POLITECHNIKA GDAŃSKA
WYDZIAŁ FIZYKI TECHNICZNEJ I MATEMATYKI STOSOWANEJ

Imię i nazwisko studenta: Ewa Sobieniak
Nr albumu: 126438
Studia drugiego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Fizyka Techniczna
Specjalność/profil: Informatyka stosowana

PRACA DYPLOMOWA MAGISTERSKA

Tytuł pracy w języku polskim: Implementacja wybranego modelu zderzeń w grach dwuwymiarowych

Tytuł pracy w języku angielskim: Implementation of the selected collisions model in two-dimensional games

Potwierdzenie przyjęcia pracy	
Opiekun pracy *podpis*	Kierownik Katedry/Zakładu *podpis*
dr inż. Paweł Syty	

Data oddania pracy do dziekanatu:

POLITECHNIKA GDAŃSKA
WYDZIAŁ FIZYKI TECHNICZNEJ I MATEMATYKI STOSOWANEJ

OŚWIADCZENIE

Imię i nazwisko studenta: Ewa Sobieniak
Data i miejsce urodzenia: 23.01.1990, Gdańsk
Nr albumu: 126438
Wydział: Wydział Fizyki Technicznej i Matematyki Stosowanej
Kierunek studiów: Fizyka Techniczna
Poziom studiów: Studia drugiego stopnia
Forma studiów: stacjonarne

Ja, niżej podpisana, wyrażam zgodę na korzystanie z mojej pracy dyplomowej zatytułowanej: Implementacja wybranego modelu zderzeń w grach dwuwymiarowych do celów naukowych lub dydaktycznych.[1]

Gdańsk, dnia ....

podpis studenta

Świadoma odpowiedzialności karnej z tytułu naruszenia przepisów ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz. U. z 2006 r., nr 90, poz. 631) i konsekwencji dyscyplinarnych określonych w ustawie Prawo o szkolnictwie wyższym (Dz. U. z 2012 r., poz. 572 z późn. zm.),[2] a także odpowiedzialności cywilno-prawnej oświadczam, że przedkładana praca dyplomowa została opracowana przeze mnie samodzielnie.

Niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadaniem tytułu zawodowego.

Wszystkie informacje umieszczone w ww. pracy dyplomowej, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami zgodnie z art. 34 ustawy o prawie autorskim i prawach pokrewnych.

Potwierdzam zgodność niniejszej wersji pracy dyplomowej z załączoną wersją elektroniczną.

Gdańsk, dnia ....

podpis studenta

Upoważniam Politechnikę Gdańską do umieszczenia ww. pracy dyplomowej w wersji elektronicznej w otwartym, cyfrowym repozytorium instytucjonalnym Politechniki Gdańskiej oraz poddawania jej procesom weryfikacji i ochrony przed przywłaszczaniem jej autorstwa.

Gdańsk, dnia ....

podpis studenta

---

[1] Zarządzenie Rektora Politechniki Gdańskiej nr 34/2009 z 9 listopada 2009 r., załącznik nr 8 do instrukcji archiwalnej PG.

[2] Ustawa z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym:

Art. 214 ustęp 4. W razie podejrzenia popełnienia przez studenta czynu podlegającego na przypisaniu sobie autorstwa istotnego fragmentu lub innych elementów cudzego utworu rektor niezwłocznie poleca przeprowadzenie postępowania wyjaśniającego.

Art. 214 ustęp 6. Jeżeli w wyniku postępowania wyjaśniającego zebrany materiał potwierdza popełnienie czynu, o którym mowa w ust. 4, rektor wstrzymuje postępowanie o nadanie tytułu zawodowego do czasu wydania orzeczenia przez komisję dyscyplinarną oraz składa zawiadomienie o popełnieniu przestępstwa.

SPIS TREŚCI

WSTĘP

Celem pracy dyplomowej jest omówienie części teoretycznej zagadnienia zderzenia idealnie sprężystego i opracowanie opartego na niej systemu umożliwiającego przeprowadzanie symulacji dla zadanych parametrów. Proponowany system jest zaprojektowanym od postaw modelem o działaniu edukacyjnym. Aplikacja ma na celu możliwość przeprowadzania obliczeń oraz wizualizacji przebiegu zderzenia idealnie sprężystego w jednym z wielu proponowanych przypadków użycia na płaszczyźnie. Symulacja badanego układu skupia się na opisie ruchu i zachowań cząstek o kształcie okręgów na płaszczyźnie przy następujących założeniach: ciężar każdego z obiektów rozłożony jest równomiernie, a więc środek masy znajduje się w środku geometrycznym okręgu o zadanym promieniu. Płaszczyzna, przedstawiona jako pudło symulacyjne, ograniczona jest z czterech stron prostopadłymi ścianami o nieskończenie wielkiej masie. Wypadkowa sił zewnętrznych działających na układ jest zerowa (tzw. układ izolowany), a więc trajektoria ruchu cząstki może zmienić się jedynie w skutek zderzenia ze ścianą lub drugą cząstką. Dodatkowo żadna cząstka nie opuszcza układu, ani do niego nie przybywa (tzw. układ zamknięty). Założono również, że powierzchnie kul są idealnie gładkie, a więc pomiędzy obiektami nie występuje tarcie. Ze względu na powyższe fakty, pomiędzy obiektami występują zderzenia idealnie sprężyste. Z tej samej przyczyny, opis kinematyki bryły sztywnej sprowadza się do zagadnienia ruchu postępowego.

Interakcja użytkownika z aplikacją obejmuje obszar zadawania warunków początkowych dla danego układu - ilość kul, ich pozycje, masy i prędkości - oraz możliwość obserwacji parametrów wybranego obiektu w trakcie symulacji. System składa się z zarówno gotowych układów przedstawiających różne aspekty zderzenia idealnie sprężystego jak i specjalnego pustego układu, który użytkownik może modyfikować od samego początku poprzez dodawanie kolejnych kul. Proponowany model może mieć zastosowanie w badaniu dynamiki obiektów w wielu obszarach zarówno nauki - symulowanie ruchu gazu idealnego - jak i życia codziennego - zderzenia kul bilardowych, przy założeniu braku tarcia pomiędzy powierzchniami - jednak przede wszystkim pozwala zrozumieć zagadnienie zderzenia idealnie sprężystego.

W skład pracy wchodzi opis podstaw mechaniki klasycznej układu zamkniętego izolowanego - pierwsza i druga zasada dynamiki Newtona, zasada zachowania pędu oraz

zasada zachowania energii. W kolejnej części skupiono się na stworzeniu procedur zgodnie z którymi należy przeprowadzać detekcję, a także tych które opisują jak prędkość cząstek będzie zmieniać się w skutek zderzenia. Rozpatrzony został zarówno przypadek zderzenia cząstki z obiektem o nieskończenie wielkiej masie - przykładowo ściany pudła symulacyjnego - jak i zderzenia pomiędzy cząstkami w przypadku ogólnym, gdy cząstki różnią się masami oraz w przypadku szczególnym, gdy cząstki są identyczne.

Na podstawie opisanej części teoretycznej została zaprojektowana biblioteka silnika fizyki wraz z aplikacją graficzną z niej korzystającą. W skład pracy wchodzi pełna dokumentacja projektowa zawierająca raport wykonalności, wymagania funkcjonalne, przypadki użycia, diagram klas oraz diagram sekwencji dla algorytmu obliczania najbliższego zderzenia. Najciekawsze elementy implementacji przedstawiono w postaci fragmentów kodu wraz z opisem. Sposób realizacji projektu opisano w części zawierającej instrukcję obsługi systemu, gdzie wyjaśniono jak można generować własne układy lub modyfikować już istniejące. Silnik fizyki został napisany w języku C#, a aplikacja zrealizowana w środowisku Microsoft XNA Game Studio 4.0.

1 FIZYKA BADANEGO UKŁADU

1.1 Kinematyka i dynamika ruchu jednostajnego prostoliniowego

W celu zaprojektowania silnika fizyki badanego układu należy przede wszystkim zrozumieć jak poruszają się jego cząstki i zgodnie z którymi zasadami fizyki. Marta Skorko w swojej "Fizyce" napisała, że "przez ruch ciała rozumiemy zmianę jego położenia w stosunku do innych ciał, które uważamy za nieruchome" [1]. Zgodnie z pierwszą zasadą dynamiki, jeżeli na ciało nie działa żadna siła lub działające na nie siły równoważą się, obiekt pozostaje w spoczynku lub zmiana jego położenia następuje zgodnie z ruchem jednostajnym prostoliniowym. Biorąc pod uwagę powyższe fakty, cząstki w symulowanym układzie będą poruszać się ruchem opisanym przez równania:

$$v = \frac{ds}{dt} = const \tag{1.1}$$

$$s(t) = s_0 + vt \tag{1.2}$$

gdzie v, s, s_0 oznaczają odpowiednio prędkość, położenie w czasie *t* oraz położenie początkowe. Wzory (1.1) i (1.2) odnoszą się do wszystkich składowych wektorów prędkości i położeń obiektów.

1.2 Zasada zachowania pędu

Warto zauważyć, że druga zasada dynamiki mówi o tym, że wszelkie zmiany prędkości mogą zachodzić jedynie pod wpływem działania siły. "Siła [ta] jest proporcjonalna do przyspieszenia, które wywołuje" [1]. Stwierdzenie to można przedstawić za pomocą równania:

$$\boldsymbol{F} = m\boldsymbol{a} \tag{1.3}$$

gdzie $\boldsymbol{F}$ oznacza siłę, m masę, $\boldsymbol{a}$ przyspieszenie. Przedstawiając przyspieszenie z równania (1.3) jako iloraz różnic prędkości i czasu, wzór można przekształcić do postaci:

$$\boldsymbol{F}(t_2 - t_1) = m\boldsymbol{v_2} - m\boldsymbol{v_1} \tag{1.4}$$

Popęd siły, będący wektorem o kierunku zgodnym z kierunkiem wektora $\boldsymbol{F}$ definiujemy jako iloczyn siły i czasu jej działania. Z kolei iloczyn masy i prędkości

nazywamy pędem, który jest wektorem o kierunku zgodnym z wektorem prędkości $\boldsymbol{v}$. Dlatego też równanie (1.4) oznacza, że "wektor popędu siły jest równy wektorowemu przyrostowi pędu wywołanemu przez tę siłę" [1].

Chcąc znaleźć chwilową siłę w upływie czasu Δt zmierzającym do zera, okazuje się ona być pochodną pędu względem czasu:

$$\boldsymbol{F} = \frac{d(m\boldsymbol{v})}{dt} = \frac{d\boldsymbol{p}}{dt} \tag{1.5}$$

Wprowadzając założenia o układzie jako izolowanym i zamkniętym, działająca na niego wypadkowa siła $\boldsymbol{F}$ jest zerowa, a więc pochodna pędu po czasie również jest równa zeru czyli:

$$\boldsymbol{p} = const \tag{1.6}$$

Cytując opis z "Podstaw fizyki" Davida Hallidaya, oznacza to, że "jeżeli na układ cząstek nie działają siły zewnętrzne lub ich wypadkowa jest równa zeru, to całkowity pęd $\boldsymbol{p}$ układu nie ulega zmianie" [2]. Powyższe stwierdzenie nosi nazwę zasady zachowania pędu. Innymi słowy oznacza to, że całkowity pęd początkowy zamkniętego układu izolowanego jest równy całkowitemu pędowi końcowemu.

1.3 Zasada zachowania energii

W opisanym układzie spełniona jest również zasada zachowania energii. Dotyczy ona wszelkich odmian energii i mówi o tym, że zmiana całkowitej energii układu równa jest energii dostarczonej do układu lub odebranej. Wynika z tego, że w układzie izolowanym zamkniętym, całkowita energia układu pozostaje stała. Oznacza to, że wewnątrz układu mogą zachodzić jedynie przemiany energetyczne jednej energii w drugą - energia nie może być ani tworzona, ani niszczona.

W przypadku badanego układu, w którym zmienia się jedynie energia kinetyczna poszczególnych obiektów, zasadę zachowania energii można przedstawić za pomocą wzoru:

$$\Delta E_k = \frac{1}{2} m({v_2}^2 - {v_1}^2) = const \tag{1.7}$$

2 ZDERZENIE IDEALNIE SPRĘŻYSTE

Cząstki w układzie poruszają się ruchem opisanym za pomocą wzoru (1.2). Pomiędzy zderzeniami, w trakcie ruchu jednostajnego prostoliniowego spełnione jest również równanie (1.1). Jednak z punktu widzenia symulacji dużo ciekawszym aspektem jest kwestia tego, kiedy i jak obiekty będą zderzać się między sobą, bo to w trakcie zderzeń będzie zmieniać się wektor prędkości $\boldsymbol{v}$ opisujący ruch każdej z cząstek uczestniczących w zderzeniu. Dlatego też kluczowym fragmentem projektowania silnika fizyki symulatora jest rozważenie zagadnienia detekcji zderzeń oraz przekazywanej w trakcie zderzenia energii, a więc prędkości cząstek tuż po zderzeniu czyli odpowiedzi zderzenia. Zgodnie z definicją Davida M. Bourga, "detekcja zderzenia jest komputerowym problemem geometrycznym, którego rozwiązanie prowadzi do ustalenia, czy i gdzie nastąpiło zderzenie [...]. Odpowiedź zderzenia to problem fizyczny ruchu dwóch lub więcej obiektów" [3].

Głównym założeniem jest ograniczenie wszelkich zderzeń do zderzenia pomiędzy dwoma obiektami. W przypadku zderzeń wielu obiektów, zostaną one rozpatrzone po kolei jako sekwencja poszczególnych zderzeń dwóch obiektów. W symulacji występują dwa rodzaje zderzeń:

- zderzenie typu kula-kula
- zderzenie typu kula-ściana

Każdy z przypadków należy rozpatrzyć osobno.

2.1 Detekcja zderzenia

Detekcja kolejnego zderzenia polega na określeniu czasu, w którym ono nastąpi. Najprostszym algorytmem jest wyliczanie dla każdego kroku czasowego czy nastąpiło już zderzenie dla każdej pary kula-kula oraz kula-ściana. Rozwiązanie to ma złożoność obliczeniową rzędu $O(N^2)$ (dla N cząstek), a do tego wprowadza utrudnienia w postaci dobrania na tyle małego kroku czasowego aby nie pominąć czasu żadnego zderzenia, ale też na tyle dużego żeby być w stanie w każdym kroku wykonać serię obliczeń dla każdego obiektu. Z tego względu zaproponowano inny algorytm, który polega na wykonaniu poniższych kroków:

1) Dla i-tej kuli, gdzie $i \in < 1, N >$, znajdź czas, dla którego nastąpi zderzenie z każdą ze ścian, w kierunku których kula porusza się.

2) Następnie znajdź najmniejszą wartość czasu i zapamiętaj jako t_{min}.

3) Dla każdej pary kul, jeżeli nastąpi pomiędzy nimi zderzenie, oblicz czas zderzenia.

4) Znajdź najmniejszą wartość czasu i jeżeli jest mniejsza od t_{min}, ustaw nową wartość zmiennej.

Opis działania dla punktów 1) oraz 3) przedstawiono w kolejnych podrozdziałach.

Zaletą tego algorytmu jest brak potrzeby wykonywania zbędnych obliczeń dla kroków, w których zderzenie nie nastąpi i dokładna znajomość czasu, w którym najbliższe zderzenie będzie miało miejsce, dzięki czemu problem pominięcia kolizji zostaje wyeliminowany.

2.1.1 Detekcja zderzenia cząstki z obszarem granicznym pudła symulacyjnego

Ponieważ kule znajdują się w prostokątnym pudle o czterech różnych pionowych lub poziomych ścianach, w przypadku zderzenia typu kula-ściana należy określić, która ze ścian będzie uczestniczyć w zderzeniu. W tym celu można wykorzystać informację o kierunku ruchu kuli zawartą w jej wektorze prędkości. Ponieważ symulacja jest wyświetlana na monitorze, układ współrzędnych został przyjęty zgodnie z układem ekranu: jeżeli składowa x-owa prędkości kuli jest dodatnia, kula porusza się w kierunku prawej ściany. Jeżeli jest mniejsza od zera, w kierunku lewej. Z kolei w przypadku składowej y-owej, dodatnia wartość oznacza ruch w kierunku ściany dolnej, a ujemna w kierunku ściany górnej.

Dysponując powyższą informacją można określić, z którą ze ścian nastąpi kolejne zderzenie. W kolejnym kroku należy wyliczyć czas zderzenia. Przykładowo dla zderzenia ze ścianą dolną, jeżeli kula o promieniu r znajduje się w odległości s od krawędzi ściany (mierząc od środka kuli), a jej prędkość wynosi:

$$\boldsymbol{v} = [v_x, v_y] \tag{2.1}$$

Cząstka poruszając się ruchem jednostajnym prostoliniowym pokona dystans do dolnej ściany:

$$s - r = v_y t \tag{2.2}$$

w czasie:

$$t = \frac{s - r}{v_y} \tag{2.3}$$

2.1.2 Detekcja zderzenia dwóch cząstek

W przypadku dwóch kul o prędkościach $\boldsymbol{v_1}$, $\boldsymbol{v_2}$ ich zderzenie nastąpi w momencie, gdy odległość s pomiędzy środkami kul będzie równa sumie ich promieni. Oznaczając współrzędne środków okręgów jako (x_1, y_1) i (x_2, y_2) oraz korzystając z faktu, że poruszają się ruchem jednostajnym prostoliniowym (zgodnie ze wzorem (1.2)), dystans s można określić jako:

$$s = r_1 + r_2 = \sqrt{((x_2 + v_{2,x}t) - (x_1 + v_{1,x}t))^2 + ((y_2 + v_{2,y}t) - (y_1 + v_{1,y}t))^2} \tag{2.4}$$

Równanie (2.4) można doprowadzić do postaci równania kwadratowego względem niewiadomego czasu t:

$$\begin{aligned} 0 = t^2 \left[\left(v_{x,2} - v_{x,1}\right)^2 + \left(v_{y,2} - v_{y,1}\right)^2 \right] \\ +2t\left[(x_2 - x_1)\left(v_{x,2} - v_{x,1}\right) + (y_2 - y_1)\left(v_{y,2} - v_{y,1}\right) \right] \\ +(x_2 - x_1)^2 + (y_2 - y_1)^2 - (r_1 + r_2)^2 \end{aligned} \tag{2.5}$$

Rozwiązując równanie (2.5) względem niewiadomej t należy rozpatrzyć tylko te przypadki, w których $\Delta \gg 0$, ponieważ w przeciwnym razie kule nie zderzą się. Ponieważ szukamy najbliższego, ale równocześnie kolejnego zderzenia, z wyliczonych czasów t_1, t_2 należy wybrać mniejszy, ale równocześnie większy od zera.

2.2 Odpowiedź zderzenia

Znając czas kolejnego zderzenia należy rozpatrzyć jak tuż przed, w jego trakcie oraz tuż po zderzeniu zachowają się obiekty w nim uczestniczące. Czym właściwie jest zderzenie? Potocznie mówiąc są to wszelkiego rodzaje kolizje pomiędzy obiektami. Chcąc podać bardziej precyzyjną definicję można powiedzieć, że "zderzenie zachodzi wtedy, gdy dwa lub więcej ciał (partnerów zderzenia) działa na siebie stosunkowo dużymi siłami w stosunkowo krótkim przedziale czasu" [2]. Zderzenie sprężyste jest tego typu zderzeniem, w którym spełniona jest zasada zachowania energii. Należy pamiętać, że w przypadku badanego układu dodatkowo spełniona jest również zasada zachowania pędu. Gdy energia nie jest zachowana mamy do czynienia z tak zwanym zderzeniem niesprężystym. W życiu codziennym często

spotykamy się ze zderzeniami w przybliżeniu sprężystymi. Przykładem takiego zderzenia jest kolizja dwóch kul bilardowych, w skutek której bardzo mała, praktycznie pomijalna część energii jest przekazywana w postaci fali dźwiękowej towarzyszącej hukowi przy zderzeniu. Przeważnie jednak ilość oddanej energii jest pomijalnie mała i zderzenie można uznać za niemal sprężyste. W rozważaniach pomijane jest również tarcie pomiędzy powierzchniami obiektów uczestniczących w zderzeniu, a więc współczynnik resuscytacji zderzenia jest równy jedności.

Zderzenie można również podzielić ze względu na kierunek ruchu obiektów przed i po kolizji. Zderzenie centralne ma miejsce, gdy wszystkie ciała uczestniczące w zderzeniu poruszają się przed i po zderzeniu wzdłuż tej samej prostej - krótko mówiąc, jest to przypadek jednowymiarowy. Oznacza to, że wektory prędkości obiektów przed i po zderzeniu układają się wzdłuż jednej prostej. Z kolei zderzenie niecentralne ma miejsce wtedy, gdy obiekty po zderzeniu poruszają się w innych kierunkach niż przed - a więc przypadek na płaszczyźnie.

Aby zrozumieć istotę zderzenia sprężystego w dwóch wymiarach najpierw należy przeprowadzić analizę dla analogicznej kolizji w jednym wymiarze, które sprowadza się do przypadku zderzenia centralnego. Dwie kule o masach m_1, m_2 i prędkościach v_1, v_2 poruszają się wzdłuż tej samej prostej (ze względu na rozważania w jednym wymiarze, prędkość nie jest już wektorem, a skalarem). Gdy dochodzi do kolizji, prędkości po zderzeniu v_1', v_2' możemy wyliczyć korzystając z zasady zachowania pędu:

$$m_1 v_1 + m_2 v_2 = m_1 v_1' + m_2 v_2' \tag{2.6}$$

Może zmienić się energia kinetyczna poszczególnych obiektów, ale sumaryczna wartość układu musi być zachowana zgodnie z zasadą zachowania energii:

$$\frac{1}{2} m_1 {v_1}^2 + \frac{1}{2} m_2 {v_2}^2 = \frac{1}{2} m_1 {v_1}'^2 + \frac{1}{2} m_2 {v_2}'^2 \tag{2.7}$$

Aby uzyskać wartości prędkości tuż po zderzeniu, należy rozwiązać układ równań (2.6)-(2.7) względem v_1' i v_2', podstawiając za v_1' i v_2' kolejno:

$$v_1' = \frac{m_1 v_1 + m_2 v_2 - m_2 v_2'}{m_1} \tag{2.8}$$

$$v_2' = \frac{m_1 v_1 + m_2 v_2 - m_1 v_1'}{m_2} \tag{2.9}$$

Ostatecznie otrzymamy dwie prędkości v_1', v_2' po zderzeniu opisane wzorami:

$$v_1{}' = \frac{(m_1 - m_2)v_1 + 2m_2 v_2}{m_1 + m_2} \tag{2.10}$$

$$v_2{}' = \frac{-(m_1 - m_2)v_2 + 2m_1 v_1}{m_1 + m_2} \tag{2.11}$$

2.2.1 Przypadek ogólny zderzenia typu kula-kula

W przypadku ruchu kul na płaszczyźnie, ruch musi być opisany przy pomocy wektorów przestrzeni $\mathbb{R}^2$. Mając dane wektory prędkości przed zderzeniem określone zgodnie ze wzorem (2.1) jako:

$$\boldsymbol{v_1} = [v_{1,x}, v_{1,y}]$$
$$\boldsymbol{v_2} = [v_{2,x}, v_{2,y}]$$

wektory po zderzeniu określamy analogicznie:

$$\boldsymbol{v_1}' = [v_{1,x}{}', v_{1,y}{}']$$
$$\boldsymbol{v_2}' = [v_{2,x}{}', v_{2,y}{}']$$

mając na uwadze ich zależność od wektorów przed zderzeniem.

Celem jest obliczenie wektorów po zderzeniu. Jednym ze sposobów rozwiązania tego problemu jest wzięcie pod uwagę punktu styku kul w momencie zderzenia oraz rozłożeniu wektorów prędkości w kierunku:

- normalnym
- stycznym

do powierzchni stykających się kul. Oznaczając współrzędne środków okręgów jako (x_1, y_1) i (x_2, y_2), wektor normalny jednostkowy do okręgu pierwszego w punkcie styczności będzie miał postać:

$$\boldsymbol{n} = [n_x, n_y] = \frac{[x_2 - x_1,\ y_2 - y_1]}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}} \tag{2.12}$$

natomiast wektor styczny:

$$\boldsymbol{t} = [t_x, t_y] = [-n_y, n_x] = \frac{[-(y_2 - y_1),\ x_2 - x_1]}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}} \tag{2.13}$$

Zakładając, że powierzchnie obu kul są idealnie gładkie, a więc nie występuje pomiędzy nimi tarcie, możemy wyjść z założenia, że po zderzeniu zmianie ulegną jedynie składowe prędkości normalne. Możemy więc potraktować zachowanie składowych normalnych jak w przypadku zderzenia jednowymiarowego. Składowe styczne pozostaną bez zmian.

Obliczenia należy rozpocząć od wykonania rzutów wektorów prędkości $\boldsymbol{v_1}'$, $\boldsymbol{v_2}'$ na osie lokalnego układu współrzędnych wyznaczonego przez wektor normalny i styczny $\boldsymbol{n}$, $\boldsymbol{t}$:

$$\begin{cases} v_{i,n} = \boldsymbol{v_i} \circ \boldsymbol{n} \\ v_{i,t} = \boldsymbol{v_i} \circ \boldsymbol{t} \end{cases} \quad i \in \{1,2\} \tag{2.14}$$

gdzie $\circ$ oznacza iloczyn skalarny, a więc:

$$v_{1,n} = \boldsymbol{v_1} \circ \boldsymbol{n} = v_{1,x} n_x + v_{1,y} n_y \tag{2.15}$$

$$v_{1,t} = \boldsymbol{v_1} \circ \boldsymbol{t} = v_{1,x} t_x + v_{1,y} t_y \tag{2.16}$$

$$v_{2,n} = \boldsymbol{v_2} \circ \boldsymbol{n} = v_{2,x} n_x + v_{2,y} n_y \tag{2.17}$$

$$v_{2,t} = \boldsymbol{v_2} \circ \boldsymbol{t} = v_{2,x} t_x + v_{2,y} t_y \tag{2.18}$$

Zgodnie z założeniami, składowe styczne przed i po zderzeniu pozostaną bez zmian:

$$v_{1,t}{}' = v_{1,t} \tag{2.19}$$

$$v_{2,t}{}' = v_{2,t} \tag{2.20}$$

Natomiast składowe normalne zmienią się zgodnie ze wzorami (2.10)-(2.11) dla przypadku jednowymiarowego w kierunku wyznaczonym przez wektor normalny:

$$v_{1,n}{}' = \frac{(m_1 - m_2) v_{1,n} + 2 m_2 v_{2,n}}{m_1 + m_2} \tag{2.21}$$

$$v_{2,n}{}' = \frac{-(m_1 - m_2) v_{2,n} + 2 m_1 v_{1,n}}{m_1 + m_2} \tag{2.22}$$

Aby uzyskać wartości wektorów prędkości po zderzeniu w pierwotnym układzie współrzędnych, prędkości należy przetransformować:

$$\boldsymbol{v_1}' = v_{1,n}{}' \boldsymbol{n} + v_{1,t}{}' \boldsymbol{t} \tag{2.23}$$

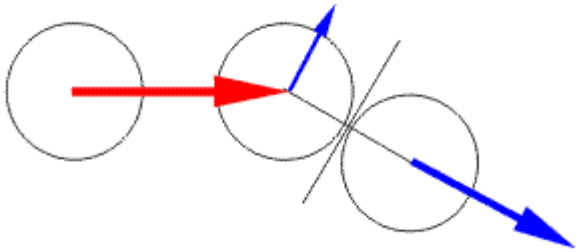
$$\boldsymbol{v_2}' = v_{2,n}{}' \boldsymbol{n} + v_{2,t}{}' \boldsymbol{t} \tag{2.24}$$

Analizując otrzymane wartości wektorów prędkości po zderzeniu wyraźnie widać, że spełniają one zasadę zachowania energii dla energii kinetycznej. Dodatkowo w przypadku

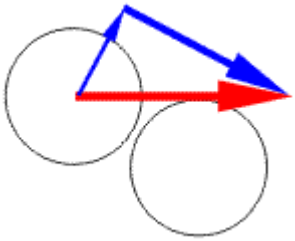
uproszczonym, gdy zderzają się obiekty o równych masach - pierwszy ruchomy, drugi stacjonarny - zasada zachowania energii sprowadza się do równania:

$$v_1{}^2 = v_1{}'^2 + v_2{}'^2 \tag{2.25}$$

Wizualnym potwierdzeniem poprawności powyższego wyprowadzenia jest fakt, że dla opisanego przypadku przedstawionego na rysunku Rys2.1, tworząc trójkąt prostokątny z prędkością początkową jako przeciwprostokątną i prędkościami po zderzeniu jako przyprostokątnymi okazuje się, że do tego samego równania doprowadza twierdzenie Pitagorasa. Trójkąt prostokątny został przedstawiony na rysunku Rys2.2.



Rys2.1 Rozkład prędkości kul o tych samych masach przed i po zderzeniu. Grafika zapożyczona z artykułu z Wikipedii [4].



Rys2.2 Suma kwadratów prędkości przed zderzeniem i po zderzeniu spełniona jest zarówno przez zasadę zachowania energii jak i twierdzenie Pitagorasa. Grafika zapożyczona z artykułu z Wikipedii [4].

2.2.2 Przypadek szczególny zderzenia typu kula-kula - zderzenie dwóch cząstek o tych samych masach i promieniach

W przypadku zderzeń jednakowych kul, wzory (2.10)-(2.11) uproszczą się ze względu tą samą wartość masy $m_1 = m_2 = m$:

$$v_1{}' = v_2 \tag{2.26}$$

$$v_2{}' = v_1 \tag{2.27}$$

a więc poszczególne składowe (2.19)-(2.22) w równaniach (2.23)-(2.24) będą wynosić odpowiednio:

$$v_{1,t}{}' = v_{1,t} \tag{2.28}$$

$$v_{2,t}{}' = v_{2,t} \tag{2.29}$$

$$v_{1,n}{}' = v_{2,n} \tag{2.30}$$

$$v_{2,n}{}' = v_{1,n} \tag{2.31}$$

Oznacza to, że dla szczególnego przypadku zderzenia ruchomej bili z bilą stacjonarną o takiej samej masie, w skutek zderzenia pierwsza kula zatrzyma się, a druga zacznie poruszać z prędkością pierwszej z sprzed zderzenia.

2.2.3 Zderzenie cząstki z obszarem granicznym pudła symulacyjnego

Z kolei w przypadku zderzenia kuli ze ścianą, korzystając ze wzorów (2.10)-(2.11) i biorąc pod uwagę, że dla cząstki o małej masie m_1 i prędkości przed zderzeniem $\boldsymbol{v_1 \neq 0}$ oraz ścianie o bardzo dużej masie m_2 pozostającej w spoczynku ($\boldsymbol{v_2 = 0}$):

$$\lim_{m_2 \to \infty} \frac{m_1 - m_2}{m_1 + m_2} = -1 \tag{2.32}$$

$$\lim_{m_2 \to \infty} \frac{m_1}{m_1 + m_2} = 0 \tag{2.33}$$

Oznacza to, że po zderzeniu kuli ze ścianą, ściana pozostanie nadal nieruchoma (co jest raczej intuicyjne), a cząstka będzie poruszać się z wektorem prędkości o składowych:

$$v_{1,n}{}' = -v_{1,n} \tag{2.34}$$

$$v_{1,t}{}' = v_{1,t} \tag{2.35}$$

Co dla przykładowego odbicia od ściany poziomej, leżącej na dolnej krawędzi obszaru będzie oznaczało jedynie zmianę składowej y-owej na przeciwną.

3 DOKUMENTACJA PROJEKTOWA

3.1 Raport wykonalności

3.1.1 Cel i zakres dokumentu

Raport wykonalności ma na celu przeprowadzenie analizy założeń i przebiegu projektu oraz usystematyzowanie wynikających z nich informacji. Dokument składa się z podrozdziałów zawierających wstępny opis systemu, analizę popytu i konkurencji, makrootoczenie, analizę możliwości i szans oraz ocenę kosztów i zysków projektu wraz z podsumowaniem. W powyższych punktach zostały opisane główne cele zespołu projektowego w kontekście uwarunkowań rynkowych, prawnych, czasowych oraz ogólnych możliwości.

3.1.2 Odbiorcy

Adresatami odbiorców jest autorka projektu dyplomowego oraz kadra naukowa Politechniki Gdańskiej reprezentowana przez promotora i zleceniodawcę, dr inż. Pawła Sytego, wraz z recenzentem pracy.

3.1.3 Wstępny opis systemu

Proponowany system jest zaprojektowanym od postaw modelem o działaniu edukacyjnym. Aplikacja ma na celu możliwość przeprowadzenia obliczeń oraz wizualizacji przebiegu zderzenia idealnie sprężystego w jednym z wielu proponowanych przypadków użycia na płaszczyźnie. System obejmuje zarówno zderzenia obiektów o różnych masach jak i prędkościach ograniczając je do dwóch typów: stacjonarna pozioma lub pionowa ściana o nieskończenie wielkiej masie oraz cząstka o zadanej masie, promieniu, położeniu początkowemu i prędkości. Pierwsze z nich mają za zadanie symulowanie obszarów granicznych pudła symulacyjnego zaś drugie użytkownik może definiować zgodnie z upodobaniem. Po zadaniu warunków początkowych i rozpoczęciu symulacji użytkownik może obserwować aktualne parametry wybranego obiektu zmieniające się w czasie, informacje o detekcji kolejnych zderzeń oraz stan całkowitej energii kinetycznej układu.

3.1.4 Analiza popytu i konkurencji

Odbiorcy

Docelową grupą odbiorców systemu są wszelkie osoby zainteresowane zagadnieniem zderzeń (w szczególności typu idealnie sprężystego). Mogą to być zarówno studenci uczelni, pracownicy naukowi jak osoby zewnętrzne chcące zgłębić swoją wiedzę lub przekonać się jaki będzie efekt zderzenia dla zadanej konfiguracji. W projekcie założono, że użytkownik końcowy nie musi znać szczegółów technicznych użytego środowiska programistycznego, a więc interfejs graficzny wymaga jedynie obsługi z poziomu poszczególnych znaków na klawiaturze. Dzięki zastosowaniu skalowanych do w przybliżeniu rzeczywistych rozmiarów cząstek, model może również zostać wykorzystany przez bilardzistów w ramach ćwiczeń rozgrywania niestandardowych ustawień. Opracowanie może być wstępem do rozważać na temat zderzeń cząstek materii np. gazów.

Popyt

Dzięki intuicyjnemu i czytelnemu interfejsowi graficznemu oraz zastosowanej dokładności i precyzji, system może znaleźć uznanie wśród każdego użytkownika chcącego badać niestandardowe ustawienia cząstek lub kul zarówno z punktu widzenia naukowego jak i w ramach rozrywki.

Konkurencja

Istnieje wiele symulatorów zderzeń jednak zwykle ograniczają się albo do wykorzystania jednej płaszczyzny (zderzenia centralne) albo nie dają możliwości parametryzowania zarówno masy, promienia jak i prędkości dowolnej ilości obiektów. Dodatkowym atutem symulatora jest rzeczywista skalowalność układu - wszystkie parametry podawane są w jednostkach SI, a promień kul na wyświetlaczy w przybliżeniu odpowiada jego rzeczywistej wartości, a więc wyniki układu są całkowicie odtwarzalne poza symulacją.

3.1.5 Makrootoczenie projektu

Otoczenie prawne

Zarówno silnik systemu jak i interfejs użytkownika wraz z wizualizacją graficzną zostały stworzone w środowisku Microsoft XNA Game Studio 4.0. Aplikacja może być udostępniana w ramach licencji produktu Microsoftu jednak niemożliwe jest wykorzystywanie jej w celach komercyjnych.

Otoczenie ekonomiczne i społeczne

Ze względu na brak licencji dla użytku komercyjnego, aplikacja nie może przynosić korzyści majątkowych. Natomiast udostępniane przez nią treści mogą być wykorzystane w szerokiej grupie społecznej, ograniczonej jedynie przez możliwość dostępu do komputera, zarówno w celach naukowych i prawnych.

3.1.6 Analiza możliwości i szans projektu (SWOT)

Szanse

Różnorodność możliwości wykorzystania przedstawionego systemu stwarza duże szanse powodzenia na rynku aplikacji.

Zagrożenia

Jedynym zagrożeniem dla systemu jest wykorzystane środowisko programistyczne, które może być wykorzystane bezproblemowo jedynie w systemach firmy Microsoft.

Atuty

Podstawowym atutem systemu jest jego elastyczność i funkcjonalność. Dodatkowo forma dostarczenia produktu końcowego w postaci instalatora umożliwia bezproblemowe zainstalowanie niezbędnych bibliotek i łatwe uruchomienie systemu.

Słabości

Ze względu na funkcjonalność w postaci skalowalności wymiarów obliczeń na rzeczywiste jednostki SI, parametry opisujące cząstki (masa, promień, położenie, prędkość) zostały ograniczone odgórnie i oddolnie tak aby wyświetlanie kul było możliwe do wyświetlania na ekranie o przeciętnej rozdzielczości. W związku z powyższym z poziomu interfejsu użytkownika niemożliwe jest wykonywanie obliczeń dla układów o rozmiarach w skali nanometrycznej. Należy jednak zwrócić uwagę na to, że jest to jedynie ograniczenie interfejsu użytkownika, a nie samej biblioteki z zaimplementowanym silnikiem fizyki.

3.1.7 Analiza wykonalności systemu

Wykonalność pod względem technicznym

Aplikacja została zrealizowana w środowisku Microsoft XNA Game Studio 4.0 z założeniem uruchamiania jej pod systemem operacyjnym Windows. Korzysta ona z oddzielnie zaprojektowanej i zaimplementowanej zewnętrznej biblioteki będącej silnikiem

fizycznym dla zagadnienia zderzenia idealnie sprężystego. Wykorzystanym językiem programistycznym jest C#.

Wykonalność pod względem organizacyjnym

Zleceniodawca wyznaczył termin odbioru systemu na październik 2014, a zleceniobiorca zobowiązał się do realizacji i dostarczenia projektu przed jego upływem. W związku z powyższym czas przeznaczony na stworzenie dokumentacji, poznanie niezbędnych technologii, zaprojektowanie, implementację oraz testowanie systemu to około pół roku. Z powodu ograniczeń czasowych istotne jest wykonanie harmonogramu pracy uwzględniającego dodatkowy czas na testowanie i ewentualne poprawki i jedynie jedną wersję publikacji.

Wykonalność pod względem prawnym

Zleceniobiorca zobowiązuje się do wykonania projektu zgodnie z obowiązującym prawem.

3.1.8 Analiza kosztów i zysków

Koszty

Jedynym, chociaż nie mniej istotnym, kosztem projektu jest koszt czasowy. W koszty zleceniobiorcy wchodzi więc całkowity czas poświęcony na realizację projektu zarówno na etapie projektowania, dokumentacji, implementacji jak i testowania systemu.

Zyski

System realizowany jest w ramach projektu magisterskiego dlatego też poprawne wykonanie go, wraz z pozytywną oceną z egzaminu dyplomowego, zapewni zleceniobiorcy tytuł naukowy magistra. Dodatkowym zyskiem jest bezcenna wiedza teoretyczna i techniczna zdobyta podczas realizacji systemu.

3.1.9 Podsumowanie

Z przeprowadzonego raportu wykonalności wynika, że projekt jest możliwy do zrealizowania na czas zgodnie z wymaganiami zleceniodawcy, bez ponoszenia kosztów finansowych, przestrzegając obowiązujących przepisów prawnych. Skończony projekt ma szansę być wykorzystywany zarówno w celach naukowych jak i rozrywkowych.

3.2 Spis wymagań systemowych

Wymagania systemowe projektu składają się z serii wymagań funkcjonalnych przedstawionych w tabeli Tab3.1. Ze względu na charakter pracy harmonogram uwzględnia jedynie jedną wersję publikacji aplikacji dlatego też kolumna ta została pominięta.

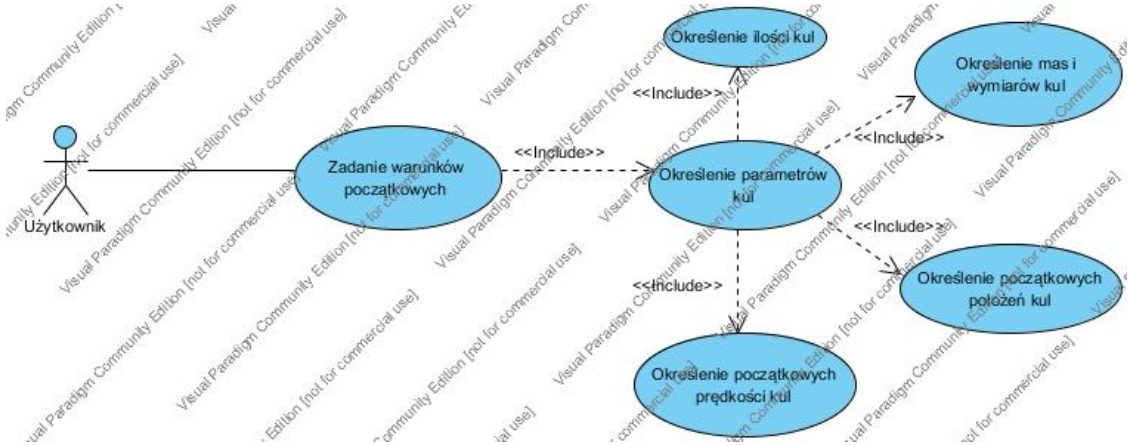
Tab3.1 Spis wymagań funkcjonalnych systemu wraz z opisem

ID	**Opis**	**Priorytet**	**Krytyczność**
WF01	System zapewnia gotowe układy symulacyjne	Średni	Średnia
WF02	System umożliwia stworzenie własnego układu symulacyjnego od zera	Wysoki	Wysoka
WF03	System umożliwia dodawanie do zadanego układu kolejnych cząstek	Wysoki	Wysoka
WF04	System umożliwia zmianę parametrów już istniejących cząstek	Wysoki	Wysoka
WF05	System waliduje parametry cząstek	Niski	Średnia
WF06	System umożliwia wstrzymanie i wznowienie przebiegu symulacji	Niski	Średnia
WF07	Obiekty symulacji są skalowane	Wysoki	Średnia
WF08	Wyniki obliczeń są realistyczne (wykonywane są na rzeczywistych jednostkach SI) i w przybliżeniu zgodne z oczekiwanymi wynikami teoretycznymi (po uwzględnieniu wszystkich założeń)	Wysoki	Wysoka
WF09	System przeprowadza symulację ruchu cząstek jako ruch jednostajnie prostoliniowy	Wysoki	Wysoka
WF10	System przeprowadza symulację zderzeń z rozróżnieniem na typ i ilość uczestniczących w nim obiektów	Wysoki	Wysoka

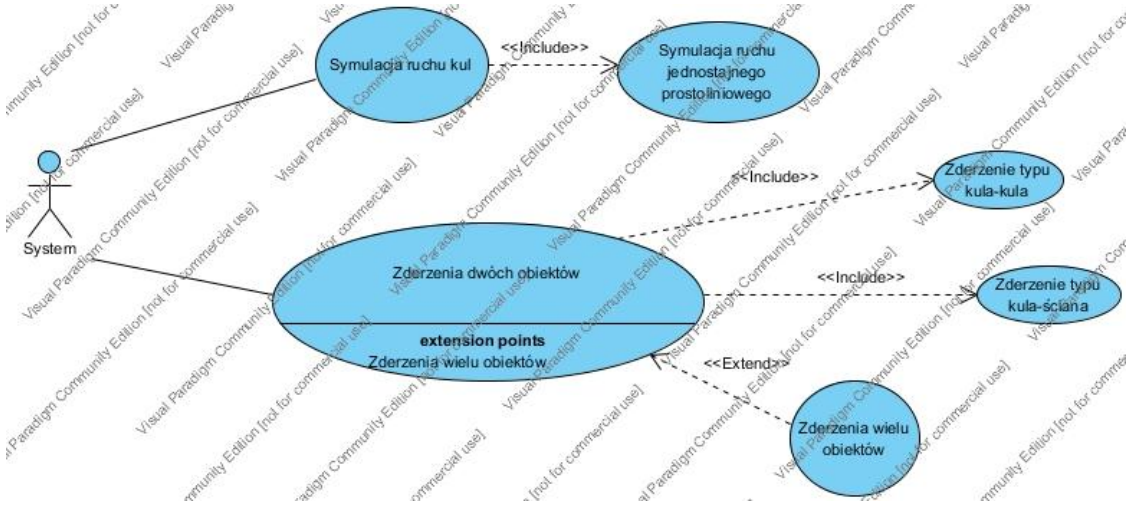
3.3 Diagramy UML

3.3.1 Diagramy przypadków użycia

Większość kluczowych wymagań systemowe zostało przedstawione za pomocą diagramów przypadków użycia. Wymagania WF01-WF04 zostały przedstawione na diagramie pierwszym (Rys3.1) traktując użytkownika jako głównego aktora i moment modyfikacji parametrów jako rozpoczęcie nowej symulacji. Z kolei wymagania WF09-WF10 zostały przedstawione na diagramie drugim (Rys3.2) z samym systemem jako aktorem.



Rys3.1 Diagram przypadków użycia przedstawiający interakcje aktora-użytkownika z systemem.



Rys3.2 Diagram przypadków użycia przedstawiający wewnętrzne wymagania funkcjonalne systemu.

3.3.2 Diagram klas

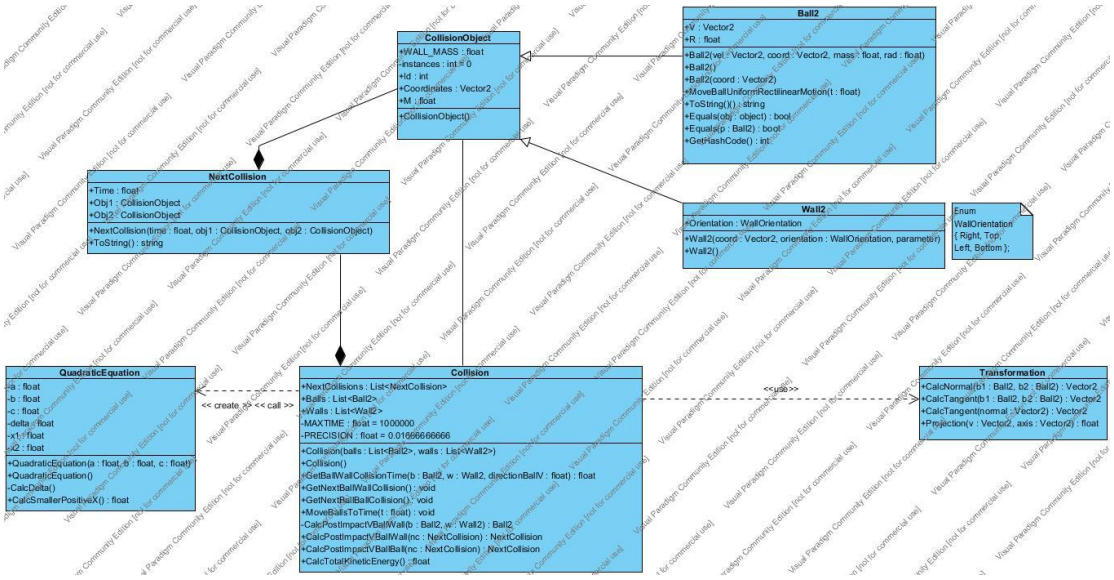
Projektując silnik fizyczny aplikacji należy zastanowić się, które funkcjonalności można pogrupować ze względu na wykonywaną przez nie czynność oraz atrybuty, na których pracują. W tym celu stworzono diagram klas przedstawiony na Rys3.3.

Najważniejszą klasą biblioteki jest klasa `Collision`. Zawiera ona informację o kolejnych zderzeniach jako obiektach `NextCollision` oraz stanie wszystkich cząstek (lista obiektów `Ball2`) i ścian (lista obiektów `Wall2`). To w ramach jej funkcji obliczane są parametry kolejnych zderzeń oraz prędkości po zderzeniach korzystając z pomocniczych klas

do rozwiązywania równania kwadratowego `QuadraticEquation` oraz transformacji na płaszczyźnie `Transformation`. Klasa `Collision` pozwala również przesuwać cząstki i monitorować stan całkowitej energii kinetycznej układu.

Informacje o kolejnym zderzeniu przechowywane są w obiekcie klasy `NextCollision`. Zawiera ona dane o czasie zderzenia i dwóch obiektach typu `CollisionObject` w nim uczestniczących.

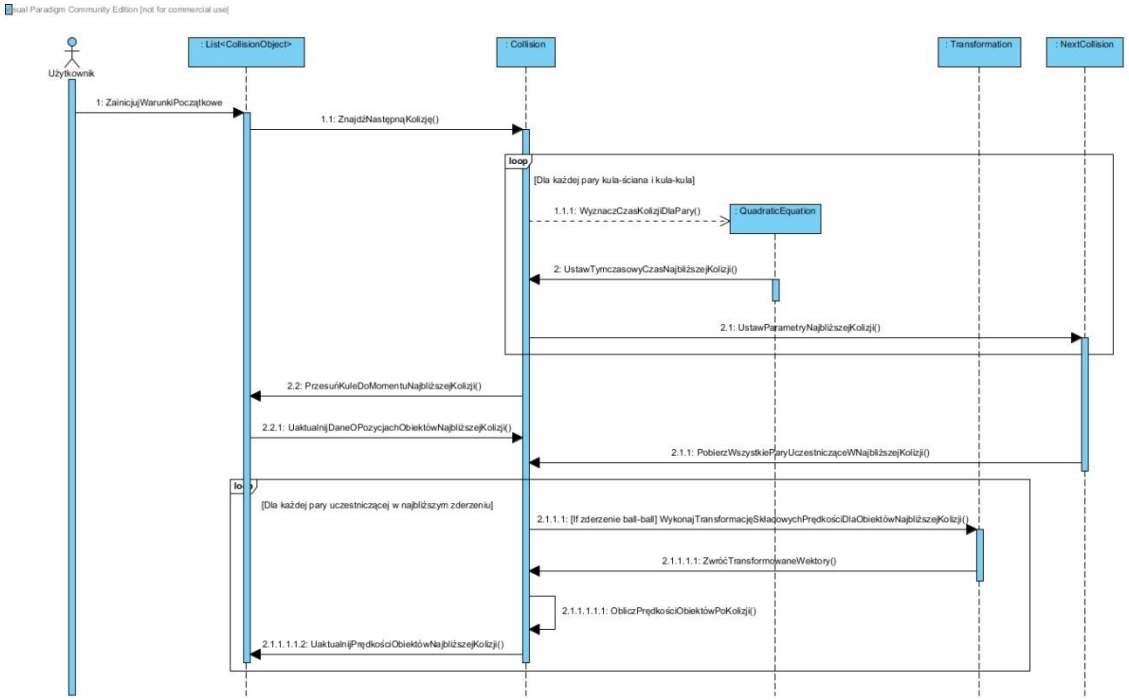
Klasa `CollisionObject` jest klasą nadrzędną dla typów `Ball2` i `Wall2`. Przechowuje część wspólną informacji o kuli i ścianie czyli unikatowy numer identyfikacyjny obiektu, masę i współrzędne. Dziedzicząca po niej klasa `Ball2` uzupełniona jest o wektor prędkości i wielkość promienia. Dodatkowo zawiera funkcję, dzięki której można zmienić pozycję cząstki zgodnie z ruchem jednostajnym prostoliniowym. Z kolei klasa `Wall2` zawiera informację o orientacji położenia ściany jako górna, dolna, prawa lub lewa.



Rys3.3 Diagram klas biblioteki z silnikiem fizyki zderzenia idealnie sprężystego.

3.3.3 Diagram sekwencji

Posiadając informację o klasach biblioteki z silnikiem fizyki, kolejnym bardzo istotnym aspektem jest przepływ danych wśród poszczególnych klas. Diagram sekwencji przedstawiony na rysunku Rys3.4, ilustruje sposób przepływu danych w ramach wykonywania wszystkich operacji mających na celu znalezienie kolejnego najbliższego zderzenia.



Rys3.4 Diagram sekwencji dla przepływu danych podczas znajdywania kolejnego najbliższego zderzenia.

4 IMPLEMENTACJA WYBRANYCH ZAGADNIEŃ

Kluczowymi zagadnieniami do zaimplementowania są funkcje odpowiedzialne za detekcję kolizji oraz zachowanie obiektów tuż po zderzeniu. W poniższych podrozdziałach przedstawiono najważniejsze fragmenty kodu silnika fizyki i aplikacji graficznej wykorzystujące teorię zawartą w rozdziale drugim wraz z opisem.

4.1 Detekcja kolizji

Jak wspomniano w części teoretycznej, detekcja kolizji sprowadza się do dwóch zagadnień. Pierwszym z nich jest detekcja kolizji kuli ze ścianą, która została zrealizowana za pomocą funkcji `GetNextBallWallCollision` klasy `Collision`:

```
1.    public void GetNextBallWallCollision()
2.    {
3.           this.NextCollisions = new List<NextCollision>();
4.           NextCollision nc = new NextCollision(MAXTIME, new CollisionObject(), new
5.                        CollisionObject());
6.           for (int i = 0; i < Balls.Count; i++)
7.           {
8.                  for (int j = 0; j < Walls.Count; j++)
9.                  {
10.                        if(Balls[i].V.X > 0 && Walls[j].Orientation ==
11.                                     WallOrientation.Right) // ball move in right
12.                                                            wall direction
13.                        {
14.                               float time = GetBallWallCollisionTime(Balls[i],
15.                                     Walls[j], Balls[i].V.X);
16.                               if (time >= 0.0f && (time + TIME_TO_NEXT_COLLISION) <
17.                                     nc.Time)
18.                               {
19.                                     nc = new NextCollision(time, Balls[i], Walls[j]);
20.                                     this.NextCollisions = new List<NextCollision>();
21.                                     this.NextCollisions.Add(nc);
22.                               }
23.                               else if (time >= 0.0f && (time - TIME_TO_NEXT_COLLISION)
24.                                     <= nc.Time)
25.                               {
26.                                     nc = new NextCollision(time, Balls[i], Walls[j]);
27.                                     this.NextCollisions.Add(nc);
28.                               }
29.                        }
30.                 ...
31.          }
32.   }
```

W ramach funkcji najpierw tworzona jest nowa lista obiektów kolejnych najbliższych zderzeń, która na wstępie zainicjowana jest sztucznym obiektem kolizji o nieskończenie odległym czasie w postaci wartości zmiennej `MAXTIME` (linie 3-5). Następnie w podwójnej pętli dla każdej cząstki sprawdzane jest z którą ze ścian nastąpi jej zderzenie. W liniach 10-29 przedstawiony został fragment kodu dla przypadku kolizji kuli z prawą pionową ścianą.

Najpierw wyliczany jest czas kolizji ze ścianą za pomocą funkcji `GetBallWallCollisionTime` (kod funkcji został omówiony poniżej). Następnie jeżeli otrzymany czas jest zarówno większy od zera (zderzenie nastąpi) jak i mniejszy od dotychczasowej najmniejszej wartości (linie 16-17), tworzona jest nowa lista kolizji i zderzenie jest do niej dodane. Należy zwrócić uwagę na dodatkowy parametr `TIME_TO_NEXT_COLLISION`, który jest stałą klasy `Collision` określającą przedział czasowy pomiędzy, którymi wyświetlane są kolejne klatki animacji, korzystające z biblioteki. W przypadku środowiska XNA domyślnie funkcja wykonująca aktualizację danych wywoływana jest sześćdziesiąt razy na sekundę, dlatego też parametr ten został ustawiony na tą wartość. Celem dodania tego parametru jest złapanie wszystkich możliwych najbliższych kolizji, które nastąpią w różnym czasie, ale nie większym niż o `TIME_TO_NEXT_COLLISION` w stosunku do najmniejszego z nich. Opisany problem przedstawiają linie 23-28. Pozostałe przypadki - kolizja z lewą, górną i dolną ścianą - przebiegają w analogiczny sposób.

Funkcja `GetBallWallCollisionTime` pobiera jako argumenty obiekt kuli i ściany oraz odpowiednią składową prędkości kuli, która określana jest przed wywołaniem funkcji w metodzie `GetNextBallWallCollision`. Kod źródłowy przedstawiony jest poniżej:

```
1.    public float GetBallWallCollisionTime(Ball2 b, Wall2 w, float directionBallV)
2.    {
3.           float s = 0.0f;
4.           if (w.Orientation == WallOrientation.Bottom || w.Orientation ==
5.                  WallOrientation.Top)
6.           {
7.                  s = Math.Abs(b.Coordinates.Y - w.Coordinates.Y);
8.           }
9.           else if (w.Orientation == WallOrientation.Left || w.Orientation ==
10.                 WallOrientation.Right)
11.          {
12.                 s = Math.Abs(b.Coordinates.X - w.Coordinates.X);
13.          }
14.
15.          return (s - b.R) / Math.Abs(directionBallV);
16.   }
```

Funkcja ta dzieli się na dwa etapy ze względu na orientację ściany. W pierwszym kroku - linie 4-8 - uwzględniony jest przypadek orientacji poziomej. Następnie - linie 9-13 - sprawdzany jest warunek dla orientacji pionowej. W zależności od przypadku wyliczana jest odległość środka kuli od ściany. Ostatecznie w linii 15 zwracany jest czas pokonania drogi wyliczonej odległości pomniejszonej o promień kuli z odpowiednią prędkością kuli.

Drugim zagadnieniem detekcji kolizji jest przypadek zderzenia kuli z drugą kulą, który został zrealizowany w funkcji `GetNextBallBallCollision` przedstawionej poniżej:

```
1.    public void GetNextBallBallCollision()
2.    {
3.           float curr_time;
4.           if (NextCollisions.Count > 0) curr_time = NextCollisions[0].Time;
5.           else curr_time = MAXTIME;
6.           NextCollision nc = new NextCollision(curr_time, new CollisionObject(), new
7.                  CollisionObject());
8.           for (int i = 0; i < Balls.Count; i++)
9.           {
10.                 for (int j = i + 1; j < Balls.Count; j++)
11.                 {
12.                        float xdiff = Balls[j].Coordinates.X - Balls[i].Coordinates.X;
13.                        float ydiff = Balls[j].Coordinates.Y - Balls[i].Coordinates.Y;
14.                        float vxdiff = Balls[j].V.X - Balls[i].V.X;
15.                        float vydiff = Balls[j].V.Y - Balls[i].V.Y;
16.                        float xdiff2 = xdiff * xdiff;
17.                        float ydiff2 = ydiff * ydiff;
18.                        float vxdiff2 = vxdiff * vxdiff;
19.                        float vydiff2 = vydiff * vydiff;
20.                        float r_sum = Balls[i].R + Balls[j].R;
21.                        float r_sum2 = r_sum * r_sum;
22.                        if (xdiff2 + ydiff2 >= r_sum2)
23.                        {
24.                               float a = vxdiff2 + vydiff2;
25.                               float b = 2.0f * xdiff * vxdiff + 2.0f * ydiff * vydiff;
26.                               float c = xdiff2 + ydiff2 - r_sum2;
27.                               QuadraticEquation qe = new QuadraticEquation(a, b, c);
28.                               float time = qe.CalcSmallerPositiveX();
29.                               if ((float)Math.Round((decimal)time, 5) > 0.0f && (time
30.                                      + TIME_TO_NEXT_COLLISION) < nc.Time)
31.                               {
32.                                      nc = new NextCollision(time, Balls[i], Balls[j]);
33.                                      this.NextCollisions = new List<NextCollision>();
34.                                      this.NextCollisions.Add(nc);
35.                               }
36.                               else if ((float)Math.Round((decimal)time, 5) > 0.0f &&
37.                                       (time - TIME_TO_NEXT_COLLISION) <= nc.Time)
38.                               {
39.                                      nc = new NextCollision(time, Balls[i], Balls[j]);
40.                                      this.NextCollisions.Add(nc);
41.                               }
42.                        }
43.                 }
44.          }
45.
46.   }
```

Ponieważ funkcja z założenia ma być wywoływana po zakończeniu funkcji wyliczającej czasy najbliższych zderzeń kul ze ścianami, w linii 4 ustawiana jest zmienna określające czas najbliższego zderzenia na najbliższe zderzenie wyliczone przez poprzednią funkcję. Następnie w linii 8 i 10 rozpoczyna się podwójna pętla sprawdzająca zderzenia dla każdej z kul pomiędzy każdą z pozostałych. Warto zwrócić uwagę na wartość początkową iteratora drugiej pętli, która celowo nie została ustawiona na zero, tak aby nie wykonywać podwójnych obliczeń dla już sprawdzonej pary. W liniach 12-21 wyliczane są wszystkie parametry niezbędne do rozwiązania równania kwadratowego z równania (2.5). Następnie w linii 22 sprawdzany jest dodatkowy warunek definiujący to, że zderzenie jeszcze nie nastąpiło, równocześnie eliminujący ewentualne nadmiarowe zderzenia wynikające z błędu

zaokrągleń operacji na typach zmiennoprzecinkowych pojedynczej precyzji. W liniach 24-28 obliczane są parametry równania kwadratowego oraz najmniejszy czas, równocześnie większy od zera. Ostatecznie w liniach 29-41 wykonywana jest analogiczna operacja jak w przypadku obliczania najbliższej kolizji kuli ze ścianą. Brany jest pod uwagę przedział czasowy uwzględniający parametr `TIME_TO_NEXT_COLLISION` oraz dodatkowo sprawdzany jest warunek wartości czasu większej od zera dla precyzji pięciu liczb - jest to dodatkowe zabezpieczenie przed nadmiarowymi zderzeniami w sytuacji, gdy kolejne najbliższe zderzenie liczone jest w momencie, gdy kule już stykają się.

4.2 Odpowiedź zderzenia

Odpowiedź zderzenia w przypadku pierwszego rodzaju kolizji kula-ściana jest raczej trywialna. Dużo ciekawsza jest implementacja przypadku zderzenia kula-kula, która została zrealizowana w funkcji `CalcPostImplactVBallBall`, która jako parametr przyjmuje obiekt najbliższego zderzenia `NextCollision`, a zwraca ten sam obiekt ze zmodyfikowanymi prędkościami:

```
1.    public NextCollision CalcPostImpactVBallBall(NextCollision nc)
2.    {
3.           if (nc != null && nc.Obj1 != null && nc.Obj2 != null)
4.           {
5.                  if (nc.Obj1.M < CollisionObject.WALL_MASS && nc.Obj2.M <
6.                         CollisionObject.WALL_MASS)
7.                  {
8.                         Ball2 b1 = (Ball2)nc.Obj1;
9.                         Ball2 b2 = (Ball2)nc.Obj2;
10.
11.                        Vector2 normal = Transformation.CalcNormal(b1, b2);
12.                        Vector2 tangent = Transformation.CalcTangent(normal);
13.
14.                        float v1n = Transformation.Projection(b1.V, normal);
15.                        float v1t = Transformation.Projection(b1.V, tangent);
16.                        float v2n = Transformation.Projection(b2.V, normal);
17.                        float v2t = Transformation.Projection(b2.V, tangent);
18.
19.                        float massSum = b1.M + b2.M;
20.                        float factor1 = b1.M / massSum;
21.                        float factor2 = b2.M / massSum;
22.
23.                        float v1nPost = factor1 * v1n - factor2 * v1n
24.                               + 2 * factor2 * v2n;
25.                        float v2nPost = -factor1 * v2n + factor2 * v2n
26.                               + 2 * factor1 * v1n;
27.
28.                        b1.V = new Vector2(v1nPost * normal.X + v1t * tangent.X,
29.                               v1nPost * normal.Y + v1t * tangent.Y);
30.                        b2.V = new Vector2(v2nPost * normal.X + v2t * tangent.X,
31.                               v2nPost * normal.Y + v2t * tangent.Y);
32.
33.                        return new NextCollision(nc.Time, b1, b2);
34.                 }
35.          }
```

```
36.         return nc;
37.   }
```

W pierwszej kolejności funkcja sprawdza przekazane parametry czy nie są wartościami `null`. Następnie w linii 5 sprawdzane jest czy oba obiekty zderzenia są cząstkami. Wykorzystano to stałą `WALL_MASS` klasy `CollisionObject`, która symbolizuje nieskończenie wielką masę zarezerwowaną tylko dla obiektów typu `Wall2`. W liniach 8-9 oba obiekty rzutowane są na typ `Ball2` aby móc wykonać na nich kolejne operacje wyliczania wektorów normalnego i stycznego (linie 11-12), rzutowania wektorów prędkości na lokalny układ współrzędnych wyznaczony przez wektor normalny i styczny (linie 14-17), obliczania prędkości po zderzeniu (linie 23-26) i rzutowania ich na pierwotny układ współrzędny (linie 28-31). Na końcu funkcja zwraca zmodyfikowany o prędkości po zderzeniu obiekt następnej kolizji.

4.3 Sekwencja głównego algorytmu

Tworzenie nowej aplikacji w środowisku Microsoft XNA Game Studio 4.0 w rzeczywistości opiera się na zaprojektowaniu zawartości dwóch funkcji - `Draw` i `Update`. Zgodnie z tłumaczeniem Roba Milesa, "kiedy framework XNA <<chce>> narysować ekran, używa opisywanej metody [`Draw`]" [5]. Z kolei "zadaniem metody `Update` jest aktualizacja danych świata gry" [5]. W trakcie działania aplikacji metody `Draw` i `Update` wywoływane są w stałych odstępach czasu. Za pomocą metody `Draw` za każdym razem rysowany jest ekran przedstawiający pudło symulacyjne, dolny panel informacyjny oraz kule w ich obecnych położeniach. Aby ruch kul był rzeczywisty istotne jest aby za pomocą metody `Update` odpowiednio uaktualniać ich położenie. Tak jak wspomniano we wcześniejszych podrozdziałach, w przypadku zaprojektowanego systemu metoda `Update` wywoływana jest raz na sześćdziesiąt sekund, dlatego też istotne jest aby w przemyślany sposób zawrzeć w niej odpowiednią kolejność obliczania kolejnych zderzeń i aktualizowania danych. W tym celu zaimplementowano w niej algorytm z rozdziału 2.1 wzbogacony o aktualizację położenia kul i innych parametrów. Kod źródłowy funkcji został przedstawiony poniżej.

```
1.    protected override void Update(GameTime gameTime)
2.    {
3.          UpdateKeyboardInput();
4.          if (isRunning)
5.          {
6.                // TODO: Add your update logic here
7.                float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;
8.                collisionTimer -= elapsed;
9.                collision.MoveBallsToTime(elapsed);
10.               if (isNextCollisionTimeKnown && collisionTimer <= 0)
```

```
11.                 {
12.                         collision.MoveBallsToTime(collisionTimer);
13.                         int i = 0;
14.                         foreach (NextCollision nc in collision.NextCollisions)
15.                         {
16.                                 if ((nc.Obj1.M < Wall2.WALL_MASS
17.                                         && nc.Obj2.M >= Wall2.WALL_MASS)
18.                                         || (nc.Obj1.M >= Wall2.WALL_MASS
19.                                         && nc.Obj2.M < Wall2.WALL_MASS))
20.                                 {
21.                                         collision.CalcPostImpactVBallWall(nc);
22.                                 }
23.                                 else
24.                                 {
25.                                         collision.CalcPostImpactVBallBall(nc);
26.                                 }
27.                                 if (i + 1 < collision.NextCollisions.Count)
28.                                 {
29.                                         collision.MoveBallsToTime(
30.                                                 collision.NextCollisions[i + 1].Time
31.                                                 - nc.Time);
32.                                 }
33.                                 counter++;
34.                                 i++;
35.                         }
36.                         isNextCollisionTimeKnown = false;
37.                         kin_energy = collision.CalcTotalKineticEnergy();
38.                         CalcAndSetNextCol();
39.                 }
40.         }
41.
42.         base.Update(gameTime);
43.   }
```

Wszelkie interakcje z użytkownikiem przeprowadzane są za pomocą poszczególnych przycisków klawiatury. Dlatego też należy sprawdzać stan klawiatury jak najczęściej aby nie pominąć przykładowego wstrzymania symulacji. W tym celu na początku funkcji `Update` wywoływana jest metoda `UpdateKeyboardInput`, która obsługuje wszelkie zmiany stanu klawiatury. Następnie w linii 4 sprawdzana jest wartość flagi `isRunning`, która jest zmienną typu `bool` oznaczającą czy symulacja została wstrzymana. Jeżeli flaga ustawiona jest na wartość negatywną, oznacza to, że należy wykonać kolejny krok symulacji. W linii 7 pobierany jest czas ostatniego wywołania metody `Update`, `elapsedTime` (który dla ustawień aplikacji jest zawsze równy 1/60 sekundy). Następnie zmienna `collisionTimer` odpowiedzialna za odliczanie czasu do najbliższej kolizji pomniejsza jest o pobraną wartość. Zmienna `collisionTimer` ma ustawianą wartość przy inicjalizacji aplikacji oraz po obliczeniach dla kolejnych zderzeń. W linii 9 wszystkie kule przesuwane są o czas `elapsedTime`. Następnie w linii 10 sprawdzany jest warunek czy znany jest czas następnego zderzenia (`isNextCollisionTimeKnown = true`) oraz czy powinien on zostać wykonany w bieżącym wywołaniu funkcji `Update` (`collisionTimer <= 0`). Jeżeli warunek nie jest spełniony, funkcja zakańcza swoje działanie. Jeżeli jednak powinny zostać przeprowadzone

obliczenia związane z kolejnym zderzeniem, funkcja wchodzi w blok rozpoczynający się linią 11. W linii 12 wszystkie kule przesunięte są o czas `collisionTimer`. Jest to zabieg mający na celu ustawienie położeń kul na wartości występujące dokładnie w momencie kolejnego zderzenia. Następnie w bloku 14-35 dla każdej pary, wchodzącej w zakres najbliższych zderzeń, obliczane są wartości prędkości po zderzeniu wraz z każdorazowym zwiększeniem licznika kolizji `counter`. Na koniec w liniach 36-38 ustawiana jest flaga `isNextCollisionTimeknown` na fałsz, przeliczana jest na nowo całkowita energia kinetyczna układu i obliczane są następne najbliższe zderzenia za pomocą metody `CalcAndSetNextCol`, która wywoływana jest również przy inicjalizacji programu. Kod funkcji `CalcAndSetNextCol` przedstawiony jest poniżej:

```
1.    private void CalcAndSetNextCol()
2.    {
3.           collision.GetNextBallWallCollision();
4.           collision.GetNextBallBallCollision();
5.           if (collision.NextCollisions.Count > 0)
6.           {
7.                  List<NextCollision> sortedNextCollisions =
8.                         collision.NextCollisions.OrderBy(x => x.Time).ToList();
9.                  collision.NextCollisions = sortedNextCollisions;
10.                 collisionTimer = collision.NextCollisions[0].Time;
11.                 isNextCollisionTimeKnown = true;
12.                 // setting next collision string
13.                 nextCollision = "";
14.                 foreach (NextCollision nc in collision.NextCollisions)
15.                 {
16.                        nextCollision += " / " + nc.ToString();
17.                 }
18.          }
19.          else
20.          {
21.                 isNextCollisionTimeKnown = false;
22.                 nextCollision = " / unknown";
23.          }
24.   }
```

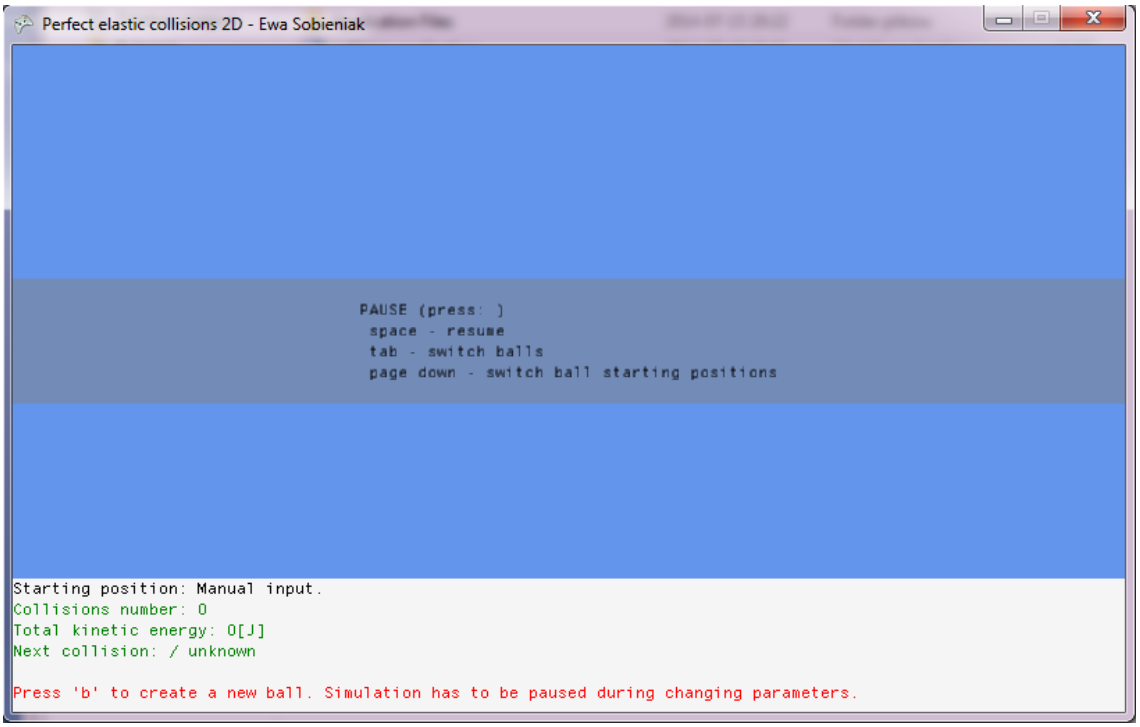
Na początku w liniach 3-4 wywoływane są metody detekcji najbliższych zderzeń dla kolizji typu kula-ściana i kula-kula (które zostały omówione wcześniej w tym rozdziale). Następnie, jeżeli znaleziono jakiekolwiek zderzenie, w liniach 7-9 zderzenia są sortowane po czasie kolizji od najbliższego do najdalszego. W kolejnym kroku zmienna `collisionTimer` zostaje ustawiona na czas najbliższej kolizji (linia 10), a flaga `isNextCollisionTimeKnown` na prawdę. W liniach 13-17 ustawiany jest fragment panelu informacyjnego dotyczący najbliższych zderzeń. Jeżeli wywołanie metod detekcji nie znalazło żadnego najbliższego zderzenia - przykładowo dla pustego układu - w bloku linii 19-22 flaga `isNextCollisionTimeKnow` ustawiana jest ponownie na fałsz, a informacja o następnym zderzeniu na nieznaną.

Należy mieć na uwadze, że jest to jedynie bardzo pobieżny opis sposobu działania środowiska XNA, niezbędny do zrozumienia działania danego systemu. Więcej szczegółów dotyczących Microsoft XNA Game Studio 4.0 można znaleźć we wspomnianej książce Roba Milesa [5].

5 INSTRUKCJA OBSŁUGI SYSTEMU

5.1 Instalacja i uruchamianie aplikacji

System został dostarczony w formie instalatora *setup.exe*, który należy włączyć przed pierwszym uruchomieniem aplikacji. W trakcie instalacji sprawdzane są podstawowe wymogi środowiska, a więc obecność platformy .NET 4.0 oraz bibliotek XNA. Jeżeli nie zostaną znalezione, instalator umożliwia pobranie ich i instalację. Po zainstalowaniu aplikacji, system należy uruchomić po przez plik *collision.application*. Ekran tuż po uruchomieniu przedstawia rysunek Rys5.1.



Rys5.1 Ekran systemu tuż po uruchomieniu aplikacji.

5.2 Obsługa wybranego układu

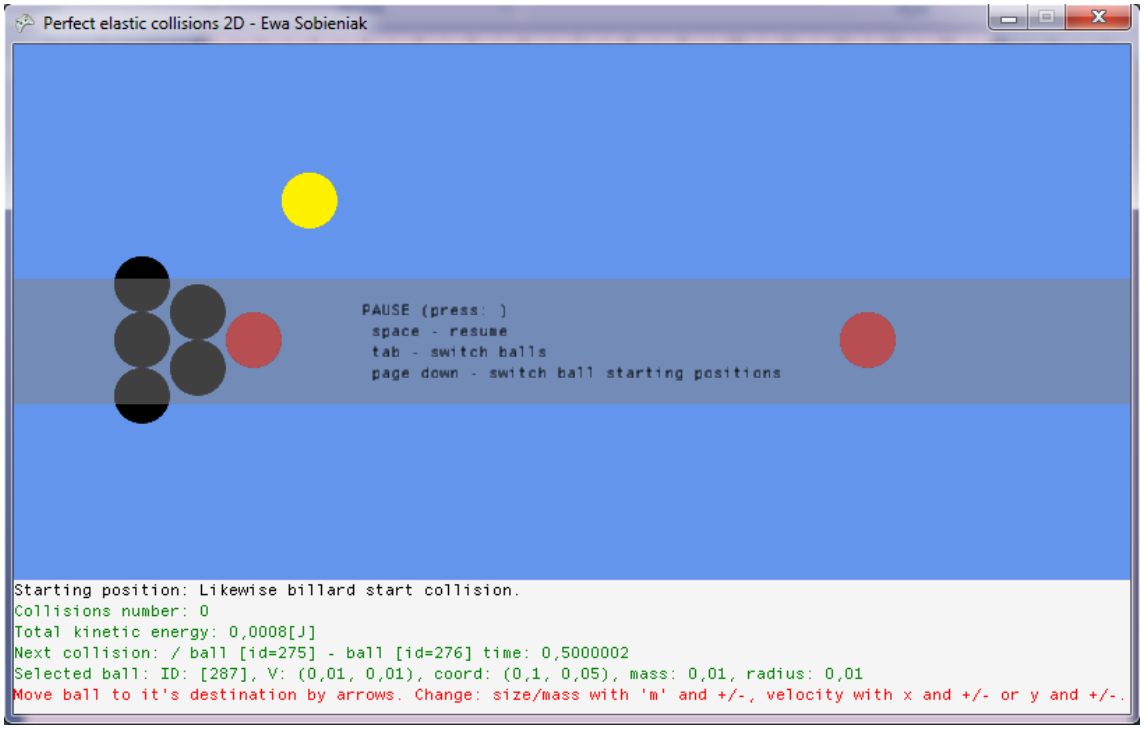
Aplikacja składa się z sześciu różnych układów symulacyjnych:

- układ pusty
- układ zderzeń centralnych horyzontalnych
- układ zderzenia centralnego wertykalnego
- układ zderzenia niecentralnego (skośnego)
- układ zderzenia trzech obiektów

- układ typu rozbicie kul bilardowych

Użytkownik może przełączać poszczególne układy za pomocą przycisku page down. W każdym z układów wszystkie kule mają domyślnie kolor czarny. Na czerwono zaznaczone są tylko te kule, które będą uczestniczyć w kolejnym zderzeniu. Z kolei na żółto zaznaczona jest kula, której parametry wyświetlone są w panelu informacyjnym na dole ekranu. Za pomocą przycisku tabulacji można zmieniać wybraną kulę tak aby móc zobaczyć parametry wszystkich kul.

Symulację rozpoczyna się i zatrzymuje za pomocą spacji. Każdy z układów można modyfikować dodając nową kulę za pomocą przyciśnięcia klawisza b. Dodatkowo wybraną, zaznaczoną na żółto kulę, można przenosić za pomocą strzałek, zwiększać i zmniejszać masę wybierając m i odpowiednio + lub - na klawiaturze numerycznej. Zmiana masy powoduje wprowadzenie analogicznej zmiany dla promienia kuli tak aby było widać graficznie cięższe cząstki. Ostatnią zmianą jest modyfikacja wektora prędkości po przez wciśnięcie kombinacji x i +/- oraz y i +/- dla odpowiednich składowych. Przykładową modyfikację oraz instrukcje przedstawia rysunek Rys 5.2.



Rys5.2 Zrzut ekranu z etapu modyfikacji układu dla układu typu rozbicie kul bilardowych.

Aplikacja na bieżąco wyświetla informację o ilości odbytych zderzeń, parametrach kolejnego zderzenia oraz stanie całkowitej energii kinetycznej. Należy mieć na uwadze, że ta ostatnia może zmieniać się o wartości rzędu 10^{-7} ze względu na ograniczenie precyzji obliczeń do typu zmiennoprzecinkowego pojedynczej precyzji.

6 TESTOWANIE SYSTEMU

6.1 Testy jednostkowe silnika fizyki

W trakcie projektowania i implementowania biblioteki z silnikiem fizyki, dla głównych klas stworzono ich odpowiedniki w ramach testowania jednostkowego. Do kodu źródłowego załączony jest projekt `CollisionLibraryTestProject` składający się z klas: `Ball2Test`, `CollisionTest`, `QuadraticEquationTest` oraz `TransformationTest`.

Testy jednostkowe opierają się na sprawdzaniu działania najważniejszych funkcji poprzez porównywanie wyniku otrzymanego z wynikiem oczekiwanym. W przypadku otrzymania niezgodności w wynikach, zgłaszane są tak zwane asercje powodujące negatywny wynik egzekucji testu. Jeżeli wyniki zgadzają się, egzekucja testu zwraca rezultat pozytywny i jest on uznany za zdany. Przykładowo w klasie testującej klasę `Collision`, test jednostkowy sprawdzający czy prawidłowo zostanie obliczona prędkość kuli po zderzeniu z lewą ścianą, wygląda następująco:

```
1.    [TestMethod()]
2.    public void CalcPostImpactVBallWallTestLeftWallFirstObjBall()
3.    {
4.           Ball2 ball = new Ball2(new Vector2(-1.5f, 6.0f), new
5.                  Vector2(2.01f, -3.0f), 0.01f, 0.01f);
6.           Wall2 wall = new Wall2(new Vector2(2.0f, -3.0f),
7.                  WallOrientation.Left);
8.           Collision target = new Collision();
9.           NextCollision nc = new NextCollision(0.0f, ball, wall);
10.          Ball2 expectedBall = new Ball2(new Vector2(1.5f, 6.0f), new
11.                 Vector2(2.01f, -3.0f), 0.01f, 0.01f);
12.          NextCollision actual;
13.          actual = target.CalcPostImpactVBallWall(nc);
14.          Ball2 actualBall = (Ball2)actual.Obj1;
15.          Assert.AreEqual(expectedBall, actualBall);
16.   }
```

W powyższym fragmencie kodu najpierw tworzone są obiekty uczestniczące w zderzeniu: kula o wektorze prędkości $\boldsymbol{v} = [-1.5, 6.0]$, położeniu $[2.01, -3.0]$, masie równej 0.01kg i promieniu 0.0m oraz ściana o orientacji pionowej, położona po lewej stronie w punkcie o tych samych współrzędnych przesuniętym jedynie o wielkość promienia kula na lewo. W kolejnych krokach stworzony jest obiekt klasy `Collision` i nadane parametry kolejnego zderzenia, w którym uczestniczą stworzone kula i ściana. Następnie został stworzony jeszcze jeden obiekt klasy `Ball2` będący oczekiwaniem rezultatem zderzenia, a więc różniący się od pierwszej kuli jedynie wektorem prędkości. Zgodnie z częścią

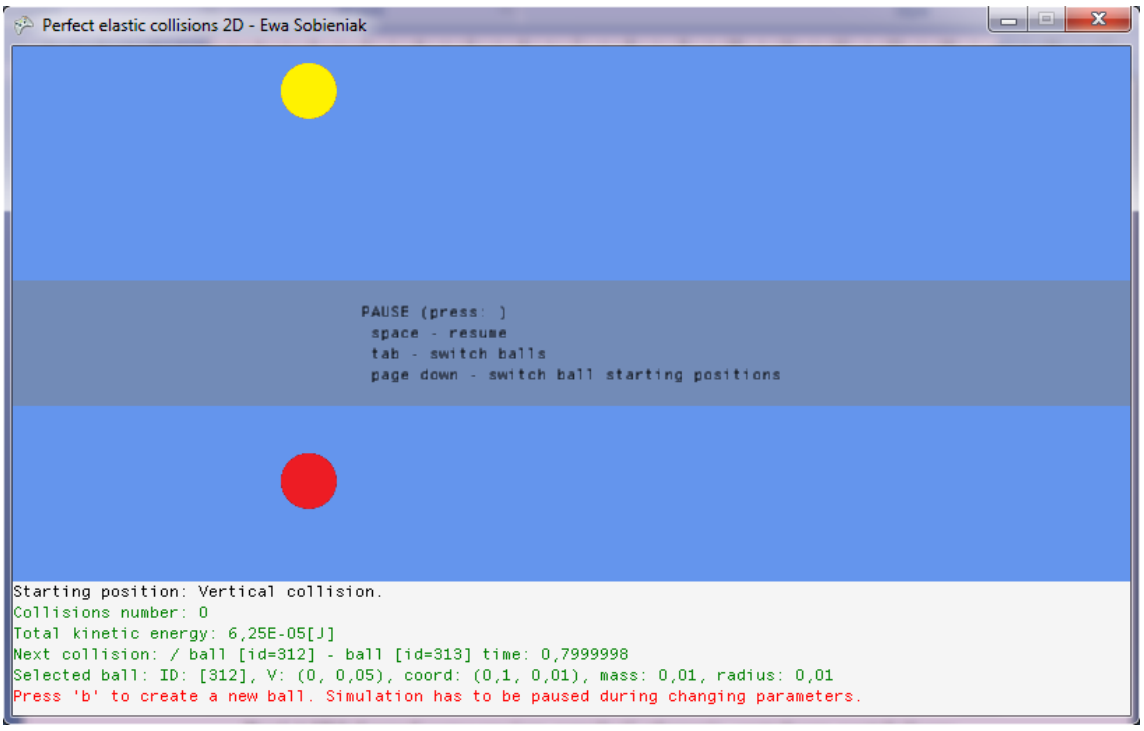
teoretyczną mówiącą o zderzeniach cząstek z obiektami o bardzo dużej masie, w przypadku kuli poruszającej się na lewo i zderzenia jej ze ścianą, zmianie ulegnie jedynie jej x-owa składowa prędkości zmieniając się na wartość przeciwną. Przedostatnim krokiem jest wywołanie metody obliczającej prędkość kuli po zderzeniu ze ścianą. Ostatecznie wywołany jest test asercji na rzecz otrzymanego w skutek obliczeń i teoretycznego obiektu, który w tym przypadku powinien zakończyć się powodzeniem.

6.2 Testy końcowe

Końcowym etapem przed publikacją aplikacji jest wykonanie serii testów z poziomu graficznego interfejsu użytkownika. W tym celu zaprojektowano serię układów różniących się zawartymi w nich kulami zarówno pod kątem parametrów jak i ilości obiektów. Poniżej przedstawiono zestawienie wyników oczekiwanych z rezultatami otrzymanymi za pomocą systemu dla przykładowych zderzeń typu kula-kula.

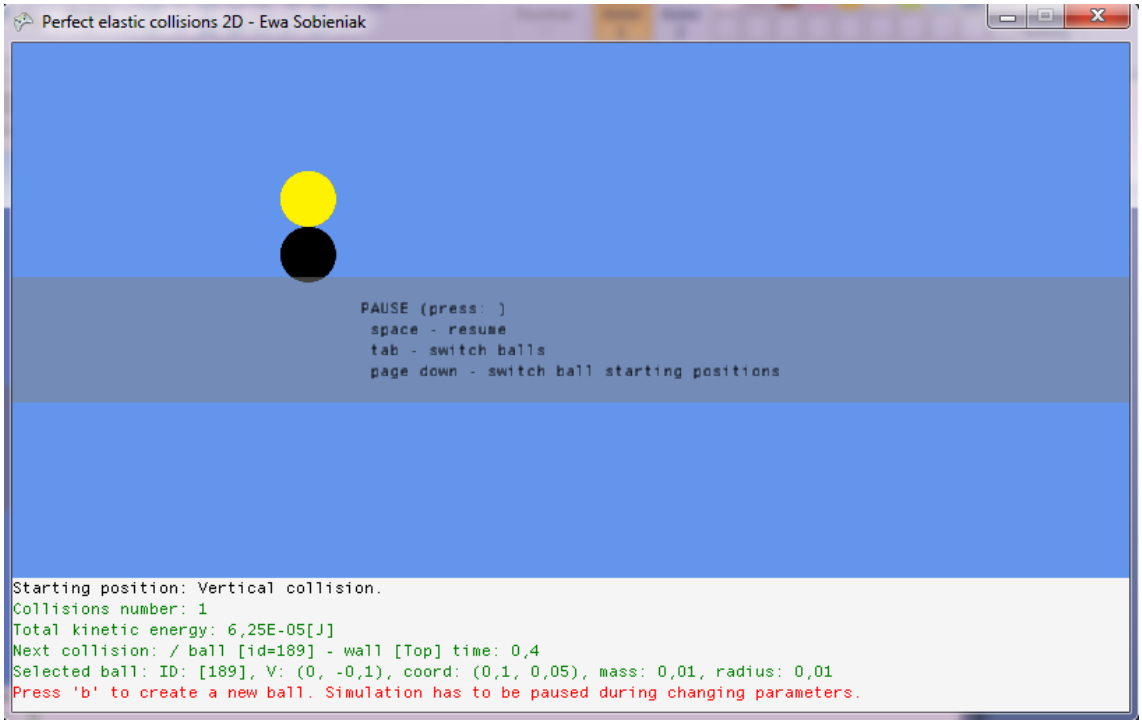
6.2.1 Zderzenie centralne wertykalne identycznych kul

Obiektami zderzenia są dwie kule o wektorach prędkości skierowanych wzdłuż tej samej pionowej osi, ale o przeciwnym zwrocie. Przed rozpoczęciem symulacji pierwsza kula ma położenie $[0.1, 0.01]$ i prędkość $\boldsymbol{v} = [0.0, 0.5]$, zaś druga kula ma położenie $[0.1, 0.15]$ i prędkość $\boldsymbol{v} = [0.0, -0.1]$. Obie kule mają tę samą masę 0.01kg i promienie 0.01m. Opisany układ kul ilustruje rysunek Rys6.1 będący zrzutem ekranu z wykonanej aplikacji.



Rys6.1 Układ przed zderzeniem centralnym wertykalnym identycznych kul.

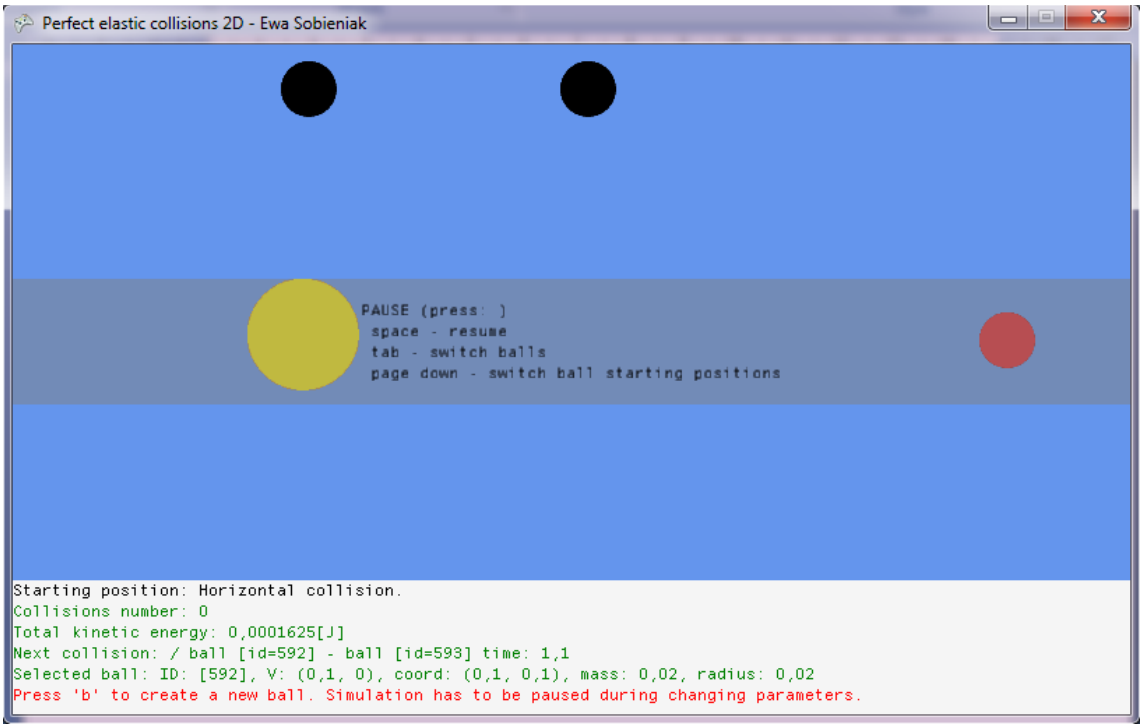
Ponieważ jest to zderzenie centralne horyzontalne, można rozpatrzyć je jako zderzenie w jednej płaszczyźnie. Podstawiając odpowiednie parametry do wzorów (2.26) i (2.27) kule po zderzeniu przekażą sobie swoje prędkości, a więc pierwsza kula poruszać się będzie z prędkością $\boldsymbol{v} = [0.0, -0.1]$, a druga $\boldsymbol{v} = [0.0, 0.5]$. Po uruchomieniu symulacji, aplikacja potwierdza spodziewane wyniki - kule "zamieniły się" prędkościami. Rysunek Rys6.2 przedstawia zrzut ekranu aplikacji tuż po zderzeniu, na którym widać, że pierwsza kula ma teraz prędkość jaką miała kula druga tuż przed zderzeniem.



Rys6.2 Układ tuż po zderzeniu centralnym wertykalnym identycznych kul.

6.2.2 Zderzenie centralne horyzontalne kul o różnych masach

W drugim teście obiektami zderzenia są dwa zestawy dwóch kul o wektorach prędkości skierowanych wzdłuż tych samych poziomych osi, ale o przeciwnym zwrocie. Test skupia się na drugiej parze, w której przed rozpoczęciem symulacji pierwsza kula ma położenie $[0.1, 0.1]$ i prędkość $\boldsymbol{v} = [0.1, 0.0]$, zaś druga kula ma położenie $[0.35, 0.1]$ i prędkość $\boldsymbol{v} = [-0.1, 0.0]$. Kule różnią się swoimi promieniami i masami - pierwsza z nich ma masę równą 0.02kg i promień 0.02m, zaś druga jest o połowę mniejsza i lżejsza. Opisany układ kul ilustruje rysunek Rys6.3 będący zrzutem ekranu z wykonanej aplikacji.



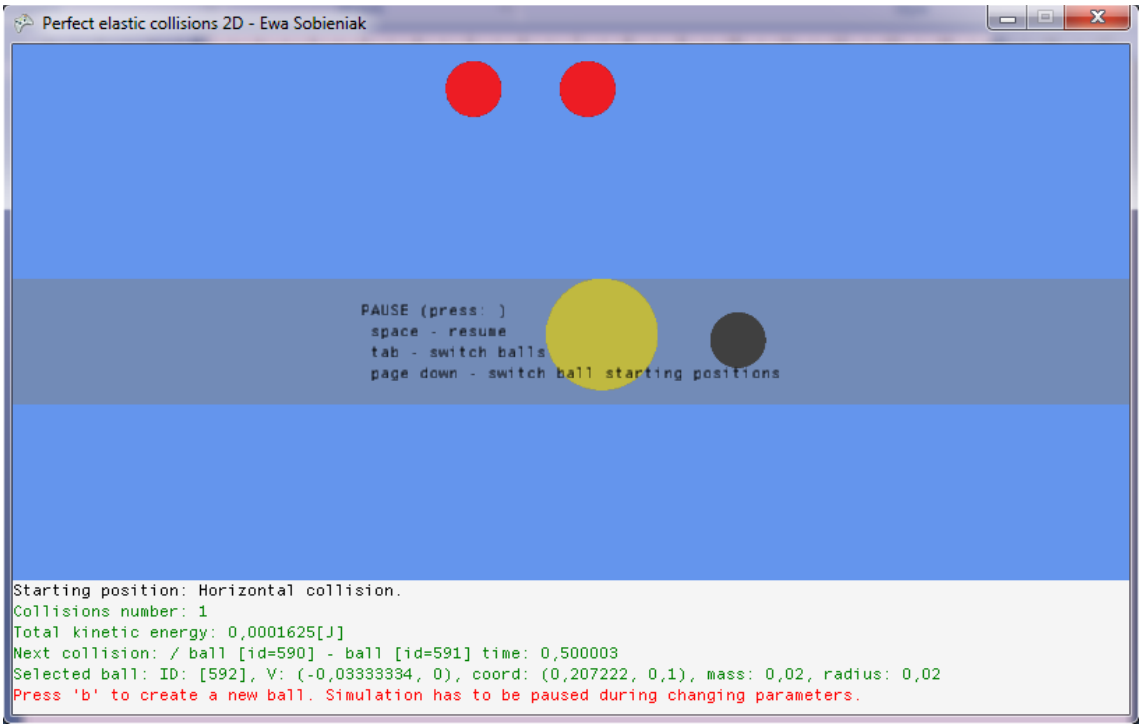
Rys5.3 Układ przed rozpoczęciem symulacji zderzenia centralnego horyzontalnego kul o różnych masach i promieniach.

Ponieważ jest to zderzenie centralne wertykalne, można rozpatrzyć je jako zderzenie w jednej płaszczyźnie. Tym razem kule różnią się masami, a więc należy skorzystać ze wzorów (2.10) i (2.11). Podstawiając odpowiednie parametry do wzorów kule po zderzeniu będą miały prędkości:

$$v_1' = \frac{(m_1 - m_2)v_1 + 2m_2v_2}{m_1 + m_2} = \frac{(0.02 - 0.01) \cdot 0.1 + 2 \cdot 0.01 \cdot (-0.1)}{0.02 + 0.01} = -0.0(3)$$

$$v_2' = \frac{-(m_1 - m_2)v_2 + 2m_1v_1}{m_1 + m_2} = \frac{-(0.02 - 0.01) \cdot (-0.1) + 2 \cdot 0.02 \cdot 0.1}{0.02 + 0.01} = 0.1(6)$$

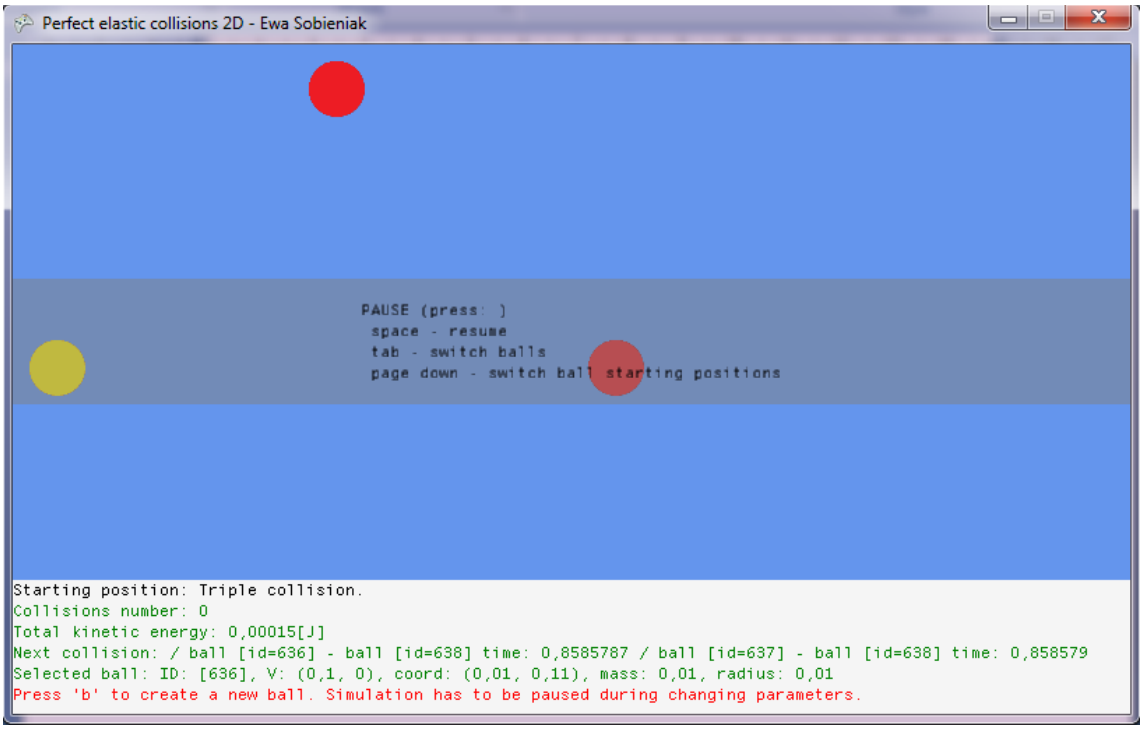
Rysunek Rys6.4 przedstawia zrzut ekranu aplikacji tuż po zderzeniu wraz z parametrami pierwszej z kul. Symulacja ponownie potwierdziła poprawność obliczeń. Warto zwrócić uwagę, że całkowita energia kinetyczna układu przed i po zderzeniu pozostała bez zmian.



Rys6.4 Układ tuż po zderzeniu centralnym horyzontalnym kul o różnych masach i promieniach.

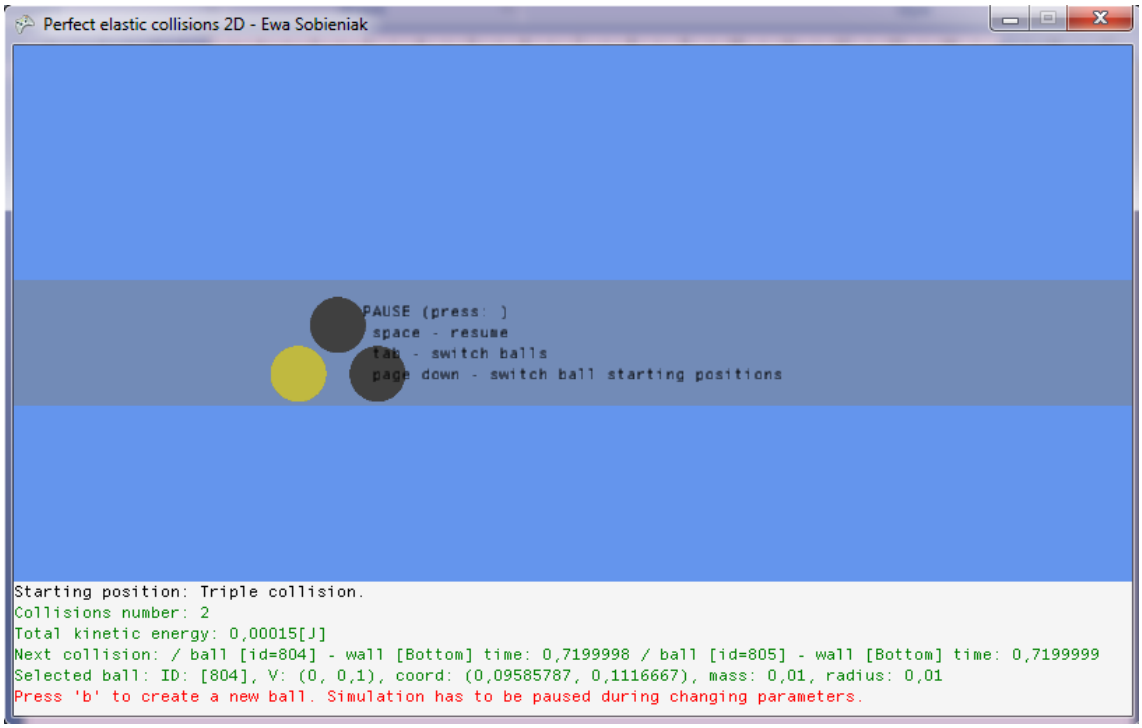
6.2.3 Zderzenie trzech obiektów

W kolejnym teście obiektami zderzenia są trzy kule. Dwie z nich ułożone są jak w jednym kierunku i dążą do zderzenia centralnego horyzontalnego jak w poprzednim przykładzie. Ich wektory prędkości to kolejno $\boldsymbol{v} = [0.1, 0.0]$ i $\boldsymbol{v} = [-0.1, 0.0]$. Trzecia kula ułożona jest powyżej linii wyznaczonej przez środki dwóch pierwszych kul, a jej wektor prędkości $\boldsymbol{v} = [0.0, 0.1]$ nakazuje jej ruch w kierunku pionowym w dół, tak aby zderzyła się z każdą z pozostałych dwóch kul zanim nastąpi zderzenie pomiędzy nimi. Wszystkie trzy kule mają taką samą masę $m = 0.01\text{kg}$ i promień $r = 0.01\text{m}$. Opisany układ ilustruje rysunek Rys6.5.



Rys6.5 Układ przed rozpoczęciem symulacji zderzenia trzech identycznych kul.

Ze względu na położenie i rozmiar kul w zderzeniu będą uczestniczyć trzy kule. Zgodnie z założeniami systemu, zderzenie można rozłożyć na dwa osobne zderzenia: zderzenie pierwszej kuli poziomej z kulą pionową i zderzenie drugiej kuli poziomej z pionową. Tym razem żadne ze zderzeń nie jest zderzeniem centralnym - wektory prędkości tuż przed zderzeniem wyznaczają kąt prosty. W tym przypadku należy skorzystać z procedury opisanej w punkcie 2.2.1 dla przypadku ogólnego zderzeń kul. Po wyznaczeniu wektorów normalnych i stycznych, zrzutowaniu wektorów prędkości na lokalny układ współrzędny, wyliczeniu prędkości zgodnie ze wzorami (2.10) i (2.11) i ponownym zrzutowaniu ich na pierwotny układ współrzędny, kule będą miały prędkości $\boldsymbol{v} = [0.0, 0.1]$ i $\boldsymbol{v} = [0.0, -0.1]$. Oczywiście wektor prędkości dla pionowej kuli zmieni się jeszcze w skutek zderzenia z drugą poziomą kulą. Efekt zderzenia przedstawia rysunek Rys6.6 potwierdzający rozważania teoretyczne.



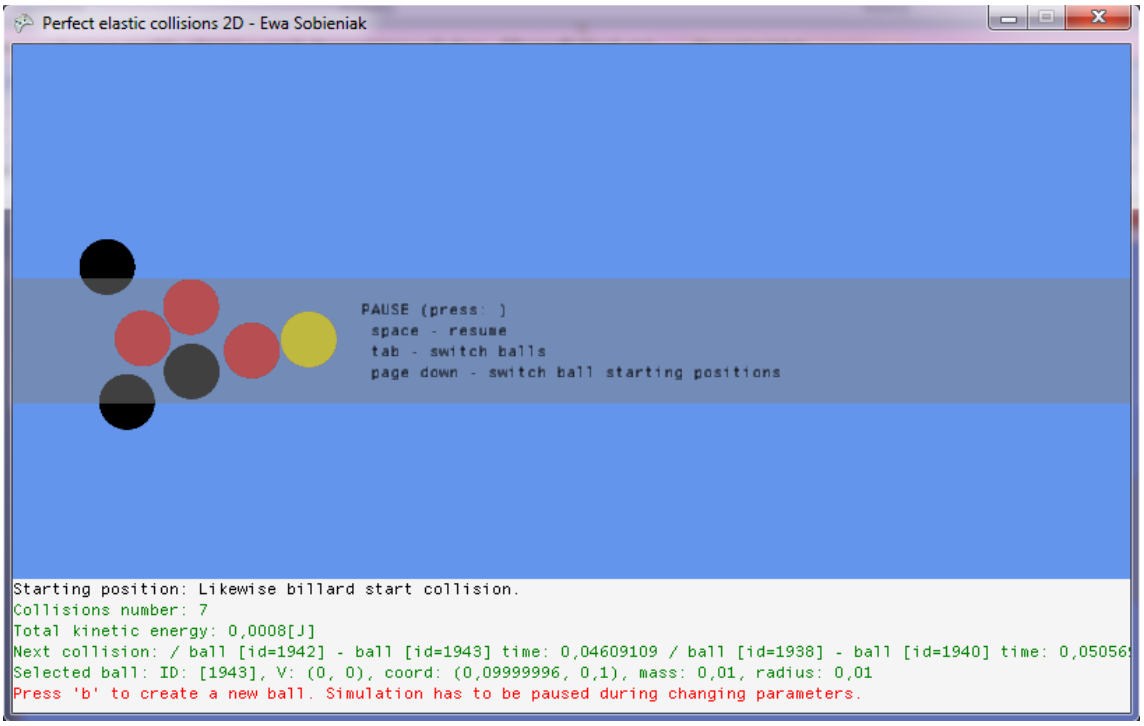
Rys6.6 Układ tuż po zderzeniu trzech identycznych kul.

6.2.4 Przykładowe rozbicie w grze typu bilard

Ostatnim przypadkiem testowym jest układ identycznych stacjonarnych cząstek przypominający początkowe ustawienie w grze typu bilard. Ostatnia z kul ustawiona jest po przeciwnej stronie ekrany z zadaną prędkością początkową. Opisany układ przedstawia Rys6.7.

Rys6.7 Układ przed rozpoczęciem symulacji przykładowego rozbicia w grze typu bilard.

Ze względu na ilość uczestniczących w zderzeniu obiektów, a więc rozpatrzenie go jako sekwencję zderzeń par obiektów, spodziewany efekt przeprowadzonego testu jest intuicyjnym ustawieniem, którego spodziewa się każdy, kto chociaż raz grał w bilard. Układ po zderzeniu wygenerowany w skutek symulacji przedstawia rysunek Rys6.8. Realistyczne ułożenie obiektów tuż po rozbiciu sugeruje poprawne działanie systemu.



Rys6.8 Układ tuż po rozpoczęciu symulacji przykładowego rozbicia w grze typu bilard.

7 PODSUMOWANIE I WNIOSKI

Praca dyplomowa miała na celu zgłębienie teorii zagadnienia zderzenia idealnie sprężystego i opracowanie opartego na niej systemu umożliwiającego przeprowadzanie symulacji dla zadanych parametrów. W pracy zawarto poszczególne etapy od opisu teoretycznego praw fizyki rządzących układem, algorytmów detekcji i odpowiedzi zderzenia po projekt biblioteki silnika fizyki wraz z dokumentacją projektową, opisem stworzonego systemu i sposobem wykorzystania zastosowanych rozwiązań wraz z częścią opisującą przebieg testowania aplikacji.

W ramach testowania systemu sprawdzono go zarówno za pomocą testów jednostkowych na poziomie implementacji biblioteki jak i testów końcowych stworzonej aplikacji. Model przetestowano pod kątem poprawności wyników dla przypadku zderzenia centralnego wertykalnego dwóch identycznych kul jak i zderzenia centralnego horyzontalnego kul o różnych masach. Sprawdzono również zachowanie systemu dla zderzeń niecentralnych oraz działanie algorytmu obsługujące przypadek zderzenia więcej niż dwóch obiektów na raz. Wykazano, że biorąc pod uwagę wszystkie założenia, model sprawdza się zarówno w przypadku testowania prostych układów dwóch kul dając wyniki zgodne z teoretycznymi oraz daje realistyczne efekty dla układów wielu obiektów na przykładzie rozbicia kul bilardowych.

Autorka pracy w trakcie zgłębiania części teoretycznej, projektowania i implementacji usystematyzowała zagadnienia części mechaniki klasycznej wymagane dla zrozumienia istoty zderzenia idealnie sprężystego. Dzięki wykorzystaniu zdobytej wiedzy, powstała biblioteka silnika fizyki, która została wykorzystana w interfejsie graficznym. Wizualizacja symulacji wraz z interakcją z użytkownikiem została zrealizowana w nowo poznanym systemie Microsoft XNAGame Studio 4.0, dzięki czemu autorka miała możliwość poszerzenia swojej wiedzy o znajomość nowego środowiska.

Praca dyplomowa zawiera opis wiedzy teoretycznej niezbędnej do zrozumienia zagadnienia zderzenia idealnie sprężystego, a także najciekawsze problemy, które należało rozwiązać aby móc zaimplementować projekt. Przedstawiony powyżej projekt jest jedynie wstępem do świata zderzeń, który ma na celu zainteresować potencjalnych odbiorców

mechaniką klasyczną, jak i daje możliwość przetestowania niestandardowych ustawień i konfrontacji oczekiwanych wyników z wyliczonymi w trakcie symulacji.

SPIS LITERATURY

[1] M. Skorko, Fizyka, Warszawa: Państwowe Wydawnictwo Naukowe, 1973.

[2] D. Halliday, R. Resnick, J. Walker, "Podstawy fizyki, tom I", Warszawa: Wydawnictwo naukowe PWN, 2007.

[3] D. M. Bourg, Fizyka dla programistów gier, Helion, 2003.

[4] Wikipedia: zderzenia sprężyste, http://pl.wikipedia.org/wiki/Zderzenie_sprężyste.

[5] R. Miles, Microsoft XNA Game Studio 4.0, Projektuj i buduj gry dla konsoli Xbox 360, urządzeń z systemem Windows Phone 7 i własnego PC, Helion, 2012.

ZAŁĄCZNIK

Do pracy dyplomowej załączono archiwum całego projektu w postaci płyty CD zawierającej poniższe katalogi i pliki:

1. Katalog "*collision"* zawierający kod źródłowy finalnego produktu - bibliotekę CollisionLibrary, testy jednostkowe biblioteki CollisionLibraryTestProject oraz projekt aplikacji XNA collision wykorzystującej ją.
2. Katalog "*publish"* zawierający pliki instalacyjne i samą aplikację
3. Katalog "*diagrams"* zawierający pliki graficzne z diagramami UML
4. Treść powyższej pracy w pliku *"Implementacja wybranego modelu zderzeń w grach dwuwymiarowych.pdf"*