Politechnika Łódzka
Instytut Informatyki

Wykorzystanie GPU urządzeń mobilnych w symulacji dynamiki tkanin

Praca dyplomowa inżynierska

Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej
Promotor: dr inż. Dominik Szajerman
Dyplomant: Marcin Wawrzonowski
Nr albumu: 180729
Kierunek: Informatyka
Specjalność: Technologie Gier i Symulacji Komputerowych

Łódź, 2016

Instytut Informatyki
90-924 Łódź, ul. Wólczańska 215, budynek B9
tel. 042 631 27 97, 042 632 97 57, fax 042 630 34 14 email: office@ics.p.pl

Spis treści

Rozdział 1

Wprowadzenie

Szybki rozwój technologiczny i wydajnościowy sprzętu komputerowego zazwyczaj idzie w parze z pojawianiem się coraz większej liczby jego zastosowań. Twórcy wizualizacji graficznych oraz gier komputerowych ciągle próbują jak najwierniej oddać wirtualne środowisko wraz ze wszystkimi jego elementami. Wśród nich znajdują się także różnego rodzaju obiekty złożone z materiałów tekstylnych, na przykład ubrania postaci, dywany, gobeliny czy sztandary. Realistyczna animacja ich ruchu oraz interakcji z pozostałymi obiektami, charakterystyczne zmarszczki i zagięcia stanowią dodatek bardzo istotny w zwiększaniu autentyczności wirtualnej sceny.

Jednakże ręczne tworzenie tej dynamiki przez animatora to bardzo trudne i zazwyczaj nie odznaczające się satysfakcjonującymi efektami zajęcie. W związku z tym zaproponowano modele symulacji dynamiki tkanin, gdzie położenie wyznaczonych punktów masy ubrania czy materiału jest obliczane w czasie rzeczywistym. Niestety, koszt obliczeniowy cechuje szybki wzrost wraz ze zwiększaniem dokładności. Rozwiązanie to przeprowadzanie obliczeń przy użyciu kart graficznych – GPU, jako że przesunięcia poszczególnych wierzchołków siatki wielokątowej tkaniny można obliczać niezależnie od siebie, a wieloprocesorowe urządzenie komputera PC idealnie się do tego nadaje.

Od dłuższego czasu urządzenia mobilne także zaczęto wyposażać w wyspecjalizowane układy GPU. Głównym zadaniem niniejszej pracy jest sprawdzenie, czy także i one wydajnościowo podołają zadaniu zasymulowania ruchu tkaniny oraz jej reagowania z otoczeniem.

1.1. Cel pracy

Symulacja tkanin to zagadnienie, z którego dopracowaniem już od dawna boryka się wielu ekspertów. Skutkiem jest istnienie kilku różnych rozwiązań tego problemu, każde ma swoje wady i zalety. Niniejsza praca postawiła sobie za cel zaimplementowanie dwóch metod symulacji – modelu masy na sprężynie i opartego na pozycji. Pierwszy z nich cechuje prostota koncepcyjna, drugi – wydajność, elastyczność i odporność na błędy numeryczne. Dokonane

zostanie porównanie pod względem szybkości działania, stabilności oraz, subiektywnie, wierności odwzorowania rzeczywistego zachowania tkaniny.

Drugim kluczowym elementem niniejszej pracy jest wykorzystanie GPU do przeprowadzenia obliczeń, ponieważ teoretycznie charakteryzują się one dużo większą wydajnością dla zagadnień mogących być przetwarzanymi równolegle. Bez wątpienia taka tendencja istnieje na platformie PC, gdzie prym wiodą wysokowydajne karty graficzne. Co w takim razie z coraz bardziej popularnymi urządzeniami mobilnymi? Instalowane tam układy GPU nie cechuje duża szybkość, a raczej niski pobór energii i mały rozmiar. Czy także tutaj uzyskają przewagę wydajnościową nad procesorami CPU? Na przykładzie symulacji tkanin postarano się w niniejszej pracy na to pytanie odpowiedzieć, implementując trzy różne wersje symulatora – z użyciem GPU, sekwencyjnie na CPU oraz wielowątkowo na CPU. Następnie porównano wydajność ich pracy dla zbiorów danych o różnej wielkości, co pozwala dowiedzieć się, czy faktycznie wykorzystanie kart graficznych urządzeń mobilnych do obliczeń GPGPU ma sens.

Ostatnią kwestią była także chęć pokazania dokładnych różnic między platformami PC oraz mobilną. Stworzona została osobna wersja programu na każdą z nich tak, aby współdzieliły ze sobą jak największą część kodu i działały możliwie identycznie. Wyciągając wnioski z implementacji oraz testowania obu, można ocenić dokładną przewagę wydajnościową tej pierwszej, a także omówić unikalne możliwości interakcji z użytkownikiem drugiej, zwracając jednak uwagę na występujące tu ograniczenia sprzętowe i programowe.

1.2. Założenia

Została stworzona aplikacja prezentująca symulację tkanin z użyciem dwóch różnych metod, jej kolizje z obiektami wirtualnej sceny oraz reakcje na interakcję użytkownika. W skład prezentowanej sceny, oprócz symulowanej tkaniny wchodzą płaska podłoga oraz przedmiot kolizyjny reprezentowany przez sferę bądź prostopadłościan. Użytkownik wpływa na zachowanie tkaniny, przesuwając ten obiekt przy pomocy przycisków, bądź wykonując ruchy palcem po ekranie dotykowym. Ponadto istnieje możliwość zmiany różnych parametrów, takich jak model symulacyjny, gęstość siatki czy współczynnik sztywności. Da się także ustawić, czy symulacja będzie przetwarzana przez procesor, czy kartę graficzną. Sam program występuje w wersjach na dwie różne platformy – mobilną i PC, reprezentowane odpowiednio przez systemy operacyjne Android oraz Windows. Urządzenia testowe to smartfon LG Nexus 4 i średniej klasy komputer PC.

Założeniami pracy były także, w miarę możliwości, stworzenie samemu całego silnika, w którym można osadzić symulację oraz osiągnięcie jak największej wydajności. W związku z tym zdecydowano się na użycie natywnych bibliotek Androida – Android NDK i API

graficznego OpenGL ES 3.0, zapewniających szczegółową kontrolę nad działaniem aplikacji, a także dużą szybkość przetwarzania. Chcąc mieć jak najlepsze porównanie, na platformie Windows użyto w tym celu OpenGL w starszej już dziś wersji 3.3, z racji tego iż zbiorem funkcji odpowiada ona edycji ES 3.0.

1.3. Zakres pracy

Aby Czytelnik nie miał problemów ze zrozumieniem całości zagadnienia, rozdział 2 zawiera w sobie całą wiedzę teoretyczną dotyczącą omawianych dziedzin. Przedstawione zostaną koncepcyjne i matematyczne podstawy obu modeli symulacji tkanin oraz system detekcji jej kolizji z otoczeniem i samą sobą, wraz z odpowiednimi niezbędnymi strukturami kolizyjnymi. Następnie zwrócona jest uwaga na problematykę programowania ogólnego przeznaczenia na GPU (GPGPU), które cechuje inne podejście, niż klasyczne sekwencyjne, omówiona zostanie architektura przykładowego GPU wraz ze strukturą oraz kolejnością wykonywanych na nim działań. Jako przykłady podane zostaną technologia CUDA oraz klasyczny potok renderingu, którego część także można wykorzystać do GPGPU. Na koniec porównano architektury, wydajności i charakterystyczne cechy CPU oraz GPU ze wskazaniem kluczowych różnic oraz omówiono, jak wykorzystanie GPU w symulacji tkanin pozwala przyspieszyć proces obliczeniowy.

Rozdział 3 skupia się na wykorzystanych w niniejszej pracy technologiach. Dokonano analizy możliwości urządzeń mobilnych w kwestii GPGPU, zastanawiając się nad sensownością ich użycia do wspomnianych zastosowań, scharakteryzowano urządzenie testowe i porównano je z przykładową konfiguracją PC. Skupiono się także na przewagach platformy mobilnej, omawiając jej unikalne możliwości interakcji oraz dostępność. Następnie opisane zostały wszystkie użyte na potrzeby pracy technologie programistyczne – API, biblioteki, kluczowe funkcje oraz środowisko programowania.

W rozdziale 4 zaprezentowano architekturę autorskiego silnika aplikacji, w którym osadzono symulator tkanin. Przedstawiono wszystkie najważniejsze komponenty, każdy pokrótce scharakteryzowano, zwrócono także uwagę na połączenia między nimi. Kluczowym dla zrozumienia działania programu elementem są tu diagramy klas oraz ogólne algorytmy pracy silnika. Następnie skupiono się na modelu encji systemu, zaprojektowanym z myślą o złożeniu każdej z poszczególnych komponentów. Ważną omawianą kwestią jest także komunikacja aplikacji z systemem Android, czyli obsługa zdarzeń. Na koniec opisano, wraz z podaniem odpowiednich algorytmów, proces renderingu poszczególnych obiektów 3D oraz scharakteryzowano system interfejsu użytkownika, zwracając uwagę zarówno na to, co da się z jego pomocą stworzyć, jak i na to, co w efekcie użytkownik może robić w programie.

W rozdziale 5 skupiono się wyłącznie na samym symulatorze tkanin. Zebrano w jedną całość wszystkie poczynione założenia projektowe, omówiono je i następnie pokazano, jak zostały one zrealizowane. Podano algorytmy działania symulatora w każdej z trzech implementacji, opisano wszystkie parametry biorące udział w symulacji oraz ich wpływ na jej wyniki. Sam ten proces podzielono na trzy etapy, każdy z nich dokładnie scharakteryzowano, załączono też fragmenty kodu GLSL, uruchamianego na karcie graficznej.

Rozdział 6 zawiera wyniki działania symulatora tkanin. Zebrano je w trzech kategoriach. Czas wykonania obrazują tabele oraz wykres, gdzie ukazana została jej zależność od gęstości symulowanej siatki dla każdej z implementacji. Podobnie wygląda sytuacja w kwestii stabilności, gdzie jako miarę przyjęto średnie oscylacje przykładowego wierzchołka, w zależności od współczynnika sztywności. Ostatnie kryterium – efekt wizualny obrazują zrzuty ekranu wykonane dla różnych metod oraz parametrów, dających pełne spektrum tego, co można uzyskać przy pomocy symulatora.

Wyniki pozwalają wyciągnąć wnioski na temat przewagi jednej metody, bądź implementacji, nad drugą. Właśnie tego dokonano w rozdziale 7, będącym podsumowaniem pracy. Przy użyciu danych z rozdziału 6, przeprowadzono porównanie metod masy na sprężynie i opartej na pozycji, implementacji CPU oraz GPU, zwracając uwagę na ich wady, zalety, wymieniając możliwe zastosowania oraz mówiąc, co można poprawić, by uzyskać lepszy efekt. Porównano także implementacje na różnych platformach – Windows oraz Android, głównie pod kątem wydajności, oraz finalnie zastanowiono się nad sensownością wykorzystania oraz praktycznymi możliwościami tej drugiej w kwestii symulacji tkanin.

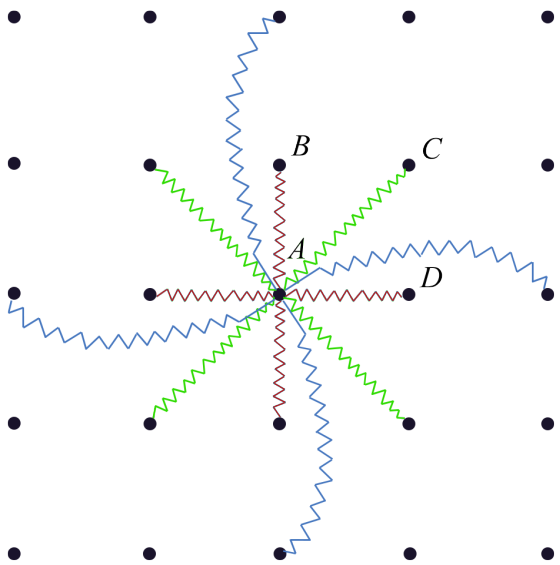
Rozdział 2

Algorytmy symulacji tkanin

2.1. Analiza istniejących modeli symulacji tkanin

2.1.1. Model masy na sprężynie

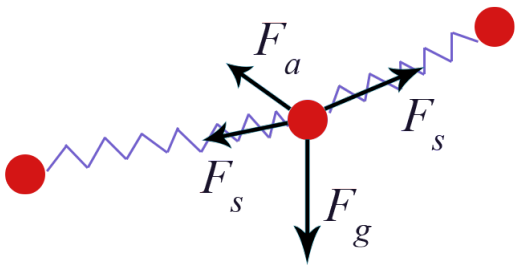
Pierwszym z rozważanych w niniejszej pracy modeli symulacji tkanin jest model masy na sprężynie. Rysowana przez API graficzne tkanina jest w postaci siatki wielokątowej. Siatka taka składa się z punktów w przestrzeni 3D – wierzchołków. Na potrzeby symulacji przyjęto, że każdy z tych wierzchołków ma określoną masę i poddano go działaniu sił, w wyniku których następuje przemieszczenie. Aby zachować kształt i odpowiednie dla tkaniny zachowanie się siatki, wierzchołki połączone są sprężynami o określonych współczynnikach sprężystości oraz tłumienia drgań.



Rysunek 2.1. Schemat modelu masy na sprężynie. Kolorami zaznaczono wszystkie sprężyny biorące udział w obliczeniach położenia wierzchołka A. Źródło: opracowanie własne.

Rysunek 2.1 przedstawia przykładowy fragment tkaniny opartej o model masy na sprężynie. Założono dla uproszczenia, iż poszczególne wierzchołki tworzą kształt prostokątów, aczkolwiek w praktyce mogą przyjmować dowolne ustawienia. Ważne jednak dla zachowania poprawnej symulacji jest to, by punkty masy były równomiernie rozłożone na całej powierzchni tkaniny. W omawianym przypadku tak właśnie jest.

Można zaobserwować trzy rodzaje sprężyn, jakie występują w modelu na rysunku 2.1. Kolorem czerwonym zostały oznaczone sprężyny strukturalne, które służą do utrzymania ogólnego kształtu tkaniny. Jednakże one same nie są w stanie zasymulować zachowania tkaniny w poprawny fizycznie i wizualnie sposób. Kolorem zielonym narysowane zostały sprężyny odpowiedzialne za wierne oddanie zgięć tkaniny, położone są one wzdłuż diagonalnych krawędzi siatki. Kolorem niebieskim zaznaczono sprężyny, których obecność zapewnia tkaninie odpowiednią elastyczność i chroni przed nadmiernym jej rozciąganiem. Łączą one nie sąsiednie wierzchołki, lecz następne za sąsiadem, w tym samym kierunku. Każdy z rodzajów sprężyn może być opisany innymi współczynnikami sprężystości i tłumienia drgań, co pozwala na uzyskanie innych zachowań tkaniny w symulacji.



Rysunek 2.2. Siły działające na pojedynczy wierzchołek. Źródło: opracowanie własne.

Na rysunku 2.2 widać, jakie siły oddziałują na każdy punkt masy – wierzchołek siatki tkaniny. Siły można zaklasyfikować jako wewnętrzne i zewnętrzne. Z zewnętrznych wyróżniono siłę grawitacji, opisaną wzorem [1]:

$$\mathbf{F_g} = m_i \cdot \mathbf{g} \ . \tag{2.1}$$

Kolejna siła zewnętrzna to siła oporu powietrza. Zgodnie z Prawem Stokesa jest ona proporcjonalna do prędkości punktu masy oraz pewnego współczynnika oporu k [1]:

$$\mathbf{F_a} = -k_a \cdot \mathbf{v_i} \ . \tag{2.2}$$

Siłami wewnętrznymi działającymi na wierzchołki tkaniny są oczywiście siły sprężystości, wynikające z istnienia omówionych wyżej sprężyn. Do wyznaczenia jej wartości wykorzystuje

się Prawo Hooke'a, mówiące, że siła sprężystości oraz jej kierunek i zwrot są proporcjonalne do wychylenia sprężyny, tj. różnicy odległości między jej aktualną długością, a długością w stanie spoczynku [1]:

$$\mathbf{F_{se}} = -\sum_{n=0}^{n<12} k_s(|\mathbf{x_i} - \mathbf{x_j}| - l_{(i,j)}) \cdot \frac{\mathbf{x_i} - \mathbf{x_j}}{|\mathbf{x_i} - \mathbf{x_j}|} \ , \tag{2.3}$$

gdzie k_s – współczynnik sprężystości, $\mathbf{x_i}$ oraz $\mathbf{x_j}$ – położenia wierzchołków połączonych daną sprężyną, $l_{(i,j)}$ – odległość między tymi punktami w stanie spoczynku.

Zgodnie z [7], wprowadzono także siłę tłumienia drgań sprężystych, aby zminimalizować niepotrzebne, nierealistyczne drgania oraz ryzyko wymknięcia się symulacji spod kontroli. Nazywa się to „wybuchnięciem" – obliczane siły są takie, że pozycje wierzchołków dążą do nieskończoności. Tłumienie przedstawia się następującym wzorem:

$$\mathbf{F_{sd}} = \sum_{n=0}^{n<12} k_d(\frac{|\mathbf{x_i} - \mathbf{x_j}| \cdot |\mathbf{v_i} - \mathbf{v_j}|}{l_{(i,j)}}) \ , \tag{2.4}$$

gdzie oznaczenia są takie same, jak wyżej z tym, że k_d jest tutaj współczynnikiem tłumienia drgań, $|\mathbf{v_i} - \mathbf{v_j}|$ oznacza różnicę prędkości obu wierzchołków, a działanie $|\mathbf{x_i} - \mathbf{x_j}| \cdot |\mathbf{v_i} - \mathbf{v_j}|$ to iloczyn skalarny wektora różnicy położenia i wektora różnicy prędkości. Końcowa siła sprężystości jest sumą dwóch powyższych wzorów. Można zapisać ją w postaci:

$$\mathbf{F_s} = \mathbf{F_{se}} + \mathbf{F_{sd}} \ , \tag{2.5}$$

$$\mathbf{F_s} = \sum_{n=0}^{n<12} -k_s(|\mathbf{x_i} - \mathbf{x_j}| - l_{(i,j)}) \cdot \frac{\mathbf{x_i} - \mathbf{x_j}}{|\mathbf{x_i} - \mathbf{x_j}|} + k_d(\frac{|\mathbf{x_i} - \mathbf{x_j}| \cdot |\mathbf{v_i} - \mathbf{v_j}|}{l_{(i,j)}}) \ . \tag{2.6}$$

W sumie dla każdego wierzchołka rozważono 12 sił sprężystości dla wszystkich przyłączonych do niego sprężyn, siłę grawitacji oraz oporu powietrza. Należy zauważyć, że nie uwzględniono tutaj reakcji na kolizje – będą one rozwiązane w późniejszej sekcji algorytmu. Wzór na siłę wypadkową działającą na pojedynczy wierzchołek tkaniny oraz wypadkowe przyspieszenie można zapisać w postaci:

$$\mathbf{F} = \mathbf{F}s + \mathbf{F}g + \mathbf{F}d \tag{2.7}$$

$$\mathbf{a}(t) = \frac{\mathbf{F}}{m_i} \tag{2.8}$$

W celu wyznaczenia zmiany położenia punktu masy w danym kroku symulacji, posłużono się, zgodnie z [1] i [7], całkowaniem Verleta, będącym jedną z technik całkowania numerycznego nie wprost. Opisane jest ono wzorem:

$$\mathbf{x}(t + \delta t) = 2\mathbf{x}(t) - \mathbf{x}(t - \delta t) + \mathbf{a}(t)\delta t^2 \ , \tag{2.9}$$

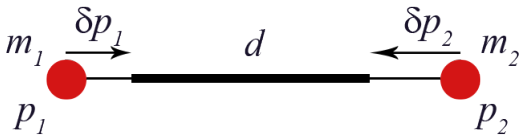
gdzie $\mathbf{a}(\mathbf{t})$ jest przyspieszeniem, a $\mathbf{x}(t+\delta t)$, $\mathbf{x}(t)$ i $\mathbf{x}(t-\delta t)$ oznaczają położenia wierzchołka w następnym, obecnym oraz poprzednim kroku symulacji.

Całkowanie Verleta jest równie proste obliczeniowo i implementacjyjnie, jak najzwyklejsza metoda Eulera, a zapewnia dużo stabilniejsze zachowanie się symulacji i minimalizację tendencji do „wybuchnięcia". Niejawnie obliczona zostaje tutaj aktualna prędkość wierzchołka, co sprawia, że do symulatora trzeba dostarczyć nie tylko obecną pozycję każdego punktu masy, ale także położenie poprzednie. Zwiększa to koszt pamięciowy symulacji względem innych technik całkowania, lecz zapewnia bardzo szybkie obliczenia i stabilne ich wyniki.

2.1.2. Model oparty na pozycji

Model oparty na pozycji i model masy na sprężynie mają pewną część wspólną – jest to obliczanie przesunięć wynikających z sił grawitacji oraz oporu powietrza za pomocą całkowania Verleta. Także tkanina ma w tym modelu taką samą postać, jak i w poprzednim – zbiór wierzchołków, połączonych krawędziami, tworzących prostokątne kształty. Podejście do symulacji ruchów wynikających ze struktury samej tkaniny jest jednak kompletnie odmienne. Ponadto zaznaczyć należy, że w niniejszej pracy została wykorzystana tylko część aktualnie omawianego modelu, bardziej szczegółowo opisanego w [2]. Prezentowane tu rozwiązanie nie bierze pod uwagę ograniczników zginania oraz całego systemu detekcji kolizji, zaproponowanego w rozdziale 4 [2].

Przesunięcia wynikające z sił zewnętrznych działających na układ nazywamy tutaj przesunięciami przewidywanymi. Każdy wierzchołek tkaniny opisywany jest, oprócz, masy, pozycji i prędkości, także przez zbiór tzw. ograniczników. Każdy z nich definiowany jest poprzez pewną funkcję $C_j : R^{3n_j} \to R$, zestaw indeksów $\{i_1, \ldots, i_{n_j}\}, i_k \in [1, \ldots, N]$ i parametr sztywności $k \in [0 \ldots 1]$. Ogranicznik może być typu równości, co oznacza, że jego ograniczenie jest spełnione, kiedy $C_j(x_{i_1}, \ldots, x_{i_{n_j}}) = 0$. Może być także typu nierówności, i spełnia je warunek $C_j(x_{i_1}, \ldots, x_{i_{n_j}}) \geq 0$. W omawianym przypadku będą rozważane tylko ograniczniki pierwszego typu. Kluczowym elementem jest oczywiście funkcja C_j, która określa sposób, w jaki przewidywana pozycja zostanie poprawiona, od czego ta poprawa zależy – czyli w efekcie zachowanie się tkaniny. Funkcja ta może być zupełnie inna, gdy obliczane są ograniczenia przesuwania, a inna przy ograniczeniach zginania bądź kolizjach. Widać, że dzięki temu model ten charakteryzuje się pewną elastycznością, a ograniczniki są narzędziem, przy pomocy którego można realizować wiele rozmaitych założeń dotyczących ruchu tkaniny. Przesunięcia wynikające z nałożonych ograniczników są obliczane po kolei, następnie pozycja przewidywana jest poprawiana do takiej, która spełnia wszystkie warunki, równości bądź nierówności, dla funkcji C_j ograniczników. Proces ten autorzy [2] nazywają projekcją. Pod uwagę brana jest także sztywność ogranicznika, czyli procent, w jakim się go stosuje.



Rysunek 2.3. Schemat działania ograniczników między dwoma punktami masy. Źródło: opracowanie własne.

Podstawowym typem ogranicznika jest ogranicznik rozciągania. To on definiuje ogólny kształt i odpowiednie zachowanie tkaniny. Jego funkcja ma postać:

$$C(\mathbf{p_1}, \mathbf{p_2}) = |\mathbf{p_1} - \mathbf{p_2}| - d \ . \tag{2.10}$$

Gdzie $\mathbf{p_1}$ i $\mathbf{p_2}$ są pozycjami rozpatrywanych wierzchołków, a d - początkową odległością między nimi. Na rysunku 2.3 widać efekt projekcji. W [2] wyprowadzone zostają wzory na jej obliczenie, na podstawie funkcji $C_j(x_{i_1}, \ldots, x_{i_{n_j}})$:

$$s = \frac{C_j(\mathbf{p_{i_1}}, \ldots, \mathbf{p_{i_{n_j}}})}{\sum_j w_j |\nabla_{\mathbf{p_j}} C_j(\mathbf{p_{i_1}}, \ldots, \mathbf{p_{i_{n_j}}})|^2} \ , \tag{2.11}$$

$$\delta\mathbf{p_i} = -s w_i \nabla_{\mathbf{p_i}} C_j(\mathbf{p_{i_1}}, \ldots, \mathbf{p_{i_{n_j}}}) \ . \tag{2.12}$$

Gdzie w_i jest odwrotnością masy wierzchołka tkaniny. Biorąc pod uwagę, iż przemieszczenie jest do niej wprost proporcjonalne, można łatwo wyobrazić sobie dowód na poprawność tego podejścia – w przypadku, gdy masa cząstki jest nieskończona, przesunięcie będzie równe zeru. Kiedy na miejsce funkcji $C_j(x_{i_1}, \ldots, x_{i_{n_j}})$ wstawi się $C(\mathbf{p_1}, \mathbf{p_2}) = |\mathbf{p_1} - \mathbf{p_2}| - d$, otrzymane zostaną po przekształceniach następujące wzory:

$$\delta\mathbf{p_1} = -\frac{w_1}{w_1 + w2}(|\mathbf{p_1} - \mathbf{p_2}| - d)\frac{\mathbf{p_1} - \mathbf{p_2}}{|\mathbf{p_1} - \mathbf{p_2}|} \ , \tag{2.13}$$

$$\delta\mathbf{p_2} = \frac{w_2}{w_1 + w2}(|\mathbf{p_1} - \mathbf{p_2}| - d)\frac{\mathbf{p_1} - \mathbf{p_2}}{|\mathbf{p_1} - \mathbf{p_2}|} \ . \tag{2.14}$$

Da się zauważyć, że wzory (2.13) i (2.14) są bardzo podobne do wzoru (2.3). Tak samo, jak w modelu masy na sprężynie, „siła" ogranicznika zależy od różnicy aktualnej odległości pomiędzy punktami masy i odległości spoczynkowej. Rolę współczynnika sprężystości pełni tutaj parametr sztywności, przez który pomnożono na koniec przesunięcie będące wynikiem

projekcji. Dla k równego 0 ogranicznik nie będzie w ogóle brany pod uwagę, a dla równego 1 – punkt nigdy nie zmieni swojej początkowej pozycji.

W rozważanym przypadku nie zastosowano ograniczników zginania, użyto także innej metody detekcji kolizji. Autorzy [2] używają ograniczników rozciągania biorąc pod uwagę tylko wierzchołki leżące w sąsiedztwie danego punktu. Okazuje się, że podobny do użycia ograniczników zginania efekt można osiągnąć zwiększając zbiór rozpatrywanych wierzchołków o te leżące jedną pozycję w siatce dalej - tak jak sprężyny oznaczone kolorem niebieskim na rysunku 2.1. Przykładowo, ograniczając elastyczność przemieszczania wierzchołka A względem C ograniczamy przecież tak naprawdę możliwość zginania się trójkątów ABD i BCD na wspólnej krawędzi BD. Nie jest to tak dokładna metoda jak ograniczniki zginania, gdzie regulujemy kąt pomiędzy trójkątami, lecz mimo to daje ona poprawny wizualny efekt, jak pokazane zostanie w rozdziale 5. Jest też szybsza do obliczenia przez procesor, jako że sam wzór jest prostszy.

2.1.3. Porównanie powyższych metod

Podsumowując przedstawienie zastosowanych metod symulacji tkaniny, należy zauważyć, że każda z nich ma swoje plusy i minusy. Największą zaletą modelu masy na sprężynie jest z pewnością prostota i łatwość implementacji. Nietrudno wyobrazić sobie tkaninę, jako zbiór wierzchołków połączonych elastycznymi sprężynami, których siły sprężystości są wyliczane przy pomocy prostych praw fizyki. Trochę gorzej jest w przypadku drugiej omawianej metody – tutaj zrozumienie zasady działania modelu wymaga wgłębienia się w wyprowadzenia wzorów matematycznych. Jednakże sama implementacja okazuje się równie prosta, ilość kodu jaką programista musi napisać, jest tak naprawdę podobna.

Z pewnością największą zaletą modelu opartego na pozycji jest przewaga wydajnościowa. Wynika ona z braku konieczności użycia całkowania numerycznego. Zachowanie tkaniny nie jest określane przez zbiór sił sprężystości, a ograniczników, od razu modyfikujących pozycję. Całkować trzeba co najwyżej chcąc obliczyć siły zewnętrzne. Pozwala to na duże oszczędności obliczeniowe. W przypadku modelu masy na sprężynie nie można uniknąć już tych obliczeń dla każdej ze sprężyn. W celu uzyskania dokładniejszych wyników należy zastosować bardziej skomplikowane metody całkowania. Prowadzi to do spadku wydajności.

Wyłania się tutaj także kolejna zaleta metody numer dwa. Numeryczne obliczanie przemieszczenia punktu masy wiąże się z ryzykiem utraty stabilności symulacji i jej „wybuchnięciem". Przez większość czasu może ona pracować bezawaryjnie, dopóki nie zajdą pewne szczególne okoliczności, kiedy to wymyka się spod kontroli. Przykładem jest znaczna, skokowa zmiana parametru δt we wzorze (2.9) Doprowadzi ona do błędnego obliczenia wartości sił we wszystkich sprężynach – będzie ona za duża, co z kolei spowoduje niekontrolowane drgania, a w konsekwencji – „wybuch". W modelu opartym na pozycji

korekcja przemieszczenia nie jest obliczana numerycznie i gwarantuje to dużą stabilność działania tej metody.

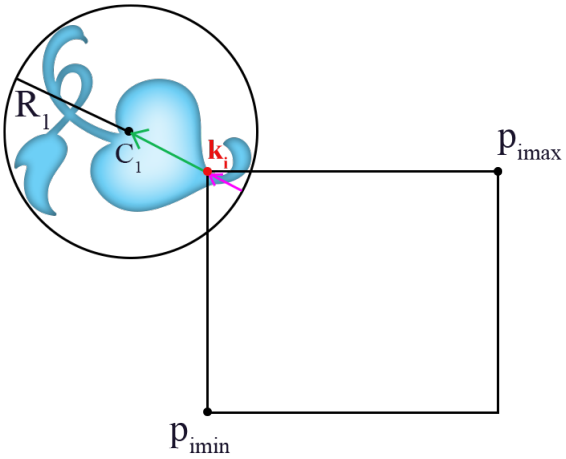
Jak widać, druga rozważana metoda wydaje się mieć przewagę nad pierwszą. Model oparty na pozycji daje nam bezpieczną i wydajną obliczeniowo symulację, dlatego jest stosowany w wiodących na rynku, komercyjnych silnikach fizycznych. Jednakże model masy na sprężynie dużo łatwiej zrozumieć początkującemu programiście, a do prostych zastosowań o stałych parametrach nadaje się on równie dobrze.

2.1.4. Detekcja kolizji

Kluczowym aspektem symulacji tkaniny jest wykrywanie kolizji i rozwiązywanie ich, czyli przemieszczanie wierzchołków tak, by obiekty się nie przenikały. Zwiększa to wrażenie autentyczności, realizmu symulacji z punktu widzenia użytkownika. Do uzyskania odpowiedniego efektu musimy jednak zadbać także o stabilność algorytmu rozwiązywania kolizji, chociażby po to, aby uniknąć znanego chyba wszystkim użytkownikom wirtualnych symulacji i gier efektu „podrygiwania". Obliczenia mogą pochłonąć tu bardzo dużo czasu procesora, dlatego kluczowy jest dobór odpowiednio wydajnego algorytmu, szczególnie, że w niniejszej pracy rozważa się implementację symulacji na urządzeniu mobilnym. Dlatego właśnie w tej kwestii uwagę poświęcono najprostszym rozwiązaniom.

2.1.4.1. Kolizje zewnętrzne

Poprzez kolizje zewnętrzne rozumiemy kolizje wierzchołków tkaniny z innymi obiektami sceny, np. podłogą pomieszczenia. Wykrywanie ich oraz generowanie odpowiedniej reakcji jest absolutnie niezbędne do zasymulowania jakiejkolwiek interakcji z resztą wirtualnego świata. W tym celu zastosowano sfery okalające, zwane dalej BS od angielskiego *Bounding Sphere* oraz prostopadłościany okalające, prostopadłe do osi układu współrzędnych, zwane dalej AABB od angielskiego *Axis-Aligned Bounding Box*.



Rysunek 2.4. Przykład kolizji sfer okalających. Źródło: opracowanie własne.

Sfery okalające są najprostszą w implementacji i najszybszą obliczeniowo metodą detekcji kolizji. Na rysunku 2.4 widać, jak wygląda przykładowy układ dwóch BS, oraz w jaki sposób ustawiona może być BS względem obiektu. Wykrycie przecięcia się polega na porównaniu sumy długości promieni R_1 i R_2 obu sfer z odległością pomiędzy ich środkami d. Jeżeli

$$R_1 + R_2 < d \ , \tag{2.15}$$

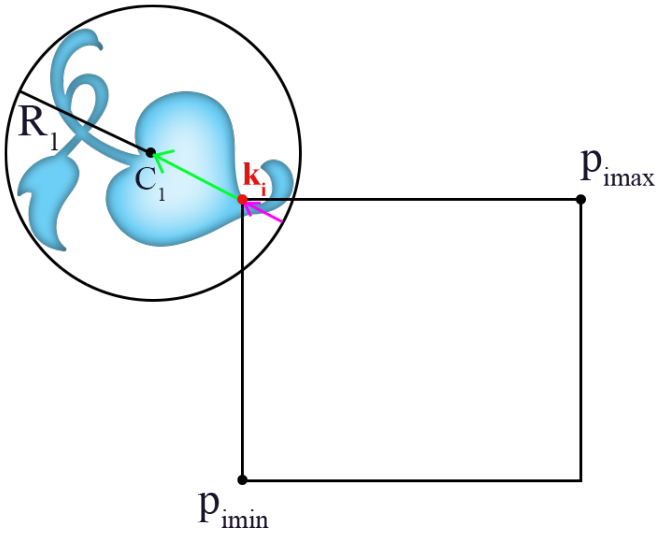
to znaczy, że doszło do kolizji i sferę, która przemieszczając się, do niej doprowadziła, należy odpowiednio przesunąć tak, by się one nie przecinały. Mowa tutaj o tzw. rozwiązaniu kolizji. Dokonano tego w prosty sposób [8]:

$$\mathbf{p}_i = \mathbf{p}_i + (\frac{\mathbf{p}_{C1} - \mathbf{p}_{C2}}{|\mathbf{p}_{C1} - \mathbf{p}_{C2}|})(R_1 + R_2 - d) \ . \tag{2.16}$$

Jak można się domyślić, wadą sfer okalających jest ich niska dokładność – obiekty idealnie kuliste rzadko występują w świecie rzeczywistym. Na potrzeby symulowania świata otaczającego tkaninę w niniejszej pracy jednak takie rozwiązanie w zupełności wystarcza. Przyjęto, że każdy wierzchołek posiada swoją własną BS, o pozycji środka identycznej z pozycją punktu oraz o promieniu równym pewnemu procentowi najkrótszej odległości między dwoma wierzchołkami, mniejszemu niż jej połowa. W każdym kroku symulacji każdy z nich zostaje sprawdzony, czy nie zaszła kolizja z którymś, bądź wieloma, obiektami sceny. Jeśli tak, poprawia się jego położenie.

Jest to rozwiązanie szybkie, jednak ma jedną istotną wadę - BS nie pokrywają całej powierzchni siatki tkaniny, a tylko obszary w okolicach wierzchołków. Kolizje na środkach trójkątów nie są sprawdzane, co może doprowadzić do przenikania obiektów przez tkaninę przy niskiej jej rozdzielczości. Założono jednak, że jest to rozwiązanie wystarczające – nie obciążamy urządzenia mobilnego nadmiarem obliczeń związanych z detekcją kolizji. Poza

tym gęstość siatki potrzebna do odwzorowania realistycznego wyglądu tkaniny jest na tyle duża, by szczeliny pomiędzy BS były na tyle małe, aby sensownie zminimalizować ryzyko wystąpienia przenikania.



Rysunek 2.5. Przykład kolizji BS z prostopadłościanem AABB. Źródło: opracowanie własne.

Obiekty zewnętrzne znajdujące się w świecie symulacji mogą posiadać dwa rodzaje detektorów kolizji. Są albo sferami okalającymi, w którym to przypadku do wykrycia przecięć zastosowane zostają opisane powyżej wzory (2.15) i (2.16), albo AABB – prostopadłościany, których położenie i rozmiary są wyznaczane przez dwa punkty w przestrzeni: minimum $\mathbf{p}_{i_{min}}$ oraz maksimum $\mathbf{p}_{i_{max}}$. W jaki sposób figury są tworzone na podstawie tych dwóch pozycji – widać na rysunku 2.5. Można zauważyć, że ten sposób interpretacji dwóch wymienionych wyżej danych skutkuje niską zajętością pamięciową (tylko dwa wektory trójwymiarowe reprezentują AABB) oraz prostotą wzorów wykrycia przecięcia, a co za tym idzie – szybkością obliczeń tychże. Ewidentną wadą jest jednak znowu niska dokładność – fakt, że kolejne ścianki prostopadłościanu zawsze będą prostopadłe do kolejnych osi układu współrzędnych. Utrudnia to odwzorowanie dokładnych kolizji np. dla obiektów pochyłych. Detekcja i rozwiązanie przecięć pomiędzy AABB i BS następuje według nieco innych zasad, niż w przypadku dwóch BS [8, 9]:

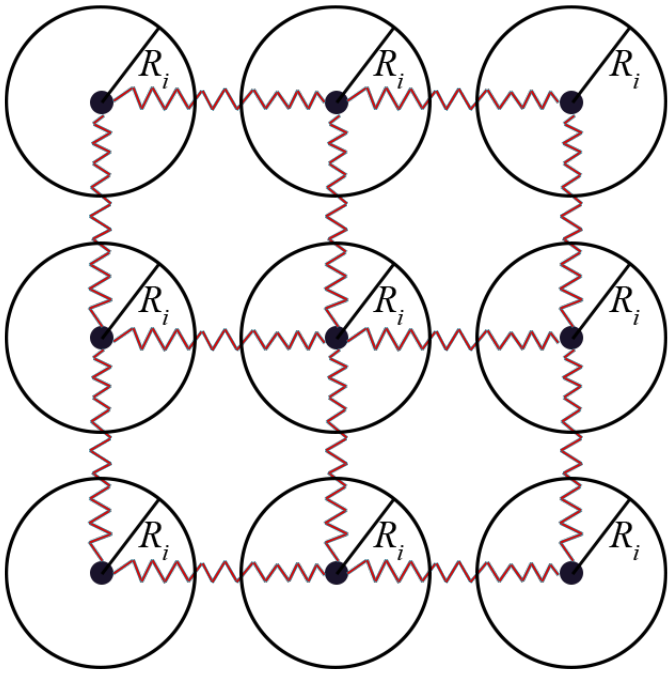
$$\mathbf{k}_i = min(max(\mathbf{p}_{C1}, \mathbf{p}_{i_{min}}), \mathbf{p}_{i_{max}}) \ . \tag{2.17}$$

$$\mathbf{p}_i = \mathbf{p}_i + \frac{\mathbf{p}_{C1} - \mathbf{k}_i}{|\mathbf{p}_{C1} - \mathbf{k}_i|}(R1 - |\mathbf{k}_i - \mathbf{p}_{C1}|) \ . \tag{2.18}$$

Punkt $\mathbf{k}_i$ obliczany wzorem (2.17) jest widoczny na rysunku 2.5, zaznaczony na czerwono. Wektor reakcji na kolizję otrzymano następnie dzięki odjęciu pozycji sfery i punktu $\mathbf{k}_i$ (zielony

wektor) oraz odpowiedniej zmianie jego długości. Ta ostatnia została dokonana poprzez kolejno: normalizację i pomnożenie przez różnicę promienia sfery okalającej i długości wektora $\mathbf{k}_i - \mathbf{p}_{C1}$. Działanie to obrazuje wzór (2.18), a wynik oznaczono na rysunku 2.5 wektorem w kolorze magenta.

2.1.4.2. Kolizje wewnętrzne



Rysunek 2.6. Ułożenie sfer kolizyjnych w wierzchołkach tkaniny. Źródło: opracowanie własne.

Podczas bardziej dynamicznych i gwałtownych ruchów symulowanej siatki, może dochodzić do przenikania się przez siebie jej fragmentów. Rozwiązując kolizje wewnętrzne, postarano się wyeliminować ten niepożądany artefakt wizualny. Na rysunku 2.6 widać, jak wygląda mechanizm dla pojedynczego wierzchołka. W sekcji 2.1.4.1 ustalono, że wszystkie punkty masy posiadają swoją własną sferę okalającą. Tutaj dalej z tego faktu skorzystano, po prostu rozwiązując kolizje BS każdego wierzchołka z BS jego ośmiu sąsiadów położonych w najbliższym otoczeniu. Zapewnia to redukcję minimalnych przecięć w bezpośrednim sąsiedztwie punktów masy, które pojawiałyby się już przy swobodnym ruchu tkaniny, z oddziałującymi tylko siłami grawitacji i oporu powietrza.

Zastosowany mechanizm z oczywistych względów nie jest doskonały. Nie zabezpiecza przed przenikaniem się w sytuacji, gdy np. róg tkaniny przemieści się przez jej środek. Jednakże zarówno bardziej wyrafinowane metody detekcji przecięć, jak i sprawdzanie kolizji BS – BS na zasadzie „każdy z każdym” wiążą się z bardzo dużym narzutem obliczeniowym,

wielokrotnie większym niż przesuwanie wierzchołków będące efektem działania modelu symulacji. W związku z tym na potrzeby niniejszej pracy została użyta bardzo prosta, acz niedoskonała, opisana powyżej technika. Biorąc pod uwagę, iż tkanina zawieszona jest na stałe za dwa, nie poruszające się, górne narożniki, prawdopodobieństwo wystąpienia przecięcia nie jest duże. Założono więc, że sprawdzanie kolizji tylko w najbliższym otoczeniu każdego z wierzchołków okaże się wystarczające wizualnie.

2.2. Obliczenia GPGPU a symulacja tkanin

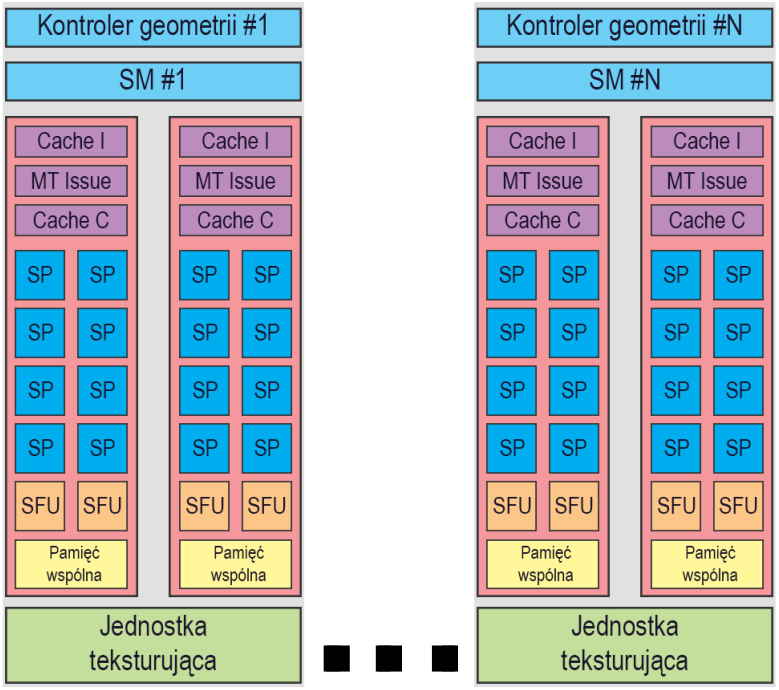
2.2.1. Definicja GPGPU

Pierwsze układy GPU – *Graphics Processing Unit* – zwane też kartami graficznymi, pojawiły się już w latach siedemdziesiątych XX wieku [10]. Od początku były to wyspecjalizowane urządzenia peryferyjne służące do wyświetlania grafiki, początkowo 2D, później, w miarę ewolucji, 3D. Przez bardzo długi czas pracowały one w układzie zamkniętym – programista dostarczał tylko określone dane, potrzebne do narysowania obiektu, np. bufory pozycji i indeksów wierzchołków, a GPU rysowało je w taki sposób, w jaki zostało zaprogramowane w fabryce. Twórca gry bądź wizualizacji graficznej, nie miał kompletnie wpływu na przebieg procesu rysowania, ani na to, jakie dokładnie obliczenia zostaną wykonane.

Zmieniło się to na początku XXI wieku, wraz z premierą układu NVIDIA GeForce 3 [10]. Wprowadził on otwarty, programowalny potok renderingu. Pisząc tzw. vertex i pixel shadery, programista od tego momentu mógł kontrolować sposób, w jaki rysowanie przebiegało, mając możliwość ulepszenia starych, wysłużonych technik.

Chcąc sprostać zapotrzebowaniu na coraz to bardziej realistyczną grafikę, producenci z biegiem lat szybko zwiększali moc obliczeniową kart graficznych. Były one jednak cały czas urządzeniami przeznaczonymi ściśle do generowania grafiki. Aby wykorzystać ich potęgę do innych celów niż rendering, należało dopasować pisane programy do API graficznego, co bardzo utrudniało proces ich powstawania. Dopiero wejście na rynek technologii takich, jak NVIDIA CUDA bądź OpenCL zapewniło dostęp do wszystkich funkcji GPU z wykorzystaniem danej platformy, niezwiązanej z renderingiem i niewymagającej jego znajomości. Obliczenia ogólnego przeznaczenia, najczęściej wykonywane za pomocą wymienionych wyżej API, nazywa się właśnie GPGPU – *General Purpose computing on Graphics Processing Units*. Ogólnie można powiedzieć, że każdy język, czy technologia, która pozwala na pobranie wynikowych danych programu uruchamianego na GPU daje możliwość wykorzystania go do GPGPU [10]. Mają one zastosowanie w wielu dziedzinach, chociażby w modelowaniu matematycznym, statystyce, medycynie bądź silnikach i symulacjach fizycznych – chociażby właśnie modelowania tkaniny.

2.2.2. Architektura i programowanie GPU



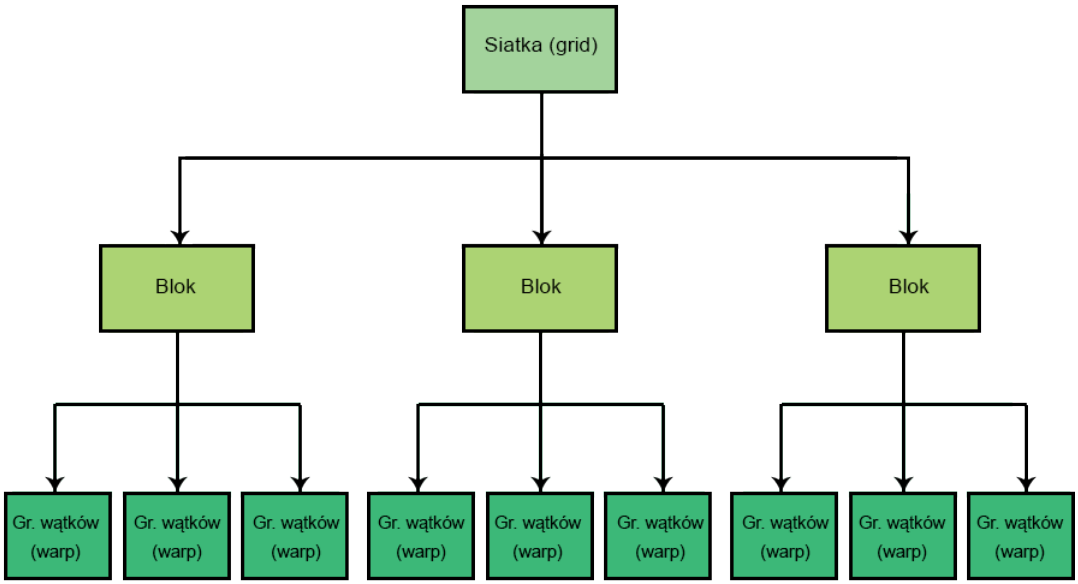
Rysunek 2.7. Typowa architektura GPU Źródło: opracowanie własne w oparciu o źródło zewnętrzne[1].

GPU jest układem złożonym z wielu jednostek obliczeniowych, pracujących ze sobą równolegle. Zrównoleglenie obliczeń wynika z charakterystyki pierwotnego zastosowania GPU, czyli renderingu grafiki. Obliczenia na każdym wierzchołku, bądź fragmencie obrazu są zazwyczaj niezależne od siebie i takie same dla każdego z nich. Można więc dla tychże elementów identyczny ciąg instrukcji wykonywać równolegle, dostarczając do niego inne dane. GPU jest sztandarowym przykładem wdrożenia architektury SIMD (*Single Instruction, Multiple Data* [12]). Jak się szybko okazało, ten model przetwarzania znalazł zastosowanie nie tylko przy rysowaniu trójkątów, ale też w wielu innych dziedzinach, często niezwiązanych z grami czy wizualizacjami.

Na rysunku 2.7 widać budowę przykładowego układu, jednakową dla większości tego typu urządzeń. Nadrzędną jednostką, z której składa się GPU jest multiprocesor strumieniowy (SM – *Streaming Multiprocessor*), będący samodzielną jednostką obliczeniową, aczkolwiek dzielący część pamięci z innymi SM. Przykładowo karta grafiki NVIDIA GeForce 9800 GTX ma osiem SM. Każdy z nich dzieli się na procesory strumieniowe (SP – *Streaming Processor*),

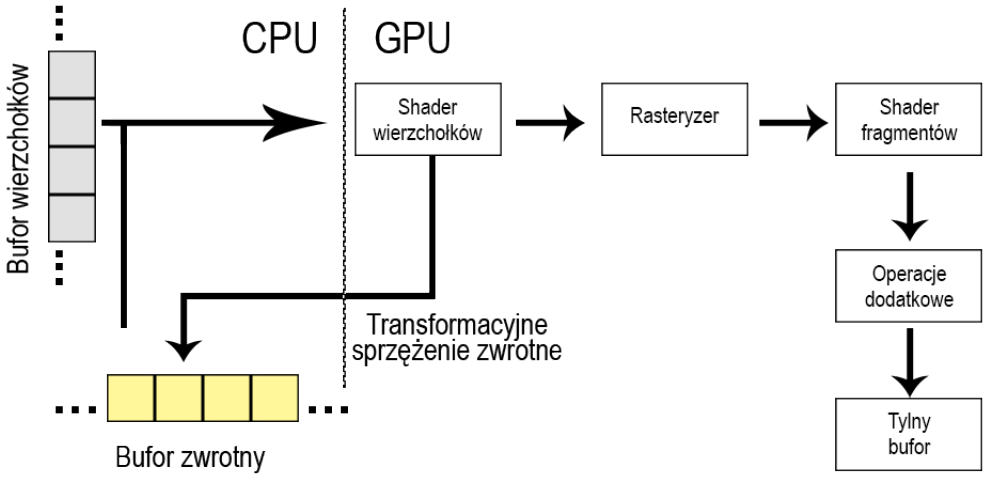
[1] https://www.google.pl/search?q=gpu+architecture&tbm=isch&biw=1920&bih=955

w danym przypadku jest ich w sumie 128 na całym GPU. SP są tak skonstruowane, by móc przetwarzać jak najwięcej operacji równolegle. W technologiach stricte GPGPU programista zależnie od potrzeb może uruchomić konkretną liczbę SP, na każdym zaprzęgając do pracy ustaloną z góry liczbę wątków, nieprzekraczającą jednak maksymalnej wartości, zależnej od modelu urządzenia [3].



Rysunek 2.8. Schemat tzw. gridu w technologii CUDA [3].

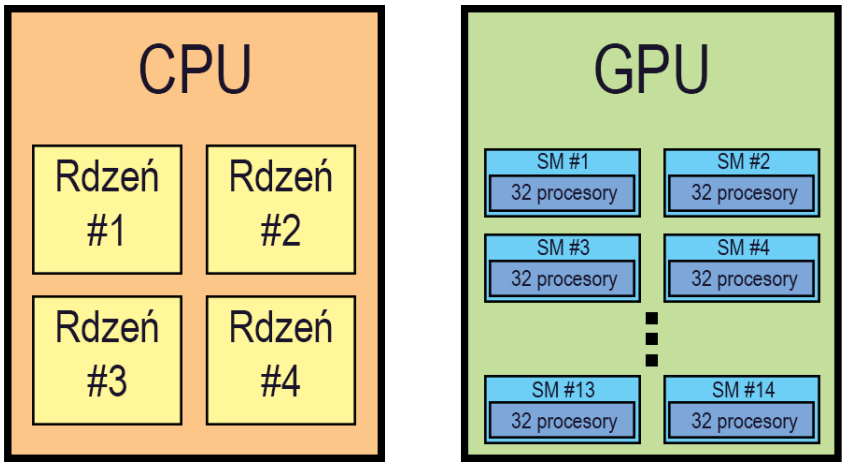
Jednakże zazwyczaj nie ma bezpośredniej kontroli nad elementami warstwy fizycznej wymienionymi wyżej. Przykładowo, CUDA zapewnia programiście warstwę logiczną podziału na jednostki przetwarzające (zwane tu blokami) i wątkami. Widać to na rysunku 2.8. W przypadku OpenCL sytuacja wygląda podobnie. Programista może oznaczyć, jak chce podzielić zadania, jednak przypisanie ich do konkretnych procesorów leży już w gestii samego API i sterownika. Na GPU stworzonym w architekturze Fermi istnieje możliwość zdefiniowania w sumie $65\,535 \times 65\,535 \times 1536$ wątków pracujących współbieżnie [3].



Rysunek 2.9. Potok renderingu i transformacyjne sprzężenie zwrotne. Źródło: opracowanie własne.

Chcąc korzystać z API graficznego do obliczeń GPGPU, programista zostaje skazany na pisanie programów w natywnym języku shaderów, najczęściej HLSL (dla API DirectX) lub GLSL (dla API OpenGL). Te wysokopoziomowe są bardzo podobne do zwykłego C, więc opanowanie ich nie powinno nastręczać problemów, jednakże wymagają wiedzy na temat przebiegu procesu renderingu. Kontrola nad przydziałem i liczbą jednostek przetwarzających także wygląda w inaczej. Jedyną możliwością jej zdefiniowania jest ustalenie danej, konkretnej liczby wierzchołków przesyłanych w buforze do GPU, które, w zależności od potrzeb, mogą być z logicznego punktu widzenia faktycznymi wierzchołkami, bądź po prostu zbiorami danych o z góry ustalonej strukturze. Rysunek 2.9 obrazuje proces przetwarzania danych zaszytych w wierzchołkach. Twórcy API graficznych od pewnego momentu starają się także umożliwić i ułatwić obliczenia GPGPU za ich pomocą, udostępniając takie funkcje, jak np. transformacyjne sprzężenie zwrotne, czy bufory teksturowe w OpenGL [7].

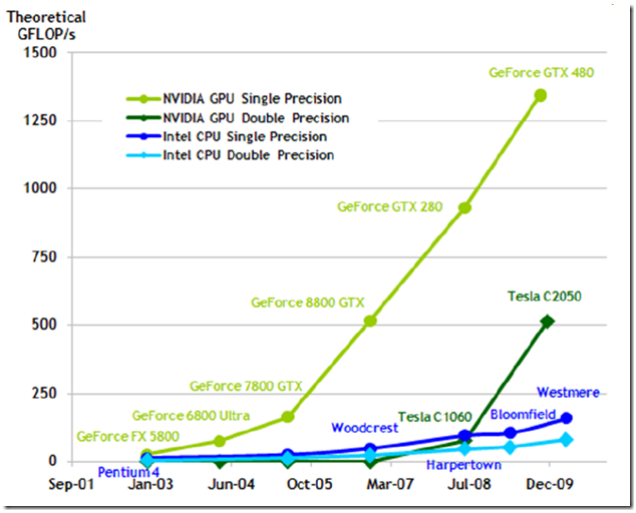
2.2.3. Porównanie GPU i CPU pod kątem architektury i zastosowań



Rysunek 2.10. Porównanie budowy CPU i GPU Źródło: opracowanie własne w oparciu o źródło zewnętrzne[2].

Rysunek 2.10 przedstawia uproszczone schematy architektury CPU i GPU, co doskonale obrazuje różnice między nimi, a także pozwala wywnioskować zastosowania. W rozdziale 2.2.2 podano, że GPU jest urządzeniem zbudowanym z wielu procesorów pracujących równolegle, natomiast CPU składa się z niewielkiej liczby jednostek obliczeniowych (w dzisiejszych czasach 4 to standardowa liczba dla zapotrzebowań osobistych), jednakże dużo szybszych. Procesory CPU są dużo bardziej zaawansowane technologiczne, wyposażone w większą liczbę zaawansowanych instrukcji oraz algorytmów przyspieszających ich działanie w specjalnych sytuacjach. Przykładem może być predykcja wyboru instrukcji warunkowych, znacznie skracająca czas obliczeń bloków `if-else` [11]. Z kolei w GPU ta technologia jest dużo uboższa. Wynika z tego, że sekwencyjny kod z dużą liczbą rozgałęzień szybciej działa na CPU, a GPU nadaje się lepiej do obliczeń równoległych, w których warunków należy unikać.

[2] http://blog.goldenhelix.com/mthiesen/video-graphics-and-genomics-a-real-game-changer/



Rysunek 2.11. Różnica rozwoju wydajności CPU i GPU na przestrzeni ostatnich lat[3].

Na rysunku 2.11 widać, że na przestrzeni ostatnich lat teoretyczna wydajność kart graficznych, liczona w gigaflopach na sekundę, wzrosła znacznie w stosunku do procesorów. Można zaobserwować kolejny rozstrzał między tymi platformami, CPU nie robi większej różnicy precyzja liczb, na których wykonywane jest przetwarzanie. Natomiast GPU zoptymalizowana jest pod kątem obliczeń na zmiennych typu `float` – pojedynczej precyzji. Prawdopodobnie po raz kolejny widać skutki najczęstszego stosowania GPU do renderingu grafiki trójwymiarowej, gdzie pojedyncza precyzja jest wystarczająca.

CPU zapewnia także dużo potężniejszy i bardziej złożony system komunikacji między pracującymi wątkami oraz większą liczbę struktur synchronizacyjnych. W przypadku GPU synchronizacja opiera się głównie o bariery pamięciowe, pozwalające wstrzymać wątki, dopóki inne nie dotrą do tego samego miejsca kodu, jednakże tylko w ograniczonym zakresie, np. na poziomie bloku, oraz o operacje atomowe [3]. Trudno uzyskać także specjalizację wątków do określonych zadań, jako, że ten sam program, zwany *kernelem*, jest uruchamiany na nich wszystkich w jednym momencie. To, nad czym programista ma kontrolę, to ich liczba oraz początkowy zbiór danych.

Największym problemem CPU jest to, że osiągnęły one szczyt swoich możliwości w kwestii szybkości taktowania zegara. Nie jest możliwe dalsze jej zwiększanie bez gwałtownego wzrostu ilości produkowanego ciepła. Wymaga to zaawansowanych rozwiązań chłodzących i dla zwykłego zjadacza chleba staje się nieosiągalne finansowo. W kontraście, procesory GPU cały czas pracują z relatywnie niskimi częstotliwościami, a ich rozwój polega w skrócie na dokładaniu kolejnych jednostek przetwarzających i zmniejszaniu zapotrzebowania na energię.

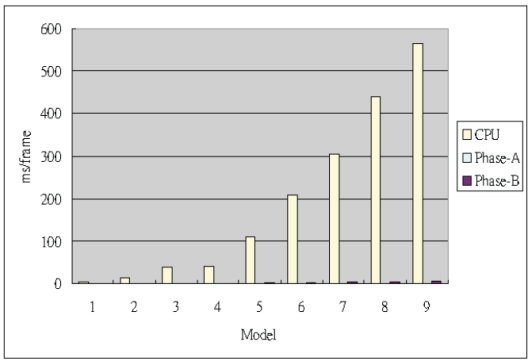
[3] https://algconsultings.wordpress.com/2011/02/28/can-we-afford-to-have-a-supercomputer

2.2.4. Zastosowanie GPU w symulacji tkanin

Siłą rzeczy narzuca się pytanie, zadane w kontekście niniejszej pracy – co lepiej nadaje się do obliczeń związanych z symulacją tkaniny? CPU czy GPU? Odpowiedź zdaje się być w miarę oczywista. Rozważany układ jest zbiorem wierzchołków, do których przyłączona jest pewna z góry określona ilość sprężyn bądź ograniczników. Każdy z nich jest oddzielnym bytem i dla każdego wykonujemy tak naprawdę te same obliczenia (tzn. ten sam ciąg instrukcji). Sensownym pomysłem w takiej sytuacji wydaje się przeprowadzenie ich równolegle.

Można podejść do tego na dwa sposoby. Pierwszym jest przyjęcie punktu masy tkaniny jako „jednostki obliczeniowej" i rozwiązanie sił bądź przesunięć dla każdej z przyłączonych do niego sprężyn. Jak jednak widać na rysunku 2.1, wierzchołki współdzielą sprężyny, tzn. wartość siły dla AB będzie taka sama jak dla BA, tylko nada się jej przeciwny zwrot. Przedstawione powyżej podejście doprowadzi do redundancji obliczeń. Drugi sposób zakłada policzenie wpierw sił dla wszystkich sprężyn, a następnie dopiero zmodyfikowanie pozycji wierzchołków na ich podstawie. Wadą tego rozwiązania jest konieczność uruchamiania dwóch różnych programów na GPU i spowolnienie programu poprzez nieuchronny powrót sterowania do CPU pomiędzy obliczeniami, ewentualnie konieczność synchronizacji dostępu do danej komórki pamięci przechowującej pozycję danego wierzchołka, jako, że jedna sprężyna wpływa na dwa punkty masy. W związku z tym w niniejszej pracy zdecydowano się na metodę pierwszą.

W przypadku chęci stworzenia implementacji symulacji na CPU, należałoby sekwencyjnie wykonywać wszystkie obliczenia na kolejnych wierzchołkach, kolejno w pętli. W najlepszym przypadku można podzielić przetwarzanie na wątki, jednak maksymalnie i tak byłoby dostępnych niewiele jednostek przetwarzających. W GPU jest ich dużo więcej, oprócz tego urządzenie z definicji przystosowano do obliczeń równoległych, co pozwala uzyskać maksymalną wydajność. Jedyną sytuacją, w której CPU mogłoby mieć przewagę, jest detekcja dużej liczby kolizji, z racji konieczności użycia tutaj instrukcji warunkowych, a jak wiemy, GPU nie najlepiej sobie z nimi radzi.



Rysunek 2.12. Porównanie wydajności implementacji CPU i GPU [4].

Autorzy [4] wykonali porównanie szybkości przetwarzania symulacji tkaniny osobno dla CPU i GPU. Widać tutaj miażdżącą przewagę karty graficznej nad procesorem w kwestii wydajności, co pozwala wnioskować o sensowności zastosowania GPU do rozwiązania omawianego problemu.

Rozdział 3

Wykorzystane technologie

3.1. Analiza możliwości urządzeń mobilnych

3.1.1. Sensowność wykorzystania urządzeń mobilnych w symulacji tkanin

W związku z faktem, iż niniejsza praca zajmuje się symulacją tkaniny na urządzeniach mobilnych, naturalnie nasuwa się pytanie o sens realizacji tego typu obliczeń z użyciem smartfonu bądź tabletu. Prawdą jest, że rozwiązanie zagadnienia związane jest ze sporym zapotrzebowaniem na moc obliczeniową, zwiększającym się proporcjonalnie do oczekiwanej dokładności rozwiązania, a konkretnie - do gęstości siatki tkaniny. Z drugiej strony, urządzenia mobilne, w przeciwieństwie do platformy PC, szczytowymi osiągnięciami technologii pod względem wydajności nigdy nie były – raczej starano się tu osiągnąć kompromis między sensowną mocą obliczeniową a niskim zużyciem energii. Jednakże smartfony i inne tego rodzaju urządzenia cechują się dostępnością dla użytkownika praktycznie zawsze, co nie do końca może być powiedziane o PC, oraz unikalnymi metodami interakcji, takimi jak ekran dotykowy, bądź akcelerometr.

Mimo oczywistych wad, także i na platformach mobilnych symulacja tkanin znajduje zastosowanie. Jako pierwszy i najważniejszy przykład należy wskazać rynek gier i wizualizacji 3D. Dawno minęły już czasy, gdy najpopularniejszą grą na telefonach komórkowych był kultowy, dwuwymiarowy „*Snake*". Obecnie spora część rynku koncentruje się na złożonych grach trójwymiarowych, z coraz ładniejszą grafiką, na potrzeby których tworzy się coraz bardziej zaawansowane silniki graficzne i korzysta z najnowszych technologii. Podobnie, jak na platformie PC, także i tutaj możliwe, a nawet pożądane jest użycie symulacji tkanin do m.in. realistycznej animacji elementów stroju bohaterów, flag powiewających na wietrze oraz innych przedmiotów tekstylnych.

Obecnie wielu producentów i sprzedawców z różnych branż decyduje się na stworzenie i wypuszczenie na rynek własnej, wyspecjalizowanej aplikacji dla urządzeń mobilnych, pozwalającej w prosty, przyjazny sposób przeglądać oferty, oglądać towary i dokonywać zakupów.

Zdecydowanie zwiększa to przychody danej firmy. Branżą, która mogłaby skorzystać na zastosowaniu symulacji tkanin w swoich aplikacjach jest oczywiście branża włókiennicza i odzieżowa. Przykładem może być chociażby stworzenie „wirtualnej przymierzalni" [5], przy pomocy której klient byłby w stanie „ubrać się" w każdy wybrany element odzieży. Aplikacja pozwoliłaby mu chociażby na obejrzenie go ze wszystkich stron, sprawdzenie elastyczności i zachowania w różnych pozach. A to wszystko na ekranie tabletu, dostępne w każdym możliwym miejscu.

Oczywistym jest, że niższa wydajność urządzeń mobilnych wiąże się z niższą jakością symulacji. Warto jednak pamiętać, iż wyświetlacze urządzeń mobilnych z reguły są mniejsze od monitorów komputerowych. Co za tym idzie – można zastosować siatkę tkaniny o mniejszej gęstości i wykonywać mniej dokładne obliczenia np. detekcji kolizji bez dużego spadku jakości wizualnej. Ten fakt, oraz omówiona wcześniej mnogość zastosowań sprawiają, że symulacja tkanin na urządzeniach mobilnych zdaje się jak najbardziej mieć sens. W rozdziałach 6 i 7 zostanie wykazane, w jakim stopniu.

3.1.2. Konfiguracja sprzętowa urządzeń mobilnych i porównanie z konfiguracją PC

Jak już wyżej wspomniano, wydajność konfiguracji sprzętowej urządzeń mobilnych to drobny ułamek wydajności komputerów klasy PC. Warto dokładniej zwrócić uwagę, jaka jest między tymi platformami różnica i z jakie ograniczenia można napotkać, tworząc symulację tkanin na tej pierwszej. Porównanie zostanie dokonane na przykładzie urządzenia testowego – smartfona LG Nexus 4 E960. Dane techniczne zaczerpnięto z [16], [17], [18] i [19].

Urządzenie oparto o mikrokontroler APQ8064 Snapdragon S4 Pro. Serce układu to czterordzeniowy procesor Krait o taktowaniu 1.5 GHz i architekturze ARMv7-A. Szybkość zegara jest przeszło dwa razy mniejsza niż w przeciętnym odpowiedniku PC. Można stąd wnioskować, że wydajność okaże się dwukrotnie mniejsza, jednak diabeł tkwi w szczegółach. Dzisiejsze procesory architektury x86 dysponują szerokim wachlarzem specjalnych instrukcji, takich jak SSE czy AVX, bardzo przyspieszających operacje wektorowe, typu SIMD. Jedynym ich odpowiednikiem w omawianym układzie są instrukcje NEON, dużo mniej wydajne. A zatem, zgodnie z [15], procesor Krait cechuje ponad dziesięciokrotnie mniejsza wydajność niż jego przykładowego odpowiednika z komputera klasy PC – Intel Core i7-4770.

Układ Snapdragon jest także wyposażony w dedykowane GPU, specjalnie na potrzeby renderingu grafiki 2D i 3D. To Adreno 320, cechujące się taktowaniem zegara 400 MHz i w sumie 64 procesorami strumieniowymi. Karta graficzna osiąga wydajność ok. 60 GFLOPS[1]. Przykładem porównawczym może być średniej klasy GPU komputerów stacjonarnych sprzed

[1] *Floating-point Operations Per Second* – liczba operacji na liczbach zmiennoprzecinkowych w czasie 1 sekundy.

kilku lat, GeForce GTX 750. Jego zegar to 1020 MHz, ma ono 512 SP, a wydajnościowo plasuje się trochę ponad 1 TFLOPS. Jedno z najpotężniejszych GPU obecnie, GeForce GTX Titan, cechuje z kolei 3072 procesorów strumieniowych i ok. 6 TFLOPS. Pod uwagę wzięto oczywiście obliczenia na liczbach zmiennoprzecinkowych pojedynczej precyzji. Widać więc, że mobilne GPU wydajnościowo stanowią zaledwie ułamek ich „pełnowymiarowych" odpowiedników.

Ważną kwestią jest też dostępność i obsługa odpowiednich technologii, a w szczególności API graficznych i GPGPU. Tutaj na szczęście sytuacja ma się dużo lepiej. Adreno 320 wspiera zaawansowane już całkiem OpenGL ES 3.0 [17]. Patrząc na sprawę pod kątem symulacji tkanin, brakuje tylko obsługi OpenGL ES 3.2, który wprowadził bardzo przydatne w omawianym problemie bufory teksturowe. Mimo to jednak API graficzne spełnia wymagania niniejszej pracy. Oczywiście z ogólnego punktu widzenia, brakuje wielu nowych rozwiązań, wprowadzonych w najnowszych „dużych" wersjach OpenGL. Do obliczeń GPGPU wspierane są technologie OpenCL 1.1, będący open-source'owym odpowiednikiem CUDA, oraz RenderScript. W tej pracy jednak okazało się niemożliwe użycie żadnego z nich. Obsługa OpenCL została wycofana na urządzeniach Google, w tym na Nexusie 4, z powodów marketingowych, a konkretnie, z powodu promowania drugiej wymienionej technologii [2]. Tej z kolei nie można było użyć z racji tego, iż API nie pozwala jawnie wybrać i ustalić, czy obliczenia będą dokonywane na GPU, czy na CPU [3].

3.1.3. Unikalne możliwości platform mobilnych w interakcji użytkownika z tkaniną

Urządzenia mobilne cechuje jedna ważna przewaga nad komputerami PC – charakterystyczny tylko dla nich interfejs. Sztandarowym przykładem jest oczywiście ekran dotykowy. W rozważanej kwestii, z jego pomocą użytkownik może ruchami palca po ekranie przemieszczać i rozciągać tkaninę. Pozwala to na nadanie jej ruchu w odpowiednią stronę, sprawdzanie elastyczności oraz dokładne, precyzyjne ustawienie w wirtualnym świecie poprzez sprowokowanie kolizji z obiektami otoczenia. Przydatne może być szczególnie w omawianym wcześniej przypadku wykorzystania symulacji do aplikacji typu „przymierzalnia". Klient jest w stanie realistycznie założyć oglądany element ubioru na swój awatar i dokładnie go dopasować.

Kolejnym charakterystycznym dla platformy mobilnej urządzeniem wejścia jest akcelerometr. Najważniejszy przykład jego wykorzystania to możliwość zmiany kierunku siły

[2] http://www.anandtech.com/show/7191/android-43-update-for-nexus-10-and-4-removes-unofficial-opencl-drivers

[3] http://stackoverflow.com/questions/18753935/forcing-renderscript-to-run-on-cpu-or-gpu-atleast-for-performance-tuning-purpos

grawitacji działającej na symulowany układ, poprzez obracanie telefonu w odpowiednim kierunku. Użytkownik będzie miał wrażenie obracania układem, a siła grawitacji pozostanie stała i skierowana w dół względem niego. Stwarza to kolejne możliwości zmiany położenia tkaniny, dokładnego jej układania i odwzorowania realizmu świata wirtualnego.

3.2. Wybrane technologie i narzędzia

3.2.1. Android NDK

Wiodącym językiem programowania na platformie Android jest język Java. Wykorzystujące go Android SDK posiada bardzo szeroką dokumentację i pomoc techniczną. Można na jego temat znaleźć wiele publikacji, a w przypadku, gdy początkujący programista ma z nim jakieś trudności, w Internecie na pewno znajdzie rozwiązanie. Istnieje jednak jeden duży problem – Java nie jest językiem natywnym i uruchamia się na maszynie wirtualnej. Przez to nie nadaje się do aplikacji, w których kluczowa jest wydajność, takich jak na przykład symulacja fizyczna tkanin.

Jest to powód, dla którego cały projekt został napisany w języku C++. Umożliwiło to wykorzystanie Android NDK[4], czyli zbioru większości funkcji Androida, dostępnych z poziomu C bądź C++. Kod ten zostaje połączony z inicjalizacyjnym kodem Javy przy pomocy technologii JNI[5], będącej swoistym pomostem pomiędzy dwoma platformami. Cykl życia aplikacji kontrolowany z poziomu C++ jest niemalże identyczny, jak w przypadku języka Java, opiera się na zdarzeniach wywoływanych przez system operacyjny. W ten sposób zarządza się m.in. inicjalizacją i zwalnianiem pamięci oraz sygnałami przychodzącymi z urządzeń wejściowych, takich jak ekran dotykowy czy akcelerometr. Za pomocą plików konfiguracyjnych i tzw. *Android Manifest* kontrolowane są wszelkie opcje konfiguracyjne dotyczące aplikacji. Można tam ustalić np. jej nazwę, widoczność elementów interfejsu systemu oraz to, czy aplikacja sama potrafi obsłużyć zmianę orientacji ekranu, czy należy ją wtedy zresetować.

Jedna z implementacji symulacji zakłada przetwarzanie jej wielowątkowo na CPU. Jako, że Android wywodzi się z rodziny UNIX-ów, istnieje możliwość wywołania funkcji biblioteki *pthread* z poziomu natywnego kodu C++. Umożliwia ona łatwe i przejrzyste uruchamianie nowych wątków, zarządzanie nimi oraz ich synchronizację. W wersji na platformę Windows skorzystano z darmowej implementacji biblioteki *pthreads-win32*[6].

[4] Native Development Kit, http://developer.android.com/tools/sdk/ndk/index.html
[5] Java Native Interface, http://docs.oracle.com/javase/7/docs/technotes/guides/jni/
[6] https://www.sourceware.org/pthreads-win32/

3.2.2. OpenGL

Stworzenie samej symulacji tkanin nie ma sensu, jeśli nie jest się w stanie pokazać jej działania. Konieczne więc było stworzenie silnika wizualizującego wygląd i zachowanie tkaniny. OpenGL to darmowe API graficzne, czyli zestaw bibliotek służących do komunikacji z GPU i w efekcie – rysowania grafiki 2D i 3D. Najważniejsza cecha tej technologii to jej dostępność na praktycznie wszystkich platformach. Właśnie z tego powodu została ona użyta w niniejszej pracy, jako że jest obsługiwana zarówno na smartfonach z systemem Android, jak i na komputerach PC z Windows. W pierwszym przypadku wykorzystano wersję OpenGL ES 3.0, będącą okrojoną i przystosowaną do użytku na platformach mobilnych oraz innych systemach wbudowanych. Na PC użyto standardowego OpenGL 3.3. Nie są to najnowsze edycje OpenGL, jednak oferują już programowalny potok renderingu i systemy wspierające GPGPU, takie jak transformacyjne sprzężenie zwrotne. Uznano je więc za wystarczające do zrealizowania niniejszej pracy.

Jak można przeczytać w [6], OpenGL wymaga także API pomocniczego, zarządzającego tworzeniem okien, do których rysowana jest grafika, przydzielaniem kontekstów graficznych, wymaganych by cokolwiek dało się wyrenderować, oraz innymi zasobami. W przypadku programu na platformie Android, użyte zostało tu EGL – specjalne API do systemów mobilnych i wbudowanych, dzieła twórców OpenGL, dostępne razem z OpenGL ES. W wersji „pecetowej" wykorzystano darmowe API GLFW[7] oraz GLEW[8].

Ważną kwestią w tworzeniu zarówno symulacji fizycznych, jak i samego silnika graficznego jest odpowiednia baza matematyczna. W niniejszej pracy użyta została darmowa biblioteka GLM[9], zalecana jako nieodłączny element programowania w OpenGL. Zapewnia ona dostęp do wielu wygodnych i przydatnych funkcji oraz struktur matematycznych, ułatwiających szczególnie obliczenia na wektorach oraz macierzach, kluczowe w grafice 3D. GLM w swoim założeniu ma jak najbardziej przypominać składnię i semantykę związanego z OpenGL języka programów cieniujących – GLSL, o którym więcej w podrozdziale 3.2.2.2.

Do wczytywania i przetwarzania tekstur wykorzystano także darmową bibliotekę SOIL2[10]. Tekstury są w programie niezbędne, z racji tego, iż jedno z założeń to wyświetlanie elementów interfejsu użytkownika – przycisków i tekstu. SOIL2 pozwala na łatwe i szybkie załadowanie tekstury w jednym z wielu obsługiwanych formatów, a następnie utworzenie odpowiedniego obiektu OpenGL. Biblioteka jest częścią domeny publicznej i została dołączona do kodu źródłowego aplikacji, z powodu potrzeby skompilowania jej dla architektury ARM.

[7] http://www.glfw.org/
[8] http://glew.sourceforge.net/
[9] http://glm.g-truc.net/0.9.7/index.html
[10] https://bitbucket.org/SpartanJ/soil2

3.2.2.1. OpenGL ES 3.0 kontra OpenGL 3.3

Według [6], każda wersja OpenGL ES to podzbiór funkcji pewnej „dużej" wersji OpenGL. W przypadku użytego tutaj Nexusa 4, obsługiwana jest edycja ES 3.0, a jej odpowiednikiem na większych platformach – wersja 3.3. Właśnie dlatego ta ostatnia posłużyła do stworzenia „pecetowego" modelu symulacji tkanin. Dzięki temu można było zapewnić jak największą kompatybilność i łatwość przeniesienia z jednej platformy na drugą.

Filozofią przyświecającą deweloperom przy tworzeniu edycji OpenGL ES jest przede wszystkim to, by każda czynność, którą można zrobić przy pomocy API, była osiągalna tylko w jeden, konkretny sposób. Idąc tym tropem, ujednolicono wiele funkcji i usunięto te, które się dublowały, bądź w dalszej perspektywie dawały identyczne efekty. Dzięki temu API stało się bardziej przejrzyste oraz łatwiej się z niego korzysta. Warto zaznaczyć, że użyteczność nie zmniejszyła się względem wersji API 3.3, a po prostu „zrobiono porządek".

3.2.2.2. GLSL

GLSL jest językiem specjalistycznym dla API OpenGL, przy pomocy którego pisane są programy cieniujące, czyli tzw. *shadery*. W niniejszej pracy wykorzystany został do stworzenia podstawowego, prostego modelu cieniowania opartego o wzory Phonga-Blinna. Z racji opisanego w podrozdziale 3.1.2 braku innych sensownych technologii na wybranej platformie sprzętowej, użyto go także do napisania obliczeń symulacji tkanin na GPU, tj. wyliczenia przesunięć wierzchołków, rozwiązania kolizji zewnętrznych i wewnętrznych oraz przeliczenia wektorów normalnych.

Niewątpliwą zaletą GLSL jest jego integracja z API graficznym i brak konieczności dołączania dodatkowych bibliotek. Chcąc prowadzić specjalistyczne obliczenia dla tkaniny, można korzystać z tych samych buforów wierzchołków, których używa się do jej rysowania. Bardzo łatwo uniknąć jest niepotrzebnego kopiowania danych. Co za tym idzie – GLSL jest optymalnym pod względem wydajności rozwiązaniem. Wart wspomnienia jest także fakt jego dostępności, można z niego skorzystać wszędzie tam, gdzie możliwe jest uruchomienie biblioteki OpenGL w wersji obsługującej programowalny potok renderingu.

GLSL ma jednak kilka wad, które dają o sobie znać w momencie, gdy zachodzi potrzeba wykorzystania go do obliczeń GPGPU. Obojętnie, jaki problem trzeba rozwiązać, trzeba dostosować dane oraz algorytmy do struktur danych potoku renderingu i do funkcji API graficznego. W rozważanym przypadku nie jest to jednak duży problem. Gorszą bolączką jest tak naprawdę niewielkie wsparcie dla GPGPU w OpenGL ES 3.0, omówione zostanie ono w następnym podrozdziale. Warto wspomnieć, że najnowsze wersje API mają już pełną obsługę obliczeń GPGPU, czego przejawem jest chociażby obecność *Compute Shaderów*, czyli programów ogólnego przeznaczenia uruchamianych na GPU.

3.2.2.3. Bufory teksturowe i bufory jednorodne

Dużym minusem urządzenia testowego jest brak obsługi OpenGL ES 3.2, w którym dodane zostały bufory teksturowe (ang. *texture buffers*). Zostały one wykorzystane do zaimplementowania prostej symulacji tkanin w [7], co dowodzi ich kluczowej przydatności. Wykonując obliczenia na każdym wierzchołku, należy mieć dostęp do pozycji wierzchołków sąsiednich, aby obliczyć odległości między nimi a tym aktualnym. Prowadzi to do konieczności posiadania dostępu do bufora pozycji obecnych oraz poprzednich wszystkich wierzchołków przez każdy uruchomiony na GPU kernel.

Można tego dokonać idealnie wykorzystując bufory teksturowe, będące *de facto* jednowymiarową teksturą, której zbiór danych da się ustalić jako jakikolwiek istniejący na karcie graficznej bufor. Jeden bufor teksturowy może pomieścić co najmniej 128 MiB, zależnie od pamięci konkretnego GPU [20]. Zapewnia też bardzo szybki swobodny odczyt danych [21].

Zamiast tego konieczne było użyć innej techniki, a mianowicie tzw. *Uniform Buffer Objects*, co można przetłumaczyć jako bufory jednorodne. Wykorzystywane są one do łączenia danych trafiających do programów cieniujących w jeden ciągły blok i dają możliwość szybkiego wykorzystania tego bloku w wielu programach [20]. Ich zastosowanie jest, jak widać, trochę inne niż to, o które w rozwiązaniu omawianego problemu chodzi, jednak w efekcie spełniają podobną rolę, jak bufory teksturowe. Należy się jednak liczyć z faktem, że maksymalna ich wielkość to tylko 64 KB danych, co ogranicza maksymalną możliwą gęstość siatki tkaniny. Odczyt jest tu także dużo wolniejszy, niż w pierwszym przypadku [21].

3.2.2.4. Transformacyjne sprzężenie zwrotne

O mechanizmie OpenGL zwanym *Transform Feedback*, czyli transformacyjnym sprzężeniu zwrotnym, wspomniano już w rozdziale 2.2.2. Znajdujący się tam rysunek 2.9 przedstawia m.in. ogólną zasadę działania tej technologii. Jak wiadomo, dane w potoku renderingu koniec końców trafiają do bufora ramki, bądź tylnego bufora i są po drodze przetwarzane w kilku różnych etapach. Aby w ogóle móc dokonywać obliczeń GPGPU korzystając z API graficznego, trzeba mieć możliwość odczytu danych wyjściowych. Następnie albo zostaje uznane, że są to pożądane wyniki obliczeń, albo podaje się je znowu na wejście potoku do ponownego przetworzenia.

Taką możliwość udostępnia właśnie transformacyjne sprzężenie zwrotne. Dane trafiają do shadera wierzchołków, gdzie program cieniujący wykonuje na nich ciąg operacji. Następnie zamiast zostać podane do rasteryzera, zapisywane są do ustalonego wcześniej, specjalnego bufora, oznaczonego jako bufor sprzężenia zwrotnego. Po zakończeniu działania *Transform Feedbacku* jest on dostępny tak samo, jak każdy inny bufor w OpenGL. Opisywana symulacja tkanin to idealny przykład zastosowania tego rozwiązania, jako, że takie same obliczenia

wykonywane są w każdym kroku, a następnie po prostu zamienia się miejscami bufory wejściowe i wyjściowe.

3.2.3. Visual Studio 2015 Community + Cross-platform Development Kit

W pracy, jako środowisko programistyczne, wykorzystano najnowszą edycję Microsoft Visual Studio w darmowej wersji Community. Wybór został dokonany ze względu na jego wszechstronność i możliwości. Nowością w wydaniu 2015 jest pakiet Cross-platform Development Kit. Umożliwia on budowanie programów na platformy mobilne, takie jak Android lub iOS. Dotychczas było to wspierane tylko w Eclipse, bądź Android Studio, będącym jego specjalistyczną edycją dla programowania na tę platformę. Nowy pakiet Microsoftu pozwala na tworzenie aplikacji w natywnym języku C++, z użyciem opisanych w podrozdziale 3.2.1 bibliotek. Zapewnia także integrację z systemem operacyjnym, co jest dużym ułatwieniem dla początkujących programistów, którzy nie muszą się zajmować zawiłościami inicjalizacji natywnego kodu.

Wiąże się z tym jednak poważne ograniczenie – oznacza to bowiem, że nie ma dostępu do części aplikacji w Javie, tworzonej tu automatycznie. Język ten jest w ogóle nieobsługiwany przez Visual Studio, ale teoretycznie wszystkie potrzebne zasoby, np. referencję do menedżera assetów[11], dostaje się bezpośrednio przy starcie programu. Jeżeli jakieś rozwiązanie wymaga bardziej zaawansowanej komunikacji między Javą a C++ – nie można z niego skorzystać.

Microsoft pomyślał też i o tym, wprowadzając pakiet Xamarin. Daje on możliwość pisania „wysokopoziomowej" części kodu przeznaczonego na platformy mobilne w C#. Jednakże jest to już opcja płatna, a edycja darmowa wprowadza duże ograniczenia odnośnie budowanych aplikacji, ograniczając chociażby rozmiar pliku wykonywalnego do 128 kiB i uniemożliwiając łączenie go z bibliotekami tworzonymi w innych językach, jak C/C++ lub Java.

[11] Zasobów dodatkowych, potrzebnych do działania aplikacji, np. tekstur, dźwięków, siatek geometrycznych, itp.

Rozdział 4

Budowa i działanie aplikacji

4.1. Cele oraz możliwości aplikacji

Na potrzeby niniejszej pracy została stworzona aplikacja prezentacyjna. Realizuje ona kilka kluczowych celów. Po pierwsze, ma za zadanie zaprezentować działanie dwóch omówionych w rozdziale 2 modeli symulacji tkanin. Musi pozwalać na porównanie ich pod względem wydajności, stabilności i efektu wizualnego. Wydajność rozumie się jako czas potrzebny na obliczenie jednego kroku symulacji, im mniejszy, tym oczywiście lepiej. Aplikacja informuje o nim użytkownika, wyświetlając stosowną informację w formie tekstowej. Jeśli chodzi o dwa następne czynniki, najlepiej ocenić zachowanie tkaniny wizualnie. W tym celu program rysuje ją w przestrzeni 3D.

Kluczową kwestią jest tutaj interakcja z innymi obiektami. Aplikacja tworzy wirtualną scenę i umieszcza w niej pewną liczbę podstawowych kształtów geometrycznych, takich jak płaszczyzna, prostopadłościan, bądź sfera. Dzięki temu można sprawdzić, jak zachowa się tkanina, wchodząc w kolizje z tymi elementami. Istnieje także opcja przemieszczania wybranego obiektu po scenie, co pozwala na tworzenie różnych konfiguracji zderzeń pomiędzy nim a przedmiotem symulacji. Warto wspomnieć, że inne elementy sceny także kolidują ze sobą.

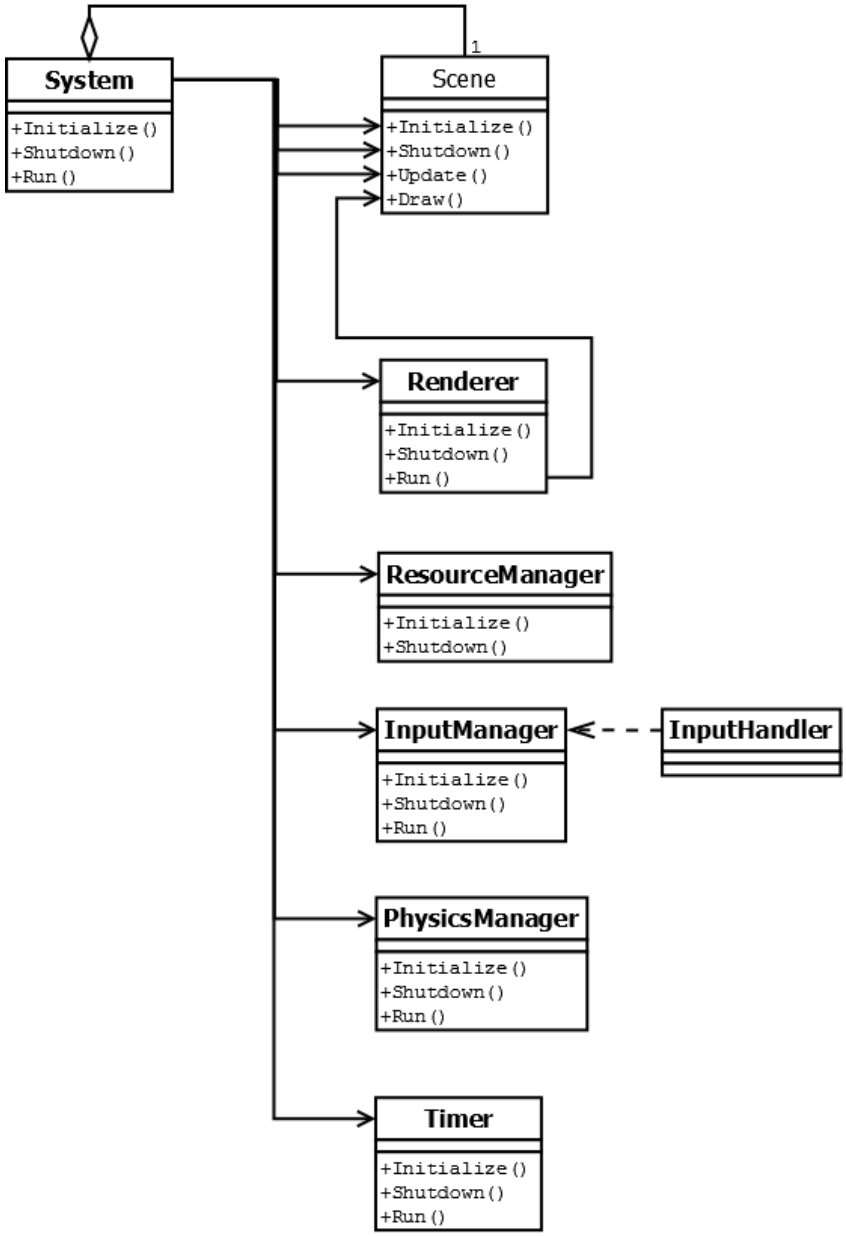
Celem była także możliwość porównania prędkości obliczeń symulacji modeli tkanin na CPU i GPU oraz zbadanie różnicy wydajności GPU urządzenia mobilnego i GPU komputera PC. Aplikacja umożliwia określenie, który z wymienionych wyżej komponentów sprzętowych będzie przetwarzać symulację. Występuje ona także w wersjach na obie rozpatrywane platformy, pozwalając na dokonanie wszelkich niezbędnych porównań.

Ważną kwestią w niniejszych rozważaniach są możliwości interakcji z tkaniną, jakie udostępnia smartfon. Program umożliwia ją, pozwalając przemieszczać fragment symulowanego obiektu poprzez przesuwanie palca po ekranie dotykowym.

Oprócz tego aplikacja dysponuje sporą liczbą ułatwień i udogodnień dla użytkownika. Pozwala na sterowanie kamerą, także przy pomocy ekranu dotykowego, przesuwanie jej po

płaszczyźnie XZ, obrót w dowolnym kierunku, przybliżanie i oddalanie. Kluczowe informacje wyświetlane są w formie tekstu: wspomniany wcześniej czas trwania obliczeń jednego kroku symulacji, liczba klatek na sekundę, czy też aktualnie przetwarzany model tkaniny. Zmienić można wiele różnych parametrów, opisanych szerzej w rozdziale 5 oraz tryb rysowania obiektów, jeżeli użytkownik chciałby zwrócić baczniejszą uwagę na zachowanie siatki – tzw. tryb *wireframe*, polegający na renderowaniu tylko krawędzi.

4.2. Ogólna architektura aplikacji



Rysunek 4.1. Najważniejsze elementy aplikacji i wzajemne powiązania. Źródło: opracowanie własne.

Na rysunku 4.1 przedstawione zostały główne komponenty silnika oraz zależności pomiędzy nimi. Klasy, których nazwy napisano pogrubioną czcionką są singletonami. Strzałka z linią ciągłą oznacza, że klasa A wywołuje funkcje klasy B i działanie B zależy od A. Jeżeli owa strzałka biegnie do nazwy klasy, znaczy to, iż wszystkie jej metody są wywoływane, a jeśli do konkretnej nazwy funkcji – tylko ona.

Większość singletonów, ale też i ważniejszych klas programu, została napisana zgodnie z prostą architekturą *Initialize – Run – Shutdown*. W przypadku sceny, encji, komponentów (klasy `Component`) i ich pochodnych, funkcja `Run` została rozbita na oddzielne `Update` i `Draw`. Wywoływane są one w głównej pętli programu. Jak łatwo się domyślić, `Initialize` i `Shutdown` uruchamia się odpowiednio przy starcie i wyłączaniu aplikacji. Takie podejście zapewnia dużą przejrzystość w strukturze wywołań funkcji silnika.

Podstawowym elementem spajającym działanie całego silnika jest klasa `System`. Odpowiada ona za inicjalizację wszystkich singletonów – menedżerów, ich aktualizację w głównej pętli programu, zwalnianie pamięci przy wyłączaniu aplikacji oraz za obsługę zdarzeń przychodzących z systemu Android. W jej gestii leży uśpienie i wznowienie programu, gdy takie żądanie zostanie wywołane. Przechowuje także aktualnie wczytaną scenę (klasa `Scene`) i w każdej klatce wywołuje jej metodę `Update`, odświeżając stan wszystkich encji. Także tutaj znajduje się referencja do struktury `Engine`. Ta struktura jest utrzymywana głównie na potrzeby komunikacji z Androidem i dzięki niej istnieje dostęp do wszystkich danych, jakie aplikacja dostaje od systemu. Wśród nich są m.in. wskaźnik do androidowej struktury `android_app`, gdzie przechowywane są te informacje, rozmiary ekranu oraz identyfikatory kontekstu graficznego, powierzchni rysowania i wyświetlacza, niezbędne bibliotece EGL w inicjalizacji OpenGL.

Drugim najważniejszym singletonem systemu jest klasa `Renderer`. Razem z klasami pochodnymi `Mesh` skupia w sobie wszystkie funkcje dotyczące renderingu grafiki 3D. Do jego odpowiedzialności należy inicjalizacja bibliotek EGL oraz OpenGL, odpowiedni wybór parametrów okna, utworzenie powierzchni rysowania oraz kontekstu. Następnym krokiem jest załadowanie wszystkich potrzebnych shaderów (plików z rozszerzeniem *.glsl*) i ustawienie wybranych parametrów OpenGL. Do tych ostatnich zalicza się m.in. wybór koloru, jakim czyszczony jest bufor ramki, włączenie testu głębokości, odcinania tylnych ścianek wielokątów, uruchomienie i ustawienie funkcji mieszania przezroczystości. Oczywiście przy wywołaniu metody `Shutdown` usuwane są wszelkie dane związane z renderingiem oraz niszczone wymienione wyżej elementy. Jego funkcja `Run` zawiera przede wszystkim obsługę zmiany rozmiaru ekranu, a co za tym idzie, parametrów okna i powierzchni rysowania, obsługę przełączania trybu wyświetlania obiektów, a wreszcie – renderingu elementów sceny oraz interfejsu użytkownika poprzez wywołanie metod `Draw` i `DrawGUI` obiektu typu `Scene`. Założeniem projektowym dla tej klasy była enkapsulacja większości wywołań funkcji OpenGL, tak, by kod renderingu dało się łatwo wymienić na inny. W związku z tym, klasa `Renderer` posiada także specjalistyczne funkcje do wczytywania shaderów, kerneli i tekstur oraz zwalniania pamięci po tych zasobach graficznych. Warto wspomnieć o tym, że z projektowego punktu widzenia, w aplikacji rozróżnia się pomiędzy shaderem (używanym do rysowania obiektów) a kernelem (używanym do obliczeń GPGPU, w transformacyjnym

sprzężeniu zwrotnym), chociaż z punktu widzenia ich wczytywania, są *de facto* tym samym – kodem GLSL, który przekształcony zostaje w tzw. *program* OpenGL.

Kluczowym dla działania symulacji komponentem jest klasa `Timer`. Jak sama nazwa wskazuje, zajmuje się ona wszystkimi czynnościami dotyczącymi zliczania czasu. Do pobrania aktualnego t użyto funkcji `clock_gettime`, zawartej w bibliotece *time.h*. Oferuje ona dokładność co do nanosekund, jednak na potrzeby systemu symulacyjnego wszelkie wielkości czasowe są przechowywane w formacie milisekund – jest to wystarczająca precyzja. `Timer` udostępnia następujące dane: czas całkowity, który upłynął od startu programu, czas, jaki mija pomiędzy kolejnymi krokami głównej pętli – δt, czyli tzw. *delta time*, liczba klatek na sekundę (FPS – *Frames Per Second*), będąca odwrotnością δt, liczba kroków głównej pętli programu od początku jego działania oraz tzw. *fixed delta time*, czyli stała, uśredniona wartość czasu pomiędzy krokami pętli, obliczona na podstawie ich pierwszych dziesięciu. Klasa `Timer` posiada też funkcję zapisywania stempli czasowych, umożliwiając łatwe odmierzanie czasu pomiędzy pewnymi wydarzeniami.

`ResourceManager` jest odpowiedzialny za zarządzanie zasobami symulacji. W tym przypadku ich rolę pełnią tylko tekstury, shadery i kernele. Kluczową kwestią tutaj jest działanie funkcji rodziny `Load`: `LoadTexture`, `LoadShader`, `LoadKernel`. Zasoby są trzymane w przeznaczonych do tego kolekcjach. Podczas wywołania tej funkcji sprawdzona zostaje najpierw odpowiednia kolekcja na obecność żądanego zasobu. Jeżeli takowy istnieje, jest od razu zwracany. Jeśli go nie ma, dopiero wtedy rozpoczyna się proces załadowania go z pliku. Dzięki takiemu podejściu w żadnym miejscu kodu nie trzeba martwić się o to, czy wczytano zasób, czy jeszcze nie.

Do obowiązków klasy `PhysicsManager` należy tak naprawdę tylko rozwiązywanie kolizji oraz przechowywanie wszelkich danych z tym związanych, tj. klas kolizyjnych, tzw. *colliderów*, pochodnych klasy `Collider`. Omawiany singleton zawiera także aktualny wektor grawitacji. W celach optymalizacyjnych, kolizje nie są sprawdzane na zasadzie „każdy obiekt z każdym", ale tylko dla tych encji, które w danej klatce zmieniły swoje położenie. Wyjątkiem jest sama tkanina – dla niej kolizje rozwiązywane są w każdym kroku symulacji i poza klasą `PhysicsManager`. Z racji konieczności wykonywania tych obliczeń na GPU było to najwygodniejszym podejściem.

Ostatnimi omawianymi singletonami są `InputManager` oraz `InputHandler`, który go opakowuje. W ich gestii leży obsługa zdarzeń pochodzących z urządzeń wejściowych, takich jak ekran dotykowy na platformie mobilnej. W przypadku „pecetowej" wersji programu, nie ma tu mowy o jakichkolwiek zdarzeniach, a zapewniony jest po prostu interfejs do odpytywania systemu operacyjnego o wciśnięcie konkretnych klawiszy na konkretnych urządzeniach. W kodzie samej logiki wirtualnego świata korzysta się jednak z funkcji klasy

`InputHandler`, zamieniającej „surowe” dane o stanie przycisków bądź ekranu na informacje o możliwości wykonania akcji przez program.

Dla jasności poniżej umieszczone zostały algorytmy 1, 2 i 3 dotyczące uproszczonego działania całego programu:

Algorytm 1: Inicjalizacja silnika symulacji.

Inicjalizuj połączenie z systemem Android.
    Pobierz struktury danych z Androida.
    Ustaw funkcje obsługujących zdarzenia systemu.
    Uruchom kolejkę zdarzeń.
    Czekaj na sygnał od systemu, mówiący, że można inicjalizować resztę programu.
Inicjalizuj Renderer.
    Inicjalizuj EGL i OpenGL.
    Wczytaj shadery.
    Ustaw zmienne OpenGL.
Inicjalizuj Menedżer zasobów.
    Wczytaj początkowo wymagane zasoby.
Inicjalizuj Menedżer interfejsu.
Inicjalizuj Menedżer fizyki.
Inicjalizuj Timer.
    Pobierz od systemu operacyjnego czas startu aplikacji.
Inicjalizuj scenę.
    Utwórz obiekty sceny i ich komponenty.
    Utwórz obiekt tkaniny i komponent symulatora tkaniny.
    Utwórz kamerę.
    Utwórz światła.
    Utwórz interfejs użytkownika.
    Utwórz obiekt z komponentem zarządzającym interfejsem.

Algorytm 2: Praca silnika symulacji.

```
while true do
    Wyciągnij i obsłuż wszystkie zdarzenia z kolejki.
    if m_running then
        Aktualizuj timer.
        Aktualizuj dane z urządzeń wejściowych.
        Aktualizuj encje sceny.
            Rozwiąż ewentualne kolizje między obiektami.
            Oblicz jeden krok symulacji tkaniny.
        Narysuj jedną klatkę wizualizacji symulacji.
    end
end
```

Algorytm 3: Uśpienie i wyłączenie silnika symulacji.

Uśpienie:

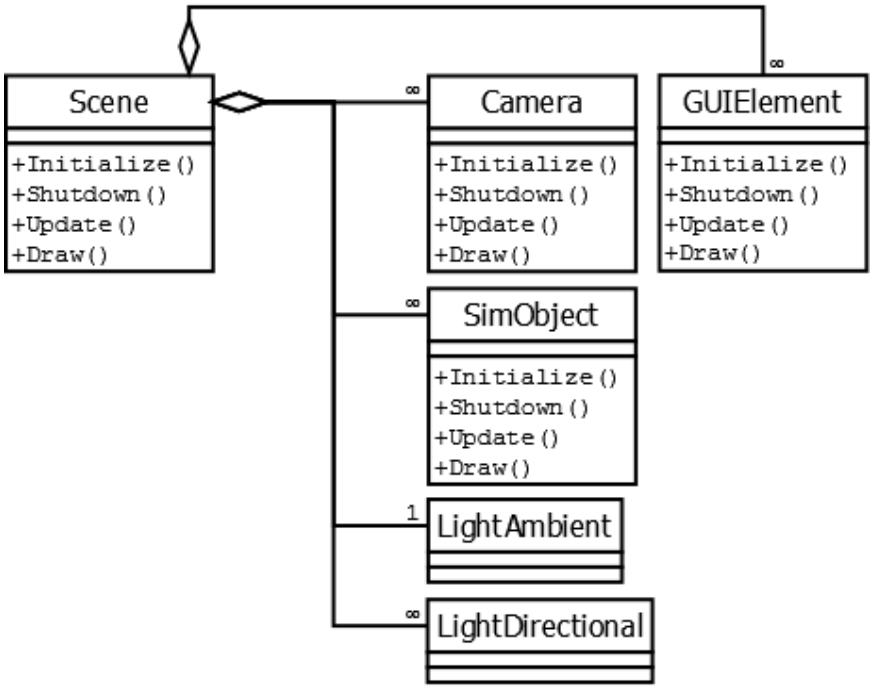
- Zamknij encję sceny i ją samą.
- Zamknij wszystkich menedżerów.
- Ustaw zmienną logiczną Systemu *m_running* na *false*.

Zamknięcie:

- Zamknij encję sceny i ją samą.
- Zamknij wszystkie menedżery.
- Wyślij do systemu informację o tym, że następuje ostateczne zamknięcie.

4.3. Budowa i działanie silnika dla wizualizacji i zarządzania symulacją

4.3.1. Encje systemu



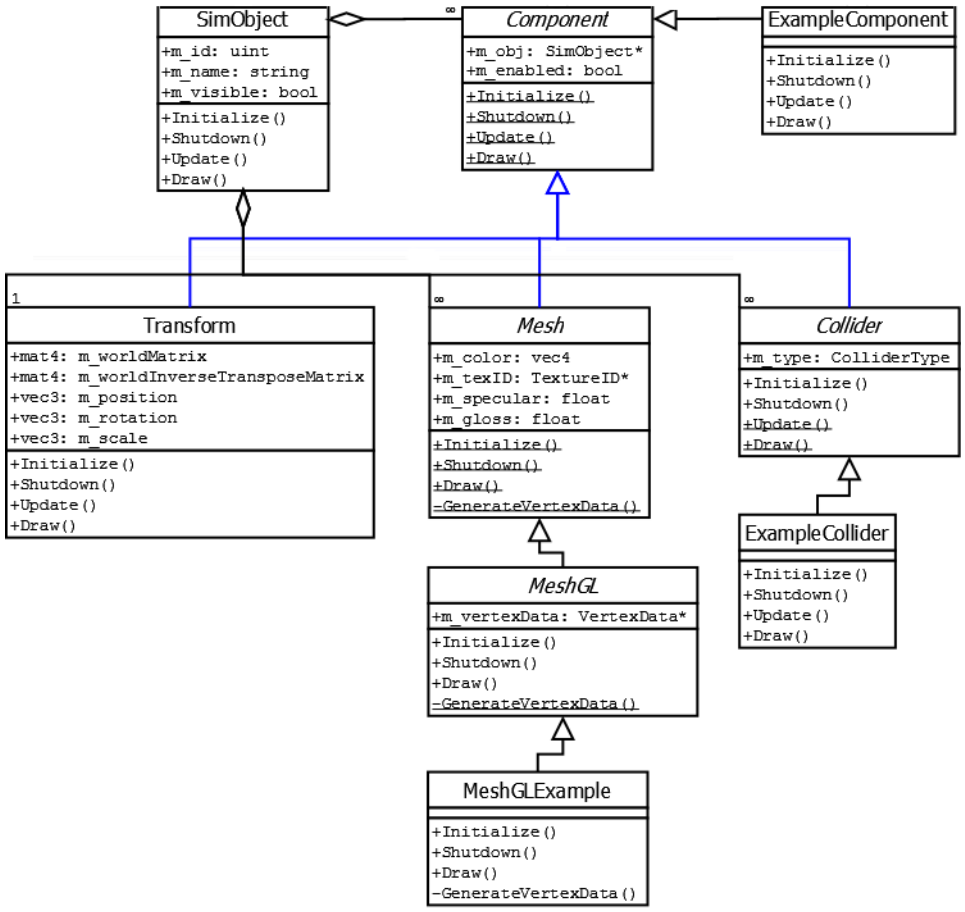
Rysunek 4.2. Architektura wirtualnej sceny. Źródło: opracowanie własne.

Diagram 4.2 przedstawia ogólną architekturę bardzo ważnego elementu silnika, jakim jest wirtualna scena, realizowanego przez klasę `Scene`. Zgodnie z przyjętym założeniem projektowym, zajmuje się ona przechowywaniem i obsługą wszystkich obiektów, z których stworzono świat symulacji.

Aby można było cokolwiek zobaczyć, niezbędna jest wirtualna kamera, której funkcjonalność enkapsuluje klasa `Camera`. Określające ją dane to macierze widoku i projekcji oraz wszelkie informacje potrzebne do ich wygenerowania. Zalicza się do nich wektory pozycji kamery, miejsca, w które patrzy się kamera, oraz kierunku „do góry". Ostatni wektor ma kluczowe znaczenie, jeżeli wystąpiłaby konieczność obrotu w osi równoległej do kierunku kamery (tzn. różnica wektorów pozycji i celu). Oprócz tego są tu dostępne informacje dotyczące własności macierzy projekcji, czyli pionowy kąt widzenia (tzw. FOV – *Field of View*), format obrazu oraz położenie płaszczyzn odcinania. Iloczyn macierzy widoku i projekcji jest niezbędny w procesie rysowania obiektów.

Scena przechowuje w kolekcjach referencje do znajdujących się w niej kamer, obiektów, elementów interfejsu oraz świateł kierunkowych, dzięki czemu możliwe jest stworzenie ich

w dowolnej liczbie. Da się łatwo ustawić, która kamera jest bieżąca, z pozycji której wyrenderowany zostanie obraz. Światła to po prostu proste kontenery danych, przechowujące kolor, kolor rozbłysku czy kierunek padania. Dane te są przekazywane do shadera podczas rysowania sceny. Elementy interfejsu zostaną dogłębniej omówione w podrozdziale 4.3.4.



Rysunek 4.3. Architektura przykładowej encji systemu. Źródło: opracowanie własne.

Diagram 4.3 opisuje budowę klasy `SimObject`, której obiekty to główne elementy wirtualnego świata symulacji. Została ona zaprojektowana zgodnie z architekturą komponentową – każdy `SimObject` zawiera w sobie kolekcję obiektów dziedziczących po klasie abstrakcyjnej `Component`. Wszystkie znajdujące się w tej kolekcji komponenty są aktualizowane podczas aktualizacji `SimObjectu`. Wpływają one na zachowanie encji w scenie i określają jej rolę z punktu widzenia całego systemu. Symulator tkaniny także jest komponentem (nosi nazwę `ClothSimulator`) i może zostać przypisany do dowolnego `SimObjectu`. Prostszy przykład to napisany do celów testowych komponent `RotateMe`, który sprawia, iż obiekt posiadający go obraca się w osi Y ze stałą prędkością. Klasa `Component` ma wśród swoich składników referencję do obiektu klasy `SimObject`, w którego skład komponentów wchodzi. Umożliwia

to łatwy i szybki dostęp oraz modyfikację wszystkich ważnych elementów i parametrów tego `SimObjectu`.

Inne elementy: `Transform`, `Mesh` i `Collider` dziedziczą po `Component`, dzięki czemu realizują funkcje komponentu, lecz dla przejrzystości traktowane są przez `SimObject` jako osobne byty. Pierwszy z komponentów jest kluczowy do określenia pozycji obiektu w scenie. Odpowiada on za przechowywanie i generowanie tzw. macierzy świata, będącej złożeniem informacji o przemieszczeniu, obrocie i skali. Komponent udostępnia także te parametry w formie trójelementowych wektorów, automatycznie aktualizując macierz świata w momencie ich zmiany z zewnątrz. Przy próbie modyfikacji pozycji automatycznie weryfikowane jest, za sprawą `PhysicsManagera`, czy po takim przesunięciu obiekt nie będzie przenikał przez inne obiekty. Sprawdzone zostają kolizje ze wszystkimi pozostałymi encjami, w oparciu o `Collidery` znajdujące się w osobnej kolekcji. `Collider` to klasa abstrakcyjna, a każda z jej implementacji stanowi inną strukturę okalającą, która musi umieć obsłużyć przecięcia z pozostałymi dostępnymi. Na potrzeby symulacji zostały stworzone, omawiane w Rozdziale 2.1.4.1 sfery okalające i prostopadłościany AABB. Klasa `Mesh` i jej podklasa `MeshGL` enkapsulują wszystkie właściwości dotyczące siatki geometrycznej obiektu, rysowanej na ekranie. Ta ostatnia także jest abstrakcyjna, a jej oferowane implementacje pozwalają na wyrenderowanie: prostopadłościanu, sfery, najprostszej płaszczyzny składającej się z czterech wierzchołków, płaszczyzny o dowolnej gęstości (używanej jako model tkaniny) oraz prostokąta w przestrzeni ekranu, na którym zostanie przedstawiony element interfejsu. Działanie będzie omówione szerzej w podrozdziale 4.3.3.

4.3.2. Komunikacja z Androidem

Konieczność komunikacji z systemem operacyjnym Android występuje zarówno podczas inicjalizacji programu, jak i w trakcie jego działania. Spora część operacji jest zautomatyzowana przez kod z pakietu Cross-platform Development, jednak o kluczowe czynności musi zadbać programista. Algorytm 1 przedstawia ogólnie ich ciąg. Na samym początku funkcja `main` otrzymuje jako argument wskaźnik do pewnego zestawu danych, opisanych strukturą `android_app`. Jej zawartość składa się z referencji do obiektów udostępnianych przez zautomatyzowany kod Javy oraz trzech ważnych wskaźników na funkcję. Do pierwszego z nich przypisany zostaje adres metody `AHandleCmd` klasy `System`, pozwalając na wywoływanie jej przez system w momencie, gdy zajdzie potrzeba obsługi zdarzenia związanego z cyklem życia aplikacji. Do drugiego wskaźnika należy podpiąć metodę odpowiedzialną za obsługę zdarzeń dla urządzeń wejściowych i jest to funkcja `AHandleInput` klasy `InputManager`. Ostatni wskaźnik ustawiono na adres funkcji `AHandleResize` klasy `Renderer`, wykonującej zmianę rozmiaru obszaru rysowania zależnie od zmiany wymiarów wyświetlacza, pojawiającej się przy obróceniu ekranu. Następnie uruchomiona zostaje kolejka zdarzeń, która później

w każdym kroku pętli głównej zostaje sprawdzona. Na koniec należy poczekać, aż Android przekaże aplikacji wymagane zasoby, m.in. kontekst renderingu, niezbędny do poprawnej inicjalizacji OpenGL. To kolejna niewygodna właściwość tej platformy.

Charakterystyczna dla systemu Android jest konieczność szczegółowego dbania o cykl życia aplikacji, czyli określanie co aplikacja ma zrobić, gdy zostanie przełączony kontekst, nastąpi uśpienie, wznowienie czy całkowite wyłączenie programu. O tych wszystkich akcjach system powiadamia aplikację, korzystając ze zdarzeń, które w każdym kroku działania są wyciągane z kolejki i obsługiwane przez wspomnianą wyżej funkcję `AHandleCmd`. Po uśpieniu program utrzymuje działającą tylko pętlę sprawdzającą kolejkę, żaden z komponentów nie jest aktualizowany. Podczas wznawiania wszystkie inicjalizowane są od nowa. Uniemożliwia to zapis stanu aplikacji przy jakiejkolwiek wymuszonej przez użytkownika przerwie w jej działaniu, ale zabezpiecza przed powstaniem w ten sposób jakichkolwiek błędów symulacji.

Jedną z największych różnic pomiędzy Androidem a platformą PC, np. Windows, jest sposób obsługi urządzeń wejścia. W drugim przypadku udostępniane są zazwyczaj funkcje pozwalające w każdym momencie „zapytać" konkretne urządzenie o stan jednego bądź wszystkich przycisków. W przypadku smartfona, nie ma klawiatury ani myszy, a ekran dotykowy. Sposób uzyskiwania informacji o tym, co się z nim aktualnie dzieje, także jest inny. Funkcja `AHandleInput` zostanie wywołana za każdym razem, gdy będzie mieć miejsce określone przez system operacyjny zdarzenie dotyczące tzw. sensorów, czyli właśnie ekranu dotykowego, akcelerometru, a nawet podpiętej do urządzenia klawiatury. Na poziomie tej metody dokonuje się filtrowania informacji tak, by reagować tylko na to, co jest interesujące z punktu widzenia aplikacji. Wykrywane i udostępniane są przesunięcia palca, wciśnięcia i puszczenia ekranu. Jako osobny gest obsługiwane jest także przesunięcie dwoma palcami naraz i tzw. *uszczypnięcie*.

Ostatnią ważną na platformie mobilnej kwestią jest obsługa obrotu ekranu, jako że użytkownik może chcieć oglądać aplikację, trzymając telefon pionowo (tryb *portrait*) lub poziomo (tryb *landscape*). Za każdym razem, gdy orientacja wyświetlacza zostanie zmieniona, system operacyjny wysyła odpowiednie zdarzenie. Obsługuje je metoda `AHandleResize`. Jej działanie jest bardzo proste – odczytane zostają nowe wymiary ekranu, a następnie wywołana zostaje funkcja `glViewport`, zmieniająca rozmiary wewnętrznego okna, do którego OpenGL rysuje. Następnie zostaje przeliczona na nowo, z aktualnymi parametrami, macierz projekcji kamery, aby uniknąć rozciągnięcia obrazu. Odpowiednio przeskalowane są także elementy interfejsu.

4.3.3. Rendering

Aby przedstawić daną encję w świecie symulacji, trzeba ją narysować na ekranie. Niezbędny w tym celu jest model 3D i wszystkie związane z nim dane potrzebne do

wyrenderowania go. Klasa `SimObject` posiada kolekcję obiektów typu `Mesh`, będącego typem abstrakcyjnym. Jedyne informacje, jakie posiada, to bardzo podstawowe parametry wyglądu, takie jak kolor, współczynnik rozbłysku, czy identyfikator tekstury. Dopiero dziedzicząca po nim klasa `MeshGL` zapewnia niemalże całą OpenGL-ową implementację. Zastosowano takie rozwiązanie, aby umożliwić łatwą podmianę kodu korzystającego z aktualnie używanej biblioteki graficznej, a co za tym idzie – łatwe przełączanie między tymi bibliotekami. Poniżej podano dokładny przebieg inicjalizacji (algorytm 4) i rysowania (algorytm 5).

Algorytm 4: Inicjalizacja modelu

Utwórz struktury przechowujące dane siatki.

Wypełnij te struktury danymi.

Wygeneruj VAO (*Vertex Array Object*) i wszystkie niezbędne bufory na GPU oraz wypełnij je danymi.

Algorytm 5: Rysowanie modelu

while *m_visible* **do**

Pobierz i włącz aktualnie używany przez `Renderer` shader.

Ustaw wszystkie parametry jednorodne (*uniforms*) shadera, w tym macierze, pozycję oka, kierunki i kolory świateł oraz parametry materiału obiektu.

Przypisz teksturę do shadera.

Włącz tablice atrybutów wierzchołków (pozycja, koordynat UV, wektor normalny, kolor, koordynat barycentryczny i indeks).

Narysuj siatkę przy pomocy funkcji `glDrawElements`.

Wyłącz tablice atrybutów wierzchołków.

end

Etap *Wypełnij te struktury danymi.* nie bez powodu został napisany kursywą. Klasa `MeshGL` jest bowiem w dalszym ciągu klasą abstrakcyjną, a funkcja odpowiedzialna właśnie za tę czynność to jej jedyna abstrakcyjna funkcja. Każda klasa dziedzicząca po `MeshGL` może we własny sposób wypełnić tablice wierzchołków oraz ich parametrów, za każdym razem w efekcie tworząc inną siatkę geometryczną. Takie podejście pozwala na proste i wygodne oddzielenie kodu samego rysowania od kodu generującego konkretny model 3D.

Sam rendering przebiega w najprostszym trybie *forward*, oznacza to, że wszystkie obiekty program rysuje prosto do bufora ramki, bądź tylnego bufora, w kolejności zgodnej z ich kolejnością występowania w kolekcji sceny i kolekcjach poszczególnych `SimObjectów`, używając testu bufora głębi do poprawnego umiejscowienia względem odległości od kamery. Rozwiązanie to ogranicza możliwość wprowadzenia dodatkowych efektów graficznych i większej liczby świateł, jednak w niniejszej symulacji nie są one potrzebne.

Wśród zasobów programu znajdują się w sumie trzy główne shadery, przy pomocy których

aplikacja rysuje scenę. Pierwszy, podstawowy, renderuje obiekty korzystając z modelu oświetlenia Phonga - Blinna, z obsługą rozbłysku specular. Obliczany jest on w shaderze fragmentów, zapewniając dużo lepszą jakość obrazu, niż w przypadku użycia do tego shadera wierzchołków, kosztem wydajności. Można sobie na to pozwolić, z racji małej liczby elementów sceny. Cel działania drugiego shadera to pokazanie struktury siatki geometrycznej obiektów – rysuje on tylko krawędzie pomiędzy wierzchołkami. W szczególnych przypadkach zwiększa to znacznie czytelność obrazu i można wykorzystać tę funkcję kiedy np. tkanina w niepoprawny sposób się zawinie, a użytkownik będzie chciał obejrzeć dokładnie, gdzie znajdują się wierzchołki w tym miejscu i jak przebiegają połączenia między nimi. Nie zostały tu użyte żadne modele oświetlenia, a po prostu narysowany zostaje jednolity kolor, przeciwny do koloru wierzchołka, tzn. $1 - k$, gdzie k – kolor wierzchołka w zakresie $[0, 1]$. Trzeci shader łączy w sobie efekty działania dwóch poprzednich.

4.3.4. Interfejs użytkownika

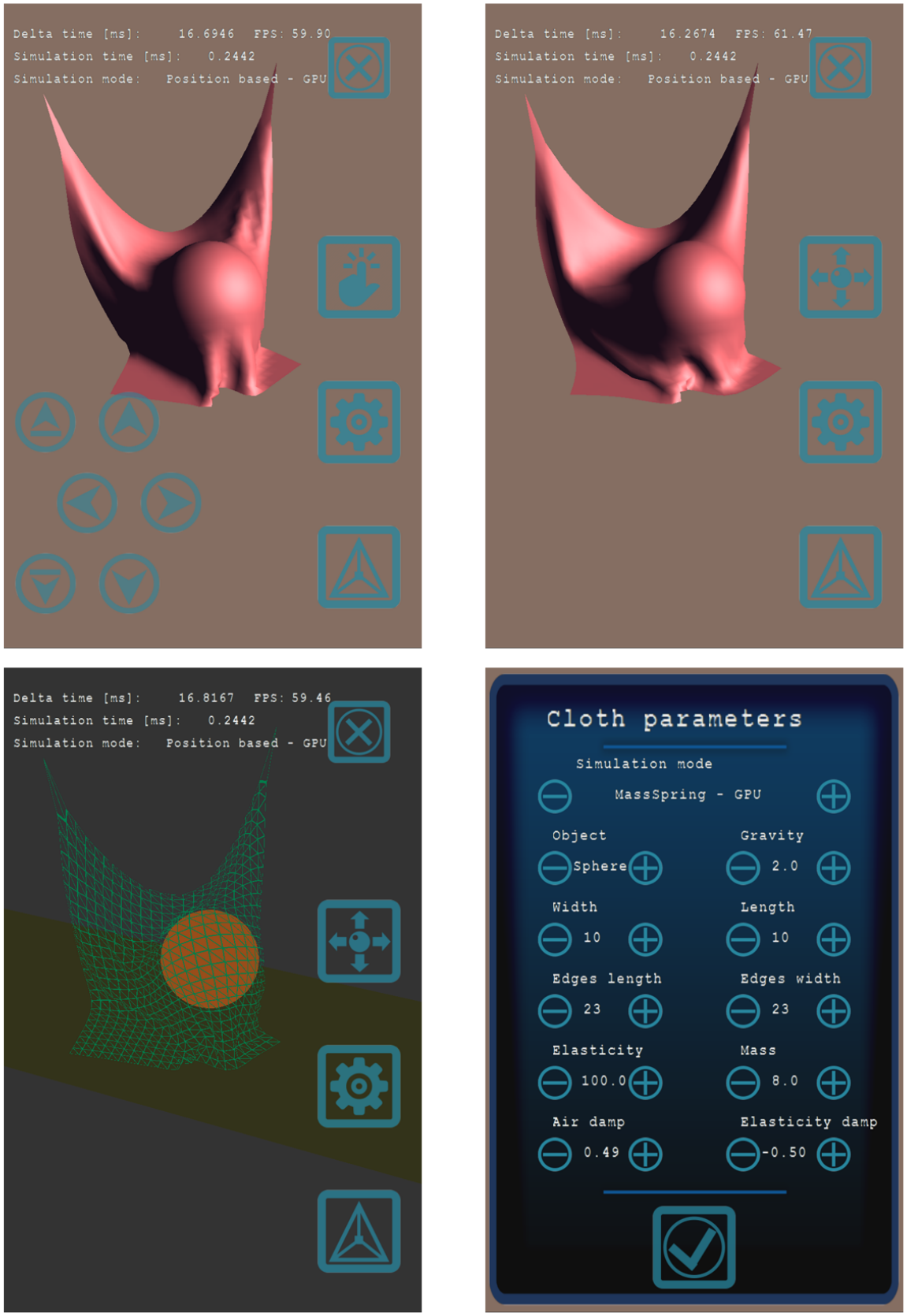
Niezbędnym do satysfakcjonującej wizualizacji symulacji tkanin i możliwości interakcji z nią użytkownika jest stworzenie odpowiedniego interfejsu graficznego, zwanego w skrócie GUI (*Graphical User Interface*). Dzięki niemu zaistnieje możliwość zarówno podejmowania pewnych akcji w symulowanym świecie, bądź poza nim, jaki i przedstawienie użytkownikowi pewnych informacji zwrotnych dotyczących głównie statystyk działania programu. GUI umożliwiło zarówno łatwe i szybkie zebranie wyników testów w Rozdziale 6, jak i miłe dla oka zaprezentowanie symulacji.

Urządzenia wejściowe obsługują klasy `InputManager` oraz `InputHandler`. Są one ze sobą nierozerwalnie związane. Pierwsza zapewnia niskopoziomową obsługę wszelkich zdarzeń pochodzących z urządzeń wskazujących oraz wyciągnięcie z nich informacji o np. wciśniętych klawiszach, o ile to możliwe. Dane, które zawarte są w owych zdarzeniach zostają sformułowane i udostępnione w postaci wygodnych elementów, tj. zmiennych logicznych, umożliwiających przykładowo sprawdzenie, czy aktualnie ekran dotykowy jest wciśnięty, oraz wektorów dwuwymiarowych odzwierciedlających pozycję wciśnięcia oraz kierunek przesunięcia palca, bądź palców po ekranie. Przeciągnięcie dwoma palcami `InputManager` także obsługuje, tak samo jak gest „uszczypnięcia", czyli tzw. *pinch*. Z kolei `InputHandler` przekuwa informacje o stanie urządzeń wejściowych na dane o możliwości wykonania konkretnych akcji systemu. Przykładowo, jego funkcja `GetCameraMovementVector` odnosi się do metody `GetDoubleTouchDirection InputManagera`. Takie podejście pozwala nam na szybką zmianę sterowania systemem bez potrzeby przerabiania wszystkich zaangażowanych w to komponentów, a jedynie zmieniając implementację funkcji klasy `InputHandler`.

Program potrafi rysować dwuwymiarowe elementy GUI w przestrzeni ekranu, takie jak: tekst, obrazki oraz dwustanowe przyciski. Tekst może być dynamicznie zmieniany w każdym

kroku pętli głównej. Klasą bazową przedstawiającą abstrakcyjny element interfejsu jest klasa `GUIElement`, skupiająca w sobie funkcje wspólne dla każdego z wymienionych wyżej rodzajów. Dziedziczą po niej m.in. `GUIText`, `Picture` czy `GUIButton`. Składniki GUI mogą być łączone w hierarchię, co pozwala na łatwe wyłączenie lub włączenie pewnej części interfejsu, np. tekstu oraz na optymalizację wykrywania kliknięcia – pozycja palca przy naciśnięciu ekranu nie musi być sprawdzana dla wszystkich elementów. Za większość głównych akcji, jakie użytkownik może wykonać odpowiadają przyciski, dla których można zdefiniować osobne zbiory operacji zarówno przy zwykłym krótkim wciśnięciu, jak i przytrzymaniu.

Komponent `GUIController` jest ostatnim, acz bardzo ważnym ogniwem systemu interfejsu. Zamienia on sygnały wejściowe, udostępniane przez klasę `InputHandler` na konkretne działanie w systemie. Zawiera kod obsługujący ruchy kamerą, logikę przycisków i pozostałych elementów GUI. Aktualizuje także informacje tekstowe na ekranie.



Rysunek 4.4. Interfejs użytkownika aplikacji. Źródło: opracowanie własne.

Projekt aplikacji zakłada, że użytkownik musi mieć możliwość zmiany położenia, obrotu i przybliżenia kamery, resetowania symulacji i modyfikacji jej parametrów, zmiany trybu wyświetlania obiektów oraz interakcji z tkaniną na dwa sposoby – przemieszczając obiekt wchodzący z nią w kolizje lub przesuwając ją samą przy pomocy ruchów palca. Wymagane jest także informowanie użytkownika o szybkości działania symulacji oraz o tym, jakie ma ona aktualnie parametry i jakiego jest typu.

Rysunek 4.4 pokazuje, w jaki sposób zostały zrealizowane te założenia. Tekst wyświetlany u góry ekranu zawiera wszelkie dane, pozwalające użytkownikowi dowiedzieć się o wydajności symulacji i wybranym jej modelu. Przycisk z rysunkiem siatki trójkąta przełącza tryby wyświetlania, czego efekt widać na lewym dolnym obrazku – rysowane są tylko krawędzie geometrii tkaniny. Przycisk w kształcie rączki bądź kulki ze strzałkami zmienia sposób, w jaki użytkownik dokonuje interakcji. Może on przemieszczać wybrany obiekt kolidujący przy pomocy umieszczonych w lewym dolnym rogu ekranu strzałek, bądź w drugim trybie przesuwać tkaninę palcem – strzałki wtedy znikają. Przycisk z zębatką otwiera menu wyboru parametrów aplikacji, dokładniej opisanych w Rozdziale 5.3.2, gdzie korzystając z ikonek plus i minus da się je zmieniać. Użytkownik ma możliwość opuszczenia aplikacji, używając przycisku „X” w prawym górnym rogu ekranu.

Rozdział 5

Budowa i działanie symulatora tkaniny

5.1. Założenia projektowe

Całość funkcji dotyczących symulacji tkaniny skupiono w klasie `ClothSimulator`. Dziedziczy ona po typie abstrakcyjnym `Component`. Za sprawą tego bardzo dobrze komponuje się z architekturą silnika, bez problemu można ją dodać do dowolnego `SimObjectu` oraz powielić, a także włączać lub wyłączać jej działanie.

W prezentowanej aplikacji został umieszczony jednak tylko jeden `ClothSimulator`, aby uprościć pracę programu i ułatwić pomiary testowe. Za cel części praktycznej pracy postawiono sobie możliwość zasymulowania zachowania pojedynczej tkaniny o prostokątnym kształcie (rysowane czworokąty mają kształt prostokątów), zawieszonej sztywno w powietrzu za dwa sąsiednie narożniki, poddającej się działaniom sił grawitacji oporu powietrza i wchodzącej w interakcje z obiektami sceny oraz sygnałami od użytkownika, za pośrednictwem ekranu dotykowego smartfona. Symulacja miała udostępniać opcję bycia obliczaną dwoma metodami – masy na sprężynie i bazującą na pozycji oraz zostać zaimplementowana na trzy sposoby – przy użyciu GPU, sekwencyjnie na CPU oraz współbieżnie na CPU, z wykorzystaniem czterech wątków roboczych. Użytkownikowi pozwolono na zmianę istotnych parametrów symulacji poprzez ich wybór z ustalonego zakresu. Wszystkie wymienione wyżej cele zostały zrealizowane. Przeznaczeniem symulatora jest jednak nie tylko wizualizacja, ale też i udostępnienie możliwości oceny modeli tkanin oraz ich implementacji pod kątem wydajności, dlatego program wyświetla informacje o czasie trwania pojedynczego kroku symulacji, a także o całościowym okresie jednego przebiegu pętli głównej programu.

5.2. Wydajność a użycie pamięci

Jako, że flagowym celem niniejszej pracy było zaimplementowanie wydajnej symulacji z użyciem GPU, nadrzędne założenie przy projektowaniu to jak największe przyspieszenie przetwarzania kosztem większego użycia pamięci. Wszystkie możliwe dane, które nie muszą

być przeliczane w każdym kroku, zostają obliczone podczas inicjalizacji symulatora, a wyniki są po prostu przesyłane do odpowiednich funkcji w trakcie działania programu. Wpasowuje się to doskonale w metodykę programowania GPU, za sprawą chociażby minimalizacji liczby instrukcji warunkowych oraz uniknięcia zbędnych obliczeń, które byłyby wielokrotnie wykonywane.

Przykładowo, sprawdzając sąsiadów wierzchołka, należy obliczyć ich identyfikatory (tzn. poznać, które to konkretnie są wierzchołki) oraz zawsze mieć pewność, iż takowy istnieje – nie wszystkie mają czterech sąsiadów, a dokładnie – nie posiadają tylu te znajdujące się na zewnętrznych krawędziach siatki. Problem zostaje rozwiązany, gdy każdemu wierzchołkowi przypisane zostaną, ustalone przy starcie komponentu, lista identyfikatorów oraz mnożników, które wynoszą 1 gdy sąsiad istnieje, bądź 0 gdy go nie ma, i w tym przypadku obliczona siła bądź przesunięcie nie biorą udziału w dalszym przetwarzaniu. Wyeliminowana jest także konieczność użycia instrukcji warunkowej, co dodatkowo poprawia wydajność.

Niestety, takie podejście zwiększa zużycie pamięci. Jak można się domyślić, jest ono proporcjonalne do ustalonej przez użytkownika gęstości siatki tkaniny. Ponadto, wszystkie wierzchołki wraz z ich parametrami są przechowywane dwukrotnie, z racji konieczności posiadania informacji o pozycjach poprzednich, dla całkowania Verleta (wzór (2.9), Rozdział 2.1.1). W przypadku, gdy wybranym trybem symulacji jest tryb GPU, wszystkie te dane zostają dodatkowo skopiowane do pamięci karty graficznej. Dochodzą jeszcze do tego parametry pomocnicze, takie jak wymienione wyżej listy identyfikatorów sąsiadów. Przykładowo, użytkownik chce stworzyć siatkę o $m \times n$ dodatkowych krawędzi bocznych. Liczba wierzchołków będzie wynosić $(m+2)(n+2)$. Z każdym z nich wiążą się następujące atrybuty:

— Pozycja (16 B),
— Koordynat teksturowania (8 B),
— Wektor normalny (16 B),
— Kolor (16 B),
— Koordynat barycentryczny (16 B),
— Indeks (4 B).

Rozmiar niektórych elementów został sztucznie zwiększony tak, aby był wielokrotnością 4 B. Dzięki temu dane zostały poprawnie ułożone w buforach jednorodnych, omawianych w rozdziale 3.2.2.3. W sumie jeden wierzchołek zajmuje 152 B pamięci – wzięto pod uwagę podwójne występowanie. Należy jednak pamiętać jeszcze o właściwych dla symulacji parametrach, opisanych w podrozdziale 5.3.2, z których część jest przypisywana każdemu wierzchołkowi oddzielnie. Dodają one do naszych obliczeń kolejne 128 B. W sumie cała tkanina wraz z parametrami waży:

$$s = 280(m+2)(n+2) + 32 \ , \tag{5.1}$$

gdzie s to rozmiar w bajtach. Ostatnia wartość wynika z konieczności przechowywania pojedynczego wektora początkowych odległości między danym a sąsiednimi wierzchołkami, takich samych dla każdego, gdyż wzięto pod uwagę jednorodną prostokątną siatkę, oraz wektora przesunięcia palca użytkownika po ekranie dotykowym, kluczowego dla realizacji opisanej w podrozdziale 5.3.5 interakcji. Przykładowo, dla gęstej, jak na warunki urządzenia mobilnego, siatki 98×98 krawędzi, zajętość pamięci wynosi 2800032 B, czyli ok. 2,7 MiB. Jest to liczba spora jak na jeden obiekt logiczny, lecz w żadnym wypadku nie krytyczna dla testowego smartfona, wyposażonego w 2 GB RAM. Oczywiście w naszych obliczeniach zignorowane zostały rozmaite zmienne pomocniczych dla symulatora tkanin, identyfikatory buforów i struktur GPU oraz zapisane wartości parametrów, jednakże ich rozmiar jest pomijalnie mały.

5.3. Zasada działania

5.3.1. Ogólny algorytm

Algorytm działania symulacji jest taki sam dla wszystkich trzech obsługiwanych implementacji – GPU, CPU i CPU na 4 wątkach. Różnice pojawiają się tylko w wywołaniach funkcji.

Dla pierwszego przypadku w każdym kroku należy przeprowadzić przypisanie GPU odpowiednich tablic atrybutów wierzchołków, które zawierają niezbędne symulacji, wygenerowane uprzednio dane. Następnie zostaje ustawiony dla OpenGL program wykonujący obliczenia, ustawione są wszystkie zmienne jednorodne, związane bufory jednorodne i uruchomione sprzężenie transformacyjne. Wywołanie `glDrawArrays` rozpoczyna obliczenia.

W przypadku CPU sprawa jest dużo prostsza, jako, że wszystkie dane i tablice są już zainicjalizowane. Można od razu przystąpić do działania. Sytuacja komplikuje się, gdy wykorzystana została opcja wielowątkowości. W tej metodzie rozłożono pracę na cztery wątki robocze, z racji tego, iż urządzenie dysponuje czterema fizycznymi jednostkami przetwarzającymi. Algorytm podziału jest prosty – liczbę wierzchołków dopełnia się do liczby podzielnej bez reszty przez 4 i dzieli ją na cztery równe zakresy, przy czym brane pod uwagę są pierwsze trzy, a ostatni jest równy liczbie pozostałych wierzchołków. Do synchronizacji użyto muteksów i licznika wątków, które zakończyły pracę. Działaniem wątków roboczych zarządza wątek główny. Każdy z tych pierwszych jest uśpiony na muteksie, dopóki nie nadejdzie wywołanie funkcji `Update` symulatora. Wtedy zostają one obudzone i rozpoczynają

obliczenia. Po ich zakończeniu podnoszą licznik i czekają na następnym muteksie. Główny wątek z kolei czeka, aż licznik osiągnie wymaganą wartość i odblokowuje następny etap obliczeń.

Cały proces podzielono na trzy oddzielne etapy, omówione dalej. Pierwszy to obliczenia ruchu tkaniny, zgodnie z przyjętym modelem symulacji, drugi – rozwiązywanie kolizji oraz zaaplikowanie ruchu tkaniny wynikłego z interakcji użytkownika. Te dwie kwestie opisano w oddzielnych podrozdziałach, jednak z punktu widzenia implementacji wchodzą one w skład jednego etapu. Ostatnim jest przeliczenie wektorów normalnych. Po dwóch pierwszych etapach następuje zamiana identyfikatorów struktury danych wejściowych ze strukturą danych wyjściowych, zastosowano tu tzw. metodę ping–pongową. Gdyby tego nie robić, efektem byłby szybki „wybuch" symulacji, jako że obliczenia na danym wierzchołku mogłyby pobierać część danych już w danym kroku zaktualizowanych, a część nie. W przypadku obu implementacji na CPU po zakończeniu kroku przetwarzania należy jeszcze przesłać nowe dane o pozycjach i wektorach normalnych wierzchołków do GPU, w celu umożliwienia ich narysowania. Dzieje się to przy pomocy funkcji `glBufferSubData`.

Poniżej podano algorytmy 6, 7 i 8, osobno dla każdej z implementacji. Dzięki temu można zaobserwować kluczowe różnice w tych podejściach, a także ocenić skomplikowanie kodu.

Algorytm 6: Symulacja na GPU.

Inicjalizuj parametry tkaniny.
Utwórz identyfikatory buforów GPU.
Przypisz istniejące już identyfikatory buforów siatki modelu tkaniny do odpowiednich identyfikatorów w klasie symulacji.
Utwórz VAO i bufory dla wszystkich wymaganych parametrów tkaniny.
Załaduj kernel rozwiązywania kolizji, kalkulacji wektorów normalnych i symulacji tkaniny przy użyciu wybranego modelu.
Utwórz obiekt transformacyjnego sprzężenia zwrotnego dla wszystkich etapów i odpowiednio dla każdego z nich bufory zwrotne oraz bufory jednorodne.
while *m_running* **do**
    Pobierz wymagane dane z systemu.
    Przypisz wszystkie tablice atrybutów wierzchołków z parametrami tkaniny właściwymi każdemu wierzchołkowi. (`glBindBuffer(GL_ARRAY_BUFFER, ...)`, `glEnableVertexAttribArray`, `glVertexAttribPointer`)
    Użyj kernela obliczeń ruchu tkaniny. (`glUseKernel`)
    Ustaw zmienne jednorodne i bufory jednorodne. (`glBindBufferBase(GL_UNIFORM_BUFFER, ...)`, `glUniform...`)
    Wyłącz rasteryzer. (`glEnable(GL_RASTERIZER_DISCARD)`)
    Zwiąż obiekt transformacyjnego sprzężenia zwrotnego. (`glBindTransformFeedback`)
    Zwiąż odpowiednie bufory zwrotne. (`glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, ...)`)
    Uruchom transformacyjne sprzężenie zwrotne. (`glBeginTransformFeedback(GL_POINTS)`)
    Rozpocznij obliczenia. (`glDrawArrays`)
    Włącz rasteryzer. (`glDisable(GL_RASTERIZER_DISCARD)`)
    Przełącz identyfikatory buforów odczytu i zapisu.
    Użyj kernela rozwiązywania kolizji.
    Dokonaj operacji przygotowawczych i uruchamiających obliczenia w podobny sposób.
    Przełącz identyfikatory buforów odczytu i zapisu.
    Użyj kernela kalkulacji wektorów normalnych.
    Dokonaj operacji przygotowawczych i uruchamiających obliczenia w podobny sposób.
    Przełącz identyfikatory buforów odczytu i zapisu.
end

Algorytm 7: Symulacja na CPU.

Inicjalizuj parametry tkaniny.

while *m_running* **do**

- Pobierz wymagane dane z systemu.
- Dla każdego wierzchołka oblicz nowe położenie zgodnie z przyjętym modelem symulacji.
- Przełącz identyfikatory buforów odczytu i zapisu.
- Dla każdego wierzchołka rozwiąż kolizje i przesuń zgodnie z sygnałem z urządzenia wejściowego.
- Przełącz identyfikatory buforów odczytu i zapisu.
- Dla każdego wierzchołka oblicz nowy wektor normalny.
- Zaktualizuj bufor pozycji wierzchołków na GPU. (`glBufferSubData`)
- Zaktualizuj bufor wektorów normalnych wierzchołków na GPU.

end

Algorytm 8: Symulacja na CPU z użyciem 4 wątków roboczych.

Wątek główny:

Inicjalizuj parametry tkaniny.

Utwórz muteksy: trzy dla każdego etapu obliczeń, dla licznika wątków i zmiennej logicznej kończącej obliczenia – `m_threadsRunning`. (`pthread_mutex_init`)

Zablokuj muteksy odpowiadające etapom obliczeń. (`pthread_mutex_lock`)

Utwórz struktury danych dla wątków – podziel dane tkaniny na 4 części.

Uruchom wątki. (`pthread_create`)

while *m_running* **do**

Dla dwóch pierwszych etapów obliczeń:

Odblokuj muteks danego etapu.

Czekaj, aż licznik wątków osiągnie wartość liczby wątków roboczych.

Zablokuj muteks danego etapu.

Ustaw licznik wątków na zero.

Przełącz identyfikatory buforów odczytu i zapisu.

Odblokuj muteks etapu trzeciego.

Czekaj, aż licznik wątków osiągnie wartość liczby wątków roboczych.

Zablokuj muteks etapu trzeciego.

Ustaw licznik wątków na zero.

Zaktualizuj bufor pozycji wierzchołków na GPU. (`glBufferSubData`)

Zaktualizuj bufor wektorów normalnych wierzchołków na GPU.

end

Wątek roboczy:

while *m_threadsRunning* **do**

Czekaj na odblokowanie muteksu etapu 1.

Pobierz wymagane dane z systemu.

Dla każdego przydzielonego wierzchołka oblicz nowe położenie zgodnie z ustalonym modelem symulacji.

Inkrementuj licznik wątków, które ukończyły pracę.

Czekaj na odblokowanie muteksu etapu 2.

Dla każdego przydzielonego wierzchołka rozwiąż kolizje i przesunięcie zgodnie z sygnałem z urządzenia wejściowego.

Inkrementuj licznik wątków, które ukończyły pracę.

Czekaj na odblokowanie muteksu etapu 3.

Dla każdego przydzielonego wierzchołka, oblicz nowy wektor normalny.

Inkrementuj licznik wątków, które ukończyły pracę.

end

5.3.2. Parametry symulacji

Symulacja tkanin do działania wymaga zdefiniowania pokaźnej liczby parametrów. Niektóre z nich mogą być różne dla każdego wierzchołka, dlatego przekazywane są do GPU w postaci tablic atrybutów. Inne będą zawsze takie same, niezależnie od tego, który wierzchołek jest przetwarzany i przesłanie ich jako parametry jednorodne cechuje się większą optymalnością. Większość parametrów użytkownik może zmienić, wybierając wartość ze z góry ustalonego zakresu bądź zbioru. Poniżej wymienione i opisane są wszystkie kluczowe parametry dostępne do modyfikacji przez użytkownika, wraz z ich znaczeniem dla symulacji. W programie, dla wygody, zostały one upakowane w strukturach `SimParams` i `SimData`. Pierwsza z nich znajduje zastosowanie podczas pobierania parametrów z interfejsu użytkownika i przekazywania ich do klasy `ClothSimulator`. Zawartość drugiej powstaje w procesie inicjalizacji tablic danych, gotowych do wykorzystania podczas obliczeń.

Tryb symulacji Zakres: Masa na sprężynie – GPU, Masa na sprężynie – CPU, Masa na sprężynie – CPUx4, Oparty na pozycji – GPU, Oparty na pozycji – CPU, Oparty na pozycji – CPUx4.

Obiekt wchodzący w kolizje Zakres: prostopadłościan, sfera. Jest to obiekt umiejscowiony na tyle blisko tkaniny tak, aby mogła ona wchodzić z nim w kolizje. Użytkownik ma możliwość przesuwania go we wszystkich osiach układu współrzędnych.

Przyspieszenie grawitacyjne Zakres: $0.1 - 10\ \frac{m}{s^2}$. Używane do obliczenia siły grawitacji działającej na każdy wierzchołek na etapie kalkulacji ruchu tkaniny.

Masa Zakres: $1 - 50\ kg$. Używana do obliczenia przyspieszenia wierzchołka, wymaganego jako dana dla całkowania Verleta.

Współczynnik oporu powietrza Zakres: $0 - 0.99$. Używany do obliczenia siły oporu powietrza, działającej na każdy wierzchołek mający różną od zera prędkość.

Współczynnik elastyczności Zakres: $0 - 1000$. Używany zarówno jako współczynnik elastyczności sprężyn tkaniny w modelu masy na sprężynie, jak i jako parametr sztywności ograniczników w modelu opartym na pozycji, po odpowiednim przeskalowaniu.

Współczynnik tłumienia drgań Zakres: $0 -$ -20. Używany w obliczeniach sił sprężystości dla modelu masy na sprężynie. Dzięki niemu minimalizowane jest ryzyko wpadnięcia tkaniny w niekontrolowane drgania.

Szerokość Zakres: $1 - 50\ m$. Rozmiar tkaniny w osi X układu współrzędnych, zakładając, że siatka tkaniny jest równoległa do płaszczyzny XZ.

Długość Zakres: $1 - 50\ m$. Rozmiar tkaniny w osi Z układu współrzędnych, zakładając, że siatka tkaniny jest równoległa do płaszczyzny XZ.

Ilość krawędzi poziomych Zakres: $0 - 126$. Ten i następny są kluczowymi parametrami dla testów wydajności i efektu wizualnego. Liczba krawędzi ma wpływ na gęstość siatki

tkaniny, a co za tym idzie na dokładność oraz ciężar obliczeniowy symulacji. Wartości tych parametrów są ograniczone do 126, czyli maksymalnie można wygenerować siatkę o gęstości 128×128 krawędzi. Limit ten wynika z opisanej w Rozdziale 3.2.2.3 konieczności zastosowania buforów jednorodnych, które mają niewielką pojemność.

Ilość krawędzi pionowych Zakres: $0 - 126$.

Natomiast na potrzeby samej symulacji używane są także poniższe parametry opisujące tkaninę, ustalone na stałe, bądź modyfikowane w każdym kroku symulacji. Użytkownik zmienić niektóre z nich tylko nie wprost, poprzez opisaną dalej interakcję z tkaniną.

Początkowa pozycja tkaniny Ustalona poza klasą `ClothSimulator` – w komponencie `Transform` tego samego obiektu. Wynosi 10 m ponad początkiem układu współrzędnych. Reprezentowana w postaci macierzy świata o wymiarach 4×4.

Wektor położenia wierzchołka Składa się z czterech komponentów. Ostatni nie bierze udziału w obliczeniach, a ma znaczenie dla kwestii poprawnego ułożenia danych w pamięci. Aktualizowany w każdym kroku symulacji.

Wektor normalny wierzchołka Jak wyżej. Służy do wyliczenia równania oświetlenia podczas rysowania tkaniny. Także aktualizowany w każdym kroku.

Indeksy sąsiadów wierzchołka Używane do wyboru wierzchołków, z którymi przetwarzany aktualnie wierzchołek połączono sprężynami bądź ogranicznikami. Jest ich 12 – 8 dla najbliższych sąsiadów i 4 dla sąsiadów oddalonych o 2 pozycje, tylko po krawędziach prostopadłych do osi układu współrzędnych.

Mnożniki sąsiadów wierzchołka Jest ich także 12. Algorytm wybierający sąsiadów, przydziela ich zawsze taką samą liczbę, nie biorąc pod uwagę, czy sąsiad faktycznie istnieje. Może tak być dla wierzchołków znajdujących się na zewnętrznych krawędziach siatki. Mnożniki przyjmujące wartości 0 lub 1 rozwiązują ten problem – dla nieistniejącego sąsiada obliczone przesunięcie nie bierze udziału w dalszym przetwarzaniu.

Mnożnik blokady wierzchołka Może wynosić 0, wtedy wierzchołek nigdy się nie poruszy i nie podlega symulacji, bądź 1. Używany, by „zawiesić" tkaninę za narożniki.

Promień sfery okalającej wierzchołka Kluczowy przy detekcji kolizji. Każdy wierzchołek posiada własną sferę okalającą, a jej promień jest obliczany wzorem podanym w algorytmie 9.

Odległości początkowe od sąsiadów wierzchołka Niezbędne w obliczaniu sił sprężystości bądź ograniczników. Każdy ze wzorów użytych w modelach symulacji wymaga dostarczenia tych danych. Obliczane we wzorze podanym w algorytmie 9.

Proces wspomnianej wyżej inicjalizacji struktur parametrów, z których korzystać będzie symulacja, można opisać podanym poniżej algorytmem 9. Ma on miejsce zawsze, kiedy

użytkownik wybierze opcję zatwierdzenia ustawień na ekranie ich wyboru. W algorytmie zastosowano następujące oznaczenia: C – liczba wierzchołków, E_w – całkowita liczba krawędzi prostopadłych do osi X układu współrzędnych, E_l – całkowita liczba krawędzi prostopadłych do osi Z układu współrzędnych (przy założeniu, że siatka tkaniny jest równoległa do płaszczyzny XZ), p_i – położenie wierzchołka, M – masa całkowita, będąca parametrem wprowadzonym przez użytkownika.

Algorytm 9: Inicjalizacja parametrów tkaniny.

Alokuj pamięć na potrzebne struktury danych i bufory.

Dla każdego wierzchołka:

Oblicz i przypisz odległości początkowe pomiędzy wierzchołkami.

Odległość względem długości: $b_l = (p_{0_z} - p_{(C-1)_z})/(E_w - 1)$.

Odległość względem szerokości: $b_w = (p_{0_x} - p_{(C-1)_x})/(E_l - 1)$.

Odległość diagonalna: $b_d = \sqrt{(b_l)^2 + (b_w)^2}$.

Zapisz mnożnik dla odległości od wierzchołków następnych po sąsiadach: 2 .

Oblicz i przypisz masę pojedynczego wierzchołka: $m_i = \frac{M}{\sqrt{(min(C,0.02)}}\ kg.$

Oblicz i przypisz promień sfery okalającej: $r_i = min(\frac{min(b_l,b_w)}{2}, 0.35)\ m.$

Przypisz współczynniki: elastyczności, tłumienia drgań, oporu powietrza.

Oblicz i przypisz indeksy wierzchołków sąsiednich oraz ich mnożników.

Przypisz mnożnik blokady (wartość 0) do górnych wierzchołków $i = 0$ i $i = C - E_l$.

5.3.3. Obliczenia ruchu tkaniny

Jest to pierwszy etap symulacji i zarazem najważniejsza jego część. To tutaj obliczane są przemieszczenia wierzchołków w taki sposób, aby odwzorować zachowanie tkaniny. Poniżej podano fragmenty kodu w języku GLSL najpierw dla modelu masy na sprężynie, a następnie – modelu opartego na pozycji. Należy pamiętać, że wykonuje się on dla pojedynczego wierzchołka. Kod implementacji na CPU jest analogiczny, przy czym zamiast równoległych obliczeń użyto pętli `for`.

Listing 5.1. Obliczanie przesunięcia – model masy na sprężynie.

```
vec3 CalcSpringForce(vec3 mPos, vec3 mPosLast, vec3 nPos, vec3 nPosLast,
   float sLength, float elCoeff, float dampCoeff)
{
   vec3 ret = vec3(0.0f, 0.0f, 0.0f);
   vec3 mVel = (mPos - mPosLast) / DeltaTime;
   vec3 nVel = (nPos - nPosLast) / DeltaTime;
   vec3 f = mPos - nPos;
   vec3 n = normalize(f);
   float fLength = length(f);
   float spring = fLength - sLength;
   vec3 springiness = - elCoeff * spring * n;
   vec3 dV = mVel - nVel;
   float damp = dampCoeff * (dot(dV, f) / fLength);
   vec3 damping = damp * n;
   float sL = length(springiness);
   damping = n * min(sL, damp);
   ret = (springiness + damping);
   return ret;
}

void main()
{
   int mID = gl_VertexID;
   vec3 mPos = vec3(Pos);
   vec3 mPosLast = vec3(PosLast);
   vec3 mVel = (mPos - mPosLast) / DeltaTime;
   vec3 mForce = vec3(0.0f, 0.0f, 0.0f);
   // wyznaczanie tablic z dlugosciami sprzezyn, dla ulatwienia dostepu z petli
   float sls1[4] = float[4](
      SpringLengths.y, SpringLengths.x, SpringLengths.y, SpringLengths.x
   );
   ...
   // obliczanie sil sprzezystosci dla wszystkich sasiadow
   for(int i = 0; i < 4; ++i)
   {
      int nID = int(roundEven(Neighbours[i]));
      vec3 nPos = vec3(InPosBuffer[nID]);
      vec3 nPosLast = vec3(InPosLastBuffer[nID]);
```

```
        vec3 force = CalcSpringForce(..);
        mForce += force * NeighbourMultipliers[i];
    }
    ...
    // obliczanie przemieszczenia
        vec3 newPos;
        vec3 acc = mForce / ElMassCoeffs.y;
        newPos = 2.0f * vec3(Pos) - vec3(PosLast) + acc * DeltaTime * DeltaTime;
    // aktualizacja pozycji
        OutPos = vec4(newPos, 1.0f);
        OutPosLast = Pos;
        gl_Position = Pos;
}
```

Listing 5.2. Obliczanie przesunięcia – model oparty na pozycji

```
void CalcDistConstraint(vec3 mPos, vec3 nPos, float mass,
    float sLength, float elCoeff, float dampCoeff,
    out vec4 constraint)
{
    elCoeff = clamp(elCoeff, 0.0f, 1.0f);
    vec3 diff = mPos - nPos;
    float cLength = length(diff);
    vec3 dP = (2.0f * mass) * (cLength - sLength) * (diff / cLength) * elCoeff;
    constraint.xyz = dP;
    constraint.w = 1.0f / mass;
}

void main()
{
    ...
    ////////////////////////////////////////////////////
    // kalkulacja sil
    vec3 mForce = vec3(0.0f);
    vec3 posPredicted = vec3(0.0f);
    ...
    vec3 acc = mForce / ElMassCoeffs.y; // masa
    posPredicted = 2.0f * vec3(Pos) - vec3(PosLast) + acc
```

```
        * DeltaTime * DeltaTime;
    // ograniczniki
    float elBias = 0.0005;
    vec3 cPos = vec3(0.0f);
    for(int i = 0; i < 2; ++i)
    {
        int id = fOrder[i];
        int nID = int(roundEven(Neighbours[id]));
        vec3 nPos = vec3(InPosBuffer[nID]);
        vec3 nPosLast = vec3(InPosLastBuffer[nID]);
        // ogranicznik pozycji. XYZ to pozycja, W - odwrotnosc masy
        vec4 constraint;
        CalcDistConstraint(...);
        cPos -= constraint.xyz * constraint.w *
            NeighbourMultipliers[id] * Multipliers.x;
    }
    posPredicted += cPos;
    cPos = vec3(0.0f);
    for(int i = 2; i < 4; ++i)
    {
        int id = fOrder[i];
        int nID = int(roundEven(Neighbours[id]));
        vec3 nPos = vec3(InPosBuffer[nID]);
        vec3 nPosLast = vec3(InPosLastBuffer[nID]);
        vec4 constraint;
        CalcDistConstraint(...);
        cPos -= constraint.xyz * constraint.w *
            NeighbourMultipliers[id] * Multipliers.x;
    }
    posPredicted += cPos;
    ...
    // aktualizacja pozycji
    OutPos = vec4(finalPos, 1.0f);
    OutPosLast = Pos;
    gl_Position = Pos;
}
```

Jak widać, implementacje obu modeli symulacji mają ze sobą sporo wspólnego i tak naprawdę ich główna różnica polega na podejściu do kalkulacji sprężyn bądź ograniczników. Zauważyć można zastosowanie wzorów z rozdziałów 2.1.1 i 2.1.2. Cały proces dotyczący obliczania sił zewnętrznych wygląda w obu przypadkach tak samo i został tu pominięty. Metody różnią się także kolejnością wykonywania działań. W modelu masy na sprężynie siły sprężystości są obliczane na początku, chociaż kolejność nie ma tu tak naprawdę znaczenia. Z kolei ma ona znaczenie w modelu opartym na pozycji – tutaj ograniczniki nałożone zostają już na obliczoną pozycję, zmienioną pod wpływem sił zewnętrznych.

5.3.4. Rozwiązywanie kolizji

Etapem drugim jest rozwiązanie kolizji zewnętrznych, wewnętrznych, a także uwzględnienie działania użytkownika na tkaninę. Aby dokonać pierwszej z tych czynności, należy mieć dane wszystkich struktur okalających dla każdego obiektu w scenie. Zapewnia je nam klasa–singleton `PhysicsManager`, gdzie przechowywane są one w formie upakowanych struktur, o rozmiarze wyrównanym do wielokrotności 4 B. Dzięki temu można, korzystając z buforów jednorodnych, bezpośrednio wysłać je do GPU jako parametry jednorodne. Warto pamiętać także o tym, że każdy wierzchołek tkaniny ma własną sferę okalającą. Na tej podstawie da się sprawdzić i rozwiązać kolizje z innymi obiektami. Poniżej zaprezentowano fragment kodu GLSL, zawartego w kernelu *ClothCollisionKernel.glsl*, odpowiedzialnego za tę czynność.

Listing 5.3. Rozwiązywanie kolizji.

```
void Vec3LengthSquared(in vec3 vec, out float ret)
{
   ret = vec.x * vec.x + vec.y * vec.y + vec.z * vec.z;
}
void CalculateCollisionSphere(vec3 mCenter, float mRadius, vec3 sphereCenter,
   float sphereRadius, float multiplier, inout vec3 ret)
{
   vec3 diff = mCenter - sphereCenter;
   float diffLength;
   Vec3LengthSquared(diff, diffLength);
   ret = vec3(0.0f);
   if
   (
      diffLength < (mRadius + sphereRadius) * (mRadius + sphereRadius) &&
      diffLength != 0.0f
```

```
    )
    {
        diff = normalize(diff);
        diff = diff * ((mRadius + sphereRadius) - sqrt(diffLength)) * multiplier;

        ret = diff;
    }
}
void CalculateCollisionBoxAA(vec3 mCenter, float mRadius, vec3 bMin,
    vec3 bMax, float multiplier, inout vec3 ret)
{
    vec3 closest = min(max(mCenter, bMin), bMax);
    float dist;
    Vec3LengthSquared(closest - mCenter, dist);
    ret = vec3(0.0f);
    if(dist < (mRadius * mRadius) && dist != 0.0f)
    {
        closest = mCenter - closest;
        ret = normalize(closest) * (mRadius - sqrt(dist)) * multiplier;
    }
}
void main()
{
    vec3 colOffset = vec3(0.0f, 0.0f, 0.0f);
    vec3 mPos = vec3(WorldMatrix * Pos);
    vec3 totalOffset = vec3(0.0f);
    float mR = Multipliers.y;
    // rozwiazanie kolizji z prostopadloscianami
    for(int i = 0; i < BoxAAColliderCount; ++i)
    {
        mat2x4 box = baaBuffer[i];
        vec3 bMin = vec3(box[0][0], box[0][1], box[0][2]);
        vec3 bMax = vec3(box[1][0], box[1][1], box[1][2]);

        CalculateCollisionBoxAA(mPos, mR, bMin, bMax, 1.0f, colOffset);
        mPos += colOffset;
        totalOffset += colOffset;
    }
    // rozwiazanie kolizji ze sferami
```

```
        for(int i = 0; i < SphereColliderCount; ++i)
        {
           vec4 sphere = sBuffer[i];
           vec3 sPos = vec3(sphere);
           float sR = sphere.w;

           CalculateCollisionSphere(mPos, mR, sPos, sR, 1.0f, colOffset);
           mPos += colOffset;
           totalOffset += colOffset;
        }
        // rozwiazanie kolizji wewnetrznych - z sasiadami
        for (int i = 0; i < 4; ++i)
        {
           vec3 wnPos = vec3(WorldMatrix * InPosBuffer[int(Neighbours[i])]);
           CalculateCollisionSphere(mPos, mR, wnPos, mR, 0.5f, colOffset);
           mPos += colOffset;
           totalOffset += colOffset;
        }
        ...
        vec4 finalPos = vec4(vec3(Pos) + totalOffset, Pos.w);
    }
```

5.3.5. Interakcja z użytkownikiem

Istnieją dwa sposoby interakcji użytkownika z tkaniną. Poprzez przesuwanie obiektu sfery bądź prostopadłościanu, z którym wchodzi ona w kolizje, albo przy pomocy ekranu dotykowego. W pierwszym przypadku, efekty są aplikowane w trakcie rozwiązywania kolizji zewnętrznych. Dla drugiego sposobu, należy dokonać specjalnych obliczeń, by wiedzieć, które wierzchołki trzeba dodatkowo przesunąć, w którą stronę i w jakim stopniu. Dzieje się to w ramach etapu drugiego, w kernelu *ClothCollisionKernel.glsl*.

Jedynymi danymi wejściowymi są dwuwymiarowe tzw. wektory dotyku. Jeden określa miejsce na ekranie, w którym nastąpiło dotknięcie ekranu, a drugi – kierunek, w jakim przesuwa się palec. Wyrażono je w przestrzeni ekranu. Aby na ich podstawie dokonać przesunięcia wierzchołka, trzeba mieć jego wektor położenia także w tej przestrzeni. Uzyskiwany jest poprzez pomnożenie go przez macierze, kolejno świata, widoku i projekcji oraz podzielenie wyniku przez jego komponent w. W ten sposób otrzymano położenie wierzchołka w zakresie $< -1, 1 >$, takim samym jak wektor dotyku. Następnie przy pomocy wzoru

Gaussa obliczono współczynnik c określający, w jakim stopniu nastąpi przesunięcie. Jest ono wprost proporcjonalne do odległości pozycji wierzchołka od punktu dotyku:

$$c = A \exp \frac{(\mathbf{p}_{t_x} - \mathbf{p}_{i_x})^2 + (\mathbf{p}_{t_y} - \mathbf{p}_{i_y})^2}{2\sigma} \ . \tag{5.2}$$

Gdzie parametry A i σ są odgórnie określonymi stałymi i wynoszą odpowiednio: 200 i 300. Natomiast $\mathbf{p}_t$ to pozycja dotyku, a $\mathbf{p}_i$ – wierzchołka. Mając już przesunięcie, trzeba wyrazić je z powrotem w koordynatach modelu. Dokonuje się tego mnożąc przez odwrotności wspomnianych wyżej macierzy – projekcji, widoku i świata. Na koniec można po prostu dodać wektor przesunięcia do aktualnej pozycji.

Listing 5.4. Obliczenie reakcji tkaniny na dotyk ekranu.

```
vec4 mPosScreen = ProjMatrix * (ViewMatrix * (WorldMatrix * finalPos));
vec4 mPosScreenNorm = mPosScreen / mPosScreen.w;
vec4 fPosScreen = vec4(TouchVector.x, TouchVector.y, 0.0f, mPosScreenNorm.w);
vec4 fDirScreen = vec4(TouchVector.z, TouchVector.w, 0.0f, 0.0f);
float A = 200.0f;
float s = 300.0f;
float coeff = A * exp(-(
   (fPosScreen.x - mPosScreenNorm.x) *
      (fPosScreen.x - mPosScreenNorm.x) +
   (fPosScreen.y - mPosScreenNorm.y) *
      (fPosScreen.y - mPosScreenNorm.y))
   / 2.0f * s);
fDirScreen *= mPosScreen.w;
fDirScreen = inverse(WorldMatrix) *
   (inverse(ViewMatrix) *
      (inverse(ProjMatrix) * fDirScreen));
fDirScreen *=
   coeff * length(vec2(TouchVector.z, TouchVector.w)) * Multipliers.x;
finalPos.xyz += fDirScreen.xyz;
```

5.3.6. Przeliczenie wektorów normalnych

Proces obliczenia wektorów normalnych wierzchołków jest ostatnim etapem symulacji. Nie ma on wpływu na samo zachowanie tkaniny, jednak posiada kluczowe znaczenie przy jej wizualizacji. Na początku siatka przyjmuje postać prostokąta położonego równolegle

do płaszczyzny XZ układu współrzędnych, a wektory normalne są skierowane w kierunku dodatnich wartości osi Y. Jeśli nie zostaną one za każdym razem przeliczone na nowo, tkanina pozostanie oświetlona zawsze w taki sam sposób, niezależnie od jej ruchu. Poskutkuje to bardzo niesatysfakcjonującym efektem wizualnym.

Proces obliczenia wektora normalnego jest bardzo prosty. Wykorzystano tutaj własność iloczynu wektorowego, który daje nam zawsze wektor prostopadły do płaszczyzny tworzonej przez wektory wchodzące w skład działania. Obliczono iloczyn dla każdej grupy: dany wierzchołek – jego dwóch kolejnych sąsiadów. Pozwala to wziąć pod uwagę wszystkie płaszczyzny „schodzące" się w przetwarzanym wierzchołku. Wyniki uśredniono poprzez ich zsumowanie, a następnie normalizację otrzymanego wektora. Proces ten opisują poniższy wzór oraz fragment kodu GLSL:

$$\mathbf{n}_i = normalize(\sum_{n=0}^{n<8}((\mathbf{p}_i - \mathbf{p}_n) \times (\mathbf{p}_i - \mathbf{p}_{(n+1)\ MOD\ 8}))) \ . \tag{5.3}$$

Listing 5.5. Przeliczenie wektorów normalnych.

```
for(int i = 0; i < 8; ++i)
{
   int nID1 = int(roundEven(ids[i]));
   int nID2 = int(roundEven(ids[(i + 1) % 8]));

   vec3 diff1 = mPos - vec3(InPosBuffer[nID1]);
   vec3 diff2 = mPos - vec3(InPosBuffer[nID2]);

   normal = normal + (cross(diff1, diff2) *
      mpliers[i] * mpliers[(i + 1) % 8]);
}
normal = normalize(normal);
```

Rozdział 6

Wyniki testów symulatora

6.1. Czas wykonania

Czas wykonania jest rozumiany jako czas potrzebny na przetworzenie jednego pełnego kroku symulacji tkaniny. Wyrażony został w milisekundach. To najważniejsze kryterium porównawcze, gdyż mówi nam, jak bardzo obliczenia obciążają sprzęt, jak duży procent całości pracy silnika stanowią i w efekcie – czy działanie symulatora cechuje płynność.

Wpływ na czas wykonania ma ilość przetwarzanych danych, czyli gęstość siatki tkaniny, oraz wybrana implementacja. Pierwszą zależność przedstawiono w formie tabel oraz wykresów, osobno dla każdej metody i implementacji. Liczbę wierzchołków można w aplikacji łatwo modyfikować, zmieniając liczbę krawędzi poziomych i pionowych. Przyjęto zakres od siatki posiadającej 10×10 wszystkich krawędzi (100 wierzchołków) do 120×120 (14400 wierzchołków), z krokiem co 10 krawędzi poziomych i pionowych.

Warto wspomnieć, że do zachowania pełnej płynności obrazu na ekranie należy rysować jedną jego klatkę przynajmniej 30 razy na sekundę. Oznacza to, iż czas wykonania symulacji nie może być większy niż ok. 33 ms. Najbardziej satysfakcjonującym wynikiem byłoby osiągnięcie go niższego niż ok. 16 ms, co równe jest 60 klatkom na sekundę – to zazwyczaj maksymalna szybkość renderingu przy włączonej synchronizacji pionowej obrazu. Założono, że pozostałe obliczenia związane z pracą silnika symulacji są pomijalnie krótkie.

Przyjęto oznaczenia:

1. C – liczba wszystkich wierzchołków.
2. MS-GPU-A – Model masy na sprężynie, implementacja GPU, platforma Android.
3. PB-GPU-A – Model oparty na pozycji, implementacja GPU, platforma Android.
4. MS-GPU-W – Model masy na sprężynie, implementacja GPU, platforma Windows.
5. PB-GPU-W – Model oparty na pozycji, implementacja GPU, platforma Windows.
6. MS-CPU-A – Model masy na sprężynie, implementacja CPU, platforma Android.
7. PB-CPU-A – Model oparty na pozycji, implementacja CPU, platforma Android.

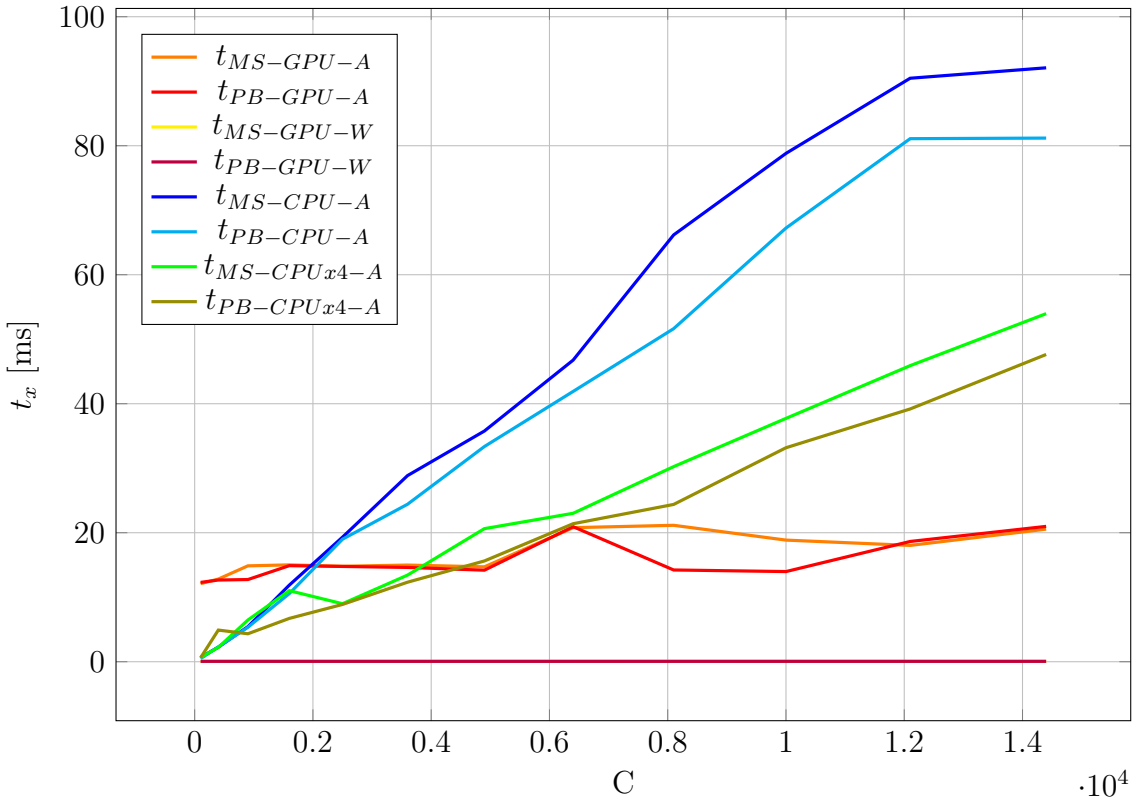
8. MS-CPUx4-A – Model masy na sprężynie, implementacja CPU (4 wątki robocze), platforma Android.
9. PB-CPUx4-A – Model oparty na pozycji, implementacja CPU (4 wątki robocze), platforma Android.

Tablica 6.1: Czas wykonania dla różnych gęstości siatki.

C	$t_{MS-GPU-A}$	$t_{PB-GPU-A}$	$t_{MS-GPU-W}$	$t_{PB-GPU-W}$
100	12.1	12.3	$6.94 \cdot 10^{-2}$	$6 \cdot 10^{-2}$
400	12.82	12.67	$6.03 \cdot 10^{-2}$	$6.05 \cdot 10^{-2}$
900	14.86	12.74	$6 \cdot 10^{-2}$	$6 \cdot 10^{-2}$
1,600	15.02	14.89	$5.97 \cdot 10^{-2}$	$6 \cdot 10^{-2}$
2,500	14.8	14.77	$6.11 \cdot 10^{-2}$	$6 \cdot 10^{-2}$
3,600	14.99	14.62	$6 \cdot 10^{-2}$	$6 \cdot 10^{-2}$
4,900	14.71	14.19	$6.48 \cdot 10^{-2}$	$6 \cdot 10^{-2}$
6,400	20.78	20.94	$6.03 \cdot 10^{-2}$	$6 \cdot 10^{-2}$
8,100	21.15	14.23	$6.14 \cdot 10^{-2}$	$6.05 \cdot 10^{-2}$
10,000	18.86	13.98	$6.05 \cdot 10^{-2}$	$6.05 \cdot 10^{-2}$
12,100	18.03	18.64	$6.03 \cdot 10^{-2}$	$6.05 \cdot 10^{-2}$
14,400	20.55	20.98	$6.05 \cdot 10^{-2}$	$6.03 \cdot 10^{-2}$

Tablica 6.2: Czas wykonania dla różnych g. siatki. (cd.)

C	$t_{MS-CPU-A}$	$t_{PB-CPU-A}$	$t_{MS-CPUx4-A}$	$t_{PB-CPUx4-A}$
100	0.66	0.49	0.64	0.61
400	2.26	2.29	2.23	4.91
900	5.43	5.34	6.47	4.33
1,600	11.87	10.55	11.02	6.71
2,500	19.31	18.99	8.99	8.88
3,600	28.86	24.42	13.43	12.33
4,900	35.77	33.36	20.63	15.62
6,400	46.76	41.9	23.01	21.39
8,100	66.17	51.64	30.25	24.39
10,000	78.81	67.24	37.73	33.18
12,100	90.46	81.09	45.9	39.19
14,400	92.08	81.18	53.97	47.64



Rysunek 6.1. Wykres zależności czasu wykonania od liczby wierzchołków. Źródło: opracowanie własne.

Wykres pokazuje dużą przewagę wydajnościową metod implementowanych na GPU. W przypadku Androida, czas obliczeń jest niemalże stały, niezależnie od liczby wierzchołków tkaniny. Drobne wahania wynikają głównie z błędu pomiaru (rzędu kilku ms). Niewielki wzrost czasu przetwarzania w końcowej fazie testów może wynikać nie tyle z samego narzutu obliczeniowego, ile z rosnącej temperatury urządzenia i związanego z tym stopniowego obniżania wydajności przez system operacyjny.

Niemożność uzyskania czasu obliczeń niższego niż ok. 12–15 ms wynika prawdopodobnie z faktu wymuszenia synchronizacji pionowej przez implementację transformacyjnego sprzężenia zwrotnego w sterowniku karty graficznej Adreno. Jak można się było spodziewać, wersję GPU na platformie PC cechuje dużo większa wydajność. W omawianym przypadku jest ona niemal 300-krotnie większa. Co ciekawe, problem z synchronizacją pionową tu nie występuje, choć czas przetwarzania także utrzymuje się na stałym poziomie.

Osobną kwestią są implementacje na CPU. Można zauważyć, iż czas przetwarzania rośnie liniowo wraz z liczbą wierzchołków i bardzo szybko osiąga wartości, które uniemożliwiają generowanie płynnego obrazu. Jedynie dla niskiej gęstości siatki uzyskano przewagę nad GPU, z racji wspomnianego wcześniej problemu. Widać także, że spadek wydajności dla

implementacji z użyciem 4 wątków roboczych jest ok. dwukrotnie mniejszy niż w przypadku podejścia sekwencyjnego.

W przypadku GPU nie zarejestrowano znaczących różnic czasu wykonania pomiędzy metodami symulacji, aczkolwiek na CPU model oparty na pozycji osiągał dla dużych liczby wierzchołków minimalnie lepsze wyniki niż jego rywal.

6.2. Stabilność

Drugim najważniejszym problemem symulacji jest jej niestabilność, rozumiana jako skłonność do wpadania siatki tkaniny w niekontrolowane drgania, co w efekcie może prowadzić do „eksplozji". Nawet jeśli się tak nie stanie, ciągłe ruchy układu skutkują nierealistycznym efektem wizualnym. Zjawisko to jest więc bardzo niepożądane i często zmusza do uruchomienia symulatora od początku.

Trudno określić, które dokładnie parametry mają wpływ na stabilność tkaniny. Z pewnością najważniejszym z nich jest sztywność – większe siły sprężystości bądź większy udział ograniczników mogą prowadzić do powstawania anomalii w procesie symulacji. Dla modelu masy na sprężynie znaczenie w redukcji drgań ma także współczynnik ich tłumienia. Nie bez wpływu pozostają też takie zmienne jak gęstość siatki, masa czy siła grawitacji.

Na potrzeby testów wybrano jeden z położonych w środku tkaniny wierzchołków oraz zbadano jego drgania w stanie spoczynku, tj. średnią różnicę pomiędzy położeniem obecnym a poprzednim, w każdym kroku symulacji. Pomiarów dokonano dla różnych współczynników sztywności, a następnie przedstawiono tę zależność w postaci tabel i wykresów. Przy każdej metodzie zostały zbadane dwa przypadki, uwzględniające inne masy, siły grawitacji, współczynniki tłumienia oraz gęstości siatki. Stan spoczynku określono jako stan, w którym tkanina opadnie swobodnie z pozycji poziomej do pionowej, zawieszonej w dwóch punktach, i przestanie się poruszać. Warto przypomnieć, że dla modelu opartego na pozycji parametr sztywności (s) został odpowiednio przeskalowany tak, by mieścił się w wymaganym zakresie [0, 1] i niósł ze sobą podobny efekt, co jego odpowiednik w modelu masy na sprężynie. Platformą testową jest mobilna wersja aplikacji, z implementacją na GPU.

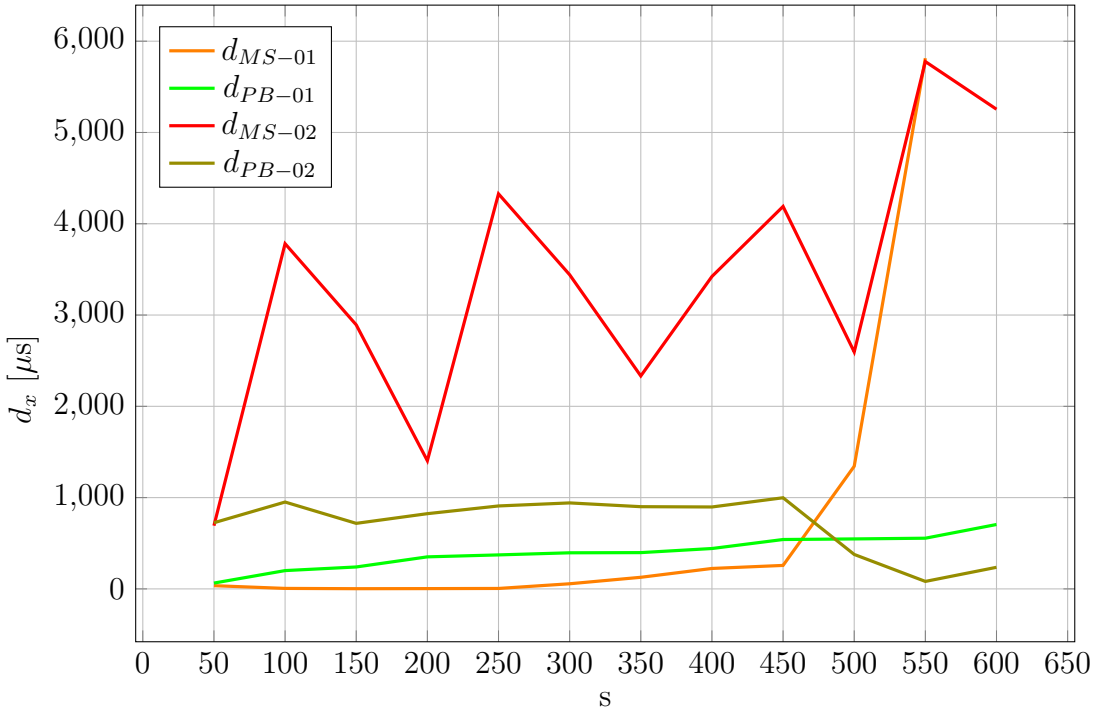
Pomiar pierwszy – sztywność: [50, 600], krok 50; masa: 0.2 kg; grawitacja: 1 $\frac{m}{s^2}$; współczynnik tłumienia: -0.5; gęstość siatki: 625 wierzchołków.

Pomiar drugi – sztywność: [50, 600], krok 50; masa: 0.7 kg; grawitacja: 2 $\frac{m}{s^2}$; współczynnik tłumienia: -10; gęstość siatki: 6400 wierzchołków.

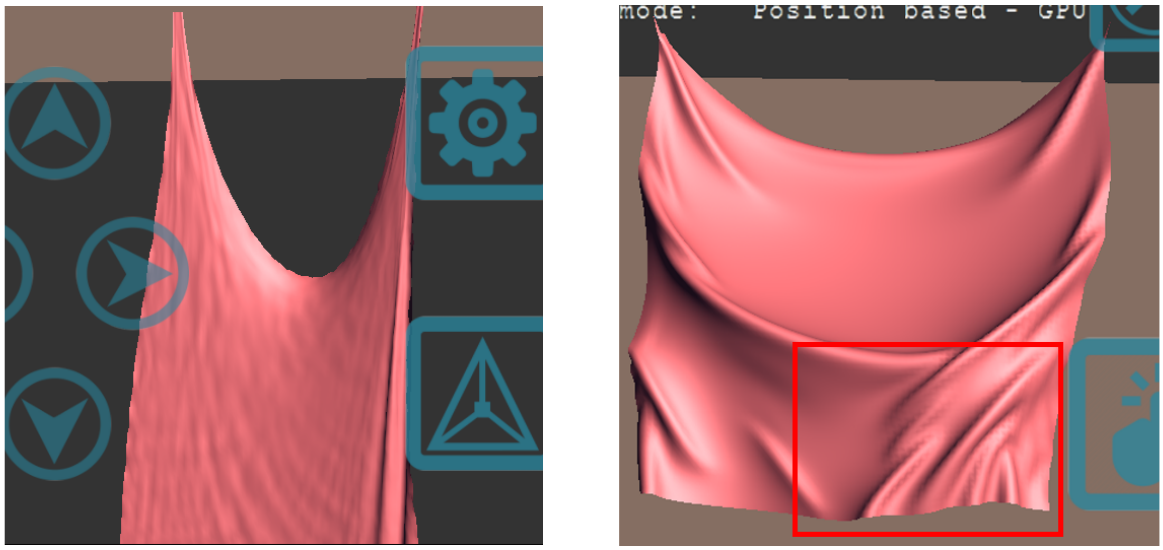
Drgania (d_x) podano w mikrometrach. Ponadto przyjęto oznaczenia: $A - n$, gdzie A – rodzaj zastosowanego modelu symulacji (MS – masy na sprężynie, PB – oparty na pozycji), n – numer pomiaru.

Tablica 6.3: Drgania dla różnych wsp. sztywności.

s	d_{MS-01}	d_{PB-01}	d_{MS-02}	d_{PB-02}
50	35.28	63.8	692.28	724.77
100	5.28	200.46	3,781.76	951.42
150	2.18	240.19	2,892.51	717.9
200	3.12	350.94	1,404.27	824.07
250	4.8	372.36	4,327.69	908.59
300	56.12	395.42	3,441.4	941.9
350	127.1	397.41	2,332.79	900.62
400	223.9	442.35	3,422.49	897.37
450	257.2	540.99	4,189.68	998.34
500	1,343.29	547.4	2,595.97	377.35
550	5,813.93	555.29	5,776.74	81.94
600	NaN	705.42	5,254.65	236.4



Rysunek 6.2. Wykres zależności drgań od współczynnika sztywności. Źródło: opracowanie własne.



Rysunek 6.3. Niestabilności występujące w obu modelach symulacji. Źródło: opracowanie własne.

Główna różnica pomiędzy modelami masy na sprężynie i opartym na pozycji ukazuje się właśnie tutaj. Widać, że w pierwszym przypadku, dla pierwszej próby, z początku zarejestrowano najniższą ze wszystkich oscylację drgań, jednak rośnie ona szybko wraz ze wzrostem parametru sztywności, dla najwyższej jego wartości doprowadzając nawet do „wybuchu" symulacji. Jeśli chodzi o drugie podejście, można zaobserwować duże oscylacje praktycznie niezależnie od elastyczności tkaniny, co pozwala wnioskować, iż zagęszczanie siatki także ma niebagatelny wpływ na drgania. Były one obecne praktycznie przez cały czas symulacji, widać je na rysunku 6.3 (po lewej). Ciągle poruszające się drobne zniekształcenia bardzo negatywnie wpływają na odbiór wizualny i w jakichkolwiek zastosowaniach praktycznych byłyby nie do zaakceptowania.

Testy udowodniły, iż model oparty na pozycji cechuje wyjątkowa stabilność – oscylacje są czasem nieznacznie większe niż u rywala, jednakże w obu próbach utrzymywały się na stałym poziomie, niezależnie od zwiększania parametru sztywności czy liczby wierzchołków. Drugi test ukazał jednak, że dla małej elastyczności i gęstej siatki, tkanina zaczyna wchodzić w niekontrolowane kolizje z samą sobą. Jest ona na tyle sztywna, by przy odpowiednim ułożeniu cząstek masy doprowadzić do „zawiśnięcia samej na sobie" i unieruchomieniu się w powietrzu, de facto ignorując siłę grawitacji. Efekt ten można zaobserwować w prawej części rysunku 6.3. Takie zachowanie tkaniny także jest nie do zaakceptowania w warunkach praktycznych, jednak należy zaznaczyć, iż nie dochodzi tu do „wybuchu" a niestabilności nie mają charakteru drobnych, szybkich drgań, a raczej niekontrolowanego falowania. Spadek

mierzonych oscylacji w przypadku współczynnika sztywności większego niż 450, w drugiej próbie, da się wytłumaczyć sytuacją widoczną na rysunku 6.3 (po prawej). Silne falowanie miało miejsce głównie w części siatki oznaczonej czerwonym prostokątem, a obszar środkowy, z którego pobrana została próbka, pozostawał we względnym spoczynku.

6.3. Efekt wizualny

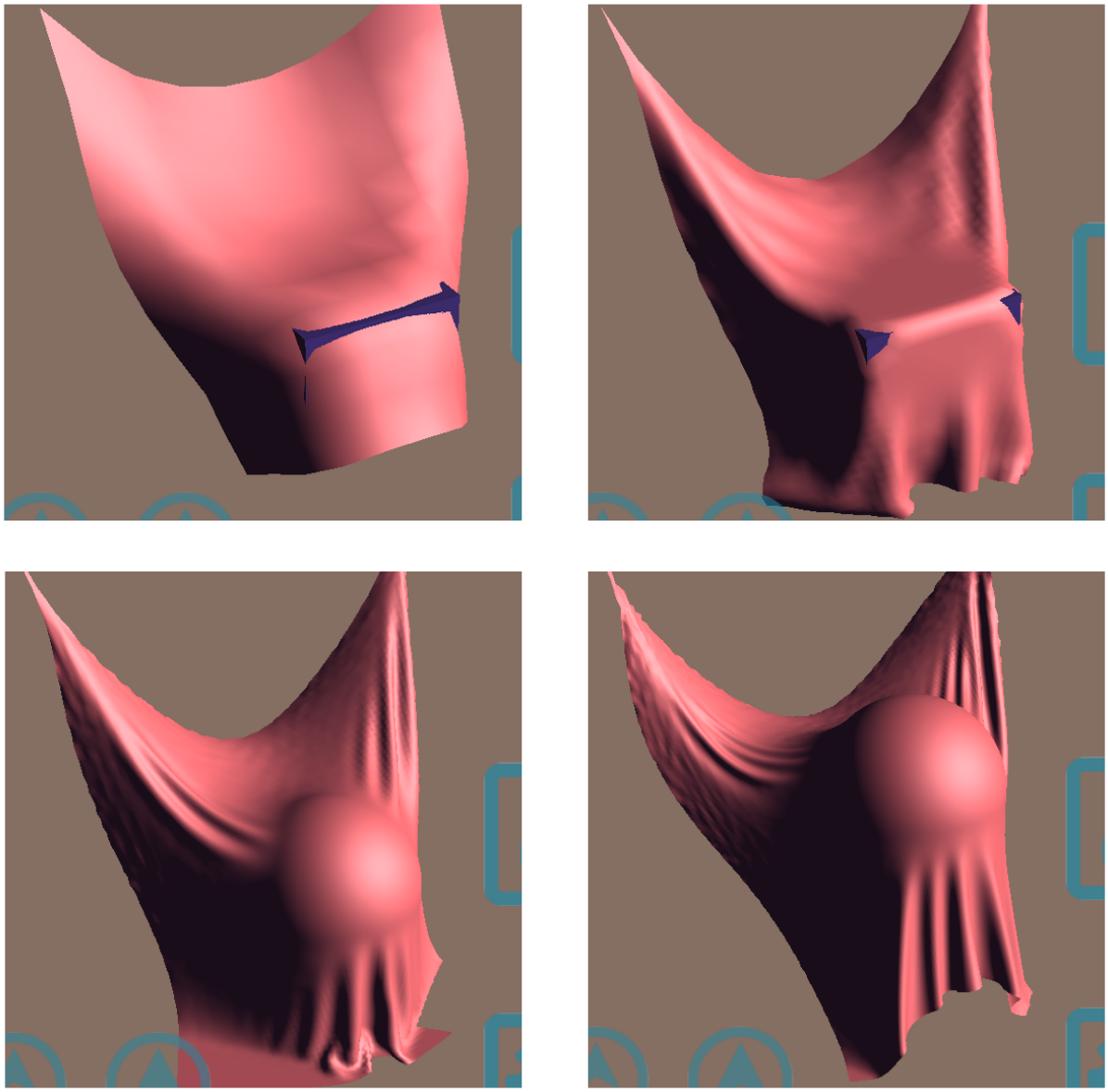
Ostatnim kryterium oceny to tzw. „efekt wizualny". Przyjęta nazwa oznacza po prostu stopień, w jakim zachowanie i wygląd symulowanej tkaniny odzwierciedla rzeczywistość. Wyznacznik ten jest całkowicie subiektywny, jednak na pewno można zauważyć wprost proporcjonalną zależność pomiędzy jakością a gęstością siatki. Mała liczba wierzchołków fizycznie nie pozwala na wygenerowanie realistycznych zmarszczek ani zagięć, tak charakterystycznych elementów animacji tkanin. Dla każdego modelu symulacji zaprezentowane zostaną zrzuty ekranu, prezentujące „efekt wizualny" z różnymi liczbami krawędzi pionowych oraz poziomych, a także innymi współczynnikami. Platformą testową jest mobilna wersja aplikacji.

Na zrzutach ekranu wchodzących w skład rysunku 6.4 przedstawiono wygląd tkaniny symulowanej modelem masy na sprężynie. Zgodnie z ruchem wskazówek zegara, począwszy od lewego górnego obrazka przyjęto następujące parametry:

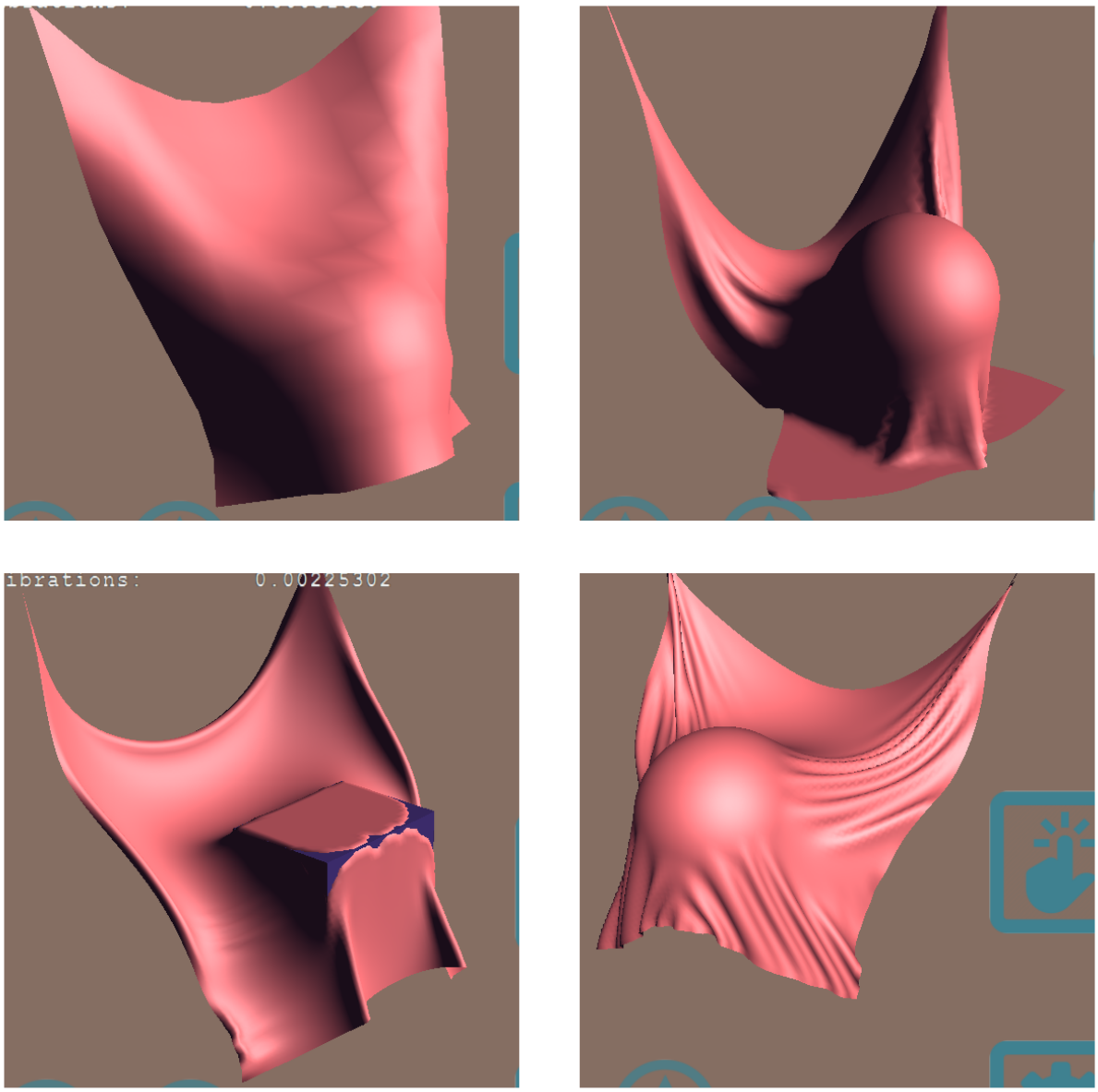
1. Siatka 10×10 krawędzi, współczynnik sztywności 200, współczynnik tłumienia -0.5, przyspieszenie grawitacyjne 5 $\frac{m}{s^2}$, masa 0.8 kg;
2. Siatka 40×40 krawędzi, współczynnik sztywności 500, współczynnik tłumienia -3.3, przyspieszenie grawitacyjne 1 $\frac{m}{s^2}$, masa 0.9 kg;
3. Siatka 80×80 krawędzi, współczynnik sztywności 500, współczynnik tłumienia -3.3, przyspieszenie grawitacyjne 0.5 $\frac{m}{s^2}$, masa 0.5 kg;
4. Siatka 120×120 krawędzi, współczynnik sztywności 700, współczynnik tłumienia -10, przyspieszenie grawitacyjne 0.5 $\frac{m}{s^2}$, masa 0.5 kg.

Natomiast na rysunku 6.5 pokazano efekt wizualny dla modelu opartego na pozycji:

1. Siatka 10×10 krawędzi, współczynnik sztywności 50, przyspieszenie grawitacyjne 5 $\frac{m}{s^2}$, masa 2 kg;
2. Siatka 40×40 krawędzi, współczynnik sztywności 100, przyspieszenie grawitacyjne 5 $\frac{m}{s^2}$, masa 0.1 kg;
3. Siatka 80×80 krawędzi, współczynnik sztywności 200, przyspieszenie grawitacyjne 5 $\frac{m}{s^2}$, masa 0.1 kg;
4. Siatka 120×120 krawędzi, współczynnik sztywności 300, przyspieszenie grawitacyjne 0.5 $\frac{m}{s^2}$, masa 0.1 kg.



Rysunek 6.4. Wygląd tkaniny dla różnych parametrów (model masy na sprężynie). Źródło: opracowanie własne.

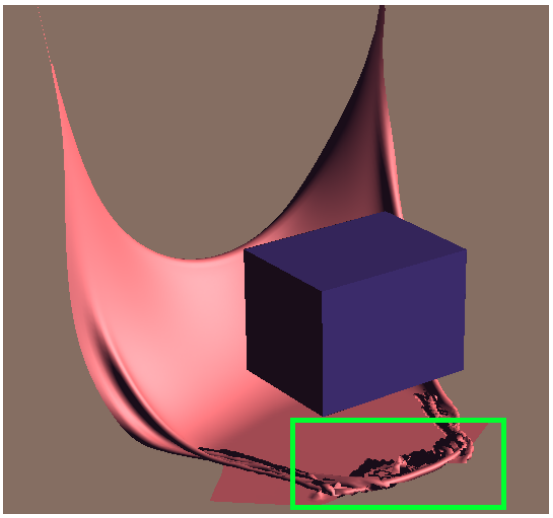


Rysunek 6.5. Wygląd tkaniny dla różnych parametrów (model oparty na pozycji). Źródło: opracowanie własne.

Jak może się wydawać z pobieżnych oględzin zrzutów ekranu, rozbieżności w wyglądzie tkaniny symulowanej różnymi modelami nie są duże. Faktem jest, iż dla małej liczby krawędzi tkanina wygląda i zachowuje się niemal tak samo. Zmarszczki i kształt ułożenia na obiekcie faktycznie wydają się wyglądać tym lepiej i bardziej realistycznie, im gęstsza jest siatka, zarówno w pierwszej, jak i w drugiej metodzie.

Podobieństwa kończą się jednak w momencie wzięcia pod uwagę parametrów, jakich użyto do uzyskania pokrewnych efektów – są zupełnie inne. Ogólnie rzecz biorąc, model oparty na pozycji generuje sztywniejszą tkaninę niż jego rywal. Czasem skutkuje to omówionymi wyżej błędami, lecz nie występują tu mikrodrgania widoczne na prawym dolnym zrzucie ekranu rysunku 6.4. Duże znaczenie ma także szybkość samej animacji tkaniny, powinna ona opadać i reagować na interakcje z poruszającymi się obiektami mniej więcej tak szybko, jak w rzeczywistości. Mimo swoich anomalii oraz trudności uzyskania sztywnego modelu, dla gęstszych siatek metoda masy na sprężynie daje w tej kwestii lepsze rezultaty. Z drugiej strony metodę opartą na pozycji cechuje dużo łatwiejsza regulacja elastyczności i większa stabilność, jednak mogą się pojawić tu problemy z ustaleniem odpowiedniej szybkości animacji. Założono stałość parametru δt przesyłanego do symulatora. W obu metodach dobranie parametrów dla uzyskania pożądanej prędkości jest tym łatwiejsze, im mniej wierzchołków posiada siatka.

W przypadku małej liczby krawędzi można zaobserwować niedokładne wykrywanie kolizji pomiędzy tkaniną a prostopadłościanem (rysunek 6.4, oba górne zrzuty ekranu). Nie jest to jednak regułą, jako że problem pojawia się też dla gęstszych siatek (rysunek 6.5, lewy dolny obrazek). Tutaj jednak winę ponosi także brak implementacji siły tarcia, co sprawia, że wierzchołki prześlizgują się po prostych ściankach obiektu, rozciągając tkaninę i tworząc większe otwory w miejscu przebicia. Dla sfery okalającej, ze względu na jej obły kształt, problemy przebicia nie występują. Wyjątkiem są szybko poruszające się obiekty, które mogą zwyczajnie przeskoczyć przez tkaninę, w jednym kroku obliczeń znajdując się przed nią, a w następnym – już za. Należałoby zastosować ciągłą metodę wykrywania kolizji, dużo bardziej skomplikowaną matematycznie, lecz usuwającą takie zjawiska.



Rysunek 6.6. Wygląd tkaniny po przeniknięciu przez prostopadłościan (model oparty na pozycji). Źródło: opracowanie własne.

Co gorsza, w przypadku prostopadłościanu ześlizgiwanie się wierzchołków może stopniowo prowadzić także do kompletnego zsunięcia się tkaniny z obiektu kolizyjnego, poprzez powolne przenikanie przez niego kolejnych punktów masy. Efekt tego widać na rysunku 6.6. Zdecydowanie sytuację poprawiłoby wprowadzenie siły tarcia bądź dokładniejszej metody detekcji kolizji, gdzie brane pod uwagę byłyby trójkąty siatki, a nie tylko same wierzchołki.

Zieloną ramką oznaczono fragment tkaniny, gdzie wystąpił błąd rozwiązywania kolizji wewnętrznych. Widać, że przeniknął on przez inną część modelu i zawinął się w drugą stronę. Stało się tak z powodu bardzo uproszczonej techniki sprawdzania tych kolizji, biorącej pod uwagę tylko cztery sąsiednie wierzchołki. Najprostszym rozwiązaniem problemu byłoby zwiększenie ich liczby, jednak wiąże się to z coraz większym kosztem obliczeniowym. Wyjściem jest także zastosowanie innej metody detekcji, o której mowa w poprzednim akapicie.

Rozdział 7

Podsumowanie i wnioski

7.1. Porównanie obu modeli symulacji

Porównania omówionych w niniejszej pracy modeli symulacji tkanin należy dokonać z uwzględnieniem wielu różnych czynników, tak aby na koniec móc jasno określić, który nadaje się lepiej do zastosowań praktycznych.

Aby móc zaimplementować którąkolwiek z metod, programista musi wpierw dobrze zrozumieć jej działanie. Podstawy teoretyczne modelu masy na sprężynie są dużo prostsze, jako że oparto go o łatwy do wyobrażenia system punktów masy połączonych sprężynami, których parametry, takie jak np. współczynnik sprężystości, wpływają na zachowanie tkaniny. Całość działa, wykorzystując znane z podstaw fizyki prawo Hooke'a. Sprawia to, że nawet początkującemu programiście, nieobeznanemu z meandrami matematyki, łatwo przyjdzie pojęcie i przedstawienie sobie wizualnie tej metody. Z kolei model oparty na pozycji wykorzystuje do swoich obliczeń dużo bardziej skomplikowane pojęcia i może nastręczyć takiej osobie niemałych trudności.

Sytuacja wygląda inaczej, jeśli weźmie się pod uwagę poziom trudności implementacji. Tworzenie omawianej aplikacji wykazało, że jest on niemal identyczny dla obu modeli symulacji. Każdy z nich da się zaprogramować tak, aby korzystał z tych samych danych oraz funkcji, różniąc się jedynie obliczeniami wykonywanymi przy konkretnych sprężynach bądź ogranicznikach. Obie techniki podczas swojego przetwarzania wykorzystują różnicę pomiędzy odległością spoczynkową a aktualną między wierzchołkami. Niebotyczny wpływ na taki stan rzeczy miała na pewno decyzja o nieużywaniu ograniczników zginania w metodzie opartej na pozycji. Zaimplementowanie wszystkich rozwiązań opisanych w [2] na pewno zmieniłoby stan rzeczy na korzyść modelu masy na sprężynie. Należy jednak zaznaczyć, że system ograniczników zapewnia dużo większy wachlarz zastosowań, umożliwiając wykorzystanie ich nie tylko do obliczeń ruchu, ale też i np. rozwiązywania kolizji. Dołożono starań, by wyodrębnić jak największą część wspólną obliczeń dla obu modeli i dzięki temu do detekcji

kolizji, reakcji na sygnał od użytkownika oraz przeliczenia normalnych używane są te same instrukcje, niezależnie od wybranego modelu symulacji.

W najważniejszej kwestii, czyli wydajności, okazuje się, iż obie metody także wykazują się podobnymi rezultatami, z bardzo niewielkim zwycięstwem modelu opartego na pozycji w implementacjach CPU. Nie można było dokładnie zbadać różnic w przypadku GPU, jako że objętość buforów jednorodnych nie pozwoliła na wygenerowanie tkaniny o tak dużej liczbie wierzchołków, by czas wykonania wzrósł powyżej 20 ms. Biorąc pod uwagę podobny poziom skomplikowania samego kodu, należy przypuszczać, że także byłaby ona niewielka. Podczas testów zauważono, iż znaczną część czasu obliczeń zajmuje część algorytmu odpowiedzialna za rozwiązywanie kolizji. Może to wynikać z faktu użycia instrukcji warunkowych w kodzie wykonywanym na GPU. Większa liczba obiektów w scenie wiązałaby się na pewno z koniecznością głębszej optymalizacji tego zagadnienia, na przykład ograniczając liczbę potencjalnych encji mogących wejść w kolizje z tkaniną jeszcze na poziomie CPU.

Oba wykorzystane modele symulacji charakteryzuje pewna, zależna od parametrów, niestabilność, jednak jest ona dużo większa w przypadku modelu masy na sprężynie. Niewątpliwe zaletą okazuje się fakt, iż w skład wzoru, na podstawie którego obliczane są siły działające na wierzchołek wchodzi komponent odpowiedzialny za tłumienie drgań, a użytkownik może go regulować. Tę metodę charakteryzuje wzrost oscylacji siatki wraz ze wzrostem współczynnika elastyczności. Mają one postać małych, acz szybkich wibracji na całej powierzchni tkaniny. Z kolei dla dużych liczb krawędzi potrzeba dużej sztywności, aby zachować odpowiedni kształt, co jeszcze bardziej powiększa problem. Duże drgania z pozoru sprawiają wrażenie, że utrzymują się na stałym poziomie, jednak przy jakiejkolwiek nagłej zmianie położenia wierzchołków, np. przy kolizji, mogą doprowadzić do nagłego „wybuchu" symulacji, co jest w praktyce niedopuszczalne. W przypadku modelu masy na sprężynie także zaobserwowano zależność pomiędzy wzrostem gęstości siatki, elastyczności a utratą stabilności. Jednakże ta metoda w każdym momencie i tak jest stabilniejsza niż jej rywalka, co pokazały wyniki zaprezentowane w rozdziale 6.2. Drgania są tutaj dużo wolniejsze i mają postać delikatnego, niekontrolowanego falowania, co dużo mniej zwraca uwagę użytkownika. Dużym plusem jest brak występowania efektu „eksplozji", niezależnie od ustawionych parametrów. Efekt ten udało się uzyskać poprzez pewnego rodzaju „trik" implementacyjny – pozycja wierzchołka przesyłana do funkcji obliczającej ogranicznik jest aktualizowana na bieżąco tylko w ramach sąsiadów położonych w jednej linii. Minusem modelu opartego na pozycji w niniejszej implementacji jest tendencja do blokowania się tkaniny samej na sobie, przy dużych współczynnikach elastyczności.

Obie metody symulacji tkaniny generują pożądany efekt graficzny, czyli realistyczne zagięcia i zmarszczki tkaniny oraz jej charakterystyczne ułożenie na obiekcie. Ich jakość

jest minimalnie lepsza dla modelu opartego na pozycji, m. in. z racji nie występowania tam drobnych drgań oraz większej responsywności na zmiany współczynnika sztywności. Należy zauważyć, że np. na potrzeby gier w wielu przypadkach nie ma konieczności szczegółowo odwzorowywać detale tkanin, te można uzyskać przy pomocy map normalnych. Wystarcza przybliżona symulacja ruchu tkaniny. Dwie omawiane metody cechuje wierne odwzorowanie tego aspektu nawet dla małej liczby krawędzi. Z kolei gdy wziąć pod uwagę gęste siatki, nasuwa się problem związany z szybkością animacji. Duża liczba wierzchołków wymaga odpowiedniego współczynnika elastyczności, a ten spowalnia przesuwanie tkaniny. Widać to szczególnie w modelu opartym na pozycji. Rozwiązaniem mogłoby być bardziej dokładne dopasowanie współczynników bądź zwiększenie parametru δt, niezmiennego w omawianej symulacji. Poprawę sytuacji prawdopodobnie można też uzyskać poprzez ustawianie innych parametrów sztywności dla każdej z grup sprężyn, bądź ograniczników (tj. równoległych do krawędzi tkaniny, leżących po przekątnej oraz takich jak pierwsze, lecz położonych o jedną pozycję dalej). Dużym minusem w kwestii efektu wizualnego okazała się zastosowana metoda detekcji kolizji. Nie rozwiązuje ona w satysfakcjonujący sposób kolizji wewnętrznych i często występują błędy kolizji zewnętrznych, np. w przypadku opadnięcia tkaniny na prostopadłościan. Aby naprawić problem, należałoby zaimplementować inną technikę. Jednak na pewno wiązałby się z tym ubytek w wydajności i tak najcięższego obliczeniowo komponentu symulacji.

Biorąc pod uwagę wszystkie wymienione powyżej czynniki, można uznać iż do zastosowań praktycznych model oparty na pozycji nadaje się bardziej. Lepiej radzi on sobie z dużymi liczbami wierzchołków, cechuje go względna stabilność i jest minimalnie wydajniejszy. Niezależnie od pożądanej jakości wizualizacji, w większości przypadków parametry da się ustalić tak, by uzyskać dobry efekt graficzny. Nie znaczy to jednak, że w obecnej implementacji jest on wolny od wad, a różnice między metodami zanikają wraz ze spadkiem gęstości siatki – wtedy i model oparty na pozycji, i masy na sprężynie cechują bardzo podobny wygląd oraz zachowanie.

7.2. Porównanie implementacji CPU i GPU smartfona

Kolejnym aspektem niniejszej pracy było wykazanie wyższości zastosowania kart graficznych urządzeń mobilnych nad zastosowaniem procesorów tychże w dziedzinie symulacji fizycznej tkanin. Testy zaprezentowane w rozdziale 6 jasno wskazują zwycięzcę tego porównania pod względem wydajności. GPU są wielokrotnie szybsze od CPU podczas obliczeń zagadnień mogących być przetwarzanymi równolegle, a takim właśnie jest omawiany problem. Rozłożenie pracy nad wierzchołkami siatki tkaniny na poszczególne jednostki przetwarzające GPU to wręcz intuicyjne, a zarazem skuteczne i wydajne rozwiązanie pomimo

występującej tu redundancji. Zarejestrowana szybkość działania okazuje się być taka sama, niezależnie od ilości danych, czego nie można powiedzieć o implementacjach CPU, gdzie maleje ona liniowo. Podział na wątki robocze zwiększa ją dwukrotnie, co trochę poprawia sytuację, jednak w przypadku szczegółowych tkanin i tak wydajność jest za niska. Z kolei na GPU zaobserwowano blokowanie liczby renderowanych klatek na sekundę do wartości zgodnej z odświeżaniem ekranu przez transformacyjne sprzężenie zwrotne. Jest to wada, nie pozwalająca w pełni ocenić wydajności i blokująca prędkość działania aplikacji, jeśli nie dba się o występowanie efektu „tearingu", a chce uzyskać jak najszybsze działanie. Problem być może rozwiązałaby zmiana testowego sprzętu na inny, bądź wykorzystanie innego API do obliczeń GPGPU.

Wszystko to nie zmienia jednak faktu, iż implementacja CPU także ma swoje zastosowania i zalety. Użycie jej jest konieczne w sytuacji, gdy urządzenie nie obsługuje wersji OpenGL ES 3.0, ani żadnej z specjalistycznych API, jak OpenCL. Możliwe, że uzyskałaby ona przewagę wydajnościową w sytuacji, gdy platforma testowa dysponowałaby GPU z bardzo niskiej półki. Da się ją także z pewnością zastosować, gdy zbiorem danych jest jedynie bardzo niewielka liczba wierzchołków, bądź jeśli wybrano animację tylko w przestrzeni 2D. Należy także zaznaczyć, że symulację tkanin dużo łatwiej zaimplementować na CPU, gdyż nie wymaga to głębszej znajomości API graficznego ani żadnych innych oraz tworzenia dość złożonego sterowania buforami, zmiennymi jednorodnymi, programami i transformacyjnym sprzężeniem zwrotnym.

7.3. Porównanie implementacji GPU smartfona i GPU PC

W rozdziale 3 przewidywano, że wydajność urządzeń mobilnych w omawianym zagadnieniu będzie wielokrotnie niższa, niż komputerów klasy PC. Stworzenie dwóch wersji aplikacji, jednej na platformę Android, drugiej – na platformę Windows pozwoliło potwierdzić to przypuszczenie. Różnica szybkości działania jest ok. 300-krotna, w dodatku na PC znika problem dotyczący blokady liczby renderowanych klatek na sekundę. Pozostaje więc na tej podstawie odpowiedzieć na fundamentalne pytanie: czy implementacja symulacji tkanin na urządzeniach mobilnych ma w ogóle sens, jeśli do dyspozycji są dużo szybsze komputery stacjonarne? Okazuje się, że tak, aczkolwiek na trochę mniejszą skalę. O ile smartfon z użyciem GPU może bezproblemowo animować tkaninę o naprawdę gęstej siatce, wystarczającej do odwzorowania większości szczegółów, to tę platformę dręczy kilka istotnych problemów.

Pierwszym jest wspominany już wielokrotnie brak buforów teksturowych na części urządzeń, co ogranicza maksymalną możliwą do uzyskania jakość. Pamiętać należy także o ogólnie niższej ilości pamięci karty graficznej, wprowadzającej kolejne limity w tej kwestii.

Symulacja tkanin mimo wszystko bardzo mocno wykorzystuje tu możliwości sprzętowe, doprowadzając do przegrzewania się urządzenia. Prowadzi to do redukcji wydajności przez system operacyjny, co z kolei skutkuje znacznym wydłużeniem przetwarzania i miało wpływ na wyniki testów w rozdziale 6.1. Tworząc aplikacje, które cechuje znaczne zapotrzebowanie na moc obliczeniową należy o tym fakcie pamiętać. To, że program początkowo działa bez zarzutu nie znaczy, iż za parę minut nie może zwolnić, gdy temperatura osiągnie wysokie wartości. Intensywna eksploatacja zasobów sprzętowych skutkuje także szybkim zużyciem baterii, co nie pozostaje bez znaczenia dla użytkownika końcowego i powinno być istotne podczas projektowania aplikacji.

Trzeba także wspomnieć o różnicach w API dostępnych na obu platformach. W przypadku PC, poczyniono ostatnimi czasy duże kroki w celu udostępnienia programistom kart graficznych jako urządzeń do obliczeń ogólnego przeznaczenia. Jest tu dostępne wiodące w tej dziedzinie API, czyli CUDA, cechujące się bardzo prostą obsługą i dużą elastycznością. Ponadto także twórcy oprogramowania graficznego starali się udostępnić te funkcje, wprowadzając Compute Shadery. Implementacja symulacji tkanin z ich użyciem byłaby zdecydowanie łatwiejsza. Należy pamiętać, że wsparcie dla GPGPU na urządzeniach mobilnych nie jest tak zaawansowane, jak na PC i w efekcie programista musi się poruszać w oprogramowaniu starszej o kilka lat generacji. Mimo to symulację można tu stworzyć, choć nastręcza to więcej trudności.

Bibliografia

[1] Hanwen Li, Yi Wan, Guanghui Ma, *A CPU-GPU Hybrid Computing Framework for Real-time Clothing Animation*, School of Information Science and Engineering, Lanzhou University, Lanzhou, China, 2011

[2] Matthias Müller, Bruno Heidelberger, Marcus Hennix, John Ratcliff, *Position Based Dynamics*, 3rd Workshop in Virtual Reality Interactions and Physical Simulation "VRIPHYS",

[3] Shane Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, Elsevier Inc., 2013

[4] Louis Li-Fang Chang, Damon Shing-Min Liu, *Deformable Object Simulation in Virtual Environment*, Association for Computing Machinery, Inc., 2006

[5] Bart Kevelham, Nadia Magnenat-Thalmann, *Virtual Try On: An application in need of GPU optimization*, ATIP/ACRC HPC Workshop, 2012

[6] Mark Segal, Kurt Akeley, *The OpenGL Graphics System: A Specification (Version 4.5 (Core Profile) – May 28, 2015)*, The Kronos Group Inc., 2015

[7] Muhammad Mobeen Movania, *OpenGL Receptury dla programisty*, Helion, 2015

[8] *Circle-Circle Collision Tutorial* http://ericleong.me/research/circle-circle/, stan na dzień 04.01.2016

[9] *Static Sphere vs AABB* http://blog.nuclex-games.com/tutorials/collision-detection/static-sphere-vs-aabb/, stan na dzień 04.01.2016

[10] *Graphics processing unit*, https://en.wikipedia.org/wiki/Graphics_processing_unit, stan na dzień 04.01.2016

[11] *Branch prediction*, https://en.wikipedia.org/wiki/Branch_predication, stan na dzień 05.04.2016

[12] *SIMD* https://en.wikipedia.org/wiki/SIMD, stan na dzień 23.01.2016

[13] *FLOPS* https://en.wikipedia.org/wiki/FLOPS, stan na dzień 23.01.2016

[14] *Geekbench 3 Results – Geekbench Browser*, http://browser.primatelabs.com/geekbench3/, stan na dzień 08.01.2016

[15] *Qualcomm Adreno 320 vs Nvidia GeForce GTX 750*, http://versus.com/en/qualcomm-adreno-320-vs-nvidia-geforce-gtx-750, stan na dzień 08.01.2016

[16] *LG Nexus 4 Technical Specifications*, http://www.lg.com/uk/mobile-phones/lg-E960-nexus-4-by-lg, stan na dzień 08.01.2016

[17] *Adreno*, https://en.wikipedia.org/wiki/Adreno, stan na dzień 08.01.2016

[18] *GeForce 700 Series*, https://en.wikipedia.org/wiki/GeForce_700_series, stan na dzień 08.01.2016

[19] *GeForce 900 Series*, https://en.wikipedia.org/wiki/GeForce_900_series, stan na dzień 08.01.2016

[20] *OpenGL Wiki*, https://www.opengl.org/wiki/, stan na dzień 10.01.2016

[21] *Uniform Buffers VS Texture Buffers*, http://rastergrid.com/blog/2010/01/uniform-buffers-vs-texture-buffers/, stan na dzień 15.01.2016

Abstract

The realistic simulation of cloths is nowadays a key to produce good-quality, authentic graphical visualizations of various fabrics, such as characters' garment elements, flags or curtains. This can be computationally expensive, more and more as number of particles, which fabric is divided into, increases. The solution to this matter was to use GPU – *Graphic Processing Unit* and perform all calculations on this device. On PC platform, this technique proved to be much faster than the standard CPU approach.

The main purpose of this work is to check whether this solution could also be introduced on the mobile devices. Most of them nowadays also have their own specialized GPU chips, but will they prove to be computationally faster than mobile CPUs? Is it possible and worth one's while to create visually appealing and efficient cloth simulation here? And how big is the difference between PC and mobile platform in GPGPU performance? This paper answers these questions.

A test application was created, one of its main purposes being the visualisation of two selected simulation methods – mass–spring and position–based model. It was equally important to show cloth's collisions between other objects in scene and itself. The user is allowed to set various parameters that influence the simulation, such as the aforementioned method type, mesh density and dimensions or elasticity coefficient. He can also impact the movement of the cloth, swiping his finger along the device's touch screen, which is something unique to the mobile platform. To fully measure every important factor of the simulation, its three implementations were created – one using GPU for computing and the other two using GPU, in sequential and multi-threaded approach. To have a comparison between mobile and PC platform, a PC version of the application was created, both similar and sharing as much code with each other as possible.

In chapter 2 of this paper there are described theoretical basis of incorporated simulation methods and collision resolving. General Purpose GPU Computing is also mentioned, along with GPU framework and a comparison between it and a CPU is made, in the matter of architecture and performance. Chapter 3 analyses abilities of mobile devices, also mentioning their unique UI capabilities after comparing test device – LG Nexus 4 – to an example PC.

All used APIs, libraries and most important functions are also described. Among them are OpenGL ES 3.0, OpenGL 3.3 and Android NDK, as test application is completely written in C++. Chapter 4 shows architecture of its engine, magnifying characteristics of the most important components. Algorithms of the engine's operation are also given, including such actions as updating scene's entities, communication with Android OS, rendering or UI. In Chapter 5 attention is turned to the cloth simulator only, describing its general work flow, divided into three stages – particle movement computation, collision solving and recalculation of normal vectors. Every stage is then thoroughly described, with their most important fragments's code given. Finally, in chapters 6 and 7, results of test application's work are shown, in the matter of both simulation models' computation time, stability and visual appeal, considering all three implementations and two tested platforms. In the end it is proved that the cloth simulation can be implemented on mobile devices and the average one's GPU can perform very well, producing smooth animation of fabric's dense mesh, but not without a few important limitations. These include less useful API functions, shorter work time on battery as a result of intensive computations and tendency to overheating.

Spis listingów

Spis rysunków

Spis tablic

Lista Algorytmów

Łódź, dn. .....…….

OŚWIADCZENIE

Marcin Wawrzonowski
(Imię i nazwisko studenta)

ul. 1 Maja 17/56, 99-300 Kutno
(Adres)

180729
(Nr albumu)

Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej
(Jednostka organizacyjna prowadząca studia)

Informatyka, specjalność Technologie Gier i Symulacji Komputerowych
(Kierunek studiów)

Studia inżynierskie, stacjonarne
(Poziom kształcenia i forma studiów)

Oświadczam, że poinformowano mnie o zasadach dotyczących kontroli oryginalności pracy dyplomowej w systemie antyplagiatowym.

Wyrażam zgodę na przetwarzanie*) mojej pracy dyplomowej, a także na przechowywanie jej w celu realizowania procedury antyplagiatowej w bazie danych systemu antyplagiatowego.

…….....
(Podpis studenta)

*) Przez przetwarzanie pracy rozumie się porównywanie przez system antyplagiatowy jej treści z innymi dokumentami (w celu ustalenia istnienia nieuprawnionych zapożyczeń), generowanie Raportu Podobieństwa oraz przechowywanie pracy w bazie danych systemu antyplagiatowego.

Łódź, dn. .....……

OŚWIADCZENIE

o samodzielności wykonania i oryginalności pracy dyplomowej

Marcin Wawrzonowski
(Imię i nazwisko studenta)

ul. 1 Maja 17/56, 99-300 Kutno
(Adres)

180729
(Nr albumu)

Wydział Fizyki Technicznej, Informatyki
i Matematyki Stosowanej
(Jednostka organizacyjna prowadząca studia)

Informatyka, specjalność Technologie Gier
i Symulacji Komputerowych
(Kierunek studiów)

Studia inżynierskie, stacjonarne
(Poziom kształcenia i forma studiów)

Świadomy/a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że przedkładana praca dyplomowa inżynierska na temat:

„Wykorzystanie GPU urządzeń mobilnych w symulacji dynamiki tkanin”

została wykonana przeze mnie samodzielnie.

Jednocześnie oświadczam, że ww. praca:

- nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2006 r. Nr 90, poz. 631, z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym, a także nie zawiera danych i informacji, które uzyskałem/am w sposób niedozwolony,
- nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów wyższej uczelni lub tytułów zawodowych.

…….....
(Podpis studenta)

Łódź, dn. .....……

OŚWIADCZENIE

o zgodności wersji elektronicznej pracy dyplomowej z przedstawionym wydrukiem komputerowym

Marcin Wawrzonowski
(Imię i nazwisko studenta)

ul. 1 Maja 17/56, 99-300 Kutno
(Adres)

180729
(Nr albumu)

Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej
(Jednostka organizacyjna prowadząca studia)

Informatyka, specjalność Technologie Gier i Symulacji Komputerowych
(Kierunek studiów)

Studia inżynierskie, stacjonarne
(Poziom kształcenia i forma studiów)

Świadomy odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że przedkładana na nośniku elektronicznym praca dyplomowa inżynierska na temat:

„Wykorzystanie GPU urządzeń mobilnych w symulacji dynamiki tkanin”

zawiera te same treści, co oceniany przez promotora i recenzenta wydruk komputerowy.

Jednocześnie oświadczam, że jest mi znany przepis art. 233 § 1 Kodeksu karnego określający odpowiedzialność za składanie fałszywych zeznań.

…….....
(Podpis studenta)