

SFML - Izrada jednostavnog izbornika

Objektno programiranje - 12. vježbe

dr. sc. Sebastijan Horvat

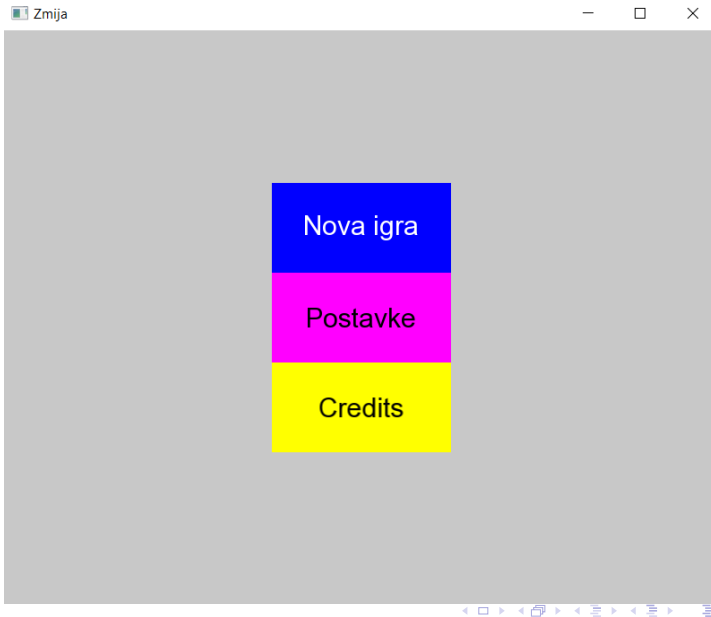
Prirodoslovno-matematički fakultet,
Sveučilište u Zagrebu

11. lipnja 2025. godine

Preuzeti kod s web stranice kolegija

- na ovim vježbama nastavljamo raditi na kodu za igru *Zmija* (prethodno smo dodali upravljanje događajima)
- kod koji trenutno imamo može se preuzeti na web stranici kolegija u materijalima za 11. vježbe (pod *Kod za igru Zmija s upravljanjem događajima*)
- dodat ćemo jednostavan izbornik - imat će tri gumba na koje korisnik može kliknuti
- prikazat ćemo postupak samo za prvi od ta tri gumba (gumb za pokretanje nove igre) - ostali se mogu samostalno implementirati za vježbu

Prikaz izbornika koji želimo dobiti



- napravimo novu datoteku `Izbornik.h` (te ju dodamo u naš projekt s igrom *Zmija*)
- u njoj napravimo sljedeće *include*ove:

```
#pragma once
#include "SFML/Graphics.hpp"
#include "Prozor.h"
#include "EventManager.h"
#include <vector>
#include <iostream>
```

- izbornik će imati tri gumba pa možemo napraviti posebnu klasu `Gumb` - svaki gumb sastoji se od pravokutnika i teksta na njemu (s time da klasa `sf::Text` zahtijeva `sf::Font`)

```
class Gumb {
public:
    // konstruktor prima: tekst za prikaz na gumbu,
    // boju teksta, boju pozadine, dimenzije i poziciju
    Gumb(const std::string&,
          const sf::Color&, const sf::Color&,
          const sf::Vector2f&, const sf::Vector2f&);
    void Renderiraj(Prozor*);
    bool SadrziKoordinate(float x, float y) {
        return pravokutnik.getGlobalBounds().contains(x,y);
    }
private:
    sf::Font font;
    sf::Text tekst;
    sf::RectangleShape pravokutnik;
};
```

Dodatno objašnjenje uz kod s prethodnog slajda

- funkcija `Renderiraj` služi za crtanje gumba na prozor (koji se ne kopira pa funkcija prima pokazivač na njega)
- funkcija `SadrziKoordinate` provjerava pripadaju li koordinate (x,y) gumbu (kako bi kasnije mogli provjeriti je li kliknuto na taj gumb)

Konstruktor klase Gumb (1. dio)

```
Gumb::Gumb(const std::string& t,  
    const sf::Color& bt, const sf::Color& bp,  
    const sf::Vector2f& dim, const sf::Vector2f& poz) {  
    if (!font.loadFromFile("Arial.ttf"))  
        std::cout << "Font nije ucitan!" << std::endl;  
    tekst.setFont(font);  
    tekst.setString(t);  
    tekst.setCharacterSize(30);  
    tekst.setFillColor(bt);  
    // tekst u sredini gumba  
    sf::FloatRect dimTeksta = tekst.getLocalBounds();  
    tekst.setOrigin(dimTeksta.left + dimTeksta.width / 2,  
        dimTeksta.top + dimTeksta.height / 2);  
    tekst.setPosition(sf::Vector2f(  
        poz + dim / 2.f  
    ));
```

Drugi dio konstruktora i renderiranje

```
    pravokutnik.setSize(dim);  
    pravokutnik.setFillColor(bp);  
    pravokutnik.setPosition(poz);  
}  
  
void Gumb::Renderiraj(Prozor* p) {  
    p->crtaj(pravokutnik);  
    p->crtaj(tekst);  
}
```

- istaknimo da se prvo mora iscrtati pravokutnik gumba, a tek onda tekst na njemu (u obratnom poretku bi pravokutnik prekrivio tekst i ne bismo ga mogli čitati)

```
class Izbornik {  
public:  
    Izbornik(Prozor*);  
    void Renderiraj();  
    void Updejt() {}  
    bool OdabranoPokretanje(float, float);  
private:  
    Prozor* p;  
    std::vector<Gumb> gumbi;  
};
```

- u ovoj verziji implementacije odlučili smo u konstruktoru izbornika primiti pokazivač na prozor u kojem će se iscrtavati (to će biti korisno, primjerice, u konstruktoru kako bi saznali dimenzije prozora radi centriranja gumba izbornika)
- funkcija `Updejt` u našem slučaju neće raditi ništa (naš izbornik je statičan - općenito to može služiti za neku animaciju izbornika)

Klasa Izbornik - konstruktor (1. dio)

```
Izbornik::Izbornik(Prozor* prozor) {  
    p = prozor;  
    sf::Vector2f dimGumba = sf::Vector2f(200, 100);  
    sf::Vector2f dimProzora = sf::Vector2f(  
        p->dohvatiVelicinu().x,  
        p->dohvatiVelicinu().y  
    );  
    sf::Vector2f sredinaProzora = dimProzora*0.5f;
```

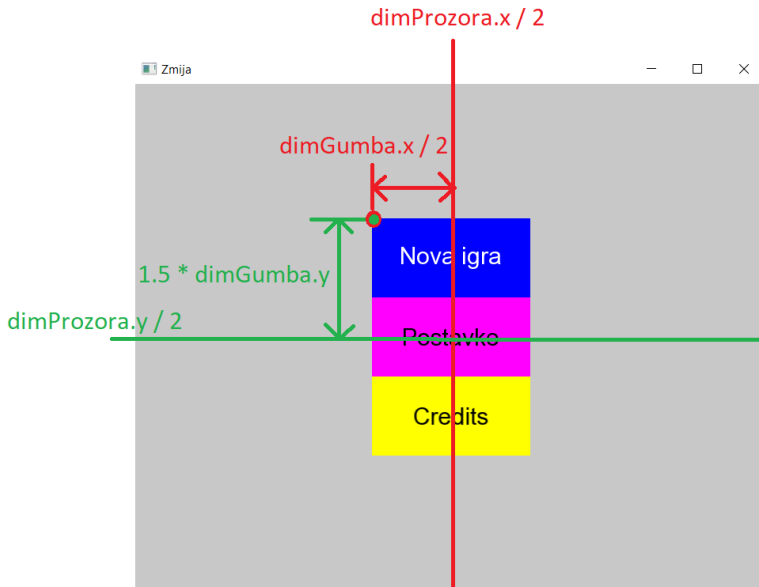
- radi jednostavnosti, odaberemo da svi gumbi imaju veličinu 200×100 piksela
- preko pokazivača `p` dohvatimo veličinu prozora (par `dimProzora`) - množenjem tog para brojeva s `0.5` dobivamo koordinate koje se nalaze na sredini prozora

Klasa Izbornik - konstruktor (2. dio)

```
// gumbi.clear();  
gumbi.reserve(3);  
gumbi.emplace_back("Nova igra",  
    sf::Color::White, sf::Color::Blue,  
    dimGumba, sf::Vector2f(  
        (dimProzora.x - dimGumba.x) / 2,  
        dimProzora.y / 2 - 1.5 * dimGumba.y  
    )  
);
```

- nije nužno čišćenje vektora `gumbi` obzirom da je po *defaultu* vektor prazan
- pomoću funkcije `reserve` unaprijed alociramo mjesta u vektoru za tri elementa (tj. gumba) - tako izbjegavamo kopiranje elemenata pri realokaciji vektora zbog kasnijih ubacivanja elemenata (kopiranje problematično zbog teksta koji ima samo pokazivač na pripadni font - zbog toga koristimo i `emplace_back` umjesto `push_back`)

Ilustracija izračuna koordinata za prvi gumb



Klasa Izbornik - konstruktor (3. dio)

- slično, odredimo koordinate za preostale gumbе:

```
gumbi.emplace_back("Postavke",  
    sf::Color::Black, sf::Color::Magenta,  
    dimGumba, sf::Vector2f(  
        (dimProzora.x - dimGumba.x) / 2,  
        dimProzora.y / 2 - 0.5 * dimGumba.y  
    )  
);  
gumbi.emplace_back("Credits",  
    sf::Color::Black, sf::Color::Yellow,  
    dimGumba, sf::Vector2f(  
        (dimProzora.x - dimGumba.x) / 2,  
        dimProzora.y / 2 + 0.5 * dimGumba.y  
    )  
);  
}
```

Klasa Izbornik - renderiranje i provjera pokretanja

- `Renderiraj` funkcija renderira svaki gumb iz vektora `gumbi` (pozivom njihove metode `Renderiraj`)
- funkcija `OdabranoPokretanje` provjerava (pozivom metode prvog gumba iz vektora - to je gumb za pokretanje nove igre) pripadaju li koordinate (x,y) gumbu za pokretanje igre (tu ćemo funkciju koristiti kasnije)

```
void Izbornik::Renderiraj() {  
    for (Gumb& gumb : gumbi)  
        gumb.Renderiraj(p);  
}
```

```
bool Izbornik::OdabranoPokretanje(float x, float y) {  
    return gumbi[0].SadrziKoordinate(x, y);  
}
```

Dodamo u `Igra.h` datoteku

- enumeriramo sva moguća stanja koja naša igra može imati (podsjetnik: u prezentaciji se bavimo samo stanjem `Intro` i stanjem `GlavniIzbornik`)
- ovdje smo stanja enumerirali od broja 1 (mogli smo i bez toga pa bi enumeracija krenula od 0)

```
#include "Izbornik.h"
```

```
enum class Stanje {  
    Intro = 1,  
    GlavniIzbornik,  
    IgraTraje,  
    Pauzirano,  
    GameOver,  
    Credits  
};
```

Dodamo u Igra klasu

- u klasi `Igra` pamtimo trenutno stanje igre u varijabli `stanje` (primjerice, jesmo li u početnom izborniku ili trenutno traje igra)
- također imamo i instancu klase `Izbornik` - nju smo stavili na kraj klase (prvo trebamo stvoriti prozor kako bi konstruktoru izbornika mogli proslijediti pokazivač na njega)

```
class Igra {  
    ...  
private:  
    ...  
    Stanje stanje;  
    Izbornik izbornik; // mora na kraj tako da se  
    // prvo prozor p stvori  
};
```

Novo u konstruktoru klase Igra

```
Igra::Igra() : p("Zmija", sf::Vector2u(800, 640)),  
             svijet(16, p.dohvatiVelicinu(), &textbox),  
             zmija(16, &textbox),  
             izbornik(&p)  
{  
    stanje = Stanje::GlavniIzbornik;
```

- u konstruktoru klase Igra, inicijaliziramo izbornik (treba pokazivač na prozor pa dajemo adresu od p)
- početno stanje stavljamo stanje glavnom izbornika (obzirom da imamo samo glavni izbornik i stanje dok traje igra)

Nova verzija Igra::Renderiraj

```
void Igra::renderiraj() {  
    p.ocisti();  
  
    switch (stanje)  
    {  
        case Stanje::IgraTraje:  
            svijet.Renderiraj(p);  
            zmija.Renderiraj(&p);  
            p.crtaj(vidljivo);  
            textbox.Renderiraj(p);  
            break;  
        case Stanje::GlavniIzbornik:  
            izbornik.Renderiraj();  
            break;  
    }  
  
    p.prikazi();  
}
```

Nova verzija Igra::update

```
void Igra::update() {  
    p.update();  
    float vrijemeIteracije = 1.0f /  
        zmija.DohvatiBrzinu();  
    switch (stanje) {  
        case Stanje::IgraTraje:  
            if (vrijeme.asSeconds() >= vrijemeIteracije) { ... }  
            break;  
        case Stanje::GlavniIzbornik:  
            izbornik.Updejt();  
            break;  
    }  
}
```

- u funkcijama za renderiranje i updejt ovisno o stanju radimo renderiranje ili updejt tog stanja (primjerice, glavnog izbornika ako smo u stanju Stanje::GlavniIzbornik)

Reagiranje na događaje

- želimo omogućiti da korisnik može klikom lijeve tipke miša odabrati neki tipku u glavnom izborniku
- dodamo u `Keys.cfg` datoteku (događaj klika lijevom tipkom miša - tu funkcionalnost nazovemo *klikMisem*):

```
klikMisem 9:0
```

- dodamo u klasu `Igra` (morat ćemo provjeriti je li kliknuto na područje koje zauzima neki gumb):

```
void ProvjeriKlik(EventDetails*);
```

- dodamo u konstruktor igre (registriramo poziv odgovarajuće funkcije za funkcionalnost koju smo nazvali *klikMisem*):

```
p.dohvatiEventManager()->AddCallback(  
    "klikMisem", &Igra::ProvjeriKlik, this  
);
```

Funkcija ProvjeriKlik

```
void Igra::ProvjeriKlik(EventDetails* ed) {  
    sf::Vector2i misPozicija  
        = p.dohvatiEventManager()->GetMousePos(  
            p.dohvatiProzor());  
    // koordinate obzirom na prozor, ne obzirom na Desktop  
    std::cout << "Kliknuto na koordinate ("  
        << misPozicija.x << "," << misPozicija.y << ")" << std::endl;  
    if (izbornik.OdabranoPokretanje(misPozicija.x, misPozicija.y)) {  
        std::cout << "Klik na gumb za pokretanje igre!" << std::endl;  
        stanje = Stanje::IgraTraje;  
    }  
}
```

Pokrenuti i testirati - uočiti i kad se igra pokrene da imamo reakciju (u obliku ispisa u konzoli) na klik na mjesto gdje je bio gumb. Također i ako u izborniku pomaknemo vidljivost (*Shift* + klik lijevom tipkom miša) to se vidi kad se pokrene igra.

Popravak problema

- ako se maknemo iz stanja glavnog izbornika, treba maknuti i *callback* (poziv odgovarajuće funkcije) za funkcionalnost *klikMisem* - to mićemo pomoću *RemoveCallback* funkcije (usporediti sa „suprotnom” funkcijom *AddCallback*)
- također tada pomoću pomoćne metode koju ćemo napisati, dodaje *callbacke* za funkcionalnosti tijekom igre (poput one za pomicanje zmije desno)
- dodamo u `if` blok iz funkcije s prethodnog slajda:

```
p.dohvatiEventManager()->RemoveCallback  
    ("klikMisem");  
RegistrirajEventeIgre();
```

- dodamo potrebnu deklaraciju nove metode u klasu `Igra` (pod `public`, iako može i drugačije):

```
void RegistrirajEventeIgre();
```

Funkcija RegistrirajEventeIgre

- prebacimo iz konstruktora klase `Igra` dodavanje *callbackova* u novu funkciju `RegistrirajEventeIgre`:

```
void Igra::RegistrirajEventeIgre() {
    p.dohvatiEventManager()->AddCallback("pomakGore",
        &Igra::ZmijaSmjerGore, this);
    p.dohvatiEventManager()->AddCallback("pomakDolje",
        &Igra::ZmijaSmjerDolje, this);
    p.dohvatiEventManager()->AddCallback("pomakDesno",
        &Igra::ZmijaSmjerDesno, this);
    p.dohvatiEventManager()->AddCallback("pomakLijevo",
        &Igra::ZmijaSmjerLijevo, this);
    p.dohvatiEventManager()->AddCallback("promjenaVidljivosti",
        &Igra::pomakniPoljeVidljivosti, this);
}
```

- funkcije poput *update* i *Renderiraj* klase `Igra` pozivaju odgovarajuće funkcije za ažuriranje i renderiranje stanja koje je trenutno aktivno
- pri prelasku iz jednog stanja u drugo, poželjno je imati pomoćne funkcije koje dodaju ili uklanjaju potrebne *callbackove*
- za upravljanje složenijim stanjima može se napraviti posebna klasa `stateManager` (kao što za upravljanje događajima imamo `eventManager`) - to je opisano u 5. poglavlju knjige: Raimondas Pupius, *SFML Game Development by Example*, Packt Publishing, 2015