Razvoj grafičke aplikacije u biblioteci Qt5

Štifanić, Tihana

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Tihana Štifanić

RAZVOJ GRAFIČKE APLIKACIJE U BIBLIOTECI Qt5

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, rujan 2021.

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

Qt je skup biblioteka namijenjen razvoju aplikacija s grafičkim korisničkim sučeljem za velik broj različitih platformi, kako onih za stolna i prijenosna računala, tako i mobilnih, kao i onih prilagođenih uklopljenim (eng. *embedded*) sustavima. Biblioteka Qt je podijeljena u više modula, od kojih svaki nudi različite funkcionalnosti, od izrade grafičkog korisničkog sučelja, preko rada s mrežom i multimedijom, do modula za upravljanjem bazom podataka.

Ovaj rad podijeljen je u četiri poglavlja, od kojih je prvo posvećeno osnovnim informacijama i mehanizmima uvedenima Qt bibliotekom. Dan je kratak pregled odabranih modula i primjer minimalne Qt aplikacije. Potom je detaljnije opisan centralni mehanizam Qt biblioteke, sustav signala i utora, koji služi povezivanju različitih objekata i uvelike se koristi kod izgradnje grafičkih korisničkih sučelja. Nadalje, ukratko je opisan *meta-object* sustav koji omogućuje korištenje sustava signala i utora, kao i Qt-ov sustav za izgradnju aplikacija. Konačno, prikazani su glavni dijelovi razvojne okoline namijenjene razvoju Qt aplikacija, Qt Creatora, i njegovog alata za izradu grafičkih korisničkih sučelja, Qt Designera.

U drugom poglavlju opisan je proces izgradnje grafičkog korisničkog sučelja uz pomoć modula Qt Widgets. Dan je kratak pregled dostupnih grafičkih elemenata, kao i primjeri njihovog korištenja. Osim toga, opisani su elementi za definiranje razmještaja grafičkih elemenata koji osiguravaju konzistentnost sučelja pri različitim veličinama glavnog prozora aplikacije. Prikazan je i primjer stvaranja sučelja kroz Qt Designer te načini kojima se tada može upravljati grafičkim elementima kroz programski kod.

Treće poglavlje posvećeno je sustavu za prikaz grafičkog sadržaja i crtanje na ekran, obuhvaćeno modulom Qt GUI. Opisan je način na koji Qt iscrtava grafičke elemente na ekran, kao i glavne klase koje u tome sudjeluju. Dalje je dan pregled metoda za crtanje, uključujući načine crtanja primitiva, teksta i slika. Također, opisani su mehanizmi za promjenu svojstava crtanja, poput debljine i boje olovke.

Konačno, u svrhu demonstracije opisanih mehanizama, u sklopu rada izrađena je jednostavna aplikacija za crtanje. Njene glavne funkcionalnosti i izgled grafičkog korisničkog sučelja predstavljeni su u četvrtom poglavlju. U njemu su također prikazani određeni dijelovi implementacije, poput korištenih grafičkih elemenata i crtanja na ekran.

Poglavlje 1

Općenito o Qt biblioteci

Qt je skup biblioteka ili razvojni okvir (eng. *development framework*) koji podržava razvoj aplikacija s grafičkim korisničkim sučeljem za veliki broj platformi. Tako omogućuje razvoj za Linux, Windows i macOS operativne sustave, za mobilne platforme (Android i iOS) te za određene platforme koje se koriste na uklopljenim (eng. *embedded*) sustavima. Qt je napisan u programskom jeziku C++ te je primarno namijenjen za korištenje s istim. Međutim, dostupan je i velik broj modula izrađen od strane drugih kompanija koji omogućuje korištenje biblioteke Qt s drugim programskim jezicima (vidi [1]). Neki od dostupnih modula su PyQt, namijenjen razvoju u Pythonu i QtJambi, za korištenje biblioteke Qt s programskim jezikom Java. U ovom radu je korištena Qt5 biblioteka, čija je najnovija i konačna verzija Qt 5.15, izdana u ožujku 2020. godine. U prosincu 2020. godine predstavljena je novija verzija, Qt6, koja donosi niz promjena u određenim dijelovima biblioteke.

Biblioteka Qt5 se primarno koristi za razvoj aplikacija s grafičkim korisničkim sučeljem, no osim alata za izradu i rad s istim, Qt5 nudi i mnogo drugih mogućnosti. Tako su, primjerice, dostupni alati za rad s multimedijom, mrežom, bazama podataka, kao i alati za pisanje i provedbu jediničnih testova (eng. *unit tests*). Ove funkcionalnosti su podijeljene u module, a neki od njih su

- **Qt Core** — bazični modul koji sadrži neke od osnovnih značajki specifičnih za Qt razvojni okvir. Osim toga, sadrži i vlastite implementacije često korištenih spremnika, poput liste, vektora, stoga, reda i drugih,
- **Qt GUI** — modul čije klase Qt uglavnom koristi interno za iscrtavanje grafičkog korisničkog sučelja. Za sam razvoj sučelja, preporuča se korištenje nekih od modula koji nude alate na višoj razini, navedenih u nastavku. Međutim, može se i koristiti direktno, na primjer onda kada je potrebna veća razina kontrole ili za crtanje na ekran,

- **Qt Widgets** — nudi klase za izradu grafičkog korisničkog sučelja na višoj razini. Sadrži raznolike grafičke elemente te načine razmještaja istih na ekran kako bi se osigurala konzistentnost sučelja kod različitih veličina ekrana. Pomoću ovog modula, grafička korisnička sučelja se stvaraju na imperativan način, koristeći dostupne metode koje mijenjaju stanje sustava,
- **Qt Quick** — modul koji se također koristi za izradu grafičkih korisničkih sučelja, ali se pomoću njega sučelja stvaraju na deklarativan način, koristeći druge alate. Dijelovi sučelja se opisuju jezikom QML, a mogu sadržavati i dijelove koda napisane u JavaScriptu,
- **Qt QML** — nudi klase za razvoj pomoću jezika QML i JavaScript, a najčešće se koristi kod stvaranja grafičkih korisničkih sučelja koristeći modul Qt Quick,
- **Qt Multimedia** — sadrži klase za rad s multimedijom. Primjerice, podržava prikaz audio zapisa, slika i video zapisa. Također, nudi i alate za pristup resursima uređaja, poput kamere ili mikrofona, u svrhu stvaranje novih multimedijskih zapisa,
- **Qt Network** — omogućava rad s aplikacijama koje koriste TCP/IP grupu mrežnih protokola, nudeći operacije poput slanja zahtjeva i datoteka putem HTTP protokola te rada s kolačićima,
- **Qt SQL** — modul koji podržava rad s bazama podataka temeljenim na SQL (eng. *Structured Query Language*) jeziku,
- **Qt Test** — nudi klase za pisanje i provedbu jediničnih testova koji se koriste za testiranje Qt aplikacija.

Osim ovih, postoje i dodatni (eng. *add-on*) moduli, od kojih su neki dostupni samo na određenim platformama. Oni uglavnom proširuju postojeće funkcionalnosti, nudeći primjerice podršku za rad s 3D grafikom, vizualizaciju podataka ili rad s drugim hardverskim resursima uređaja, na primjer Bluetoothom ili senzorima.

Prije nastavka, još je važno napomenuti da su sve Qt aplikacije programi vođeni događajima (eng. *event-driven*). To znači da je glavni dio svake Qt aplikacije jedna beskonačna petlja, takozvana petlja događaja (eng. *event loop*). Unutar nje, aplikacija čeka na interakciju korisnika s grafičkim sučeljem, čime se osigurava pravovremena reakcija na korisnikove zahtjeve. Primjer minimalnog izvornog koda bilo koje Qt aplikacije vidljiv je u isječku koda 1.1. Moguće je uočiti upotrebu klase `QApplication`, koja se brine za inicijalizaciju svih potrebnih elemenata za prikaz aplikacije. Također je zadužena za pokretanje petlje događaja, koje se ostvaruje pozivom metode `QApplication::exec()`.

```
#include <QApplication>

int main(int argc, char **argv)
{
    QApplication app (argc, argv);
    return app.exec();
}
```

Isječak koda 1.1: Primjer minimalnog izvornog koda Qt aplikacije

U nastavku ovog poglavlja pobliže su opisani neki od centralnih dijelova Qt biblioteke, poput mehanizma signala i utora, kao i *meta-object* sustav koji omogućuje njegovo korištenje. Također, ukratko se opisuje proces automatske izgradnje Qt projekata, kao i projektna datoteka koja je za to potrebna. Konačno, dan je kratak pregled alata često korištenog za razvoj Qt aplikacija — Qt Creatora — koji dodatno olakšava i ubrzava proces razvoja.

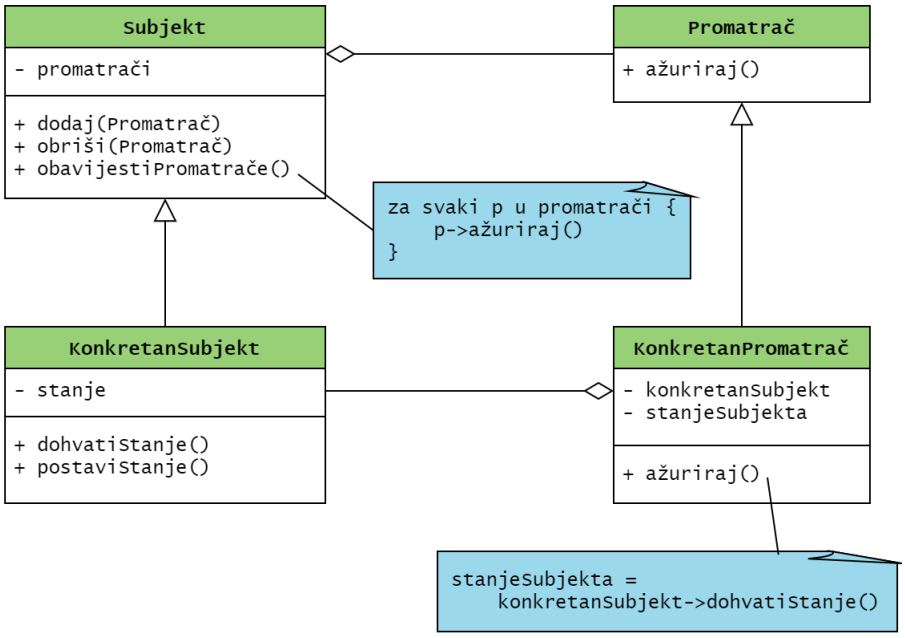
1.1 Sustav signala i utora

Kad god je riječ o grafičkim korisničkim sučeljima, važnu ulogu igra uspostavljanje mehanizma za povezivanje elemenata korisničkog sučelja s ponašanjem aplikacije. Primjerice, sadrži li sučelje gumb čija je namjena zatvaranje prozora unutar kojeg se aplikacija prikazuje, tada je potrebno detektirati pritisak gumba i definirati takav odgovor aplikacije na isti. Kao rješenje ovog problema, često se nameće oblikovni obrazac promatrača.

Oblikovni obrasci (eng. *design patterns*) su recepti za rješenja nekih problema koji se često susreću u razvoju računalnih sustava. Oblikovni obrazac promatrača (eng. *observer*) primjenjiv je u situacijama kada različiti objekti (*promatrači*) ovise o stanju nekog drugog, istaknutog, objekta (*subjekta*). Tada je prilikom svake promjene stanja subjekta nužno o tome obavijestiti sve promatrače, kako bi mogli pravilno i pravodobno reagirati.

Dijagramatski prikaz oblikovnog obrasca promatrača vidljiv je na slici 1.2. Apstraktni subjekt sadrži listu referenci na pretplaćene promatrače i metode za njeno modificiranje, kao i metodu za obavještavanje svih pretplaćenih promatrača. Konkretan subjekt sadrži stanje (koje je promatračima od interesa) te metode za njegovo dohvaćanje i postavljanje. Prilikom svake promjene stanja subjekta, poziva se metoda `ažuriraj()` svih pretplaćenih promatrača. Važno je primijetiti da subjekt ne treba znati ništa o prirodi konkretnog proma-

trača, već je važno jedino da on implementira svoju metodu `ažuriraj()`. S druge strane, konkretan promatrač sadrži referencu na konkretni subjekt čije su mu promjene stanja od zanimanja. Ovo je potrebno kako bi se konkretan promatrač uopće mogao inicijalno pretplatiti, pozivajući subjektovu metodu `dodaj()`, kao i u svrhu dohvaćanja njegovog trenutnog stanja. Također, konkretan promatrač pamti trenutno stanje subjekta, koje osvježava u svojoj metodi `ažuriraj()`.



Slika 1.2: Dijagram oblikovnog obrasca promatrača

Kod grafičkih korisničkih sučelja, subjekti su često grafički elementi. Na primjer, polje za unos teksta može preuzeti ulogu subjekta koji sve svoje promatrače mora obavijestiti o događaju unosa, brisanja ili općenito promjene teksta. Promatrač u ovom slučaju je sama aplikacija te je tada moguće definirati način na koji ona treba odgovoriti na ove događaje.

Biblioteka Qt5 koristi oblikovni obrazac promatrača kroz svoj sustav signala i utora. Sustav signala i utora (eng. *signals and slots*) jedan je od centralnih mehanizama uvedenih s Qt bibliotekom, kao i značajka koja najviše razlikuje Qt od ostalih, sličnih, razvojnih okvira (vidi [6]). Ovaj sustav se koristi kao način ostvarenja komunikacije među objektima, a njegova najčešća primjena je upravo u izradi grafičkih korisničkih sučelja.

Nastavno na oblikovni obrazac promatrača, u sustavu signala i utora, subjekt prilikom promjene svojeg stanja emitira određeni *signal*. Tada promatrač, koji je zainteresiran za promjenu stanja tog objekta, može definirati *utor* — funkciju koja se pokreće kao odgovor na određeni signal. Kako bi ovo funkcioniralo, odgovarajuće signale i utore je po-

trebno spojiti, što se u Qt-u odvija izvan objekata koji igraju uloge promatrača i subjekata. Važno je primijetiti da u ovoj primjeni postoji određeno odstupanje od opisanog oblikovnog obrasca promatrača. Naime, subjekt ne mora znati ništa o promatračima koji su zainteresirani za njega niti održavati listu takvih objekata, već samo emitira signal koji je dohvatljiv svima. Slično, promatrač ne treba posjedovati referencu na subjekt, već je samo važno da je njegov utor povezan sa signalom za kojeg je zainteresiran.

Qt5 sadrži velik broj predefiniranih signala i utora, koje je, kao što je ranije navedeno, potrebno samo spojiti u slučaju da trebaju ovisiti jedan o drugome. Za spajanje signala i utora, koristi se metoda `QObject::connect()`. Klasa `QObject` je posebna klasa unutar Qt biblioteke, bez koje nije moguće koristiti sustav signala i utora, o kojoj je više riječi u narednom potpoglavlju.

Primjer spajanja predefiniranog signala i utora u jednostavnoj Qt aplikaciji vidljiv je u isječku koda 1.3. Kao i u isječku koda 1.1 koji prikazuje minimalnu Qt aplikaciju, koristi se klasa `QApplication` za inicijalizaciju osnovnih dijelova aplikacije. Nakon toga, stvoren je objekt tipa `QPushButton` koji predstavlja gumb iz modula Qt Widgets. Objekti tog tipa imaju predefinirani signal `clicked()`, koji se emitira prilikom pritiska gumba. S druge strane, korišten je predefinirani utor `quit()`, dostupan u klasi `QApplication`, koji gasi aplikaciju. Kako bi se signal `clicked()` povezao s utorom `quit()`, funkciji `QObject::connect()` redom treba predati adresu pošiljatelja (gumba), ciljanog signala, primatelja (aplikacije) te željenog utora.

```
1 int main(int argc, char * argv[])
2 {
3     QApplication app(argc, argv);
4     QPushButton *gumbZatvori = new QPushButton("Zatvori prozor");
5
6     QObject::connect(gumbZatvori, &QPushButton::clicked, &app,
      ↪  &QApplication::quit);
7
8     return app.exec();
9 }
```

Isječak koda 1.3: Primjer spajanja predefiniranih signala i utora

Osim predefiniranih signala i utora, moguće je definirati i vlastite. I signali i utori su u osnovi obične funkcije s povratnim tipom `void`, s tom razlikom da funkciju koja pred-

stavlja signal ne treba implementirati, već ju je dovoljno samo deklarirati, dok se za ostalo brine Qt. Pored deklaracije, signal je na odabranom mjestu potrebno emitirati koristeći ključnu riječ `emit`. Vlastite signale i utore moguće je spojiti na isti način kao i predefinirane, pozivanjem metode `QObject::connect()`. Funkcije signala i utora mogu primati argumente, ali prilikom spajanja je potrebno osigurati da primaju istu listu parametara.

1.2 *Meta-Object* sustav

Iako jednostavan za korištenje, sustav signala i utora je vrlo kompleksan (vidi [7]). Zbog toga Qt uvodi takozvani *meta-object* sustav. Iako je primaran razlog njegova uvođenja bio implementacija sustava signala i utora, donosi i druge dobrobiti poput sigurnog otkrivanja i pretvaranja tipova podataka za vrijeme izvršavanja programa, uvođenje svojstava (eng. *property*) u klase i mnogih drugih. Glavni dijelovi na kojima sustav počiva su

- klasa `QObject`,
- makro `Q_OBJECT`,
- *meta-object* prevodioc,

detaljnije pojašnjeni u nastavku.

Klasa `QObject` i makro `Q_OBJECT`

Kako bi neki objekt mogao koristiti svojstva i dobrobiti *meta-object* sustava, potrebno je da je tipa klase koja nasljeđuje `QObject`. Pored toga, nužno je smjestiti `Q_OBJECT` makro u privatni dio ciljane klase. Ovaj makro će prilikom prevođenja biti zamijenjen C++ kodom koji omogućuje korištenje sustava signala i utora, za što se brine *meta-object* prevodioc, detaljnije opisan u nastavku.

Osim mogućnosti korištenja sustava signala i utora, klasa `QObject` uvodi i sustav roditeljstva (eng. *parenting system*). Objekti svake njene podklase mogu imati definiranog roditelja i djecu te su dostupne metode za pronalazak istih. Ono što je kod ovog sustava od iznimne važnosti jest da se prilikom uništavanja takvog objekta i sva njegova djeca uništavaju zajedno s njim. Na taj je način programeru olakšano korištenje ovakvih objekata te je znatno smanjena mogućnost nepoželjnog curenja memorije (eng. *memory leak*). Osim toga, kada je riječ o izgradnji grafičkog korisničkog sučelja i prikazu elemenata na ekran, svaki objekt koji predstavlja grafički element (primjerice, gumb ili polje za unos teksta) se automatski prikazuje unutar svojeg roditelja. U tom slučaju, onaj element čiji roditelj nije definiran postaje glavni prozor aplikacije.

Meta-object prevodioc

Meta-object prevodioc (eng. *Meta-Object Compiler*) je alat koji se brine za implementaciju ranije navedenih svojstava uvedenih *meta-object* sustavom. Jedna od njegovih zadaća jest, primjerice, čitanje datoteka zaglavlja (eng. *header file*) i generiranje novih datoteka izvornog koda onda kada naiđe na spomenuti makro `Q_OBJECT`. Nakon toga, generirane datoteke moraju biti prevedene (eng. *compiled*) i povezane (eng. *linked*) zajedno s implementacijom ciljane klase. Koristeći Qt-ov sustav za automatsku izgradnju, detaljnije opisan u nastavku, *meta-object* prevodioc nije potrebno koristiti i pozivati direktno, već je to sastavni dio procesa izgradnje.

1.3 Sustav za izgradnju

Kako se Qt projekti obično sastoje od većeg broja datoteka, biblioteka Qt5 dolazi u paketu s alatom `qmake` koji olakšava i automatizira proces izgradnje projekta. Sastavni dio Qt-ovog sustava za izgradnju je projektna datoteka, prepoznatljiva po nastavku `.pro`. Unutar nje je dovoljno samo popisati korištene module, datoteke izvornog koda i zaglavlja te preostale datoteke koje su eventualno dio projekta.

Primjer jedne projektne datoteke vidljiv je u isječku koda 1.4. U ovom primjeru, podrazumijeva se da su sve navedene datoteke u istom direktoriju kao i projektna datoteka. U suprotnom je potrebno navesti putanju do ciljane datoteke. Sve vrijednosti koje se nalaze s lijeve strane znaka za pridruživanje su zapravo predefinirane varijable, a one predstavljaju

- `QT` — popis modula koji se koristi u projektu. U ovom slučaju, to su moduli Qt Core, Qt GUI te Qt Widgets,
- `CONFIG` — općenite konfiguracijske opcije. U ovom primjeru, vrijednost `c++11` označava da u ovom projektu treba biti podržano korištenje C++11 standarda,
- `SOURCES` — popis datoteka izvornog koda koje projekt sadrži. U ovom slučaju to su datoteke `main.cpp` te `mainwindow.cpp`,
- `HEADERS` — popis zaglavlja koje projekt sadrži. Jedino zaglavlje u ovom slučaju je `mainwindow.h`.

Osim ovih, postoji još nekoliko predefiniranih varijabli kojima se u projektnoj datoteci može pridijeliti vrijednost. Osim toga, moguće je uključivati i druge Qt projekte, uvjetno uključivati datoteke (ovisno o, primjerice, platformi), dodavati druge biblioteke i još mnogo toga (vidi [3]).

Nakon što je projektna datoteka spremna, na temelju nje je naredbom `qmake` moguće automatski generirati *makefile*. Na platformama baziranim na Unixu, *makefile* datoteka je

```
QT       += core gui widgets

CONFIG += c++11

SOURCES += \
    main.cpp \
    mainwindow.cpp

HEADERS += \
    mainwindow.h
```

Isječak koda 1.4: Primjer projektne `.pro` datoteke

datoteka koja sadrži naredbe ljuske (eng. *shell commands*) potrebne za cjelovito prevođenje i spajanje datoteka izvornog koda. Konačno, koristeći naredbu `make` započinje izgradnja cijelog projekta na temelju generiranog *makefile*a. Time je proces izgradnje maksimalno olakšan, i nije potrebno brinuti o prevođenju i spajanju eventualnog velikog broja datoteka koje čine jedan Qt projekt.

1.4 Razvojno okruženje Qt Creator

Qt Creator je integrirana razvojna okolina (eng. *Integrated Development Environment*) dostupna za platfome Linux, macOS i Windows. Primarno je namijenjen razvoju C++ aplikacija, a posebno je prilagođen razvoju Qt aplikacija. Koristeći metode opisane u prethodnim poglavljima, Qt projekte moguće je stvarati ručno, pišući vlastitu projektnu datoteku i prevodeći je koristeći opisani sustav za izgradnju. Međutim, korištenje Qt Creatora znatno ubrzava i olakšava proces razvoja Qt aplikacija.

Nudi mnogo značajki za lakše pisanje koda poput automatskog dovršavanja koda (eng. *code completion*) za vrijeme pisanja, bojenje ključnih riječi (eng. *syntax highlighting*), provjeru i analizu koda te alate za lakše refaktoriranje koda (primjerice, promjene imena varijabli). Osim toga, nudi alat za otkrivanje pogrešaka (eng. *debugger*), alat za analizu performansi programa (eng. *performance analyzer*) te alat za automatsko provođenje jediničnih testova i prikaz njihovih rezultata.

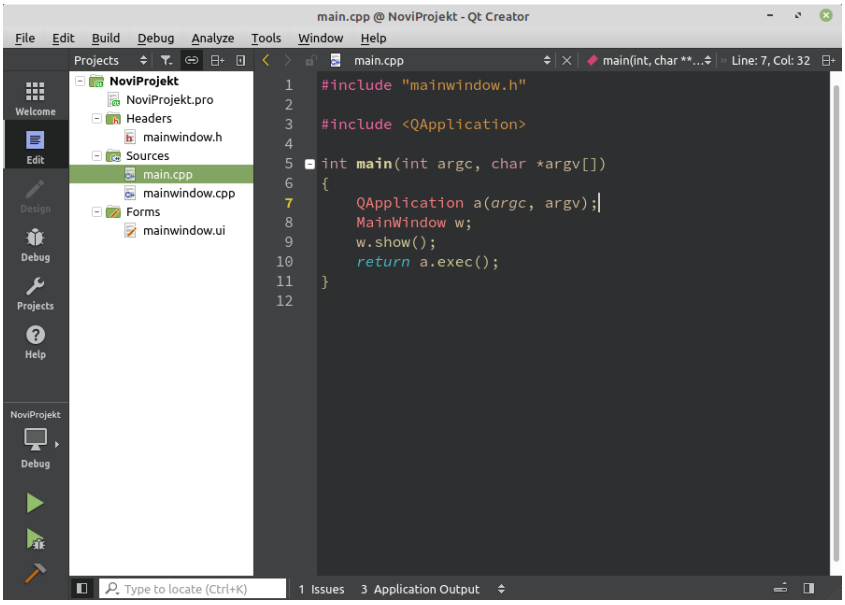
Stvaranje novog, praznog, Qt projekta pomoću Qt Creatora vrši se kroz izbornik, biranjem opcije File → New File or Project... . Potom je moguće odabrati tip projekta, a

neke od ponuđenih vrsta aplikacija su

- `Qt Console Application` — obična Qt aplikacija,
- `Qt Widgets` i `Qt Quick Application` — Qt aplikacije u kojima se grafičko korisničko sučelje gradi pomoću modula Qt Widgets, odnosno Qt Quick,
- `Qt for Python` — Qt aplikacija namijenjena korištenju programskog jezika Python,
- `Plain C` i `Plain C++ Application` — obične C, odnosno C++ aplikacije, u kojima nije predviđeno korištenje Qt biblioteke.

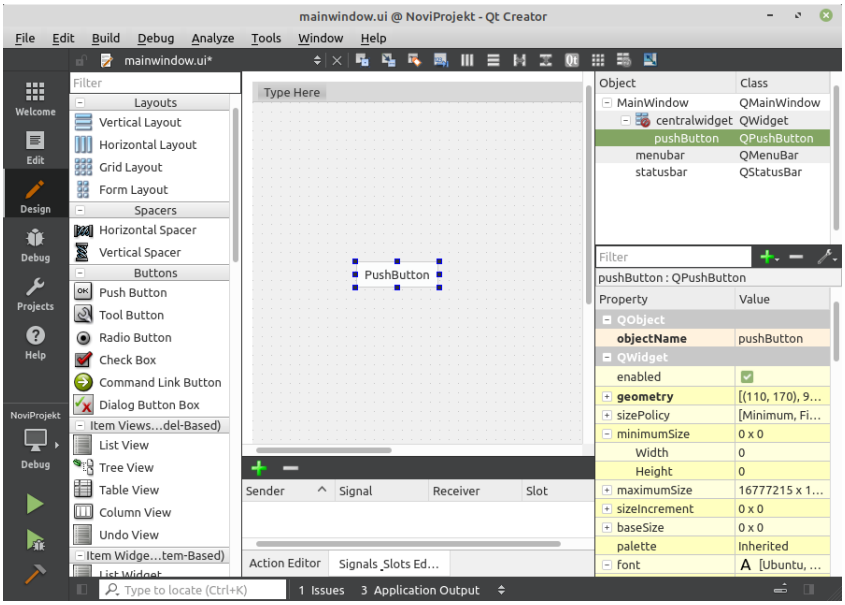
Nakon odabira vrste projekta, moguće je odabrati sustav za izgradnju, što je inicijalno `qmake`, no dostupni su i drugi, primjerice `CMake`. Projekt je također moguće dodati u Git, najpoznatiji sustav za verzioniranje koda (eng. *Version Control System*). Odabirom neke od Qt aplikacija, Qt Creator automatski generira projektnu datoteku, u kojoj su već uključeni svi potrebni moduli (primjerice, Qt Widgets) u ovisnosti o vrsti projekta.

Primjer otvorenog projekta u Qt Creatoru prikazan je na slici 1.5. U desnom dijelu prozora nalazi se uređivač teksta pomoću kojeg se uređuju datoteke izvornog koda, dok je s lijeve strane dostupan prikaz strukture projekta. Sve poruke nastale prilikom izgradnje programa ili ispisane prilikom izvršavanja programa, prikazuju se u konzoli na dnu ekrana. Konačno, uz donji dio lijevog ruba prozora, dostupne su brze opcije za izgradnju i pokretanje programa u običnom ili načinu rada za otklanjanje pogrešaka (eng. *debug mode*).



Slika 1.5: Otvoreni projekt u Qt Creatoru

Kako se biblioteka Qt5 često koristi za izradu aplikacija s grafičkim korisničkim sučeljem, tako Qt Creator uključuje i alat za izradu sučelja, Qt Designer, prikazan na slici 1.6. U centralnom dijelu prozora nalazi se prikaz izrađenog grafičkog korisničkog sučelja. S lijeve strane prozora vidljiv je popis svih dostupnih grafičkih elemenata, koji se u sučelje mogu smjestiti na željeno mjesto povlačenjem i ispuštanjem (eng. *drag-and-drop*). Popis svih grafičkih elemenata koji su trenutno dio sučelja, kao i sva njihova svojstva koje je moguće mijenjati direktno u Qt Designeru, prikazana su na desnoj strani prozora.



Slika 1.6: Qt Designer

Poglavlje 2

Grafičko korisničko sučelje uz modul Qt Widgets

Veliki dio biblioteke Qt5 su alati za izradu grafičkog korisničkog sučelja. Modul Qt Widgets nudi velik broj elemenata koji se često koriste u grafičkim korisničkim sučeljima, poput gumba, polja za unos teksta, padajućih izbornika i mnogih drugih. Također, sadrži klase za lakši razmještaj elemenata na ekran, nudeći razmještaje poput, primjerice, vertikalno ili horizontalno nanizanih elemenata te elemenata poslaganih u rešetku.

Najlakši način stvaranja grafičkog korisničkog sučelja je korištenjem alata Qt Designer. Tada se za stvoreno sučelje automatski generira XML (eng. *Extensible Markup Language*) datoteka koja ga opisuje i koju se u pravilu ne bi trebalo ručno mijenjati. Prema stvorenoj XML datoteci, prevoditelj za grafičko sučelje (eng. *User Interface Compiler*) generira C++ kod koji postavlja sve definirane elemente sučelja. Ovaj proces se pokreće automatski prilikom izgradnje projekta te prevoditelj za grafičko sučelje nije potrebno koristiti direktno.

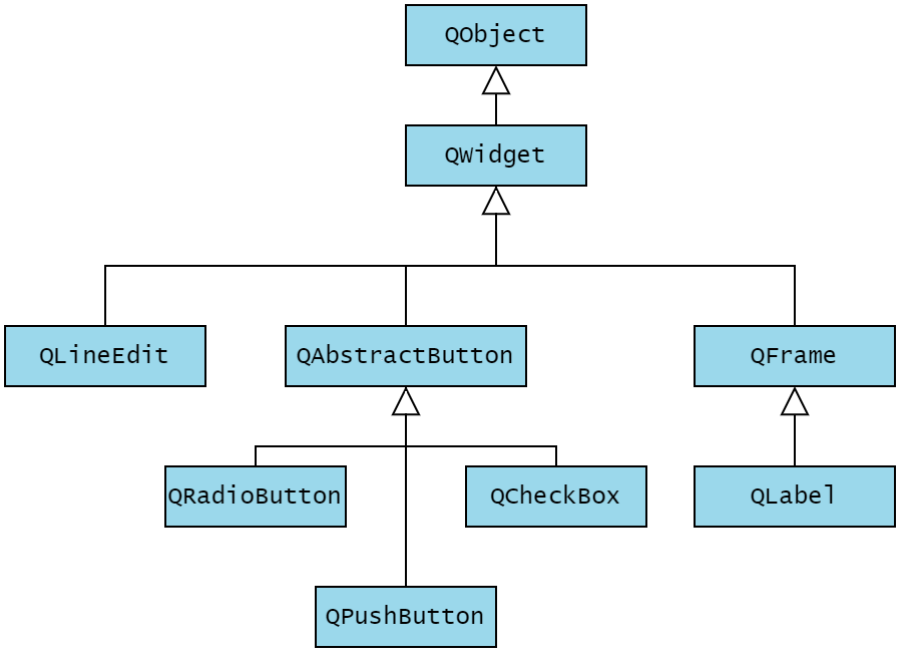
Prije detaljnijeg opisa elemenata pomoću kojih se gradi grafičko korisničko sučelje, valja napomenuti da je, osim modula Qt Widgets, za izradu grafičkih korisničkih modula moguće koristiti i modul Qt Quick. Tada se sučelja stvaraju na deklarativan način, koristeći jezik QML (eng. *Qt Modeling Language*), a moguće je uključiti i dijelove koda napisane u JavaScriptu. On se uobičajeno koristi za izradu grafičkih korisničkih sučelja aplikacija za mobilne platforme, dok se Qt Widgets prvenstveno koristi kod aplikacija namijenjenih stolnim i prijenosnih računalima.

2.1 Grafički elementi

Osnovni dio svakog grafičkog korisničkog sučelja su grafički elementi (eng. *widgets*) koji omogućuju interakciju korisnika s aplikacijom. Modul Qt Widgets nudi velik broj klasa

koje implementiraju raznovrsne grafičke elemente. Ovisno o grafičkom elementu, moguće je postaviti svojstva poput veličine, smještaja na ekranu, teksta koji se prikazuje i slično.

Sve klase koje predstavljaju grafičke elemente nasljeđuju klasu `QObject`, a time i ranije opisana svojstva poput mogućnosti korištenja sustava signala i utora te sudjelovanja u sustavu roditeljstva. Još jedna klasa koja je zajednička svim grafičkim elementima je `QWidget`. Ona, a time i svi elementi koji je nasljeđuju, ima svojstva poput širine i visine, vidljivosti ili koordinata pozicije lijevog gornjeg kuta u odnosu na roditeljski element, kao i metode za njihovo modificiranje. Mnogi grafički elementi ne nasljeđuju `QWidget` direktno, već postoje apstraktne klase zajedničke grupi grafičkih elemenata sa sličnim ponašanjem. Tako, primjerice, klasa `QAbstractButton` predstavlja klasu čiji objekti imaju mogućnosti poput postavljanja teksta, ikone i njene veličine, prečaca na tipkovnici koji aktivira objekt i drugih. Također, nudi predefinirane signale koji su emitirani prilikom dugog držanja, otpuštanja ili općenito pritiska na objekt. Konačno, objekti iz klasa koje nasljeđuju ovu su konkretni grafički elementi koje je moguće koristiti u grafičkim korisničkim sučeljima. To su, primjerice, `QPushButton` koji predstavlja običan gumb, `QCheckBox` koji predstavlja kućicu za označavanje ili `QRadioButton` koji je također moguće označiti, a najčešće se koristi za označavanje jedne od ponuđenih unutar grupe opcija. Ilustracija ove hijerarhije klasa grafičkih elemenata vidljiva je na slici 2.1.



Slika 2.1: Dio hijerarhije klasa grafičkih elemenata

Uz ranije navedene grafičke elemente, još neki često korišteni elementi dostupni u biblioteci Qt5 su

- `QLabel` — obična tekstualna oznaka,
- `QComboBox` — padajući izbornik za prikaz većeg broja opcija i odabir jedne,
- `QLineEdit` — polje za unos jedne linije teksta,
- `QTextEdit` — polje za unos više linija teksta. Također podržava formatiranje teksta koristeći oznake slične onima u HTML-u ili Markdownu,
- `QSpinBox` — polje za unos cjelobrojne vrijednosti,
- `QDoubleSpinBox` — polje za unos realnog broja dvostruke preciznosti.

Primjer korištenja nekih od navedenih grafičkih elemenata vidljiv je u isječku koda 2.2. Konstruktori objekata klase `QPushButton` te `QCheckBox` kao parametre primaju tekst koji će biti prikazan na gumbu te pokazivač na roditeljski objekt, čija je zadana (eng. *default*) vrijednost jednaka nul-pokazivaču. Tako objekt `gumb1` nema roditelja, budući da on nije predan konstruktoru, dok su njegova djeca objekti `gumb2` i `kucica`. Nakon konstrukcije objekta `gumb2`, metodom `QWidget::setGeometry(x, y, w, h)` zadaje se smještaj njegovog gornjeg lijevog kuta kao `(x, y)` te širina `w` i visina `h`.

```
1  int main(int argc, char *argv[])
2  {
3      QApplication app(argc, argv);
4
5      QPushButton gumb1 ("Gumb roditelj");
6      QPushButton gumb2 ("Gumb dijete", &gumb1);
7      gumb2.setGeometry(40, 40, 150, 50);
8      QCheckBox kucica ("Kućica za označavanje", &gumb1);
9
10     gumb1.show();
11     return app.exec();
12 }
```

Isječak koda 2.2: Grafičko sučelje s dva gumba i kućicom za označavanje

Rezultat pokretanja isječka koda 2.2 prikazan je slikom 2.3. Prema ranije opisanom sustavu roditeljstva, objekt `gumb1` nema roditelja te zbog toga postaje glavni prozor aplikacije. Kako su mu djeca drugi gumb i kućica za označavanje, oni su smješteni unutar njega. Pozicija kućice za označavanje nije specificirana pa se ona prikazuje u gornjem lijevom uglu, što je ishodište koordinatnog sustava, dok je drugi gumb prikazan na odabranoj poziciji.

Slika 2.3: Grafičko sučelje s dva gumba i kućicom za označavanje

Prozori

Jedan od istaknutih grafičkih elemenata unutar modula Qt Widgets je prozor, realiziran klasom `QMainWindow`. Prozore nije nužno koristiti, budući da element bez roditelja automatski postaje glavni prozor, no oni nude lako postavljanje tipičnih elemenata prozora, poput trake s izbornicima, alatne ili statusne trake.

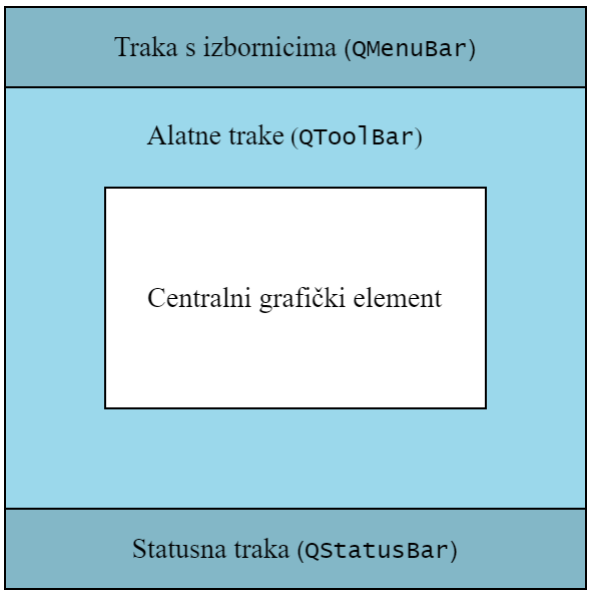
Metode potrebne za dodavanje i prilagođavanje trake s izbornicima uključene su unutar klase `QMenuBar`. Izbornici se prikazuju na vrhu ekrana kao gumbi čijom se aktivacijom prikazuje padajući izbornik s dostupnim akcijama. Akcije unutar izbornika su implementirane klasom `QAction`. Ona, između ostalog, omogućuje postavljanje imena i ikone akcije. Također, prilikom pritiska akcije, emitira predefinirani signal `triggered()`, kojeg je potrebno spojiti sa željenim utorom.

Klasa `QToolBar` predstavlja alatnu traku. U nju je moguće smjestiti proizvoljne grafičke elemente, no uobičajeno, kao i izbornici, sadrži elemente tipa `QAction`. U slučaju da je

postavljena ikona ciljane akcije, ona će se prikazati u alatnoj traci. Moguće je koristiti iste akcije unutar izbornika i alatne trake, čime se često korištene stavke izbornika mogu učiniti lako dostupnima. Prema svom zadanom ponašanju, alatna traka je plutajuća i moguće ju je po želji premještati i usidriti u različite rubove prozora, no `QToolBar` nudi metode čime se ovo ponašanje može onemogućiti. Tada alatna traka ostaje trajno usidrena u gornji rub prozora.

Konačno, klasa `QStatusBar` predstavlja statusnu traku koja se prikazuje na donjem rubu ekrana. Primarna namjena joj je prikaz privremenih statusnih poruka, primjerice obavještavanja korisnika o trenutnom statusu aplikacije ili napretka kod izvršavanja složenih operacija. Osim ovoga, unutar nje je moguće smjestiti proizvoljan grafički element, koji se također može prikazivati samo privremeno, ili pak biti trajno smješten i prikazan unutar statusne trake.

Vizualni prikaz razmještaja ovih elemenata unutar prozora vidljiv je na slici 2.4. Kako je rečeno, traka s izbornicima je smještena uz gornji rub prozora, a statusna traka uz donji. Alatna traka može biti usidrena uz bilo koji rub prozora. Također, korištenjem metode `QMainWindow::setCentralWidget()`, moguće je postaviti centralni grafički element koji se prikazuje u sredini prozora i zauzima sav preostali prostor.



Slika 2.4: Razmještaj elemenata unutar prozora

Kao i svi drugi grafički elementi, prozori se mogu koristiti direktno unutar funkcije `main()`. Međutim, zbog boljeg strukturiranja izvornog koda aplikacije, preporučeno je stvoriti vlastitu klasu koja nasljeđuje `QMainWindow` te implementirati sve funkcionalnosti aplikacije unutar nje. Stoga, Qt Creator nudi automatsko generiranje kostura jedne ovakve

klase prilikom stvaranja novog projekta. Također nudi stvaranje *forme*, koja se koristi prilikom stvaranja sučelja koristeći Qt Designer, a o kojoj će biti riječi kasnije.

Zaglavlje takve generirane klase vidljivo je u isječku koda 2.5. Generirana klasa je imena `MainWindow`, a nasljeđuje klasu `QMainWindow`. Također se može primijetiti i makro `Q_OBJECT`, koji omogućuje korištenje sustava signala i utora u klasi prozora. Također, klasa u privatnom dijelu sadrži pokazivač na objekt tipa `Ui::MainWindow`, koji se koristi za dohvat grafičkih elemenata ako je korisničko sučelje izrađeno koristeći Qt Designer.

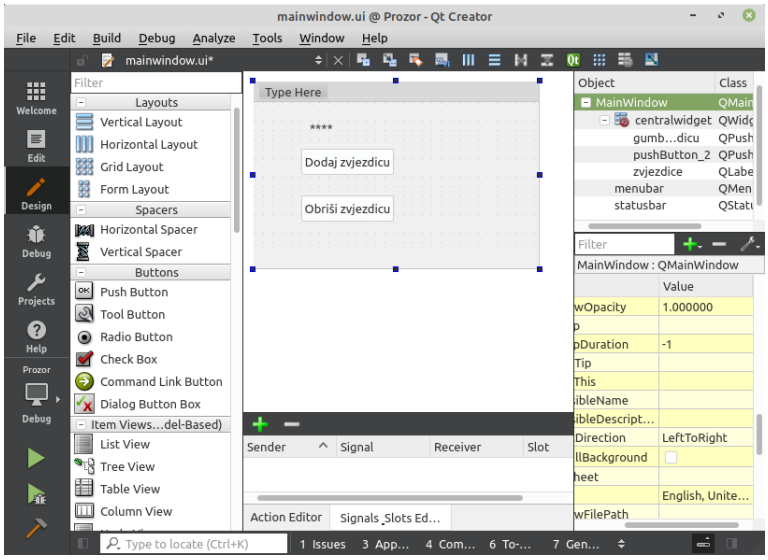
```
1  class MainWindow : public QMainWindow
2  {
3      Q_OBJECT
4
5  public:
6      MainWindow(QWidget *parent = nullptr);
7      ~MainWindow();
8
9  private:
10     Ui::MainWindow *ui;
11 };
```

Isječak koda 2.5: Zaglavlje klase koja predstavlja prozor

Postavljanje grafičkih elemenata kroz Qt Designer

Postavljanje velikog broja grafičkih elemenata kroz programski kod je poprilično zamorno, a Qt Designer znatno olakšava ovaj proces. Kako bi bilo moguće koristiti ga, potrebno je koristiti formu, prepoznatljivu po nastavku `.ui`. Kako je ranije napomenuto, prilikom stvaranja novog projekta, Qt Creator nudi stvaranje klase prozora, ali i pripadne datoteke forme. Njenim uređivanjem, automatski se otvara Qt Designer. Grafičko korisničko sučelje se kreira povlačenjem i ispuštanjem dostupnih grafičkih elemenata. Primjer grafičkog korisničkog sučelja izrađenog koristeći alat Qt Designer vidljivo je na slici 2.6. Sučelje sadrži jednu tekstualnu oznaku i dva gumba.

Otvaranjem datoteke forme u načinu za uređivanje, moguće je vidjeti sav XML kod koji definira stvoreno grafičko sučelje. Qt Creator inicijalno ne dozvoljava uređivanje koda ove datoteke, kako ne bi došlo do njene slučajne promjene, no ovu postavku je moguće promijeniti.



Slika 2.6: Primjer grafičkog korisničkog sučelja unutar Qt Designera

Dohvaćanje grafičkih elemenata postavljenih kroz Qt Designer moguće je kroz varijablu članicu `ui` klase prozora, vidljivu u isječku koda 2.5. Na taj je način moguće mijenjati svojstva grafičkih elemenata, ali i koristiti ih u sustavu signala i utora. Primjerice, prvi gumb je moguće dohvatiti kao `ui->gumbDodajZvjezdicu`, gdje je drugi dio ime objekta koji predstavlja gumb, definirano kroz Qt Designer.

2.2 Interakcija s korisnikom

Kako bi se grafički elementi mogli povezati s odgovarajućim akcijama, čime se konačno ostvaruje interakcija korisnika i aplikacije, koristi se sustav signala i utora. Za predefinirane grafičke elemente, uglavnom je dovoljno koristiti njihove predefinirane signale, no utore je vrlo često potrebno pisati samostalno.

Nastavno na primjer grafičkog korisničkog sučelja prikazanog na slici 2.6, u isječku koda 2.7 definira se i povezuje utor gumba koji dodaje zvjezdicu. Željeni utor je implementiran metodom `dodajZvjezdicu()`. Unutar nje se dohvaća trenutni tekst tekstualne oznake koristeći `ui->tekstZvjezdice` te se produžava još jednom zvjezdicom. Nakon toga se, u konstruktoru klase `MainWindow`, koja predstavlja prozor, obavlja povezivanje signala `QPushButton::pressed()` i definiranog utora. Time je ostvareno željeno ponašanje gumba te će se pritiskom na njega tekstualna oznaka nadopuniti dodatnom zvjezdicom.

Konačno, utore je moguće vrlo jednostavno definirati kroz Qt Designer. Desnim klikom na željeni element, otvara se izbornik u kojemu je potrebno odabrati ponuđenu opciju

```
void MainWindow::dodajZvjezdicu() {
    // Dohvaćanje trenutnog teksta tekstualne oznake.
    QString zvjezdice = ui->tekstZvjezdice->text();
    // Postavljanje novog teksta tekstualne oznake.
    ui->tekstZvjezdice->setText(zvjezdice + "*");
}

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    // Povezivanje signala pressed() i utora dodajZvjezdicu().
    QObject::connect(ui->gumbDodajZvjezdicu,
                     &QPushButton::pressed,
                     this,
                     &MainWindow::dodajZvjezdicu);
}
```

Isječak koda 2.7: Povezivanje pritiska gumba i akcije za dodavanje zvjezdice

`Go to slot...` . Tada se otvara prozor u kojem su ponuđeni signali svih klasa koje ciljani element nasljeđuje, primjerice `clicked()` , `pressed()` ili `released()` za gumb. Odabirom jedne od opcija, Qt Creator stvara utor čije je ime oblika `on_imeObjekta_imeSignala()` , kojeg je samo potrebno implementirati. U ovom slučaju, ručno spajanje signala i utora koristeći metodu `QObject::connect()` nije potrebno. Za to se brine ranije spomenuti *meta-object* sustav, koji prilikom prevođenja traži metode čije je ime navedenog oblika i proširuje programski kod spajanjem signala `imeSignala()` koji pripada objektu `imeObjekta` s definiranim utorom.

Kao primjer, isječkom koda 2.8 prikazana je implementacije jedne ovakve metode, koja se nastavlja na izrađeno grafičko korisničko sučelje prikazano na slici 2.6. Dakle, Qt će automatski povezati signal `pressed()` koji emitira `gumbObrisiZvjezdicu` s utorom de-

finiranim u prikazanom isječku koda. Time je i drugi gumb, za brisanje zvjezdice, povezan s odgovarajućom akcijom i grafičko korisničko sučelje je dovršeno.

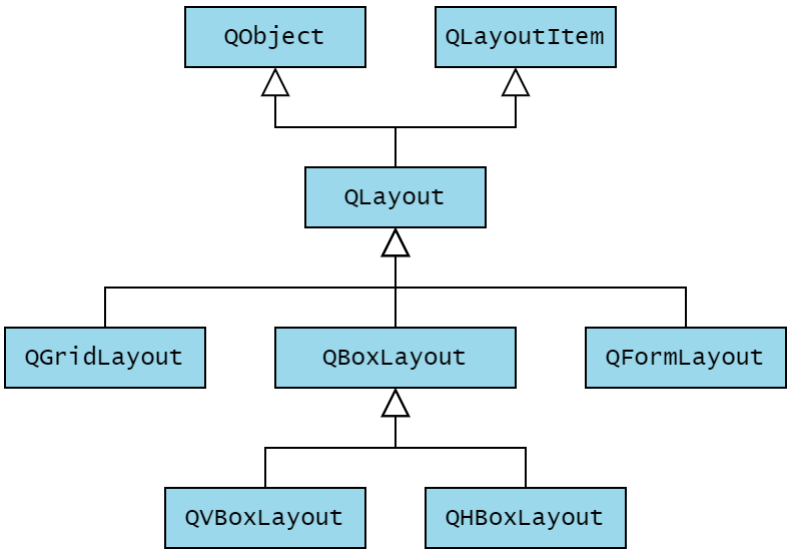
```
1  void MainWindow::on_gumbObrisiZvjezdicu_pressed()
2  {
3      // Dohvaćanje trenutnog teksta tekstualne oznake.
4      QString zvjezdice = ui->tekstZvjezdice->text();
5      // Postavljanje novog teksta tekstualne oznake.
6      if (zvjezdice > 0)
7          ui->tekstZvjezdice->setText(
8              zvjezdice.left(zvjezdice.length() - 1)
9              );
10 }
```

Isječak koda 2.8: Povezivanje pritiska gumba i odgovarajuće akcije

2.3 Razmještaj elemenata

Dosad predstavljenim metodama, grafički elementi mogli su se programski postaviti na željenu lokaciju unutar prozora koristeći metodu `QWidget::setGeometry()` ili ručnim smještajem elementa koristeći Qt Designer. Međutim, prilikom promjene veličine prozora, svi elementi smješteni na takav način, ostat će iste veličine i na istoj poziciji (u odnosu na gornji lijevi kut prozora). To uglavnom nije željeno ponašanje, već bi se veličina elemenata, kao i njihov razmještaj, trebali automatski prilagoditi trenutnoj veličini prozora.

U tu svrhu moguće je koristiti klase za razmještaj elemenata, koje sve nasljeđuju klasu `QLayout`. Postavljanje razmještaja moguće je za sve grafičke elemente, a obavlja se metodom `QWidget::setLayout()`. Na taj način će sva djeca odabranog grafičkog elementa biti smještena u skladu s odabranim razmještajem elemenata, a njihova veličina i pozicija će se mijenjati prilikom promjene veličine roditelja. Važno je napomenuti da klasa `QLayout` ne pripada skupu grafičkih elemenata, budući da ne nasljeđuje klasu `QWidget`. No, prilikom izrade kompleksnijih grafičkih sučelja, moguće je ugnježđivanje ovakvih elemenata za razmještaj. Dio hijerarhije klasa za razmještaj elemenata vidljiv je na slici 2.9.



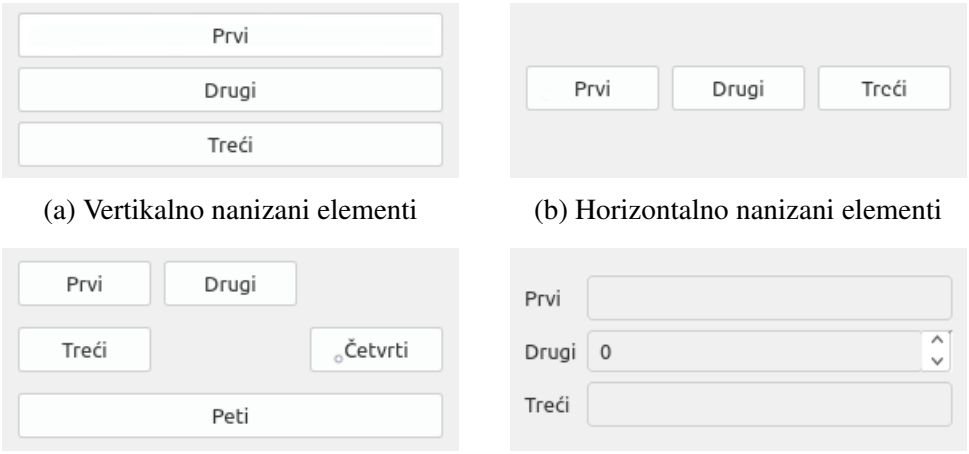
Slika 2.9: Dio hijerarhije klasa za razmještaj elemenata

Neke od često korištenih klasa za razmještaj elemenata su

- `QVBoxLayout` — elementi su nanizani vertikalno jedan ispod drugog,
- `QHBoxLayout` — elementi su nanizani horizontalno jedan pored drugog,
- `QGridLayout` — elementi su poredani u rešetku (eng. *grid*). Pojedini grafički element može zauzimati više redaka ili stupaca,
- `QFormLayout` — elementi su poredani u tipičan razmještaj forme, rešetku s dva stupca i više redaka. Obično se koristi u formi za unos podataka, gdje su u prvom stupcu tekstualne oznake, a u drugom polja za unos teksta ili drugi elementi za odabir opcija.

Primjer različitih razmještaja nekoliko grafičkih elemenata vidljiv je na slici 2.10.

Konačno, određivanje razmještaja elemenata moguće je vrlo jednostavno postići koristeći Qt Designer. Desnim klikom na element čiji razmještaj želimo postaviti, moguće je odabrati opciju `Lay out` i potom željeni razmještaj.



(a) Vertikalno nanizani elementi

(b) Horizontalno nanizani elementi

(c) Razmještaj u rešetki

(d) Razmještaj u obliku forme

Slika 2.10: Primjer različitih razmještaja grafičkih elemenata

Poglavlje 3

Sustav za prikaz grafičkog sadržaja i crtanje pomoću klase `QPainter`

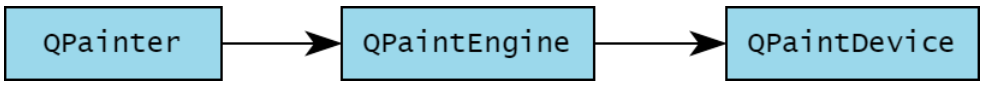
Biblioteka Qt5 nudi niz klasa korisnih za stvaranje i prikaz grafičkog sadržaja na ekranu. One su uglavnom dio modula Qt GUI, a najvažnija od njih je `QPainter` te klase povezane s njom. Interno, Qt koristi mehanizam koji uključuje `QPainter` za prikaz i iscrtavanje grafičkih elemenata korisničkog sučelja na ekranu. Koristeći jedan od ranije spomenutih modula više razine za stvaranje sučelja, primjerice Qt Widgets, ovo je za programera potpuno transparentno i ne mora se brinuti o tome. Međutim, Qt nudi i direktno korištenje ove klase. Tako se, primjerice, može upotrijebiti za crtanje slika, primitiva ili teksta na ekran, a podržava crtanje bazirano i na rasterskoj i vektorskoj grafici. Također se može koristiti za stvaranje vlastitih grafičkih elemenata.

U nastavku ovog poglavlja dan je pregled korištenja klase `QPainter` za crtanje na ekran, uključujući mogućnost crtanja raznovrsnih primitiva, slika i teksta, kao i kratak opis mehanizama korištenih prilikom crtanja na ekran. Prije toga, valjda još napomenuti da je crtanje na ekran moguće ostvariti koristeći modul Qt Quick i deklarativno programiranje. Također, putem posebnih klasa, podržano je korištenje nativnog OpenGL koda te iscrtavanje elemenata koristeći OpenGL.

3.1 Općenito o sustavu za iscrtavanje na ekran

Qt-ov sustav za iscrtavanje elemenata na ekran može se opisati kao cjevovodni sustav (eng. *pipeline*), čiji su sastavni dijelovi klase `QPainter`, `QPaintDevice` i `QPaintEngine`, ilustriran slikom 3.1.

U nastavku ovog potpoglavlja, slijede nešto detaljniji opisi klasa `QPaintDevice` te `QPaintEngine`.



Slika 3.1: Qt-ov cjevovodni sustav za iscrtavanje na ekran

Klasa `QPaintDevice`

Apstraktna klasa `QPaintDevice` predstavlja dvodimenzionalno platno na koje je moguće crtati koristeći `QPainter`. Kako bi se to ostvarilo, konstruktoru objekta klase `QPainter` se predaje referenca na željeni objekt tipa `QPaintDevice`. Alternativno, moguće ga je predati metodi `QPainter::begin()` koja označava početak crtanja. Stvaranje vlastitog platna za crtanje podrazumijeva nasljeđivanje klase `QPaintDevice` i preradu njene `paintEngine()` metode, o čemu je više riječi u narednom odjeljku. Međutim, u većini slučajeva je dovoljno korištenje već postojećih klasa koje nasljeđuju `QPaintDevice`, od kojih su neke

- `QWidget` — na ekran iscrtava svoj grafički prikaz koristeći `QPainter`. Svaki put kada je prikaz potrebno ponovno iscrtati (primjerice, kod promjene veličine) emitira se poseban događaj tipa `QPaintEvent`. U vezi s time, moguće je preraditi metodu `QWidget::paintEvent()`, u kojoj su definirani dodatni postupci koji će se provesti prilikom ponovnog iscrtavanja grafičkog elementa,
- `QImage` — klasa korištena za zapis i uređivanje slikovnog sadržaja. Nudi manipulaciju slikama na razini piksela te osigurava njihovu točnost i preciznost neovisno o platformi na kojoj se aplikacija koristi. Također, podržava operacije crtanja u zasebnoj programskoj niti. Ovo je posebno korisno kod zahtjevnijih operacija, kako glavna programska nit ne bi bila blokirana i kako bi mogla nesmetano primati i obrađivati korisnikov unos,
- `QPixmap` — također nudi funkcionalnosti za rad sa slikovnim sadržajem, no dizajnirana je i optimirana primarno za njihov prikaz. Manipulacije na razini piksela nisu podržane, već je za to potrebno koristiti pretvorbu u klasu `QImage`. Zbog toga se preporučuje njeno korištenje samo onda kada je potreban isključivo prikaz slikovnog sadržaja, dok je u slučaju potrebe vršenja dodatne obrade bolje koristiti `QImage`.

Opsežniji popis postojećih klasa koje nasljeđuju `QPaintDevice` moguće je pronaći na [4].

Klasa `QPaintEngine`

Klasu `QPaintEngine` moguće je opisati kao klasu čiji objekti definiraju način na koji `QPainter` treba crtati na zadani `QPaintDevice`. Sadrži niz virtualnih metoda kojima

je definiran način na koje se pojedine primitive (linije, pravokutnici, poligoni, elipse) trebaju nacrtati. Tako, primjerice, metoda `QPaintEngine::drawEllipse()` služi za definiranje načina na koji se iscrtava elipsa. Većina ovih virtualnih metoda ima zadanu implementaciju, no sve ih je moguće preraditi u slučaju stvaranja vlastitih podklasa klase `QPaintEngine`.

U slučaju korištenja nekih od postojećih klasa koje nasljeđuju `QPaintDevice` kao platna za crtanje, nije potreban direktan rad s klasom `QPaintEngine`, već se implicitno koriste predefinirani načini crtanja. Međutim, kako je ranije napomenuto, u slučaju definiranja vlastitih klasa koje predstavljaju platno za crtanje, potrebno je preraditi metodu `QPaintDevice::paintEngine()`. Ona vraća pokazivač na objekt tipa `QPaintEngine` koji se koristi, a kojeg je tada nužno izraditi samostalno. U tu svrhu je potrebno preraditi sve virtualne metode klase `QPaintEngine` koje nemaju zadanu implementaciju.

3.2 Crtanje primitiva, slika i teksta

Kako bi crtanje na ekran pomoću klase `QPainter` bilo što jednostavnije, dostupan je velik broj metoda za crtanje primitiva, slika i teksta. Većina njih je preopterećena (eng. *overloaded*), nudeći uporabu s različitim brojem i tipovima parametara, čime ove metode postaju fleksibilnije i jednostavnije za korištenje.

Metode za crtanje primitiva mogu se prepoznati po prefiksu `draw` iza kojeg slijedi ime primitive, a neki od dostupnih oblika za crtanje su

- **linije**, zadane s dvije točke,
- **pravokutnici**, zadani pozicijom lijevog gornjeg kuta te širinom i visinom. Također su dostupni i pravokutnici zakrivljenih uglova,
- **elipse**, zadane svojim središtem i duljinom horizontalne i vertikalne osi. Alternativno, moguće ih je nacrtati specificiranjem pravokutnika u koji se trebaju moći upisati,
- **poligoni**, zadani nizom točaka navedenih redoslijedom kojim trebaju biti spojene,
- **kružni lukovi**, zadani pravokutnikom u kojem bi pripadna kružnica trebala moći biti upisana te početnim i završnim kutom.

Cjelovit popis metoda za crtanje primitiva dostupan je na [5].

Primjer korištenja metoda za crtanje primitiva prikazan je isječkom koda 3.2. U ovom primjeru, crtanje se obavlja u metodi `paintEvent()` glavnog prozora. Konstruktoru objekta `QPainter` predaje se pokazivač na glavni prozor, na koji se primitive direktno iscrtavaju. Konačno, crta se linija, kvadrat i elipsa. Prikaz koji se dobije pokretanjem aplikacije koja sadrži ovaj isječak koda prikazan je na slici 3.4.

```
void MainWindow::paintEvent(QPaintEvent *)
{
    QPainter crtac(this);
    crtac.drawLine(5, 5, 50, 50);
    crtac.drawRect(50, 50, 100, 100);
    crtac.drawEllipse(QPoint(150, 150), 50, 70);
}
```

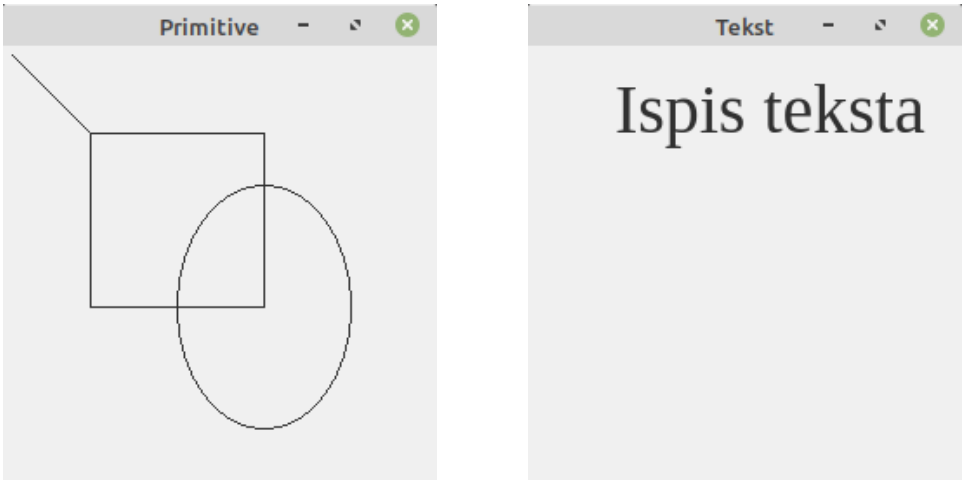
Isječak koda 3.2: Crtanje primitiva pomoću `QPainter`a

Osim metoda za crtanje primitiva, `QPainter` sadrži i veći broj preopterećenih metoda za ispis teksta. One također imaju različit broj i vrste parametara, a uglavnom se svode na specificiranje pozicije i teksta koji će biti ispisan. Odabir izgleda slova vrši se pozivom metode `QPainter::setFont()`, kojoj se predaje objekt tipa `QFont`. Njegov konstruktor pak podržava specifikaciju dizajna slova (eng. *font family*), veličine te debljine ili eventualne nakošenosti slova. Primjer ispisa teksta pomoću `QPaintera` prikazan je u isječku koda 3.3, a rezultat koji se dobije na slici 3.4.

```
void MainWindow::paintEvent(QPaintEvent *)
{
    QPainter crtac(this);
    crtac.setFont(QFont("Times New Roman", 30));
    crtac.drawText(50, 50, "Ispis teksta");
}
```

Isječak koda 3.3: Crtanje teksta pomoću `QPainter`a

Konačno, `QPainter` nudi nekoliko preopterećenih metoda za crtanje slika. Pomoću njih je moguće odabranu sliku nacrtati na definiranoj poziciji ili pak unutar definiranog pravokutnika, čime se njena veličina automatski mijenja ako je to potrebno. Sliku je prvo potrebno učitati iz memorije u objekt tipa `QImage`, koji se tada predaje jednoj od metoda za crtanje slika. Primjer iscrtavanja slike na ekran prikazan je isječkom koda 3.5. Putanja



(a) Primitive iscrtane isječkom koda 3.2 (b) Tekst iscrtan isječkom koda 3.3

Slika 3.4: Primjer crtanja primitiva i ispisa teksta pomoću `QPainter` a

slike zadaje se relativno u odnosu na datoteku u kojoj su smještene datoteke izgrađenog projekta. Nakon učitavanja, slika se iscrtava na ekran tako da joj je gornji lijevi ugao u ishodištu koordinatnog sustava.

```
1 void MainWindow::paintEvent(QPaintEvent *)
2 {
3     QImage slika("slika.png");
4     QPainter crtac(this);
5     crtac.drawImage(0, 0, slika);
6 }
```

Isječak koda 3.5: Crtanje slike pomoću `QPainter` a

3.3 Olovke i kistovi

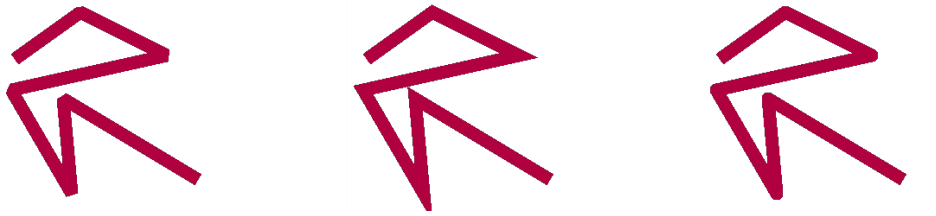
Kako bi se prilikom crtanja primitiva mogla mijenjati osnovna svojstva, poput debljine i boje linije, ili svojstava ispune, `QPainter` nudi metode `setPen()` i `setBrush()`. U

tu svrhu, potrebno je korištenje objekata klasa `QPen` te `QBrush`, detaljnije opisanih u nastavku.

Klasa QPen

Glavna svrha objekata `QPen` klase jest definiranje načina crtanja vanjskih linija primitiva, kao i boje ispisa teksta. Nudi nekoliko različitih konstruktora kojima je moguće jednostavno definirati željeni stil te niz metoda kojima je moguće odabrati vrijednosti dodatnih svojstava. Neka od često korištenih svojstava su

- **stil linije** — postoji nekoliko stilova linija definiranih unutar enumeracije `Qt`. Tako je moguće odabrati uobičajene `Qt::SolidLine`, iscrtkane `Qt::DashLine` ili istočkane `Qt::DotLine` linije. Također su dostupni stilovi u kojima se crtice i točke izmjenjuju. Za crtanje ispunjenih poligona bez rubova, postoji `Qt::NoPen`, čijim odabirom linija uopće neće biti nacrtana. Ovo svojstvo je moguće postaviti pozivom metode `QPen::setStyle()`,
- **stil krajeva linija** — definira način na koji su nacrtani krajevi linija, što je posebno korisno kod debljih linija. Moguće je odabrati jedan od stilova s ravnim, odrezanim krajevima `Qt::SquareCap` ili `Qt::FlatCap` te `Qt::RoundCap` u kojem su krajevi linija zaobljeni. Svojstvo se postavlja korištenjem metode `QPen::setCapStyle()`,
- **stil spajanja linija** — određuje način spajanja dviju linija. Način `Qt::BevelJoin` popunjuje preostali prostor između linija, a `Qt::MiterJoin` produljuje linije dok se njihovi krajevi ne sretnu. Konačno, `Qt::RoundJoin` popunjuje preostali prostor koristeći kružni isječak, u svrhu dojma zaobljenosti spojeva. Ovi stilovi spajanja ilustrirani su slikom 3.6.



(a) `Qt::BevelJoin` (b) `Qt::MiterJoin` (c) `Qt::RoundJoin`

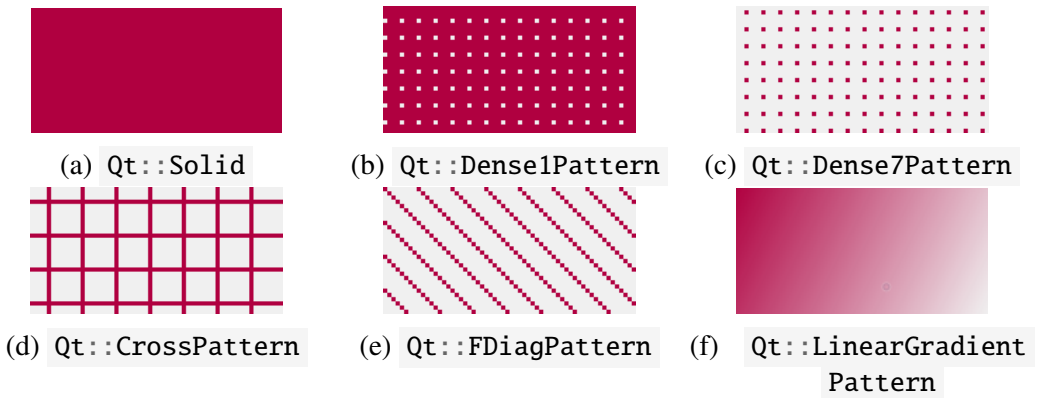
Slika 3.6: Različiti stilovi spajanja linija

Klasa QBrush

Osnovna namjena klase `QBrush` je definiranje načina ispuna primitiva. Koristeći ovu klasu, moguće je odabrati jedan od uzoraka ispuna koji su definirani unutar enumeracije `Qt::BrushStyle`. Pozivom metode `QBrush::setColor()` određuje se boja ispune. Neki od dostupnih uzoraka su

- **bez ispune** — kako bi se nacrtala primitiva bez ispune, moguće je koristiti vrijednost `Qt::NoBrush`,
- **jednobojna ispuna** — postiže se korištenjem kista `Qt::SolidBrush`,
- **točkasta ispuna** — dostupno je više vrsta ovakvih ispuna koje su određene vrijednostima od `Qt::Dense1Pattern` do `Qt::Dense7Pattern`,
- **šrafirana ispuna** — ispuna koja se sastoji od razmaknutih usporednih linija. Moguće je birati između horizontalnih `Qt::HorPattern` ili vertikalnih `Qt::VerPattern` te ukriženih horizontalnih i vertikalnih linija `Qt::CrossPattern`. Slične vrijednosti dostupne su i za dijagonalne linije,
- **gradijentna ispuna** — također postoji više vrsta ovakvih ispuna, a neke od njih su linearni gradijent, predstavljen vrijednošću `Qt::LinearGradientPattern` i radijalni gradijent, kojem odgovara vrijednost `Qt::RadialGradientPattern`. Postoje i pripadne klase koje sve nasljeđuju `QGradient`, a pomoću kojih se mogu finije odrediti svojstva poput smjera gradijenta ili boje,
- **ispuna teksturom** — moguće je definirati i ispunu slikom, koja se u svojoj originalnoj veličini ponavlja onoliko puta koliko je potrebno za ispunu cijele primitive.

Neki od ovih stilova ispune prikazani su na slici 3.7.



(a) `Qt::Solid` (b) `Qt::Dense1Pattern` (c) `Qt::Dense7Pattern`

(d) `Qt::CrossPattern` (e) `Qt::FDiagPattern` (f) `Qt::LinearGradient Pattern`

Slika 3.7: Različiti stilovi ispune

3.4 Transformacije koordinatnog sustava

Prilikom crtanja objekata na ekran, klasa `QPainter` koristi svoj logički koordinatni sustav kako bi odredila smještaj i veličinu crtanih objekata unutar platna za crtanje, to jest objekta tipa `QPaintDevice` (vidi [2, str. 117]). Ishodište koordinatnog sustava nalazi se u lijevom, gornjem uglu, a vrijednosti x, odnosno y osi rastu prema dolje, odnosno desno. Klasa `QPainter` podržava transformacije koordinatnog sustava, od kojih su neke

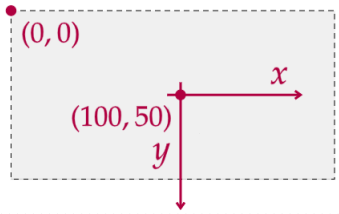
- **translacija**, određena odmakom u horizontalnom i vertikalnom smjeru,
- **rotacija** za određeni broj stupnjeva u smjeru kazaljke na satu,
- **skaliranje** odabranim faktorom u horizontalnom i vertikalnom smjeru.

Važno je napomenuti da ovo nisu transformacije objekta, već koordinatnog sustava. To je posebno vidljivo prilikom rotacije, budući da se oni elementi koji nisu u ishodištu koordinatnog sustava neće rotirati oko svoje osi, već oko ishodišta. Primjer korištenja koordinatnih transformacija prikazan je isječkom koda 3.8. Nakon konstrukcije `QPainter` objekta, trenutna postavka njegovog koordinatnog sustava sprema se naredbom `save()`. Potom se vrše translacija i rotacija koordinatnog sustava, čemu slijedi crtanje pravokutnika. Naredbom `restore()` vraćaju se ranije spremljene postavke, čime se ishodište koordinatnog sustava vraća u gornji lijevi ugao, nakon čega se crta još jedan pravokutnik.

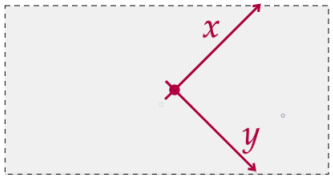
```
1  void MainWindow::paintEvent(QPaintEvent *) {
2      QPainter crtac(this);
3
4      crtac.save();
5      crtac.translate(100, 50);
6      crtac.rotate(-45);
7      crtac.drawRect(-10, -25, 20, 50);
8      crtac.restore();
9
10     crtac.drawRect(-10, -25, 20, 50);
11 }
```

Isječak koda 3.8: Primjer koordinatnih transformacija

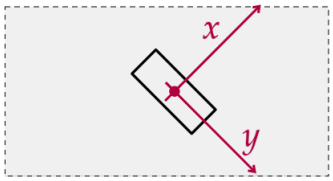
Rezultat ovih transformacija ilustriran je slikom 3.9. U ovom slučaju, prozor je širine 200, a visine 100 piksela. Na slici 3.9a demonstrirano je stanje koordinatnog sustava nakon naredbe `translate(100, 50)`, čime je ishodište pomaknuto u točku (100, 50). Potom se cijeli koordinatni sustav naredbom `rotate(-45)` rotira za 45° u negativnom, to jest smjeru suprotnom od kazaljke na satu, što je ilustrirano slikom 3.9b. Nakon toga, crta se pravokutnik odabranih dimenzija pozivom naredbe `drawRect()`, prikazan slikom 3.9c. Pravokutnik je nacrtan prema prenesenim parametrima, a relativno o trenutnim postavkama koordinatnog sustava. Potom se naredbom `restore()` poništavaju sve transformacije načinjene prije poziva metode `save()`, čime se koordinatni sustav rotira, a njegovo ishodište vraća se na inicijalnu poziciju. Konačno, crta se još jedan pravokutnik, čiji je dio vidljiv u lijevom gornjem uglu. Važno je primijetiti da su metodi `drawRect()` u oba slučaja predani isti parametri, što je vidljivo u linijama 6 i 9 isječka koda 3.8. Međutim, zbog transformacija koordinatnog sustava koje su izvedene između dva crtanja, nalaze se na različitim pozicijama te nisu isto orijentirani.



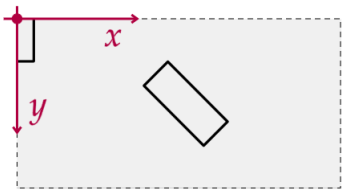
(a) Rezultat translacije koordinatnog sustava



(b) Rezultat rotacije koordinatnog sustava



(c) Rezultat crtanja prvog pravokutnika



(d) Rezultat vraćanja postavki koordinatnog sustava i crtanja drugog pravokutnika

Slika 3.9: Transformacije koordinatnog sustava prilikom izvođenja isječka koda 3.8

Poglavlje 4

Primjer jednostavne aplikacije za crtanje

U svrhu demonstracije opisanih tema, u sklopu ovog rada izrađena je jednostavna aplikacija za crtanje koristeći biblioteku Qt5. Zadatak aplikacije je korisniku omogućiti prostoručno crtanje na ekran pomoću miša, kao i mogućnost odabira boje i debljine linije za crtanje. Također je bilo potrebno pružiti mogućnost spremanja i učitavanja izrađenih slika u nekom od često korištenih slikovnih formata.

Kako je aplikacija namijenjena stolnim i prijenosnim računalima, za izradu grafičkog korisničkog sučelja korišten je modul Qt Widgets. Crtanje na ekran izvedeno je pomoću klase `QPainter`, koja je dio modula Qt GUI. Prilikom implementacije funkcionalnosti crtanja, korištene su mnoge od ranije navedenih metoda za crtanje primitiva, kao i metoda za odabir svojstava poput debljine ili boje kista kojim se crta.

4.1 Opis funkcionalnosti

Osim navedenog, izrađena aplikacija nudi još nekoliko alata za crtanje. Oni su

- **prostoručno crtanje** — pritiskom lijeve tipke miša i njegovim pomicanjem, moguće je crtati prostoručne linije,
- **gumica za brisanje** — vrlo slična prostoručnom crtanju, s tom razlikom što je njena boja trajno postavljena na bijelu,
- **crtanje elipsi i pravokutnika** — pritiskom na platno za crtanje, oblici se povlačenjem proširuju do željene veličine te crtaju na platno otpuštanjem tipke miša. Držeći tipku Shift pritisnutom, moguće je crtanje kvadrata i kružnica,

- **ispis teksta** — odabirom opcije za pisanje teksta na ekran, otvara se dijaloški okvir za unos teksta koji će biti prikazan. Potom se tekst pomacima miša može smjestiti na željenu poziciju, a pritiskom lijeve tipke miša, „lijepi" se na platno. Dostupne su i opcije za promjenu korištenog *fonta*,

- **dodavanje slika** — pokretanjem odgovarajuće akcije, prikazuje se dijaloški okvir za odabir slike iz pohrane računala. Nakon odabira, smještaj slike može se prilagoditi pomicanjem miša te se, kao i u slučaju teksta, njegova konačna pozicija potvrđuje pritiskom lijeve tipke miša. Osim odabira slike putem dijaloškog okvira, željenu sliku moguće je dovući iz odabranog direktorija na računalu i ispustiti unutar aplikacije, čime se pokreće opisana akcija za pozicioniranje slike. U ovom načinu crtanja, ponuđena je mogućnost promjene veličine odabrane slike.

Sve linije i oblici crtaju se u zadanoj boji, zadanom debljinom linije, dok se sav tekst ispisuje odabranom bojom. Moguće je odabrati jednu od predefiniranih opcija, ali i vlastitu boju ili debljinu linije u pikselima. Također, dostupne su akcije za poništavanje (eng. *undo*) i ponavljanje (eng. *redo*) određenog broja najkasnije nacrtanih linija ili oblika.

Pored crtanja, izrađene slike je moguće spremiti u JPG, BMP ili PNG formatu te ih kasnije učitati. Prilikom otvaranja ili učitavanja slike, prikazuje se dijaloški okvir za odabir mjesta pohrane na računalu. Pri pokretanju aplikacije, inicijalno je prikazano prazno platno širine i visine 500 piksela, no moguće je stvoriti novo platno proizvoljnih dimenzija.

4.2 Grafičko korisničko sučelje

Izrađeno grafičko korisničko sučelje prikazano je na slici 4.1. Kako bi bili dostupni tipični elementi prozora, cjelokupno sučelje smješteno je unutar prozora tipa `QMainWindow`. Glavni elementi koji se mogu uočiti su:

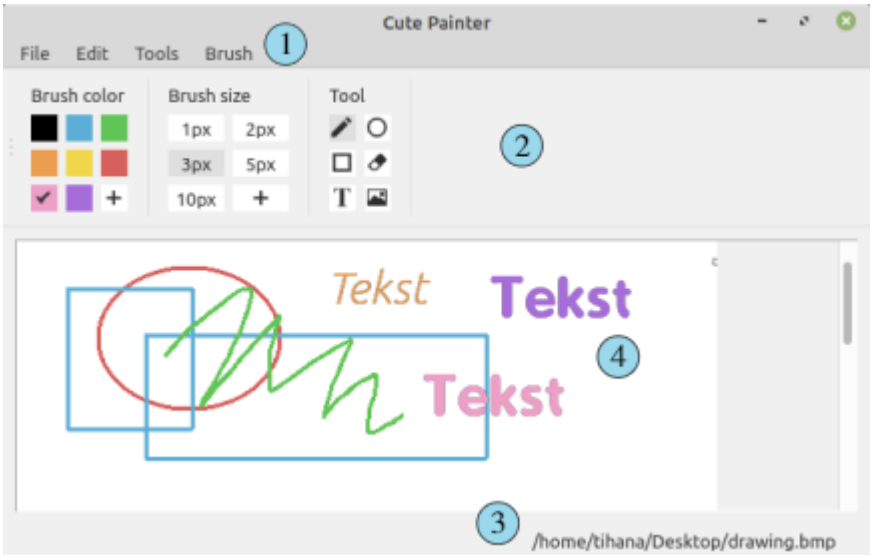
1. **Traka s izbornicima** — implementirana klasom `QMenuBar`, nudi četiri izbornika. Unutar izbornika File dostupne su akcije za rad s datotekama, poput spremanja i učitavanja slika. Ovdje je također prisutna opcija otvaranja čistog platna odabranih dimenzija. Izbornik Edit sadrži opcije za poništavanje Undo i ponavljanje Redo linija i oblika koji su najkasnije nacrtani. Isto tako, ovdje su raspoložive opcije za promjenu veličine crteža i platna za crtanje. Korištenjem izbornika Tools, moguće je odabrati jedan od ranije navedenih alata za crtanje. Pritiskom tipke Esc otkazuje se trenutno crtanje i odabire se alat za prostoručno crtanje. Konačno, izbornik Brush nudi nekoliko predefiniranih opcija za izbor boje, odnosno debljine kista, kao i opciju za proizvoljan odabir njihovih vrijednosti. U slučaju odabira proizvoljne vrijednosti boje kista, otvara se dijaloški okvir za odabir boje unutar kojeg je moguće

zadati vrijednosti boje u HSV, RGB ili heksadecimalnom formatu. Boje je također moguće odabrati iz spektra boja.

2. **Alatna traka** — plutajuća alatna traka implementirana klasom `QToolBar` može se povlačenjem i ispuštanjem usidriti na bilo koji od rubova ekrana, a inicijalno je prikazana na vrhu. Sadrži presliku onih opcija koje su dostupne u traci s izbornicima, podijeljenih u grupe ovisno o vrsti. Nudi brz pristup promjeni alata za crtanje, kao i odabiru debljine i boje kista. Gumbi za odabir boje unutar alatne trake realizirani su koristeći `QToolButton`, čija je boja pozadina postavljena tako da odražava boju kista koja se odabire njihovim pritiskom.

3. **Statusna traka** — unutar statusne trake implementirane klasom `QStatusBar` nalazi se ispis putanje trenutno otvorene datoteke.

4. **Platno za crtanje** — centralni dio aplikacije po kojem je moguće crtati. Kod većih dimenzija platna za crtanje, prikazuju se horizontalni i vertikalni klizači kojima je moguće pomicati platno za crtanje. Na taj način osigurana je vidljivost svih dijelova platna i onda kada je dimenzija većih od prozora.



Slika 4.1: Grafičko korisničko sučelje aplikacije za crtanje

4.3 Implementacija

U svrhu bolje organizacije koda, izvorni kod aplikacije podijeljen je u nekoliko većih klasa, koje su

- klasa `MainWindow` — obavlja radnje vezane uz grafičko korisničko sučelje te se brine za interakciju korisnika sa sučeljem putem mehanizma signala i utora,
- klasa `ImageBuffer` — pomoćna klasa koja implementira kružni spremnik slika tipa `QImage`, potreban za implementaciju opcija za ponavljanje i poništavanje najkasnije nacrtanog elementa,
- klasa `Canvas` — klasa koja predstavlja platno za crtanje i unutar koje se nalaze sve metode povezane s crtanjem na ekran.

Svaka od ovih klasa detaljnije je opisana u nastavku.

Klasa `MainWindow`

Klasa `MainWindow` predstavlja glavni prozor aplikacije i nasljeđuje `QMainWindow`. Sadrži metode za postavljanje osnovnih elemenata grafičkog korisničkog sučelja, poput trake s izbornicima i alatne trake. Unutar njih, akcije tipa `QAction` se povezuju s odgovarajućim gumbima tipa `QToolBar` prisutnima u alatnoj traci. Na ovaj način se povezuju svi gumbi za odabir alata, boje i debljine linije za crtanje. Također se označavaju gumbi onih opcija koje su inicijalno postavljene, što su alat za prostoručno crtanje crnim kistom debljine dva piksela. Pored toga, za sve akcije za rad s datotekama se metodom `QAction::setShortcut()` postavljaju prečaci na tipkovnici kojima ih je moguće aktivirati.

Jedan od važnijih dijelova ove klase je platno za crtanje, objekt tipa `Canvas`. Ova klasa je detaljnije opisana u nastavku, no za sada je bitno napomenuti da sadrži javne metode kojima joj `MainWindow` može dostaviti informacije o korisnikovom unosu, primjerice, prilikom promjene boje ili debljine kista. Zbog rada s datotekama, također može dohvatiti nacrtanu sliku ili proslijediti platnu za crtanje učitanu sliku.

Osim navedenog, klasa `MainWindow` sadrži niz utora kojima definira ponašanje pritiska gumba za odabir boje, debljine linije ili alata. Ovi utori se ne povezuju sa signalima samih gumba, već njihovih pripadnih akcija, koji se emitiraju prilikom pritiska na gumb. Kako opcije unutar izbornika na vrhu ekrana i alatne trake dijele iste akcije, time je osiguran jednak odgovor aplikacije prilikom odabira željene opcije na bilo koji od dva načina, bez potrebe za ponavljanjem programskog koda. Imena svih utora su oblika `on_imeObjekta_imeSignala()`, čime nije potrebno dodatno pozivanje metode za povezivanje signala i utora `QObject::connect()`, kako je opisano u potpoglavlju 2.2. Kao

primjer, prikazan je jedan od utora za promjenu boje kista, vidljiv u isječku koda 4.2. Unutar ovog utora poziva se odgovarajuća javna metoda klase `Canvas`. Boja `colorBlue` tipa `QColor` definirana je unutar klase `MainWindow`, uključujući još nekoliko često korištenih boja.

```
1 void MainWindow::setCanvasBrushColor(const QColor &color) {
2     canvas.setBrushColor(color);
3 }
4 void MainWindow::on_actionBlue_triggered() {
5     setCanvasBrushColor(colorBlue);
6 }
```

Isječak koda 4.2: Utor za odabir boje kista

Još jedan primjer utora, onaj za učitavanje nove slike, vidljiv je u isječku koda 4.4. Ako postoje nespremljene promjene, metodom `showUnsavedChangesDialogAndSave()` prikazuje se jednostavni dijalog koji o tome upozorava korisnika, i daje mu mogućnost spremanja trenutnog crteža prije otvaranje nove slike. Odluči li korisnik otkazati akciju, daljnje radnje se obustavljaju i može nastaviti s crtanjem. U suprotnom, poziva se metoda `showLoadImageDialog()`, prikazana isječkom koda 4.3, koja pokazuje dijaloški okvir za odabir slike iz pohrane računala koristeći metodu `QFileDialog::getOpenFileName()` te vraća odabranu putanju. Konačno, metoda `openAndLoadImage()` učitava odabranu sliku koristeći putanju koja se predaje kao argument i postavlja je na platno za crtanje.

```
1 QString MainWindow::showLoadImageDialog() {
2     QString fileName = QFileDialog::getOpenFileName(
3                 this, "Open file", ".",
4                 "Image files (*.jpg *.jpeg *.bmp *.png)");
5     return fileName;
6 }
```

Isječak koda 4.3: Implementacija metode `showLoadImageDialog()`

```
void MainWindow::on_actionOpen_triggered() {
    bool actionCancelled = false;
    if (canvas.getUnsavedChanges())
        actionCancelled = showUnsavedChangesDialogAndSave();

    if (!actionCancelled) {
        QString fileName = showLoadImageDialog();
        if (!fileName.isEmpty())
            openAndLoadImage(fileName);
    }
}
```

Isječak koda 4.4: Utor za učitavanje nove slike

Klasa `ImageBuffer`

Ova klasa predstavlja kružni spremnik slika `QImage`. Njeno čitavo zaglavlje prikazano je u isječku koda 4.5. Osim pokazivača na spremnik `buffer`, sadrži svojstva poput ukupnog kapaciteta i preostalog prostora. Također, sadrži indekse najstarijeg, najnovijeg i trenutnog elementa.

Metodom `getCurrent()` moguće je dohvatiti trenutnu sliku, dok se metodama imena `currentNext()` i `currentPrevious()` pomiče indeks trenutnog elementa, u ovisnosti o tome koliko je prostora dostupno. Ovo ponašanje bilo je potrebno u svrhu implementacije *undo* i *redo* funkcionalnosti. Odabirom jedne od navedenih opcija, klasa `Canvas`, koja sadržava ovakav kružni spremnik, poziva odgovarajuću metodu čime se pomiče indeks trenutnog elementa. Prilikom svakog poteza napravljenog tijekom crtanja, poziva se metoda `newImage()` klase spremnika. Unutar nje se najnovija slika kopira na sljedeće mjesto, dok se indeks trenutnog elementa pomiče za jedan. Na taj je način osigurano spremanje slike nastale netom nakon poteza crtanja, koja se kasnije lako može dohvatiti kad korisnik poželi poništiti svoj potez.

Preopterećene metode `newBuffer()` koriste se onda kada postoji potreba za novim spremnikom. Parametri koje prima određuju prvu sliku spremnika, odnosno njene dimenzije. Sljedeća ponuđena metoda, `resizeImage()`, stvara novu sliku unutar spremnika tako da kopira trenutnu i promijeni joj veličinu, dok `resizeCanvas()` kopira trenutnu sliku i reže je ili nadopunjuje praznim prostorom ovisno o tome je li nova dimenzija platna veća

ili manja od trenutne slike.

```
class ImageBuffer
{
    QImage *buffer;
    int capacity, spaceLeft;
    // Najstariji, najnoviji i trenutni element.
    int oldest, current, newest;
public:
    ImageBuffer(int width, int height, int capacity = 20);
    ~ImageBuffer();

    QImage &getCurrent(); // Dohvaćanje trenutnog elementa.
    void newImage(); // Stvaranje nove u redu slika.
    void newImage(const QImage &image);
    void currentNext(); // Pomak trenutne slike na sljedeću.
    void currentPrevious(); // Pomak trenutne slike na prethodnu.
    // Priprema za novo platno zadanih dimenzija.
    void newBuffer(int width, int height);
    void newBuffer(const QImage &image);
    // Promjena veličine slike ili platna.
    void resizeImage(int width, int height);
    void resizeCanvas(int width, int height);
};
```

Isječak koda 4.5: Zaglavlje klase `ImageBuffer`

Klasa Canvas

Sve metode za crtanje sadržane su unutar klase `Canvas`. Ona nasljeđuje klasu `QWidget`, koja od sustava za upravljanje prozorom prima događaje vezane uz korištenje miša i tipkov-

nice. Klasa `QWidget` nudi niz metoda koje `Canvas` prerađuje, a koje definiraju ponašanje aplikacije prilikom ovih događaja, primjerice pritiska ili otpuštanja tipke miša.

Osim metoda za crtanje, klasa `Canvas` sadrži enumeraciju `DrawingMode` koja označava jedan od osam načina crtanja — prostoručno crtanje, gumica za brisanje, crtanje pravokutnika, kvadrata, elipsi ili kružnica, ispis teksta i dodavanje slika. Sadrži i nekoliko olovki tipa `QPen` koje definiraju način crtanja vanjskih linija. Olovka `eraserPen` koristi se kad je odabrani alat gumica za brisanje, a boja joj je trajno postavljena na bijelu. Oblici, linije i tekst crtaju se olovkom `drawingPen`, čija se boja i debljina mijenjaju prema korisnikovom odabiru. Kod obje olovke, odabrani stil krajeva linije je `Qt::RoundCap`, a spajanja `Qt::RoundJoin`, čime se postiže bolji izgled svih nacrtanih linija i oblika.

Prva od prerađenih metoda je `QWidget::mousePressEvent()`, prikazana isječkom programskog koda 4.6. Ova metoda poziva se prilikom svakog pritiska miša. Prikazane zastavice `isMousePressed` i `unsavedChanges` sadržane su u privatnom dijelu klase `Canvas`, a označavaju je li miš trenutno pritisnut, odnosno postoje li nespremljene promjene. Kako je ranije opisano, u ovom trenutku potrebno je stvoriti novu sliku unutar kružnog spremnika `imageBuffer` kako bi bila omogućena funkcionalnost *undo*. Konačno, vrijednost varijable članice `mousePressedAt`, koja predstavlja koordinate početka crtanja, postavlja se na poziciju pritiska miša, dohvatljivu unutar parametra tipa `QMouseEvent`.

```
1 void Canvas::mousePressEvent(QMouseEvent *e) {
2     if (e->button() == Qt::LeftButton) {
3         isMousePressed = true;
4         unsavedChanges = true;
5         imageBuffer.newImage(); // Za undo i redo.
6         mousePressedAt = e->pos();
7     }
8 }
```

Isječak koda 4.6: Prerađena metoda `QWidget::mousePressEvent()`

Sljedeća prerađena metoda je `QWidget::mouseMoveEvent()`, koja se poziva prilikom svakog pomaka miša, a prikazana je isječkom koda 4.7. Kako bi korisnik u svakom trenutku mogao imati bolji dojam trenutne veličine kista, na poziciju miša iscrtava se kružnica promjera jednakog odabranoj debljini kista. Ovo se obavlja metodom `drawCursorSize()`, neovisno o tome je li miš trenutno pritisnut ili ne. Također, u slučaju da je trenutni način

crtanja pisanje teksta, odnosno dodavanje slike, poziva se metoda `drawText()`, odnosno metoda `drawImage()`, koje primaju trenutnu poziciju miša. Ako ništa od toga nije slučaj i lijeva tipka miša je pritisnuta, pokreću se metode za crtanje linija i oblika. Ako je trenutno odabrani alat jedan od onih za crtanje oblika, poziva se metoda `drawShape()`, a u suprotnom metoda `drawLine()`. Obje metode primaju poziciju miša unutar prozora.

```
1  void Canvas::mouseMoveEvent(QMouseEvent *e) {
2      drawCursorSize(e->pos());
3
4      if (drawingMode == DrawingMode::Text)
5          drawText(e->pos());
6      else if (drawingMode == DrawingMode::Image)
7          drawImage(e->pos());
8      else if (e->buttons() == Qt::LeftButton && isMousePressed) {
9          if (isDrawingModeShape())
10             drawShape(e->pos());
11         else
12             drawLine(e->pos());
13     }
14 }
```

Isječak koda 4.7: Prerađena metoda `QWidget::mouseMoveEvent()`

Metoda `drawLine()` za prostoručno crtanje prikazana je isječkom koda 4.8. Konstruktoru objekta za crtanje tipa `QPainter` predaje se trenutna slika u kružnom spremniku kao platno za crtanje. Ovisno o trenutnom načinu crtanja, bira se jedna od ranije spomenutih olovki. Konačno, metodom `QPainter::drawLine()` crta se linija koja spaja prethodnu i trenutnu poziciju miša, koja se potom ažurira.

Crtanje oblika implementirano je metodom `drawShape()`, prikazanom isječkom programskog koda 4.9. Konstruktoru objekta za crtanje tipa `QPainter` ovaj put se predaje privremeno platno za crtanje `temporaryImg` tipa `QImage`. Dok god je tipka miša pritisnuta, oblik se iznova crta na privremeno platno, prije čega se ono čisti, a dodaje se na pravo platno tek onda kada crtanje završi. Oblik nije moguće crtati direktno na trenutnu sliku unutar kružnog spremnika, kako je to izvedeno kod crtanja linije, budući da bi to rezultiralo neželjenim ponavljanjem istog oblika u različitim veličinama. Naime, kako se

```
void Canvas::drawLine(QPoint position) {
    QPainter painter(&imageBuffer.getCurrent());
    if (drawingMode == Canvas::DrawingMode::Eraser)
        painter.setPen(eraserPen);
    else if (drawingMode == Canvas::DrawingMode::Brush)
        painter.setPen(drawingPen);

    painter.drawLine(mousePressedAt, position);
    mousePressedAt = position;
    this->update(); // Ponovno iscrtaj ekran.
}
```

Isječak koda 4.8: Metoda `drawLine()` za prostoručno crtanje

`drawShape()` poziva u metodi `mouseMoveEvent()`, tako bi na platno bila nacrtana nova kopija odabranog oblika drugačije veličine prilikom svakog pomaka miša. Nadalje se, ovisno o trenutno odabranom načinu crtanja, računa širina i visina novog oblika i pozivaju se metode `QPainter::drawRect()` za crtanje pravokutnika ili `QPainter::drawEllipse()` za crtanje elipse. Parametri potrebni za iscrtavanje likova određuju se na osnovu trenutne i pozicije na kojoj je prvotno pritisnuta tipka miša.

```
void Canvas::drawShape(QPoint position) {
    temporaryImg.fill(qRgba(0, 0, 0, 0));
    QPainter painter(&temporaryImg);
    painter.setPen(drawingPen);

    int x = mousePressedAt.x(), y = mousePressedAt.y();
    int width = position.x() - x, height = position.y() - y;

    if (drawingMode == DrawingMode::Circle
            || drawingMode == DrawingMode::Square)
        height = width;
```

```
12      if (isDrawingModeCircularShape())
13          painter.drawEllipse(x, y, width, height);
14      else if (isDrawingModeRectangularShape())
15          painter.drawRect(x, y, width, height);
16
17      this->update(); // Ponovno iscrtaj ekran.
18  }
```

Isječak koda 4.9: Metoda `drawShape()` za crtanje oblika

Metode za crtanje teksta i slika funkcioniraju na veoma sličan način kao ona za crtanje oblika, utoliko što se crtanje također prvo obavlja na privremenom platnu. Metoda za pisanje teksta `drawText()` prikazana je isječkom koda 4.9. Varijabla `font` je članica čiju vrijednost klasa `MainWindow` mijenja ovisno o korisnikovom odabiru. Slično, u slučaju načina crtanja slike, koristi se varijabla članica `imageToShow` unutar koje je pohranjena slika koju je potrebno dodati na platno. Njeno korištenje i crtanje obavljeno je unutar metode `drawImage()`, prikazane isječkom koda 4.10.

```
1   void Canvas::drawText(QPoint position) {
2       temporaryImg.fill(qRgba(0, 0, 0, 0));
3       QPainter painter(&temporaryImg);
4       painter.setPen(drawingPen);
5       painter.setFont(font);
6       painter.drawText(
7                   QRect(position.x(), position.y(),
8                         textToWriteWidth, temporaryImg.height()),
9                   Qt::TextWrapAnywhere, textToWrite);
10
11      this->update(); // Ponovno iscrtaj ekran.
12  }
```

Isječak koda 4.9: Metoda `drawText()` za ispis teksta

```
void Canvas::drawImage(QPoint position) {
    temporaryImg.fill(qRgba(0, 0, 0, 0));
    QPainter painter(&temporaryImg);
    painter.drawImage(position.x(), position.y(), imageToShow);

    this->update(); // Ponovno iscrtaj ekran.
}
```

Isječak koda 4.10: Metoda `drawImage()` za crtanje slike

Konačno, završetak crtanja označen je otpuštanjem tipke miše i implementiran unutar metode `mouseReleaseEvent()`, prikazane isječkom koda 4.11. Unutar nje, otpuštanje miša signalizira se promjenom vrijednosti zastavice `isMousePressed`. Kao što je ranije napomenuto, ako se trenutno crta neki od oblika, tekst ili slika, privremeno platno je potrebno nalijepiti na platno unutar kružnog spremnika slika. Potom se privremeno platno čisti kako bi bilo spremno za iduću upotrebu.

```
void Canvas::mouseReleaseEvent(QMouseEvent *e) {
    if (e->button() == Qt::LeftButton && isMousePressed) {
        isMousePressed = false;
        // Zalijepi privremenu sliku na pravu.
        if (isDrawingModeShape() || isDrawingModeTextOrImage()) {
            QPainter painter(&imageBuffer.getCurrent());
            painter.drawImage(temporaryImg.rect(), temporaryImg);
            // Očisti privremenu sliku.
            temporaryImg.fill(qRgba(0, 0, 0, 0));
        }
    }
}
```

Isječak koda 4.11: Prerađena metoda `QWidget::mouseReleaseEvent()`

Do sada prikazanim, linije su se iscrtavale na platno unutar kružnog spremnika, a oblici, tekst i slike na privremeno platno, što se sve događa izvan ekrana. Kako bi se izrađena slika konačno i prikazala na ekranu, bilo ju je potrebno iscrtati na sam objekt tipa `Canvas`, u svrhu čega je prerađena metoda `QWidget::paintEvent()`, prikazana isječkom koda 4.12. Kako je ranije rečeno, ona se poziva onda kada je objekt tipa `QWidget` potrebno ponovno iscrtati na ekran. Onda kada postoji potreba njenog ručnog pozivanja, moguće je iskoristiti metodu `QWidget::update()`. U ovom slučaju, to je bilo, primjerice, unutar metode `drawLine()`, prikazane isječkom koda 4.8 ili `drawShape()`, vidljive u isječku koda 4.9. Važno je napomenuti da je direktno crtanje na objekt tipa `QWidget` moguće izvesti isključivo u njegovoj `paintEvent()` metodi (ili nekoj drugoj pozvanoj unutar nje), stoga se i crtanje na `Canvas` objekt izvodi u istoj. Konstruktoru klase `QPainter` predaje se pokazivač `this` te se potom trenutna slika unutar kružnog spremnika crta na ekran. Kod načina crtanja oblika, teksta ili slike, privremeno platno `temporaryImg` sadrži samo trenutno nacrtani element te se potom i ono crta na ekran, preko trenutne slike kružnog spremnika. Konačno, na ekran se iscrtava pomoćno platno `cursorImg`, koje sadrži kružnicu promjera trenutne debljine kista. Time su svi elementi iscrtani na `Canvas` i postaju vidljivi na ekranu.

```
1  void Canvas::paintEvent(QPaintEvent *) {
2      QPainter painter(this);
3      painter.drawImage(imageBuffer.getCurrent().rect(),
       ↪  imageBuffer.getCurrent());
4
5      if (isDrawingModeShape() || isDrawingModeTextOrImage())
6          painter.drawImage(temporaryImg.rect(), temporaryImg);
7
8      if (!isDrawingModeTextOrImage())
9          painter.drawImage(cursorImg.rect(), cursorImg);
10 }
```

Isječak koda 4.12: Prerađena metoda `QWidget::paintEvent()`

Zaključak

Velikim brojem modula koji nudi, biblioteka Qt5 uvelike olakšava razvoj raznovrsnih aplikacija. Među njima, posebno se ističe modul Qt Widgets i brojni grafički elementi uključeni u njega, koji omogućavaju izradu jednostavnijih, ali i složenih grafičkih korisničkih sučelja. Mehanizmom signala i utora moguće je na jednostavan i za programera veoma intuitivan način postići komunikaciju među objektima, uključujući i grafičke elemente. Time se vrlo efikasno ispunjava glavna svrha korisničkih sučelja, ostvarenje interakcije korisnika s aplikacijom. Razvojna okolina Qt Creator, posebno namijenjena razvoju Qt aplikacija, dodatno pojednostavljuje i ubrzava proces razvoja, dok uključeni alat Qt Designer omogućuje brzu i jednostavnu izradu grafičkih korisničkih sučelja.

Pored toga, biblioteka Qt5 nudi moćan i učinkovit sustav za iscrtavanje elemenata na ekran, podržan modulom Qt GUI. U ovome se posebno važnom pokazuje klasa `QPainter`, koju Qt interno koristi za iscrtavanje složenih grafičkih elemenata na ekran. Osim ovoga, moguće ju je koristiti i direktno za iscrtavanje primitiva, slika ili teksta, ili pak za stvaranje vlastitih grafičkih elemenata. Kako bi crtanje elemenata na ekran bilo što jednostavnije i fleksibilnije, klasa `QPainter` nudi niz preopterećenih metoda za crtanje velikog broja primitiva, linija, teksta i slika. Također, pruža nekoliko klasa kojima je moguće dodatno prilagoditi svojstva poput stila, boje i debljine linije kojom se crta te načina ispuna primitiva. Time je moguće iscrtati vrlo složene i vizualno atraktivne elemente. U kombinaciji s drugim dijelovima Qt5 biblioteke, poput objekata klase `QWidget`, crtanje na ekran moguće je povezati s korištenjem ulaznih uređaja poput miša i tipkovnice. Ovo je demonstrirano jednostavnom aplikacijom za crtanje izrađenom u sklopu ovog rada, u kojem se crtanje na ekran pomoću klase `QPainter` povezuje s pomacima miša, u svrhu crtanja prostoručnih linija i drugih oblika.

Bibliografija

[1] *Qt Wiki: Language Bindings*, https://wiki.qt.io/Language_Bindings, posjećena u kolovozu 2021.

[2] L. Z. Eng, *Qt5 C++ GUI Programming Cookbook: Practical recipes for building cross-platform GUI applications, widgets, and animations with Qt 5*, Packt Publishing Ltd, United Kingdom, 2019.

[3] The Qt Company Ltd, *Qt Documentation: Creating Project Files*, https://doc.qt.io/qt-5/qmake-project-files.html, posjećena u kolovozu 2021.

[4] _____, *Qt Documentation: Paint Devices and Backends*, https://doc.qt.io/qt-5/paintsystem-devices.html, posjećena u kolovozu 2021.

[5] _____, *Qt Documentation: QPainter class*, https://doc.qt.io/qt-5/qpainter.html, posjećena u kolovozu 2021.

[6] _____, *Qt Documentation: Signals & Slots*, https://doc.qt.io/qt-5/signalsandslots.html, posjećena u kolovozu 2021.

[7] _____, *Qt Documentation: Why Does Qt Use Moc for Signals and Slots?*, https://doc.qt.io/qt-5/why-moc.html, posjećena u kolovozu 2021.

Sažetak

U ovom diplomskom radu opisana je biblioteka Qt5 za razvoj aplikacija s grafičkim korisničkim sučeljem. Predstavljena je njena struktura, kao i osnovni mehanizmi koje uvodi. Opisan je proces izrade grafičkog korisničkog sučelja, glavni elementi koji u njemu sudjeluju te način ostvarenja interakcije s korisnikom. Prezentiran je Qt-ov sustav za prikaz grafičkog sadržaja, kao i metode kojima se može ostvariti crtanje na ekran. Opisane su funkcionalnosti i implementacija jednostavne aplikacije za crtanje izrađene u sklopu rada.

Summary

This thesis describes the Qt5 framework, used in developing cross-platform applications that incorporate a graphical user interface. The paper briefly outlines its basic structure, along with some of the central mechanisms it introduces. It also provides an overview of graphical elements frequently used in graphical user interfaces, as well as the process of creating one using the Qt5 framework. Moreover, it portrays Qt's graphic content display system, including techniques that enable performing on-screen drawing operations. To demonstrate the presented methods, a basic painting application was created, whose functionalities and implementation are described.

Životopis

Rođena sam 7. lipnja 1996. godine u Puli, gdje započinjem obrazovanje u Osnovnoj školi Kaštanjer. Nakon toga upisujem Tehničku školu u Puli, koju završavam 2015. godine. Iste godine upisala sam preddiplomski sveučilišni studij Matematika na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta Sveučilišta u Zagrebu. Završetkom preddiplomskog studija 2019. godine, obrazovanje nastavljam na diplomskom sveučilišnom studiju Računarstvo i matematika na istom fakultetu.