Programiranje video igara u biblioteci SFML

Šimunović, Zvonimir

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Zvonimir Šimunović

PROGRAMIRANJE VIDEOIGARA U BIBLIOTECI SFML

Diplomski rad

Voditelj rada:
Prof. dr. sc. Mladen Jurak

Zagreb, studeni 2019.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

U današnjem svijetu neobično je sresti neku osobu koja se nije susrela barem s jednom videoigrom. Igre više nisu samo na računalima i konzolama, prisutne su i na našim mobilnim uređajima. Uz razne moguće podjele jedna od postojećih podjela je na igre koje su u dvije (2D) i igre koje su u tri dimenzije (3D). 2D igre značajno su dominirale tržištem do 90-ih godina dok nismo razvili dovoljno dobru tehnologiju kako bismo mogli efikasno pokretati 3D igre.

Premda su najveće i najpoznatije igre današnjice najčešće 3D, 2D je još uvijek prisutan i često ga susrećemo, pogotovo u svijetu nezavisnih (*indie*) igara. Svijet 2D-a prepun je mogućnosti, pa se takve igre neće tako skoro prestati programirati. Pojavom modernijih, objektno orijentiranih jezika došle su i biblioteke koje omogućavaju pisanje koda za igre. Ti jezici i biblioteke dopuštaju brzo, efikasno i pregledno pisanje koda.

Jedna od takvih biblioteka za jezik C++ i 2D igre je *Simple and Fast Multimedia Library* ili SFML. Ona donosi moćne alate za programiranje igara koji omogućavaju jednostavno i brzo programiranje.

Sama biblioteka nije dovoljna za pisanje kvalitetnoga koda za igre, već je bitna i organizacija koda. Objektno orijentirani jezici omogućavaju nam organizaciju koda tako da je taj kod spreman za održavanje i nadogradnju. Još su jedna pomoć u tome oblikovni obrasci. To su isprobani načini organizacije koda koji rješavaju neke standardne probleme s kojima se susreću programeri.

Ovaj rad prikazuje biblioteku SFML, neke njezine osnovne dijelove i mogućnosti. Specifično se osvrće i na neke oblikovne obrasce koji su posebno efikasni u svijetu programiranja igara te ih prikazuje s problemima koje oni rješavaju. Vidjet ćemo kako nam ti obrasci omogućavaju pisanje kvalitetnog i efikasnog koda za videoigre.

Poglavlje 1

Kratki uvod u C++

C++ je programski jezik s podrškom za objektno orijentirano programiranje. Razvio ga je Bjarne Stroustrup 80-ih godina prošloga stoljeća. Napisao ga je kao proširenje programskom jeziku C, pa je originalno bio nazvan "C s klasama". Jezik se razvijao s vremenom i još uvijek se razvija. Zadnji standard koji smo dobili je C++17, a C++20 sljedeći je koji se očekuje. C++ nam omogućava upravljanje memorijom na niskim razinama i zato ga koristimo ako su nam bitne performanse i ograničeni smo resursima.

Među ostalim, danas se koristi često u velikim zahtjevnim sustavima, aplikacijama kojima su performanse bitne, i *desktop*-aplikacijama. Često je korišten za programiranje igara i neki poznatiji *game enginei* (aplikacije koje se koriste za razvoj igara i koje rade na višim razinama razvoja za razliku od SFML-a), poput *Unreala* i *Source enginea*, pisani su u njemu. C++ nam je bitan u ovome radu jer je SFML primarno C++ biblioteka za razvoj igara.

C++ danas je sveprisutan i razvojem jezika postat će još bolji i pristupačniji, pa je vjerojatna njegova duga upotreba i u budućnosti. Ipak, jedna od najvećih kritika jeziku njegova je kompleksnost i činjenica da je učenje toga jezika teže od većine drugih. Kada je pak u pitanju upravljanje memorijom, onda dobivamo bolje performanse, ali treba paziti pri takvom programiranju jer se mogu pojaviti neke greške koje je teško riješiti (npr. može se dogoditi curenje memorije ili *memory leak*). Zbog toga dosta programera radije piše kod u malo jednostavnijim jezicima ako performanse nisu toliko bitne.

Ovaj diplomski rad pretpostavlja osnovno poznavanje jezika C++ ili barem nekog drugog programskog jezika. Za početak učenja jezika C++ dobra je knjiga Johna Hortona *Beginning C++ Game Programming* [6] koja uči C++ baš preko SFML-a i programiranja videoigara.

Poglavlje 2

SFML

Kada je u pitanju programiranje igara u 2D-u, imamo različite mogućnosti u različitim jezicima. Neke biblioteke nam nude više, a neke manje mogućnosti. U nekima možemo upravljati dijelovima programa bolje nego u drugima. Jedna od biblioteka koja dopušta više mogućnosti je *Simple and Fast Multimedia Library* odnosno SFML. To je također biblioteka pisana u jeziku C++ koji sam po sebi omogućava preciznije upravljanje objektima i memorijom koju oni zauzimaju.

Kako bismo koristili SFML, potrebno je skinuti biblioteku sa službene stranice [4]. Tu možemo pronaći i detaljnu dokumentaciju za instaliranje i upotrebu. SFML biblioteka dostupna je na više platformi i čak je, zahvaljujući njenoj aktivnoj zajednici, dostupna za više programskih jezika, npr. *Java* jezik i *Python*, premda je službena verzija vezana s C i .Net jezicima. SFML je također biblioteka otvorenoga koda, što znači da svatko može čitati njegov kod u cijelosti i pogledati kako je nešto implementirano.

U ovom su poglavlju prikazani neki osnovni segmenti vezani sa SFML-om uz jedan osnovan primjer. Objašnjeni su i neki koncepti poput glavne petlje igre (*game loop*), iscrtavanja i organizacije koda.

2.1 Moduli SFML-a

Kao što joj samo ime govori (Jednostavna i brza biblioteka), SFML biblioteka jednostavna je za korištenje i omogućava brze programe i brzo pisanje kodova. Nudi jednostavno aplikacijsko programsko sučelje (engl. *application programming interface* ili API) koje je pregledno i pristupačno za korisnike.

Biblioteke za programiranje igara moraju biti i multimedijske. Ne možemo imati igru koja nema grafiku i zvučne efekte. Zato SFML nudi podršku za korištenje raznih medija u našim igrama. Ta je podrška podijeljena u 5 modula.

- **System** (Sustav): Ovo je centralni modul oko kojeg se vrte ostali moduli. Nudi nam vektore, satove, podršku za dretve i još mnogo toga.
- **Window** (Prozor): Omogućava stvaranje prozora i prikupljanje korisnikovih unosa kao što je unos preko miša ili tipkovnice.
- **Graphics** (Grafika): Nudi svu podršku potrebnu za 2D iscrtavanje. Preko njega možemo učitati teksture i prikazati ih na zaslonu. Možemo također prikazivati oblike i tekstove.
- **Audio** (Zvuk): Modul koji omogućava učitavanje zvukova i puštanje korisniku preko njegovih zvučnika.
- **Network** (Mreža): Podrška za slanje podataka preko mreže, bilo lokalne ili preko interneta.

Ovdje neće biti detaljan prikaz svih modula, već ćemo samo proći glavne klase. Za detaljnije informacije o svim modulima preporučujem pogledati dokumentaciju na službenoj stranici.

Za uvoz ovih modula koristimo naredbu include:

```
#include <SFML/Graphics.hpp>
```

Ili samo za dio nekog modula:

```
#include <SFML/Audio/Sound.hpp>
```

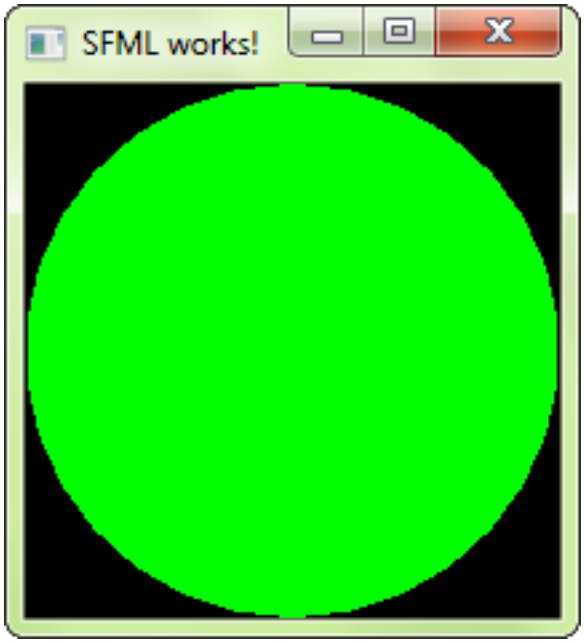
2.2 Osnovni primjer

Pogledat ćemo jedan jednostavan primjer programa pisanog u SFML-u i u komentarima ukratko objasniti što se događa.

```
#include <SFML/Graphics.hpp>

int main()
{
  // Otvara prozor s naslovom "SFML works!".
  sf::RenderWindow window(sf::VideoMode(200, 200),
  "SFML works!");

  // Definira krug.
  sf::CircleShape shape(100.f);
  // Popunjava krug zelenom bojom.
  shape.setFillColor(sf::Color::Green);

  // Petlja koja vrti igru dok je prozor otvoren.
  while (window.isOpen())
  {
    // Događaj koji provjerava zatvara li se prozor.
    sf::Event event;
    while (window.pollEvent(event))
    {
      if (event.type == sf::Event::Closed)
      window.close();
    }

    // Briše prethodno te crta i prikazuje novo.
    window.clear();
    window.draw(shape);
    window.display();
  }

  return 0;
}
```

Na slici 2.1 vidimo što se dobije kada se kod kompajlira i pokrene.

Vidimo da je otvoren prozor i da je u njemu nacrtan krug zelene boje. Prvo se inicijalizira taj zeleni krug i onda se u glavnoj petlji iscrtava u prozoru sve dok korisnik na neki



Slika 2.1: Prvi primjer

način ne zatvori prozor.

2.3 Prozor i događaji

Kao što se vidi u našem osnovnom primjeru, kako bismo pokrenuli igru i nešto crtali, trebamo otvoriti neki prozor. U tom prozoru odvijaju se svi glavni događaji naše igre i po njemu možemo crtati ono što nam treba. Za otvaranje i upravljanje prozorom koristimo `sf::Window` klasu. Jedan osnovan primjer izgledao bi ovako:

```
#include <SFML/Graphics.hpp>

int main()
{
  sf::Window window(sf::VideoMode(500, 500),
          "Moj naslov!");
  window.display();
  sf::sleep(sf::seconds(3));
  return 0;
}
```

U ovom primjeru inicijalizira se varijabla `window` kojoj kasnije pozovemo funkciju članicu `window.display()` kojom prikazujemo korisniku prozor. U ovom će se slučaju taj prozor otvoriti i nakon tri sekunde zatvoriti. Prozor će biti veličine 500 x 500 piksela i imat će naslov ”Moj naslov!”.

Problem koji tu postoji je da se prozor zatvara samostalno. Ono što nam je potrebno je da taj prozor ostane otvoren dok korisnik to želi i naravno, da se nešto događa u tom prozoru. SFML nam omogućava razne mogućnosti crtanja po zaslonu, ali ono što želimo je da tim likovima možemo na neki način upravljati. Tu u priču ulaze događaji. Jedan od tih je osnovni događaj zatvaranja prozora. Prvo popravljamo naš prijašnji kod i crtamo jedan osnovan oblik. Dodatno dodajemo osnovnu petlju (*Game Loop*) u kojoj će se događati sve bitne stvari u našoj igri.

```
#include <SFML/Window.hpp>

int main()
{
  sf::Window window(sf::VideoMode(200, 200),
          "SFML works!");

  sf::CircleShape shape(100.f);
  shape.setFillColor(sf::Color::Green);

  // Game loop
  while (window.isOpen())
```

```
    {
      sf::Event event;
      while (window.pollEvent(event))
      {
        if (event.type == sf::Event::Closed)
        window.close();
      }

      window.clear();
      window.draw(shape);
      window.display();
    }

    return 0;
}
```

Sada imamo prozor koji će ostati otvoren sve dok ga korisnik ne zatvori, a u njemu će se crtati zeleni krug. Jedan bitan dio, koji smo ovdje uveli, glavna je petlja igre. Njezin je zadatak držati prozor otvorenim kako bi korisnik mogao igrati igru dok god on želi. Svaka glavna petlja igre ima tri stadija:

1. Čitanje unosa – gledamo što je korisnik unio preko uređaja za unos i prozora
2. Ažuriranje slike – na osnovu korisnikova unosa ažuriramo stanje objekata na zaslonu
3. Iscrtavanje slike – crtamo novu sliku s novim stanjima objekata.

Unose čitamo korištenjem klase `sf::Event` preko koje određujemo događaje koje je korisnik pokrenuo. Ti će događaji najčešće utjecati na objekte na zaslonu, bilo to pomicanje objekata ili neke radnje (napad, obrana, magija i sl.). Te ćemo objekte crtati u novim (ili istim ako nije bilo promjene) stanjima. Za crtanje u prozoru koristi se *Double buffer* oblikovni obrazac o kojem će biti govora malo poslije. Ukratko, dok je nešto nacrtano u prozoru, u pozadini pripremamo ono što se crta sljedeće i kada dođe vrijeme, izbrišemo trenutno stanje i nacrtamo novo.

Jedna bitna klasa koja je vezana s prikazivanjem događaja na zaslonu je `sf::View`. Ona predstavlja 2D kameru koja definira koji je dio igre prikazan na zaslonu. Pogled možemo koristiti na više načina. Nudi nam opciju da pomičemo trenutni pogled i tako pokrećemo svijet. Na primjer, ako želimo da se naš lik penje nekamo, možemo pomicati pogled prema gore, a ne sve elemente igre prema dolje. `sf::View` nudi i sve ostale transformacije iz SFML-a. Također, možemo napraviti dva primjerka klase `View` i podijeliti zaslon na dva dijela pa tako imati igru za više igrača.

Događaji (*Events*)

Čitanje događaja koje stvara korisnik jedna je od najbitnijih značajki vezanih s programiranjem igara. Igre su po samoj definiciji interaktivne, pa je čitanje korisnikovih unosa ključno. Događaje možemo čitati preko `sf::Window` instance ili čitajući stanje samih događaja u stvarnom vremenu. Ako čitamo iz prozora, pozivamo `bool pollEvent(event)` koja vraća *true* dok god ima novih događaja koje treba čitati i popunjava varijablu `event` s podatcima samog događaja. Bitno je znati da može postojati više od jednog događaja (npr. možemo stisnuti i miš i tipkovnicu u isto vrijeme) i onda moramo paziti da uhvatimo sve događaje. Zbog toga će kod izgledati ovako:

```
// Game loop
while (window.isOpen())
{
  sf::Event event;

  // Dok postoje događaji koje treba obraditi.
  while (window.pollEvent(event))
  {
    // Obradi trenutni događaj.
    if (event.type == sf::Event::EventType::Closed)
    {
      window.close();
    }
  }

  // Ažuriraj i crtaj novu sliku.
}
```

Ovaj specifičan kod omogućava nam da pritisnemo ”X” na prozoru i zatvorimo ga. Vidimo da ćemo obradu događaja početi provjerom njihova tipa. Događaji su tipa `Event::EventType`, što je enumeracija unutar `Event` klase. Nakon što odredimo njegov tip, gledamo koji se specifični događaj toga tipa odvio. Zatim njega prevodimo u neku akciju (npr. skok nakon pritiska tipke *space*). Naše događaje možemo logički podijeliti u 4 grupe i u svakoj grupi postoje tipovi događaja:

1. Prozor – mijenjanje veličine ili fokusa i zatvaranje
2. Tipkovnica – pritiskanje ili puštanje tipke i unos teksta
3. Miš – micanje miša i pritiskanje ili puštanje tipki na mišu
4. *Joystick* – micanje miša, pritiskanje ili puštanje tipki i povezivanje *joysticka*.

Sada možemo provjeravati unose korisnika i na osnovu tih unosa utjecati na likove na zaslonu. U slučaju da koristimo `pollEvents`, od koristi nam može biti `switch` u kojem su slučajevi tipovi događaja.

```
void Game::processEvents()
{
  sf::Event event;
  while (mWindow.pollEvent(event))
  {
    switch (event.type)
    {
      case sf::Event::KeyPressed:
        if (event.key.code == sf::Keyboard::Space)
          shooting == true;
      break;
      case sf::Event::KeyReleased:
        if (event.key.code == sf::Keyboard::Space)
          shooting == false;
      break;
      case sf::Event::Closed:
        mWindow.close();
      break;
    }
  }
}
```

U ovom primjeru provjeravamo pritiskuje li korisnik tipku za pucanje koja je u ovom slučaju *space*. Na ovakav sličan način možemo pomicati naše likove po zaslonu, pucati iz oružja ili izvesti neku drugu radnju koju zahtijeva naša igra.

Drugi način čitanja događaja je u stvarnom vremenu. SFML nam omogućava provjeru stanja entiteta u svakom trenutku. Ti entiteti su miš, tipkovnica ili *joystick*. Ovdje ne obrađujemo događaje kako se pojavljuju, nego samo provjeravamo za svaki događaj je li se dogodio u trenutnom ciklusu. Ovaj pristup drukčiji je od obrade događaja koju smo do sada koristili, ali je odličan za primjere poput prošlog u kojem želimo obrađivati kretnje lika. Tako bi analizu toga pritiskuje li korisnik tipku za pucanje iz prošlog primjera sveli na sljedeće:

```
shooting = sf::Keyboard::isKeyPressed(sf::Keyboard::Space);
```

Ovakav pristup očitavanja događaja omogućava nam smanjivanje međusobne zavisnosti među klasama. Ako imamo `Game` klasu u kojoj je funkcija članica `run()` koja pokreće

igru, ne moramo u njoj čitati sve korisnikove unose. Ovime obradu korisnikova unosa možemo prepustiti klasama kojima je taj unos bitan.

Sada kada imamo mogućnost komunikacije s korisnikom, rezultat te komunikacije trebamo prikazati na zaslonu. Korisnik će pomicati likove, izvoditi radnje poput pucanja ili skakanja, odabirati neku opciju u izborniku i slično. Dalje nam je važno crtati nešto po zaslonu i to nacrtano mijenjati ovisno o korisnikovim željama.

2.4 Iscrtavanje

U početku svijeta igara nije bilo mnogo mogućnosti pri njihovom programiranju. Sve igre morale su biti jednostavne jer sami hardver nije imao mogućnosti za velike svjetove i kompliciranu grafiku. Ako je igra bila zahtjevnija, onda se često radilo o igri s jednostavnijom grafikom i mnogo teksta. Neke su avanture čak znale biti u potpunosti tekstualne. Danas svaka igra ima grafičko sučelje i prikazuje nešto na zaslonu, od samog lika igrača do informacija kao što je količina metaka i sl. Praktički je nemoguće zamisliti modernu igru bez grafike i zato će i ovdje biti korištena. SFML nam nudi mogućnosti prikazivanja oblika i tekstura na zaslonu. Možemo crtati neke oblike, poput krugova i kvadrata, ili učitavati i crtati naše slike pohranjene na tvrdom disku.

Prikazano je kako otvoriti prozor i čitati korisnikove unose, sada je potrebno to dalje obraditi. Taj unos možemo iskoristiti tako da prikažemo neki lik na zaslonu i njime upravljamo. Te likove i slike crtamo u prozoru koji smo otvarali u dosadašnjim primjerima, ali sada koristimo primjer klase `sf::RenderWindow`, a ne `sf::Window`. Možemo ponovno uzeti naš početni primjer kako bismo demonstrirali ovu funkcionalnost:

```
#include <SFML/Graphics.hpp>

int main()
{
  sf::RenderWindow window(sf::VideoMode(200, 200),
              "Crtanje");

  sf::CircleShape shape(100.f);
  shape.setFillColor(sf::Color::Green);

  while (window.isOpen())
  {
    sf::Event event;
    while (window.pollEvent(event))
    {
      if (event.type == sf::Event::Closed)
        window.close();
    }

    window.clear();
    window.draw(shape);
    window.display();
  }
```

```
    return 0;
}
```

Kao što vidimo u ovom primjeru, u svakom ciklusu prvo se briše sve u prozoru. Nakon toga će se naš oblik nacrtati i na kraju prikazati. Ovaj dio svake glavne petlje igre pisan je u *Double Buffer* oblikovnom obrascu. Jednu sliku prikazujemo dok drugu pripremamo za prikazivanje. To nam omogućava da se igra bez problema iscrtava na zaslonu, ali o tome više u jednom od idućih poglavlja.

Oblici i transformacije

Kada je u pitanju crtanje oblika, SFML nam nudi poznate oblike, poput kruga i pravokutnika, no možemo i sami napraviti vlastiti oblik. Te oblike stvaramo klasama `CircleShape` , `RectangleShape` i `ConvexShape`. `CircleShape` nam omogućava crtanje kruga tako da mu zadamo polumjer, a pomoću `RectangleShape` crtamo pravokutnike sa zadanom širinom i visinom. `ConvexShape` je oblik koji ima proizvoljni broj vrhova koji mi zadamo. Kao što je navedeno, broj vrhova je proizvoljan, ali pod uvjetom da čini vrhove oblika koji se može nacrtati.

Svaki oblik nasljeđuje klasu `Shape` i tako nadjačava funkcije članice `setFillColor()`, `setOutlineColor()` i `setOutlineThickness()`. Ove nam funkcije omogućavaju da postavljamo izgled oblika onako kako nam odgovara. Pokazat ćemo njihovu upotrebu u sljedećem primjeru:

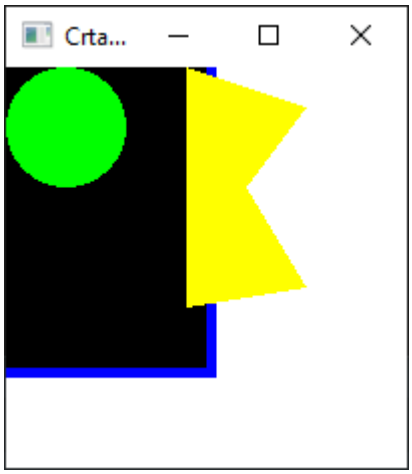
```
sf::CircleShape circleShape(30);
circleShape.setFillColor(sf::Color::Green);

sf::RectangleShape rectangleShape(sf::Vector2f(100, 150));
rectangleShape.setFillColor(sf::Color::Black);
rectangleShape.setOutlineColor(sf::Color::Blue);
rectangleShape.setOutlineThickness(5);

sf::ConvexShape pentagonShape;
pentagonShape.setPointCount(5);
pentagonShape.setPoint(0, sf::Vector2f(90, 0));
pentagonShape.setPoint(1, sf::Vector2f(90, 120));
pentagonShape.setPoint(2, sf::Vector2f(150, 0));
pentagonShape.setPoint(3, sf::Vector2f(120, 60));
pentagonShape.setPoint(4, sf::Vector2f(150, 110));
rectangleShape.setFillColor(sf::Color::Yellow);
```

Jedna bitna klasa koja se ovdje pojavljuje je `Vector2f`. Ona predstavlja 2D vektor koji sadržava dvije `float` vrijednosti. Postoje različite verzije vektora u SFML-u. Tu je `Vector2i`, koji sadrži dva cijela (*integer*) broja, i `Vector3i`, koji sadrži tri. Postoje i druge verzije koje sadrže različite vrijednosti i vektor koji nam omogućava pohranu tipova koje želimo, `Vector2<class>` i `Vector3<class>`. Vektore ćemo često koristiti u crtanju za označavanje dimenzija, smjerova, pozicija i slično.

Sada želimo na zaslonu prikazati naše oblike, pa ćemo koristiti objekt tipa `RenderWindow` koju smo naveli na početku ovog poglavlja:

```
// Nakon što sve izbriše, crta bijelu pozadinu.
window.clear(sf::Color::White);

window.draw(rectangleShape);
window.draw(circleShape);
window.draw(pentagonShape);

window.display();
```

U primjeru koristimo funkciju `RenderWindow::draw()` kojom crtamo u prozoru. Na slici 2.2 prikazan je rezultat pokretanja koda.

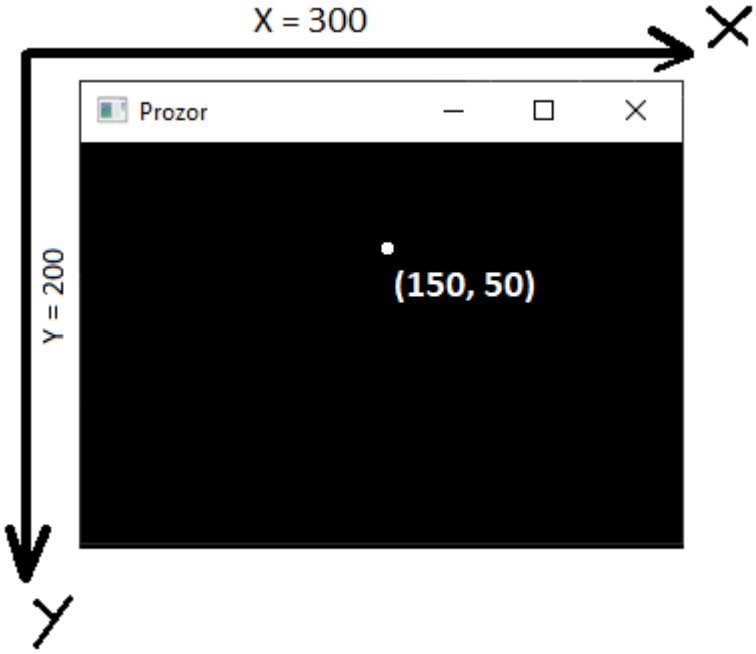


Slika 2.2: *Shape* primjer

Jedan detalj koji se vidi u ovome primjeru je bitnost redoslijeda crtanja oblika. Vidimo da je crtanje prije pozvano na pravokutniku nego na krugu, zato je pravokutnik na slici ispod kruga.

Transformacije

Samo crtanje likova nije nam dovoljno za pravu igru, također nam trebaju i neke transformacije. Želimo likove okretati, pomicati, povećavati i smanjivati. Prije transformiranja potrebno je razumjeti kako funkcionira koordinatni sustav u igrama. Njegova je orijentacija drukčija od standardne na koju smo navikli. Gornji lijevi kut prozora predstavlja točku (0, 0). Os y raste prema dolje, a os x raste prema desno. Kako to izgleda, možemo vidjeti na slici 2.3.



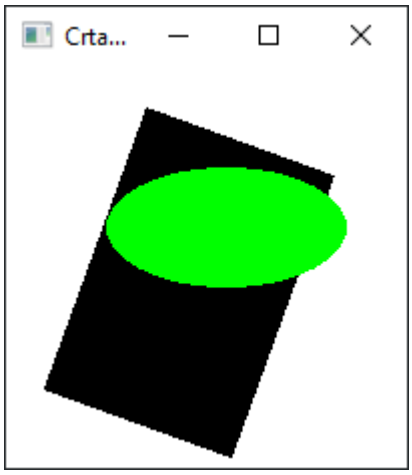
Slika 2.3: Koordinatni sustav

Sada znamo kako će se ponašati naše transformacije, pa ih možemo početi koristiti. Do sada smo koristili klasu `Shape` za sve naše oblike i da bismo ih crtali, ali ona nudi i još više mogućnosti. Ona nasljeđuje `sf::Transformable` koja nam nudi neke funkcije za transformiranje. `setPosition()` postavlja oblik na željenu poziciju, `setRotation()` koristimo za rotaciju oblika, a `setScale()` skalira. U nastavku je primjer upotrebe ovih funkcija:

```
sf::RectangleShape rectangleShape(sf::Vector2f(100, 150));
rectangleShape.setFillColor(sf::Color::Black);
rectangleShape.setPosition(sf::Vector2f(70,20));
rectangleShape.setRotation(20);
```

```
sf::CircleShape circleShape(30);
circleShape.setFillColor(sf::Color::Green);
circleShape.setPosition(sf::Vector2f(50,50));
circleShape.setScale(sf::Vector2f(2,1));
```

Pokretanjem programa dobivamo rezultat kao na slici 2.4. Vidimo da smo krug i pravokut-



Slika 2.4: Transformacija oblika

nik pomaknuli. Krug smo skalirali tako da bude dvostruko širi nego početno, a pravokutnik smo rotirali za 20 stupnjeva. Vidimo da nam transformacije daju velik broj mogućnosti u radu s oblicima. Uz funkciju `setPosition`, koja pomiče oblik na neku apsolutnu poziciju, možemo koristiti `Transformable::move()` tako da joj predamo vektor koji govori koliko želimo da se oblik pomakne od trenutne pozicije.

Još jedna funkcija koja je, uz transformacije, bitna je `Transformable::setOrigin()`. Nju koristimo kako bismo postavili izvor oblika. On se koristi kao centralna točka oko koje se obavljaju sve transformacije. Izvor se gleda relativno na oblik kojemu ga mijenjamo. U početku je za svaki oblik postavljen na (0, 0), tj. u gornji lijevi kut. Ako ostavimo tako, onda će se pri promjeni rotacije ona događati oko te točke. Ako promijenimo izvor na sredinu oblika, onda će se rotacija događati oko te sredine, tj. oblik će se rotirati oko svoga centra.

Teksture i *spriteovi*

Za učitavanje i prikazivanje slika u SFML-u primarno su zadužene tri klase. To su `sf::Image` i `sf::Texture` (učitavanje slika) i `sf::Sprite` (prikaz slika). Možemo koristiti i oblike za prikazivanje slika, ali zbog svoje se jednostavnosti *spriteovi* češće koriste.

Učitavanje slika

Za učitavanje slika u SFML-u koristimo klase `sf::Image` i `sf::Texture`. Razlika između ovih dviju klasa u mogućnostima je manipuliranja i prikazivanja. `Image` klasu koristimo za učitavanje i spremanje slika te manipulaciju piksela, a `Texture` koristimo za prikazivanje slika (*render*). One obje u sebi sadrže niz piksela koje možemo mijenjati. Zbog toga nam SFML nudi lako prebacivanje iz jedne klase u drugu, ali taj proces može biti skup, pa ga treba pažljivo koristiti.

U `Image` klasu možemo učitavati slike, ali također i crtati u nju. Nudi nam mogućnost stvaranja slike preko primjerka klase `sf::Color`:

```
sf::Image img;
img.create(100, 50, sf::Color::Blue);
```

Prva dva argumenta predstavljaju širinu i visinu, a treći predstavlja boju popune. Kao treći argument također možemo predati niz brojeva tipa `sf::Uint8` koji bi predstavljali pojedinačne piksele kao RGB vrijednosti.

Možemo, naravno, učitati slike s tvrdoga diska.

```
sf::Image img;
if(!img.loadFromFile("mojaSlika.png"))
{
  // Nije uspjelo učitavanje datoteke.
  return -1;
}
```

Bitan je dio `Image` to što nam omogućava razne funkcije upravljanja slikom, npr. mijenjanje vrijednosti individualnoga piksela pomoću `Image::setPixel()`. Nudi nam pristup nizu vrijednosti koje predstavljaju sve piksele ili jednom individualnom pikselu. Sliku možemo okrenuti vertikalno pomoću `Image::flipVertically()` ili horizontalno koristeći `Image::flipHorizontally()`. Sliku na kraju svega možemo spremiti na disk preko funkcije `Image::saveToFile()`.

Nakon što smo upravljali slikama, trebalo bi od njih napraviti teksture. Klasa `Texture` nudi nam slične funkcije kao i `Image`. Obje mogu pozivati `loadFromFile()` funkciju, postoji samo jedna bitna razlika. U klasi `Texture` postoji mogućnost prosljeđivanja još jednoga argumenta uz ime datoteke, a to su dimenzije i pozicija pravokutnika koji pred-

stavlja samo dio slike. Taj pristup omogućava postojanje više tekstura za jednu sliku i učitavanje samo dijelova. To nam štedi vrijeme i memoriju.

```
sf::Texture texture;
if(!texture.loadFromFile("mojaSlika.png",
        sf::IntRect(20, 0, 50, 50));
{
  // Nije uspjelo učitavanje datoteke.
  return -1;
}
```

Također možemo učitati teksturu iz primjerka klase `Image` tako da pozovemo funkciju `sf::Texture::loadFromImage()` i predamo joj kao argument sliku. Još nam je samo ostalo prikazati tu teksturu na zaslonu.

Teksture i oblici

Već smo spomenuli da postoji mogućnost učitavanja tekstura preko oblika. Naše teksture možemo postaviti na oblike poput pravokutnika i tako ih prikazivati na zaslonu. Oblicima možemo limitirati količinu slike koja će se prikazati. Kao primjer uzet ćemo prikaz šume koju vidimo na slici 2.5.

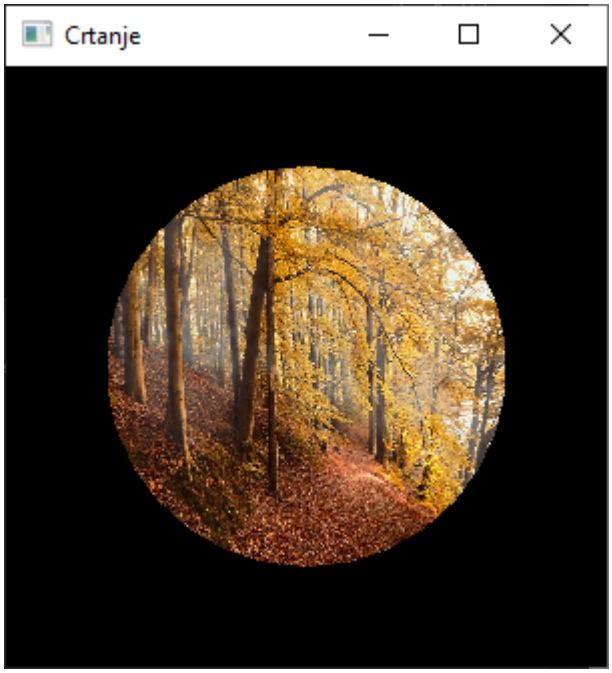
Slika 2.5: Šuma

Slijedi kod koji to omogućava:

```
#include <SFML/Graphics.hpp>

int main()
{
  sf::RenderWindow window(sf::VideoMode(300, 300),
              "Crtanje");

  sf::Texture texture;
```

```
    texture.loadFromFile("autumn-forest.jpg");

    sf::CircleShape circleShape(100);
    circleShape.setTexture(&texture);
    circleShape.setPosition(50, 50);

    // Game loop
}
```

Izvršavanjem ovoga koda dolazimo do rezultata na slici 2.6.



Slika 2.6: Šuma u krugu

Uz oblike postoji opcija crtanja tekstura preko *spriteova*.

Spriteovi

Spriteovi su, poput oblika, površina po kojoj crtamo teksture i među njima postoje razlike. Prva je razlika da se *sprite* uvijek iscrtava kao pravokutnik, pa njime ne možemo rezati dio teksture kao s oblicima. Njegova veličina je onolika kolika i teksturina. Možemo ju jedino mijenjati tako da koristimo transformacije. `sf::Sprite` nasljeđuje `Transformable` i `Drawable`, pa ih možemo crtati i na njima raditi transformacije isto kao i kod oblika.

Bez obzira što oblici nude više mogućnosti, svejedno ćemo češće koristiti *spriteove* zbog njihove jednostavnosti.

Njihova je svrha samo učitati teksturu i prikazati je (uz neke osnovne transformacije). Bitno je još znati da, dok oblike možemo koristiti i bez tekstura, *spriteovi* ih zahtijevaju za korištenje.

Pogledajmo jedan osnovan primjer:

```
sf::Texture texture;

// Učitaj teksturu.

// Shape
sf::RectangleShape rect(sf::Vector2f(200, 100));
rect.setTexture(&texture);

// Sprite
sf::Sprite sp(texture);
```

Sada vidimo tu jednostavnost. Bitno je i ovdje naglasiti da, ako želimo nacrtati cijelu teksturu preko pravokutnika, situacija se još malo komplicira. Bez obzira na jednostavnost *sprite* svejedno nudi dovoljno mogućnosti da ga ima smisla koristiti.

Pomicanje oblika

Oblici nisu zanimljivi ako su nepomični i ako se sami pokreću. Želimo imati mogućnost upravljanja tim oblicima. Ovdje ćemo iskoristiti događaje i oblike kako bismo napravili jednu interaktivnu igru. Događaje ćemo gledati u realnom vremenu jer je, kao što je već objašnjeno, praktičnije za ovakve primjere. Jedna mana ovakva pristupa je što će učitavati unose čak i ako korisnik nije fokusiran na glavni prozor, pa je potrebno imati to na umu pri korištenju.

Prikazat ćemo ovdje jedan osnovan primjer da bismo vidjeli kako često u praksi pomičemo likove. Lik će se pomicati dolje ako pritisnemo tipku za dolje, a rotirat će se za jedan stupanj oko vlastite osi po sličici ako držimo slovo "R".

```
#include <SFML/Graphics.hpp>

int main()
{
  sf::RenderWindow window(sf::VideoMode(500, 500),
              "Crtanje");

  sf::RectangleShape rectangleShape(sf::Vector2f(50, 50));
  rectangleShape.setFillColor(sf::Color::White);
  rectangleShape.setPosition(sf::Vector2f(100, 100));
  rectangleShape.setOrigin(sf::Vector2f(25, 25));

  while (window.isOpen())
  {
    sf::Event event;
    while (window.pollEvent(event))
    {
      if (event.type == sf::Event::Closed)
        window.close();
    }

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::R))
    {
      rectangleShape.rotate(1.f);
    }

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down))
    {
```

```
            rectangleShape.move(sf::Vector2f(0, 1));
        }
        window.clear();
        window.draw(rectangleShape);
        window.display();
    }

    return 0;

}
```

Problem koji se javlja u ovome primjeru je taj da će na bržim računalima biti više rotacije i pokreta u sekundi nego na sporijima. To su problemi koje treba rješavati u glavnoj petlji igre. O tome više u jednom od idućih poglavlja. Jedno od mogućih rješenja tog problema u SFML-u je korištenje funkcije `Window::setFramerateLimit()`. Ako postavimo limit, funkcija `Window::display()` usporit će crtanje po potrebi. Kao što ćemo vidjeti poslije, to nije najbolje rješenje, pogotovo kada su u pitanju lošija računala koja teško pokreću našu igru.

Sada znamo crtati neke osnovne oblike i pokretati ih. Možemo u svojoj igri konstruirati pravila fizike i određivati što korisnik može ili ne može napraviti. To su centralni dijelovi velike većine igara i moramo se potruditi da rade dobro.

U ovome primjeru korišteni su oblici, ali isto bi bilo i sa *spriteovima*, samo bismo umjesto crtanog oblika učitali teksturu i prikazali je preko *spritea* i na kraju taj *sprite* transformirali.

2.5 Zvuk

Zasad su pokazani centralni dijelovi bez kojih igra ne može. U većini slučajeva poželjno je poboljšati to iskustvo zvukom. On omogućava da se korisnicima naznači da se nešto dogodilo, poput pucnja iz oružja ili koraka neprijatelja. Korisniku se može i puštati glazba koja uvijek upotpunjuje sve dijelove igre i čini ih življima. Zvuk omogućava slanje povratne informacije korisniku i izazivanje određene emocije. Ovdje je objašnjeno na koje načine SFML omogućava korištenje zvuka.

Slično kao i kod slika, postoje dvije mogućnosti učitavanja zvuka, a to je preko klasa `sf::Music` i `sf::Sound`. Razlika između te dvije klase osnovna je, ali vrlo bitna. `sf::Sound` učitava glazbu pohranjenu na tvrdom disku u radnu memoriju te potom pušta zvuk iz memorije. `sf::Music` otvara tok (engl. *stream*) prema datoteci na tvrdom disku i učitava dio po dio zvučne datoteke. Obje nasljeđuju klasu `SoundSource` preko koje imaju neke osnovne glazbene funkcionalnosti. Također pokreću zvuk u zasebnoj dretvi kako ne bi blokirale trenutnu. Podržani formati su WAV, OGG/*Vorbis* i FLAC. Bitno je primijetiti da SFML ne podržava MP3 format.

Očito je `Sound` klasu bolje koristiti kada su u pitanje manje zvučne datoteke koje će se često ponavljati (poput zvuka skoka ili pucnja). Ta se klasa češće koristi jer su većina zvučnih datoteka u igrama efekti. Ovom klasom nakon učitavanja u RAM možemo, kada god treba, dobiti brz pristup zvučnom zapisu. `Music` koristi se kada su u pitanju velike datoteke, a programer je ograničen količinom memorije. Budući da ova klasa učitava dijelove datoteke, ona ima prirodni zastoj zbog posla koji obavlja. Ponuđena nam je i mogućnost puštanja 3D zvukova. Takvi se zvukovi čuju iz više različitih smjerova i daju dojam treće dimenzije zvuka. Unutar SFML-a možemo i snimati zvuk koristeći klasu `SoundRecorder`, ali ovdje će fokus biti na puštanju zvuka.

sf::Sound

Klasa `sf:Sound` koristi se kao omotač oko primjerka klase `sf::SoundBuffer`. `SoundBuffer` predstavlja zvuk u memoriji, a `Sound` koristimo za puštanje te glazbe. Ova struktura i odnos između klasa isti su kao i kod `Texture` i `Sprite` klasa. To nam omogućava da jedan primjerak klase `SoundBuffer` koristimo više puta i tako uštedimo na memoriji.

Prvo pomoću `SoundBuffer` učitamo datoteku na disku preko funkcija oblika `loadFromX()`. Nama je najbitnija `loadFromFile`. Nakon što konstruiramo `Sound` objekt kojemu proslijedimo zvuk koji smo prethodno učitali, možemo ga puštati, zaustavljati ili pauzirati. Nude se i funkcije koje javljaju status pjesme (je li pauzirana i sl.) i još neke druge. Osnovan primjer isječka koda izgledao bi ovako:

```
sf::SoundBuffer sBuffer;
```

```
// Vraca "false" ako se dogodila greska pri ucitavanju.
if(!sBuffer.loadFromFile("MojZvuk.ogg"))
{
  return -1;
}


sf::Sound sound(buffer);

sound.setLoop(true);
sound.play();
// Pomice zvuk dvije sekunde naprijed.
sound.setPlayingOffset(sf::seconds(2.f));
```

sf::Music

Klasa `sf::Music` je klasa koju koristimo zasebno (ne trebamo koristiti drugu klasu) za puštanje zvukova. Preporučuje se kod većih zvukova kada smo ograničeni memorijom. Za upravljanje glazbom koristimo iste metode kao i u klasi `sf::Sound`. Za razliku od `SoundBuffer` klase koja učita cijelu pjesmu u memoriju, ova klasa otvara datoteku za pristup, tj. koristi `openFromFile()`, a ne `load()`. Isječak koda:

```
sf::Music music;
if(!music.openFromFile("MojZvuk.flac"));
  return -1;
music.play();
```

2.6 Mrežna komunikacija

Pojavom i populariziranjem interneta javila se želja za njegovim korištenjem u nekim igrama. On nam omogućava brzu komunikaciju s drugim računalima diljem svijeta i time igre možemo igrati zajedno ili protiv drugih ljudi. Danas je veza s internetom čest dio igara. Jedna je bitna značajka da preko interneta skidamo nova ažuriranja igara u redovnim intervalima i tako naše igre dobivaju ili popravke *bugova* ili nove sadržaje, što im produžuje vijek. Prije su se nova izdanja morala kupovati u trgovinama kako bi se riješili takvi problemi. Unatoč tome najbitnija novina koju je internet donio u igre je komponenta igranja s drugim igračima. To je omogućilo stvaranje kompetitivnih mrežnih igara. Neke su igre danas čak isključivo za igranje preko interneta. Iz svega toga izrasla je i e-sport scena u kojoj se može natjecati u igrama i osvojiti novčane nagrade.

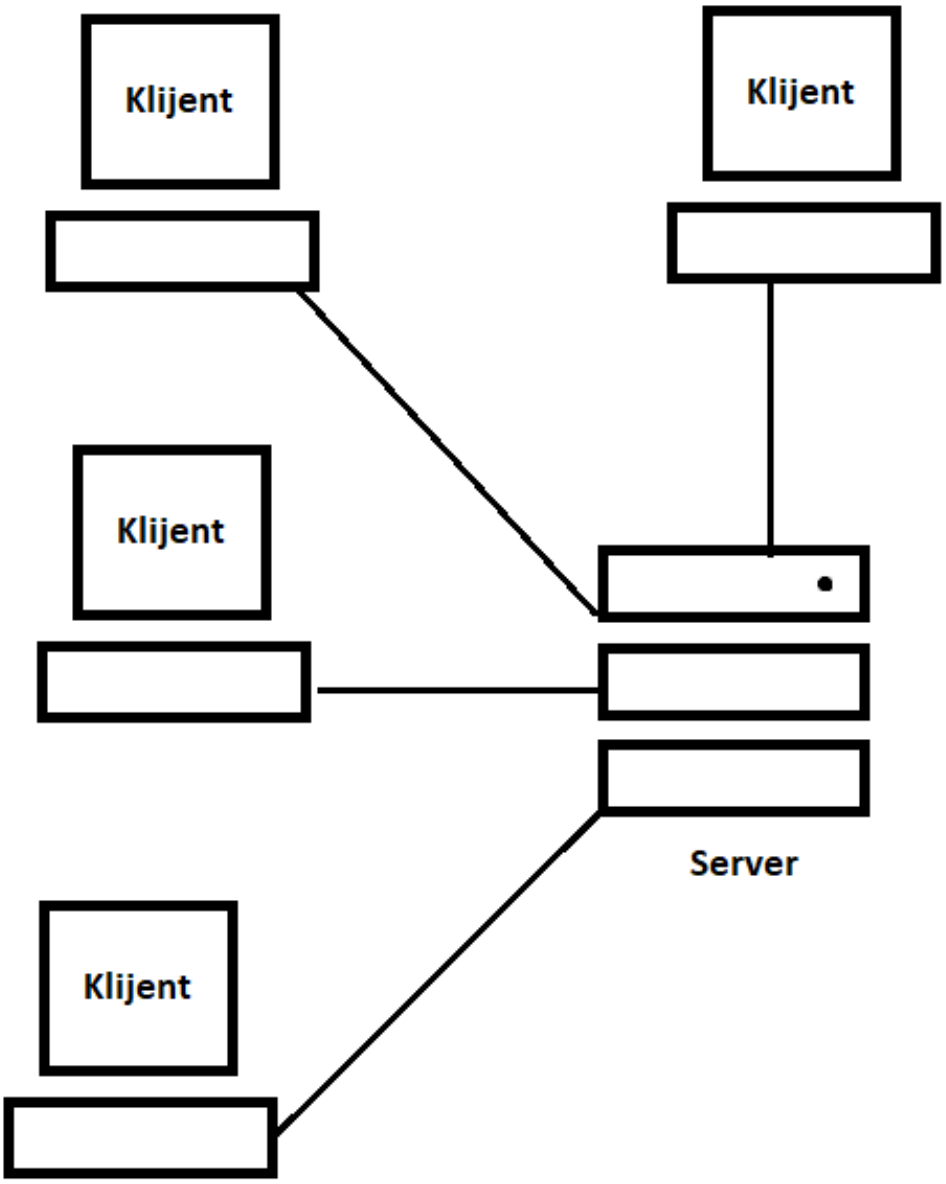
SFML nudi razne mogućnosti kada je u pitanju komunikacija preko interneta. Ovdje će biti prikazane osnovne. Arhitektura koju ćemo koristiti za komunikaciju bit će klijent-server. Ona funkcionira tako da se klijenti spajaju na jedan centralni server koji sinkronizira događaje i šalje im trenutno stanje. Kako bismo znali kamo putuje informacija, trebamo znati IP adresu koja će voditi naš paket do odredišta. U kontekstu igara klijenti bi slali informacije poput pucnja, skoka i sl., a server bi primao informacije i analizirao ih. Tako bi znali je li metak nekoga pogodio ili ne. Nakon toga server šalje trenutno stanje u igri svim klijentima. Taj je odnos prikazan na dijagramu 2.7.

Kada je u pitanju transport, postoje dva protokola: TCP i UDP. Razlike su u pouzdanosti i performansama. Oni rješavaju probleme prebacivanja informacija s jednog računala na drugo. U svakom od primjera slat ćemo neke jednostavne podatke, ali se u pravilu šalju paketi podataka. U SFML-u je njihova podrška preko klase `sf::Packet` i u nju možemo pohraniti više informacija različita tipa i slati ih preko interneta. Nadalje, u svakome od protokola drukčije se programira, pa ćemo ih pogledati zasebno.

TCP

Transmission Control Protocol (TCP) koristimo kada želimo osigurati stizanje paketa na njegovo odredište. To ga čini pouzdanim protokolom. Zbog toga se šalje više informacija nego što originalno postoji jer pratimo je li stigao svaki paket ili ne. Ako jedan paket ne stigne kamo treba, onda se on šalje ponovno sve do njegova dolaska. Protokol tako osigura da dođu svi paketi, i to u pravilnom redoslijedu. TCP je zbog svega ovoga sporiji i koristimo ga kad god brzina nije presudna. U većini slučajeva i neće biti, osim u brzim akcijskim igrama u kojima je svaki djelić vremena bitan. Većina interneta komunicira preko ovoga protokola zato što je uglavnom važnija točnost pristiglih informacija nego gubitak malo vremena.

Za spajanje nam trebaju IP adresa odredišta i *port* na koji se spajamo. TCP koristimo



Slika 2.7: Klijent-server model

preko dvije klase: `sf::TcpSocket` (utičnica), koja uspostavlja vezu (odnosno klijent), i `sf::TcpListener`, koja prihvaća vezu (server). Ako je veza uspješna, `sf::TcpListener` otvara `sf::TcpSocket` koji se spaja na prethodnu utičnicu i uspostavlja vezu između njih.

Prvo gledamo primjer klijenta:

```
sf::TcpSocket socket;

// Funkcija connect vraca status konekcije.
if (socket.connect("192.168.1.10",
  12345) != sf::Socket::Done)
{
  // Neuspjela veza.
  return -1;
}

// Uspostavljena je veza, pa saljemo poruku.
const int mSize = 100;
char msg[mSize] = "Ja koristim SFML. Ti?";
if( socket.send(msg, mSize) != sf::Socket::Done)
{
  // Greska pri slanju.
}
```

Zatim gledamo primjer servera:

```
// Slusa na portu 12345.
sf::TcpListener listener;
listener.listen(12345);

// Prihvaćamo vezu.
sf::TcpSocket socket;

if (listener.listen(socket) != sf::Socket::Done)
  return -1;

// Uspostavljena je veza, pa primamo poruku.
const std::size_t mSize = 100;
char msg[mSize];
std::size_t readSize;
if( socket.recieve(msg, mSize,
  readSize) != sf::Socket::Done)
```

```
{
  // Greska pri primanju informacija.
  return -1;
}

// Obrada podataka.

socket.disconnect();
```

TCP stvara vezu između dvije utičnice i kada ih spoji, obavlja razmjenu podataka. Nakon obrade zatvaramo utičnicu i prekidamo vezu.

UDP

User Datagram Protocol (UDP) nepouzdani je protokol koji ne prati ima li paketa koji nisu stigli na svoje odredište ili je li redoslijed paketa pravilan. Zbog toga je ovaj protokol brži i zahtijeva manje memorije za svaki paket. Tada i odredišno računalo ne treba čekati izgubljene pakete, nego samo nastavlja izvršavanje. Koristimo ga kada je u našim aplikacijama bitna brzina. Dakle, ako treba mnogo informacija uskladiti u kratkom vremenu, onda nam je UDP poželjniji.

Slanje podataka preko UDP-a slično je kao i preko TCP-a, samo koristimo `sf::UdpSocket`. Za razliku od TCP-a u kojemu jedna utičnica može biti povezana samo s jednom drugom, ovdje jedna utičnica može slati podatke na više odredišta. Točnije, veza se ne uspostavlja, nego se samo pošalju podatci.

Opet prvo gledamo primjer klijenta:

```
sf::UdpSocket socket;

// Saljemo poruku.
const int mSize = 100;
char msg[mSize] = "Ja koristim SFML. Ti?";
if( socket.send(msg, mSize, "192.168.1.10",
   12345) != sf::Socket::Done)
{
  // Greska pri slanju.
}
```

Zatim gledamo primjer servera:

```
sf::UdpSocket socket;
```

```
// Vežemo utičnicu za port da zna gdje primiti podatke.
socket.bind(12345);

// Primamo poruku.
const std::size_t mSize = 100;
char msg[mSize];
std::size_t readSize;
sf::IpAdress clientIP;
unsigned short remotePort;
if( socket.recieve(msg, mSize, readSize
  clientIP, remotePort) != sf::Socket::Done)
{
  // Greska pri primanju informacija.
  return -1;
}

// Obrada podataka.

socket.unbind();
```

Ovo je osnovan primjer mrežne komunikacije u UDP-u, a prethodno smo pokazali i u TCP-u. Vidimo sada kako proširenjem ovoga i korištenjem paketa možemo imati kompleksne programe koji bi nam omogućili igranje igara preko mreže.

Poglavlje 3

Oblikovni obrasci u igrama

U početcima programiranja jezici su bili proceduralni. Ti su programi bili brzi i efikasni, ali takav način pisanja nepraktičniji je što je kod veći. Tu nam pomažu klase. S klasama i polimorfizmom možemo imati velike i pregledne kodove koji su pogodni za napredak i održavanje. Također nam pomažu kod ponovne upotrebe koda jer su takvi dijelovi često odvojeni od ostatka koda, pa se lako iskoriste negdje drugdje. Još jedan segment koji nam u tome može pomoći su oblikovni obrasci. To su provjereni načini pisanja koda koji rješavaju neke određene probleme te ukazuju na dobru praksu pisanja koda.

Obrasci su bitni i za svijet igara. Pomažu nam da naše igre rade dobro i da su otvorene dodavanju novih dijelova igre. Ovdje će biti prikazani neki oblikovni obrasci koji se često koriste u igrama te su uvelike korisni. Naravno, postoje mnogi drugi obrasci koji mogu pomoći u pisanju koda za naše igre [9] ili za bilo koji program koji možda pišemo u nekom objektno orijentiranom jeziku. [3]

3.1 *Game loop*

Motivacija

Game loop centralni je obrazac za igre koji nam služi za pokretanje igara. Dok pokretači igre sami obavljaju glavnu petlju, SFML nam omogućava da osobno programiramo glavnu petlju igre. Svaki put kada se odvrti petlja, naš program učita korisnikov unos, ažurira stanje igre i prikazuje ju korisniku. Ovo je najbitniji oblikovni obrazac kada je u pitanju programiranje igara i skoro ga svaka igra ima.

Kada su u pitanju skripte koje se pokreću u komandnoj liniji, radi se o programima koji, nakon što se pokrenu, odrade što trebaju sekvencijalno i prestanu s radom. Dosta programa s grafičkim sučeljima funkcionira slično. Ne rade ništa dok čekaju da korisnik ne ponudi neki unos. Kada korisnik nešto unese, aplikacija odradi što treba i opet čeka novi unos.

To nam predstavlja problem jer su igre interaktivne i nešto se konstantno događa na zaslonu. Igra se treba kretati čak i kada nema korisnikova unosa. Ako igramo igru u kojoj upravljamo avionom, on neće čekati da korisnik nešto unese, već se kreće i bez unosa. To je zapravo prva ključna ideja petlje igre, ona čita korisnikov unos bez da ga čeka. Točnije, petlja se vrti i obavlja sve radnje, ali neće stati s izvođenjem kako bi čekala korisnikov unos. U našem je primjeru to bilo:

```
while (true)
{
  processInput();
  update();
  render();
}
```

Vidimo da svaki put kada petlja prolazi, naša igra čita korisnikov unos, ali ako nema unosa, onda `processInput()` neće ništa napraviti. Funkcija `update()` ažurirat će stanje na zaslonu neovisno o tome ima li korisnikova unosa ili ne. Ako ima, onda ažurira u skladu s unosom. Na kraju svega toga funkcija `render()` nacrtat će na zaslon sve potrebno. Dodatno, negdje u kodu bit će uvjet koji će omogućiti izlazak iz ove petlje. Ovo je bio osnovni konceptualni primjer. Imali smo takav u poglavlju koje nas je uvelo u SFML, gdje se u petlji samo čekalo da korisnik zatvori prozor.

Sada postavljamo logično pitanje: koliko se brzo ova petlja izvodi? Petlja će se izvoditi onoliko puta koliko računalo dopušta da se izvodi. Jedan izraz koji se ovdje često pojavljuje su sličice po sekundi ili FPS (*frames per second*). Taj pojam označava koliko sličica u jednoj sekundi naša igra crta, odnosno koliko se puta u sekundi petlja igre vrti. Više sličica znači fluidniju igru i sliku, a manje sličica može biti čak neigrivo. Naravno, ne znači da zaslon može toliko FPS-a i prikazati. To je ovisno o brzini osvježavanja zaslona.

Često želimo i upravljati tim brojem. Očito je da mali broj sličica nije poželjan, ali isto tako ne želimo nužno uvijek da je velik. Naime, u tom slučaju naša igra može opteretiti logičke jedinice računala i grafičke kartice. Ogromne su količine FPS-a ponekad i nepotrebne, pa ćemo često ograničiti broj izvođenja petlje u sekundi. Na starijim računalima ovo je bio problem kada su igre ovisile o hardveru na kojem su se izvodile. Brzina nekih igara znala je biti uvjetovana time na kojem računalu se izvodila. Danas to više nije tako i zapravo je jedan od bitnih poslova petlje igre održavanje konzistentne brzine igre neovisno o hardveru.

Bitno je imati na umu i da će se ova petlja vrtjeti velik broj puta u sekundi, zato je važno učiniti je što efikasnijom.

Primjer

Primjer koda za ovaj obrazac poprilično je izravan. Ovdje ćemo se više baviti nekim detaljima oko same implementacije glavne petlje te prednostima i manama određenih implementacija. Pozivat ćemo neke standardne funkcije u igrama, ali ih nećemo implementirati jer u tome ipak nije poanta ovog obrasca. Fokus će biti na samoj glavnoj petlji.

Prvo ćemo pogledati najosnovniji oblik ovakve petlje kakav je već prikazan:

```
while (true)
{
  processInput();
  update();
  render();
}
```

Ovakva petlja vrtjet će se onoliko brzo koliko je računalo u mogućnosti. Problem s takvim pristupom je što nemamo kontrolu nad brzinom kojom će se igra vrtjeti. Na brzim će se računalima igra vrtjeti prebrzo, a na slabijim će računalima biti prespora za igranje. Ako je u pitanju pokretanje nekoga lika, onda će se na brzom računalu lik prebaciti preko cijeloga zaslona u manje od pola sekunde, a na slabom računalu trebat će nekoliko sekundi. Zbog toga je poželjno imati određenu kontrolu nad glavnom petljom.

Upravljanje glavnom petljom

Prvi pristup rješavanju ovog problema ujedno je i najjednostavniji. Strategija je određivanje koliko ćemo puta izvršiti petlju i zatim čekanje određene količine vremena. Dakle, ako želimo imati petlju koja traje 20 ms, a na jednom računalu se izvede u 12 ms, onda ćemo pozvati `sleep(8 ms)` i time dobiti željenu duljinu petlje. Duljina petlje zapravo znači koliko će sličica (ili *frameova*) biti poslano na crtanje (*render*). To bi značilo da zapravo ograničavamo igru da postiže željeni FPS. Taj bi kod izgledao ovako:

```
double time_per_loop = 1000/desired_FPS;
while (true)
{
  double loop_start = getCurrentTime();
  processInput();
  update();
  render();
  sleep(loop_start + time_per_loop - getCurrentTime());
}
```

Ovdje funkcija `sleep()` omogućava da se petlja ne izvršava prečesto. Sada dolazimo do velike mane ovoga pristupa, a to su računala kojima će trebati previše vremena da izvrše

petlju. Igra će tada usporiti i neće biti željene brzine. Jedno rješenje je da ponudimo korisnicima više opcija za grafiku i detalje u igri tako da petlja traje kraće, ali uvijek će postojati još slabija računala koja i na najmanjoj rezoluciji neće moći postići zadovoljavajuće rezultate. Zato ta opcija i nije najpoželjnija.

Pomicanje ovisno o vremenu

Problem u prošlom rješenju nastajao je kada je trebalo više vremena za obradu jedne sličice. Jedno rješenje toga pomicanje je igre za vrijeme koje nam nedostaje. Dakle, ideja je ta da, ako petlja traje 20 ms, a nama treba više vremena za obradu, onda igru pomaknemo za 20 + višak ms. Točnije, ako se lik pomakne 100 piksela u 20 ms, a nama je potrebno 30 ms, onda ćemo pomaknuti lik 150 piksela u jednom izvođenju petlje.

Trebamo odlučiti koliko će se igra pomaknuti za određeno vrijeme i onda gledamo koliko je stvarno trebalo vremena za petlju. Ovisno o tome koliko je vremena prošlo, toliko će naša funkcija pomaknuti igru. Kod bi izgledao ovako:

```
double last_time = getCurrentTime();
while (true)
{
  double current_time = getCurrentTime();
  double elapsed_time = current_time - last_time;
  processInput();
  update(elapsed_time);
  render();
  last_time = current_time;
}
```

Vidimo da `update` funkcija prima vrijeme koje je prošlo. U toj ćemo funkciji pomaknuti igru ovisno o prošlom vremenu. Za udaljenost koju će lik prijeći to bi bila brzina lika pomnožena s vremenom trajanja petlje. To znači da bi se tijekom dužeg vremena lik više pomicao. Time postižemo jednako pomicanje igre na računalima različite snage.

Ovakav pristup donosi sa sobom probleme zbog toga što računala zaokružuju pri operacijama s tipom `double`. Zamislimo da jedan korisnik u jednoj sekundi prođe petlju 10 puta, a drugi je prođe 60 puta. Korisnik koji prođe glavnu petlju više puta obavit će više operacija i podložniji je većem broju grešaka. U tom trenutku možemo imati dva događaja koja izgledaju drukčije iako su izvedena na isti način. Zato nam je teško predvidjeti što se može dogoditi na različitim računalima jer manje greške u računanju mogu dovesti do većih problema pri izračunu pravila fizike u igri. Zbog te nestabilnosti nije pametno koristiti ovo rješenje.

Više koraka u jednom

Vidjeli smo da nam je problem što ili prečesto mijenjamo igru ili prerijetko. Taj problem možemo riješiti tako da uvijek crtamo posljednju sliku (što nije zahtjevna operacija), ali da ažuriranje slike obavimo ili više puta u jednoj petlji ili nijednom. To nam omogućava fiksan broj ažuriranja koji mi odaberemo. Na sporijim računalima ažuriranje će biti češće, a na bržima rjeđe, no ukupan broj bit će jednak, unatoč tome što će brže računalo i brže crtati trenutno stanje. Ovdje ćemo pratiti koliko je vremena prošlo od zadnje sličice, a ažuriranja ćemo provoditi ovisno o tome. Kod izgleda ovako:

```
double previous_time = getCurrentTime();
double lag = 0.0;
while (true)
{
  double current_time = getCurrentTime();
  double elapsed_time = current_time - previous_time;
  previous_time = current_time;
  lag += elapsed_time;
  processInput();
  while (lag >= time_per_update)
  {
    update();
    lag -= time_per_update;
  }
  render();
}
```

Sada vidimo da će se `update` funkcija pozivati više puta ako računalu treba više vremena za obradu, a na brzom računalu neće uopće ažurirati par sličica. Svakim novim prolaskom petljom provjeravamo koliko je vremena prošlo od zadnjeg prolaska i akumuliramo to vrijeme. Ako je računalo sporo, onda će zaostatak biti veći od željenog vremena, pa će se izvršiti onoliko puta koliko je zaostatak veći. Ako je računalo prebrzo, `lag` će se povećavati tijekom više prolazaka petljom dok ne dođemo do željene veličine, tek tada će se stanje ažurirati. Ovime smo postigli jednak broj ažuriranja na različitim računalima. Na nekim će računalima prikazivanje možda biti sporije, a igra malo usporenija, ali broj ažuriranja igre bit će isti.

Pri ovoj implementaciji mogu se javiti problemi zbog različitih vremena ažuriranja i crtanja. Ako se crtanje događa na pola puta između dva ažuriranja, onda možemo očekivati da se igra nalazi u međustanju, ali će zapravo biti iscrtana u stanju prvog ažuriranja. To možemo popraviti tako da crtanje, tj. `render()` funkcija, prima kao argument vrijeme od zadnjeg ažuriranja te crta na osnovi toga. Ovo može dovesti do toga da se objekti nalaze u

nedozvoljenim pozicijama (npr. objekt uđe par piksela u zid), a to će primijetiti tek iduće ažuriranje. Srećom, ovakve su greške minimalne u praksi, pa je ovakav pristup i dalje praktičan.

Zaključak

Za razliku od drugih, ovaj je obrazac takav da većina igara ne može funkcionirati bez njega. Dobrodošao je čak i u igrama koje ga tehnički ne trebaju. Takve igre bez njega ne bi mogle imati grafičke efekte ili zvukove. Bilo da sami pišemo petlju svojom bibliotekom ili da je *game engine* vlasnik petlje, svejedno nam je potreban kako bi naša igra bila igra.

3.2 *Double Buffer*

Motivacija

Kada programiramo igru, potrebno je neke informacije prikazati igraču. Pozadinu, likove, efekte i ostalo treba crtati po zaslonu, ali to nije jednostavan zadatak. Objekti na zaslonu prikazuju se tako da prvo crtamo od pozadine prema naprijed. Ono što želimo je da je to crtanje fluidno, brzo i efikasno. Želimo da se svaka prikazana sličica na zaslonu vidi dobro i u cijelosti.

S tim problemom pomaže nam oblikovni obrazac *Double Buffer*, odnosno dvostruka međupohrana kojom jednu sličicu pripremamo dok se druga prikazuje. Da bi razumjeli zašto i kada je to potrebno, moramo se malo upoznati s načinom na koji računalo prikazuje sličice na zaslonu. Ono što slijedi je pojednostavljenje toga problema.

Računalo prikazuje sličice na način da crta piksel po piksel i tako red po red piksela od vrha do dna. Kada dođe na dno, vrati se na početak i počne ponovno crtati. Brzinu crtanja određuje brzina osvježavanja monitora koja najčešće iznosi 60 puta u sekundi. Računalo određuje piksele koje će crtati tako da čita *framebuffer*. To je niz piksela u radnoj memoriji koji sadržava informacije o svakom pikselu koji treba nacrtati. Ako ne koristimo ovaj oblikovni obrazac, onda zapravo pri izračunima nekih vrijednosti (kao što je pozicija ili rotacija lika) u tom trenutku crtamo direktno u *framebuffer*.

To znači da čitamo iz toga niza ono što treba prikazati na zaslonu i tu se zapravo javlja problem koji se ovdje pokušava riješiti. Do njega dolazi ako brzina čitanja i crtanja na zaslonu nije usklađena s našom brzinom pisanja u sami *framebuffer*. Može se dogoditi da se igra brže crta nego što mi pišemo. Ono što se tada dogodi je to da ćemo početi pisati neku sliku u *framebuffer*, npr. neki automobil. U isto vrijeme ažuriramo igru, računamo raznorazne vrijednosti te crtamo sve to na zaslonu. Dok dođemo do, na primjer, guma od automobila, slika se već nastavila crtati bez da smo uspjeli napisati u *frambuffer* što dalje želimo. Zato ćemo dobiti sliku automobila koji nema gume. Računalo zapravo nastavi

čitati *framebuffer*, ali kako nismo stigli u njega sve napisati, tamo nisu vrijednosti koje želimo. Tada se događa *tearing* ili kidanje zaslona.

Taj se problem riješi postojanjem dvaju *framebuffera*. Iz jednoga će računalo čitati što će crtati na zaslon, a u drugi ćemo zapisivati što će biti na sljedećoj sličici. Tako zapravo odvojimo crtanje i pisanje u *framebufferu* na trenutni i sljedeći *buffer*. U pozadini pripremamo ono što će se crtati na idućoj sličici i zatim na kraju ciklusa predamo računalu nove podatke. On ih crta, a mi opet u pozadini pripremamo sljedeću sličicu. Informacije se čitaju samo iz trenutnog, a pišu samo u sljedeći *buffer*. Kada obavimo te operacije, izmijenimo ta dva *buffera*.

Ovo je jedna od centralnih ideja u modernim igrama. Starije konzole nisu nužno to radile, nego su usklađivale crtanje s brzinom osvježavanja zaslona. To nije bilo jednostavno, ali je bilo potrebno na hardveru koji je imao ograničene mogućnosti. U svijetu izvan igara ovaj obrazac može biti koristan ako se u nekom programu podatcima pristupa u isto vrijeme kada se u njih i piše. Ono čega moramo biti svjesni je da samo mijenjanje *buffera* može nekada biti vremenski zahtjevno. Nadalje, moramo biti svjesni da sada imamo dva *buffera* i da je to opterećenje memorije. Ako koristimo neke uređaje koji su ograničeni memorijom, vjerojatno bi neka druga opcija bila bolja.

Primjer

Kao primjer konstruirat ćemo jedan jednostavan oblik grafičkoga sustava. On će nam omogućavati pisanje po *framebufferu*. Ovdje zapravo implementiramo ono što već postoji na nižim razinama grafičkoga sustava, ali ovaj će nam primjer pomoći da razumijemo o čemu se zapravo radi. Prvo ide samo *framebuffer*:

```
class Framebuffer
{
  public:
    Framebuffer() { clear(); }

    // 0 - bijeli pikseli, 1 - crni pikseli.
    void clear()
    {
      for (int i = 0; i < width * height; ++i)
      {
        pixels[i] = 0;
      }
    }

    void draw(int x, int y)
```

```
    {
      pixels[(width * y) + x] = 1;
    }

    const int* getPixels()
    {
      return pixels;
    }

  private:
    const int width = 200;
    const int height = 300;
    int pixels[width * height];
};
```

Napravili smo funkcije kojima crtamo po *bufferu* i resetiramo sve na bijelu boju. Također možemo pristupiti cijelom nizu boja preko funkcije `getPixels()` koju će zvati *video driver* tako da bi čitao *buffer* i crtao na zaslon.

Sada ćemo dodati klasu preko koje ćemo crtati u *buffer*.

```
class Scene
{
  public:
    void draw()
    {
      buffer.clear();

      buffer.draw(1, 1);
      buffer.draw(2, 1);
      buffer.draw(3, 1);
      buffer.draw(4, 1);
      buffer.draw(1, 2);
      buffer.draw(4, 2);
      buffer.draw(1, 3);
      buffer.draw(2, 3);
      buffer.draw(1, 4);
      buffer.draw(2, 4);
      buffer.draw(3, 4);
      buffer.draw(4, 4);
    }
    Framebuffer& getBuffer() { return buffer; }
```

```
  private:
    Framebuffer buffer;
};
```

Funkcija `draw()` crta kvadrat veličine 4 x 4 na zaslonu.

Pri svakom pozivu funkcija `draw()` briše sve na zaslonu i crta kvadrat. Točnije, ona isprazni *buffer* i nakon toga piše u njega. Tu imamo i funkciju `getBuffer()` preko koje *video driver* može pristupiti *bufferu*.

Na prvi pogled ne čini se kako bi ovako mogao nastati neki problem, ali do njega će doći kada su naša scena i *video driver* neusklađeni. On može u bilo kojem trenutku zatražiti piksele iz *buffera* i crtati ih po zaslonu, pa možemo dobiti ovakvu situaciju:

```
buffer.clear();

buffer.draw(1, 1);
buffer.draw(2, 1);
buffer.draw(3, 1);
buffer.draw(4, 1);
buffer.draw(1, 2);
buffer.draw(4, 2);
// Ovdje video driver čita piksele.
buffer.draw(1, 3);
buffer.draw(2, 3);
buffer.draw(1, 4);
buffer.draw(2, 4);
buffer.draw(3, 4);
buffer.draw(4, 4);
```

U tom će se slučaju na zaslon u toj sličici iscrtati samo gornji dio kvadrata. U narednim crtanjima može prekinuti pisanje u bilo kojem trenutku, pa dolazi do treperenja na zaslonu. Ovo se događa zato što čitamo iz istog *buffera* u koji i pišemo. Ovaj problem možemo riješiti uvođenjem drugog *buffera*:

```
class Scene
{
  public:
    Scene()
    : current(&buffers[0]),
    next(&buffers[1])
    {}

    void draw()
```

```
    {
      buffer.clear();

      next.draw(1, 1);
      next.draw(2, 1);
      next.draw(3, 1);
      next.draw(4, 1);
      next.draw(1, 2);
      next.draw(4, 2);
      next.draw(1, 3);
      next.draw(2, 3);
      next.draw(1, 4);
      next.draw(2, 4);
      next.draw(3, 4);
      next.draw(4, 4);
    }
    Framebuffer& getBuffer() { return *current; }
  private:
    void swap()
    {
    // Zamjena pokazivača.
      Framebuffer* temp = current;
      current = next;
      next = temp;
    }
    Framebuffer buffers[2];
    Framebuffer* current;
    Framebuffer* next;
};
```

Sada imamo dva *buffera*. Ono što smo ovdje postigli je to da se nikada neće čitati iz *buffera* u koji se piše i obrnuto. Uvijek pišemo samo u `next` i čitamo samo iz `current`. Nakon što završimo crtanje iduće scene u `next`, napravimo zamjenu *buffera* pozivom funkcije `swap()`. Sada vidimo da *video driver* može pozvati `getBuffer()` u bilo kojem trenutku i problem neće nastati. Na zaslonu će se nacrtati potpuna slika bez ikakva treperenja.

Zaključak

Double buffer nam omogućava da prikazujemo naše igre fluidno i u cijelosti. Pod cijenu memorije i brzine dobivamo igru koju je lakše crtati te ne trebamo paziti kakav je zaslon

na koji se iscrtava igra. Možemo manje razmišljati o hardveru, a više se fokusirati na svoj kod.

3.3 *State*

Motivacija

Ovaj oblikovni obrazac koristimo kako bismo predstavljali stanja u kojima može biti naša igra ili objekti u našoj igri. Igra može imati više različitih situacija u kojima se drukčije ponaša. Na primjer, najčešće će postojati glavni izbornik iz kojega možemo ili ugasiti igru ili prijeći u stanje igranja. Ono što korisniku prikazujemo možemo pamtiti jednom varijablom, ali to stvara probleme kada želimo proširiti kod tako da dodamo u izborniku opciju kojom korisnik može mijenjati postavke ili pogledati upute kako igrati igru. Dodana su dva nova stanja koja opet pratimo tom jednom varijablom. U tom slučaju može doći do toga da je glavni dio koda jedna velika `switch` naredba koja ima puno slučajeva. Odmah je očito zašto je to loše i nepoželjno. Uvijek želimo da je naš kod što pregledniji i pogodniji za razvijanje, zato je bolje odvajanje u zasebne klase.

Klase koje kreiramo bit će razna stanja i kod treba imati mogućnost njihova mijenjanja. Objekt će izgledati kao da je mijenjao klasu, ali zapravo je samo došlo do promjene njegova stanja. To će nam smanjiti velike dijelove koda u kojima provjeravamo puno uvjeta prije neke radnje odvajanjem tih radnji u zasebne klase.

Primjer

Kao primjer uzmimo igru u kojoj je glavni lik čovjek koji se može transformirati u lava čiji će napad biti jači. Prvi dio pokazat će zašto koristimo ovaj obrazac. Zamislimo klasu `Player` koja predstavlja naš glavni lik koji ima mogućnost transformiranja. Jedan osnovan način rješavanja ovog problema bilo bi korištenje enumeracije:

```
enum class States
{
  Human,
  Lion
}
```

Klasa `Player` u tom bi slučaju sadržavala varijablu koja bi označavala trenutno stanje. Sada treba čitati unos korisnika koji može napasti čudovište pritiskom na tipku *space*:

```
void Player::processInput()
{
  sf::Event event;
```

```
    while (mWindow.pollEvent(event))
    {
      switch (event.type)
      {
      case sf::Event::KeyPressed:
        if (event.key.code == sf::Keyboard::Space)
        {
          if(state == States:Human)
          {
            // 30 oznacava koliko je snazan napad
            attack(30);
          } else if( state == States:Lion )
          {
            attack(80);
          }
        }
      break;
        case sf::Event::Closed:
        mWindow.close();
      break;
      }
    }

}
```

Već je ovdje očito da se stvari u kodu lako zakompliciraju kada obrađujemo stanja na ovakav način. Još bi veći problem bio kada bismo htjeli da se naš lik može transformirati u vuka ili medvjeda. Dodatne komplikacije nastaju ako želimo da to mijenja i brzinu. To može dovesti do koda koji je prekompliciran i neodrživ. Zato želimo ove dijelove prebaciti u zasebne klase koje će sve naslijediti `State` klasu koja će imati čisto virtualnu `processInput()` člansku funkciju. Tako možemo iskoristiti polimorfizam za poboljšanje kvalitete koda.

Uz već spomenutu klasu imat ćemo i `HumanState` i `LionState` koje će implementirati `processInput()` funkciju. `Player` klasa sadržavat će pokazivač koji pokazuje na trenutno stanje. Kod sada izgleda ovako:

```
Class State
{
  virtual void processInput(sf::Event event) = 0;
}
// ...
```

```
Class HumanState : public State
{
  void processInput(sf::Event event)
  {
    if ( event.type == sf::Event::KeyPressed
      && event.key.code == sf::Keyboard::Space)
    {
      attack(30); // 30 oznacava koliko je snazan napad
    }
  }
}
// ...
Class LionState : public State
{
  void processInput(sf::Event event)
  {
    if ( event.type == sf::Event::KeyPressed
      && event.key.code == sf::Keyboard::Space)
    {
      attack(80);
    }

  }
}
// ...
void Player::processInput()
{
  // ...
  while (mWindow.pollEvent(event))
  {
    currentState->processInput(event);
    // ...
  }

}
```

Očito je ovaj kod puno pregledniji i lakši za održavanje. Omogućava nam jednostavno dodavanje novih stanja.

U zadnjem se primjeru samo čitaju unosi korisnika, ali je dobra ideja ovako pozivati još i `update()` i `draw()` funkcije jer ćemo ovisno o trenutnom stanju pomicati i crtati glavni

lik.

Zaključak

Glavna primjena ovoga obrasca bolja je organizacija koda. Bez njega neke funkcije mogu biti ogromne i neodržive. Ako je funkcija dovoljno velika i ima mnogo `if`/`else` naredbi ili prevelik `switch`, treba nam `State` oblikovni obrazac. Točnije, potreban je ako neki objekt mijenja ponašanje ovisno o stanju. Njime jednostavno mijenjamo stanje glavnoga lika i dodajemo nove dijelove po potrebi.

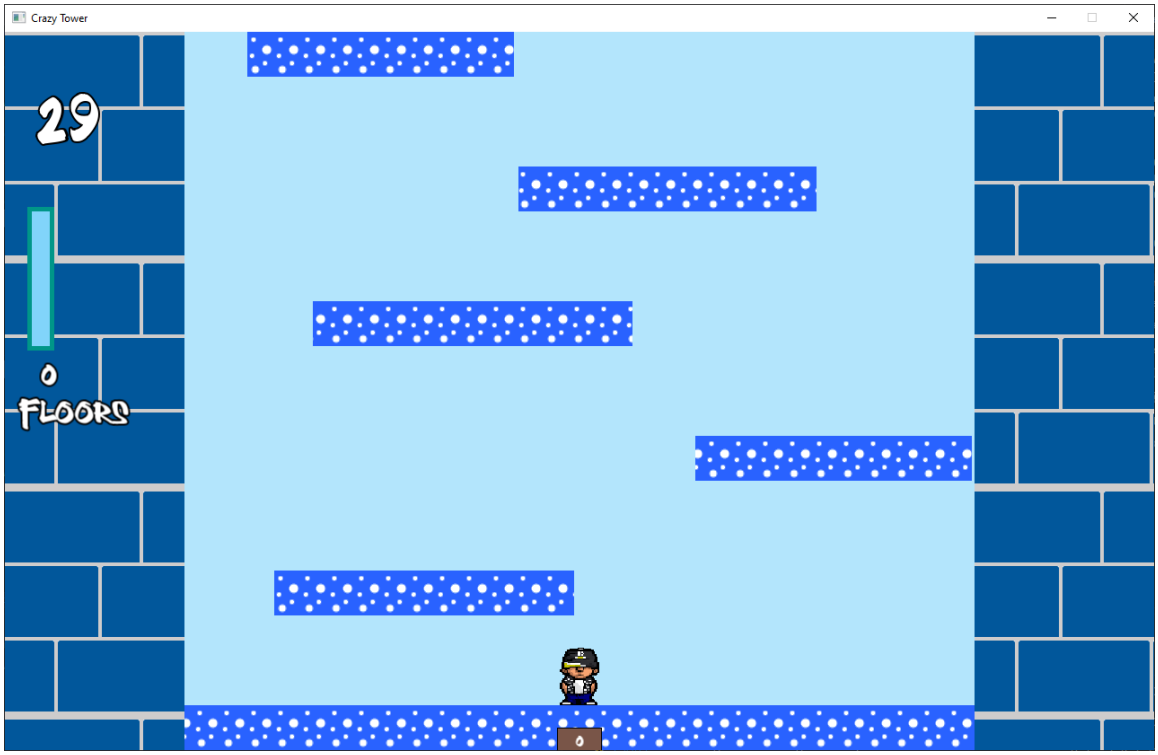
Poglavlje 4

Implementacija igre u SFML-u

Ovo poglavlje demonstrirat će jednu implemetaciju igre u SFML-u. Bit će prikazana struktura implementacije i neki dobri običaji pri programiranju igara. Igra u pitanju zove se *Crazy Tower*. To je 2D platformer u kojem glavni lik skače po tornju. Cilj je igre doseći što viši kat i postići što veći *combo*. U početku igrač stoji na početnoj platformi i pomicanjem pritiskom na tipke desno i lijevo i skakanjem na tipku *space* pokrećemo naš lik. Zaslon izgleda kao na slici 4.1.

4.1 Crazy Tower – pravila

Kada se lik počne kretati, i sama se igra nakon par katova počinje pokretati. Zaslon se kreće prema gore, a igrač mora ostati iznad dna zaslona. Ako igrač padne ispod dna, trenutna je igra gotova. Tada se naš najveći *combo* i najviši kat uspoređuju s rezultatima drugih igrača i postavljaju na ljestvicu. Najviši kat postižemo tako da likom skačemo na sve više katove. Taj rekord nije toliko teško postići koliko je teško ostvariti velik *combo*. On se postiže tako što igrač preskoči više od jednog kata u jednom skoku. Tada igrač uđe u *combo* stanje u kojem se svaki put kada preskoči više od jednoga kata, brojka preskočenih katova pribraja samom *combo* broju. Postoji i vremenski brojač od tri sekunde koji se ponovno postavlja svaki put kada povećamo *combo*. Ako to vrijeme istekne ili ako igrač ne preskoči više od jednoga kata, *combo* se postavlja na nulu. Slika 4.2 prikazuje gdje se nalazi trenutni *combo*. Odmah iznad njega vremenski je brojač za *combo*, a povrh njega je sat za povećanje brzine. On povećava brzinu kretanja zaslona svakih trideset sekundi i tako čini igru težom nakon određena vremena.



Slika 4.1: Crazy Tower

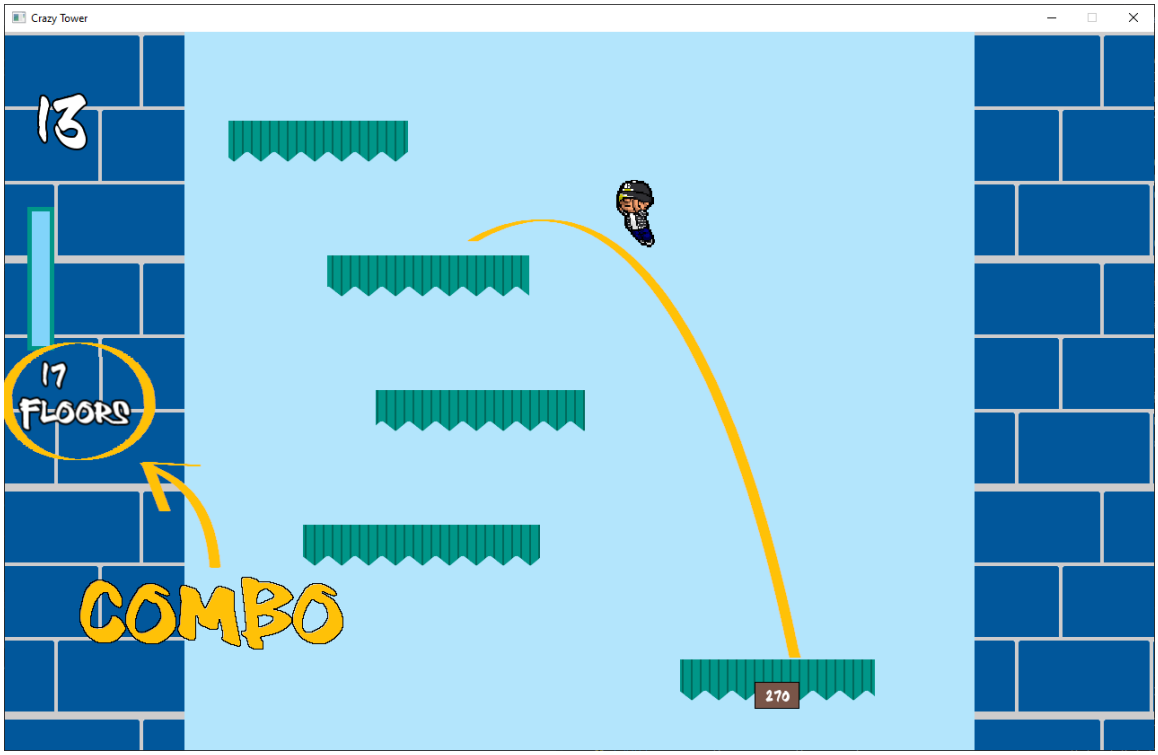
4.2 Implementacija

Igra je implementirana korištenjem više standardnih metoda i oblikovnih obrazaca za igre. Oni služe tome da igra ima kod koji je organiziran, čitljiv i lako se može unaprijediti. Korišteni su obrasci poput *State* i *Game Loop* te neke standardne metode poput klasa `Game` i `Player`.

Game

Aplikacije pisane u jeziku C++ počinju svoje izvršavanje iz `main()` funkcije. U igrama se teži postići što manje logike i programa u glavnoj funkciji. Zbog toga se često uvede nova klasa `Game` koja obavlja svu glavnu logiku igre. Ona je centralna klasa u kojoj cijela igra počinje svoje izvršavanje. `Game` ima sljedeću strukturu:

```
class     Game
{
public:
```



Slika 4.2: Crazy Tower *combo*

```
                Game();
  void          run();

private:
  void          processInput();
  void          update(sf::Time dt);
  void          render();

};
```

Ovo je standardna struktura glavne igrine klase. Igra počinje pozivom `run()` u kojoj se pozivaju ostale funkcije igre. Ovdje se koristi prvi obrazac u igri: *Game loop*. Iz glavne petlje počinje sva centralna logika igre i obrazac koristimo kao što je već opisano u jednom od prethodnih poglavlja. Naravno, svu logiku igre nećemo prepustiti ni samoj klasi `Game`, nego ćemo je rasporediti u više klasa koje će imati specifične funkcije. U ovom slučaju u njoj glavna petlja ograničava igru na 60 sličica po sekundi.

Stanja

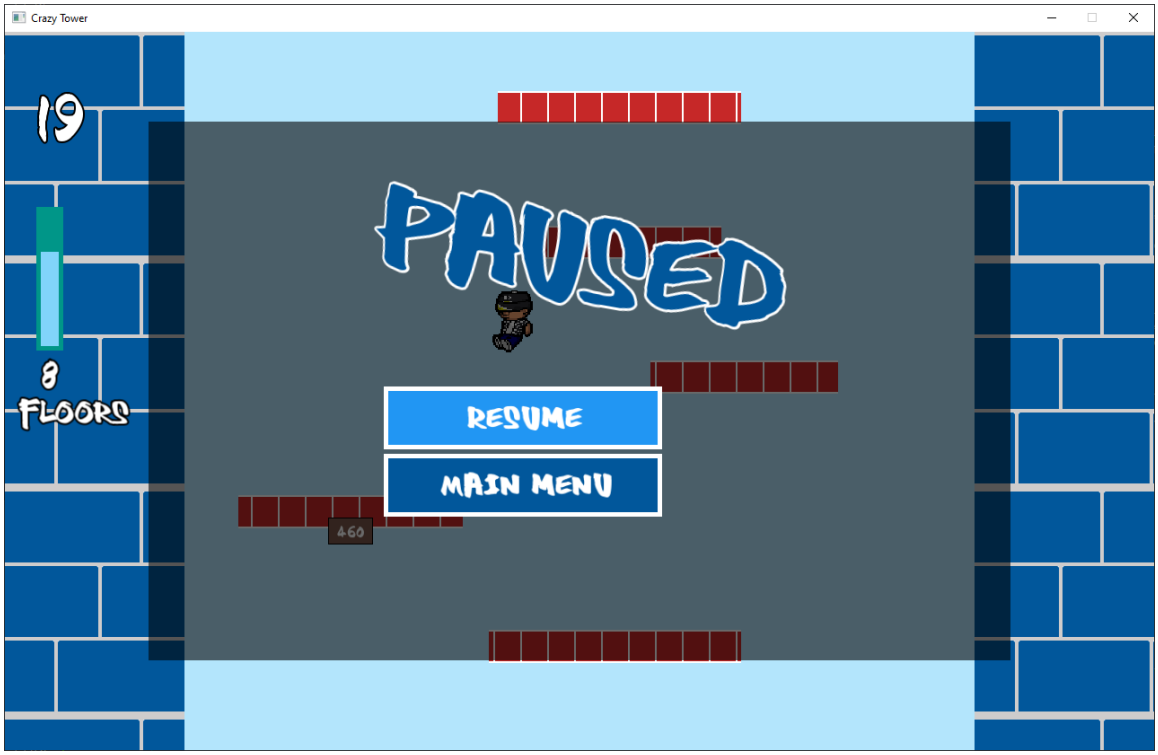
Kao i većina igara ova neće početi na način da igrača odmah ubaci u igru. Prvo se pojavi glavni zaslon s određenim opcijama iz kojega možemo ući u igru. Za vrijeme igre želimo da postoji mogućnost pauziranja igre kako bi igrač mogao zaustaviti pa nastaviti igru (ili se vratiti na početni zaslon). Sada je očito da su sve ovo određena stanja u kojima se igra može nalaziti, pa ćemo zbog toga iskoristiti *State* oblikovni obrazac. Imat ćemo više različitih stanja u kojima se igra može nalaziti te ćemo ih stavljati na stog svih stanja po potrebi. Svako stanje nasljeđuje klasu `State` koja ima sljedeću glavnu strukturu:

```
class State
{
public:
  typedef std::unique_ptr<State> Ptr;

  struct Context
  {
    Context( ... );
    // Ovdje idu varijable koje ce biti dio konteksta
  };

public:
            State(StateStack& stack, Context context);
  virtual void    draw() = 0;
  virtual bool    update(sf::Time dt) = 0;
  virtual bool    handleEvent(const sf::Event& event) = 0;

protected:
  void        stackPush(GameStates::ID stateID);
  void        stackPop();
  void        stackClear();

  Context       context;

private:
  StateStack*     stack;

};
```

Sva stanja imat će funkcije bitne za glavne radnje u igri (crtanje, ažuriranje i događaje). Bitno je primijetiti da ažuriranje i obrada događaja imaju povratni tip `bool`. Ovisno o

tome što te funkcije vraćaju, bit će određeno hoće li se stanja na stogu ispod njih ažurirati ili ne, kao i hoće li obrađivati događaje ili ne.

Dakle, ideja je ta da stanja stavljamo na stog i onda se ona ažuriraju ovisno o stanjima iznad. Korist ovoga pristupa očita je na primjeru pauziranoga stanja. U toj situaciji ne želimo da se igra resetira, nego da je u istome stanju koje je bilo prije pauziranja. To znači da želimo da se igra iscrtava, ali ne i ažurira. To ćemo postići tako da funkcije `update` i `handleEvent` vrate vrijednost `false` i time naznače stogu da se stanja ispod trenutnog ne ažuriraju. Kada se igra pauzira, ona zapravo gura pauzirano stanje na stog i dolazi do toga da je stanje igre ispod toga pauziranog stanja. Kada smo gotovi s pauzom, samo mičemo element s vrha stoga i time se nastavlja trenutna igra. Ako se stog isprazni, znamo da je igra gotova i zatvaramo prozor. Pauzirani zaslon izgleda kao na slici 4.3.



Slika 4.3: Pauziran zaslon

Sva će stanja koja nude neke mogućnosti odabira sadržavati u sebi jedan izbornik. On će biti implementiran preko dvije klase, a to su `Menu`, koja predstavlja jedan izbornik, i `MenuOption`, koja predstavlja jedan izbor. Ona simulira gumb preko klase `sf::RectangleShape`. Ovim je pristupom lakše ubaciti izbornike u više različitih stanja.

Kako bi se razna stanja mogla prikazivati u istome prozoru i kako bi mogla međusobno

komunicirati, potrebna nam je jedna klasa koja bi sadržavala sve bitne informacije. To nam nudi `State::Context` koji će u sebi sadržavati sve objekte koje razna stanja trebaju i dijele. On nam je potreban da neke podatke ne učitavamo više puta, čime štedimo i procesorsko vrijeme i memoriju. To nam je posebno poželjno kada su u pitanju slike i zvukovi. Njih ćemo također spremati na poseban način.

Resursi

Resurse nije praktično i pregledno držati u zasebnim varijablama. Rastom njihovoga broja (što se često dogodi u igrama) možemo imati velik broj varijabli, a to lako dovodi do nekih *bugova*. Ono što želimo, neki je organizator resursa preko kojeg možemo učitati resurse i kasnije im jednostavno pristupiti. Njega bismo onda slali stanjima preko konteksta i time osigurali da dodavanje resursa ne zahtijeva velike promjene u kodu. Nakon što učitamo neki resurs, pridružit ćemo mu neku identifikaciju (ID) u obliku enumeracije. Primjer takve klase izgledao bi ovako:

```
template <typename Resource, typename Identifier>
class ResourceHolder
{
public:
  void load(Identifier id, const std::string& filename);
  Resource& get(Identifier id);
  const Resource& get(Identifier id) const;

private:
  std::map<Identifier, std::unique_ptr<Resource>>
        mResourceMap;
};
```

Koristimo *template* kako bi ova klasa bila što općenitija. Želimo preko nje mogućnost pristupa i teksturama, i zvukovima, i svim ostalim mogućim resursima. Tako imamo samo jednu klasu za sve resurse.

Uz ovo moramo imati neku identifikaciju koju ćemo predstaviti enumeracijama:

```
namespace Textures
{
  enum class ID
  {
    Background,
    Character,
    Platform,
```

```
    };
}

namespace SoundEffect
{
    enum class ID
    {
        Jump,
        Fall,
        MenuSelect
    };
}
```

Svakoj identifikaciji pripada točno jedan resurs. Tako je lakše pristupiti potrebnom resursu i kod je manje sklon greškama. U početku izvođenja aplikacije unutar klase `Game` učitaju se svi resursi i onda njima pristupamo preko njihovoga ID-a. Kako njih šaljemo unutar konteksta, svako će stanje imati pristup.

Sve ove centralne klase omogućavaju nam dobro organiziran kod koji nije međusobno ovisan, ali koji održava dobru komunikaciju. Ovakav kostur postavlja dobre temelje za preglednu i lako održivu aplikaciju.

Graf igre

Najvažniji dio svake igre onaj je u kojem je igramo, ostalo su sve dijelovi koji služe kako bi on bio bolji. Taj je dio dosta kompliciran za crtanje, pa se koristi neki malo poželjniji pristup. Kada bi sve crtanje obavljali u jednoj klasi, to bi zahtijevalo velik broj varijabli koje moramo sve pripremiti i održavati na jednom mjestu. Pri svakom dodavanju novog dijela ili mijenjanju starog naš bi kod bio otvoren greškama i sve manje pregledan. Zbog toga ćemo crtanje i ažuriranje prebaciti u čvorove grafa naše aplikacije. Taj će graf biti podijeljen u slojeve i svaki sloj igre imat će svoj korijen i djecu koja će predstavljati dijelove toga sloja.

Igra je prvo podijeljena u slojeve: pozadinu, platforme, igrača i zidove tornja. Za svaki sloj grafa njegov korijen predstavljen je čvorom i označen imenom toga sloja. Svaki od njih imat će barem jedno dijete i nekada će ta djeca imati još djece. U slučaju sloja s platformama prvo dijete bit će klasa `Platforms` koja će pokazivati na više djece tipa `Platform`, a svaki će taj čvor predstavljati jednu platformu. Svaka od njih crtat će se u odnosu na poziciju roditelja, tj. objekta tipa `Platforms`. To znači da ako postavimo sve platforme na poziciju (50, 60) i onda stavimo jednu platformu na poziciju (3, 6), njena će pozicija u prozoru biti (53, 66). Dakle, dijete radi transformacije relativno o roditelju.

Ovakav pristup omogućava nam dobru organizaciju koda što se tiče ažuriranja i crtanja te nam omogućava pozicioniranje elemenata igre u odnosu na druge. To nam je posebno poželjno kad, na primjer, imamo lik koji se može klonirati i njegov ga klon prati u svakom koraku. Ovakvim je pristupom to jednostavno napraviti, samo u grafu klona postavimo kao dijete lika. Tako će on lako pratiti poziciju originala.

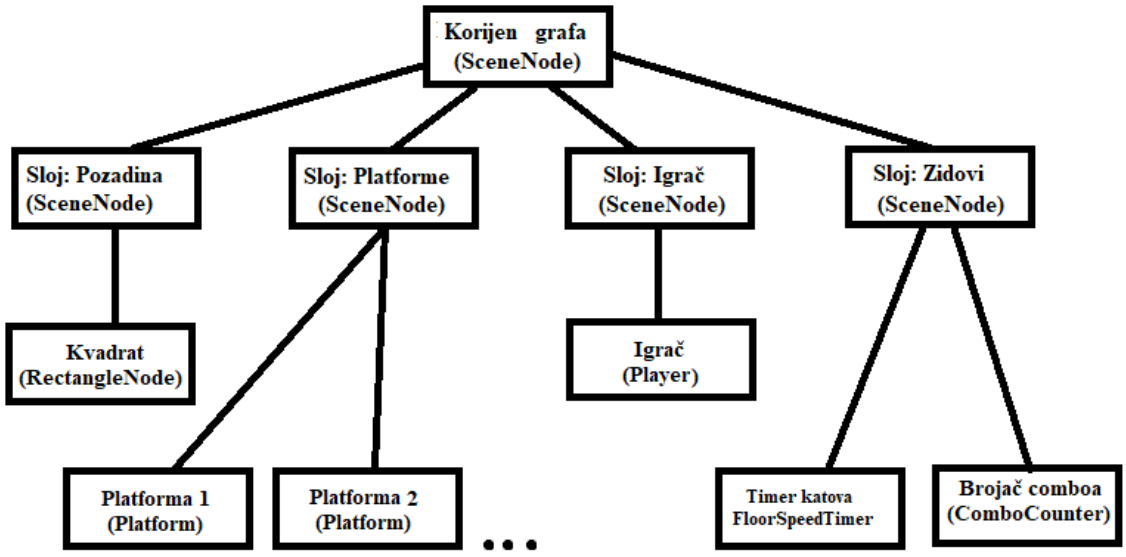
Sve počinje od klase `SceneNode` koju će svaki od čvorova naslijediti.

```
class SceneNode :
  public sf::Transformable,
  public sf::Drawable
{
public:
                SceneNode();
  void          attachChild(std::unique_ptr<SceneNode> child);
  Ptr           detachChild(const SceneNode& node);

  void          update(sf::Time dt);

  sf::Vector2f      getWorldPosition() const;
  sf::Transform     getWorldTransform() const;

private:
  virtual void      updateCurrent(sf::Time dt);
  void          updateChildren(sf::Time dt);

  virtual void      draw(sf::RenderTarget& target,
                sf::RenderStates states) const final;
  virtual void      drawCurrent(sf::RenderTarget& target,
                sf::RenderStates states) const;
  void          drawChildren(sf::RenderTarget& target,
                sf::RenderStates states) const;

  SceneNode* parent;

protected:
  std::vector<std::unique_ptr<SceneNode>>   children;

};
```

Nasljeđivanje `Transformable` omogućava da nad čvorovima radimo transformacije, a `Drawable` omogućava da zovemo `window.draw(node)` i tako crtamo čvorove.

U njoj možemo dodavati i izbacivati djecu čvorova. Budući da se čvorovi transformiraju u odnosu na roditelje, nude se funkcije `getWorldPosition()` i `getWorldTransform()` koje nam pružaju globalne pozicije i transformacije. One služe kada trebamo provjeriti imaju li dva elementa u igri dodira (stoji li igrač na platformi i sl.). Funkcije `update()` i `draw()` zapravo pozivaju crtanje i ažuriranje prvo na trenutnom čvoru, a zatim na njegovoj djeci, uz napomenu da `draw()` prvo primijeni sve transformacije roditelja pa onda crta trenutni čvor.

Prije svega ovoga postoji klasa `Tower` koja prikazuje svijet koristeći mogućnosti klase `sf::View`. Ona će stvoriti korijen grafa koji će za djecu imati čvorove koji pak predstavljaju slojeve. Zbog toga će graf igre izgledati kao na dijagramu 4.4.



Slika 4.4: Čvorovi grafa

U toj glavnoj klasi svi se početni čvorovi pozicioniraju po zaslonu i omogućava im se međusobna komunikacija.

Klasa `Platforms` služi za inicijaliziranje i ažuriranje svih platformi. Ona organizira platforme i služi kao most za komuniciranje među platformama i drugim klasama. `Platform` predstavlja jednu platformu. `RectangleNode` je pozadina igre. `TowerWalls` su zidovi desno i lijevo od glavnoga dijela igre. Oni na sebi imaju sat koji mjeri kada treba ubrzati kretanje igre, sat koji mjeri trajanje *comboa* i brojač za *combo*.

Na kraju, `Player` klasa predstavlja igrača. U njoj se analiziraju unosi korisnika koji ga pokreću. Sadržava svu logiku vezanu uz pokrete i ažuriranje pozicije. Ona je centralna

klasa u ovom grafu s kojom svaka druga na neki način komunicira. Preko nje upravljamo našim igračem sve dok ne padne ispod zaslona i ne pojavi se *Game over* zaslon. Na njemu pak upisujemo naš rezultat ako smo srušili neke rekorde. Cilj je, kao što smo već rekli, postići što bolji rekord.

Bibliografija

[1] *Wikipedia - The Free Encyclopedia*, https://en.wikipedia.org/.

[2] M. Barbier, *SFML Blueprints*, Pact Publishing, 2015, https://www.packtpub.com/game-development/sfml-blueprints.

[3] Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides, *Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994, https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612.

[4] Laurent Gomila, *Web stranica SFML-a*, https://www.sfml-dev.org/.

[5] J. Gregory, *Game Engine Architecture*, Wellesley, Massachusetts, 2009, https://www.amazon.com/Engine-Architecture-Third-Jason-Gregory/dp/1138035459.

[6] J. Horton, *Beginning C++ Game Programming*, Pact Publishing, 2016, https://www.packtpub.com/game-development/beginning-c-game-programming.

[7] M. G. Milchev, *SFML Essentials*, Pact Publishing, 2015, https://www.packtpub.com/game-development/sfml-essentials.

[8] A. Moreira, H. V. Hansson i J. Haller, *SFML Game Developement*, Pact Publishing, 2013, https://www.packtpub.com/game-development/sfml-game-development.

[9] R. Nystrom, *Game Programming Patterns*, Genever Benning, 2014, https://gameprogrammingpatterns.com/.

[10] R. Pupius, *SFML Game Developement By Example*, Pact Publishing, 2015, https://www.packtpub.com/game-development/sfml-game-development-example.

Sažetak

U ovom radu vidjeli smo kako funkcionira SFML. To je C++ biblioteka koja omogućava programiranje dvodimenzionalnih računalnih igara. SFML nudi razne mogućnosti u razvoju igara koje su ponuđene preko 5 glavnih modula, a to su sustav, prozor, grafika, zvuk i mreža. S njima možemo crtati po zaslonu, upravljati likom i ostalim dijelovima programa, puštati zvukove i igrati preko mreže. Sve te mogućnosti dovode do toga da možemo programirati kvalitetne i komplicirane igre. Kako bi taj kod bio pregledan i održiv, koristimo oblikovne obrasce. Oni su isprobane metode organiziranja koda koje dovode do toga da je kvalitetnije napisan. Bez njih bi kodove s velikim brojem linija bilo teško čitati i održavati. Oni daju višu razinu apstrakcije kodu. Na kraju smo vidjeli kako implementirati jednu igru u SFML-u. Pri tome smo koristili njegove module i oblikovne obrasce. Na tome primjeru vide se mogućnosti SFML-a i zašto su korisni obrasci. Također su korištene neke standardne prakse pri programiranju igara koje olakšavaju organizaciju i dodavanje novih dijelova igre.

Summary

In this paper, we have seen the capabilities of SFML. It is a C++ library that allows us to program two-dimensional video games. SFML offers different possibilities when developing games and they are offered through 5 main modules, these are system, window, graphics, sound and network. We can use them to draw on the screen, manipulate the character and other parts of the game, play sounds and play over a network. All those possibilities give us the option to develop high quality complex games. To make that code readable and sustainable, we use design patterns. They are tried and tested methods of organizing code that leads to it being written better. Without them, codes that have many lines would be difficult to read and maintain. They provide a higher level of abstraction to the code. Finally, we saw how to implement a game in SFML. While doing so, we used its modules and design patterns. That example showed the capabilities of SFML and usefulness of design patterns. Some standard game programming practices have also been used which facilitate the organization and addition of new parts to the game.

Životopis

Zvonimir Šimunović rođen je 27. listopada 1993. u Splitu, a odrastao je u gradu Imotskom. Tu završava Osnovnu školu "Stjepan Radić" i prirodoslovno-matematičku gimnaziju u Gimnaziji dr. Mate Ujevića.

Nakon toga u Zagrebu upisuje preddiplomski studijski program Matematika na Prirodoslovno-matematičkom fakultetu. Nakon završavanja preddiplomskog upisuje dimplomski Računarstvo i matematika.