

SFML - Upravljanje događajima

Objektno programiranje - 11. vježbe

dr. sc. Sebastijan Horvat

Prirodoslovno-matematički fakultet,
Sveučilište u Zagrebu

4. lipnja 2025. godine

Upravljanje događajima

- prikazat ćemo upravljanje događajima na primjeru naše igre *Zmija* (kod se može preuzeti s web-stranice kolegija)
- podsjetnik na situaciju koju imamo trenutno u kodu te igre glede provjere događaja:
 - koje funkcije u `main` funkciji izravno ili neizravno provjeravaju događaje:

```
Igra igra;  
while (!igra.gotovo()) {  
    igra.obradiUlaz();  
    igra.update();  
    igra.renderiraj();  
    igra.restartSata();  
}
```

Gdje provjeravamo događaje

- funkcija `void Igra::update()` poziva za prozor `igra.p` metodu `void Prozor::update()` koja radi sljedeće:

```
sf::Event event;
while (prozor.pollEvent(event))
{
    if (event.type == sf::Event::Closed)
        gotov = true;
    else if (event.type == sf::Event::KeyPressed
        && event.key.code == sf::Keyboard::F5)
        prebaciNaCijeli();
}
```

- uočite uokvireno: osim **tipa događaja** pritiska tipke provjeravamo i **kod tipke** koja je bila pritisnuta

Gdje još provjeravamo događaje

- u funkciji `void Igra::obradiUlaz()` imamo hrpu `else-if`-ova:

```
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up)
    && zmija.dohvatiFizickiSmjer() != Smjer::Dolje) {
    zmija.PostaviSmjer(Smjer::Gore);
} else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down)
    && zmija.dohvatiFizickiSmjer() != Smjer::Gore) {
    zmija.PostaviSmjer(Smjer::Dolje);
} else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left)
    && zmija.dohvatiFizickiSmjer() != Smjer::Desno) {
    zmija.PostaviSmjer(Smjer::Lijevo);
} else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right)
    && zmija.dohvatiFizickiSmjer() != Smjer::Lijevo) {
    zmija.PostaviSmjer(Smjer::Desno);
}
```

- uočite problem s održavanjem (više tipaka, kombinacija tipaka, tipke miša, igraće palice ili *joysticka*, ...)

Dva broja potrebna za određivanje događaja

- podsjetnik na trenutni kod:

```
sf::Event event;
while (prozor.pollEvent(event))
{
    if (event.type == sf::Event::Closed)
        gotov = true;
    else if (event.type == sf::Event::KeyPressed
        && event.key.code == sf::Keyboard::F5)
        prebaciNaCijeli();
}
```

- ako želimo saznati je li pritisnuta tipka $F5$ prvo provjerimo radi li se o **tipu** događaja pritiska tipke te tek onda možemo provjeriti radi li se o **kodu** tipke $F5$
- uočimo da tu dodatnu informaciju (kod tipke) nemamo za tip događaja zahtjeva zatvaranja prozora

Prvi broj - enumeracija za tip događaja

- klasa `Event` ima enumeraciju tipova događaja (primjerice, `sf::Event::Closed` ima vrijednost 0, `sf::Event::Resized` ima vrijednost 1, itd.)
- „poseban tip” `Count` predstavlja koliko ima tipova događaja (npr. ako mu je vrijednost 23 u nekoj verziji SFML-a tada imamo ukupno 23 vrste događaja)

```
enum EventType
{
    Closed,
    Resized,
    LostFocus,
    GainedFocus,
    TextEntered,
    KeyPressed,
    KeyReleased,
    MouseWheelMoved,
    MouseWheelScrolled,
    MouseButtonPressed,
    MouseButtonReleased,
    MouseMoved,
    MouseEntered,
    MouseLeft,
    JoystickButtonPressed,
    JoystickButtonReleased,
    JoystickMoved,
    JoystickConnected,
    JoystickDisconnected,
    TouchBegan,
    TouchMoved,
    TouchEnded,
    SensorChanged,

    Count
};
```

Drugi broj - dodatna informacija

- primjerice, klasa `Keyboard` ima enumeraciju za kod tipke (na slici desno) (`sf::Keyboard::A` ima vrijednost 0, `sf::Keyboard::B` ima vrijednost 1, itd.)

- postoje i za neke druge tipove događaja, primjerice, za tipke miša postoji enumeracija u klasi `Mouse`:

```
enum Button
{
    Left,
    Right,
    Middle,
    XButton1,
    XButton2,

    ButtonCount
};
```

```
enum Key
{
    Unknown = -1,
    A = 0,
    B,
    C,
    D,
    E,
    F,
    G,
    H,
    I,
    J,
    K,
    L,
    M,
    N,
    O,
    P,
    Q,
    R,
    itd.
```

Primjer za određivanje dva broja

- u kodu (unutar Visual Studio; može i preko SFML-ovih web-stranica) stavimo pokazivač na `sf::Event::KeyPressed` te na `sf::Keyboard::F5` kako bi dobili kojim cjelobrojnim vrijednostima odgovaraju u enumeracijama:

```
sf::Event event;
while (prozor.pollEvent(event))
{
    if (event.type == sf::Event::Closed)
        gotov = true;
    else if (event.type == sf::Event::KeyPressed
        && event.key.code == sf::Keyboard::F5)
        prebaciNaCijeli();
}
```

- time dobivamo sljedeće informacije:

```
enum sf::Event::EventType::KeyPressed = 5
enum sf::Keyboard::Key::F5 = 89
```


Nastavak primjera

- dakle, par brojeva koji odgovara događaju pritiska tipke `F5` je

$(5, 89)$

- taj par brojeva pisat ćemo ovako:

$5:89$

- za tipove događaja koji nemaju dodatnu informaciju, stavljamo broj koji predstavlja dodatnu informaciju da je jednak 0
- primjerice, za tip događaja `sf::Event::Closed` imamo njegovu cjelobrojnu vrijednost 0, a kako on nema dodatnu informaciju i drugi broj para stavljamo na 0
- dakle, zahtjevu za zatvaranje prozora (tipa `sf::Event::Closed`) pridružujemo par

$0:0$

- za bolje razumijevanje, prvo ćemo koristiti već implementiranu klasu u datoteci `EventManager.h`, a kasnije ćemo analizirati njen kod

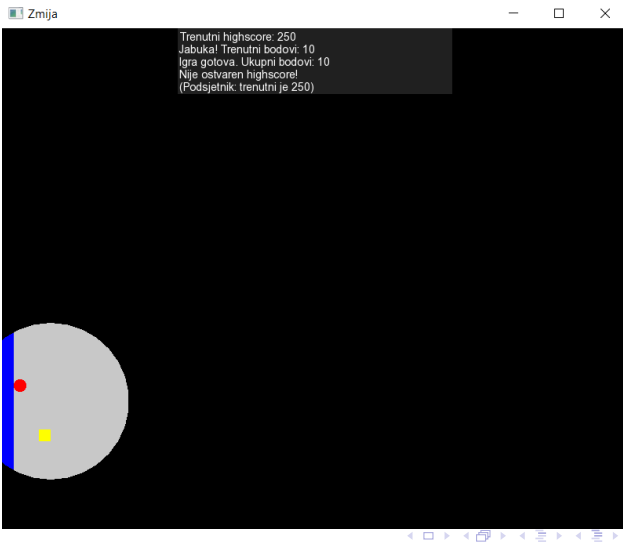
Preuzeti datoteku `EventManager.h` i dodati ju u projekt s našom igrom *Zmija*.

Ukratko što radi *event manager*:

- iz **konfiguracijske datoteke** učitava tzv. *bindinge*
- **binding** je kombinacija događaja koja je potrebna za nešto (npr. istodobno pritisnuta tipka *Shift* i pritisak lijeve tipke miša)
- takvu kombinaciju povezujemo sa stringom koji predstavlja neku **funkcionalnost**
- pojedinom stringu pridružimo **funkciju** koja se treba izvršiti kad njegova kombinacija događaja bude ispunjena

Primjer - što ćemo napraviti u igri *Zmija*

- dodamo krug (transparentan s ja...ako velikim crnim obrubom) koji predstavlja područje prozora koje igrač trenutno može vidjeti:



Na koje događaje ćemo reagirati

Uz svaki smislimo neki **naziv funkcionalnosti**:

- zahtjev za zatvaranje prozora
 - potrebno je zatvoriti prozor (**Window_close**)
- pritisak na tipku F5
 - mijenjamo u način prikaza preko cijelog ekrana (ili izlazimo iz takvog načina prikaza) (**Fullscreen_toggle**)
- pritisak na tipku gore, dolje, lijevo ili desno
 - zahtjev¹ za promjenu smjera kretanja zmije prema gore, dolje, lijevo ili desno (**pomakGore**, **pomakDolje**, **pomakLijevo**, **pomakDesno**)
- pritisak lijeve tipke miša za vrijeme držanja tipke *Shift*
 - postavljanje kruga vidljivosti tako da mu je središte na koordinatama prozora tamo gdje je kliknuto (**promjenaVidljivosti**)

¹sjetite se da gledamo i tzv. *fizički smjer zmije*

Parovi brojeva koji odgovaraju tim događajima

- zahtjev za zatvaranje prozora:

0 : 0

- pritisak tipke prema dolje:

5 : 74

- pritisak tipke `F5`:

5 : 89

- pritisak tipke prema lijevo:

5 : 71

- pritisak tipke prema gore:

5 : 73

- pritisak tipke prema desno:

5 : 72

- pritisnuta je tipka *Shift* za vrijeme čega se javi pritisak lijeve tipke miša:

9 : 0 24 : 38

Napomena: U posljednjem smo mogli navesti i u obratnom poretku (prvo 24:38, a onda 9:0, kako i je u tekstu opisa). Broj 24 nije službeno iz SFML-ove enumeracije tipova događaja, nego je definiran u `EventManager` klasi (je li tipka trenutno pritisnuta - to inače provjera funkcija `sf::Keyboard::isKeyPressed`).

Napomena o događajima koje želimo

- treba razmisliti kakve događaje točno želimo - primjerice:
 - ako želimo izvršiti neku funkciju kad korisnik pritisne dvije tipke (dva `KeyPressed` događaja) korisnik ih mora pritisnuti u točno isti trenutak (što je rijetko moguće)
 - ako želimo izvršiti neku funkciju kad korisnik drži pritisnutima dvije tipke, tada će se pozivi funkcije ponoviti više puta (sve dok ih korisnik drži pritisnutima)
- mi smo se odlučili na to da korisnik drži pritisnutom tipku *Shift*, a onda će se željena funkcija izvršiti jednom kad korisnik pritisne lijevu tipku miša

Konfiguracijska datoteka

- napravimo datoteku naziva *Keys.cfg* (*Configuration File* - za spremanje postavki programa) u istoj mapi gdje imamo u projektu kod za igru *Zmija*
- sadržaj te datoteke treba biti sljedeći:

```
Window_close 0:0  
Fullscreen_toggle 5:89  
pomakGore 5:73  
pomakDolje 5:74  
pomakLijevo 5:71  
pomakDesno 5:72  
promjenaVidljivosti 9:0 24:38
```

- svaki redak te datoteke sadrži: naziv funkcionalnosti (string bez razmaka) te niz parova brojeva odvojenih razmakom - oni predstavljaju koji se sve događaji moraju (istovremeno!) dogoditi kako bi se pokrenula tražena funkcionalnost

Gdje dodati *EventManager* objekt

- podsjetnik: u `main` funkciji u glavnoj petlji poziva se `igra.update()`; koja poziva `p.update()`; (gdje je `p` klase `Prozor` koji igra klase `Igra` koristi)
- funkcija `update` klase `Prozor` ima petlju po svim događajima koji su se dogodili:

```
void Prozor::update() {  
    sf::Event event;  
    while (prozor.pollEvent(event))  
    {  
        ...   provjere događaja   ...  
    }  
}
```

- prema tome, dodat ćemo `EventManager` objekt u klasu `Prozor` - on će u `update` funkciji analizirati sve događaje i posebno pratiti jesu li se dogodili oni koji su nam potrebni za tražene funkcionalnosti

Dodavanje *EventManager* objekta

- u datoteku `Prozor.h` dodamo potrebni `include`:

```
#include "EventManager.h"
```

- klasa `Prozor` imat će svoju instancu `EventManager` klase:

```
class Prozor {  
    ...  
    private:  
        ...  
        EventManager eventManager;  
};
```

- klasa `EventManager` u svom konstruktoru poziva svoju metodu

```
void EventManager::LoadBindings()
```

koja će učitati *bindingse* iz datoteke `Keys.cfg`

Praćenje ima li prozor fokus

- ne želimo obrađivati događaje ako prozor nije u fokusu
- kako praćenje fokusa može biti općenito korisno, dodat ćemo u klasu `Prozor` varijablu koja će nam reći je li prozor trenutno u fokusu (te funkciju da se njena vrijednost može provjeriti izvana)

```
class Prozor {  
    public:  
        ...  
        bool imaFokus() {  
            return fokus;  
        }  
    private:  
        ...  
        bool fokus; // je li prozor u fokusu  
};
```

Početno postavljanje vrijednosti za fokus

- podsjetnik: oba konstruktora klase `Prozor`:

```
Prozor::Prozor(const std::string&,
               const sf::Vector2u&);
Prozor::Prozor();
```

pozivaju metodu `Postavi` koja postavlja početne vrijednosti dijelova klase `Prozor` pa ćemo tamo početno `fokus` postaviti na `true`:

```
void Prozor::Postavi(...) {
    ...
    fokus = true;
    //otvaranje prozora nakon popunjavanja postavki
    Stvori();
}
```

Promjena vrijednosti od `fokus`

- vrijednost od `fokus` mijenjamo pojavom događaja tipa gubitka i dobivanja fokusa za `prozor` (klase `RenderWindow`)
- u petlji koja prolazi svim događajima, posebno gledamo događaje tih tipova:

```
void Prozor::update() {  
    sf::Event event;  
    while (prozor.pollEvent(event)) {  
        if (event.type == sf::Event::LostFocus) {  
            fokus = false;  
        }  
        else if (event.type == sf::Event::GainedFocus) {  
            fokus = true;  
        }  
        ...  
    }  
}
```

Povezivanje funkcionalnosti s funkcijama

- iz konfiguracijske datoteke smo primjerice učitali

```
Window_close 0:0
```

- prema tome, kad se dogodi događaj kojem odgovara 0:0 ispunjeni su svi uvjeti za funkcionalnost (što je za nas string!)
"Window_close"
- potrebno je prvo napisati funkciju koja će se tada pozvati - to treba biti funkcija koja će u petlji u funkciji `Prozor::Update` zamijeniti sljedeće:

```
if(event.type == sf::Event::Closed)
    gotov = true;
```

- dakle, funkcija treba samo `gotov` postaviti na `true` (to će biti metoda klase `Prozor`)

Dodavanje funkcije za funkcionalnost `Window_close`

```
class Prozor {  
    public:  
        ...  
        void zatvori(EventDetails* l_details) {  
            gotov = true;  
        }  
        ...  
};
```

- funkcija koju ćemo dodati nekoj funkcionalnosti mora (zbog načina na koji je implementiran `EventManager`) imati prototip
`void naziv_funkcije(EventDetails*);`
- funkcija prima pokazivač na strukturu `EventDetails` koja je također definirana u datoteci `EventManager.h`
- ta struktura na koju pokazuje `l_details` je već popunjena i sadrži dodatne informacije o događaju (više o tome kasnije) - uočite da je moramo navesti zbog propisanog prototipa iako nam `l_details` neće ovdje trebati

Povezivanje funkcije s funkcionalnošću

- to se radi pomoću metode `AddCallback`
- napraviti ćemo to u funkciji `Postavi` klase `Prozor` koju pozivaju konstruktori te klase (uočite: to će se dogoditi nakon što je konstruiran `EventManager` te klase, tj. već smo učitali *bindinge*)

```
void Prozor::Postavi(...) {  
    ...  
    eventManager.AddCallback("Window_close",  
                             &Prozor::zatvori, this);  
    //otvaranje prozora nakon popunjavanja postavki  
    Stvori(); }  

```

- prvi argument je funkcionalnost (string), drugi argument pokazivač na metodu (odgovarajućeg prototipa), a treći pokazivač na instancu klase `Prozor` kojoj ta metoda pripada (to je upravo ova koja poziva svoju `Postavi` metodu)

Povezivanje funkcionalnosti za *fullscreen*

- u konfiguracijskoj datoteke je učitano:

```
Fullscreen_toggle 5:89
```

- prema tome, kad se dogodi događaj kojem odgovara 5:89 ispunjeni su svi uvjeti za funkcionalnost

```
"Fullscreen_toggle"
```

- u petlji u funkciji `Prozor::Update` želimo zamijeniti sljedeće:

```
if (event.type == sf::Event::KeyPressed  
&& event.key.code == sf::Keyboard::F5)  
    prebaciNaCijeli();
```

- dakle, već imamo funkciju `prebaciNaCijeli` koju treba pozvati za tu funkcionalnost - samo joj treba promijeniti prototip

Prilagodba funkcije za *fullscreen* funkcionalnost

- prilagodimo postojeću funkciju `prebaciNaCijeli` tako da ima traženi prototip:

```
class Prozor {  
    public:  
        ...  
        void prebaciNaCijeli(EventDetails*);  
        ...  
};  
  
...  
  
void Prozor::prebaciNaCijeli(EventDetails* ed) {  
    ...  
}
```

Povezivanje funkcije s funkcionalnošću

- povezujemo u funkciji `Postavi` klase `Prozor` koju pozivaju konstruktori te klase (slično kao i prije):

```
void Prozor::Postavi(...) {  
    ...  
    eventManager.AddCallback("Fullscreen_toggle",  
        &Prozor::prebaciNaCijeli, this);  
    //otvaranje prozora nakon popunjavanja postavki  
    Stvori();  
}
```

Novi kod izmijenjene funkcije `Prozor::update`

```
void Prozor::update() {
    sf::Event event;
    while (prozor.pollEvent(event)) {
        if (event.type == sf::Event::LostFocus) {
            fokus = false;
            eventManager.SetFocus(false);
        }
        else if (event.type == sf::Event::GainedFocus) {
            fokus = true;
            eventManager.SetFocus(true);
        }
        eventManager.HandleEvent(event);
    }
    eventManager.Update();
}
```

- objašnjenje gornjeg koda nalazi se na idućim slajdovima

Postavljanje fokusa za eventManager

- gubitak ili dobivanje fokusa bilježimo i u klasi `EventManager`
- ta klasa ima privatan dio

```
bool m_hasFocus;
```

koji mijenjamo pomoću metode `EventManager::SetFocus`

```
void SetFocus(const bool& l_focus) {  
    m_hasFocus = l_focus;  
}
```

- praćenje fokusa je važno jer ne želimo izvršavati funkcije za odgovarajuće funkcionalnosti kao nema fokusa - zato metoda `EventManager::Update` za provjeru ispunjenja uvjeta funkcionalnosti i pokretanja funkcija na početku svog koda ima:

```
if (!m_hasFocus)  
    return;
```

Obrada svakog događaja u eventManageru

- svaki događaj u petlji

```
while (prozor.pollEvent(event)) {  
    ...  
}
```

prosljeđujemo metodi `EventManager::HandleEvent`

- ta metoda analizira događaj na sljedeći način:
 - pogleda zanima li nas taj događaj - odgovara li njegov tip (i dodatna informacija ako ju ima) nekom događaju iz niza *bindinga* koje imamo
 - jedan *binding* odgovara retku iz konfiguracijske datoteke - naziv funkcionalnosti i niz parova brojeva za događaje - primjerice:

`promjenaVidljivosti 9:0 24:38`

- u konfiguracijskoj datoteci smo još naveli četiri funkcionalnosti:

`pomakGore 5:73`

`pomakDolje 5:74`

`pomakLijevo 5:71`

`pomakDesno 5:72`

`promjenaVidljivosti 9:0 24:38`

Dodamo u klasu Igra

- dodamo u klasu Igra (u datoteci Igra.h):

```
class Igra {  
    public:  
        ...  
        void ZmijaSmjerGore (EventDetails*);  
        void ZmijaSmjerDolje (EventDetails*);  
        void ZmijaSmjerLijevo (EventDetails*);  
        void ZmijaSmjerDesno (EventDetails*);  
        void pomakniPoljeVidljivosti (EventDetails*);  
    private:  
        ...  
        sf::CircleShape vidljivo;  
};
```

- dodali smo i krug za dio prozora koji će korisnik moći vidjeti

Dodavanje početnih postavki za polje vidljivosti

- dodajemo u konstruktor klase `Igra` početne postavke za krug vidljivosti:

```
Igra::Igra() :...{  
    vidljivo.setFillColor(sf::Color::Transparent);  
    vidljivo.setOutlineColor(sf::Color::Black);  
    vidljivo.setOutlineThickness(  
        p.dohvatiVelicinu().x);  
    vidljivo.setRadius(100.f);  
    vidljivo.setOrigin(vidljivo.getRadius(),  
        vidljivo.getRadius());  
    vidljivo.setPosition(100.f, 100.f);  
    ...  
}
```


- očekivano, to polje kao i ostale objekte crtamo u `Igra::Renderiraj` funkciji:

```
void Igra::renderiraj() {  
    p.ocisti();  
    svijet.Renderiraj(p);  
    zmija.Renderiraj(&p);  
    p.crtaj(vidljivo);  
    textbox.Renderiraj(p);  
    p.prikazi();  
}
```

Dodavanje funkcija za funkcionalnosti

```
void Igra::ZmijaSmjerGore(EventDetails* ed) {  
    if (zmija.dohvatiFizickiSmjer() != Smjer::Dolje)  
        zmija.PostaviSmjer(Smjer::Gore);  
}  
  
void Igra::ZmijaSmjerDolje(EventDetails* ed) {  
    if (zmija.dohvatiFizickiSmjer() != Smjer::Gore)  
        zmija.PostaviSmjer(Smjer::Dolje);  
}  
  
void Igra::ZmijaSmjerLijevo(EventDetails* ed) {  
    if (zmija.dohvatiFizickiSmjer() != Smjer::Desno)  
        zmija.PostaviSmjer(Smjer::Lijevo);  
}  
  
void Igra::ZmijaSmjerDesno(EventDetails* ed) {  
    if (zmija.dohvatiFizickiSmjer() != Smjer::Lijevo)  
        zmija.PostaviSmjer(Smjer::Desno);  
}
```

Dodavanje funkcija za funkcionalnosti

- za dohvat koordinata na koje je korisnik kliknuo mišem, koristimo metodu `GetMousePos` klase `EventManager`
- ako ne želimo koordinate obzirom na radnu površinu (*desktop*) nego obzirom na prozor klase `RenderWindow` u kojem se program izvršava, dajemo toj funkciji referencu na taj prozor

```
void Igra::pomakniPoljeVidljivosti(EventDetails* ed) {  
    sf::Vector2i misPozicija  
        = p.dohvatiEventManager()->GetMousePos(  
            p.dohvatiProzor()  
        );  
    vidljivo.setPosition(misPozicija.x, misPozicija.y);  
    std::cout << "Pomicanje vidljivosti na: "  
        << misPozicija.x << ", " << misPozicija.y  
        << std::endl;  
}
```

- za ovo nam je trebao dohvat `EventManager` instance koju ima prozor `p` te i sam taj prozor, pa dodamo te metode u klasu `Prozor` (kod je na idućem slajdu)

Nove metode u klasi Prozor

```
class Prozor {  
    public:  
        ...  
        sf::RenderWindow* dohvatiProzor() {  
            return &prozor;  
        }  
        EventManager* dohvatiEventManager() {  
            return &eventManager;  
        }  
        ...  
};
```

Povezivanje funkcija s funkcionalnostima

- povezujemo u konstruktoru klase Igra:

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

Više nema potrebe za funkcijom obradiUlaz

- sad više nema potrebe za funkcijom `obradiUlaz` klase `Igra` pa njen poziv zakomentiramo u `main` funkciji:

```
int main() {  
    Igra igra;  
    while (!igra.gotovo()) {  
        //igra.obradiUlaz(); // zakomentirano  
        igra.update();  
        igra.renderiraj();  
        igra.restartSata();  
    }  
    return 0;  
}
```

Više nema potrebe za funkcijom `obradiUlaz`

- možemo ju također zakomentirati (ili ju izbaciti) i u klasi `Igra`:

```
class Igra {
public:
    Igra();
    ~Igra();
    //void obradiUlaz(); // zakomentirano
    void update();
    void renderiraj();

/*void Igra::obradiUlaz() {
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up)
        && zmija.dohvatiFizickiSmjer() != Smjer::Dolje) {
        zmija.PostaviSmjer(Smjer::Gore);
    } else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down)
        && zmija.dohvatiFizickiSmjer() != Smjer::Gore) {
        zmija.PostaviSmjer(Smjer::Dolje);
    } else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left)
        && zmija.dohvatiFizickiSmjer() != Smjer::Desno) {
        zmija.PostaviSmjer(Smjer::Lijevo);
    } else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right)
        && zmija.dohvatiFizickiSmjer() != Smjer::Lijevo) {
        zmija.PostaviSmjer(Smjer::Desno);
    }
} */
```