Autor

Podstawy mechaniki klasycznej dla programistów gier, czyli rzecz o tym, jak całkować równanie ruchu

Gry oparte na fizyce tworzone są przynajmniej od lat osiemdziesiątych zeszłego wieku. Wystarczy wspomnieć klasyczne już symulatory lotu, gry samochodowe, czy „strzelanki", w których bardzo realistycznie zachowują się ciała zastrzelonych przeciwników. Ostatnio coraz popularniejsze stają się także gry typu *casual*, także te przeznaczone na urządzenia przenośne, w których fizyka odgrywa kluczową rolę. Dobrym przykładem jest mój tablet, na którym dzieci w ciągu pierwszego miesiąca zainstalowały już *Cut the rope*, *Where is my water*, *World of Goo* i oczywiście kilka wersji *Angry Birds*. Wszystkie je łączy wykorzystanie fizyki. Robią to wprawdzie w różnym stopniu, ale to właśnie ten element w dużym stopniu decyduje o ich dużej grywalności. Ja sam bardzo lubię grać w *Trine* i *Trine 2* oraz w gry z serii *Half-Life* w których fizyka jest również kluczowa.

Zmorą wszystkich fizyków starających się pisać artykuły popularne jest „klątwa Hawkinga". Stephen Hawking w swojej *Krótkiej historii czasu* napisał, że każdy wzór w tekście zmniejsza liczbę czytelników o połowę. Ponieważ w tym artykule jest 20 wzorów, istnieje groźba, że na jego przeczytanie zdecyduje się tylko $1/2^{20}$, czyli 1/1048576 potencjalnych czytelników! Biorąc pod uwagę nakład *Programisty*, a nawet populację polskich programistów, oznaczałoby to maksymalnie jednego czytelnika. Trochę to mnie oczywiście niepokoi. Niestety inaczej, tzn. bez wzorów, o fizyce w grach opowiedzieć się nie da.

Załóżmy, że i my chcemy wprowadzić do tworzonej przez nas gry fizykę. Musimy się zastanowić czego ma ona dotyczyć i jakie obiekty będą jej podlegać. Jeżeli przez „fizykę" rozumiemy ruch i zderzenia brył sztywnych takich, jak pudełka, elementy konstrukcji czy pojazdy – wówczas najlepiej i najwygodniej jest użyć gotowych silników fizycznych, jak choćby *NVIDIA PhysX*, *Box2D*, *Bullet*, *Farseer*, czy jeden z wielu innych dostępnych w Internecie. Równania ruchu brył sztywnych są trudne, za trudne aby implementować je samemu. Ich implementacja, późniejsze tropienie błędów i optymalizacja zajęłyby zbyt wiele czasu, który warto poświęcić raczej na szlifowanie innych elementów gry decydujących o jej przyszłym powodzeniu. Zupełnie samodzielnie możemy natomiast zaimplementować ruch i zderzenia punktów materialnych. Tyle wystarczy chociażby do przygotowania początkowych poziomów *World of Goo*.

KRÓTKIE POWTÓRZENIE SZKOLNEJ MECHANIKI

Ruch punktów materialnych opisuje równanie odkryte pod koniec XVII wieku przez Isaaca Newtona:

1. $m\ddot{\vec{r}}=\vec{F}(\vec{r},\dot{\vec{r}},t)$

W tym równaniu $\vec{r}$ oznacza położenie obiektu w trójwymiarowej przestrzeni, m to jego masa, a $\vec{F}$ – siła działająca na ten obiekt. Kropka nad wektorem oznacza różniczkowanie po czasie. Jest to tzw. równanie różniczkowe zwyczajne, z angielskiego określane skrótem ODE (ang. *ordinary differential equation*).

Równanie to można przepisać dla wektora prędkości $\vec{v}$, czyli pierwszej pochodnej położenia

2. $\vec{v}(t)=\dot{\vec{r}}(t)$

Ma ono wówczas postać:

3. $m\dot{\vec{v}}=\vec{F}(\vec{r},\vec{v},t)$

Mówi ono mniej więcej tyle, że wektor opisujący zmianę prędkości obiektu ma kierunek siły działającej na ów obiekt. Wielkość tej zmiany w określonym czasie jest tym większa im mniejsza jest masa obiektu. Masa jest wobec tego miarą bezwładności ciała, czyli tego jak trudno je rozpędzić. Zwróćmy uwagę, że z tego równania nie wynika, że ruch obiektu musi następować w kierunku, w którym działa siła (tak mylnie uważał Arystoteles). Tak nie jest chociażby w ruchu po okręgu, w którym siła dośrodkowa, a tym samym przyspieszenie, jest zawsze prostopadła do prędkości.

Równanie Newtona (1) możemy rozwiązać jeżeli za wektor $\vec{F}$ wstawimy jakąś konkretną postać siły – funkcję położenia, prędkości i ewentualnie czasu – opisującą oddziaływania w konkretnym układzie. Wówczas uzyskamy równanie ruchu. Rozwiązując je (całkując) znajdziemy prędkość i położenie ciała w funkcji czasu, czyli $\vec{v}(t)$ i $\vec{r}(t)$. W nielicznych przypadkach, np. jeżeli symulowany obiekt podlega sile grawitacji, która w pobliżu powierzchni Ziemi zależy tylko od masy ciała:

4. $\vec{F}=m\vec{g}$

lub sile harmonicznej

5. $\vec{F}(\vec{r})=-k\vec{r}$

szczególnie często wykorzystywanej w grach, rozwiązanie równania ruchu można znaleźć bez użycia komputerów. Każda komplikacja, chociażby

uwzględnienie interakcji z użytkownikiem, uniemożliwia takie podejście i symulacja komputerowa staje się jedynym możliwym podejściem. We wzorze (4) wektor $\vec{g}$ to przyspieszenie Ziemskie, a stała *k* we wzorze (5) nazywana jest współczynnikiem sprężystości. Ciekawostka: zwróćmy uwagę, że we wzorze (4) masa *m* nie jest już miarą bezwładności, a miarą podatności na oddziaływanie grawitacyjne.

To siłą harmoniczną, choć uzupełnioną o tłumienie, opisane są wiązania między kulkami Goo w grze *World of Goo*. Tej siły należy również użyć chcąc symulować ciała sprężyste: linę, siatkę lub kostkę galaretki. W takim przypadku rozwiązujemy równania ruchu osobno dla wielu obiektów. Jak zwróciłem uwagę, dla obu sił, przyciągania ziemskiego i siły harmonicznej, równania ruchu można rozwiązać analitycznie (na papierze). W pierwszym przypadku, musimy scałkować równanie ruchu postaci:

6. $$m\dot{\vec{v}}(t)=m\vec{g}$$

Masa z lewej strony tego równania (masa bezwładna) i prawej strony (masa grawitacyjna), choć opisujące różne cechy ciała, są wedle naszej wiedzy równe. Można je wobec tego w tym równaniu pominąć. W efekcie prędkość ciała nie zależy od jego masy. Oznacza to, że piórko w próżni powinno opadać w taki sam sposób, jak metalowa kulka. Całkując równanie (6) w przedziale czasu od 0 do *t* uzyskamy wzór na prędkość w ruchu jednostajnie przyspieszonym:

7. $$\vec{v}(t)=\vec{v}_0+\vec{g}t$$

Z prawej strony pojawiła się prędkość w chwili początkowej $\vec{v}_0$. Musimy ją znać, aby móc obliczyć prędkość w późniejszych chwilach – w końcu różniczkowe równanie Newtona wyznacza jedynie zmianę prędkości. Jeżeli następnie scałkujemy prędkość ze wzoru (7), obliczymy położenie obiektu w funkcji czasu – wzór dobrze znany ze szkoły:

8. $$\vec{r}(t)=\vec{r}_0+\vec{v}_0 t+\vec{g}\,t^2/2$$

Tu pojawia się kolejna stała całkowania – położenie początkowe $\vec{r}_0$. Położenie i prędkość początkowa tworzą warunki początkowe określające stan układu, z którego rozpoczynamy symulację.

METODA EULERA I METODA EULERA-CROMERA

Załóżmy zatem, że stajemy przed koniecznością rozwiązania równania (3) dla znanej siły $\vec{F}(\vec{r},\dot{\vec{r}},t)$ i znanych warunków początkowych. Jak się do tego zabrać? W pierwszym odruchu zapewne zaczniemy przeszukiwać Internet. A tam najczęstszą podpowiedzią jest metoda Eulera. W metodzie tej pochodne zastępuje się ilorazami różnicowymi:

9. $$\dot{f}(t)\approx\frac{f(t+\Delta t)-f(t)}{\Delta t}$$

czyli pomijając granicę, w której długość kroku czasowego zmierza do zera. Jasne jest wobec tego, że przybliżenie to powinno działać tym lepiej, im mniejszy jest krok czasowy Δt w symulacji. Stosując metodę Eulera dla równania ruchu (3) otrzymamy jego przybliżenie:

10. $$m\frac{\vec{v}(t+\Delta t)-\vec{v}(t)}{\Delta t}=\vec{F}(\vec{r},\vec{v},t)$$

W tym wzorze interesuje nas oczywiście prędkość w następnej chwili czasu $\vec{v}(t+\Delta t)$. Przekształćmy wobec tego powyższy wzór do postaci:

11. $$\vec{v}(t+\Delta t)=\vec{v}(t)+\frac{\Delta t}{m}\vec{F}(\vec{r},\vec{v},t)$$

Uzyskaliśmy w ten sposób przepis na obliczanie prędkości po kolejnych krokach symulacji. Musimy jedynie znać poprzedni wektor prędkości i przyspieszenie działające na obiekt. Stosując to samo podejście dla wzoru (2) uzyskamy:

12. $$\frac{\vec{r}(t+\Delta t)-\vec{r}(t)}{\Delta t}=\vec{v}(t)$$

Zatem położenie w kolejnym kroku czasowym jest następujące:

13. $$\vec{r}(t+\Delta t)=\vec{r}(t)+\Delta t\,\vec{v}(t)$$

We wzorach (11) i (13) położenie i prędkość w chwili *t* są znane z poprzedniego kroku symulacji. Metoda Eulera wymaga wobec tego, abyśmy dla każdego obiektu przechowywali dwa wektory: jego położenie i prędkość. To niewiele, zaledwie sześć liczb dla każdego punktu materialnego w symulowanym zbiorze. Ponadto widać, że wzory (11) i (13) są proste do implementacji i niekosztowne obliczeniowo.

I wszystko byłoby pięknie, gdyby nie fakt, iż stosując ten przepis nie uzyskamy dobrych wyników. Metoda Eulera jest bowiem zawsze rozbieżna. Możemy ją stosować tylko jeżeli symulacja nie będzie zbyt długa lub obiekt jest dodatkowo kontrolowany przez gracza, co wymusza okresowe zmiany położenia obiektu i tym samym dzieli symulację na krótkie fragmenty z wymuszanymi warunkami początkowymi. Możemy się oczywiście starać poprawić wyniki zmniejszając krok czasowy. To jednak prowadzi do zwiększenia ilości kroków, co oznacza zwiększenie kosztu obliczeń, a przy tym narastanie błędów zaokrągleń, które mogą istotnie wpływać na wyniki. Poza tym żadne zmniejszenie kroku czasowego nie zmusi metody Eulera do dobrego działania, wydłuży jedynie okres, w którym błędy będą akceptowalne.

Jednak metodę Eulera można poprawić i to stosunkowo łatwo. Wystarczy we wzorze (13) zastąpić prędkość z chwili *t* prędkością z chwili $t+\Delta t$ tzn. uzyskanej w tym samym kroku symulacji ze wzoru (11):

14. $$\vec{r}(t+\Delta t)=\vec{r}(t)+\Delta t\,\vec{v}(t+\Delta t)$$

Metoda ta nazywana jest zmodyfikowaną lub poprawioną metodą Eulera, metodą Eulera pół-implicite, metodą Eulera-Cromera lub metodą NSV. Formalnie nie zwiększa ona dokładności obliczeń (nadal zachowywane są tylko wyrazy z Δt^1 – por. ramka o szeregu Taylora), ale w odróżnieniu od podstawowej metody Eulera lepiej zachowuje energię układu.

IMPLEMENTACJA

Implementacja metody Eulera nie jest trudna. Wymaga raptem dwóch linii kodu. Warto jednak przygotować klasę, która przechowuje parametry opisujące obiekt (są to pola przechowujące masę, położenie i prędkość, a także pola pomocnicze – następne położenie i następna prędkość), a także ułatwia symulacje dla wielu punktów. Pola przykładowego szablonu klasy o nazwie `TPunktMaterialny<T>` napisanego w C++ widoczne są na Listingu 1. Do przechowywania wszystkich wektorów używany jest szablon struktury `TWektor<T>`, który dostępny jest w projekcie dołączonym do artykułu (do pobrania ze strony WWW magazynu) . Metoda Eulera zaimplementowana jest w bardzo prostej metodzie `TPunktMaterialny<T>::PrzygotujRuch_Euler` widocznej na Listingu 2. Każdemu całkowaniu odpowiada zaledwie jedna linijka jej kodu. Na tym samym listingu pokazana jest zarówno podstawowa wersja tej metody, jak i poprawiona. Bezwzględnie należy korzystać z tej drugiej.

Listing 1. Pola szablonu klasy opisującej obiekt

```
template<typename T> class TPunktMaterialny
{
    private:
        TWektor<T> polozenie;
        TWektor<T> predkosc;
        TWektor<T> poprzedniePolozenie;
        TWektor<T> nastepnePolozenie;
        TWektor<T> nastepnaPredkosc;
        T masa;
        ...
```

Autor

Kluczowym dla zrozumienia większości metod numerycznych jest pojęcie szeregu Taylora. Pozwala on na przedstawienie funkcji wokół interesującego nas punktu w postaci wielomianu. Współczynniki tego wielomianu są pochodnymi oryginalnej funkcji. Jeżeli Czytelnik nie zna tego pojęcia, to jest dobra okazja, aby je poznać. Dobrym źródłem wiedzy na jego temat, oczywiście poza podręcznikami do analizy matematycznej, jest choćby Wikipedia (http://pl.wikipedia.org/wiki/Wz%C3%B3r_Taylora), jeszcze lepszym witryna MathWorld (http://mathworld.wolfram.com/TaylorSeries.html).

Na nasze potrzeby wystarczy powiedzieć, że wartość funkcji $f(t)$ w dowolnym punkcie t_1 można przedstawić jako nieskończony szereg o następującej postaci:

$$f(t_1)=f(t)+\frac{\dot{f}(t)}{1!}(t_1-t)^1+\frac{\ddot{f}(t)}{2!}(t_1-t)^2+\ldots+\frac{f^{(n)}(t)}{n!}(t_1-t)^n+\ldots=\sum_{n=0}^{\infty}\frac{f^{(n)}(t)}{n!}(t_1-t)^n$$

Nazywa się to rozwinięciem wokół punktu *t*, ponieważ wartość wszystkich pochodnych obliczana jest w tym punkcie. To najbardziej typowa postać wzoru ora. Na potrzeby metod numerycznych warto go jednak nieco przeformułować, wprowadzając nowe oznaczenie $\Delta t=t_1-t$:

$$f(t+\Delta t)=f(t)+\frac{\dot{f}(t)}{1!}\Delta t^1+\frac{\ddot{f}(t)}{2!}\Delta t^2+\ldots+\frac{f^{(n)}(t)}{n!}\Delta t^n+\ldots=\sum_{n=0}^{\infty}\frac{f^{(n)}(t)}{n!}\Delta t^n$$

Wzór Taylora to sposób na obliczenie wartości funkcji w punkcie $t+\Delta t$ na podstawie wartości funkcji $f(t)$ i wszystkich jej pochodnych w punkcie t . Jeżeli ilość znanych pochodnych jest ograniczona, możemy przynajmniej obliczyć przybliżoną wartość funkcji w nowym punkcie, który w tym przypadku nie powinien być jednak zbyt odległy (Δt powinno być małe). Zauważmy, że użycie metody Eulera oznacza tyle, że pomijamy wszystkie wyrazy tego rozwinięcia poza pierwszymi dwoma – stałym i liniowym.

Listing 2. Implementacja metod Eulera, Eulera-Cromera i Verleta

```
//wersja podstawowa metody Eulera - nie należy z niej korzystać
template<typename T>
void TPunktMaterialny<T>::PrzygotujRuch_Euler(TWektor<T> przyspieszenie,T krokCzasowy)
{
   nastepnaPredkosc=predkosc+przyspieszenie*krokCzasowy;
   nastepnePolozenie=polozenie+predkosc*krokCzasowy;
}

//zmodyfikowana wersja metody Eulera
template<typename T>
void TPunktMaterialny<T>::PrzygotujRuch_Euler(TWektor<T> przyspieszenie,T krokCzasowy)
{
   nastepnaPredkosc=predkosc+przyspieszenie*krokCzasowy;
   nastepnePolozenie=polozenie+nastepnaPredkosc*krokCzasowy;
}

#define SQR(x) ((x)*(x))

template<typename T>
void TPunktMaterialny<T>::PrzygotujRuch_Verlet(TWektor<T> przyspieszenie,T krokCzasowy)
{
   if(numerKroku==0) PrzygotujRuch_Euler(przyspieszenie,krokCzasowy);
   else
   {
      nastepnePolozenie=2.0*polozenie-poprzedniePolozenie+przyspieszenie*SQR(krokCzasowy);
      nastepnaPredkosc=(nastepnePolozenie-poprzedniePolozenie)/(2*krokCzasowy);
   }
}

template<typename T>
void TPunktMaterialny<T>::PrzygotujRuch(TWektor<T> sila,T krokCzasowy,Algorytm algorytm)
{
   TWektor<T> przyspieszenie=sila/masa;
   switch(algorytm)
   {
      case algorytmEulera:
         PrzygotujRuch_Euler(przyspieszenie,krokCzasowy);
         break;
      case algorytmVerleta:
         PrzygotujRuch_Verlet(przyspieszenie,krokCzasowy);
         break;
   }
}

template<typename T>
void TPunktMaterialny<T>::WykonajRuch()
{
   poprzedniePolozenie=polozenie;
   polozenie=nastepnePolozenie;
   predkosc=nastepnaPredkosc;
   ++numerKroku;
}
```

Na stronie http://programistamag.pl/download dostępny jest projekt Visual C++ 2012, w którym użyty został szablon klasy **TPunktMaterialny<T>** oraz towarzyszący jej szablon **TZbiorPunktowMaterialnych<T>** implementujący zbiór obiektów połączonych siłami harmonicznymi z tłumieniem. W projekcie tym symulowana jest sprężysta kostka. Wyniki wyświetlane są za pomocą OpenGL (Rysunek 1). Każdy punkt reprezentowany przez sferę można złapać myszką i w ten sposób poruszać kostką. Warto zwrócić uwagę, że wiązanie kursora myszy z punktem nie jest bezpośrednie (położenia kursora nie wyznacza wprost położenie punktu), a poprzez dodatkową siłę harmoniczną. W praktyce oznacza to, że łączymy je silnie tłumioną sprężynką.

METODA VERLETA

Niezaprzeczalną zaletą metody Eulera jest jej intuicyjność - używając tej metody zakładamy po prostu, że w krótkich przedziałach czasu całkujemy stałą wartość przyspieszenia lub prędkości. Za tym idzie prostota wzorów i w konsekwencji łatwość implementacji. Niewielkie prawdopodobieństwo popełnienia błędu podczas implementacji jest nie do przecenienia. Jednak co z tego, jeżeli wyniki uzyskane w ten sposób nie są poprawne. Oczywiście nie zawsze pełna poprawność jest konieczna. Dla przykładu jeżeli modelujemy warkocz Lary Croft składający się z kilku punktów materialnych powiązanych siłami sprężystymi z tłumieniem, wystarczy gdy uzyskamy rozwiązanie, które jest wiarygodne. W takim przypadku nie potrzebna nam duża dokładność. Jednak jeżeli metodą Eulera chcemy rozwiązać równania dla kulek *Goo*, uzyskany w ten sposób wynik może być bardzo rozczarowujący.

Warto wobec tego wiedzieć, że jest inna metoda, której złożoność jest niewiele większa od metody Eulera, przy znacznie lepszej jakości otrzymywanych rozwiązań. Zawdzięczamy ją dwóm fizykom: Carlowi Størmerowi, który na początku XX wieku używał jej na papierze oraz Loupowi Verletowi, który w drugiej połowie XX wieku, już używając komputerów, zastosował ją na potrzeby dynamiki molekularnej. Dynamika molekularna to część biofizyki, w którym sporych rozmiarów cząsteczki (np. białka) bada się metodami mechaniki klasycznej tj. traktując poszczególne atomy jak kulki połączone oddziaływaniami harmonicznymi. Innymi słowy dla poszczególnych atomów rozwiązuje się równanie (3) z siłami, które są rozbudowanymi wersjami siły (5). Podobnie, jak w przypadku gier, w których liczy się przede wszystkim finalny obraz, tak i w dynamice molekularnej interesujące są przede wszystkim położenia symulowanych obiektów. Prędkości to tylko informacja uzupełniająca. Metoda Verleta pozwala na obliczanie kolejnych położeń z pominięciem prędkości:

$$15.\quad \vec{r}(t+\Delta t) \approx -\vec{r}(t-\Delta t) + 2\vec{r}(t) + \ddot{\vec{r}}(t)\Delta t^2$$

gdzie druga pochodna położenia to nic innego jak przyspieszenie, czyli siła podzielona przez masę. Jak widać z powyższego wzoru do obliczenia kolejnego położenia już nie potrzebujemy prędkości. Zamiast niej musimy jednak przechowywać położenie nie tylko z poprzedniego kroku czasowego, ale również z jeszcze wcześniejszego.

Skąd wziął się ten wzór? Najprościej wyprowadzić go korzystając z opisanego w ramce szeregu Taylora. Jak już wiemy pozwala on znaleźć położenie w chwili $t+\Delta t$ o ile znamy prędkość, przyspieszenie i wyższe pochodne położenia w chwili *t*:

$$16.\quad \vec{r}(t+\Delta t) \approx \vec{r}(t) + \frac{\dot{\vec{r}}(t)}{1!}\Delta t^1 + \frac{\ddot{\vec{r}}(t)}{2!}\Delta t^2 + \frac{\dddot{\vec{r}}(t)}{3!}\Delta t^3 + \frac{\vec{r}^{(4)}(t)}{4!}\Delta t^4$$

Zwróćmy uwagę, że równie dobrze można znaleźć położenie dla chwili wcześniejszej, czyli dla $t-\Delta t$:

$$17.\quad \vec{r}(t-\Delta t) \approx \vec{r}(t) - \frac{\dot{\vec{r}}(t)}{1!}\Delta t^1 + \frac{\ddot{\vec{r}}(t)}{2!}\Delta t^2 - \frac{\dddot{\vec{r}}(t)}{3!}\Delta t^3 + \frac{\vec{r}^{(4)}(t)}{4!}\Delta t^4$$

Dodając te dwa rozwinięcia stronami i porządkując uzyskane wyrazy otrzymamy przepis na następne położenie, czyli wzór (18). Jak widać złożoność obliczeniowa tego wzoru nie jest znacznie większa od złożoności dwóch razem wziętych wzorów (11) i (13) (lub (14)), a dokładność rośnie ogromnie. Pomijane są bowiem dopiero wyrazy z Δt^4 , podczas gdy w metodzie Eulera (także w tej zmodyfikowanej) pomijane są już wyrazy z Δt^2 . Implementacja wzoru (18) widoczna jest na Listingu 2 w metodzie `PrzygotujRuch_Verlet`.

Skoro algorytm Verleta korzysta z dwóch punktów czasowych do obliczenia następnego, to pojawia się problem w pierwszym kroku symulacji. Użycie tego algorytmu nie jest wówczas możliwe, bo niezdefiniowana jest jeszcze wartość pola `poprzedniePolozenie` (zob. Listing 2). Najprostsze co możemy zrobić, to w momencie tworzenia obiektu zainicjować to pole wartością identyczną jak `polozenie`. Wówczas w pierwszym kroku otrzymamy taki sam wynik jak korzystając z algorytmu Eulera. Równoważnie w pierwszym kroku możemy jawnie użyć metody implementującej algorytm Eulera. W metodzie z Listingu 2 korzystamy właśnie z tego rozwiązania.

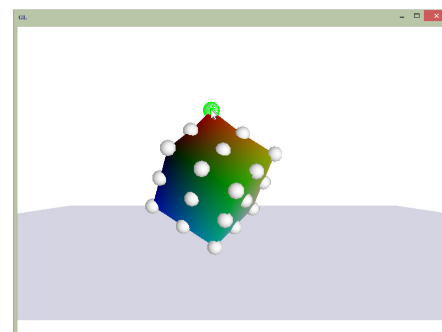
WERYFIKACJA ROZWIĄZAŃ, CZYLI CAŁKI RUCHU

Jak zweryfikować poprawność uzyskanych w symulacji numerycznej rozwiązań równań ruchu? Dobrą metodą jest kontrolowanie wielkości, o której wiemy, że powinna być stała (całki ruchu) i sprawdzenie w jakim zakresie się ona zmienia. O tym, jakie wielkości powinny pozostawać stałe, mówią zasady zachowania będące fundamentem fizyki. W przypadku oscylatora harmonicznego bez tłumienia, który opisuje siła (5), jak również w przypadku spadku ciała w polu grawitacyjnym (siła (4)), obowiązuje zasada zachowania energii mechanicznej. Oznacza to, że suma energii kinetycznej i potencjalnej powinna być stała i równa wartości z chwili początkowej. W przypadku oscylatora oznacza to, że stała powinna być równa:

$$18.\quad mv^2(t) + kx^2(t) = mv_0^2 + kx_0^2 = const$$

(energia została pomnożona przez 2). Powinniśmy tak dobrać krok czasowy, aby wielkość ta była zachowana. Nie jest zaskoczeniem, że krok czasowy będzie mógł być większy w przypadku metody Verleta niż Eulera. W tym drugim przypadku zmiana wartości całek ruchu jest zresztą nieunikniona.

Więcej na temat całkowania równań ruchu, także na potrzeby gier, znajdzie czytelnik w książce: *Grafika. Fizyka. Metody numeryczne. Symulacje fizyczne z wizualizacją 3D*, Jacek Matulewski, Tomasz Dziubak, Marcin Sylwestrzak, Radosław Płoszajczak, Wydawnictwo Naukowe PWN 2010.



Rysunek 1. Przykładowa symulacja korzystająca z metody Verleta

Jacek Matulewski

Fizyk zajmujący się na co dzień optyką kwantową i układami nieuporządkowanymi na Wydziale Fizyki, Astronomii i Informatyki Stosowanej UMK w Toruniu. Od 1998 r. interesuje się programowaniem dla systemu Windows, w szczególności platformą .NET i językiem C#. Autor serii książek poświęconych programowaniu. Większość ukazała się nakładem wydawnictwa Helion. Wierny użytkownik kupionego w połowie lat osiemdziesiątych "komputera osobistego" ZX Spectrum 48k.