howTo :: Proste GUI wewnątrz gry (sheadovas/poradniki/howto/howto-proste-gui-wewnatrz-gry/)

Paź 10, 2015 / howTo (sheadovas/category/poradniki/howto/)

O napisaniu prostego systemu wyświetlającego GUI wewnątrz gry.

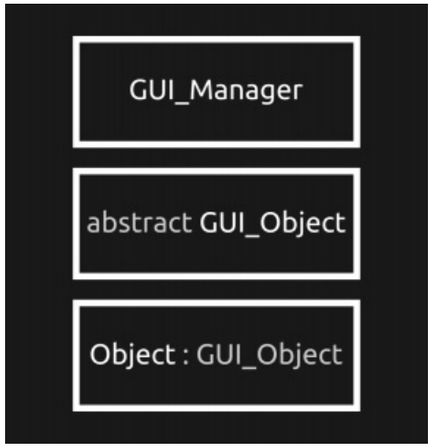
W (prawie) każdej grze potrzebny jest jakiś interfejs, który będzie zezwalał na wyświetlenie potrzebnych informacji graczowi na ekranie, czy to będzie guzik, czy pole do wpisania swojego imienia, czy też najzwyklejszego napisu tak nie każda biblioteka graficzna (czy silnik) oferuje system spełniający naszym wymaganiom (np. SFML wcale takiego systemu nie posiada).

W tym wpisie chciałbym Wam pokazać jak w stosunkowo łatwy sposób można osiągnąć łatwe do rozbudowania GUI.

Dziel i zwyciężaj!

Problem stworzenia pozwalającego na stworzenie prostego GUI może wydawać się skomplikowany, jednak jeżeli zgodnie z zasadą D&Q podzielimy go na mniejsze problemy to jesteśmy w stanie robić nasz problem na 3 główne składowe, z których będzie składało się GUI:

1. Klasy bazowej (*GUI_Manager*), która będzie zarządzała naszym GUI, sprawiała że nasza komunikacja/interakcja pomiędzy klasą, a biblioteką którą wybraliśmy będzie obsłużona właśnie w tej klasie (także przechwytywanie zdarzeń, itp).
2. Klasy abstrakcyjnej (*GUI_Object*), która jest bazowym budulcem reszty obiektów.
3. Klas stworzonych przy użyciu *GUI_Object*, są to fizycznie tworzone obiekty które jesteśmy w stanie narysować i nimi zarządzać.



(https://i0.wp.com/www.shead.ayz.pl/wp-content/uploads/2015/10/gui-schme.png)

Abstrakcyjny obiekt (GUI_Object)

Najbardziej podstawowym elementem jest klasa, z której zbudujemy resztę obiektów, w nawet najbardziej prymitywnym GUI powinna zawierać flagę informującą o tym czy obiekt powinien zostać rysowany, pozycję, a także metodę rysującą.

Przykładowy wygląd klasy — C++

```
class GUI_Object
{
public:
 GUI_Object(GUI_Object *parent = 0) { _parent = parent; }

 virtual void setActive(bool active) = 0;
 inline bool isActive() { return _isActive; }

 virtual void setPosition(Vector2 pos) = 0;
 inline Vector2 getPosition() { return _position; }

 virtual void draw(Window &window) = 0;

protected:
 GUI_Object *_parent;
 bool _isActive;
 Vector2 _position;
};
```

Ta klasa posiada konstruktor ustawiający rodzica (w momencie gdy rodzic zostanie przesunięty to także zrobi to dziedzic, pozycja będzie liczona względem rodzica, a nie całej sceny itd.), metody pozwalające na podstawowe manipulowanie obiektem (kontrola flagi aktywności, pozycji), a także metoda dla menadżera, która rysuje obiekt, jako argument podałem obiekt, który służy do rysowania standardowych obiektów w bibliotece, być może w niektórych bibliotekach jest zbędny.

Menadżer GUI (GUI_Manager)

Jest to klasa, która może zarządzać zdarzeniami generowanymi przez inne obiekty (nie uwzględniłem tego w przykładzie powyżej), a także zajmuje się rysowaniem wszystkich obiektów, które do niego dodamy, w naszym przypadku rysowane będą w takiej kolejności w jakiej je dodamy (tzn. pierwszy dodany obiekt, będzie rysowany jako pierwszy, drugi będzie rysowany na pierwszym, ...).

Przykładowy wygląd klasy C++

```
class GUI_Manager
{
public:
 GUI_Manager(Window *window) { _window = window; }

 void addObject(GUI_Object *object);
 void removeAllObjects();

 void drawObjects();

private:
 Window *_window;
 list <GUI_Object*> _objects;
};
```

Klasa składa się z listy zawierającej wskaźniki na wszystkie obiekty, które zostały dodane do menadżera i mają zostać narysowane.

Klasy pochodne GUI_Object

W tym paragrafie chciałbym wyjaśnić dlaczego przykładowa implementacja tego segmentu schematu się nie pojawi. Odpowiedź jest dość prosta: wiązałoby się to z dość mocnym użyciem jakiejś biblioteki, a chcę zachować tą część wpisów *howTo* jak najbardziej uniwersalną.

Jak cały system działa w praktyce pokażę dopiero przy implementacji.

Implementacja

Materiał pokazuje przykładową implementację prostego GUI (bez systemu zdarzeń) przy użyciu SFML 2.3.

(Link do playlisty zawierającą implementację pojawi się za jakiś czas)

Code ON!