Fizyka i silniki fizyczne
nVidia PhysX3

Część 3: Transformacje, siły, materiały, optymalizacja

Podstawowe klasy geometryczne

Podstawowe klasy związane z organizacją przestrzeni 3D i transformacjami to:

- **PxVec3**, **PxVec4** – wektor 3 i 4-elementowy złożony z liczb typu PxReal (float 32-bity);
- **PxMat33**, **PxMat44** – macierz 3x3 i 4x4 złożona z liczb typu PxReal;
- **PxQuat** – obrót reprezentowany przez kwaternion;
- **PxTransform** – reprezentacja transformacji złożona z wektora translacji (**PxVec3**) i kwaterniona obrotu (**PxQuat**).

Klasa PxVec3 i PxVec4

Metoda (PxVec3)	Opis
PxVec3()	Domyślny konstruktor (bez inicjalizacji danych)
PxVec3(PxReal a)	Konstruktor skalarny → wartość „a” przypisywana do wszystkich pól
PxVec3(PxReal nx, PxReal ny, PxReal nz)	Konstruktor z 3 współrzędnymi podanymi wprost
bool **isNormalized**()	Zwraca „true” jeżeli wektor jest znormalizowany
PxReal **magnitude**()	Długość wektora
PxReal **magnitudeSquared**()	Kwadrat długości wektora
PxReal **normalize**()	Normalizacja wektora
float **minElement**()	Minimalna wartość z elementów wektora
float **maxElement**()	Maksymalna wartość z elementów wektora
PxVec3 **abs**()	Moduł elementów wektora jako nowy wektor

Operacje na wektorach

Klasy PxVec3 i PxVec4 posiadają przeciążone operatory, między innymi:

Operator	**Opis**
=	Przypisanie (kopia danych wektora)
[]	Dostęp do pól wektora (dozwolony zakres 0..2 lub 0..3)
==, !=	Porównanie dwóch wektorów
- (jeden argument)	Zmiana znaku wektora → odwrócenie
+, -	Dodanie, odjęcie dwóch wektorów
*, /	Mnożenie, dzielenie przez pojedynczą liczbę

Mnożenie wektorów (PxVec3 i PxVec4)

Metoda (PxVec3)	**Opis**
PxReal **dot**(const PxVec3 &v)	Iloczyn skalarny dwóch wektorów
PxVec3 **cross**(const PxVec3 &v)	Iloczyn wektorowy
PxVec3 **multiply**(const PxVec3 &a)	Iloczyn skalarny odpowiadających sobie współrzędnych, wynik w formie wektora

Macierze

Macierze **PxMat33** i **PxMat44** zbudowane są z wektorów:

- **PxMat33** z trzech wektorów typu **PxVec3** dostępnych jako **column0**, **column1** i **column2**;
- **PxMat44** z czterech wektorów typu **PxVec4** dostępnych jako **column0**, **column1**, **column2** i **column3**.

Analogicznie jak wektory, klasy posiadają przeciążone operatory matematyczne. Operacja mnożenia (*) może dotyczyć wartości skalarnej, wektora (**PxVec3** lub **PxVec4**) lub innej macierzy o takich samych rozmiarach.

Klasa PxMat33 i PxMat44

Metoda (PxMat33)	Opis
PxMat33()	Domyślny konstruktor, bez inicjalizacji
PxMat33(const PxVec3 &col0, const PxVec3 &col1, const PxVec3 &col2)	Konstruktor składający macierz z kolumn podanych jako wektory
PxMat33(PxReal r)	Konstruktor tworzący macierz jednostkową pomnożoną przez „r”
PxMat33(PxReal values[])	Konstruktor tworzący macierz z 9 podanych wartości
PxMat33(const PxQuat &q)	Utworzenie macierzy obrotu na podstawie podanego kwaterniona
PxMat33 **getTranspose**()	Zwraca macierz transponowaną
PxMat33 **getInverse**()	Zwraca macierz odwrotną
PxReal **getDeterminant**()	Policzenie wyznacznika macierzy (tylko 3x3)

Kwaterniony

Kwaternion (klasa **PxQuat**) reprezentuje obrót z pomocą czterech współrzędnych typu PxReal: **x**, **y**, **z** i **w.**

Konstruktor	**Opis**
PxQuat()	Konstruktor domyślny, bez inicjalizacji
PxQuat(PxReal r)	Konstruktor skalarny, wartość „r” przypisuje wartości skalarnej „w”, pozostałe współrzędne na zero
PxQuat(PxReal nx, PxReal ny, PxReal nz, PxReal nw)	Konstruktor tworzący kwaterniona z czterech bezpośrednio podanych współrzędnych
PxQuat(PxReal angleRadians, const PxVec3 &unitAxis)	Konstrukcja kwaterniona na podstawie kąta i osi obrotu (podanej jako wektor)
PxQuat(const PxMat33 &m)	Kwaternion z macierzy obrotu 3x3

Metody PxQuat

Metoda	Opis
bool **isUnit**()	Zwraca „true” jeżeli kwaternion jest jednostkowy
void **toRadiansAndUnitAxis** (PxReal &angle, PxVec3 &axis)	Zwraca reprezentacje obrotu w postaci osi i kąta obrotu
PxReal **getAngle**()	Kąt między kwaternionem bieżącym i jednostkowym
PxReal **getAngle**(const PxQuat &q)	Kąt między kwaternionem bieżącym i podanym
PxReal **dot**(const PxQuat &v)	Iloczyn skalarny dwóch kwaternionów
PxReal **normalize**()	Normalizacja kwaterniona

Przeciążone operatory PxQuat

Operator	Opis
==	Porównanie czy kwaterniony są identyczne
=	Przypisanie (kopia danych kwaterniona)
*	Mnożenie przez inny kwaternion lub liczbę
+, -	Dodanie, odjęcie dwóch kwaternionów
- (jeden argument)	Zmiana znaku kwaterniona

Transformacje

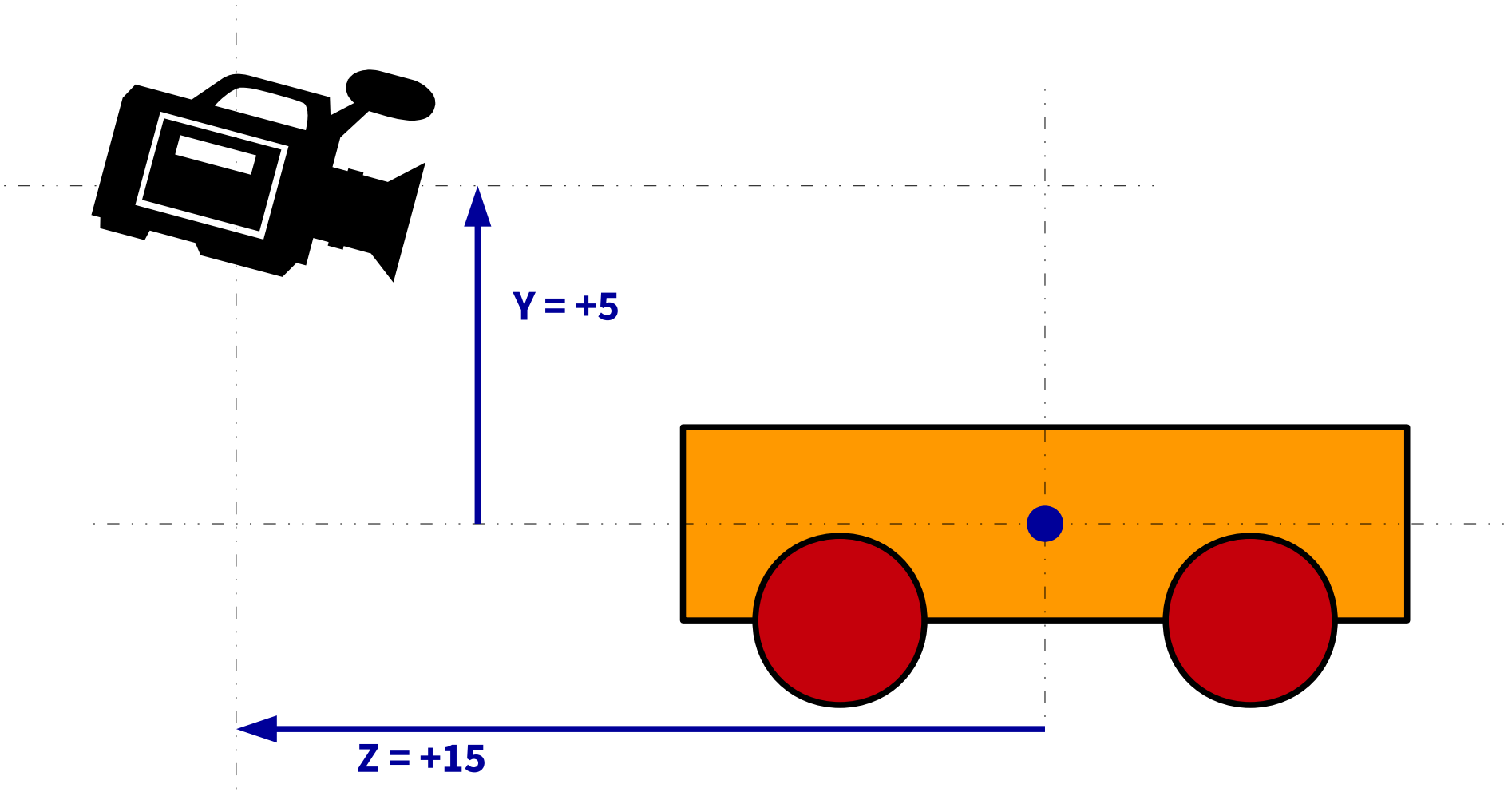
Klasa **PxTransform** reprezentuje transformację w przestrzeni złożoną z wektora translacji i kwaterniona rotacji.

Konstruktor	**Opis**
PxTransform()	Konstruktor domyślny, bez inicjalizacji
PxTransform(const PxVec3 &position)	Konstruktor z wektorem przesunięcia, bez rotacji
PxTransform(const PxQuat &orientation)	Konstruktor z rotacją, bez przesunięcia (0, 0, 0)
PxTransform(const PxVec3 &p0, const PxQuat &q0)	Transformacja złożona z wektora przesunięcia i rotacji
PxTransform(const PxMat44 &m)	Transformacja opisana macierzą 4x4 (rotacja i przesunięcie)

PxTransform – metody

Metoda	Opis
PxTransform **getInverse**()	Zwraca transformację odwrotną (przeciwną)
PxVec3 **transform**(const PxVec3 &input)	Transformuje podany wektor (rotacja i translacja)
PxVec3 **transformInv**(const PxVec3 &input)	Jak wyżej, ale w przeciwną stronę
PxTransform **transform**(const PxTransform &src)	Łączenie dwóch transformacji – zwraca nowy obiekt który wykonuje najpierw transformację „src” a następnie obiektu którego jest to metoda
* (operator)	Łączenie dwóch transformacji (jak wyżej)

Transformacje – przykład



Transformacje – przykład

```
// Elementy pojazdu:
PxRigidDynamic *a_carbody;
PxRigidDynamic *a_wheel[4];

// ...

// Odczytanie pozycji pojazdu i stosowne ustawienie kamery:
PxTransform cpose = a_carbody->getGlobalPose();
PxVec3 cp = cpose.transform(PxVec3(0, 0, 0));
PxVec3 ce = cpose.transform(PxVec3(0, 5, 15));

if(ce.y < 5)   ce.y = 5;

// Ustawienie kamery śledzącej pojazd:
gluLookAt(ce.x, ce.y, ce.z, cp.x, cp.y, cp.z, 0, 1, 0);
```

Parametry materiałów

Materiały

Materiał definiuje właściwości fizyczne powierzchni aktora. Materiały tworzy się z pomocą metody **createMaterial()** klasy **PxPhysics** i przypisuje do aktorów w czasie ich tworzenia.

```
PxMaterial *createMaterial ( PxReal staticFriction,
                             PxReal dynamicFriction,
                             PxReal restitution )
```

- **staticFriction** – wsp. tarcia statycznego (0..PX_MAX_F32);
- **dynamicFriction** – wsp. tarcia dynamicznego (0..PX_MAX_F32);
- **restitution** – wsp. sprężystości (0..1).

Wybrane metody klasy PxMaterial

`void` **`setDynamicFriction`** `(PxReal coef)`

Ustawienie współczynnika tarcia dynamicznego (0..PX_MAX_F32).

`void` **`setStaticFriction`** `(PxReal coef)`

Ustawienie współczynnika tarcia statycznego (0..PX_MAX_F32).

`void` **`setRestitution`** `(PxReal rest)`

Ustawienie współczynnika sprężystości (0..1).

`void` **`setFrictionCombineMode`** `(PxCombineMode::Enum combMode)`

Określenie sposobu wyznaczania wypadkowego tarcia przy zetknięciu dwóch aktorów.

`void` **`setRestitutionCombineMode`** `(PxCombineMode::Enum combMode)`

Określenie sposoby wyznaczania wypadkowej sprężystości przy zderzeniu dwóch aktorów.

Wypadkowa sprężystość i tarcie

Jeżeli aktory korzystające z dwóch różnych materiałów zderzają się ze sobą, domyślnie ich współczynniki są uśredniane.

Możliwe tryby wyznaczania wypadkowego tarcia i sprężystości określone są przez typ **PxCombineMode** (enum):

eAVERAGE	Uśrednianie: (a + b)/2
eMIN	Wartość minimalna: min(a, b)
eMAX	Wartość maksymalna: max(a, b)
eMULTIPLY	Iloczyn: a * b

Wypadkowa sprężystość i tarcie

Tylko jeden tryb mieszania współczynników jest brany pod uwagę przy kolizji dwóch aktorów.

Jeżeli aktory wykorzystują materiały z różniącym się trybem mieszania, wybierane są tryby o najwyższym priorytecie zgodnie z kolejnością:

1) **eMAX** (najwyższy priorytet)
2) **eMULTIPLY**
3) **eMIN**
4) **eAVERAGE** (najniższy priorytet)

Niszczenie materiałów

Zmniejszenie licznika użycia materiału:

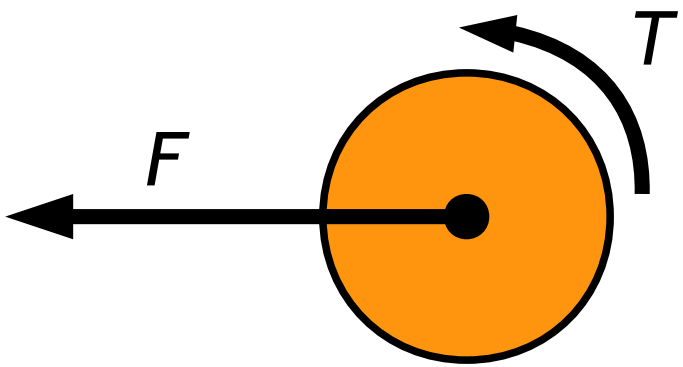
```
void PxMaterial::release ()
```

Obiekt reprezentujący materiał zostanie faktycznie zniszczony, tylko jeżeli nie jest przypisany do żadnego aktora!

Siły działające na aktorów

Siły działające na aktorów

Do każdego aktora (klasa PxActor) można przyłożyć działającą na niego w dowolnym kierunku **siłę** (ang. *force*) lub **moment obrotowy** (ang. *torque*).



Przyłożenie siły – PxRigidBody

```
void addForce(
        const PxVec3 &force,
        PxForceMode::Enum mode = PxForceMode::eFORCE,
        bool autowake = true )
```

- **force** - wartość siły/impulsu w formie wektora we współrzędnych globalnych;
- **mode** - sposób przyłożenia siły*;
- **autowake** - automatyczne budzenie aktora TAK/NIE.

```
void addTorque(
            const PxVec3 &torque,
            PxForceMode::Enum mode = PxForceMode::eFORCE,
            bool autowake = true )
```

- **torque** - wartość siły/impulsu rotacji w formie wektora - rotacja wokół każdej osi we współrzędnych globalnych;
- **mode** - sposób przyłożenia siły*;
- **autowake** - automatyczne budzenie aktora TAK/NIE.

PxForceMode

Charakter siły/rotacji którą można przyłożyć do aktora:

Stała	Opis
eFORCE	Jednostka to masa * przyspieszenie (F = m*a).
eIMPULSE	Jednostka to masa * prędkość (p = m * v).
eVELOCITY_CHANGE	Jednostka to droga / czas (v = s/t). Zmiana prędkości niezależna od masy.
eACCELERATION	Jednostka to droga / czas do kwadratu (a = s/t2).

Przyłożenie siły – PxRigidBodyExt

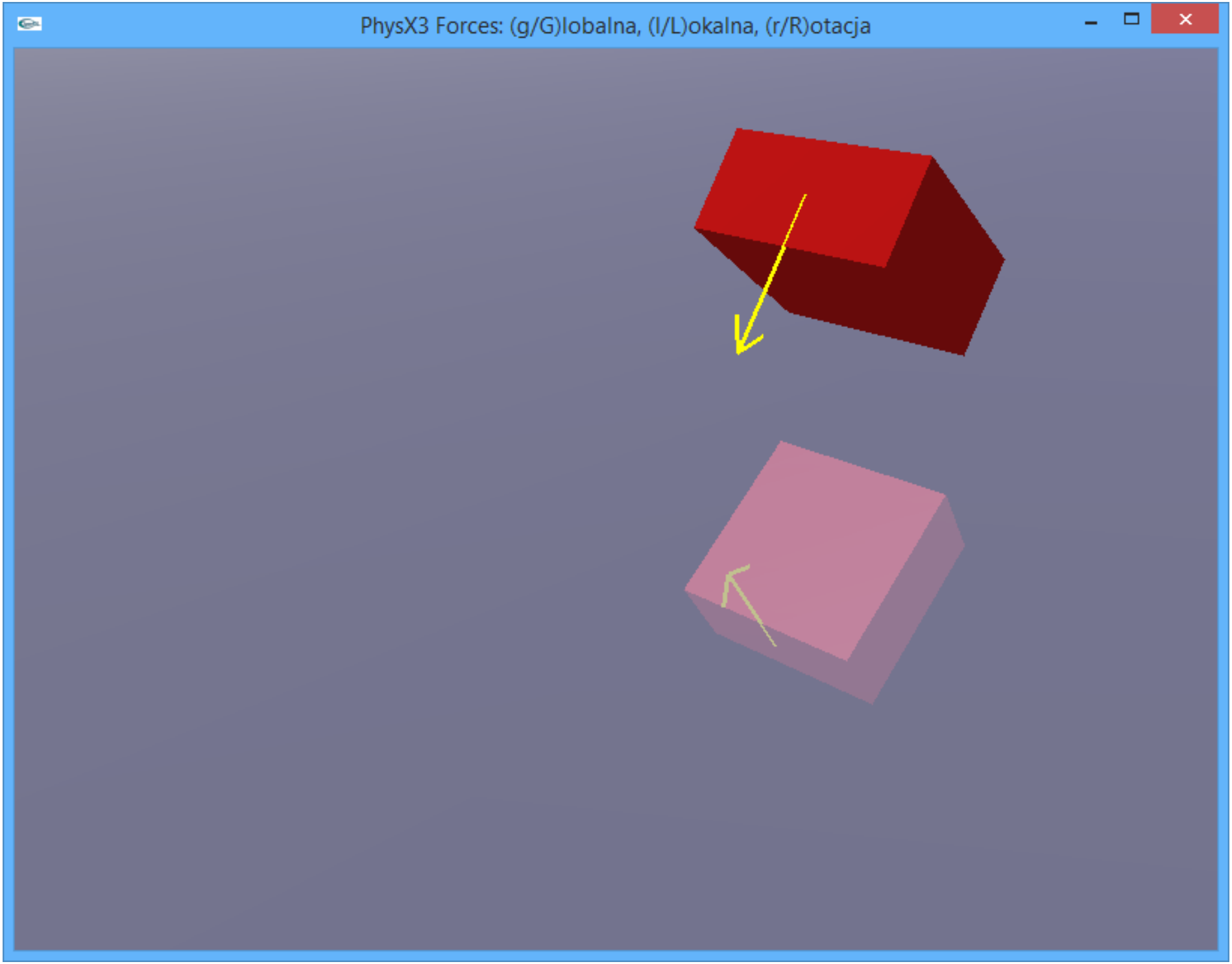
```
void addForceAtPos(
      PxRigidBody &body,
      const PxVec3 &force, const PxVec3 &pos,
      PxForceMode::Enum mode = PxForceMode::eFORCE,
      bool wakeup = true)
```

- **body** - aktor dynamiczny do którego przykładamy siłę,
- **force** - wektor siły,
- **pos** - punkt zamocowania,
- **mode** - sposób przyłożenia (dozwolone tylko eFORCE i eIMPULSE),
- **wakeup** - wybudzenie ze stanu uśpienia TAK/NIE.

Metoda	**PxVec3 *force***	**PxVec3 *pos***
addForceAtPos	globalny	globalny
addForceAtLocalPos	globalny	lokalny (obiektu)
addLocalForceAtPos	lokalny (obiektu)	globalny
addLocalForceAtLocalPos	lokalny (obiektu)	lokalny (obiektu)

Przyłożenie siły – przykład

```
// Prostopadłościan:
PxRigidDynamic *a_box;

// …

PxVec3 _force_vec = PxVec3(0, 0,  3000);
// Lokalna siła w lokalnym punkcie:
PxRigidBodyExt::addLocalForceAtLocalPos(*a_box, _force_vec,
                            PxVec3(2, 0, 0), PxForceMode::eIMPULSE);

// Globalna siła w lokalnym punkcie:
PxRigidBodyExt::addForceAtLocalPos(*a_box, _force_vec,
                            PxVec3(2, 0, 0), PxForceMode::eIMPULSE);

// …

// Konwersja wektora siły rotacji do współrzędnych lokalnych:
PxVec3 siła(0, 0, 10000);
PxTransform pose = a_box->getGlobalPose();
a_box->addTorque(pose.transform(siła), PxForceMode::eIMPULSE);
```


PhysX3 Forces: (g/G)lobalna, (l/L)okalna, (r/R)otacja

Siła/rotacja o charakterze trwałym

Wszystkie metody określania siły i rotacji są **funkcją czasu** – efekt znika samoczynnie po chwili czasu.

Aby siła była przyłożona w sposób trwały, należy cyklicznie ponawiać wywołanie metod **addForce()** / **addTorque()**.

Aktory kinematyczne

Aktory kinematyczne

Aktor kinematyczny – to specyficzny rodzaj aktora dynamicznego z nieskończenie dużą masą, na którego nie działają żadne siły (również grawitacja).

Aktor kinematyczny nie oddziałuje (nie zderza się) z innymi aktorami kinematycznymi i statycznymi.

W przeciwieństwie do aktorów statycznych, aktory kinematyczne mogą się przemieszczać. Szczególnie dobrze nadają się do symulowania obiektów których ruch należy precyzyjnie kontrolować, np. windy, ruchome platformy itp.

Aktory kinematyczne – tworzenie

Aktory kinematyczne tworzy się tak samo jak dynamiczne (**PxRigidDynamic**), lecz po jego utworzeniu należy ustawić flagę **eKINEMATIC**.

```
// Cztery prostopadłościany:
PxRigidDynamic *a_box[4];

a_box[0]->setRigidDynamicFlag(
                PxRigidDynamicFlag::eKINEMATIC, true);
```

Drugi równoważny sposób to użycie funkcji **PxCreateKinematic**(), która automatycznie ustawia tę flagę.

Aktory kinematyczne – przemieszczanie

Aktorów kinematycznych można przemieszczać przy pomocy metody **setKinematicTarget**() klasy **PxRigidDynamic**.

```
void setKinematicTarget(const PxTransform &destination)
```

Metoda nadaje aktorowi prędkość, która w kolejnym kroku symulacji spowoduje że aktor znajdzie się w miejscu (i rotacji) wskazanym przez parametr „destination”.

Aby uzyskać ciągły ruch aktora (np. po prostej) należy tę metodę wywoływać w każdym kroku symulacji, z odpowiednio przesuniętą pozycją.

Usypianie aktorów

Usypianie aktorów

Mechanizm usypiania aktorów (ang. sleep) to jedna z podstawowych **metod optymalizacji** obliczeń w bibliotece PhysX.

W idealnej sytuacji tylko aktory znajdujące się w bryle widzenia powinny być aktywne! Wszystkie inne powinny być w stanie uśpienia.

Aktory usypiane są automatycznie, jeżeli ich parametry spełnią określone warunki.

Moment uśpienia aktora

Parametry bezpośrednio i pośrednio określające moment uśpienia i budzenia aktorów to:

1) minimalna wartość **energii kinetycznej** (ang. *sleep energy threshold*) poniżej której aktor może zostać uśpiony;
2) wartość **licznika budzenia** (ang. wake counter) określająca czas symulacji po której nieaktywny aktor może być uśpiony;
3) grubość „skóry” aktora (ang. *contact offset*).

Jeśli energia kinetyczna aktora spadnie poniżej określonego progu i upłynie określony czas symulacji, jest on automatycznie **usypiany**. Grubość skóry określa sposób reakcji aktorów z innymi aktorami i w efekcie wpływa na moment ich „wybudzenia”.

Metody klasy PxRigidDynamic związane z usypianiem

`void` **`setSleepThreshold`** `( PxReal threshold )`

Ustawienie progu energii kinetycznej znormalizowanej względem masy. Jeżeli energia kinetyczna aktora podzielona przez jego masę jest niższa niż zadany próg, aktor może przejść w stan uśpienia.

Wartość domyślna: $5 * 10^{-5} * V_s = 0{,}0005$

Gdzie V_s = PxTolerancesScale::speed (domyślnie 10)

`void` **`setWakeCounter`** `( PxReal wakeCounterValue )`

Określenie minimalnego czasu, po upłynięciu którego aktor może przejść w stan uśpienia.

Wartość domyślna: *0.4* (s)

Metody klasy PxRigidDynamic związane z usypianiem

`void` **`putToSleep ( )`**

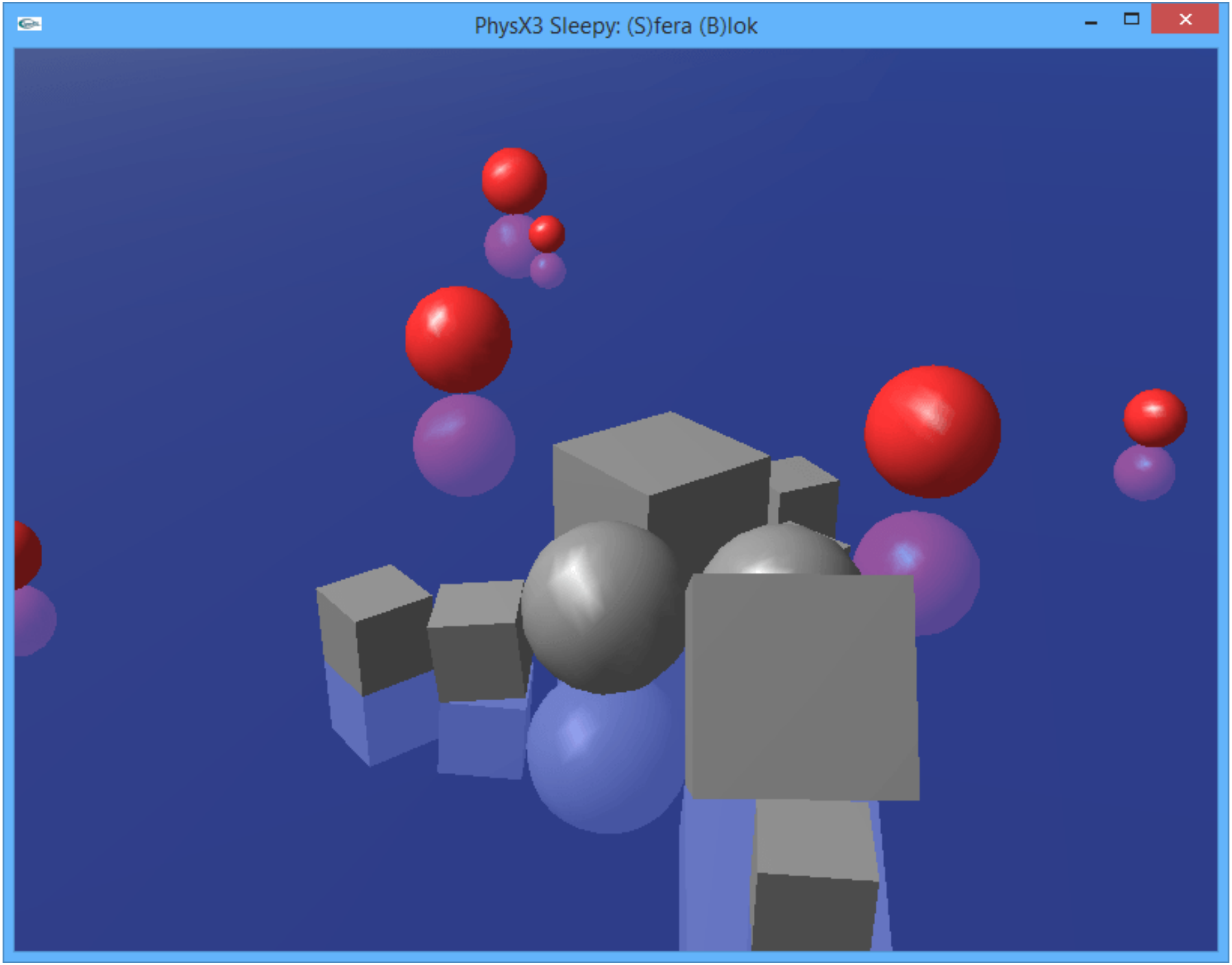
Natychmiastowe uśpienie aktora. Przy kolejnym wywołaniu metody simulate() może zostać automatycznie wybudzony.

`void` **`wakeUp ( )`**

Obudzenie aktora.

`bool` **`isSleeping ( )`**

Zwraca „true” jeżeli aktor jest w stanie uśpienia.


PhysX3 Sleepy: (S)fera (B)lok

Grubość skóry

Grubość „skóry” aktora (parametry „*rest offset*” i „*contact offset*”) określa głębokość na jaką aktory mogą się nawzajem „penetrować”. Różna grubość „skóry” to różnego rodzaju materiały:

- **Gruba skóra** symuluje miękkie, uginające się materiały jak karton, gąbka, guma itp.
- **Cienka skóra** odpowiada materiałom twardym takim jak beton, metal itp.

Aktory posiadające cienką skórę są łatwiej budzone z uśpienia, ponieważ ich reakcja na kolizje jest szybsza i bardziej „nerwowa”.

Grubość skóry to bardzo istotny parametr. Dobranie odpowiednich ustawień dla aktorów zapewnia większy realizm symulacji i mniejsze wymagania obliczeniowe (aktory częściej są usypiane).

Określenie grubości skóry – klasa PxShape

```
void setContactOffset ( PxReal contactOffset )
```

Jeżeli aktory zbliżą się do siebie na odległość mniejszą niż suma ich wartości „*contact offset*”, uważa się że aktory się zetknęły.

Zakres: *[maks(0, restOffset), PX_MAX_F32)*

Wartość domyślna: *0,02* (m)

```
void setRestOffset ( PxReal restOffset )
```

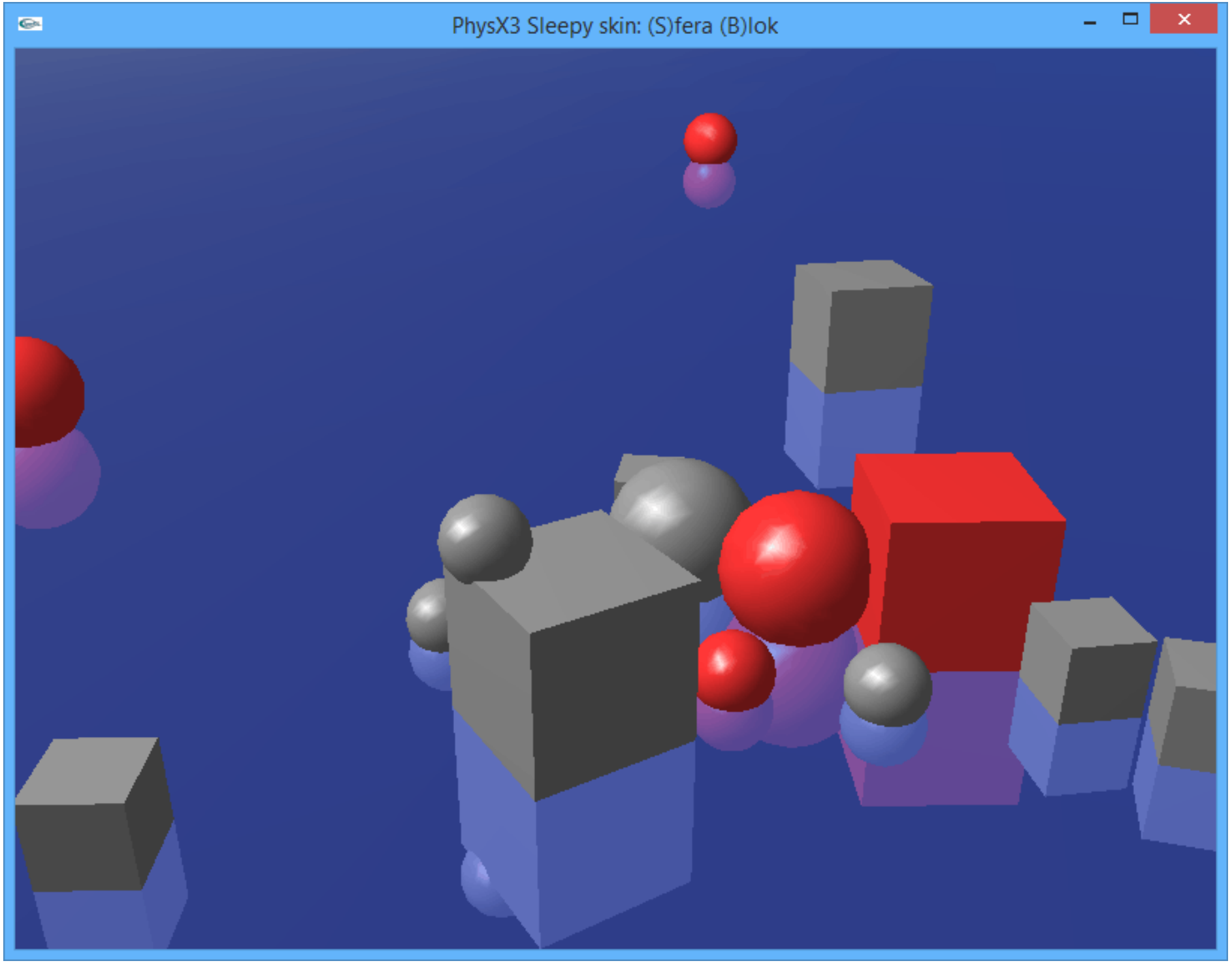
Jeżeli aktory zbliżą się do siebie na odległość mniejszą niż suma ich wartości „*rest offset*”, uważa się że powierzchnie aktorów fizycznie się zetknęły (zderzają się).

Zakres: *(-PX_MAX_F32, contactOffset)*

Wartość domyślna: *0,0*

Zmiana grubości „skóry” - przykład

```
PxRigidDynamic* make_sfera(PxMaterial *mat, float radius,
                                            float x, float y, float z) {
    // Położenie aktora i kształt sfery:
    PxTransform transform(PxVec3(x, y, z));
    PxSphereGeometry geometry(radius);

    // Przygotowanie kształtu:
    PxShape *kształt = pphys->createShape(geometry, *mat);
    // Zmiana parametrów skóry:
    kształt->setContactOffset(0);
    kształt->setRestOffset(-0.25);

    // Utworzenie aktora:
    PxRigidDynamic *actor = PxCreateDynamic(*pphys, transform,
                                                    *kształt, 20.0f);
    gsc->addActor(*actor);
    return(actor);
}
```


PhysX3 Sleepy skin: (S)fera (B)lok