tradycyjnie" napisanych aplikacjach przekaz informacji następuje poprzez współdzielenie stanu określonych obiektów, przez co dostęp do tego stanu musi być chroniony przez kosztowne obliczeniowo mechanizmy synchronizacji. Wspomniany paradygmat odwraca sposób komunikacji między wątkami – informacje przekazywane są w postaci obiektów w pamięci przenoszonych przez kanały, w swoim działaniu nieco przypominającymi potoki

znane z systemów zgodnych ze standardem POSIX. W zaimplementowanej aplikacji kanały posiadają tylko część funkcjonalności oryginału z języka *Go*, a mianowicie posiadają tylko nieblokujące operacje typu ***try_pop(...)*** i ***try_push(...)*** oraz ***try_push_all(...)*** *i* ***try_pop_all(...)***. Zaimplementowane zostały za pomocą kolejek i muteksów (***std::deque<>, std::mutex***). Mają one zdecydowanie większy narzut w porównaniu z oryginałem, jednakże nie ma to większego znaczenia, ponieważ stanowi bardzo niewielką część czasu wykonywania jednego kroku czasowego symulacji. Zasadniczą zaletą tego podejścia jest zawężenie miejsc w kodzie, gdzie wymagane jest zachowanie szczególnej ostrożności – możliwe jest pisanie kodu obsługującego symulację i mechanizmy OpenCL oraz interfejsu wizualizacyjnego przy założeniu, że będą one wykonywane na oddzielnych wątkach bez bezpośredniego „kontaktu" ze sobą. Dzięki temu możliwe jest uniknięcie szerokiej gamy błędów programistycznych związanych z wielowątkowością. Komunikacja z światem zewnętrznym odbywa się tylko przez kanały, co pozwala na uniknięcie wyścigów danych. Nie jest także możliwe wystąpienie zakleszczeń (ang. *deadlock*), ponieważ nie następuje wykonywanie większości funkcji przez więcej niż jeden wątek (wyjątkiem są funkcje należące do kanałów) i związana z nim rekursywna akwizycja muteksów. Z punktu widzenia architektury programu-gospodarza jest jeszcze jedna zaleta tego rozwiązania - klasy komunikujące się przez kanały są bardzo luźno ze sobą powiązane (ang. loose coupling), co bardzo ułatwia analizę ich kodu i działania. Obiekt reprezentujący stan i inwarianty symulacji komunikuje się z obiektem reprezentującym interfejs wizualizacyjny przy pomocy dwóch kanałów. Jeden z nich przechowuje kolejne „migawki" stanu symulacji, który ma zostać zaprezentowany użytkownikowi, w postaci gęstości barwnika unoszonego przez przepływ płynu. Komunikacja odbywa się przez niego jednostronnie, od obiektu symulacyjnego do obiektu interfejsu. Drugi kanał, służy do transportu wydarzeń związanych z interakcją użytkownika z interfejsem. Program obecnie obsługuje 2 takie wydarzenia – dodanie impulsu siłowego i dodanie barwnika, jednakże nic nie stoi na przeszkodzie by rozszerzyć funkcjonalność tego kanału. Komunikacja odbywa się tutaj w kierunku odwrotnym do tego w kanale 1. Schemat na rysunku 5.1 przedstawia komunikację między obiektem klasy Simulation (reprezentującym symulację) i obiektem klasy MainWindow reprezentującym interfejs użytkownika. Event to struktura przechowująca dane na temat wydarzeń w interfejsie użytkownika, np. położenie wydarzenia w układzie współrzędnych symulacji, wartość impulsu czy ilość dodanego barwnika. ScalarField to alias dla typu ***std::vector<cl_float>*** przechowującego pole skalarne reprezentujące gęstość barwnika. Oprócz kanałów komunikacyjnych stosowana jest też flaga sygnalizująca to, czy symulacja ma zostać zakończena, czy nie. Jest ona instancją typu ***std::atomic<bool>***. Dostęp do wartości tej zmiennej jest możliwy tylko poprzez operacje atomowe, które zazwyczaj mają odwzorowanie 1 do 1 z instrukcjami atomowymi procesora. Instrukcje te pozwalają na komunikację z małym narzutem poprzez uniknięcie konieczności użycia muteksów. Ponieważ nie interesuje nas to, w jakiej względnej kolejności odbywają się zapisy i odczyty. W zasadzie ważne jest dla działania programu to, że wartość ***false*** tej zmiennej, oznaczająca zakończenie programu, zostanie w pewnym, nieokreślonym momencie rozpropagowana do innych wątków. Powód opóźnionej propagacji wartości zmiennych z tzw. zrelaksowaną semantyką porządkowania dostępów do pamięci (relaxed memory ordering) leży w architekturze nowoczesnych CPU i tzw. protokołach spójności cache'u, co znajduje się poza zakresem tej pracy.

Dokładniejszy opis i zastosowania operacji atomowych można znaleźć w pozycji [6].



Rys. 5.1. Schemat komunikacji międzywątkowej w programie-gospodarzu.

W ramach nie pogarszania czytelności kodu, zdecydowałem się użyć wyjątków do obsługi błędów OpenCL, ponieważ większość operacji związanych z OpenCLem zwraca kody liczbowe oznaczające sukces, bądź też wystąpienie błędu. Ze względu na naturę aplikacji wszystkie wyjątki rzucone przez OpenCL nie są obsługiwane i powodują zakończenie działania programu, wszakże nie ma sensu kontynuacja działania programu jeśli skończyły się zasoby na urządzeniu obliczeniowym i jest ono w nieokreślonym stanie. Wbrew powszechnie panującemu przekonaniu wyjątki nie powodują znaczącego spadku wydajności. Jest nawet możliwe, w niektórych sytuacjach, że obsługa błędów poprzez instrukcje warunkowe i kody zwrotne będzie wolniejsza od wyjątków, zakładając, że rzucane są one rzadko.

6. Symulacja – inicjalizacja stanu i kolejkowanie kerneli obliczeniowych

W klasie reprezentującej symulację przepływu płynu można wyróżnić zasadniczo trzy funkcjonalności:

1) zarządzanie czasem życia i inicjalizacja obiektów w pamięci urządzenia OpenCL,
2) kolejkowanie kerneli obliczeniowych,
3) pobieranie wyniku symulacji z urządzenia obliczeniowego i komunikacja z interfejsem wizualizacyjnym.

Pierwsza wymieniona funkcjonalność mieści się całkowicie w konstruktorze klasy *Simulation*. Można wyróżnić dwa etapy inicjalizacyjne – utworzenie(przy jednoczesnej inicjalizacji) buforów pamięci oraz tworzenie obiektów reprezentujących kernele obliczeniowe i następnie ustawienie części z ich argumentów, a dokładniej tych argumentów, które nie zmieniają się w trakcie symulacji. Te stałe to np. krok czasowy δt przekazywany przy wywoływaniu kernela obliczającego etap adwekcji.
Druga wspomniana funkcjonalność klasy reprezentującej symulację to kolejkowanie kerneli obliczeniowych. Aplikacja zawiera 2 rodzaje kerneli – kernele „wewnętrzne", które wykonywane są dla każdej komórki nie będącej komórką brzegową oraz brzegowe wykonywane dla komórek brzegowych. Operację kolejkowania kernela w kolejce poleceń urządzenia obliczeniowego można podzielić na dwie mniejsze – jedna wspólna dla wszystkich kerneli danego rodzaju, zaś ta druga specyficzna dla każdego kernela (polegająca przede wszystkim na ustawieniu parametrów nie będących stałymi). Istnieją dwie metody służące zakolejkowaniu kerneli w kolejce urządzenia obliczeniowego. Ich kod przedstawiony jest na listingu 6.1.

```
void Simulation::enqueue_boundary_kernel(cl::Kernel& boundary_kernel) const
{
	const auto boundary_cell_count = cell_count - 2;
	const auto horizontal_NDRange  = cl::NDRange{boundary_cell_count, 1};
	const auto vertical_NDRange    = cl::NDRange{1, boundary_cell_count};

	//Pierwszy i ostatni wiersz
	boundary_kernel.setArg(1, Offset{0, 1});
	cmd_queue.enqueueNDRangeKernel(boundary_kernel, cl::NDRange{1, 0}, horizontal_NDRange);
	boundary_kernel.setArg(1, Offset{0, -1});
	cmd_queue.enqueueNDRangeKernel(boundary_kernel, cl::NDRange{1, cell_count - 1}, horizontal_NDRange);

	//Pierwsza i ostatnia kolumna
	boundary_kernel.setArg(1, Offset{1, 0});
	cmd_queue.enqueueNDRangeKernel(boundary_kernel, cl::NDRange{0, 1}, vertical_NDRange);
	boundary_kernel.setArg(1, Offset{-1, 0});
	cmd_queue.enqueueNDRangeKernel(boundary_kernel, cl::NDRange{cell_count - 1, 1}, vertical_NDRange);

	cmd_queue.enqueueBarrierWithWaitList();
}

void Simulation::enqueue_inner_kernel(const cl::Kernel& kernel) const
{
	const auto range  = cl::NDRange{workgroup_size, workgroup_size};

	for (uint y = 1; y < cell_count - 2; y += workgroup_size) {
		for (uint x = 1; x < cell_count - 2; x += workgroup_size) {
			cmd_queue.enqueueNDRangeKernel(kernel, cl::NDRange{x, y}, range);
		}
	}

	cmd_queue.enqueueBarrierWithWaitList();
}
```

Listing 6.1. Dodawanie kerneli do kolejki poleceń urządzenia obliczeniowego

Metoda ***enqueue_boundary_kernel(...)*** służy zakolejkowywaniu kerneli liczących warunki brzegowe. Istnieją 4 zakresy brzegowych komórek symulacji: górny i dolny wiersz i pierwsza i ostatnia kolumna. Każdy zakres zawiera N-2 komórek, gdzie N to wymiar przestrzeni symulacji w danym wymiarze. Odjęcie 2 w tym wypadku mówi o tym, że pomijamy komórki brzegowe znajdujące się w wierzchołkach prostokąta reprezentującego przestrzeń symulacji, ponieważ nie graniczą one z żadną komórką niebrzegową. Przed zakolejkowaniem każdego z zakresów ustawiany jest argument kerneli brzegowych – offset. Mówi on kernelowi, na podstawie której komórki symulacji ma on obliczyć warunek brzegowy. Przykładowo, dla pierwszego wiersza offset w kierunku x wynosi 0, zaś w kierunku y 1, co oznacza, że komórka poniżej komórki brzegowej jest źródłem wartości na podstawie której obliczany jest dany warunek brzegowy. Należy tutaj pamiętać, że oś y ma zwrot przeciwny do zwrotu najczęściej stosowanego w matematyce, tzn. wartości y rosną „w dół”. Konwencja ta stosowana jest przy tworzeniu interfejsów graficznych. W mojej implementacji jest ona użyta, by uprościć odwzorowanie układu współrzędnych interfejsu na układ współrzędnych symulacji i vice versa. Ostatnią operacją dodaną w tej metodzie do kolejki poleceń urządzenia obliczeniowego jest bariera z listą oczekiwania. W tym przypadku lista oczekiwania jest pusta, a zatem najpierw wykonane zostaną polecenia sprzed bariery, a następnie te znajdujące się po niej. Stosowanie barier przy każdym zakolejkowaniu kernela jest konieczne, ponieważ zapobiegają one powstawaniu wyścigów danych między kolejnymi grupami roboczymi lub sąsiednimi kernelami. Druga metoda widoczna na listingu 6.1 to ***enqueue_inner_kernel(…)*** służąca do kolejkowania kerneli wykonujących obliczenia dla wewnętrznych komórek symulacji. Pętle ***for*** służą w tej metodzie optymalnemu podziałowi przestrzeni symulacji na prostokąty, tzn. w taki sposób, by zminimalizować narzut związany z kolejkowaniem kerneli, co oznacza kolejkowanie grup roboczych o rozmiarze taki jak maksymalny obsługiwany rozmiar dla wybranego urządzenia obliczeniowego. W przypadku urządzeń, na których testowana była aplikacja, maksymalny rozmiar grupy roboczej wynosi 256. Zakładam tutaj, że liczba komórek wewnętrznych symulacji jest podzielna przez 256. Tak jak w przypadku poprzedniej metody, tutaj także konieczne jest użycie bariery, by uniknąć problemów z niesynchronizowanymi zapisami i odczytami. Większość operacji składających się na symulację przepływu płynu Metodą Stabilnych Płynów używa wyżej wymienionych metod w bardzo prosty sposób – ustawiają argumenty kerneli i wywołują jedną z nich. Jednakże w przypadku etapów polegających na rozwiązaniu równań Poissona wymagany jest większy stopień złożoności kodu. Listing 6.2 przedstawia to na przykładzie etapu rzutowania opisanego w rozdziale 2., a dokładniej jego zasadniczej części – poszukiwania pola skalarnego opisującego ciśnienie płynu. Na początku następuje wypełnienie zerami bufora pamięci zawierającego wartość początkową ciśnienia co zmniejsza ilość iteracji niezbędnych w celu osiągnięcia zbieżności. Następnie ustawiony jest jeden z argumentów kernela obliczeniowego, który nie jest zmieniany podczas kolejnych iteracji algorytmu rozwiązującego równania Poissona. Operacja ta (jak zresztą większość operacji składających się na symulację) nie może być wykonana w miejscu, tzn. wymagany jest dodatkowy bufor przechowujący wynik każdej z iteracji. Pod koniec każdej iteracji zamieniamy bufory, korzystając z powszechnie stosowanego idiomu – użycie funkcji ***swap(...)***, której odpowiednie przeciążenie wybierane jest za pomocą ADL (ang. Argument-Dependent Lookup – Wyszukiwanie [przeciążeń]

Zależne od Argumentów). Celem tego idiomu jest umożliwienie użycia specjalizacji funkcji ***swap(...)*** jeżeli została ona zadeklarowana w przestrzeni nazw zawierającej zamieniany typ lub użycie szablonu ***std::swap(…)*** w przypadku przeciwnym.

```
void Simulation::zero_fill_scalar_field(cl::Buffer& field)
{
	cmd_queue.enqueueFillBuffer(field, Scalar{0.0}, 0, total_cell_count);
	cmd_queue.enqueueBarrierWithWaitList();
}

void Simulation::calculate_p()
{
	zero_fill_scalar_field(p);
	scalar_jacobi_kernel.setArg(1, divergence_w);

	for (int i = 0; i < jacobi_iterations; ++i) {
		apply_scalar_boundary_conditions(p);

		scalar_jacobi_kernel.setArg(0, p);
		scalar_jacobi_kernel.setArg(2, temporary_p);
		enqueueInnerKernel(cmd_queue, scalar_jacobi_kernel);

		using std::swap;
		swap(p, temporary_p);
	}

	apply_scalar_boundary_conditions(p);
}
```

Listing 6.2 Obliczanie ciśnienia

Warto tutaj wspomnieć, że następuje jedynie zamiana uchwytów do buforów znajdujących się w przestrzeni adresowej gospodarza, a zatem operacja ich zamiany jest tania pod względem obliczeniowym. Ponadto, nie są konieczne żadne mechanizmy synchronizacji po stronie urządzenia obliczeniowego, ponieważ z jego punktu widzenia nie wykonywane są żadne operacje na wspomnianych buforach.
Ostatnią zasadniczą funkcjonalnością klasy reprezentującej symulację wymienioną na początku rozdziału jest komunikacja z interfejsem wizualizacyjnym i pobieranie wyniku każdego kroku czasowego symulacji, tzn. pola skalarnego reprezentującego gęstość barwnika unoszonego przez przepływ płynu. Listing 6.3 przedstawia metodę zawierającą wymienione operacje. Pewne etapy symulacji zostały pominięte w celu zachowania czytelności kodu w tej pracy. Komunikacja z interfejsem odbywa się poprzez dwa kanały komunikacyjne: ***events_from_ui*** i ***to_ui.*** Metoda ***update()*** reprezentuje jeden krok czasowy symulacji. Na początku wykonywany jest etap adwekcji. Następnie, następuje próba pobrania kontenera (w tym wypadku ***std::deque***) zawierającego wydarzenia opisujące interakcję użytkownika z symulacją, a dokładniej operacje dodania barwnika i przyłożenia siły. W zależności od typu wydarzenia wywoływana jest odpowiednia metoda. Następnie wykonywane są pozostałe etapy symulacji (pominięte na listingu), takie jak dodanie stałych źródeł sił, dyfuzja, przywrócenie wirowości itd. Ostatnim etapem związanym z samą symulacją jest obliczenie pola wynikowego u opisującego przepływ. Tak jak to zostało opisane w rozdziale 2., pole wektorowe u można obliczyć przez zastosowanie operatora rzutowania P. Zadaniem ostatniego bloku kodu jest kopiowanie wynikowego pola skalarnego ***dye***, opisującego gęstość barwnika, z pamięci urządzenia obliczeniowego do pamięci gospodarza. Warto tu wspomnieć, że operacja ta nie wymaga tutaj bariery, gdyż zawarta jest ona w funkcji ***cl::copy(…).*** Na końcu następuje próba wypchnięcia wyniku do kanału komunikacyjnego. Jeżeli nie powiedzie się ona, to następuje dodanie wyniku do kolejki oczekującej. Dzięki temu w następnym kroku symulacji może zostać użyta metoda ***try_push_all(…)*** o mniejszym narzucie, która podejmuje próbę przesłania wszystkich elementów znajdujących się w kolejce oczekujących.

```
void Simulation::update()
{
	calculate_advection();

	//Komunikacja Simulation <- MainWindow
	auto received_events = events_from_ui->try_pop_all();
	for (const auto& simulation_event : received_events) {
		if (simulation_event.type == Event::Type::ADD_DYE) {
			add_dye(simulation_event);
		} else if (simulation_event.type == Event::Type::APPLY_FORCE) {
			apply_impulse(simulation_event);
		}
	}

	//Obliczenie dyfuzji, dodanie sił stałych, sił wyporu, przywrócenie wirowości itd.

	apply_projection_operator();

	//Komunikacja Simulation -> MainWindow
	ScalarField output_buffer;
	output_buffer.resize(total_cell_count);
	cl::copy(cmd_queue, dye, output_buffer.begin(), output_buffer.end());

	if (dye_buffers_wait_list.empty()) {
		if (not to_ui->try_push(output_buffer)) {
			dye_buffers_wait_list.emplace_back(std::move(output_buffer));
		}
	} else {
		dye_buffers_wait_list.emplace_back(std::move(output_buffer));
		to_ui->try_push_all(dye_buffers_wait_list);
	}
}
```

Listing 6.3 Komunikacja z interfejsem i kopiowanie wyniku kroku symulacji

7. Kernele obliczeniowe

Tak jak to zostało wspomniane w rozdziale 4. kernele obliczeniowe to po prostu funkcje wykonywane na urządzeniu obliczeniowym, przy czym każde wywołanie takiej funkcji typowo odbywa się dla każdej z komórek symulacji. Przy pisaniu kerneli obliczeniowych pomocne okazały się funkcje pomocnicze, które widoczne są na listingu 7.1.

```
inline size_t AT(size_t x, size_t y)
{
	return y*SIZE + x;
}

inline size_t AT_POS(Point pos)
{
	return AT(pos.x, pos.y);
}

inline Point getPosition()
{
	Point point = {get_global_id(0), get_global_id(1)};
	return point;
}
```

Listing 7.1 Funkcje pomocnicze kerneli obliczeniowych

Mimo, że nie jest to konieczne, funkcje te zostały oznaczone słowem kluczowym ***inline***, ponieważ, wspomniany został wcześniej fakt, że nie jest możliwe „typwe" wywoływanie funkcji z poziomu kernela obliczeniowego, aczkolwiek możliwe jest statyczne „wklejenie" przez kompilator ciała funkcji wywoływanej do kodu wywołującego ją kernela. Zatem to słowo kluczowe ma tutaj przede wszystkim funkcje informatywną dla programisty, by było możliwe łatwe odróżnienie funkcji pomocniczych od kerneli. Pierwsze dwie funkcje służą obliczeniu indeksu w jednowymiarowym buforze danych reprezenującym dwuwymiarową przestrzeń symulacji. Ostatnia funkcja zwraca pozycję wykonywanego work-itema w grupie roboczej w postaci koordynatów w przestrzeni symulacji. Funkcje te znacząco poprawiają czytelność kodu poprzez ukrycie tych szczegółów w cienkiej warstwie abstrakcji, co dobrze widać na przykładzie kernela obliczającego jedną iterację metody Jacobiego dla pola wektorowego, widoczną na listingu 7.2.

```
kernel void vector_jacobi_iteration(const GlobalVectorField x,
                                    const GlobalVectorField b,
                                          GlobalVectorField x_out,
                                    const Scalar alpha,
                                    const Scalar beta_reciprocal)
{
	const Point position = getPosition();
	const int index = AT_POS(position);

	const Vector x_left = x[AT(position.x - 1, position.y)];
	const Vector x_right = x[AT(position.x + 1, position.y)];
	const Vector x_top = x[AT(position.x, position.y + 1)];
	const Vector x_bottom = x[AT(position.x, position.y - 1)];

	x_out[index] = (x_left + x_right + x_top + x_bottom + alpha * b[index]) * beta_reciprocal;
}
```

Listing 7.2 Kernel obliczający jedną iterację metody Jacobiego

GlobalVectorField to alias dla typu ***global Vector****, czyli tablicy przechowującej pole wektorowe, znajdującej się w globalnej przestrzeni adresowej. Ponieważ kernel ten

nie może wykonywać obliczeń w miejscu, konieczna jest tablica „źródłowa” i wynikowa, co widać w argumentach tego kernela (***x*** i ***x_out***). Można wyróżnić 3 etapy w wykonaniu tego kernela:

1) obliczenie pozycji w przestrzeni symulacji i indeksu w tablicy,
2) odczytanie wartości ***x*** sąsiednich komórek symulacji (zmienne ***x_left***, ***x_right***, ***x_top*** i ***x_bottom***),
3) obliczenie wyniku, zgodnie z równaniem w rozdziale 2., i zapisanie go w odpowiednim miejscu w tablicy wynikowej.

Jak widać kernele obliczeniowe mają bardzo prostą budowę, głównie przez to, że nie zawierają żadnej logiki, a głównie operacje arytmetyczne oraz operacje odczytu/zapisu. Nie jest to cecha przypadkowa – urządzenia GPGPU bardzo sprawnie wykonują operacje arytmetyczne na typach zmiennoprzecinkowych, natomiast radzą sobie słabo przy wykonywaniu instrukcji warunkowych (np. ***if***), co zostało opisane w rozdziale 4.

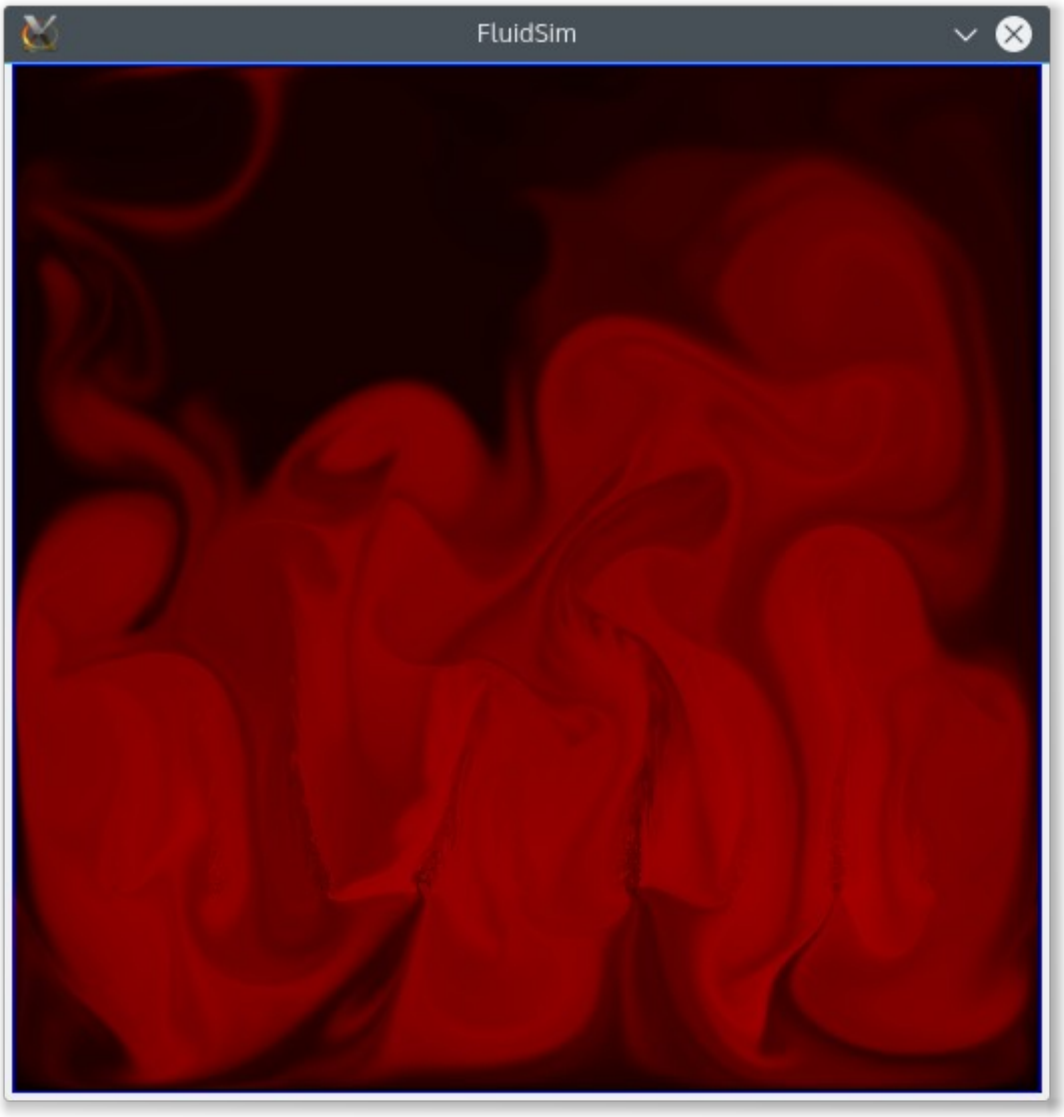
8. Interfejs wizualizacyjny

Interfejs jest w zasadzie najmniej skomplikowaną częścią opisywanej aplikacji. Został napisany w oparciu o bibliotekę SDL2. Decyzja ta była kierowana przede wszystkim prostotą. W porównaniu do innych bibliotek SDL2 nie wymaga dużej ilości kodu by otrzymać prosty, działający interfejs. Najciekawszymi metodami są te odpowiedzialne za odbieranie wyniku symulacji z kanału komunikacyjnego oraz dokonujące mapowania wartości gęstości barwnika w danym punkcie na odpowiedni prostokąt w oknie wizualizacji. Przedstawione zostało one na listingu 8.1

```
void retrieve_dye_field_queue()
{
	std::deque<ScalarField> field_queue = dye_field_to_ui->try_pop_all();
	if (not field_queue.empty()) {
		if (not this->field.empty()) {
			std::swap(this->field, field_queue.back());
		}

	}
}

void paint()
{
	auto renderer = this->renderer.get();
	SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
	SDL_RenderClear(renderer);
	SDL_SetRenderDrawColor(renderer, 0, 0, 255, 255);
	SDL_RenderDrawRect(renderer, &boundary_rect);
	SDL_Rect rect;
	rect.w = pixels_per_cell;
	rect.h = pixels_per_cell;

	retrieve_dye_field_queue();

	for (uint x = 1; x < cells - 2; ++x) {
		for (uint y = 1; y < cells - 2; ++y) {
			rect.x = x * pixels_per_cell;
			rect.y = y * pixels_per_cell;
			auto field_val = field[y * cells + x];
			SDL_SetRenderDrawColor(renderer, std::min(255 * field[y * cells + x], 255.0f), 0, 0, 255);
			SDL_RenderFillRect(renderer, &rect);
		}
	}

	SDL_RenderPresent(renderer);
}
```

Listing 8.1 Metody aktualizująca wizualizację i odbierająca wynik symulacji

W zmiennej ***this->field_queue*** przechowywana jest aktualne pole skalarne reprezentację gęstość barwnika. Jest ona potrzebna, ponieważ aktualizacja wizualizacji może nastąpić zanim zakończy się obliczanie wyniku następnego kroku czasowego symulacji (np. gdy okno zostanie przesunięte na ekranie przez użytkownika). Okno symulacji podzielone jest na N-2 prostokątów, każdy o rozmiarze ***pixels_per_cell***, reprezentujących wewnętrzne komórki symulacji. Gęstość barwnika mapowana jest liniowo na kolor czerwony danego prostokąta, przy czym wartość jest ograniczona do 255. Warto tutaj zauważyć, że interfejs wymaga, by rozmiar okna w każdym wymiarze był większy niż liczba wewnętrznych komórek symulacji. Zrzut ekranu na rysunku 8.2 przedstawia jak przykładowo wygląda wizualizacja przy 8 stałych źródłach sił skierowanych w górę. Należy zauważyć, że niewielka nawet interakcja ze strony użytkownika może spowodować duże różnice w stanie symulacji, zatem ciężko jest odtworzyć stan po restarcie symulacji, jeśli wystąpiła interakcja użytkownika z aplikacją w postaci przyłożenia sił. Stałe opisujące płyn zostały tak dobrane, by symulowany był przepływ cieczy zbliżonej właściwościami fizycznymi do wody zamkniętej w pudełku o względnie niewielkiej wysokości.



Rys. 8.2 Okno wizualizacji

9. Podsumowanie

Realizując projekt aplikacji wykonującej symulację przepływu płynu udało się zrealizować prawie wszystkie założenia przedstawione na wstępie. Stworzona aplikacja jest w stanie wygenerować zadowalającą ilość klatek na sekundę na testowanym sprzęcie, dla całkiem dużej liczby komórek (uzyskano około 20 klatek na sekundę dla przestrzeni symulacji o rozmiarze 512x512 komórek na urządzeniu GPGPU AMD Radeon™ HD 6870). Dla porównania, uzyskanie podobnej liczby klatek na sekundę(w wypadku średnio uzyskano 15) na CPU (Intel Haswell i5-3210M) wymaga zmniejszenia 4-krotnie liczby komórek symulacji (do 256x256). Niestety, nie udało się osiągnąć jednego z założeń – stabilności symulacji. Przyłożenie zbyt dużej siły blisko wierzchołków przestrzeni symulacji powoduje destabilizację numeryczną na etapie opisanym przez operator rzutowania P, przedstawiony w rozdziale 2. Teoretycznie, wszystkie zastosowane metody numeryczne są stabilne, jednakże, przy rozwiązywaniu równań Poissona stosowana jest stała liczba iteracji algorytmu Jacobiego, co powoduje, że może nie nastąpić zbieżność w pewnych, wyżej opisanych warunkach. W praktyce jednak sytuacja nie zdarza się często, w zasadzie występuje jedynie, gdy użytkownik celowo próbuje ją zdestabilizować. Poprawienie tego problemu niestety nie jest łatwe, ponieważ wymagałoby dodania dodatkowej logiki badającej, czy konieczna jest następna iteracja algorytmu, co mogłoby się wiązać z pogorszeniem wydajności i wymagałoby całkowitą migrację do OpenCLa w wersji 2.0, ponieważ wcześniejsze wersje nie pozwalają na kolejkowanie poleceń z poziomu kodu wykonywanego na urządzeniach obliczeniowych. Mimo, to, udało się zaobserwować kilka zjawisk typowych dla przepływu nieściśliwego płynu w zamkniętej przestrzeni, np. powstawanie wirów, wynikających z zasady zachowania masy, czy też „odbijanie” się i dyfuzja barwnika, gdy przepływ skierowany jest bezpośrednio w stronę granicy symulacji.

OpenCL okazał się narzędziem o bardzo dużych możliwościach, które względnie łatwo jest użyć w celu rozwiązywania równań różniczkowych. Moim zdaniem jego pozycja wśród rozwiązań z dziedziny HPC (ang. High Performance Computing) będzie się w przyszłości umacniać, ze względu na to, że dosyć duża klasa rozwiązań problemów daje się zrównoleglić w sposób „przyjazny” dla urządzeń typu GPGPU. Rozwój OpenCLa zmierza w stronę dalszej rozbudowy możliwości przez niego oferowanych, np. w wersji 2.1 (ustandaryzowanej na początku 2015 roku) oferuje on możliwość pisania kerneli obliczeniowych w tzw. statycznym podzbiorze języka C++14, który w porównaniu do obecnie stosowanego języka bazującego na C99 zapewnia programiście niesłychanie użyteczne narzędzia do tworzenia abstrakcji.

Istnieje wiele możliwości rozbudowy stworzonej aplikacji. Można osiągnąć zwiększenie wydajności poprzez użycie obrazów i niestandardowych rozszerzeń OpenCLa pozwalających na współpracę z OpenGLem (Open Graphics Library, biblioteka służąca wydajnemu renderowaniu grafiki z użyciem akceleracji sprzętowej oferowanej przez GPU). Umożliwiłoby to uniknięcie konieczności kopiowania wyniku symulacji do pamięci hosta, a zatem cały stan symulacji mógłby zostać po stronie urządzenia obliczeniowego. Użycie obrazów zamiast buforów pozwoliłoby także na użycie wbudowanego w urządzenia GPGPU mechanizmu sprzętowej interpolacji współrzędnych na obrazie, co znacząco przyspieszyłoby wykonywanie etapu adwekcji. Jednakże, zastosowanie tego rozwiązania wymagałoby zwiększenia

złożoności kodu dokonującego mapowania współrzędnych w interfejsie wizualizacyjnym na współrzędne w przestrzeni symulacji. Ponadto, jeden z mankamentów obecnej implementacji zostałby praktycznie sam rozwiązany, a dokładniej wymaganie co do minimalnego rozmiaru okna. Dzięki użyciu OpenGLa, możliwe byłoby łatwe skalowanie wynikowego obrazu, reprezentującego gęstość barwnika, do odpowiedniego rozmiaru. Kolejne rozszerzenie – dodanie 3-ciego wymiaru także wymagałoby użycia OpenGLa w celu wizualizacji przepływu 3-wymiarowego. Sama metoda stabilnych płynów nie wymagałaby zbyt wielu modyfikacji – w zasadzie konieczne byłoby uwzględnienie dodatkowego wymiaru w etapach, które korzystają z sąsiednich punktów, lub manipulują składniki wektorów wzdłuż jednej z osi. Dla przykładu algorytm Jacobiego wymagałby uwzględnienia 2 dodatkowych sąsiadów wzdłuż wymiaru z, ponieważ symulacja podzielona jest na sześciany. Do 3-wymiarowej symulacji można by także dodać kolejny interesujący efekt – granice między różnymi płynami, np. powierzchnia wody odgraniczająca ją od powietrza. Algorytm rozwiązywania równań Poissona, który został użyty w tej aplikacji – algorytm Jacobiego nie jest zbyt wydajny obliczeniowo. Zamiast niego można by zastosować metody osiągające zbieżność szybciej, np. relaksacyjną metodę Gaussa-Seidla.

10. Bibliografia

[1] J. Stam „*Stable Fluids.”*, 1999
[2] M.J Harris, *”GPU Gems”*, University of North Carolina at Chapel Hill, 2007
[3] *„AMD Accelerated Parallel Processing – OpenCL User Guide”,* 2014
[4] J. Stam, „*Real-Time Fluid* Dynamics *for Games*”, 2003
[5] B. R. Gaster, L. Howes, *„The OpenCL C++ Wrapper API”*, Khronos OpenCL Working Group, wersja 1.2.6
[6] A. Williams, *„C++ Concurrency in Action – Practical Multithreading”,* wydawnictwo Hanning, 2012
[7] „Argument-dependent lookup” http://en.cppreference.com/w/cpp/language/adl
[8] „Bilinear interpolation” https://en.wikipedia.org/wiki/Bilinear_interpolation