
POLITECHNIKA WARSZAWSKA

WYDZIAŁ MATEMATYKI
I NAUK INFORMACYJNYCH

PRACA DYPLOMOWA INŻYNIERSKA

# Symulacja oddziaływań grawitacyjnych na urządzeniach mobilnych Android.

Autor:
Sławomir Płodczyk

Współautorzy:
Mateusz Kaczmarski,
Jakub Ruszkowski

Promotor:
dr inż. Joanna Porter-Sobieraj

WARSZAWA, LUTY 2015 r.

……
podpis promotora

……
podpis autora

# Streszczenie

Niniejszy dokument jest opisem projektu inżynierskiego realizującego temat *Symulacja oddziaływań grawitacyjnych na urządzeniach mobilnych Android*. W ramach tego projektu została wykonana gra zręcznościowa w technologii 3D. Polega ona na sterowaniu kulką umieszczoną na ekranie poprzez odpowiednie przechylanie telefonu. Na kulkę działa siła grawitacji zdefiniowana w lokalnym układzie poprzez położenie urządzenia w świecie rzeczywistym. Odpowiednio przechylając telefon, przemieszczamy kulkę zgodnie z działającą na nią siłą. Celem gry jest takie sterowanie kulką, aby dotrzeć z nią od punktu początkowego do mety.

Aplikacja została wykonana w technologii Android. Korzysta ona z danych pobieranych z akcelerometru urządzenia mobilnego. Obiekty z gry wyświetlane na ekranie rysowane są z użyciem biblioteki OpenGL ES 2.0.

Gra składa się z 5 poziomów o różnej trudności. W grze kulka porusza się po ograniczonych platformach. Elementy znajdujące się na planszy zmieniają tor ruchu oraz prędkość kulki. Wpływ na ostateczny wynik mają zbierane diamenty i klepsydry. Osiągnięcie punktu mety jest możliwe tylko po odwiedzeniu wszystkich punktów kontrolnych na planszy. Przekroczenie limitu czasowego lub spadek z platformy skutkuje przegraną na danym poziomie. Ostateczny wynik zależny jest od zdobytych diamentów i pozostałych do ukończenia etapu sekund i przeliczany jest w skali 0-3 gwiazdki.

Aplikację można podzielić na 3 moduły. Pierwszy z nich – moduł fizyczny odpowiedzialny jest za część fizyczną związaną z poruszaniem się kulki, poruszaniem się elementów ruchomych na planszy oraz odbiciami kulki od innych obiektów. Drugi moduł – moduł graficzny odpowiedzialny jest za generowanie i wyświetlanie obiektów trójwymiarowych znajdujących się na planszy. Dodatkowo ta część ma za zadanie generowanie i wyświetlanie cieni. W ramach trzeciego modułu - modułu interfejsu został wykonany dwuwymiarowy interfejs graficzny. Zawiera on możliwość sprawdzenia podstawowych danych o grze i zmiany kilku opcji.

Aplikacja wymaga współdziałania wszystkich modułów. Za stan gry odpowiada moduł interfejsu. W trakcie rozgrywki wywołuje on moduł fizyczny w celu odświeżenia stanu obiektów. Następnie, aby odrysować obiekty na nowych pozycjach używany jest moduł graficzny. Po wygraniu lub przegraniu poziomu aplikacją zarządza moduł interfejsu.

# Spis treści

# 1 . Wstęp

W trakcie rozwoju informatyki cel i sposób prezentacji pisanych programów ulegał wielu zmianom. Obecnie odchodzi się od aplikacji pisanych jako programy na komputer osobisty. Popularne stały się aplikacje korzystające z połączenia internetowego w architekturze klient-serwer. Rozwój urządzeń mobilnych spowodował możliwość tworzenia aplikacji przeznaczonych wyłącznie na te urządzenia. Aplikacje te mogą dodatkowo korzystać z informacji o swojej lokalizacji lub przestrzennym położeniu otrzymywanych od hardware'u telefonu lub tabletu. Dzięki tym danym programy projektowane na te urządzenia udostępniają nowe funkcjonalności. Wzrost mocy obliczeniowych kart graficznych i ich dostępność na urządzeniach mobilnych dają możliwość płynnego wyświetlania ruchu obiektów. Elementy mobilności, oddziaływania ze światem rzeczywistym oraz grafiki 3D są podstawowymi cechami charakteryzującymi grę *Paranormal Ball* wykonaną w ramach projektu inżynierskiego.

Dokument ten zawiera opis wyglądu aplikacji, a także szczegółów rozwiązań technicznych zawartych w programie. Został on podzielony na kilka rozdziałów.

Rozdział pierwszy jest wprowadzeniem do pracy. Zawiera informację o podziale pracy pomiędzy członków zespołu. Na koniec umieszczony został spis i wyjaśnienia trudniejszych pojęć używanych w opisie.

Rozdział drugi przedstawia całą logikę gry, a także opis interfejsu użytkownika. Na początku zawarto skrócony zarys aplikacji. Później przedstawione są elementy znajdujące się na planszach. Potem zawarta jest instrukcja użytkownika objaśniająca ekrany wyświetlane podczas rozgrywki. Na koniec umieszczone są zasady gry oraz informacje o elementach dodatkowych – dźwięku i wibracjach.

Rozdział trzeci opowiada o szczegółach technicznych zaimplementowanych w aplikacji. Rozpoczyna się on opisem technologii, z których korzystaliśmy w projekcie oraz opisem układu współrzędnych zdefiniowanym w aplikacji. Następnie przedstawione są aspekty odnoszące się grafiki 3D. Podrozdział ten zawiera informacje o sposobie generowania modeli obiektów oraz ich rysowania na ekranie urządzenia mobilnego. Kolejny podrozdział opowiada o części fizycznej aplikacji. Zawarty jest tam zarys sposobu odświeżania położenia elementów, opis modeli wszystkich obiektów znajdujących się na planszy, a także sposób rozwiązania równania ruchu dla kulki oraz kolizji kulki z innymi elementami. Na koniec umieszczony jest używany przez nas algorytm rozwiązywania równań różniczkowych. Następy podrozdział przedstawia sposoby przetwarzania danych planszy oraz opis kontrolek

używanych w aplikacji. Przedstawione są w nim mechanizmy, którymi posługuje się aplikacja, aby zapisywać najlepsze wyniki gracza czy jego preferencje dotyczące ustawień gry. Ostatni podrozdział części technicznej mówi o elementach towarzyszących projektowi – dokumentacji kodu oraz testom jednostkowym.

Rozdział czwarty podsumowuje naszą aplikację oraz zarysowuje wnioski, które nasunęły się nam podczas projektowania i implementacji gry.

## Podział obowiązków

W pracy inżynierskiej zastosowaliśmy poniższy podział obowiązków:

- Szata graficzna, animacja, tworzenie modeli 3D, projekty plansz, opracowanie definicji sceny, rozdziału o części graficznej oraz informacji o bibliotece OpenGL ES 2 - Mateusz Kaczmarski.
- Zaprojektowanie i implementacja modułu fizycznego aplikacji, opracowanie rozdziału o części fizycznej aplikacji oraz informacji o akcelerometrze- Sławomir Płodczyk.
- Wykonanie okien menu oraz paska postępu na ekranie gry, projekty plansz i ich utworzenie, opracowanie opisu gry oraz części technicznej odnoszącej się systemu Andoid – Jakub Ruszkowski.

Promotorem naszej pracy jest dr inż. Joanna Porter-Sobieraj, której chcielibyśmy tutaj podziękować za pomoc i współpracę.

## Słownik

- elementy bonusowe – obiekty zdefiniowane w układzie sceny, ich zdobycie pozwala na uzyskanie lepszego wyniku
- elementy na planszy – obiekty zdefiniowane w układzie sceny, które zmieniają tor ruchu kulki
- Paranormal Ball- nazwa aplikacji na urządzenia mobilne; gra, której implementacja jest celem projektu inżynierskiego
- plansza – przestrzeń trójwymiarowa zawierająca płaską powierzchnię oraz elementy na planszy i elementy bonusowe, zdefiniowana w układzie sceny i rzutowana na ekran urządzenia mobilnego
- układ sceny – układ współrzędnych zdefiniowany w aplikacji. Względem niego określana jest pozycja wszystkich innych elementów w aplikacji

- urządzenie mobilne – urządzenie posiadające system operacyjny Android, a także wyposażone w podstawowy sensor – akcelerometr
- ekran– sformułowanie określające pojedyncze okno gry np.: menu główne, ekran wyboru planszy, ekran pauzy itd.
- bryła zasięgu widzenia/światła (ang.frustum) – figura geometryczna opisująca pole widzenia kamery lub zasięg promieni światła. Ma kształt ściętego ostrosłupa o podstawie prostokąta. Kamera lub źródło światła znajduje się w punkcie przecięcia przedłużenia krawędzi bocznych tego ostrosłupa.

# 2 . Gra ParanormalBall

## *2.1 Wstęp*

Gra składa się z 5 poziomów. Użytkownik na każdym z poziomów ma za zadanie przenieść kulkę od punktu startowego do mety. Na kulkę umieszczoną na ekranie działa siła grawitacji skierowana prostopadle do powierzchni ziemi. W celu ukończenia danej planszy użytkownik przechyla urządzenie mobilne w taki sposób, aby kulka przesuwała się w odpowiednim kierunku. Kamera podczas rozgrywki porusza się za kulką. Dodatkowo istnieje możliwość obrotu kamery poprzez przesuwanie palcem po ekranie. Zdobyty przez użytkownika wynik przeliczany jest na gwiazdki. Najlepsze wyniki na danych poziomach są zapamiętywane.

Ekrany służące obsłudze gry zostały wykonane w technologii 2D. Użytkownik ma możliwość rozpoczęcia gry i wyboru poziomu, zmiany poszczególnych opcji oraz uzyskania informacji odnośnie pomocy i sterowania w grze.

Do zarządzania głównymi oknami gry wykorzystywane są aktywności. Każda z nich odpowiada za jedno z okien (menu główne, opcje, pomoc, wybór poziomu, ekran gry). Do wyświetlania danych używane są klasy kontrolek dziedziczące po podstawowych oraz rozszerzone o wybór czcionki oraz animacje. Mimo niewielkich różnic pomiędzy kontrolkami dla każdego okna zdefiniowano inne klasy kontrolek tak aby w przyszłości można było w łatwy sposób zmienić zachowanie kontrolek konkretnego okna.

## *2.2 Ekrany aplikacji*

Po za ekranem gry wszystkie okna aplikacji posiadają teksturę w tle [1].

### 2.2.1 Menu główne

Przy uruchomieniu aplikacji pierwszym oknem, które widzi użytkownik jest menu główne (Rysunek 1: Menu główne). Jego zadaniem jest umożliwienie na proste nawigowanie pomiędzy oknami aplikacji.


*Rysunek 1: Menu główne*

Posiada ono 4 przyciski, które pozwalają na następujące akcje:

1. "START" - przejście do ekranu wyboru poziomu, z którego można rozpocząć rozgrywkę
2. "OPCJE" - umożliwia użytkownikowi dostosowanie opcji gry do swoich preferencji
3. "POMOC" - pozwala na zaznajomienie się z głównymi zasadami rozgrywki
4. KONIEC" - zamyka aplikację

### 2.2.2 Opcje

Okno opcji (Rysunek 2: Opcje cz. 1) umożliwia dostosowanie ustawień rozgrywki do swoich preferencji. Wybrane opcje zostaną zapamiętane i odtworzone przy kolejnych uruchomieniach aplikacji.


*Rysunek 2: Opcje cz. 1*


*Rysunek 3: Opcje cz. 2*

*Rysunek 4: Opcje cz. 3*

W oknie dostępne są przyciski:

1. "MUZYKA" - zaznaczenie tej opcji powoduje odtwarzanie muzyki w tle podczas rozgrywania poziomów
2. "DŹWIĘKI" - włączenie/ wyłączenie pozostałych dźwięków w aplikacji
3. "WIBRACJE" - włączenie / wyłączenie wibracji
4. "PL" - zmiana języka na polski
5. "EN" - zmiana języka na angielski
6. "CIENIE" - włączenie / wyłączenie generowania cieni w grze
7. Przewijany pasek z teksturami - kliknięcie na jedną z dostępnych w nim tekstur powoduje zmianę tekstury kulki w grze
8. Okienko z tekturą - pokazuje aktualnie wybraną dla kulki tekstur ę
9. "ANULUJ" - zamyka okno opcji i odrzuca wprowadzone zmiany
10. "ZAPISZ" - zamyka okno opcji i zapisuje wprowadzone zmiany

Jedynie w przypadku wyboru języka oraz tekstury użytkownik ma do dyspozycji kilka możliwości. W pozostałych przypadkach jest to wybór typu tak / nie.

### 2.2.3 Pomoc

Okno pomocy (Rysunek 5: Pomoc) pozwala na zapoznanie się z głównymi zasadami gry.


*Rysunek 5: Pomoc*

Na środku jest wyświetlony przewijany panel z informacjami, pod którym dostępne są przyciski:

1. "MENU" - zamyka okno, wraca do menu głównego
2. "DALEJ" - wyświetla kolejną wskazówkę dotyczącą rozgrywki

### 2.2.4 Wybór poziomu

Ekran wyboru poziomu (Rysunek 6: Wybór poziomu) pozwala na rozegranie wybranej gry oraz sprawdzenie najlepszych wyników uzyskanych na poszczególnych planszach.


*Rysunek 6: Wybór poziomu*


*Rysunek 7: Komunikat*

Dostępne są tutaj:

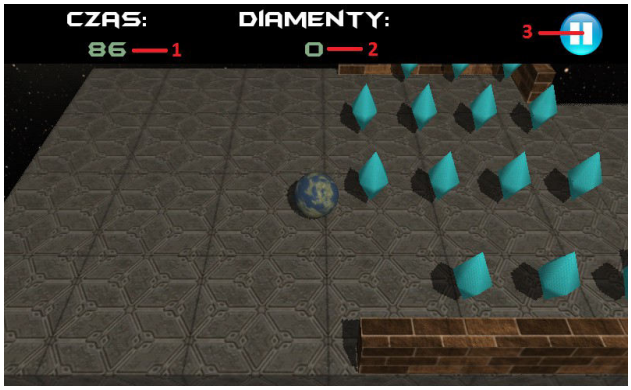
1. Przewijany panel z numerowanymi poziomami - pozwala na wybór gry. Niebieska ramka wskazuje, który poziom jest aktualnie wybrany.
2. Blokada - oznacza, że dany poziom jest jeszcze zablokowany. Aby go odblokować należy przejść wszystkie poprzedzające go.

3. Zdjęcie planszy - po wybraniu poziomu, zostaje wyświetlone w tym miejscu zdjęcie planszy.
4. Panel z gwiazdkami [2] - pokazuje uzyskaną przez gracza liczbę gwiazdek na danym poziomie.
5. Najlepszy wynik punktowy na danym poziomie.
6. "WSTECZ" - zamyka okno i powraca do menu głównego
7. "START" - rozpoczyna grę na wybranym poziomie

Po wybraniu zablokowanego poziomu zostaje wyświetlony komunikat informujący o tym (Rysunek 7: Komunikat). Naciśnięcie „OK” zamyka komunikat i pozostawia poprzedni wybór poziomu.
Przy zmianie wyboru poziomu ładowane jest jego zdjęcie oraz wyświetlany zostaje najlepszy dotychczas uzyskany na nim wynik oraz liczba zdobytych gwiazdek.
Przy przejściu do tego okna z menu głównego aplikacja automatycznie wybiera do rozegrania ostatni odblokowany przez użytkownika poziom (czyli taki, którego jeszcze nie ukończył lub ostatni jeśli ukończył już wszystkie). Natomiast po rozegraniu konkretnego poziomu i powrotu do tego okna z ekranu gry aplikacja wybiera ten sam poziom w przypadku porażki lub kolejny (jeśli istnieje) w przypadku zwycięstwa.
Początkowo jedynie pierwszy poziom gry jest odblokowany i dostępny do rozegrania.

### 2.2.5 Ekran gry

Ekran gry (Rysunek 8: Ekran gry) przedstawia aktualną rozgrywkę oraz informuje gracza o najważniejszych informacjach.



*Rysunek 8: Ekran gry*

*Rysunek 9: Menu pauzy*

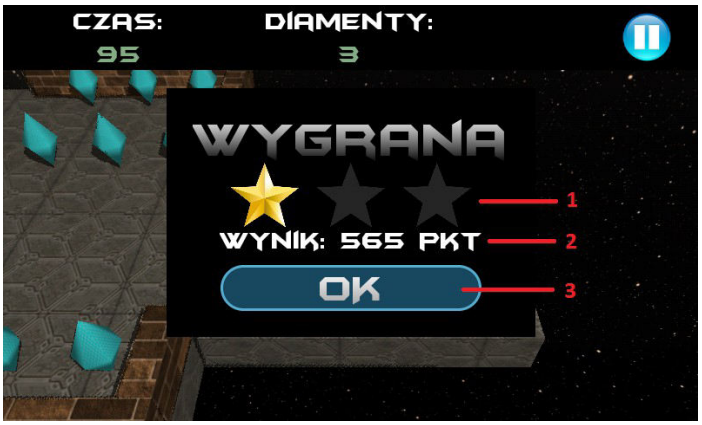
1. Czas - pozostały w sekundach czas na ukończenie poziomu.
2. Diamenty - zebrane na danym poziomie diamenty.

3. Pauza - przycisk pozwalający na zatrzymanie gry.

Po wciśnięciu pauzy [3] ukazuje się menu pauzy (Rysunek 9: Menu pauzy), w którym użytkownik ma dostępne następujące opcje:

1. "WZNÓW" - wznawia zatrzymaną grę
2. "ZACZNIJ OD NOWA" - rozpoczyna od nowa rozgrywkę na danym poziomie (czas, zebrane diamenty oraz wszystkie elementy planszy zostają ustawione na domyślne wartości)
3. "OPCJE" - pozwala na przejście do okna opcji i zmianę pewnych ustawień (muzyki, dźwięków oraz wibracji)
4. "MENU" - kończy daną rozgrywkę (nie zapisuje postępu) i wraca do menu głównego

Wznowienie gry można również spowodować przez powtórne naciśnięcie przycisku pauzy lub wciśnięcie przycisku "wstecz" na urządzeniu.
W przypadku wygranej lub przegranej na danym poziomie wyświetlany jest kolejny komunikat informujący o rezultacie (Rysunek 10: Komunikat o rezultacie).




*Rysunek 10: Komunikat o rezultacie*

Informuje on użytkownika o uzyskanym wyniku punktowym oraz zdobytych gwiazdkach:

1. Gwiazdki - informacja o liczbie uzyskanych gwiazdek
2. Wynik - informacja o liczbie zdobytych punktów
3. "OK" - zamyka okno gry i powraca do ekranu wyboru poziomu.

## *2.3 Elementy planszy*

Na kulkę poruszającą się po planszy mogą oddziaływać różne elementy, które mogą pomagać lub przeszkadzać w odpowiednim zaliczeniu poziomu.
Elementy planszy są pokryte różnymi teksturami [4].

### Ściany

Elementy te mogą znajdować się w każdym miejscu na planszy. Jeżeli kulka dotknie się ściany, to odbija się od niej sprężyście zgodnie z prawami fizyki.

### Przepaście

Dookoła planszy oraz czasami wewnątrz niej znajdują się przepaście. Jeżeli kulka znajduje się nad przepaścią, to spada w otchłań. Spadek kulki z płaszczyzny poziomu kończy rozgrywkę.

### Lepkie powierzchnie

Oprócz zwykłych płaszczyzn znajdujących się na planszy można spotkać inne płaszczyzny oznaczone inną teksturą. W rejonie tym siła tarcia działająca na kulkę od strony płaszczyzny wzrasta i powoduje spadek prędkości kulki.

### Windy - podnośniki

Wśród różnych płaszczyzn znajdujących się na planszy można znaleźć takie płaszczyzny, które poruszają się w kierunku góra-dół względem płaszczyzny pierwotnej. Umieszczenie kulki na takiej platformie umożliwia jej przedostanie się na wyższy poziom.

### Belki

Są to elementy, które poruszają się w sposób zorganizowany w trakcie danej rozgrywki. Zetknięcie się kulki z rozpędzoną belką powodują zmianę toru ruchu kulki (belka nie zmienia swojej prędkości).

### Obowiązkowe punkty kontrolne

Są to przeważnie okrągłe platformy, przez które użytkownik zobowiązany jest przejść, aby ukończyć daną planszę.

### Elementy bonusowe

Na każdym poziomie znajdują się elementy dodatkowe, które pomagają użytkownikowi w osiągnięciu jak najlepszego wyniku.

*Rysunek 11: Model diamentu*

*Rysunek 12: Model klepsydry*

***Diamenty***

Elementy te zwiększają wynik osiągnięty na danej planszy. Ich zebranie nie jest niezbędne do ukończenia danego poziomu.

Diament w grze jest reprezentowany przez bryłę złożoną z 2 ostrosłupów prawidłowych sześciokątnych złączonych ze sobą postawami (Rysunek 11: Model diamentu).

***Klepsydry***

Zebranie tej rzeczy z planszy powoduje, że czas pozostały do ukończenia planszy na danym poziomie wydłuża się. Pozostały czas ma także znaczenie przy obliczaniu wyniku końcowego dla danej planszy.

Klepsydra w grze jest reprezentowana przez bryłę złożoną z 2 przenikających się stożków (Rysunek 12: Model klepsydry).

## *2.4 Plansze*

Poziom trudności plansz wzrasta wraz z przechodzeniem kolejnych etapów. Początkowa plansza zaprojektowana została z myślą o niedoświadczonych graczach - jest łatwa co daje użytkownikowi możliwość zaznajomienia się ze sterowaniem oraz celem gry. Kolejne plansze są już trudniejsze i wymagają od użytkownika większej koncentracji i precyzji aby utrzymać się na platformie i dostać się do punktu końcowego.

## *2.5 Zasady gry*

### 2.5.1 Warunki wygranej i przegranej

Przegrana w grze następuje w wymienionych sytuacjach:

- Kulka spadnie poniżej wszystkich platform rozmieszczonych na danym poziomie
- Koniec czasu

W obu przypadkach gracz nie dostaje punktów ani gwiazdek za ukończenie poziomu.

Wygrana następuję jedynie w przypadku kiedy gracz doprowadzi kulkę do mety (aktywnej).
Aby aktywować metę (oznaczona przez czerwoną poświatę, a potem zieloną kiedy staje się aktywna) należy doprowadzić kulkę do wszystkich punktów kontrolnych (oznaczonych żółtą poświatą) rozmieszczonych na planszy.

### 2.5.2 Punktacja

Kiedy gracz wygra na danym poziomie, następuje przeliczenie zdobytych punktów.
Punkty przyznawane są za:

- Zdobyte diamenty - 30pkt za każdy diament
- Pozostały czas- 5pkt za każdą sekundę

Następnie na podstawie liczby punktów gracz otrzymuje gwiazdki: pierwszą za samo ukończenie poziomu a kolejne za przekroczenie odpowiednich progów punktowych. Każdy z etapów ma osobny sposób przeliczania zebranych diamentów i pozostałego czasu na punkty.
Pośrednio na punktację mogą mieć wpływ klepsydry - zebranie ich powoduje zwiększenie pozostałego na danym poziomie czasu.

### 2.5.3 Informacje dodatkowe

- Przy pierwszym włączeniu aplikacji użytkownik ma odblokowany jedynie pierwszy poziom.
- Podczas każdej rozgrywki użytkownikowi przysługuje jedno *życie.* W przypadku gdy nie uda się przejść poziomu istnieje możliwość rozegrania go kolejny raz.
- Aby odblokować poziom należy przejść wszystkie poprzedzające go.
- Po przejściu poziomu po raz pierwszy, użytkownik może poprawić osiągnięty wynik, wybierając ponownie z menu dany poziom..

## *2.6 Dźwięk i wibracje*

W trakcie trwania rozgrywki w tle jest odtwarzana muzyka [5]. Zdobycie elementu dodatkowego oraz wygranie lub przegranie poziomu  sygnalizowane jest sygnałem dźwiękowym. Muzykę oraz sygnały dźwiękowe można wyłączyć w menu opcji. Regulacja głośności efektów dźwiękowych możliwa jest za pomocą regulacji głośności urządzenia mobilnego. W przypadku zderzenia się ze ścianą lub belką na planszy następuje efekt wibracji telefonu. Można go także wyłączyć w menu opcji.

# 3 . Techniczny opis aplikacji

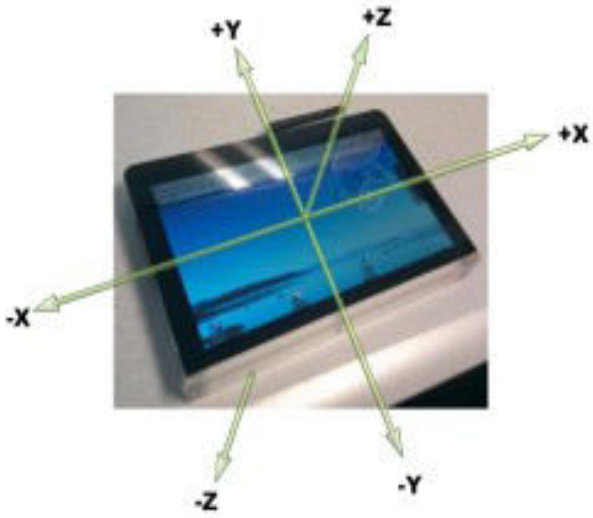
## *3.1 Użyte technologie*

### 3.1.1 Akcelerometr

Akcelerometr jest niewielkim urządzeniem, które pozwala na zmierzenie wartości przyspieszenia ziemskiego lub przyspieszenia związanego z nagłą zmianą położenia. Zbudowany jest z pewnej masy przymocowanej do sprężystych belek. Konstrukcja ta stanowi elektrodę w układzie kondensatora pomiarowego. Zmiana położenia masy pociąga za sobą zmianę pojemności i napięcia wyjściowego. Urządzenie uzyskuje dane o przyspieszeniu w trzech kierunkach. Akcelerometr charakteryzuje się niewielkim poborem prądu.[6]
Sensory te mają wiele zastosowań. Wykorzystuje się je do konstrukcji krokomierza lub stabilizacji obrazu podczas wykonywania zdjęć. W naszej aplikacji używamy akcelerometru, aby możliwe było sterowanie kulką nie używając żadnych przycisków.
System Android umożliwia skorzystanie z danych, które mierzy akcelerometr w bardzo prosty sposób[7]. Jak wspomniano wcześniej, urządzenie to mierzy przyspieszenie w 3 kierunkach. Osie wyznaczające te kierunki pokazano na rysunkach 13. oraz 14.

Aby skorzystać z danych uzyskiwanych przez akcelerometr należy najpierw zgłosić chęć uzyskiwania i przetwarzania poprzez wywołania kilku funkcji. Dodatkowo klasa, która będzie przetwarzała dane z akcelerometru musi implementować interface **SensorEventListener**. Jedna z metod wymaganych do zaimplementowania wywołuje się zawsze, kiedy wartości przekazywane przez sensor zmienią się. Wartości przyspieszenia można w tej metodzie odpowiednio przetworzyć, zapisać i skorzystać z nich w odpowiedniej chwili.


*Rysunek 13: Osie wzdłuż których mierzone jest przyspieszenie na telefonie (za [8])*




*Rysunek 14: Osie wzdłuż których mierzone jest przyspieszenie na tablecie(za [8])*

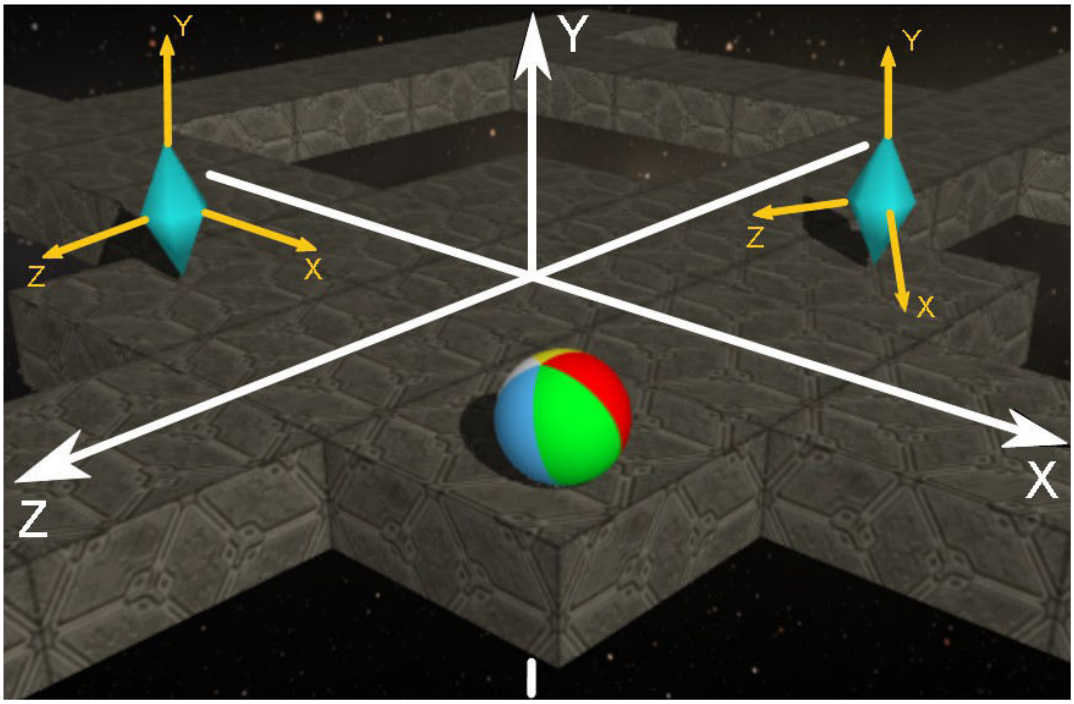
### 3.1.2 Biblioteka OpenGL ES 2.0

Do wizualizacji obiektów na ekranie urządzenia mobilnego wykorzystaliśmy powszechnie stosowaną bibliotekę graficzną OpenGL ES 2.0 (ang. OpenGL for Embedded Systems) [9]. Dlatego też, do prawidłowego działania aplikacji niezbędne jest urządzenie z systemem Android, który posiada dla niej wsparcie (systemy Android od wersji 2.2). OpenGL ES jest uproszczoną wersją wieloplatformowej biblioteki OpenGL przeznaczoną dla przenośnych urządzeń o ograniczonych zasobach, w tym dla telefonów komórkowych oraz tabletów. Wykorzystuje procesor graficzny (układ GPU) do realizacji obliczeń procedur, co znacząco przyspiesza proces tworzenia grafiki 3D i umożliwia uzyskanie płynności animacji nawet dla skomplikowanych obiektów. Niestety wyniki działania biblioteki mogą się różnić w zależności od sposobu implementacji potoku graficznego OpenGL na podzespołach urządzeń mobilnych, przez co wygląd klatki obrazu wygenerowany na różnych urządzeniach może być odmienny.

### *3.2 Definicja sceny*

Podczas trwania rozgrywki, na ekranie urządzenia mobilnego wyświetlana jest część sceny, która w danym momencie znajduje się w polu widoku kamery. Poruszająca się po planszy kulka jest nieustannie obserwowana przez kamerę, która zmienia swoje położenie utrzymując stałą odległość od śledzonego obiektu. Początkowo, kamera ustawiona jest pod kątem 45° do płaszczyzny OXZ. Użytkownik ma jednak możliwość swobodnego obrotu jej pozycji względem środka kulki co pozwala dostosować pole widoku do swoich aktualnych potrzeb.

Każda z plansz występująca w aplikacji zdefiniowana jest w globalnym układzie współrzędnym przedstawionym na rysunku 15. Oś Y tego układu jest prostopadła do płaszczyzny, po której porusza się kulka, z kolei osie X oraz Z określają odpowiednio szerokość i wysokość ekranu urządzenia mobilnego.




*Rysunek 15: Położenie globalnego układu współrzędnych sceny oraz lokalnych układów modeli.*

Jako jednostkę planszy przyjęliśmy długość promienia kulki. To względem niej wyznaczane są rozmiary wszystkich obiektów. Każdy model elementów planszy zdefiniowany jest przez pewną siatkę trójkątów, która z zadaną dokładnością przybliża rzeczywisty kształt obiektu. Modele generowane są w lokalnych układach współrzędnych oznaczonych na rysunku 15 żółtym kolorem. Ustawienie obiektów na planszy polega na odpowiednim przesunięciu ich lokalnych układów oraz zastosowaniu dla nich rotacji w przypadku wystąpienia obrotu elementu.

## *3.3 Moduł graficzny*

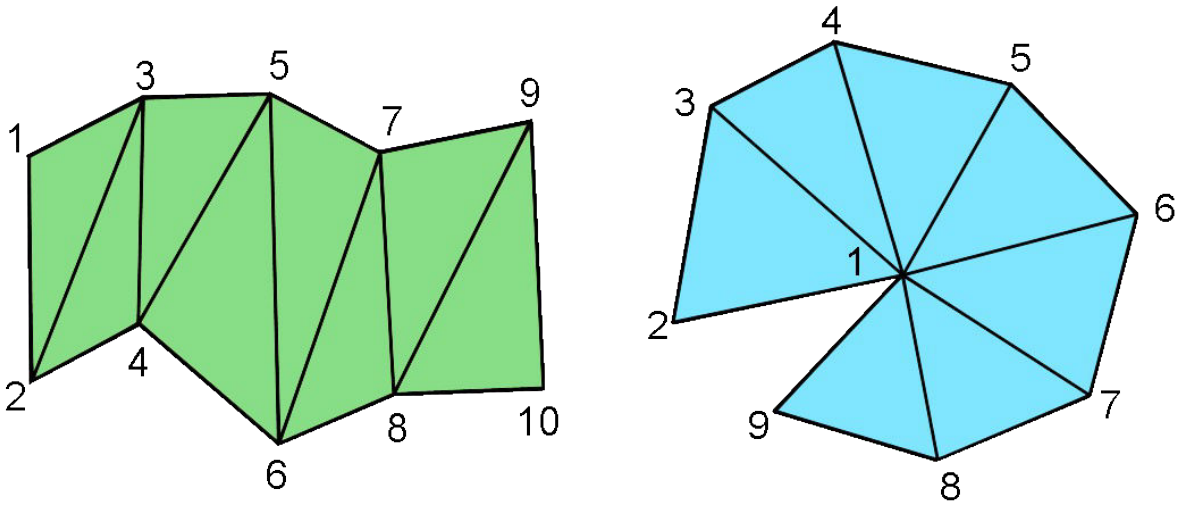
Rozdział ten zawiera informacje o sposobie wyświetlania obiektów na ekranie urządzenia mobilnego oraz metodach tworzenia obiektów. Opisane są w nim również zastosowane algorytmy wyznaczania koloru pikseli oraz mechanizmy optymalizacyjne proces renderowania.

### 3.3.1 Tworzenie modeli

Dane opisujące modele obiektów przechowywane są w klasie **TriangleMeshData** przedstawionej na diagramie 16. Tablica *vertexData* zawiera atrybuty wszystkich wierzchołków siatki trójkątów definiujące między innymi ich położenie w lokalnym układzie, kierunek wektora normalnego skierowanego na zewnątrz obiektu oraz współrzędne tekstur. Dodatkowo, aby móc skorzystać z dostępnych w bibliotece OpenGL ES narzędzi optymalizujących proces renderowania, ważne jest, aby wierzchołki wszystkich trójkątów definiowane były w kolejności odwrotnej do kierunku ruchu wskazówek zegara patrząc na zewnętrzną stronę obiektu. Informacje na temat wykorzystywanych w aplikacji metod optymalizacyjnych znajduje się w rozdziale 3.3.5. *Metody przyspieszające renderowanie sceny.*


*Rysunek 16: Diagramy klasy TrangleMeshData oraz interfejsu DrawCommand.*

Lista *drawCommands* określa sposób rysowania modelu złożonego z wierzchołków zdefiniowanych w tablicy *vertexData*. Sposób ten przechowywany jest w postaci metody *draw* w interfejsie **DrawCommand**. OpenGL ES 2.0 zapewnia możliwość wyświetlania na ekranie jedynie podstawowych obiektów geometrycznych jak punkty, linie oraz trójkąty. Biblioteka ta dostarcza jednak narzędzia pozwalające rysować bardziej skomplikowane kształty złożone z prostych figur. Do tworzenia modeli w aplikacji zastosowaliśmy między innymi wstęgi (ang. triangle strip) oraz wachlarze trójkątów (ang. triangle fan). Sposób ich definiowania zaprezentowany został na poniższym rysunku.



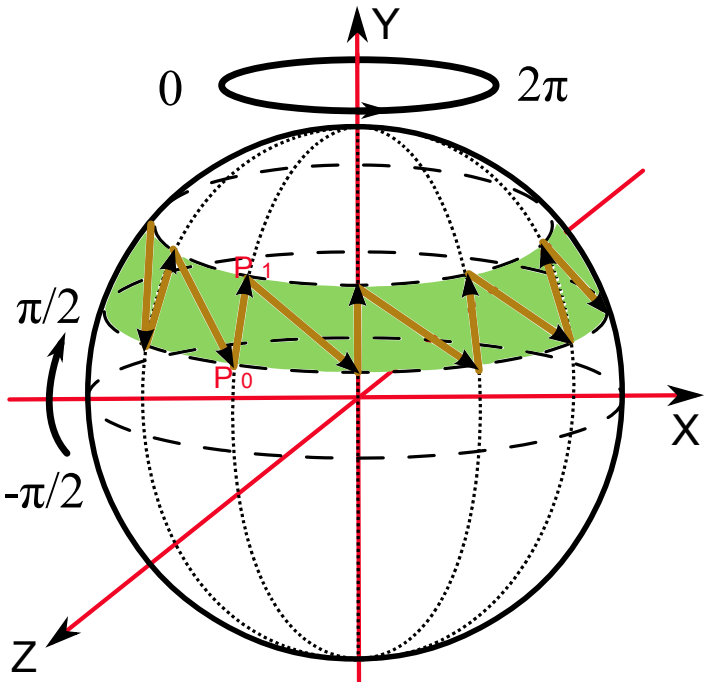

*Rysunek 17: Przykłady dostępnych w OpenGL ES narzędzi modelujących. Od lewej: wstęga oraz wachlarz trójkątów.*

Za pomocą wachlarza trójkątów, z łatwością można z danym przybliżeniem wygenerować koło, czy też boczne ściany ostrosłupa wykorzystywane w modelu diamentu. Z kolei do budowy prostopadłościanów, sfer oraz powierzchni bocznych ściętych stożków posłużyliśmy się wstęgami trójkątów. Sposób ich użycia przedstawiony został na przykładzie schematu 18. generowania sfery. Mając daną długość promienia sfery (zmienna *radius* typu float) oraz liczbę określająca stopień przybliżenia sfery (zmienna *numPoints* typu int) wyznaczane są kolejne wierzchołki siatki trójkątów według algorytmu:

```
for(int i=0; i<numPoints; i++){
        float i0_radian =  −π/2 + i/numPoints · π
        float i1_radian =  −π/2 + (i+1)/numPoints · π
        for(int j=numPoints; j>=0; j--){
                float j_radian =  j/numPoints · 2π

                //Współrzędne punktu P0:
                X0 = r · cos(i0_radian) · cos(j_radian);
                Y0 = r · cos(i0_radian) · sin(j_radian);
                Z0 = r · sin(i0_radian);

                //Współrzędne punktu P1:
                X1 = r · cos(i1_radian) · cos(j_radian);
                Y1 = r · cos(i1_radian) · sin(j_radian);
                Z1 = r · sin(i1_radian);
        }
}
```




*Rysunek 18: Schemat generowania modelu sfery.*

Liczba trójkątów użytych do modelowania sfery przez powyższy algorytm o złożoności kwadratowej wynosi 2 · *numPoints* · *numPoints*. W aplikacji zastosowaliśmy numPoints o wartości 25.

W aplikacji, klasami odpowiedzialnymi za tworzenie modeli wszystkich obiektów są klasy **ObjectBuilder** oraz **ObjectGenerator** zaprezentowane na rysunku 19. Klasa **ObjectBuilder** zawiera szereg metod typu *appendGeometry*, które w zależności od typu obiektu geometrycznego, rozszerzają tablicę *vertexData* o dodatkowe wierzchołki i dodają do listy *drawCommands* reguły rysowania dodanych wierzchołków. Metoda *build* zwraca obiekt **TriangleMeshData** utworzony na podstawie *vertexData* oraz *drawCommands*.


*Rysunek 19: Diagramy klas generujących modele: ObjectBuilder oraz ObjectGenerator.*

Klasa **ObjectGenerator** posiada szereg metod *generateObject*, które korzystając z klasy **ObjectBuilder**, generują docelowe modele definiujące elementy plansz.

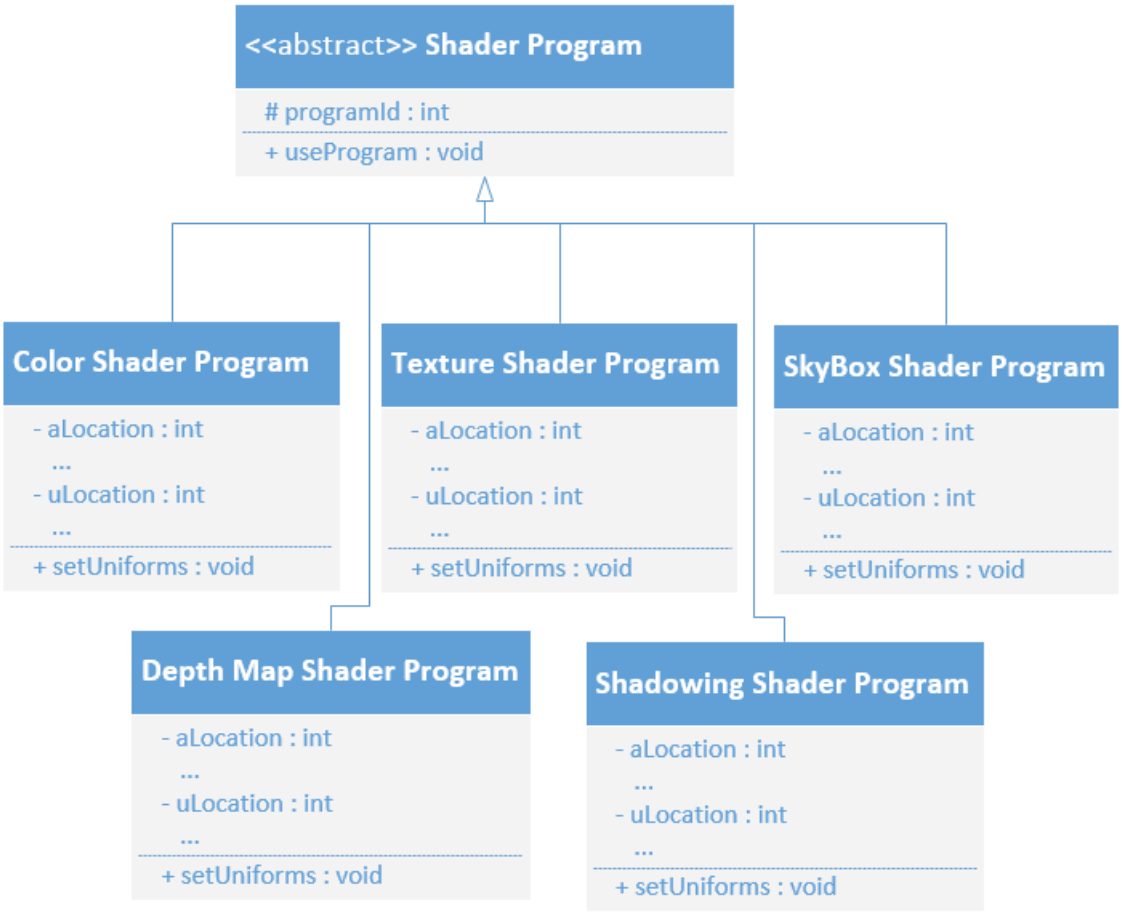
### 3.3.2 Tworzenie programów

Programy (ang. shader program) są to procedury uruchamiane na karcie graficznej. Przekazuje się do nich atrybuty wierzchołków oraz tzn. wartości stałe (ang. uniform) dla danego wywołania. Programy służą do obliczania pozycji obiektów na ekranie urządzenia, wyznaczania kolorów poszczególnych pikseli oraz do tworzenia mapy głębokości cieni. Każdy program składa się z dwóch podprogramów cieniujących: programu cieniowania wierzchołków (ang. vertex shader) oraz programu cieniowania pikseli (ang. fragment shader).

Do rysowania różnych typów obiektów stosowane są różne typy programów opisanych przez klasy dziedziczące po klasie **ShaderProgram**. Posiadają one metodę *setUniforms*, która przekazuje do utworzonego w konstruktorze programu odpowiednie zmienne skojarzone z wartościami stałymi użytymi w plikach kodów programów. W aplikacji zastosowaliśmy pięć rodzai programów:

- **ColorShaderProgram** do rysowania modeli obiektów przy użyciu jednego, określonego koloru,
- **TextureShaderProgram** do mapowania tekstur na modele obiektów (bez generowania ich cieni),
- **SkyBoxShaderProgram** do rysowania sześcianu tła,
- **DepthMapShaderProgram** do tworzenia mapy głębokości cieni stosowaną do generowania cieni obiektów,
- **ShadowingShaderProgram** do mapowania tekstur na modele obiektów wraz z generowaniem ich cieni.

Na rysunku 20. przedstawiony został diagram klas wykorzystywanych programów.



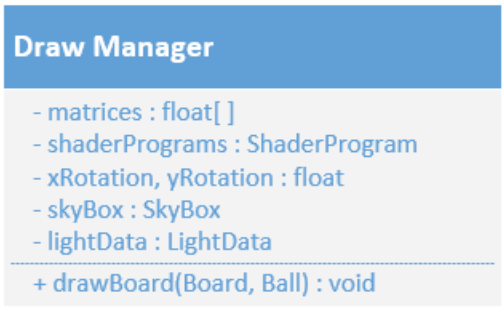

*Rysunek 20: Diagram klas programów.*

Abstrakcyjna klasa **ShaderProgram** zawiera pole *programId* typu int oraz stałe tekstowe określające nazwy atrybutów i wartości stałych (ang. uniform) stosowanych w kodach programów. Posiada także metodę *useProgram*, która zawiadamia OpenGL, aby przy kolejnym procesie renderowania obiektu zastosować dany program.

Powyższe klasy korzystają z pomocniczej klasy **ShaderHelper**, która odpowiedzialna jest za kompilacje przekazanych w parametrze kodów programów dla wierzchołków i fragmentów, łączenie ich w pojedynczy program, sprawdzenie poprawność utworzonego programu i zwrócenie jego identyfikatora OpenGL w przypadku pozytywnego wyniku walidacji.

### 3.3.3 Generowanie klatki obrazu

Klasą odpowiedzialną za generowanie pojedynczej klatki obrazu jest klasa **DrawManager** przedstawiona na poniższym diagramie.




*Rysunek 21: Diagram klasy odpowiedzialnej za generowanie pojedynczej klatki obrazu.*

Zawiera ona wszystkie narzędzia niezbędne do poprawnego rysowania obiektów na ekranie, w tym:

- Macierze do wyliczania pozycji modeli obiektów na ekranie:
  - float[] *modelsMatrix* – macierz modelu stosowana do umiejscowienia modeli obiektów w odpowiedniej pozycji we współrzędnych świata. Wyznaczana osobno dla każdego obiektu występującego na planszy.
  - float[] *viewMatrix* – macierz widoku kamery określająca położenie kamery oraz kierunek jej patrzenia.
  - float[] *projectionMatrix* – macierz projekcji kamery definiująca pole widzenia kamery (określa bryłę zasięgu widoku).
  - float[] *viewProjectionMatrix* – macierz pomocnicza; jest wynikiem iloczynu macierzy projekcji i macierzy widoku.
  - float[] *modelViewProjectionMatrix* – macierz pomocnicza; jest wynikiem iloczynu pomocniczej macierzy viewProjectionMatrix i macierzy modelu.
  - float[] *normalsRotationMatrix* – macierz służąca do odpowiedniej rotacji wektorów normalnych w przypadku obrotu modelu obiektu. Wyznaczana osobno dla każdego obracającego się obiektu.
  - float[] *lightsViewMatrix* – macierz widoku światła określająca położenie jego źródła oraz nominalny kierunek świecenia.
  - float[] *lightsProjectionMatrix* – macierz projekcji światła definiująca zasięg padania jego promieni.

- float[] *lightsViewProjectionMatrix* – macierz pomocnicza; jest wynikiem iloczynu macierzy projekcji i widoku światła.
- float[] *skyBoxViewProjectionMatrix* – macierz będąca wynikiem iloczynu macierzy projekcji kamery i widoku tła. Macierz widoku tła wyznaczana jest na podstawie kątów obrotu kamery i określa część tła, która w danym momencie jest wyświetlana.

- SkyBox *skyBox* – obiekt definiujący przestrzenne tło sceny. Wszystkie plansze umieszczone są w środku sześcianu, którego wewnętrzne ściany mają nałożone tekstury kosmosu.
- Procedury określające metodę rysowania danego modelu:
  - ColorShaderProgram *colorShaderProgram***,**
  - TextureShaderProgram *textureShaderProgram***,**
  - SkyBoxShaderProgram *skyBoxShaderProgram***,**
  - DepthMapShaderProgram *depthMapShaderProgram***,**
  - ShadowingShaderProgram *shadowingShaderProgram***.**
- LightData *lightData* – informacje o położeniu oraz parametrach światła: współczynnik światła otoczenia (ang. ambient) oraz współczynnik światła rozproszonego (ang. diffuse), oba typu float.
- float *xRotation*, *yRotation* – kąty obrotu kamery odpowiednio wokół osi Y i X.
- boolean *withShadow* – informacja o tym, czy opcja generowania cieni jest włączona czy wyłączona.
- int[] *frameBufferObjectId*, *depthTextureId*, *renderTextureId* – identyfikatory przechowujące informacje o buforze klatki wykorzystywanym do tworzenia mapy głębokości cieni.
- int *displayWidth*, *displayHeight*, *depthMapWidth*, *depthMapHeight* – wymiary ekranu oraz mapy cieni.

Główną metodą klasy generującą pojedynczą klatkę jest metoda *drawBoard*, która przyjmuje jako parametr obiekt typu Board zawierający wszystkie elementy występujące na planszy oraz obiekt kulki typu Ball.

```
drawBoard(Board board, Ball ball){
    czyszczenie obszaru rysowania;
    ustawienie nowego pola widoku kamery oraz zasięgu promieni światła na podstawie położenia kulki oraz kąta
    obrotu kamery;
    for(obiekt :  elementy board oraz ball)
```

```
            wyliczenie wartości wszystkich macierzy niezbędnych do prawidłowego umieszczenia obiektu na
                scenie (macierz modelu, macierz obrotu wektorów normalnych, pomocnicza
            macierz modelViewProjection)
            ustawienie wartości stałe (ang. uniform) dla programu;
            powiązanie obiektu z odpowiednim programem (w zależności od tego czy wybrano opcję z
                generowaniem cieni obiektów sceny, czy dany obiekt ma mieć nałożoną teksturę,
            czy też być w jednolitym kolorze);
            obiekt.draw();
}
```

W aplikacji zastosowaliśmy dynamiczne źródło światła, które zmienia swoje położenie wraz z ruchem kulki na scenie. Znajduje się ono zawsze w niewielkiej odległości od kamery, przez co uzyskuje się efekt, jakby gracz przemieszczał się po planszy wyposażony w latarkę. Jedną z wad takiego podejścia jest obowiązek wyliczania nowego pola widoku kamery i zasięgu promieni światła dla każdej klatki. Aby wygenerować cienie wszystkich obiektów w polu widzenia kamery, bryła widoku powinna zawierać się w bryle zasięgu światła. Dodatkowo, aby algorytm generowania cieni był optymalny, należało zadbać o to, aby bryła światła był jak najmniejsza.

### 3.3.4 Algorytmy cieniowania obiektów

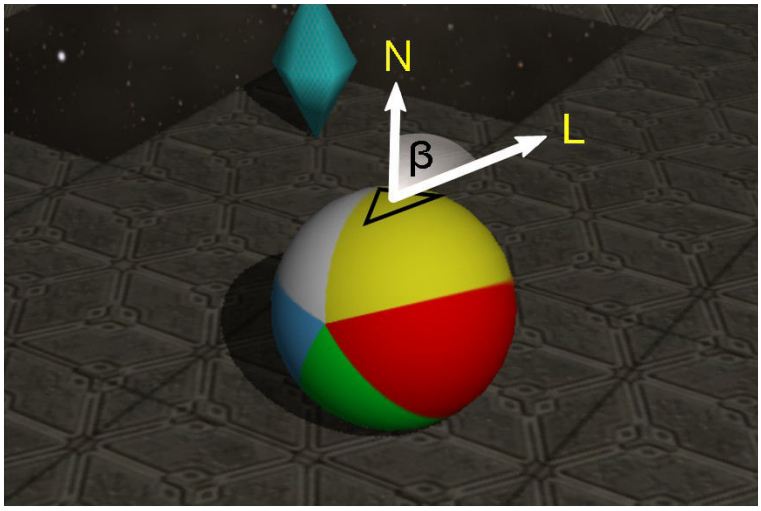
Do cieniowania obiektów na planszy zastosowaliśmy model uwzględniający jedynie odbicie rozproszone, pochodzące od oświetlenia otoczenia oraz źródła światła zdefiniowanego na scenie. Natężenie światła, które dociera do punktu obiektu, dla którego wyliczamy kolor jest sumą natężenia światła otoczenia (ang. ambient) oraz natężenia światła rozproszonego (ang. diffuse).

$$I_{light} = I_{ambient} + I_{diffuse}$$

Natężenie światła otoczenia ma stałą wartość zdefiniowaną przez współczynnik ambient w obiekcie LightData. Z kolei natężenie światła rozproszonego obliczana jest dla każdego punktu modelu według wzoru:

$$I_{diffuse} = k_{diffuse} \cdot \cos \beta$$

gdzie $k_{diffuse}$ jest stałym współczynnikiem rozproszenia, a $\beta$ jest kątem między wektorem normalnym a wektorem określającym położenie źródła światła (rysunek 22).




*Rysunek 22: Kąt β między wektorem normalnym a wektorem światła na przykładzie jednego z trójkątów modelu kulki.*

Ostatecznie, wartość uzyskanego natężenia światła mnożona jest przez wartość koloru danego piksela. Wartość koloru zdefiniowana jest za pomocą modelu RGBA i przechowywana w czteroelementowej tablicy liczb zmiennoprzecinkowych (float) z przedziału od 0.0 do 1.0.

Pierwotny kolor piksela obiektu (np. kolor z tekstury przypisanej danemu obiektowi):

$$natural\_color = [r_{nat}, g_{nat}, b_{nat}, a_{nat}]$$

Kolor piksela po zastosowaniu cieniowania:

$$after\_shadowing\_color = I_{light} \cdot natural\_color = [I_{light} \cdot r_{nat}, I_{light} \cdot g_{nat}, I_{light} \cdot b_{nat}, I_{light} \cdot a_{nat}]$$

Do generowania cieni obiektów wykorzystaliśmy mapę głębokości cieni, którą uzyskujemy poprzez renderowanie sceny z punktu widzenia światła a następnie zapisujemy do niej informację o głębokości [10][11][12]. Podczas wyliczania ostatecznego koloru piksela sprawdzane jest, czy głębokość tego piksela w układzie współrzędnych światła jest większa niż jego głębokość odczytana z mapy cieni. Jeśli tak, to piksel znajduje się w cieniu i jest zaciemniany.

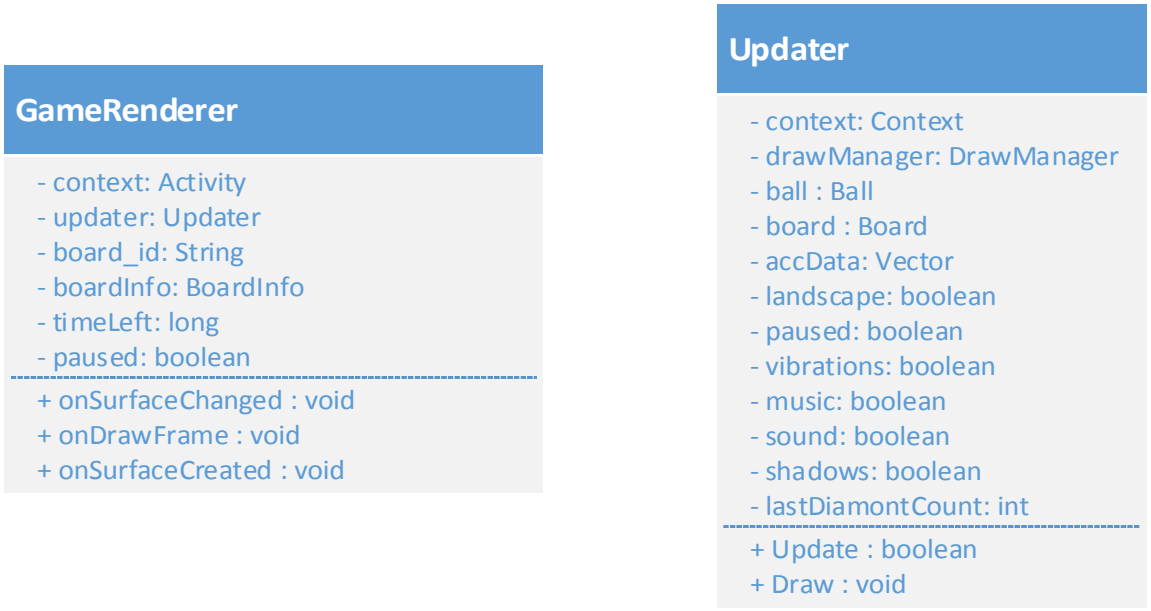
Podczas implementacji generowania cieni natknęliśmy się na problem występowania ciemnych artefaktów na oświetlanych obiektach. Częściowo usunęliśmy je stosując dynamiczną wartość przesunięcia (ang. dynamic bias) głębokości piksela odczytaną z mapy cieni [13]. Jednak dla pewnej konfiguracji położenia i obrotu kamery można w dalszy ciągu zauważyć niewielkie artefakty.

### 3.3.5 Metody przyspieszające renderowanie sceny.

Podczas implementacji aplikacji, szczególny nacisk kładliśmy na zapewnienie płynności animacji. Aby ją uzyskać, zastosowaliśmy kilka metod przyspieszających tworzenie klatki obrazu. Jedną z nich jest tzw. back face culling, czyli usuwanie niewidocznych powierzchni. Podczas procesu renderowania, pomijane są tylne ściany modeli obiektów, co pozwoliło na uzyskanie poprawy wydajności o 40-50%. Podobny mechanizm użyliśmy przy generowaniu mapy głębokości cieni. W tym przypadku, do mapy zapisywane są informacje o głębokości cienia przy pominięciu cienia tworzonego przez przednie ściany modelu. Kolejnym sposobem optymalizacji renderowania sceny jest odrzucenie części obiektów znajdujących się poza polem widoku kamery (ang. viewing frustum culling).

## *3.4 Moduł fizyczny*

### 3.4.1 Odświeżanie położenia obiektów




*Rysunek 23: Klasy odpowiedzialne za odświeżanie położenia obiektów*

Klasami odpowiedzialnymi za zmianę położenia elementów w czasie, a także wywołującymi odpowiednie metody za nie odpowiedzialne są **GameRenderer** i **Updater**.

Klasa **GameRenderer** – implementuje interfejs **Renderer**, do odświeżania pozycji korzysta z odpowiednich metod klasy **Updater**.

Klasa **Updater –** odświeża pozycje elementów oraz zarządza procesem aktualizacji danych. Implementuje interfejs umożliwiający pobieranie danych z akcelerometru. Zawiera pola:

- obiekt typu **Board** opisujący bieżącą planszę (główną powierzchnię, przeszkody, bonusy, punkty początkowe)
- obiekt typu **Ball** opisujący kulkę
- liczbę diamentów pozostałych do zebrania
- dane odnośnie przyspieszenia pobrane z akcelerometru
- pola zawierające aktualne ustawienia dotyczące obecności muzyki, cieni itp., a także informację o orientacji ekranu urządzenia
- obiekt typu **DrawManager** wykorzystywany przy odrysowywaniu planszy

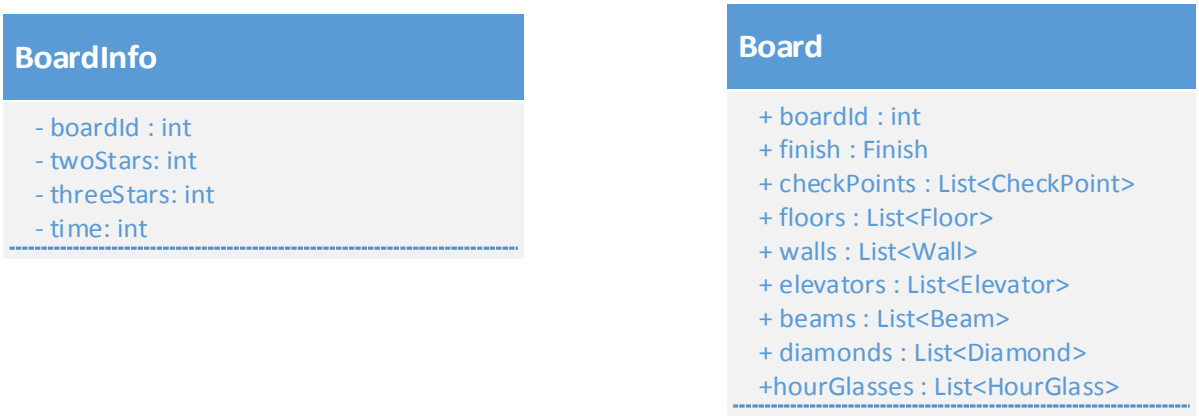
Udostępnia ona także następujące metody:

- *Update* - metoda dla danych z akcelerometru oraz aktualnych prędkości i położeń dla wszystkich elementów oblicza kolejne położenia elementów. Przyjmuje jako argument czas, który mija pomiędzy kolejnymi rysowanymi pozycjami w symulacji.

```
UpdateResult Update(float dt){
        sprawdzamy, czy kulka znajduje się na podłodze lub windzie;
        for(obiekt : beams oraz elevators)
                odświeżamy pozycję obiektu;
        odświeżamy pozycję kulki;
        jeśli kulka znajduje się poniżej wszystkich podłóg i wind
                gra kończy się porażką;
        for(obiekt: elementy board)
                jeśli występuje kolizja z obiektem
                        dokonujemy zmiany stanu gry lub korygujemy położenie kulki;
        zwracamy informację o stanie gry;
}
```

### 3.4.2 Organizacja planszy

Do pogrupowania danych dla planszy służą następujące klasy:



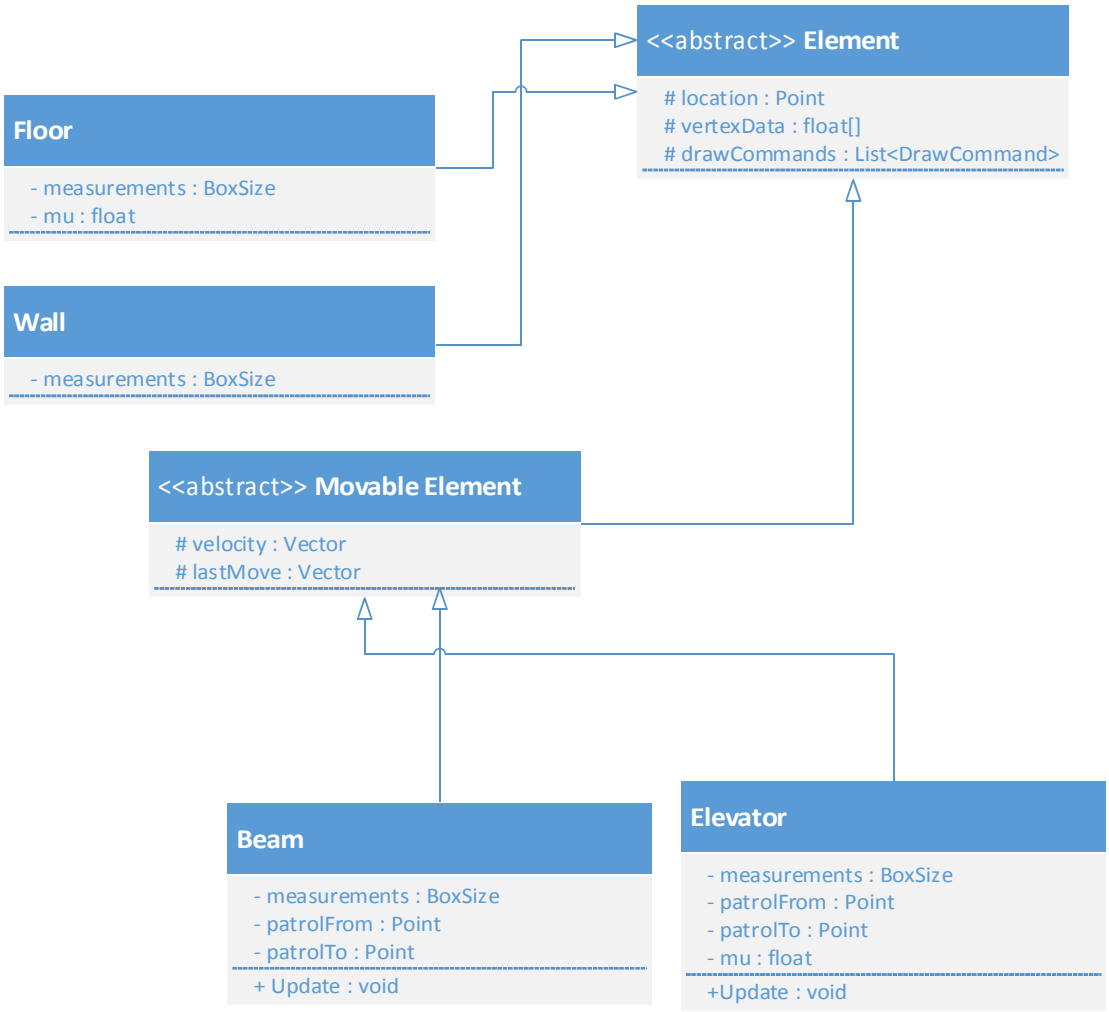

*Rysunek 24: Klasy służące pogrupowaniu danych*

Klasa **Board –** klasa grupująca informacje o aktualnej sytuacji na planszy. Posiada listy wszystkich elementów poza kulką, które znajdują się na planszy.

Klasa **BoardInfo** – zawiera dodatkowe dane na temat planszy. Posiada informację o czasie, w jakim należy pokonać daną planszę oraz przeliczniki konkretnych wyników na gwiazdki w końcowej ocenie.

### 3.4.3 Elementy na planszy

Dla każdego obiektu na planszy stworzono odpowiadający mu model. Na rysunkach 25 i 26 przedstawiono klasy odpowiadające elementom znajdującym się na planszy:




*Rysunek 25: Klasy odpowiadające elementom na planszy cz. 1*

Klasa **Element** – klasa abstrakcyjna, podstawowa klasa dla wszystkich elementów znajdujących się na planszy, zawiera informacje o położeniu środka elementu, a także dane dla modelu pomagające w wyświetleniu danego obiektu. Posiada także metodę *draw***()**, która wywołuje po kolei wszystkie metody *draw*() na liście *drawCommands* oraz metodę *bindAttributes* przyjmującą jako parametr obiekt klasy **ShaderProgram** oraz typ wyliczeniowy określający rodzaj podanego programu cieniującego. Metoda ta wiąże atrybuty wierzchołków modelu danego elementu z atrybutami występującymi w plikach podprogramów używanych przez przekazany program cieniujący.

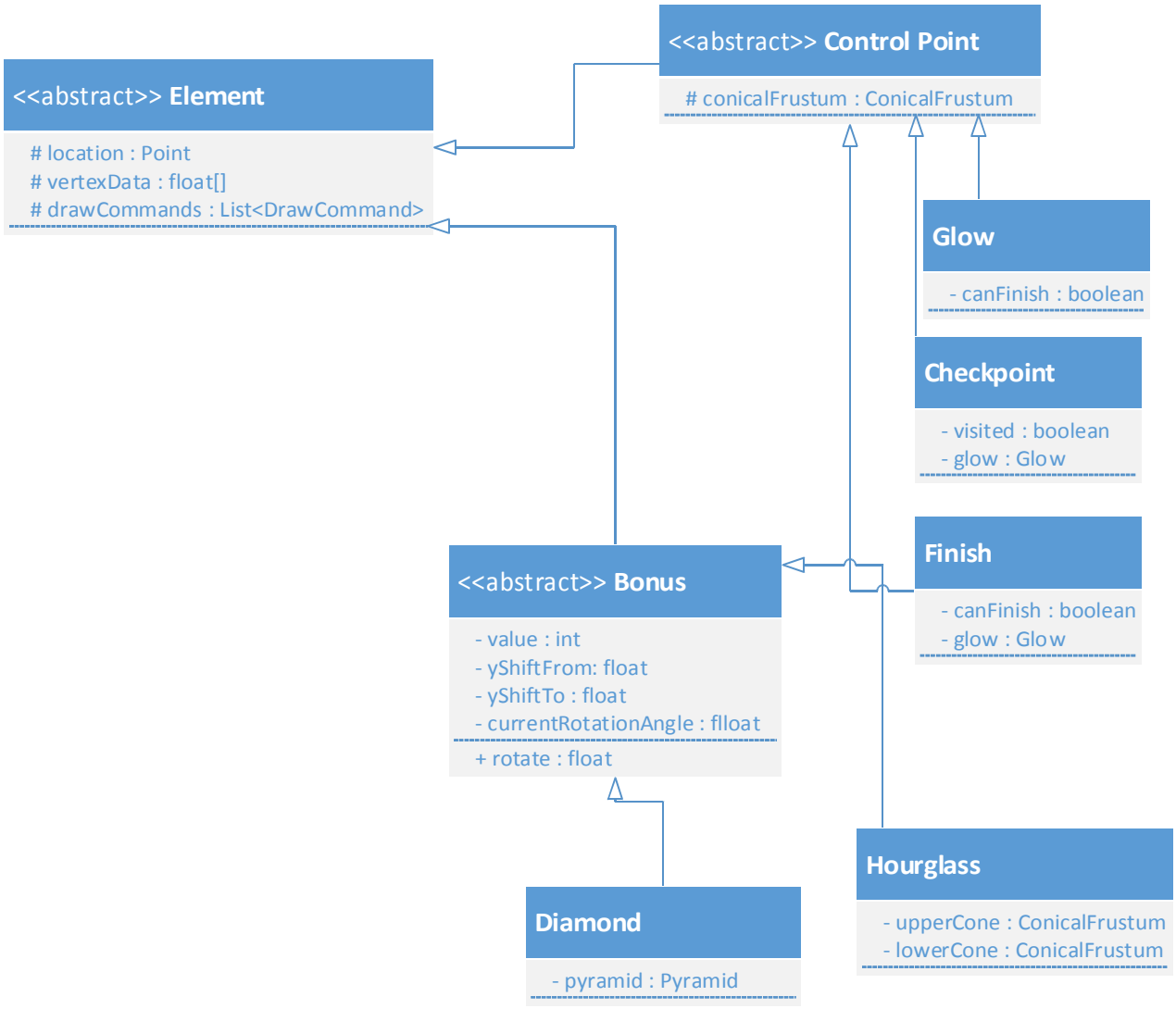
Klasa **Wall** – klasa dziedzicząca po **Element**, odpowiada ścianom znajdującym się na planszy, zawiera informację o rozmiarze elementu. Elementy tej klasy są prostopadłościanami równoległymi do osi układu współrzędnych.

Klasa **Floor** – klasa dziedzicząca po **Element**, odpowiada podłodze znajdującej się na planszy, zawiera informację o rozmiarze elementu. Elementy tej klasy są prostopadłościanami równoległymi do osi układu współrzędnych. Dodatkowo klasa ta zawiera informację o współczynniku tarcia dla powierzchni.

Klasa **MovableElement** – klasa abstrakcyjna dziedzicząca po **Element**, zawiera informację o aktualnej prędkości elementu – wektor *velocity* oraz przebytym pomiędzy 2 ostatnimi klatkami odcinku drogi – wektor przesunięcia *lastMove*.

Klasa **Elevator** – klasa dziedzicząca po **MovableElement**, odpowiada windom – podnośnikom. Zawiera informacje o rozmiarze oraz maksymalnym wychyleniu w jedną i drugą stronę. Elementy tej klasy są prostopadłościanami równoległymi do osi układu współrzędnych. Dodatkowo klasa ta zawiera informację o współczynniku tarcia dla powierzchni. Klasa udostępnia także metodę Update służącą do odpowiedniej zmiany położenia elementu w zależności od czasu, który upłynął pomiędzy 2 ostatnimi klatkami.

Klasa **Beam** – klasa dziedzicząca po **MovableElement**, odpowiada belkom. Zawiera informacje o rozmiarze oraz ekstremalnym położeniu w jedną i drugą stronę. Elementy tej klasy są prostopadłościanami równoległymi do osi układu współrzędnych. Klasa udostępnia także metodę Update służącą do odpowiedniej zmiany położenia elementu w zależności od czasu, który upłynął pomiędzy 2 ostatnimi klatkami.




*Rysunek 26: Klasy odpowiadające elementom na planszy cz. 2*

Klasa **Bonus** – klasa abstrakcyjna dziedzicząca po **Element**, zawiera informację o wartości bonusu związanego z elementem. Dla uatrakcyjnienia rozgrywki elementy – bonusy poruszają się, tzn. obracają się i delikatnie podnoszą się do góry i opadają. Klasa ta zawiera zatem informacje o kącie obrotu elementu wzdłuż osi OY oraz górnej i dolnej granicy położenia elementu. Udostępnia dodatkowo metodę *rotate*() służącą do implementacji obrotu elementu znajdującego się na planszy.

Klasa **Diamond** – klasa dziedzicząca po **Bonus**, reprezentuje diament. Diament rysowany jest w postaci 2 ostrosłupów, których wysokości są równoległe do osi OY układu współrzędnych; klasa zawiera informacje o rozmiarach ostrosłupów.

Klasa **Hourglass** - klasa dziedzicząca po **Bonus**, reprezentuje klepsydrę. Klepsydra rysowana jest w postaci 2 ściętych stożków, których wysokości są równoległe do osi OY układu współrzędnych; klasa

zawiera informacje o rozmiarach tych brył. Klasa ta zawiera pomocniczą klasę **HourGlassWoodenParts**, która odpowiada za poprawne rysowanie drewnianych części klepsydry.

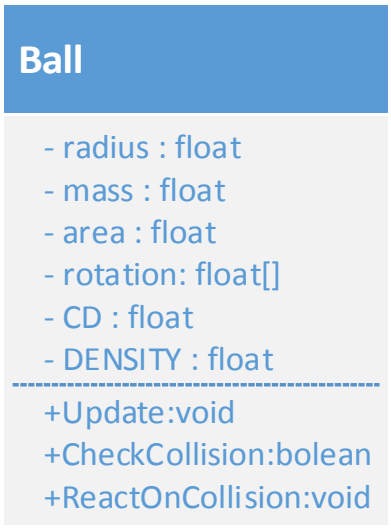
Klasa **ControlPoint –** klasa abstrakcyjna dziedzicząca po **Element**, reprezentuje punkty mety i punktu kontrolnego. Punkty te rysowane są w postaci ściętego stożka.

Klasa **Glow –** dziedziczy po **ControlPoint**, pomocnicza klasa służąca do rysowania poświaty dla obiektów klasy **ControlPoint**. Zawiera zdefiniowane w modelu RGBA możliwe kolory poświaty.

Klasa **Finish –** klasa dziedzicząca po **ControlPoint**, reprezentuje punkt mety. Zawiera pole określające, czy możemy już ukończyć poziom, a także pomocniczy element klasy **Glow**.

Klasa **CheckPoint –** klasa dziedzicząca po **ControlPoint**, reprezentuje punkt kontrolny. Zawiera pole określające, czy dany punkt został już odwiedzony, a także pomocniczy element klasy **Glow**.

### 3.4.4 Kulka




*Rysunek 27: Klasa kulki*

Klasa odpowiadająca kulce. Dziedziczy po klasie **MovableElement**. Zawiera ona następujące pola:

- float *radius* – długość promienia kulki
- float[] *rotation* – stan obrotu kulki zapisany w postaci macierzy obrotu
- float *mass* – masa kulki
- float *area* – powierzchnia czołowa – pole przekroju kuli prostopadłego do wektora prędkości postępowej

- float *CD* – współczynnik dla oporu powietrza, wartość stała
- float *DENSITY* – gęstość powietrza, wartość stała

#### *3.4.4.1 Odświeżanie stanu kulki*

Do odświeżania stanu kulki służy następująca metoda:

- void *Update*(float dt, Vector accelerometrData, float mu) – aktualizuje położenie kulki, bierze pod uwagę tylko aktualne wskazania akcelerometru oraz fakt, czy kulka znajduje się na płaszczyźnie, rozwiązuje równanie różniczkowe do zmiany stanu (położenia i prędkości) kulki

```
void Update(float dt, Vector accData, float mu){
        aktualizujemy pozycję i prędkość korzystając ze wszystkich danych przekazywanych do funkcji,
        korzystamy z metody równań różniczkowych
        //podczas rozwiązywania korzystamy z tego, że:
        //if(mu>=0)
        //      znajdujemy wartość przyspieszenia związanego z ruchem po powierzchni
        //else
        //      znajdujemy wartość przyspieszenia, gdy kulka nie znajduje się na powierzchni

        następnie obliczamy zmianę położenia kulki i na tej podstawie zmieniamy macierz obrotu kulki
}
```

**Obliczanie nowego położenia i prędkości**

Zakładamy, że kulka w czasie t:

- znajduje się w punkcie **P**=(x,y,z)
- ma prędkość **v**=($v_x$,$v_y$,$v_z$)

Aby określić położenie kulki w następnej chwili czasu należy wyznaczyć siłę wypadkową **F** działającą na nią. Następnie dzieląc uzyskaną siłę przez masę kulki *m*( $a = \frac{F}{m}$ ) otrzymujemy wartość przyspieszenia **a**=($a_x$,$a_y$,$a_z$) działającego na kulkę.

Wówczas, mając dane przyspieszenie działające na kulkę możemy uzyskać wartość prędkości kulki **v'**=($v_x$',$v_y$',$v_z$') i położenia **P'**=(x',y',z') w czasie t+Δt następująco:

- $\boldsymbol{v}' = \boldsymbol{v} + \int_t^{t+\Delta t} \boldsymbol{a}\, dt$
- $\boldsymbol{P}' = \boldsymbol{P} + \int_t^{t+\Delta t} \boldsymbol{v}\, dt$

Powyżej uzyskaliśmy 6 równań różniczkowych. Rozwiązywane one będą numerycznie za pomocą metody Rungego-Kutty rzędu czwartego. Opis tej metody znajduje się w rozdziale 3.4.6.

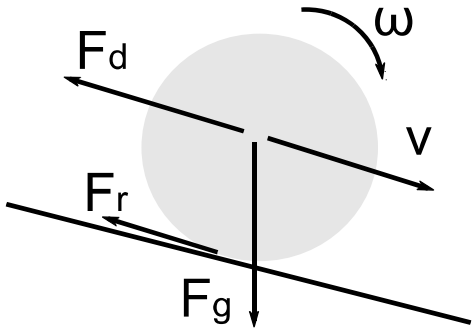
Siła wypadkowa zależy od aktualnego położenia kulki[14]. Rozważamy 2 możliwości:

1.Kulka porusza się po płaszczyźnie bez poślizgu. Wówczas działają następujące siły:

$F_g$ – siła grawitacji,

$F_r$ – siła tarcia,

$F_D$ – siła oporu powietrza.




*Rysunek 28: Schemat ruchu kulki na płaszczyźnie*

Dodatkowo zostały zaznaczone:

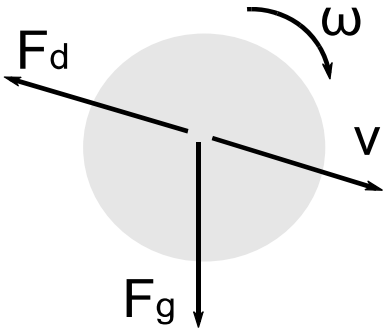
v – prędkość ruchu postępowego kulki,

ω – prędkość ruchu obrotowego kulki.

2.Kulka nie znajduje się na powierzchni. Wówczas działają następujące siły:

$F_g$ – siła grawitacji,

$F_D$ – siła oporu powietrza,




*Rysunek 29: Schemat ruchu kulki znajdującej się w locie*

Dodatkowo zostały zaznaczone:

v – prędkość ruchu postępowego kulki,

ω – prędkość ruchu obrotowego kulki.

Powyższe siły można obliczyć następująco:

- $F_g = mg$ gdzie m- masa kulki, g-przyspieszenie ziemskie
- $F_r = \mu_r mgcos(\varphi)$ gdzie $\mu_r$ - współczynnik tarcia,
  φ-kąt nachylenia płaszczyzny do powierzchni Ziemi
- $F_D = \frac{1}{2} C_D \rho v^2 A$ gdzie ρ - gęstość powietrza, v - prędkość kulki,
  A – powierzchnia czołowa obiektu, zapisana w polu *area* kulki; dla kuli $A=\pi r^2$,
  $C_D$ – współczynnik oporu, dla kulki w ruchu laminarnym $C_D$=0.4-0.47

**Obliczanie stanu obrotu kulki**

W danej chwili czasu t mamy dane:

- Vector *lastMove* – zmiana położenia kulki w ciągu ostatniej klatki
- float[] *rotate* – stan obrotu kulki zapisany w macierzy obrotu

Wówczas nową wartość dla macierzy obrotu możemy wyliczyć następująco:

rzutujemy wektor *lastMove* na płaszczyznę OXZ, zapisujemy wektor jako *difference*
oś obrotu = wektor znajdujący się w płaszczyźnie OXZ prostopadły do wektora *difference*
kąt obrotu = stosunek długości wektora *difference* do obwodu koła wielkiego kulki
tworzymy macierz obrotu z wyliczonych wartości osi i kąta obrotu (metoda *setRorateM* z OpenGL)
mnożymy utworzoną macierz z lewej strony przez macierz *rotate*, wynik zapisujemy w macierzy *rotate*

### *3.4.4.2 Rozwiązywanie kolizji*

Algorytmy związane z kolizjami kulki i odpowiedziami na nie znajdują się w opisie klasy Collisions.

- boolean *CheckCollision*(Floor, Wall, Bar, Elevator, Diamond, HourGlass lub ControlPoint element) – metody sprawdzające, czy elementy kolidują z kulką,

```
boolean CheckCollision(Floor floor){
```

floor jest prostopadłościanem równoległym do osi układu współrzędnych
sprawdzamy czy ten prostopadłościan koliduje z kulką
}

w analogiczny sposób wyglądają metody *CheckCollision* z elementami **Wall**, **Bar** i **Elevator**

boolean CheckCollision(Diamond diamond){
sprawdzamy czy walec opisany na diamencie koliduje z kulą
}

w analogiczny sposób wygląda metoda *CheckCollision* z elementem **HourGlass**

boolean CheckCollision(Finish finish){
finish jest elementem składającym się z 2 ściętych stożków
sprawdzamy czy kulka znajduje się na kole, które jest podstawą dolnego ściętego stożka
}

w analogiczny sposób wygląda metoda *CheckCollision* z elementem **CheckPoint**

- *ReactOnCollision*(Floor, Wall, Bar lub Elevator element) – metody, które zmianiają położenie i prędkość kulki w przypadku kolizji z innym elementem

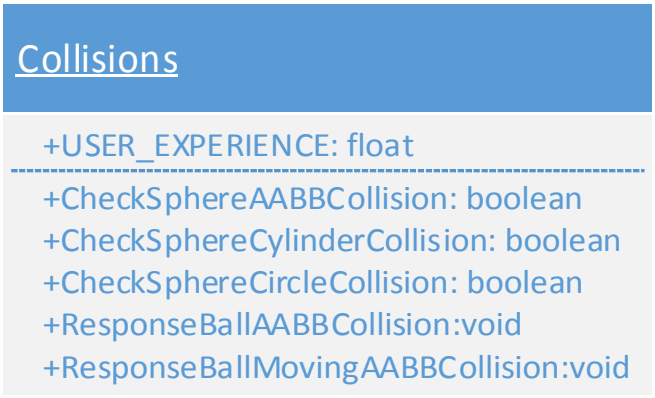
void ReactOnCollision(Floor element){
floor jest nieruchomym prostopadłościanem równoległym do osi układu współrzędnych
zmieniamy stan kulki ze względu na kolizję z tym elementem
}

w analogiczny sposób wygląda metoda *ReactOnCollision* z elementem **Wall**

void ReactOnCollision(Beam beam){
beam jest poruszającym się prostopadłościanem równoległym do osi układu współrzędnych
zmieniamy stan kulki ze względu na kolizję z tym elementem
}

w analogiczny sposób wygląda metoda *ReactOnCollision* z elementem **Elevator**

### 3.4.5 Klasa Collisions

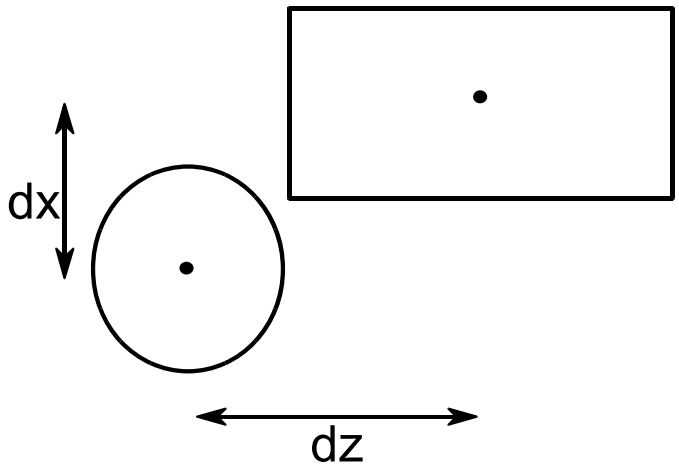



*Rysunek 30: Klasa Collisions*

Klasa statyczna odpowiadające za sprawdzanie i rozwiązywanie problemów związanych z kolizjami kulki z elementami. Posiada pole USER_EXPERIENCE – niewielka liczba, wartość pomocnicza używana przy sprawdzaniu równości lub nierówności liczb przy obliczeniach. Używana jest w związku z niedokładnością obliczeń numerycznych komputera. Zawiera następujące metody:

- boolean *CheckSphereAABBCollision*(Sphere sphere,Box box) – metoda sprawdzająca, czy kula koliduje z prostopadłościanem równoległym do osi układu współrzędnych (AABB) [15]

```
public static boolean CheckSphereAABBCollision(Sphere sphere, Box box) {
        dla każdej ze współrzędnych
                obliczamy po współrzędnej różnicę pomiędzy środkiem kuli a środkiem prostopadłościanu
                (przykład na rys.31) zapisujemy jako d
                dif =max(różnica d i połowy rozmiaru prostopadłościanu względem tej współrzędnej, 0)
        distance =suma kwadratów wartości dif dla każdej ze współrzędnych;
        kolizja występuje jeśli distance<=kwadrat promienia kuli
}
```



*Rysunek 31: Sposób obliczania różnicy po współrzędnej*

- boolean *CheckSphereCylinderCollision*(Sphere sphere,Cylinder cylinder) – metoda sprawdzająca, czy kula koliduje z walcem, którego wysokość jest równoległa do osi Y układu współrzędnych

```
public static boolean CheckSphereCylinderCollision(Sphere sphere, Cylinder cylinder) {
        dla współrzędnej Y
                obliczamy po współrzędnej różnicę pomiędzy środkiem kuli a środkiem walca
                i zapisujemy jako d
                difstance=kwadrat max(różnica d i połowy rozmiaru prostopadłościanu względem współrzędnej y, 0)
        obliczamy kwadrat odległości rzutów środka kuli oraz środka walca na płaszczyznę OXZ, wartość tę dodajemy
        do wartości distance
        kolizja występuje jeżeli distance <=(promień kuli + promień podstawy walca)
}
```

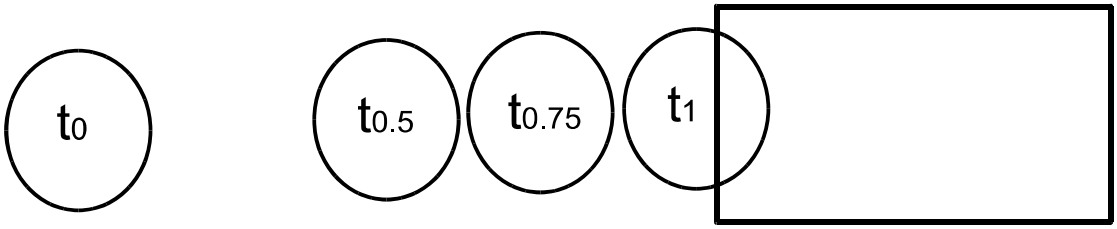
- boolean *CheckSphereCircleCollision*(Sphere sphere, Circle circle) – metoda sprawdzająca, czy kula znajduje się na kole

```
public static boolean CheckSphereCircleCollision(Sphere sphere, Circle circle) {
        sprawdzamy czy odległość rzutów środka kuli i środka koła na płaszczyznę OXZ jest mniejsza niż promień kuli
        oraz czy różnica po współrzędnej y środka kuli od środka koła jest równa promieniowi
}
```

- void *ResponseBallAABBCollisions*(Ball ball, Box box) – metoda rozwiązująca sytuację kolizji piłki ze statycznym prostopadłościanem równoległym do osi układu współrzędnych

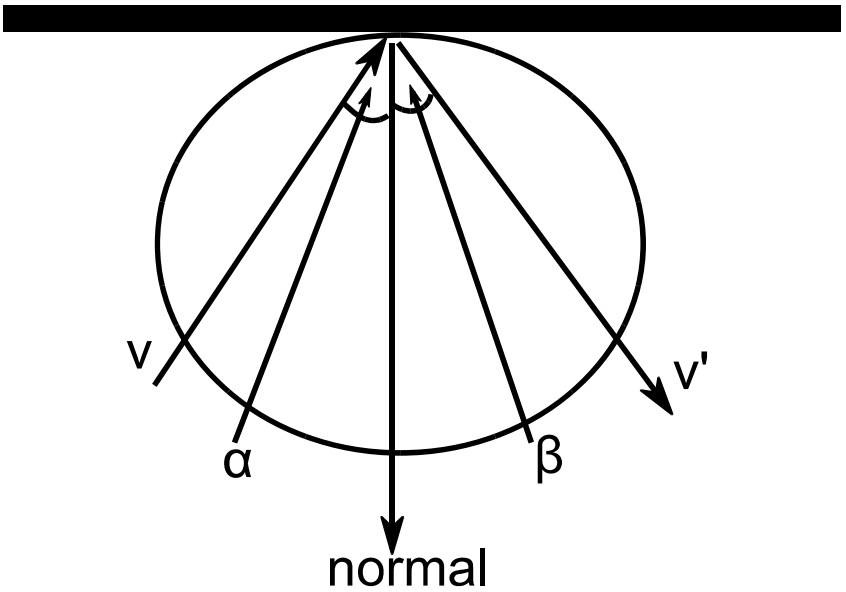
```
public static void ResponseBallAABBCollisions(Ball ball, Box collidedBox) {
```

korzystając z położenia kuli przed kolizją i po kolizji metodą bisekcji szukamy położenia kuli oraz momentu w czasie, w którym nastąpiło zetknięcie[16]
zapisujemy wartość położenia kuli w halfLocation, a czas kolizji w halfTime




*Rysunek 32: Metoda bisekcji poszukiwania momentu kolizji*

znajdujemy wektor normalny do płaszczyzny kolizji i zapisujemy w normal
jeśli wektor normalny nie zawiera się w płaszczyźnie równoległej do płaszczyzny OXY, OXZ lub OYZ, to
ustalamy normal.y=0
korzystając z unormowanej wartości prędkości kuli oraz unormowanego wektora normalnego obliczamy wartość prędkości kuli po kolizji korzystając z faktu, że kąt padania=kąt odbicia




*Rysunek 33:Sposób rozwiązywania kolizji: normal – wektor normalny do płaszczyzny kolizji,*
v – wektor prędkości kulki przed kolizją
v' – wektor prędkości kulki po kolizji

*α=β (kąt padania=kąt odbicia)*

ustalamy nowe położenie kuli symulując ruch dla niej od położenia halfLocation korzystając z nowo wyliczonej wartości prędkości, zapisujemy nową wartość prędkości

jeśli nadal występuje kolizja

znajdujemy wektor normalny do płaszczyzny kolizji i zapisujemy w distance

metodą bisekcji przesuwamy kulę w kierunku distance aby usunąć kolizję z elementem i ustawić kulę jak najbliżej elementu z którym kolidowała

}

- void *ResponseBallMovingAABBCollisions*(Ball ball, MovableElemenet element) – metoda rozwiązująca sytuację kolizji kulki z poruszającym się prostopadłościanem równoległym do osi układu współrzędnych. Jej działanie jest podobne do metody *ResponseBallAABBCollisions*

public static void ResponseBallMovingAABBCollisions(Ball ball, MovableElement element) {

korzystając z położenia kuli i elementu przed kolizją i po kolizji metodą bisekcji szukamy położenia kuli i elementu oraz momentu w czasie, w którym nastąpiło zetknięcie

zapisujemy wartość położenia kuli w halfLocation, położenia elementu w halfMovableElement a czas kolizji w halfTime (patrz rys.32)

znajdujemy wektor normalny do płaszczyzny kolizji i zapisujemy w normal

jeśli wektor normalny nie zawiera się w płaszczyźnie równoległej do płaszczyzny OXY, OXZ lub OYZ, to ustalamy normal.y=0

w przypadku kolizji z windą oraz zachodzenia warunku: normal.y jest różny od 0

metodą bisekcji przesuwamy kulę w kierunku wektora normalnego aby usunąć kolizję z elementem i ustawić kulę jak najbliżej elementu z którym kolidowała

w przeciwnym przypadku

korzystając z unormowanej wartości prędkości kuli oraz unormowanego wektora normalnego

obliczamy wartość prędkości kuli po kolizji korzystając z faktu, że kąt padania=kąt odbicia

(patrz rys. 33)

jeśli następowała kolizja z windą

symulujemy ruch kuli od momentu kolizji zgodnie z ruchem windy

ustalamy nowe położenie kuli symulując ruch dla niej od położenia halfLocation korzystając z nowo wyliczonej wartości prędkości, zapisujemy nową wartość prędkości

jeśli nadal występuje kolizja z belką

znajdujemy wektor normalny do płaszczyzny kolizji i zapisujemy w distance

metodą bisekcji przesuwamy kulę w kierunku distance aby usunąć kolizję z elementem i ustawić kulę jak najbliżej elementu z którym kolidowała

}

### 3.4.6 Rozwiązywanie równań różniczkowych

Równania różniczkowe występują podczas obliczania kolejnych położeń i prędkości kulki w czasie trwania ruchu. W naszej aplikacji rozwiązujemy za pomocą metody Rungego-Kutty rzędu 4[17]. Korzystamy z następującego algorytmu:

Niech $\Delta x = \int_{t_1}^{t_1+\Delta t} f(t,x)\,dt$ - zmiana wartości x w czasie Δt.

Obliczamy następujące wartości:

- $\Delta x_1 = f(t,x)\,\Delta t$
- $\Delta x_2 = f\left(t+\frac{\Delta t}{2}, x+\frac{\Delta t}{2}\Delta x_1\right)\Delta t$
- $\Delta x_3 = f\left(t+\frac{\Delta t}{2}, x+\frac{\Delta t}{2}\Delta x_2\right)\Delta t$
- $\Delta x_4 = f\left(t+\Delta t, x+\Delta t\,\Delta x_3\right)\Delta t$

Wówczas $\Delta x = \frac{\Delta x_1 + 2\Delta x_2 + 2\Delta x_3 + \Delta x_4}{6}$

## *3.5 Część Android*

### 3.5.1 Pliki XML

#### *3.5.1.1 Pliki opisujące plansze*

Do przechowywania danych o strukturze planszy wykorzystywane są pliki xml.

Korzeniem dokumentu jest element o nazwie **Board**, zawierający atrybut id. Jest to element zawierający informację o wszystkich obiektach składowych planszy.

Nazwy elementów potomnych pokrywają się z nazwami klas odpowiadającym im obiektom, tzn.: Bar, CheckPoint, Diamond, Elevator, Finish, Floor, HourGlass, Start, Wall.

Dodatkowo każdy z tych elementów zawiera informację o położeniu na planszy. Są za to odpowiedzialne 3 atrybuty: location_x, location_y oraz location_z.

Elementy Bar oraz Elevator oprócz położenia muszą mieć zdefiniowane 2 punkty, pomiędzy którymi będą się poruszać: point_from_x, point_from_y, point_from_z, point_to_x, point_to_y, point_to_z.
Elementy Floor oraz Wall z kolei muszą posiadać informację o rozmiarze, tzn.: boxsize_x, boxsize_y, boxsize_z.
Element Floor posiada dodatkowo informację o współczynniku tarcia: friction, a element HourGlass o liczbie dodatkowych sekund: value

Przykładowy plik xml:

```
<Board id="board_1">
    <Floor location_x="0.00" location_y="-0.10" location_z="0.00" boxsize_x="0.70" boxsize_y="0.20" boxsize_z="1.60" friction="0.05" />
    <Floor location_x="0.00" location_y="-0.10" location_z="0.00" boxsize_x="1.60" boxsize_y="0.20" boxsize_z="0.70" friction="0.05" />
    <Start location_x="0.00" location_y="0.10" location_z="0.00" />
    <Finish location_x="0.50" location_y="0.10" location_z="0.00" />
</Board>
```

#### *3.5.1.2 Pliki z progami punktowymi*

Do przechowywania informacji o progach punktowych są wykorzystywane odpowiednie pliki xml.
Korzeniem dokumentu jest element o nazwie **Board_Information**.
Elementami potomnymi są elementy o nazwie **Board** zawierające następujące atrybuty: id, two_stars, three_stars, time.
Atrybuty two_stars oraz three_stars przechowują informację o wymaganej ilości punktów potrzebnych do otrzymania odpowiednio 2 oraz 3 gwiazdek. Pierwsza gwiazdka jest przyznawana za samo ukończenie poziomu, więc nie trzeba przechowywać dla niej progu punktowego.
Atrybut time mówi o liczbie sekund, którą gracz ma na przejście danego poziomu.
Wartość atrybutu id powinna być taka sama jak w przypadku pliku opisującego daną planszę, tak aby oba pliki mogły być przypisane do konkretnego poziomu i stanowiły jego pełny opis.

Przykładowy plik xml:

```
<Board_Information>
    <Board id="board_1" two_stars="250" three_stars="350" time="100" />
    <Board id="board_2" two_stars="300" three_stars="500" time="100"  />
```

```
    <Board id="board_3" two_stars="500" three_stars="750" time="150"  />
</Board_Information>
```

### 3.5.2 Kontrolki

Interfejs użytkownika korzysta ze stworzonych przez nas kontrolek. Rozszerzają one podstawowe kontrolki o pewne funkcjonalności oraz korzystają z jednakowej czcionki [18].

- **FontTextView** - klasa dziedzicząca po TextView, pozwalająca na proste ustawienie czcionki
- **...TextView** - grupa klas, które rozszerzają TextView o animację [19] i gradient
- **...ImageView** - grupa klas, które rozszerzają ImageView o animację
- **OptionCheckBox** - rozszerza CheckBox o animację

### 3.5.3 Aktywności

Z każdym głównym oknem aplikacji powiązana jest klasa aktywności. Oprócz wyświetlania zawartości okna mają one także inne zadania.
Aktywność odpowiedzialna za wyświetlanie menu głównego ma jeszcze jedną ważną funkcję. Za pomocą mechanizmu SharedPreferences [20] wczytuje ona ustawienia zapisane wcześniej przez użytkownika lub jeśli aplikacja została uruchomiona po raz pierwszy to uzupełnia je poprzez przypisanie domyślnych wartości. Użytkownik może zmienić te ustawienia przechodząc do okna opcji. Aktywność powiązana z oknem opcji także korzysta z mechanizmu SharedPreferences, aby wczytać ustawienia wybrane przez użytkownika lub wprowadzić zapisane przez niego zmiany. Dodatkowo przy zmianie wyboru języka musi ona przeładować zawartość okna, aby dostosować ją do wyboru.
W oknie wyboru poziomu użytkownik może sprawdzić najlepsze uzyskane dotychczas wyniki na kolejnych planszach. Aktywność pobiera te informacje w taki sam sposób jak przy ustawieniach – za pomocą mechanizmu SharedPreferences.
Aktywność odpowiadająca za ekran gry jest wywoływana przez aktywność wyboru poziomu za pomocą metody "startActivityForResult". Dzięki temu przy zakończeniu aktywności odpowiedzialnej za grę, na poziomie aktywności wyboru planszy możemy odczytać rezultat jakim zakończyła się rozgrywka i odpowiednio zareagować. W przypadku przechodzenia do okna opcji wysyłana jest informacja o tym, że należy pokazać ograniczony wybór ustawień (jedynie muzyka, dźwięki oraz wibracje).

Podczas gry użytkownik steruje kulką poprzez ruch urządzenia, a więc nie musi dotykać ekranu. Może

to spowodować automatyczne zablokowanie ekranu urządzenia, przed czym należało się uchronić. Jest to rozwiązane za pomocą mechanizmu PowerManager.WakeLock [21] uruchamianego podczas tworzenia aktywności gry. Przy zakończeniu rozgrywki następuje zwolnienie zasobów związanych z tym mechanizmem.

Dodatkowo aktywność odpowiadająca za ekran gry musi kontrolować jaką domyślną orientację ma urządzenie, na którym uruchomiona jest aplikacja. Jest to o tyle ważne, że w przypadku urządzeń o domyślnej orientacji poziomej inaczej ustawione są osie współrzędnych (co ma wpływ na sterowanie kulką z użyciem akcelerometru).

## *3.6 Elementy towarzyszące aplikacji*

### 3.6.1 Dokumentacja kodu

Razem z aplikacją została wykonana dokumentacja kodu. Znajduje się ona na płycie dołączonej do pracy inżynierskiej w folderze *Dokumentacja kodu.* Wyświetlenie jej w przeglądarce internetowej jest możliwe poprzez otworzenie pliku *Index.html*.

### 3.6.2 Testy jednostkowe

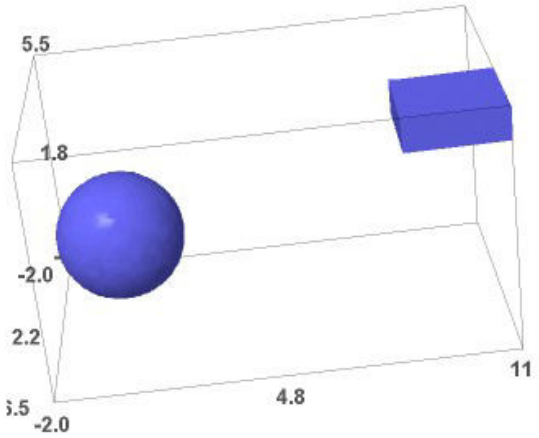
Dla sprawdzenia poprawności działania aplikacji zostały zaprojektowane i przeprowadzone testy jednostkowe. Znajdują się one na płycie w folderze z kodem aplikacji jako osobny projekt. Poniżej został zaprezentowany jeden z zaimplementowanych scenariuszy testowych wraz z kilkoma przypadkami testowymi.

**Scenariusz testowy**

Celem tego testu jest sprawdzenie poprawności metody, która stwierdza, czy kula i prostopadłościanem równoległy do osi układu współrzędnych przecinają się. W tym celu definiujemy położenie środka kuli i jej promień, a także położenie środka prostopadłościanu oraz jego rozmiary. Metoda zwraca wartość *true,* jeśli kolizja występuje, w przeciwnym wypadku wartość *false.* Przytoczymy następujące przypadki testowe:

1. Sfera: położenie środka - (0, 0, 0), długość promienia kuli - 2

   Prostopadłościan: położenie środka - (5, 10, 5), wymiary prostopadłościanu - (3, 3, 1)

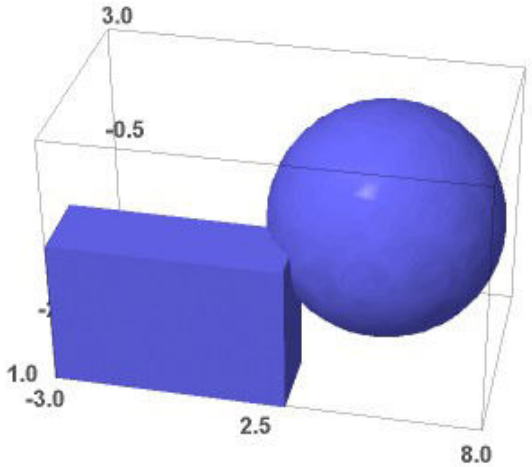



*Rysunek 34: Wizualizacja przypadku 1. Z użyciem[22]*

Obiekty nie przecinają się, metoda zwróciła wartość *false*

2. Sfera: położenie środka -(-2, 5, 0), długość promienia kuli - 3

   Prostopadłościan: położenie środka - (0, 0, -2), wymiary prostopadłościanu - (2, 6, 4)

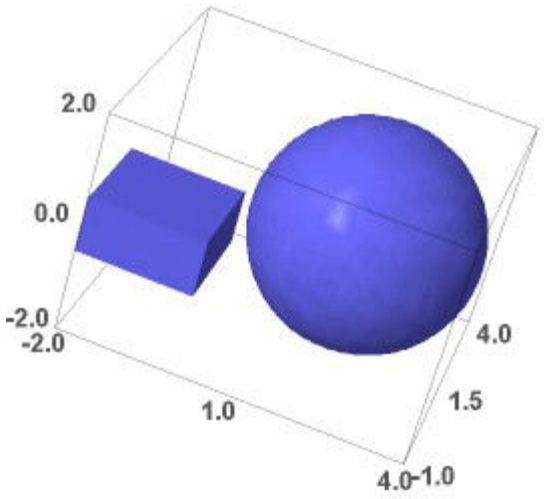



*Rysunek 35: Wizualizacja przypadku 2. Z użyciem[22]*

Obiekty przecinają się, metoda zwróciła wartość *true.*

3. Sfera: położenie środka - (2, 2, 0), długość promienia kuli - 2

Prostopadłościan: położenie środka - (-1, 0, 0), wymiary prostopadłościanu -(2, 2, 1)




*Rysunek 36: Wizualizacja przypadku 3. Z użyciem[22]*

Obiekty nie przecinają się, metoda zwróciła wartość *false.*

# 4 . Podsumowanie

Projekt inżynierski opisany w tej pracy jest połączeniem kilku nowoczesnych technologii. Aplikacja mobilna z interesującą treścią jest dobrą reklamą współczesnej informatyki. Zastosowanie technologii OpenGL ES 2.0 umożliwiło uzyskanie płynności animacji niezbędnej w dynamicznych grach zręcznościowych. Naszym celem było zaprojektowanie ładnego i przejrzystego interfejsu użytkownika oraz zaimplementowanie intuicyjnego sterowania, co pozwoliłoby przyciągnąć szersze grono użytkowników. System wczytywania plansz, jaki zastosowaliśmy w aplikacji, umożliwia łatwe dodawanie kolejnych. Ponadto, aplikacja może zostać uruchomiona na większości urządzeń z systemem Android, zarówno na telefonach jak i tabletach.

W przyszłości projekt może zostać jeszcze rozbudowany. Możliwe, że zostanie umieszczony w Sklepie Play, aby był dostępny dla większej grupy użytkowników. Planujemy wtedy zaprojektować dodatkowe plansze, tak aby gra nie kończyła się zbyt szybko. Możliwe byłoby też stworzenie nowych elementów planszy, które urozmaiciłyby rozgrywkę. Dodatkowo chcielibyśmy dodać ranking najlepszych wyników (aktualizowany za pomocą Web Service), żeby wprowadzić element rywalizacji pomiędzy graczami.

Zakres projektu umożliwiał wygodny podział pracy pomiędzy autorów. Mogliśmy pracować równolegle nad osobnymi fragmentami nie martwiąc się o problemy z integracją. Istniały klasy, w których przenikały się nasze odpowiedzialności, jednakże, dla ułatwienia współpracy, podzieliliśmy je. Ostatecznie, moduły współpracują ze sobą bez większych kłopotów. Pracę wspólną umożliwił nam system wersjonowania plików – GIT. Dzięki temu projektowi udało nam się bliżej zapoznać i nauczyć współpracy z taką aplikacją.

# 5 . Bibliografia


[1] http://pichost.me/1430204/

[2] http://bloodbrothersgame.wikia.com/wiki/File:Star.png

[3] http://icons.iconarchive.com/icons/hopstarter/soft-scraps/96/Button-Pause-icon.png

[4] http://www.spiralgraphics.biz/packs/browse.htm

[5] http://www.flashkit.com/loops/

[6] http://elektronikab2b.pl/technika/12098-zyroskopy-i-akcelerometry-mems-w-elektronice-uzytkowej

[7] http://developer.android.com/guide/topics/sensors/sensors_motion.html#sensors-motion-accel

[8] http://www.badlogicgames.com/wordpress/wp-content/uploads/2011/06/orientation.png

[9] Kevin Brothaler, OpenGL ES 2 for Android, The Pragmatic Programmers, 2013

[10] http://www.codeproject.com/Articles/822380/Shadow-Mapping-with-Android-OpenGL-ES

[11] http://fabiensanglard.net/shadowmapping/index.php

[12] https://msdn.microsoft.com/en-us/library/windows/desktop/ee416324(v=vs.85).aspx

[13] http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/

[14] Grant Palmer,Physics for GameProgrammer, Apress, 2005, str.83-116

[15] http://stackoverflow.com/questions/15247347/collision-detection-between-a-boundingbox-and-a-sphere-in-libgdx

[16] http://www.emunix.emich.edu/~evett/GameProgramming/LectureNotes/4.2%20Collision%20Detection.pdf

[17] Grant Palmer,Physics for GameProgrammer, Apress, 2005, str.55-67//

[18] http://www.fontsquirrel.com/fonts/sonsie-one

[19] http://stackoverflow.com/questions/8978036/android-translateanimation-animation

[20] http://www.tutorialspoint.com/android/android_shared_preferences.htm

[21] http://developer.android.com/reference/android/os/PowerManager.html

[22] http://www.sagenb.org/, program Jmol