Jak należy pisać aktualizację stanu gry.

Czytelniku, jeżeli napisałeś w swoim życiu jakąkolwiek grę, to zapewne jesteś świadom fundamentalnej koncepcji programowania gier, jaką jest pętla główna aplikacji[1]. Być może więc widziałeś kod pisany podobnie jak poniższy:

```
float dt = 0.0f;                                    //czas od ostatniej aktualizacji
float lastUpdateTime = GetCurrentTime();    //czas ostatniej aktualizacji
                                                    //przykladowa funkcja GetCurrentTime() pobiera
                                                    //nam od systemu aktualny czas w sekundach

while(true)
{
        dt = GetCurrentTime() - lastUpdateTime;     //obliczenie czasu od ostatniej klatki
        lastUpdate += dt;                           //podmiana

        GrabInput();            //<-- zbieranie wejscia z klawiatury, myszki, sieci, itp.
        UpdateGame(dt);                 //<-- aktualizacja fizyki i logiki gry
        RenderGame();           //<-- wyswietlenie aktualnego stanu na ekranie
}
```

Wszyscy pisaliśmy kiedyś taki kod. Jeśli i Ty tak piszesz, to przestań. W tym krótkim artykule chciałbym przekonać Cię, że taka pętla główna jest zła, Na szczęście niewiele trzeba, by ją naprawić.

Głównym problemem powyższego kodu jest sposób, w jaki aktualizuje on fizykę i logikę gry. Po wyliczeniu czasu od ostatniej aktualizacji (co jest właściwą rzeczą) uruchamia on aktualizację gry z obliczonym krokiem czasowym (wartość dt). Taki sposób pisania pętli nazywamy **metodą zmiennokrokową** – wyliczony krok zależy od tego, ile trwały obliczenia w poprzedniej klatce i przez to będzie zmieniał się z każdym obiegiem. Jeśli z jakiegoś powodu rysowanie potrwa dłużej lub włączy się Garbage Collector (hello, Java), to wartość dt w kolejnym obiegu pętli będzie się znacząco różniła od poprzedniej.

Zdecydowanie lepszym podejściem jest tak zwana **metoda stałokrokowa** – odpowiedni, poprawiony kod pętli głównej zamieszczam niżej:

```
float dt = 0.0f;                                    //czas od ostatniej aktualizacji
float lastUpdateTime = GetCurrentTime();    //czas ostatniej aktualizacji
                                                    //przykladowa funkcja GetCurrentTime() pobiera
                                                    //nam od systemu aktualny czas w sekundach
float accumulator = 0.0f;
const float TIME_STEP = 0.03;               //krok czasowy, a zarazem czas trwania ramki
                                                    //fizyki w sekundach; tutaj 30 milisekund, czyli
                                                    //ok. 30 aktualizacji na sekundę

while(true)
{
```

[1] Temat ten dobrze omówił Xion w artykule na swoim blogu - http://xion.org.pl/productions/texts/coding/game-programming/real-time-loop/

```
        dt = GetCurrentTime() - lastUpdateTime;    //obliczenie czasu od ostatniej klatki
        lastUpdate += dt;                  //podmiana
        accumulator += dt;

        GrabInput();          //<-- zbieranie wejscia z klawiatury, myszki, sieci, itp.
        while(accumulator > TIME_STEP)
        {
               UpdateGame(TIME_STEP);              //<-- aktualizacja fizyki i logiki gry
               accumulator -= TIME_STEP;
        }
        RenderGame();         //<-- wyswietlenie aktualnego stanu na ekranie
}
```

Zasadnicza idea obydwu przykładów jest taka sama – w nieskończonej pętli głównej wykonujemy aktualizację logiki gry oraz rendering grafiki. Różnica polega na tym, że w drugim przypadku krok czasowy dla aktualizacji gry jest zawsze stały i wynosi dokładnie TIME_STEP. Podobnie jak poprzednio liczymy „deltę czasu" (parametr dt), czyli czas od poprzedniej klatki. Tutaj jednak wykorzystujemy ją do określenia ile (jeśli w ogóle) aktualizacji gry ze stałym krokiem czasowym należy wykonać, by „nadgonić" czas rzeczywisty – na przykład, jeżeli od poprzedniej klatki minęło 15 milisekund, to nie aktualizujemy gry w ogóle; jeżeli między klatkami minęły 62 milisekundy, to UpdateGame() wykona się dwa razy przed renderingiem.

Zapytasz – „ale po co mi to?". Cóż, jeżeli chciałbyś umieścić w swojej grze replay'e, cofanie czasu, dokładną fizykę lub wykonywane co jakiś czas trudne obliczenia, albo po prostu zależy Ci na tym, by w Twoją produkcję grały też osoby ze sprzętem ledwo spełniającym minimalne wymagania gry, to właściwy sposób aktualizacji gry jest **obowiązkowy**! Oto, co daje nam symulacja stałokrokowa:

- Stabilność fizyki,
- Możliwość polegania na kodzie fizycznym,
- Przewidywalność gry (determinizm).

Poniżej chciałbym rozwinąć te myśli.

Wybuchowa fizyka

Pisanie fizyki zmiennokrokowej to proszenie się o kłopoty. Całkowanie Eulera, podstawowy (wbrew strasznie brzmiącej nazwy) sposób na obliczanie fizyki w grach jest sam w sobie niestabilny; dodanie mu zmiennego kroku czasowego pomaga jeszcze szybciej eksplodować. Właśnie… na czym polega „eksplozja" fizyki?

Rozważmy model prostej sprężyny[2]. W szkole podawane jest prawo Hooke'a, które głosi, że siła działająca na ciało zaczepione na sprężynę wynosi w danej chwili czasu:

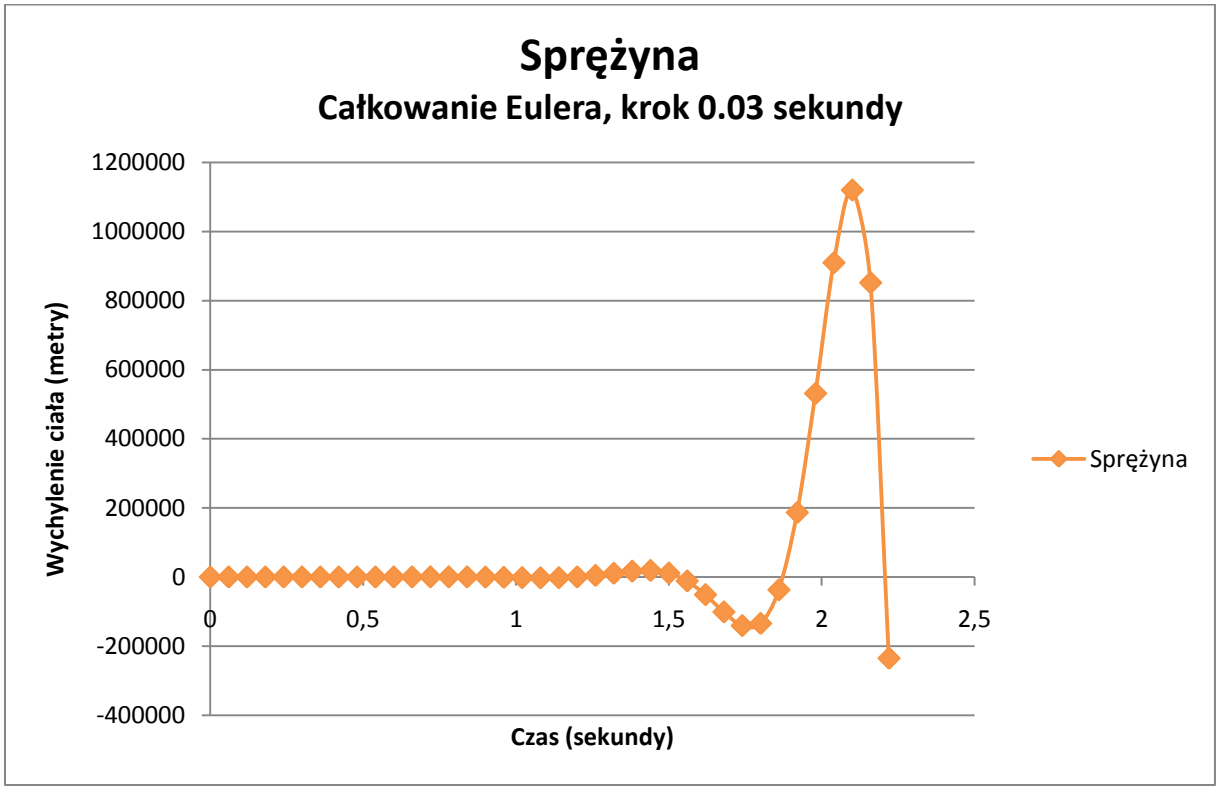
$$F = -kx$$

Współczynnik k, zwany stałą sprężystości to pewna liczba mówiąca o tym, czy sprężyna jest miękka jak ta z długopisu, czy twarda jak amortyzator rowerowy. Czynnik x to tak zwane wydłużenie sprężyny, czyli odległość między jej początkiem a końcem. Wzór ten mówi nam, że siła jest

[2] Rozważana sprężyna ma zerową długość – ciało zaczepione na niej jest w spoczynku wtedy, gdy znajduje się w punkcie zaczepienia sprężyny. Ogólny wzór, uwzględniający spoczynkową długość sprężyny l, wygląda następująco: $F = -k(x - l)$.

proporcjonalna do wydłużenia sprężyny i działa w kierunku jej punktu równowagi – rozciągnięta lub ściśnięta sprężyna chce wrócić do stanu, jaki miała na początku.

Wyobraźmy sobie teraz, że symulujemy taką sprężynę z zaczepionym na końcu ciałem. Jeżeli naciągniemy ją wystarczająco mocno, to siła działająca na jej koniec będzie tak duża, że w ciągu jednego kroku czasowego zawieszone ciało „przeskoczy" na jej drugą stronę. Tam, w kolejnym kroku symulacji, na ciało zadziała nowa siła, która każe mu wrócić. Teraz wyobraźmy sobie co by było, gdyby taka siła spowodowała w kolejnych dwóch – trzech krokach przeskoczenie takiego ciała na drugą stronę sprężyny, ale na odległość większą niż miało dotąd. W następnych krokach działająca siła zawróci je jeszcze silniej. Ciało wyleci jeszcze dalej od punktu równowagi, co spowoduje działanie jeszcze mocniejszej siły zawracającej… Zwykle po kilkudziesięciu krokach symulacji ciało oddala się do (komputerowej) nieskończoności. Taki efekt nazywamy „eksplozją" modelu fizycznego.



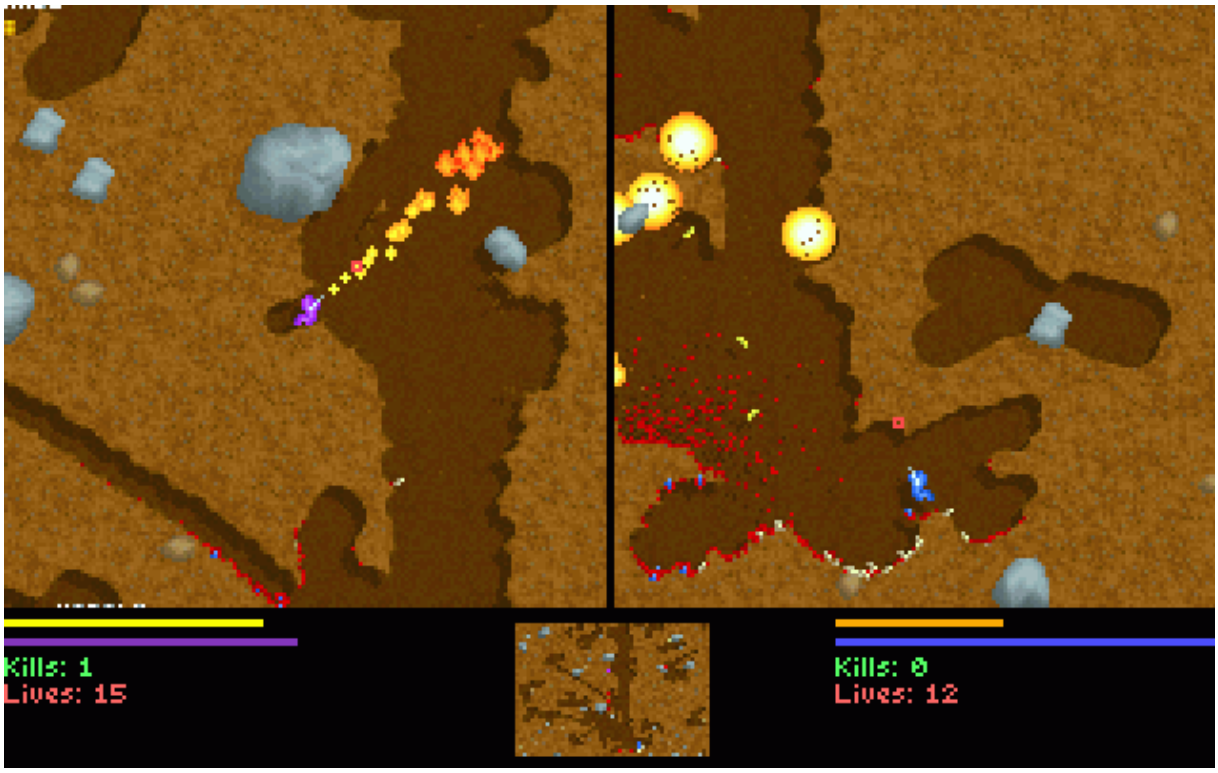
Rysunek 1 - Eksplozja modelu sprężyny. Na sprężynie zawieszono ciężarek. Wykres przedstawia wychylenie ciężarka w kolejnych krokach symulacji stałokrokowej. Należy zauważyć, że przy tej skali wykresu drgania sprężyny w pierwszej sekundzie symulacji są praktycznie niewidoczne.

Należy podkreślić, że możliwość eksplozji systemu fizycznego jest nie do uniknięcia – wynika ona z tego, że posiadamy krok czasowy; znikłaby, gdybyśmy umieli liczyć w nieskończenie małych odstępach czasu. Da się jednak nad nią zapanować pilnując, by siły i współczynniki były odpowiednio małe. Jednak w sytuacji, gdy stosujemy naiwną metodę zmiennokrokową, to o jakiejkolwiek kontroli nie ma mowy. Jeżeli z jakiegoś powodu (na przykład chwilowego obciążenia systemu, zrzucania pamięci na dysk czy uruchomienia się Garbage Collectora w Javie) krok czasowy zamiast ok. 30ms nagle wyniesie np. 500ms, to mamy praktycznie po symulacji. Taka mała „czkawka" wystarczy, by szybko „bujający się" obiekt na sprężynie wyleciał niebezpiecznie daleko poza punkt równowagi i uruchomił eksplozję systemu. Nigdy nie wiemy, kiedy to się stanie – takie zjawisko jest całkowicie zależne od obciążenia systemu naszego gracza. Dla odmiany, kod stałokrokowy jest zupełnie

niewrażliwy na powyższy problem – w przypadku długiej klatki, symulacja fizyczna zostanie po prostu wykonana kilka razy, by „nadgonić” czas rzeczywisty.

Fizyka, na której można polegać

Jako gracz, zapewne widziałeś taką sytuację wiele razy – szybko poruszająca się piłka magicznie „przelatywała” przez klocki, w które miała uderzyć. Pociski czasem przenikały przez ściany. Spadające przedmioty potrafiły wylecieć za planszę. Osobom, które nie widziały czegoś takiego wcześniej, to polecam zagrać w Liero – tam bronie odpychające gracza wystarczająco mocno były używane do teleportowania go w losowe miejsce planszy.



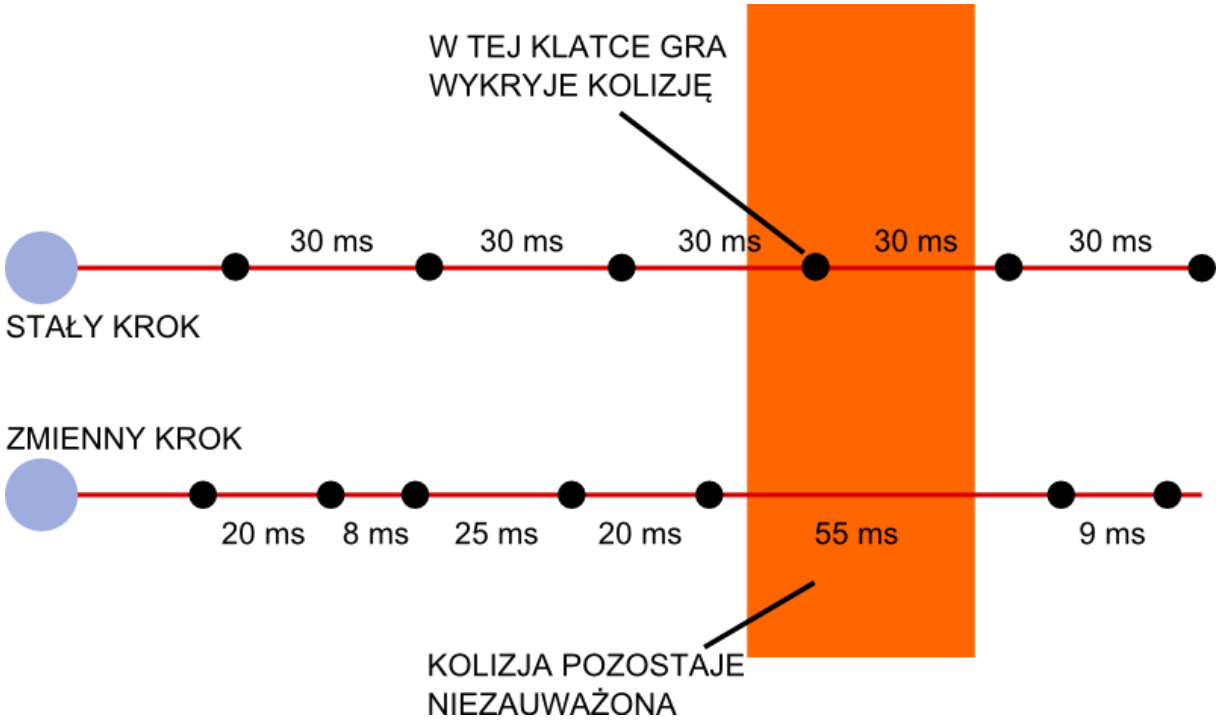
Rysunek 2 - Gra Liero znana ze stworzonych przez fanów broni, które oszukując system kolizji wykonywały teleportację gracza. Screen pochodzi ze strony http://www.datarealms.com/stuff/games/liero.gif.

Przyczyna tego problemu jest bardzo prosta – typowy kod wykrywania zderzeń sprawdza, czy dwa obiekty się przecinają w aktualnej klatce[3]. Jeżeli tak, sygnalizuje kolizję. Jednak odpowiednio szybka piłka przy odpowiednio dużym kroku czasowym może przebyć w pojedynczej „klatce” symulacji odległość większą, niż szerokość klocka. W tej sytuacji kod wykrywania zderzeń nie zauważy, że obiekty zderzyły się „w międzyczasie”. Piłka magicznie teleportuje się za klocek. Taka sytuacja może być wbrew pozorom częstsza niż się wydaje. Oto krótkie wyliczenie:

Typowa ilość klatek na sekundę to 30. Czas pojedynczej klatki wynosi więc 1/30 sekundy. Załóżmy, że gramy w rozdzielczości 800x600. Wyobraźmy sobie małą piłkę, która leci na tyle szybko, że potrafi dotrzeć z góry ekranu na sam dół w ciągu jednej sekundy. Porusza się więc ona z szybkością 600

[3] W szczególności, typowy kod nie sprawdza czy dwa obiekty się przecięły od ostatniej klatki – sprawdza jedynie, czy w aktualnej klatce się nie przecinają. Ten pierwszy typ sprawdzeń jest dużo trudniejszy do napisania.

pikseli na sekundę. **W ciągu pojedynczej klatki przeleci ona więc 1/30 s * 600 px/s = 20px**. To dość sporo. To więcej niż rozmiar ikony w tray'u (pasku powiadomień systemu Windows), to dwa razy tyle co wysokość zegara systemowego w Windows – 20 pikseli to na tyle duża odległość, że taka piłka może bez problemu przeskoczyć cienkie murki. Nawet jeśli dołożymy do tego wymiary piłeczki (powiedzmy 10 x 10 pikseli), to wciąż będzie mogła ona niezauważenie przeskoczyć przez obiekty grubości 10 pikseli (zegara systemowego) – na przykład pociski, małe przeszkody.



Rysunek 3 - Różnice w wykrywalności zderzeń między stałokrokową i zmiennokrokową symulacją.

Ponieważ w symulacji zmiennokrokowej czas trwania pojedynczej klatki może wynosić, zależnie od humoru, 1/100 sekundy, 1/15 sekundy lub nawet i pół sekundy, to jesteśmy „ugotowani" – sprawność wykrywania kolizji zależy w dużym stopniu od tego, czy gracz nie uruchomił sobie w tle cięższych programów.

Chociaż efekty przeskakiwania oraz eksplozji fizyki występują także w symulacjach stałokrokowych, zasadnicza różnica polega na tym, że w przypadku stałego kroku **możemy te efekty przewidywać i kontrolować**. Na przykład, jeśli wiemy, że najcieńszy obiekt w naszej grze ma grubość 10 jednostek, to maksymalna bezpieczna szybkość dla dowolnego (tj. niezależnie od jego rozmiaru) obiektu w grze wynosi: 10 / dt. Wstawiając krok czasowy równy przykładowo 1/30 sekundy uzyskamy szybkość 300 jednostek na sekundę. Obiekty poruszające się szybciej mają sporą szansę na „przenikanie przez ściany". Ponieważ wyliczyliśmy maksymalną dopuszczalną prędkość, możemy upewnić się, że nie zostanie ona nigdy przekroczona – w ten sposób wyeliminujemy problemy z kolizjami w naszej grze. W podobny sposób możemy zapewnić, że fizyka nigdy nie eksploduje – dobierając odpowiednie współczynniki sprężystości oraz górne ograniczenia na siły. Podobnych obliczeń nie moglibyśmy wykonać dla zmiennego kroku czasowego.

Przewidywalność gry

Fizyka klasyczna mówi, że jeżeli znamy stan początkowy (położenie, prędkość, masa) danego ciała oraz dokładnie znamy wszystkie działające na nie siły to możemy przewidzieć stan tego ciała w dowolnym późniejszym momencie czasu[4]. Podobnie może być w naszej grze – jeżeli zachowamy stały krok czasowy, to nasza symulacja fizyczna będzie się wykonywała **za każdym razem dokładnie w taki sam sposób** – niezależnie od obciążenia komputera. Ta właściwość jest fundamentalna dla pewnych „fajnych" rzeczy w grach, takich jak replay'e czy metody „cofania czasu".

Przewidywalność (inaczej determinizm) gry jest tak istotna dla replayów, ponieważ pozwala nam zapisywać jedynie wejście od gracza (klawiatura, mysz, sieć, itp.) oraz początkowe ustawienie generatorów liczb losowych[5] – mając te informacje, możemy dokładnie odtworzyć całe zachowanie gry. Byłoby to niemożliwe w przypadku zastosowania zmiennego kroku czasowego.

Przewidywalność gry pomaga także w wyszukiwaniu błędów – daje nam gwarancję, że kod symulacji zawsze wykona się tak samo w kolejnych próbach. Jest to bardzo istotne, ponieważ w przypadku metody zmiennokrokowej pewna część błędu może pochodzić z samego faktu, że krok czasowy był chwilami za długi lub za krótki.

Mity o symulacji stałokrokowej

Ostatnio słyszałem kilka ciekawych (i uważam, że nieuzasadnionych) uwag skierowanych przeciwko metodzie stałokrokowej.

„Metoda jest skomplikowana"

Mam nadzieję, że początek tego artykułu przekonał Cię, Czytelniku, że napisanie symulacji stałokrokowej to w stosunku do symulacji zmiennokrokowej tylko kilka dodatkowych linii kodu w głównej pętli. Sama idea jest bardzo prosta i zupełnie niewidoczna dla aktualizowanego kodu gry.

„Metoda tworzy sztuczne powiązanie między fizyką a logiką"

Chociaż głównym zadaniem symulacji stałokrokowej jest uzyskanie dokładniejszej i stabilniejszej fizyki, to nie należy zapominać, iż **jest ona istotna dla „logicznej" części gry** – na przykład do wspomnianego już zapisywania replayów czy mechanizmów cofania się w czasie. Stały krok w symulacji **nie jest** po prostu 'kwestią implementacji fizyki'. Można go więc użyć także do aktualizacji logiki. Z resztą, na swój sposób logika i fizyka w grze są ze sobą bardzo związane.

---

[4] Przestraszonym etycznymi i filozoficznymi konsekwencjami tego faktu śpieszę przypomnieć, że mechanika kwantowa pokazała, iż świat nie jest w ten sposób przewidywalny – w odpowiednio małej skali (na poziomie pojedynczych atomów) wszystkim rządzi prawdopodobieństwo.

[5] Tak zwane „generatory liczb losowych" stosowane w programowaniu tak naprawdę są w pełni przewidywalne – wykonują one jedynie bardzo wymyślne obliczenia matematyczne na poprzednio wyliczonej wartości tak, by nowa wartość była zupełnie inna, a kolejne wartości nie były ze sobą w widoczny sposób związane. Znając więc początkowe ustawienie generatora (tzw. seed), możemy przewidzieć kolejne wyliczone wartości. Właściwa nazwa na takie narzędzia to „generatory liczb pseudolosowych". Istnieją jednak generatory liczb prawdziwie losowych (np. HotBits - http://www.fourmilab.ch/hotbits/), oparte o zjawiska kwantowe, które z natury są procesami losowymi.

Jeszcze jedna poprawka…

Jest jeszcze jedna rzecz, którą możemy poprawić w kodzie pętli głównej. Otóż w pewnych specyficznych sytuacjach może się zdarzyć, że wyliczony czas od ostatniej klatki będzie ujemny[6]. Oprócz tego, jeżeli z jakiegoś powodu aktualizacja stanu gry oraz rendering zaczynają zabierać bardzo dużo czasu, to gra może nie nadążać z nadganianiem czasu rzeczywistego – w każdym kolejnym obiegu pętli głównej wykonywać się będzie coraz więcej kroków aktualizacji[7]. Musimy więc ograniczyć nasz czas z dołu i z góry. Poniżej prezentuję ostateczny przykład kodu:

```
float dt = 0.0f;                                    //czas od ostatniej aktualizacji
float lastUpdateTime = GetCurrentTime();    //czas ostatniej aktualizacji
                                                    //przykladowa funkcja GetCurrentTime() pobiera
                                                    //nam od systemu aktualny czas w sekundach
float accumulator = 0.0f;
const float TIME_STEP = 0.03;                       //krok czasowy, a zarazem czas trwania ramki
                                                    //fizyki w sekundach; tutaj 30 milisekund, czyli
                                                    //ok. 30 aktualizacji na sekundę
const float MAX_ACCUMULATED_TIME = 1.0;     //maksymalny czas zgromadzony w pojedynczym
                                                    //obiegu petli glownej

while(true)
{
        dt = GetCurrentTime() - lastUpdateTime;    //obliczenie czasu od ostatniej klatki
        lastUpdate += dt;                          //podmiana
        dt = std::max(0, dt);                      //upewniamy sie, ze dt >= 0
        accumulator += dt;
        accumulator = clamp(accumulator, 0, MAX_ACCUMULATED_TIME);      //zapobiegamy
                                                    //zbyt duzej ilosci aktualizacji w danym obiegu
                                                    //petli glownej


        GrabInput();          //<-- zbieranie wejscia z klawiatury, myszki, sieci, itp.
        while(accumulator > TIME_STEP)
        {
                UpdateGame(TIME_STEP);             //<-- aktualizacja fizyki i logiki gry
                accumulator -= TIME_STEP;
        }
        RenderGame();         //<-- wyswietlenie aktualnego stanu na ekranie
}
```

Pogrubieniem zaznaczyłem trzy nowe linie kodu – druga z nich zapewnia, że obliczony czas między klatkami jest nieujemny. Dzięki trzeciej ograniczamy sobie maksymalny zebrany w akumulatorze czas do jednej sekundy – mamy w tym momencie co najwyżej 1 FPS wyświetlania, ale teoretycznie gra dalej chodzi poprawnie!

Wyjaśnienia wymaga jedynie funkcja **clamp()** – jej zadaniem jest „obcięcie” pierwszego parametru do przedziału ustalonego przez drugi i trzeci argument. Innymi słowy, jeżeli pierwszy parametr leży poza przedziałem <a; b>, to zwracany jest jeden z końców przedziału; w przeciwnym wypadku zwracana

[6] Może się tak na przykład zdarzyć, gdy system „zawinie” swój wewnętrzny licznik czasu i zacznie liczyć go od nowa. Jest to całkiem normalne zjawisko.

[7] Poniekąd przypomina to eksplozję układu fizycznego – zamiast ciała zawieszonego na sprężynie uciekającego do nieskończoności mamy liczbę aktualizacji do zrobienia w pojedynczym obiegu pętli głównej; liczba ta zaczyna zmierzać do wartości nieskończonej.

jest sprawdzany parametr. Jest to bardzo przydatny fragment kodu, który powinien znaleźć się w osobistej biblioteczce matematycznej. Można tę funkcję napisać na przykład tak:

```
template <typename T>
T clamp(const T& what, const T& a, const T& b)
{
	return std::min(b, std::max(what, a));
}
```

Mam nadzieję, że przekonałem Cię, Czytelniku, iż właściwe napisanie aktualizacji gry w pętli głównej aplikacji nie jest niczym strasznym. Ostatecznie dopisaliśmy jedynie kilka linii kodu, jednak osiągnęliśmy w ten sposób trzy ważne rzeczy: stabilność fizyki, możliwość polegania na kodzie fizycznym oraz przewidywalność (czyli determinizm) gry. Życzę więc, by poprawnie stworzona symulacja stałokrokowa dała Ci możliwość pisania ciekawych gier opartych o fizykę lub nowatorski gameplay.

Jacek Złydach, TeMPOraL
http://temporal.pr0.pl/devblog
Wersja oryginalna: 17.04.2010
Ostatnie poprawki: 23.04.2010