SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Zvonimir Šimunović

PROGRAMIRANJE VIDEO IGARA U BIBLIOTECI SFML

Diplomski rad

Voditelj rada:
Prof. dr. sc. Mladen Jurak

Zagreb, Rujan, 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 omogućuju pisanje koda za igre. Ti jezici i biblioteke omogućuju brzo, efikasno i pregledno pisanje koda za igre.

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 omogućuju jednostavno i brzo programiranje.

Sama biblioteka nije dovoljna za pisanje kvalitetnog koda za igre, već je bitna i organizacija koda. Objektno orijentirani jezici omogućuju 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 diplomski 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. Te obrasce gledat ćemo kako nam omogućuju 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 omogućuje 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 Unreal 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 [3] 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 za jezik C++ koji sam po sebi omogućuje preciznije upravljanje objektima i memorijom koju oni zauzimaju.

Kako bismo koristili SFML, potrebno je skinuti biblioteku sa službene stranice [2]. 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 open-source, 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 osnovni primjer. Objašnjeni su i neki koncepti poput glavne petlje igre (game loop), renderiranja 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 omogućuje 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 renderanje. 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 to lokalne ili preko interneta.

Ovdje neće biti detaljan prikaz svih modula, već ćemo ih samo prikazati. 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>
```

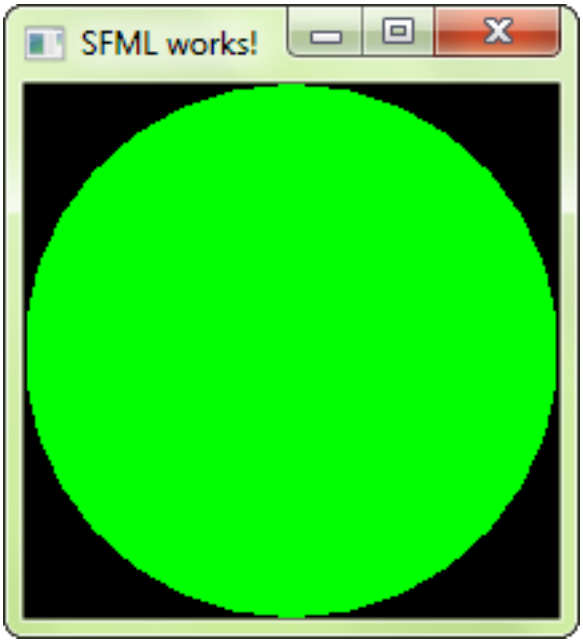
2.2 Osnovni primjer

Pogledat ćemo jedan osnovan 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 pokrene događaj ili event zatvaranja prozora.

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 3 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 omogućuje razne mogućnosti crtanja po zaslonu, ali ono što mi želimo je da tim likovima možemo na neki način upravljati. Tu u priču ulaze **događaji**. Jedan od takvih 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 je glavna 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. Crtanje 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 ekranu, 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.

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 Window::pollEvent(sf::Event\& e`

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 omogućuje 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 omogućuje 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 omogućuje nam da smanjimo međusobnu zavisnost 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 Crtanje

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` klase, 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 igrine petlje pisan je u "Double Buffer" oblikovnom obrascu. Jednu sliku prikazujemo dok drugu pripremamo za prikazivanje. To nam omogućuje da se igra bez problema iscrtava na zaslonu, ali o tome više u drugom poglavlju.

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 omogućuje 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 `Shape::setFillColor()`, `Shape::setOutlineColor()` i `Shape::setOutlineThickness()`. Ove nam funkcije omogućavaju da postavljamo izgled oblika onako kako nam odgovara. Pokazat ćemo njihovu upotrebu u sljedećem primjeru:

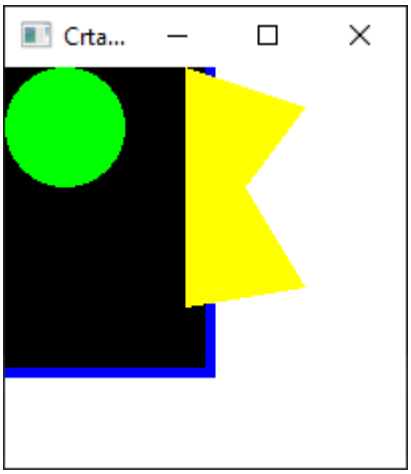
```
sf::CircleShape circleShape(30);
circleShape.setFillColor(sf::Color::Green);

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 omogućuje da pohranjujemo tipove koje želimo, `Vector2\textless class\textgreater` i `Vector3\textless class\textgreater`. Vektore ćemo često koristiti u crtanju za označavanje dimenzija, smjerova, pozicija i slično.

Sada želimo na zaslonu prikazati naše oblike, pa ćemo koristiti varijablu 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. U nastavku je prikazan rezultat pokretanja koda:

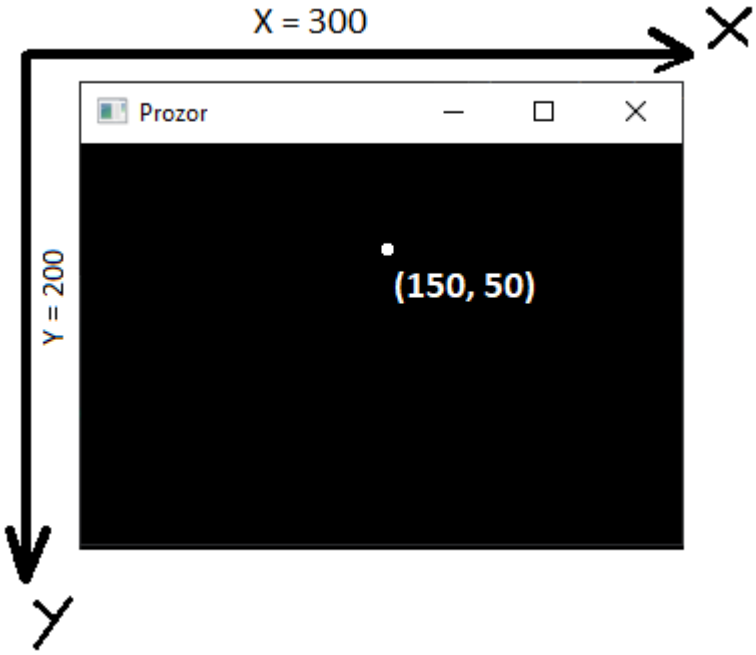


Slika 2.2: Shape primjer 1

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 razumiti kako funkcionira koordinatni sustav u igrama. Njegova orjentacija je drugačija od standadrdne na koju smo navikli. Gornji lijevi kut prozora predstavlja točku (0, 0). Os y raste prema dolje dok os x raste prema desno. Kako to izgleda možemo vidjeti u priloženom crtežu:



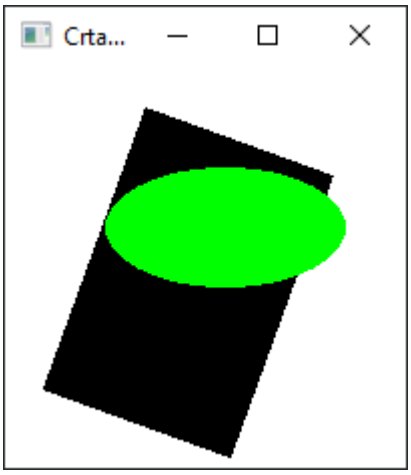
Slika 2.3: Koordinatni sustav

Sada znamo kako će se ponašati naše transformacije pa ih možemo i početi koristiti. Do sada smo koristili klasu `Shape` za sve naše oblike i da bi ih crtali, ali ona nudi i još više mogućnosti. Ona nasljeđuje `sf::Transformable` koja nam nudi neke funkcije za transformiranje. `Transformable::setPosition()` postavlja oblik na željenu poziciju, `Transformable::setRotation()` koristimo za rotaciju oblika, a `Transformable::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 sljedeći rezultat: Vidimo da smo krug i pravokutnik po-



Slika 2.4: Shape primjer transform

maknuli. 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 svog centra.

Teksture i spriteovi

Za učitavanje i prikazivanje slika u SFML-u primarno su zadužene 3 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 jednostavnosti spriteovi se češće koriste.

Učitavanje slika

Za učitavanje slika u SFML-u koristimo klase `sf::Image` i `sf::Texture`. Razlika između ovih dviju klasa je u mogućnostima manipuliranja i prikazivanja. `Image` klasu koristimo za učitavanje i spremanje slika, te manipulaciju piksela dok `Texture` koristimo za prikazivanje slika (render). One obe u srži sadržavaju u sebi niz piksela ali se razlikuju po nekim mogućnostima koje svaka klasa nudi. Zbog toga SFML nam nudi lako prebacivanje iz jedne klase u drugu ali taj proces može biti skup pa treba ga pažljivo koristiti.

Počinjemo prvo sa `Image` klasom. U nju možemo učitavati slike ali 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. Također možemo predati ze treći argument niz brojeva tipa `sf::Uint8` koji bi prezentirali individualne piksele kao RGB vrijednosti.

Možemo, naravno, učitati slike se tvrdog diska.

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

Bitan dio `Image` je da nam omogućuje razne funkcije za upravljanje slikom poput mijenjanja vrijednosti individualnog piksela sa `Image::setPixel()`. Nudi nam pristup nizu vrijednosti koje predstavljaju sve piksele ili jednom individualnom pikselu. Sliku možemo okrenuti vertikalno sa `Image::flipVertically()` ili horizontalno sa `Image::flipHorizontally()`. Na kraju svega možemo i sliku spremiti na disk preko funkcije `Image::saveToFile()`. Nakon što smo upravljali slikama sada bi trebalo napraviti od njih teksture.

Klasa `Texture` nam nudi slične funkcije kao i `Image`. Obe mogu pozivati `loadFromFile()` funkciju samo postoji jedna bitna razlika. U klasi `Texture` imamo mogućnost proslijediti još jedan argument uz ime datoteke a to su dimenzije i pozicija pravokutnika koji predstavlja samo dio slike. Taj pristup nam omogućuje da imamo više tekstura za jednu sliku i da

možemo koristiti jednu sliku i učitavati samo dijelove. To je posebno koristimo jer onda možemo cijeli ciklus animacije jednog lika imati samo u jednoj slici.

```
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 koju želimo. Još nam je ostalo samo prikazati tu teksturu na zaslonu.

Teksture i oblici

Već smo spomenuli da imamo mogućnost učitavanja tekstura preko oblika. Možemo naše teksture postaviti na oblike poput pravokutnika i tako ih prikazivati na zaslonu. Oblicima možemo limitirati količinu slike koja će se prikazati. Za primjer želimo sada ovu sliku šume nacrtati u krugu:

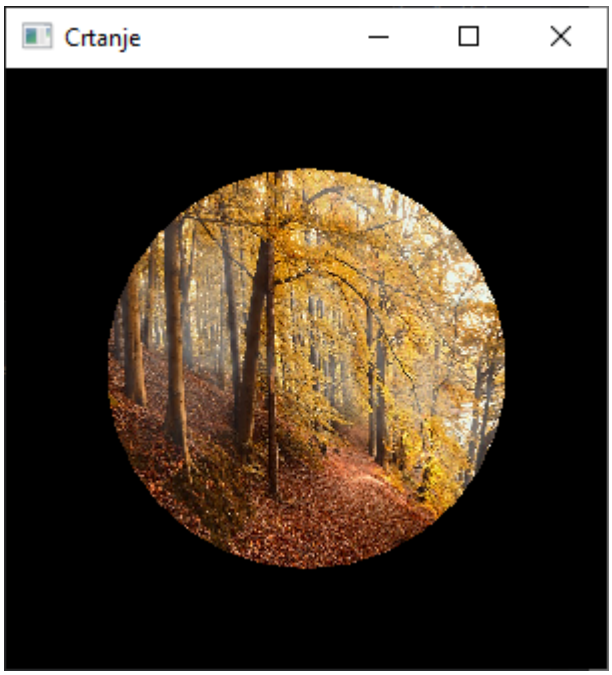
Slika 2.5: Autumn forest

Slijedi kod koji to omogućuje:

```
#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 ovog koda dobijemo sljedeći rezultat:



Slika 2.6: Autumn forest circle

Uz oblike imamo opciju crtanja tekstura preko spriteova.

Spriteovi

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

kao i oblike. Bez obzira što oblici nude više mogućnosti mi ćemo svejedno češće koristiti spriteove zbog njihove jednostavnosti.

Njihova je svrha samo učitati teksturu i prikazati je (uz moguće neke osnovne transformacije). To se vidi i u tome da, dok oblike možemo koristiti i bez tekstura, spriteovi ih zahtjevaju za korištenje.

Pogledajmo jedan osnovni primjer u kojem je očita ta jednostavnost:

```
sf::Texture texture;

// Load the texture

// 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, onda se situacija 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. Mi 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 ovakvoga 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 1 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 igrinoj petlji. O tome više u drugom poglavlju. 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 ovom primjeru su korišteni oblici ali isto bi bilo i sa spriteovima. Samo bi umjesto crtanog oblika učitali teksturu i prikazali je preko spritea i na kraju taj sprite transformirali.

2.5 Zvuk

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

Slično kao i za slike, za zvuk postoje dvije mogućnosti učitavanja, 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 (en. stream) prema datoteci na tvrdom disku i učitava dio po dio zvučne datoteke. Obje nasljeđuju klasu `SoundSource` preko koje imaju neke osnovne glazbene fukncionalnosti. Također pokreću zvuk u zasebnoj dretvi tako da ne blokiraju 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` se koristi kada su u pitanju velike datoteke, a programer je ograničen količinom memorije. Budući da ova klasa učitava dijelove datoteke, ona ima prirodni zastoj zbog toga posla koji obavlja. Ponuđena nam je i mogućnost puštanja 3D zvukova. Takvi zvukovi se čuju iz više različitih smjerova i daju dojam treće dimenzije zvuka. Unutar SFML-a možemo i snimati zvuk koristeći klasu `SoundRecorder` ali ovdje će biti fokus na puštanje zvuka.

sf::Sound

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

Prvo pomoću `SoundBuffer` učitamo sa diska datoteku preko fukncija oblika `SoundBuffer::loadF`
Nama je najbitnija `loadFromFile`. Nakon što konstruiramo `Sound` objekt kojemu proslijedimo prethodni zvuk kojeg smo učitali radimo što nam treba. Zvukove možemo puštati, zaustavljati ili pauzirati. Nude se i funkcije koje javljaju status pjesme (je li pauzirana i sl.) i još neke druge. Osnovni primjer isjecka koda bi izgledao ovako:

```
sf::SoundBuffer buffer;
```

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

sf::Sound sound(buffer);

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

sf::Music

Klasa `sf::Music` je klasa koju koritimo zasebno (ne treba nam druga klasa za korištenje) za puštanje zvukova. Preporučuje se kod većih zvukova kada smo ograničeni memorijom. Za upravljanje glazbom koristimo ise fukncije kao i prije. Za razliku od `SoundBuffer` klase koja učita cijelu pjesmu u memoeriju, 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 da neke igre koriste internet. On nam omogućuje brzu komunikaciju sa drugim računalima diljem svijeta i time možemo igrati igre zajedno ili protiv drugih ljudi. Danas je povezanost na internet čest dio igara. Jedna bitna stavka je da preko njega skidamo nova ažuriranja za igre u redovnim intervalima i tako naše igre dobivaju ili popravke bugova ili nove sadržaje što im produžuje vijek. Prije se za sve moralo kupovati novo izdanje u trgovinama da bi se riješili takvi problemi. Unatoč tome najbitnija novina koju je internet donio u igre je komponenta igranja sa drugim igračima. To je omogućilo stvaranje kompetitivnih online igara. Neke igre danas su čak samo moguće za igranje preko interneta. Iz svega toga je izrasla i e-sport scena u kojoj se može natjecati u igrama i osvojiti novčane nagrade.

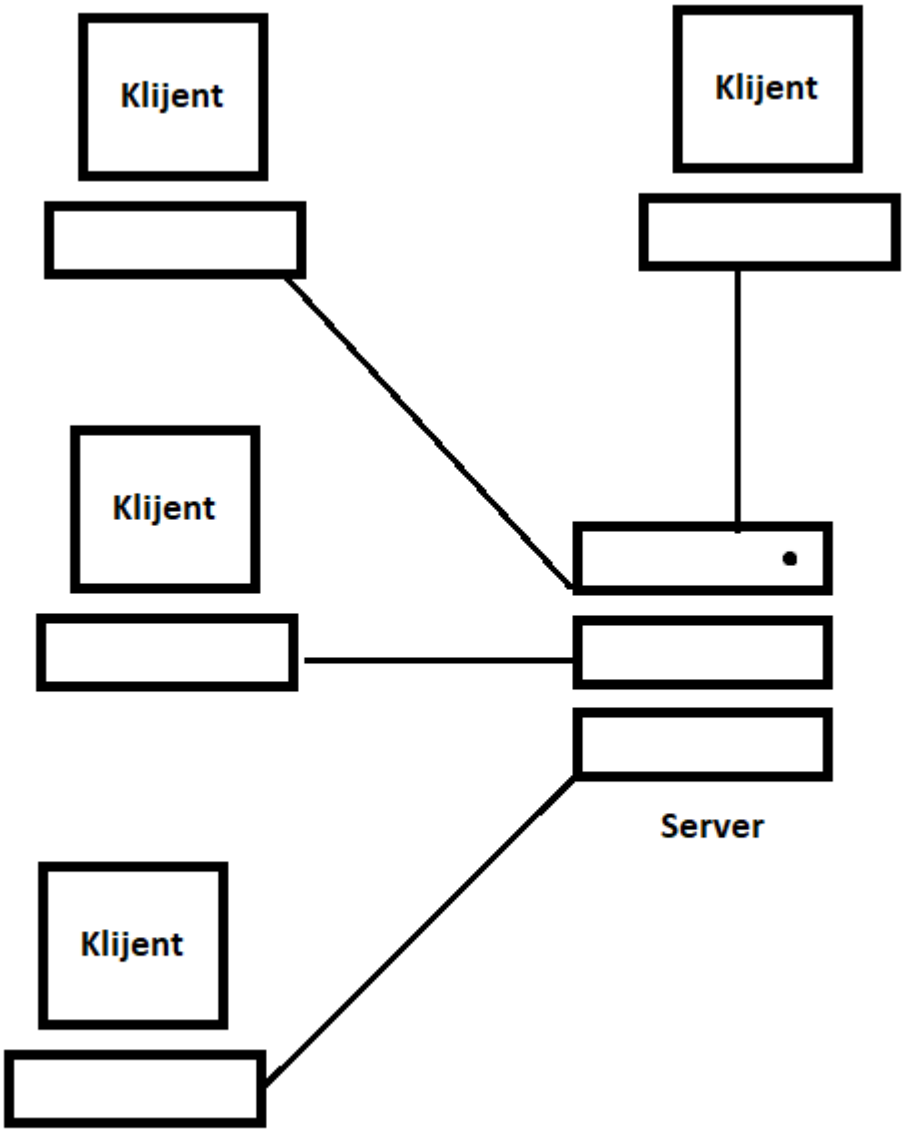
SFML nudi razne mogućnosti i detalje kada je u pitanju komunikacija preko interneta. Mi ćemo ovdje proći neke osnovne. Arhitektura koju ćemo koristiti za komunikaciju će biti klijent-server. Ona funkcionira tako da se klijenti spajaju na jedan centralni server koji sinkronizira događaje i šalje njima trenutna stanja. Da bi znali gdje putuje informacija trebamo znati IP adresu koja će voditi naš paket do odredišta. U kontekstu igara bi klijenti slali informacije poput pucnja, skoka i slično dok bi server primao informacije i analizirao bih. Tako bi znali je li metak nekoga pogodio ili ne i nakon toga server šalje svim klijentima trenutno stanje u igri. Taj odnos je kao na prikazanome dijagramu:

Kada je u pitanju transport onda imamo dva protokola i to su TCP i UDP. Razlike su u pouzdanosti i performansama. Oni rješavaju probleme prebacivanja informacija sa jednog računala na drugo. U svakom od primjera ćemo slati neke jednostavne podatke ali u pravilu ćemo slati pakete. U SFML-u njihova podrška je preko klase `sf::Packet` i u njega možemo pohraniti više informacija različitog tipa i slati ih preko interneta. Također, u svakome od njih se drukčije programira pa ćemo ih pogledati zasebno.

TCP

Transmission Cotrol Protocol (TCP) koristimo kada želimo osigurati da je paket stigao na svoje 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 gdje treba onda se on šalje opet sve dok ne dođe. On osigura da dođu svi paketi i to u pravilnom redoslijedu. Zbog ovoga svega TCP je sporiji i koristimo ga gdje god brzina nije presudna. U većini slučajeva neće biti osim u brzim, akcijskim igrama u kojima je svaki komadić vremena bitan. Većina interneta komunicira preko ovog protokola zato što je većinom bitnija točnosti pristiglih informacija nego što ćemo izgubiti malo vremena.

Za spajanje nam trebaju IP adresa odredišta i port na koji se spajamo. TCP se obavlja preko dvije klase `sf::TcpSocket` (utičnica) koja uspostavlja vezu (odnosno klijent) i



Slika 2.7: Client server model

`sf::TcpListener` koji prihvaća vezu (server). Ako je veza uspješna onda `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 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:

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

// Prihvaćamo vezu
sf::TcpSocket socket;

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

// Uspostavljena 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 onda kada ih spoji radi razmjenu podataka. Nakon obrade zatvaramo utičnicu i prekidamo vezu.

UDP

User Datagram Protocol (UDP) je nepouzdani 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 zahtjeva manje memorije za svaki paket. Zbog toga i odredišno računalo ne treba čekati izgubljene pakete nego samo nastavlja izvršavanje. Koristimo ga kada je brzina bitna u našim aplikacijama. Ako nam treba puno informacija uskladiti u kratkom vremenu onda nam je UDP poželjniji.

Slanje podataka preko UDP-a je slično kao i preko TCP-a ali samo koristimo `sf::UdpSocket`. Za razliku od TCP-a u kojemu jedna utičnica može biti povezana samo sa drugom nekom, 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 osnovni primjer mrežne komunikacije u UDP a prethodno smo pokazali i u TCP. 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 za dodavanje 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 [4] ili za bilo koji program koji možda pišemo u nekom objektno orijentiranom jeziku. [1]

3.1 Game loop

Motivacija

Game loop je centralni obrazac za igre koji nam služi za pokretanje igara. Dok game enginei sami obavljaju glavnu petlju, SFML nam omogućava da mi osobno isprogramiramo glavnu igrinu petlju. 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 svaka igra ga 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 bez njega. To je zapravo prva ključna ideja igrine petlje, ona čita korisnikov unos bez da ga čeka. Točnije, petlja se vrti i obavlja sve radnje, ali neće stati s izvođenjem da 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 je u petlji samo čekano 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 igrina petlja 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.

Mi također često želimo 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 količine FPS-a su 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 igrine petlje održavanje konzistentne brzine igre neovisno o hardveru.

Bitno je imati na umu i da će se ova petlja vrtiti 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 vrtit će se onoliko brzo koliko je računalo u mogćnosti. Problem s takvim pristupom je što nemamo kontrolu nad brzinom kojom će se igra vrtiti. Na brzim računalima će se igra vrtiti prebrzo, a na slabijim će računalima biti prespora za igranje. Ako je u pitanju pokretanje nekog lika, onda će se na brzom računalu lik prebaciti preko cijelog zaslona u manje od pola sekunde, a na slabom računalu trebat će nekoliko sekundi. Zbog toga je poželjnu 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 mi 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 omogućuje da se petlja ne izvršava prečesto. Sada dolazimo do velike mane ovog 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 duže vremena za obradu jedne sličice. Jedno rješenje toga je pomicanje 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 fukncji 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 duljeg 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 korsnik 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 omogućuje 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;
  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 različita 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 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 nedozvo-

ljenim 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 mi 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 mi početi pisati neku sliku u framebuffer, npr. neki automobil. Mi u isto vrijeme ažuriramo igru, računamo raznorazne vrijednosti te crtamo sve to na zaslonu. Dok mi dođemo do, na primjer, guma od automobila, slika se već nastavila crtati bez da smo mi uspjeli napisati u frambuffer što dalje želimo. Zato ćemo dobiti sliku automobila koji nema gume. Računalo

zapravo nastavi čitati framebuffer, ali kako mi nismo stigli u njega sve napisati, tamo nisu vrijednosti koje mi 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 vremenski koštati. Dodatno, 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 videodriver 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.

Svaku novu sličicu 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 videodriver 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 videodriver 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 videodriver 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 videodriver 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 omogućuje 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 da bi predstavljali stanja u kojima može biti naša igra ili objekti u našoj igri. Igra može imati više različitih situacija u kojima se drugačije ponaša. Na primjer, najčešće će imati glavni izbornik iz kojeg možemo ili ugasiti igru ili prijeći u stanje igranja igre. Mi možemo ono što korisniku prikazujemo pamtiti jednom varijablom ali to stvara probleme ako želimo proširiti kod tako da dodamo u izborniku opciju da korisnik može mijenjati postavke ili pogledati upute kako igrati igru. Dodana su dva nova stanja koja opet pratimo sa 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 loša praksa. Uvijek želimo da je naš kod pregledniji i pogodniji za razvijanje i zato je bolje odvojiti u zasebne klase.

Takve klase će biti razna stanja i kod treba imati mogućnost da ih mijenja. Objekt će izgledati kao da je mijenjao klasu ali zapravo je samo došlo do promjene njegovog stanja. To će nam smanjiti velike dijelove koda u kojima provjeravamo puno uvjeta kod objekata prije neke radnje tako da će ih odvojiti u zasebne klase.

Primjer

Za primjer uzimamo igru u kojoj je glavni lik čovjek koji se može transformirati u lava čiji će napad biti jači. Prvi dio će pokazati zašto koristimo ovaj obrazac. Zamislimo klasu `Player` koja predstavlja našeg glavnog lika koji ima mogućnost transformiranja u lava u čijem obliku ima jači napad i brže kretanje. Jedan osnovni način rješavanja ovog problema bi bilo koristiti enumeracije:

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

Klasa `Player` bi u tome 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 u kodu da se stvari lako zakompliciraju kada obrađujemo stanja na ovakav način. Još bi veći problem bio kada bi htjeli da se naš lik može transformirati u vuka ili medvjeda. Dodatne komplikacije 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 da bi poboljšali kvalitetu koda.

Uz već spomenutu klasu ćemo imati i `HumanState` i `LionState` koje će implementirati `processInput()` funkciju. `Player` klasa će sadržavati 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. Omogućuje nam jednostavno dodavanje novih stanja.

U zadnje primjeru se 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 glavnog

lika.

Zaključak

Glavna primjena ovog obrasca je zbog bolje organizacije koda. Bez njega neke funkcije mogu biti ogromne i neodržive. Ako je funkcija dovoljno velika i ima puno `if`/`else` ili prevelik `switch` onda nam treba `State` oblikovni obrazac. Točnije ako neki objekt treba mijenjati ponašanje ovisno o stanju. Sa njime jednostavno mijenjamo stanje glavno lika i dodajemo nove dijelove po potrebi.

3.4 Component

Motivacija

Primjer

Zaključak

Poglavlje 4

Implementacija igre u SFML-u

Bibliografija

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

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

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

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

Sažetak

U ovom radu upoznali smo se s mogućnostima programiranja igara u SFML-u. Pokazali smo što sve SFML može te sve njegove glavne dijelove. Uz to je još objašnjeno kako organizirati kod kada je u pitanju programiranje igara kako bi kod bio spreman za održavanje.

Summary

In this ...

Životopis

Zvonimir Šimunović rođen je 27.10.1993. 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.