

# RPG na Androida - ćwiczenia

## Część 1

21.05.2019

## 1 Game Loop - podstawy

### 1.1 Istota oraz funkcje

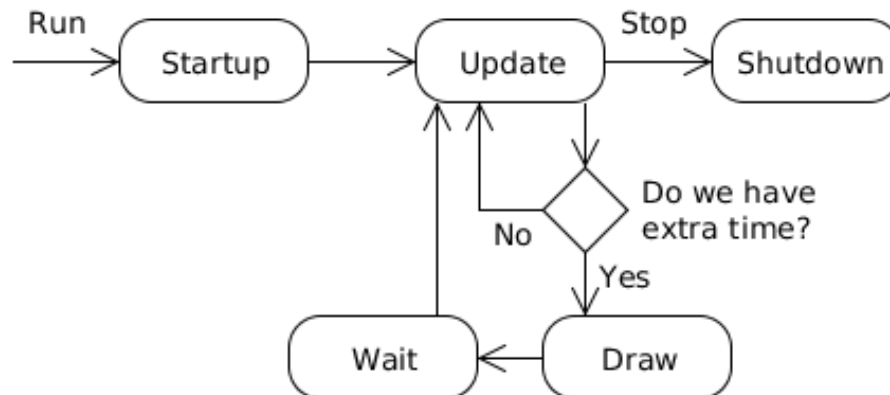
Game loop, czyli podstawowy element silnika gry komputerowej.

Każda gra realizuje trzy podstawowe funkcje:

- pobranie danych od użytkownika
- aktualizacja stanu gry
- renderowanie świata gry

Ze względu na to, że wszystkie te operacje muszą się cyklicznie powtarzać to programując zamykamy je w pętli i ta właśnie pętla nazywana jest Game Loop-em.

Podstawowy schemat takiej pętli wygląda następująco:



Natomiast do najprostszej implementacji omawianej pętli potrzeba zaledwie 4 linijek kodu...

```
while (running) {  
    update();  
    render();  
}
```

- boolean running - pozwala nam kontrolować czy np. gra nie została wstrzymana i nie chcemy przestać renderować animacji które działają w tle
- metoda update() odpowiada za aktualizację stanu gry
- metoda render() odpowiada za renderowanie świata

Można uznać, że taka prosta pętla realizuje swoje zadanie, ale... jest z nią jeden bardzo istotny problem - czas. Działa ona tak szybko jak tylko może i jej aktualnym ograniczeniem jest wyłącznie platforma sprzętowa na której została uruchomiana. Prowadzi to do niedopuszczalnej sytuacji gdy ta sama gra działa szybciej lub wolniej zależnie od sprzętu na którym została uruchomiana. Przykładowo w grze rpg te osoby które mają szybszy komputer będą poruszały się szybciej po mapie niż te dysponujące nieco gorszym sprzętem.

Naszym celem przy implementacji Game Loop-a jest osiągnięcie stałej szybkości gry, ale przy jednoczesnym zmaksymalizowaniu szybkości renderowania świata gry - w końcu mając lepszy sprzęt oczekujemy płynniejszego obrazu.

W tym momencie warto wprowadzić dwa pojęcia:

- UPS (Updates Per Second) - definiuje nam szybkość gry, czyli to jak często aktualizowane są zmienne odpowiedzialne na przykład za położenie gracza na mapie, detekcje kolizji itp.
- FPS (Frames Per Second) - myślę, że większość osób się z tym spotkała, jest to ilość klatek na sekundę czyli to jak często obraz gry jest renderowany (render())

Istnieje wiele implementacji omawianej pętli, zależnie od aktualnych potrzeb możemy się zdecydować na jedną z nich. Poniżej pokrótce postaram się przedstawić kilka wybranych.

## 1.2 Stała szybkość gry

Naszym głównym problem w najbardziej podstawowej wersji Game Loop-a było to, że zależnie od sprzętu działała z różną szybkością, a konkretniej funkcja `update()`, która odpowiada za aktualizowanie stanu rozgrywki była wywoływana z niezdefiniowaną szybkością. Przykład rozwiązania tego problemu wygląda następująco:

```
36 public void run() {
37     long FPS = 60;
38     long sleepTime = 1000L/FPS;    //1000ms = 1s
39     while (running) {
40         update();
41         render();
42         try {
43             Thread.sleep(sleepTime);    //sleep(long millis)
44         } catch (InterruptedException e) {
45             e.printStackTrace();
46         }
47     }
48 }
```

- linia 37 - definiujemy jak szybko chcemy żeby nasza gra działała
- linia 38 - obliczamy na ile chcemy uspić główny wątek gry aby zachować wcześniej zdefiniowaną szybkość
- linia 43 - usypiamy wątek

W ten prosty sposób udało nam się rozwiązać największy problem. Taka pętla co prawda działa ale daleko jej jeszcze do optimum. Jednym z jej problemów jest to, że co prawda aktualizujemy stan gry ze stałą szybkością co było naszym zamiarem, ale przy okazji ograniczyliśmy szybkość renderowania, co nie jest pożądanym zabiegiem.

Ponadto warto zwrócić uwagę na to, że *sleepTime* nie uwzględnia jak długo będą się wykonywały metody `render()` i `update()`, co powoduje, że jednak problem braku stabilności szybkości gry nadal nie został w pełni rozwiązany. Przykładowo kilka operacji na zmiennej typu `String` powoduje znaczny spadek wydajności naszej pętli.

```

64 private void update() {
65     for(int i = 0; i < 1000; i++){
66         abc = abc.toLowerCase().toUpperCase().toUpperCase().toLowerCase().toUpperCase()
67             .toUpperCase().toUpperCase().toUpperCase().toUpperCase().toLowerCase()
68             .toLowerCase().toUpperCase().toUpperCase().toLowerCase().toUpperCase()
69             .toUpperCase().toLowerCase().toUpperCase().toUpperCase().toUpperCase()
70             .toUpperCase().toUpperCase().toLowerCase().toLowerCase().toUpperCase()
71             .toUpperCase();
72     }
73 }
74 }

```

```

1145 ms | 6 updates, 6 renders
1136 ms | 6 updates, 6 renders
1160 ms | 6 updates, 6 renders
1183 ms | 6 updates, 6 renders

```

### 1.3 Zmaksymalizowanie FPS

Skoro już udało nam się osiągnąć (prawie) stałą szybkość gry, to następnym krokiem powinno być zmaksymalizowanie fps-ów.

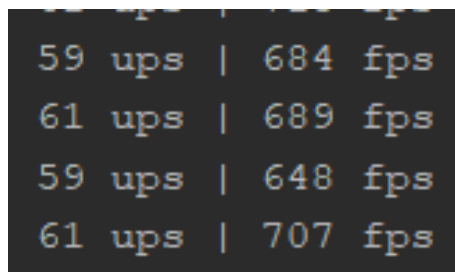
W poprzednim przykładzie metoda `render()` była wywoływana razem z metodą `update()` co znacząco ograniczało możliwości renderowania obrazu. Ponadto metoda `Thread.sleep()` jest problematyczna, więc chcemy jej unikać robiąc kolejny krok naprzód.

Poniżej przedstawiono przykład Game Loop-a, który ma znaczącą przewagę nad swoimi poprzednikami.

```

90 public void run(){
91     boolean running = true;
92     long lastTime = System.nanoTime();
93     double delta = 0.0;
94     double ns = 1000000000.0 / (float)DESIRED_FPS;
95     long timer = System.currentTimeMillis();
96     int updateCount = 0;
97     int frameCount = 0;
98     while(running) {
99         long now = System.nanoTime();
100         delta += (now - lastTime) / ns;
101         lastTime = now;
102         if (delta >= 1.0) {
103             update();
104             updateCount++;
105             delta--;
106         }
107         render();
108         frameCount++;
109         if (System.currentTimeMillis() - timer > 1000) {
110             timer += 1000;
111             System.out.println(updateCount + " ups | " + frameCount + " fps");
112             updateCount = 0;
113             frameCount = 0;
114         }
115     }
116 }

```



```
59 ups | 684 fps
61 ups | 689 fps
59 ups | 648 fps
61 ups | 707 fps
```

- linia 94 - aby uzyskać wysoką dokładność naszych obliczeń używamy `System.nanoTime()`, zmienna `ns` zapewnia konwersję z nanosekund na sekundy, ponadto dzieląc dodatkowo przez wartość `DESIRED_FPS` otrzymujemy odstęp pomiędzy kolejnymi wywołaniami metody `update()`.
- linia 95 - `System.currentTimeMillis()` zwraca ile milisekund upłynęło od 01.01.1970, zmienna `timer` używana jedynie w celu wypisywania aktualnej ilości ups/fps
- linia 100 - `(now - lastTime)` określa ile czasu potrzebowały metody `update()` i `render()`
- linia 102 - w linii 100 nastąpiła konwersja, operując już na sekundach możemy sprawdzić czy w danej iteracji mamy wywołać metodę `update()` (tak aby trzymać się stałej wartości `DESIRED_FPS` - patrz zmienna `ns`)
- linie 109 - 113 - odpowiadają za wypisywanie na konsolę aktualnych wartości fps/ups

Jak widzimy na zrzucie z konsoli uzyskany efekt odpowiada naszym założeniom, gra działa ze stałą szybkością, jednocześnie renderowanie odbywa się tak szybko jak to jest tylko możliwe, ograniczone jedynie przez platformę sprzętową.

Patrząc na zrzut z konsoli możemy mieć wrażenie, że wszystko działa tak jak powinno, ale... Powinniśmy się zastanowić co się stanie jeżeli nie będziemy w stanie uzyskać przykładowo 60 ups. Niestety gra zacznie zwalniać a do tego nie chcemy dopuścić, możemy sobie pozwolić na utratę kilku fps-ów ale za wszelką cenę chcemy utrzymać aktualizacje stanu gry na stałym poziomie i to właśnie jest kolejny krok w celu udoskonalenia silnika naszej gry.

Dodanie zmiennej `MAX_SKIPPED_FRAMES` która pozwoli nam w razie potrzeby pominąć maksymalnie pewną ilość fps ale zachować stały poziom ups w momencie gdy nasz sprzęt nie radzi sobie z aplikacją.

## 2 Zadania

Plik z zadaniami znajduje się na githubie, korzystaj z pozostawionych komentarzy w plikach z kodem.

### 2.1 Update()

W pierwszym zadaniu mamy do napisania metodę `update()`, która przy każdym wywołaniu zaktualizuje położenie prostokąta. Efekt jaki chcemy uzyskać to prostokąt poruszający się od "lewej do prawej".

Zwróć uwagę na konsolę, która pokazuje z jaką szybkością nasza aplikacja działa oraz ile klatek na sekundę jest renderowanych.

### 2.2 Stała szybkość gry

Skoro już wiemy jak aktualizować stan gry poprzez metodę `update()`, możemy się teraz zająć pierwszą optymalizacją naszego Game Loop-a. Korzystając z metody `Thread.sleep()`, która przyjmuje jako argument czas podany w milisekundach (!) zmodyfikuj metodę `run()` aby zrzut z konsoli wskazywał na wartość bliską `DESIRED_FPS`.

Uwaga: metoda `Thread.sleep()` nie jest dokładna, jako argument przyjmuje minimalny czas na jaki usypia i zależnie od innych wątków działających w systemie czy choćby Garbage Collector czas ten może się wydłużyć.

### 2.3 Game Loop w akcji

W tym zadaniu będziemy modyfikować klasę `Game`. Twoim zadaniem będzie:

- dokończyć implementację metody `render()` tak aby rysowała prostokąt na środku ekranu

- napisać metodę `update()` która zależnie od tego który klawisz (W, S, A, D) jest wciśnięty to zmieni odpowiednio położenie renderowania naszego prostokąta
- zadbać o to aby nie było możliwe wyjście poza obszar okna
- (opcjonalne) narysować w dowolnym miejscu kilka obiektów, (mogą być prostokąty) których położenie będzie niezmiennie i nie pozwolić graczowi przejść przez te obiekty (kolizja detekcji)
- (opcjonalne) spraw by wygenerowane prostokąty w poprzednim zadaniu poruszały się swobodnie

## Literatura

- [1] <https://dewitters.com/dewitters-gameloop/>
- [2] <https://gameprogrammingpatterns.com/game-loop.html>
- [3] <https://marcusman.com/>
- [4] <https://gamedev.stackexchange.com/>
- [5] <https://stackoverflow.com/>