Programiranje video igara u biblioteci Qt5

Pužar, Sara

Master's thesis / Diplomski rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:787247

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-23**

Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Sara Pužar

PROGRAMIRANJE VIDEO IGARA U BIBLIOTECI QT5

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, veljača, 2020

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. _____

Hvala roditeljima i seki na neiscrpnoj podršci i ohrabrivanju koje su mi pružili kroz godine studiranja.
Hvala dečku, prijateljima i društvu iz praktikuma na svim suzama i zabavama kojima su mi ispunili dane.
Hvala mentoru, prof. dr. sc. Mladenu Juraku, na pomoći i odgovorima na mojih bezbroj pitanja.

Sadržaj

Sadržaj **iv**

Uvod **1**

1 Qt 5 **2**
1.1 Meta-Object sustav 3
1.2 Signali i utori 7
1.3 Qt Widgets 10
1.4 Qt Quick 11
1.5 Infrastruktura za programiranje 2D igara 13

2 Oblikovni obrasci **25**
2.1 Osnovna podjela oblikovnih obrazaca 25
2.2 Oblikovni obrasci u razvoju programskih igara 26

3 Implementacija video igre **34**
3.1 Pregled igre 34
3.2 Struktura aplikacije 37
3.3 Funkcionalnosti igre 38

4 Zaključak **50**

Bibliografija **51**

Uvod

Qt je alat za razvijanje grafičkih sučelja i aplikacija. Prva verzija Qt-a objavljena je u svibnju 1995., a nastala je iz potrebe Haavard Norda i Eirik Chambe-Enga za izradom više-platformske (engl. *cross-platform*) C++ aplikacije za skladištenje ultrasound slika [4]. Danas je aktualna najnovija Qt5 verzija koja u odnosu na svoje prethodniku nudi nove funkcionalnosti i optimizirane performanse [9]. Iako primarna uloga Qt-a nije izrada video igara, s obzirom na sve veću popularnost ovog oblika zabave, Qt se počeo koristiti u svrhu razvoja igara [11]. Qt5 nudi nekoliko klasa i mehanizama koji olakšavaju implementaciju igara, čije će se mogućnosti istražiti i predstaviti u ovom radu.

Danas gotovo trećina svjetske populacije za sebe tvrdi kako igraju igre, a iako je uvriježeno mišljenje kako su igre za djecu, prosječna dob ljudi koji ih igraju je 34 godine [13]. Popularnosti igara doprinijela je i svestranost platformi na kojima se igre igraju, a industrija video igara toliko je narasla da nadmašuje filmsku i glazbenu industriju [17]. Uspjeh mobilnih igara svakom godinom raste te se približava uspjehu konzolnih i računalnih igara [14].

Ovaj rad je podijeljen u tri osnovna dijela. U prvom dijelu bit će predstavljen Qt, njegove bitne karakteristike te podrška za izradu video igara. U drugom dijelu objašnjava se pojam oblikovnih obrazaca, njihova uporaba u razvoju video igara te poveznica sa Qt5. U posljednjem poglavlju predstavljena je video igra, *The Camel Run*, koja je napravljena u sklopu ovog rada, koristeći Qt 5.13.1. biblioteku.

Poglavlje 1

Qt 5

Qt je *framework* namijenjen izradi grafičkih korisničkih sučelja i aplikacija prilagođenih različitim operacijskim sustavima. Razvijen je u programskom jeziku C++, te se najčešće koristi kod aplikacija rađenih u tom jeziku. Uz C++, Qt5 pruža mogućnost povezivanja i s drugim programskim jezicima kao što je python. Qt5 podržava vrlo jednostavno korištenje standarda C++11 i C++14, a omogućeno je i osposobljavanje standarda C++17 [18]. Uz podršku za izradu grafičkih sučelja, Qt pruža alate za razna druga područja kao što su paralelizam i mrežna komunikacija [5]. 2018. godine najavljena je objava Qt6 biblioteke za kraj 2020. godine [8] .

Verzije i instalacija

Qt framework održava *The Qt Company* te je dostupan pod različitim licencama:

1. Komercijalna verzija
   a) namijenjena izradi komercijalnih proizvoda
   b) dostupni su svi moduli
   c) Qt programska potpora za korisnike
2. Qt otvorenog koda (engl. *Open-source*)
   a) ograničena dostupnost modula
   b) obaveza korištenja jedne od sljedećih licenci: LPGL v3, GPL v2 ili GPL v3.

Instalacija Qt-a sastoji se od svega nekoliko koraka te su upute o instalaciji navedene u Qt-evoj dokumentaciji [20].

Moduli

Qt5 je sastavljen od više modula koji se razlikuju po svojoj dostupnosti. Postoji 14 osnovnih modula koji su podržani na svim razvojnim platformama te na svim testiranim ciljnim platformama. Osnovni moduli su dostupni u komercijalnoj verziji i u sklopu verzije otvorenog koda, te se smatra kako su to moduli koji implementiraju funkcionalnosti potrebne za razvoj većine Qt aplikacija. Među osnovnim modulima nalaze se i moduli koji su korišteni u razvoju igre napravljene u sklopu ovog rada:

- Qt Core
- Qt GUI
- Qt Multimedia
- Qt Widgets.

Uz osnovne module, Qt5 broji preko 40 dodatnih modula koji ne spadaju u osnovnu instalaciju. Dodatni moduli nisu nužno dostupni u obje verzije Qt5, također nisu ni podržani na svim razvojnim platformama, ni na svim testiranim ciljnim platformama. Iako u sklopu ovog rada dodatni moduli neće biti korišteni, u nastavku slijedi popis nekih od modula korisnih za izradu video igara:

- Qt 3D
- Qt Android Extras
- Qt Gamepad
- Graphical Effects

1.1 Meta-Object sustav

Jedno od osnovnih obilježja Qt-a je Meta-Object sustav. Meta-Object sustav pruža potporu za mehanizam signala i utora, o kojem će u sekciji 1.2 biti više govora, također pruža potporu za sustav svojstava (engl. *Property system*) te za dohvaćanje informacija za vrijeme izvođenja (*Run-Time type information - RTTI*).

Prema Qt-ovoj dokumentaciji [20], Meta-Object sustav bazira se na sljedećim sastavnicama:

- **QObject klasa** bazna je klasa koju nasljeđuju sve klase unutar kojih se koriste funkcionalnost koje Meta-Object sustav pruža.

- **Q_OBJECT** je makro koji se koristi unutar privatnog dijela deklaracije klase kako bi se osposobile značajke Meta-Object sustava.
- **Meta-Object Compiler (*moc*)** svaku klasu koja nasljeđuje QObject nadopunjuje potrebnim kodom koji implementira meta-object značajke.

QObject klasa

QObject je bazna klasa za sve Qt objekte. Ona je klasa koja implementira mehanizam signala i utora. QObject i klase koje ju nasljeđuju, međusobno se mogu organizirati u stablastu strukturu (hijerarhiju). Objektima `QObject` klase, objekte roditelje može se pridružiti na dva načina, kroz konstruktor - predajući pokazivač na objekt roditelj ili koristeći metodu `setParent(*parent)`. U trenutku određivanja roditelja, početni objekt se automatski dodaje na listu djece objekta roditelja. Na taj način, roditelj preuzima vlasništvo na objektom djetetom. Zahvaljujući ovakvoj strukturi, roditelj se brine o oslobađanju memorije koje dijete zauzima te automatski briše i dijete unutar svog destruktora.

Još jedna bitna stavka klase QObject jest to što pruža osnovne funkcionalnosti vremenskog brojača kroz metode `startTimer();` i `killTimer();`. Za naprednije funkcionalnosti vremenskog brojača postoji klasa `QTimer`.

Meta-Object Compiler

Meta-Object Compiler (*moc*) unatoč svom nazivu zapravo nije kompajler već generator koda. Štoviše, moc nije jedini generator koda unutar Qt biblioteke. User Interface Compiler (*uic*) je također generator koda koji iz opisa sučelja zapisanog u XML-u, generira C++ kod, dok Resource Compiler (*rcc*) generira kod kojim se resursi ugrađuju u Qt aplikaciju. Moc je sastavni dio Qt-a pomoću kojeg je omogućeno automatsko ili jednostavnije implementiranje složenih funkcionalnosti.

Moc čita datoteke zaglavlja klasa te u njima traži javljanje ključne riječi Q_OBJECT. Ukoliko pronađe javljanje ključne riječi, moc generira novu izvornu C++ datoteku u kojoj generira meta-object kod potreban za klase koje koriste Q_OBJECT makro. Generirani C++ kod sadrži implementacije metoda te instancu `QMetaObject` objekta koji sadrži meta podatke. Također, kod mora biti kompiliran i povezan (engl. *linked*) s implementacijom klase. Kod koji moc generira (*Meta-Object kod*) potreban je za implementaciju sljedećih značajki:

- signali i utori
- sustav svojstava
- informacije dobivene za vrijeme izvođenja koda (engl. *run-time type information*).

Signali i utori implementiraju se nakon što moc u kodu klase naiđe na ključne riječi `slots`, kojoj uvijek prethodi modifikator pristupa, ili `signals`.

Sustav svojstava (engl. *The Property System*), pruža mogućnost definiranja svojstava kroz korištenje `Q_PROPERTY()` makroa. Svojstva definirana na taj način ponašaju se slično kao varijable članice, klase unutar koje je definiran makro, uz dodatne mogućnosti. Primjer dodatnih značajki jest mogućnost čitanja i pisanja bez poznavanja klase kojoj to svojstvo pripada, već je dovoljno samo poznavanje imena svojstva. Čitanje i pisanje moguće je izvesti korištenjem funkcija `property()` i `setProperty()`.

Još neke ključne riječi koje moc prevađa u C++ kod su:

- `Q_ENUM()` deklarira listu enumeracija
- `Q_CLASSINFO()` omogućuje dodavanje para imena i vrijednosti nekom meta-objektu
- `Q_FLAGS()` deklarira enumeracije koje se ne isključuju međusobno.

Ključne riječi, Qt ekstenzije jeziku C++, su zapravo makroi koje moc definira u datoteci `qobjectdefs.h`, na način prikazan u isječku koda 1.1, preuzetom iz izvora [15].

Isječak koda 1.1: Definicija makroa

```
1 #define slots slots
2 #define signals signals
3 #define Q_SLOTS Q_SLOTS
4 #define Q_SIGNALS Q_SIGNALS
5 #define Q_PROPERTY(text) Q_PROPERTY(text)
6 #define emit
```

Obično kompiliranje C++ koda ide u 2 koraka:

1. Kreiranje svih datoteka objekata
2. Povezivanje datoteka objekata za program.

Kod korištenja moc-a potreban je jedan pretkorak, odnosno predkompiliranje svih potrebnih datoteka koristeći moc. Moc također upozorava o neispravnim ili opasnim izrazima unutar deklaracije klase s `Q_OBJECT` makroom.

Isječak koda 1.2: Definicija klase koja sadrži ključnu riječ `Q_OBJECT`

```
1 class Coin: public QObject, public QGraphicsEllipseItem
2 {
3     Q_OBJECT
4     Q_PROPERTY(qreal opacity READ opacity WRITE setOpacity)
5     Q_PROPERTY(QRectF rect READ rect WRITE setRect)
6
```

```
7  public:
8      explicit Coin(QGraphicsItem *parent = 0);
9      enum { Type = UserType + 4 };
10     int type() const;
11     void explode();
12     bool explosion() const;
13     void setExplosion(bool explosion);
14
15 private:
16     bool mExplosion;
17 };
```

Zašto Qt koristi moc

Dva bitna pojma vezana uz računarstvo su introspekcija i refleksija. U kontekstu računarstva, introspekcija je svojstvo programa da ispita karakteristike objekta za vrijeme izvršavanja, dok je refleksija svojstvo da proces koji se odvija ispita te modificira vlastitu strukturu te ponašanje. Pod karakteristike spadaju ime objekta, bazna klasa koju nasljeđuje, metode, varijable članice ili tip objekta [16]. Dok neki jezici kao što je Python podržavaju i introspekciju i refleksiju, C++ ima ograničenu podršku za introspekciju [15]. Tu na snagu stupa moc, je alat koji omogućuje introspekciju.

RTTI u C++

Identificiranje tipova podataka za vrijeme izvođenja (*Run-Time type identification - RTTI*) u C++ osposobljeno je kroz sljedeće funkcije:

- `dynamic_cast()` - pretvorba tipova podataka
- `typeid()` - dohvaćanje tipova podataka.

Uz navedene metode, moguće je dohvaćanje imena klase, lokaciju tablice virtualnih funkcija te hijerarhiju [12]. Unatoč tome, RTTI ima ograničenja i mane zbog kojeg se često izbjegava, a u pojedinim je projektima čak i zabranjena njegova upotreba [6].

Ograničenja i mane RTTI-a:

- RTTI se može koristiti samo s polimorfnim klasama, dakle klasama koje sadrže bar jednu virtualnu funkciju.
- Neki C++ kompajleri ne sadrže podršku za RTTI. S druge strane, zbog potrebe za dodatnim informacijama za spremanje tipova podataka, neki kompajleri koji podržavaju RTTI zahtijevaju eksplicitnu sklopku za omogućavanje RTTI-a.

- Funkcija `dynamic_cast()` nije stabilna kod korištenja dinamičkih biblioteka te su performanse nepredvidive.
- RTTI nije u skladu s C++ *zero-overhead policy*, princip prema kojem izvršavanje implementiranih apstrakcija ne smije trajati duže od vlastitih implementacija [6].

Introspekcija koristeći moc

Moc, kao što je već rečeno, je generator koda koji implementira razne metode te kreira objekt `QMetaObject`. Instanca `QMetaObject` klase je instancirana za svaku klasu koja sadrži `Q_OBJECT` makro te sadrži sve meta podatke potrebne za Qt-eve mehanizme.

Neki od podataka koji se mogu dohvatiti su:

- ime klase putem metode `className()`
- meta objekt klase koju nasljeđuje putem `superClass()`
- informacije o metodama klase putem `method()` i `methodCount()`
- informacije o svojstvima klase putem `property()` i `propertyCount()`.

Kod koji moc generira je pisan koristeći standardni C++, zbog čega ga je moguće kompilirati koristeći bilo koji C++ kompajler [20], a kao stabilniju alternativu funkciji `dynamic_cast()`, Qt nudi metode `qobject_cast()` i `qt_metacast()` [5].

Iako se introspekcija rijetko koristi za razvoj jednostavnih grafičkih aplikacija, potrebna je za izradu skriptnih izvođača te alata za izgradnju grafičkih sučelja [20]. U kontekstu Qt-a navedeni elementi introspekcije koriste se implementaciju internih funkcionalnosti, među kojima su i signali i utori. Alternativa realizaciji signala i utora pomoću introspekcije, je C++ sustav predložaka (engl. *template sustav*), međutim takva bi realizacija, prema *The Qt company*, bila manje robusnija i čitljivija [5].

1.2 Signali i utori

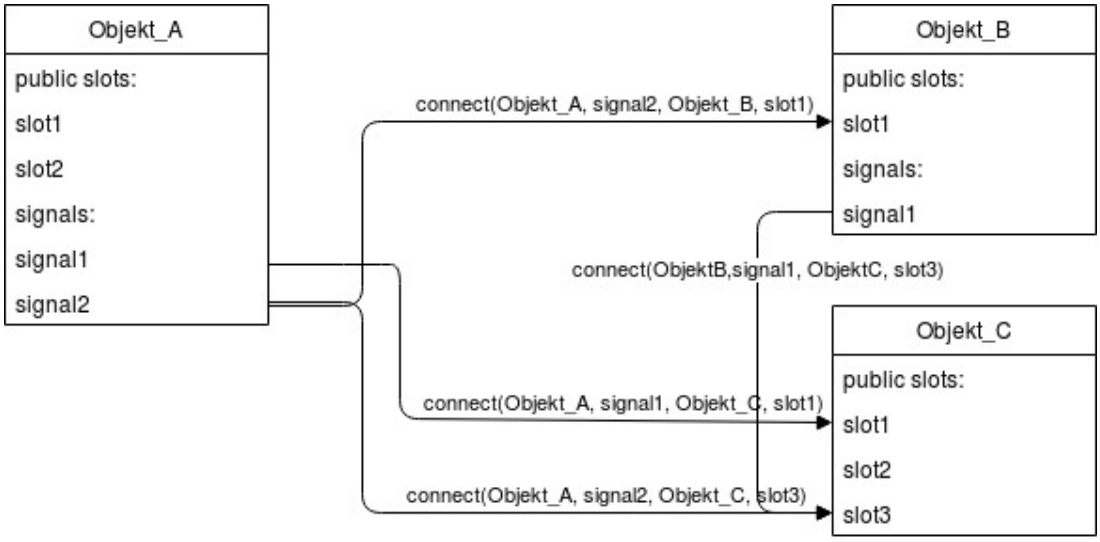
Signali i utori (engl. *Signals and slots*) jedno je od najbitnijih obilježja Qt5, a neophodan je i u izradi video igre. Signali i utori su mehanizam pomoću kojeg je implementirana komunikacija različitih objekata unutar Qt5. Primjer scenarija u kojem je korišten navedeni mehanizam je pritisak na gumb *Start* nakon kojeg se šalje signal `clicked()` povezan s metodom `startGame()`.

Komunikacija između gumba i metode jest ono u čemu se Qt5 razlikuje od većine ostalih platformi. Na drugim platformama komunikacija različitih objekata često je ostvareno korištenjem `callback()` funkcije, odnosno prosljeđivanjem pointera na funkciju

koja odrađuje posao. Takav pristup sa sobom nosi probleme prosljeđivanja krivih tipova podataka. Qt5 za razmjenu poruka između objekata koristi signale i utore. Prema [5] navedeni mehanizam je izgrađen iz tri koncepta:

- **Signal** - poruka koju objekt šalje
- **Utor** - metoda koja se izvršava kao reakcija na signal
- **Veza** - `connect` metoda koja spaja signale i utore.

Prednost signala i utora je što se jedan signal može povezati s više različitih utora, te jedan utor može biti povezan s više signala, kao što je prikazano na slici 1.1. Povezani objekti mogu biti različitih klasa, a mogu pripadati i različitim dretvama.



Slika 1.1: Shematski prikaz povezivanja signala i utora

Signali i utori deklariraju se kao metode u definiciji klase. Signali su u definiciji klase naznačeni ključnom riječi `signals`, ne smiju imati povratnu vrijednost te su uvijek javno definirani. Signali se ne implementiraju, već se implementacija generira pomoću moc-a. Utori su naznačeni ključnom riječi `slots`, također ne smiju imati povratnu vrijednost, ali mogu biti privatne, zaštićene ili javne metode. Prednost utora je što osim uloge u mehanizmu signala i utora, mogu se koristiti zasebno te pozivati kao metoda klase. Povezivanje signala i utora provodi se koristeći metodu `connect`, gdje liste tipova parametara metoda signala i utora moraju odgovarati.

Sintaksa

Postoje dvije vrste sintakse za pozivanje `connect` metodom, međutim prednost se daje novijoj sintaksi iz više razloga. Prema Qt-ovoj dokumentaciji [20], nova sintaksa omogućuje:

- automatsko pretvaranje tipova (engl. *castanje*), ako je moguća implicitna konverzija
- uporaba metode klase kao utora
- provjera postojanja signala i utora te tipova za vrijeme kompiliranja.

Stara sintaksa:
`connect(sender, SIGNAL(signal(int)),receiver, SLOT(slot(int)));`.
Nova sintaksa:
`connect(sender, &Sender::signal, receiver, &Receiver::slot);`

Poziv nove sintakse se sastoji od objekta koji šalje signal - `sender` (instanca klase `Sender`), signala koji šalje - `&Sender::signal`, objekta koji prima signal - `receiver` (instanca klase `Receiver`), te utora, metode koja se izvršava kao reakcija na signal - `&Receiver::slot`.

Primjer signala i utora

Qt-eve klase imaju preddefinirane signale i utore. Primjer je klasa `QLineEdit`, grafički element za unos teksta. Neki od signala koje sadrži su:

- `textChanged()` - signal se šalje kada se promijeni tekst
- `returnPressed()` - signal se šalje kad je pritisnuta tipka za potvrđivanje.

Neki od utora su:

- `paste()` - metoda koja unosi tekst zapamćen u međuspremniku (engl. *clipboard*)
- `clear()` - metoda koja briše tekstualni sadržaj grafičkog elementa.

Preddefinirani signali i utori mogu se povezivati s vlastitim implementacijama utora i signala. Utor `clear()` može biti povezan sa signalom koji emitira pritisak neke tipke na tipkovnici, dok se na signal `textChanged()` može vezati utor s funkcijom provjere zadovoljava li uneseni tekst određene uvjete.

Meta object kod iza signala i utora

Indeksiranje metoda

Signali, utori i metode neke klase koja sadrži `Q_OBJECT` makro popisani su u pripadajućem `QMetaObject` objektu. Svakoj je metodi pridružen indeks, počevši od 0, prateći navedeni redoslijed. Ovaj indeks naziva se relativni indeks. Češće korišten indeks naziva se apsolutni indeks. Prilikom određivanja apsolutnog indeksa metodama, uključuju se i metode svih klasa iz lanca nasljeđivanja. Tada se indeksima dodaje pomak jednak broju metoda u baznim klasama.

Signali

Slanje signala u kodu izvodi se pozivajući ključne riječi `emit` te u nastavku poziv metode signala. Ključna riječ `emit` je zapravo prazni makro, kao što je prikazano u isječku koda 1.1, koji služi programeru za lakše razumijevanje koda. Slanje signala implementira se u automatski generiranoj implementaciji signala u datoteci `moc_CLASS.cpp`, gdje je `CLASS` ime klase. Signal se implementira kao obična metoda koja poziva `activate()` metodu `QMetaObjecta` te joj prosljeđuje povratni tip i argumente kao u isječku koda 1.3.

Isječak koda 1.3: Implementacija signala, preuzeto s [15]

```
1 void className::signalName(Type arg1)
2 {
3     void *_a[] = { Q_NULLPTR, const_cast<void*>
4                    (reinterpret_cast<const void*>(&_arg1)) };
5     QMetaObject::activate(this, &staticMetaObject, 0, _a);
6 }
```

1.3 Qt Widgets

Qt Widgets je modul koji sadrži osnovne elemente korisničkog sučelja potrebne za izradu Qt aplikacija. Widgeti *(izvedenica iz riječi: Windows i Gadgets)* su osnovni blokovi od kojih se gradi grafičko korisničko sučelje. Primaju događaje (engl. *events*) uzrokovane računalnim mišem, tipkovnicom ili nekim drugim objektom, te iscrtavaju reprezentaciju sebe na prozor.

Unatoč svom vidljivom obliku, widgeti su u Qt-evoj grafici pravokutnici te se na ekran iscrtavaju u odnosu na gornji lijevi kut te prema Z-redoslijedu. Z-redoslijed (eng. *Z-order*) je redoslijed koji određuje preklapanje objekata, odnosno kojim se redoslijedom iscrtavaju. Služe prikazivanju informacija, statusa, služe korisniku za unos podataka ili kao spremište za druge widgete koje želimo grupirati. `QWidget` je osnovna klasa koju nasljeđuju svi

elementi korisničkog sučelja definirani u sklopu Qt Widgets modela, a sama `QWidget` klasa nasljeđuje *QObject* i *QPaintDevice*.

Definiranjem roditelj-potomak veze između komponenta grafičkog sučelja, olakšava se upravljanje memorijom jer objekti roditelji upravljaju oslobađanjem memorije objekata potomaka. Objekti se na navedeni način mogu povezivati, te grade hijerarhijsku strukturu. Glavnom prozoru aplikacije nikad nije pridružen objekt roditelj.

Qt Widgets uobičajeno se koristi za izradu desktop aplikacija. Widgeti su najčešće elementi za izgradnju korisničkog sučelja, primjereni za statična sučelja [20].

Qt Designer

Qt Designer je alat za izradu Qt aplikacija, dostupan u sklopu Qt Creator-a, integrirane razvojne okoline. Prednost ovog alata je jednostavno dizajniranje grafičkih sučelja. Qt Designer je grafički alat, objekti se mogu instancirati i povezivati povuci i ispusti metodom (engl. *drag and drop*). Glavni i osnovni blokovi izgradnje aplikacije su Qt Widgeti. Radna površina alata podijeljena na pet cjelina:

- glavni radni prozor s radnom reprezentacijom prozora koji izrađujemo
- popis dostupnih widgeta
- prikaz odnosa među widgetima (hijerarhija) te svojstva
- veze među objektima.

Izrađene forme i prozori pomoću Qt Designera jednostavno se integriraju s kodom te ih je moguće mijenjati dinamički koristeći kod.

1.4 Qt Quick

Qt Quick alternativa je korištenju Qt Widgets-a. Ovaj modul prilagođen je za razvoj QML (*Qt Modelling Language*) aplikacija koje se koriste na uređajima s ekranima osjetljivima na dodir [7]. Kao što je spomenuto Qt Quick-om se izrađuju QML aplikacije, međutim modul sadrži i C++ API (*Application programming interface*) za proširenje aplikacije C++ kodom.

Sastavne jedinice, iz kojih se gradi sučelje, u Qt Quick-u poznate su pod nazivom `QML types`. Dok je dostupno nekoliko osnovnih `QML type`, moguće je uvesti tipove iz drugih biblioteka ili izraditi vlastite tipove.

Prednosti Qt Quicka su što omogućuje u potpunosti prilagodljiv izgled te olakšava izradu i korištenje animacija i tranzicija. Vizualne efekte moguće je ostvariti koristeći specijalizirane komponente za čestične i sjenčane efekte.

QML

QML (*Qt Modelling Language*) je deklarativni programski jezik. Korišten je za opis korisničkog sučelja na dvije razine:

- opisuje izgled korisničkog sučelja
- opisuje ponašanje korisničkog sučelja.

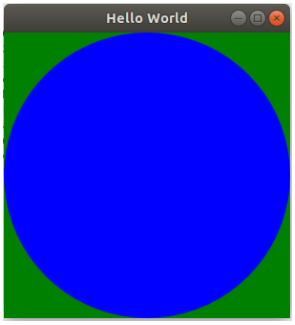
Sintaksa QML-a vrlo je slična JSON (*JavaScript Object Notion*) sintaksi, štoviše QML podržava `JavaScript` izraze u kodu ili uvoz vanjskih *JavaScript* biblioteka.

Isječak koda 1.4: Isječak QML koda

```
import QtQuick 2.12
import QtQuick.Window 2.12

Window {
    visible: true
    width: 300
    height: 300
    title: qsTr("Hello World")
    Rectangle{
        width: 300; height:300
        color: "green"
        Rectangle{
            width: 300; height: 300
            color: "blue"
            radius: 0.5*width
        }
    }
}
```

Vizualni rezultat QML koda prikazan na slici 1.2.



Slika 1.2: Vizualni rezultat QML koda

Modul koji pruža oboje, QML jezik i infrastrukturu, je Qt QML. Unatoč tome, Qt Quick pruža širok spektar vizualnih i interaktivnih komponenti te sustav za animacije. Qt Quick je alat kojim se omogućava korištenje QML-a, međutim Qt Quick nije jedini takav. Postoje i drugi sustavi koji omogućavaju razvijanje grafičkih sučelja u QML-u. Primjer takvih alata su Sailfish Silica, BlackBerry Cascade i QBS.

1.5 Infrastruktura za programiranje 2D igara

Qt je biblioteka koja je popularnost stekla podrškom za izradu grafičkih sučelja. Unatoč tome, danas se sve više razvija te nastoji pružiti što bolju podršku i za izradu 2D, ali i 3D, video igara. Osnovne stavke Qt-a za izradu video igara su klasa `QGraphicsView` te arhitektura grafičkog pogleda, podrška za detekciju kolizija te `QSettings` klasa za definiranje postavki aplikacije.

Uobičajena grafika igre sastoji se od tri osnovna dijela:

- instanca klase `QGraphicsView`, zvana *view* ili pogled
- instanca klase `QGraphicsScene`, zvana *scene* ili scena
- više instanci klase `QGraphicsItem`, zvane *items* ili grafički objekti.

Odnos među tim objektima predstavljen je slikom 1.3.

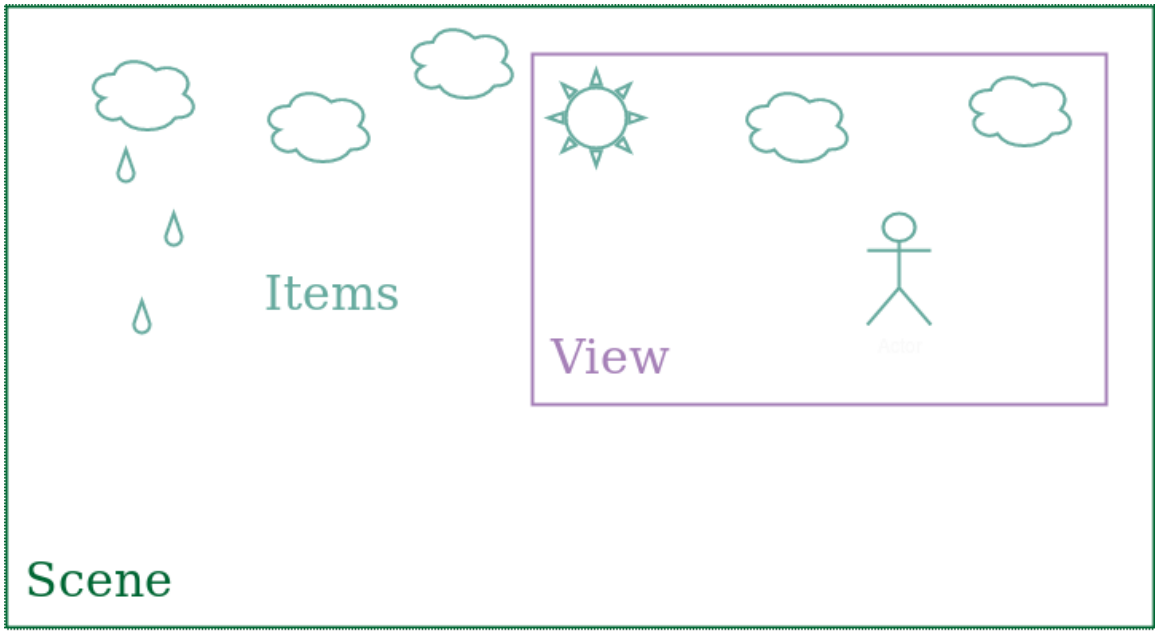
Grafički objekti predstavljaju osnovne jedinice kojima se gradi scena (i pogled). Objekte dodajemo na scenu, a potom pogled određuje koji dio scene se prikazuje. Pogled može prikazati cijelu scenu, njen dio ili se kretati te prikazivati cijelu scenu dio po dio.

Grafički objekti

Kao što je već spomenuto, grafički objekt je osnovna jedinica koja gradi scenu. `QGraphicsItem` služi kao bazna klasa za sve grafičke elemente na sceni, bilo posrednim ili neposrednim nasljeđivanjem, kao što je prikazano u slici 1.4. Objekti mogu prikazivati osnovne oblike, složenije oblike, tekst, slike, a mogu ostati i prazni. Objekti se mogu međusobno povezivati vezom roditeljstva, te mogu sačinjavati sličnu strukturu kao `QObject` objekti.

Qt pruža mogućnost implementiranja vlastite `QGraphicsItem` klase. Vlastita implementacija sastoji se od 3 ključna koraka:

- nova klasa mora nasljeđivati `QGraphicsItem`
- implementacija čiste virtualne metode `boundingRect()`, koja vraća pravokutnu aproksimaciju prostora koji zauzima objekt

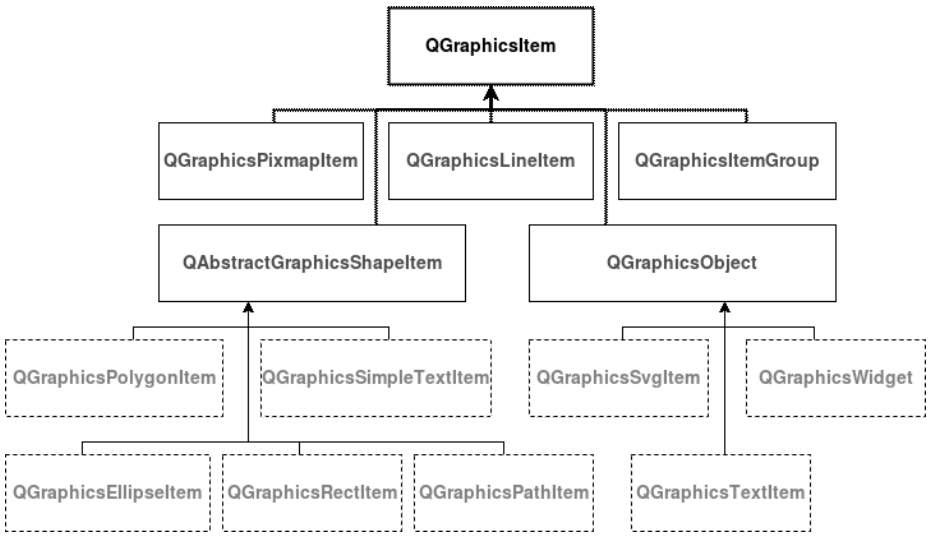


Slika 1.3: Odnos pogleda, scene i grafičkih objekata

- implementacija čiste virtualne metode `paint()`, koja implementira iscrtavanje oblika.

Kao alternativu stvaranju vlastite implementacije, Qt nudi nekoliko standardnih objekata kako bi olakšao i ubrzao razvoj programa. Standardni objekti se mogu modificirati te prilagođavati potrebama programera. U nastavku slijedi popis osnovnih oblika prema literaturi [1].

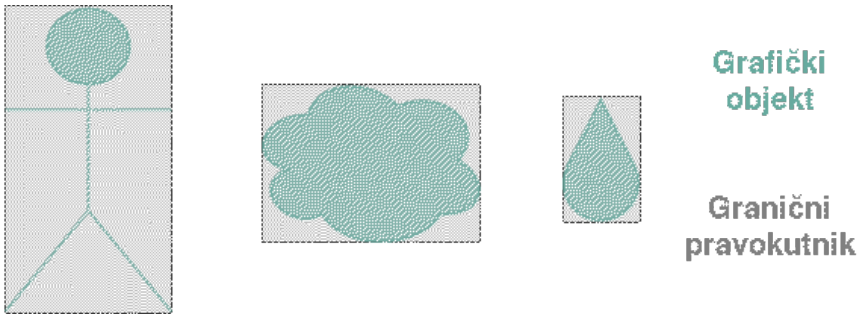
- `QGraphicsLineItem` - iscrtavanje jednostavne linije
- `QGraphicsRectItem` - iscrtavanje pravokutnika
- `QGraphicsEllipseItem` - iscrtavanje elipse
- `QGraphicsPolygonItem` - iscrtavanje poligona
- `QGraphicsSimpleTextItem` - iscrtavanje jednostavnog teksta
- `QGraphicsTextItem` - iscrtavanje tekstova složenije strukture
- `QGraphicsPixmapItem` - iscrtavanje različitih slika.



Slika 1.4: Nasljeđivanje klase `QGraphicsItem`

Svaki objekt uz standardna obilježja sadrži i granični pravokutnik (*bounding rect*) u odnosu na kojeg se postavlja pozicija.

Definicija 1.5.1 (Granični pravokutnik)**.** *Granični pravokutnik objekta O, je najmanji pravokutnik takav da se objekt O u potpunosti nalazi unutar pravokutnika.*



Slika 1.5: Granični pravokutnici grafičkih objekata

Ponašanje objekata moguće je modificirati koristeći zastavice zvane `QGraphicsItem::GraphicsItemFlag`. Po kreiranju objekata, sve su zastavice onemogućene. Neke od zastavica prikazane su u tablici 1.1.

Objekti se mogu prilagođavati potrebama metodama kao što su: `setEnabled()`, `setFocus()`, `setGraphicsEffect()`, `setOpacity()` te `setPos()`. Ovim metodama se

ItemIsMovable	Objekt podržava interaktivno pomicanje koristeći računalni miš.
ItemIsSelectable	Objekt podržava odabiranje.
ItemIsFocusable	Objekt podržava fokusiranje koristeći tipkovnicu.
ItemClipsToShape	Objekt reagira samo na događaje unutar oblika iscrtavanja, a ne ne objekte izvan oblika, unutar graničnog pravokutnika.
ItemClipsChildrenToShape	Objekt ograničava iscrtavanje svih potomaka na prostor svog oblika.
ItemIgnoresTransformations	Objekt zanemaruje sve nasljeđene transformacije.
ItemIgnoresParentOpacity	Objekt zanemaruje neprozirnost roditelja.
ItemDoesntPropagate OpacityToChildren	Objekt ne prosljeđuje neprozirnost na objekte kojima je roditelj.
ItemStacksBehindParent	Objekt se postavlja iza roditelja po Z-redosljedu.
ItemHasNoContents	Objekt ne iscrtava ništa.
ItemSendsGeometryChanges	Postavljanjem ove zastavice, metoda `itemChange()` obavještava o geometrijskim promjenama objekta.

Tablica 1.1: `QGraphicsItem` zastavice

redom određuje reagira li objekt na događaje, je li objekt fokusiran, grafički efekti, neprozirnost te pozicija. Uz naveden postoji još niz metoda kojima se objekti mogu modificirati [20].

Za upravljanje roditeljskim vezama između objekata, `QGraphicsItem` sadrži sljedeće metode: `setParentItem(QGraphicsItem *parent)` - postavlja novi objekt kao roditeljski, `parentItem()` - dohvaća roditeljski objekt, `childItems()` - vraća listu objekata koji su potomci odabranog objekta, `setFiltersChildEvents(bool enabled)` - određuje hoće li ili ne svi događaju usmjereni djeci ovog objekta ubuduće biti proslijeđeni objektu roditelju.

Pretvorba `QGraphicsItem` objekata

Funkcija koja nije metoda članica klase `QGraphicsItem`, ali je usko vezana uz nju, je `TypeT qgraphicsitem_cast(QGraphicsItem *item)`. Ako je moguće, funkcija primljeni objekt `item` (engl. *cast*) pretvara u objekt tipa `TypeT` te vraća takav objekt. Ukoliko pretvorba nije moguća, funkcija vraća `nullptr`. Kako bi se omogućilo korištenje navedene funkcije na objektima prilagođenih klasa potrebno je reimplementirati metode `type()`. Svi standardni objekti su povezani s jedinstvenom vrijednošću, koja se dohvaća metodom `type()`, dok je novoimplementiranim objektima pridana vrijednost `UserType`(65536).

Zato kako bi se naslijeđene klase mogle razlikovati, potrebno je reimplementirati navedenu metodu kako bi i ona vraćala jedinstvenu vrijednost na način prikazan u primjeru 1.5.

Isječak koda 1.5: Reimplementacija funkcije `type()`

```
class CustomItem : public QGraphicsItem
{
public:
    enum { Type = UserType + 1 };
    int type() const override
    {
        return Type;
    }
};
```

Scena

Nakon što je svladano kreiranje samih objekata, potrebno ih je dodati na scenu. Bazna klasa svih scena je `QGraphicsScene` koja nasljeđuje `Q_Object`. Dodavanje objekata na scenu izvodi se unutar metoda klasa koje nasljeđuju `QGraphicsScene` na dva načina.

- metodom `addItem()` dodaju se postojeći objekti
- metodama `addEllipse()`, `addLine()`, `addPath()`, `addPixmap()`, `addPolygon()`, `addRect()` i `addText()` se stvara novi, odgovarajući objekt, dodaje se na scenu te metoda vraća pokazivač na novostvoreni objekt.

Na navedene načine pridaje se vlasništvo nad objektom konkretnoj sceni. U svakom trenutku objekt može pripadati samo jednoj sceni pa ako se vlasništvo objekta pokuša predati drugoj sceni, veza objekta i prve scene se briše te objekt ostaje u vlasništvu samo nove scene.

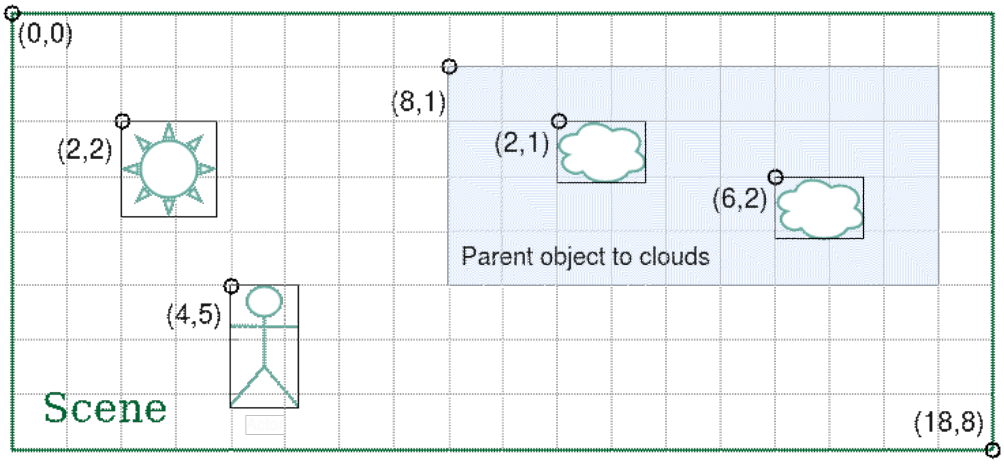
Scena je izgrađena iz tri sloja.

- `ItemLayer`
- `BackgorundLayer`
- `ForegroundLayer`

Kada `QGraphicsScene` iscrtava sadržaj, iscrtava slojeve određenim redoslijedom. Prvo se iscrtava `BackgorundLayer` pozivom virtualne metode `drawBackground()`, potom `ItemLayer` pozivom `drawItems()` te u konačnici `ForegroundLayer` pozivom metode `drawForeground()`.

Pozicioniranje komponenti na sceni

Kako je scena odgovorna za objekte koje posjeduje, u odnosu na nju određuje se i pozicija objekata. U slučajevima kada grafički objekti imaju uređenu vezu roditelj-potomak



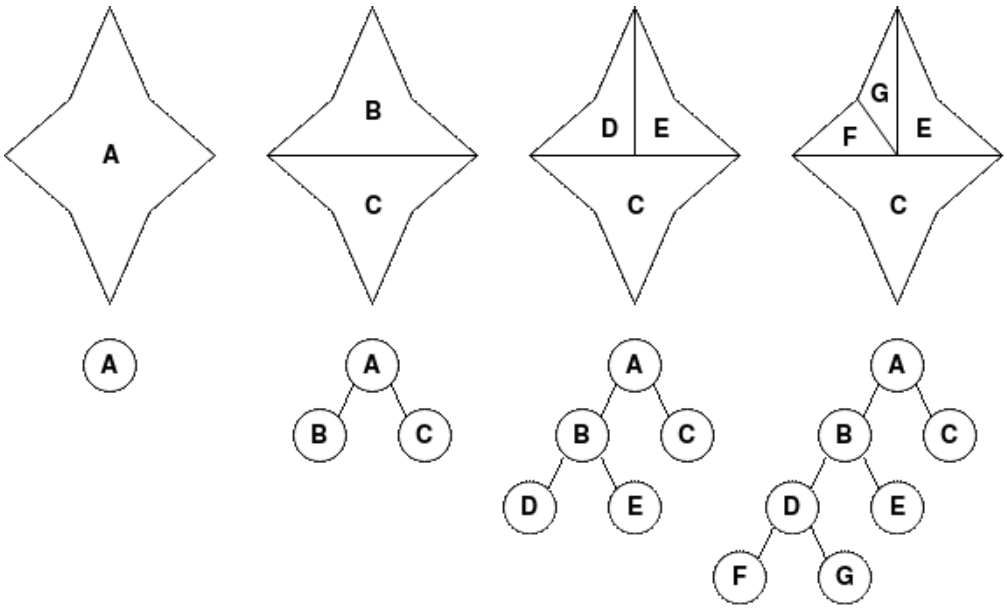
Slika 1.6: Koordinate grafičkih objekata na sceni

s drugim objektom poziciju se računa u odnosu na objekt roditelj. Slikom 1.6 vizualno je prikazano kako se računa pozicija grafičkih objekata u odnosu na scenu ili na objekt roditelj.

Prednost `QGraphicsScene` je sposobnost da brzo i efikasno određuje pozicije elemenata. Za upravljanje lokacijama komponenata, `QGraphicsScene` koristi indeksirajući algoritam (*Binary Space Partitioning tree* - BSP) binarnog particioniranja prostora. BSP je metoda kojom se prostor rekurzivno dijeli na dva konveksna podprostora koristeći hiperravnine te tvori binarno stablo [22], kao na slici 1.7. BSP algoritam je primjenjiv na velikim scenama s mnogo statičnih objekata.

Metoda `bspDepthTree()` vraća dubinu BSP stabla. Dubina BSP stabla utječe na performanse iscrtavanja scene te brzine pozicioniranja elemenata [20]. Optimalna veličina BSP stabla je kada svaki segment ima između 0 i 10 elemenata. Alternativa korištenju BSP indeksiranja je korištenje takozvanog `NoIndex` indeksiranja, objekti nisu indeksirani već se linearno pretražuju. Ovakva metoda pogodna je za scene s mnogo dinamičkih elemenata koji se često brišu i dodaju, jer je složenost tih operacija konstantna.

Metoda `items()`, koja vraća elemente na temelju njihove pozicije, ima nekoliko preopterećenja pa tako može vratiti listu objekata na sceni, objekata koji su unutar ili se presijecaju s određenim pravokutnikom ili objekata na određenoj poziciji. Lista je sortirana prema Z-redoslijedu.



Slika 1.7: Stablo binarnog prostornog particioniranja

Pogled

S kreiranom scenom te grafičkim objektima na njoj, preostalo je još samo iscrtavanje i prikazivanje željenog dijela scene. Za taj zadatak odgovorna je klasa `QGraphicsView` koja nasljeđuje `QAbstractScrollArea`. Prvi korak pri iscrtavanju scene je pridruživanje scene pogledu. Pridruživanje je moguće na dva načina, šaljući adresu scene kao argument konstruktoru pogleda, ili koristeći metodu `setScene()`. Drugi korak je pozivanje metode `render()` nad instancom `QGraphicsView` klase.

Ukoliko nije drugačije određeno, pri prvoj vizualizaciji pogleda, automatski je detektirano područje scene koje se iscrtava koristeći metodu `QGraphicsScene:itemsBoundingRect()`. Time se iscrtava granični pravokutnik koji sadrži sve grafičke komponente scene. Za određivanje prilagođenog područja koje se iscrtava, postoji metoda `setSceneRect()`.

Interakcije miša i tipkovnice s elementima na ekranu prvi obrađuje pogled koji ih prevađa u scenske događaje (`QGraphicsSceneEvents`) te ih prosljeđuje vizualiziranoj sceni, a scena ih prosljeđuje samom elementu koji ih obrađuje. Interaktivnost pogleda se može programabilno onemogućiti ili omogućiti koristeći metodu `setInteractive(bool enabled)`.

Optimizacija iscrtavanja

Pažljivi čitač zapitati će se što se događa s objektima koji nisu unutar pogleda, a na sceni su. Troše li oni resurse grafičke procesorske jedinice? Ovo pitanje je standardno pitanje koje se rješava takozvanim *view frustum culling* mehanizmom, mehanizmom koji detektira koji objekti nisu vidljivi te ih ne iscrtava. Navedeni mehanizam nije potrebno implementirati jer Qt optimizira iscrtavanje samo potrebnih elemenata [1].

Za napredno podešavanje performansi, `QGraphicsView`, pruža metode `setOptimizationFlag()` i `setOptimizationFlags`, kojima se mogu omogućiti sljedeće mogućnosti.

- `QGraphicsView::DontSavePainterState` - Postavljanjem ove zastavice, `QGraphicsView` ne vraća spremljena stanja `QPainter` objekta. Omogućavanje zastavice je korisno kada grafički objekti već sami vraćaju spremljeno stanje `QPainter` objekta.
- `QGraphicsView::DontAdjustForAntialiasing` - Postavljanjem ove zastavice, `QGraphicsView` ne povećava vidljiva područja, objekata koji iscrtavaju zaglađene linije (engl. *antialiasing*), za 2 pixela u svakom smjeru, čime se smanjuje područje koje zahtjeva iscrtavanje.

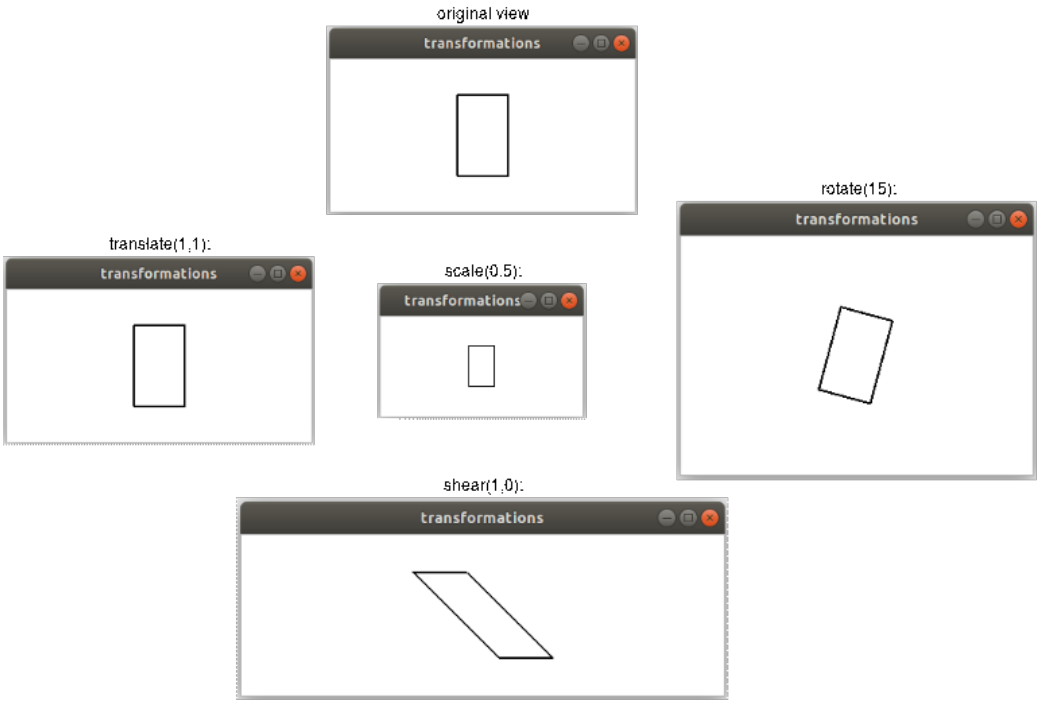
Transformacije

`QGraphicsView` podržava afine transformacije. Transformacije se mogu graditi pomoću matričnog zapisa koristeći metodu `setTransform()` ili koristeći metode `rotate()`, `scale()`, `translate()` ili `shear()`, učinak je prikazan na slici 1.8. Za vrijeme transformacija, `QGraphicsView`, održava centar pogleda fiksnim, zbog čega translacije nemaju vizualni učinak [20].

Alternativa korištenju `QGraphicsView` metoda za transformacije je korištenje instance klase `QTransform`, koja specificira transformacije 2D koordinatnog sustava. `QTransform` pruža mogućnost rotiranja, translacije, skaliranja, projekcije te smicanja. Transformacije se mogu zadati i matrično te se mogu pridruživati grafičkim objektima koristeći metodu `setTransform()`.

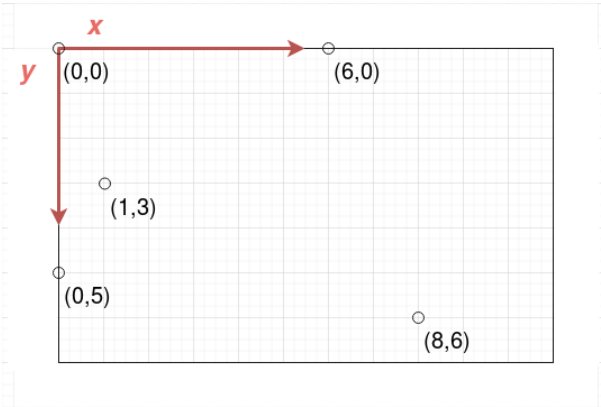
Koordinatni sustav

Jedan od osnovnih elemenata potrebnih za programiranje 2D grafike jest razumijevanje koordinatnog sustava alata koji se koristi. U sklopu Qt-eve podrške za iscrtavanje grafike bitno je istaknuti tri zasebna, ali povezana koordinatna sustava. Koordinatni sustav pogleda, sustav scene te sustav svakog pojedinog grafičkog elementa.



Slika 1.8: Transformacije pogleda

Koordinatni sustavi sastoje se od dvije koordinatne osi, x i y. X koordinata raste od izvorišta na desno, dok y koordinata raste od izvorišta na dole, kao što je i uobičajeno u području računalne grafike, slika 1.9.



Slika 1.9: Koordinatni sustav u računalnoj grafici

Kolizije

Tema detektiranja kolizija složena je tema vrijedna proučavanja, a primjenu je među ostalom našla i u programiranju video igara. Qt nudi podršku za osnovno detektiranja kolizija grafičkih objekata na sceni. Ovisno o potrebi, `QGraphicsItem` pruža dvije metode za detekciju kolizija.

Prva metoda daje odgovor na pitanje događa li se kolizija grafičkog objekta s nekim konkretnim drugim grafičkim objektom. Poziv ove metode,

```
objectA.collidesWithItem(objectB, ItemSelectionMode);
```

rezultira `true` ili `false` `bool` varijablom. Druga metoda odgovara na pitanje s kojim sve objektima konkretni objekt dolazi u koliziju. Poziv ove metode,

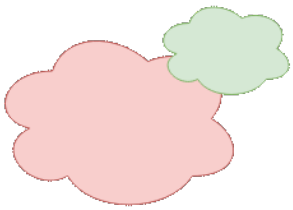
```
objectA.collidingItems(ItemSelectionMode);
```

rezultira listom objekata tipa `*QGraphicsItem`.

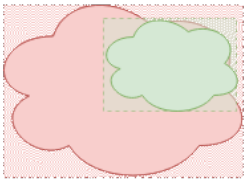
Obje metode primaju opcionalan argument koji određuje kako je definirana kolizija dva objekta. Postoje četiri moguća izbora.

Slika 1.10: Objekti u koliziji uz ContainsItemShape uvjet

`Qt::ContainsItemShape`: Kolizija je detektirana samo ako se objekti, s kojima se provjerava kolizija, u potpunosti nalaze unutar područja objekta nad kojim je pozvana metoda.



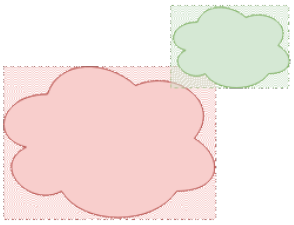
Slika 1.11: Objekti u koliziji uz IntersectsItemShape uvjet



Slika 1.12: Objekti u koliziji uz ContainsItemBoundingRect uvjet

`Qt::IntersectsItemShape`: Kolizija je detektirana ako se objekti, s kojima se kolizija detektira, barem presijecaju s područjem objekta nad kojim je pozvana metoda.

`Qt::ContainsItemBoundingRect`: Kolizija je detektirana ako se granični pravokutnici, objekata s kojima se provjerava kolizija, u potpunosti nalaze unutar graničnog pravokutnika objekta nad kojim je pozvana metoda.



Slika 1.13: Objekti u koliziji uz IntersectsItemBoundingRect uvjet

`Qt::IntersectsItemBoundingRect`: Kolizija je detektirana ako se granični pravokutnik, objekata s kojima se provjerava kolizija, presijeca s graničnim pravokutnikom objekta nad kojim je pozvana metoda.

Iako Qt nudi navedene metode za detektiranje kolizije, ne pruža mogućnost dohvaćanja točke kolizije.

QSettings

Posljednja od opisanih funkcionalnosti koje Qt sadrži, a korisne su za razvoj video igara, su postavke. Qt pruža vrlo jednostavno sučelje za trajno spremanje postavki aplikacije u obliku instance klase `QSettings`. Različite igre često imaju potrebu za realizacijom postavki unutar kojih korisnik može gasiti i paliti melodiju ili zvučne efekte, birati rezoluciju ekrana, mijenjati kontrole kojima se junak kreće ili nešto slično.

Klasa `QSettings` pruža trajno spremanje uređenih parova ključeva i vrijednosti. Ključevi su ključne riječi pod kojima se sprema ime neke opcije, dok su vrijednosti stanja

u kojima se to opcija može zateći. Ukoliko je potrebno odrediti uređeni par koji predstavlja opciju gašenja i paljenja zvuka unutar igre, ključna riječ bi mogla biti ”Zvuk”, dok bi vrijednosti bile ”Upaljeno” i ”Ugašeno”.

Isječak koda 1.6: Spremanje stavke postavke

```
1 QSettings settings;
2 settings.setValue("Sound", "On");
```

Isječak koda 1.7: Dohvaćanje stavke postavke

```
1 QSettings settings;
2 QString sound = settings.value("Sound", "").toString();
```

Kao vrijednosti, metoda `setValue()`, prima `QString` i `QVariant`. `QVariant` je klasa koja u načelu služi kao nositelj podataka širokog spektra tipova [5]. Može pohranjivati podatke osnovnih tipova kao što su `int`, `bool`, `char` i `float` te složenijih tipova kao što su `QString`, `QList` i `QUrl`. Također `QVariant` može simulirati unije za većinu korištenih tipova podataka u Qt-u, a često se koristi u svrhu serijalizacije podataka te za rad s bazama podataka unutar Qt-a.

Dohvaćanje svih vrijednosti provodi se metodom `allKeys()`, koja vraća listu objekata tipa `QString`.

Fizika svijeta

Jedna od ključnih stavki kod izgradnje video igara jest fizika svijeta u kojem je igra smještena. Dok u igrama poput vrlo poznatog *Minesweeper*a, svijet, a sa samim time i fizika svijeta, zapravo ne postoji, u igrama kao što su *Super Mario* [26], *Pocket Tanks* [25] ili *Angry Birds* [24] fizika određuje sam tok igre. Gravitacija koja djeluje na grafičke objekte, sila udarca, čvrstoća i elastičnost tijela, te putanja lansiranog objekta rezultati su zakona fizike koji vladaju stvorenim svijetom unutar igre.

Kao što je već napomenuto Qt je platforma za razvoj grafičkih aplikacija i iako pruža mnoštvo klasa i metoda s kojima je razvoj igara olakšan, ne pruža sustav za implementiranje fizike svijeta unutar video igre.

Poglavlje 2

Oblikovni obrasci

Oblikovni obrazac naziv je za prijedlog rješenja široj grupi problema koji se često javljaju pri razvoju programskih rješenja. Oblikovni obrazac sačinjava njegovo ime, rješenje koje je dokumentirano u općenitom obliku, te nije vezano uz specifičnosti jednog problema ili programskog jezika, te opis (grupe) problema na koji obrazac pruža rješenje. Obrasci se često prikazuju grafičkim prikazima (UML dijagramom).

2.1 Osnovna podjela oblikovnih obrazaca

Prema literaturi [3] oblikovni obrasci se najčešće svrstavaju u jednu od tri grupe:

- obrasci stvaranja
- strukturni obrasci
- obrasci ponašanja.

Obrasci stvaranja

Obrasci stvaranja (engl. *Creation patterns*) bazirani su na instanciranjima objekata. Pripomažu ostvarenju neovisnosti sustava od načina na koji su objekti kreirani, sastavljeni ili reprezentirani. Neki od poznatijih oblikovnih obrazaca koji spadaju u ovu kategoriju su: *Singleton, Prototype, Object Pool, Factory Method, Builder i Abstract Factory.*

Strukturni obrasci

Strukturni obrasci (engl. *Structural patterns*) bave se odnosom između klasa. Naglasak je na načinu na koji su klase i objekti komponirani u veće strukture. U nastavku su navedeni

poznatiji primjerci obrazaca koji spadaju u ovu kategoriju: *Flyweight, Composite, Bridge, Adapter, Decorator, Facade, Proxy, Private Class Data*.

Obrasci ponašanja

Obrasci ponašanja (engl. *Behavioral patterns*) usredotočeni su na komunikaciju između klasa. Bave se algoritmima i dodjelom odgovornosti među klasama. *State, Visitor, Command, Observer, Mediator, Strategy, Template method, Chain of responsability, Interpreter, Iterator i Memento* poznatiji su predstavnici kategorije obrazaca ponašanja.

2.2 Oblikovni obrasci u razvoju programskih igara

Kao i u mnogim drugim granama, u području razvoja video igara često se javljaju slični problemi, stoga se koristeći oblikovne obrasce traže univerzalna rješenja koja će biti primjenjiva u različitim okolnostima. Tematiku oblikovnih obrazaca u programiranju video igara razrađuje Nystrom u [2], a na njegovom se popisu pronašao i dio ovdje obrađenih oblikovnih obrazaca. Slijedi detaljni prikaz obrazaca *game loop*, *state* i *observer*, a potom kratki prikaz problema koje rješavaju obrasci *flyweight* i *composite*.

Game Loop

Osnovni i neizostavni oblikovni obrazac kad se govori o video igrama je *Game loop* obrazac. *Game loop* ili glavna petlja igre omogućuje protok vremena u igri, interaktivnost i dinamičnost igara.

Jednostavno rečeno, glavna petlja igre zapravo je beskonačna petlja u kojoj se obrađuje unos korisnika, promjene svih elemenata scene te se tada obrađeni elementi iscrtavaju na ekran.

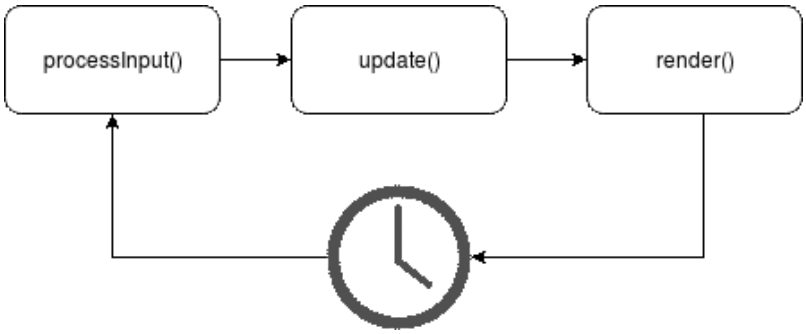
Osnovna implementacije petlje igre sastoji se od obrade unosa podataka, ažuriranja igre te iscrtavanja na scenu, prikazana isječkom 2.1.

Isječak koda 2.1: Osnovni algoritam igraće petlje

```
1 while(true){
2     processInput();
3     update();
4     redner();
5 }
```

Problem takvog algoritma je što trajanje jednog prolaza kroz petlju ne mora trajati jednako kao trajanje nekog drugog, pa zbog toga dolazi do vremenski neravnomjernog iscrtavanja te onog što nazivamo trzanje i zastoja u igrici.

Kako bi se to ispravilo uvodi se vremenski brojač kojim se vrijeme izvođenja jednog prolaza kroz petlju standardizira. Na kraj petlje postavlja se vremenski brojač koji čeka da prođe vrijeme koje treba imati svaki prolaz. Algoritam je prikazan slikom 2.1.



Slika 2.1: Sturktura glavne igraće petlje

Navedeni pristup je dobar kada je potrebno usporiti vrijeme izvođenja jednog prolaza kroz petlju, međutim ukoliko je vrijeme potrebno za obradu podata i ažuriranje duže no što je prvotno određeno za svaki prolaz potreban je drugačiji pristup.

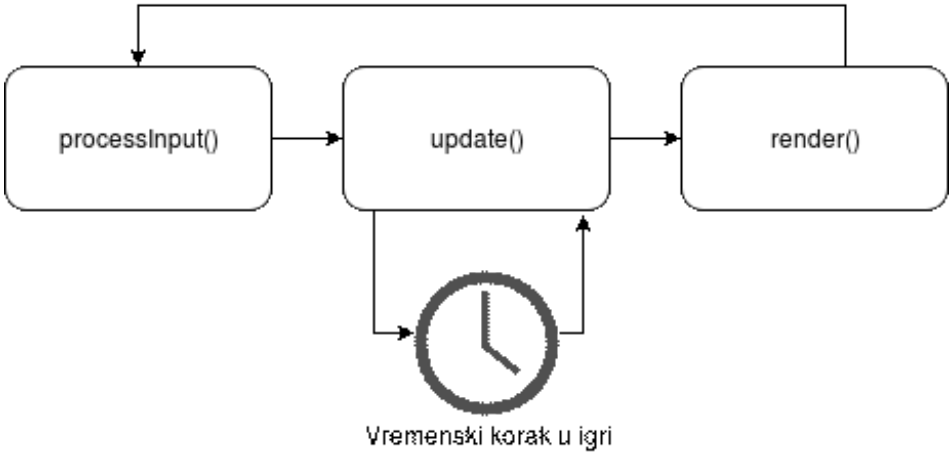
Kako bi se događaji u igri događali konstantnom brzinom, želimo da se stanja elemenata igre ažuriraju konstantnom brzinom, neovisno o tome iscrtavaju li se odmah ili ne. Kako bi nadoknadili stvarno vrijeme koje je bilo potrebno za obradu podataka u prošlom prolazu kroz petlju, u sljedećem prolazu kroz petlju ponavlja se ažuriranje podataka s fiksnim vremenskim razmakom, koji je prijevremeno određen kao vremenski korak u igri. Algoritam je prikazan slikom 2.2.

Game loop u Qt5

U pravilu Qt skriva implementaciju glavne petlje aplikacije. Nakon instanciranja objekta klase `QApplication`, nad njime se poziva metoda `exec()` kojom započinje glavna petlja događaja aplikacije koja obrađuje unos podataka.

Isječak koda 2.2: Pokretanje glavne petlje aplikacije

```
1 int main(itn argc, char argv*[])
2 {
3     QApplication app(argc, argv);
4     return app.exec();
5 }
```



Slika 2.2: Igraća petlja s vremenskim brojačem u `update()` metodi

Naravno, moguće je pristupiti izradi aplikacije na drugi način te implementirati svoju glavnu petlju aplikacije, odnosno video igre. Korisni alat u tom trenutku je `QTimer`, klasa koja pruža sučelje za brojače vremena.

State

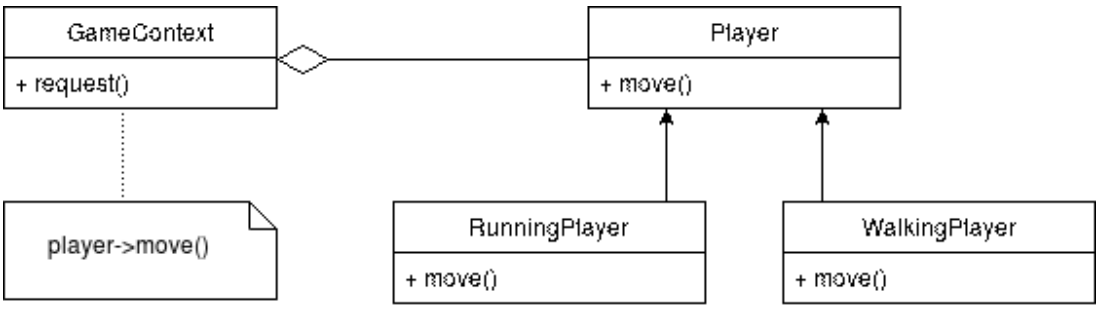
Još jedan od osnovnih oblikovnih obrazaca koji se koriste u razvoju video igara je *State* obrazac. Poanta ovog obrasca je optimiziranje koda kojim je implementiran objekt koji ima različita stanja te se u tim stanjima drugačije ponaša.

State obrazac podrazumijeva stvaranje apstraktne klase koju nasljeđuju klase koje predstavljaju pojedino stanje u kojem se objekt može zateći. Tokom igre kako objekt mijenja stanje, tako se mijenja i instancirana klasa koja ga predstavlja te implementira kako se taj objekt ponaša u pojedinmom stanju. Ovaj obrazac pomaže u restrukturiranju programskog koda koji sadrži metode s mnogo grananja kako bi pratile ponašanje stanja objekta s čitljivijim i skalabilnijim kodom u skladu s objektno orijentiranom paradigmom.

Dijagram 2.3 prikazuje moguću upotrebu *State* obrasca. *Player* je objekt koji se može zateći u dva stanja, trčećem ili hodajućem. Klase koje implementiraju ta dva stanja razlikuju se po implementaciji metode kretanja, koja pokreće odgovarajuću animaciju hodanja ili trčanja.

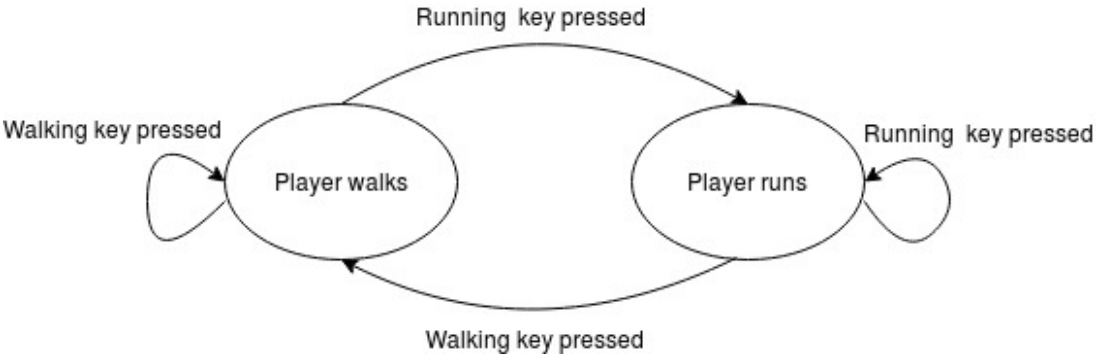
State u Qt5

Obrazac *State* usko je vezan uz konačne automate. *State* obrazac u kratkim crtama može se opisati nabrajajući stanja u kojima se tijelo može naći te kako prelazi iz jednog stanja u



Slika 2.3: Dijagram State obrasca

drugo, te skicirati, kao na skici 2.5. Ovako prezentiran *state* obrazac podsjeća na konačne



Slika 2.4: Skica stanja i prijelaza

automate. Slijedi formalna definicija konačnih automata [10].

Definicija 2.2.1 (Konačni automat)**.** *Konačni automat nad abecedom* Σ *je idealizirani (matematički) stroj* $\mathcal{M} := (\mathbf{Q}, \Sigma, \delta, q_0, F)$*, koji uz* Σ *sadrži i:*

- *konačan skup* $\mathbf{Q}$ *čije elemente nazivamo stanja*
- *istaknuti element* $q_0 \in \mathbf{Q}$ *koji nazivamo početno stanje*
- *podskup* $F \subseteq \mathbf{Q}$ *čije elemente nazivamo završna stanja,*
- *funkcija prijelaza* $\delta : \mathbf{Q} \times \Sigma \longrightarrow \mathbf{Q}$

U kontekstu ranijeg primjera 2.3, konačan automat se sastoji od: Σ - abeceda nad kojim automat radi su pritisnute tipke na tipkovnici, **Q** - stanje hodanja i trčanja, δ - funkcija prijelaza određena trenutnim stanjem i pritisnutom tipkom te rezultira novim stanjem, q_0 - početno stanje je hodanje. Navedeni primjer je jednostavno osmisliti te skicirati, kao na slici 2.5, a za implementaciju Qt sadrži *The State Machine Framework*. *The State Machine Framework* pruža niz klasa za izrade i izvršavanje grafova kojima se predstavljaju konačni automati.

Observer

Obrazac *Observer* svoju popularnost potvrđuje činjenicom da u sklopu jave postoji `java.util.Observer` i `java.util.Observable` klase. Također, *observer* je implicitno implementiran u samom jeziku `JavaScript` putem sustava događaja (engl. `events`).

Observer obrazac primjenjiv je u situacijama gdje se javlja veza jedan naprema mnogo među objektima, te je o promjeni jednog objekta potrebno obavijestiti više drugih objekata. U nastavku slijedi implementacija ovog obrasca.

Isječak koda 2.3: Observer obrazac - Observer

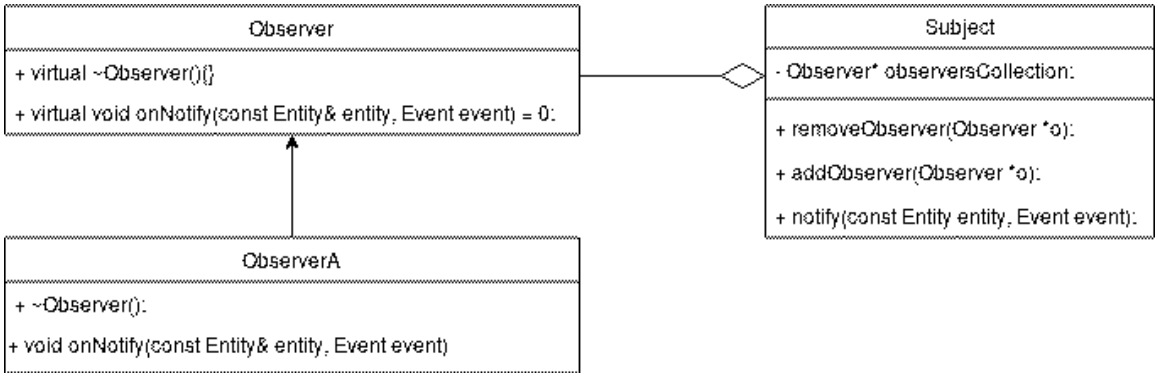
```
1  class Observer
2  {
3      public:
4      virtual ~Observer() {}
5      virtual void onNotify(const Entity& entity, Event event) = 0;
6  }
```

`Observer`, u odsječku koda 2.3, je apstraktna klasa koju nasljeđuju one klase kojima je potrebna informacija o nekom drugom objektu.

Isječak koda 2.4: Observer obrazac - Subject

```
1  class Subject
2  {
3  private:
4      Observers* mObservers[MAX_OBSERVERS];
5      int mNumObservers;
6  protected:
7      notify(const Entity& entity, Event event);
8  public:
9      void addObserve(Observer* observer);
10     void removeObserver(Observer *observer);
11 }
```

`Subject`, u odsječku koda 2.4, je objekt interesa, objekt o čijim promjenama mnogo drugih objekata ovisi. Bitno je metode kojima se dodaju i uklanjaju instance `observer`-a ostaviti javno dostupnim, kako bi vanjska okolina mogla odrediti koji će objekti biti obaviješten o promjenama, a koji ne.



Slika 2.5: Skica observers

Kako Nystrom navodi u [2], *observer* je koristan pri realizaciji, u video igrama čestog, sustava postignuća (*Achievements system*). Sustav postignuća nagrađuje junaka igre za razne događaje unutar igre, zbog toga se koristi *observer* obrazac. Razni objekti implementiraju apstraktnu klasu *observer* te obavještavaju sustav postignuća, *subject*, o specifičnom događaju.

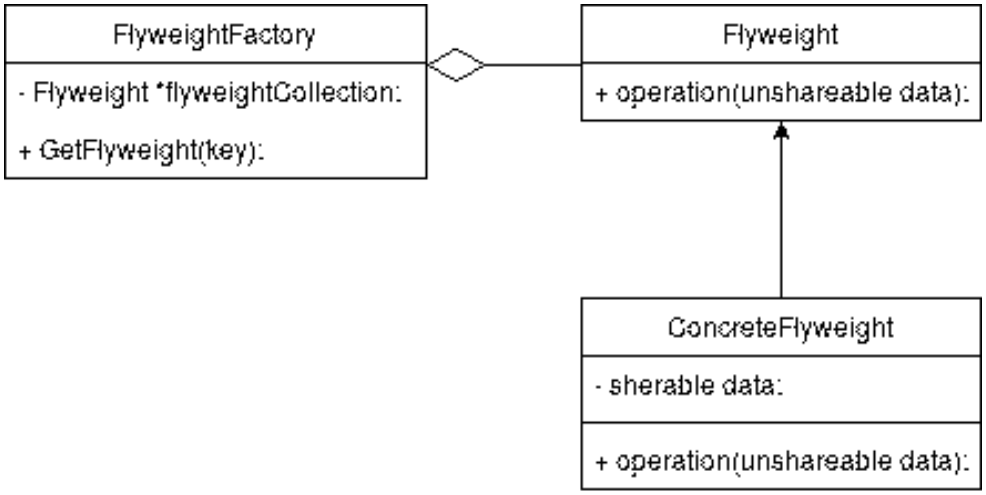
Observer u Qt5

Kao i u razvoju igara, ovaj se obrazac implicitno koristi i u samoj Qt5 biblioteci i to na dva načina. `QEvent` klasa i sve njene podklase implementiraju *observer* obrazac na vrlo niskoj razini. Također, cijeli mehanizam signala i utora implementira obrazac na nešto višoj razini, te pruža jednostavno sučelje za korištenje *observer* obrasca.

Flyweight

Kod video igara s bogatom grafikom i širokim područjem kojim se korisnik može kretati unutar igre, javlja se problem velikog broja grafičkih objekata za koje je potrebno dovoljno memorije, a potom i osigurati mogućnost prolaska podataka sa središnje jedinice na grafičku jedinicu. Jedan od obrazaca kojima se pristupa tom problemu je *Flyweight*. Ideja iza *Flyweight* obrasca je podjela podataka o nekom objektu na dva dijela, podaci o

instanci, te podaci koji su zajednički svim instancama te klase. Tada, ako postoji više objekata sa zajedničkim drugim dijelom, ti se podatci ne dupliciraju te se tako štedi na prostoru i vremenu.



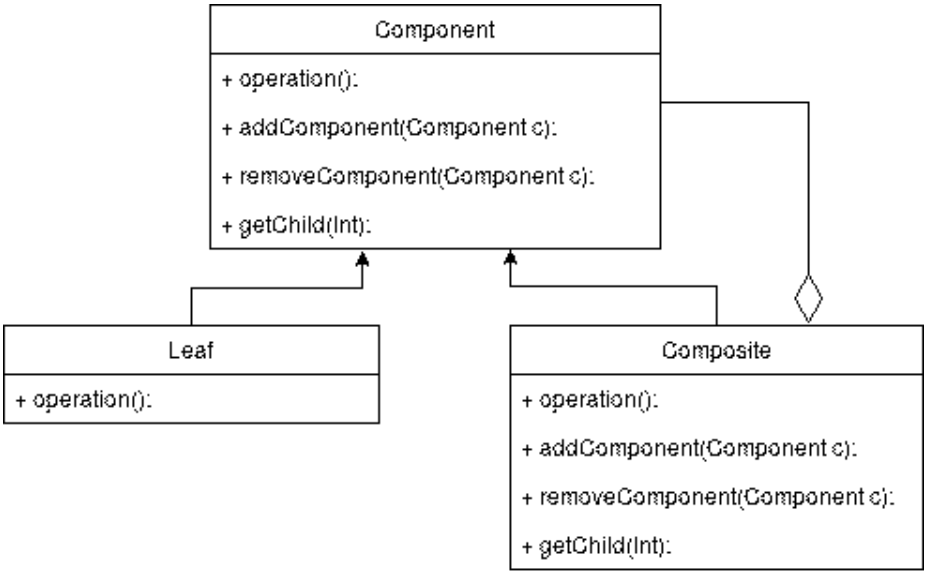
Slika 2.6: Dijagram Flyweight obrasca

Slika 2.6 prikazuje dijagram *flyweight* obrasca. *FlyweightFactory* je klasa zadužena za upravljanje *Flyweight* objektima. Svi podatci koji nisu djeljivi između objekata, uklonjeni su iz popisa atributa klase, te se direktno prosljeđuju metodi klase *ConcreteFlyweight* kojoj su potrebni. *FlyweightFactory* koristi privremenu memoriju (engl. *cache*) te koristi stare instance klase *Flyweight* [21].

Composite

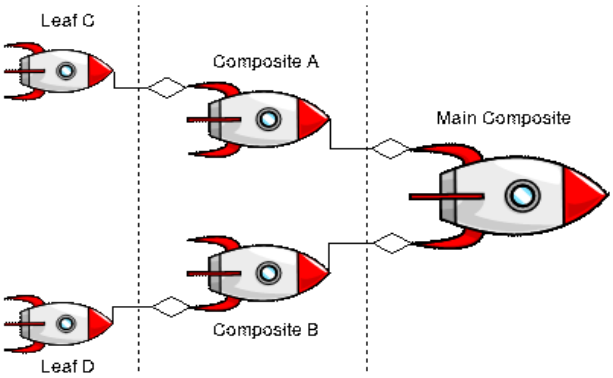
Mogući slučaj kod video igara je postojanje skupa grafičkih objekata, različitih složenosti i uloga, koji su povezani u hijerarhijsku strukturu. Ako platforma, na kojoj se video igra razvija, nema već implementiranu podršku, za ovakav slučaj često je korišten *composite* obrazac. *Composite* obrazac omogućuje da se neka grupa objekata tretira na isti način, ali istovremeno je uređena hijerarhija između svih članova obrasca. Dijagram oblikovnog obrasca prikazan je slikom 2.7.

Component je klasa koja predstavlja sučelje za sve objekte. *Leaf* predstavlja jednostavne objekte koji nemaju djece, dok je *Composite* komponenta s djecom. *Composite* sadrži reference na djecu, koja nasljeđuju *Component* klasu, dakle djeca su klase *Leaf* ili *Composite* [19].



Slika 2.7: Dijagram Composite obrasca

Na primjeru video igre, *Composite* je primjenjiv u igri koja sadrži glavni objekt kojim se upravlja te pomoćnim objektima koji prate glavni objekt prilikom kretanja. Odnosi među objektima mogu se graditi hijerarhijski kao na slici 2.8. *Main Composite* objekt je kojim korisnik upravlja te taj objekt sadrži referencu na objekte *Composite A* i *Composite B*, oni se mapom kreću tako da ostanu jednako udaljeni od *Main Composite* objekta. *Composite A* sadrži referencu na *Leaf C*, te se odnose na isti način. Analogno se ponašaju *Composite B* i *Leaf D*.



Slika 2.8: Grafički prikaz odnosa objekata u igri

Poglavlje 3

Implementacija video igre

U sklopu razrade teme ovog rada napravljena je 2D video igra koristeći Qt 5.13.1. biblioteku te Qt Creator 4.10. IDE. Igra pod nazivom *The Camel Run*, 2D je platformer, te je izrađena po uzoru na igru Super Mario. *The Camel Run* igra zamišljena je kao igra se mnogo kratkih levela, a za potrebe ovog rada, dizajnirana su prva dva levela.

Korišteni su Qt5 moduli: core, gui, widgets te multimedia, također je korišten `c++11` standard.

3.1 Pregled igre

The camel run je računalna igra. Protagonist igre jest umorna deva koja se kreće kroz pustinju ispunjenu preprekama kao što su kaktusi, škorpion te pješčane rupe, kako bi došla do oaze gdje se može odmoriti.

Upute o korištenju

Korisnik pomiče glavnog junaka igre tipkama lijevo, desno te skače razmaknicom (*space-bar*). Cilj igre je izbjegavati kaktuse i škorpione, preskakati rupe u platformi te doći do cilja.

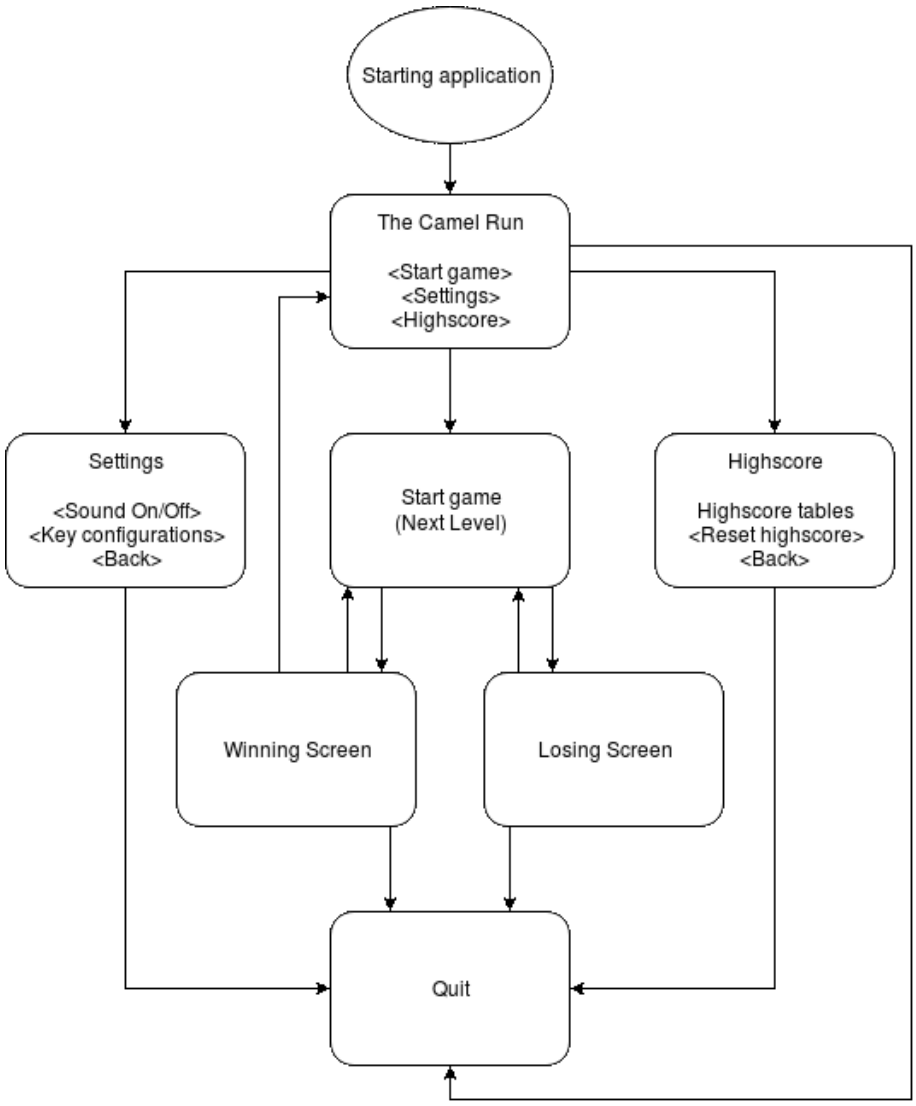
Arhitektura igre

Igra se sastoji od četiri glavna ekrana:

- glavni izbornik
- postavke

- najbolji rezultati
- igraći ekran.

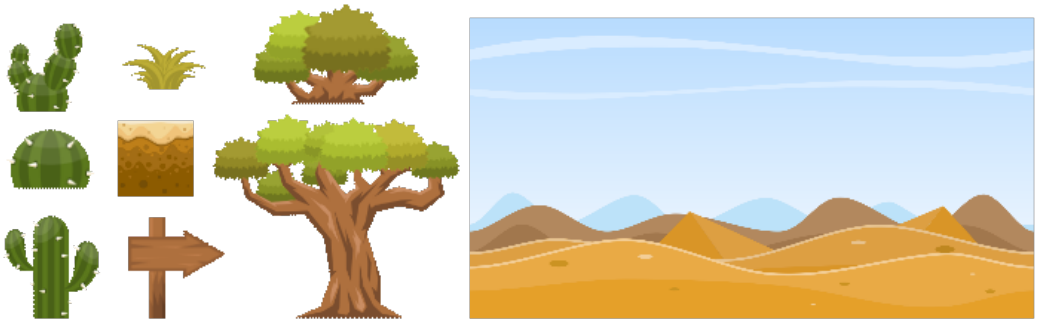
Izmjene ekrana, funkcionalnosti i tok igre prikazan je dijagramom 3.1.



Slika 3.1: Dijagram toka aplikacije

Resursi

Za potrebe izrade igre preuzeto je nekoliko javno dostupnih resursa dok je dio resursa izrađen. Sljedeći resursi preuzeti su sa stranice koja nudi mnoštvo resursa za izradu 2D video igara [27].

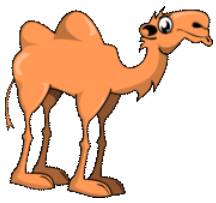


Slika 3.2: Pozadina i pustinjski objekti

Model škorpiona preuzet je sa [28].

Slika 3.3: Model škorpiona

Deva je preuzeta sa sljedećeg izvora [29].

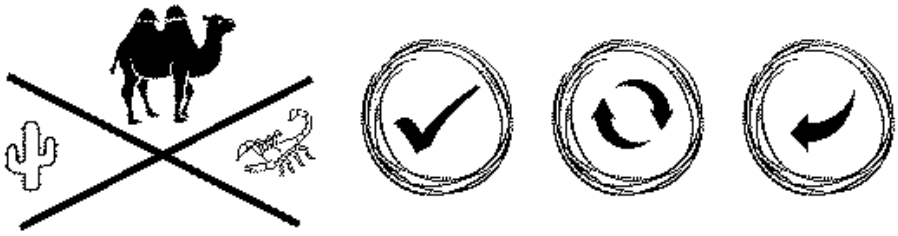


Slika 3.4: Model deve

Korišteni font preuzet je sa [30].
Ostatak resursa je izrađen.

The quick brown fox jumps over the lazy dog.

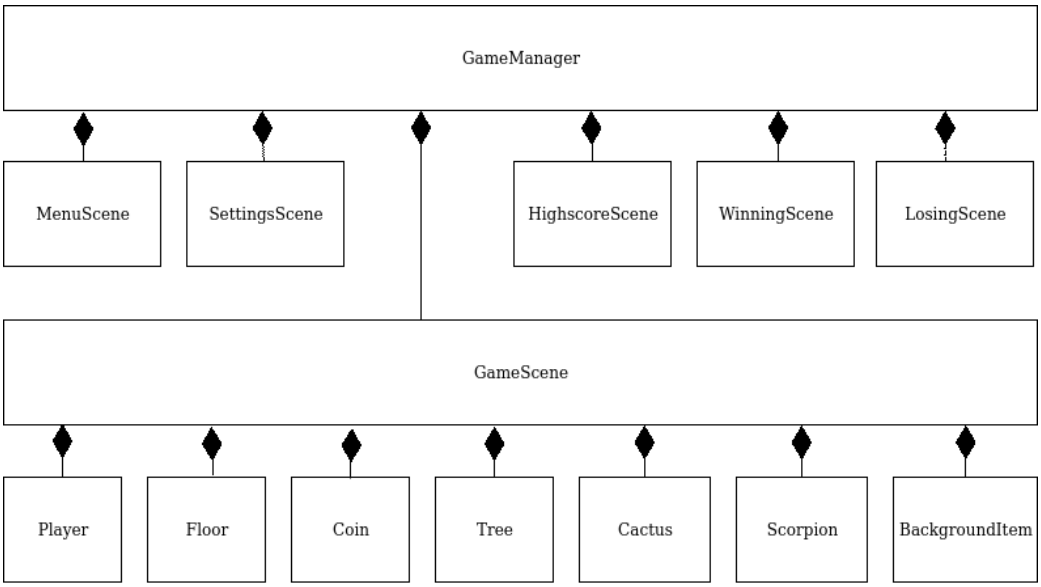
Slika 3.5: Uzorak korištenog fonta



Slika 3.6: Izrađeni resursi

3.2 Struktura aplikacije

Igra se sastoji od četrnaest klasa, čija je međusobna zavisnost prikazana dijagramom 3.7.



Slika 3.7: Class dijagram

`GameScene` sadrži prozirne `QGraphicsRectItem`-e koji potom drže pokazivače na instance klasa `Coin`, `Scorpio`, `Cactus` i `Floor`, dok `GameManager` drži po jedan pokazivač na svaku od klasa `MenuScene`, `SettingsScene`, `HighscoreScene`, `WinningScene`, `LosingScene` te `GameScene`.

Qt podrška za resurse

Uz navedene klase, aplikacija sadrži jednu `resource.qrc` datoteku.

Kako bi aplikacija imala pristup resursima, qt nudi mehanizam neovisan o platformi koji sprema odabrane resurse u binarnom zapisu unutar same aplikacije. Za korištenje sustava za resurse potrebno je izraditi datoteku s nastavkom `.qrc` (*Qt Resource Collection*).

Svi potrebni resursi, u koje mogu spadati: slike, zvukovi, fontovi ili ikone, spremaju se u tu datoteku. U kodu aplikacije, resurse pohranjene na naveden način moguće je dohvatiti pomoću njima pridruženog puta. Ovako spremljeni resursi bit će dostupni i u izvršnim datotekama aplikacije.

3.3 Funkcionalnosti igre

GameManager klasa

`GameManager` klasa nasljeđuje `QMainWindow` klasu te predstavlja glavni prozor aplikacije. Ona je odgovorna za izmjenu glavnih ekrana te iscrtavanje scena. Klasa sadrži ključnu riječ `Q_OBJECT` te koristi signale i utore za provođenje igre.

Isječak koda 3.1: Bitne stavke datoteke gamemanager.hr

```
class GameManager: public QMainWindow{
    Q_OBJECT
public:
    explicit GameManager(QWidget *parent = null);

private:
    QGraphicsView mView;
    MenuScene *mMenuScene;
    GameScene *mGameScene;
    SettingsScene *mSettingsScene;
    HighscoreScene *mHighscoreScene;
    LosingScene *mLosingScene;
    WinningScene *mWinningScene;
    QSound *mSound;

    void renderScene(QGraphicsScene*);

signals:
```

```
19      // SIGNALS
20
21  private slots:
22      // PRIVATE SLOTS
23
24  public slots:
25      //PUBLIC SLOTS
26  };
```

MenuScene klasa

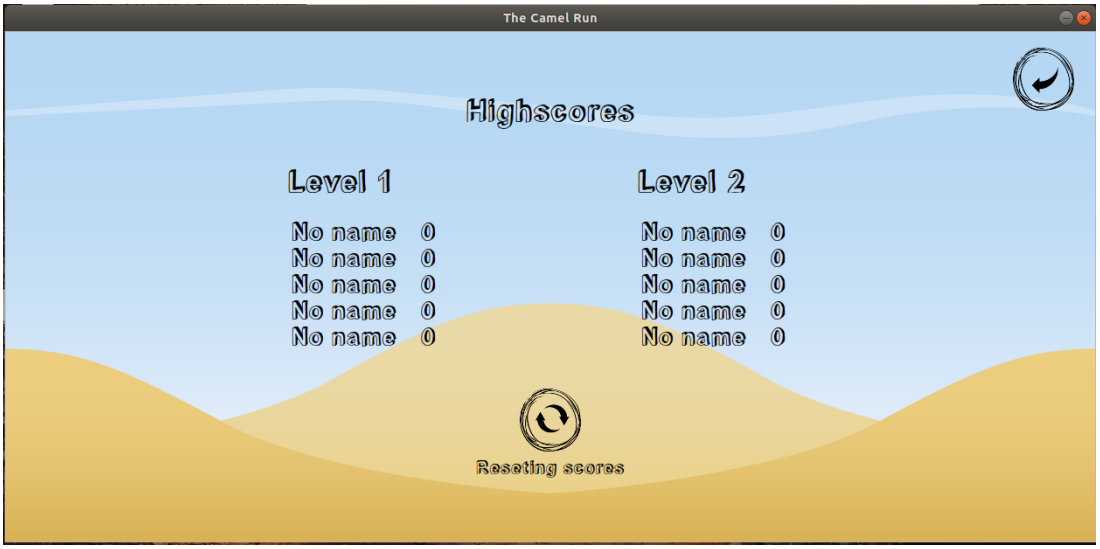
`MenuScene` klasa nasljeđuje klasu `QGraphicsScene` te implementira izgled i funkcionalnosti glavnog izbornika.

Slika 3.8: Glavni izbornik aplikacije

Klasa sadrži tri instance `QPushButton` widgeta uz pomoć čijih se signala `clicked()` komunicira s klasom `GameManager` i izmjenjuju scene.

HighscoreScene klasa

`HighscoreScene` klasa nasljeđuje klasu `QGraphicsScene`. Na sceni se ispisuju najbolji rezultati po levelima. Pri prvom pokretanju igre, liste najboljih rezultata su ispunjene vrijednostima ”No name” i brojem bodova 0.



Slika 3.9: Prikaz najboljih rezultata

Liste najboljih rezultata se dohvaćaju i spremaju pomoću instance klase `QSettings`. `QSettings` čuva uređeni par `QString`-a, koji određuje level i poziciju, te `QList`-u koja sadrži nadimak te broj bodova.

Isječak koda 3.2: Inicijalno punjenje lista najboljih rezultata

```
QSettings settings;
QList<QString> score({"No name", "0"});
QString settingsAttribute;
for(int i = 1; i < 3; i++){
    for(int j = 1; j < 6; j++){
        settingsAttribute = QString("Lvl%1_%2").arg(i).arg(j);
        settings.setValue(settingsAttribute, QVariant(score));
    }
}
```

Resetiranjem rezultata vraćaju se inicijalne vrijednosti.

SettingsScene klasa

`SettingsScene` klasa kao i prethodne dvije klase nasljeđuje klasu `QGraphicsScene`. U postavkama je omogućeno paljenje i gašenje zvuka te konfiguracija tipki s kojima se korisnik kreće u igri.

Vrijednosti o zvuku i trenutnoj konfiguraciji tipki zapamćene su koristeći instancu klase `QSettings`.

Slika 3.10: Ekran postavki

Isječak koda 3.3: Kod paljenja i gašenja zvuka

```
1
2  GameManager::GameManager(QWidget *parent):QMainWindow(parent){
3      // kod GameManager konstruktora
4      connect(mSettingsScene->mSoundBtn, &QPushButton::clicked,
5              this, &GameManager::changeSoundMode);
6      // kod GameManager konstruktora
7  }
8
9  void GameManager::changeSoundMode()
10 {
11     QSettings settings;
12     QString soundOffOn = settings.value("Sound", "").toString();
13
14     if (soundOffOn == "On"){
15         settings.setValue("Sound", "Off");
16     }
17     else{
18         settings.setValue("Sound", "On");
19     }
20     mSettingsScene->setSoundIcon();
21     playSound();
22 }
```

Signal gumba na ekranu postavki povezan je putem mehanizma signala i utora s metodom klase `GameManager`, `changeSoundMode()`.

GameScene klasa

Centralna scena igre *The Camel Run* je klasa `GameScene` koja također nasljeđuje klasu `QGraphicsScene`.

Slika 3.11: Startna pozicija igre

Glavni elementi `GameScene` su instanca klase `Player` te pravokutni prozirni grafički objekti koji su u roditeljskoj vezi s objektima klasa `Cactus`, `Scorpio`, `Coins`, `Floor` i `Tree`.

Player klasa

`Player` nasljeđuje `QGraphicsPixmapItem` te sadrži privatne varijable koje čuvaju podatke o stanju životnih bodova deve, smjeru u kojem se kreće te o količini skupljenih novčića. Kretanje i animacija skakanja deve provode se u klasi `GameManager`. Animacija skakanje deve implementirana je koristeći `QPropertyAnimation`. U definiciji klase definirani su `Q_PROPERTY, jumpFactor` te `QPropertyAnimation, jumpAnimation`. U nastavku slijedi implementacija animacije skakanja.

Definicija svojstva nalazi se u zaglavlju klase, istu klasu potrebno je označiti `Q_OBJECT` makro-om.

Isječak koda 3.4: Definicija Q_PROPERTY

```
1 Q_PROPERTY(qreal jumpFactor READ jumpFactor
2            WRITE setJumpFactor NOTIFY jumpFactorChanged)
```

`QPropertyAnimation` klasa animira `Q_PROPERTY` objekte. Potrebno je specificirati ime svojstva koje se animira te modificirati željene vrijednosti kako bi se animacija odvila na željeni način. U navedenom primjeru postavljene su početna i krajnja vrijednost svojstva, željena vrijednost u određenom vremenskom trenutku, vremensko trajanje animacije te funkcija prema kojoj se izračunavaju vrijednosti, `setEasingCurve()`. Qt nudi više od 40 definiranih tipova `QEasingCurve`, a moguće je definirati i vlastitu funkciju.

Isječak koda 3.5: Inicijalizacija QPropertyAnimation klase

```
1     mJumpAnimation = new QPropertyAnimation(this);
2     mJumpAnimation->setTargetObject(this);
3     mJumpAnimation->setPropertyName("jumpFactor");
4     mJumpAnimation->setStartValue(0);
5     mJumpAnimation->setKeyValueAt(0.5, 1);
6     mJumpAnimation->setEndValue(0);
7     mJumpAnimation->setDuration(800);
8     mJumpAnimation->setEasingCurve(QEasingCurve::OutInQuad);
```

Preostaje još samo uskladiti promjenu y koordinate objekta koji se animira sa svojstvom.

Isječak koda 3.6: Izračunavanje y koordinate i animacija skakanja

```
1  void GameScene::jump()
2  {
3      if (QAbstractAnimation::Stopped == mJumpAnimation->state()) {
4          mJumpAnimation->start();
5      }
6  }
7
8  qreal GameScene::jumpFactor() const
9  {
10     return mJumpFactor;
11 }
12
13 void GameScene::setJumpFactor(const qreal &jumpFactor)
14 {
15     if (mJumpFactor == jumpFactor) {
16         return;
17     }
18     mJumpFactor = jumpFactor;
19     emit jumpFactorChanged(mJumpFactor);
20
21     qreal groundY = mGroundLevel -
22                     mPlayer->boundingRect().height() / 2;
23     qreal y = groundY -
24               mJumpAnimation->currentValue().toReal() * mJumpHeight;
```

```
25
26      mPlayer->setY(y);
27  }
```

Cactus klasa

`Cactus` je još jedan od grafičkih objekata koji grade `GameScene`. Omogućena su tri levela kaktusa koji se razlikuju veličinom i štetom koju nanose devi. Level se određuje prilikom kreiranja instance klase.

Isječak koda 3.7: Konstruktor klase Cactus

```
1  Cactus::Cactus(QGraphicsItem *parent, int level) :
2          QGraphicsPixmapItem(parent),mDamage(1)
3  {
4      QString pixmapPath;
5      switch (level) {
6          case 1:{
7              pixmapPath = ":/cactusLvl1.png";
8              mDamage = 1;
9              break;
10         }
11         case 2: {
12             pixmapPath = ":/cactusLv2.png";
13             mDamage = 1;
14             break;
15         }
16         case 3:{
17             pixmapPath = ":/cactusLv3.png";
18             mDamage = 2;
19             break;
20         }
21     }
22     QPixmap pixmap(pixmapPath);
23     setPixmap(pixmap);
24     setOffset(-pixmap.width() / 2, -pixmap.height() / 2);
25 }
```

U ovisnosti o levelu kaktusa iscrtava se pripadajuća `pixmapa`. Kaktusi nanose štetu devi u trenutku dodira, koji se detektira navedenim metodama Qt biblioteke za detekciju kolizija. Ukoliko deva naskoči na kaktus, aktivira se animacija odskoka s tog kaktusa. Animacija odskoka je napravljena analogno animaciji za običan skok.

Scorpio klasa

`Scorpio` klasa dinamički je neprijatelj koji se kreće unutar svog fiksnog područja te nanosi štetu devi u trenutku dodira. `Scorpio` također nasljeđuje `QGraphicsPixmapItem` klasu. U trenutku stvaranja škorpiona generira se instanca klase `QTimer`, na čiji se signal `QTimer::timeout()` škorpioni kreću za određeni korak.

Floor klasa

Pod po kojem se deva kreće konstruiran je od mnogo instanci klase `Floor`. Koristeći `Floor` klasu, napravljene su zapreke koje je potrebno preskočiti. U trenutku kada deva pada s platforme, pokreće se metoda `sinking()` koja koristeći transformacije, konkretno rotacije, simulira upadanje deve u rupu u podu.

Slika 3.12: Postepena rotacije deve

Coin klasa

`Coin` klasa nasljeđuje `QGraphicsEllipseItem`. Novčić je iscrtan programabilno, koristeći klasične metode klase `QGraphicsItem`.

Isječak koda 3.8: Izrada novčića

```
Coin::Coin(QGraphicsItem *parent) :
    QGraphicsEllipseItem(parent),
    mExplosion(false)
{
    setPen(QPen(QColor(218, 165, 32), 2));
    setBrush(QColor(255, 223, 0));
    setRect(-12, -12, 24, 24);
}
```

U trenutku dodira deve s instancom klase `Coin` istovremeno se dešavaju dva ključna događaja:

- animacija eksplozije novčića
- ažuriranje brojača novčića u gornjem lijevom kutu.

Animacija eksplozije novčića sastoji se od dvije instance `QPropertyAnimation`, jedna je odgovorna za promjenu veličine novčića, dok je druga odgovorna za postepeno nestajanje novčića.

Isječak koda 3.9: Animacija novčića

```
void Coin::explode()
{
    if (mExplosion) {
        return;
    }

    mExplosion = true;
    QParallelAnimationGroup *group = new QParallelAnimationGroup(this);
    QPropertyAnimation *scaleAnimation =
                        new QPropertyAnimation(this, "rect");
    scaleAnimation->setDuration(700);
    QRectF r = rect();
    scaleAnimation->setStartValue(r);
    scaleAnimation->setEndValue(
                QRectF(r.topLeft() - r.bottomRight(), r.size() * 2));
    scaleAnimation->setEasingCurve(QEasingCurve::OutQuart);
    group->addAnimation(scaleAnimation);

    QPropertyAnimation *fadeAnimation =
                        new QPropertyAnimation(this, "opacity");
    fadeAnimation->setDuration(700);
    fadeAnimation->setStartValue(1);
    fadeAnimation->setEndValue(0);
    fadeAnimation->setEasingCurve(QEasingCurve::OutQuart);
    group->addAnimation(fadeAnimation);

    connect(group, &QParallelAnimationGroup::finished,
            this, &Coin::deleteLater);
    group->start();
}
```

Brojač novčića ažurira se koristeći mehanizam signala i utora. Nakon što dolazi do kolizije s novčićem, emitiran je signal da je novčić skupljen te se poziva metoda-utor koja obrađuje taj događaj. Na takav način obrađuje se i promjena stanja životnih bodova deve.

Isječak koda 3.10: Povezivanje signala i utora

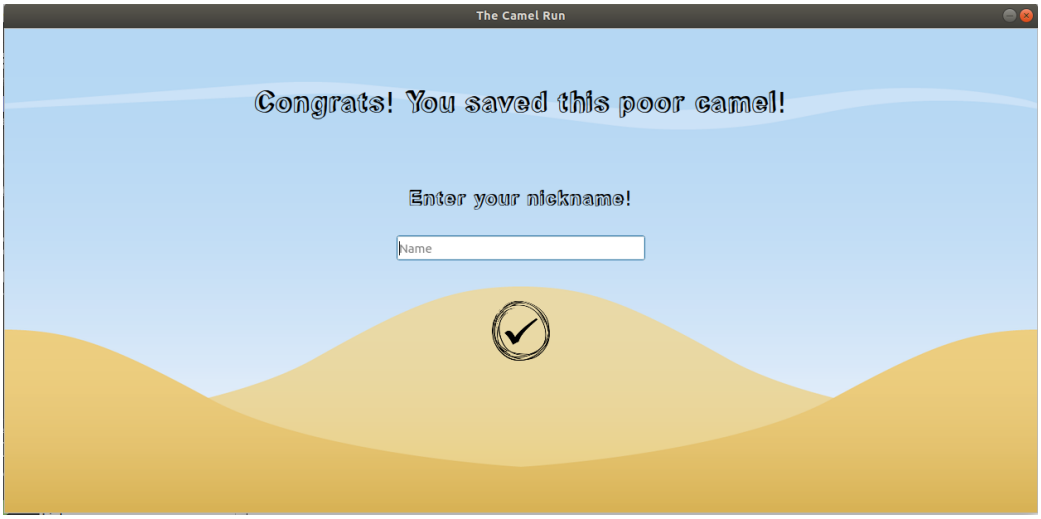
```
connect(mGameScene, &GameScene::healthBarChanged,
        mGameScene, &GameScene::updateHealthBar);
connect(mGameScene, &GameScene::coinGathered,
        mGameScene, &GameScene::updateCoinCounter);
```

Tree klasa

`Tree` nasljeđuje `QGraphicsPixmapItem`. Instanca ove klase nagoviješta kraj levela te predstavlja oazu.

Slika 3.13: Završetak levela

U trenutku kada deva prođe kroz instancu klase `Tree`, pokreće se nova scena, `WinningScene`.



Slika 3.14: Scena uslijed uspješno pređenog levela

Nakon unosa imena te pritiska gumba za potvrđivanje, započinje ili sljedeći level ili je igra završena te je korisnik vraćen na glavni izbornik. U slučaju da je korisnik izgubio igru te nije prešao level iscrtava se ekran sa slike 3.15.

Slika 3.15: Scena uslijed gubljenja života

Tada je nakon pritiska bilo koje tipke tipkovnice korisnik preusmjeren na glavni izbornik. Postupak preusmjeravanja na glavni izbornik implementiran je u 3 koraka, prikazan u isječcima koda 3.11,3.12 i 3.13 :

- slanje signala
- obrada događaja u utoru
- povezivanje signala i utora.

Isječak koda 3.11: Slanje signala pritiskom na tipku

```
1 void LosingScene::keyPressEvent(QKeyEvent *event)
2 {
3     emit backToMenu();
4 }
```

Isječak koda 3.12: Implementacija utora

```
1 void GameManager::startMenu()
2 {
3     renderScene(mMenuScene);
4 }
```

Isječak koda 3.13: Povezivanje signala i utora u konstruktoru klase GameManager

```
1 connect(mLosingScene, &LosingScene::backToMenu,
2         this, &GameManager::startMenu);
```

BackgroundItem klasa

Instancama `BackgroundItem` klase, dopunjuje se scena. To su grafički objekti koji nemaju funkcionalnu ulogu, već estetsku. Stvaraju ugođaj, odnosno suptilno usmjeravaju korisnika prema cilju.

Generiranje levela

Leveli se razlikuju po broju i razmještaju novčića, neprijatelji i zapreka. Novčići se svaki put generiraju na novim, nasumičnim mjestima, ali unutar dosega skoka deve. Rupe u koje deva može upasti su fiksne te se na određenom levelu uvijek javljaju na istim mjestima. Škorpioni i kaktusi se generiraju polu-nasumično. Kako je pod kojim se deva kreće rascjepkan rupama, generiranje škorpiona i kaktusa donekle je predodređeno rasporedom platforma koje su nastale. Ovisno o levelu instancira se određeni broj škorpion po platformi s nasumičnim položajem unutar platforme. Na isti način se i određeni broj kaktusa instancira po platformi te se nasumično pozicionira, no također u ovisnosti o levelu stvara se više kaktusa većih levela.

Poglavlje 4

Zaključak

U ovom radu ukratko je opisana Qt5 biblioteka i njena podrška za razvoj video igara. Navedeni su i objašnjeni najčešće korišteni oblikovni obrasci u programiranju igara, te njihova poveznica s Qt5 bibliotekom. U svrhu prezentacije funkcionalnosti koje Qt5 nudi, izrađena je i video igra *The Camel Run*. Igra je izrađena u C++ programskom jeziku, te su u radu predstavljene funkcionalnosti i prikaz nekih od implementiranih mogućnosti igre.

Unatoč svojoj usmjerenosti ka razvoju grafičkih sučelja te *cross-platform* aplikacija, Qt se pokazao kao zadovoljavajući alat za izradu jednostavnih 2D video igara. Pružajući sustav za grafičko iscrtavanje, osnove detekcija kolizija, te postavke, uz Qt je vrlo jednostavno implementirati 2D osnovnu video igru. Međutim za izradu složenije igre, kao što su platformeri, gdje je potrebno uspostaviti gravitaciju, čvrstoću tijela i mnoge druge aspekte umjetno stvorenog svijeta, mogućnosti Qt-a su ograničene. Dugo vrijeme razvoja funkcionalnosti koje su uklopljene unutar drugih alata, jedan je od glavnih nedostataka Qt-a u pogledu izrade složenijih video igara.

Bibliografija

[1] W. Wysota and L. Haas, *Game Programming Using Qt*. Packt Publishing Ltd., 2016.

[2] R. Nystrom, *Game Programming Patterns*. Genever Benning, 2014.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Elements of Reusable Object-Oriented Software*. Addison Wesley, 2002.

[4] J. Blanchette and M. Summerfield, *C++ GUI Programming with Qt 4*. Prentice Hall, 2008.

[5] G. Lazar and R. Penea, *Mastering Qt5*. Packt Publishing Ltd., 2016.

[6] M. Krajewski, *High Performance Programing With Qt5*. Packt Publishing Ltd., 2019.

[7] L. Eng, *Hands-On GUI Programming with C++ and Qt5*. Packt Publishing Ltd., 2018.

[8] "Qt Contributors' Summit 2018 wrap-up." `https://www.qt.io/blog/2018/06/13/qt-contributors-summit-2018-wrap`, preuzeto: prosinac, 2019.

[9] "What's new in Qt 5." `https://doc.qt.io/qt-5/qt5-intro.html`, preuzeto: siječanj, 2020.

[10] V. Čačić, "Interpretacija programa — predavanja." `https://drive.google.com/file/d/1Kk9zmnHkKGqBWGmWh5B-UAz_oF7Ha529/view`, preuzeto: siječanj, 2020.

[11] "Qt Based Games." `https://wiki.qt.io/Qt_Based_Games`, preuzeto: siječanj, 2020.

[12] C. Cattiaux and K. Szkudlapski, "Visual C++ RTTI Inspection ." `https://blog.quarkslab.com/visual-c-rtti-inspection.html`, preuzeto: veljača, 2020.

[13] V. Yanev, "Video Game Demographics – Who Plays Games in 2020." https://techjury.net/stats-about/video-game-demographics#gref, preuzeto: veljača, 2020.

[14] "Video Game Industry Overview." https://www.gamingscan.com/gaming-statistics/, preuzeto: siječanj 2020.

[15] "How Qt Signals and Slots Work." https://woboq.com/blog/how-qt-signals-slots-work.html, preuzeto: veljača, 2020.

[16] R. Lerdof, K. Tatroe, and P. MacIntyre, "Programing PHP." https://docstore.mik.ua/orelly/webprog/php/ch06_05.htm, pruzeto: veljača, 2020.

[17] "Video games market." https://www.vanillaplus.com/2018/07/05/40093-video-games-market-worth-music-movies-combined-arent-csps-launching-games-services/, preuzeto: siječanj, 2020.

[18] A. Shrestha, "Creating C++17 enabled Qt projects." https://amirkoblog.wordpress.com/2018/08/14/creating-c17-enabled-qt-projects/, preuzeto: prosinac, 2019.

[19] M. Jurak, "Principi objektnog programiranja i oblikovni obrasci." https://web.math.pmf.unizg.hr/nastava/opepp/Slides/Predavanja/html-noslides/slides-6-1.html, preuzeto, veljača, 2020.

[20] "Qt 5.14." https://doc.qt.io/qt-5/, preuzeto: siječanj, 2020.

[21] "Flyweight Design Pattern ." https://sourcemaking.com/design_patterns/flyweight, preuzeto: veljača, 2020.

[22] ""Introduction to BSP"." https://web.cs.wpi.edu/~matt/courses/cs563/talks/bsp/document.html, preuzeto: veljača, 2020.

[23] M. Haney, ""Design Patterns in Game Programming"." https://www.gamasutra.com/blogs/MichaelHaney/20110920/90250/Design_Patterns_in_Game_Programming.php, preuzeto: prosinac, 2019.

[24] "Angry Birds." https://www.angrybirds.com/, preuzeto:prosinac, 2019.

[25] "Pocket Tanks." http://www.blitwise.com/ptanks.html, preuzeto: prosinac, 2019.

[26] "Super Mario." https://supermariobros.io/, prosinac, 2019.

[27] "Free desert platformer tileset." https://www.gameart2d.com/free-desert-platformer-tileset.html, preuzeto: rujan, 2019.

[28] "Scorpion cartoon in shades of rust and sepia." https://en.wikipedia.org/wiki/File:Scorpion_cartoon_in_shades_of_rust_and_sepia.svg, preuzeto: rujan, 2019.

[29] "Cute camel." https://favpng.com/png_view/cute-camel-bactrian-camel-cartoon-clip-art-png/TauxCi99, preuzeto: rujan, 2019.

[30] "Font asset." https://www.1001freefonts.com/., preuzeto: rujan, 2019.

Sažetak

Kroz ovaj diplomski rad predstavljena je Qt5 biblioteka, osnovni principi i mehanizmi koje implementira, te njena podrška za razvoj video igara. Objašnjen je pojam oblikovnih obrazaca, te njihova primjena u programiranju igara. Najčešće korišteni oblikovni obrasci u video igrama su detaljno objašnjeni te je prikazana poveznica s Qt5 bibliotekom. Kao demonstracija mogućnosti Qt5 biblioteke u razvoju video igara, izrađena je 2D igra koja je predstavljena u sklopu diplomskog rada. Navedene su prednosti i mane korištenja Qt5 biblioteke u svrhu izrade video igre.

Ključne riječi: Qt5, video igre, oblikovni obrasci

Summary

This master thesis presents the Qt5 library, the basic principles and mechanisms it implements, and its support for video game development. The concept and application of programming design patterns in game programming are explained. The most commonly used patterns in video games are explained in detail and are presented with connections to the Qt5 library. As a demonstration of the capabilities of the Qt5 library in video game development, a 2D game was created and presented as part of the thesis. In the end, the master thesis presents a discussion about the advantages and disadvantages of using the Qt5 to create a video game.

Key words: Qt5, video games, design patterns

Životopis

Sara Pužar, rođena je 15. kolovoza 1995. godine u Puli. Školovanje započinje 2002. godine u Osnovnoj školi Veli Vrh te završava 2010. kada upisuje prirodoslovno-matematički smjer u Gimnaziji Pula. 2014. godine završava srednjoškolsko obrazovanje te upisuje nastavnički smjer sveučilišnog studija matematike na Prirodoslovno-matematičkom fakultetu u Zagrebu.

Preddiplomski studij završava 2017. godine uz priznanje za izvrsnost Matematičkog odsjeka na PMF-u kada dobiva i titulu sveučilišne prvostupnice edukacije matematike. Iste godine, 2017., upisuje diplomski sveučilišni studij računarstva i matematike za čije vrijeme sudjeluje u studentskim praksama u firmama Mireo i Syntio.