Paralelno programiranje u Qt biblioteci

Popović, Dorotea

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Dorotea Popović

PARALELNO PROGRAMIRANJE U QT BIBLIOTECI

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, veljača 2025.

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

Na današnjim operacijskim sustavima moguće je izvršavanje više neovisnih zadataka istovremeno, što se naziva paralelizmom. To se postiže korištenjem programskih niti unutar programa. Svaka programska nit izvršava svoje instrukcije neovisno o drugim programskim nitima. Svaki program ima glavnu programsku nit u kojoj se izvršava funkcija `main()`. Glavna programska nit automatski se kreira kada se pokrene program. Ako su potrebne dodatne programske niti koje će se izvršavati paralelno s glavnom programskom niti, njih treba ručno kreirati unutar programa. Ako računalo ima više procesora, svaka programska nit može raditi na svojem procesoru. Ako računalo ima samo jedan procesor ili ako je broj procesora manji od broja programskih niti, programske niti radit će naizmjenično u kratkim vremenskim intervalima. Svaka programska nit ima svoju lokalnu memoriju, ali postoji i zajednička memorija kojoj mogu pristupiti sve programske niti.

Cilj ovog diplomskog rada je proučiti podršku za paralelno programiranje s programskim nitima u Qt 6 *frameworku* za izradu grafičkih korisničkih sučelja i to usporediti sa paralelnim programiranjem u standardnoj biblioteci C++20. U prvom poglavlju opisan je Qt *framework* i njegove glavne funkcionalnosti. Naglasak je stavljen na mehanizam signala i utora koji se koristi za komunikaciju između programskih niti. U drugom poglavlju opisani su načini na koje je moguće implementirati višenitne aplikacije u Qt-u, kao i klase potrebne za zaštitu zajedničkih podataka i sinkronizaciju programskih niti u Qt-u. U slijedećem poglavlju opisane su klase iz standardne C++ biblioteke koje su potrebne za kreiranje programskih niti, međusobno isključivanje i sinkronizaciju programskih niti. Također su opisane sličnosti i razlike u odnosu na klase i mehanizme za paralelno programiranje u Qt-u. U zadnjem poglavlju opisan je primjer aplikacije s programskim nitima u Qt-u.

Poglavlje 1

Qt

Qt je *framework* za razvoj aplikacija na više platformi za stolna i prijenosna računala te mobilne uređaje. Podržane platforme su Linux, Windows, Android, iOS i druge. Qt je napisan u programskom jeziku C++ i pruža alate u raznim područjima, kao što su grafička korisnička sučelja, programske niti, umreživanje i mnoga druga.

Razvoj Qt-a započeli su 1990. godine Norveški programeri Eirik Chambe-Eng i Haavard Nord. Osnovali su tvrtku koja se zvala *Trolltech* i bavila se prodavanjem Qt licenca. U svibnju 1995. godine prva verzija Qt-a, Qt 0.90 postala je javno dostupna. Bivši *Trolltech* 2014. godine dobio je ime *The Qt Company* i postao je podružnica u potpunom vlasništvu tvrtke *Digia*. Razvojem Qt-a danas upravlja *The Qt Project* koji uključuje mnogo tvrtki i pojedinaca diljem svijeta.

1.1 Qmake

Bilo koji sustav izgradnje može se koristiti s Qt-om, ali Qt ima i svoj vlastiti sustav izgradnje *qmake*. *Qmake* je alat za izgradnju aplikacija, biblioteka i drugih komponenti koji automatizira generiranje *Makefile* datoteka. *Makefile* datoteka sadrži sve potrebne naredbe za izgradnju projekta, a generira se na temelju informacija u projektnoj datoteci. Projektna (.pro) datoteka obično sadrži popis izvornih datoteka i datoteka zaglavlja, opće informacije o konfiguraciji i sve pojedinosti specifične za aplikaciju, kao što je popis dodatnih biblioteka.

Uz pomoć *qmake*-a proces izgradnje razvojnih projekata na različitim platformama je jednostavan, bez obzira je li projekt napisan korištenjem Qt-a ili ne. Za jednostavne projekte, *qmake* je potrebno pokrenuti samo u direktoriju najviše razine projekta da bi se generirala *Makefile* datoteka. Zatim se može pokrenuti *make* alat za izgradnju projekta na temelju *Makefile* datoteke.

1.2 Qt Creator

Qt ima i svoje vlastito integrirano razvojno okruženje (eng. *integrated development environment*, IDE) koje se naziva Qt Creator. Podržan je na operacijskim sustavima Linux, OS X i Windows. Pomoću Qt Creator-a mogu se razvijati aplikacije za stolna i prijenosna računala, mobilne uređaje i web preglednike. Razlikuje se od uređivača teksta po tome što zna kako izgraditi i pokrenuti aplikacije. Razumije C++ i QML jezike kao kod, a ne samo kao običan tekst. Stoga nudi korisne značajke kao što su inteligentno dovršavanje koda, isticanje sintakse, integrirani sustav pomoći, integraciju ispravljanja pogrešaka i integraciju za sve glavne sustave za upravljanje kodom (npr. git, Bazaar). Jednom kad je aplikacija razvijena pomoću Qt-a, moguće ju je uvesti na različite platforme.

1.3 Grafička korisnička sučelja

U Qt-u grafička korisnička sučelja mogu se dizajnirati na dva načina: pomoću Qt Widgets-a i Qt Quick-a. Qt Widgets prikladan je za stvaranje aplikacija namijenjenih stolnim i prijenosnim računalima, a Qt Quick korisnička sučelja optimalna su za mobilne aplikacije.

Grafička korisnička sučelja kreirana pomoću Qt Widgets modula napisana su izravno u C++-u. Qt Widgets su elementi korisničkog sučelja koji su tipični u okruženjima za stolna računala. Dobro se integriraju u temeljnu platformu, čime pružaju izvorni izgled i dojam na Windows-u, Linux-u i macOS-u. Bogati su značajkama prikladnim uglavnom za tradicionalna korisnička sučelja. Qt nudi interaktivni grafički alat Qt Designer koji funkcionira kao generator koda za grafička korisnička sučelja temeljena na *widgets*-ima. Qt Designer može se koristiti samostalno, a također je integriran i u Qt Creator.

Qt Quick stvara dinamička i fluidna korisnička sučelja. Grafička korisnička sučelja kreirana pomoću Qt Quick-a napisana su u QML-u (eng. *Qt Modeling Language*), a logika je implementirana u C++-u. QML je deklarativni jezik za opis objekata koji integrira JavaScript za proceduralno programiranje. Modul Qt Quick nudi QML tipove kao što su gumbi, dijalozi i izbornici.

1.4 Signali i utori

U programiranju grafičkog korisničkog sučelja, važno je da različiti objekti mogu međusobno komunicirati. U Qt-u se za komunikaciju između različitih grafičkih komponenti koristi mehanizam signala i utora (eng. *signals and slots*). Isti mehanizam može se koristiti za sigurnu komunikaciju između različitih programskih niti.

Prilikom pojave određenog događaja, emitira se signal. Kao odgovor na određeni signal poziva se funkcija koju nazivamo utor. Qt Widgets modul ima mnogo unaprijed definira-

nih signala i utora. Svoje vlastite signale i utore moguće je definirati u klasama koje su naslijeđene iz klase QObject ili neke njezine podklase.

Signali i utori mogu uzeti proizvoljan broj argumenata bilo kojeg tipa, ali broj i tipovi argumenata koje uzima signal mora odgovarati broju i tipovima argumenata koje uzima odgovarajući utor. Utor može uzeti i manji broj argumenata nego signal koji prima jer može zanemariti dodatne argumente. Signal se povezuje s utorom pomoću metode `QObject::connect()`.

Klasa koja emitira signal ne zna koji utori primaju taj signal niti prima li uopće neki utor taj signal. Isto tako, utor ne zna ima li signala koji su spojeni na njega. To osigurava da su komponente potpuno nezavisne. Ako je signal povezan s utorom, mehanizam signala i utora osigurava da će utor biti pozvan s parametrima odgovarajućeg signala u pravo vrijeme.

Bilo koji broj signala može se spojiti na jedan utor, a jedan signal može se spojiti na bilo koji broj utora. Moguće je i spojiti jedan signal izravno na drugi signal. Tada će se odmah emitirati drugi signal kad god se emitira prvi signal.

Promotrimo sliku 1.1. Vidimo da je signal signal1 iz objekta *Object1* spojen na dva utora *slot1* i *slot2* u objektu *Object2*. Na slici je također vidljivo da različiti signali iz istog objekta mogu biti spojeni na utore u različitim objektima. Isto tako, signali iz različitih objekata mogu biti spojeni na utore u istom objektu. Signal *signal2* iz objekta *Object1* spojen je na utor *slot1* u objektu *Object4*, a signal *signal1* iz objekta *Object3* spojen je na utor *slot3* u objektu *Object4*.

Signali

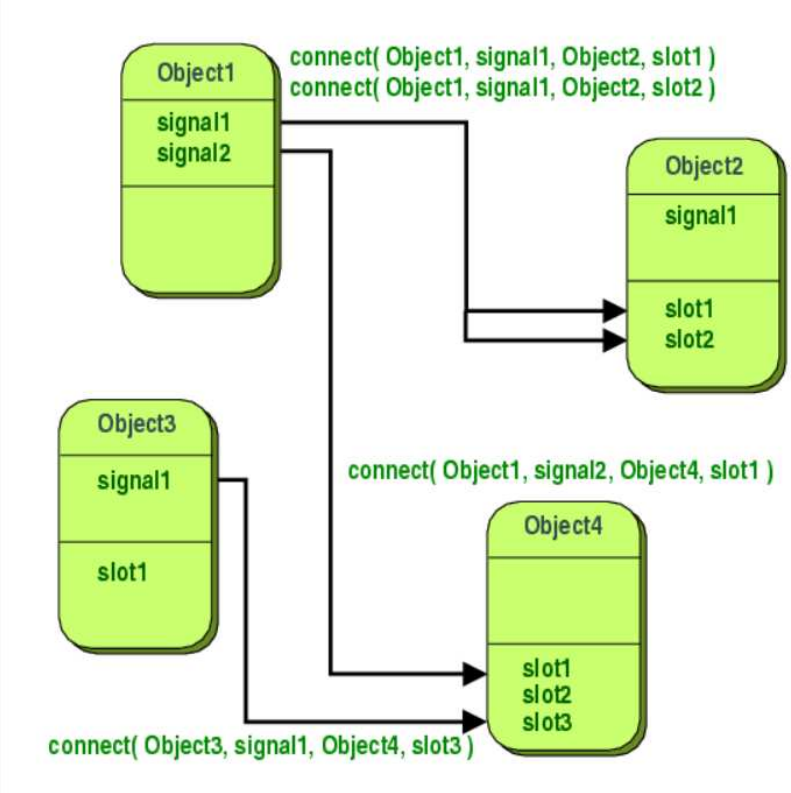
Kada se stanje objekta promijeni na neki način, on emitira signal. Kako su signali funkcije javnog pristupa, moguće ih je emitirati s bilo kojeg mjesta. Međutim, preporučuje se da se emitiraju samo iz klase u kojoj je definiran signal ili iz njezinih podklasa.

Utor povezan sa signalom izvršava se odmah kada se signal emitira. Ako je sa signalom povezano više utora, oni će se izvršavati onim redoslijedom kojim su bili spojeni, jedan za drugim. Signal se emitira naredbom `emit`. Kod koji slijedi nakon naredbe `emit` nastavlja se izvršavati tek nakon što se izvrše svi utori povezani sa emitiranim signalom.

Utori

Utor se poziva kada se povezani signal emitira. Utori su obične funkcije članice koje slijede uobičajena C ++ pravila kada se izravno pozivaju. Međutim, njihova posebna značajka je mogućnost povezivanja sa signalima. Putem veze signal-utor, može ih pozvati bilo koja klasa, bez obzira na razinu pristupa. To može uzrokovati da se kao odgovor na signal

emitiran iz instance proizvoljne klase pozove privatni utor u instanci druge, nepovezane klase.



Slika 1.1: Signali i utori [21]

Poglavlje 2

Programske niti u Qt-u

Qt pruža podršku za programske niti kroz klase koje su neovisne o platformi. Također pruža sigurno objavljivanje događaja unutar programskih niti te povezivanje signala i utora među njima. To pojednostavljuje razvoj prijenosnih Qt aplikacija s programskim nitima i efikasno korištenje višeprocesorskih sustava. Programiranje s programskim nitima također omogućuje izvođenje dugotrajnih operacija bez zamrzavanja korisničkog sučelja aplikacije.

Qt nudi mnoge klase i funkcije za rad s programskim nitima. U Qt-u moguće je implementirati višenitne aplikacije na više načina. U nastavku su opisana četiri različita pristupa: QThread, QThreadPool, Qt Concurrent i WorkerScript QML.

2.1 QThread

QThread je klasa koji služi za upravljanje programskim nitima. Svaki QThread objekt predstavlja i upravlja jednom programskom niti unutar programa. Nova programska nit može se kreirati na dva načina: direktnim kreiranjem instance QThread klase ili definiranjem klase koja proširuje klasu QThread.

Instanciranjem QThread-a kreira se paralelna petlja događaja. Izvršavanje koda u novoj programskoj niti pokreće se pozivanjem metode `start()` na QThread objektu. Metoda `start()` automatski poziva funkciju `QThread::run()` i emitira signal `started()`. Funkcija `run()` pokreće petlju događaja unutar nove programske niti pozivom funkcije `QThread::exec()`. Kako bi programska nit koja je pozvala metodu `start()` čekala da nova programska nit završi s radom, potrebno je pozvati metodu `QThread::wait()`. Metoda `wait()` će blokirati programsku nit koja ju je pozvala sve dok QThread objekt na kojem je pozvana metoda ne završi s izvođenjem `run()` funkcije. Kada funkcija `run()` završi s izvođenjem, automatski se zaustavlja petlja događaja i emitira se signal `finished()`. Slijedeći primjer prikazuje kako se kreira i pokreće nova programska nit.

```
QThread *thread = new QThread();
thread->start();
thread->wait();
```

Listing 2.1: Primjer kreiranja i pokretanja QThread-a

Instanca QThread klase živi u programskoj niti koja ju je instancirala, a ne u novoj programskoj niti koja poziva funkciju `run()`. To znači da će se svi utori klase QThread u redu čekanja izvršiti u staroj programskoj niti. Metode pozvane izravno na QThread objektu izvršit će se u programskoj niti koja je pozvala metodu.

Kada se kreira podklasa QThread klase potrebno je ponovno implementirati funkciju `run()` koja će se izvršavati u novoj programskoj niti.

```
class WorkerThread : public QThread
{
       Q_OBJECT
       void run() override {
              QString result;
              /* here is the expensive or blocking operation */
              emit resultReady(result);
       }
       signals:
       void resultReady(const QString &s);
};

void MyObject::startWorkInAThread()
{
       WorkerThread *workerThread = new WorkerThread(this);
       connect(workerThread, &WorkerThread::resultReady, this,
              &MyObject::handleResults);
       workerThread->start();
}
```

Listing 2.2: Primjer kreiranja podklase QThread klase [17]

U primjeru 2.2 kreira se klasa `WorkerThread` koja proširuje klasu QThread i pomoću nje se kreira programska nit. U novoj programskoj niti ne pokreće se petlja događaja. Kako bi se pokrenula paralelna petlja događaja u novoj programskoj niti, potrebno je iz funkcije `run()` pozvati funkciju `exec()`.

Konstruktor podklase izvršava se u staroj programskoj niti, a funkcija `run()` se izvršava u novoj programskoj niti. Dakle, ako se varijabli članici pristupa iz obje funkcije, pristupa joj se iz dvije različite programske niti, stoga je potrebno zaštititi pristup.

Još jedan način pokretanja koda u novoj programskoj niti je premještanjem objekta u novu programsku nit pomoću metode `QObject::moveToThread()`. Kako bi se zaustavilo

izvršavanje programske niti, poziva se metoda `QThread::quit()` koja zaustavlja petlju događaja programske niti.

```
class Worker : public QObject
{
        Q_OBJECT

        public slots:
        void doWork(const QString &parameter) {
                QString result;
                /* here is the expensive or blocking operation */
                emit resultReady(result);
        }

        signals:
        void resultReady(const QString &result);
};

class Controller : public QObject
{
        Q_OBJECT

        QThread workerThread;
        public:
        Controller() {
                Worker *worker = new Worker;
                worker->moveToThread(&workerThread);
                connect(&workerThread, &QThread::finished,
                        worker, &QObject::deleteLater);
                connect(this, &Controller::operate,
                        worker, &Worker::doWork);
                connect(worker, &Worker::resultReady,
                        this, &Controller::handleResults);
                workerThread.start();
        }
        ~Controller() {
                workerThread.quit();
                workerThread.wait();
        }

        public slots:
        void handleResults(const QString &);

        signals:
        void operate(const QString &);
};

int main(int argc, char *argv[])
```

```
{
        QCoreApplication app(argc, argv);
        Controller controller;
        emit controller.operate("string");
        return app.exec();
}
```

Listing 2.3: Primjer premještanja objekta u drugu programsku nit [17]

U primjeru 2.3, `Controller` kreira `Worker` objekt i premješta ga u drugu programsku nit. Time su sve kreirane signal-utor veze između objekta `Worker` i objekta `Controller` veze u redu čekanja. Kada `Controller` emitira signal `operate()`, utor `doWork()` će biti stavljen u red čekanja petlje događaja programske niti u kojoj se nalazi `Worker`. Isto tako, kada `Worker` emitira signal `resultReady()`, utor `handleResults()` će biti stavljen u red čekanja petlje događaja glavne programske niti jer se u njoj nalazi `Controller`. Pošto je programska nit u kojoj se nalazi `Worker` kreirana unutar `Controller` klase, potrebno je osigurati da ta programska nit završi izvršavanje prije nego se `Controller` objekt uništi. Zato se u destruktoru zovu metode `quit()` i `wait()`. Kada programska nit završi izvršavanje, potrebno je izbrisati `Worker` objekt koji se nalazi u toj programskoj niti. Zato je `QThread::finished()` signal povezan s utorom `QObject::deleteLater()`.

2.2 QThreadPool

Kako bi se smanjili troškovi čestog stvaranja i uništavanja programskih niti, Qt omogućuje ponovno korištenje postojećih programskih niti za nove zadatke. Klasa QThreadPool služi za upravljanje zbirkom QThread-ova za višekratnu upotrebu. Svaka Qt aplikacija ima jedan globalni QThreadPool objekt koji se može dohvatiti putem metode `QThreadPool::globalInstance()`. Globalna QThreadPool instanca automatski održava optimalan broj programskih niti na temelju broja jezgri procesora. Maksimalni mogući broj programskih niti može se dohvatiti putem metode `QThreadPool::maxThreadCount()`

Za izvršavanje koda u jednoj od programskih niti QThreadPool-a potrebno je kreirati podklasu QRunnable klase i ponovno implementirati funkciju `QRunnable::run()`. Da bi se pokrenulo izvršavanje koda u programskoj niti, QRunnable objekt potrebno je proslijediti metodi `QThreadPool::start()`. Metoda `QThreadPool::start()` stavlja QRunnable objekt u red čekanja QThreadPool-a. Kada programska nit postane dostupna, kod unutar funkcije `QRunnable::run()` izvršit će se u toj programskoj niti.

```
class HelloWorldTask : public QRunnable
{
        void run() override
        {
```

```
                qDebug() << "Hello world from thread" << QThread::
                    currentThread();
        }
};

HelloWorldTask *hello = new HelloWorldTask();
// QThreadPool takes ownership and deletes 'hello' automatically
QThreadPool::globalInstance()->start(hello);
```

Listing 2.4: Primjer pokretanja programske niti pomoću QThreadPool-a [18]

U primjeru 2.4, kod unutar funkcije `HelloWorldTask::run()` izvršit će se u jednoj od programskih niti QThreadPool-a. Kada se pokrene programska nit, QThreadPool instanca preuzima vlasništvo nad `hello` objektom i automatski ga briše kada funkcija `run()` završi izvršavanje.

2.3 Qt Concurrent

Qt Concurrent modul omogućuje programiranje s programskim nitima bez korištenja primitiva niske razine kao što su mutexi ili semafori. Kako bi se Qt Concurrent modul mogao koristiti, potrebno je u projektnu datoteku dodati liniju `QT += concurrent`.

Metoda `QtConcurrent::run()` služi za pokretanje bilo koje funkcije u drugoj programskoj niti. Funkcija koja će se izvršavati u drugoj programskoj niti prosljeđuje se metodi `run()` kao argument. Rezultat funkcije dostupan je kroz objekt QFuture. U slijedećem primjeru funkcija `aFunction()` izvršit će se u novoj programskoj niti.

```
extern void aFunction();
QFuture<void> future = QtConcurrent::run(aFunction);
```

Listing 2.5: Primjer pozivanja metode `QtConcurrent::run()` [16]

Ako funkcija prima neke argumente, njih je također potrebno proslijediti metodi `run()`. Na mjestu poziva funkcije `run()` kreiraju se kopije argumenata i prosljeđuju se programskoj niti kada počne izvršavanje funkcije. Na slijedećem primjeru prikazano je kako se prosljeđuju argumenti funkciji `aFunctionWithArguments` koja se izvršava u novoj programskoj niti.

```
extern void aFunctionWithArguments(int arg1, double arg2,
                                   const QString &string);

int integer = 1;
double floatingPoint = 2.0;
QString string = "string";
```

```
QFuture<void> future = QtConcurrent::run(aFunctionWithArguments,
                                         integer,
                                         floatingPoint,
                                         string);
```

Listing 2.6: Primjer prosljeđivanja argumenata programskoj niti [16]

Povratna vrijednost funkcije može se dohvatiti kroz QFuture objekt pomoću metode `result()`. Pozivom metode `QFuture::result()` blokira se trenutna programska nit dok rezultat funkcije ne postane dostupan.

```
extern QString functionReturningAString();
QFuture<QString> future = QtConcurrent::run(functionReturningAString);
...
QString result = future.result();
```

Listing 2.7: Primjer dohvaćanja povratne vrijednosti funkcije [16]

U primjeru 2.7 povratna vrijednost funkcije `functionReturningAString()`je tipa `QString`. Zato je predložak klase QFuture instanciran parametrom tipa `QString`.

Rezultat izračunavanja funkcije može se pohraniti u QPromise objekt. Nakon što izračunavanje završi, rezultat će biti dostupan kroz QFuture objekt. U tom slučaju funkcija koja vrši izračunavanje prima dodatni argument tipa QPromise. Argument tipa QPromise uvijek treba biti prvi argument.

```
void helloWorldFunction(QPromise<QString> &promise)
{
       promise.addResult("Hello");
       promise.addResult("world");
}

QFuture<QString> future = QtConcurrent::run(helloWorldFunction);
...
QList<QString> results = future.results();
```

Listing 2.8: Primjer prosljeđivanja rezultata funkcije [16]

U primjeru 2.8 prikazano je kako se u QPromise objekt može pohraniti više rezultata. Metoda `addResult()` pohranjuje rezultat na kraj kolekcije rezultata. U slučaju kada je pohranjeno više rezultata, na QFuture objektu poziva se metoda `results()` kako bi se dohvatili svi rezultati. Predlošci klasa QPromise i QFuture trebaju biti instancirani parametrom tipa rezultata funkcije. QPromise argument instancira se unutar `run()` funkcije i njegova referenca se prosljeđuje funkciji `helloWorldFunction()`.

QPromise također omogućuje pauziranje, nastavljanje i otkazivanje računanja, ako je to zatraženo. To je prikazano na slijedećem primjeru.

```
void aFunction(QPromise<int> &promise)
{
        for (int i = 0; i < 100; ++i) {
                promise.suspendIfRequested();
                if (promise.isCanceled())
                return;

                const int res = i ;
                promise.addResult(res);
        }
}

int main(int argc, char *argv[])
{
        QCoreApplication app(argc, argv);
        QFuture<int> future = QtConcurrent::run(aFunction);

        QThread::sleep(10);
        future.suspend();

        QThread::sleep(10);
        future.resume();

        QThread::sleep(10);
        future.cancel();

        return app.exec();
}
```

Listing 2.9: Primjer pauziranja, nastavljanja i otkazivanja računanja [16]

U primjeru 2.9 pozivom `future.suspend()` traži se zaustavljanje izvršavanja funkcije `aFunction()`. Nakon poziva funkcije `suspend()`, funkcija `aFunction()` će se zaustaviti izvršavati kada dođe do slijedećeg poziva `promise.suspendIfRequested()`. Funkcija `aFunction()` će se nastaviti izvršavati kada se pozove `future.resume()`. Nakon poziva `future.cancel()`, slijedeći poziv `promise.isCanceled()` u `aFunction()` funkciji će vratiti `true` i funkcija `aFunction()` će prekinuti s radom.

2.4 WorkerScript QML

WorkerScript QML tip omogućuje pokretanje JavaScript koda u novoj programskoj niti paralelno s glavnom programskom niti grafičkog korisničkog sučelja. To je korisno za računanje u pozadini tako da glavna programska nit ne bude blokirana. Poruke se mogu slati između nove programske niti i glavne programske niti pomoću `sendMessage()` metode i `onMessage()` *handlera*.

Svakoj WorkerScript instanci može biti pridružena jedna .js skripta. Pozivom metode `WorkerScript.sendMessage()`, skripta će se pokrenuti u zasebnoj programskoj niti. Kada skripta završi s izvođenjem, nova programska nit može poslati odgovor natrag glavnoj programskoj niti koja će pozvati *handler* `WorkerScript.onMessage()`.

2.5 Sinkronizacija programskih niti

Svrha programskih niti je omogućiti paralelno pokretanje koda, no postoje situacije u kojima jedna programska nit treba stati i čekati drugu programsku nit. Ako dvije programske niti pokušaju istovremeno pisati u istu varijablu, rezultat će biti nedefiniran. Zato je potrebno zaštititi zajedničke resurse, a to se postiže međusobnim isključivanjem. Qt pruža primitive niske razine i mehanizme visoke razine za sinkronizaciju programskih niti.

QMutex

QMutex je osnovna klasa za provođenje međusobnog isključivanja. Kako bi programska nit dobila pristup zajedničkom resursu, treba zaključati *mutex* metodom `QMutex::lock()`. Kada završi svoj zadatak, otključava *mutex* metodom `QMutex::unlock()`. Ako neka druga programska nit pokuša zaključati isti *mutex* dok je već zaključan, biti će stavljena u stanje mirovanja dok *mutex* ne bude otključan. Time je omogućeno da u svakom trenutku samo jedna programska nit može pristupiti zajedničkom resursu.

```
QMutex mutex;
int number = 6;

void method1()
{
        mutex.lock();
        number *= 5;
        number /= 4;
        mutex.unlock();
}

void method2()
{
        mutex.lock();
        number *= 3;
        number /= 2;
        mutex.unlock();
}
```

Listing 2.10: Primjer zaključavanja i otključavanja QMutex-a [11]

Ako se u primjeru 2.10 metode `method1()` i `method2()` izvršavaju istovremeno u dvjema različitim programskim nitima, potreban je QMutex jer obje metode pristupaju istoj varijabli.

Ako programska nit zaključa *mutex*, ali ga ne otključa, aplikacija se može zamrznuti jer će zajednički resurs postati trajno nedostupan drugim programskim nitima. To se može dogoditi u slučaju kada program izbaci iznimku dok je *mutex* zaključan. Zbog toga je umjesto eksplicitnog pozivana metoda `lock()` i `unlock()` sigurnije koristiti QMutexLocker koji će osigurati pravilno zaključavanje i otključavanje *mutexa*. QMutexLocker će u svojem konstruktoru zaključati *mutex*, a u destruktoru će otključati *mutex*.

```
int complexFunction(int flag)
{
        QMutexLocker locker(&mutex);

        int retVal = 0;

        switch (flag) {
                case 0:
                case 1:
                return moreComplexFunction(flag);
                case 2:
                {
                        int status = anotherFunction();
                        if (status < 0)
                        return -2;
                        retVal = status + flag;
                }
                break;
                default:
                if (flag > 10)
                return -1;
                break;
        }

        return retVal;
}
```

Listing 2.11: Primjer zaključavanja QMutex-a pomoću QMutexLocker-a [12]

Kada bi u primjeru 2.11 QMutex bio zaključan metodom `QMutex::lock()`, prije svake `return` naredbe trebalo bi pozvati metodu `QMutex::unlock()`. Zato je jednostavnije i sigurnije koristiti `QMutexLocker`. Kada se izlazi iz funkcije, QMutexLocker će biti uništen i time će se QMutex otključati.

QReadWriteLock

QReadWriteLock je klasa za zaštitu resursa kojima se može pristupiti za čitanje i pisanje. Slična je klasi QMutex, osim što ima metode `lockForRead()` i `lockForWrite()` kojima razlikuje pristup za čitanje i pristup za pisanje. Omogućuje istovremeno čitanje iste varijable iz različitih programskih niti. Ako nijedna programska nit ne zapisuje u varijablu, sigurno je da više programskih niti istovremeno čita iz nje.

```
QReadWriteLock lock;

void ReaderThread::run()
{
       lock.lockForRead();
       read_file();
       lock.unlock();
}

void WriterThread::run()
{
       lock.lockForWrite();
       write_file();
       lock.unlock();
}
```

Listing 2.12: Primjer zaključavanja QReadWriteLock-a za čitanje i za pisanje [14]

Kako bi se osiguralo da programske niti koje čitaju ne blokiraju zauvijek programske niti koje žele pisati, programska nit koja pokušava dobiti pristup za čitanje bit će blokirana ako već postoji blokirana programska nit koja čeka pristup za pisanje, čak i ako trenutno pristup imaju samo programske niti koje čitaju. Također, ako trenutno pristup ima programska nit koja piše i druga programska nit traži pristup za pisanje, ona će imati prednost nad svim programskim nitima koje čekaju pristup za čitanje.

Slično kao klasa QMutexLocker za QMutex, i za QReadWriteLock postoje klase QReadLocker i QWriteLocker koje se koriste kako bi se smanjila šansa da zajednički resurs slučajno postane trajno zaključan. QReadLocker u konstruktoru zaključava QReadWriteLock za čitanje, a u destruktoru ga otključava. QWriteLocker u konstruktoru zaključava QReadWriteLock za pisanje, a u destruktoru ga otključava. U slijedećim primjerima QReadLocker i QWriteLocker će biti uništeni pri izlasku iz funkcija i time će se QReadWriteLock otključati.

```
QReadWriteLock lock;
int data;

int readData()
```

```
{
        QReadLocker locker(&lock);
        return data;
}
```

Listing 2.13: Primjer zaključavanja QReadWriteLock-a za čitanje pomoću QReadLocker-a [13]

```
QReadWriteLock lock;
int data;

void writeData()
{
        QWriteLocker locker(&lock);
        data = 0;
}
```

Listing 2.14: Primjer zaključavanja QReadWriteLock-a za pisanje pomoću QWriteLocker-a [20]

QSemaphore

Klasa QSemaphore je generalizacija klase QMutex. Za razliku od QMutex-a koji štiti točno jedan resurs, QSemaphore može štititi određeni broj identičnih resursa. QSemaphore ima metode `acquire()` i `release()`. Pozivom metode `acquire(n)` pokušava se steći pristup na *n* resursa. Ako toliko resursa nije dostupno, poziv metode će biti blokiran dok resursi ne postanu dostupni. Broj dostupnih resursa može se dohvatiti pomoću metode `available()`. Pozivom metode `tryAcquire(n)` također se pokušava steći pristup na *n* resursa. Metoda vraća `true` ako postoji barem *n* dostupnih resursa, inače vraća `false`. Pozivom metode `release(n)` otpušta se *n* resursa.

```
QSemaphore sem(5);      // sem.available() == 5

sem.acquire(3);         // sem.available() == 2
sem.acquire(2);         // sem.available() == 0
sem.release(5);         // sem.available() == 5
sem.release(5);         // sem.available() == 10

sem.tryAcquire(1);      // sem.available() == 9, returns true
sem.tryAcquire(250);    // sem.available() == 9, returns false
```

Listing 2.15: Primjer korištenja QSemaphore-a [15]

QWaitCondition

Klasa QWaitCondition pruža uvjetnu varijablu za sinkronizaciju programskih niti. Za razliku od prethodno navedenih klasa, korištenjem QWaitCondition-a programske niti ne čekaju da zajednički resurs postane dostupan, nego da se ispuni određeni uvjet. Programska nit koja čeka da uvjet postane ispunjen poziva metodu `QWaitCondition::wait()`. Kada uvjet postane ispunjen, na QWaitCondition objektu poziva se metoda `wakeOne()` da bi se probudila jedna nasumično odabrana programska nit na čekanju ili metoda `wakeAll()` da bi se probudile sve programske niti na čekanju.

```
QMutex mutex;
QWaitCondition condition;

class Worker : public QThread
{
        protected:
        void run() override
        {
                mutex.lock();
                condition.wait(&mutex);
                /* do some work */
                mutex.unlock();
        }
};

int main(int argc, char *argv[])
{
        QCoreApplication app(argc, argv);
        Worker worker;
        worker.start();
        /* do some work */
        condition.wakeOne();
        worker.wait();
        return app.exec();
}
```

Listing 2.16: Primjer korištenja QWaitCondition-a

Redovi čekanja događaja

Qt-ov sustav događaja pruža učinkovit način komunikacije između programskih niti. Svaka programska nit može imati svoju vlastitu petlju događaja. Glavna programska nit započinje svoju petlju događaja pomoću metode `QCoreApplication::exec()`, dok ostale programske niti mogu pokrenuti svoju petlju događaja pomoću metode `QThread::exec()`. Da bi se utor pozvao u drugoj programskoj niti, taj poziv potrebno je staviti u petlju

događaja ciljne programske niti. Time se osigurava da ciljna programska nit završi svoj trenutni zadatak prije nego utor započne s radom. Početna programska nit nastavlja raditi paralelno.

Da bi se poziv stavio u petlju događaja druge programske niti, potrebno je uspostaviti signal-utor vezu u redu čekanja. Takva veza kreira se kada pošiljatelj signala i primatelj signala žive u različitim programskim nitima. Kada se emitira signal, sustav događaja zabilježit će njegove argumente. Kada se kontrola vrati u petlju događaja programske niti u kojoj živi primatelj signala, pokrenut će se utor.

Korištenje sustava događaja za sinkronizaciju programskih niti uklanja rizik od zastoja koji se može dogoditi korištenjem primitiva niske razine. Međutim, sustav događaja ne omogućuje međusobno isključivanje. Ako metode pristupaju zajedničkim podacima i dalje je potrebno zaštititi te podatke primitivima niske razine.

2.6 Sigurnost programskih niti

Funkcija je sigurna za programske niti (eng. *thread-safe*) ako se može pozvati istovremeno iz više programskih niti, čak i kada pozivi koriste zajedničke podatke. Svi pristupi zajedničkim podacima su serijalizirani. Klasa je sigurna za programske niti ako se njezine funkcije članice mogu sigurno pozvati iz više programskih niti, čak i ako sve programske niti koriste istu instancu klase.

Reentrantna funkcija (eng. *reentrant*) se također može istovremeno pozvati iz više programskih niti, ali samo ako svaki poziv koristi svoje vlastite podatke. Za klasu se kaže da je reentrantna ako se njezine funkcije članice mogu sigurno pozvati iz više programskih niti, pod uvjetom da svaka programska nit koristi različitu instancu klase.

Poglavlje 3

Programske niti u C++-u

3.1 Klasa `std::thread`

Klasa `std::thread` omogućuje paralelno izvršavanje više funkcija. Konstruktor objekta `std::thread` kao argument prima funkciju koja će se izvršavati paralelno s glavnim programom. Svaki tako kreirani `std::thread` objekt predstavlja jednu programsku nit. Funkcija koja je predana kao argument konstruktoru počet će se izvršavati u novoj programskoj niti čim se kreira `std::thread` objekt. U glavnoj programskoj niti istovremeno se nastavlja izvršavanje glavnog programa. Kada završi izršavanje glavnog programa, sve programske niti koje su u njemu pokrenute prekidaju s radom.

```
void f1()
{
        std::cout << "Thread 1 executing\n";
}

int main()
{
        std::thread t1(f1)
        t1.join();
}
```

Listing 3.1: Primjer kreiranja programske niti [1]

Kako bi se osiguralo da sve programske niti pokrenute u glavnom programu završe s radom, potrebno je pozvati metodu `std::thread::join()` ili `std::thread::detach()` na svim `std::thread` objektima prije završetka glavnog programa. Metoda `join()` će blokirati glavnu programsku nit dok programska nit na kojoj je metoda pozvana ne završi s radom. Metoda `detach()` odvaja programsku nit od `std::thread` objekta i time omogućuje da programska nit nastavi neovisno raditi i nakon što završi izvršavanje glav-

nog programa. Obje metode moguće je pozvati samo jednom na svakom `std::thread` objektu. Nakon poziva metode `join()` ili `detach()`, `std::thread` objekt nije više povezan sa programskom niti. Dakle, ponovni poziv metoda na istom `std::thread` objektu izbacuje izuzetak.

U `std::thread` klasi također postoji metoda `joinable()` koja provjerava je li moguće pozvati metode `join()` i `detach()`. Metoda `joinable()` vraća `true` ako je `std::thread` objekt povezan s programskom niti, inače vraća `false`. `std::thread` objekt kreiran dodijeljenim konstruktorom ne predstavlja programsku nit i na njemu metoda `joinable()` vraća `false`.

Klasa `std::thread` ne nudi nikakav mehanizam povrata vrijednosti iz programske niti u glavni program. Dakle, ako funkcija koja se predaje konstruktoru nije tipa void, njezina povratna vrijednost se zanemaruje. Za prosljeđivanje povratnih vrijednosti ili iznimaka u glavnu programsku nit može se koristiti `std::promise` ili `std::async`.

Objekti tipa `std::thread` ne mogu se kopirati zato što dva `std::thread` objekta ne mogu predstavljati istu programsku nit. Mogu se samo premještati, pri čemu se predaje vlasništvo nad programkom niti. Objekt koji predaje vlasništvo nije više povezan s programskom niti, a objekt koji preuzima vlasništvo postaje povezan s programskom niti ako već nije povezan s nekom drugom programskom niti.

3.2 Klasa `std::jthread`

U standardu C++20 uvedena je nova klasa `std::jthread`. Ima isto ponašanje kao klasa `std::thread`, uz nekoliko dodatnih mogućnosti.

Glavna razlika je u tome što `std::jthread` u svom destruktoru poziva metodu `join()` ako je `std::jthread` objekt povezan s nekom programskom niti. Ako je prethodno pozvana metoda `join()` ili `detach()` na `std::jthread` objektu, on više nije povezan s programskom niti i u destruktoru neće biti pozvana metoda `join()`.

Još jedna razlika u odnosu na klasu `std::thread` je u tome što je programsku nit koju predstavlja `std::jthread` objekt moguće zaustaviti usred izvršavanja. Konstruktor `std::jthread` objekta kao argument prima funkciju koja kao prvi argument prima `std::stop_token` koji omogućuje funkciji da ispita je li zatraženo zaustavljanje tijekom njezinog izvršavanja.

3.3 Prijenos parametara programskoj niti

Ako funkcija koja se izvršava u novoj programskoj niti prima argumente, onda i konstruktor `std::thread` objekta prima te iste argumente u odgovarajućem poretku nakon same funkcije. Konstruktor `std::thread` objekta može imati proizvoljno mnogo parametara.

Prvi parametar je funkcija koja se izvršava u novoj programskoj niti, a ostali parametri su argumenti koji se predaju toj funkciji.

Argumenti se funkciji prenose po vrijednost. Dakle, konstruktor radi kopije argumenata i predaje ih funkciji. Kako bi se argument mogao prenijeti po referenci, potrebno ga je zamotati u `std::ref` ili `std::cref` objekt.

```
void f1(int n)
{
        for (int i = 0; i < 5; ++i)
        {
                std::cout << "Thread 1 executing\n";
                ++n;
        }
}

void f2(int& n)
{
        for (int i = 0; i < 5; ++i)
        {
                std::cout << "Thread 2 executing\n";
                ++n;
        }
}

int main()
{
        int n = 0;
        std::thread t1(f1, n + 1); // pass by value
        std::thread t2(f2, std::ref(n)); // pass by reference
        t1.join();
        t2.join();
}
```

Listing 3.2: Primjer prijenosa parametara programskoj niti [1]

3.4 Međusobno isključivanje

Postojanje zajedničke memorije kojoj mogu pristupiti sve programske niti u programu može dovesti do problema ako više programskih niti u svojim funkcijama koriste iste podatke.

Ako postoji varijabla u programu kojoj se može pristupiti iz svih programskih niti i sve programske niti samo čitaju njezinu vrijednost iz dijeljene memorije, a nijedna programska nit ju ne mijenja, tada će sve programske niti pročitati istu vrijednost dijeljene varijable bez

obzira kojim redosljedom pristupaju varijabli. Problem nastaje ako jedna programska nit mijenja vrijednost varijable i upisuje u dijeljenu memoriju dok druga programska nit čita vrijednost varijable. Nije moguće unaprijed znati kojim redoslijedom će programske niti pristupiti varijabli i hoće li druga programska nit pročitati staru ili novu vrijednost varijable.

Taj problem nastaje zato što operacije koje mijenjaju dijeljenu varijablu nisu nedjeljive, odnosno atomske. Operacija je atomska ako je druge programske niti mogu vidjeti kao završenu ili još nezapočetu, ali ne i kao djelomično izvršenu. Dakle, potrebno je sve operacije koje mijenjaju dijeljene varijable učiniti atomskim. To se postiže uvođenjem *mutexa*. Svaka programska nit treba zaključati *mutex* prije ulaska u kritični dio koda i otključati *mutex* nakon izlaska iz kritičnog dijela. Time se spriječava da više programskih niti istovremeno pristupa dijeljenoj memoriji.

Klasa `std::mutex`

Kada više programskih niti pristupa zajedničkoj varijabli, potrebno je osigurati da joj ne pristupa više programskih niti istovremeno. Za to služi objekt tipa `std::mutex` (eng. *mutex=mutual exclusion*). Klasa `std::mutex` ima metode `lock()` i `try_lock()` za zaključavanje, te metodu `unlock()` za otključavanje.

Programska nit ima vlasništvo nad *mutexom* od trenutka kada uspješno pozove metodu `lock()` ili `try_lock()` do trenutka kada pozove metodu `unlock()`. Ako u tom vremenu neka druga programska nit pokuša pozvati metodu `lock()` nad istim *mutexom*, ta programska nit će biti blokirana sve dok *mutex* ne bude otključan.

```
int g_num = 0; // protected by g_num_mutex
std::mutex g_num_mutex;

void slow_increment(int id)
{
        for (int i = 0; i < 3; ++i)
        {
                g_num_mutex.lock();
                ++g_num;
                // note, that the mutex also syncronizes the output
                std::cout << "id: " << id
                          << ", g_num: " << g_num << '\n';
                g_num_mutex.unlock();

                std::this_thread::sleep_for(
                        std::chrono::milliseconds(234)
                );
        }
}
```

```
int main()
{
	std::thread t1{slow_increment, 0};
	std::thread t2{slow_increment, 1};
	t1.join();
	t2.join();
}
```

Listing 3.3: Primjer korištenja *mutexa* [1]

Ako nijedna programska nit nema vlasništvo nad *mutexom*, metoda `try_lock()` pozvana iz bilo koje programske niti vratit će vrijednost `true` i zaključati *mutex*. Inače, ako je *mutex* u vlasništvu neke programske niti, metoda `try_lock` pozvana iz neke druge programske niti koja nema vlasništvo nad *mutexom* vratit će vrijednost `false` i neće blokirati programsku nit iz koje je pozvana.

Objekti tipa `std::mutex` se ne mogu kopirati niti premještati.

Klasa `std::lock_guard`

Na svakom `std::mutex` objektu na kojem je pozvana metoda `lock()` treba se pozvati i metoda `unlock()`, čak i u slučaju kada se iz kritičnog dijela koda izlazi zbog izbačenog izuzetka. U suprotnom će druga programska nit koja pokušati pozvati metodu `lock()` biti trajno blokirana. Zato je sigurnije umjesto eksplicitnog pozivanja metoda `lock()` i `unlock()` koristiti klasu `std::lock_guard`.

Konstruktor klase `std::lock_guard` prima *mutex*. U konstruktoru se poziva metoda `lock()` na *mutexu*, a u destruktoru se poziva metoda `unlock()`. Ako je konstruktoru prosljeđen još i argument `std::adopt_lock`, objekt `std::lock_guard` preuzima vlasništvo nad već zaključanim *mutexom* bez da ga pokuša zaključati.

```
std::map<std::string, std::string> g_pages;
std::mutex g_pages_mutex;

void save_page(const std::string& url)
{
	// simulate a long page fetch
	std::this_thread::sleep_for(std::chrono::seconds(2));
	std::string result = "fake content";

	std::lock_guard<std::mutex> guard(g_pages_mutex);
	g_pages[url] = result;
}

int main()
{
```

```
        std::thread t1(save_page, "http://foo");
        std::thread t2(save_page, "http://bar");
        t1.join();
        t2.join();

        // safe to access g_pages without lock now,
        // as the threads are joined
        for (const auto& [url, page] : g_pages)
        std::cout << url << " => " << page << '\n';
}
```

Listing 3.4: Primjer korištenja *lock_guard*-a [1]

Ako kritični dio koda čini jedan blok zatvoren unutar vitičastih zagrada, na početku bloka potrebno je instancirati `std::lock_guard` objekt s odgovarajućim *mutexom*. Time je osigurano da će se pri izlasku iz bloka pozvati metoda `unlock()` na *mutexu*, čak i ako je izbačen izuzetak.

Objekti tipa `std::lock_guard` se ne mogu kopirati niti premještati.

Klasa `std::unique_lock`

Osim klase `std::lock_guard`, za zaključavanje *mutexa* postoji i `std::unique_lock` klasa. Isto kao klasa `std::lock_guard`, ima konstruktor koji prima *mutex* i može primati drugi argument `std::adopt_lock`, pri čemu preuzima već zaključani *mutex*.

Klasa `std::unique_lock` ima metode `lock()`, `try_lock()` i `unlock()`. Time je omogućeno kasnije zaključavanje *mutexa* (ne nužno u konstruktoru) ako je konstruktoru `std::unique_lock` objekta kao drugi argument prosljeđen `std::defer_lock`. Kako ova klasa uzima više resursa nego klasa `std::lock_guard`, koristi se samo ako je *mutex* potrebno otključati i ponovno zaključati.

```
int main()
{
        int counter = 0;
        std::mutex counter_mutex;
        std::vector<std::thread> threads;

        auto worker_task = [&](int id)
        {
                std::unique_lock<std::mutex> lock(counter_mutex);
                ++counter;
                std::cout << id << ", initial counter: "
                          << counter << '\n';
                lock.unlock();
```

```
                // don't hold the lock
                // while we simulate an expensive operation
                std::this_thread::sleep_for(std::chrono::seconds(1));

                lock.lock();
                ++counter;
                std::cout << id << ", final counter: "
                          << counter << '\n';
        };

        for (int i = 0; i < 10; ++i)
        threads.emplace_back(worker_task, i);

        for (auto& thread : threads)
        thread.join();
}
```

Listing 3.5: Primjer korištenja *unique_lock*-a [1]

Klasa `std::unique_lock` također sadrži dvije metode koje omogućuju zaključavanje *mutexa* na ograničeno vrijeme. To su `try_lock_for()` i `try_lock_until()`.

Mutex predan `std::unique_lock` objektu može se nakon zaključavanja otključati i ponovno zaključati. Zbog toga ova klasa ima metodu `owns_lock()` koja provjerava je li *mutex* `std::unique_lock` objekta zaključan ili nije.

Objekti tipa `std::unique_lock` se mogu premještati, ali ne mogu se kopirati.

Klasa `std::shared_mutex`

Međusobno isključivanje nije potrebno kada različite programske niti samo čitaju zajedničke podatke, nego samo kada programska nit mijenja vrijednost zajedničkih podataka. U tom slučaju umjesto klase `std::mutex` moguće je koristiti klasu `std::shared_mutex` koja dozvoljava dijeljeni i ekskluzivni pristup. Svaka programska nit može imati dodijeljenu samo jednu vrstu pristupa u svakom trenutku.

Ekskluzivni pristup dodijeljuje se programskoj niti pozivom `lock()` ili `try_lock()` metode. Kada jedna programska nit dobije ekskluzivni pristup, nijedna druga programska nit ne može dobiti niti dijeljeni niti ekskluzivni pristup.

Dijeljeni pristup dodijeljuje se programskoj niti pozivom metode `lock_shared()` ili `try_lock_shared()`. Kada jedna programska nit dobije dijeljeni pristup, nijedna druga programska nit ne može dobiti ekskluzivni pristup, ali sve programske niti mogu dobiti dijeljeni pristup.

Objekte tipa `std::shared_mutex` moguće je zaključavati pomoću `std::lock_guard` ili `std::unique_lock` objekta ako je potreban ekskluzivni pristup. U slučaju kada je potreban dijeljeni pristup, koristi se `std::shared_lock`. Klasa `std::shared_lock` ima

istu funkcionalnost kao i klasa `std::unique_lock`, samo umjesto metode `lock()` koristi metodu `shared_lock()`.

```
class ThreadSafeCounter
{
        public:
        ThreadSafeCounter() = default;

        // Multiple threads/readers can read the counter's value at the
           same time.
        unsigned int get() const
        {
                std::shared_lock lock(mutex_);
                return value_;
        }

        // Only one thread/writer can increment/write the counter's
           value.
        void increment()
        {
                std::unique_lock lock(mutex_);
                ++value_;
        }

        // Only one thread/writer can reset/write the counter's value.
        void reset()
        {
                std::unique_lock lock(mutex_);
                value_ = 0;
        }

        private:
        mutable std::shared_mutex mutex_;
        unsigned int value_{};
};
```

Listing 3.6: Primjer korištenja *shared_mutex*-a [1]

3.5 Sinkronizacija programskih niti

Osim zaštite zajedičkih resursa, važna je i međusobna sinkronizacija programskih niti. Može doći do situacije da jedna programska nit ne može nastaviti s radom dok neka druga programska nit ne ispuni neki preduvjet ili pošalje signal.

Uvjetne varijable

Uvjetna varijabla (eng. *condition variable*) je mehanizam za sinkronizaciju programskih niti. Omogućuje programskoj niti da obavijesti ostale programske niti kada je zadovoljen određeni uvjet. Postoje dvije klase definirane u zaglavlju ⟨`condition_variable`⟩. To su `std::condition_variable` i `std::condition_variable_any`.

Klasa `std::condition_variable` predstavlja uvjetnu varijablu povezanu s objektom `std::unique_lock`. Ima metode `notify_one()` i `notify_all()` za obaviještavanje programskih niti koje čekaju te metode `wait()`, `wait_for()` i `wait_until()` za čekanje dok se uvjet ne ispuni.

Metoda `notify_one()` obaviještava jednu od programskih niti koje čekaju. Metoda `notify_all()` obaviještava sve programske niti koje čekaju. Metoda `wait()` blokira programsku nit koja je pozvala metodu dok se uvjet ne ispuni. Metode `wait_for()` i `wait_until()` blokiraju programsku nit koja je pozvala metodu dok se uvjet ne ispuni ili dok ne prođe određeno vrijeme.

Kako bi programska nit promijenila vrijednost zajedničke varijable i obavijestila ostale programske niti nakon promjene vrijednosti, najprije je potrebno zaključati *mutex* (najćešće pomoću `std::lock_guard`) i time blokirati ostale programske niti. Programska nit treba obaviti promjenu vrijednosti varijable dok ima vlasništvo nad *mutexom*. Nakon promjene vrijednosti varijable poziva metodu `notify_one()` ili `notify_all()`, što je moguće i nakon otključavanja *mutexa*.

Programska nit koja čeka da uvjet (promjena zajedničke varijable) bude ispunjen, najprije treba zaključati mutex pomoću `std::unique_lock`. Nakon toga je potrebno provjeriti je li uvjet već ispunjen. Ako uvjet nije ispunjen, poziva se metoda `wait()`, `wait_for()` ili `wait_until()` i ponovno se provjerava je li uvjet ispunjen. Ako nije, programska nit ponovno čeka. Postupak se ponavlja sve dok uvjet ne postane ispunjen.

```
std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread()
{
        // wait until main() sends data
        std::unique_lock lk(m);
        cv.wait(lk, []{ return ready; });

        // after the wait, we own the lock
        std::cout << "Worker thread is processing data\n";
        data += " after processing";
```

```
        // send data back to main()
        processed = true;
        std::cout << "Worker thread signals"
                  << "data processing completed\n";

        // manual unlocking is done before notifying, to avoid waking up
        // the waiting thread only to block again
        lk.unlock();
        cv.notify_one();
        std::this_thread::sleep_for(3s);
}

int main()
{
        std::thread worker(worker_thread);

        data = "Example data";
        // send data to the worker thread
        {
                std::lock_guard lk(m);
                ready = true;
                std::cout << "main() signals"
                          << "data ready for processing\n";
        }
        cv.notify_one();

        // wait for the worker
        {
                std::unique_lock lk(m);
                cv.wait(lk, []{ return processed; });
        }
        std::cout << "Back in main(), data = " << data << '\n';

        worker.join();
}
```

Listing 3.7: Primjer korištenja uvjetne varijable [1]

Klase `std::condition_variable` i `std::condition_variable_any` imaju isto ponašanje, ali klasa `std::condition_variable_any` je fleksibilnija po tome što može biti povezana sa bilo kojim tipom lokota.

Objekti tipa `std::condition_variable` i `std::condition_variable_any` ne mogu se kopirati niti premještati.

Klase `std::future` i `std::promise`

Klase `std::future` i `std::promise` omogućavaju slanje podataka između dvije programske niti kroz zajedničku memorijsku lokaciju. Rezultat izračunavanja koje vrši neka druga programska nit pohranjuje se u `std::promise` objektu. Tom rezultatu se može pristupiti kroz `std::future` objekt nakon što programska nit završi izračunavanje.

Vrijednost rezultata postavlja se u `std::promise` objekt pomoću metode `set_value()`. Ova metoda može se pozvati samo jednom, inače izbacuje izuzetak.

`std::future` objekt kreira se tako da se prvo kreira `std::promise` objekt i na njemu se pozove metoda `get_future()` koja vraća odgovarajući `std::future` objekt. Vrijednost rezultata dobiva se tako da se na `std::future` objektu pozove metoda `get()`. Ako rezultat još nije izračunat, metoda zove metodu `wait()` koja blokira programsku nit sve dok izračunavanje ne završi. Metoda `get()` ne smije se pozvati više od jednom jer prvi poziv uništava dijeljenu varijablu. Zato klasa `std::future` ima metodu `valid()` koja provjerava ima li `std::future` objekt pridruženu dijeljenu varijablu.

Objekti tipa `std::future` i `std::promise` se ne mogu kopirati. Mogu se samo premještati.

```
void do_work(std::promise<int> promise)
{
	std::this_thread::sleep_for(std::chrono::seconds(1));
	promise.set_value(6);
}

int main()
{
	std::promise<int> promise;
	std::future<int> future = promise.get_future();
	std::thread work_thread(do_work, std::move(promise));

	// future::get() will wait until the future has a valid result
	// and retrieves it.
	// wait for result
	std::cout << "result=" << future.get() << '\n';
	work_thread.join(); // wait for thread completion
}
```

Listing 3.8: Primjer korištenja *future* i *promise* objekata [1]

Funkcija `std::async`

Funkcija `std::async` definirana je u zaglavlju ⟨`future`⟩ i omogućuje asinkrono izvršavanje funkcije bez potrebe ručnog kreiranja nove programske niti. Rezultat funkcije će biti

dostupan kroz `std::future` objekt nakon završetka izvršavanja funkcije.

Prvi argument funkcije `std::async` je funkcija koja se poziva. Ako ta funkcija prima neke argumente, njih je također potrebno predati funkciji `std::async`. Argumenti se funkciji `std::async` prenose po vrijednosti. Kako bi se argument mogao prenijeti po referenci, potrebno ga je zamotati u `std::ref` ili `std::cref` objekt.

Funkcija `std::async` može imati dodatni parametar tipa `std::launch` na prvom mjestu preko kojeg se kontrolira hoće li se pozvana funkcija izvršavati u posebnoj programskoj niti ili u trenutnoj programskoj niti. Dodijeljena vrijednost parametra je `std::launch::deferred | std::launch::async` i u tom slučaju implementacija odlučuje o načinu pokretanja funkcije. Ako je vrijednost parametra postavljena na `std::launch::async`, funkcija će se pokrenuti u posebnoj programskoj niti. Ako je vrijednost parametra postavljena na `std::launch::deferred`, izvršavanje funkcije će početi kada se pozove `get()` ili `wait()` na `std::future` objektu i funkcija će se tada izvršavati u trenutnoj programskoj niti.

```
std::mutex m;

struct X
{
	void foo(int i, const std::string& str)
	{
		std::lock_guard<std::mutex> lk(m);
		std::cout << str << ' ' << i << '\n';
	}

	void bar(const std::string& str)
	{
		std::lock_guard<std::mutex> lk(m);
		std::cout << str << '\n';
	}

	int operator()(int i)
	{
		std::lock_guard<std::mutex> lk(m);
		std::cout << i << '\n';
		return i + 10;
	}
};

int main()
{
	X x;
	// Calls (&x)->foo(42, "Hello") with default policy:
	// may print "Hello 42" concurrently or defer execution
```

```
        std::future<void> a1 = std::async(&X::foo, &x, 42, "Hello");
        // Calls x.bar("world!") with deferred policy
        // prints "world!" when a2.get() or a2.wait() is called
        std::future<void> a2 = std::async(std::launch::deferred,
                                          &X::bar, x, "world!");
        // Calls X()(43); with async policy
        // prints "43" concurrently
        std::future<int> a3 = std::async(std::launch::async, X(), 43);
        a2.wait();                     // prints "world!"
        std::cout << a3.get() << '\n'; // prints "53"
} // if a1 is not done at this point,
  // destructor of a1 prints "Hello 42" here
```

Listing 3.9: Primjer korištenja funkcije *async* [1]

3.6 Usporedba sa Qt-om

Kreiranje programskih niti

I u Qt-u i C++-u postoji klasa koja predstavlja jednu programsku nit. To je QThread u Qt-u i `std::thread` (ili `std::jthread`) u C++-u. Međutim, razlikuju se po načinu kreiranja programske niti.

Programsku nit u Qt-u moguće je kreirati na dva načina: instanciranjem klase QThread ili kreiranjem njezine podklase. Instanciranjem klase QThread kod unutar funkcije `QThread::run()` izvršit će se u novoj programskoj niti i pokrenut će se paralelna petlja događaja. Pri kreiranju podklase QThread klase, potrebno je ponovno implementirati funkciju `run()` koja će se izvršavati u novoj programskoj niti. Paralelna petlja događaja pokreće se pozivom funkcije `exec()` unutar funkcije `run()`. Programska nit pokreće se pozivom `QThread::start()` metode.

Kako bi se kreirala programska nit u C++-u, konstruktoru `std::thread` objekta potrebno je kao argument predati funkciju koja će se izvršavati u novoj programskoj niti. Ako ta funkcija prima argumente, njih je također potrebno predati konstruktoru. To je jedini način pokretanja paralelnog koda pomoću klase `std::thread` i njime se ne pokreće paralelna petlja događaja. Programska nit pokreće se kada se kreira `std::thread` objekt.

U Qt-u postojeće programske niti moguće je ponovno iskoristiti za nove zadatke. QThreadPool klasa služi za upravljanje zbirkom QThread-ova za višekratnu upotrebu. U C++-u ne postoji *thread pool* klasa, ali moguće je uključiti neku od biblioteka koja ju implementira ili implementirati vlastitu klasu koristeći postojeće klase `std::thread` i `std::queue`.

Međusobno isključivanje

Klase QMutex u Qt-u i `std::mutex` u C++-u služe za zaštitu zajedničkih podataka između različitih programskih niti. Obje klase funkcioniraju na isti način. Imaju metode `lock()` i `unlock()` za zaključavanje i otključavanje *mutexa*, čime omogućuju da u svakom trenutku samo jedna programska nit može pristupiti zajedničkom podatku. Programska nit treba zaključati *mutex* prije nego pristupa zajedničkom podatku, te ga otključati nakon pristupa. Ako neka druga programska nit pokuša zaključati *mutex* dok je zaključan, ona će biti blokirana sve dok *mutex* ne bude otključan.

Također postoje pomoćne klase za zaključavanje *mutexa* u Qt-u i C++-u. To su klase QMutexLocker u Qt-u i `std::lock_guard` (ili `std::unique_lock`) u C++-u. Te klase funkcioniraju na isti način. Konstruktori klasa kao argument primaju *mutex* koji treba zaključati. U konstruktoru se zaključava *mutex*, a u destruktoru se otključava.

U Qt-u je moguće razlikovati pristup zajedničkim resursima za čitanje od pristupa za pisanje pomoću klase QReadWriteLocker. U C++-u takvu funkcionalnost ima klasa `std::shared_mutex` koja dozvoljava dijeljeni pristup (za čitanje) i ekskluzivni pristup (za pisanje).

Klasa QReadWriteLock ima metode `lockForRead()` i `lockForWrite()` kojima razlikuje pristup za čitanje od pristupa za pisanje. Klasa `std::shared_mutex` ima metode `lock()` i `try_lock()` kojima dodijeljuje ekskluzivni pristup te metode `lock_shared()` i `try_lock_shared()` kojima dodijeljuje dijeljeni pristup.

Pomoćne klase QReadLocker i QWriteLocker služe za zaključavanje QReadWriteLocka te funkcioniraju na isti način kao i QMutexLocker. QReadLocker zaključava za čitanje, a QWriteLocker za pisanje. Pomoću `std::lock_guard` ili `std::unique_lock` zaključava se `shared_mutex` za ekskluzivni pristup, a pomoću `std::shared_lock` objekta zaključava se `shared_mutex` za dijeljeni pristup.

Sinkronizacija programskih niti

Sinkronizacija programskih niti vrši se pomoću uvjetnih varijabli. Qt ima klasu QWaitCondition, a C++ klasu `std::condition_variable`. Obje klase funkcioniraju na isti način, samo se razlikuju imena nekih metoda. Programska nit koja čeka da uvjet postane ispunjen poziva metodu `QWaitCondition::wait()` ili `std::condition_variable::wait()`. Kada uvjet postane ispunjen, poziva se metoda `QWaitCondition::wakeOne()` ili metoda `std::condition_variable::notify_one()` da bi se probudila jedna nasumično odabrana programska nit na čekanju, te metoda `QWaitCondition::wakeAll()` ili metoda `std::condition_variable::notify_all()` da bi se probudile sve programske niti na čekanju.

Mogućnost slanja podataka između programskih niti pomoću *future* i *promise* objekata također funkcionira na sličan način i u Qt-u i u C++-u. Za to služe klase QFuture i

QPromise u Qt-u, te klase `std::future` i `std::promise` u C++-u.

Rezultat funkcije koja je pokrenuta u drugoj programskoj niti preko `std::thread` objekta može se pohraniti u `std::promise` objekt pozivom metode `set_value()`. Nakon što programska nit završi s radom, rezultatu se može pristupiti pozivom metode `get()` na `std::future` objektu.

Kada se funkcija pokrene u drugoj programskoj niti preko `QtConcurrent::run()` metode, rezultat te funkcije dostupan je kroz QFuture objekt. Rezultat funkcije pohranjuje se u QPromise objekt pozivom metode `QPromise::addResult()`. Nakon završetka funkcije, rezultat se može dohvatiti kroz QFuture objekt pozivom metode `QFuture::result()`.

Funkcija `std::async` u C++-u ima sličnu funkcionalnost kao `QtConcurrent::run()`. Kao argument funkcijama `std::async` i `QtConcurrent::run()` se predaje funkcija koja će se izvršavati u posebnoj programskoj niti, a rezultat te funkcije će biti dostupan kroz *future* objekt nakon završetka izvršavanja. Ako ta funkcija prima argumente, njih je također potrebno predati funkciji `std::async` odnosno `run()`. Razlika je u tome što funkcija `std::async` omogućuje asinkrono izvršavanje funkcije u programskoj niti u kojoj je pozvana, a ne nužno u posebnoj programskoj niti. Može primati dodatni parametar tipa `std::launch` preko kojeg se kontrolira hoće li se pozvana funkcija izvršavati u posebnoj programskoj niti ili u trenutnoj programskoj niti.

QFuture i QPromise klase imaju još neke dodatne funkcionalnosti koje nisu dostupne u `std::future` i `std::promise` klasama. U QPromise objekt moguće je pohraniti više rezultata iz jedne funkcije. U tom slučaju svi rezultati mogu se dohvatiti pozivom metode `QFuture::results()`. Također je moguće pauzirati, nastaviti i otkazati izvršavanje funkcije pozivom `QFuture::suspend()`, `QFuture::resume()` i `QFuture::cancel()` metoda.

Komunikacija između programskih niti

Najveća razlika između ova dva sustava za paralelno programiranje je u načinu komunikacije između programskih niti. U Qt-u postoji mehanizam signala i utora koji se koristi za komunikaciju između programskih niti. C++ nema sličan mehanizam, nego programske niti mogu komunicirati samo kroz zajedničku memoriju.

Poglavlje 4

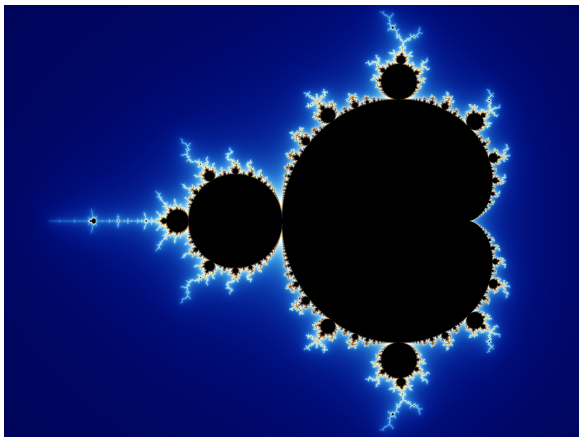
Primjer Mandelbrot

Programiranje s više programskih niti korištenjem Qt-a demonstirat ćemo aplikacijom koja iscrtava Mandelbrotov fraktal te podržava zumiranje i pomicanje fraktala.

Mandelbrotov fraktal je skup točaka c kompleksne ravnine za koje rekurzija

$$z_{n+1} = z_n^2 + c, \quad n \geq 0, \tag{4.1}$$

gdje je $z_0 = 0$, ne teži u beskonačnost kada n teži u beskonačnost. Dakle, točka c pripada Mandelbrotovom skupu ako i samo ako je apsolutna vrijednost od z_n ograničena nekim prirodnim brojem za sve $n \geq 0$.



Slika 4.1: Mandelbrotov fraktal [28]

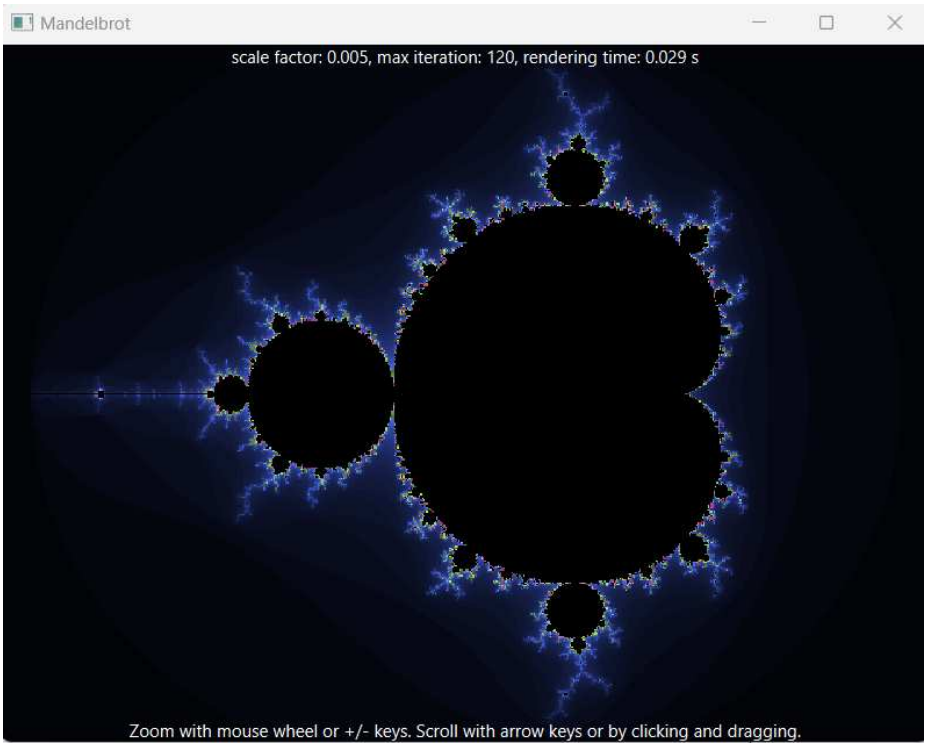
Na slici 4.1 prikazan je Mandelbrotov fraktal. Sve crne točke pripadaju Mandelbrotovom skupu, a sve ostale točke nalaze se izvan Mandelbrotovog skupa. Točke izvan Mandelbrotovog skupa obojane su različitim nijansama plave boje ovisno o brzini kojom rekurzija

4.1 teži u beskonačnost. Točke za koje je potrebno manje iteracija obojane su svjetlijim nijansama, dok su točke za koje je potrebno više iteracija obojane tamnijim nijansama.

Ponekad je potrebno mnogo iteracija kroz formulu 4.1 da bi se ustanovilo pripada li neka točka Mandelbrotovom skupu. Kako bi se ubrzao taj proces, moguće je provjeravati više točaka istovremeno u različitim programskim nitima. U našoj aplikaciji svaka programska nit provjeravat će sve točke s određenom y-koordinatom, odnosno imaginarnim dijelom. To će biti detaljno opisano u odjeljku 4.2 gdje je opisana klasa `Task`.

4.1 Opis aplikacije

Aplikacija je izrađena u Qt Creator-u. Podržava zumiranje korištenjem kotačića miša ili klikom na tipke plus/minus i pomicanje korištenjem miša ili tipka strelica. Na slici 4.2 prikazan je početni izgled aplikacije. Na sredini je prikazana početna slika Mandelbrotovog fraktala. Na vrhu pišu informacije o trenutno prikazanom fraktalu (faktor skaliranja, maksimalni broj iteracija, vrijeme izračunavanja), a na dnu pišu upute za zumiranje i pomicanje fraktala.



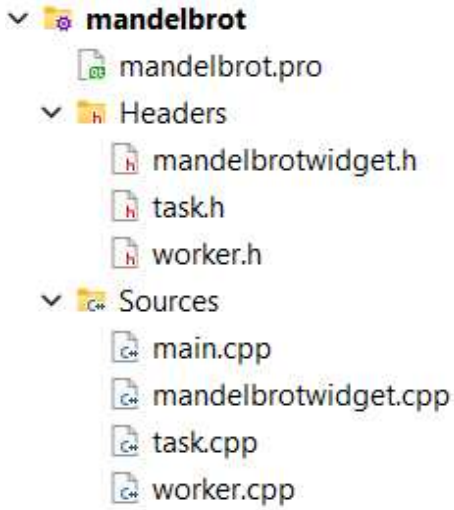
Slika 4.2: Izgled aplikacije

Izračunavanje Mandelbrotovog fraktala izvodi se kroz više radnih programskih niti kako bi se izbjeglo blokiranje petlje događaja glavne programske niti, a time i korisničnog sučelja aplikacije. Svaka radna programska nit vrši izračun za točke s određenom y-koordinatom. Radnim programskim nitima upravlja radni objekt koji živi u posebnoj programskoj niti. Radni objekt skuplja rezultate izračunavanja iz radnih programskih niti te ih prosljeđuje glavnoj programskoj niti kao konačni rezultat. Glavna programska nit služi za prikaz Mandelbrotovog fraktala te obavlja operacije potrebne za zumiranje i pomicanje fraktala.

Sva komunikacija između programskih niti obavlja se preko mehanizma signala i utora. Kada je potrebno generirati novu sliku Mandelbrotovog fraktala, glavna programska nit emitira signal. U radnom objektu se tada izvršava utor u kojem se pomoću QThreadPool-a pokreću radne programske niti. Kada radna programska nit završi izračunavanje, ona emitira signal. Nakon toga se u radnom objektu izvršava utor u kojem se svi rezultati izračunavanja spremaju u konačni rezultat. Kada su svi rezultati izračunati, radni objekt emitira signal, a nakon toga se u glavnoj programskoj niti izvršava utor koji pokreće iscrtavanje Mandelbrotovog fraktala.

4.2 Implementacija

Aplikacija se sastoji od tri klase: `MandelbrotWidget`, `Task` i `Worker`.



Slika 4.3: Struktura datoteka aplikacije

- Klasa `MandelbrotWidget` služi za iscrtavanje i prikaz Mandelbrotovog fraktala te obavlja operacije potrebne za zumiranje i pomicanje fraktala. Traži ponovno generiranje fraktala kada se fraktal zumira ili pomakne.
- Klasa `Worker` obrađuje zahtjev za generiranje fraktala. Upravlja radnim programskim nitima pomoću `QThreadPool`-a, skuplja njihove rezultate izračunavanja te ih prosljeđuje `MandelbrotWidget`-u.
- Klasa `Task` služi za izračunavanje dijela Mandelbrotovog fraktala. Svaki `Task` objekt vrši izračun za točke s određenom y-koordinatom i rezultat prosljeđuje `Worker`-u.

Projektna datoteka

U projektnoj datoteci nalaze se sve bitne informacije vezane za projekt. Ključna riječ `TEMPLATE` opisuje što gradimo. U našem slučaju to je aplikacija. `TARGET` je ime naše aplikacije. Linija `QT += core gui widgets` dodaje potrebne Qt module. Naredbama `SOURCES += main.cpp mandelbrotwidget.cpp task.cpp worker.cpp` i `HEADERS += mandelbrotwidget.h task.h worker.h` dodajemo sve izvorne datoteke i datoteke zaglavlja koje će sudjelovati u izgradnji.

```
1  TEMPLATE = app
2  TARGET = mandelbrot
3
4  greaterThan(QT_MAJOR_VERSION, 5): QT += core gui widgets
5
6  SOURCES += \
7      main.cpp \
8      mandelbrotwidget.cpp \
9      task.cpp \
10     worker.cpp
11
12 HEADERS += \
13     mandelbrotwidget.h \
14     task.h \
15     worker.h
```

Slika 4.4: Projektna datoteka

Glavni program

Naša Mandelbrot aplikacija je Qt Widget aplikacija. Glavni program započinje kreiranjem instance klase `QApplication`. Za svaku aplikaciju postoji točno jedan objekt tipa `QApplication` koji upravlja njenim tijekom kontrole. Konstruktor klase `QApplication`

prima argumente komandne linije i inicijalizira sustav prozora. Nakon toga kreira se instanca klase `MandelbrotWidget`. Naredbom `widget.show();` prikazuje se objekt `MandelbrotWidget`. Naredba `return app.exec();` pokreće glavnu petlju događaja.

```
1  #include "mandelbrotwidget.h"
2
3  #include <QApplication>
4
5  int main(int argc, char *argv[])
6  {
7      QApplication app(argc, argv);
8
9      MandelbrotWidget widget;
10     widget.show();
11     return app.exec();
12 }
```

Slika 4.5: main.cpp datoteka

Klasa Task

Klasa `Task` proširuje klasu `QRunnable` kako bi mogli ponovno implementirati funkciju `run()` koja će se izvršavati u novoj programskoj niti. `Task` također proširuje `QObject` klasu kako bi iz nje mogli emitirati signale. Svaki `Task` objekt izračunava dio točaka Mandelbrotovog fraktala s određenom y-koordinatom. Kada izračunavanje završi, emitira se signal `taskCompleted()`. Varijabla `y` predstavlja y-koordinatu točaka koje se računaju. Varijabla `origin` predstavlja gornji lijevi kut Mandelbrotovog fraktala. Varijable `scaleFactor`, `size` i `maxIterations` označuju faktor skaliranja, veličinu fraktala te maksimalni mogući broj iteracija koji ćemo proći za svaku točku. Konstruktor klase `Task` prima argumente potrebne za izračunavanje Mandelbrotovog fraktala. Varijable članice inicijaliziraju se vrijednostima tih argumenata.

```
10 class Task : public QObject, public QRunnable
11 {
12     Q_OBJECT
13 public:
14     explicit Task(int y_, QPointF origin_, double scaleFactor_, QSize size_,
15                   int maxIterations_, QObject *parent = nullptr);
16
17 signals:
18     void taskCompleted(const int &y, const QVector<int> &result, const QPointF &origin_,
19                        const double &scaleFactor_, const QSize &resultSize_);
20
21 protected:
22     void run() override;
23
24 private:
25     int y;
26     QPointF origin;
27     double scaleFactor;
28     QSize size;
29     int maxIterations;
30 };
```

Slika 4.6: task.h datoteka

```
3  Task::Task(int y_, QPointF origin_, double scaleFactor_, QSize size_,
4             int maxIterations_, QObject *parent) :
5      QObject(parent),
6      y(y_),
7      origin(origin_),
8      scaleFactor(scaleFactor_),
9      size(size_),
10     maxIterations(maxIterations_)
11 {}
```

Slika 4.7: Konstruktor klase `Task`

U funkciji `run()` inicijaliziramo vektor rezultata. Varijable `x` i `y` određuju poziciju piksela koju pretvaramo u koordinate kompleksne ravnine. Dakle, varijable `cx` i `cy` su prave vrijednosti koordinata točaka. Za svaku točku računamo rekurziju 4.1 sve dok ne dođemo do maksimalnog broja iteracija. Ako je apsolutna vrijednost trenutnog člana rekurzije prešla fiksnu granicu (u ovom slučaju 4), prekidamo računanje jer smo time ustanovili da točka nije dio Mandelbrotovog skupa. Broj iteracija koje su bile potrebne da bi se to ustanovilo spremamo u vektor rezultata. Ako smo došli do maksimalog broja iteracija, točka je dio Mandelbrotovog skupa i u tom slučaju u vektor rezultata spremamo maksimalni broj iteracija. Kada prođemo kroz sve točke, emitiramo signal `taskCompleted()` preko kojeg `Worker`-u prosljeđujemo vektor rezultata.

```
13  void Task::run()
14  {
15      const int width = size.width();
16      QVector<int> result(width);
17      const double cy = origin.y() + y * scaleFactor;
18
19      for (int x = 0; x < width; ++x) {
20          const double cx = origin.x() + x * scaleFactor;
21          double x1 = cx;
22          double y1 = cy;
23          int numIterations = 0;
24
25          do {
26              ++numIterations;
27              const double nextX = (x1 * x1) - (y1 * y1) + cx;
28              const double nextY = (2 * x1 * y1) + cy;
29              x1 = nextX;
30              y1 = nextY;
31              if ((x1 * x1) + (y1 * y1) > 4)
32                  break;
33          } while (numIterations < maxIterations);
34
35          result[x] = numIterations;
36      }
37
38      emit taskCompleted(y, result, origin, scaleFactor, size);
39  }
```

Slika 4.8: Funkcija `run()`

Klasa Worker

Klasa `Worker` proširuje klasu `QObject` kako bi u njoj mogli implementirati signale i utore. Ova klasa upravlja `Task`-ovima preko QThreadPool-a, skuplja njihove rezultate i prosljeđuje ih `MandelbrotWidget`-u. Kada `MandelbrotWidget` zatraži generiranje slike Mandelbrotovog fraktala, poziva se utor `generateMandelbrot()` u kojem se pokreću `Task`-ovi u novim programskim nitima. Kada `Task` završi izračunavanje, poziva se utor `processResult()` u kojem skupljamo rezultate iz svih programskih niti. Kada dobijemo sve rezultate, emitira se signal `mandelbrotGenerated()`. Varijable članice `origin`, `scaleFactor`, `resultSize` i `maxIteration` označuju redom gornji lijevi kut slike, faktor skaliranja, veličinu slike i maksimalni broj iteracija za provjeru točaka. Vektor `results` sadrži vektore rezultata iz svih programskih niti, a `numReceivedResults` broji koliko je rezultata primljeno. `timer` koristimo da bismo izračunali vrijeme potrebno za izračunavanje Mandelbrotovog fraktala.

```
class Worker : public QObject
{
    Q_OBJECT

public:
    explicit Worker(QObject *parent = nullptr);
    ~Worker();

signals:
    void mandelbrotGenerated(const QVector<QVector<int>> &results, const int &maxIteration,
                             const double &time);

public slots:
    void generateMandelbrot(QPointF origin_, double scaleFactor_, QSize resultSize_);
    void processResult(const int &y, const QVector<int> &result, const QPointF &origin_,
                       const double &scaleFactor_, const QSize &resultSize_);

private:
    QPointF origin;
    double scaleFactor;
    QSize resultSize;
    int maxIterations;
    QVector<QVector<int>> results;
    int numReceivedResults;
    QElapsedTimer timer;
};
```

Slika 4.9: worker.h datoteka

```
Worker::Worker(QObject *parent) :
    QObject(parent)
{}

Worker::~Worker()
{
    QThreadPool::globalInstance()->clear();
    QThreadPool::globalInstance()->waitForDone();
}
```

Slika 4.10: Konstruktor i destruktor `Worker` klase

U konstruktoru ne radimo ništa, a u destruktoru mičemo iz reda čekanja sve `Task`-ove koji još nisu započeli s radom i čekamo da završe oni koji su već počeli.

Utor `generateMandelbrot()` poziva se kada `MandelbrotWidget` zatraži generiranje slike Mandelbrotovog fraktala, to jest kada se slika zumira ili pomakne. Vrijednosti argumenata prosljeđenih iz `MandelbrotWidget`-a spremamo u varijable članice te određujemo maksimalni broj iteracija ovisno o `scaleFactor`-u. Što je slika više zumirana, postavljamo veći maksimalni broj iteracija kako bi slika bila detaljnija. Nakon toga mičemo iz

reda čekanja sve `Task`-ove koji još nisu započeli s radom i čekamo da završe oni koji su već počeli. Također resetiramo brojač koji broji primljene rezultate, inicijaliziramo novi vektor rezultata i pokrećemo `timer`. Za svaki `y` kreiramo novi `Task` s potrebnim parametrima, spajamo signal `taskCompleted()` iz kreiranog `Task` objekta sa utorom `processResult()` iz `Worker` klase te pokrećemo `Task` u novoj programskoj niti.

```
15  void Worker::generateMandelbrot(QPointF origin_, double scaleFactor_,
16                                  QSize resultSize_)
17  {
18      origin = origin_;
19      scaleFactor = scaleFactor_;
20      resultSize = resultSize_;
21
22      const int height = resultSize.height();
23      const int width = resultSize.width();
24      maxIterations = 100 + 1 / (10 * scaleFactor);
25
26      QThreadPool::globalInstance()->clear();
27      QThreadPool::globalInstance()->waitForDone();
28      numReceivedResults = 0;
29      results = QVector<QVector<int>>(height);
30
31      timer.start();
32
33      for (int y = 0; y < height; ++y) {
34          results[y] = QVector<int>(width);
35          Task *task = new Task(y, origin, scaleFactor, resultSize,
36                                maxIterations, this);
37          connect(task, &Task::taskCompleted, this, &Worker::processResult);
38          QThreadPool::globalInstance()->start(task);
39      }
40  }
```

Slika 4.11: Utor `generateMandelbrot()`

Kada `Task` završi izračunavanje i emitira signal `taskCompleted()`, poziva se utor `processResult()` koji obrađuje rezultate. Najprije provjeravamo jesu li vrijednosti argumenata prosljeđenih iz `Task` objekta jednake vrijednostima spremljenima u varijable članice. Ako je utor `generateMandelbrot()` bio pozvan prije nego su obrađeni svi rezultati od prethodnog zahtjeva, vrijednosti varijabli će se razlikovati i neće biti potrebno obraditi taj rezultat. Ako je potrebno obraditi rezultat, povećavamo brojač primljenih rezultata i spremamo dobiveni rezultat u vektor rezultata. Ako smo primili sve rezultate, zaustavljamo `timer` i emitiramo signal `mandelbrotGenerated()`.

```
42  void Worker::processResult(const int &y, const QVector<int> &result,
43                             const QPointF &origin_, const double &scaleFactor_,
44                             const QSize &resultSize_)
45  {
46      if (origin != origin_ ||
47          scaleFactor != scaleFactor_ ||
48          resultSize != resultSize_) {
49          return;
50      }
51
52      ++numReceivedResults;
53      results[y] = result;
54
55      if (numReceivedResults == resultSize.height()) {
56          const auto time = static_cast<double>(timer.elapsed()) / 1000;
57          emit mandelbrotGenerated(results, maxIterations, time);
58      }
59  }
```

Slika 4.12: Utor `processResult()`

Klasa `MandelbrotWidget`

Klasa `MandelbrotWidget` proširuje klasu `QWidget`. Služi za iscrtavanje i prikaz Mandelbrotovog fraktala. Signal `requestImage()` emitira se kada je potrebno generirati sliku Mandelbrotovog fraktala zbog zumiranja ili pomicanja slike. Kada `Worker` skupi rezultate izračunavanja Mandelbrotovog fraktala iz svih programskih niti i emitira signal `mandelbrotGenerated()`, poziva se utor `updateImage()` koji pokreće ponovno iscrtavanje slike. U ovoj klasi ponovno implementiramo funkcije za obradu događaja (eng. *event handlers*) iz klase `QWidget`. Funkcije `zoom()` i `scroll()` su pomoćne funkcije za zumiranje i pomicanje slike koje se pozivaju iz funkcija za obradu događaja. Funkcija `generateColorFromIteration()` generira boju piksela ovisno o pripadnom broju iteracija. Među privatnim varijablama nalaze se instanca klase `Worker` te instanca klase QThread koja predstavlja programsku nit u kojoj će se izvršavati `Worker`. `mandelbrotImage` je slika izračunatog Mandelbrotovog fraktala, `lastMousePosition` određuje posljednju poziciju kursora miša pri pomicanju slike, `origin` i `scaleFactor` označuju gornji lijevi kut slike i faktor skaliranja slike, a stringovi `help` i `info` sadržavaju upute za zumiranje i pomicanje slike te informacije o trenutno prikazanoj slici.

Stavljanjem `Worker`-a u posebnu programsku nit sve signal-utor veze između `Worker`-a i `MandelbrotWidget`-a biti će veze u redu čekanja. Time je omogućeno da se prilikom zumiranja i pomicanja slike izračunavanje Mandelbrotovog fraktala izvrši bez blokiranja glavne programske niti i korisničkog sučelja aplikacije. Kada bi `Worker` bio u

glavnoj programskoj niti, upravljanje `QThreadPool`-om i procesiranje rezultata iz `Task`-ova odvijalo bi se u glavnoj programskoj niti, čime bi se blokirala glavna petlja događaja dok izračunavanje Mandelbrotovog fraktala ne završi.

```
class MandelbrotWidget : public QWidget
{
    Q_OBJECT

public:
    explicit MandelbrotWidget(QWidget *parent = nullptr);
    ~MandelbrotWidget();

signals:
    void requestImage(QPointF origin_, double scaleFactor_, QSize resultSize_);

public slots:
    void updateImage(const QVector<QVector<int>> &results, const int &maxIteration,
                     const double &time);

protected:
    void paintEvent(QPaintEvent *event) override;
    void resizeEvent(QResizeEvent *event) override;
    void wheelEvent(QWheelEvent *event) override;
    void mousePressEvent(QMouseEvent *event) override;
    void mouseMoveEvent(QMouseEvent *event) override;
    void keyPressEvent(QKeyEvent *event) override;

private:
    QRgb generateColorFromIteration(int iteration, int maxIteration);
    void zoom(double zoomFactor);
    void scroll(int deltaX, int deltaY);

    Worker mandelbrotWorker;
    QThread mandelbrotThread;
    QImage mandelbrotImage;
    QPoint lastMousePosition;
    QPointF origin;
    double scaleFactor;
    QString help;
    QString info;
};
```

Slika 4.13: mandelbrotwidget.h datoteka

Na početku definiramo konstante koje će nam trebati. U konstruktoru inicijaliziramo varijablu članicu `scaleFactor` i string `help`. `Worker` objekt premještamo u novu programsku nit te povezujemo signal `requestImage()` iz `MandelbrotWidget` klase sa utorom `generateMandelbrot()` iz `Worker` objekta i signal `mandelbrotGenerated()` iz

`Worker` objekta sa utorom `updateImage()` iz `MandelbrotWidget` klase. Na kraju pokrećemo programsku nit u kojoj će se izvršavati `Worker`. U destruktoru zaustavljamo programsku nit u kojoj se izvršava `Worker`.

```
 8  constexpr double DefaultScale = 0.005;
 9  constexpr double ZoomInFactor = 0.8;
10  constexpr double ZoomOutFactor = 1 / ZoomInFactor;
11  constexpr int ScrollStep = 20;
12
13  MandelbrotWidget::MandelbrotWidget(QWidget *parent) :
14      QWidget(parent),
15      scaleFactor(DefaultScale)
16  {
17      help = QString("Zoom with mouse wheel or +/- keys. "
18                     "Scroll with arrow keys or by clicking and dragging.");
19      setWindowTitle(QString("Mandelbrot"));
20
21      mandelbrotWorker.moveToThread(&mandelbrotThread);
22      connect(this, &MandelbrotWidget::requestImage,
23              &mandelbrotWorker, &Worker::generateMandelbrot);
24      connect(&mandelbrotWorker, &Worker::mandelbrotGenerated,
25              this, &MandelbrotWidget::updateImage);
26
27      mandelbrotThread.start();
28  }
29
30  MandelbrotWidget::~MandelbrotWidget()
31  {
32      mandelbrotThread.quit();
33      mandelbrotThread.wait();
34  }
```

Slika 4.14: Konstruktor i destruktor klase `MandelbrotWidget`

Nakon što `Worker` emitira signal `mandelbrotGenerated()`, poziva se utor `updateImage()`. U utoru Inicijaliziramo novu sliku Mandelbrotovog fraktala i za svaki piksel slike generiramo boju pozivom funkcije `generateColorFromIteration()`. Na kraju pozivamo metodu `QWidget::update()` koja pokreće iscrtavanje nove slike Mandelbrotovog fraktala pozivajući `paintEvent()`.

Funkcija `generateColorFromIteration()` služi za generiranje boje piksela ovisno o broju iteracija koje su bile potrebne da bi se ustanovilo je li odgovarajuća točka dio Mandelbrotovog skupa. Ako je broj iteracija manji od `maxIteration`, točka je izvan Mandelbrotovog skupa i bojamo piksel u boju ovisno o broju iteracija. Inače je točka unutar Mandelbrotovog skupa i bojamo piksel u crno.

```
36  void MandelbrotWidget::updateImage(const QVector<QVector<int>> &results,
37                                     const int &maxIteration, const double &time)
38  {
39      info = QString("scale factor: %1, max iteration: %2, rendering time: %3 s")
40                 .arg(scaleFactor).arg(maxIteration).arg(time);
41
42      mandelbrotImage = QImage(size(), QImage::Format_RGB32);
43      mandelbrotImage.setDevicePixelRatio(devicePixelRatio());
44
45      for (int y = 0; y < height(); ++y)
46          for (int x = 0; x < width(); ++x)
47              mandelbrotImage.setPixelColor(x, y,
48                                            generateColorFromIteration(results[y][x],
49                                                                       maxIteration));
50      update();
51  }
```

Slika 4.15: Utor `updateImage()`

```
53  QRgb MandelbrotWidget::generateColorFromIteration(int iteration, int maxIteration)
54  {
55      int r = 0;
56      int g = 0;
57      int b = 0;
58
59      if (iteration < maxIteration) {
60          r = 255 * iteration / maxIteration;
61          g = static_cast<int>(255 * iteration / (0.7 * maxIteration)) % 256;
62          b = static_cast<int>(255 * iteration / (0.3 * maxIteration)) % 256;
63      }
64
65      return qRgb(r, g, b);
66  }
```

Slika 4.16: Metoda `generateColorFromIteration()`

U `paintEvent()` funkciji bojamo pozadinu u crno i iscrtavamo sliku izračunatog Mandelbrotovog fraktala. Također ispisujemo informacije o slici (faktor skaliranja, maksimalni broj iteracija, potrebno vrijeme izračunavanja) te upute kako zumirati ili pomaknuti sliku.

```
68  void MandelbrotWidget::paintEvent(QPaintEvent *event)
69  {
70      QPainter painter(this);
71      painter.fillRect(rect(), Qt::black);
72
73      painter.drawImage(event->rect(), mandelbrotImage);
74
75      if (!info.isEmpty()) {
76          painter.setPen(Qt::white);
77          painter.drawText(rect(),
78                           Qt::AlignHCenter|Qt::AlignTop|Qt::TextWordWrap,
79                           info);
80      }
81
82      painter.setPen(Qt::white);
83      painter.drawText(rect(),
84                       Qt::AlignHCenter|Qt::AlignBottom|Qt::TextWordWrap,
85                       help);
86  }
```

Slika 4.17: Metoda `paintEvent()`

```
88  void MandelbrotWidget::resizeEvent(QResizeEvent *event)
89  {
90      QSize oldSize = event->oldSize();
91      if ((oldSize.width() > 0) & (oldSize.height() > 0)) {
92          QPointF center = origin +
93                           QPointF(oldSize.width()/2 * scaleFactor,
94                                   oldSize.height()/2 * scaleFactor);
95
96          origin = center - QPointF(width()/2 * scaleFactor,
97                                    height()/2 * scaleFactor);
98      }
99      else
100         origin = QPointF(-width()/2 * scaleFactor - 0.5,
101                          -height()/2 * scaleFactor);
102
103     emit requestImage(origin, scaleFactor, size());
104 }
```

Slika 4.18: Metoda `resizeEvent()`

Funkcija `resizeEvent()` poziva se kada se pokrene aplikacija i prozor je prvi put prikazan te kada korisnik promijeni veličinu prozora. Ako je prozor prvi puta prikazan, `oldSize` će imati nevaljane dimenzije $(-1, -1)$ i tada inicijaliziramo varijablu `origin` na gornji lijevi kut. Inače, ako `oldSize` ima valjane dimenzije, računamo `origin` u novim dimenzijama tako da slika i dalje bude jednako centrirana. Na kraju emitiramo signal `requestImage()` s novom vrijednosti varijable `origin`.

Funkcija `wheelEvent()` poziva se kada korisnik pomakne kotačić miša. Ako je kotačić pomaknut u pozitivnom smjeru, pozivamo metodu `zoom()` s argumentom `ZoomInFactor`. Inače, ako je kotačić pomaknut u negativnom smjeru, pozivamo metodu `zoom()` s argumentom `ZoomOutFactor`.

```
106 void MandelbrotWidget::wheelEvent(QWheelEvent *event)
107 {
108     const int delta = event->angleDelta().y();
109     double zoomFactor = (delta > 0) ? ZoomInFactor : ZoomOutFactor;
110     zoom(zoomFactor);
111 }
```

Slika 4.19: Metoda `wheelEvent()`

```
113 void MandelbrotWidget::mousePressEvent(QMouseEvent *event)
114 {
115     if (event->button() == Qt::LeftButton) {
116         lastMousePosition = event->position().toPoint();
117     }
118 }
119
120 void MandelbrotWidget::mouseMoveEvent(QMouseEvent *event)
121 {
122     if (event->buttons() & Qt::LeftButton) {
123         QPoint delta = lastMousePosition - event->position().toPoint();
124         lastMousePosition = event->position().toPoint();
125         scroll(delta.x(), delta.y());
126     }
127 }
```

Slika 4.20: Metode `mousePressEvent()` i `mouseMoveEvent()`

Funkcija `mousePressEvent()` poziva se kada korisnik klikne tipku miša. Ako je kliknuta lijeva tipka miša, računamo novu poziciju kursora miša.

Funkcija `mouseMoveEvent()` poziva se kada korisnik pomakne kursor miša. Ako je lijeva tipka miša kliknuta dok se kursor miša pomiče, računamo koliko je kursor miša pomaknut i pozivamo metodu `scroll()`.

```
129 void MandelbrotWidget::keyPressEvent(QKeyEvent *event)
130 {
131     switch (event->key()) {
132     case Qt::Key_Plus:
133         zoom(ZoomInFactor);
134         break;
135     case Qt::Key_Minus:
136         zoom(ZoomOutFactor);
137         break;
138     case Qt::Key_Left:
139         scroll(-ScrollStep, 0);
140         break;
141     case Qt::Key_Right:
142         scroll(+ScrollStep, 0);
143         break;
144     case Qt::Key_Up:
145         scroll(0, -ScrollStep);
146         break;
147     case Qt::Key_Down:
148         scroll(0, +ScrollStep);
149         break;
150     default:
151         QWidget::keyPressEvent(event);
152     }
153 }
```

Slika 4.21: Metoda `keyPressEvent()`

Funkcija `keyPressEvent()` poziva se kada korisnik klikne tipku na tipkovnici. Za tipke plus i minus pozivamo metodu `zoom()` s argumentom `ZoomInFactor`, odnosno `ZoomOutFactor`. Za tipke strelice pozivamo `scroll()` s argumentima `-ScroolStep`, `+ScrollStep` i 0, ovisno u kojem smjeru želimo pomaknuti sliku.

U metodi `zoom()` računamo nove vrijednosti varijabli članica `scaleFactor` i `origin` tako da zumirana slika i dalje bude jednako centrirana. Nakon toga emitiramo signal `requestImage()` s novim vrijednostima varijabli `scaleFactor` i `origin`.

U metodi `scroll()` računamo novu vrijednost varijable `origin` i emitiramo signal `requestImage()` s novom vrijednosti varijable `origin`.

```
155 void MandelbrotWidget::zoom(double zoomFactor)
156 {
157     QPointF center = origin + QPointF(width()/2 * scaleFactor,
158                                       height()/2 * scaleFactor);
159     scaleFactor *= zoomFactor;
160     origin = center - QPointF(width()/2 * scaleFactor,
161                               height()/2 * scaleFactor);
162     emit requestImage(origin, scaleFactor, size());
163 }
164
165 void MandelbrotWidget::scroll(int deltaX, int deltaY)
166 {
167     origin += QPointF(deltaX * scaleFactor, deltaY * scaleFactor);
168     emit requestImage(origin, scaleFactor, size());
169 }
```

Slika 4.22: Metode `zoom()` i `scroll()`

Zaključak

Sustav za paralelno programiranje s programskim nitima u Qt `frameworku` za izradu grafičkih korisničkih sučelja vrlo je sličan sustavu za paralelno programiranje u standardnoj biblioteci C++20, ali ima i neke dodatne mogućnosti. Qt-ov sustav događaja zajedno s mehanizmom signala i utora pruža učinkovit način komunikacije između različitih programskih niti. Qt-ova podrška za programske niti omogućuje razvoj aplikacija s programskim nitima koje zahtjevaju izvođenje dugotrajnih operacija bez zamrzivanja korisničkog sučelja aplikacije. Korištenjem sekundarnih programskih niti koje će vršiti izračun dok glavna programska nit služi samo za prikaz grafičkog korisničkog sučelja izbjegava se blokiranje petlje događaja glavne programske niti, a time i korisničkog sučelja aplikacije.

Bibliografija

[1] *cppreference, Concurrency support library*, https://en.cppreference.com/w/cpp/thread, (siječanj 2025).

[2] M. Jurak, *Sinhronizacija programskih niti*, https://web.math.pmf.unizg.hr/nastava/ppr/html/Cpp/sinhronizacija.html, (siječanj 2025).

[3] M. Jurak, *Programske niti*, https://web.math.pmf.unizg.hr/nastava/ppr/html/Cpp/programske_niti.html, (siječanj 2025).

[4] M. Jurak, *Zajednički resursi*, https://web.math.pmf.unizg.hr/nastava/ppr/html/Cpp/zajednicki_resursi.html, (siječanj 2025).

[5] G. Lazar i R. Penea, *Mastering Qt 5 - Second Edition*, Packt Publishing Ltd, 2018.

[6] *Qt Documentation, Introduction to Qt*, https://doc.qt.io/qt-6/qt-intro.html, (rujan 2024).

[7] *Qt Documentation, Mandelbrot*, https://doc.qt.io/qt-6/qtcore-threads-mandelbrot-example.html, (studeni 2024).

[8] *Qt Documentation, Multithreading Technologies in Qt*, https://doc.qt.io/qt-6/threads-technologies.html, (listopad 2024).

[9] *Qt Documentation, qmake Manual*, https://doc.qt.io/qt-6/qmake-manual.html, (rujan 2024).

[10] *Qt Documentation, qmake Overview*, https://doc.qt.io/qt-6/qmake-overview.html, (rujan 2024).

[11] *Qt Documentation, QMutex Class*, https://doc.qt.io/qt-6/qmutex.html, (siječanj 2025).

[12] *Qt Documentation, QMutexLocker Class*, https://doc.qt.io/qt-6/qmutexlocker.html, (siječanj 2025).

[13] *Qt Documentation, QReadLocker Class*, https://doc.qt.io/qt-6/qreadlocker.html, (siječanj 2025).

[14] *Qt Documentation, QReadWriteLock Class*, https://doc.qt.io/qt-6/qreadwritelock.html, (siječanj 2025).

[15] *Qt Documentation, QSemaphore Class*, https://doc.qt.io/qt-6/qsemaphore.html, (siječanj 2025).

[16] *Qt Documentation, Qt Concurrent*, https://doc.qt.io/qt-6/qtconcurrent-index.html, (siječanj 2025).

[17] *Qt Documentation, QThread Class*, https://doc.qt.io/qt-6/qthread.html, (siječanj 2025).

[18] *Qt Documentation, QThreadPool Class*, https://doc.qt.io/qt-6/qthreadpool.html, (siječanj 2025).

[19] *Qt Documentation, QWaitCondition Class*, https://doc.qt.io/qt-6/qwaitcondition.html, (listopad 2024).

[20] *Qt Documentation, QWriteLocker Class*, https://doc.qt.io/qt-6/qwritelocker.html, (siječanj 2025).

[21] *Qt Documentation, Signals and Slots*, https://doc.qt.io/qt-6/signalsandslots.html, (kolovoz 2024).

[22] *Qt Documentation, Synchronizing Threads*, https://doc.qt.io/qt-6/threads-synchronizing.html, (listopad 2024).

[23] *Qt Documentation, Thread Support in Qt*, https://doc.qt.io/qt-6/threads.html, (listopad 2024).

[24] *Qt Documentation, User Interfaces*, https://doc.qt.io/qt-6/topics-ui.html, (rujan 2024).

[25] *Qt Documentation, WorkerScript QML Type*, https://doc.qt.io/qt-6/qml-qtqml-workerscript-workerscript.htmll, (listopad 2024).

[26] *Qt Wiki, About Qt*, https://wiki.qt.io/About_Qt, (rujan 2024).

[27] *Qt Wiki, Qt History*, https://wiki.qt.io/Qt_History, (rujan 2024).

[28] *Wikipedia, Mandelbrot set*, https://en.wikipedia.org/wiki/Mandelbrot_set, (prosinac 2024).

Sažetak

U ovom diplomskom radu obrađen je Qt *framework* za izradu grafičkih korisničkih sučelja i aplikacija na više platformi. Opisane su neke od njegovih karakteristika i funkcionalnosti, kao što je mehanizam signala i utora koji služi za komunikaciju između različitih grafičkih komponenti. Glavni dio rada fokusiran je na podršku za paralelno programiranje s programskim nitima u Qt-u. Qt sadrži brojne klase i mehanizme za rad s programskih nitima. U ovom radu također je opisan sustav za paralelno programiranje u standardnoj C++ biblioteci i uspoređen sa sustavom za paralelno programiranje u Qt-u. Glavna razlika je u tome što Qt ima mehanizam signala i utora koji može koristiti za komunikaciju između različitih programskih niti. Na primjeru iscrtavanja Mandelbrotovog fraktala s mogućnošću zumiranja i pomicanja, prikazano je kako se može implementirati aplikacija sa programskim nitima u Qt-u. Glavna programska nit služi samo za prikaz grafičkog korisničkog sučelja, dok sporedne programske niti vrše izračunavanje Mandelbrotovog fraktala kako bi se izbjeglo blokiranje korisničkog sučelja aplikacije.

Summary

This thesis describes Qt framework for creating graphical user interfaces and cross-platform applications. Some of its described characteristics and functionalities include signals and slots mechanism used for communication between different graphical components. The main part of the thesis focuses on support for parallel programming in Qt. Qt consists of many classes and mechanisms for working with multiple threads. The thesis also describes parallel programming system in the standard C++ library and compares it to parallel programming system in Qt. The main difference is that Qt contains signals and slots mechanism which can be used for communication between different threads. The example of rendering the Mandelbrot fractal with zoom and scroll functionality demonstrates how an application with threads can be implemented in Qt. The main thread is used only for displaying the graphical user interface, while the secondary threads perform the calculation of the Mandelbrot fractal to avoid blocking the application's user interface.

Životopis

Dorotea Popović rođena je 5. travnja 1998. godine u Zagrebu. Pohađala je Osnovnu školu Ljube Babića u Jastrebarskom. Nakon završenog osnovnoškolskog obrazovanja, 2013. godine upisuje Srednju školu Jastrebarsko, smjer opća gimnazija. 2017. godine završava srednjoškolsko obrazovanje i upisuje preddiplomski sveučilišni studij Matematika; smjer: nastavnički na Prirodoslovno-matematičkom fakultetu u Zagrebu. Nakon uspješnog završetka preddiplomskog studija, 2021. godine upisuje diplomski sveučilišni studij Računarstvo i matematika na istom fakultetu. Tijekom posljednje godine studija radi u tvrtci *Ericsson Nikola Tesla* na poziciji softver developera.