Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki

Katedra Informatyki

Praca magisterska

Piotr Janik

Wydajna symulacja dynamiki płynów i przewodnictwa cieplnego w środowisku przeglądarki internetowej

Promotor:
prof. dr hab. inż. Witold Dzwinel

Kraków 2012

OŚWIADCZENIE AUTORA PRACY

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie, i nie korzystałem ze źródeł innych niż wymienione w pracy.

.....

PODPIS

AGH
University of Science and Technology in Krakow

Faculty of Electrical Engineering, Automatics, Computer Science and Electronics

Department of Computer Science

Master of Science Thesis

Piotr Janik

Efficient Simulation of Fluid Dynamics and Heat Transfer in the Web Browser Environment

Supervisor:
Witold Dzwinel, Prof.

Krakow 2012

Serdecznie dziękuję ...

Streszczenie

Celem niniejszej pracy było stworzenie wydajnej symulacji dynamiki płynów oraz przewodnictwa cieplnego działającej w środowisku przeglądarki internetowej. Aplikacja ma charakter edukacyjny, a jej głównym zadaniem jest wspieranie użytkownika w dogłębnym zrozumieniu procesu transferu energii, w tym zjawisk takich jak przewodnictwo cieplne, konwekcja czy różnorodne przepływy gazów i cieczy. Projekt został zrealizowany przy współpracy z The Concord Consortium, amerykańską organizacją non-profit zajmującą się wspieraniem edukacji poprzez technologię.

Implementacja symulatora w przeglądarce internetowej była niezwykle istotna ze względu na jego edukacyjne zastosowanie – kluczowym wymaganiem była dostępność dla jak najszerszego grona użytkowników, przenośność, wieloplatformowość oraz możliwość łatwego osadzania w wirtualnych podręcznikach szkolnych nowej generacji. Z drugiej strony, o jakości symulacji fizycznej w dużej mierze decyduje jej wydajność czyli efektywne wykorzystanie zasobów. Do niedawna stało to w sprzeczności z implementacją w języku *JavaScript* oraz wykonywaniem w środowisku przeglądarki internetowej.

Realizacja tych pozornie wykluczających się wymagań została osiągnięta dzięki implementacji uwzględniającej budowę i ograniczenia silników *JavaScript* w nowoczesnych przeglądarkach oraz dzięki przeniesieniu większości obliczeń na procesor karty graficznej wykorzystując technologię *WebGL*. Jest to podejście nowatorskie, gdyż dopiero wraz z niedawnym rozpowszechnieniem się standardu *HTML5* zasoby kart graficznych stały się dostępne dla aplikacji internetowych w tak szerokim zakresie.

W pracy zostały przedstawione najważniejsze, nowoczesne techniki umożliwiające tworzenie wydajnych, równoległych aplikacji działających w przeglądarce internetowej z wykorzystaniem języka *JavaScript* oraz technologii *WebGL*. Szczegółowo zostały również opracowanie rezultaty oraz korzyści płynące ze zrównoleglenia symulacji fizycznej. W efekcie powstał wydajny symulator, który dzięki swojej dostępności oraz jakości może mieć niezwykle szeroki wpływ na edukację i zrozumienie fizyki przez użytkowników na różnych etapach procesu kształcenia.

Spis treści

1. Wprowadzenie

Większość ludzi w swoich domach i pracy może korzystać z dorobku technologicznej rewolucji, która miała miejsce w ostatnich latach. Jednak w niektórych dziedzinach życia zmiany następują znacznie wolniej - jedną z nich jest edukacja. Komputery i internet stały się dostępne w większości szkół, jednak programy i metody nauczania często nie nadążają za postępem technologicznym, są wciąż reliktem poprzedniej epoki. Uczniowie pracują na komputerach najnowszej generacji, jednak zwykle wykorzystują tylko ułamki ich możliwości, nie mając dostępu do narzędzi, które faktycznie mogłyby przyczynić się do lepszego zrozumienia poruszanych na lekcjach zagadnień. Najczęściej komputer i internet stają się po prostu źródłem łatwo dostępnej wiedzy czy też miejscem gdzie pewne problemy można próbować rozwiązywać wspólnie. Jest to oczywiście prawidłowe i wartościowe wykorzystanie nowoczesnej technologi, ale jednocześnie też dosyć powierzchowne i niewyczerpujące jej pełnych możliwości. Problemy i wyzwania stojące przed współczesną edukacją szerzej porusza Andrew A. Zucker [Zuc09].

Edukacja może skorzystać na technologicznej rewolucji w znacznie większym stopniu - jednym z pomysłów są wirtualne laboratoria, które pozwolą uczniom eksplorować wybrane zagadnienia w sposób interaktywny, szczególnie podczas nauczania przedmiotów ścisłych i przyrodniczych, tak istotnych w dzisiejszych czasach. Cyfryzacja powinna zmienić tradycyjne oblicze lekcji z podręcznikiem i zeszytem na pracę przy narzędziach edukacyjnych nowej generacji, wykorzystując powszechną dostępność nowoczesnych technologii. Takie aplikacje również doskonale wpisują się w popularną ideę cyfrowych podręczników. Dosłowne przeniesienie zawartości papierowych książek na ekrany komputerów nie wiązałoby się ze znaczącymi zmianami - inny byłby tylko nośnik słów, wiedzy. Bez zmian natomiast pozostałby sam proces i metody uczenia przez uczniów i studentów. Jednak jeśli wirtualny podręcznik zostanie zintegrowany z interaktywnymi aplikacjami, pozwoli to zupełnie zmienić oblicze nauki. Uczeń będzie miał możliwość prawdziwej eksploracji zagadnień, eksperymentowania we własnym domu, przed własnym komputerem, podczas codziennej nauki, która może zamienić się w prawdziwą, wartościową i przede wszystkim rozwijającą przygodę.

Najlepszym pomysłem dla podręczników przyszłości wydaje się umiejscowienie ich

w internecie. Dzięki dystrybucji poprzez to medium można uzyskać niezwykle łatwy i powszechny dostęp, jako że połączenie z internetem jest w dzisiejszych czasach czymś w pełni osiągalnym. Internetowa dystrybucja niesie również mnóstwo korzyści nie tylko dla użytkowników podręczników, ale także dla ich twórców - wystarczy wymienić zalety takie jak łatwość aktualizacji i docierania do odbiorców. W związku z tym, również narzędzia stanowiące interaktywne elementy podręczników przyszłości powinny być przystosowane do działania w środowisku przeglądarki internetowej. Jest to zadanie wymagające, jednak niedawny rozwój technologii i standardów internetowych takich jak HTML5 oraz WebGL, jak również gwałtowne zmiany w samych przeglądarkach internetowych, dają ogromne możliwości w tej materii.

Wymienione pomysły nie są tylko planami na przyszłość - te zmiany już powoli następują, cyfrowe podręczniki i wirtualne laboratoria są trakcie rozwoju. Jedną z organizacji zajmujących się wprowadzaniem najnowszych osiągnięć techniki do szkół jest The Concord Consortium.

1.1. Cel pracy

W wyniku współpracy ze wspomnianą organizacją The Concord Consortium powstał wydajny symulator fizyczny prezentujący zjawisko przewodnictwa cieplnego oraz dynamikę płynów działający w środowisku przeglądarki internetowej o nazwie *Energy2D*. Ta interaktywna aplikacja doskonale wpisuje się w przedstawioną ideę wirtualnych podręczników i laboratoriów, umożliwiając użytkownikom łatwiejsze zrozumienie praw fizyki, które rządzą transferem energii.

Celem niniejszej pracy jest przedstawienie rozwiązań, które umożliwiły powstanie symulatora, ze szczególnym naciskiem na technologię WebGL, której niestandardowe i nowatorskie zastosowanie pozwoliło zrównoleglić obliczenia fizyczne i tym samym osiągnąć znaczący wzrost wydajności.

1.2. Organizacja dokumentu

Dalsze rozdziały przedstawiają kolejno:

- Wprowadzenie do problematyki symulacji dynamiki płynów i przewodnictwa cieplnego, jej znaczenie dla edukacji oraz przegląd istniejących rozwiązań.
- Opis podstawowej implementacji aplikacji, ze szczególnym uwzględnieniem zagadnień związanych z jej architekturą i wnioskami, które można rozszerzyć na ogół złożonych systemów *JavaScript*.

- Prezentację kluczowych technik dzięki którym udało się zrównoleglić silniki fizyczne symulatora przy pomocy technologii *WebGL*.
- Ocenę systemu, w szczególności testy jakościowe, wydajnościowe oraz badanie jak konfiguracja sprzętowa użytkownika wpływa na odbiór i jakość symulacji.
- Podsumowanie, wnioski, oraz pomysły na dalszy rozwój aplikacji.

2. Symulacja dynamiki płynów i przewodnictwa cieplnego jako wartościowe narzędzie edukacyjne

2.1. Zastosowanie oraz wpływ na edukację

2.2. Motywacja i uzasadnienie osadzenia symulacji w środowisku przeglądarki internetowej

2.3. Dostępne, istniejące rozwiązania

2.3.1. Przegląd

2.3.2. Prekursor systemu - aplikacja Energy2D

2.4. Zjawiska modelowane przez symulator

2.4.1. Przewodnictwo cieplne

2.4.2. Dynamika płynów

2.5. Zastosowane silniki fizyczne

2.5.1. Równanie przewodnictwa cieplnego

2.5.2. Równanie Naviera-Stokesa

3. Implementacja symulatora w środowisku przeglądarki internetowej

Niniejszy rozdział omawia najważniejsze zagadnienia związane z implementacją symulatora *Energy2D* w środowisku przeglądarki internetowej, wykorzystując język programowania *JavaScript*.

Technologie te nie były w zamierzeniu tworzone z myślą o dużych, złożonych aplikacjach. Przeciwnie, język *JavaScript* miał być językiem wyjątkowo prostym, łatwym do nauczenia, odpornym na błędy i przeznaczonym głównie do tworzenia prostych skryptów wzbogacających treść stron internetowych. Jednak gwałtowny rozwój technologii internetowych zmienił sytuacje. Aplikacje dostępne przez przeglądarkę stały się zaawansowane i złożone, często wypierając swoje odpowiedniki instalowane bezpośrednio na komputerach użytkowników. Jednak sama technologia niewiele się zmieniła od swojego powstania. Pierwsza specyfikacja *ECMAScript* (definiująca język *JavaScript*) pojawiła się w roku 1997, została uaktualniona również w latach 1998 oraz 1999, a przez następnych dziesięć lat, aż do roku 2009, nie wprowadzono żadnych zmian. Obrazuje to fakt, iż sam rozwój języka nie nadąża za jego zaawansowanymi zastosowaniami.

W związku z tą sytuacją, tworząc złożone aplikacje w języku *JavaScript*, programiści mogą napotkać wiele trudności. Brakuje dostępu do rozwiązań, które często uznawane są za podstawowe i niezbędne. Różne są też koncepcje i podejścia do zagadnień takich jak programowanie obiektowe czy rozwiązywanie zależności. Jednak należy podkreślić, że przy tym przeglądarka internetowa i *JavaScript* stanowią doskonałe środowisko uruchomieniowe dla aplikacji. Przeglądarka stanowi dziś podstawowe oprogramowanie praktycznie każdego komputera. Aplikacje napisane w języku *JavaScript* są z natury wieloplatformowe, gdyż wykonywane przez interpreter. Co więcej, przez wieloplatformowość można rozumieć nie tylko różne systemy operacyjne komputerów osobistych ale także urządzenia mobilne, które również posiadają zaawansowane przeglądarki. Aplikacje *JavaScript* nie wymagają również od użytkownika instalacji, uprawnień administratora, a aktualizacje są niezwykle łatwe do przeprowadzenia. Jest to potencjał, który zdecydowanie jest wart poradzenia sobie z problemami wymienionymi wcześniej. Szczególnie dla aplikacji

edukacyjnej, która musi być łatwo dostępna dla jak najszerszego grona odbiorców.

Poniżej zaprezentowane są najważniejsze zagadnienia związane z implementacją symulatora *Energy2D*. Jej głównymi założeniami było zniwelowanie wad wymagającego środowiska przeglądarki internetowej oraz optymalne wykorzystanie jego zalet.

3.1. Język programowania – JavaScript

JavaScript nie narzuca żadnego stylu programowania. Jest on językiem skryptowym, zorientowanym obiektowo lecz o prototypowym modelu dziedziczenia. Ponadto jest dynamiczny, słabo typowany, a funkcje są obiektami pierwszoklasowymi. Doskonale wspiera wiele paradygmatów programowania, takich jak programowanie zorientowane obiektowo, imperatywne czy też funkcyjne.

W związku z tym twórcy aplikacji mają bardzo dużą dawkę elastyczności w podejściu do strukturyzacji swojej pracy. Nic nie jest z góry narzucone i możliwe jest skorzystanie ze wszystkich cech języka w zależności od potrzeb. Jednak ta elastyczność również może stanowić przyczynę tworzenia aplikacji nieczytelnych, nieustrukturyzowanych, w sposób chaotyczny mieszających różne paradygmaty i wzorce. Czyli aplikacji trudnych w rozwoju oraz utrzymaniu. Dlatego też możliwości *JavaScript* powinny być stosowane z pełną świadomością korzyści i konsekwencji. Natomiast architektura aplikacji powinna się opierać na solidnych, sprawdzonych wzorcach, aby minimalizować omówione powyżej ryzyko.

3.2. Architektura systemu – wzorzec Model–View–Controller

Na etapie projektowania aplikacji zostały wyodrębnione dwa główne zadania, które muszą być realizowane przez aplikację:

- Obliczenia fizyczne.
- Wizualizacja.

Zarówno silniki fizyczne wraz z potrzebnymi modelami danych jak i metody wizualizacji wymagają dość złożonych algorytmów. Oczywistym stało się, że należy starać się stworzyć jak najbardziej niezależne moduły odpowiedzialne za realizację każdego z tych zadań. Jest to możliwe – wizualizacja korzysta z wyników obliczeń fizycznych, jednak same obliczenia o fakcie istnienia wizualizacji nie muszą nic wiedzieć. Umożliwiło to wprowadzenie jasnego podziału na luźno powiązane grupy komponentów, które realizują dwa wymienione wcześniej cele.

Uwzględniając powyższe obserwacje, ostateczna architektura aplikacji powstała na bazie wzorca projektowego *Model-View-Controller*. Schemat ideowy systemu *Energy2D* przedstawia rysunek 3.1. Użyty wzorzec projektowy jest powszechnie stosowany w aplikacjach, które posiadają graficzny interfejs użytkownika. Skupia się na oddzieleniu logiki biznesowej oraz model danych od ich reprezentacji. Ujmując to ogólniej, zapewnia rozdzielenie obowiązków (ang. Separation of Concerns[1]). Wpisuje się to doskonale we wspomniane powyżej potrzeby symulatora *Energy2D*, dlatego też właśnie ten wzorzec został użyty jako podstawa architektury.

Wyraźny podział funkcjonalności przyniósł wiele korzyści, z których najważniejsze to:

- Czytelna, jasna organizacja kodu.
- Łatwe utrzymanie aplikacji.
- Niepowiązane, samodzielne obiekty, które realizują zadania ogólne i mogą być ponownie użyte (np. większość wizualizacji).
- Brak związania silników fizycznych z przeglądarką internetową. Wynika to właśnie z oddzielenia od warstwy prezentacji. Dzięki temu możliwe jest użycie efektywnych testów zautomatyzowanych. Ta kwestia jest omówiona szerzej w sekcji 3.4.

Ponadto, należy podkreślić, iż architektura *Energy2D* skupia się na stworzeniu grupy jak najbardziej niezależnych obiektów i modułów, które są ze sobą powiązane czytelną siecią zależności oraz realizują jasno określone zadania.

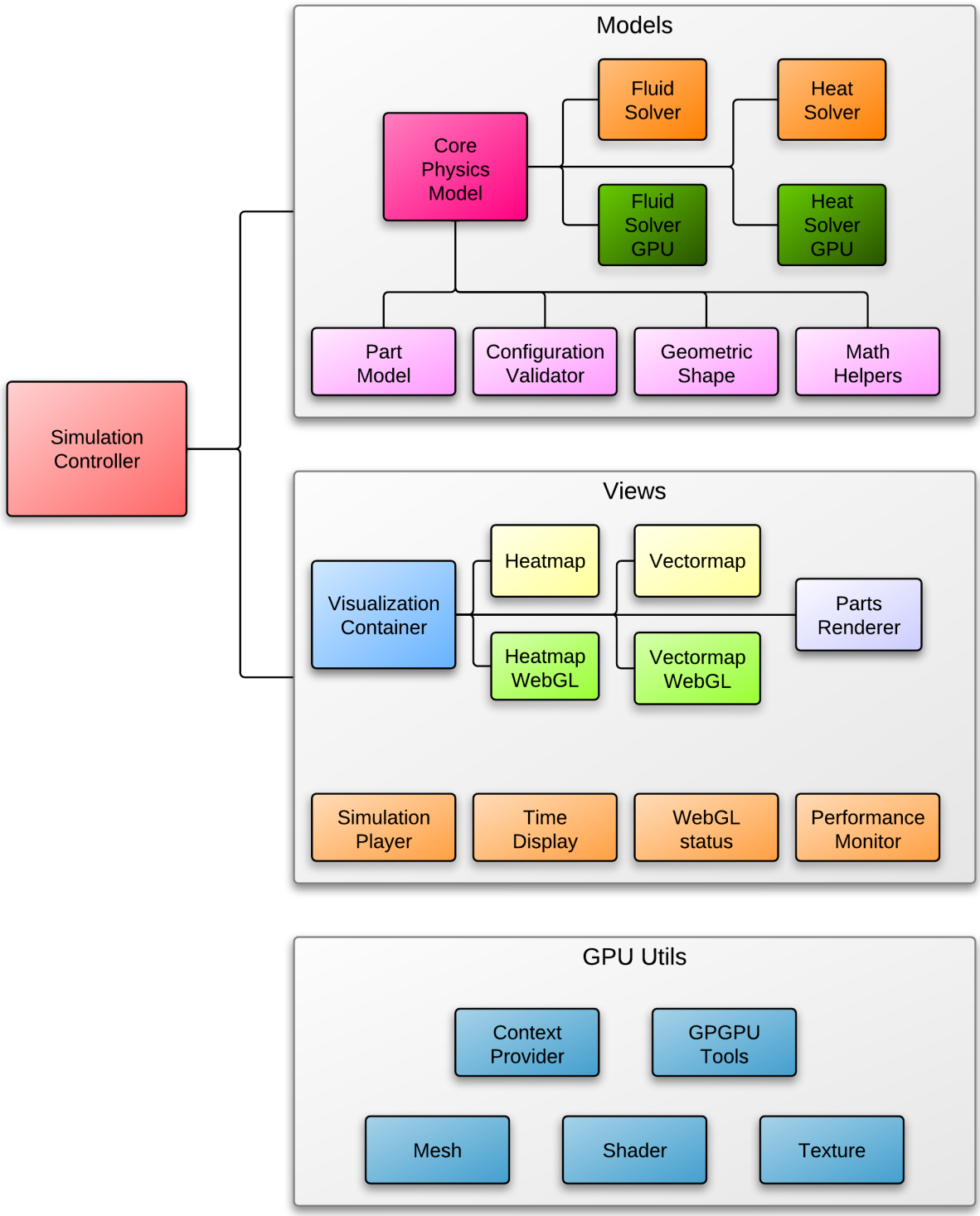
3.3. Przegląd najistotniejszych jednostek symulatora

W strukturze aplikacji można wyróżnić kilka wyraźnych grup charakteryzujących się odmiennymi celami zasadniczymi. Wizualizacje tej struktury przedstawia rysunek 3.1, a kolejne sekcje przybliżają poszczególne grupy oraz ich najważniejsze składniki.

3.3.1. Modele

Jest to zbiór obiektów odpowiedzialnych za przechowywanie i manipulowanie danymi aplikacji. To właśnie w tej grupie znajdują się kluczowe dla całej aplikacji silniki fizyczne odpowiadające za symulację przewodnictwa cieplnego oraz dynamiki płynów.

[1] `http://en.wikipedia.org/wiki/Separation_of_concerns`



Rysunek 3.1: Schemat ideowy architektury symulatora *Energy2D*

Core Physics Model tworzy i zarządza wszystkimi niezbędnymi dla obliczeń fizycznych danymi. Można do nich zaliczyć siatki symulacyjne (reprezentowane przez tablice *JavaScript* bądź też dwuwymiarowe tekstury *WebGL*) oraz zestaw parametrów charakteryzujących warunki początkowe symulacji. Parametry te są definiowane przez specjalny plik konfiguracyjny w formacie *JSON*, dlatego też można stworzyć bardzo

wiele przypadków modelujących różnorakie zjawiska fizyczne. Do zarządzania konfiguracją wykorzystywany jest jeden z pomocniczych obiektów **Configration Validator**. Obliczenia fizyczne nie są przeprowadzane przez sam obiekt *Core Physics Model*. Deleguje on te zadania do innych obiektów, tym samym realizując wzorzec projektowy strategii.

Heat Solver oraz Fluid Solver to obiekty implementujące algorytmy fizyczne przewodnictwa cieplnego oraz dynamiki płynów przedstawione w sekcji 2.5. Implementacja jest wyłącznie w języku *JavaScript* dlatego też obliczenia wykonywane są na procesorze głównym komputera.

Heat Solver GPU oraz Fluid Solver GPU to odpowiedniki obiektów przedstawionych powyżej, jednak implementujące algorytmy fizyczne przy wykorzystaniu technologii *WebGL*. Stąd, główne obliczenia wykonywane są na procesorze karty graficznej. Zapewnia to wielokrotnie lepszą wydajność. Jednak jako, iż *WebGL* jest technologią dość nową, obiekty te implementują identyczną logikę jak wersje bazowe i są używane wyłącznie jeżeli konfiguracja użytkownika spełnia odpowiednie wymagania.

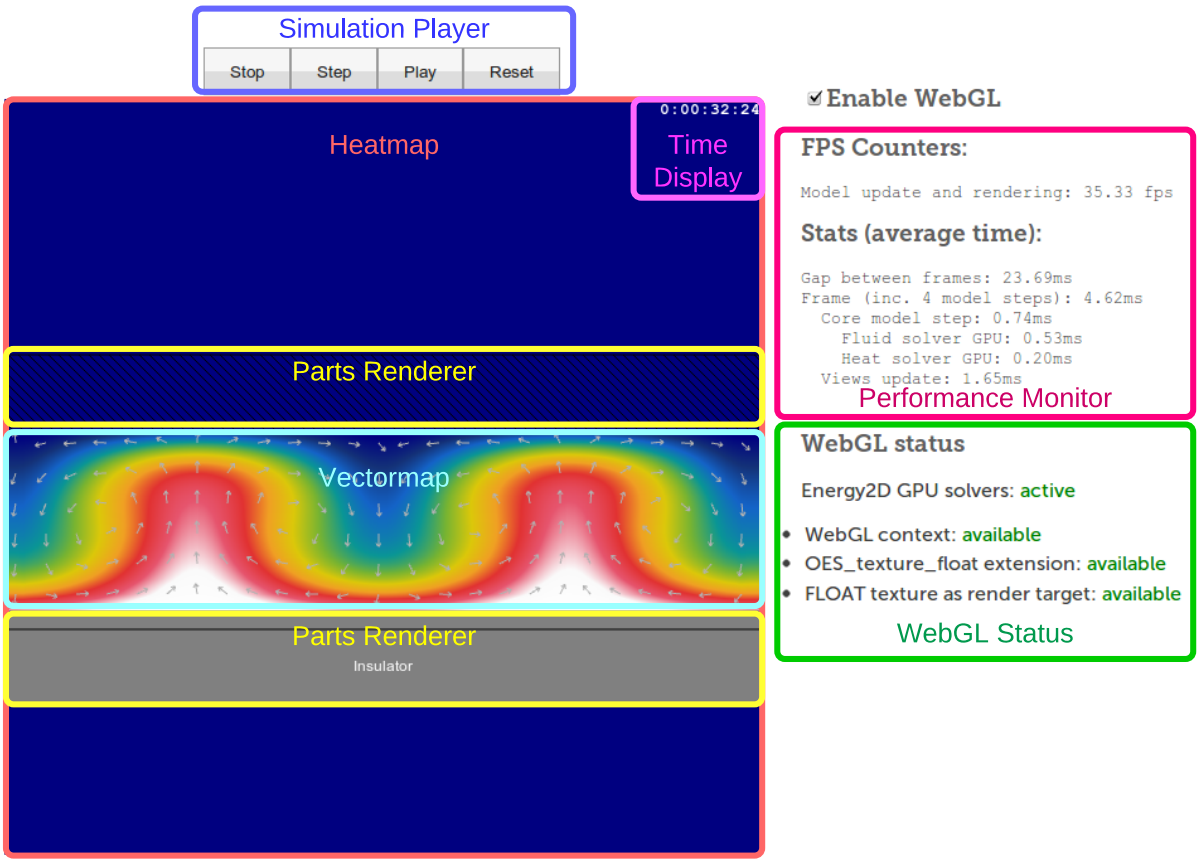
Part Model to obiekt reprezentujący element symulacji nie będący płynem (cieczą lub gazem). Dzięki obecności takich obiektów, możliwe jest stworzenie ciekawych warunków początkowych odzwierciedlających rzeczywistość. Zadaniem elementów stałych symulacji jest tworzenie barier przepływu dla płynów. Ponadto, elementy stałe symulacji mogą przewodzić ciepło co dodatkowo wzbogaca symulacje i pozwala zaprezentować np. efekt wpływu pojemności cieplnej na zdolność przewodzenia ciepła.

Geometric Shape oraz Math Helpers to obiekty pomocnicze, wykorzystywane głównie przez obiekty *Part Model*. Definiują różne reguły matematyczne pomocne przy reprezentowaniu i wizualizowaniu kształtów geometrycznych.

3.3.2. Widoki

Jest to zbiór obiektów odpowiedzialnych za aspekt wizualny symulacji. Ich głównym zadaniem jest wizualizacja czyli prezentowanie przedstawionych powyżej modeli. Przykładowy zrzut ekranu aplikacji *Energy2D* wraz z zaznaczonymi obszarami wygenerowanymi przez poszczególne widoki prezentuje rysunek 3.2.

Heatmap to jeden z najważniejszych widoków, gdyż jest odpowiedzialny za wizualizację temperatury. To dzięki temu możliwe jest obserwowanie zjawisk zachodzących podczas symulacji. Wzorowany jest na obrazie kamer termowizyjnych. Dostępnych jest także kilka palet kolorów.



Rysunek 3.2: Prezentacja poszczególnych widoków aplikacji *Energy2D*

Vectormap to widok odpowiedzialny za wyświetlanie strzałek prezentujących prędkość i kierunek przepływu płynu. Umożliwia to lepsze zrozumienie zjawisk prezentowanych podczas symulacji.

Heatmap WebGL oraz Vectormap WebGL to odpowiedniki powyższych widoków, lecz korzystające z technologii *WebGL*. Dzięki temu nie jest potrzebny czasochłonny transfer danych z tekstury w pamięci karty graficznej do pamięci głównej komputera.

Parts Renderer to widok mający za zadanie prezentacje elementów stałych symulacji (czyli niebędących płynem).

Visualization Container to obiekt grupujący powyższe widoki, zapewniający odpowiednie ich ułożenie oraz udostępniający pomocniczy interfejs do zarządzania poszczególnymi podwidokami.

Simulation Player umożliwia sterowanie przebiegiem symulacji (uruchomienie, zatrzymanie, odtwarzanie po jednym kroku, resetowanie do stanu początkowego).

WebGL Status ma zadanie informacyjne, prezentuje czy konfiguracja użytkownika wspiera technologię *WebGL* oraz niezbędne jej rozszerzenia.

Performance Monitor prezentuje dane dotyczące wydajności, takie jak ilość wyświetlanych klatek na sekundę oraz czasy wykonywania poszczególnych etapów algorytmu. Dzięki niemu było możliwe przeprowadzenie testów wydajnościowych omówionych w rozdziale 5.2.

3.3.3. Kontroler

Aplikacja *Energy2D* definiuje tylko jeden kontroler – **Simulation Controller**. Jest on odpowiedzialny za wiele zadań, ale przede wszystkim stanowi spoiwo łączące moduły odpowiedzialne za przeprowadzanie obliczeń fizycznych oraz moduły odpowiedzialne za wizualizację. Zarządza relacjami między tymi obiektami oraz steruje również przebiegiem symulacji.

3.3.4. Narzędzia pomocnicze związane z GPU

Podczas pracy nad optymalizacją polegającą na przeniesieniu obliczeń fizycznych na GPU (por. rozdział 4), pojawiła się potrzeba stworzenia narzędzi, które stanowiłyby opakowanie dla niskopoziomowych funkcji API *WebGL*.

Shader, Texture oraz Mesh to obiekty przeznaczone do zarządzania typowymi dla programowania grafiki trójwymiarowej strukturami – odpowiednio: programami jednostek cieniujących, teksturami oraz geometrią.

GPGPU Tools to moduł udostępniający zestaw funkcji ułatwiających wykonywanie obliczeń ogólnego przeznaczenia na procesorze karty graficznej. Jest głównie wykorzystywany przez obiekty takie jak *Fluid Solver GPU* oraz *Heat Solver GPU*.

3.4. Zgodność ze środowiskiem Node.js oraz przeglądarką internetową

Node.js[2] to dynamicznie rozwijające się środowisko *JavaScript*, stworzone na bazie silnika *V8* opracowanego na potrzeby przeglądarki internetowej Google Chrome. Aby skrypt mógł być wykonywany w tym środowisku, musi być niezależny od cech charakterystycznych dla środowiska przeglądarki internetowej.

Niezależność od przeglądarki można zdefiniować jako niekorzystanie z metod globalnie dostępnych obiektów takich jak *window* czy też *document*. Innym działaniem, które wiąże skrypt z przeglądarką i tradycyjnym zastosowaniem jest np. trawersowanie (oraz

[2]Strona domowa projektu: `http://nodejs.org/`

ewentualne modyfikowanie) drzewa HTML DOM. Jednak, przy założeniu o niedostępności zewnętrznych bibliotek, można to sprowadzić do nieużywania metod wspomnianego powyżej obiektu *document*.

Poprzedni podrozdział zaprezentował modułową budowę symulatora *Energy2D* na bazie wzorca projektowego *Model-View-Controller*. Architektura opierająca się na jak najbardziej niezależnych modułach, pełniących jasno zdefiniowaną funkcję jest ważna nie tylko ze względu na łatwiejszy rozwój i utrzymanie aplikacji. To podejście pozwoliło wyodrębnić jednostki symulatora, które są zupełnie niezależne od przeglądarki wg. kryteriów zdefiniowanych powyżej. Warunek ten spełnia grupa obiektów przedstawiona wcześniej jako „Modele" (por. 3.3.1). Dzięki niezależności od przeglądarki, możliwe stało się używanie tych obiektów w przedstawionym powyżej środowisku *Node.js*.

Główną korzyścią jaka z tego płynie jest możliwość stosowania wydajnych, efektywnych testów zautomatyzowanych. Istnieją narzędzia umożliwiające testowanie kodu *JavaScript* związanego z przeglądarką, ale są to rozwiązania znacznie mniej wydajne oraz wygodne. Oparcie testów na skryptach wykonywanych przez *Node.js* pozwoliło zintegrować testy z system ciągłej integracji *Travis Continuous Integration*[3]. Ponadto, ułatwione zostało ewentualne użycie zaawansowanych silników fizycznych *Energy2D* przez inne aplikacje działające w środowisku *Node.js*.

3.5. Zarządzanie zależnościami w złożonych aplikacjach JavaScript

Język *JavaScript* natywnie nie wspiera żadnego mechanizmu zarządzania zależnościami. Nie posiada instrukcji takich jak *import* czy *require*, znanych z innych popularnych języków programowania, które pozwalają organizować źródła aplikacji. W związku z tym twórcy aplikacji są zmuszeni rozwiązać ten problem własnoręcznie. Najpopularniejsze podejścia do tego problemu to:

- Manualne zarządzanie zależnościami przez tagi *<script>* bezpośrednio w kodzie źródłowym strony HTML. Jest to rozwiązanie najprostsze jednak programista jest zmuszony do ręcznego śledzenia zależności oraz zdefiniowania skryptów w odpowiedniej kolejności. Z tego też powodu sprawdza się w praktyce wyłącznie w małych, prostych aplikacjach.

- Własne skrypty budujące aplikację, najczęściej poprzez zautomatyzowane łączenie poszczególnych plików źródłowych w jeden plik wynikowy. Wykorzystywane są do

[3]Strona domowa projektu: `http://travis-ci.org/`

tego różne technologie. We wczesnym etapie rozwoju aplikacji *Energy2D* właśnie tak zarządzano zależnościami. Wykorzystywaną technologią był program *make*, dlatego pliki źródłowe zdefiniowane były w pliku *Makefile*. Podejście to jest znacznie lepsze niż poprzednie. Przede wszystkim pozwala zbudować jeden wynikowy plik, który wystarczy dołączyć do strony HTML. Możliwe staje się łatwe rozpowszechnianie aplikacji. Jednak wciąż programista musi ręcznie śledzić i definiować zależności oraz zapewniać ich rozwiązywanie.

- Technologie dedykowane. Niwelują one wady poprzednich podejść poprzez automatyzacje całego procesu. Przykładem może być definicja modułów wg. standardu CommonJS, używana np. w środowisku *Node.js*. Innym standardem jest AMD – Asynchronous Module Definition. Potrzebne pliki źródłowe (moduły) są wczytywane asynchronicznie, można też zadbać o to, żeby były dołączane wyłącznie wtedy kiedy faktycznie są potrzebne. Implementacje tej technologii istnieją zarówno dla przeglądarki jak i środowiska *Node.js*. Dedykowane technologie same rozwiązują zależności. Programista tylko je definiuje, dlatego też jest to rozwiązanie korzystne (czasami wręcz niezbędne) w przypadku złożonych aplikacji.

W symulatorze *Energy2D* wybrane zostało ostatnie z omówionych podejść. Zastosowano implementację standardu AMD w postaci biblioteki RequireJS[4]. Wybór samego podejścia jest uzasadniony dużą złożonością aplikacji. Natomiast wybór tego konkretnego standardu definicji modułów oraz biblioteki, która go implementuje podyktowany był przede wszystkim zgodnością zarówno z przeglądarką internetową jak i środowiskiem *Node.js*.

RequireJS posiada wiele alternatyw, jednak ta konkretna biblioteka jest rozwijana i wspierana od długiego czasu, posiada ugruntowaną pozycję i rokuje długie wsparcie w przyszłości. Definiowanie modułów odbywa się przez instrukcję *define* o następującej postaci:

```
1 define(['helper/utils', 'mymodule'], function (utils, mymodule) {
2     var api = { /*...*/ };
3     return api;
4 });
```

Listing 3.1: Definicja modułu przy użyciu technologii RequireJS

Dopuszczalna jest też alternatywna składnia, zbliżona do standardu CommonJS:

```
1 define(function (require) {
2     var utils    = require('helper/utils'),
3         mymodule = require('mymodule'),
```

[4]Strona domowa biblioteki: `http://requirejs.org/`

```
4         api = { /*...*/ };
5     return api;
6 });
```

Listing 3.2: Alternatywna składnia definicji modułu przy użyciu technologii RequireJS

Odpowiada ona dokładnie wersji podstawowej przedstawionej powyżej. W kodzie *Energy2D* jest używana najczęściej ze względu na subiektywne wrażenie większej czytelności. Ponadto, umożliwia bardzo łatwe konwertowanie modułów napisanych pierwotnie w standardzie CommonJS, co również było użyteczne podczas pracy nad aplikacją *Energy2D* (gdyż pierwotnie silniki fizyczne były właśnie modułami zdefiniowanymi w standardzie CommonJS).

4. Przeniesienie obliczeń fizycznych na procesor karty graficznej

Niniejszy rozdział prezentuje techniki zastosowane w celu przeniesienia głównych obliczeń fizycznych na kartę graficzną w symulatorze *Energy2D* będącym przedmiotem tej pracy. Na początku opisane są tradycyjne podejścia do tego problemu dla aplikacji działających w natywnym środowisku systemu operacyjnego oraz ich odniesienie do środowiska oferowanego przez przeglądarki internetowe. Następnie zaprezentowane zostały najważniejsze zagadnienia dotyczące implementacji równoległych silników fizycznych *Energy2D* działających na procesorze karty graficznej. Informacje te mogą być szczególnie użyteczne przy próbach podobnych optymalizacji innych aplikacji.

Dzięki przeniesieniu obliczeń fizycznych na GPU uzyskano istotny wzrost wydajności. Dokładna analiza zysków ze zrównoleglenia symulacji jest przedstawiona w rozdziale 5.

4.1. Typowe metody przenoszenia obliczeń na GPU, a przeglądarka internetowa

Współczesna karty graficzne posiadają ogromną moc obliczeniową – wielokrotnie większą od centralnego procesora przy założeniu, że obliczenia da się wykonywać w sposób równoległy. Pomysł aby przenieść część obliczeń ogólnego zastosowania na kartę graficzną pojawił się wraz z dynamicznym rozwojem procesorów graficznych. Szczególnie istotnym momentem było wprowadzenie programowalnych jednostek cieniujących (specyfikacja *DirectX 8*). Dalszy rozwój obliczeń ogólnego zastosowania na kartach graficznych (ang. *General-Purpose Computing on Graphics Processing Units*, w skrócie GPGPU) miał miejsce wraz z wprowadzeniem technologii, które ukryły złożoność dostępu do zasobów karty graficznej i udostępniły interfejs wysokiego poziomu. Wiodącymi technologiami tego typu są *OpenCL* (rozwiązanie otwarte) oraz *CUDA* (zamknięte rozwiązanie firmy NVIDIA, działające wyłącznie na sprzęcie tego producenta).

Wspomniane powyżej technologie dotyczą oczywiście aplikacji pisanych w natywnym środowisku systemu operacyjnego. Aby programować jednostki cieniujące wystarczy pod-

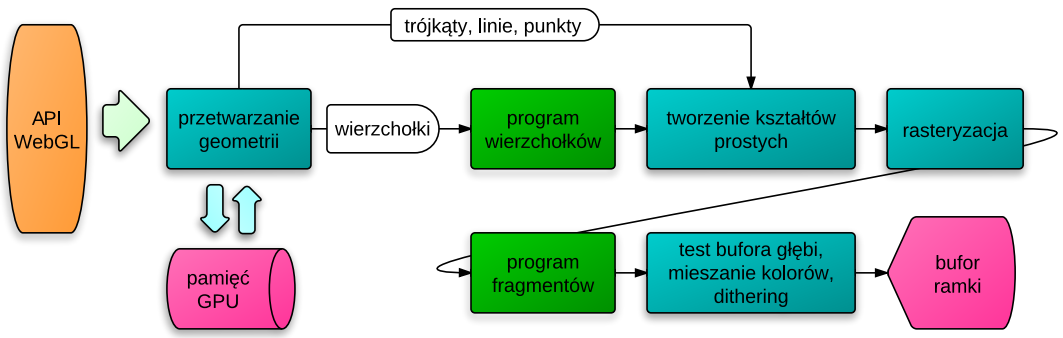
stawowy dostęp do standardowego interfejsu *OpenGL* bądź *Direct3D*. Implementacje tych interfejsów można znaleźć dla prawie każdego współczesnego języka programowania. Interfejsy wyższego poziomu (*OpenCL*, *CUDA*) również posiadają implementacje w różnych językach programowania, choć najczęstszym środowiskiem ich działania są aplikacje napisane w C bądź C++.

Poniżej krótko przedstawione są sposoby przeprowadzania obliczeń ogólnego zastosowania na kartach graficznych oraz ich związek ze specyficznym środowiskiem przeglądarki internetowej.

4.1.1. Niskopoziomowe programowanie jednostek cieniujących

Jest to najstarsze podejście do przeprowadzania obliczeń ogólnego zastosowania na procesorze karty graficznej. Programista w swoisty sposób „oszukuje" kartę graficzną, przeprowadzając renderowanie prostej geometrii wyłącznie w celu uruchomienia własnych programów jednostek cieniujących, które wykonują obliczenia często nie mające nic wspólnego z generowaniem obrazu.

W przypadku technologii *WebGL*, która posłużyła do zaimplementowania silników fizycznych *Energy2D*, diagram potoku renderowania prezentuje rysunek 4.1. Zielonym kolorem zostały oznaczone procesy, które są programowalne. Są to dwa rodzaje programów jednostek cieniujących – wierzchołków oraz fragmentów. Nie ma możliwości bezpośredniego sterowania pozostałymi procesami. Odbywają się one automatycznie, ewentualnie możliwa jest konfiguracja pewnych parametrów. Dlatego też cały algorytm przeznaczony do zrównoleglenia musi zostać zapisany wyłącznie przy użyciu programów wierzchołków oraz fragmentów. Ponadto dane powinny się znajdować w pamięci karty graficznej, najczęściej w postaci dwuwymiarowych tekstur.



Rysunek 4.1: Diagram potoku renderowania *WebGL* / *OpenGL 3.x*

Takie podejście jednak jest wymagające i obarczone pewnymi problemami. Łatwo popełnić błędy, wymagana jest też przynajmniej podstawowa wiedza o programowaniu grafiki trójwymiarowej. Najważniejsze koncepcje niskopoziomowego programowania jednostek cieniujących zostały doskonale przedstawione przez Marka Harrisa w jednym z jego artykułów do popularnej serii *GPU Gems* [Har05].

4.1.2. Technologie wyższego poziomu

Technologie, które przyczyniły się do gwałtownego wzrostu popularności obliczeń na kartach graficznych to szczególnie *OpenCL* oraz *CUDA*. Udostępniają one znacznie wyższy poziom abstrakcji – programista jest zwolniony z obowiązku przyswojenia sobie niskopoziomowych mechanizmów rządzących działaniem kart graficznych (choć ta wiedza pozwala tworzyć aplikacje efektywniejsze). Udostępniony jest specjalny interfejs oraz składnia, co znacznie ułatwia pracę, szczególnie programistom bez doświadczenia w pracy z programowaniem grafiki trójwymiarowej.

4.1.3. Możliwości w środowisku przeglądarki internetowej

Należy uściślić, że „środowisko przeglądarki internetowej" to zbiór możliwości nowoczesnych, wiodących przeglądarek internetowych, rozpowszechniony na tyle, aby był dostępny dla większości użytkowników internetu. Równocześnie możliwości te nie mogą wymagać instalowania żadnych dodatków i rozszerzeń, gdyż znacznie ogranicza to ich dostępność. Jest to szczególnie istotne dla aplikacji edukacyjnych, które wymagają łatwego i powszechnego dostępu.

Przeglądarka internetowa z założenia oferuje środowisko bardzo ograniczone, przede wszystkim ze względów bezpieczeństwa. Jednak niedawno, wraz z nadejściem standardu HTML5, został dodany podstawowy dostęp do zasobów karty graficznej. Realizuje go technologia WebGL, która jest implementacją standardu *OpenGL ES 2.0*. Tym samym pojawiła się możliwość programowania jednostek cieniujących kart graficznych, a więc i przeprowadzania na nich obliczeń ogólnego zastosowania (4.1.1).

Technologie programowania kart graficznych wyższego poziomu (4.1.2) nie są (jeszcze) dostępne w przeglądarce internetowej. Aktualnie trwają intensywne prace nad implementacją standardu OpenCL o roboczej nazwie WebCL. Jednak technologia ta w aktualnym momencie jest na bardzo wczesnym etapie rozwoju. Więcej informacji można znaleźć na stronie internetowej: `http://www.khronos.org/webcl/`.

Dlatego też jedyną metodą na przeprowadzenie obliczeń ogólnego zastosowania na procesorze karty graficznej w środowisku przeglądarki internetowej jest sposób zaprezentowany w sekcji 4.1.1. Właśnie taka, niskopoziomowa metoda została zastosowana dla

silników fizycznych symulatora *Energy2D*. Opis najważniejszych zagadnień związanych z implementacją prezentuje następna sekcja (4.2).

4.2. Implementacji silników fizycznych Energy2D przy użyciu WebGL

Symulator *Energy2D* składa się z dwóch kluczowych silników fizycznych - przewodnictwa cieplnego oraz dynamiki płynów. Są one dokładnie przybliżone w sekcji 2.5. Oba te silniki są niezwykle wymagające obliczeniowo. Dlatego też ich optymalizacja była zadaniem kluczowym, aby stworzyć aplikację wartościową edukacyjnie. Zbyt wolny przebieg symulacji może skutecznie zniechęcić większość potencjalnych użytkowników. Jednym z podstawowych wymagań było, aby wirtualne laboratorium było aplikacją w pełni interaktywną czyli również działającą jak najpłynniej.

Poniżej przedstawione są najważniejsze zagadnienia związanie z przeniesieniem obliczeń fizycznych *Energy2D* na kartę graficzną.

4.2.1. Analiza algorytmów silników fizycznych pod kątem przetwarzania równoległego

Wszystkie algorytmy rozwiązujące równania fizyczne w symulatorze *Energy2D* zostały poddane analizie pod kątem możliwości równoległego przetwarzania. Zostały wyróżnione następujące kryteria, które algorytm musi spełniać, aby było to możliwe:

- Możliwość reprezentowana danych w pamięci karty graficznej.
- Niezależność wyniku algorytmu od sekwencji wykonywania obliczeń.

Jeżeli kryteria te są spełnione, powinna istnieć możliwość zaimplementowania algorytmu w sposób równoległy. Oczywiście zmienia się strona techniczna, użyte rozwiązania, język programowania oraz techniki, jednak sama koncepcja algorytmu i główne kroki powinny pozostać bez większych zmian. Problem pojawia się wtedy, gdy któryś z algorytmów nie spełnia jednego z tych kryteriów. W takim przypadku konieczne są gruntowne modyfikacje lub rezygnacja z implementacji takiego algorytmu.

Pierwszy warunek spełniają wszystkie algorytmy, jako że obliczenia są wykonywane na prostokątnych siatkach symulacyjnych. Można je reprezentować w pamięci karty graficznej przy użyciu dwuwymiarowych tekstur. Temat organizacji danych w pamięci GPU oraz związane z tym problemy porusza sekcja 4.2.4.

Drugi warunek nie został spełniony przez wszystkie zastosowane algorytmy. Problematyczne okazały się metody rozwiązywania układów równań liniowych metodą relaksacji

Gaussa-Seidela (por. [MC89]). Podczas jednego przebiegu przez wszystkie komórki macierzy, nowa wartość komórki (i, j) jest zależna od wartości komórek sąsiednich. Następnie komórka macierzy jest niezwłocznie aktualizowana. Powoduje to, iż przy sekwencyjnym przetwarzaniu wszystkich komórek, nowa wartość dla każdej komórki jest zależna od wartości obliczonych zarówno w poprzednim kroku relaksacji jak i kroku aktualnym. W sposób uproszczony schemat takiej relaksacji prezentuje algorytm 4.1.

Algorytm 4.1 Relaksacja metodą Gaussa-Seidela na CPU

for $0 \rightarrow relaxation\_steps$ **do**
  for all grid cells **do**
    $new\_val \leftarrow f(x_{i,j}, x_{i+1,j}, x_{i-1,j}, x_{i,j+1}, x_{i,j-1})$
    $x_{i,j} \leftarrow new\_val$
  end for
end for

Niestety, przy obliczeniach równoległych nie można polegać na takiej zależności. Każda komórka zostanie zaktualizowana na podstawie wartości komórek sąsiednich wyłącznie z poprzedniego kroku relaksacji. Co więcej, ograniczeniem są też kwestie czysto technologiczne - tekstury w których przechowywane są dane mogą być podczas obliczeń wyłącznie przeznaczone do odczytu lub do zapisu. Tak więc niemożliwy jest jednoczesny odczyt z danej tekstury oraz natychmiastowy zapis.

Problem ten został rozwiązany przez zmianę algorytmu rozwiązywania układów równań liniowych na metodę relaksacji *Jacobiego*. Główną różnicą jest moment aktualizowania komórek siatki symulacji. W przeciwieństwie do metody *Gaussa- Seidela* nie następuje to natychmiast po obliczeniu nowej wartości dla danej komórki, ale dopiero obliczeniu nowych wartości dla wszystkich komórek. Uproszczony schemat tej relaksacji prezentuje algorytm 4.2.

Algorytm 4.2 Relaksacja metodą Jacobiego na CPU

for $0 \rightarrow relaxation\_steps$ **do**
  for all grid cells **do**
    $temp_{i,j} \leftarrow f(x_{i,j}, x_{i+1,j}, x_{i-1,j}, x_{i,j+1}, x_{i,j-1})$
  end for
  for all grid cells **do**
    $x_{i,j} \leftarrow temp_{i,j}$
  end for
end for

Taki algorytm można już przetworzyć na wersję równoległą. Macierze x oraz *temp* zostają zamienione na odpowiednie tekstury. Również, w celach wydajnościowych zawartość tekstur nie jest przepisywana tylko zamieniane są ich referencje. Uproszczony schemat relaksacji metodą *Jacobiego* na GPU prezentuje algorytm 4.3.

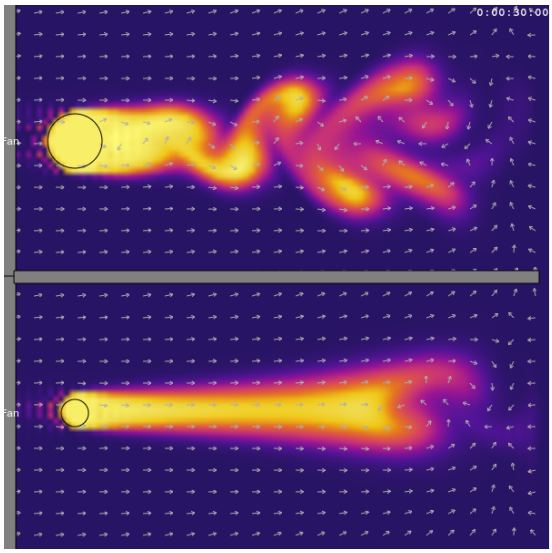
Algorytm 4.3 Relaksacja metodą Jacobiego na GPU

for $0 \rightarrow relaxation\_steps$ **do**
  for all grid cells **do** [IN PARALLEL]
    $temp_{i,j} \leftarrow f(x_{i,j}, x_{i+1,j}, x_{i-1,j}, x_{i,j+1}, x_{i,j-1})$
  end for
  swap x with *temp*
end for

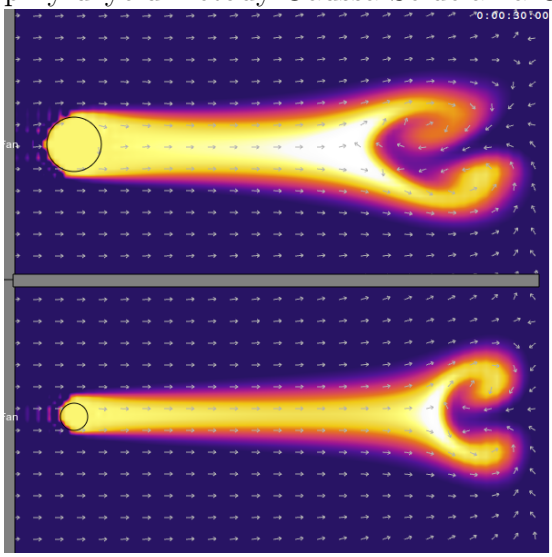
Oczywiście zmiana algorytmu rozwiązywania równań liniowych niesie ze sobą pewne konsekwencje. Inna jest konwergencja tych dwóch algorytmów. Metoda *Gaussa-Seidela* jest szybciej zbieżna niż metoda *Jacobiego*. Efekty symulacji dla różnej konfiguracji algorytmów rozwiązywania układów równań liniowych prezentują rysunki 4.2, 4.3 oraz 4.4.

Na ww. rysunkach można zaobserwować wyraźne różnice w rezultatach symulacji. W przypadku przedstawionej symulacji oczekiwanym wynikiem było powstane wiru Kármána dla większej przeszkody. Przy implementacji równoległej metody *Jacobiego* widać, iż porządany efekt symulacji zostaje utracony. Dlatego też należy wykonać więcej kroków relaksacji. Okazało się, że wartością wystarczającą jest dziesięć. Wartość ta została ustalona empirycznie, tak aby wyniki symulacji odpowiadały oczekiwaniom oraz były jak najbardziej zbliżone do rezelutatów sekwencyjnych silików fizycznych. Jest to niezbędne, ponieważ wersja równoległa aplikacji *Energy2D* powinna być w pełni zgodnym i kompatybilnym rozszerzeniem wersji podstawowej (sekwencyjnej, napisanej bez użycia technologii *WebGL*). Oczywiście wiąże się to ze spowolnieniem symulacji, jednak w żadnym wypadku nie neguje opłacalności przeniesienia obliczeń na kartę graficzną – relaksacja na GPU jest wciąż dużo szybszych niż na CPU mimo konieczności wykonania dwukrotnie większej liczby kroków.

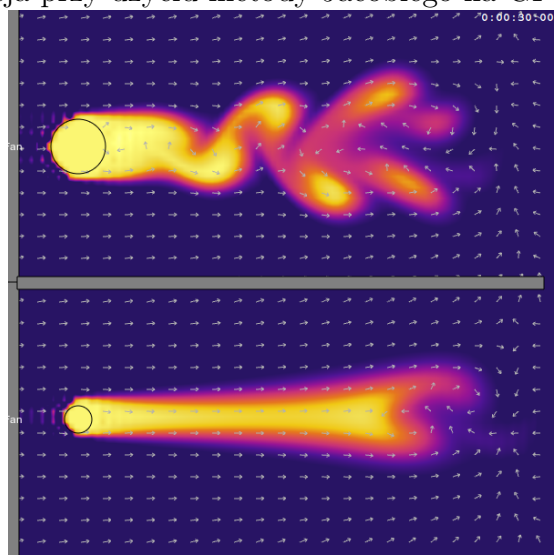
Istnieją również implementacje algorytmu *Gaussa-Seidela* na maszyny równoległe. Nie są to dokładne kopie wersji sekwencyjnej, lecz imitują obliczanie nowych wartości komórek na podstawie wartości z kroku relaksacji poprzedniego i aktualnego. Doskonałe opracowanie zagadnień algorytmów rozwiązujących układy równań liniowych na CPU oraz GPU, które okazało się niezwykle przydatne podczas implementacji równoległych algorytmów dla *Energy2D*, zostało przygotowane przez G. Amadora oraz A. Gomesa [AG09]. Niestety niskopoziomowa implementacja algorytmu *Gaussa-Seidela* na GPU wymaga wykonania



Rysunek 4.2: Symulacja przy użyciu metody Gaussa-Seidela na CPU, 5 kroków relaksacji



Rysunek 4.3: Symulacja przy użyciu metody Jacobiego na GPU, 5 kroków relaksacji



Rysunek 4.4: Symulacja przy użyciu metody Jacobiego na GPU, 10 kroków relaksacji

dwóch procesów renderowania do tekstury dla jednego kroku relaksacji – powoduje to narzut czasowy, który w efekcie niweczy zysk z szybszej konwergencji algorytmu.

4.2.2. Podstawowy zarys implementacji

Zgodnie z wnioskami sekcji 4.1.3, ze względu na ograniczenia środowiska przeglądarki internetowej, wymuszona została implementacja niskopoziomowa przy użyciu technologii *WebGL*. Opiera się ona na programowaniu jednostek cieniujących karty graficznej, głównie wykorzystując programy fragmentów (ang. fragment programs). Taka metoda przeprowadzania obliczeń ogólnego przeznaczenia na karcie graficznej wymaga zrozumienia podstawowych zagadnień związanych z programowaniem grafiki trójwymiarowej (por. sekcja 4.1.1).

W przypadku implementacji *Energy2D* podstawowy schemat przeprowadzania obliczeń fizycznych na GPU z wykorzystaniem technologii *WebGL* wygląda następująco:

- Przygotowanie oraz kompilacja programów wierzchołków oraz fragmentów, które zawierają implementację algorytmów silników fizycznych.
- Przygotowanie tekstur, które przechowują dane symulacyjne.
- Przygotowanie danych geometrii płaszczyzny pokrywającej cały zakres współrzędnych, które mieszczą się w obszarze renderowania.
- Wykonywanie kolejnych kroków algorytmów fizycznych, co sprowadza się do:
  - Renderowania wcześniej przygotowanej geometrii płaszczyzny przy użyciu wcześniej przygotowanych programów wierzchołków i fragmentów.
  - Kopiowania danych z bufora ramki do wybranej tekstury przechowującej dane symulacji [1].

Poszczególne podpunkty w szerszym zakresie przybliżają sekcje od 4.2.3 do 4.2.6.

4.2.3. Programy wierzchołków oraz fragmentów

Właściwa implementacja algorytmów w programach wierzchołków i fragmentów decyduje o poprawności oraz wydajności całej aplikacji. Bardzo często w przypadku obliczeń ogólnego zastosowania na GPU, a także w przypadku symulatora *Energy2D*, większość pracy przypada na programy fragmentów. Program wierzchołków zwykle sprowadza się do

[1] Odbywa się to z użyciem obiektu bufora ramki (ang. FrameBuffer Object) z dołączoną do niego teksturą

skopiowania wejściowych współrzędnych wierzchołka i tekstury. Nie ma potrzeby jakiejkolwiek modyfikacji geometrii renderowanej płaszczyzny. Dlatego też większość programów wierzchołków w symulatorze Energy2D odpowiada poniższej implementacji:

```
attribute vec4 vertexPos;
attribute vec4 texCoord;

varying vec2 coord;
void main() {
  coord = texCoord;
  gl_Position = vertexPos;
}
```

Listing 4.1: Typowa implementacja programu wierzchołków w symulatorze *Energy2D* (język GLSL)

Programy fragmentów, jako że wykonują właściwe obliczenia, nie są już tak trywialne. Bardzo ważne jest właściwe określenie współrzędnych tekseli tekstury. Posiadając siatkę symulacji o wymiarach NxN, należy ją zrzutować na zakres domyślnych współrzędnych tekstury, które zawierają się w przedziale [0, 1]. Jeżeli zrobi się to nieprawidłowo, trudno będzie wykryć taki błąd, gdyż domyślnie tekstury interpolują wartości leżące pomiędzy rzeczywistymi danymi. Może to prowadzić do nieoczekiwanych rezultatów, stąd niezwykle istotne jest precyzyjne określanie współrzędnych. W tym celu, praktycznie każdy program fragmentów zawierał wektor o nazwie *grid* równy (1.0 / N, 1.0 / N), gdzie NxN to wymiary siatki symulacyjnej. Dodając lub odejmując odpowiedni jego komponent można uzyskać dokładną wartość komórki sąsiedniej. Warto też pamiętać o fakcie, iż pierwsza kolumna tekseli nie ma współrzędnej X równej 0.0, lecz 0.5 / N. To samo dotyczy pierwszego rzędu współrzędnej Y równej 0.5 / N. Błędne założenie, iż te kolumny mają współrzędne 0.0 prowadzi do problemów przy wymuszaniu warunków brzegowych podczas symulacji. Ostatecznie, typowy szkielet programów fragmentów wygląda następująco:

```
uniform sampler2D simulationData;
uniform vec2 grid;
varying vec2 coord;

vec4 F(vec4 data) {
  // Funkcja wykonująca właściwe obliczenia dla danego kroku symulacji.
  // ...
}

void main() {
  vec4 data = texture2D(simulationData, coord);
  // Instrukcja warunkowa sprawdzająca czy nie są przetwarzane brzegi
     siatki.
```

```
13   if (coord.x > grid.x && coord.x < 1.0 - grid.x &&
14       coord.y > grid.y && coord.y < 1.0 - grid.y) {
15     data = F(data);
16   }
17   gl_FragColor = data;
18 }
```

Listing 4.2: Szkielet implementacji programu fragmentów w symulatorze *Energy2D* (język GLSL)

Oczywiście program, który wymuszał warunki brzegowe posiadał odwrotny warunek w liniach 13 oraz 14. Implementacja funkcji F nie jest przytoczona, gdyż nie da się wyróżnić jakiegoś ogólnego jej schematu czy wzoru. Można powiedzieć, że przenosząc dany krok algorytmu do języka GLSL, funkcja F stanowi implementację „wewnętrznych" instrukcji zagnieżdżonych pętli iterujących po wszystkich komórkach symulacji.

4.2.4. Organizacja danych w pamięci karty graficznej

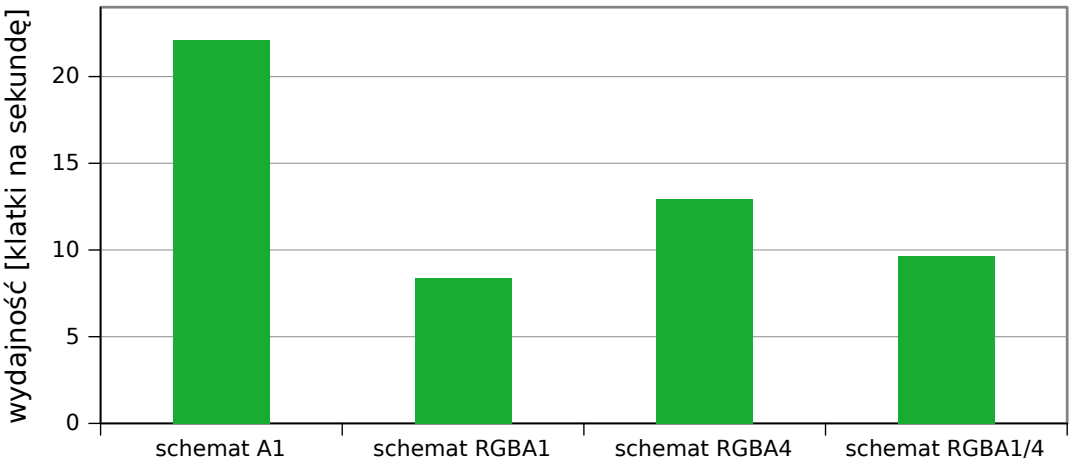
Dane symulacji (takie jak np. macierz temperatury czy macierz prędkości płynu) przechowywane są w dwuwymiarowych teksturach zmiennoprzecinkowych. Tego typu tekstury nie wchodzą w skład podstawowej specyfikacji *WebGL 1.0* ([Web11]). W związku z tym wymagane jest użycie rozszerzenia *OES_texture_float*, które jest dostępne na większości współczesnych urządzeń ([TODO: wspomnieć o rozdziale z testami]).

Niezwykle ważną kwestią jest organizacja danych w teksturze. Jest sporo możliwości ponieważ tekstura z zasady nie jest wierną kopią tablicy JavaScript, a obiektem przystosowanym do przechowywania obrazów (posiada na przykład kanały kolorów). Dlatego też można rozważyć kilka potencjalnych sposobów na rozmieszczenie danych.

- Schemat 1 – tekstury jednokanałowe (format ALPHA lub LUMINANCE), jedna tekstura odpowiada jednej tablicy JavaScript. Dalej nazywany schematem **A1** na potrzeby niniejszego opracowania.
- Schemat 2 – tekstury czterokanałowe (format RGBA), dane tylko w jednym kanale, jedna tekstura odpowiada jednej tablicy JavaScript. Dalej nazywany schematem **RGBA1** na potrzeby niniejszego opracowania.
- Schemat 3 – tekstury czterokanałowe (format RGBA), dane w każdym z kanałów, jedna tekstura odpowiada czterem tablicom JavaScript. Dalej nazywany schematem **RGBA4** na potrzeby niniejszego opracowania.
- Schemat 4 – tekstury czterokanałowe (format RGBA), dane w każdym z kanałów, jedna tekstura odpowiada jednej tablicy JavaScript, rozmiar tekstury zredukowany

czterokrotnie, gdyż każdy kanał odpowiada jednej ćwiartce tablicy. Dalej nazywany schematem **RGBA1/4** na potrzeby niniejszego opracowania.

Każdy z powyższych schematów został przetestowany podczas implementacji symulatora *Energy2D*. Wyniki testów wydajnościowych przedstawia wykres 4.5.



Rysunek 4.5: Wpływ organizacji danych w teksturach na wydajność symulacji *Energy2D*

Rozwiązaniem optymalnym ze względu na wydajność oraz czytelność kodu wydaje się schemat A1. Umożliwia on przeniesienie danych z tablic w relacji 1:1 do tekstur. Do tego, każdy odczyt czy zapis do tekstury dotyczy tylko jednego kanału. Niestety, na większości urządzeń nie ma możliwości dołączenia tekstury typu ALPHA lub LUMINANCE do własnego obiektu bufora ramki (ang. FrameBuffer Object) – czyli renderowania do takiej tekstury [2]. Wyklucza to możliwość użycia tego rozwiązania mimo oczywistych zalet. Być może w przyszłości rozwój specyfikacji *WebGL* na to pozwoli.

Warto tutaj nadmienić, że specyfikacja *WebGL* ([Web11]) nie gwarantuje, iż jakikolwiek z formatów tekstur zmiennoprzecinkowych będzie zaakceptowany jako cel renderowania. Programista powinien wykonać test, aby sprawdzić czy maszyna użytkownika wspiera daną konfigurację. Można to zrobić w następujący sposób:

```
1 var gl      = getWebGLContext(),
2     texture = gl.createTexture(),
3     fbo     = gl.createFramebuffer();
4
5 if (!gl.getExtension('OES_texture_float')) {
6     throw new Error("Rozszerzenie OES_texture_float niedostępne.");
```

[2] Testy zostały przeprowadzone na systemie Linux (Ubuntu 12.04) i przeglądarce Google Chrome w wersji 22, która korzysta z natywnego sterownika *OpenGL*. Natomiast na tej samej konfiguracji sprzętowej, ale działającej pod kontrolą systemu Windows nie udałoby się uruchomić aplikacji. Podobnie w przypadku równie popularnego systemu operacyjnego OS X.

```
7  }
8  gl.bindTexture(gl.TEXTURE_2D, texture);
9  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, 128, 128, 0, gl.RGBA, gl.FLOAT,
       null);
10 gl.bindFramebuffer(gl.FRAMEBUFFER, fbo);
11 gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0, gl.
       TEXTURE_2D, texture, 0);
12 if (gl.checkFramebufferStatus(gl.FRAMEBUFFER) !== gl.
       FRAMEBUFFER_COMPLETE) {
13     throw new Error("Dana tekstura nie jest wspierana jako cel
           renderowania.");
14 }
```

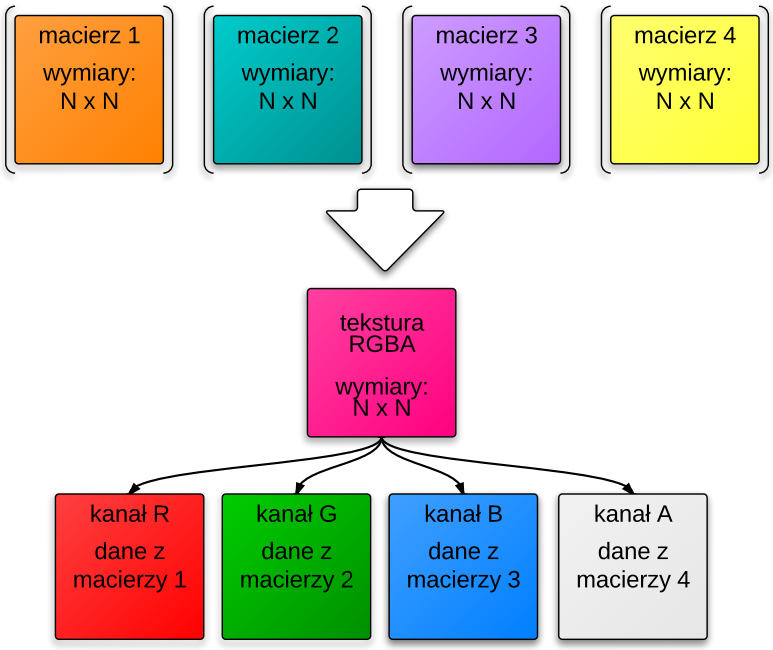
Listing 4.3: Weryfikacja poprawności formatu i typu tekstury używanej jako cel renderowania

W praktyce tekstury zmiennoprzecinkowe posiadające cztery kanały kolorów są najczęściej akceptowanym formatem do którego można zapisywać dane podczas renderowania. Dlatego też schematy organizacji danych RGBA1, RGBA4 oraz RGBA1/4 używają takiego formatu tekstury.

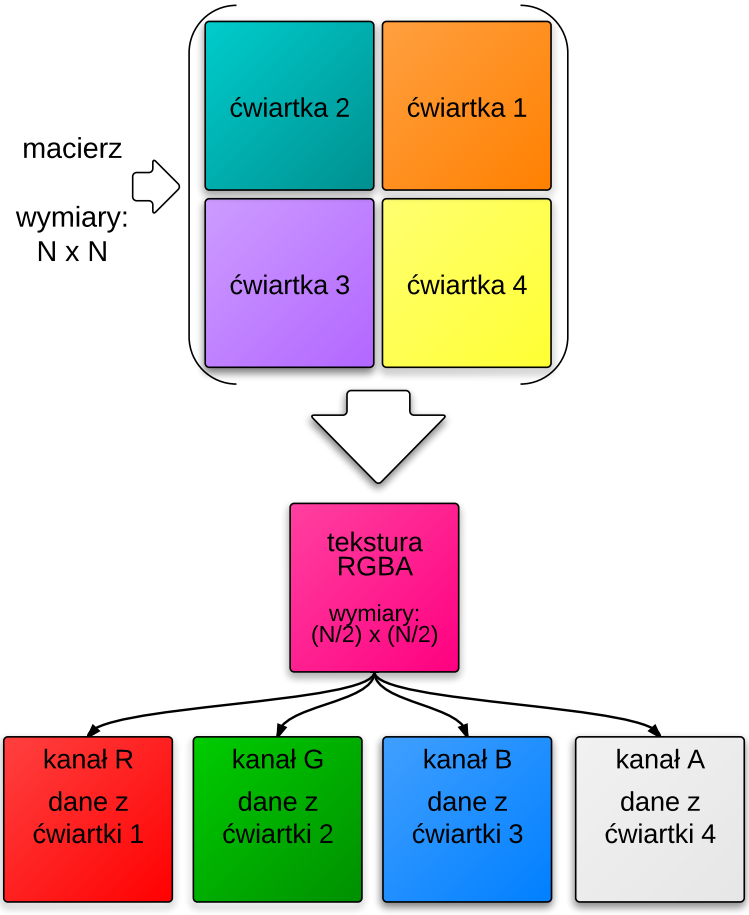
Schemat RGBA1 posiada te same zalety co A1 jeśli chodzi o organizacje i czytelność kodu źródłowego, jednak w tym przypadku dochodzi do dużego narzutu wydajności związanego z odczytem i zapisem tekstur. Przy każdej z tych operacji karta graficzna musi odczytać cztery kanały, jednak praktycznie wykorzystywany jest tylko jeden z nich. Operacje dostępu do pamięci są czasochłonne, dlatego też taka organizacja danych nie jest korzystna ze względów wydajnościowych.

Rozwiązaniem tego problemu są schematy RGBA4 oraz RGBA1/4. Organizacja danych wg. trzeciego schematu pozwala zredukować narzut związany z odczytem oraz zapisem pod warunkiem dobrej organizacji danych w teksturach. Pomysł ten obrazuje diagram 4.6.

Przy korzystnym ułożeniu danych jest możliwe praktycznie całkowite zredukowanie narzutu związanego z odczytem wszystkich czterech kanałów tekstury, jednak w praktyce jest to często niewykonalne. Przez korzystne rozmieszczenie danych przyjmuje się takie ich ułożenie, żeby program jednostek cieniujących odczytując teksturę faktycznie korzystał z danych zawartych w każdym z kanałów. Podobnie przy zapisie, program renderujący powinien modyfikować wszystkie cztery kanały. W przypadku symulatora *Energy2D* udało się uzyskać taką organizację danych, żeby odczyt był w znacznym stopniu zoptymalizowany, jednak podczas zapisu modyfikowany był tylko jeden lub dwa kanały (kanał zawierający dane o temperaturze lub kanały zawierające komponenty wektorów prędkości). Mimo nie do końca optymalnego ułożenia kanałów, schemat RGBA4 okazał się wydajniejszy około 54% od RGBA1.



Rysunek 4.6: Schemat organizacji danych z czterech macierzy w jednej teksturze



Rysunek 4.7: Schemat organizacji danych z macierzy NxN w teksturze $(N/2)x(N/2)$

Ciekawą organizacją danych może wydawać się również pomysł przedstawiony w schemacie RGBA1/4. Pozwala przechowywać tablicę JavaScript o wymiarach *NxN* w teksturze o wymiarach *(N/2)x(N/2)*. Pomysł ten obrazuje diagram 4.7.

Taki układ danych teoretycznie posiada sporo zalet jak w przypadku użycia tekstur jednokanałowych. Jednak znacznie zmniejsza on czytelność kodu i komplikuje implementacje. Utrudnione zostaje przede wszystkim kontrolowanie warunków brzegowych, programy jednostek cieniujących przetwarzają cztery pola jednocześnie i stają się bardzo skomplikowane. W przypadku *Energy2D* komplikacje programów fragmentów były tak znaczne, że w efekcie odnotowano bardzo słabe rezultaty pod względem wydajności. Symulacja okazała się być wolniejsza około 34% od symulacji korzystającej ze schematu RGBA4.

Dlatego też, ostatecznie w *Energy2D* zastosowano schemat RGBA4. Zapewnia on najlepszy kompromis pomiędzy dobrą wydajnością oraz wsparciem przez większość dostępnych obecnie urządzeń.

4.2.5. Geometria

Podczas wykonywania obliczeń ogólnego przeznaczenia przy pomocy bezpośredniego programowania jednostek cieniujących karty graficznej, niezbędne jest stworzenie obiektu (a właściwie jego geometrii), który będzie renderowany. W teorii może być to dowolna bryła. Jednak najczęściej pożądane jest, aby program fragmentów wykonał się dla każdej komórki siatki symulacyjnej, którą stanowi piksel tekstury. Można to osiągnąć renderując płaszczyznę, która pokrywa całą dostępną przestrzeń renderowania.

Również w przypadku *Energy2D* również wykorzystywany jest rendering takiej płaszczyzny. Do przechowywania jej własności wykorzystane zostały bufory wierzchołków (ang. vertex buffers) oraz indeksów (ang. index buffers) znajdujące się w pamięci karty graficznej. Dzięki temu, podczas wielokrotnego renderowania tej samej płaszczyzny, nie jest konieczne nieustanne przesyłanie atrybutów wierzchołków do pamięci GPU.

W finalnej implementacji *Energy2D* zostały przygotowane klasy pomocnicze zarządzające geometrią oraz buforami. Jednak najprostszy sposób na stworzenie płaszczyzny pokrywającej cały obszar renderowania oraz umieszczenie jej atrybutów w pamięci karty graficznej zaprezentowany jest poniżej:

```
1 var gl           = getWebGLContext(),
2     vertexBuffer = gl.createBuffer(),
3     indexBuffer  = gl.createBuffer(),
4     vertexData,
5     indexData;
6
7 // Współrzędne wierzchołków.
```

```
8  vertexData = new Float32Array([
9      -1, -1,
10      1, -1,
11     -1,  1,
12      1,  1
13 ]);
14 // Transfer do bufora wierzchołków.
15 gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
16 gl.bufferData(gl.ARRAY_BUFFER, vertexData, gl.STATIC_DRAW);
17 // Indeksy trojkątów płaszczyzny.
18 indexData = new Uint16Array([
19     0, 1, 2,
20     2, 1, 3
21 ]);
22 // Transfer do bufora indeksów.
23 gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
24 gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indexData, gl.STATIC_DRAW);
```

Listing 4.4: Definicja geometrii płaszczyzny pokrywającej cały obszar renderowania

4.2.6. Wykonywanie kroków algorytmów na GPU

Dysponując skompilowanymi programami wierzchołków oraz fragmentów, danymi symulacji zapisanymi w teksturach dwuwymiarowych oraz niezbędną geometrią, można przejść do faktycznej realizacji kroków algorytmów zapisanych w programach jednostek cieniujących.

W przypadku technologii WebGL narzuca się schemat związany z techniką renderowania do tekstury przy użyciu obiektu bufora ramki (ang. *Frame Buffer Object*). Związanie takiego obiektu z wybraną teksturą, a następnie uaktywnienie przed właściwym renderowaniem, powoduje iż karta graficzna automatycznie kopiuje zawartość bufora ramki do tekstury po zakończonym renderowaniu.

Technika ta ma jednak kilka ograniczeń. Tekstura związana z obiektem bufora ramki (czyli przeznaczona do zapisu) nie może być używana jednocześnie do odczytu wartości. W związku z tym zawsze trzeba używać minimalnie jednej tekstury tymczasowej. Następnie należy kopiować jej zawartość do tekstury docelowej lub podmienić referencje. Oczywiście modyfikacja wyłącznie referencji jest znacznie efektywniejsza przez co i częściej stosowana. Całościowo taki schemat nazywany jest „ping-pong rendering”.

Implementacja w języku *JavaScript* przy użyciu technologi *WebGL* nie odbiega znacząco od implementacji w innych językach przy użyciu „tradycyjnego” API OpenGL. Mark Harris zaprezentował najważniejsze aspekty tej techniki w swoich artykułach opublikowanych w serii *GPU Gems* ([Har05] oraz [Har04]).

5. Ocena aplikacji

W tym rozdziale zaprezentowana została kompleksowa ocena aplikacji *Energy2D*. Jest ona podzielona na kilka części. Następnie zaprezentowane są testy jakościowe, które opierają się na próbie odzwierciedlenia wybranych zjawisk fizycznych przy pomocy symulatora. Z kolei testy wydajnościowe wersji podstawowej oraz równoległej *Energy2D* pokazują zysk jaki dało przeniesienie obliczeń na kartę graficzną. Na podstawie tych wyników zanalizowana została dostępność aplikacji dla potencjalnych użytkowników na różnych urządzeniach oraz ich konfiguracjach. Rozdział kończy ocena stopnia realizacji założeń projektowych oraz krótkie podsumowanie.

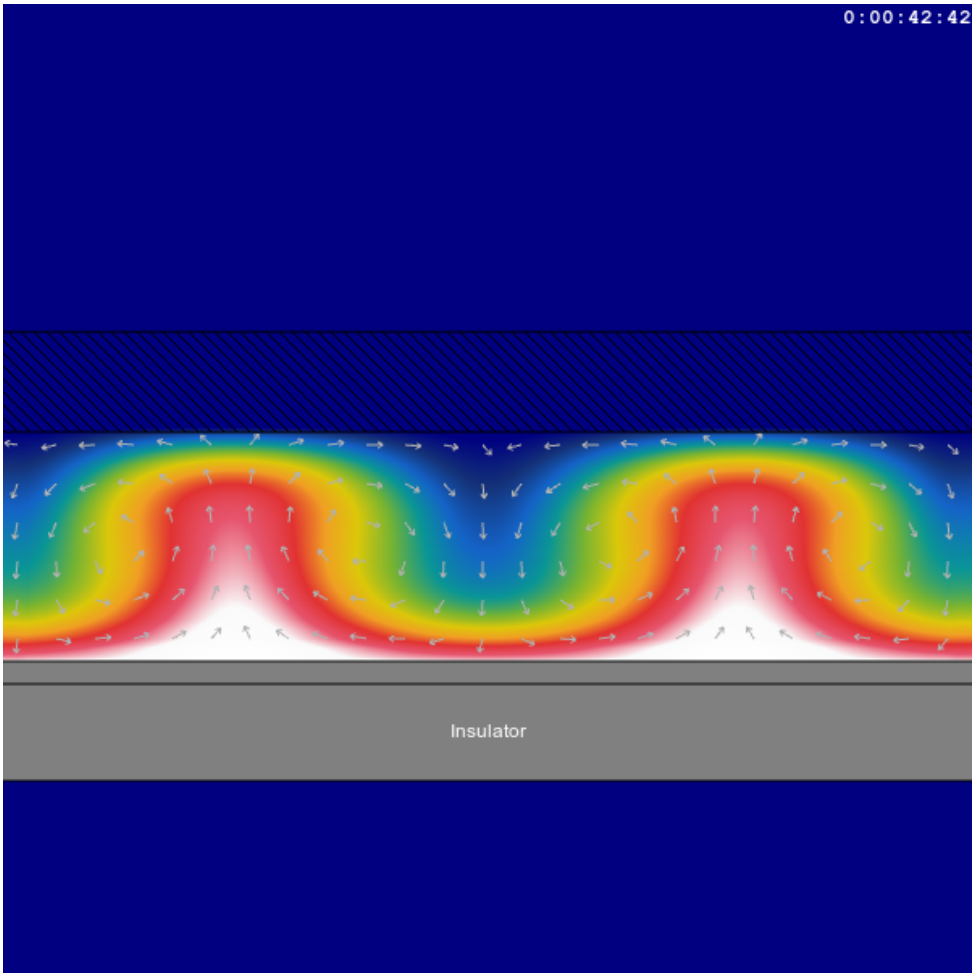
5.1. Modelowanie wybranych zjawisk fizycznych jako testy jakościowe symulatora

W przypadku aplikacji edukacyjnej jaką jest symulator *Energy2D* niezwykle istotne jest aby symulowane zjawiska odzwierciedlały w sposób wiarygodny rzeczywistość. Z drugiej jednak strony, aplikacja musi być interaktywna i działać w czasie rzeczywistym. Dlatego też nie można sobie pozwolić na zbyt długi czas wykonywania, co zwykle idzie w parze z dokładnymi algorytmami i obliczeniami. Z tego też powodu zastosowane silniki (algorytmy) fizyczne balansują pomiędzy poprawnością fizyczną, a wydajnością (por. rozdział 2.5). Jest to dopuszczalne, ponieważ symulacja jest zorientowana wyłącznie na aspekt wizualny.

Z powodu tego kompromisowego podejścia do pełnej dokładności obliczeń, niezwykle istotne były testy aplikacji pod kątem podstawowej poprawności fizycznej. W tym celu zostały przygotowane przypadki testowe, które miały za zadanie modelować powszechnie znane zjawiska fizyczne związane z przewodnictwem cieplnym oraz dynamiką płynów. Poniżej przedstawione są wyniki tych testów.

5.1.1. Komórki Bénarda

Modelowanie komórek Bénarda to jeden z podstawowych testów aplikacji symulujących dynamikę płynów. Są to komórki konwekcyjne powstające w płynie podgrzewanym od spodu. Rysunek 5.1 prezentuje wyniki symulacji przeprowadzonej przez *Energy2D*.



Rysunek 5.1: Symulacja formowania się komórek Bénarda

Wyraźnie widać formowanie się oczekiwanych komórek. Ich układ oraz kształt stabilizuje się po bardzo krótkim czasie symulacji. Wyniki można uznać za w pełni satysfakcjonujące.

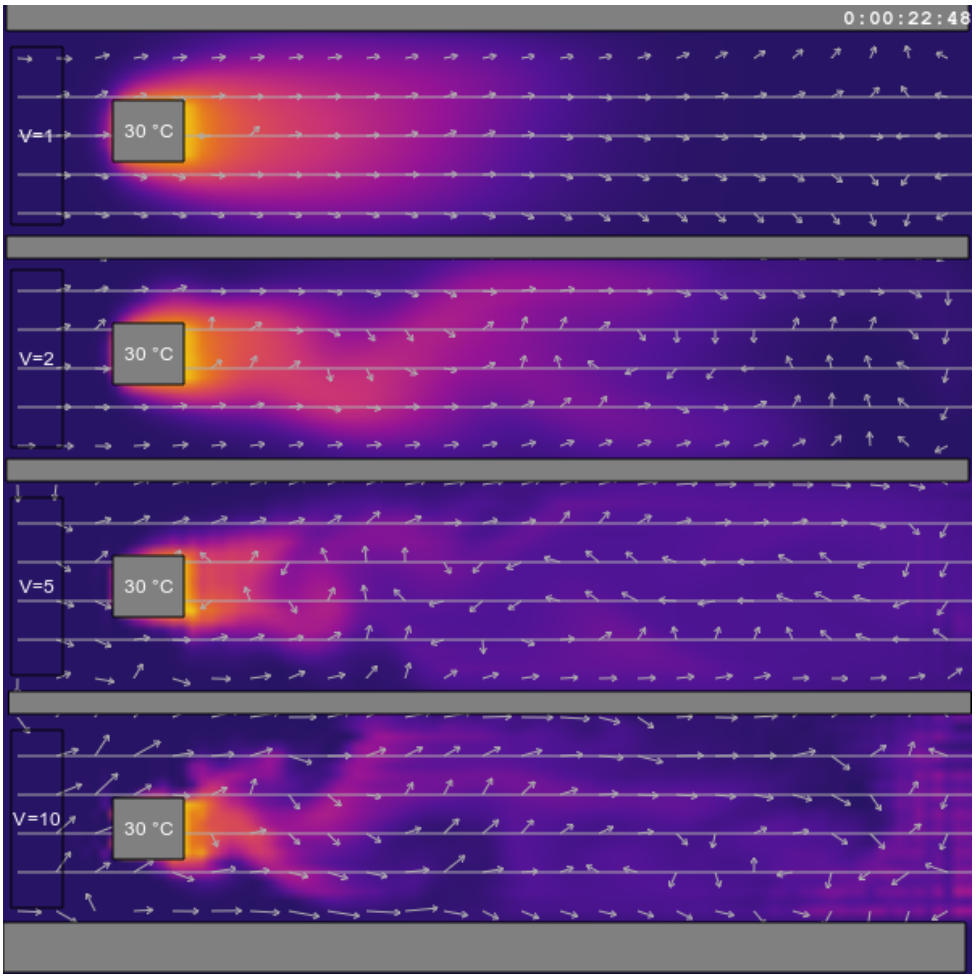
5.1.2. Przepływ laminarny oraz turbulentny

Symulacja przepływów laminarnych oraz turbulentnych to następny test silnika dynamiki płynów. Rodzaj przepływu, w przypadku gdy jest on zakłócony przez obecność przeszkody, w głównym stopniu determinuje liczba Reynoldsa. Jest ona zależna od:

- lepkości płynu,

- prędkości przepływu,
- średnicy przeszkody.

W związku z tym został przygotowany przypadek testowy, w którym płyn ma stałą lepkość, a przeszkody identyczne wymiary. Zmienna jest tylko prędkość przepływu. Dla mniejszych prędkości oczekiwanym wynikiem był przepływ laminarny, dla większych przepływ turbulentny. Rysunek 5.2 prezentuje wyniki takiej symulacji przeprowadzonej przez *Energy2D*.



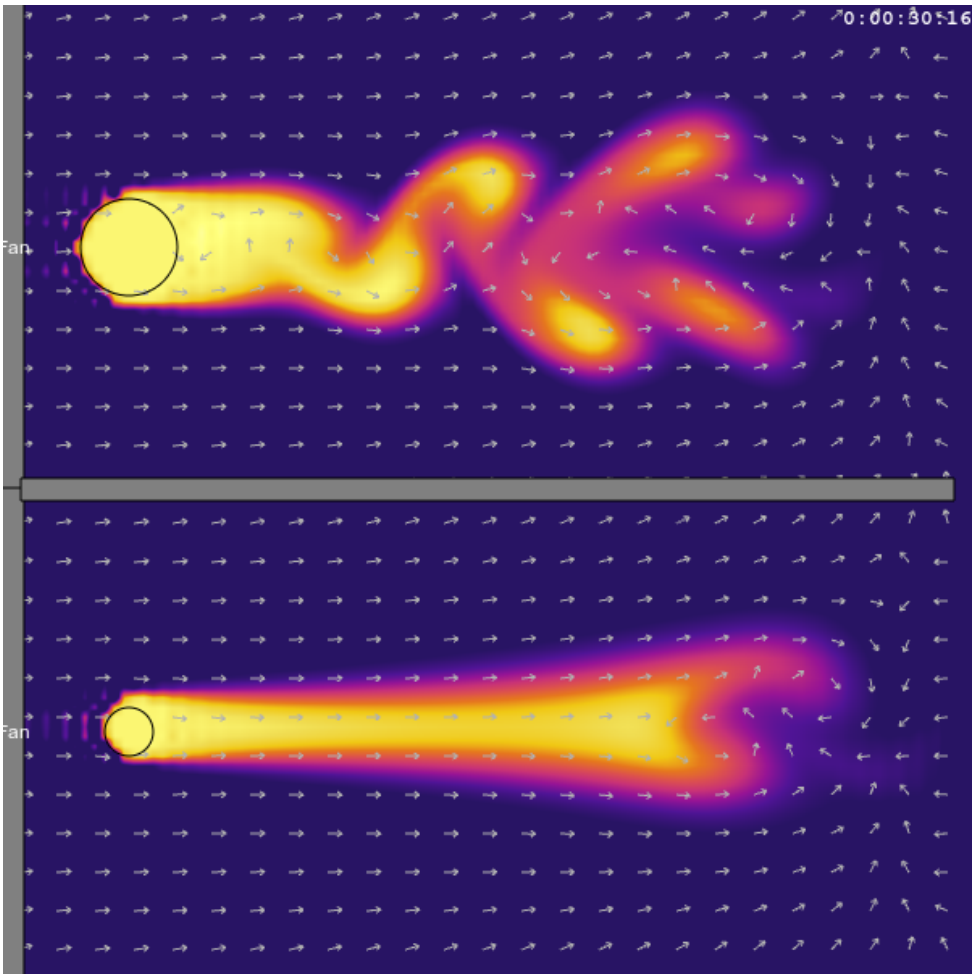
Rysunek 5.2: Symulacja wpływu liczby Reynoldsa na rodzaj przepływu

Rezultaty są zgodne z oczekiwaniami. Wyraźnie widać wpływ liczby Reynoldsa (determinowanej przez prędkość płynu) na rodzaj przepływu.

5.1.3. Ścieżka wirowa von Kármána

Ścieżka wirowa von Kármána to kolejny doskonały test silnika dynamiki płynów. Jest to szczególny rodzaj przepływu turbulentnego, który powstaje wyłącznie dla pewnego zakresu wartości liczby Reyonldsa (por. 5.1.2).

W związku z tym został przygotowany przypadek testowy, w którym płyn ma stałą lepkość oraz prędkość przepływu, natomiast zmienna jest tylko średnica przeszkody. Zgodnie z powyższymi założeniami, powinno być możliwe dobranie takich średnic przeszkód, aby wiry Kármána powstały tylko za większą z nich. Rysunek 5.3 prezentuje wyniki takiej symulacji przeprowadzonej przez *Energy2D*.



Rysunek 5.3: Symulacja formowania się wirów Kármána

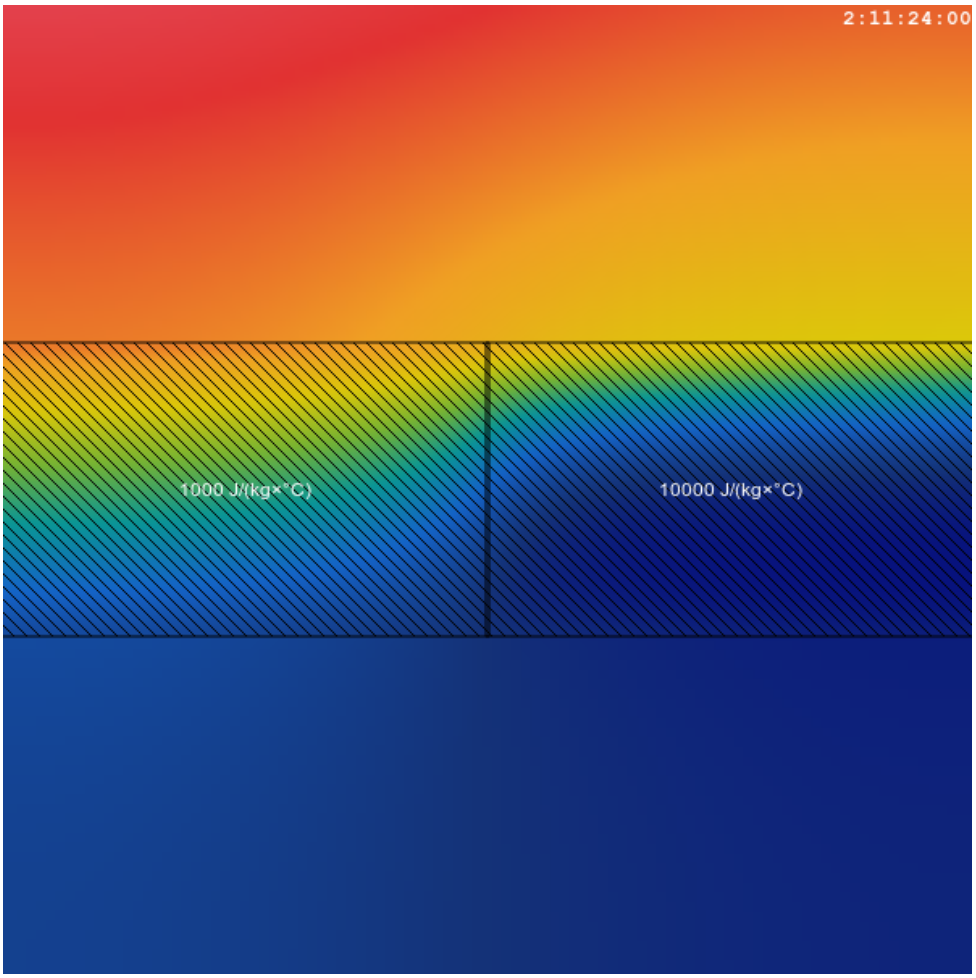
Rezultaty są zgodne z oczekiwaniami. Wir powstał tylko w warunkach, w których liczba Reynoldsa była większa.

5.1.4. Pojemność cieplna

Tym razem przypadek testowy dotyczy silnika przewodnictwa cieplnego. Pojemność cieplna jest wielkość fizyczna, która charakteryzuje ilość ciepła, jaka jest niezbędna do zmiany temperatury ciała o jednostkę temperatury. Poprawna symulacja powinna uwzględniać ten parametr materiałów.

Aby to sprawdzić przygotowany został przypadek testowy, w którym znajdują się dwa materiały o różnej pojemności cieplnej. Materiał o większej pojemności powinien przewo-

dzić ciepło znacznie lepiej niż ten o pojemności mniejszej. Wyniki takiego eksperymentu przeprowadzonego przez symulator *Energy2D* prezentuje rysunek 5.4.



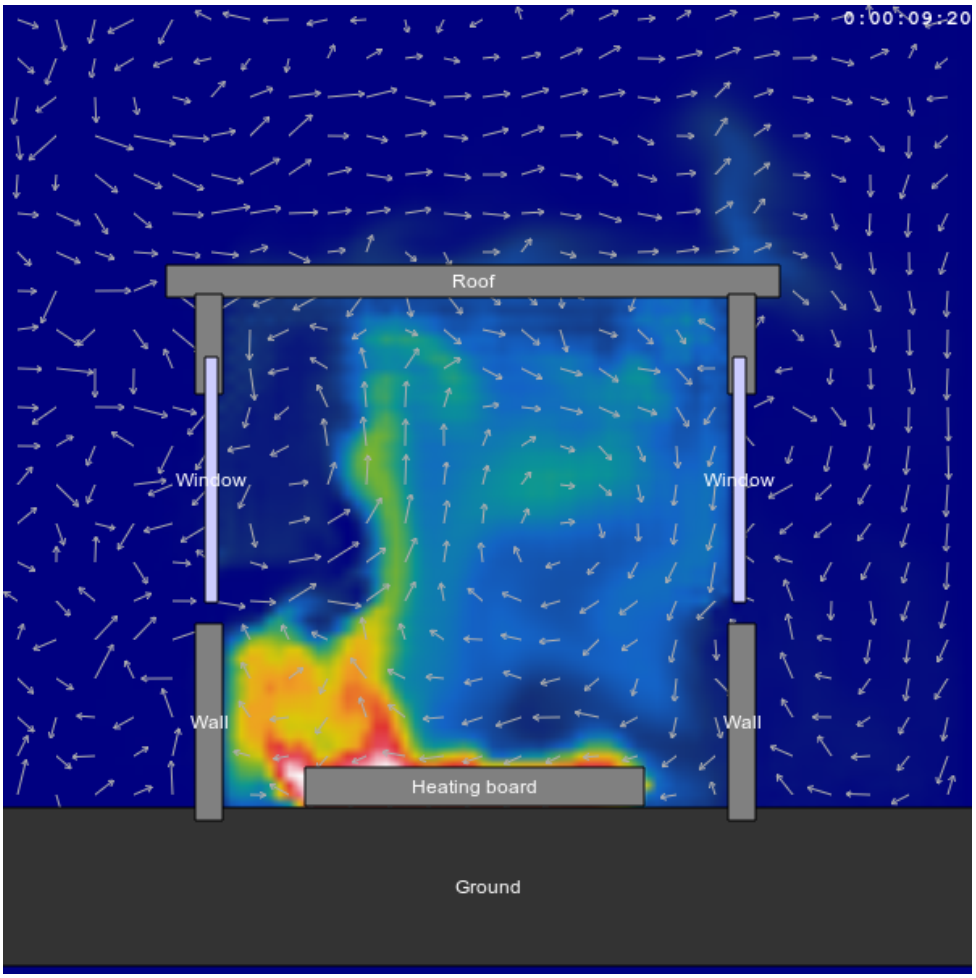
Rysunek 5.4: Symulacja wpływu pojemności cieplnej na przewodnictwo cieplne

Rezultaty po raz kolejny są zgodne z oczekiwaniami. Widać wyraźnie, iż materiał o większej pojemności cieplnej znacznie lepiej przewodzi ciepło.

5.1.5. Aspekt edukacyjny

Ostatni z zaprezentowanych testów jest bardziej złożony. Nie prezentuje on jednego, konkretnego zjawiska fizycznego jak poprzednie przykłady. Skupia się on na zaprezentowaniu przykładowych możliwości symulacji oraz na potencjalnym aspekcie edukacyjnym.

Przygotowana została scena zawierająca nieszczelnie izolowane pomieszczenie, będące metaforą mieszkania. Jest ono ogrzewane przez element grzewczy o stałej mocy. W jego otoczeniu został wymuszony dość mocny przepływ symulujący wiatr. Miało to na celu pokazanie użytkownikom różnych, potencjalnych dróg ucieczki ciepła z pomieszczeń mieszkalnych.



Rysunek 5.5: Symulacja ucieczki ciepła z pomieszczeń mieszkalnych

Rysunek 5.5 prezentuje efekty takiej symulacji. Widoczne są dwa wyraźne zjawiska – strata ciepła w okolicy nieszczelnych okien oraz strata ciepła związana z przewodnictwem cieplnym dachu. Pozwala to lepiej zrozumieć użytkownikowi (docelowo uczniowi na początkowym etapie edukacji) wpływ różnych aspektów pomieszczeń mieszkalnych (takich jak szczelność czy izolacja) na straty ciepła. Może przyczynić się to do rozsądniejszego i ekonomiczniejszego gospodarowania energią w jego własnym mieszkaniu czy też domu. W dobie szczególnej dbałości o środowisko naturalne jest to niezwykle cenny i pożądany efekt edukacyjny.

5.2. Testy wydajnościowe

Wydajność jest kluczowym aspektem symulacji ukierunkowanej na cele edukacyjne. Tylko odpowiednia prędkość symulacji może przyciągnąć uwagę użytkownika i zachęcić go do dalszej eksploracji zagadnienia.

W celu osiągnięcia zadowalającej wydajności na różnych urządzeniach zastosowano

kilka technik. Podstawową jest świadoma implementacja polegająca na unikaniu zbędnych, czasochłonnych operacji, wynikająca ze znajomości środowiska przeglądarki internetowej. Jednak krokiem, który najbardziej przyczynił się do powstania naprawdę wydajnej aplikacji było przeniesienie obliczeń związanych z fizyką na procesor karty graficznej.

Niniejsza sekcja przedstawia zyski z zastosowania tej optymalizacji. Omówiona jest także kwestia wpływu konfiguracji sprzętowej użytkownika oraz związana z tym ogólna dostępność symulatora dla szerokiego grona odbiorców.

5.2.1. Metodologia testów

Za wyznacznik wydajności została przyjęta liczba klatek na sekundę animacji, którą jest w stanie generować działająca aplikacja *Energy2D*. Jedna klatka domyślnie składa się z:

- czterech kroków symulacji,
- odświeżenie wizualizacji.

Takie też ustawienia zostały zastosowane podczas wszystkich testów.

Aplikacja wymusza kolejne klatki poprzez użycie metody *setInterval()* z czasem 0. Powoduje to, iż przeglądarka nie wprowadzi żadnych dodatkowych opóźnień między klatkami i przejdzie do generowania kolejnej natychmiast, gdy będzie to możliwe. Nie została użyta zalecana w przypadku aplikacji *WebGL*funkcja *requestAnimationFrame()*, ponieważ wprowadza ona limit 60 klatek na sekundę i koncentruje się na utrzymaniu stałego tempa animacji, a nie osiągnięciu maksymalnej wydajności. Zaburzało to w istotny sposób wiarygodność wyników.

Na potrzeby testów zostały przygotowane specjalne przypadki testowe. Różnią się one między sobą:

- układem sceny,
- modelowanym zjawiskiem fizyczny,
- użytymi silnikami fizycznymi:
  - wyłącznie silnik przewodnictwa cieplnego,
  - silnik przewodnictwa cieplnego oraz silnik dynamiki płynów,
- rozmiarem siatki symulacyjnej.

Tabela 5.1 prezentuje symboliczne nazwy przypadków testowych wraz z ich krótką charakterystyką. Skrót HT pochodzi od ang. Heat Transfer i oznacza, że dany przypadek

Tablica 5.1: Zestawienie charakterystyki przypadków testowych do pomiaru wydajności

Nazwa testu	Siatka	HT	CFD	Opis
ht1-100	$100x100$	✓	×	symulacja przewodnictwa cieplnego
ht1-512	$512x512$	✓	×	symulacja przewodnictwa cieplnego
ht1-1024	$1024x1024$	✓	×	symulacja przewodnictwa cieplnego
ht2-100	$100x100$	✓	×	symulacja przewodnictwa cieplnego
ht2-512	$512x512$	✓	×	symulacja przewodnictwa cieplnego
ht2-1024	$1024x1024$	✓	×	symulacja przewodnictwa cieplnego
cfd1-100	$100x100$	✓	✓	symulacja dynamiki płynów
cfd1-256	$256x256$	✓	✓	symulacja dynamiki płynów
cfd2-100	$100x100$	✓	✓	symulacja dynamiki płynów
cfd2-256	$256x256$	✓	✓	symulacja dynamiki płynów

testowy korzysta z silnika przewodnictwa cieplnego. Z kolei skrót CDF pochodzi od ang. Computational Fluid Dynamics i oznacza, że dany przypadek testowy korzysta z silnika dynamiki płynów. Wizualizację symulacji przypadków testowych prezentują rysunki 5.6, 5.7, 5.8 oraz 5.9.

5.2.2. Konfiguracje sprzętowe przeznaczone do testów

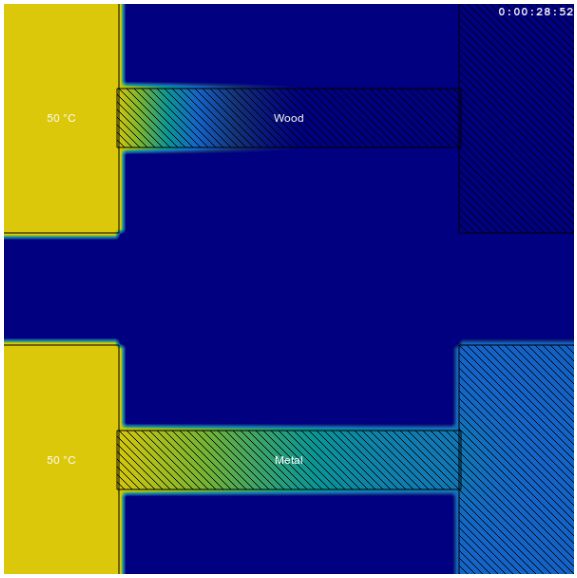
Testy zostały przeprowadzone na następujących komputerach:

- laptop Samsung QX510,
- laptop Apple MacBook Pro, wersja z roku 2010,
- laptop Apple MacBook Pro, wersja z roku 2012,
- komputer stacjonarny.

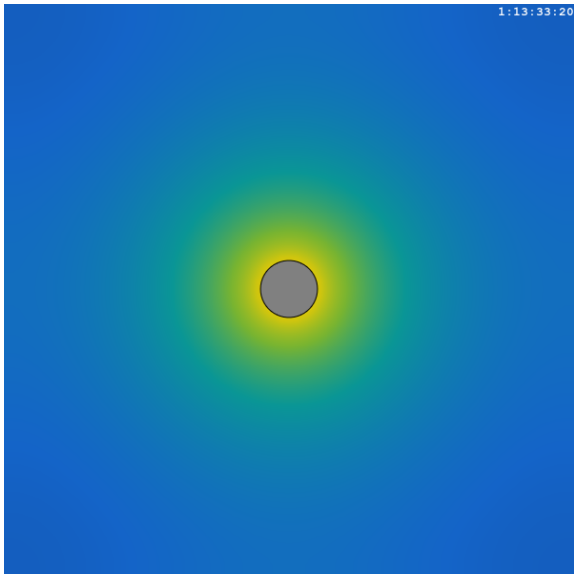
Tablica 5.2: Charakterystyka komputerów testowych

Nazwa	Procesor	Karta graficzna	System operacyjny
Samsung QX510	Intel Core i5 2.66GHz	NVIDIA GT 420M	Ubuntu 12.04
MacBook Pro 2010	Intel Core i7 2.66GHz	NVIDIA GT 330M	Mac OS X 10.6.8
MacBook Pro 2012	Intel Core i7 2.3GHz	NVIDIA GT 650M	Mac OS X 10.7.4
stacjonarny	Intel Core2Quad 2.5GHz	NVIDIA 9600GT	Windows 7

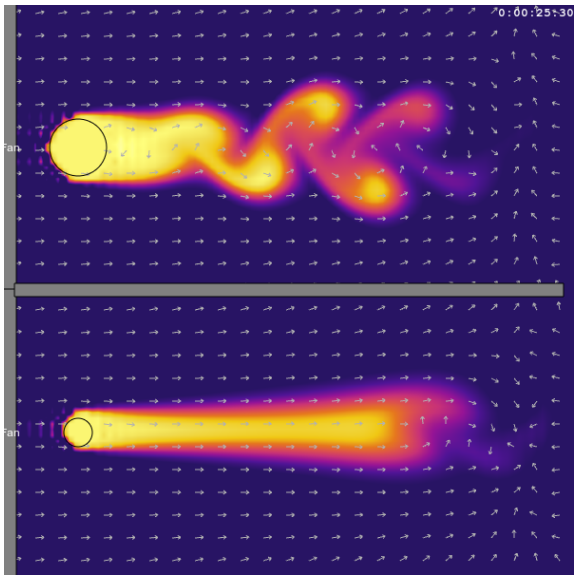
Ich dokładną specyfikację prezentuje tabela 5.2. Konfiguracje znacznie różnią się od siebie. Działają pod kontrolą trzech najpopularniejszych systemów operacyjnych – Linux



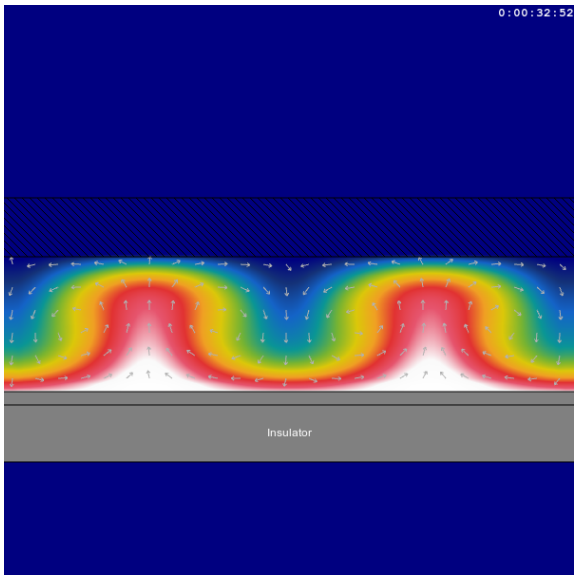
Rysunek 5.6: Symulacja ht1-100/512/1024



Rysunek 5.7: Symulacja ht2-100/512/1024



Rysunek 5.8: Symulacja cfd1-100/256



Rysunek 5.9: Symulacja cfd2-100/256

(Ubuntu), Mac OS X oraz Windows 7. Również moc obliczeniowa kart graficznych jest bardzo zróżnicowana.

5.2.3. Porównanie wydajności w różnych przeglądarkach internetowych

Aplikacja od początku rozwoju była głównie testowana w przeglądarce Google Chrome, gdyż wyraźnie było widać jej ogromną przewagę w kwestii wydajności silnika *JavaScript*. Ponadto, wg. ostatnich raportów, jest to najpopularniejsza przeglądarka na świecie, tak więc potencjalnie najwięcej użytkowników *Energy2D* będzie z niej korzystać. Raport [Bro12] pokazuję, iż w lipcu 2012 roku z przeglądarki Google Chrome korzystało 42.9% użytkowników internetu. Drugie miejsce należało do przeglądarki Mozilla Firefox z udziałem 33.7%. Łącznie te dwie przeglądarki posiadają 77.6% „rynku", dlatego są traktowane priorytetowo. Kolejna popularna przeglądarka Internet Explorer została wyłączona z testów z powodu niedostępności poza środowiskiem Windows oraz braku wsparcia dla technologii *WebGL*. Podobnie Safari, które wspiera *WebGL* jednak działa tylko na systemie operacyjnym Mac OS. Dlatego też testy zostały przeprowadzone wyłącznie z użyciem następujących przeglądarek:

- Google Chrome v. 22
- Mozilla Firefox v. 16
- Opera v. 12.50

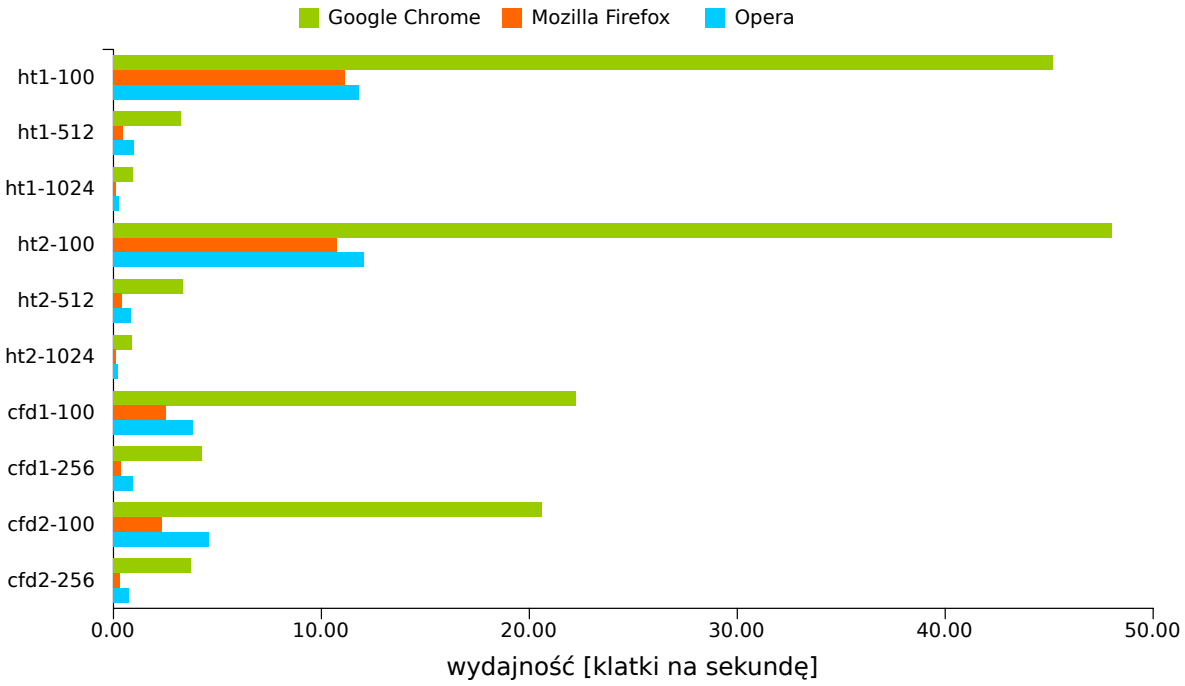
Środowiskiem testowym był laptop Samsung QX510 (por. tablica 5.2).

Obliczenia fizyczne wykonywane na CPU

Wyniki pomiarów wydajności symulatora w przypadku obliczeń przeprowadzanych na CPU prezentuje tabela 5.3 oraz wykres 5.10. Nietrudno dostrzec ogromną przewagę przeglądarki Google Chrome. Jest to efekt zastosowanego w niej niezwykle wydajnego silnika *JavaScript* o nazwie V8. Opera oraz Mozilla Firefox posiadają zbliżoną wydajność, jednak Opera jest nieznacznie szybsza. Warto też zauważyć, iż kiedy obliczenia fizyczne przeprowadzane są na CPU, jedyną przeglądarką, która zapewnia odpowiednią prędkość wykonywania się symulacji jest Google Chrome.

Tablica 5.3: Wydajność symulacji na CPU w zależności od przeglądarki

Test	Google Chrome	Mozilla Firefox	Opera
ht1-100	45.20	11.12	11.83
ht1-512	3.24	0.45	0.98
ht1-1024	0.92	0.11	0.25
ht2-100	48.04	10.73	12.05
ht2-512	3.34	0.40	0.84
ht2-1024	0.89	0.11	0.22
cfd1-100	22.25	2.51	3.84
cfd1-256	4.24	0.38	0.93
cfd2-100	20.60	2.35	4.58
cfd2-256	3.71	0.30	0.73



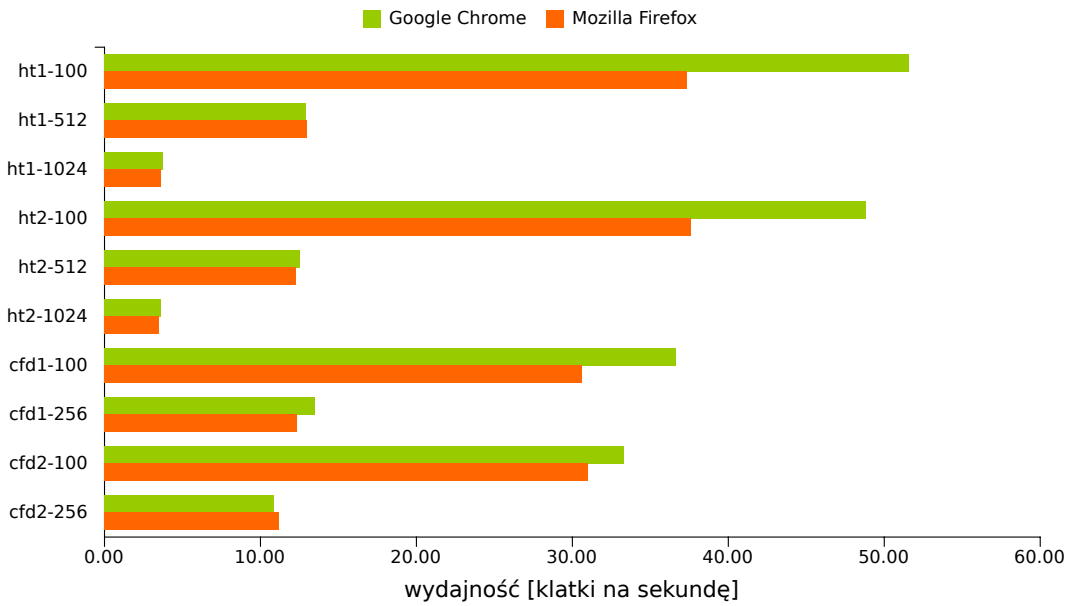
Rysunek 5.10: Wydajność symulacji na CPU w zależności od przeglądarki

Obliczenia fizyczne wykonywane na GPU

W przypadku symulacji, która obliczenia fizyczne przeprowadza na GPU, nie została przetestowana Opera. Wynika to z faktu, iż w dniu kiedy testy były przeprowadzane, wsparcie dla technologii *WebGL* przez Operę wyłącznie eksperymentalne w wyniku czego nie było dostępu do niezbędnego rozszerzenia *OES_texture_float*. Wyniki testów dla przeglądarek Google Chrome oraz Mozilla Firefox przedstawiają tabela 5.4 oraz wykres 5.11.

Tablica 5.4: Wydajność symulacji na GPU w zależności od przeglądarki

Test	Google Chrome	Mozilla Firefox
ht1-100	51.57	37.33
ht1-512	12.94	12.99
ht1-1024	3.73	3.64
ht2-100	48.84	37.59
ht2-512	12.52	12.28
ht2-1024	3.60	3.47
cfd1-100	36.67	30.59
cfd1-256	13.47	12.35
cfd2-100	33.28	31.01
cfd2-256	10.85	11.18



Rysunek 5.11: Wydajność symulacji na GPU w zależności od przeglądarki

W przypadku obliczeń na GPU sytuacja diametralnie się zmienia. Różnice między przeglądarkami, tak istotne przy obliczeniach na CPU, stają się znikome. Wynika to z

faktu, iż najbardziej obciążające obliczenia przeniesione są na kartę graficzną i to jej wydajność jest decydująca, a nie wydajność silnika *JavaScript*. Potwierdza to fakt, iż przewagę Google Chrome widać wyłącznie dla testów działających na najmniejszych siatkach. Są to jedyne przypadki, gdzie wydajność samego interpretera *JavaScript* jest jeszcze istotna.

Innym ważnym wnioskiem jest to, iż użycie technologii *WebGL* pozwala zniwelować różnice między wydajnością przeglądarek. O ile w przypadku obliczeń na CPU jedynym rozsądnym środowiskiem wykonywania się symulacji była przeglądarka Google Chrome, to przy zastosowaniu *WebGL* możliwe staje się również użycie Mozilli Firefox oraz każdej innej przeglądarki, która wspiera *WebGL* (np. Safari czy też w niedalekiej przyszłości także Opery).

5.2.4. Analiza zysku wydajności wynikający z przeniesienia obliczeń na GPU

Kluczową optymalizacją symulatora *Energy2D* było przeniesienie obliczeń związanych z fizyką na procesor karty graficznej przy wykorzystaniu technologii *WebGL*. Zagadnienie to porusza rozdział 4. Niniejsza sekcja przedstawia i analizuje efekty jakie przyniosła ta optymalizacja.

Metodologię testów definiuje podrozdział 5.2.1.

Konfiguracje sprzętowe użyte podczas testów przybliża z kolei podrozdział 5.2.2.

Wybór przeglądarki internetowej do testów jest niezwykle istotnym czynnikiem, który w największym stopniu determinuje ocenę optymalizacji. Wynika to z bardzo dużych różnic między wydajnością interpreterów *JavaScript* w różnych przeglądarkach (por. podrozdział 5.2.3). Oczywistym jest, że przeglądarka o niższej wydajności silnika *JavaScript* zanotuje znacznie większe przyspieszenie niż przeglądarka o silniku bardzo wydajnym. Dlatego też do testów została użyta **najwydajniejsza** z dostępnych przeglądarek – Google Chrome. Dzięki temu testy przedstawiają **minimalne, pesymistyczne** oczekiwane przyspieszenie związane z przeniesieniem obliczeń na GPU. Jednocześnie należy podkreślić, iż na mniej wydajnych przeglądarkach optymalizacja przynosi jeszcze więcej korzyści i często staje się niezbędna aby w ogóle możliwe było jej używanie.

Wyniki testów na różnych komputerach przestawiają tabele od 5.5 do 5.8. Zgodnie z przyjętą metodologią, miarą wydajności jest liczba klatek na sekundę jaką jest w stanie wygenerować przeglądarka podczas działania symulacji. Kolumna *CPU* prezentuje wydajność symulacji na procesorze głównym, a kolumna *GPU* na procesorze karty graficznej.

Z kolei kolumna *GPU/CPU* prezentuje stosunek wartości z kolumny *GPU* do wartości z kolumny *CPU*. Jest to pomocniczy współczynnik, który bardzo dobrze charakteryzuje i wizualizuje zysk z optymalizacji.

Tablica 5.5: Wydajność symulacji na CPU oraz GPU – laptop Samsung QX510

Test	CPU	GPU	**GPU/CPU**
ht1-100	41.98	68.73	**1.64**
ht1-512	3.15	12.30	**3.90**
ht1-1024	0.71	3.58	**5.04**
ht2-100	51.37	62.36	**1.21**
ht2-512	2.86	12.61	**4.41**
ht2-1024	0.80	3.75	**4.69**
cfd1-100	18.84	43.76	**2.32**
cfd1-256	3.88	13.56	**3.49**
cfd2-100	20.60	43.28	**1.62**
cfd2-256	3.71	10.85	**2.92**

Tablica 5.6: Wydajność symulacji na CPU oraz GPU – laptop Apple MacBook Pro, wersja 2010

Test	CPU	GPU	**GPU/CPU**
ht1-100	43.10	92.60	**2.15**
ht1-512	3.20	15.50	**4.84**
ht1-1024	0.91	4.90	**5.38**
ht2-100	43.70	92.10	**2.11**
ht2-512	2.91	15.50	**5.33**
ht2-1024	0.73	4.55	**6.23**
cfd1-100	21.00	55.10	**2.62**
cfd1-256	2.30	14.60	**6.35**
cfd2-100	8.00	48.00	**6.00**
cfd2-256	1.70	12.20	**7.18**

Tablica 5.7: Wydajność symulacji na CPU oraz GPU – laptop Apple MacBook Pro, wersja 2012

Test	CPU	GPU	**GPU/CPU**
ht1-100	57.00	116.00	**2.04**
ht1-512	4.10	53.60	**13.07**
ht1-1024	1.10	14.90	**13.55**
ht2-100	79.80	118.80	**1.49**
ht2-512	4.60	50.30	**10.93**
ht2-1024	0.96	14.20	**14.79**
cfd1-100	17.10	101.00	**5.91**
cfd1-256	2.80	36.60	**13.07**
cfd2-100	19.90	93.00	**4.67**
cfd2-256	2.80	36.60	**13.07**

Tablica 5.8: Wydajność symulacji na CPU oraz GPU – komputer stacjonarny

Test	CPU	GPU	**GPU/CPU**
ht1-100	35.00	78.00	**2.23**
ht1-512	2.50	37.00	**14.80**
ht1-1024	0.69	9.00	**13.04**
ht2-100	39.00	83.00	**2.13**
ht2-512	2.25	37.00	**16.44**
ht2-1024	0.65	10.60	**16.31**
cfd1-100	16.55	64.10	**3.87**
cfd1-256	2.50	30.00	**12.00**
cfd2-100	12.80	65.20	**5.09**
cfd2-256	2.15	26.30	**12.23**

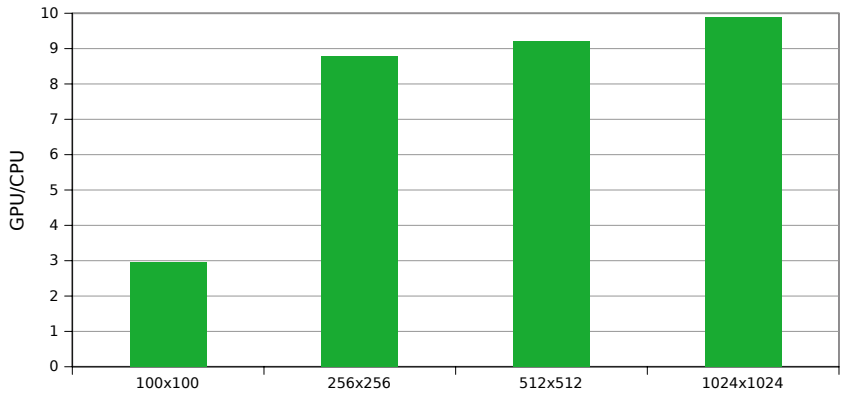
Bardzo wyraźna jest zależność wyników od konfiguracji sprzętowej. Najlepsze rezultaty osiągnął nowoczesny laptop Apple MacBook Pro z roku 2012 dzięki wydajnej karcie graficznej. Pozostałe dwa laptopy notują najmniejsze korzyści ze zrównoleglenia, gdyż posiadają znacznie słabsze jednostki graficzne. Jedyny komputer stacjonarny w zestawieniu, mimo iż jest starszy niż pozostałe jednostki, również wykazuje bardzo duże zyski z przeniesienia obliczeń na GPU. Karta graficzna NVIDIA 9600GT nie jest najnowszą propozycją, jednak jako jedyna nie jest też w wersji mobilnej, które cechują znacznie gorsze parametry.

Warto zauważyć, iż komputery nowsze, o większej mocy obliczeniowej, korzystają ze zrównoleglenia obliczeń znacznie bardziej niż komputery słabsze i starsze. Jest to obiecująca tendencja, gdyż prognozuje coraz większe korzyści z przenoszenia obliczeń ogólnego zastosowania na procesory kart graficznych.

Zdecydowanie największy wpływ na przyrost wydajność podczas obliczeń równoległych ma rozmiar siatki symulacyjnej. Relacje przyspieszenia oraz rozmiaru siatki prezentuje tabela 5.9 oraz odpowiadający jej wykres 5.12. Dane uśredniono z testów na wszystkich czterech konfiguracjach sprzętowych.

Tablica 5.9: Stosunek wydajności obliczeń na GPU do CPU w zależności od rozmiaru siatki symulacyjnej

Siatka	**GPU/CPU**
100x100	**2.94**
256x256	**8.79**
512x512	**9.21**
1024x1024	**9.88**



Rysunek 5.12: Stosunek wydajności obliczeń na GPU do CPU w zależności od rozmiaru siatki symulacyjnej

Wyniki udowadniają wartość zastosowanej optymalizacji. Już dla najmniejszej siatki wzrost wydajności jest blisko **trzykrotny**. Dla większych siatek wydajność jest blisko **dziesięć razy większa** niż podczas obliczeń na CPU.

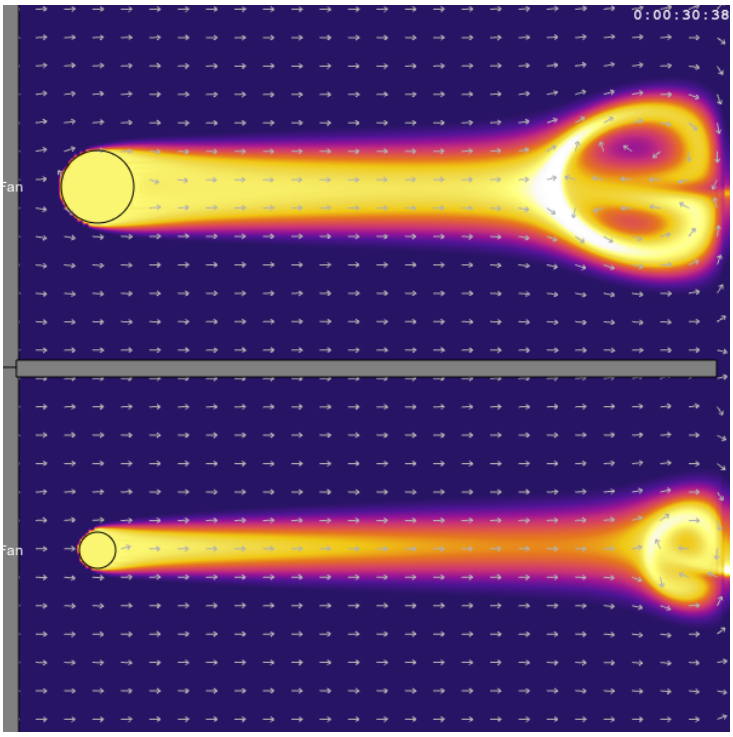
Odpowiednio gęsta siatka pokazuje potencjał jednostek graficznych. Ponadto, lepsze wyniki wiążą się też zapewne z niwelowaniem znaczenia instrukcji w języku *JavaScript* wspólnych dla wersji zarówno sekwencyjnej jak i równoległej – gdy rozmiar siatki jest odpowiednio duży, procent czasu poświęcony na wykonywanie tych instrukcje staje się pomijalny.

5.2.5. Wpływ przeniesienia obliczeń na GPU na jakość symulacji

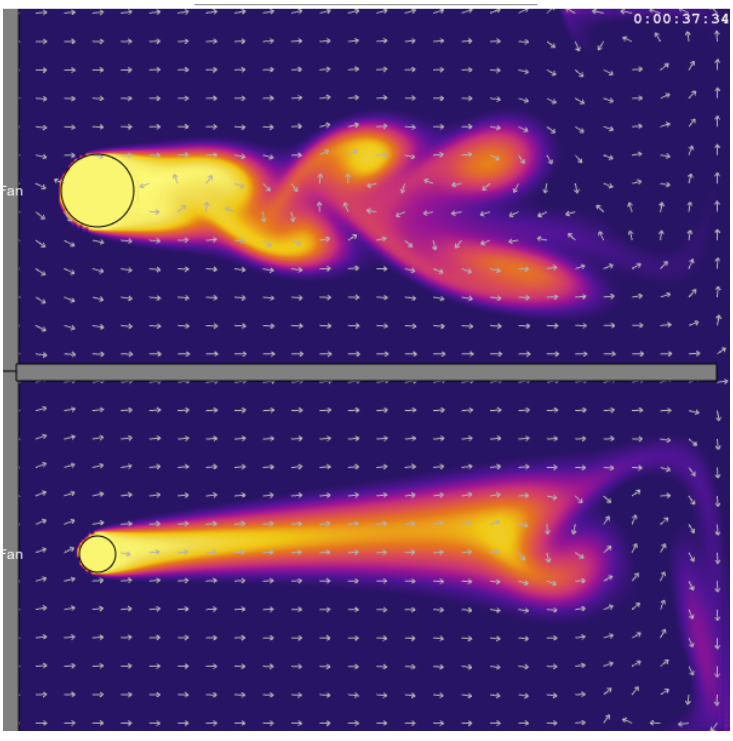
Oczywiście zrównoleglenie symulacji przekłada się na jej szybsze wykonywanie. Dokładne wyniki przedstawia poprzednia sekcja 5.2.4. W kontekście aplikacji edukacyjnej jest to bardzo ważne – symulator staje się bardziej „interaktywny", a oczekiwane rezultaty widać dużo szybciej. Doświadczenia i analizy pokazują, że zbyt wolne działanie aplikacji bardzo często skutkuje zniechęceniem użytkownika.

Dzięki przeniesieniu obliczeń na GPU, symulacja przyspieszyła około trzykrotnie już dla siatek 100x100. Często tylko zastosowanie tej optymalizacji pozwala, aby animacja była odtwarzana z prędkością większą niż 25 –30 klatek na sekundę. Jest to graniczna wartość, dla której ludzkie interpretują animację jako płynną. Dlatego też zysk z optymalizacji jest nie do przecenienia.

Użycie większej siatki symulacyjnej skutkuje jeszcze większym wzrostem wydajności podczas obliczeń równoległych. Wydawać by się mogło, iż większa siatka powinna przynieść automatycznie lepszą jakość symulacji. Jednak przy zastosowanych algorytmach fizycznych, większa siatka wprowadza też pewne problemy – trudniej jest symulować wybrane zjawiska fizyczne. Potrzebne jest znacznie dokładniejsze rozwiązywanie układów równań liniowych (więcej kroków relaksacji), co powoduje zwielokrotnienie czasu potrzebnego na obliczenia. Efekt ten przedstawiają rysunki 5.13 oraz 5.14. Przy obecnej wydajności procesorów głównych oraz graficznych, powoduje to iż używanie gęstych siatek symulacji staje się niepraktyczne. Ponadto, efekt wizualny nie jest wart tak znacznego spadku wydajności. Dlatego też większość przykładów korzysta z siatek 100x100. Taki rozmiar zapewnia dobrą jakość wizualną i merytoryczną symulacji, a dzięki przeniesieniu obliczeń na GPU również płynność działania.



Rysunek 5.13: Niewłaściwie formowanie się ścieżki wirowej von Karmana (GPU, siatka 256x256, 10 kroków relaksacji)



Rysunek 5.14: Właściwe formowanie się ścieżki wirowej von Karmana (GPU, siatka 256x256, 60 kroków relaksacji)

5.3. Ocena dostępności aplikacji dla potencjalnych użytkowników

Aplikacja do działania wymaga wyłącznie przeglądarki internetowej. Te są dostępne na każdym wiodącym systemie operacyjnym. Nie jest wymagana też żadna instalacja czy też uprawnienia administratora, aby uruchomić symulację. Dlatego też potencjalnemu użytkownikowi wystarczy komputer średniej klasy z zainstalowaną przeglądarką internetową oraz dostępem do internetu. Biorąc pod uwagę możliwości aplikacji, są to niewątpliwie niskie wymagania.

Wpływ posiadanej konfiguracji sprzętowej i oprogramowania na działanie symulatora jest znaczny. Gdy nie jest możliwe korzystanie z akceleracji na karcie graficznej, wydajność silnika *JavaScript* zastosowanego w przeglądarce internetowe ma decydujące znacznie. Testy przedstawia sekcja 5.2.3. Najlepszym wyborem w tym momencie jest przeglądarka Google Chrome, jednak w związku z dynamicznym rozwojem wszystkich wiodących przeglądarek, sytuacja ta może się zmienić w niedalekiej przyszłości.

Jednak problem zależności od konkretnej przeglądarki likwiduje przeniesienie obliczeń na kartę graficzną. Interpreter *JavaScript* traci na znaczeniu, jednak wymagane jest wsparcie przeglądarki dla technologii *WebGL*. W tym momencie zapewnia je Google Chrome, Mozilla Firefox, Safari oraz Opera (jeszcze w fazie eksperymentalnej). Czyli przeglądarki, które posiadają zdecydowaną większość rynku. Ponadto, wsparcie dla *WebGL* w niedalekiej przyszłości powinna zapewniać każda, licząca się przeglądarka internetowa.

Przy użyciu tej optymalizacji, ciężar obliczeń spada na kartę graficzną. Jednak nawet słabe, energooszczędne karty graficzne przeznaczone do laptopów radzą sobie z tym zadaniem doskonale. Dlatego też nie powinno być to problemem dla użytkownika. Uśredniając zgromadzone wyniki testów (por 5.2.4), karty graficzne osiągnęły od trzech do dziesięciu razy lepszą wydajność niż procesory główne i zapewniły doskonałą płynność symulacji.

6. Wnioski

6.1. Podsumowanie

6.2. Dalszy rozwój

Bibliografia

[AG09] G. Amador and A. Gomes. Linear Solvers for Stable Fluids: GPU vs CPU. In *Proceedings of the 17th Encontro Português de Computação Gráfica (EPCG'09)*, pages 145–153. Instituto de Telecomunicacoes, Departamento de Informatica, Universidade da Beira Interior, 2009.

[Bro12] Web Browsers Statistics and Trends. `http://www.w3schools.com/browsers/browsers_stats.asp`, 2012.

[Har04] Mark J. Harris. Fast Fluid Dynamics Simulation on the GPU. In *GPU Gems*, chapter 38, pages 637–665. Addison-Wesley, 2004.

[Har05] Mark J. Harris. Mapping Computational Concepts to GPUs. In *GPU Gems 2*, chapter 31, pages 493–508. Addison-Wesley, 2005.

[MC89] Robert W. MacCormack and Graham V. Candler. The solution of the Navier-Stokes equations using Gauss-Seidel line relaxation. *Computers & Fluids*, 17(1):135–150, 1989.

[Web11] WebGL 1.0 Specification. `https://www.khronos.org/registry/webgl/specs/1.0/`, 2011.

[Zuc09] Andrew A. Zucker. Transforming schools with technology. *Independent School*, 2009.