Uniwersytet Wrocławski

Praca magisterska

# Metody interaktywnej symulacji fizyczej w środowisku jednolitym i rozproszonym

*Autor:*
Marcin Pawłowski

*Promotor:*
dr Andrzej Łukaszewski

17 maja 2015

# Streszczenie

Celem pracy jest omówienie, implementacja oraz porównanie metod modelowania interaktywnych symulacji fizycznych w czasie rzeczywistym. Symulacje takie znajdują zastosowanie głównie w grach komputerowych, programach szkoleniowych i edukacyjnych, animacji komputerowej i robotyce. Praca może służyć za punkt wyjścia dla osób pragnących zaimplementować własną symulację lub zrozumieć schemat działania istniejącej implementacji.

Symulowane środowisko składa się z obiektów reprezentowanych przez bryły sztywne i działających na nie sił i ograniczeń. Interaktywność symulacji polega na możliwości modyfikacji sił działających na obiekty w czasie rzeczywistym. Celem symulacji jest odwzorowanie w realistyczny sposób świata rzeczywistego w świecie wirtualnym, z którym użytkownik może wchodzić w interakcję. Aby ten cel osiągnąć należy uwzględnić w niej nie tylko reakcje obiektów na siły i związany z tym ruch ale także wzajemny wpływ obiektów na siebie poprzez kolizje. Wiąże się to nie tylko z reprezentacją kształtów obiektów ale też z koniecznością rozwiązania problemów takich jak efektywne wykrywanie zaistniałych kolizji w czasie rzeczywistym czy dokładne wyznaczenie punktu kolizji – co często okazuje się bardzo trudne obliczeniowo. Obiekty mogą nie tylko odbijać się od siebie ale także swobodnie na sobie spoczywać lub trzeć o siebie. Zapewnienie, by takie obiekty wzajemnie nie mogły się przenikać stanowi osobny problem. Przeniesienie symulacji fizycznych do środowiska rozproszonego, takiego jak sieciowe gry komputerowe lub inne wirtualne rzeczywistości stanowi dodatkowe wyzwanie ze względu na konieczność synchronizacji symulacji pomiędzy odległymi użytkownikami. Trudno zapewnić wszystkim użytkownikom wrażenie spójności przedstawionej rzeczywistości w sytuacji gdy informacje o ich wpływie na nią często potrzebują setek milisekund by rozpropagować się w sieci.

# Spis treści

# Wstęp

Najpowszechniejszymi zastosowaniami symulacji fizycznych w skali makro są w dzisiejszych czasach gry komputerowe i programy symulacyjne, jak na przykład szkoleniowe symulatory lotu. W obu przypadkach przebieg symulacji zależy nie tylko od jej warunków początkowych ale też od czynnika ludzkiego w postaci osoby biorącej udział w symulacji lub grającej w grę. Wynika to z ich interaktywności - użytkownik może wpływać na symulację na z góry ustalonych zasadach, najczęściej poprzez sterowanie jednym z obiektów składających się na symulację.

Istotne jest aby taka symulacja z punktu widzenia użytkownika zachowywała się w sposób analogiczny do świata rzeczywistego. Do osiągnięcie tego celu potrzebne jest zrealizowanie wielu współdziałających ze sobą elementów symulatora. Do realizacji każdego z nich zwykle można użyć przynajmniej kilku różnych metod, których wybór może mieć znaczący wpływ na stopień w jakim symulacja przypomina rzeczywistość a także na złożoność obliczeniową programu. Często szukanie optymalnej metody rozwiązania problemu polegać będzie na poszukiwaniu złotego środka pozwalającego na prowadzenie symulacji o zadowalającym poziomie realizmu przy minimalnym nakładzie obliczeń. Konieczność poszukiwania rozwiązań o niskiej złożoności obliczeniowej wynika bezpośrednio z interaktywności symulacji - program powinien działać w czasie rzeczywistym by zapewnić użytkownikowi wrażenie zachowania płynności symulowanego środowiska.

Podstawowe elementy symulacji interaktywnej to kinematyka i dynamika brył sztywnych, odbicia brył sztywnych wraz z tarciem i spoczynek brył. Dodatkowym problemem dotyczącym prowadzenia symulacji w wieloużytkownikowym środowisku rozproszonym jest kwestia synchronizacji lokalnych kopii wirtualnego świata. Problem ten jest szczególnie ważny dla wielograczowych gier komputerowych korzystających z sieci.

Pierwszy rozdział poświęcony jest realizacji ruchu brył sztywnych i ich reakcji na siły, co wymaga zaimplementowania reprezentacji brył sztywnych w tym ich kształtu, pozycji, prędkości i innych właściwości fizycznych potrzebnych do przeprowadzenia realistycznej symulacji. Potrzebna jest także metoda numeryczna rozwiązywania równań różniczkowych umożliwiająca określanie zmian w położeniu, prędkości czy rotacji brył w miarę rozwoju symulacji.

Drugi rozdział skupia się na opisaniu metod potrzebnych do implementacji realistycznych odbić zachodzących pomiędzy bryłami pozwalających im wchodzić ze sobą w interakcję w ramach symulacji. Podstawowym elementem jest system wykrywania kolizji. Istnieje wiele metod wykrywania kolizji pomiędzy bryłami sztywnymi o różnej dokładności i złożoności obliczeniowej. Wybór właściwej metody często zależy od przeznaczenia symulacji. Wiedząc już, że pomiędzy parą brył zaszła kolizja można przystąpić do obliczania sił, które zadziałają na obiekty w chwili zderzenia i odepchną je od siebie. Wymaga to znajomości ich masy, położenia, kształtu, rotacji, prędkości, przyspieszenia i punktu zderzenia. Wszystkie z tych właściwości są z góry znane przed wykryciem kolizji, poza punktem zderzenia. Dokładne wyznaczenie takiego punktu jest problematyczne ze względu na dyskretny charakter symulacji i fakt, ze kolizje zwykle będą występować pomiędzy dwiema klatkami animacji. Jednak poprawne przybliżenie punktu jest niezmiernie ważne, gdyż w przeciwnym razie symulacja może zachowywać się w sposób nieintuicyjny i sprzeczny z zasadami fizyki.

Rozdział trzeci traktuje o osobnym zagadnieniu jakim jest problem swobodnego spoczynku brył. Gdy obiekty leżą na sobie ich względne prędkości są bliskie zeru i klasyczne metody obliczania sił odbicia nie sprawdzają się. Zamiast tego można skonstruować układ równań liniowych zapewniających, że bryły sztywne nie będą się wzajemnie przenikać. Potrzebne są więc metody numeryczne rozwiązywania układów równań liniowych — działające możliwie najszybciej i dające możliwie najlepsze przybliżenia rozwiązania dokładnego. Alternatywnie można zmodyfikować metody z rozdziału drugiego tak, by zapobiegały one przenikaniu się brył w warunkach spoczynku.

Czwarty rozdział dotyczy symulacji rozproszonej gdzie potrzebne są metody niwelujące efekty opóźnień w komunikacji pomiędzy użytkownikami. Istnieje wiele podejść do tego problemu, jednak najpopularniejszym, i tym, do którego implementacji ograniczę się w niniejszej pracy jest model klient-serwer gdzie oryginalna symulacja prowadzona jest przez serwer a użytkownicy-klienci starają się poprzez

krótkoterminowe przewidywanie rozwoju symulacji i korygowanie jej stanu po otrzymaniu informacji z serwera utrzymywać swoją lokalną kopię środowiska w stanie zsynchronizowanym z oryginałem.

Ostatni, piąty rozdział opisuje dołączoną do pracy przykładową implementację rozproszonej symulacji fizycznej wykorzystującej metody opisane poniżej.

# Rozdział 1

# Kinematyka i dynamika

Kinematyka określa sposób w jaki obiekt fizyczny porusza się w przestrzeni bez uwzględnienia jego właściwości i działających na niego sił wywołujących ten ruch. Dynamika natomiast opisuje wpływ zewnętrznych sił na ruch obiektów. Celem poniższego rozdziału jest opis i implementacja symulacji ruchu brył sztywnych pod wpływem działających na nie sił.

## 1.1 Ruch punktu w przestrzeni

Podstawowym pojęciem w kontekście symulacji fizycznej jest pojęcie ruchu. Miarą ruchu jest prędkość — pierwsza pochodna pozycji po czasie. Zarówno prędkość jak i pozycję obiektów wyrażamy za pomocą trójwymiarowych wektorów liczb rzeczywistych.

Wektor pozycji określa położenie obiektu w trójwymiarowej przestrzeni względem początku układu współrzędnych. Wybór jednostki w jakich wyrażona jest ta odległość jest kwestią umowną, jednak ze względu na przejrzystość i prostotę implementacji istotne jest aby wybór ten był konsekwentny, czyli aby przestrzeń zawsze była mierzona w tych samych jednostkach. Najczęściej stosowane są w tym celu metry.

Wektor prędkości wyraża liczbę jednostek odległości na jedną jednostkę czasu w każdym z trzech wymiarów. Podobnie jak w przypadku odległości istotna jest konsekwencja w doborze jednostek, najczęściej będą to metry na sekundę.

W najprostszym przypadku (przy nieobecności sił lub gdy siły się równoważą) aby przewidzieć położenie obiektu w dowolnej chwili $t$ wystarczy znać jego pozycję początkową w chwili $t_0$ i prędkość $\vec{v}$. Pozycję $p$ określa wtedy prosta funkcja liniowa:

$$p(t) = p(t_0) + t * \vec{v} \tag{1.1}$$

gdzie jednostką $p$ są metry, $t$ sekundy a $\vec{v}$ metry na sekundę. Odpowiada to pierwszej zasadzie dynamiki Newtona.

Niestety w praktyce sytuacja jest bardziej skomplikowana, liczne działające na obiekt siły wpływają na jego przyspieszenie, czyli pochodną prędkości. Wpływ tych sił sam w sobie także może ulegać zmianom w czasie. W efekcie w czasie zmianie ulega nie tylko pozycja ale też prędkość i przyspieszenie. Powyższa funkcja nie wystarczy aby z wiarygodnym przybliżeniem przewidywać ruch obiektów, potrzebne do tego będzie skonstruowanie odpowiednich równań różniczkowych i ich rozwiązanie metodami numerycznymi.

Brakującym pojęciem jest pojęcie siły, aby móc się nią posługiwać obiekty muszą poza pozycją, prędkością i przyspieszeniem posiadać także masę, wyrażaną w kilogramach. Tak jak pozostałe omawiane wielkości siła reprezentowana jest przez wektor liczb rzeczywistych, jej miarą są niutony. Zależność pomiędzy siłą a przyspieszeniem wyraża równanie:

$$\vec{a} = \frac{\vec{F}}{m} \tag{1.2}$$

gdzie $\vec{F}$ to wektor siły wyrażonej w niutonach, $m$ to masa w kilogramach a $\vec{a}$ to przyspieszenie w metrach na sekundę do kwadratu. Równanie to opisuje drugą zasadę dynamiki Newtona.

Poza opisanymi wyżej właściwościami do dalszych obliczeń istotne będzie czasami pojęcie pędu zdefiniowanego jako $P = mv$ gdzie $m$ to masa obiektu a $v$ to jego prędkość. Ponieważ $F = am$ gdzie $m$ jest stała a $a = v'$ możemy wyrazić zależność między siłą a pędem przez $P' = F$.

Dysponując czterema podstawowymi właściwościami punktu fizycznego (masa, położenie, prędkość i przyspieszenie) a także zakładając, że w każdej chwili jesteśmy w stanie określić siły działające na ten obiekt (będą one najczęściej zależeć od

jego pozycji i prędkości), i że masa obiektu nie zmienia się w czasie, jesteśmy w stanie przewidywać z dużą dokładnością położenie i prędkość obiektu w dowolnym punkcie w czasie znając ich wartości początkowe dla pewnego czasu $t_0$.

By to zrobić, potrzebne jest rozwiązanie równania różniczkowego zwyczajnego:

$$p''(t) = \frac{F(p(t), p'(t), t)}{m}$$
$$p(t_0) = p_0 \quad p'(t_0) = v_0$$

gdzie $F$ to znana nam funkcja określająca sumę sił działających na obiekt, zależna od jego pozycji, jego prędkości i czasu. $p(t)$ to pozycja obiektu w chwili $t$ a jej kolejne pochodne to oczywiście odpowiednio jego prędkość i przyspieszenie. Warto zauważyć, że równanie jest analogiczne do podanego wyżej równania wyrażającego zależność między siłą a przyspieszeniem.

Numeryczne rozwiązywanie równań różniczkowych znacznie ułatwia ograniczenie się do równań rzędu pierwszego, dlatego powyższe równanie drugiego rzędu zastąpimy równoważnymi dwoma równaniami:

$$v'(t) = \frac{F(p(t), v(t), t)}{m} \quad v(t_0) = v_0$$
$$p' = v \quad p(t_0) = p_0$$

Oczywiście symulacja interaktywna nie opiera się jedynie na jednorazowym obliczeniu pozycji i prędkości obiektu w pewnym docelowym $t_1$, lecz na następujących po sobie krokach symulacji najczęściej odpowiadających kolejnym klatkom animacji.

Zakładając że, w chwili $t_0$ stan obiektów jest znany i poprawnie wyświetlony w postaci klatki animacji, aby po upływie czasu $\Delta_0 t$ móc wyświetlić kolejną klatkę animacji należy rozwiązać powyższe równania dla $t_1 = t_0 + \Delta_0 t$. By wyświetlić kolejną klatkę, po upływie kolejnych $\Delta_1 t$ sekund powtarzamy obliczenia jednak zamiast $t_0$ za warunek początkowy przyjmujemy stan w chwili $t_1$, obliczyć natomiast chcemy stan w chwili $t_2 = t_1 + \Delta_1 t$. Kolejne kroki obliczamy aż do zakończenia symulacji.
Kolejne wartości $\Delta t$ mogą być zawsze równe (mówimy wtedy o symulacji ze stałym krokiem) lub zmienne (symulacja ze zmiennym krokiem). Najczęściej $\Delta t$ jest

dobrana tak, by zapewnić płynność animacji lecz nie wykonywać obliczeń częściej niż to konieczne, czyli wynosi między $\frac{1}{30}s$ a $\frac{1}{100}s$ co odpowiada oczywiście 30 i 100 klatkom wyświetlanym na sekundę.

Możliwe jest też wyświetlanie kolejnych klatek animacji częściej niż wykonywane są kolejne kroki symulacji. Wówczas stany obiektów potrzebne do wyświetlenia klatek animacji znajdujących się pomiędzy dwoma kolejnymi krokami symulacji otrzymywane są przez interpolację. Pozwala to na osiągnięcie większej płynności animacji przy zachowaniu niskiej częstotliwości wykonywania stosunkowo kosztownych obliczeń związanych z rozwiązywaniem równań różniczkowych.

## 1.2 Numeryczne rozwiązywanie równań różniczkowych

Analityczne rozwiązanie równań ruchu obiektów jest w kontekście symulacji interaktywnej niepraktyczne, dlatego symulatory ograniczają się do rozwiązań numerycznych. Poniżej omówionych zostało kilka najpopularniejszych metod, zakładają one, że szukamy rozwiązania równania różniczkowego zwyczajnego pierwszego rzędu.

### 1.2.1 Metoda Eulera

Najprostszą i, można by rzec, trywialną metodą jest metoda Eulera. Dla nieznanej funkcji $f(t)$, jej wartości początkowej $f(t_0) = f_0$ i znanej pochodnej $f'(t) = d(f(t), t)$, chcemy ustalić wartość $f(t_1)$ dla pewnego $t_1 = t_0 + \Delta t$. Metoda ta wyznacza tę wartość jako:

$$f(t_1) = f(t_0) + \Delta t\, d(f(t_0), t_0)$$

Dla równań ruchu punktu oznacza to, że znając pozycję punktu w chwili $t_0$ (czyli $p(t_0)$) i pochodną tej pozycji w tej samej chwili (czyli $v(t_0)$) możemy przybliżyć położenie punktu po upływie $\Delta t$ sekund przez $p(t_0) + \Delta t v(t_0)$. Innymi słowy zakładamy, że punkt będzie nadal przez $\Delta t$ sekund poruszał się z prędkością $v(t_0)$.

W sytuacji gdy prędkość faktycznie nie zmienia się metoda ta daje dokładne rozwiązanie, jednak w większości przypadków jest ona dalece niedokładna. Mimo to jest często stosowana ze względu na prostotę implementacji i znikomą liczbę obliczeń potrzebnych do wykonania w każdym kroku symulacji.

### 1.2.2 Metoda punktu środkowego

Metoda punktu środkowego bierze swoją nazwę z faktu, że pochodna funkcji $f$ obliczamy nie w punkcie $t_0$ ale w punkcie $t_0 + \frac{\Delta t}{2}$. Ponieważ do wyznaczenia tej pochodnej potrzebujemy znać wartość funkcji w tym punkcie wyznaczamy tę wartość metodą Eulera, co daje $f(t_0) + \frac{\Delta t}{2} d(f(t_0), t_0)$. Całość metody prezentuje się następująco:

$$f(t_1) = f(t_0) + \Delta t\, d\Big(f(t_0) + \frac{\Delta t\, d(f(t_0), t_0)}{2},\ \ t_0 + \frac{\Delta t}{2}\Big)$$

Dzięki użyciu wartości pochodnej ze środka kroku symulacji, metoda ta cechuje się znacznie szybszą zbieżnością do rozwiązania dokładnego wraz ze skracaniem długości kroku.

### 1.2.3 Metoda Rungego-Kutty

Najczęściej stosowaną w profesjonalnych symulacjach metodą numeryczną jest metoda Rungego-Kutty czwartego rzędu ze względu na wysoką dokładność przy zachowaniu względnie niskiej złożoności obliczeniowej.

Metoda ta polega na obliczeniu pochodnej szukanej funkcji $f$ w czterech punktach, wartości pochodnej oznacza się kolejno $k_1$, $k_2$, $k_3$ i $k_4$. Pierwsza wartość, to wartość pochodnej w punkcie $t_0$. Druga to wartość w punkcie $t_0 + \frac{\Delta t}{2}$ gdzie do przybliżenia wartości $f$ w tym punkcie wykorzystuje się wartość $k_1$ — tak jak w metodzie punktu środkowego. Wartość $k_3$ także przybliża wartość pochodnej w połowie przedziału, tym razem jednak wykorzystując obliczoną już wartość $k_2$. Ostatnia wartość oblicza pochodną na końcu kroku, w punkcie $t_1$, przybliżając $f(t_1)$ z wykorzystaniem $k_3$. Obliczanie wartości $k$ przebiega więc następująco:

$$k_1 = d(f(t_0), t_0)$$

$$k_2 = d\Big(f(t_0) + \frac{\Delta t\, k_1}{2},\ \ t_0 + \frac{\Delta t}{2}\Big)$$
$$k_3 = d\Big(f(t_0) + \frac{\Delta t\, k_2}{2},\ \ t_0 + \frac{\Delta t}{2}\Big)$$
$$k_4 = d\Big(f(t_0) + \Delta t\, k_3,\ \ t_0 + \Delta t\Big)$$

Dysponując tymi czterema wartościami, $f(t_1)$ przybliżane jest przez:

$$f(t_1) = f(t_0) + \frac{1}{6}\Delta t(k_1 + 2k_2 + 2k_3 + k_4)$$

## 1.3 Bryły sztywne

Do tej pory rozważaliśmy jedynie obiekty fizyczne będące punktami w przestrzeni — nie posiadającymi kształtu ani orientacji. Jednak aby symulacja w jakimkolwiek stopniu przypominała rzeczywistość, musi być wypełniona przestrzennymi kształtami. Dla uproszczenia przyjmuje się, że kształty te nie ulegają deformacji (są sztywne) i są równomiernie gęste.
Takie bryły sztywne mogą być reprezentowane przez zbiór ścian, które ograniczają ich kształt. Każda ściana określona jest przez ciąg wyznaczających ją wierzchołków. Pozycja tych wierzchołków określona jest jednak nie względem środka układu współrzędnych lecz względem środka ciężkości bryły, co znacznie ułatwia obliczenia. Zakładamy, że środek ciężkości bryły zawsze leży w punkcie $(0, 0, 0)$ jej lokalnego układu.
Ponieważ bryły posiadają kształt, istotna staje się ich orientacja i ruch obrotowy. Do ich reprezentacji i obliczeń potrzebne są nowe wartości, odpowiadające tym związanym z ruchem liniowym: orientacja odpowiadająca pozycji, prędkość kątowa i przyspieszenie kątowe a także moment siły i moment pędu odpowiadające sile i pędowi oraz moment bezwładności odpowiadający masie.

Wobec tego stan dowolnej bryły sztywnej w momencie $t$ określają jednoznacznie cztery wartości: pozycja $p(t)$ i orientacja $q(t)$, które umieszczają ją w przestrzeni a także prędkość $v(t)$ i prędkość kątowa $\omega(t)$, które opisują ruch bryły. Te wartości są wystarczające by móc wyświetlić bryłę w chwili $t$.

Orientacja bryły wyrażona jest za pomocą macierzy $3 \times 3$, którą oznaczę przez $R$, wyznacza ona jednoznacznie obrót bryły wokół jej środka ciężkości. Mając do

dyspozycji pozycję i orientację bryły w momencie $t$, oznaczone odpowiednio jako $p(t)$ i $R(t)$ jesteśmy w stanie przekształcić położenie punktu wyrażone względem jej środka na jego pozycję w globalnym układzie współrzędnych. Jeśli punkt $a$ jest określony w lokalnym układzie współrzędnych bryły to jego pozycja w globalnym układzie w chwili $t$ to:

$$a_g = R(t)a + p(t) \tag{1.3}$$

Mimo, że macierze orientacji są powszechnie stosowane na przykład w bibliotekach graficznych i z tego względu są one przydatne podczas wyświetlania obiektów na ekranie, do wewnętrznej reprezentacji orientacji lepiej nadają się kwaterniony. Kwaterniony to czwórki liczb, które będę oznaczał jako parę $[s, v]$ gdzie $s$ jest liczbą rzeczywistą a $v$ trójwymiarowym wektorem liczb rzeczywistych.
Orientację bryły wyrażoną za pomocą kwaternionu oznaczę przez $q$.

Kwaternion orientacji można wyprowadzić znając oś wokół której dokonuje się obrót bryły, oznaczoną jako wektor jednostkowy $u$ i kąt obrotu $\alpha$ wyrażony w radianach. Wówczas odpowiadający obrotowi kwaternion ma postać:

$$[\cos\frac{\alpha}{2}, u\sin\frac{\alpha}{2}] \tag{1.4}$$

Dla kwaternionu $q = [s, v]$ Zależność pomiędzy $R$ a $q$ wyraża równanie:

$$R = \begin{pmatrix} 1 - 2v_y^2 - 2v_z^2 & 2v_xv_y - 2sv_z & 2v_xv_z + 2sv_y \\ 2v_xv_y + 2sv_z & 1 - 2v_x^2 - 2v_z^2 & 2v_yv_z - 2sv_x \\ 2v_xv_z - 2sv_y & 2v_yv_z + 2sv_x & 1 - 2v_x^2 - 2v_y^2 \end{pmatrix}$$

Powodem, dla którego kwaterniony stanowią lepszą reprezentację jest fakt, że tylko niewielki podzbiór macierzy $3 \times 3$ (konkretnie, macierze o wyznaczniku równym 1) stanowi reprezentację orientacji. Pozostałe macierze poza obrotem obiektu będą go także deformować. Jest to problemem gdy weźmie się pod uwagę narastający wraz z rozwojem symulacji błąd obliczeń zmiennoprzecinkowych. W miarę narastania błędu macierz coraz bardziej odbiegać będzie od poprawnej reprezentacji orientacji.

Dla odmiany, wszystkie kwaterniony o normie równej 1 są poprawną reprezentacją orientacji bryły sztywnej. Nawet jeśli w wyniku gromadzących się błędów kwaternion straci tę właściwość, można ją łatwo przywrócić normalizując go.

Do kolejnych wyprowadzeń konieczne jest zdefiniowanie dwóch prostych operacji na kwaternionach, mnożenie:

$$[s_a, v][s_b, w] = [s_a s_b - vw, s_a w + s_b v + v \times w]$$

i dodawanie:

$$[s_a, v] + [s_b, w] = [s_a + s_b, v + w]$$

Mnożenie kwaternionu przez skalar sprowadza się do mnożenia przez kwaternion o zerowej części urojonej: $a[s, v] = [a, \mathbf{0}][s, v]$.

Zmiana, jakiej ulega orientacja w czasie określona jest przez prędkość kątową, analogicznie jak w przypadku pozycji obiektu. Jednak w przeciwieństwie do pozycji i prędkości liniowych, prędkość kątowa, której powszechnie używa się w symulacjach nie jest pochodną orientacji po czasie, mimo, że pochodna ta jest konieczna do obliczenia zmiany orientacji w toku symulacji.
Zamiast tego, prędkość kątowa wyrażona jest za pomocą trójwymiarowego wektora, którego kierunek wyznacza oś wokół której obraca się obiekt, a długość określa prędkość obrotu wyrażoną w radianach na sekundę. Tak zdefiniowany wektor, oznaczany przez $\omega$, łączy z pochodną $q$ po czasie następująca relacja (z równania 1.4):

$$q'(t) = \frac{1}{2}[0, \omega(t)]q(t) \tag{1.5}$$

Moment siły określa, w jaki sposób siła przyłożona do obiektu wpływa na jego obrót. W przypadku obiektów będących punktami przyłożenie siły wpływa jedynie na prędkość liniową i dokładne miejsce, na które działa siła nie ma znaczenia. W przypadku brył jednak, w zależności od tego w którym miejscu przyłożymy siłę, jej moment przybierze różne wartości. Jeśli $r$ jest punktem aplikacji siły w lokalnym układzie współrzędnych bryły to zależność pomiędzy siłą a momentem siły (oznaczanym $\tau$) określa równość:

$$\tau = F \times r \tag{1.6}$$

Podobnie jak w przypadku siły, przez $\tau$ będę oznaczał najczęściej sumę wszystkich momentów siły działających na dany obiekt w danej chwili.

Analogicznie do pędu wynoszącego $P = mv$, moment pędu definiujemy jako $L = I\omega$, gdzie $I$ to tensor momentu bezwładności, który, podobnie jak masa określająca jak trudno przesunąć dany obiekt, określa trudność wprawienia obiektu w ruch obrotowy. Podobnie jak w przypadku pędu, zachodzi też zależność $L' = \tau$.
Tensor momentu bezwładności to macierz rozmiaru $3 \times 3$, której poszczególne elementy określają bezwładność obrotu wokół poszczególnych osi. Element $I_{xy}$ określa wpływ momentu siły działającego wokół osi $x$ układu współrzędnych na przyspieszenie kątowe wokół osi $y$, według następującego równania:

$$\tau_x = I_{xy}\omega'_y \tag{1.7}$$

gdzie $\omega'_y$ to pochodna prędkości kątowej, czyli przyspieszenie kątowe, wokół osi $y$. Widać tutaj silną analogię do równania $F = ma$ i tym samym analogię pomiędzy masą a momentem bezwładności.
Ponieważ osie, których dotyczą elementy $I$ są osiami globalnego układu współrzędnych wartość $I$ zależy od orientacji bryły sztywnej. Ponieważ wyliczenie wartości $I$ może być kosztowne chcemy uniknąć ponawiania obliczeń za każdym razem, gdy orientacja bryły ulegnie zmianie. W tym celu obliczamy tensor $I^b$, który wyraża te same wartości bezwładności, jednak w odniesieniu do osi lokalnego układu współrzędnych bryły. Jest on niezależny od orientacji bryły w układzie globalnym i tym samym jest wartością stałą.
Zależność pomiędzy $I$ a $I^b$ wyraża równanie:

$$I = RI^bR^T \tag{1.8}$$

Obliczenie w ten sposób $I$ na podstawie $I^b$ jest dalece mniej kosztowne. Niniejsza praca nie obejmuje metod obliczania $I^b$, jednak ich wartości dla wielu powszechnych brył są znane i mogą być łatwo na stałe zakodowane w symulacji a następnie wyliczane wedle potrzeby podstawiając odpowiednią masę, wymiary i typ bryły.

## 1.4 Ruch brył sztywnych

Tak jak w rozdziale 1.1, do prowadzenia symulacji brył sztywnych potrzebne jest rozwiązanie równania różniczkowego zwyczajnego, do czego niezbędne jest obliczenie pochodnej stanu bryły. Tak jak poprzednio znane są nam pochodne pozycji (prędkość) i prędkości (przyspieszenie, obliczane bezpośrednio z funkcji $F$). Pochodną orientacji możemy obliczyć znając prędkość kątową według równania 1.5, natomiast pochodną prędkości kątowej wyraża równanie (Baraff [1]):

$$\omega'(t) = I(t)^{-1}\tau$$

gdzie $\tau$ to moment siły (będący funkcją, podobnie jak $F$, zależną od czasu i stanu bryły).

Całość równania różniczkowego prezentuje się następująco:

$$\begin{aligned} p(t_0) &= p_0 \quad p'(t) = v(t) \\ v(t_0) &= v_0 \quad v'(t) = \frac{F(t)}{m} \\ q(t_0) &= q_0 \quad q'(t) = \frac{1}{2}[0, \omega(t)]q(t) \\ \omega(t_0) &= \omega_0 \quad \omega'(t) = I(t)^{-1}\tau \end{aligned}$$

Dla uproszczenia, zależność funkcji $F$ i $\tau$ od stanu bryły w chili $t$ można pozostawić w domyśle i zapisywać je po prostu jako $F(t)$ i $\tau(t)$. Metody numeryczne z rozdziału 1.2 można łatwo rozszerzyć tak, by rozwiązywały powyższe równania, lub nawet by rozwiązywały dowolny problem składający się z wektora funkcji o znanych wartościach początkowych i pochodnych, które potrafimy wyliczyć dla czasu $t$ o ile znane są nam wartości funkcji bazowych dla czasu $t$.
Powyższe równania spełniają te warunki, znane są nam wartości funkcji $p$, $v$, $\omega$ i $q$ dla czasu $t_0$ i znane są nam ich pochodne dla każdego $t$ dla którego znane są nam ich wartości.

Poniżej znajduje się przykładowa implementacja funkcji **StateDerivative**, która dla czasu $t$ i stanu bryły w czasie $t$ zwraca pochodną jej stanu.

Poza implementacją funkcji, podane są też definicje struktury RBState i klasy RBody, z których ona korzysta.

Dysponując funkcją **StateDerivative** możemy zaimplementować krok symulacji, wykorzystując jedną z metod numerycznych z rozdziału 1.2. Użyję metody RK4, pisząc funkcję **RK4Step**.
Dzięki zaimplementowaniu operacji arytmetycznych na stanach brył, możemy rozwiązywać wszystkie cztery równania różniczkowe jednocześnie, zastępując je jednym równaniem gdzie niewiadomą jest cały stan bryły.

```
1  // bryla sztywna
2  class RBody
3  {
4      public:
5          Vector3     pos;        // pozycja
6          Vector3     vel;        // predkosc
7          Quaternion  ang_pos;    // orientacja
8          Vector3     ang_vel;    // predkosc katowa
9          double      mass        // masa
10
11         // tensory bezwladnosci w lokalnym ukladzie wspolrzednych
12         Matrix3x3   local_inertia;
13         Matrix3x3   inv_local_inertia;
14
15         Vector3 Forces(double time, RBState state) const;
16         Vector3 Torque(double time, RBState state) const;
17         RBState State() const;
18         RBState StateDerivative(double t, RBState s) const;
19 };
20
21 // stan bryly
22 class RBState
23 {
24     public:
25         Vector3     pos;        // pozycja
26         Vector3     vel;        // predkosc
27         Quaternion  ang_pos;    // orientacja
28         Vector3     ang_vel;    // predkosc katowa
29
30         // operacje arytmetyczne na stanach ulatwiaja rozwiazywanie ODE
31         RBState operator+(RBState s);
32         RBState operator*(double a);
```

```
33          RBState operator/(double a);
34  };
35
36  RBState RBody::StateDerivative(double t, RBState s) const
37  {
38      RBState derivative;
39      Matrix3x3 rot_matrix = RBody::RotationMatrix(s.ang_pos);
40      Matrix3x3 inv_inertia =
41          rot_matrix * inv_local_inertia * rot_matrix.Transpose();
42
43      derivative.pos = s.vel;
44      derivative.vel = Forces(t, s) / mass;
45      derivative.ang_pos = (Quaternion(0, s.ang_vel) * s.ang_pos) * 0.5f;
46      derivative.ang_vel = inv_inertia * Torque (t, s);
47
48      return derivative;
49  }
50
51  void RK4Step(RBody& rb, double current_time, double delta_time)
52  {
53      double ct = current_time;
54      double dt = delta_time;
55
56      RBState k1 = rb.StateDerivative(ct, rb.State());
57      RBState k2 = rb.StateDerivative(ct + dt/2.0f, rb.State() + k1*(dt/2.0f));
58      RBState k3 = rb.StateDerivative(ct + dt/2.0f, rb.State() + k2*(dt/2.0f));
59      RBState k4 = rb.StateDerivative(ct + dt, rb.State() + k3*dt);
60      RBState delta = (k1 + k2*2.0f + k3*2.0f + k4) * (1.0f/6.0f * dt);
61
62      rb.pos = rb.pos + delta.pos;
63      rb.vel = rb.vel + delta.vel;
64      rb.ang_pos = rb.ang_pos + delta.ang_pos;
65      rb.ang_vel = rb.ang_vel + delta.ang_vel;
66
67      // normalizuje kwaternion by przywrocic mu dlugosc jednostkowa
68      rb.ang_pos = rb.ang_pos.Normalize();
69  }
```

# Rozdział 2

# Kolizje

Symulowanie kolizji pomiędzy bryłami sztywnymi dzieli się na dwa odrębne etapy: wykrywanie kolizji i reakcję na kolizje. O ile reakcja sprowadza się do stosunkowo nieskomplikowanych obliczeń o stałej złożoności dla każdej kolizji, o tyle wykrywanie wymaga rozwiązania kilku problemów o wyższej złożoności, zależnej zarówno od liczby brył uczestniczących w symulacji jak i stopnia skomplikowania poszczególnych brył (ich rozmiaru lub liczby wierzchołków, ścian i krawędzi).
Istnieje także trzeci etap, który najczęściej zaliczany jest do wykrywania kolizji, ale któremu ja poświęcę osobny dział: ustalanie parametrów kolizji.

## 2.1 Wykrywanie kolizji

Jako kolizję określamy sytuację, w której kształty dwóch brył przecinają się ze sobą, czyli istnieją pomiędzy nimi punkty wspólne. Aby skutecznie i niskim kosztem wykrywać zaistnienie takiej sytuacji podczas symulacji pewnego zbioru brył $\mathcal{B}$, podzielimy algorytm na dwie fazy: fazę ogólną i fazę szczegółową. Pierwsza z nich zajmuje się znajdowaniem par brył, które z dużym prawdopodobieństwem mogą kolidować ze sobą. Druga dla każdej takiej pary upewnia się, że kolizja faktycznie zaszła.
Dzięki temu podziałowi symulacja może ograniczyć wykorzystanie kosztownych procedur wykrywania kolizji odrzucając wcześnie przypadki, w których bryły na pewno nie nachodzą na siebie.

### 2.1.1 Faza szczegółowa

Mimo, że faza ta następuje po fazie ogólnej omówię ją w pierwszej kolejności, ponieważ stanowi ona przypadek podstawowy dla wykrywania kolizji. Faza szczegółowa polega na sprawdzeniu dla danych dwóch brył sztywnych czy zachodzi pomiędzy nimi kolizja. Najpopularniejszym i najszybszym sposobem na sprawdzenie czy dwie bryły posiadają jakieś punkty wspólne jest zastosowanie twierdzenia o osi rozdzielającej (separating axis theorem).

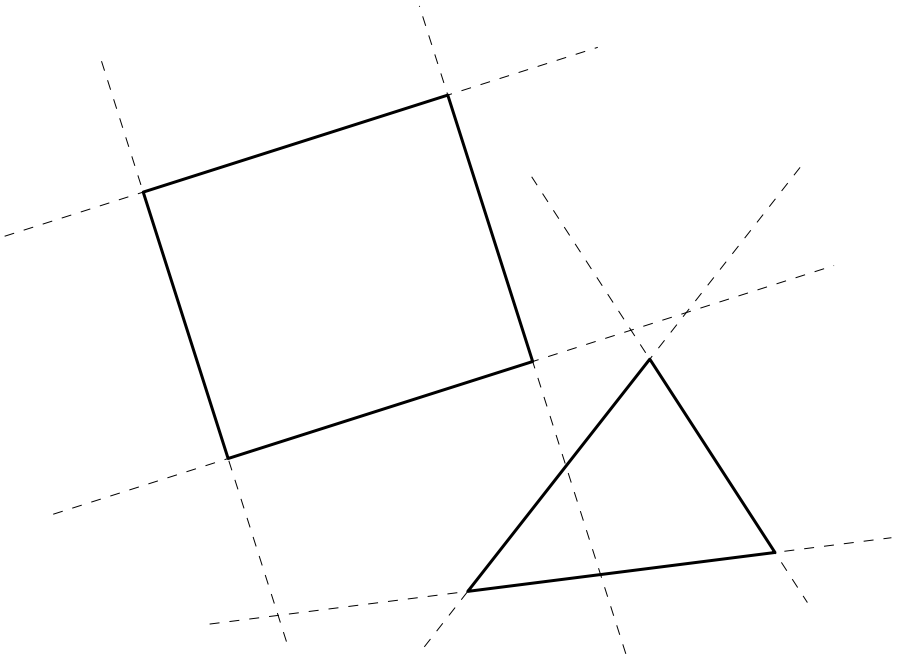
**Theorem 2.1.** *Dwa wypukłe obiekty nie nachodzą na siebie jeśli istnieje prosta (nazywana osią) taka, że ich rzuty na tę prostą nie nachodzą na siebie.*

Wydaje się, że twierdzenie to, mówiąc jedynie o obiektach wypukłych, narzuca poważne ograniczenie na klasę brył sztywnych jakie mogą zostać użyte w symulacji, jednak bryły wklęsłe możemy z łatwością reprezentować jako sumę kształtów wypukłych. Wówczas by przekonać się o istnieniu kolizji pomiędzy dwoma wklęsłymi obiektami szukamy kolizji pomiędzy dwoma z wypukłych kształtów, które się na nie składają.
W praktyce nie potrzebujemy znajdować osi, o której mówi twierdzenie. Dla dwóch brył, $A$ i $B$ wystarczy nam znalezienie takiej płaszczyzny, dla której bryła $A$ w całości leży po jednej jej stronie, zaś $B$ po drugiej. Wówczas oś z twierdzenia jest osią prostopadłą do tej płaszczyzny.

Podstawowym problemem fazy szczegółowej jest znalezienie takiej płaszczyzny lub stwierdzenie z pewnością, że płaszczyzna taka nie istnieje. Z pomocą przychodzi suma Minkowskiego. Jeśli dwie bryły wypukłe $A$ i $B$ interpretujemy jako zbiory punktów, które się w nich zawierają to ich suma Minkowskiego także będzie bryłą wypukłą, zawierającą punkty $\{a + b \mid a \in A \wedge b \in B\}$.
Można pokazać, że jeśli istnieje płaszczyzna rozdzielająca dwie bryły to jest ona równoległa do jednej ze ścian sumy Minkowskiego jednej z nich i odwrotności (odbicia względem początku układu) drugiej z nich zdefiniowanej jako:

$$A + (-B) = \{a - b \mid a \in A \wedge b \in B\}$$

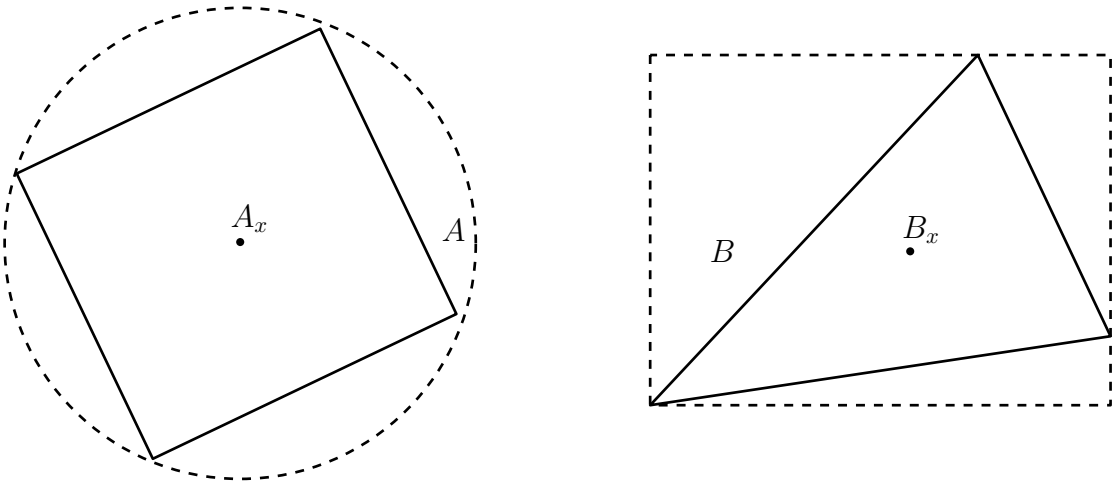
Jest tak, ponieważ punkt $o = (0, 0, 0)$ należy do tej sumy, tylko jeśli pomiędzy bryłami istnieje punkt wspólny (Yi-King Choi [2]) a ponieważ $A+(-B)$ jest kształtem wypukłym, głębokość punktu $o$ w tej wynikowej bryle jest jego odległością od jej



Rysunek 2.1: Dwie figury na płaszczyźnie wraz ze wszystkimi potencjalnymi płaszczyznami (prostymi) rozdzielającymi.

najbliższej ściany.
Ponieważ kształt sumy Minkowskiego odpowiada kształtowi osiągniętemu przez przesuwanie jednej z brył wzdłuż ścian i krawędzi drugiej bryły każda ze ścian różnicy jest wyznaczana przez ścianę jednej z oryginalnych brył lub jest równoległa do jednej z krawędzi bryły $A$ i jednej krawędzi bryły $B$ (Yi-King Choi [2])
Oznacza to, że jeśli istnieje płaszczyzna rozdzielająca dwie wypukłe bryły trójwymiarowe to w szczególności istnieje także taka, która zawiera ścianę jednej z nich albo krawędź jednej z nich i jest równoległa do krawędzi drugiej z nich.

Aby mieć pewność, że dwie bryły nie kolidują musimy zatem dla każdej ściany i każdej pary krawędzi wyznaczyć na ich podstawie płaszczyznę i sprawdzić, czy obie bryły mieszczą się w całości po jej przeciwnych stronach. Jest to zadanie czasochłonne, szczególnie dla brył o złożonej geometrii, ponieważ jego złożoność wynosi $O(n + m^2)$ gdzie $n$ ogranicza liczbę ścian a $m$ liczbę krawędzi w bryłach. Dlatego istotne jest dążenie do wykonywania tego testu możliwe jak najrzadziej. W tym celu przed przystąpieniem do sprawdzania rozłączności brył wykonuje się testy biorące pod uwagę pewne górne ograniczenie zajmowanej przez nie przestrzeni, co w niektórych przypadkach może znacznie mniejszym kosztem wykluczyć ich nachodzenie na siebie. Dwa najczęściej stosowane ograniczenia to sfera ograniczająca i $AABB$




Rysunek 2.2: Dwie bryły wraz z ich środkami ciężkości. Po lewej bryła $A$ i jej sfera ograniczająca, po prawej bryła $B$ i zajmowany przez nią obszar $AABB$.

Sfera ograniczająca to sfera dobrana w taki sposób, by bryła w całości zawierała się w jej wnętrzu. Najprostszym na to sposobem jest umieszczenie środka sfery na środku ciężkości bryły (czyli punkcie początkowym jej lokalnego układu współrzędnych) i ustalenie promienia sfery jako odległość najdalej wysuniętego wierzchołka bryły od jej środka.
Mając tak wyznaczone sfery możemy stwierdzić, że jeśli odległość pomiędzy środkami ciężkości dwóch brył jest większa niż suma promieni ich ograniczających sfer to na pewno nie zachodzi pomiędzy nimi kolizja.

*AABB* (*Axis Aligned Bounding Box*) wymaga ustalenia dla każdej bryły zajmowanego przez nią obszaru wzdłuż każdego wymiaru globalnego układu współrzędnych. Dla osi $x$, $y$, $z$ i bryły $A$ interesują nas w takim razie wartości $b_x^A$ i $e_x^A$ oznaczające pierwsze współrzędne dwóch punktów bryły najdalej wysuniętych wzdłuż osi OX a także analogiczne wartości dla osi $y$ i $z$.
Dana para brył na pewno nie nachodzi na siebie jeśli nie nachodzą na siebie parami ich przedziały $[b_x, e_x]$, $[b_y, e_y]$ ani $[b_z, e_z]$.

### 2.1.2 Faza ogólna

W fazie ogólnej operujemy całym zbiorem $\mathcal{B}$ starając się wyłonić z niego pary brył, które z dużym prawdopodobieństwem weszły ze sobą w kolizję. Oczywiście trywialnym rozwiązaniem jest sprawdzenie zachodzenia kolizji pomiędzy każdą

możliwą parą brył według technik fazy szczegółowej. Wymaga to jednak wykonania $O(n^2)$ testów. Aby poprawić ten wynik stosuje się najczęściej jedną z dwóch metod: *podziału przestrzeni* lub *sortowania i przycinania*.

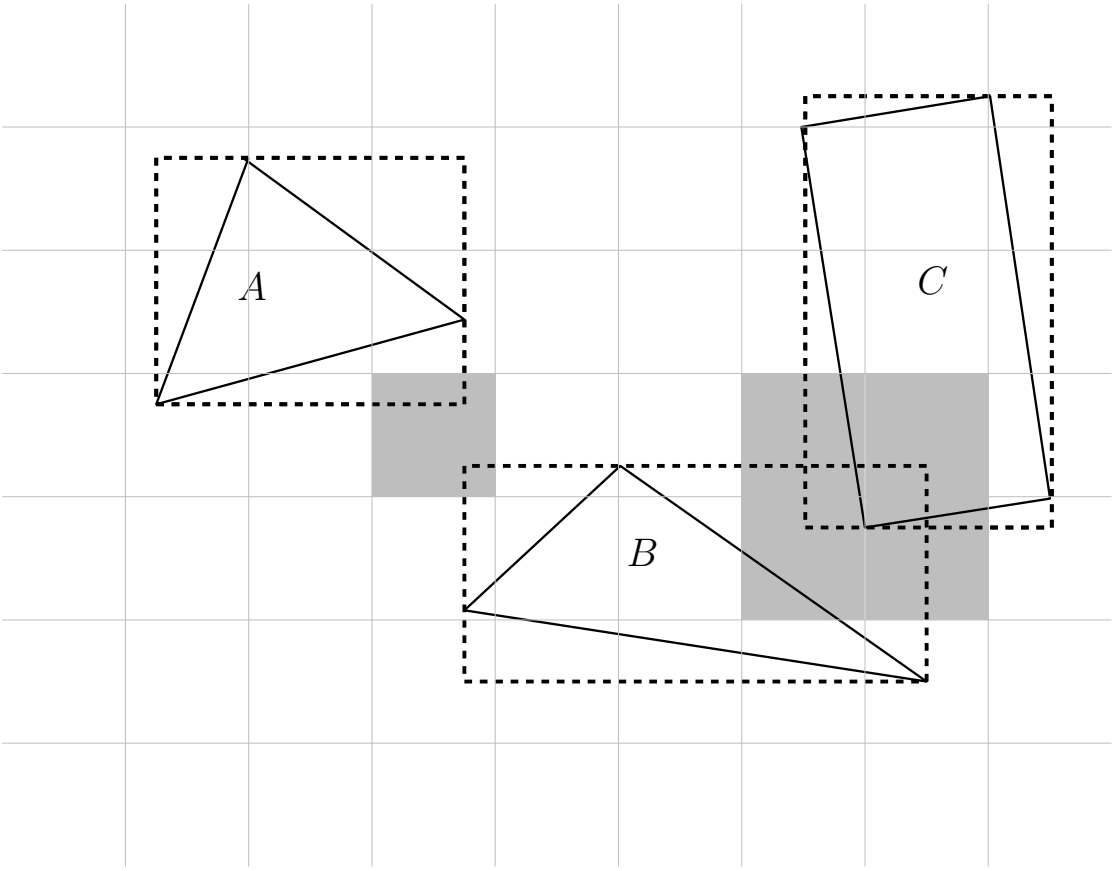
Metoda podziału przestrzeni polega na podzieleniu trójwymiarowej przestrzeni symulacji na rozłączne fragmenty, najczęściej w postaci kraty o polach jednakowej długości w każdym z wymiarów. Następnie dla każdej bryły sprawdzamy w przybliżeniu w których polach kraty się ona znajduje, stosując jako przybliżenie zajmowanej przez nią przestrzeni sferę ograniczającą lub *AABB*. Gdy proces ten zostanie wykonany dla każdej bryły iteruje się po zajętych polach oznaczając pary brył zajmujące te same pola jako potencjalne kolizje.
Ze względu na konieczność iteracji jedynie po polach zajętych, nie istnieje potrzeba tworzenia struktury danych odpowiadającej całości kraty. Potrzebna jest jedynie struktura umożliwiająca przechowywanie informacji o tym, które jej pola są zajęte i przez jakie bryły.
Praktycznym rozwiązaniem jest przechowywanie informacji o zajmowanych polach w zbalansowanym binarnym drzewie poszukiwań indeksowanym numerem pola (składającym się z trzech liczb naturalnych) i przechowującym dla każdego zajętego pola listę znajdujących się w nim brył. Po zebraniu informacji dotyczących wszystkich brył przeglądamy drzewo umieszczając każdą parę obiektów występującą w jednej liście w osobnej strukturze przechowującej informację o potencjalnych kolizjach.
Dokładny algorytm dla uproszczonego przypadku jednowymiarowego podany jest poniżej.
Użyta struktura mapy wykorzystuje w domyśle w swojej implementacji zbalansowane drzewo.

```
procedure SpatialPartition(bodies, cell_size)
    map ← empty_map()
    collision_set ← empty_set()
    for all body ∈ bodies do
        b ← get_lower_bound(body)
        e ← get_upper_bound(body)
        for i ← b, e step cell_size do
            map ← insert(map, i/cell_size, body)
        end for
    end for
    for all list ∈ values(map) do
```

```
        if length(list) > 1 then
            for each pair (a, b) ∈ list and (a, b) ∉ collision_set do
                collision_set ← insert((a, b), collision_set)
            end for
        end if
    end for
    return collision_set
end procedure
```

W ten sposób złożoność fazy ogólnej z wykorzystaniem tej metody wynosi $O(n \log n)$ przy założeniu, że maksymalny rozmiar brył sztywnych w dowolnym wymiarze jest ograniczony przez stałą. Złożoność ta wynika z konieczności umieszczenia w drzewie informacji dotyczącej $n$ brył, gdzie każde takie umieszczenie (poprzez operację *insert*) ma złożoność $O(\log n)$.
W praktyce efektywność metody waha się znacznie w zależności od doboru rozmiaru komórek kraty do rozmiarów obiektów. Zły dobór może owocować nie tylko wysoką stałą złożoności obliczeniowej ale także znacznie zawyżoną liczbą sugerowanych potencjalnych kolizji.

Metoda sortowania i przycinania opiera się na wykrywaniu nachodzących na siebie kształtów osobno w każdym z wymiarów. Dla każdej bryły i każdej z osi układu współrzędnych wyznaczane są skrajne punkty jej rzutu na tę oś. Następnie punkty skrajne wszystkich brył są sortowane, otrzymuje się w ten sposób trzy posortowane listy punktów. Przeglądając listy w posortowanym porządku można w liniowym czasie wykryć potencjalne pary nachodzących na siebie brył.
W tym celu utrzymuje się listę aktualnie aktywnych rzutów *Active*, gdy na liście punktów napotykamy punkt $b^A$ do zbioru potencjalnych kolizji dla tej osi dodajemy zbiór $\{(A, x) \mid x \in Active\}$ a do listy *Active* dodajemy bryłę $A$. Gdy napotykamy punkt $e^A$ usuwamy z listy *Active* bryłę $A$. Procedurę powtarza się dla każdej listy punktów a następnie jako wynikowy zbiór potencjalnych kolizji zwraca się przekrój zbiorów dla wszystkich osi. W ten sposób do fazy szczegółowej przechodzą bryły, które nachodzą na siebie we wszystkich trzech wymiarach.
Złożoność metody sortowania i przycinania to $O(n \log n)$ ze względu na konieczność sortowania punktów granicznych brył. Zakłada się przy tym, że liczba wierzchołków każdej bryły jest ograniczona przez stałą, ponieważ do ustalenia skrajnych współrzędnych potrzebne jest sprawdzenie wszystkich wierzchołków obiektu.




Rysunek 2.3: Przykład metody podziału przestrzeni. $A$, $B$ i $C$ są bryłami. Przerywaną linią oznaczone są górne ograniczenia na obszary przez nie zajmowane. Na szaro oznaczone są pola kraty zajmowane przez więcej niż jedno górne ograniczenie.

### 2.1.3 Optymalizacja przez zapamiętywanie

Większość przedstawionych powyżej metod wykrywania kolizji można zoptymalizować wykorzystując fakt, że zmiany pomiędzy poszczególnymi krokami symulacji są z reguły bardzo niewielkie (Baraff [1]). Zapamiętując informacje wykorzystane w poprzednim kroku możemy zaktualizować je kosztem mniejszym niż koszt ponownego zebrania kompletnych informacji lub liczyć na to, że informacje te nadal będą aktualne.

Metoda płaszczyzny rozdzielającej może zostać znacznie przyspieszona przez zapamiętanie dla każdej sprawdzanej pary płaszczyzny, która rozdzielała je podczas ostatniego testu. Jeśli dwa testy dzieli niewielka liczba kroków symulacji z dużym prawdopodobieństwem ta sama płaszczyzna nadal będzie oddzielać od siebie kształty brył. Tylko w przypadku gdy zapamiętana płaszczyzna przestaje je rozdzielać potrzebne jest obliczenie nowej lub stwierdzenie, że taka nie istnieje.

Podobnie dla fazy ogólnej metoda sortowania i przycinania może zostać poprawiona przez zapamiętanie list punktów skrajnych. W kolejnym kroku listę taką można zaktualizować w czasie liniowym obliczając nowe punkty skrajne brył a następnie posortować w czasie zależnym od stopnia w jakim jej porządek uległ zaburzeniu, wykorzystując sortowanie przez wstawianie. W przypadku gdy porządek listy nie uległ zmianie, lub zmiany są znikome, złożoność jej posortowania będzie wynosić $O(n)$.

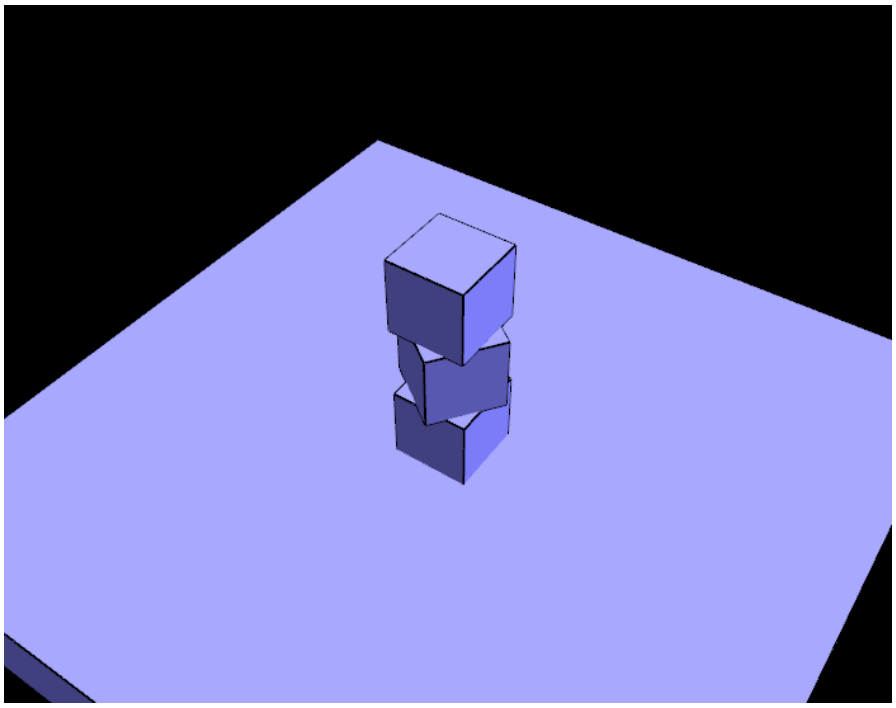
### 2.1.4 Uwagi implementacyjne, próbkowanie punktów

Wykrywanie kolizji jest często jednym z najbardziej kosztownych obliczeniowo elementów symulacji. Liczba możliwych osi rozdzielających rośnie kwadratowo wraz ze wzrostem skomplikowania brył (liczby wierzchołków i krawędzi). Ponieważ w każdym kroku symulacji orientacja tych osi może zmieniać się względem zaangażowanych brył zapamiętanie związanych z nimi rzutów jest niemożliwe, co oznacza, że dla każdej osi konieczne jest każdorazowe rzutowanie wszystkich wierzchołków obu brył.

Metoda osi rozdzielającej musi sprawdzić wszystkie możliwe osie by stwierdzić na pewno, że obiekty nachodzą na siebie, jednak może zakończyć się ona wcześniej jeśli wykryje oś, w której obiekty nie nachodzą. Oznacza to, że metoda ta sprawdza się dużo lepiej w sytuacji, gdy szansa na wystąpienie kolizji jest mała.

Aby uniknąć wysokiego kosztu sprawdzania kolizji pomiędzy parami krawędzi w sytuacji gdy kolizja najprawdopodobniej ma miejsce zaimplementowałem dodatkowy etap, który zamiast korzystać jedynie z twierdzenia o osi rozdzielającej testuje w pierwszej kolejności wybrane punkty obu brył, sprawdzając, czy znajdują się one wewnątrz drugiej bryły. Jeśli wybrany punk jednej bryły znajduje się wewnątrz drugiej to bryły na pewno nachodzą na siebie. Pozwala to znacznie szybciej stwierdzić, że pomiędzy bryłami zachodzi kolizja, niż w przypadku metody osi rozdzielającej, która musi sprawdzić w tym celu wszystkie osie.

Problemem jest jednak odpowiedni dobór takich punktów, tak by zmaksymalizować szanse na wczesne wykrycie kolizji i jednocześnie zminimalizować narzut obliczeń wymaganych do przetworzenia punktów.
Dobrymi kandydatami do sprawdzenia są wierzchołki brył. Jest ich o rząd wielkości mniej niż płaszczyzn rozdzielających, a każda kolizja wierzchołka ze ścianą

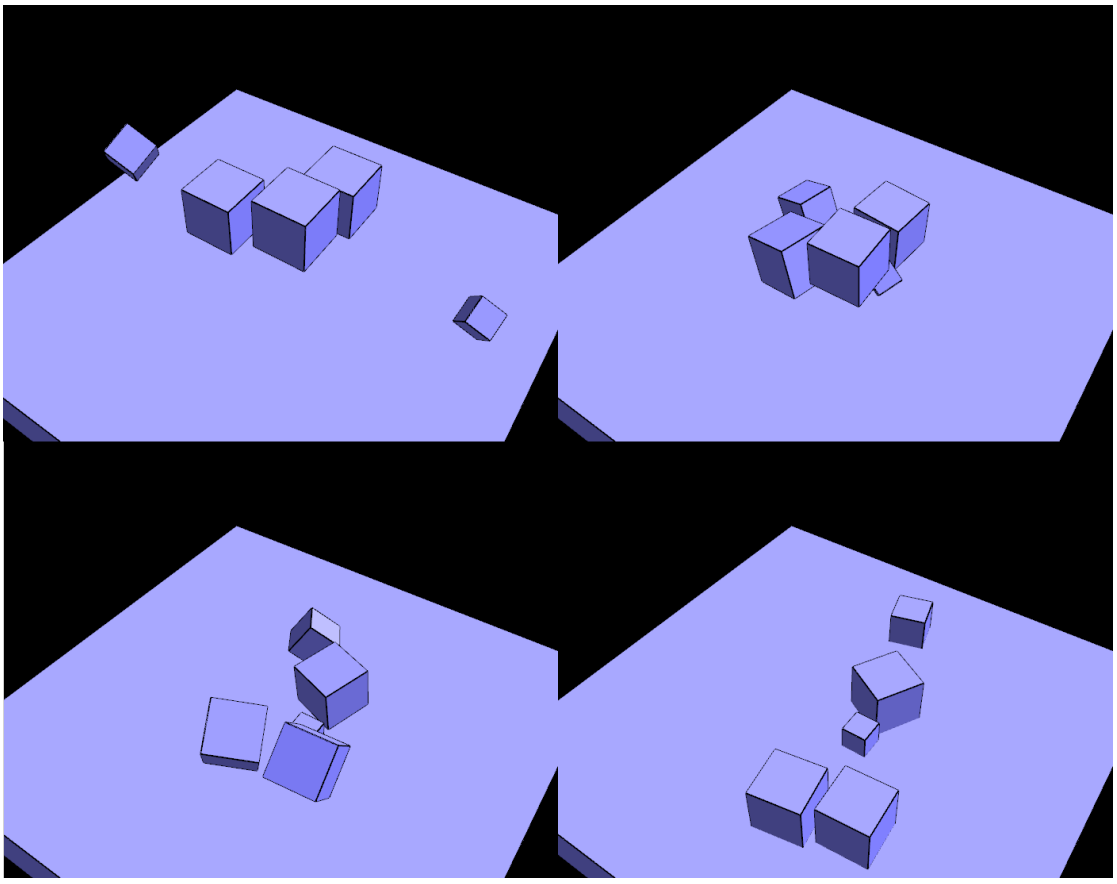


Rysunek 2.4: Scena A, trzy spoczywające na sobie w bezruchu bryły generują w każdym kroku animacji wiele jednoczesnych kolizji, które bardzo nieznacznie różnią się między sobą w czasie.

może zostać wykryta biorąc pod uwagę właśnie te punkty.
Drugim dobrym kandydatem są punkty, w których bryły nachodziły na siebie w poprzednich krokach symulacji. Każda bryła może pamiętać swoich kilka ostatnich punktów kolizji i zawsze sprawdzać je w pierwszej kolejności. Pozwala to na bardzo szybkie ustalenie, że dwie bryły faktycznie nadal na siebie nachodzą w sytuacji gdy spoczywają one na sobie przez wiele kolejnych kroków symulacji. To rozwiązanie jest szczególnie korzystne, ponieważ wykrywanie kolizji metodą osi rozdzielającej jest najbardziej kosztowne w sytuacji gdy bryły kolidują ze sobą w wielu następujących po sobie krokach, czyli wtedy gdy metoda próbkowania ostatnich punktów kolizji jest najskuteczniejsza.

Przeprowadziłem testy tego usprawnienia dodając do brył bufor przechowujący stałą liczbę ostatnio wykrytych punktów kolizji i licząc średni czas potrzebny na obliczenie kompletnej klatki symulacji.

Testy przeprowadziłem dla dwóch symulowanych scen, sceny A składającej się z brył ułożonych w wieżę i sceny B gdzie bryły wchodzą se sobą w liczne dynamiczne kolizje. Dla obu scen wyznaczyłem średni czas potrzebny na obliczenie jednego kroku symulacji poprzez wykonanie tysiąca następujących po sobie kroków.
Proces ten powtórzyłem zarówno dla przypadku gdy optymalizacja jest wyłączona

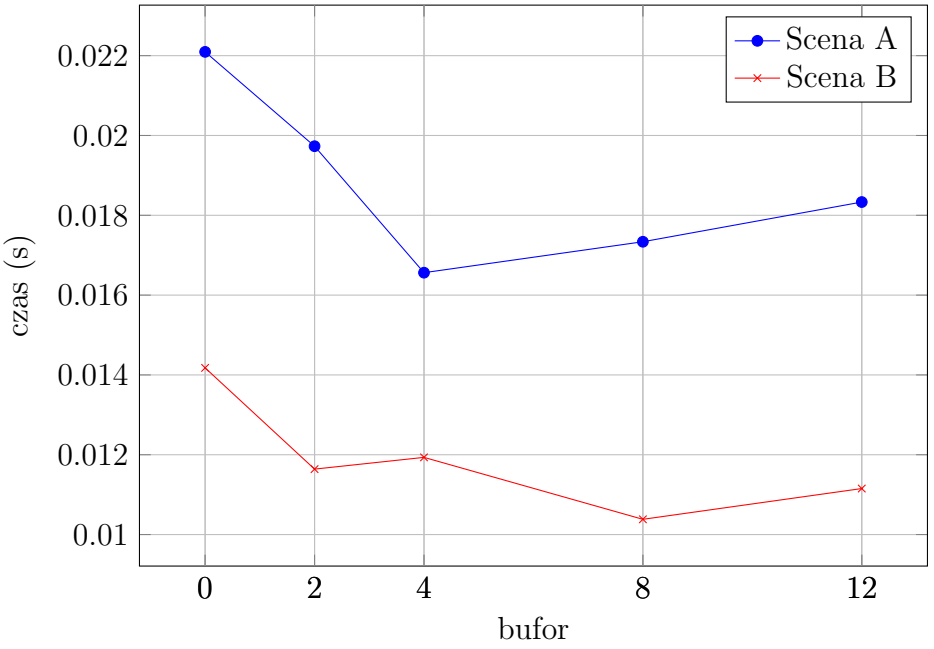


Rysunek 2.5: Scena B, pięć brył wchodzi ze sobą w liczne, następujące w krótkich odstępach czasu kolizje przed wytraceniem swoich prędkości i ustabilizowaniem się w spoczynku.

jak i dla różnych rozmiarów bufora punktów.
Wyniki podsumowuje poniższy wykres.

Testy zostały przeprowadzone dla rozmiarów bufora: 0 (bufor wyłączony), 2, 4, 8 i 12. W obu scenach można dostrzec wyraźną poprawę wydajności po dodaniu bufora punktów, jednak łatwo zauważyć pomiędzy nimi pewne różnice.
Scena A, składająca się ze spoczywających na sobie w bezruchu brył zyskuje na optymalizacji znacznie więcej ponieważ położenie punktów kolizji pomiędzy klatkami praktycznie nie ulega zmianie. Widać też, że największy zysk dał bufor wielkości czterech punktów, ponieważ w większości przypadków na właśnie tylu punktach będą spoczywać bryły. Zwiększanie bufora ponad tę wartość powoduje jedynie zwalnianie algorytmu ze względu na większą liczbę danych koniecznych do przetworzenia.
W przypadku sceny B zysk jest mniejszy, ale także więcej korzyści mogą dawać dłuższe bufory. Dzieje się tak, ponieważ gdy bryły koziołkują i odbijają się od

siebie wielokrotnie, mimo, że pomiędzy dwoma klatkami symulacji mogą nie pojawić się żadne punkty wspólne kolizji, mogą one powtarzać się na przestrzeni wielu kroków.




Podsumowując, wprowadzenie bufora punktów kolizji pozwala uzyskać czas obliczeń krótszy nawet o 25% w skrajnych przypadkach, gdy bryły spoczywają na sobie bez ruchu.

## 2.2 Znajdowanie punktów kolizji

Istnieje wiele podejść do problemu odnajdowania punktów kolizji i ich rozwiązywania. W tej sekcji przedstawione są dwie skrajne metody — rozpatrywanie punktów kolizji pojedynczo w chwili ich zaistnienia i rozwiązywanie kolizji zbiorczo, w chwili gdy je wykryjemy.

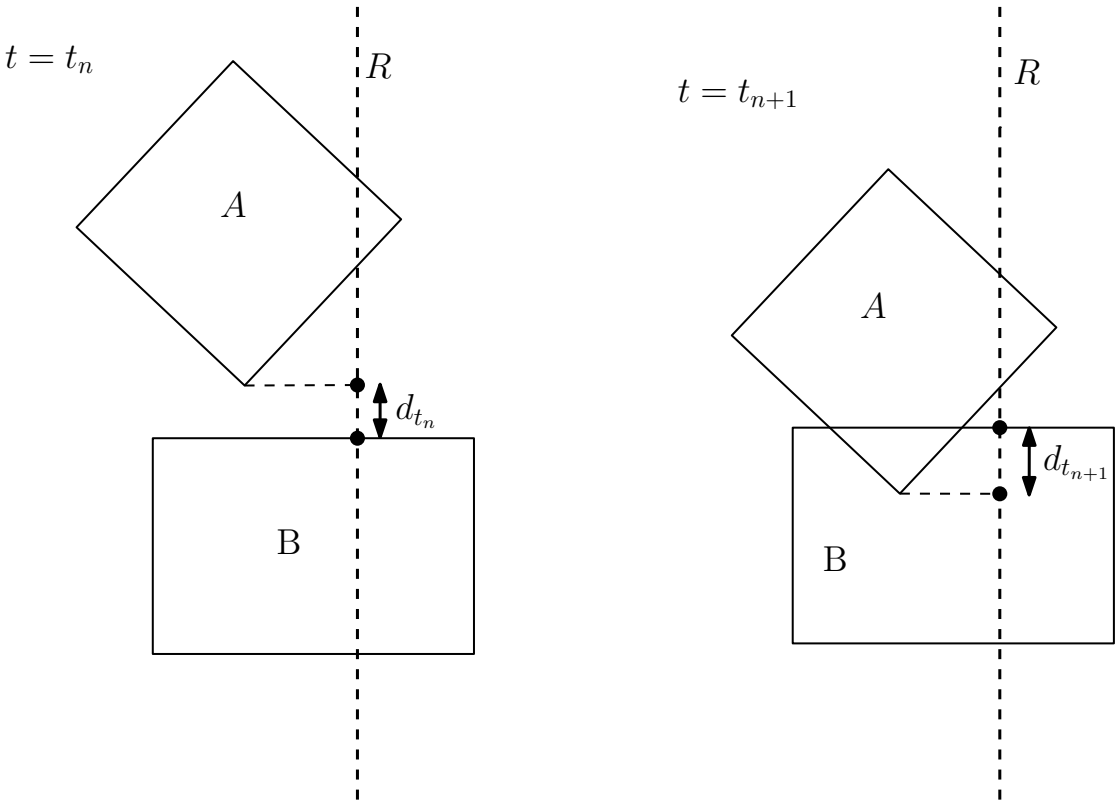
### 2.2.1 Rozwiązywanie kolizji w chwili jej wystąpienia

Jeśli chcemy zareagować na zaistnienie kolizji dokładnie w chwili, w której ona wystąpiła musimy znaleźć moment kiedy dwie bryły wchodzą ze sobą w kontakt i punkt, w którym to następuje. Takiej metody użyto w pracy Baraffa [1].

Najczęściej moment, w którym dwa kształty wchodzą w kontakt następuje pomiędzy dwoma krokami symulacji. W chwili $t_n$ bryły nie kolidują ze sobą ale w chwili $t_{n+1}$ zostaje już wykryta pomiędzy nimi penetracja. Chcemy wyznaczyć (z pewną tolerancją) moment $t_c$ taki, że $t_n \leqslant t_c \leqslant t_{n+1}$ i w którym bryły właśnie wchodzą ze sobą w kontakt — czyli ich punkty wspólne leżą tylko na ich powierzchni.
Najprostszym sposobem na wyznaczenie $t_c$ jest metoda bisekcji. Wiedząc, że szukana znajduje się gdzieś pomiędzy $t_n$ a $t_{n+1}$ wyznaczamy pozycję obiektów w chwili $\frac{t_n+t_{n+1}}{2}$. Jeśli kontakt nie nastąpił jeszcze w momencie $\frac{t_n+t_{n+1}}{2}$ to musi on nastąpić między $\frac{t_n+t_{n+1}}{2}$ a $t_{n+1}$. Jeśli kontakt nastąpił wcześniej i bryły już penetrują się wzajemnie to musiał on nastąpić pomiędzy $t_n$ a $\frac{t_n+t_{n+1}}{2}$. Procedurę tę powtarzamy rekurencyjnie do momentu spełnienia warunku stopu — na przykład do momentu gdy głębokość penetracji będzie mieścić się w przedziale $< -\epsilon, \epsilon >$ dla pewnego małego $\epsilon$.
Głębokość penetracji sprawdzamy rzutując obie bryły na oś prostopadłą do każdej z możliwych płaszczyzn rozdzielających i szukając osi, wzdłuż której zachodzi minimalne nachodzenie rzutów na siebie. Najczęściej minimalne nachodzenie będzie zachodzić wzdłuż osi prostopadłej do płaszczyzny, która jako ostatnia służyła za świadka rozłączności brył w etapie szczegółowym wykrywania kolizji.

Gdy znany jest już z akceptowalną dokładnością moment $t_c$ możemy wyznaczyć punkt lub punkty styku. Każdy taki punkt można zaklasyfikować jako kontakt wierzchołka ze ścianą lub kontakt krawędzi z krawędzią (kontakty wierzchołek-wierzchołek i wierzchołek-krawędź możemy bez zaburzania symulacji przybliżyć tymi przypadkami). W pierwszym przypadku ścianą będzie ściana wyznaczająca płaszczyznę rozdzielającą o minimalnej penetracji a wierzchołek będzie wierzchołkiem najbliższym tej płaszczyzny. Ten wierzchołek jest wówczas szukanym punktem styku.
W drugim przypadku krawędzie będą krawędziami wyznaczającymi płaszczyznę rozdzielającą a szukanym punktem będzie punkt na jednej z nich, leżący najbliżej drugiej z nich.
Jednoczesnych punktów styku dla dwóch brył może być wiele, ponieważ może istnieć wiele punktów, które znajdują się w odległości mniejszej niż $\epsilon$ od płaszczyzny rozdzielającej lub może istnieć wiele par krawędzi wyznaczających tę płaszczyznę i należących do niej z dokładnością $\epsilon$. Wszystkie te punkty przekazujemy do procedury rozwiązującej kolizje.

Wielokrotne wyznaczanie pozycji brył dla różnych $t$ wymaga pewnego stopnia



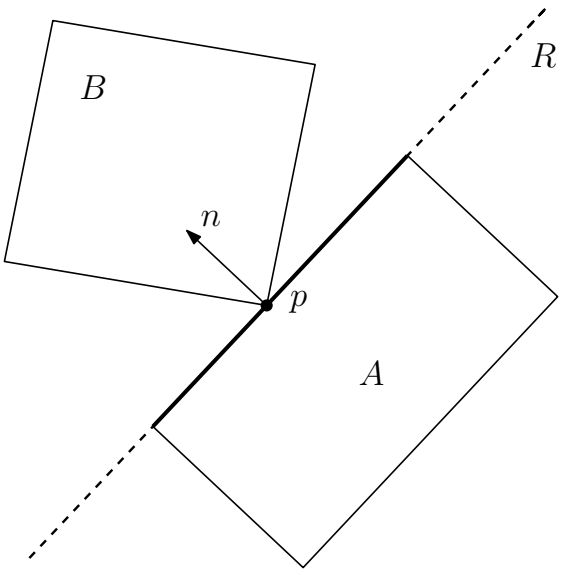

RYSUNEK 2.6: Znajdowanie momentu wystąpienia kolizji pomiędzy bryłami $A$ i $B$. W chwili $t_n$ głębokość penetracji wzdłuż osi $R$ wynosi $d_{t_n} > \epsilon$ a w chwili $t_{n+1}$ wynosi $d_{t_{n+1}} < -\epsilon$. Kolizja musiała nastąpić w chwili $t_c$ gdzie $t_n < t_c < t_{n+1}$.

interakcji modułu wykrywania kolizji z procedurą rozwiązującą równania ruchu brył. Pomiędzy $t_n$ a $t_{n+1}$ mogło zajść wiele kolizji. Zakładamy, że informacje o tych kolizjach (ich głębokości penetracji) przetrzymujemy na liście. Musimy potrafić spośród nich wybrać tę, dla której głębokość penetracji jest maksymalna i metodą bisekcji cofać lub przewijać symulację aż do znalezienia punktu $t_c$, dla którego ta maksymalna penetracja znajduje się w przedziale $< -\epsilon, \epsilon >$. Po każdej zmianie w pozycjach brył konieczne jest na nowo obliczyć głębokości penetracji na liście kolizji.

Wówczas możemy rozwiązać wszystkie kolizje, których głębokość mieści się w tym przedziale i usunąć je z listy. Następnie ponownie metodą bisekcji odnajdujemy punkt w czasie, dla którego kolejna kolizja o maksymalnej penetracji występuje z dokładnością do $\epsilon$.

W ten sposób kolizje rozpatrujemy jedna po drugiej, zawsze w chwili ich wystąpienia, aż do momentu gdy lista kolizji zaistniałych pomiędzy $t_n$ a $t_{n+1}$ będzie pusta.

Metodę tę podsumowuje poniższy fragment kodu. **bodies** i **collisions** to struktury




Rysunek 2.7: Bryły $A$ i $B$ są w chwili kontaktu rozdzielane przez płaszczyznę $R$. Ścianą biorącą udział w kolizji jest zaznaczona ściana bryły $A$ a wierzchołkiem jest zaznaczony wierzchołek bryły $B$. Wyznaczają one odpowiednio normalną kolizji $n$ i punkt styku $p$.

przechowujące odpowiednio wszystkie bryły i zaistniałe kolizje. Funkcja **RK4StepAll** rozwija symulację brył w czasie o $t =$ **delta_time**.

Funkcja **CheckAll** wykrywa wszystkie zaistniałe kolizje i umieszcza je w strukturze **collisions**. Każda umieszczona tam kolizja zawiera informację o jej głębokości i uczestniczących bryłach.

**MaxPen** zwraca maksymalną głębokość penetracji spośród wszystkich kolizji a **RecheckAll** oblicza na nowo głębokości pamiętanych penetracji. W końcu funkcja **SolveAll** rozwiązuje wszystkie kolizje, których głębokość mieści się w przedziale $< -EPS, EPS >$.

```
1  double target_time; // = t_(n+1)
2  double current_time = target_time;
3  ODESolver::RK4StepAll(&bodies, delta_time);
4  CDetector::CheckAll(&bodies, &collisions);
5
6  while (collisions.size() > 0)
7  {
8      double search_step = delta_time/2.0f;
9      double max_depth = CDetector::MaxPen(&collisions);
10
11     while (max_depth > EPS || max_depth < -EPS)
12     {
13         if (max_depth > EPS) delta_time = -search_step;
```

```
14            else delta_time = search_step;
15            search_step /= 2.0f;
16            current_time += delta_time;
17            ODESolver::RK4StepAll(&bodies, delta_time);
18            CDetector::RecheckAll(&collisions);
19            max_depth = CDetector::MaxPen(&collisions);
20        }
21        CSolver::SolveAll(&collisions, EPS);
22        delta_time = target_time - current_time;
23        ODESolver::RK4StepAll(&bodies, delta_time);
24        CDetector::RecheckAll(&collisions);
25    }
```

### 2.2.2 Rozwiązywanie kolizji bez znajomości momentu wystąpienia

W tej metodzie dla żadnej z kolizji zaistniałych pomiędzy $t_n$ a $t_{n+1}$ nie szukamy ich momentu wystąpienia. Zamiast tego szukać będziemy dla każdej z nich potencjalnych punktów penetracji i rozwiązywać je zakładając, że występują w chwili $t_n$ w punkcie o najgłębszej penetracji. Została ona opisana na przykład w pracy Guendelmana [3].
Wynikowy algorytm jest zarówno znacznie prostszy w implementacji jak i mniej złożony obliczeniowo, co pozwala na modelowanie większej liczby jednoczesnych kolizji tym samym kosztem.

Zakładając, że wszystkie pary kolidujących ze sobą brył zostały wykryte, dla każdej takiej pary $(A, B)$ znajdujemy wszystkie punkty $p$ spełniające jeden z warunków:

1. $p$ jest wierzchołkiem bryły $A$ $(B)$ i znajduje się wewnątrz bryły $B$ $(A)$
2. $p$ jest punktem należącym do jednej z krawędzi $A$ leżącym najbliżej jednej z krawędzi bryły $B$, nie jest wierzchołkiem $A$ i leży wewnątrz $B$

Są to potencjalne punkty najgłębszej penetracji dla kolizji typu, odpowiednio, wierzchołek-ściana i krawędź-krawędź. Umieszczamy je na liście. Dla każdego punktu zapamiętujemy także jego głębokość penetracji i normalną związanej z nim kolizji. Dla pierwszego przypadku będą to odpowiednio odległość $p$ od najbliższej ściany

$B$ ($A$) i wektor normalny tej ściany. Dla drugiego przypadku będą to odległość $p$ od krawędzi $B$ i wektor jednostkowy wskazujący z $p$ na najbliższy mu punkt tej krawędzi.

Po znalezieniu wszystkich punktów wycofujemy symulację do punktu $t_n$. Następnie dla każdej pary $(A, B)$ punkty te sortujemy po ich głębokości penetracji, najgłębszy punkt przyjmujemy jako punkt kolizji i rozpatrujemy ją, po czym usuwamy go z listy.
Dla pozostałych na liście punktów $p$ sprawdzamy, czy punkty $p_A$ i $p_B$, będące punktem $p$ wyrażonym w lokalnych układach współrzędnych $A$ i $B$ zbliżają czy oddalają się od siebie w kierunku normalnej kolizji. Jeśli punkty te oddalają się od siebie to w toku symulacji kolizja w punkcie $p$ zostanie rozwiązana i usuwamy go z listy. Dla wierzchołków, które pozostaną na liście powtarzamy proces aż do momentu gdy lista się opróżni.
Po rozpatrzeniu wszystkich par $(A, B)$ przewijamy symulację ponownie do punktu $t_{n+1}$.

Powyższa metoda może owocować w dalece niepoprawne wyniki w sytuacji gdy prędkości obiektów lub krok symulacji są na tyle duże, że stan obiektów w chwili $t_{n+1}$ lub $t_n$ ma bardzo mało wspólnego z ich stanem w chwili wystąpienia kolizji. W praktyce jednak długość kroku symulacji jest zwykle na tyle mała, że wyniki dawane przez tę metodę, mimo, że widocznie różne od wyników zwracanych przez metodę rozwiązywania kolizji w chwili ich wystąpienia, sprawiają wrażenie realistycznych i zachowują podstawowe prawa fizyki.

### 2.2.3 Uwagi implementacyjne, filtrowanie krawędzi i wierzchołków

Obie wymienione wyżej metody rozwiązywania kolizji wymagają odszukania wszystkich punktów penetracji, mogących być zarówno punktami gdzie wierzchołek jednej bryły wszedł w kontakt ze ścianą drugiej, lub punktami styku krawędzi dwóch brył.
Dla implementacji naiwnej, złożoność obliczeniowa sprawdzenia wszystkich wierzchołków dla penetracji wierzchołek-ściana wynosi $O(\mathbf{VF})$ gdzie $\mathbf{V}$ to liczba wierzchołków a $\mathbf{F}$ to liczba ścian w obu bryłach. Podobnie dla penetracji krawędź-krawędź złożoność to $O(\mathbf{E}^2)$ gdzie $\mathbf{E}$ to liczba krawędzi.

Czas działania symulacji można istotnie poprawić filtrując w czasie liniowym listę krawędzi i wierzchołków brył odrzucając te, które na pewno nie biorą udziału w kolizji.

Mechanizm filtrowania jest prosty, wierzchołki i krawędzie, które mogą brać udział w kolizji muszą z pewną dokładnością należeć do płaszczyzny rozdzielającej bryły. Jeśli $n$ jest wektorem normalnym tej płaszczyzny a $p$ to pewien punkt do niej należący to wierzchołek $v$ jest dobrym kandydatem na punkt penetracji jeśli:

$$|(v_g - p)n| < \epsilon + d$$

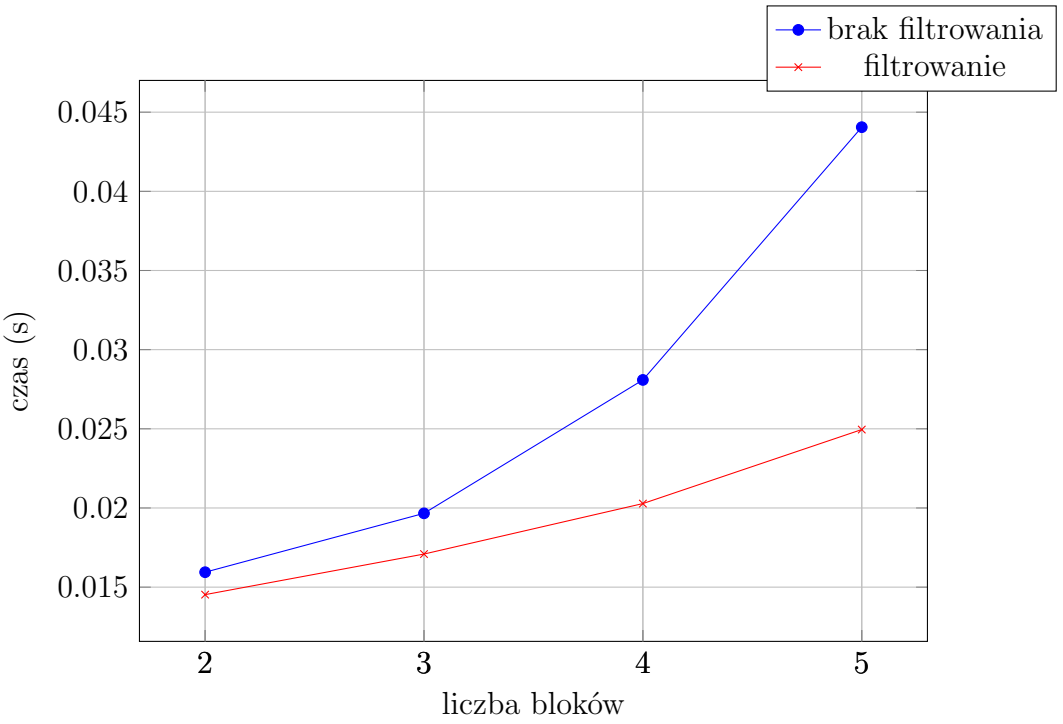
gdzie $v_g$ to pozycja wierzchołka $v$ w globalnym układzie współrzędnych, $d$ to głębokość penetracji brył wzdłuż normalnej $n$ a $\epsilon$ to pewna mała stała dodana w celu zniwelowania wpływu błędów numerycznych. Warunek ten sprawdza odległość punktu $v_g$ od płaszczyzny rozdzielającej, rzutując wektor $v_g - p$ na normalną płaszczyzny.

Podobnie krawędź $e$ jest dobrym kandydatem jeśli:

$$|(e_a - p)n| < \epsilon + d \quad \wedge \quad |(e_b - p)n| < \epsilon + d$$

gdzie $e_a$ jest początkiem krawędzi a $e_b$ jej końcem w globalnym układzie współrzędnych.

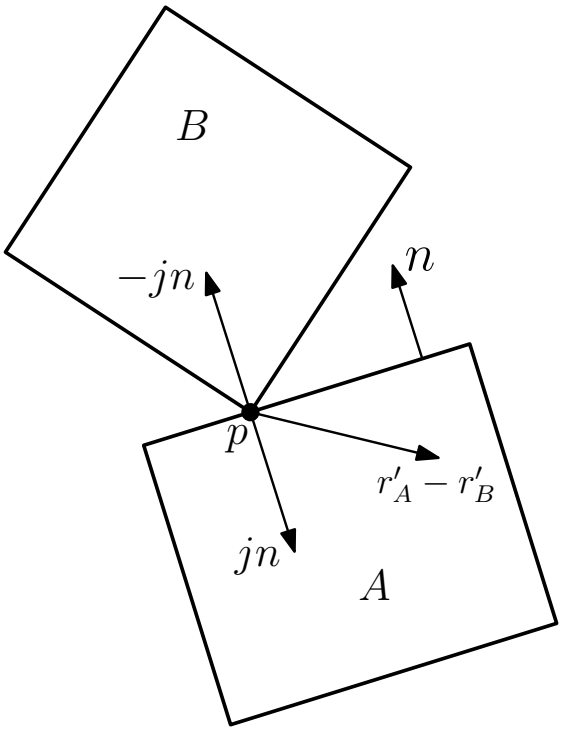
Po zaimplementowaniu tej optymalizacji przeprowadziłem testy porównujące wydajność symulacji z i bez jej udziału. W ramach testu porównałem średni czas potrzebny na wykonanie jednego kroku symulacji obliczony na podstawie tysiąca kolejnych kroków sceny składającej się z kilku spoczywających na sobie bloków. Scena została dobrana w taki sposób, by konieczne było ciągłe wykrywanie punktów kolizji, zarówno dla kontaktów punkt-ściana jak i krawędź-krawędź. Testy przeprowadziłem dla liczby bloków od dwóch do pięciu, wyniki znajdują się w tabelce poniżej:




Liczba punktów i krawędzi po przefiltrowaniu będzie zawsze znacząco mniejsza niż ich łączna liczba, ponieważ do płaszczyzny rozdzielającej może jednocześnie należeć w przybliżeniu co najwyżej jedna ściana danej bryły. Dzięki temu, przy założeniu, że liczba krawędzi i punków należących do jednej ściany ograniczona jest przez stałą, filtrowanie jest w stanie zredukować złożoność wykrycia punktów kolizji dla pojedynczej pary brył do złożoności liniowej, co zdają się potwierdzać testy.

## 2.3 Właściwe rozwiązywanie kolizji

Znając już parę brył wchodzącą w kolizję a także punkty, w których następuje kontakt brył możemy obliczyć efekt ich zderzenia. Ponieważ zakładamy, że wszystkie obiekty biorące udział w symulacji są idealnie sztywne w chwili wystąpienia kolizji zmiana kierunku ich ruchu musi nastąpić natychmiast. Z tego powodu nie można użyć do tego celu opisanego w pierwszym rozdziale mechanizmu siły, ponieważ ta potrzebuje pewnego niezerowego odcinku czasu by zmienić prędkość obiektu — w szczególności warunkiem poprawnego sformułowania równania różniczkowego jest aby prędkość, pochodna pozycji, była ciągła w czasie. Nagła zmiana kierunku ruchu tworzy nieciągłość co wymaga zatrzymania symulacji w chwili kolizji, obliczenia nowych prędkości liniowych i kątowych i wznowienie symulacji po ich




RYSUNEK 2.8: Impuls $jn$ aplikujemy w punkcie $p$ do bryły $A$ a impuls $-jn$ do bryły $B$. Wektor $r'_A - r'_B$ po rzutowaniu na normalną kolizji $n$ da wartość ujemną, co oznacza, że bryły zbliżają się do siebie w punkcie styku.

wprowadzeniu.

Tę nagłą zmianę w prędkościach określa się mianem impulsu (za książką Bourga [4]). Impuls jest wielkością wektorową, działającą podobnie do siły, jednak wpływającą nie na przyspieszenie a na prędkość obiektu. Zmiana prędkości $\Delta v$ po zaaplikowaniu impulsu $J$ wynosi (według pracy Baraffa [1]):

$$\Delta v = \frac{J}{M}$$

Gdzie $M$ to masa obiektu. W zależności od punktu $p$ (w lokalnym układzie współrzędnych bryły), w którym zaaplikujemy impuls $J$ zmianie ulega także prędkość kątowa obiektu (za pracą Baraffa [1]):

$$\Delta\omega = I^{-1}(p \times J)$$

Do obliczenia impulsu dla pary brył $A$ i $B$ kolidujących w punkcie $p$ potrzebny będzie szereg dodatkowych wielkości.

Jeśli kolizja nastąpiła pomiędzy wierzchołkiem a ścianą to bez straty ogólności

zakładamy, że bryła $A$ zawiera wierzchołek a bryła $B$ ścianę. Przez $n$ oznaczymy wektor normalny kolidującej ściany skierowany w stronę bryły $A$.
Jeśli bryły kolidują ze sobą krawędziami to przez $e_A$ oznaczymy wektor jednostkowy równoległy do kolidującej krawędzi $A$ a przez $e_B$ analogiczny wektor dla bryły $B$. Przez $n$ tym razem oznaczymy wektor prostopadły zarówno do $e_A$ i do $e_B$, tak jak poprzednio skierowany od bryły $B$ do bryły $A$.
Przez $r_A$ i $r_B$ oznaczymy punkt kolizji wyrażony w lokalnym układzie współrzędnych odpowiednio bryły $A$ i $B$, natomiast przez $r'_A$ i $r'_B$ prędkości tych punktów w chwili zaistnienia kolizji obliczone według wzoru:

$$r'_A = v_A + \omega_A \times r_A$$

$$r'_B = v_B + \omega_B \times r_B$$

Przez $v_{rel}$ oznaczymy względną prędkość punktów kolizji wzdłuż normalnej kolizji czyli:

$$v_{rel} = n(r'_A - r'_B)$$

Jeśli $v_{rel} > 0$ to bryły oddalają się od siebie w punkcie $p$ i symulację możemy kontynuować bez wprowadzania impulsu. Jeśli $v_{rel} = 0$ to mamy do czynienia ze spoczynkiem brył, którym zajmę się w następnym rozdziale. Jeśli natomiast $v_{rel} < 0$ to bryły zbliżają się do siebie i należy zaaplikować do nich impuls by zapobiec wzajemnej penetracji. Impuls aplikujemy do obu brył w punkcie kolizji $p$ a jego wielkość wynosi $jn$ dla bryły $A$ i $-jn$ dla bryły $B$, gdzie $j$ jest pewną stałą, której wielkość wyprowadzę poniżej.

Oznaczmy przez $r'^{-}_A$ prędkość punktu $r_A$ przed wystąpieniem kolizji a przez $r'^{+}_A$ po zaaplikowaniu impulsu. Analogicznie wyznaczymy wielkości $r'^{-}_B$ i $r'^{+}_B$ dla $r'_B$ a także $v^{-}_{rel}$ i $v^{+}_{rel}$ dla $v_{rel}$. Na ten moment zignorujemy tarcie występujące pomiędzy kolidującymi bryłami. Wówczas z zasady zachowania pędu wynika:

$$v^{+}_{rel} = -\epsilon v^{-}_{rel}$$

gdzie $\epsilon$ jest stałą sprężystości z przedziału $[0, 1]$ określającą jaka część energii jest zachowana podczas kolizji. $\epsilon = 1$ oznacza odbicie idealnie sprężyste, natomiast

$\epsilon = 0$ oznacza, że cała energia zostaje stracona i bryły w chwili kolizji zatrzymują się.
Po aplikacji impulsu zachodzić będzie równość:

$$r'^{+}_A = v^+_A + \omega^+_A \times r_A$$

gdzie $v^+_A$ i $\omega^+_A$ to prędkości bryły po aplikacji impulsu. Z definicji impulsu wynika, że są one równe:

$$v^+_A = v^-_A + \frac{jn}{M_A}$$
$$\omega^+_A = \omega^-_A + I^{-1}_A(r_A \times jn)$$

Podstawiając te wartości do równania otrzymujemy:

$$\begin{aligned} r'^{+}_A &= (v^-_A + \tfrac{jn}{M_A}) + (\omega^-_A + I^{-1}_A(r_A \times jn)) \times r_A \\ &= v^-_A + \omega^-_A \times r_A + (\tfrac{jn}{M_A}) + (I^{-1}_A(r_A \times jn)) \times r_A \\ &= r'^{-}_A + j(\tfrac{n}{M_A} + (I^{-1}_A(r_A \times n)) \times r_A) \end{aligned}$$

Wykonując analogiczne przekształcenia dla bryły $B$, na którą działa odwrotny impuls, otrzymamy równość:

$$r'^{+}_B = r'^{-}_B - j(\frac{n}{M_B} + (I^{-1}_B(r_B \times n)) \times r_B)$$

Wobec czego:

$$r'^{+}_A - r'^{+}_B = (r'^{-}_A - r'^{-}_B) + j(\frac{n}{M_A} + \frac{n}{M_B} + (I^{-1}_A(r_A \times n)) \times r_A + (I^{-1}_B(r_B \times n)) \times r_B)$$

Ponieważ $n(r'^{+}_A - r'^{+}_B) = v^+_{rel}$, $n(r'^{-}_A - r'^{-}_B) = v^-_{rel}$ a $nn = 1$ po pomnożeniu powyższego równania obustronnie przez $n$ otrzymujemy:

$$v^+_{rel} = v^-_{rel} + j(\frac{1}{M_A} + \frac{1}{M_B} + n(I^{-1}_A(r_A \times n)) \times r_A + n(I^{-1}_B(r_B \times n)) \times r_B)$$

A ponieważ $v_{rel}^{+} = -\epsilon v_{rel}^{-}$ można podstawić:

$$-\epsilon v_{rel}^{-} = v_{rel}^{-} + j(\frac{1}{M_A} + \frac{1}{M_B} + n(I_A^{-1}(r_A \times n)) \times r_A + n(I_B^{-1}(r_B \times n)) \times r_B)$$

Co pozwala nam ostatecznie wyprowadzić wzór na wielkość $j$ zależny jedynie od wartości sprzed aplikacji impulsu:

$$j = \frac{-(1+\epsilon)v_{rel}^{-}}{\frac{1}{M_A} + \frac{1}{M_B} + n(I_A^{-1}(r_A \times n)) \times r_A + n(I_B^{-1}(r_B \times n)) \times r_B}$$

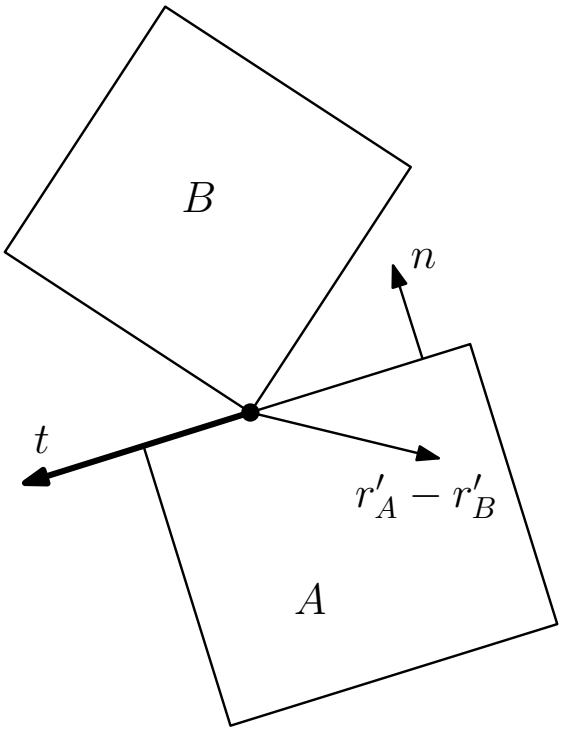
Na potrzeby większości symulacji potrzebne są specjalne obiekty, takie, których nie można poruszyć przez kolizje z innymi obiektami. Obiekty takie mogą na przykład służyć za podłogę lub powierzchnię ziemi. Chcemy jednak by inne obiekty mogły odbijać się od nich swobodnie w sposób realistyczny, więc nadal muszą one uczestniczyć w wykrywaniu i reagowaniu na zderzenia.
Dla każdego takiego obiektu $S$, niech $\frac{1}{M_S}$ będzie równe zero a $I_S^{-1}$ będzie macierzą zerową zarówno w równaniu na wielkość $j$ jak i w równaniach aplikacji impulsu:

$$\Delta v_S = \frac{J}{M_S}$$
$$\Delta \omega_S = I_S^{-1}(p \times J)$$

Będzie to odpowiadać obiektowi o nieskończonej masie i inercji, czyli takiemu, którym nie da się poruszyć.

### 2.3.1 Impuls tarcia kinetycznego

Poza impulsem $J$ działającym wzdłuż normalnej kolizji, kontakt dwóch brył powoduje także pojawienie się tarcia, które działa wzdłuż powierzchni styku, czyli w poprzek normalnej kolizji.
W myśl praw tarcia Amontonsa wielkość siły tarcia jest proporcjonalna do siły utrzymującej ciała w kontakcie. Ponieważ z wymienionych wcześniej powodów w symulacji odbić brył sztywnych interesują nas nie siły a impulsy, odpowiednikiem tego prawa dotyczącym symulacji jest proporcjonalność wielkości impulsu $J$ do




Rysunek 2.9: Wektor $t$, wzdłuż którego aplikuje się impuls tarcia jest prostopadły do wektora normalnego kolizji $n$ i zwrócony w przeciwnym kierunku do względnej prędkości punktów styku $r'_A - r'_B$.

wielkości impulsu tarcia, który oznaczymy $J_f$ (Bourg [4]). Stałą określającą stosunek $|J|$ do $|J_f|$ jest $\mu$ — współczynnik tarcia kinetycznego.

$$\mu = \frac{|J_f|}{|J|}$$

Dla różnych par materiałów jego wielkość różni się i jest ustalana empirycznie, najczęściej jednak zachodzi $0 < \mu < 1$.

Impuls $J_f$ działa w kierunku przeciwnym do względnej prędkości obiektów w poprzek normalnej kolizji. Kierunek ten oznaczymy jako $t$:

$$t = n \times (n \times (r'_A - r'_B))$$

Po obliczeniu wielkości $j$ i aplikacji impulsu $J = jn$ do bryły $A$ i przeciwnego do bryły $B$ aplikujemy więc jeszcze impuls $J_f = \mu j t_n$ do bryły $A$ i odwrotny do bryły $B$, gdzie $t_n$ to wektor jednostkowy w kierunku $t$.

Należy jednak pamiętać, że impuls ten dotyczy jedynie tarcia kinetycznego, które działa tylko na obiekty poruszające się względem siebie wzdłuż powierzchni styku. Oznacza to, że impuls $J_f$ może być co najwyżej tak duży aby kompletnie powstrzymać ten ruch, nie może jednak go odwrócić. By obliczyć maksymalną wielkość $J_f$ można posłużyć się obliczeniami, które potrzebne były do wyznaczenia $j$. Przypomnę, że jeśli $\epsilon = 0$ to $j$ jest dokładnie wielkością potrzebną by zatrzymać zbliżanie się brył w punkcie $p$. Wobec tego, jeśli za $\epsilon$ podstawimy zero, za normalną kolizji wektor jednostkowy $t_n$ a prędkość względną wyznaczymy nie w kierunku $n$ ale w kierunku $t$ to otrzymamy wielkość impulsu $J_f$ potrzebną by zatrzymać względny ruch poprzeczny.
Oznaczmy tę wielkość $j_f^{max}$:

$$j_f^{max} = \frac{-v_{rel}^t}{\frac{1}{M_A} + \frac{1}{M_B} + t_n(I_A^{-1}(r_A \times t_n)) \times r_A + t_n(I_B^{-1}(r_B \times t_n)) \times r_B}$$

gdzie:

$$v_{rel}^t = t_n(r'_A - r'_B)$$

Wówczas ostateczna wielkość impulsu $J_f$ aplikowanego do bryły $A$ wynosi:

$$J_f = min(j_f^{max}, \mu j)t$$

Do bryły $B$ aplikujemy odwrotny impuls tarcia $-J_f$.

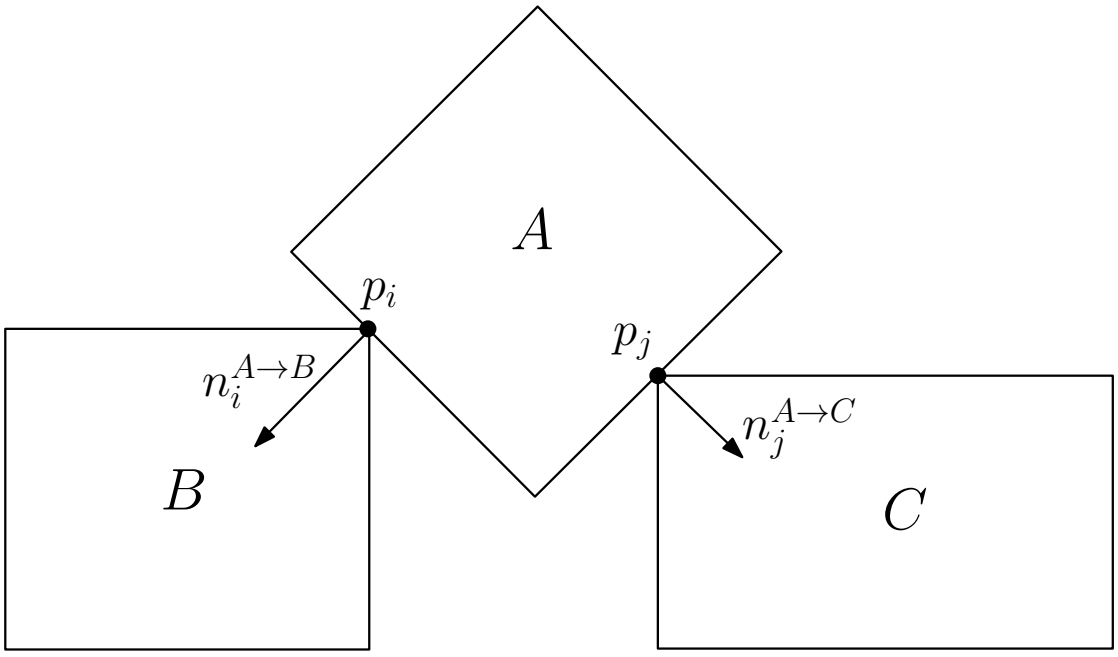
# Rozdział 3

# Spoczynek brył

Kolizje występują gdy dwie bryły najdą na siebie w wyniku zbliżania się do siebie, wówczas aplikacja impulsu proporcjonalnego do ich względnej prędkości odwraca ich ruch i sprawia, że zaczynają się od siebie oddalać. Jednak gdy jedna bryła spoczywa na drugiej ich względna prędkość wynosi zero — nie zbliżają i nie oddalają się one od siebie. Mimo to działające na nie siły (na przykład siły grawitacyjne) mogą nadal powodować, że wraz z rozwojem symulacji zajdzie pomiędzy nimi penetracja. Aby temu zapobiec należy, poprzez aplikację odpowiednich sił lub impulsów, utrzymać bryły we wzajemnym kontakcie.
Rozwiązywanie spoczynku brył zwykle następuje po wykryciu i reakcji na kolizje, ponieważ tylko pary obiektów, które znajdują się ze sobą w kontakcie, ale których względna prędkość jest zbyt niska by zaaplikować do nich impuls kolizji są kandydatami do rozwiązania ich spoczynku.
Podobnie jak w przypadku rozwiązywania kolizji, także w przypadku spoczynku spośród wielu możliwych rozwiązań skupię się na dwóch skrajnych metodach, najlepiej współdziałających z metodami z poprzedniego rozdziału.

## 3.1 Metoda sił spoczynkowych

Metoda ta opiera się na aplikacji odpowiednich sił do spoczywających na sobie brył, w odróżnieniu od rozwiązywania kolizji gdzie do brył aplikowaliśmy impulsy. Do obliczenia wielkości tych sił potrzebne jest zdefiniowanie i rozwiązanie problemu programowania kwadratowego. Celem metody jest znalezienie dla każdego punktu styku dwóch brył $p_i$ takiej minimalnej wartości siły $f_i$, która zapobiegnie




Rysunek 3.1: Układ trzech spoczywających na sobie brył. W układzie występują dwa punkty styku wraz z odpowiadającymi im normalnymi kolizji: punkt $p_i$, będący punktem kontaktu między bryłami $A$ i $B$ i punkt $p_j$ pomiędzy bryłami $A$ i $C$.

wzajemnej penetracji brył w tym punkcie.
Ponieważ konfiguracja wzajemnie spoczywających na sobie brył często będzie składać się z więcej niż jednej pary i jednego punktu kontaktu musimy wartości wszystkich sił wyliczyć jednocześnie, siły działające w jednym punkcie kontaktu mogą mieć wpływ na siły działające w innym punkcie jeśli oba punkty należą do tej samej bryły.

Załóżmy, że metoda jako dane wejściowe otrzymuje informacje o $n$ punktach styku, każdy punkt $i$ opisany jest przez biorące w nim udział bryły $A_i$ i $B_i$ a także normalną kontaktu $n_i^{B\to A}$ (skierowaną od bryły $B_i$ do bryły $A_i$) i położenie punktu styku w globalnym układzie współrzędnych $p_i$. Tak jak w przypadku kolizji siły będą działać w punkcie styku $p_i$ i prostopadle do powierzchni styku czyli w kierunku $n_i^{B\to A}$, dodatnia siła $f_i$ zadziała na bryłę $A_i$ a ujemna $-f_i$ na bryłę $B_i$.

Aby siły $f_i$ były w stanie zapobiec penetracji wszystkich brył muszą podlegać one szeregowi ograniczeń. Pierwszym ograniczeniem jest konieczność zapobieżenia penetracji brył. Jeśli jako $d_i$ oznaczymy głębokość penetracji brył $A_i$ i $B_i$ w punkcie $p_i$ to ograniczenie to możemy wyrazić jako

$$d_i = n_i^{B \to A}(p_A - p_B) \geqslant 0$$

gdzie $p_A$ i $p_B$ to punkty na bryłach $A_i$ i $B_i$ pomiędzy którymi doszło do kontaktu. Ponieważ wektor $n_i^{B \to A}$ skierowany jest od bryły $B_i$ do bryły $A_i$ wartość $d_i$ będzie ujemna gdy pomiędzy bryłami zajdzie penetracja.
Ponieważ bryły znajdują się ze sobą w kontakcie $d_i$ w tej chwili wynosi 0 (z pewną dokładnością). Aby zapobiec penetracji $d_i$ nie może się zmniejszyć, czyli jej pochodna po czasie $d_i'$ musi być nieujemna:

$$d_i' = (n_i^{B \to A})'(p_A - p_B) + n_i^{B \to A}(p_A' - p_B') \geqslant 0$$

Ponieważ w chwili kontaktu $p_A = p_B$ wyrażenie to upraszcza się do $d_i' = n_i^{B \to A}(p_A' - p_B')$ co jak łatwo zauważyć jest względną prędkością punktów $p_A$ i $p_B$, a ta według założeń w chwili kontaktu także wynosi w przybliżeniu 0. Chcemy w miarę rozwoju symulacji zapobiec zmniejszeniu się tej wartości, ponieważ oznaczałoby to, że bryły zaczynają się penetrować. W tym celu druga pochodna $d_i$ także musi być nieujemna:

$$d_i'' = ((n_i^{B \to A})''(p_A - p_B) + (n_i^{B \to A})'(p_A' - p_B')) + ((n_i^{B \to A})'(p_A' - p_B' + n_i^{B \to A}(p_A'' - p_B''))$$

ponieważ w chwili kontaktu $p_A = p_B$ wyrażenie upraszcza się do:

$$d_i'' = n_i^{B \to A}(p_A'' - p_B'') + 2(n_i^{B \to A})'(p_A' - p_B')$$

Ostatecznie by zapobiec przyszłej penetracji musi zachodzić $d_i'' \geqslant 0$, czyli względne przyspieszenie punktów $p_A$ i $p_B$ musi być nieujemne.

Ponieważ siły $f_i$ muszą odpychać bryły od siebie drugim ograniczeniem jest ich nieujemność: $f_i \geqslant 0$.

W końcu trzecie ograniczenie wynika z faktu, że siły $f_i$ powinny działać na bryły tylko w przypadku gdy jest utrzymywany między nimi kontakt czyli gdy $d_i'' = 0$. Jeśli $d_i'' > 0$ to bryły odrywają się od siebie i siły spoczynkowe powinny przestać na nie oddziaływać czyli $f_i = 0$.

Ponieważ ani $f_i$ ani $d_i''$ nie mogą być ujemne ograniczenie to oznacza, że zawsze przynajmniej jedna z tych dwóch wartości musi wynosić zero: $f_i d_i'' = 0$.

### 3.1.1 Obliczanie wartości sił

Aby móc znaleźć wartości $f_i$ spełniające te trzy ograniczenia konieczne jest znalezienie zależności pomiędzy $d_i''$ a $f_i$. Ponieważ jedna bryła może jednocześnie znajdować się w wielu spoczynkowych kontaktach wartość $d_i''$ dla danego punktu $i$ zależy nie tylko od $f_i$ ale także od wartości $f$ dla innych kontaktów. Zależność tę wyraża poniższe równanie (Baraff [5]):

$$d_i'' = a_{i1} f_1 + a_{i2} f_2 + \cdots + a_{in} f_n + b_i$$

gdzie $a_{ij}$ to elementy macierzy $\mathbf{A} \in \mathbb{R}^{n \times n}$, która zawiera informacje o masach brył i wzajemnym ułożeniu punktów kontaktu a $b_i$ to element wektora $\mathbf{b} \in \mathbb{R}^n$ zawierającego informację o zewnętrznych (względem sił spoczynkowych) siłach działających na bryły, jak na przykład siły grawitacyjne.
Przedstawiając wartości $d_i''$ i $f_i$ w postaci wektorów $\mathbf{d}$ i $\mathbf{f}$ możemy układ wszystkich równań wyrażających zależność między nimi zapisać w postaci jednego równania macierzowego:

$$\mathbf{d} = \mathbf{A}\mathbf{f} + \mathbf{b}$$

Ograniczenia $f_i \geqslant 0$ i $d_i'' \geqslant 0$ także możemy wyrazić w postaci wektorowej:

$$\mathbf{f} \geqslant \mathbf{0}$$

$$\mathbf{d} = \mathbf{A}\mathbf{f} + \mathbf{b} \geqslant \mathbf{0}$$

gdzie zapis $\mathbf{f} \geqslant \mathbf{0}$ oznacza, że każdy element $\mathbf{f}$ jest nieujemny.

Korzystając z faktu, że wszystkie elementy $d$ i $f$ są nieujemne, trzecie z podanych wcześniej ograniczeń, $f_i d_i'' = 0$, można zapisać w postaci:

$$\mathbf{f}^T \mathbf{d} = \mathbf{f}^T (\mathbf{A}\mathbf{f} + \mathbf{b}) = \mathbf{0}$$

Razem te trzy ograniczenia stanowią specyfikację problemu programowania liniowego, gdzie celem jest minimalizacja funkcji $F$ przy zachowaniu odpowiednich ograniczeń:

$$F(\mathbf{f}) = \mathbf{f}^T\mathbf{A}\mathbf{f} + \mathbf{f}^T\mathbf{b} \quad \text{przy zachowaniu} \quad \left\{ \begin{array}{l} \mathbf{f} \geqslant \mathbf{0} \\ \mathbf{A}\mathbf{f} + \mathbf{b} \geqslant \mathbf{0} \end{array} \right\}$$

Opis metody rozwiązywania tego problemu wykracza poza zakres tej pracy, istnieje jednak wiele bibliotek i programów rozwiązujących problemy programowania kwadratowego. Specjalny algorytm dostosowany do rozwiązywania konkretnie powyższego problemu jest opisany w pracy Baraffa [5].

Zakładając, że do rozwiązania zastosujemy zewnętrzny program lub bibliotekę wynikowy wektor sił $\mathbf{f}$ zaaplikujemy do brył dodając dla każdego kontaktu $i$ siłę $f_i n_i^{B\to A}$ działającą w punkcie $p_A$ na bryłę $A_i$ i siłę $-f_i n_i^{B\to A}$ działającą w $p_B$ na bryłę $B_i$. Wartości tych sił muszą zostać uwzględnione przez moduł rozwiązujący równania ruchu brył by zapobiec inter-penetracji.

Nadal jednak nieznana jest macierz $\mathbf{A}$ i wektor $\mathbf{b}$.

### 3.1.2 Obliczanie stałych w równaniach sił spoczynkowych

Stałe w macierzy $\mathbf{A}$ wyznaczają zależność wielkości $d_i''$ od poszczególnych sił $f_i$, natomiast elementy wektora $\mathbf{b}$ to składowa $d_i''$ niezależna od $\mathbf{f}$.
Najpierw skupię się na macierzy $\mathbf{A}$, przypomnę, że:

$$d_i'' = n_i^{B\to A}(p_A'' - p_B'') + 2(n_i^{B\to A})'(p_A' - p_B')$$

Druga część sumy zawiera jedynie pierwsze pochodne po czasie, odpowiadające prędkościom, i dlatego nie ma ona związku z elementami macierzy, które jako skalary sił wpływają bezpośrednio na przyspieszenie obiektu. Przyspieszenia znajdują się za to w pierwszym składniku sumy: $n_i^{B\to A}(p_A'' - p_B'')$. Rozważmy w jaki sposób $d_i''$ zależy od siły $f_j$, co determinuje element macierzy $a_{ij}$. Jeśli w kontakcie $i$ biorą udział bryły $A_i$ i $B_i$ a w kontakcie $j$ bryły $A_j$ i $B_j$ to siła $f_j$ wpływa na $d_i''$ tylko jeśli istnieją pomiędzy tymi kontaktami wspólne bryły, czyli jeśli

$$A_i = A_j \vee A_i = B_j \vee B_i = A_j \vee B_i = B_j$$

w przeciwnym razie $f_j$ nie ma wpływu na wartości $p''_A$ i $p''_B$ kontaktu $i$ wobec czego $a_{ij} = a_{ji} = 0$.
Załóżmy więc, że $A_i = A_j$, czyli na bryłę $A_i$ działa siła $f_j n_j^{B \to A}$. Wielkość $p''_A$ można wyliczyć jako:

$$p''_A = v'_A + \omega'_A \times r_A + \omega_A \times (\omega_A \times r_A)$$

gdzie $r_A$ to punkt $p_A$ wyrażony w lokalnym układzie współrzędnych bryły $A$, $v'_A$ i $\omega'_A$ to przyspieszenia liniowe i kątowe bryły $A$ a $v_A$ i $\omega_A$ to jej prędkości liniowa i kątowa. Spośród wszystkich tych wartości $f_j$ ma wpływ jedynie na przyspieszenie. Wpływ siły na przyspieszenie liniowe to siła podzielona przez masę bryły, i taki właśnie jest wpływ $f_j$ na $p''_A$:

$$\frac{f_j n_j^{B \to A}}{M_A}$$

Podobnie $f_j$ wywiera wpływ na $p''_A$ wpływając na $\omega'_A$. Jak opisano w rozdziale pierwszym, przyspieszenie kątowe wynosi:

$$\omega'_A = I_A^{-1}((I_A \omega_A) \times \omega_A + \tau_A)$$

gdzie $I_A$ to tensor inercji a $\tau_A$ to moment siły. Aplikacja siły $f$ do bryły $A$ a punkcie $r$ (w lokalnym układzie bryły $A$) wypływa na jej moment siły według równania:

$$\Delta\tau_A = f \times r$$

Oznacza to, że $f_j$ wpływa na $\omega'_A$ zmieniając $\tau_A$ o $(f_j n_j^{B \to A}) \times r_j^A$ gdzie $r_j^A$ to punkt kontaktu $p_j$ leżący na bryle $A$ i wyrażony w jej lokalnym układzie współrzędnych. Tym samym wartość $\omega'_A$ zmienia się o:

$$I_A^{-1}((f_j n_j^{B \to A}) \times r_j^A)$$

a wartość $p''_A$ o:

$$I_A^{-1}((f_j n_j^{B\to A}) \times r_j^A) \times r_A$$

Łącząc wpływ jaki $f_j$ wywiera na $p''_A$ przez przyspieszenie liniowe i kątowe otrzymujemy kompletny wpływ $f_j$ na $p''_A$:

$$\Delta p''_A = \frac{f_j n_j^{B\to A}}{M_A} + I_A^{-1}((f_j n_j^{B\to A}) \times r_j^A) \times r_A = f_j(\frac{n_j^{B\to A}}{M_A} + I_A^{-1}(n_j^{B\to A} \times r_j^A) \times r_A)$$

Wpływ $f_j$ na $p''_B$ liczymy analogicznie, pamiętając, że na bryłę $B$ działa siła o przeciwnym zwrocie:

$$\Delta p''_B = -f_j(\frac{n_j^{B\to A}}{M_B} + I_B^{-1}(n_j^{B\to A} \times r_j^B) \times r_B)$$

Pamiętając, że $d''_i = n_i^{B\to A}(p''_A - p''_B) + 2(n_i^{B\to A})'(p'_A - p'_B)$ kompletny wpływ $f_j$ na $d''_i$ wynosi:

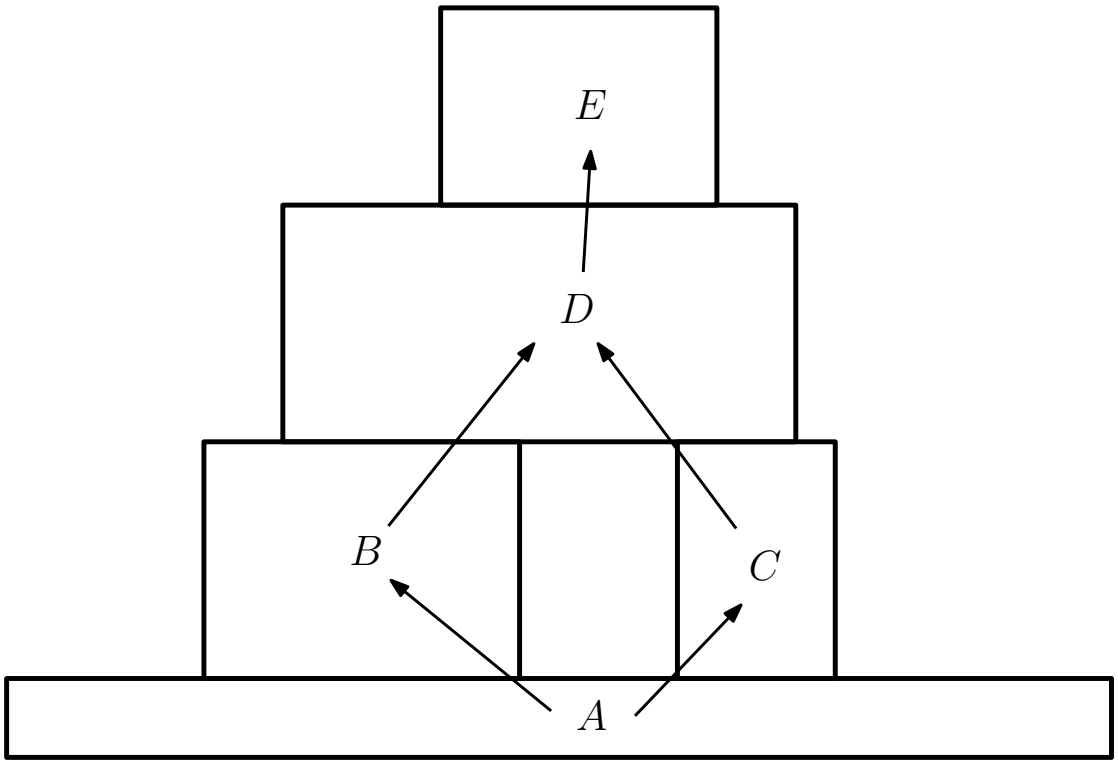
$$a_{ij} = n_i^{B\to A}(\Delta p''_A - \Delta p''_B)$$

Analogiczne obliczenia stosujemy dla wszystkich pozostałych par kontaktów by obliczyć wszystkie elementy macierzy $\mathbf{A}$. Wszystkie elementy $d''_i$, na które siły z wektora $\mathbf{f}$ nie mają wpływu muszą znaleźć się w wektorze $\mathbf{b}$. W celu ich obliczenia oznaczmy przez $F_A$ siły zewnętrzne działające na bryłę $A$ a przez $\tau_A$ działający na nią zewnętrzny moment siły.
Wstawiając te wartości do wzoru na $p''_A$ otrzymujemy:

$$(p''_A)^b = \frac{F_A}{M_A} + (I_A^{-1}((I_A\omega_A) \times \omega_A + \tau_A)) \times r_A + \omega_A \times (\omega_A \times r_A)$$

Analogicznie obliczamy wartość $(p''_B)^b$. Łącząc te wartości, mnożąc je przez $n_i^{B\to A}$ i dodając element $d''_i$ niezależny od sił otrzymujemy wartość $b_i$:

$$b_i = n_i^{B\to A}((p''_A)^b - (p''_B)^b) + 2(n_i^{B\to A})'(p'_A - p'_B)$$




Rysunek 3.2: Przykładowy układ pięciu spoczywających na sobie brył wraz z odpowiednim grafem kontaktów. Kontakty będziemy rozwiązywać aplikując impulsy zaczynając od najniżej położonych brył.

## 3.2 Metoda iteracyjnej aplikacji impulsów

Metodą alternatywną do metody sił spoczynkowych jest metoda, która podobnie jak w etapie rozwiązywania kolizji aplikuje do spoczywających brył impulsy. Została ona opisana w pracy Guendelmana [3]. Aby zapewnić odpowiednią propagację impulsów w układach wielu spoczywających na sobie wzajemnie brył impulsy aplikujemy do każdej kontaktującej pary wielokrotnie, w odpowiedniej kolejności.

Pierwszym krokiem jest skonstruowanie grafu kontaktów, który będzie służył do wyznaczenia kolejności aplikacji impulsów. Początkowo graf zawiera wierzchołki odpowiadające bryłom biorącym udział w symulacji. Każda bryła $A$ kolejno bierze udział w indywidualnym kroku symulacji, dla obecnej chwili $t$ wyznaczana jest jej pozycja dla chwili $t + \Delta t$ i sprawdzane są kolizje jakie zajdą między nią a innymi bryłami. Dla każdej takiej kolizji z bryłą $B$ w grafie umieszczana jest krawędź prowadząca z $B$ do $A$ a z wierzchołkiem $B$ kojarzona jest informacja o kolizji. Następnie symulacja bryły jest wycofywana do chwili $t$ i proces zostaje powtórzony dla kolejnego obiektu.

Tak powstały graf następnie sortuje się topologicznie, jednak ponieważ mogą występować w nim cykle, najpierw wszystkie wierzchołki należące do jednego cyklu łączymy w jeden, zawierający informację o wszystkich kolizjach związanych z wierzchołkami cyklu.
Uzyskane sortowanie jest kolejnością rozwiązywania kolizji, odpowiada ono w przybliżeniu kolejności w jakiej bryły spoczywają na sobie, zaczynając od bryły znajdującej się najbardziej na spodzie, kończąc na najbardziej wierzchniej bryle. Niestety mimo to rozwiązanie kontaktu często zaburza kontakty rozwiązane przed nim, dlatego konieczne jest wielokrotne przejście przez posortowaną listę kontaktów. Kontakty znajdujące się w jednym wierzchołku rozwiązywane są w dowolnej kolejności, jednak ponieważ są one mocniej ze sobą powiązane są przetwarzane kilkukrotnie w ramach jednego odwiedzenia wierzchołka.

Do rozwiązywania kontaktów i aplikacji impulsów użyta jest metoda podobna do metody z rozdziału 2.2.2, jednak zakłada się, że kontakty pomiędzy bryłami występują dokładnie w momencie ich rozpatrywania.
Fakt, że kontakty dotyczą statycznego spoczynku brył na sobie sugeruje, że jako stałej sprężystości $\epsilon$ najlepiej użyć 0, co spowoduje, że bryły nie odbiją się od siebie i pozostaną w kontakcie. W praktyce większą dokładność symulacji otrzymuje się zmieniając wartość $\epsilon$ z kolejnymi iteracjami.
W początkowych iteracjach, gdy układ spoczywających obiektów jest najbardziej niestabilny wartość z zakresu $(-1, 0)$ spowoduje, że zamiast odbijać się od siebie lub zatrzymywać się, bryły będą jedynie spowalniać swój ruch. Wraz z kolejnymi iteracjami $\epsilon$ stopniowo jest zwiększany by w końcu osiągnąć wartość 0 w ostatniej iteracji, określanej mianem *propagacji uderzeniowej*.
Ta ostatnia, specjalna iteracja gwarantuje, że pomiędzy obiektami nie znajdującymi się w jednym cyklu nie zajdzie penetracja. Charakteryzuje się ona nie tylko $\epsilon = 0$ ale także faktem, że rozpatrując kolizję obiektu $A$ z rozpatrywanego obecnie wierzchołka z obiektem $B$ znajdującym się w wierzchołku wcześniejszym, przyjmujemy:

$$\frac{1}{M_B} = 0$$
$$I_B^{-1} = \mathbf{0_{3\times 3}}$$

gdzie $M_B$ oznacza masę bryły $B$, $I_B^{-1}$ odwrotność jej tensora bezwładności a $\mathbf{0_{3\times 3}}$

to macierz zerowa $3 \times 3$. W ten sposób w ostatniej iteracji każda "warstwa" obiektów, po rozpatrzeniu jest zamrażana. Jej prędkość nie ulega dalszym zmianom (ze względu na nieskończoną masę i bezwładność) a odpowiedzialność za niedopuszczenie do penetracji spoczywa w całości na obiektach z wyższych warstw.
W ten sposób algorytm zmusza symulację do zapobieżenia penetracji obiektów, pozwalając jednak obiektom dowolnie wpływać na siebie na przestrzeni kilku wstępnych iteracji.

Dodatkową zaletą iteracyjnej aplikacji impulsów jest możliwość łatwego rozszerzenia jej o element tarcia. Wystarczy poza impulsem kolizji aplikować każdorazowo impuls tarcia kinetycznego tak jak opisano to w rozdziale 2.3.1.

### 3.2.1 Naturalne rozróżnienie kolizji i spoczynku

Metoda sił spoczynkowych zakładała, że spoczywające na sobie bryły mają w punktach kolizji względną prędkość wynoszącą zero. W algorytmie numerycznym sytuacja taka będzie zachodzić niezwykle rzadko i kontakt pomiędzy dwoma bryłami uznaje się za spoczynkowy gdy względna prędkość jest odpowiednio mała, na przykład mieści się w przedziale $(-e, e)$ dla pewnego małego $e$.
W przypadku metody iteracyjnej Guendelman [3] proponuje alternatywne rozwiązanie. Zamiast rozwiązywać równania numeryczne dla pozycji i prędkości brył jednocześnie, są one rozwiązywane w dwóch osobnych krokach. Niestety wymaga to zastosowania do rozwiązania równań prostej metody numerycznej, w której rozwiązania tych równań nie są od siebie wzajemnie zależne. Metodą użytą w pracy Guendelmana [3] jest metoda Eulera.
Wówczas schemat pojedynczego kroku symulacji dla obecnej chwili $t$ i długości kroku $\Delta t$ prezentuje się następująco:

1. Wstępnie oblicz prędkości a następnie pozycje obiektów w chwili $t + \Delta t$
2. Wykryj kolizję dla obliczonych pozycji i prędkości
3. Rozwiąż kolizje używając pozycji i prędkości z chwili $t$
4. Ponownie rozwiąż równania numeryczne dla prędkości używając wartości przyspieszenia zmodyfikowanych przez proces rozwiązywania kolizji

5. Dla obiektów pomiędzy którymi zaszły kolizje, i które nadal zbliżają się do siebie rozwiąż spoczynek brył

6. Rozwiąż równania numeryczne dla pozycji

Zalety tej metody mogą nie być oczywiste. Załóżmy, że w symulacji znajdują się zarówno bryły spoczywające na sobie jak i takie, które w obecnym kroku powinny się od siebie odbić.
Ponieważ kolizje wykrywamy dla pozycji przewidzianych po uwzględnieniu przyspieszenia i prędkości, zarówno bryły spoczywające jak i kolidujące zostaną wykryte w kroku drugim. Jednak ponieważ do rozwiązania kolizji używamy pozycji i prędkości z chwili $t$, obiekty, które przed wykonaniem kroku nie zbliżały się do siebie nie zostaną odbite, ich względna prędkość w chwili $t$ wynosi $v_{rel} \leqslant 0$. Jest to pożądany wynik, ponieważ obiekty, pomiędzy którymi zachodzi kontakt, ale które nie zbliżały się do siebie w poprzednim kroku najprawdopodobniej spoczywają na sobie i nie chcemy by doszło pomiędzy nimi do odbicia.
W kolejnym kroku prędkości wszystkich obiektów są aktualizowane z uwzględnieniem zmian jakie zaszły w wyniku rozwiązywania kolizji. Dopiero teraz, po uwzględnieniu działających na obiekty sił (na przykład sił grawitacyjnych), obiekty spoczywające na sobie zyskały względną prędkość i zaczęły się do siebie zbliżać, tak jak w przewidzianym stanie z pierwszego kroku. Ich kontakt został już poprawnie wykryty w kroku drugim i może zostać rozwiązany. Dopiero po uwzględnieniu zmian w prędkościach zaszłych w wyniku rozwiązywania spoczynku pozycje obiektów zostają zaktualizowane do nowych wartości zwróconych przez metodę Eulera.

### 3.2.2 Tarcie w spoczynku, tarcie statyczne

Siła tarcia statycznego to siła przeciwdziałająca próbom przesunięcia przedmiotu wzdłuż powierzchni innego obiektu gdy ich względna prędkość w punkcie styku wynosi zero. Problem uwzględnienia tej siły, a także siły tarcia dynamicznego, w metodzie sił spoczynkowych jest nietrywialny i został omówiony w pracy Baraffa [5].
W metodzie iteracyjnej aplikacji impulsów rozwiązanie jest znacznie prostsze, ponieważ do tarcia dynamicznego można wprost zastosować metodę opisaną w rozdziale dotyczącym kolizji — impulsy aplikowane w trakcie kolizji działają na tej samej zasadzie co impulsy aplikowane w spoczynku.

Jednak aby uwzględnić tarcie statyczne należy nieznacznie zmodyfikować metodę z działu 2.3.1. Poniższa metoda została opisana w pracy Bendera [6].

Tarcie statyczne objawia się poprzez zupełne zatrzymanie ruchu względnego wzdłuż powierzchni styku dwóch brył i zachodzi gdy nacisk w tym punkcie jest odpowiednio duży w stosunku do sił próbujących przesunąć bryły. Metoda bierze to pod uwagę wprowadzając współczynnik tarcia statycznego $\mu_s$, który może być zdefiniowany jako stała globalna dla całej symulacji lub być zależny od pary trących o siebie brył. Pozwala on ustalić próg, poniżej którego zachodzi tarcie statyczne. Będzie ono występować jeśli względna prędkość obiektów w punkcie styku w poprzek normalnej kolizji wynosi zero i zachodzi:

$$j_t \leqslant \mu_s j_n$$

gdzie $j_n$ to wielkość impulsu działającego w punkcie styku wzdłuż normalnej kolizji a $j_t$ to wielkość impulsu w poprzek normalnej. W takim przypadku w fazie aplikacji impulsu zamiast poprzecznego impulsu o wielkości $j_t$ aplikuje się impuls, który dokładnie zatrzyma względy ruch poprzeczny, czyli jak pokazałem w rozdziale 2.3.1:

$$j_f^{max} = \frac{-v_{rel}^t}{\frac{1}{M_A} + \frac{1}{M_B} + t_n(I_A^{-1}(r_A \times t_n)) \times r_A + t_n(I_B^{-1}(r_B \times t_n)) \times r_B}$$

Nie uwzględnienie tarcia statycznego w procesie rozwiązywania spoczynku brył może owocować drganiem i spontanicznym przesuwaniem się spoczywających na sobie brył w wyniku działania tarcia dynamicznego podczas iteracyjnej aplikacji impulsów.

## 3.3 Rozsuwanie penetrujących brył

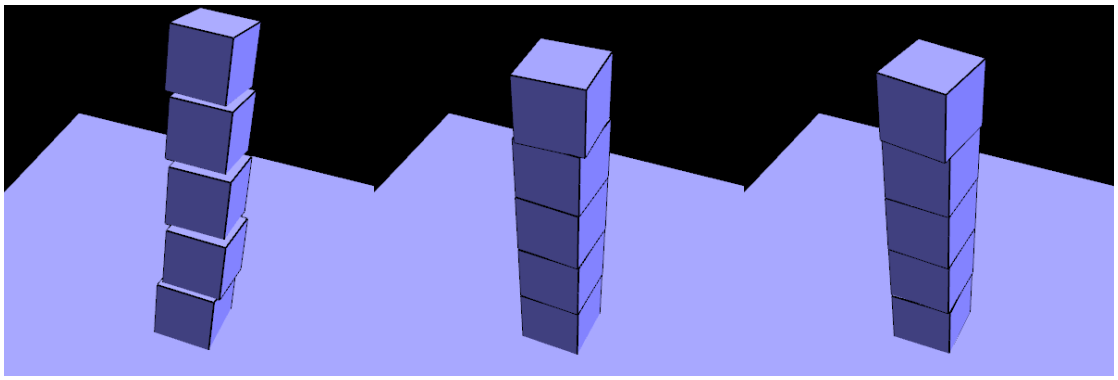
W podsumowaniu pracy Guendelmana [3] wspomniana jest możliwość rozszerzenia metody iteracyjnej aplikacji impulsów przez rozsuwanie nachodzących na siebie brył modyfikując bezpośrednio ich położenie.
Proces ten nie ma przełożenia na żadne rzeczywiste zjawisko i nie biorą w nim

udziału ani siły ani nawet impulsy, pozwala on jednak pozbyć się niewielkich niedokładności w symulacji wynikających z błędów obliczeń zmiennoprzecinkowych lub z niedostatecznej liczby iteracji procesu spoczynku brył.

W dynamicznych kolizjach brył wpływ rozsuwania jest znikomy, jednak podczas spoczynku może ono wspomagać a nawet w pewnym stopniu zastępować aplikacje impulsów spoczynkowych, ponieważ zarówno impulsy spoczynkowe jak i rozsuwanie brył ma za zadanie zapobiegać ich penetracji przy zachowaniu minimalnego wpływu na ich położenie.
Rozsunięcie brył nie ma bezpośredniego wpływu na poprawienie wydajności symulacji i bardzo znikomy wpływ na poprawę realizmu symulowanej sceny. Jednak pozwala ono zmniejszyć nacisk na dokładne odwzorowanie spoczynku brył. Jeśli niewielkie penetracje jesteśmy w stanie zniwelować przez prostą modyfikacje położenia brył to przestaje być konieczne jak najdokładniejsze zapobiegnięcie tej penetracji w etapie symulowania spoczynku.
Oznacza to, że przez zastosowanie rozsuwania możemy ograniczyć liczbę iteracji symulacji spoczynku i tym samym poprawić wydajność zachowując zbliżony stopień realizmu. Niestety ciężko zmierzyć korzyści jakie daje ta metoda, ponieważ należy rozważyć stosunek łatwo mierzalnego zysku wydajności i trudno mierzalnej utraty realizmu wynikającej z częściowego zastąpienia fizycznie poprawnej metody impulsów metodą nie mającej przełożenia na żadne zjawisko fizyczne.
Postanowiłem zaimplementować metodę rozsuwania i sprawdzić jak duży zysk wydajności można uzyskać przy zachowaniu różnic w przebiegu symulacji na tyle małych, że nie są one zauważalne dla niewprawnego oka.

Najprostszą metodą rozsunięcia brył, ingerującą w symulację w najmniejszym stopniu jest rozsunięcie ich wzdłuż osi minimalnej penetracji, po etapie rozwiązywania kolizji i spoczynku. Każdą z brył przesuwam o odległość równą głębokości penetracji przemnożonej przez stosunek jej masy do sumy mas obu brył. W ten sposób lżejsze bryły będą bardziej podatne na rozsuwanie niż ciężkie, co znacząco poprawia realizm.
Przy założeniu, że $n^{B \to A}$ to wektor równoległy do osi minimalnej penetracji (wyznaczonej metodą SAT) skierowany od bryły $B$ do bryły $A$, pozycje obu brył zostają zmodyfikowane w następujący sposób:

$$pos'_A = pos_A + n^{B \to A} * (m_A/(m_A + m_B))$$



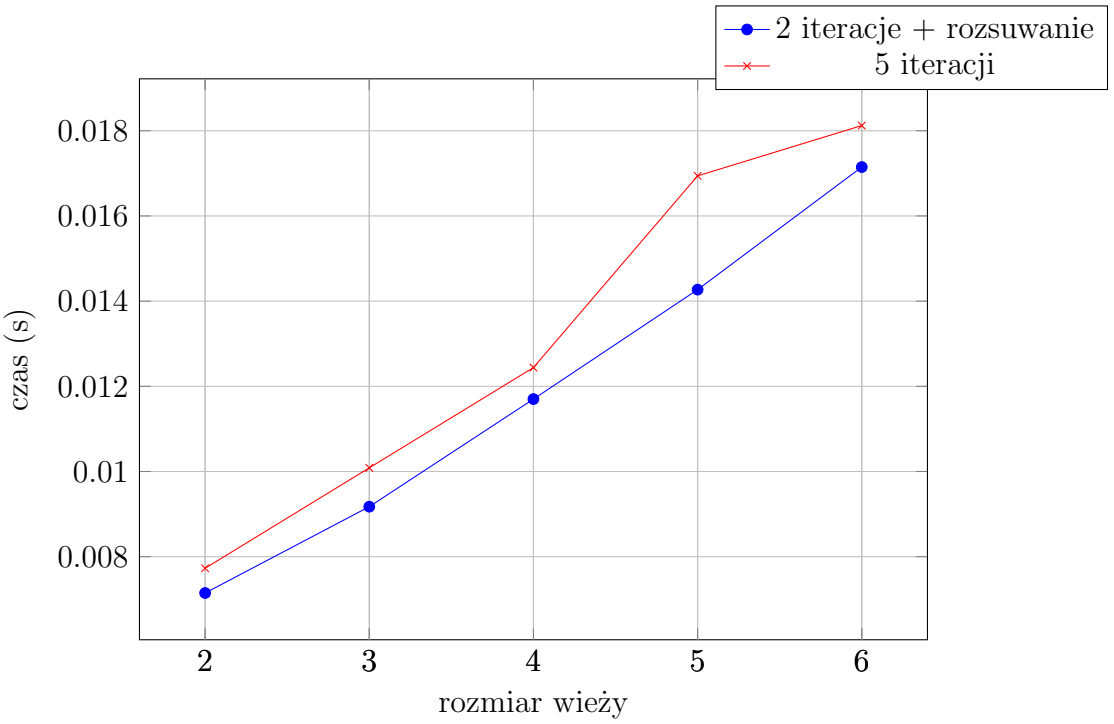
RYSUNEK 3.3: Wynik symulacji sceny przez 2000 kroków. Od lewej: dwie iteracje spoczynkowe, pięć iteracji spoczynkowych i dwie iteracje spoczynkowe wzbogacone o rozsuwanie.

$$pos'_B = pos_B - n^{B \to A} * (m_B/(m_A + m_B))$$

gdzie $m_A$ i $m_B$ to masy odpowiednio bryły $A$ i $B$, $pos_A$ i $pos_B$ to ich pozycje przed rozsunięciem a $pos'_A$ i $pos'_B$ to pozycje po rozsunięciu.

Okazuje się jednak, że przemieszczanie brył o pełną głębokość penetracji skutkuje w bardzo gwałtowne, nierealistyczne ruchy. Dlatego postanowiłem w każdym kroku symulacji przemieszczać je jedynie o pewien ułamek tej wielkości. W ten sposób bryły rozsuwają się stopniowo, na przestrzeni kilku następujących po sobie kroków symulacji, a proces symulacji kolizji i spoczynku ma szanse zareagować na zmiany w położeniu obiektów. Najbardziej wiarygodne wyniki dawał ułamek wynoszący poniżej 0.1, jednak wartości mniejsze od 0.01 skutkują w rozsuwanie zbyt niewielkie by robić istotną różnicę dla symulacji.

Przeprowadziłem testy mające na celu określić jaki wpływ na realizm i wydajność symulacji może mieć wprowadzenie etapu rozsuwania brył. Dodając ten etap jednocześnie zmniejszałem liczbę iteracji w etapie rozwiązywania spoczynku i przeprowadzałem symulację sceny składającej się z pięciu spoczywających na sobie pudełek ułożonych w wieżę.
Metodą prób i błędów, subiektywnie oceniając realizm symulacji doszedłem do wniosku, że symulacja wzbogacona o rozsuwanie potrzebuje jedynie dwóch iteracji (w tym jednej iteracji propagacji uderzeniowej) by osiągnąć efekt zbliżony do symulacji wykonującej pięć iteracji. Jednocześnie są to najmniejsze liczby iteracji (odpowiednio dla symulacji z i bez rozsuwania) zapewniające realizm i stabilność symulowanej sceny.




Następnie przeprowadziłem porównanie średniego czasu wykonywania jednego kroku symulacji spoczynku brył dla obu wariantów (dwóch iteracji z rozsuwaniem i pięciu iteracji bez rozsuwania). Dla pięciu różnych rozmiarów wieży klocków (od dwóch do sześciu brył) zmierzyłem średni czas obliczony z tysiąca kolejnych kroków. Wyniki znajdują się na załączonym wykresie.

Jak widać zysk z zastosowania rozsuwania, choć niewielki, jest zauważalny i wynosi w rozpatrywanych przypadkach średnio 9%

# Rozdział 4

# Środowisko rozproszone

Wirtualne światy symulacji fizycznych są często dzielone pomiędzy wieloma użytkownikami, najczęściej w postaci sieciowych gier wideo wykorzystujących protokół IP do komunikacji. Każdy z użytkowników posiada wówczas na swojej maszynie lokalną kopię symulowanego świata a zadaniem symulacji jest zapewnienie, że kopie są do siebie możliwe najpodobniejsze, by stworzyć dla użytkowników iluzję, że interagują oni z jedną wspólną wirtualną rzeczywistością.
Niezależnie jednak od protokołu, komunikacja na odległość musi wiązać się z opóźnieniami w dostarczaniu informacji. Jednocześnie interaktywność symulacji oznacza, że jeden lub więcej z użytkowników posiada możliwość ingerencji w symulacje w czasie rzeczywistym, zaburzając jej przebieg. Z tych dwóch faktów rodzi się problem — jak utrzymać kopie w stanie zsynchronizowanym gdy informacje o zachodzących w niej zmianach docierają do użytkowników z dużym opóźnieniem.

## 4.1 Model serwer-klient

Synchronizację symulacji znacznie ułatwia wyróżnienie jednego z użytkowników jako posiadacza "właściwej"kopii wirtualnego świata. Użytkownik taki, nazywany serwerem, informuje swoich klientów (pozostałych użytkowników) o zmianach zachodzących w symulacji a także odbiera od nich informacje o próbach wprowadzenia przez nich zmian. Klienci nie komunikują się pomiędzy sobą, każda komunikacja zachodzi pomiędzy serwerem i jednym z klientów (van Waveren [7]).
W takim modelu zadaniem jest utrzymanie przebiegu symulacji lokalnej kopii każdego klienta jak najbliżej przebiegu symulacji serwera. Serwer zawsze jako pierwszy

rozpoczyna symulację, a kolejni łączący się z nim klienci synchronizują z nim swoje lokalne kopie świata.

## 4.2 Spójność symulacji

Aby możliwa była równoległa, zsynchronizowana symulacja na wielu niezależnych maszynach przy zachowaniu minimalnej komunikacji konieczne jest by symulacja była w pełni deterministyczna (van Waveren [7]). We wcześniejszych rozdziałach mowa była o długości kroku symulacji i czasie symulacji. Oba wyrażane były w sekundach a długość kroku mogła być dowolna, w szczególności zmieniać się w trakcie symulacji. Jednak ze względu na niedokładność numerycznych rozwiązań równań różniczkowych ruchu obiektów będą one różnić się między sobą w zależności od dobranej długości kroku. W efekcie dwie symulacje o tych samych warunkach początkowych ale różnych krokach będą różnić się swoim przebiegiem.
Aby temu zapobiec, konieczne jest ustalenie stałej długości kroku, wspólnej dla wszystkich lokalnych kopii. Mierzenie czasu symulacji i datowanie nim istotnych wydarzeń także staje się dzięki temu prostsze. Zamiast mierzyć symulowane sekundy wystarczy liczyć kolejne kroki.

Dodatkowym, mniej oczywistym problemem w utrzymaniu spójności symulacji jest kwestia precyzji wykorzystywanych typów zmiennoprzecinkowych. Jeśli będzie ona różnić się pomiędzy kopiami, różnić się będą także wyniki obliczeń, nawet dla identycznych danych wejściowych. Konieczne jest zapewnienie, że każda lokalna kopia operuje na takich samych typach danych i w dokładnie ten sam sposób.

## 4.3 Metoda predykcji i korekty

Sposób w jaki lokalne kopie symulacji są utrzymywane w stanie zsynchronizowanym jest określany mianem metody predykcji i korekty. Zarówno serwer jak i klienci nie posiadają pełnej wiedzy na temat obecnego stanu symulacji ze względu na opóźnienia w przekazywaniu informacji. Informacja o zmianie w symulacji, która zaszła w danym momencie, dotrze do nich w przyszłości. Z tego powodu prowadzenie symulacji nazywane jest predykcją (van Waveren [7]), kolejne kroki symulacji są przewidywane na podstawie aktualnej wiedzy. W chwili gdy informacja o pewnej pominiętej zmianie dotrze do klienta lub serwera musi on dokonać

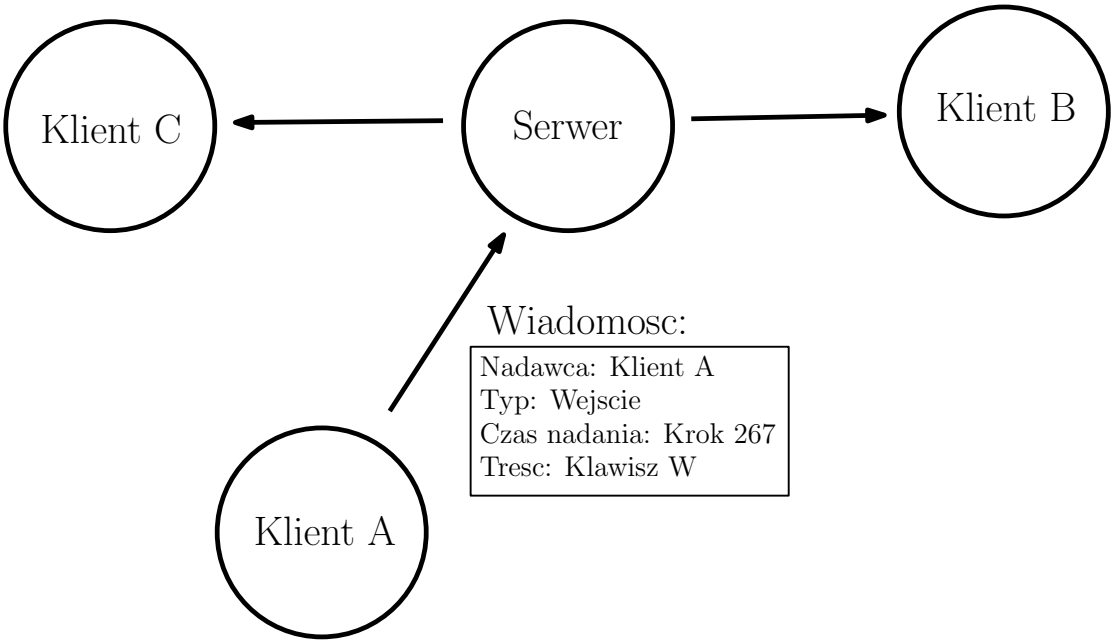
korekty, czyli cofnąć swoją lokalną kopię w czasie do chwili wysłania tej informacji, zaaplikować zmianę, a następnie ponownie przeprowadzić cofniętą symulację.
Metoda predykcji zachowania się obiektów na podstawie ich obecnego stanu (łącznie z prędkościami, przyspieszeniami i działającymi na niego siłami) nazywana jest metodą *dead-reckoning* (Smed [8]).

### 4.3.1 Komunikacja

Ponieważ cofanie symulacji jest kosztowną operacją istotne jest zminimalizowanie opóźnień w dostarczaniu informacji. Najprostszym na to sposobem jest ograniczenie ilości przesyłanych informacji do koniecznego minimum. Aby komunikacja pomiędzy serwerem i klientami pozwalała na synchronizację kopii symulacji konieczne są dwa rodzaje wiadomości (van Waveren [7]).

Pierwszy to synchronizacja początkowa, jest to wiadomość wysyłana od serwera do klienta w chwili gdy ten drugi nawiąże połączenie. Wiadomość ta zawiera informację o wszystkich biorących udział w symulacji obiektach i ich aktualnych stanach a także aktualny czas symulacji. Po otrzymaniu takiej wiadomości klient ustawia czas swojej lokalnej kopii na odebrany czas symulacji i wypełnia ją obiektami o odpowiednich stanach. Po odebraniu tej wiadomości symulacje serwera i klienta będą przebiegały w identyczny sposób aż do momentu, gdy jeden z użytkowników wprowadzi w symulacji zmianę.

Drugi rodzaj wiadomości to informacja o wprowadzonych danych wejściowych. Zmiany wprowadzane w symulacji najczęściej mają postać sił przykładanych do znajdujących się w niej brył, ale mogą także polegać na dodawaniu lub usuwaniu obiektów lub dowolnej modyfikacji ich stanu.
Opisanie tych zmian bezpośrednio może okazać się bardzo skomplikowane a istotne jest minimalizowanie rozmiaru przekazywanych przez sieć danych. Z tego względu wygodniejsze jest przekazanie wejścia użytkownika, które spowodowało te zmiany, na przykład informacji o naciśniętych klawiszach. Każda kopia symulacji powinna wiedzieć w taki sposób dane wejście wpływa na jej przebieg.
Ten rodzaj wiadomości może być wysyłany i odbierany zarówno przez serwer jak i klientów. Poza informacją o wejściu użytkownika musi ona zawierać także czas symulacji, w którym zostało ono wprowadzone a także, w przypadku gdy ma to znaczenie, informację o tym, który użytkownik je wprowadził.
Klient lub serwer po odebraniu tych informacji musi wycofać swoją kopię do chwili




Rysunek 4.1: Klient A wysyła do serwera informację o wprowadzonym przez użytkownika wejściu wraz z czasem nadania. Serwer rozsyła tę informację do swoich pozostałych klientów w niezmienionej postaci.

wysłania wiadomości, dokonać zmian wynikających z danych wejściowych a następnie powtórnie wykonać cofnięte kroki (van Waveren [7]). Dodatkowo w przypadku serwera rozsyła on tę wiadomość do wszystkich klientów poza jej nadawcą, dzięki czemu wszystkie kopie wprowadzą zaistniałą zmianę.

### 4.3.2 Czas i cofanie symulacji

Uwzględnienie wydarzenia, które miało miejsce w kroku wcześniejszym niż ten obecnie symulowany wymaga wycofania symulacji. Odwrócenie wszystkich wykonanych operacji — ruchu obiektów i ich kolizji — jest nie tylko bardzo kosztowne, ale też trudno zagwarantować, że pozwoli przywrócić całość symulacji do stanu identycznego jak przed wykonaniem cofanych kroków. Lepszym rozwiązaniem jest przechowywanie bufora przeszłych stanów dla każdego symulowanego obiektu. Bufor może mieć postać stosu na który odkładamy stany po wykonaniu kroku symulacji, i z którego te stany ściągamy cofając się.
Ponieważ czas mierzony jest w wykonanych krokach, wystarczy od aktualnego czasu lokalnej kopii symulacji odjąć czas otrzymany w wiadomości by otrzymać liczbę kroków o jaką należy się cofnąć przed zastosowaniem zakomunikowanej zmiany. Następnie przed przywróceniem normalnego toku symulacji ta sama liczba

kroków jest nadrabiana, czyli wykonywana ponownie z uwzględnieniem zaszłej zmiany.

Ponieważ opóźnienia w komunikacji pomiędzy użytkownikami mogą ulegać ciągłym zmianom, może zdarzyć się, że kiedy początkowo duże opóźnienie znacznie się zmniejszy czas przychodzących wiadomości będzie późniejszy niż obecny czas lokalnej kopii. Konieczne jest wówczas zapamiętanie takiej wiadomości i odczekanie, aż lokalny czas zrówna się z jej czasem. Wówczas można zaaplikować związaną z nią zmianę bez cofania symulacji.

Istnieje pewien szczególny przypadek, który musi zostać rozpatrzony by cofanie i powtarzanie symulacji wprowadzało jedynie zmiany wynikające z otrzymanej wiadomości. Jeśli pomiędzy aktualnym czasem symulacji i czasem wysłania wiadomości lokalna kopia otrzymała inne dane wejściowe (w postaci wiadomości od innego użytkownika lub bezpośrednio z urządzenia wejściowego) to po cofnięciu i powtórzeniu symulacji ich wpływ zostanie utracony. Dzieje się tak ponieważ cofając wykonane kroki cofamy także zmiany wprowadzone w ich trakcie.
Aby temu zapobiec poza stosem stanów obiektów potrzebny jest także bufor wejścia. Dla każdego kroku symulacji zapamiętanego na stosach obiektów zapamiętywane jest także wejście jakie zostało wprowadzone w tym kroku. W chwili powtarzania cofniętego kroku efekt wprowadzonego w nim wejścia jest także powtarzany. Wejście musi być zapamiętywane w buforze za każdym razem gdy jest wprowadzane lub otrzymywane w wiadomości. Ponieważ nie zawsze będziemy otrzymywać je w porządku chronologicznym (na przykład wejście otrzymane w wiadomości może być starsze niż ostatnie wejście lokalnego użytkownika) stos nie jest w tym przypadku dobrym rozwiązaniem. Lepsza może okazać się lista, która pozwoli na wstawianie elementów w środek i na sekwencyjny dostęp do elementów.

Poniższy fragment kodu przestawia przykładową implementację bufora przeszłych stanów dla brył sztywnych reprezentowanych przez klasę **RBState**.
Dwie przedstawione metody to **PushState**, odkładająca na szczyt stosu bufora obecny stan bryły, i **RevertState** cofająca stan bryły o pewną dodatnią liczbę kroków. Metoda **State** zwraca obecny stan obiektu a stała **STATE_LIST_LEN** ogranicza długość bufora stanów.

```
1 void RBody::PushState()
2 {
3   past_states.push_back(State());
```

```
4     if (past_states.size() > STATE_LIST_LEN)
5       past_states.pop_front();
6   }
7
8   void RBody::RevertState(int backward_count)
9   {
10    if (backward_count <= 0)
11      return;
12
13    for (int i = 0; i < backward_count-1; i++)
14      past_states.pop_back();
15
16    ApplyState(past_states.back());
17    past_states.pop_back();
18  }
```

## 4.4 Poprawność

Ze względu na opóźnienia w przekazywaniu informacji pomiędzy lokalnymi kopiami i możliwość pojawienia się nowych danych wejściowych ze strony dowolnego klienta w każdej chwili, kopie przez znaczącą część czasu symulacji nie znajdują się w stanie zsynchronizowanym, a jedynie w stanie do takiej synchronizacji dążącym. Dodatkowo zegar symulacji każdej z tych kopii może być przesunięty względem zegara serwera o różną niezerową wartość. W efekcie pojawia się problem z definicją poprawnego przebiegu symulacji. Nie można z pewnością stwierdzić czy przebieg rozproszonej symulacji fizycznej jest poprawny dopóki się on nie zakończy, a nawet wówczas należy przyjąć i wziąć pod uwagę pewne górne ograniczenie na opóźnienie w propagacji informacji pomiędzy użytkownikami a także maksymalne przesunięcie zegarów pomiędzy nimi.

Jeśli opóźnienie może osiągnąć wartość $n$ sekund a maksymalne przesunięcie $m$ sekund, to pomiędzy wprowadzeniem ostatnich danych wejściowych a testem poprawności musi upłynąć minimum $n + m$ sekund aby informacje o tych danych miały szanse rozpropagować się. W przypadku modelu klient-serwer, jeśli $n$ to maksymalne opóźnienie pomiędzy dowolnym klientem a serwerem, to czas pomiędzy ostatnimi danymi a testem musi wynosić minimum $2n + m$ ze względu na

konieczność przekazania informacji pomiędzy klientem a serwerem a następnie pomiędzy serwerem a pozostałymi klientami. Zakładamy tutaj oczywiście, że upływ czasu symulacji odpowiada dokładnie upływowi czasu rzeczywistego.

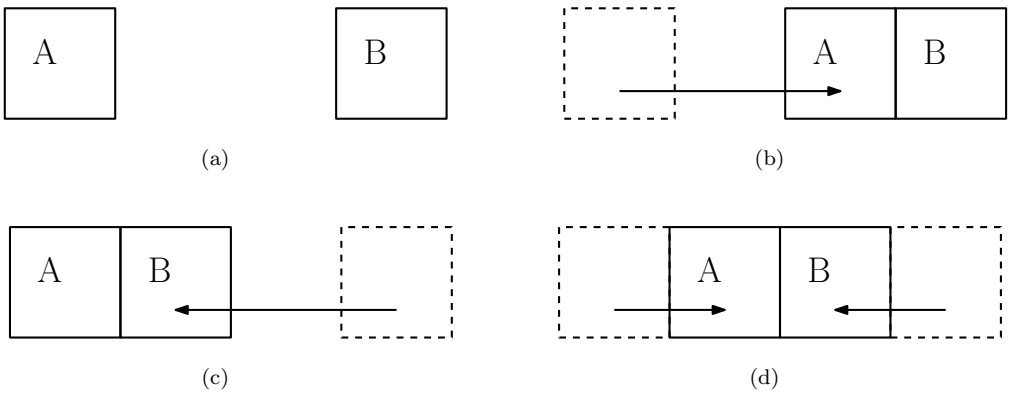
Przy zachowaniu powyższego warunku, przebieg rozproszonej symulacji możemy uznać za poprawny jeśli dla każdego momentu symulacji spełniającego powyższy warunek stany wszystkich obiektów w każdej lokalnej kopii są identyczne w odpowiadających sobie krokach symulacji.

Przedstawiona wcześniej metoda predykcji-korekty gwarantuje poprawność przy zachowaniu pewnych warunków:

- Nieograniczony stos przeszłych stanów
- Nieograniczony bufor danych wejściowych
- Gwarancja propagacji informacji
- Gwarancja pełnego determinizmu symulacji

Niestety w praktyce jedynie dwa ostatnie warunki są możliwe do spełnienia. Gwarancję propagacji informacji można użyskać na przykład w przypadku użycia niezawodnego protokołu komunikacyjnego TCP, który gwarantuje dostarczenie wysłanych pakietów do adresata. Nie daje on jednak żadnego górnego ograniczenia na czas potrzebny na ich dostarczenie.

W praktycznych implementacjach rozproszonych symulacji najczęściej tworząc je przyjmuje się pewne ograniczenia, w ramach których symulacja będzie poprawna. Dotyczą one maksymalnego czasu propagacji informacji (z którego wynika konieczny rozmiar stosu przeszłych stanów) i częstotliwości z jaką mogą pojawiać się nowe dane wejściowe.
Zakładając, że symulacja wykonuje $N$ kroków na sekundę, a górne ograniczenie na czas propagacji informacji wynosi $m$ sekund, to rozmiar stosu stanów powinien wynosić $mN$. Taki sam rozmiar powinien mieć bufor danych wejściowych, zakładając, że wiele danych pojawiających się w tym samym kroku symulacji (na przykład kombinacje wciśniętych klawiszy) może być przechowywanych w jednym polu bufora.




Rysunek 4.2: Kolizja pomiędzy dwoma zbliżającymi się do siebie bryłami może zajść w różnych punktach w zależności od tego, które komunikaty zdążyły dotrzeć do lokalnej kopii symulacji.

### 4.4.1 Kolejność docierania komunikatów

Kolejność w jakiej informacje będą docierać do serwera i do jego klientów ma oczywiście znaczenie dla ich lokalnego przebiegu symulacji. Jednak dla dowolnego momentu symulacji, dla którego możliwe jest ustalenie jej poprawności, kolejność w jakiej docierały wcześniejsze komunikaty nie ma wpływu na stan symulacji. Dzieje się tak dzięki metodzie predykcji-korekty, która po otrzymaniu każdej wiadomości cofa symulację do chwili jej wysłania w celu skorygowania przebiegu.

Rozważmy przykład dwóch nieruchomych brył $A$ i $B$. Początkowo ich stan odpowiada rysunkowi 4.2(a). Jeśli w tym samym kroku symulacji klient $K_A$ przyłoży do bryły $A$ siłę skierowaną w prawo a klient $K_B$ do bryły $B$ siłę skierowaną w lewo, obaj wyślą do serwera komunikat o wprowadzonych przez siebie zmianach. Jednak w zależności od tego, który z nich zostanie dostarczony jako pierwszy, symulacja lokalnej kopii serwera będzie przebiegać inaczej. Rysunek 4.2(b) przedstawia stan, jaki osiągnie symulacja jeśli komunikat od klienta $K_A$ dotrze w pierwszej kolejności, bryła $A$ poruszy się, ale była $B$ pozostanie na miejscu. Analogicznie rysunek 4.2(c) przedstawia sytuację, w której informacja od klienta $K_B$ została dostarczona pierwsza.
Niezależnie jednak od kolejności dostarczania wiadomości, w chwili gdy oba komunikaty dotrą do serwera dokona on korekty i stan jego lokalnej kopii symulacji będzie zgodnie z oczekiwaniami odpowiadał rysunkowi 4.2(d).

# Rozdział 5

# Przykładowa implementacja

## 5.1 Opis implementacji

Do pracy dołączona została przykładowa implementacja rozproszonej symulacji fizycznej. Została ona wykorzystana do przeprowadzenia testów i porównań z rozdziału 3.3.
Aby przebieg symulacji był deterministyczny, i tym samym by możliwa była synchronizacja w rozproszonym środowisku, zastosowałem stałą długość kroku wynoszącą $40ms$. Do wykrywania kolizji w fazie wąskiej użyłem metody płaszczyzny rozdzielającej a w szerokiej sfer ograniczających. Dodatkowo wprowadziłem do etapu wykrywania kolizji optymalizacje opisane w dziale 2.1.4.
Do ustalenia punktów kolizji wykorzystałem metodę rozwiązującą wiele kontaktów jednocześnie, co znacznie upraszcza strukturę programu w przypadku przyjętego przeze mnie stałego kroku symulacji. Także w tym przypadku zaimplementowałem opisane przez siebie w rozdziale 2.2.3 optymalizacje.
Do napisania części związanej z komunikacją w środowisku rozproszonym wykorzystałem bibliotekę ENet, która bazując na protokole sieciowym UDP tworzy wysokopoziomowy protokół komunikacji gwarantujący dostarczenie pakietów przez ponawianie próby ich wysłania aż do otrzymania potwierdzenia zwrotnego. Dzięki niej byłem w stanie zbudować model klient-serwer o stosunkowo (w porównaniu z protokołem TCP) niskim narzucie komunikacyjnym i szybkich czasach dostarczania pakietów zachowując niezawodność komunikacji.
Rozmiar stosu przeszłych stanów wynosi 50 pozycji, co dla kroku symulacji długości $40ms$ gwarantuje poprawność dla czasów propagacji informacji pomiędzy

klientami nie przekraczających dwóch sekund.
Reprezentacja graficzna symulacji została zaprogramowana z użyciem biblioteki OpenGL.

## 5.2 Uruchamianie i obsługa programu

Program może zostać uruchomiony w dwóch trybach: serwera lub klienta. W trybie serwera program przyjmuje jako parametr plik tekstowy opisujący stan początkowy symulowanej sceny. Plik ten składa się z następujących po sobie opisów biorących udział w symulacji brył. Opisane są ich stałe właściwości fizyczne jak masa i kształt, a także ich stan początkowy składający się z pozycji, orientacji, prędkości i momentu obrotowego. Początkowe przyspieszenie kątowe i liniowe wynosi zawsze zero.
W trybie klienta program jako argument przyjmuje adres IP serwera, z którym będzie próbował nawiązać połączenie używając w tym celu portu 1234. Po ustaleniu łączności serwer wysyła do klienta kompletny opis obecnego stanu sceny wraz z numerem kroku symulacji, co pozwala klientowi zsynchronizować z serwerem swoja lokalna kopię symulowanego środowiska.

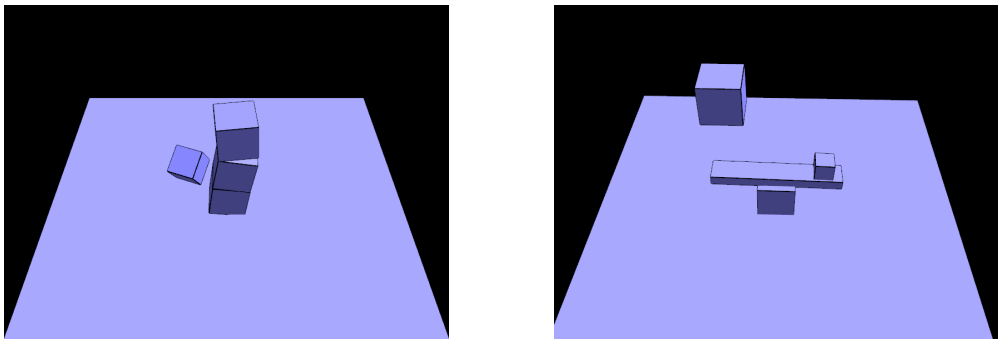
Aby skompilować program wystarczy wywołać polecenie **make** w katalogu z jego źródłami. Do kompilacji potrzebne są:

- make
- gcc
- ENet
- GLUT

Przykładowe wywołanie programu w trybie serwera, dane sceny zostaną wczytane z pliku **scene01**:

```
./implementacja h scene01
```

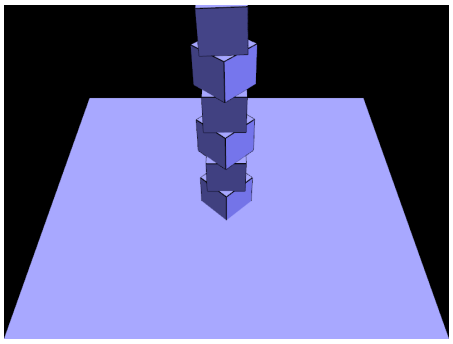
Przykładowe wywołanie w trybie klienta:



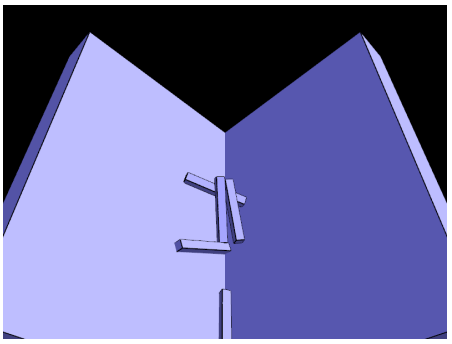
(a)



(b)



(c)



(d)

RYSUNEK 5.1: Cztery spośród przykładowych scen dołączonych do programu.
5.1(a) plik **scene02**, bryła uderzająca w wieżę
5.1(b) plik **scene04**, katapulta
5.1(c) plik **scene05**, wieża o wysokości sześciu brył
5.1(d) plik **scene06**, wąskie bryły i dwie pochylnie

```
./implementacja c 127.0.0.1
```

Do programu dołączonych jest kilka przykładowych scen w postaci plików tekstowych. Na obrazku 5.1 można zobaczyć część z nich.

Każdy z użytkowników, zarówno w trybie serwera jak i klienta, może kontrolować jedną z brył biorących udział w symulacji. Dla serwera będzie to pierwsza w kolejności bryła opisana w pliku sceny, a dla użytkownika dołączającego do symulacji jako $N$-ty klient będzie to bryła $N + 1$ w kolejności. W przypadku gdy użytkowników jest więcej niż brył biorących udział w symulacji, ostatni użytkownicy nie kontrolują żadnej z nich.
Kontrola nad bryłą odbywa się poprzez przykładanie do niej siły zwróconej w jednym z czterech kierunków świata i wynoszącej 20N. Każdy z klawiszy **'W'**, **'A'**, **'S'** i **'D'** odpowiada jednemu z kierunków a siła jest przykładana do środka

ciężkości bryły tak długo jak długo dany klawisz pozostaje wciśnięty.

Dodatkowo wciśnięcie klawisza **'Q'** powoduje zamknięcie programu, wciśnięcie spacji powoduje chwilowe wstrzymanie symulacji (ponowne wciśnięcie spowoduje jej wznowienie).
Po naciśnięciu lewego klawisza myszki wewnątrz okna symulacji możliwe jest obracanie sceny wokół jej środka poruszając myszą. Rolką myszki możliwe jest oddalanie i przybliżanie widoku kamery.

## 5.3 Format opisu scen

Stany początkowe symulowanych scen są odczytywane przez program z plików tekstowych o następującym formacie.

Opis sceny rozpoczyna się po pierwszym wystąpieniu znaku '~', wszystkie znaki go poprzedzające są ignorowane.

Opis składa się z oddzielonych od siebie dowolną liczbą białych znaków opisów brył, gdzie każdy opis bryły składa się z występujących kolejno i oddzielonych od siebie dowolną liczbą białych znaków właściwości:

1. Liczby zmiennoprzecinkowej określającej gęstość bryły w kilogramach na metr sześcienny
2. Trzech liczb zmiennoprzecinkowych określających kolejno szerokość, wysokość i głębokość bryły w metrach (dla uproszczenia program generuje jedynie prostopadłościany)
3. Ciągu znaków alfanumerycznych stanowiących nazwę bryły
4. Zmiennej przyjmującej wartość 0 lub 1 określającej czy bryła jest obiektem statycznym (wartość 1 oznacza obiekt statyczny)
5. Trzech liczb zmiennoprzecinkowych określających kolejne współrzędne położenia środka ciężkości bryły w metrach
6. Trzech liczb zmiennoprzecinkowych określających kolejne współrzędne prędkości bryły w metrach na sekundę (tylko w przypadku bryły niestatycznej)

7. Czterech liczb zmiennoprzecinkowych określających kolejne współrzędne kwaternionu orientacji bryły

8. Trzech liczb zmiennoprzecinkowych określających kolejne współrzędne wektora prędkości kątowej bryły. Kierunek wektora wyznacza oś obrotu a jego długość prędkość obrotu wyrażoną w radianach na sekundę (tylko w przypadku bryły niestatycznej)

Po ostatniej właściwości ostatniej bryły należącej do opisu występuje bezpośrednio znak **';'**.

# Bibliografia


[1] David Baraff. Physically based modeling: Rigid body simulation. Online SIGGRAPH 2001 Course Notes, 2001. URL http://www.pixar.com/companyinfo/research/pbm2001/pdf/notesg.pdf.

[2] Yi-King Choi, Xueqing Li, Wenping Wang, and Stephen Cameron. Collision detection of convex polyhedra based on duality transformation. *The University of Hong Kong*, 2005.

[3] Eran Guendelman, Robert Bridson, and Ronald Fedkiw. Nonconvex rigid bodies with stacking. *ACM Transactions on Graphics (TOG)*, Volume 22(3): 871–878, July 2003.

[4] David M. Bourg. *Physics for Game Developers*. O'Reilly, 2002.

[5] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *SIGGRAPH '94 Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 23–34, 1994.

[6] Jan Bender and Alfred Schmitt. Constraint-based collision and contact handling using impulses. In *Proceedings of the 19th International Conference on Computer Animation and Social Agents, Geneva (Switzerland)*, pages 3–11, 2006.

[7] J.M.P. van Waveren. The doom iii network architecture. id Software, 2006. URL http://mrelusive.com/publications/papers/The-DOOM-III-Network-Architecture.pdf.

[8] Jouni Smed, Timo Kaukoranta, and Harri Hakonen. A review on networking and multiplayer computer games. In *Multiplayer Computer Games, Proc. Int. Conf. on Application and Development of Computer Games in the 21st Century*, pages 1–5, 2002.

[9] J. Bender, K. Erleben, and J. Trinkle. Interactive simulation of rigid body dynamics in computer graphics. *Computer Graphics Forum*, 33(1):246–270, February 2014.