Maszyny stanów

Tomasz Zakrzewski

Motywacja

- Aplikacje jako zbiory stanów: zależnie od wartości różnych zmiennych, nasza aplikacja znajduje się w innym stanie.
- Warunki przebywania w danym stanie mogą być skomplikowane (kolejny slajd...).

Motywacja c.d.

```
if (input == BUTTON_UP) {
  if (!hero.jumping) {
    hero.jump();
  } else if (!hero.doubleJumping) {
    hero.doubleJump();
  }
} else if (input == BUTTON_DOWN) {
  if (hero.jumping || hero.doubleJumping) {
    hero.dive();
  } else if (hero.standing) {
    hero.duck();
  } else if (hero.v_x != 0.0f && !hero.sliding &&
     !hero.jumping && !hero.doubleJumping && ...)
    hero.slide();
}
...
```

Motywacja c.d.

- “Eeee tam... Nie potrzebuję tego!”
- “We Don’t Need One Until We Do” (http://www.skorks.com/2011/09/why-developers-never-use-state-machines/)

Jak?

- Można samemu (o tym za chwilę...)
- QStateMachine (Qt)
- Meta State Machine (boost)

Projektowanie własnej maszyny

Czego oczekujemy? Musimy mieć:

- Stany,
- Przejścia (“tranzycje”),

Projektowanie własnej maszyny c.d.

. . . Ale może jeszcze chcemy mieć akcje w momencie wejścia/wyjścia ze stanu?
Zalety:

- Łatwo zaimplementować zmianę stanów obiektów (np. grafiki postaci w grze).
- Można zapamiętywać informacje w stanach i jakoś je modyfikować (np. liczba odwiedzeń stanu)

Jak?

Użyjemy C++11:

- Stany i tranzycje będące dowolnymi klasami podawanymi przez programistę (template)
  Np. stanami i przejściami mogą być enumy; stanami mogą być jakieś liczby całkowite, a przejściami litery (< 3 JaO); ...
- Każdy stan ma 2 przypisane sobie obiekty `std::function<void(void)>` - funkcje wywoływane na wejściu i wyjściu ze stanu.
  Dlaczego to jest fajne? Bo możemy zbindować **dowolną** funkcję (włącznie z metodami klas na rzecz konkrentych obiektów).
  Możemy nawet bindować metody klasy, której obiekty trzymamy jako stany!
- Dla wygody i wydajności użyjemy `std::unordered_map`.

Jak? c.d.

Problem z `std::hash`

`std::unordered_map<T, S>` używa `std::hash<T>`, żeby wyliczyć hasza każdego klucza. Dlatego, musimy dostarczyć implementację `std::hash<T>` dla naszej klasy będącej stanem oraz dla klasy będącej tranzycją. Ale...
Pewnie rzadko kiedy będziemy chcieli używać stanów i tranzycji innych niż enumy i/lub typy, dla których w STL jest zaimplementowany hash. Domyślnie nie da się łatwo haszować enum class bez pisania haszowania dla każdej enum classy osobno, ale udało mi się to objejść...

Implementacja...

QStateMachine

Pewnie już myśleliście, że nie powiem nic o Qt. ☺[2]
QStateMachine działa w pewnym sensie podobnie do naszej implementacji.

- Też mamy stany, ale są one dla nas przygotowane (`QState`). Możemy też dodawać własne korzystając z dziedziczenia (potrzebujemy podtypu `QAbstractState`).
- Tranzycje realizowane są głównie za pomocą signali. W momencie puszczenia sygnału, jeśli stan ma tranzycję czekającą na dany sygnał, to jest ona wykonywana.
- Zamiast naszych funkcji we własnej maszynie, stany QStateMachine mogą w momencie wejścia zmieniać properties i wywoływać metody QObjectów

Przykład

Diaballik!

Przykład

Czy może być tak pięknie i kolorowo?

Nie...

Na co trzeba uważać w QStateMachine?

Dwie rzeczy, które złożone razem mogą powodować totalną katastrofę:

- QStateMachine działa na sygnałach. Sygnały są przetwarzane w eventloopie (nie-natychmiast)!
- Sygnały są podpinane do tranzycji stanu dopiero po wejściu do niego (bo nie chcemy np. uruchamiać tranzycji z nieaktywnych stanów, które czekają na ten sam sygnał) i odpinane przy wyjściu.

Pytanie za milion

Na podstawie wyliczanki wyżej, co może nie zadziałać?

Jeszcze jeden problem...

Sygnały są dosyć wolne.