WERSJA	DATA	ZMIANY
0.1	20.12.2022	*Pierwsza publikacja*
1.0	13.01.2023	*Uzupełnienie o diagramy UML*
1.1	16.01.2023	*Aktualizacja tabeli bugów*
1.2	17.01.2023	*Drobne poprawki*

SYMULATOR LECĄCYCH CZĄSTEK

Autor: Piotr Wilkoń

Akademia Górniczo-Hutnicza

Spis treści

Lista oznaczeń

API	Application Programming Interface
COR	Coefficient of Restitution
GLU	OpenGL Utility
GLUT	OpenGL Utility Toolkit
OOP	Object-Oriented Programming
STL	Standard Template Library

1. Wstęp

Dokument dotyczy opracowania symulatora lecących cząstek w przestrzeni trójwymiarowej. Podstawowym zadaniem tego oprogramowania jest symulacja zachodzących zjawisk fizycznych (przede wszystkim mechanicznych) i zobrazowanie przestrzeni przy pomocy interfejsu graficznego. Założeniem projektu jest stworzenie maksymalnie niezależnych od siebie podsystemów, aby każdy mógł być używany oddzielnie, a także aby ułatwić dodawanie nowych funkcjonalności i poprawianie błędów.

1.1 Założenia projektu

Podstawowymi założeniami projektu są:

1. Przygotowanie abstrakcyjnego opisu i materiałów potrzebnych do zaimplementowania algorytmów.
2. Określenie wymagań i opracowanie architektury podsystemu graficznego.
3. Określenie wymagań i opracowanie architektury podsystemu silnika fizyki.
4. Określenie sposobu komunikacji pomiędzy podsystemem graficznym i silnika fizyki przy zachowaniu maksymalnej abstrakcji warstw.
5. Określenie wymagań interfejsu użytkownika.
6. Implementacja podsystemu graficznego i jego API.
7. Test podsystemu graficznego dla obiektów statycznych.
8. Implementacja podsystemu silnika fizyki i jego API.
9. Test podsystemu silnika fizyki.
10. Implementacja interfejsu do komunikacji między podsystemami.
11. Implementacja interfejsu użytkownika.
12. Testy końcowe.

2. Funkcjonalność

1. Funkcjonalności interfejsu graficznego
    - Obrazowanie przestrzeni trójwymiarowej w oknie o określonych wymiarach
    - Ustawianie kamery na podstawie współrzędnych sferycznych lub kartezjańskich z określonym polem widzenia
    - Generacja obiektów graficznych (płaszczyzn i sfer) o dowolnych wymiarach i kolorach
    - Ustawianie położenia obiektów i ich obrotu
2. Funkcjonalności silnika fizyki
    - Symulacja zachowania ciał fizycznych w przestrzeni o określonych parametrach: przyspieszeniu grawitacyjnym, gęstości, współczynniku skalowania czasu i kroku czasowym symulacji
    - Generacja ciał fizycznych (punktów materialnych) o dowolnej masie, położeniu początkowym, prędkości początkowej, współczynniku odbicia (COR) i promieniu (w celu zmapowania ich do obiektów graficznych)
    - Obsługa zderzeń (sprężystych i niesprężystych – w zależności od COR) między ciałami
3. Funkcjonalności warstwy łączącej podsystemy
    - Zestawienie ciała fizycznego i jego graficznej reprezentacji w postaci jednego obiektu
    - Przekazywanie danych o chwilowym położeniu i innych parametrach z silnika fizyki do podsystemu graficznego
4. Funkcjonalności interfejsu użytkownika
    - Uruchomienie programu z argumentami umożliwiającymi ustawienie:
        - Wymiarów okna
        - Skalowania czasu
        - Liczby losowo wygenerowanych obiektów
        - Automatycznego startu symulacji
        - Użycia lub nie podsystemu graficznego
    - Generacja zadanej liczby losowych obiektów
    - Tworzenie obiektów w trakcie działania programu za pomocą odpowiedniej komendy
    - Zatrzymywanie i wznawianie symulacji za pomocą klawiszy
    - Sterowanie kamerą za pomocą klawiszy

3. Analiza problemów

3.1 Podsystem graficzny

3.1.1 Generacja kształtów, tablica wierzchołków i indeksów

Biblioteki do obsługi grafiki komputerowej, takie jak OpenGL, nie udostępniają metod do tworzenia większości kształtów, nawet dwuwymiarowych. Podstawowym kształtem (z ang. *primitive)* jest trójkąt, z którego buduje się bardziej złożone struktury. Trójkąt taki opisywany jest przede wszystkim za pomocą trzech zestawów współrzędnych, opisujących jego wierzchołki. Najprostszym do zbudowania kształtem (innym niż trójkąt) jest czworokąt, który składa się z dwóch trójkątów.

Niech wierzchołki takiego czworokąta, w przestrzeni dwuwymiarowej, dane będą jako $A(x_A, y_A)$, $B(x_B, y_B)$, $C(x_C, y_C)$ i $D(x_D, y_D)$. Wówczas można utworzyć dwa trójkąty o wierzchołkach, na przykład, $A(x_A, y_A)$, $B(x_B, y_B)$ i $D(x_D, y_D)$ oraz $B(x_B, y_B)$, $C(x_C, y_C)$, $D(x_D, y_D)$. Tak zdefiniowane wierzchołki umieszcza się w tablicy zwanej *tablicą wierzchołków* i przekazuje do odpowiedniej funkcji biblioteki graficznej. Narysowanie takich trójkątów skutkuje wyświetleniem czworokąta.

W analogiczny sposób można tworzyć dowolne kształty, gdzie np. koło można przybliżyć jako identyczne trójkąty równoramienne przyległe do siebie dłuższymi bokami. Podobnie tworzy się kształty trójwymiarowe, przy czym wówczas należy uwzględnić trzecią współrzędną każdego punktu.

Przy bardzo dużej ilości wierzchołków problemem może zacząć być ilość zajętej pamięci. Na przykładzie czworokąta można zauważyć, że trójkąty posiadają jeden wspólny bok, tj. dwa wspólne wierzchołki, B i D. Tworząc trójkąty oddzielnie, wierzchołki te są powielone. Do rozwiązania tego problemu używa się *tablicy indeksów*, który wskazuje na indeks wierzchołka w tablicy wierzchołków, który będzie użyty jako kolejny wierzchołek do zbudowania kształtu. Budowę tablicy wierzchołków i indeksów dla przykładu z czworokątem przedstawiono w **Tabeli 1-1.**

Indeksy tablic:	0	1	2	3	4	5
Tablica wierzchołków:	A	B	C	D	*pusty*	*pusty*
Tablica indeksów wierzchołków:	0 *(A)*	1 *(B)*	3 *(D)*	1 *(B)*	2 *(C)*	3 *(D)*

Tabela 1-1. Tablice wierzchołków i indeksów dla czworokąta

O ile dla tak prostego kształtu zastosowanie indeksów może nie mieć sensu, to przy bardziej złożonych kształtach, na przykład tam, gdzie z każdej strony danego trójkąta przylega do niego inny trójkąt, pozwala to oszczędzić znaczne ilości pamięci.

Dobrym przykładem dla pokazania oszczędności pamięci przy zastosowaniu indeksowania wierzchołków jest koło zbudowane z N trójkątów równoramiennych. Takie koło ma wówczas N wierzchołków na obwodzie i jeden wierzchołek w środku, do którego zbiegają się wszystkie trójkąty. Przyjmujemy, że każdy wierzchołek i każdy indeks zapisuje się w zmiennej o wielkości 32 bitów (4

bajty). Gdyby nie została użyta tablica indeksów, to potrzebne jest $3N$ wierzchołków (i każdy wierzchołek posiada 2 składowe), a zatem zajęta pamięć (w bajtach) to:

$$C_0 = 4 * 2 * 3N = 24N$$

Gdyby wykorzystać tablicę indeksów, to wystarczy, jak wspomniano wcześniej, $N + 1$ wierzchołków, a także $3N$ indeksów (każdy indeks opisuje jeden wierzchołek trójkąta). Wówczas zajęta pamięć (w bajtach) to:

$$C = 4 * [2 * (N + 1) + 3N] = 20N + 8$$

Łatwo zauważyć, że oszczędność pamięci pojawia się już dla $N \geq 3$, a zatem jeszcze przy figurach nieprzypominających koła.

3.1.2 Generacja sfery

W przestrzeni trójwymiarowej jednym z podstawowych kształtów jest sfera, która, podobnie jak każdy inny kształt, musi być zbudowana z płaskich trójkątów.

Budowę sfery (a w zasadzie jej przybliżenia) najprościej rozpocząć od stworzenia wierzchołków na jej powierzchni[1]. Do określenia współrzędnych punktów używa się współrzędnych sferycznych[2]:

$$x = r \cos\theta \cos\varphi$$

$$y = r \cos\theta \sin\varphi$$

$$z = r \sin\theta$$

Gdzie $\varphi \in [0, 2\pi]$ – długość azymutalna, $\theta \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ – odległość zenitalna, r – stały promień sfery

Współrzędne są liczone dla dyskretnych punktów (ich skończonej ilości) przez ustawienie odpowiedniego kroku zmiany kątów. Tak wygenerowane wierzchołki są umieszczane w tablicy wierzchołków.

Konieczne jest zmapowanie wierzchołków tak, aby tworzyły trójkąty. W tym celu brane są takie cztery wierzchołki, które tworzą czworokąt, tj. dwie pary wierzchołków – o takiej samej długości azymutalnej i o takiej samej odległości zenitalnej, a następnie tworzy się z nich dwa trójkąty. Indeksy wierzchołków tworzące trójkąty wpisuje się do tablicy indeksów.

3.1.3 Przemieszczanie kamery

W przestrzeni trójwymiarowej ustawienie kamery opisują trzy wektory:

1. położenia
2. obserwowanego punktu
3. obrotu (wskazujący „górę”), ang. *up vector*, zazwyczaj $(0, 1, 0)$

W toku projektowania przyjęto, że stale obserwowany będzie środek układu współrzędnych (punkt $(0, 0, 0)$), natomiast możliwa będzie zmiana położenia kamery. Do ustalenia współrzędnych położenia wykorzystano układ współrzędnych sferycznych, który przedstawiono w poprzednim punkcie (3.1.2), przy czym promień (odległość od obserwowanego punktu) może być regulowana.

---

[1] http://www.songho.ca/opengl/gl_sphere.html

[2] Spherical coordinates. Encyclopedia of Mathematics: http://encyclopediaofmath.org/index.php?title=Spherical_coordinates&oldid=48774

3.2 Silnik fizyki

3.2.1 Ruch ciał

Jednym z podstawowych pojęć w mechanice jest pojęcie siły, które związane jest z przyspieszeniem ciała. Związek ten opisuje wzór:

$$\vec{F} = m\vec{a}$$

gdzie m jest masą ciała, a $\vec{a}$ – wektorem przyspieszenia ciała.

Wiadomo również, że prędkość dana jest ogólnym wzorem $\vec{v} = \int \vec{a}\, dt$. Dla skończonych przyrostów czasu Δt można wykorzystać operator sumy: $\vec{v} = \sum \vec{a}\Delta t$, zatem każdy składowy przyrost prędkości opisuje wzór:

$$\Delta\vec{v} = \vec{a}\Delta t = \frac{\vec{F}}{m}\Delta t$$

3.2.2 Siły środowiskowe

Na etapie projektowania silnika przyjęto uwzględnienie trzech sił, których wartości opisywane są przez parametry środowiska (otoczenia).

Podstawową siłą jest siła grawitacji, którą w przybliżeniu można opisać za pomocą przyspieszenia grawitacyjnego, zależnego od masy i promienia planety:

$$|\overrightarrow{a_g}| = g$$

Wektor tej siły, a zatem i wektor przyspieszenia, jest skierowany w dół (przeciwnie do osi *y* w układzie współrzędnych używanych przez silnik grafiki).

Kolejną siłą jest siła wyporu, której odpowiada przyspieszenie dane wzorem[3]:

$$|\overrightarrow{a_w}| = \frac{\rho g V}{m}$$

gdzie ρ jest gęstością ośrodka, g – przyspieszeniem grawitacyjnym, V – objętością ciała, a m – jego masą.

Wektor tego przyspieszenia jest skierowany przeciwnie do wektora przyspieszenia grawitacyjnego.

Ostatnią siłą jest siła oporu ruchu, której odpowiada przyspieszenie dane wzorem (w przybliżeniu)[4]:

$$|\overrightarrow{a_o}| = \frac{C_D \rho |\vec{v}|^2 A}{2m}$$

gdzie C_D jest współczynnikiem oporu dla danego ciała, $\vec{v}$ – wektorem prędkości ciała, A – powierzchnią przekroju ciała, ρ – gęstością ośrodka, a m – masą ciała.

Wektor tego przyspieszenia jest skierowany przeciwnie do wektora prędkości ciała.

Wektor zmiany prędkości $\overrightarrow{\Delta v}$ dla każdego przyspieszenia obliczany jest ze wzoru przedstawionego w punkcie 3.2.1. Wektory te są sumowane i dodawane do aktualnej prędkości ciała.

Uwzględnienie wyżej wymienionych sił pozwala na symulację zachowania ciała nie tylko w powietrzu, ale także np. w cieczach, gdzie opory ruchu i wyporność są znaczące.

---

[3] https://wiki.ubc.ca/Buoyancy,_Pressure,_Bernoulli%27s_Equation

[4] https://www.grc.nasa.gov/www/k-12/rocket/drageq.html

3.2.3 Zderzenia

W projekcie silnika fizyki uwzględniono implementację mechanizmu wykrywania i obsługi zderzeń. Dla realizacji zderzeń niesprężystych przyjęto istnienie współczynnika COR – stosunku prędkości po odbiciu do prędkości przed odbiciem.

3.2.3.1 Wykrywanie zderzeń

Do wykrywania zderzeń powszechnie wykorzystuje się bryły ograniczające, będące najmniejszymi bryłami, w które można wpisać dane ciało w całości[5].

W projekcie silnika fizyki jako bryły ograniczające przyjęto sfery, których dowolną ilość można wykorzystać do przybliżenia najmniejszej bryły otaczającej ciało. Każda sfera opisana jest przez wektor położenia jej środka względem środka ciała i promień. Jeśli wektory globalnych współrzędnych sfer wchodzących w skład dwóch różnych ciał oznaczymy przez $\overrightarrow{x_A}$ i $\overrightarrow{x_B}$, a promienie tych sfer przez R_A i R_B, to ustalenie, czy doszło do zderzenia, sprowadza się do sprawdzenia prawdziwości wzoru:

$$|\overrightarrow{x_A} - \overrightarrow{x_B}| \leq R_A + R_B$$

Test taki przeprowadzany jest dla każdej pary sfer. Jeśli wzór ten jest spełniony w chociaż jednym przypadku, to doszło do zderzenia.

3.2.3.2 Zderzenie ciał

Na etapie projektowania silnika przyjęto zastosowanie ciał o jednorodnej gęstości masy, które nie podlegają obrotom, zatem są to punkty materialne o niezerowych wymiarach.

Wektory zmian prędkości każdego obiektu po zderzeniu można obliczyć używają odpowiednich wzorów.[6] Początkowo obliczana jest wypadkowa prędkość zderzenia:

$$v_w = \vec{n} \circ (\overrightarrow{v_1} - \overrightarrow{v_2})$$

gdzie $\vec{n}$ jest jednostkowym wektorem łączącym środki ciał, v_k są prędkościami ciał, a operator $\circ$ oznacza iloczyn skalarny.

Następnie obliczany jest popęd siły działającej podczas zderzenia:

$$J = (1 + \epsilon_1\epsilon_2)\frac{m_1 m_2}{m_1 + m_2} v_w$$

gdzie ϵ_k są współczynnikami COR ciał, m_k – masami ciał, a v_w – wcześniej obliczoną prędkością.

Uwaga: W toku projektowania przyjęto dla uproszczenia, że każde ciało posiada swój współczynnik COR (opisujący zmianę prędkości przy odbiciu od sztywnego i stacjonarnego ciała), a wypadkowy współczynnik COR dla dwóch ciał jest równy ich iloczynowi. Dzięki temu, jeśli jedno z ciał jest całkowicie niesprężyste, to nie dojdzie do żadnego odbicia.

Ostatecznie wektory zmian prędkości są wyliczane jako:

$$\Delta\overrightarrow{v_1} = -\frac{J}{m_1}\vec{n}$$

$$\Delta\overrightarrow{v_2} = \frac{J}{m_2}\vec{n}$$

i dodawane do aktualnego wektora prędkości odpowiedniego ciała.

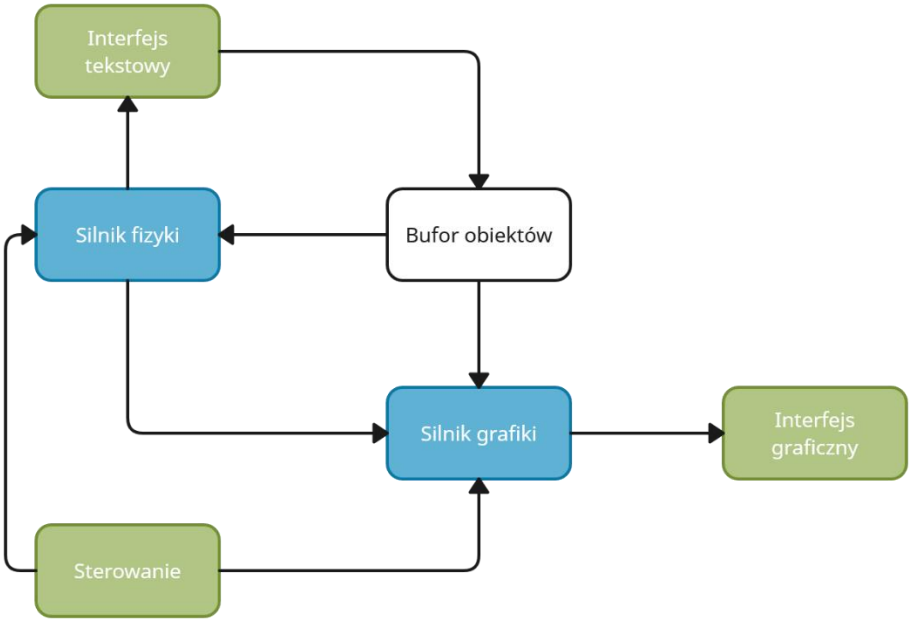
---

[5] https://en.wikipedia.org/wiki/Collision_detection

[6] https://physics.stackexchange.com/questions/681396/elastic-collision-3d-eqaution

4. Projekt techniczny

Budowę systemu w sposób najbardziej ogólny i abstrakcyjny przedstawiono za pomocą schematu blokowego na rysunku 4-1.



Rysunek 4-1. Budowa systemu

4.1 Silnik fizyki

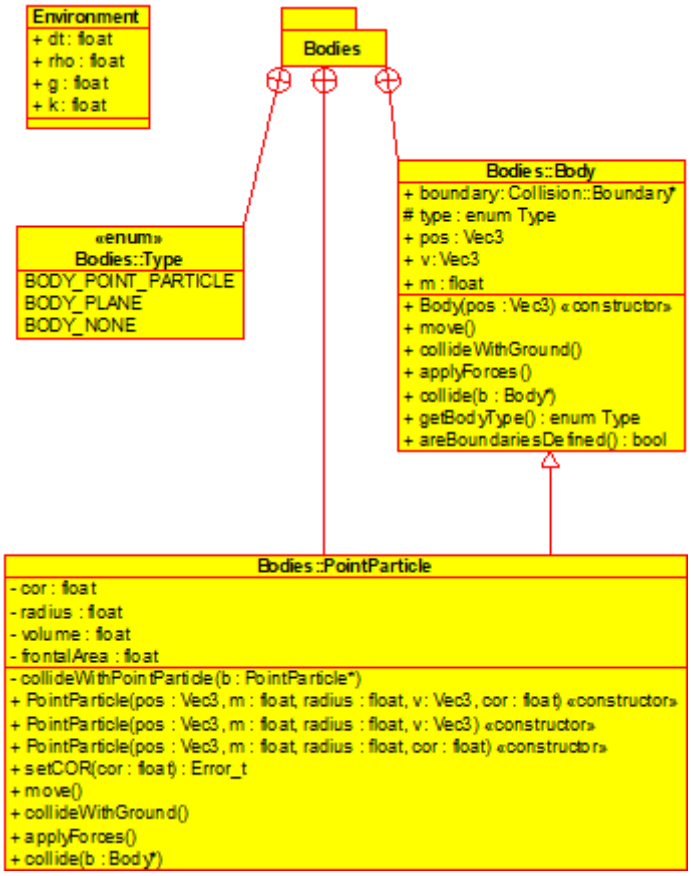
Silnik fizyki odpowiada za obsługę wszelkich zdarzeń fizycznych. Cały podsystem podzielony jest na dwa zasadnicze bloki: obsługę zachowań ciała i obsługę brył ograniczających.

4.1.1 Obsługa zachowań ciała (przestrzeń nazw *Bodies*)

Klasy w tej przestrzeni odpowiadają za tworzenie ciał fizycznych, ustawianie ich parametrów oraz obsługę:

- sił
  - ciążenia
  - wyporu
  - oporu
- ruchu (na podstawie chwilowej prędkości)
- zderzeń

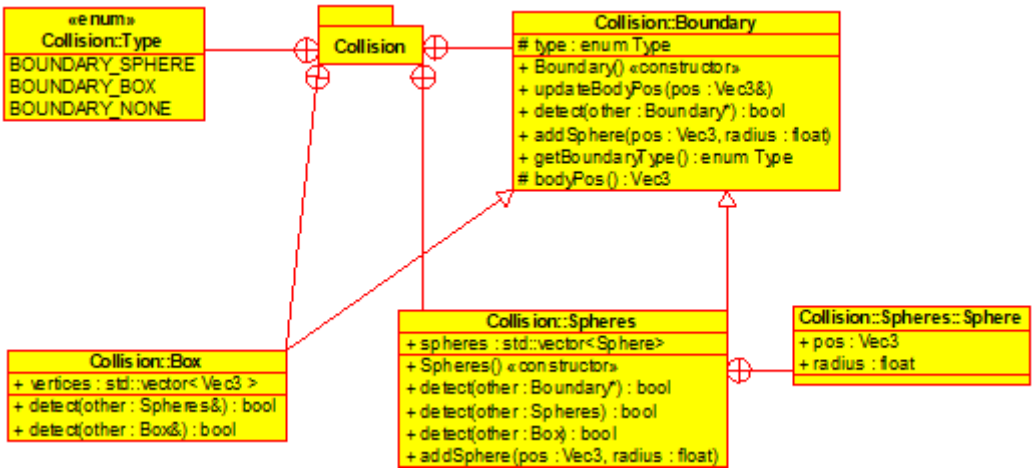
Diagram klas w tej przestrzeni przedstawiono na rysunku 4-2.



Rysunek 4-2. Diagram klas w przestrzeni *Bodies*. Dodatkowo pokazano strukturę *Environment* przechowującą ustawienia środowiska

4.1.2 Obsługa brył ograniczających (przestrzeń nazw *Collision*)

Klasy w tej przestrzeni odpowiadają za tworzenie brył ograniczających oraz wykrywanie zderzenia obiektów (nachodzenia brył ograniczających na siebie). Diagram klas w tej przestrzeni przedstawiono na rysunku 4-3.



Rysunek 4-3. Diagram klas w przestrzeni *Collision*

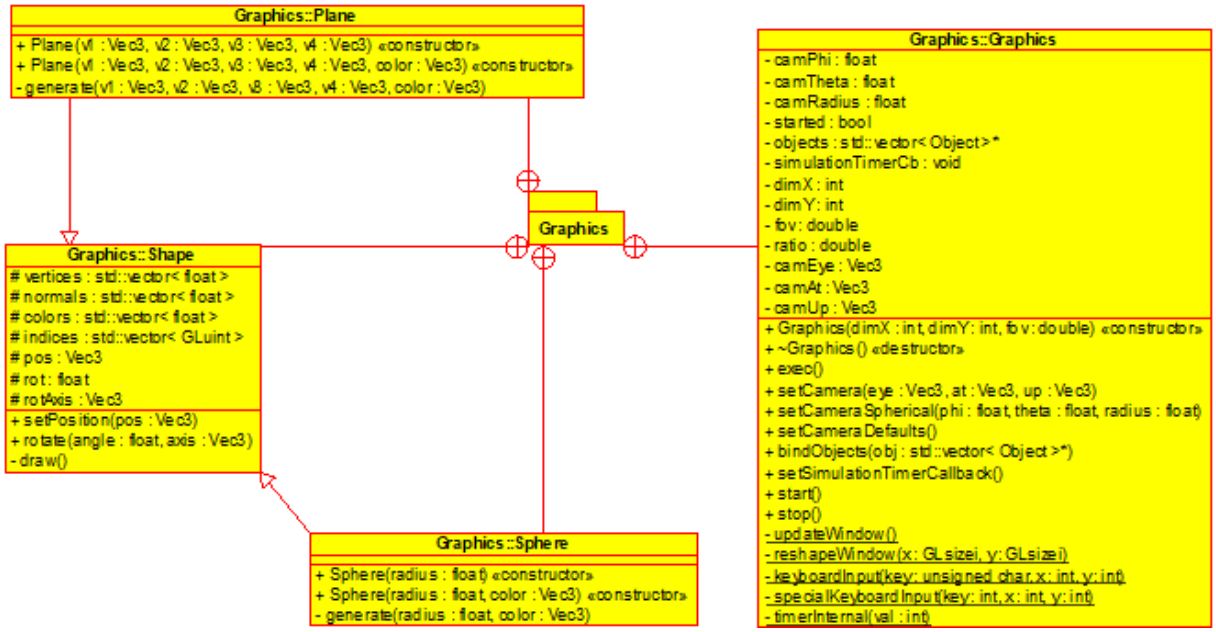
Klasa *Collision::Box* jest niezaimplementowana.

4.2 Silnik grafiki (przestrzeń nazw *Graphics*)

Silnik grafiki odpowiada za:

- tworzenie obiektów (kształtów) graficznych
- rysowanie (renderowanie) obiektów
- umiejscowienie obiektów w przestrzeni
- obrót obiektów
- ustawianie kamery
- inicjalizację środowiska, stworzenie okna, ustawienie światła

Ponadto silnik graficzny obsługuje sterowanie poprzez klawiaturę. Diagram klas w tej przestrzeni przedstawiono na rysunku 4-4.



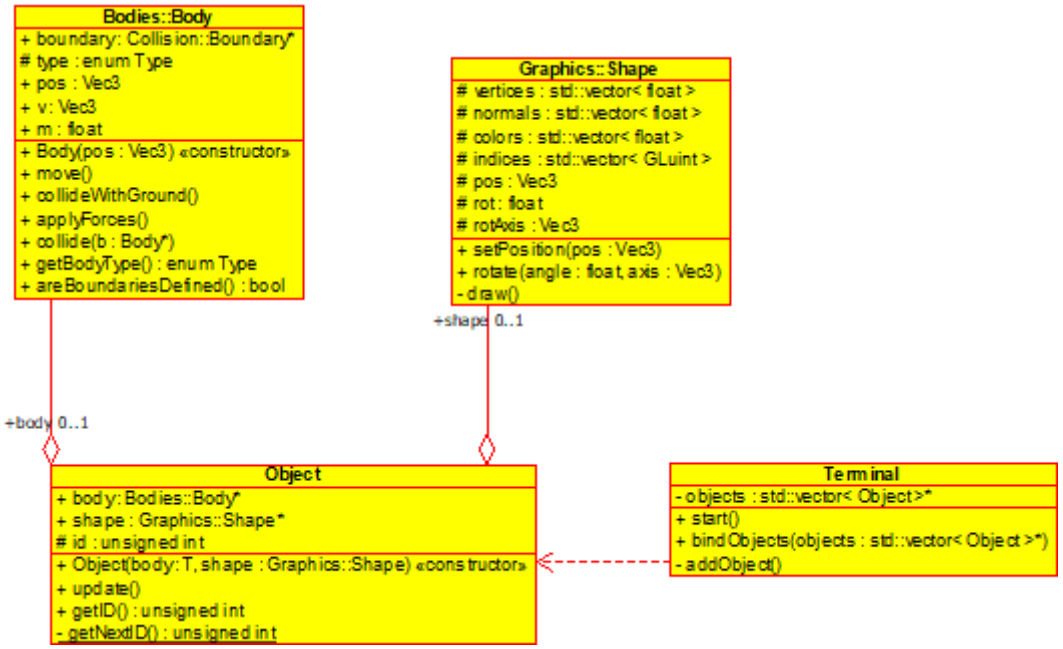
Rysunek 4-4. Diagram klas w przestrzeni *Graphics*

4.3 Magazyn obiektów i obsługa terminala (klasy *Object* i *Terminal*)

Jako interfejs umożliwiający zestawienie ciała fizycznego z graficznym kształtem wykorzystana jest klasa *Object*. Klasa ta przechowuje obiekt ciała (*Body*) i kształtu (*Shape*). Obiekty tej klasy przechowywane są w odpowiednim wektorze, zwanym dalej magazynem obiektów. W toku symulacji każdy obiekt z magazynu obiektów jest przetwarzany, tj. podlega symulacji fizycznej i przedstawieniu graficznemu.

Klasa terminala umożliwia dodawanie nowych obiektów o zadanych parametrach do magazynu obiektów.

Diagram tych klas przedstawiono na rysunku 4-5.



Rysunek 4-5. Diagram klas *Object* i *Terminal*

4.4 Sposób opisu kodu źródłowego

Jako sposobu opisu kodu źródłowego, przede wszystkim klas i ich metod, użyto standardu Doxygen. Ponadto sam kod opatrzony jest dużą ilością komentarzy wewnątrz funkcji, których celem jest wyjaśnienie mechanizmu ich działania.

Zgodnie ze standardem Doxygen, metody opisywane są w następujący sposób:

```
/**
* @brief Ogólny opis funkcji
* @param a Opis parametru a
* @param b Opis parametru b
* @return Opis wartości zwracanych przez funkcję
*
* Szczegółowy opis funkcji
**/
int add(int a, int b){
…
```

Poniżej przedstawiono przykładowy opis rzeczywistej metody:

```
/**
* @brief Set camera by specyfing vectors
* @param eye Eye position
* @param at Point to look at
* @param up Camera orientation vector
**/
void setCamera(Vec3 eye, Vec3 at, Vec3 up);
```

5. Opis realizacji

Projekt został zrealizowany w oparciu o standardowe biblioteki dostarczone wraz z kompilatorem (m. in. STL, OpenGL, GLU). Jako podstawę do obsługi interfejsu graficznego wykorzystana została biblioteka OpenGL. Dodatkowo użyta została biblioteka FreeGLUT[7], będąca nakładką umożliwiającą prostsze wykorzystanie OpenGL, przede wszystkim inicjalizację, tworzenie okna czy obsługę klawiatury. Silnik fizyki wykorzystuje wyłącznie standardowe biblioteki.

Oprogramowanie tworzone było w środowisku Visual Studio Code w języku C++ na 64-bitowej platformie Microsoft Windows 10. Cały program kompilowano przy użyciu MinGW-64. Jako system budowania projektu wykorzystano CMake, a kontrola zmian odbywała się poprzez system Git, wraz z regularną synchronizacją z serwisem GitHub.

Prekompilowane oprogramowanie działa wyłącznie na systemach 64-bitowych. Do uruchomienia wymagane są dołączone biblioteki współdzielone (DLL) oraz obsługa OpenGL w wersji 2.0 lub nowszej.

Oprogramowanie zostało tworzone modułowo. Początkowo powstał system graficzny, umożliwiający wizualną obserwację działania całości. Po pomyślnym jego zaimplementowaniu, gdy możliwe było tworzenie kształtów i zmiana perspektywy obserwacji, przystąpiono do budowy silnika fizyki. Rozpoczęto od implementacji wpływu sił na ciała, a następnie wdrożono wykrywanie i obsługę zderzeń. Cały program został napisany zgodnie z paradygmatem OOP. Uwagę poświęcono przede wszystkim zachowaniu maksymalnej abstrakcji między klasami, dzięki czemu możliwe jest niezależne uruchomienie podsystemu fizyki i grafiki, a także zapewnienie jak największej i najłatwiejszej skalowalności, aby nie były konieczne modyfikacje już istniejących klas.

Diagramy UML, ukazujące budowę programu, przedstawiono w sekcji 4.

[7] https://freeglut.sourceforge.net/

6. Opis wykonanych testów - lista bugów, uzupełnień, itd.

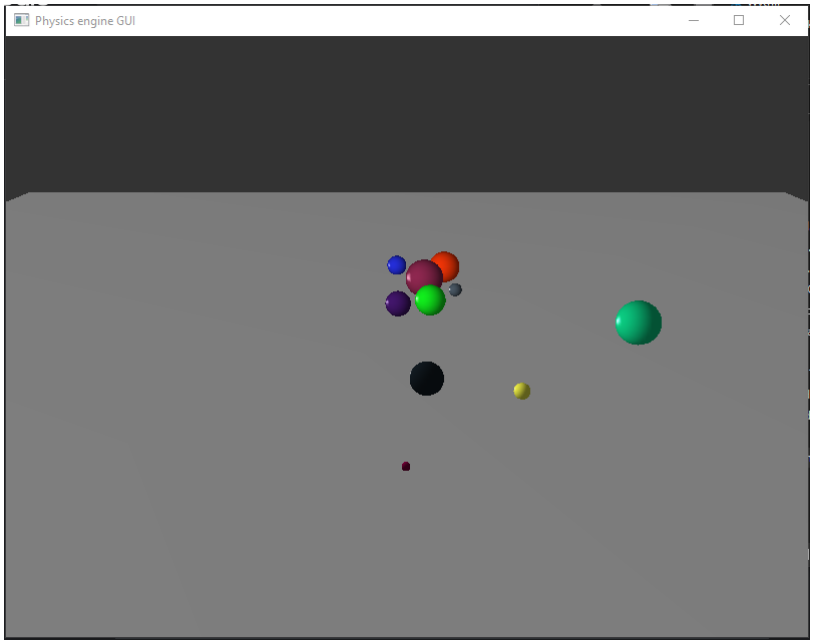
Kod usterki	**Data**	**Autor**	**Opis**	**Stan**
#BP1	10.01.2023	Piotr Wilkoń	Ciała początkowo nachodzące na siebie pozostają złączone	Rozwiązany (16.01.2023)
#BG1	10.01.2023	Piotr Wilkoń	Brak oświetlenia przy kompilacji *Release*	Nierozwiązany
#BG2	16.01.2023	Piotr Wilkoń	Brak widoku przy ustawieniu kamery u góry prostopadle do ziemi	Rozwiązany (16.01.2023)

7. Podręcznik użytkownika

7.1 Uruchomienie programu

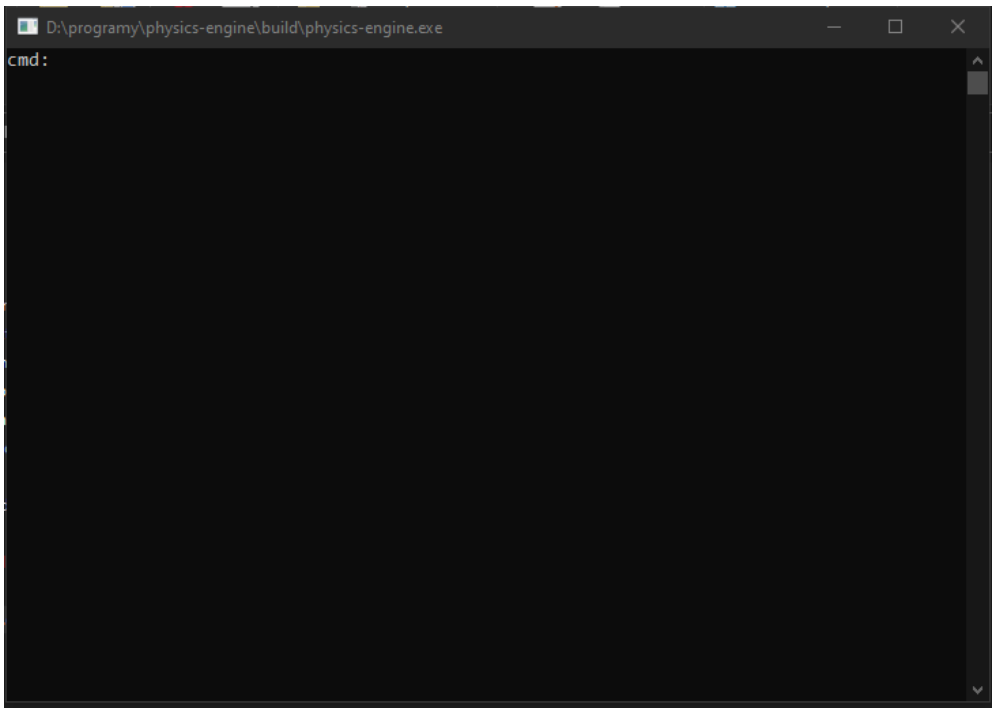
Oprogramowanie należy uruchamiać na 64-bitowej platformie Microsoft Windows 10 z obsługą OpenGL w wersji 2.0 lub nowszej. Do uruchomienia konieczne są odpowiednie biblioteki współdzielone, które są załączone wraz ze skompilowanym programem.

Program otwiera się w jednym lub dwóch oknach. Okno przedstawione na Rysunku 7-1 jest oknem interfejsu graficznego. Jest ono widoczne tylko wówczas, gdy użytkownik umyślnie nie wyłączy interfejsu graficznego.



Rysunek 7-1. Okno interfejsu graficznego

Drugim oknem jest okno interfejsu tekstowego (terminala), w którym można wprowadzać komendy. Okno to jest zawsze widoczne i przedstawiono je na rysunku 7-2.



Rysunek 7-2. Okno interfejsu tekstowego

7.1.1 Argumenty startowe

Program uruchamiany bez argumentów startowych przyjmuje opcje domyślne.

Do uruchomienia programu można zastosować następujące argumenty (przełączniki):

- *-k skala* – ustawia skalowanie czasu symulacji
- *-n ilość* – ustawia liczbę losowo generowanych obiektów
- *-x wymiar* – ustawia szerokość okna w pikselach
- *-y wymiar* – ustawia wysokość okna w pikselach
- *-g* – wyłącza interfejs graficzny. Interfejs graficzny jest domyślnie włączony.
- *-a* – wyłącza automatyczne startowanie symulacji. Automatyczny start jest domyślnie włączony.
- *-h* – pokazuje stronę pomocy

Aby sprawdzić domyślne wartości startowe, należy uruchomić program z argumentem *-h*.

7.1.2 Interfejs graficzny

W interfejsie graficznym możliwe jest użycie klawiszy do zmiany perspektywy obserwacji, zatrzymania i wznowienia symulacji.

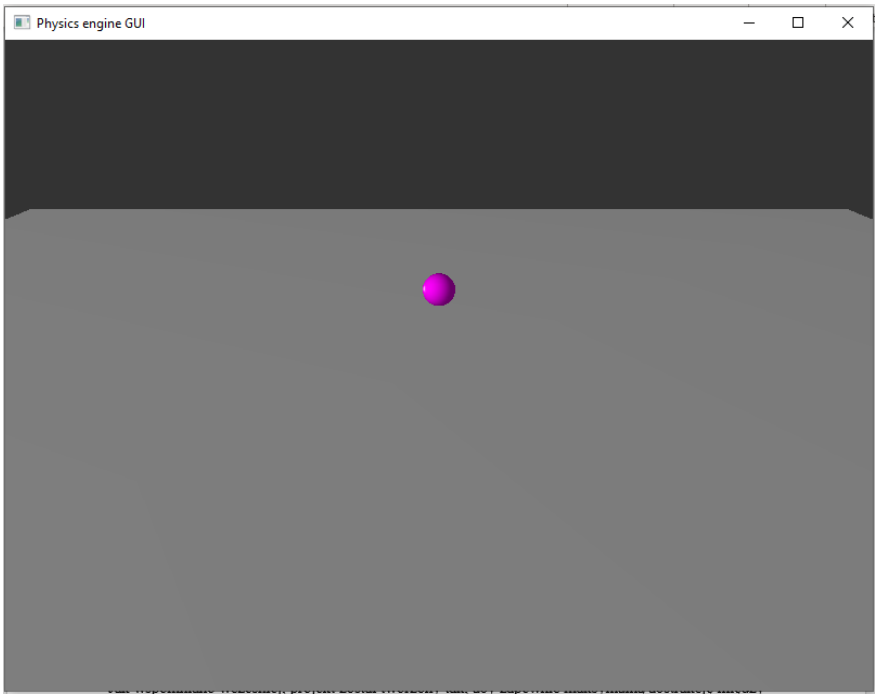
- Spacja – zatrzymuje i wznawia symulację
- Strzałki – zmieniają pozycję kamery wg współrzędnych sferycznych
- Z, X – odpowiednio: oddala i przybliża kamerę

7.1.3 Interfejs tekstowy

Interfejs tekstowy umożliwia dodanie obiektów w trakcie działania programu. Dostępna jest komenda *add*, która prowadzi użytkownika przez kreator dodawania obiektu, jak przedstawiono na rysunkach 7-3 oraz 7-4.

```
cmd: add
Position x: 0
Position y: 1
Position z: 0
Radius: 0.2
Mass: 1
Set initial velocity (y/N)?
Set coefficient of restitution (y/N)? n
Set color (y/N)? y
Color R: 1
Color G: 0
Color B: 1
Object added
cmd:
```

Rysunek 7-3. Dodawanie obiektu z użyciem polecenia *add*



Rysunek 7-4. Dodany obiekt widoczny w przestrzeni

8. Metodologia rozwoju i utrzymania systemu

Jak wspomniano wcześniej, projekt został tworzony tak, aby zapewnić maksymalną abstrakcję między zasadniczymi blokami. Dodawanie kolejnych typów ciał fizycznych wymaga jedynie utworzenia nowej klasy dla danego ciała, dziedziczącej po bazowej klasie dla wszystkich ciał fizycznych, a także dodania nowego typu ciała do odpowiedniego typu wyliczeniowego. Obiekt takiej klasy można stworzyć, powiązać z obiektem graficznej reprezentacji i obiektem opisującym bryłę ograniczającą. Identycznie jest w przypadku tworzenia nowego typu kształtu graficznego. Dzięki niezależności warstwy fizycznej i graficznej fizyczne ciało może nie być powiązane z żadnym konkretnym typem kształtu graficznego i wówczas podlega tylko symulacji fizycznej, bez graficznej reprezentacji, co może ułatwić jednostkowe sprawdzenie nowo dodanej funkcjonalności.

Bibliografia

[1] Cyganek B.: Introduction to Programming with C++ for Engineers, Wiley-IEEE Press, 2020.

[2] Halliday D., Resnick R., Walker J.: Fundamentals of Physics. Part 1, Wiley, 2021.

[3] Korn G. A., Korn T. M.: Mathematical Handbook for Scientists and Engineers, McGraw-Hill, 1961.

[4] Stroustrup B.: The C++ Programming Language, Pearson Education, 2013.

[5] Song Ho Ahn: OpenGL Tutorials - http://www.songho.ca/opengl/

[6] OpenGL Documentation - https://docs.gl/

[7] GLU Documentation - https://registry.khronos.org/OpenGL-Refpages/gl2.1/xhtml/

[8] FreeGLUT Documentation - https://freeglut.sourceforge.net/docs/api.php