Jak szybko i w prosty sposób zaprojektować aplikację przenośną. Optymalizacja

Wbrew powszechnej opinii, tworzenie aplikacji na wszelkie dostępne systemy operacyjne i urządzenia nie jest ani kosztowne, ani czasochłonne, ani trudne. Dodatkowo, dzięki rozwojowi inżynierii oprogramowania, nie jest to także trudne.

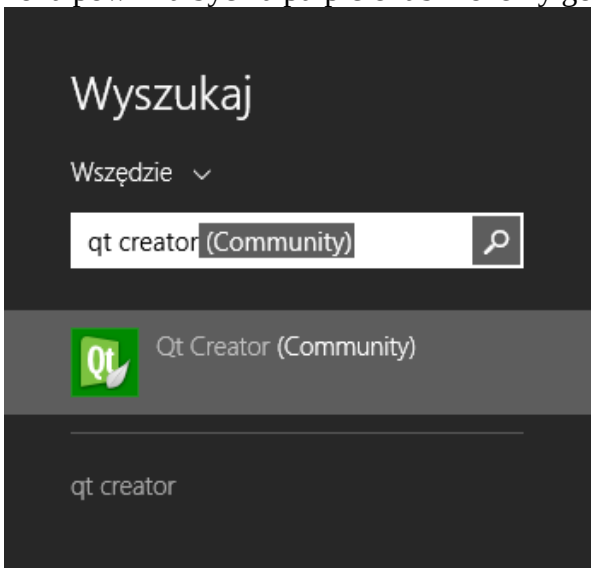
Poniższe ćwiczenie zostanie zrealizowane przy wykorzystaniu bibliotek Qt. Biblioteki te zostały zaprojektowane z myślą o języku C++, jednak można spotkać ich implementację także w innych językach. Siłą bibliotek jest fakt, że ich podstawowa wersja jest zupełnie darmowa i w zasadzie niczym nie różni się ona od wersji płatnej (poza brakiem kilku dodatków, bez których można spokojnie pracować). Ponadto raz napisana aplikacja może być uruchamiana, po wykonaniu kompilacji, na dowolnych systemach operacyjnych i urządzeniach – Windows, Linux czy OS X, Windows Phone, Android czy iOS.

Qt pozwala w bardzo łatwy i przyjemny sposób tworzyć aplikacje dzięki językowi QML. Został stworzony z myślą o tworzeniu aplikacji ze wsparciem dla ekranów dotykowych, można w nim jednak tworzyć także programy dla standardowych ekranów. Generalnie język przypomina składnią CSS (HTML5), same funkcje to natomiast czysty język skryptowy JavaScript, również wykorzystywany na stronach WWW. Dzięki temu każdy, kto potrafi tworzyć strony internetowe bez problemu powinien poradzić sobie z napisaniem prostej aplikacji.

I. Podstawy tworzenia aplikacji

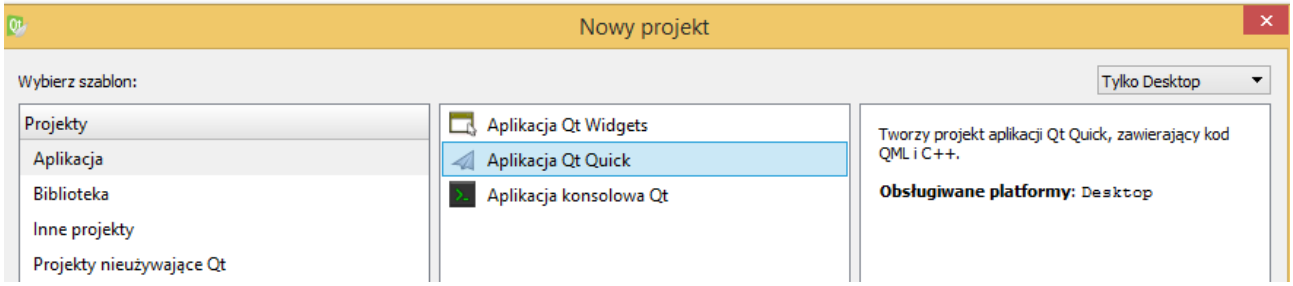
W ramach przykładu utworzymy prostą, acz popularną grę Pamięć (Memory). Naszą aplikację będziemy rozwijać w prostym, acz funkcjonalnym narzędziu o nazwie Qt Creator. W kolejnych punktach przedstawione zostaną pierwsze kroki tworzenia naszej gry.

1. Otwieramy Qt Creator (ikona powinna być na pulpicie lub możemy go wyszukać)

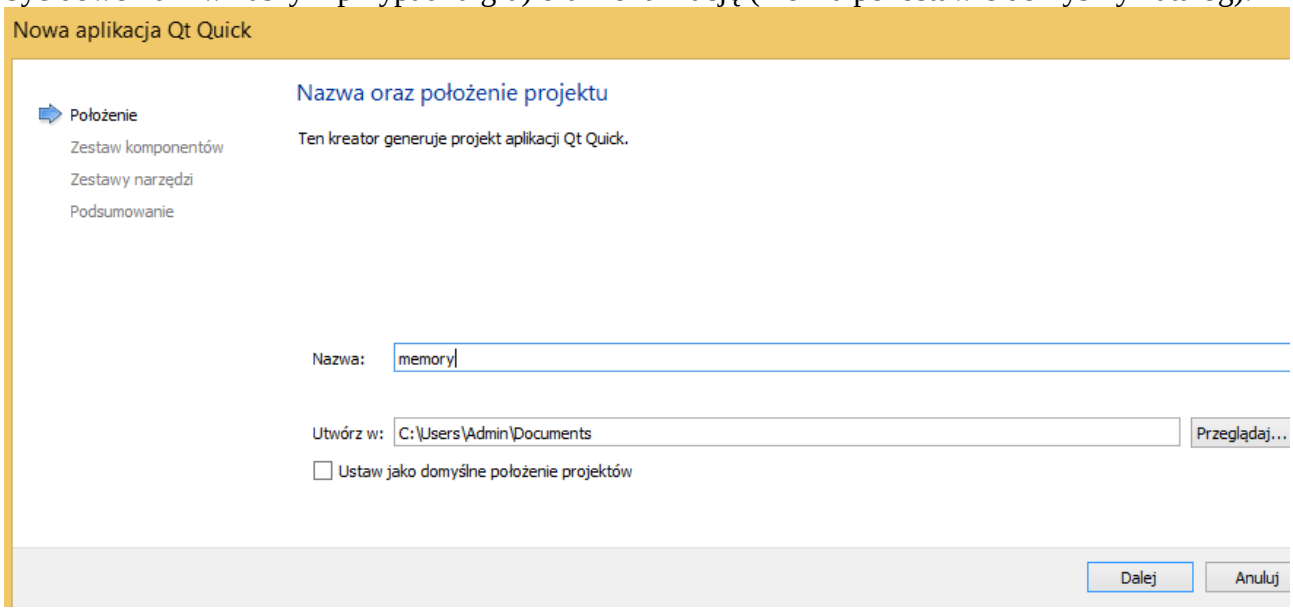


2. W zakładce Powitanie wybieramy „Nowy projekt”

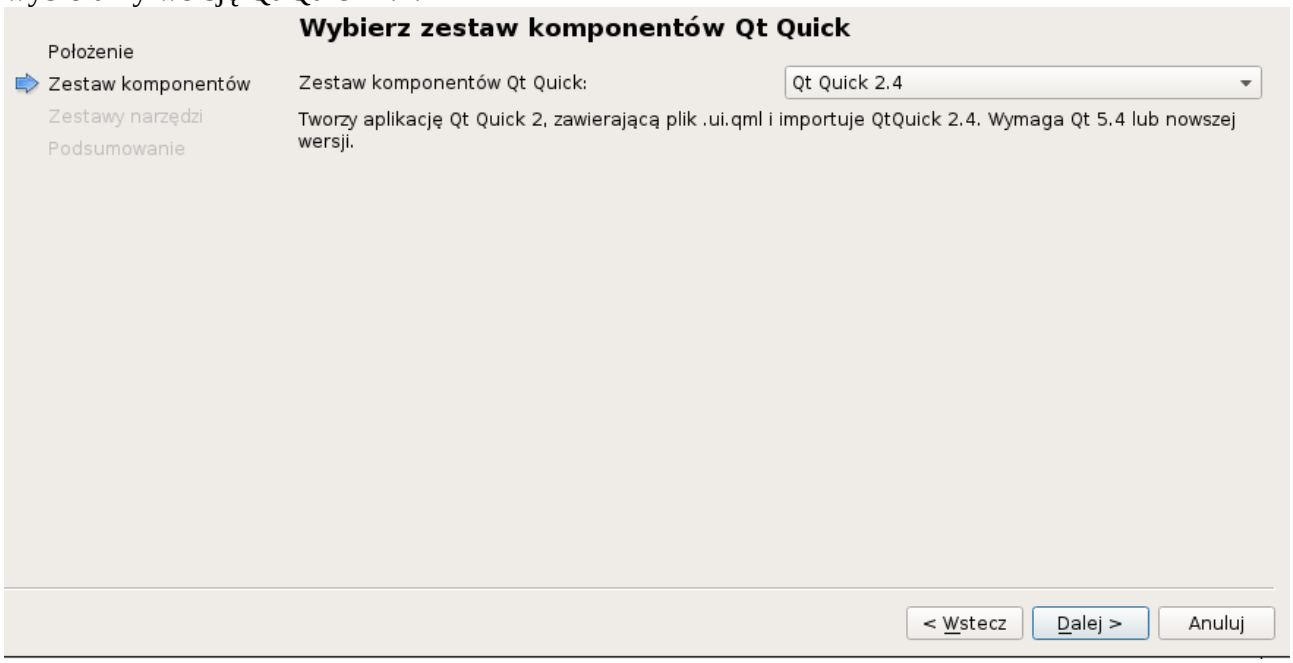
3. Jako nowy projekt wybieramy „Aplikacje”; typ interesującej nas aplikacji jest „Aplikacja Qt Quick”



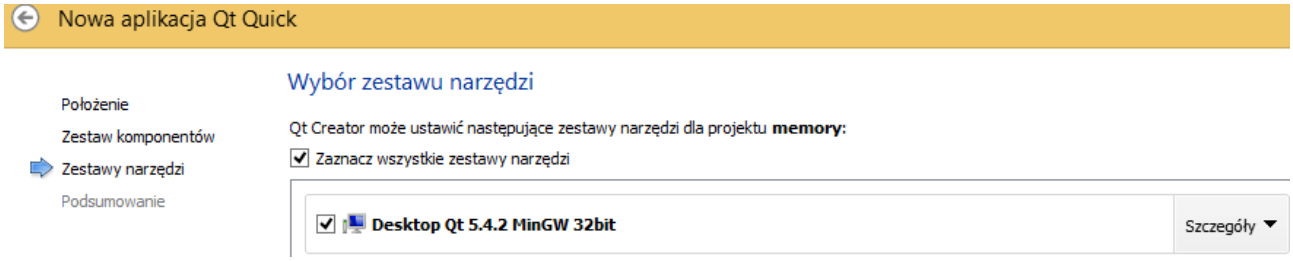
4. Rozpocznie się konfiguracja nowego projektu. Po pierwsze podajemy jego nazwę (nazwa może być dowolna – w naszym przypadku gra) oraz lokalizację (można pozostawić domyślny katalog).



5. Wybieramy zestaw komponentów Qt Quick. Ponieważ tworzymy aplikacje od podstaw wybieramy wersję Qt Quick 2.4.



6. Wybieramy zestaw narzędzi. Tutaj po prostu akceptujemy domyślny wybór.



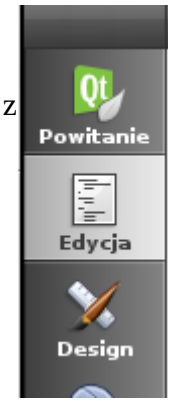
7. Wyświetli się nam podsumowanie tworzonego projektu. Po prostu je akceptujemy.

Mamy już utworzony nowy, czysty projekt. Qt Creator tworzy nam przykładową, prostą aplikację. Możemy ją uruchomić klikając na zieloną strzałkę lub poprzez kombinację klawiszy [CTRL+R]. Jeżeli wszystko jest jak należy, na ekranie powinna pojawić nam się okno naszej aplikacji z napisem Hello World. Jeżeli klikniemy w dowolne miejsce okna to aplikacja zakończy swoje działanie. Czas na wyjaśnienia.

8. Tworzona aplikacja składa się z kilku plików źródłowych. Pierwszym z nich jest plik z rozszerzeniem pro. W nim znajduje się wstępna konfiguracja naszego projektu. W naszym wypadku można pozostawić go nietkniętym.

9. W dalszej części natkniemy się na katalog deployment. W nim znajduje się niejako podprojekt naszej aplikacji odpowiadający za jej poprawne wyświetlanie. Tutaj także nie jest potrzebna nasza ingerencja.

10. Katalog „Źródła” zawiera główny kod naszej aplikacji. Na chwilę obecną znajdziemy w nim tylko plik main.cpp z główną funkcją programu – main. Ponieważ jednak nasz przykład nie będzie wymagał ingerencji w C++, pozostawiamy go w nietkniętym stanie.



11. Folder „Zasoby” to ten, który nas interesuje. To w nim znajduje się plik qml.qrc – dołączane pliki do naszego projektu. Domyślnie są tutaj dwa pliki; main.qml – główny plik programu (uchwyt okna) oraz MainForm.ui.qml, który przejrzymy by zapoznać się z budową aplikacji w QML (i na którym będziemy pracować. Plik nie otworzy się automatycznie do podglądu kodu – trzeba kliknąć na niego, a później na ikonę Edycja. Prześledźmy wskazany plik linia po linii.

```
import QtQuick 2.4                          //1

Rectangle {                                 //2
property alias mouseArea: mouseArea         //3
    width: 360                              //4
    height: 360                             //5
   MouseArea {                              //6
        id: mouseArea                       //7
        anchors.fill: parent                //8
    }
    Text {                                  //9
        text:  "Hello World"                //10
        anchors.centerIn: parent            //11
    }
}
```

//1 – w tej linii importowana jest biblioteka z typami obiektów QML. By wszystko działało jak

należy nie może zostać skasowana; nie powinna też być zmieniania

//2 – tworzony jest pierwszy obiekt QML. Jak jego nazwa wskazuje, jest to po prostu prostokąt. Generalnie w QML typy obiektów mają „przyjazne” nazwy, których brzmienie określa ich przeznaczenie (tzw. semantyka).

//3 – tworzymy tutaj zmienną-alias (odnośnik, odwołanie), która pozwala na używanie wskazanego przez nią elementu (ten po dwukropku) w dowolnym innym miejscu (np. innymi pliku, takim jak main.qml)

//4 – w tej linii określamy szerokość naszego okna. Wartość ta podawana jest w pikselach (punkty wyświetlane na ekranie)

//5 – określamy wysokość; analogicznie jak linia wyżej

//6 – MouseArea (obszar myszy) nie jest elementem wizualnym (nie możemy go zobaczyć). Pozwala on natomiast przechwycić zdarzenia, które użytkownik „tworzy” klikając myszą, przesuwając jej kursor nad elementami aplikacji itp.

//7 – wskazujemy id naszego elementu. Jest to unikatowy identyfikator elementu – NIE MOŻE ISTNIEĆ taki drugi w całym pliku qml. Dzięki temu możemy odwoływać się do zawartości teg elementu w dowolnej linii naszego programu

//8 – zakotwiczenie obszaru elementu wskazane jest jako „wypełnij” (fill). Oznacza to, że ma on objąć dokładnie taki sam obszar, jaki posiada wskazany inny element. Ponieważ elementem wskazywanym jest rodzić (parent) oznacza to, że obszar myszy wypełni nasz prostokąt

//9– tworzymy kolejny obiekt – tekstowy.

//10 – linia ta odpowiada za ustawienie elementu tekstowego na przestrzeni naszego prostokąta. Zakotwiczenie (umiejscowienie) go wskazane jest na punkt środkowy elementu nadrzędnego (jakim jest nasz prostokąt).

//11 – dodajemy tekst, który ma być wyświetlany na ekranie; możemy wpisać każdy tekst, jaki nam się zamarzy.

Jak widać, budowa najprostszego programu nie należy do specjalnie trudnych. Oczywiście nie jest to program, którym moglibyśmy się pochwalić. W przykładzie ukazane zostały raptem 3 z dziesiątek elementów wbudowanych w QML. Przejdźmy więc dalej, zmodyfikujmy aplikację poprzez dodanie kolejnych elementów.

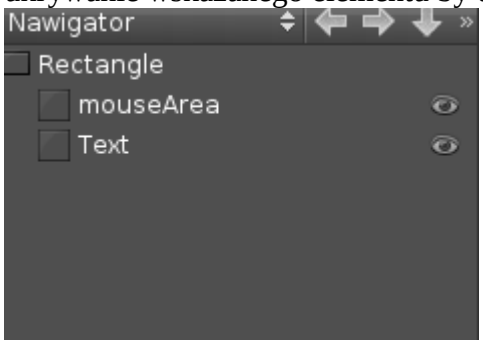
II. Graficzne projektowanie wyglądu programu.

Jeżeli ustawianie elementów w kodzie wydaje się zbyt skomplikowane – nie ma co panikować. Qt Creator posiada bowiem prosty i intuicyjny moduł graficznego projektowania aplikacji. Aby się do niego przenieść należy zaznaczyć plik QML (prawdopodobnie już otwarty) i kliknąć zakładkę „Design”. Teraz spróbujmy zmodyfikować naszą aplikację.

WAŻNE – Proszę zmienić nazwę pliku z MainForm.ui.qml na MainForm.qml.

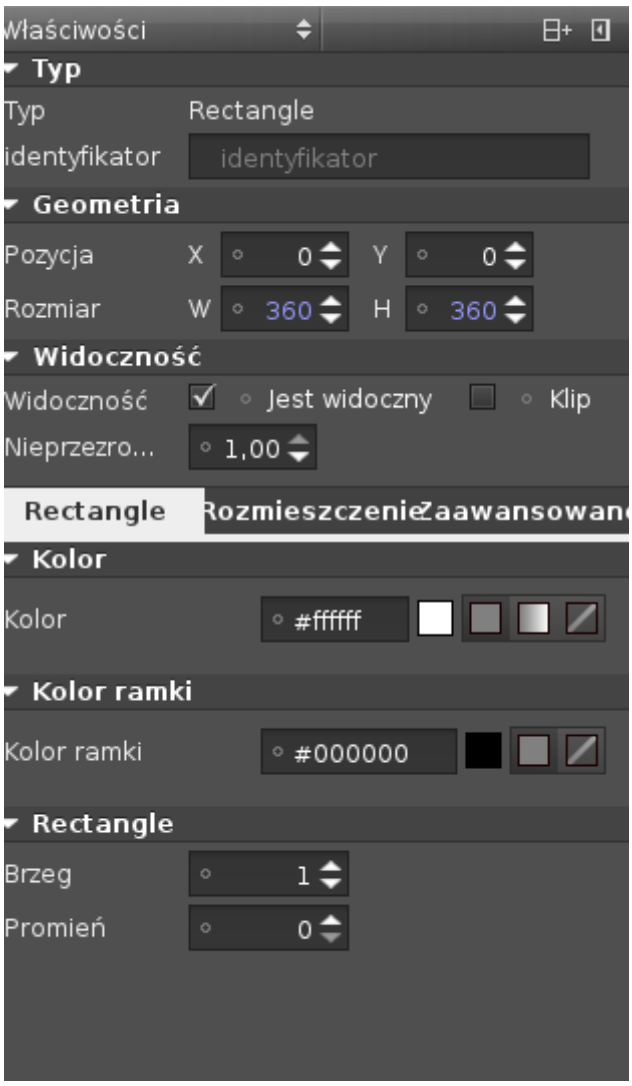
1. Przede wszystkim należy tutaj zwrócić uwagę na fragment okna zatytułowane „Nawigator”. Zawiera bowiem wszystkie elementy aktualnie dodane do naszej aplikacji. Na chwilę obecną mamy

jeden prostokąt (Rectangle), jedno pole tekstowe (Text) oraz jeden obszar myszy (MouseArea). Klikając kolejne elementy na opisywanej liście możemy się pomiędzy nimi przełączać by zmieniać ich właściwości, przesuwać je itp. Widać też zależności pomiędzy poszczególnymi elementami (jeżeli elementy wypisane są po wcięciu to znaczy, że są dziećmi elementu powyżej). Mało z tego, możemy przeciągać i upuszczać poszczególne elementy na inne, by przerzucać je pomiędzy sobą. Dodatkowo ikona oka pozwala ukrywanie wskazanego elementu by ułatwić pracę z innymi.



2. Drugim ciekawym fragmentem okna jest Biblioteka. W niej znajdują się wszystkie aktualnie załadowane elementy QML. Możemy z niej dodawać każdy element, jaki zapragniemy np. poprzez metodę przeciągnij i upuść.

3. W części opisanej „Właściwości” wyświetlać się będą natomiast wszelkie parametry wskazanego elementu. Tutaj możemy ustawić np. wysokość oraz szerokość elementu, jego ułożenie względem pozostałych obiektów oraz kolor, obramowanie czy też wyświetlany tekst.



4. Fragment okna z widoczną nazwą naszego edytowanego pliku to tzw. pasek stanów aplikacji. Tutaj, przy pomocy klikania na kwadrat z plusem, kopiujemy aktualny wygląd naszej aplikacji, po czym możemy zmieniać w nim dowolne rzeczy – np. usuwać dodane elementy, zmieniać ich kolor, rozmiar itp. Wszystkie te zmiany są niezależne względem poprzedniego stanu. Dzięki temu możemy zaprojektować jak nasza aplikacja ma wygląda w przypadku kliknięcia np. przycisku Start aplikacji.

5. Pozostała część okna to nasza przestrzeń robocza, potocznie nazywana pulpitem/stołem montażowym. To tutaj mamy podgląd aktualnie wybranego stanu naszego programu. Tutaj też przeciągamy elementy z biblioteki by dodać je do naszego programu.

Spróbujmy teraz zmienić aplikację pod nasz cel.

6. Klikamy na liście element Rectangle. Pojawią się jego właściwości. Zmieńmy teraz rozmiar okna na 1024 x 726 pikseli. W tym celu we Właściwościach odnajdujemy pozycje Geometria i zmieniamy elementy Rozmiar W (szerokość) na 1024, a H (wysokość) na 726. Od teraz nasz prostokąt stał się większy.

7. Teraz usuńmy elementy, które znajdują się w naszym prostokącie (bo na nic nam się nie przydadzą). W tym celu wybieramy element Text oraz MouseArea jeden po drugim z wciśniętym klawiszem CTRL. Teraz trzeba kliknąć klawisz DELETE na klawiaturze – elementy znikną.

8. Załóżmy, że nasza gra ma mieć łącznie 8 kart – czyli będziemy musieli znaleźć podczas gry 4 pary. Teraz musimy wyobrazić sobie jak będzie wyglądać każda z kart. Przede wszystkim każda „karta" powinna posiadać swój przód i tył. Tak się fortunnie składa, że QML posiada taki element – nazywa się on Flipable. Pozwala on na umieszczenie w sobie dwa różne elementy – jeden z przodu, drugi z tyłu. Elementem osadzonym może być np. obrazek, czyli dokładnie to, czego potrzebujemy.

9. Zacznijmy tworzenie. Wstawmy jeden element Flipable (przeciągnijmy go na nasze okno). Po dodaniu proszę zauważyć, że element ten zawiera w sobie od razu dwa podelementy – front oraz back. Jeden odpowiada za wygląd elementu z przodu, drugi z tyłu. Teraz spróbujmy dodać dwa obrazy.

10. Po pierwsze – musimy mieć obrazy. Na cel ćwiczenia zostaną one dołączone. Oczywiście można wybrać dowolne inne obrazy – wystarczy poszukać w sieci. W naszym folderze znajduje się 5 obrazów – 1-4.jpg (kolejne obrazy do naszej pamięciówki) oraz back.jpg, który będzie na każdej karcie. WSKAZANE JEST dodać folder z obrazami do miejsca, w którym mamy pliki wykonywalne naszego programu (w przypadku programów w wersji testowej katalog debug – zostanie to opisane na końcu materiału).

11. Przeciągamy element Image na nasz projekt. Początkowo nie będzie on dodany do naszego elementu Flipable. Mając już element Image dodany do naszego projektu przeciągamy go w nawigatorze na element back w naszym elemencie Flipable. Od teraz obrazek jest z nim związany. Teraz musimy jeszcze „rozmieścić" nasz obrazek na całej powierzchni elementu obrotowego.

12. Wybieramy dodany obraz Image1. We właściwościach wybieramy zakładkę Rozmieszczenie. Teraz mamy widok na wszystkie możliwe do zastosowania kotwice elementu do nadrzędnego elementu. Nas będzie obchodziła kotwica wypełnienia w elemencie (trzecia licząc od prawej strony). Element powinien rozciągnąć się do poprzedniego.

13. Dodajmy obrazek. W tym celu klikamy zakładkę Image we właściwościach. Znajdziemy tam pole Źródło. Klikamy na przycisk z trzema kropkami i wybieramy obrazek, który ma zostać załadowany do elementu. Nie zobaczymy tego obrazu! Dzieje się tak dlatego, że na chwilę obecną element obrotowy ustawiony jest na widok przodu, nie tyłu.

14. Powtarzamy operację z dodaniem obrazu na przód elementu. Teraz zobaczymy załadowany obrazek.

15. Teraz spróbujmy uruchomić naszą aplikację i sprawdźmy jak ona się zachowuje... Niestety, obraz się nie obraca. Tutaj niestety trzeba dodać trochę kodu – wizualizacja nic nie pomoże. Po pierwsze musimy dodać do naszego elementu Flipable1 obiekt MouseArea. Obiekt ten powinien być dodany jako potomny Flipable lecz nie powinien należeć ani do przodu, ani tyłu. Rozciągamy go na całą powierzchnię.

WAŻNE – może się zdarzyć, że uruchomiona aplikacja będzie znacznie mniejsza niż założyliśmy! Nie ma co panikować bądź, co gorsza, w ogóle się nie uruchomi. Trzeba po prostu wybrać plik main.qml, skasować wszystko co się w nim znajduje, i wkleić zamiast tego taki kod:

```
import QtQuick 2.4
import QtQuick.Window 2.2
```

```
Window {
   visible: true
   width: 1024
   height: 726
   MainForm {

   }
}
```

Od tego momentu wszystko powinno działać jak należy.

16. Przechodzimy do edycji kodu. Żeby nasz obrazek się obracał (sprawiając wrażenie karty) musimy dodać nowe zdarzenie dla obszaru myszy. Ponieważ zmiana ma następować przy kliknięciu myszy na kartę toteż dodajemy zdarzenie onClicked.

17. Teraz jedna ważna sprawa – musimy wiedzieć, kiedy nasza karta ma się przewrócić. Musimy więc 'zapamiętywać' która strona jest aktualnie w użyciu. W tym celu dodajemy nową zmienną o wartości true (prawda) lub false (fałsz). Zmienne deklaruje się w QML nastepująco:

```
property bool flipped: false
```

Powyższa linia mówi programowi, że ma utworzyć nową właściwość (w tym wypadku zmienną). Właściwość ta jest typu bool - może przyjmować więc albo wartość prawdziwą (true), albo fałszywą (false). Trzeci człon to nazwa zmiennej – może być dowolna (nie można używać jedynie nazw poleceń i słów zastrzeżonych w języku). Po dwukropku natomiast umieszcza się wartość początkową zmiennej (można pozostawić zmienną bez dwukropka i wartości początkowej; w naszym wypadku będzie ona jednak potrzebna).

18. Utworzoną zmienną najlepiej jest umieścić zaraz na początku definicji naszego elementu, np. zaraz po właściwości id.

19. Następnym krokiem będzie dodanie „przekształcenia” (transform) naszej karty tak, by wyświetlał się domyślnie jej tył (a nie jak dotychczas przód). Wymagać to będzie od nas dodania nowego elementu do karty, kryjącego się pod właściwością transform. Oto jego kod:

```
transform: Rotation {
                id: rotation1
                origin.x: flipable1.width/2
                origin.y: flipable1.height/2
                axis.x: 0; axis.y: 1; axis.z: 0
                angle: 180
            }
```

Jak to działa? Po pierwsze obiekt Rotation MUSI posiadać swoje id.

Id w QML spełnia bardzo ważną rolę – pozwala jednoznacznie odwołać się do wskazanego elementu z dowolnego punktu naszego pliku qml (niezależnie od położenia). Warunek jest taki, że nie może być dwóch id z dokładnie taką samą nazwą, numerem, opisem. Muszą być jednoznaczne (czyli unikatowe). Id można nadawać każdemu elementowi QML (chociaż nie zawsze trzeba).

Każdy obrót musi mieć swój punkt zaczepienia (początek działania na osi XYZ). W powyższym kodzie punktem tym staje się środek naszego elementu nadrzędnego (jakim jest

flipable1).

Przedostatnia linia to tak naprawdę 3 OSOBNE DEFINICJE (jest to przykład, że i tak można definiować wartości właściwości). Obiekt axis określa osie w przestrzeni XYZ, po których ma nastąpić przekształcenie. Ponieważ chcemy, aby karta obracała się w pionie – musi obracać się wedle osi Y (natomiast NIE MOŻE obrać się po osiach XZ). Dlatego do zmiennych axis x oraz z ustawiona została wartość 0, a pod wartość y 1.

Ostatnia właściwość, kąt (angle), określa o ile ma zostać obrócony nasz element. Domyślnie obrót naszego elementu Flipable wynosi 0 stopni. My jednak chcemy, by karta pokazywała swoje plecy – dlatego na powyższym przykładzie została tam ustawiona wartość 180.

20. Niestety – to nie wszystko. Teraz po uruchomieniu programu karta obrócona jest zawartością na dół. Nie da się jej natomiast obrócić – pomimo że na nią klikamy. Sprawę można rozwiązać na kilka sposobów. Tutaj zostanie pokazane rozwiązanie, jakie zalecają twórcy języka. Dodajemy taki kod pod właściwością transform (dodaną wcześniej):

```
states: State {
            name: "flipCard"
            PropertyChanges { target: rotation1; angle: 0 }
            when: flipable1.flipped
        }
```

Właściwość states może zawierać wiele stanów (stąd nazwa states). Każdy stan musi posiadać swoją jednoznaczna nazwę (name). Nazwa jest dowolna; w przykładzie stan został nazwany flipCard jednak można go nazwać w dowolny inny sposób. W drugiej linii mamy kolejny podelement o nazwie PropertyChanges. Element ten wskazuje na cel naszego działania (w tym wypadku na id elementuRotation oraz na właściwość, jaką w nim zmieniamy (w tym wypadku tylko kąt obrotu). Ponieważ ustawiliśmy wcześniej wartość 180 (o tyle nasza karta została obrócona względem osi y) to teraz zmieniamy ten obrót na 0 stopni (stan początkowy karty – czyli jej przód).

Ostatnia linia (właściwość when) mówi naszej aplikacji kiedy dodany stan ma zadziałać – a ma zadziałać dokładnie w chwili, kiedy wcześniej ustawiana przez nas zmienna będzie miała wartość true. Tutaj posłużono się pewnym skrótem, bardzo chętnie stosowanym w informatyce. Linia

```
when: flipable1.flipped
```

jest równoważna zapisowi

```
when: flipable1.flipped == true
```

Jak widać, oszczędzamy sobie na pisaniu niepotrzebnej wartości. Jeżeli opisywana zmienna zmieni swoją wartość na false opisywany stan przestanie być aktywny i karta wróci do swojej pierwotnej pozycji.

21. Możemy sprawdzić działanie programu. Karta zacznie zmieniać swoją zawartość (obróci się) ale robi to natychmiastowo, przez co nam z kolei wydaje się ta zamiana bardzo nienaturalna. Żeby to zmienić trzeba dodać jeszcze jedną właściwość:

```
transitions: Transition {
                NumberAnimation { target: rotation1; property: "angle"; duration: 400 }
            }
```

Podobnie jak to miało miejsce przy stanach, tak i tutaj właściwość przejścia (transitions) może

zawierać kilka obiektów opisujących przejścia poszczególnych elementów. W naszym wypadku jest to jeden element Transition (Przejście). Zawiera on z kolei jeden element NumberAnimation (AnimacjaLiczbowa). Animacja ta jest stosowana w przypadku zmian wartości liczbowych (tutaj mamy z taką do czynienia – wartość obrotu karty).

Proszę zauważyć, że animacja jak swój cel (target) obiera element obrotu karty (rotation1) i reaguje tylko na właściwość kąta (angle). Ostatnia właściwość, czas trwania (duration), został ustaiony na 400 ms. Jeżeli chcilibyśmy by animacja byłą wolniejsza/szybsza trzeba modyfikować właśnie ten parametr.

22. Ostatnia, bardzo ważna sprawa. W naszym przypadku właściwości id takich elementów jak Image czy MouseArea zupełnie się nie przydadzą, a wręcz będą przeszkadzać. Zostały one dodane przez edytor graficzny. Usuńmy je po prostu.

Można rzecz, że właśnie skończyliśmy tworzyć pierwszą kartę w grze. Teraz wystarczy skopiować tak utworzoną kartę siedem razy. Na poszczególnych kartach należy zmieniać obrazki (żeby zabawa miała sens). Karty możemy kopiować poprzez edytor graficzny lub poprzez kod (znacznie lepszy pomysł). W przypadku kodu musimy jedynie pamiętać o zmianie parametrów x, y oraz wszystkie id. SZCZEGÓLNIE WAŻNE jest podmienienie wartości id we wszystkich elementach Rotation, State Transition oraz MouseArea (np. z flipable1 na flipable2 itd.)

III. „Ożywienie” programu poprzez elementy sprawdzające i decyzyjne.

Wygląd to nie wszystko. W tej chwili nasze karty odwracają się lecz nic poza tym się nie dzieje. Zmodyfikujemy aktualny kod tak by wszystko zaczęło działać.

1. Dodajmy do naszego głównego prostokąta (najlepiej przed właściwościami width oraz height) następujący fragment kodu:

```
id: mainRectangle
   property int pairs: 4
   property int steps: 0
   property int lastChoose: -1
```

Po pierwsze nasz główny prostokąt zyskał id (będziemy się d niego odwoływać). Kolejne trzy właściwości:
- pairs określa ile mamy par w grze do odkrycia (8 kart to 4 pary)
- steps będzie zmienną przechowującą ilość kliknięć użytkownika na karty (aby było wiadome ile razy musiał kliknąć nim odnalazł pary); wartość będzie zwiększana o jeden za każdym wywołaniem zdarzenia onClicked
- lastChoose to zmienna pomocnicza. Jeżeli gracz kliknie na jakąś kartę to zmienna ta będzie przechowywać jej numer. Gdy gracz kliknie na innej karcie – zmienna porówna jej numer z numerem przechowanym pod tą zmienną. W przypadku identycznej wartości program będzie wiedział, że została odkryta para albo, że karty nie są parą i zmieni jej wartość z powrotem na -1.

2. Pod ostatnim elementem (ostatnią kartą) dodajemy takie dwa elementy tekstowe:

```
Text {
      id: moveText
      anchors.bottom: parent.bottom
      anchors.left: parent.left
      anchors.bottomMargin: 20
      anchors.leftMargin: 20
```

```
        font.pointSize: 17

        text: qsTr("Aktualnie wykonałeś ruchów: " + mainRectangle.steps)
    }

    Text {
        id: pointText
        anchors.bottom: parent.bottom
        anchors.right: parent.right
        anchors.bottomMargin: 20
        anchors.rightMargin: 20
        font.pointSize: 17

        text: qsTr("Pozostało par: " + mainRectangle.pairs)
    }
```

UWAGA! Elementy można było wstawić w dowolnym momencie (nawet na początku okna). Chodzi jedynie o przejrzystość kodu!

Pierwszy z nich odpowiada za wyświetlanie aktualnej ilości kliknięć jakich dokonał użytkownik. Drugi – ile par gracz musi jeszcze odkryć. Z nowości jakie zostały tutaj dodane są 2 właściwości:
- anchors.rightMargin/anchors.bottomMargin, itp. - właściwości te pozwalają na przesunięcie elementu od wskazanych krawędzi (lewo, prawo, góra, dół) o określoną ilość pikseli.
- font.pointSize – zmienia wielkość czcionki na określoną ilość punktów. Punkty to domyśla wielkość czcionki w niemal wszystkich programach tekstowych (włączając w to edytory tekstowe) i równa się 1/72 cala.

3. Napisy się już wyświetlają, jednak nie zmieniają się wartości w nich wyświetlane. Żeby tego dokonać trzeba dodać do każdej funkcji onClicked (we wszystkich kartach) taką linię:

```
steps++;
```

powyższy zapis jest równoważny:

```
steps = steps + 1;
```

czyli, innymi słowy, zmieniamy wartość naszej zmiennej steps o jeden przy każdym kliknięciu na kartę.
Wygląd zmodyfikowanego zdarzenia onClicked w pierwszej karcie:

```
onClicked: {
            steps++;
            flipable1.flipped = !flipable1.flipped;
}
```

Teraz powinniśmy, za każdym kliknięciem karty, zobaczyć zwiększającą się ilość kliknięć.

4. Teraz trzeba dodać trochę logiki wynajdywania par. Po pierwsze musimy zacząć wykrywać czy któraś z kart nie została aktualnie odwrócona. W tym celu do funkcji onClicked musimy dodać następujący kod:

```
onClicked: {
```

```
            steps++;
            flipable1.flipped = !flipable1.flipped;
            if (mainRectangle.lastChoose == -1)                    //1
                mainRectangle.lastChoose = 1;
            else if (flipable1.flipped) {                          //2
                if (flipable1.pair == mainRectangle.lastChoose) {
                    mainRectangle.pairs--;
                }
                mainRectangle.lastChoose = -1;
            }
            else                                                   //3
                mainRectangle.lastChoose = -1;
        }
```

W powyższym kodzie założono, że każda z par kart ma swój numer-identyfikator – numeracja odwzorowuje numery obrazów kart.

Kod zaznaczony na czerwono:

//1 - w tym punkcie sprawdzamy, czy zmienna lastChoose ma ustawioną zmienną -1 (żadna karta nie była wybrana). Jeżeli tak, to nadajemy tejże zmiennej wartość pary kart, której jedna została odkryta.
//2 – ten punkt wykona się w przypadku, gdy //1 zwróci fałsz – zmienna będzie miała nadany numer inny od -1. Sprawdzamy tutaj czy nasz użytkownik odkrył kartę. Jeżeli tak to sprawdzamy czy numer aktualny karty równy jest karcie ostatnio odsłoniętej. W przypadku prawdy – zmniejszamy ilość par do odkrycia. Niezależnie od drugiego warunku zmieniamy stan zmiennej lastChoose na -1.
//3 – linia ta została dodana na wypadek gdyby żaden z warunków nie został wykonany. Jeżeli tak właśnie się zdarzy należy zmienić zawartość zmiennej lastChoose na -1. Inaczej gra mogłaby nie działać prawidłowo.

Proszę zauważyć, że w powyższym kodzie pojawiła się zmienna pair w odniesieniu do naszej karty. Oznacza to, że musimy taką zmienną dodać. Najlepiej dodać ją zaraz pod id:

```
    Flipable {
        id: rectangle1
        property int pair: 1
//reszta kodu
```

Wartość zmiennej pair w tym wypadku równa się numerowi załadowanego obrazu – NIE KARTY.

Jeżeli wszystko zostało poprawnie ustawione to odpowiednie sparowanie kart powinno odliczać nam pary do końca gry.

5. Trzeba przyznać, że nadal nasza „gra” nie przypomina do końca gry – odliczanie par do końca to nie wszystko. Przede wszystkim powinniśmy odwracać karty w przypadku niesparowania ich. Karty sparowane powinny natomiast znikać by nie klikać ich ponownie. Wreszcie na koniec gry powinniśmy dostawać stosowny komunikat. Zacznijmy od odwracania kart w przypadku nie trafienia w parę.
Na początku musimy dodać odpowiedni kod odpowiadający za obracanie obu kart. Zrealizujemy to poprzez dodanie nowej funkcji do elementu mainRectangle (np. przed pierwszym elementem Flipable):

```
 function flipCards() {
```

```
        var i;
        for(i = 0; i < mainRectangle.children.length; i++) {
            if (mainRectangle.children[i].flipped) {
                mainRectangle.children[i].flipped = false;
            }
        }
    }
```

Jak funkcja działa? Tutaj wykorzystywana jest zasada, że nasz główny prostokąt jest rodzicem dla kolejnych elementów w nim tworzonych. Wszystkie te elementy (karty) są dostępne pod zmienną children (dzieci). Dzieci są tutaj w postaci zmiennej tablicowej (listy). Zmienna ta więc posiada, w postaci listy, dostęp do wszystkich naszych obiektów!
Funkcja stosuje tutaj jedną z podstawowych pętli for; pętla ta ma za zadanie wykonywać określoną liczbę razy instrukcje znajdujące się w jej ciele. Posiada ona trzy parametry.

Pierwszy z nich to najczęściej liczba całkowita, od której rozpoczyna się liczenie. W naszym przypadku rozpoczynamy liczenie od 0.

Drugi to warunek do kiedy funkcja ma się wykonywać w naszym wypadku chcemy wykonywać kod pętli for do chwili, kiedy wartość zmiennej i jest mniejsza od ilości elementów na liście. Ostatni parametr to równanie o ile ma zwiększać się zmienna i przy każdorazowym powtórzeniu pętli (w naszym wypadku zwiększa się o jeden).

W ciele pętli mamy sprawdzenie (if) czy element posiada zmienną flipped oraz czy zmienna ta ma wartość true (sprawdzanie dwa w jednym). Jeżeli tak, to zmieniamy wartość tejże zmiennej na false, co wiąże się bezpośrednio ze zmiana stanu obiektu na początkowy (odwrócona karta).

Funkcja sama z siebie się nie wywoła. To my decydujemy kiedy wykonają się zawarte w niej instrukcje. Ponieważ mają się wykonać tylko wtedy, kiedy nie trafimy w parę kart należy odpowiednio zmodyfikować zdarzenie onClicked (pokazane na przykładzie z pierwszej karty):

```
onClicked: {
            steps++;
            flipable1.flipped = !flipable1.flipped;
            if (mainRectangle.lastChoose == -1)
                mainRectangle.lastChoose = 1;
            else if (flipable1.flipped) {
                if (flipable1.pair == mainRectangle.lastChoose) {
                    mainRectangle.pairs--;
                }
                else
                    flipCards();
                mainRectangle.lastChoose = -1;
            }
            else
                mainRectangle.lastChoose = -1;
        }
```

I to wszystko. Jeżeli druga karta nie będzie należała do pary – wszystkie karty zostaną obrócone. Powyższą zmianę trzeba zaaplikować we wszystkich elementach.

6. Teraz powoli wszystko zaczyna wyglądać w miarę w porządku. Problem wzbudza jedynie moment, gdy odkryjemy parę – karty zostają i w zasadzie nic nie możemy z nimi zrobić. Rozwiązanie jest proste – należy je „usunąć” czyli po prostu przestać je wyświetlać. Posłużymy się dodaniem kolejnej funkcji:

```
function dropCards() {
    var i, j;
    for(i = 0; i < mainRectangle.children.length; i++) {
        if (mainRectangle.children[i].pair === lastChoose)
            mainRectangle.children[i].visible = false;
}
```

Funkcja jest modyfikacją poprzedniej flipCards(). W tym wypadku, jeżeli numer pary karty jest identyczny z tym ostatnio wybranym to widoczność tejże karty (visible) ma zostać ustawiona na false (czyli obiekt ma być niewidoczny). Wraz z niewidocznością obiekt przestaje reagować na naciśnięcia, to z kolei eliminuje go z gry. Wystarczy teraz dodać odpowiednia funkcję do onClicked:

```
onClicked: {
            steps++;
            flipable8.flipped = !flipable8.flipped;
            if (mainRectangle.lastChoose == -1)
                mainRectangle.lastChoose = 2;
            else if (flipable8.flipped) {
                if (flipable8.pair == mainRectangle.lastChoose) {
                    mainRectangle.pairs--;
                    dropCards();
                }
                else
                    flipCards();
                mainRectangle.lastChoose = -1;
            }
            else
                mainRectangle.lastChoose = -1;
        }
```

i gotowe!

7. Karty się odsłaniają, pary są eliminowane, ruchy są liczone... wydawać by się mogło, że to już koniec. Jednak nie! Na końcu gry powinniśmy mieć przecież możliwość zobaczenia wyniku, rozpoczęcia gry na nowo lub wyjścia z niej. To niestety wymaga dodania kolejnych elementów do naszego widoku.

Najprościej będzie dodać nowy element prostokąta:

```
Rectangle {
    id: winInfo
    width: parent.width / 2
    height: parent.height / 2
    border.color: "#ff0000"
    border.width: 2
    anchors.centerIn: parent
    z: 1000
    visible: false
}
```

Nowościami są:

- szerokość (width) liczona jest względem rodzica (parent – nasz mainRectangle). Oznacza to, że ten prostokąt będzie miał połowę szerokości rodzica (dzielona przez dwa)
- analogicznie wysokość (height)
- dodane jest obramowanie (border). Ma ono otrzymać odpowiedni kolor (#ff0000 – po prostu czerwony) oraz ma mieć grubość 2 piksele (by było ładnie widoczne).
- parametr z jest ustawiony na 1000 by mieć pewność, że przykryje wszystkie dotychczasowe obiekty
- widoczność (visible) ustawiona jest na false (obiekt nie będzie widoczny). Tym samym nie będzie nam na razie przeszkadzał.

8. Powinniśmy wyświetlać w naszym oknie tekst w przypadku, gdy gracz wygra. Dodajmy więc odpowiedni element tekstowy to naszego elementu winInfo:

```
Text {
        id: winText
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.top: parent.top
        anchors.topMargin: 50
        font.pointSize: 14
        font.bold: true
        text: qsTr("Wygrałeś wykonując " + mainRectangle.steps + " ruchów!. Chcesz powtórzyć?")
    }
```

Nowością jest właściwość font.bold (czcionka zostaje pogrubiona).

9. Pasowałoby dodać także jakieś przyciski. Tutaj przyciskiem będą po prostu odpowiednio ustawione prostokąty, z polem na przechwycenie myszy:

```
    Rectangle {
        id: restartBtn
        width: 100
        height: 50
        anchors.left: winInfo.left
        anchors.leftMargin: 30
        anchors.bottom: winInfo.bottom
        anchors.bottomMargin: 20
        color: "#0000ff"

        Text {
            id: restartText
            anchors.centerIn: parent
            color: "#ffffff"
            text: qsTr("Ok")
        }

        MouseArea {
            z: 100
            anchors.fill: parent
            hoverEnabled: true
            onClicked: restartGame()
            onEntered: {
                restartBtn.color = "#00ff00";
```

```
            restartText.color = "#ff0000";
        }
        onExited: {
            restartBtn.color = "#0000ff";
            restartText.color = "#ffffff";
        }
    }
}
```

Nowością jest:

- hoverEnabled – właściwość wskazuje, że element ma reagować także na moment, gdy kursor myszy znajduje się nad nim (nie tylko na kliknięcie przycisków).
- przy onClicked pojawia się funkcja restartująca naszą grę (dodamy ją za chwilę)
- zdarzenie onEntered – wywołuje się w chwili gdy mysz znajdzie się nad naszym przyciskiem (zmienia kolor przycisku)
- zdarzenie onExited – wywołuje się w chiwli gdy mysz znajdzie się poza naszym przyciskiem (powrót właściwości).

INFORMACJA – zmiany te można było wykonać poprzez stany, jednak autor miał w zamyśle pokazać także inne metody modyfikacji właściwości elementów.

Trzeba utworzyć także drugi przycisk, zamykający grę:

```
Rectangle {
        id: quitBtn
        width: 100
        height: 50
        anchors.right: winInfo.right
        anchors.rightMargin: 30
        anchors.bottom: winInfo.bottom
        anchors.bottomMargin: 20
        color: "#0000ff"

        Text {
            id: quitText
            anchors.centerIn: parent
            color: "#ffffff"
            text: qsTr("Zamknij")
        }


        MouseArea {
            z: 100
            anchors.fill: parent
            hoverEnabled: true
            onClicked: Qt.quit()
            onEntered: {
                quitBtn.color = "#00ff00";
                quitText.color = "#ff0000";
            }
            onExited: {
```

```
                quitBtn.color = "#0000ff";
                quitText.color = "#ffffff";
            }
        }
    }
```

10. Funkcja do przycisku restartującego grę:

```
function restartGame() {
    var i;
    for(i = 0; i < mainRectangle.children.length; i++) {
        if (mainRectangle.children[i].flipped)
            mainRectangle.children[i].flipped = false;
         mainRectangle.children[i].visible = true;
    }
    pairs = 4;
    steps = 0;
    lastChoose = -1;
    winInfo.visible = false;
}
```

11. Teraz tylko musi zostać wywołane pojawienie się okna z informacją o zwycięstwie. W tym celu dodamy pewne zdarzenie (najlepiej zaraz przy zmiennej, do której zdarzenie będzie się odnosić czyli pairs)

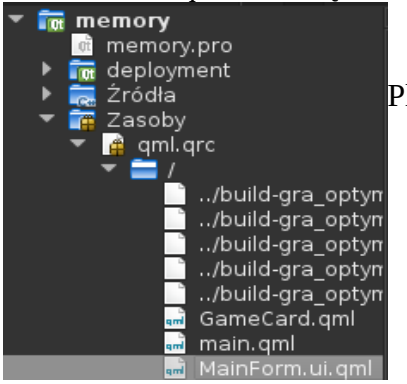
```
onPairsChanged: {
    if (pairs == 0)
        winInfo.visible = true;
}
```

Od tego momentu, gdy liczba par spadnie do zera pojawi nam się wcześniej tworzony prostokąt z informacją o wznowieniu gry.

IV. Czas na prawdziwe tworzenie aplikacji! (bardziej zaawansowane)

Mamy już stworzoną grę. W zasadzie moglibyśmy na tym poprzestać. Jednak czy nasz kod jest odpowiedni? Czy w razie gdybyśmy chcieli dodać nowe karty, albo zmienić ich rozmiar, poszłoby na to łatwo? Odpowiedź brzmi – nie! Głównym powodem jest projektowanie graficzne – tutaj dodawanie kolejnych elementów wiązało się z dodaniem niezależnych nowych elementów. Niestety przy jakiejkolwiek próbie modyfikacji jesteśmy zmuszeni modyfikować wszystkie elementy. To jest zbyt czasochłonne i nieefektywne.

Przede wszystkim musimy zacząć od nowego projektu – ten przedstawiony powyżej jest raczej nierozwijalny (ang. immutable). Mając nowy projekt pierwsze co robimy kasujemy MainForm.ui.qml – nie będziemy się kusić na edycję wizualną.



Plik MainForm.ui.qml, który należy skasować.

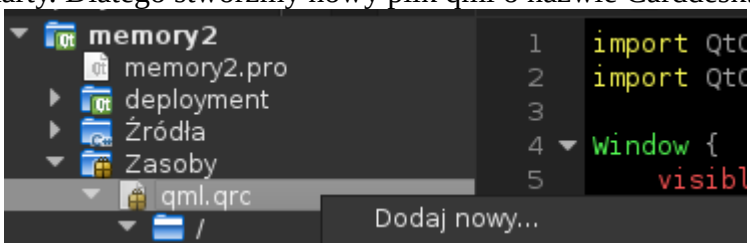
Teraz czas zaplanować jak nasza gra ma się zachowywać. Załóżmy, że użytkownik sam może wybrać ilość par kart, które będzie musiał odkryć. Jako minimum ustalimy 4 pary (8 kart na planszy), jako maksimum zaś 12 (24 karty). Do tego można dorzucić podwaliny wyzwania – maksymalnej ilości kroków, w jakiej możemy odkryć wszystkie karty.

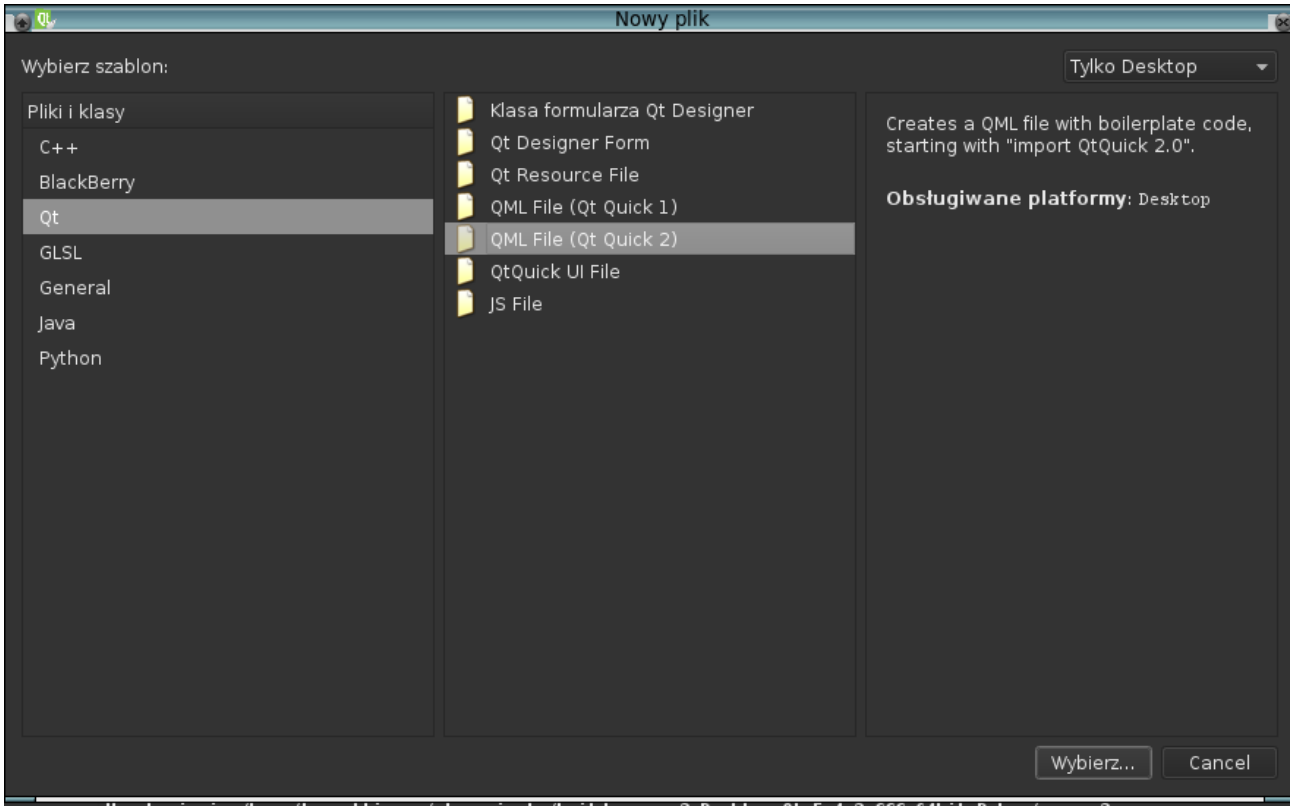
Pierwszym krokiem będzie zdobycie 12 dowolnych wizerunków kart, które chcielibyśmy użyć w grze. Dla potrzeb naszego kodu wybrane zostały losowe obrazki owoców (niekoniecznie zoptymalizowane – udostępnione są jedynie w celach pokazowych).

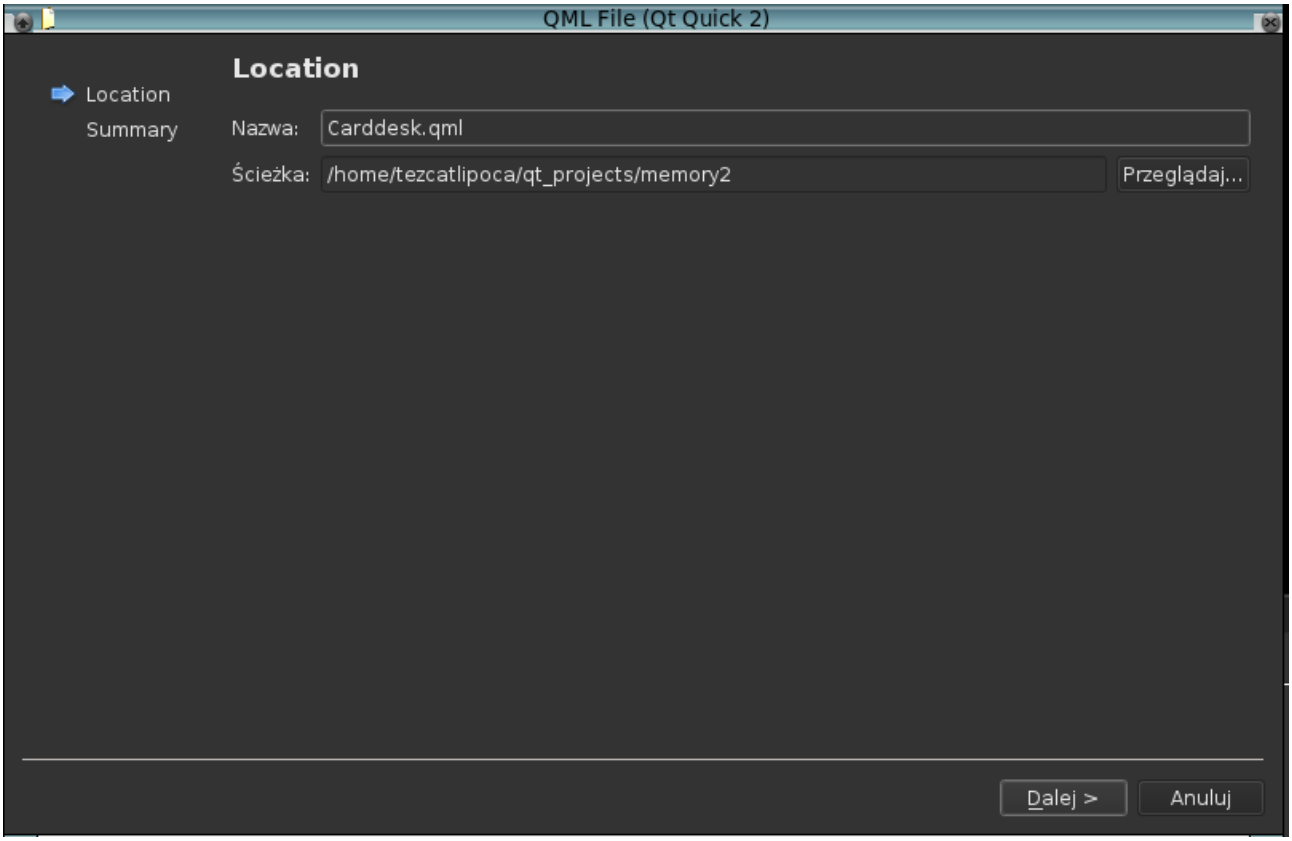
Teram możemy zacząć tworzyć nasz program. Obecnie naszym projekcie mamy tylko plik main.qml (główne okno/widok). Odwołujemy się w nim do zawartości okna głównego, które nie istnieje (skasowany plik qml). Trzeba to zmienić.

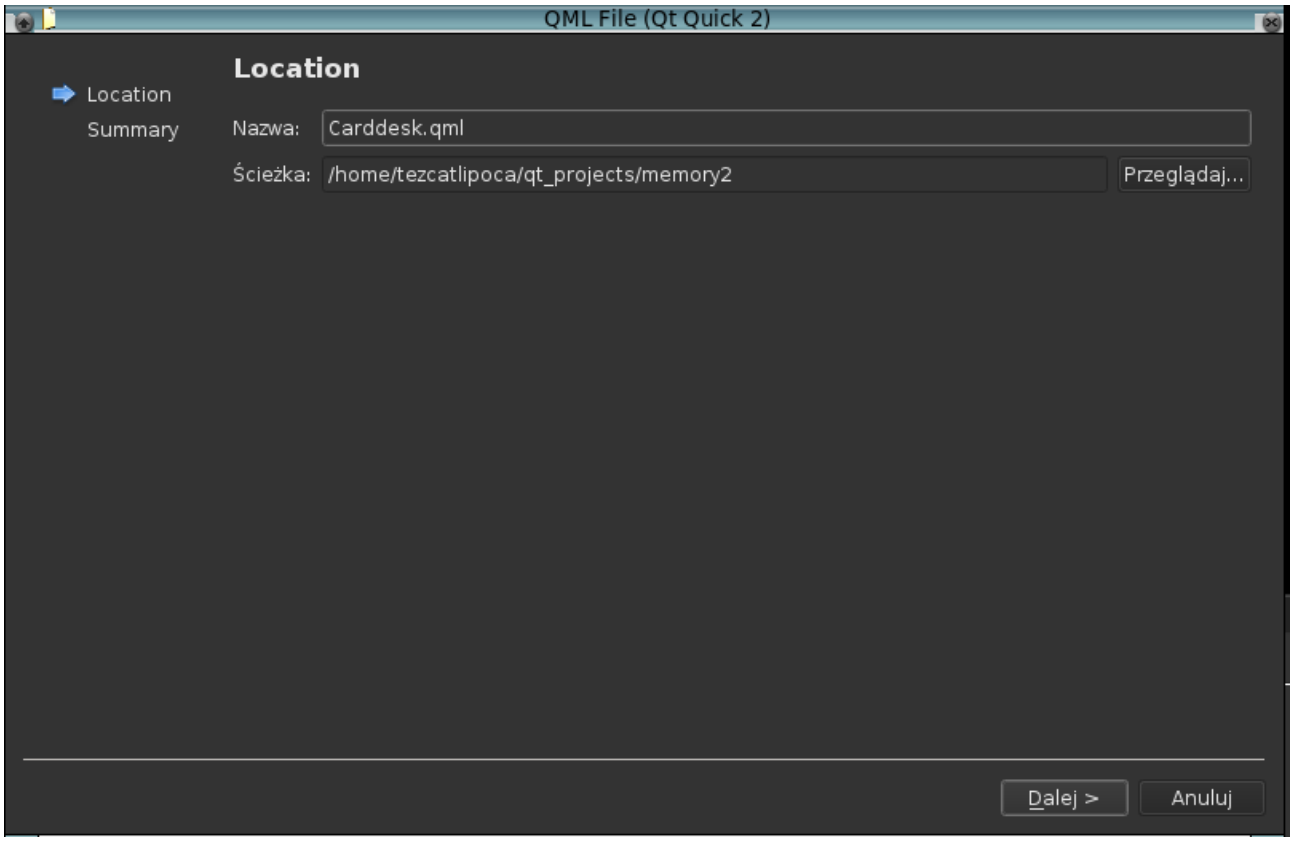
INFORMACJA: Teoretycznie moglibyśmy tworzyć wszystko w jednym pliku. Nie jest to jednak eleganckie rozwiązanie z wielu powodów. Dla nas, początkujących, powód jest jeden – nie robić bałaganu w kodzie by móc jak najlepiej nauczyć się programować!

Gra powinna mieć główny element w postaci przestrzeni, na której będziemy tasować i rozdawać karty. Dlatego stwórzmy nowy plik qml o nazwie Carddesk.qml









Plik ten będzie przez nas wykorzystywany do układania kart, wyświetlania wykonanych kroków przez gracza oraz pozostała ilość par na stole.

W podobny do opisywanego sposobu utwórzmy jeszcze pliki Gamecard.qml oraz Startwindow.qml. Gdy mamy je już wszystkie zacznijmy kodować.

1. Zacznijmy od pliku Gamecard.qml (tworzyliśmy już podobny wcześniej). Po pierwsze utworzymy odpowiednie właściwości (zmienne):

```
property alias cardImg: flipFront.source
property alias cardWidth: flipable1.width
property alias cardHeight: flipable1.height
property bool cardFlipped: false

property int cardPair
```

- cardImg – pozwala na nadanie karcie odpowiedniego obrazka
- cardWidth – pozwala na zmianę szerokości karty
- cardHeight – pozwala na zmianę wysokości karty
- cardFlipped – przechowuje informacje, czy karta ma zostać obrócona obrazkiem (domyślne false mówi, że karta ma być obrócona do nas tyłem)
- cardPair – zmienne przechowuje numer pary drugiej karty na stole (czyli wartość od 0 do 12)

W dalszej części ustalamy

```
id: cardItem
width: flipable1.width
height: flipable1.height
x: 0
y: 0
```

- id naszej karty (czyli unikatową zmienną elementu – pamiętajmy NIE MOŻE SIĘ POWTARZAĆ W OBRĘBIE PROJEKTU!)
- szerokość (width) naszej karty – pobierana jest ona wartości szerokości karty
- wysokość (height) – analogicznie do szerokości
- pozycja na osi x naszego elementu
- pozycja na osi y naszego elementu

Poniżej zaczyna się definicja właściwa naszego elementu (deklarujemy użycie obiektu Flipable). Nie będziemy tutaj prezentować całego kodu (dostępny w postaci zip do niniejszego pdf). Ciekawszym miejscem jest definicja przodu oraz tyłu elementu:

```
front: Image {
    id: flipFront
    anchors.fill: parent
}

back: Image {
    anchors.fill: parent
    source: "back.jpg"
}
```

Przód (front) jaki widać nie posiada domyślnie żadnego obrazu; ma jedynie wypełniać cały element nadrzędny (Flipable). Tył karty zaś ma wypełniać obraz zapisany pod nazwą back.jpg (jeszcze nie dodany, dodamy do za chwilę do projektu – wraz z innymi).

Pozostałe linie były już omawiane wcześniej.

2. Przejdźmy do pliku Carddesk.qml. Zaweira on następujące właściwości (zmienne):

```
property alias deskWidth: cardDesk.width
property alias deskHeight: cardDesk.height

property int lastChoose: -1
property int steps: 0
property int pairsLeft: 4
property int stepsLeft: -1
```

- deskWidth/deskHeight – analogicznie jak przy elemencie karty
- lastChoose – przechowuje wartość pary ostatnio wybranej karty (w przypadku nie wybrania wartość -1)
- steps – ilość ruchów, jaką wykonał gracz
- pairsLeft – ilość par, które pozostały na stole
- stepsLeft – ilość ruchów, jaką gracz może wykonać; jeżeli wartość -1 – nie ma limitu ruchów

Dodane zostały dwa zdarzenia wywoływane bezpośrednio po zmianie pairsLeft oraz stepsLeft:

```
    onPairsLeftChanged: {                                           //1
        pairText.text = pairText.staticText + pairsLeft;
        if (pairsLeft == 0) {
            var c = Qt.createComponent("Startwindow.qml");
            var window = c.createObject(cardDesk);
            window.windowText = "Wygrałeś!\n Ustaw nową grę lub wyjdź.";
        }
    }

    onStepsChanged: {                                               //2
        stepText.text = stepText.staticText + steps;
        if (stepsLeft != -1) {
            stepText.text += "/" + stepsLeft;
            if (stepsLeft <= steps) {
                var c = Qt.createComponent("Startwindow.qml");
                var window = c.createObject(cardDesk);
                window.windowText = "Przegrałeś!\n Ustaw nową grę lub wyjdź.";
            }

        }
    }
```

Pierwsze zdarzenie sprawdza ile pozostało par na stole. Jeżeli, otwiera ono okno startowe, które zawierać będzie odpowiedni komentarz.
Analogicznie działa drugie zdarzenie. Jeżeli ilość kroków jest różna od -1 (brak reakcji) oraz liczba kroków jest mniejsza bądź równa ilości aktualnie wykonanych kroków to następuje wyświetlenie stosownego komunikatu o przegranej i propozycji nowej rozgrywki.

UWAGA: Powyższe zdarzenia (zmiany) generowane są przez QML AUTOMATYCZNIE – dla każdej zmiennej!

Dalej większość funkcji była już wcześniej opisywana (punkt III materiału). Nowością jest ta oto funkcja:

```
function generateDesk(pairs) {
        cleanDesk();
        pairsLeft = pairs;

        var cardWidth = cardDesk.width;
        var cardHeight = cardDesk.height - 100;
        var widthRatio = pairs;
        var heightRatio = 2;
        if (pairs % 8 != 0 && pairs < 8)
            widthRatio = (pairs % 8);
        else if (pairs > 8)
            widthRatio = 8;
        cardWidth /= widthRatio;

        if (pairs > 8)
            heightRatio = 3;
        cardHeight /= heightRatio;
```

```
        var cardsPairTable = [];
        for (var i = 0; i < pairs; i++)
            cardsPairTable[i] = 0;

        for (var i = 0; i < heightRatio; i++) {
            for (var j = 0; j < widthRatio; j++) {
                var component = Qt.createComponent("Gamecard.qml");
                var card = component.createObject(cardDesk);
                var cardPlace = -1;
                do {
                    cardPlace = Math.floor(Math.random() * pairs);
                    if (cardsPairTable[cardPlace] < 2) {
                        cardsPairTable[cardPlace]++;
                    }
                    else
                        cardPlace = -1;

                } while(cardPlace == -1);
                card.objectName = "playCard";
                card.cardPair = cardPlace;
                card.cardWidth = cardWidth;
                card.cardHeight = cardHeight;
                card.x = 5 + (j * cardWidth);
                card.y = 5 + (i * cardHeight);
                card.cardImg = cardPlace + ".jpg";
            }
        }
    }
```

Sprawdźmy jak działa.

Na początek wywołana zostaje inna funkcja, cleanDesk(); kasuje ona wszystkie karty ze stołu (jeżeli istnieją) oraz zmienia wartości zmiennych na domyślne.

Zmienia wartość pairsLeft na podaną w parametrze funkcji.

Ustawia wartości 4 zmiennych, które są niezbędne do poprawnego wygenerowania wielkości kart. cardWidth/cardHeight będą przechowywać wielkości kolejnych generowanych kart. widthRatio/heightRatio z kolei to współczynniki, o jaki zostaną pomniejszone nasze karty. Proszę zauważyć, że domyślnie wielkości karty ustawione są na całą szerokość stołu/wysokość – 100 pikseli (chodzi o napisy na dole okna), z kolei współczynnik dla szerokości domyślnie równa się ilości par, a wysokości wynosi 2 (2 rzędy).

Teraz sprawdzane są zależności, pozwalające dokładnie określić współczynniki szerokości i wysokości. Nie wdając się w szczegóły – jeżeli par jest mniej niż 8 współczynnik ustalany jest na podstawie reszty z dzielenia przez 8, w przeciwnym wypadku współczynnik wynosi 8. Z współczynnik wysokości zmienia się na 3 jeżeli na stole będzie więcej niż 8 par kart.

W tym miejscu tworzymy zmienną tablicową, która pozwoli naszemu prostemu algorytmowi zorientować się, które karty już znajdują się na stole (by nie utworzył ich za dużo). Następnie wypełniamy tę tablicę 0 (wielkość tablicy równa jest ilości par na stole)

INFORMACJA: Zmienne tablicowe są bardzo często używane. Ich zaletą jest możliwość dokładania kolejnych wartości bez każdorazowego inicjowania (ustalania) nowej zmiennej. Właściwość ta jest o tyle ważna, że program nie może generować w locie nowych zmiennych (wyjątek – języki skryptowe tworzące nowe skrypty/programy kompilowane tworzące własne skrypty). Pod jedną nazwą zmiennej mamy więc wiele wartości, które możemy wykorzystywać, zmieniać czy usuwać.

Przechodzimy do właściwej części kodu, czyli generowania kart na stole. Posłuży nam do tego najpopularniejsza chyba pętla for (zaraz po while... albo i przed). Działanie takiej pętli jest niezwykle proste. Najlepiej przedstawi do składnia:

for (var i =0; i <warunek; i++)

Pętla ta tak naprawdę nie przyjmuje zmiennych ani warunków (jak to ma miejsce w funkcjach). W nawiasie, po średnikach są tak naprawdę wywołane 3 kolejne instrukcje językowe, które mogą posiadać dowolną składnię. W tym wypadku pierwsza instrukcja inicjuje nową zmienną i, której nadaje wartość 0 (może być dowolna inna). W drugiej instrukcji następuje sprawdzenie (warunek) czy i NADAL jest mniejsze od zmiennej warunek. Ostatnia instrukcja nakazuje zwiększenie wartości i o jeden.
Działanie takiej instrukcji sprowadza się do postaci: wykonaj instrukcję drugą (sprawdź warunek). Jeżeli jest spełniony wykonaj kod pomiędzy klamrami {}. Zwiększ wartość zmiennej o jeden i przejdź do instrukcji 2 for. Cykl POWTARZA się do chwili niespełnienia warunku!

W opisywanej funkcji mamy zainicjowane dwie pętle. Dlaczego dwie? Ponieważ generujemy (wstawiamy) karty zarówno w rzędzie, jaki i pionie. I właśnie pierwsza pętla odpowiada za rzędy kart, natomiast druga pętla za kolejne kolumny (czyli kolejne karty w rzędzie).

Za każdym razem w drugiej pętli tworzony jest nowy element z pliku Gamecard.qml. Następnie losowana jest para naszej karty; tutaj użyta została inna pętla – do {} while. Nie wdając się za specjalnie w szczegóły, pętla ta zawsze wykona się PRZYNAJMNIEJ JEDEN RAZ. Po jej wykonaniu następuje sprawdzenie warunku (dostępnego za słowem while). Jeżeli nie został spełniony pętla więcej nie wykona się; w przeciwnym wypadku kod w niej zawarty wykona się ponownie (aż do skutku).

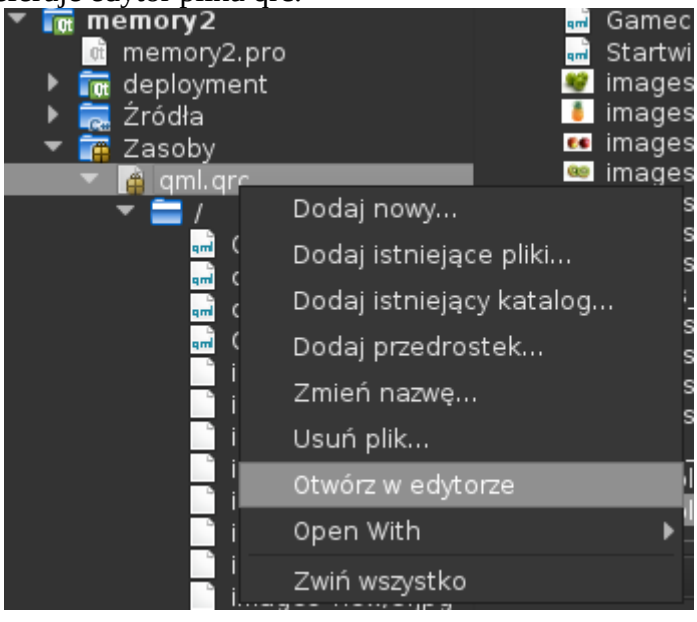
W rzeczonej pętli za każdym razem losujemy, poprzez użycie algorytmu losującego zmienne pseudolosowe, numer pary, który potencjalnie chcielibyśmy umieścić na stole. Sprawdzamy, czy na stole nie leżą już dwie takie karty – jeżeli to prawda (nie ma żadnej takiej karty/jest tylko jedna) to dokładamy kartę i kończymy losowanie. Jeżeli obie karty leżą już na stole to powtarzamy losowanie (aż do znalezienia wolnej pary kart).

Na sam koniec ustawiamy parametry tworzonej karty – jej umiejscowienie, obrazek oraz nazwę obiektu (nazwy mogą się powtarzać!).

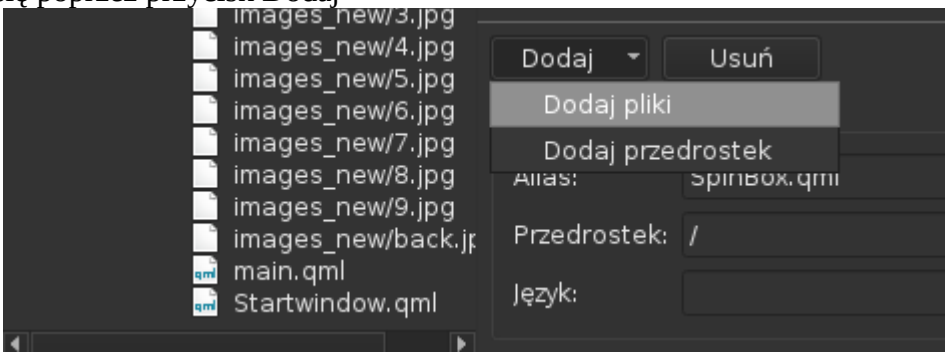
3. Ostatnim dodanym plikiem jest Startwindow.qml. Opisywanie go nie ma większego sensu, gdyż stanowi on zbiór odpowiednich elementów, takich jak przyciski, teksty oraz elementy pozwalające na wybór ilości par/ilości ruchów do wykonania.

Ponieważ elementy te (przyciski oraz pola wyboru par/ruchów) są znacznie rozbudowane zostały zaimportowane (dołączone) z innego projektu. Nie będzie też szczegółowo przedstawiany ich kod – można go przeanalizować w ramach samokształcenia.

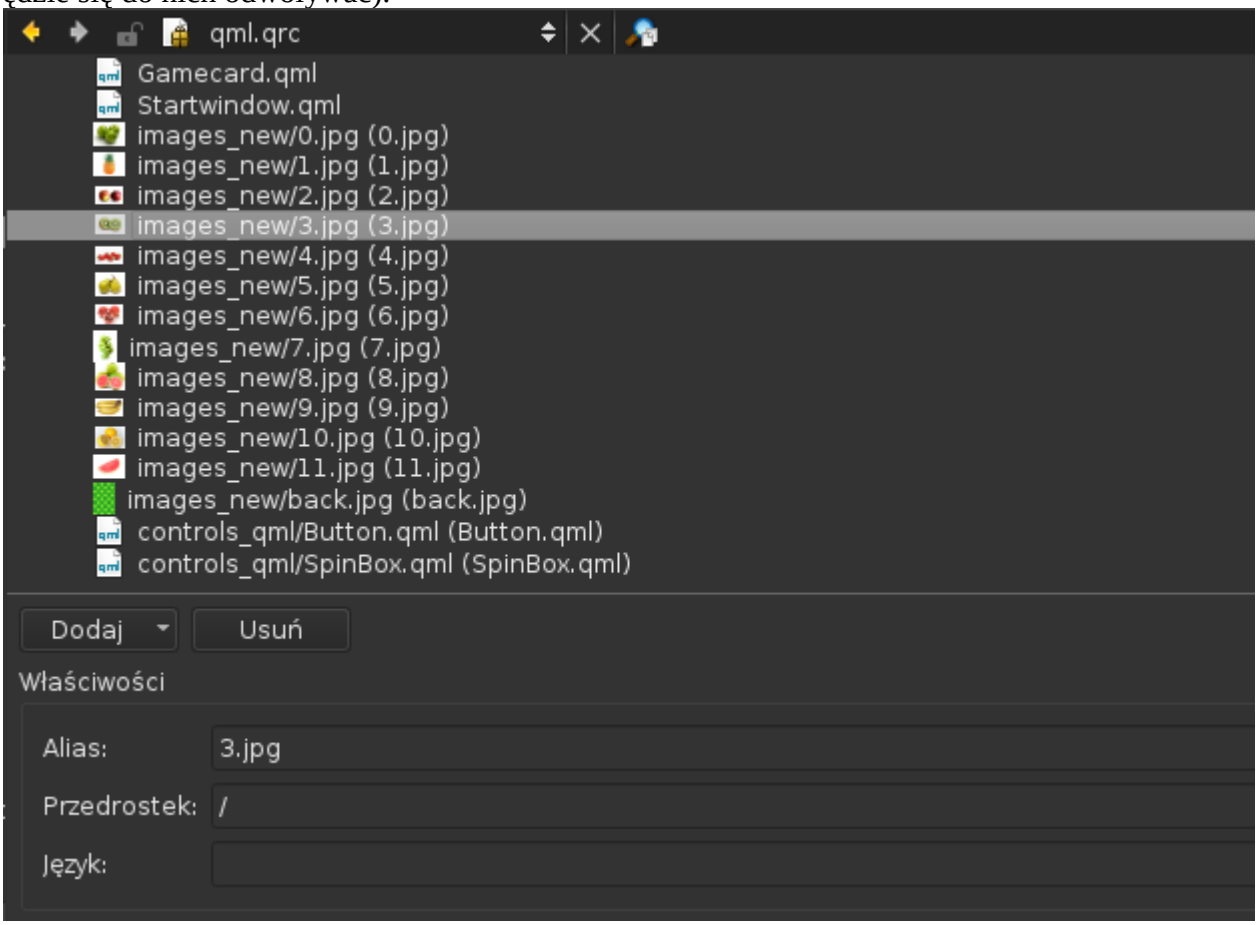
4. Kiedy mamy już kod naszej aplikacji przyszedł czas dodania plików graficznych oraz dwóch elementów QML do naszego projektu. Dokonać można tego na kolka sposobów, autor preferuje edytor pliku qrc.



Pliki dodaje się poprzez przycisk Dodaj



Pliki dodaje się w ten sam sposób jak w innych aplikacjach/do innych aplikacji. Po dodaniu plików najlepiej jest dodać alias (tzw. pseudonim pliku/synonim) dla każdego z nich (inaczej bardzo trudno będzie się do nich odwoływać).



To wszystko! Można kompilować i sprawdzać działanie naszego programu – ćwiczenia.

V. Wnioski i spostrzeżenia końcowe

Dobiegamy końca z naszym przykładem tworzenia aplikacji w C++ z bibliotekami Qt. Jak widać, tworzenie w ten sposób aplikacji jest dosyć proste i intuicyjne, szczególnie dla osób obeznanych z tworzeniem stron internetowych. Mało tego, aplikacja ta będzie znacznie lżejsza od aplikacji HTML5; ponadto zadziała na każdym dostępnym urządzeniu na rynku – wystarczy ją przekompilować.

Powyższy projekt nie jest oczywiście w pełni skończony. Przykładowo dotyka go błąd zbyt szybkiego gracza – jeżeli szybko będziemy klikać na kolejne karty to gra nie zaliczy nawet poprawnie odkrytej karty. Ponadto by gra miała jaki urok potrzebny byłby ranking gracza, czas

ukończenia partii i inne rozwiązania, spotykane w podobnych produktach.

Można też tworzyć bardziej ambitne projekty, takie jak programy narzędziowe, multimedialne czy nawet własne przeglądarki stron WWW (odpowiedni moduł WebKit). Do tego nic nas nie ogranicza – jeżeli potrzebujemy większej wydajności albo chcemy stworzyć nowy, niedostępny w QML obiekt – możemy po prostu go sobie napisać w C++. Łączenie C++ z QML (i na odwrót) również nie jest trudne, a pozwala np. na wczytywanie informacji z baz danych, nawiązywanie połączeń sieciowych, tworzenie aplikacji wielowątkowych itd.

Materiały pomocnicze:
http://wiki.qt.io/
http://doc.qt.io/qt-5
http://doc.qt.io/qt-5/qtqml-cppintegration-interactqmlfromcpp.html
http://stackoverflow.com/questions/28507619/how-to-create-delay-function-in-qml
http://planetatechnika.pl (w przygotowaniu!)