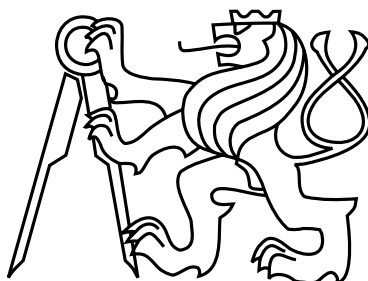


Jazyk pro definici uživatelského rozhraní

- Prostudujte nástroje pro tvorbu moderních grafických uživatelských rozhraní. Zaměřte se na jazyk *QML*.
- Navrhněte jazyk syntakticky co nejvíce podobný *QML* přímo integrovatelný do *C/C++* aplikace.
- Navržený jazyk popište gramatikou a implementujte jeho překladač do pomocného *C/C++* kódu.
- Implementujte podpůrnou knihovnu pro běh grafických aplikací napsaných navrženým jazykem.
- Diskutujte výhody a nevýhody navrženého řešení, řešení porovnejte s *QML*.

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačové grafiky a interakce



Diplomová Práce

Jazyk pro definici uživatelského rozhraní

Bc. Michal Hotovec

Vedoucí práce: Ing. Tomáš Barák

Studijní program: Otevřená informatika, Magisterský

Obor: Počítačová grafika a interakce

15. května 2015

Poděkování

Tímto bych rád poděkoval vedoucímu práce Ing. Tomáši Barákovi, za poskytnuté rady a konzultace.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v příloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Pečkách dne 15. 5. 2015

.....

Abstract

This thesis deals with issues of designing graphical user interface. The thesis focuses on design by usage of declarative models especially on declarative language *QML*. The acquired knowledge is used to design new declarative language for the purposes of definition of user interface. The result of the thesis is a declarative language for definition of graphical user interface, a compiler and a support runtime library for graphical applications. The compiler allows generation of *C++* source code from the newly designed language. The translated *C++* source code can be integrated into user application, which allows to display the defined user interface with the help of the runtime library. The inseparable part of the thesis is a user application, on which this approach was tested. The tested approach is compared with designing graphical user interface by using the *QML* language.

Keywords: Declarative language, Graphical User Interface, *GUI*, Declarative Models

Abstrakt

Tato práce se zabývá problematikou tvorby grafických uživatelských rozhraní. Soustředí se na tvorbu prostřednictvím deklarativních modelů, zejména pak na deklarativní jazyk *QML*. Zjištěné znalosti jsou využity při návrhu nového deklarativního jazyka pro definici grafického uživatelského rozhraní. Výsledkem práce je deklarativní jazyk pro definici uživatelského rozhraní, překladač a podpůrná knihovna pro běh v rámci grafických aplikací. Překladač umožňuje z definice grafického rozhraní v navrženém jazyce vygenerovat zdrojový kód v jazyce *C++*, který lze zahrnout do uživatelské aplikace a pomocí podpůrné knihovny dané grafické rozhraní zobrazit. Součástí práce je i uživatelská aplikace, na níž je otestována funkčnost tohoto přístupu a je provedeno jeho srovnání s tvorbou grafického uživatelského rozhraní pomocí jazyka *QML*.

Klíčová slova: Deklarativní jazyk, Grafické uživatelské rozhraní, *GUI*, Deklarativní modely

Obsah

1	Úvod	1
2	GUI	3
2.1	Historie <i>GUI</i>	3
2.1.1	Počátky <i>GUI</i>	3
2.1.2	První <i>GUI</i>	4
2.1.3	Rozvoj v osmdesátých letech	5
2.2	Deklarativní modely pro tvorbu <i>GUI</i>	7
2.3	<i>GUI</i> Editory	10
2.4	Deklarativní <i>GUI</i>	10
2.5	Jazyk <i>QML</i>	11
2.5.1	Qt	11
2.5.2	Qt Quick	11
2.5.3	<i>QML</i>	11
3	Jazyk <i>CQML</i>	17
3.1	Návrh syntaxe	17
3.1.1	Komponenty	18
3.1.2	Použití externí logiky	18
3.2	Volba gramatiky	19
4	Návrh nástrojů	21
4.1	Deklarace vestavěných typů	22
4.2	Překladač	24
4.2.1	Zpracování kódu v jazyce <i>C++</i>	25
4.2.2	Výstup	27
4.3	Preprocesor datových typů	27
4.3.1	Generování výstupního kódu	27
4.3.2	Hašování	27
4.4	Knihovna	29
4.4.1	Vstup	29
4.4.2	Správa <i>GUI</i>	30
4.4.3	Vykreslování	31
4.4.4	Inicializace	32
4.4.5	Variant	32

5	Implementace	35
5.1	Přidávání vestavěných typů	35
5.2	Překladač	37
5.2.1	Zpracování vstupu	37
5.2.2	Konstrukce a zpracování pomocných struktur	37
5.2.3	Zpracování kódu v jazyce <i>C++</i>	38
5.2.4	Výstup	38
5.3	Preprocesor vestavěných typů	39
5.3.1	Inicializace	39
5.3.2	Generování hašovacích funkcí	39
5.3.3	Generování výstupního kódu	40
5.4	Knihovna	41
5.4.1	Inicializace	41
5.4.2	Update	42
5.4.3	Vykreslování	42
5.4.4	Zpracování vstupu	43
5.4.5	Správa externích zdrojů	44
5.4.6	Datový typ <i>Variant</i> a <i>VariantRef</i>	45
5.4.7	Datové typy <i>CQML</i>	46
5.5	Uživatelská aplikace	46
6	Výsledky	49
6.1	Tvorba <i>GUI</i> pomocí <i>QML</i> a <i>CQML</i>	49
6.1.1	Hlavní nabídka	49
6.1.2	Kalkulačka	51
6.1.3	Jednoduchá hra	52
6.2	Porovnání <i>QML</i> a <i>CQML</i>	54
6.2.1	Tvorba více oken	55
6.2.2	Reference na atributy	55
6.2.3	Dynamická změna hierarchie elementů	56
6.2.4	Rozšiřitelnost	56
7	Závěr	57
A	Gramatika	63
B	Použité technologie	65
C	Syntaxe jazyka <i>CQML</i>	67
C.1	Tvorba elementů	67
C.2	Přiřazení hodnoty atributu	67
C.3	Provázání atributů	68
C.4	Použití identifikátoru pro přístup k elementu	68
C.5	Rozšíření elementu o nový atribut	68
C.6	Reference na element	69
C.7	Zpracování událostí	70

C.8	Použití externí logiky	70
C.9	Použití komponenty	70
D	Datové typy <i>CQML</i>	73
D.1	<i>Grafické elementy</i>	73
D.1.1	<i>Element</i>	73
D.1.2	<i>Rectangle</i>	74
D.1.3	<i>MouseArea</i>	74
D.1.4	<i>Image</i>	75
D.1.5	<i>ScaledImage</i>	75
D.1.6	<i>Text</i>	76
D.2	<i>Hodnotové typy</i>	76
D.2.1	<i>Font</i>	76
D.2.2	<i>Img</i>	76
D.2.3	<i>Color</i>	77
E	Návod na použití <i>CQML</i>	79
E.1	Použití překladače	79
E.1.1	Spuštění překladu	79
E.1.2	Výstupní soubory	79
E.2	Použití knihovny	80
E.2.1	Kompilace uživatelské aplikace	80
E.2.2	Inicializace rozhraní	80
E.2.3	Zpracování vstupu	80
E.2.4	Rozhraní pro zpracování externích zdrojů	80
E.2.5	Vykreslovací rozhraní	80
E.2.6	Funkce volané z hlavní programové smyčky	83
F	Rozšiřování <i>CQML</i>	85
G	Obsah CD	87

Seznam obrázků

2.1	Počítač Xerox Alto [3].	4
2.2	Počítač Xerox Alto, nekterém běží jedno z prvních GUI [3].	5
2.3	Grafické rozhraní počítače Macintosh [6].	6
2.4	Grafické rozhraní systému Windows 1.0 [7].	6
2.5	Grafické rozhraní systému Windows 2.0 [7].	7
2.6	Grafické rozhraní počítače Amiga Workbench [8].	7
2.7	Grafické rozhraní Arthur společnosti Acorn Computers [9].	8
2.8	Diagram aplikace s používající popis GUI vygenerovaný z modelu.	9
4.1	Návrh struktury aplikace.	21
4.2	Návrh struktury aplikace rozšířený o části pro předzpracování datových typů.	22
4.3	Návrh struktury překladače.	24
4.4	Diagram tříd symbolů.	26
4.5	Návrh realizace fronty.	30
4.6	Návrh komunikace mezi knihovnou a uživatelskou aplikací.	30
6.1	Obrázek jednoduché nabídky realizované v jazyce <i>QML</i>	50
6.2	Obrázek jednoduché nabídky realizované v jazyce <i>CQML</i>	51
6.3	Obrázek kalkulačky realizované pomocí deklarace v jazyce <i>QML</i>	52
6.4	Obrázek kalkulačky realizované pomocí deklarace v jazyce <i>CQML</i>	53
6.5	Obrázek jednoduché hry realizované pomocí deklarace v jazyce <i>QML</i>	54
6.6	Obrázek jednoduché hry realizované pomocí deklarace v jazyce <i>CQML</i>	55
D.1	Diagram ilustrující rozřezání obrázku pomocí elementu <i>ScaledImage</i> [29].	75

Kapitola 1

Úvod

Většina moderních softwarových aplikací používá grafické uživatelské rozhraní (dále *GUI*) pro komunikaci s uživatelem. *GUI* umožňuje uživateli ovládat aplikaci prostřednictvím interaktivních grafických elementů a aplikace uživateli poskytuje zpětnou vazbu například změnou podoby daných grafických elementů. Návrh i tvorba (resp. implementace) *GUI* pro softwarové aplikace je časově (tudíž i finančně) velmi náročná činnost.

Podobně jako je tomu u programování aplikační logiky, existuje několik základních přístupů i pro tvorbu uživatelského rozhraní. Jedním z nich je přímé psaní zdrojového kódu *GUI* v nějakém imperativním programovacím jazyce, například s využitím různých grafických knihoven. Tento přístup patří k časově nejnáročnějším a vzhledem k tomu, že každá změna návrhu vyžaduje manuální zásah do zdrojového kódu, tak zpravidla neumožňuje rychlé prototypování. Touto metodou práce nezabývá.

Dalšími přístupy jsou tvorba pomocí speciálních grafických editorů, kde mohou návrh i tvorba být prováděny současně, či automatické generování rozhraní z deklarativních modelů popisujících navržené *GUI*. Modelem v tomto případě může být i popis *GUI* pomocí nějakého deklarativního jazyka. Jedním z takových jazyků je jazyk *QML*, který se používá především v aplikacích programovaných v jazyce *C/C++* a k jeho přednostem patří zejména možnost rapidního prototypování a poskytuje rozmanité množství grafický elementů, které poskytuje knihovna *Qt*.

Cílem této práce je vytvořit jazyk pro tvorbu grafického uživatelského rozhraní, který by poskytoval funkčnost podobnou jazyku *QML*. Ale oproti jazyku *QML*, který je interpretovaný za běhu uživatelské aplikace, půjde vytvořený jazyk přeložit do zkompileovatelného kódu *C/C++*. Z tohoto důvodu je součástí práce i implementace překladače pro daný jazyk, který ověří validitu vstupu a přeloží jej do kódu *C/C++*, který půjde integrovat do zdrojových *C/C++* kódů uživatelské aplikace a následně půjde zkompileovat a spustit. Tato uživatelská aplikace by následně měla zobrazit *GUI* definované v navrženém jazyce.

Pro účely správy *GUI* v rámci uživatelské aplikace, je nutné implementovat podpůrnou knihovnu, kterou bude uživatelská aplikace za běhu používat. Aby tato knihovna mohla být používána na co nejvíce platformách, podporuje vyměnitelné externí rozhraní pro vstup a výstup, který je implementován v rámci uživatelské aplikace.

Na závěr práce je vytvořeno jednoduché *GUI* v jazyce *QML* i v nově navrženém jazyce, na čemž je demonstrována podobnost i rozdíly mezi jednotlivými řešeními. Aby bylo možné přeložené zdrojové kódy v jazyce *C/C++* použít pro zobrazení daného *GUI*, je také vytvo-

řena uživatelská aplikace využívající vytvořenou podpůrnou knihovnu, na níž je ukázána funkčnost použitého přístupu.

Kapitola 2

Grafická uživatelská rozhraní

Tato kapitola se zabývá shrnutím poznatků o *GUI*. V první řadě zmapuje historický rozvoj *GUI* a následně uvádí několik přístupů pro tvorbu moderního *GUI* pomocí deklarativních modelů. Následně se zaměří na deklarativní jazyky, především na jazyk *QML*.

2.1 Historie *GUI*

První počítače neumožňovaly žádný grafický výstup a komunikovali s uživatelem prostřednictvím děrných štítků, avšak s nástupem pokročilejších přístrojů pro vstup a výstup přišla i disciplína tvorby uživatelského rozhraní.

2.1.1 Počátky *GUI*

Za počátek vzniku disciplíny vývoje uživatelského rozhraní pro počítače je považováno publikování článku *Augmenting Human Intellect* [1] od D. C. Engelbarta v roce 1962. Kde autor argumentoval, že schopnost lidského intelektu zkoumat a řešit komplexní problémy a situace může být rozšířena pomocí vhodných nástrojů, v tomto případě počítačů. V zápětí bylo založeno výzkumné centrum *Augmentation Human Intellect (AHI) Research Center* (také známé pod zkratkou *ARC*) ve Stanfordském výzkumném institutu, které fungovalo až do roku 1989, kdy byla jeho činnost ukončena z finančních důvodů. V rámci centra *AHI* započal vývoj experimentálních technologií pro rozšíření schopností lidského intelektu.

6. prosince 1968 byly veřejně předvedeny vyvinuté technologie ve formě *NLS (oN-Line System)* na konferenci *Fall Joint Computer Conference* v San Franciscu. Na tomto systému byl demonstrován nový přístup k interakci uživatele s počítačem. Jednalo se o distribuovaný systém několika počítačů. Pro vstup uživatele bylo používáno několik zařízení, konkrétně klávesnice, několik posuvníků a úplně první počítačové myši. Systém používal komerčně dostupný displej, který umožňoval vykreslování znaků a vektorové zobrazení grafiky v podobě čar. Ukazatel myši byl na obrazovce zobrazen pomocí svislé šipky. Během ukázky bylo demonstrováno sdílení grafického výstupu na více obrazovek. Dále zde byla demonstrována úprava dokumentů, možnosti jejich zobrazování a úpravy jejich hierarchie. Z technologií a přístupů prezentovaných v rámci ukázky tohoto systému vyšly v budoucnosti, některé společnosti při návrhu svých uživatelských rozhraní [2].

2.1.2 První GUI

V roce 1970 byl společností *Xerox* založen výzkumný institut *PARC* (*Palo Alto Research Center*). Během prvních let fungování bylo vyvinuto několik dodnes využívaných technologií, mezi něž patří i technologie laserového tisku vyvinutá v roce 1971. V tehdejší době neexisto-



Obrázek 2.1: Počítač Xerox Alto [3].

valy počítače, které by umožnily manipulaci s podobou tištěných dokumentů pomocí jiného než textové rozhraní. Proto byl v *PARC* zahájen vývoj počítače, který byl v roce 1973 světu představen jako počítač *Alto Personal Workstation*. Tento počítač byl během let postupně vylepšován a postupně se stal prvním počítačem, který používal bitmapový displej a umožnil grafickou úpravu dokumentů. V roce 1972 byl v institutu *PARC* vyvinut první objektově orientovaný jazyk zvaný *SmallTalk*, který umožňoval jednotlivé součásti programu rozdělit do samostatných objektů, které mohly být následně používány i v jiných programech. V roce 1975 bylo vyvinuto první *GUI*, které se svou funkcí velice podobalo dnešnímu. Zahrnovalo mimo jiné ikony, překrývající se okna, vyskakovací nabídky a mohla být ovládána pomocí myši [5]. Počítač *Alto* však nebyl nikdy komerčně k dispozici a až v roce 1981 byla



Obrázek 2.2: Počítač Xerox Alto, nekterém běží jedno z prvních GUI [3].

dána veřejně k prodeji jeho variace pod názvem *Xerox Star 8010 Document Processor*.

2.1.3 Rozvoj v osmdesátých letech

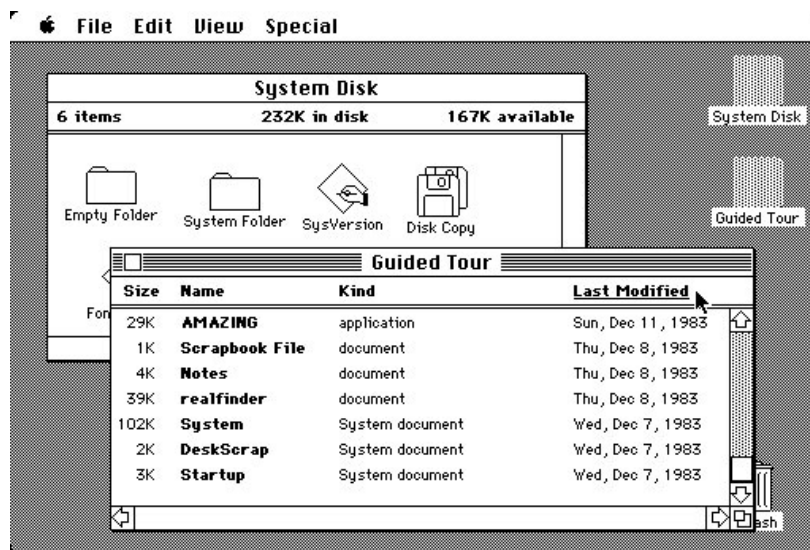
V osmdesátých letech dvacátého století se začaly na trhu objevovat první počítače poskytující GUI [4].

Apple

Za jednoho z dalších průkopníků je považována společnost *Apple* založená v roce 1976. Která za svůj počáteční úspěch vděčila populárnímu počítači *Apple II*, který umožňoval zobrazení textu i grafiky, avšak poskytoval pouze textové vstupní rozhraní. Společnost byla ovlivněna vlivem institutu *PARC* a započala v roce 1979 vývoj počítače *Lisa*, který měl být jejím prvním počítačem poskytující ovládání pomocí GUI. Vývoj tohoto počítače byl dokončen v roce 1983. Díky vysoké ceně však nebyl počítač příliš komerčně úspěšný a v roce 1984 byla trhu představena jeho levnější verze - počítač *Macintosh*, který, ačkoli měl nižší výkon než *Lisa*, zachoval většinu schopností jejího GUI.

Microsoft Windows

V roce 1983 Microsoft ohlásil vývoj systému *Windows 1.0*, který byl dokončen v roce 1985. Na rozdíl od moderních rozhraní tento systém používal okna, která byla ve svých



Obrázek 2.3: Grafické rozhraní počítače Macintosh [6].

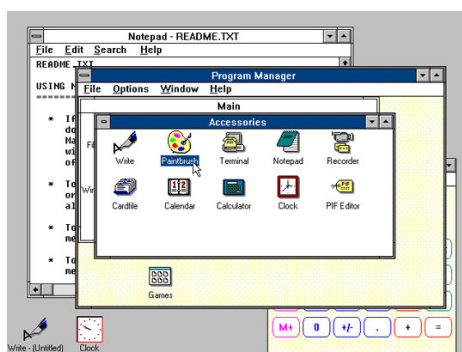
vyhrazených prostorech na obrazovce a nemohla překrývat. V roce 1987 byla vydána verze *Windows 2.0*, která zavedla tradiční systém překrývajících se oken, která se dala přesunovat pomocí myši.



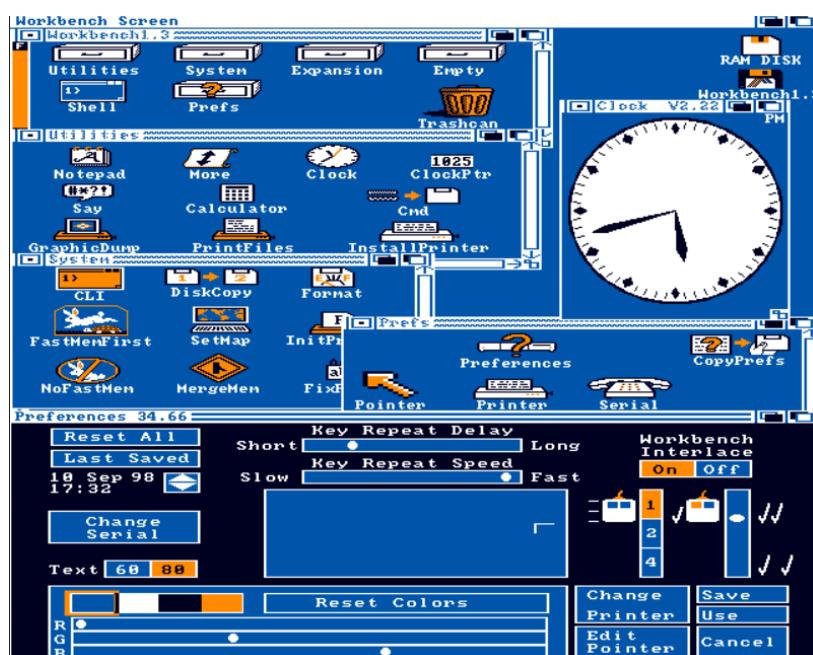
Obrázek 2.4: Grafické rozhraní systému Windows 1.0 [7].

Amiga Workbench

V roce 1985 společnost *Commodore* představila počítač *Amiga*, který poskytoval své GUI *Amiga Workbench*. Toto rozhraní umožňovalo ručně přehazovat pořadí oken na obrazovce (odpředu dozadu) a umožňovalo manipulovat pomocí myši s jednotlivými okny tradičním způsobem. Díky manuálnímu způsobu pořadí vykreslování oken bylo umožněno pracovat s oknem, aniž by se přesunulo dopředu.



Obrázek 2.5: Grafické rozhraní systému Windows 2.0 [7].



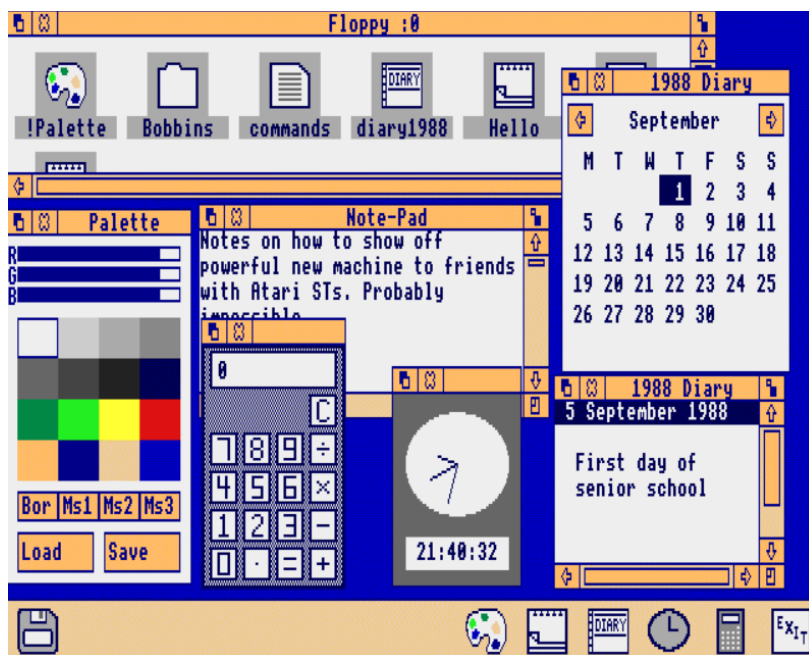
Obrázek 2.6: Grafické rozhraní počítače Amiga Workbench [8].

Acorn

V roce 1987 společnost *Acorn Computers* vyvinula své GUI zvané *Arthur*, které jako první umožnilo zobrazení fontů v šestnácti barvách s použitím anti-aliasingu.

2.2 Tvorba GUI pomocí deklarativních modelů

Podobně jako u programování běžných aplikací existuje deklarativní a imperativní přístup k tvorbě GUI aplikací. Imperativní přístup spočívá v tom, že programátor v rámci



Obrázek 2.7: Grafické rozhraní Arthur společnosti Acorn Computers [9].

aplikačního kódu přímo naprogramuje struktury uživatelského rozhraní spolu s jejich funkcí a vzhledem.

Jelikož předmětem této práce je vyjít z existujícího deklarativním jazyka pro tvorbu GUI, se zabývá především tvorbou GUI pomocí deklarativních modelů. Základem tohoto přístupu je, že vývojář popíše vzhled a strukturu GUI pomocí nějakého modelu. Model obsahuje informace nutné pro vytvoření samotného uživatelského rozhraní. Tento model je přístupný aplikaci, která podle něj vytvoří deklarované uživatelské rozhraní[10].

Existuje několik běžně užívaných modelů pro deklarativní tvorbu GUI.

Datové modely Datové modely byly prvními používanými modely v rámci modelově založeného vývoje GUI. Jsou přímo odvozeny z datových struktur aplikace, proto jsou zejména užitečné při generování GUI výběrem součástí podle toho, z jakého typu byly odvozeny. Některé systémy dokážou z datového modelu pomocí algoritmů přímo vygenerovat úplné uživatelské rozhraní [11].

Doménové modely Doménový model reprezentuje objekty v doméně a jejich vzájemné vztahy. Patří mezi ně některé objektově orientované datové modely. Tyto modely vedly k plně deklarativním doménovým modelům, které dokáží efektivně popsat vztahy mezi objekty jednotlivých domén a umožňují generování GUI s dynamickým chováním v rámci statického rozvržení[12].

Aplikační modely Specifikují služby, které aplikace poskytuje. Zpravidla se jedná o objektově orientované modely. Kde objekty uchovávají stav jednotlivých součástí v rámci GUI a také operací, které ovlivňují stav těchto součástí. Cílem těchto modelů je usnadnit deklaraci chování GUI.

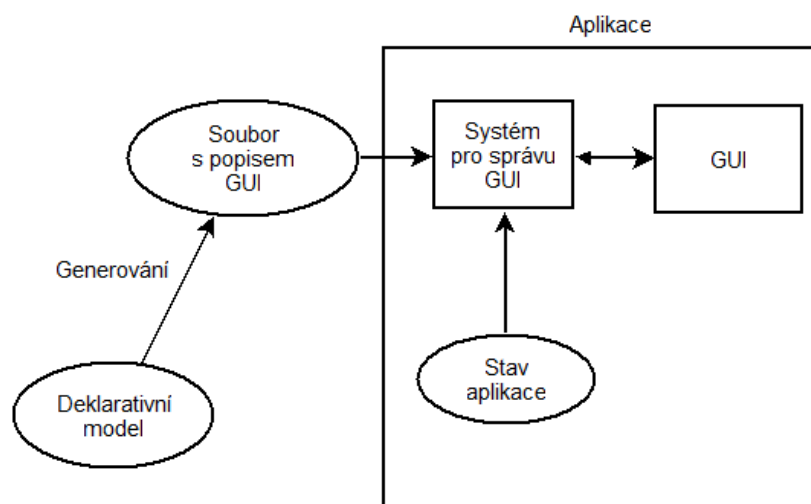
User-task model Tento model je součástí user-centered design (designu cíleného na uživatele). Popisuje úkony, které koncový uživatel může provádět a jaké interakční možnosti musí být navrženy. Model zahrnuje prvky, jako jsou cíle, úkony, a doménové objekty. Cíle definují požadované stavy. Sekvence úkonů definuje proceduru k dosažení určitého cíle. Doménové objekty reprezentují jednotlivé elementy, které musí být zobrazeny v rámci provedení jednotlivých úkonů.

Dialogový model Dialogový model popisuje komunikace člověk-počítač. Specifikuje druhy funkcí, které může koncový uživatel zavolat, například stisky tlačítek nebo příkazy. Dále specifikuje, jakými způsoby bude počítač od uživatele vyžadovat vstup nebo mu poskytovat zpětnou vazbu.

Prezentační model Prezentační model uvádí jak jednotlivé objekty a interakce vypadají v rámci odlišných dialogových stavů. Také zpravidla popisují hierarchickou dekompozici jednotlivých obrazovek na jejich samostatné součásti. Prezentační a dialogové modely jsou si velmi blízké, a některé přístupy tvorby GUI používají oba tyto modely.

Behaviorální model Tento model specifikuje chování vstupu. Kombinace prezentačního a behaviorálního modelu umožňuje specifikovat rozvržení a dynamické chování GUI nezávisle na sobě.

Popsané modely se používají různými způsoby. Záleží především na systému použití GUI, v některých případech je GUI aplikace generováno z modelu přímo za jejího běhu. V jiných případech je pomocí externích nástrojů vygenerován popis GUI ve specifickém formátu, který je následně použit v rámci systému pro správu uživatelského rozhraní (viz. obrázek 2.8), který aplikace používá. Součástí systému pro správu uživatelského rozhraní může být například interpret jazyka, v němž byl popis GUI vygenerován [13].



Obrázek 2.8: Diagram aplikace s používající popis GUI vygenerovaný z modelu.

2.3 GUI Editory

K návrhu a tvorbě GUI se používají mimo jiné i speciální editory, z nichž některé umožňují i spuštění a testování navrhovaného rozhraní. Výstupem velké části takovýchto editorů bývá zpravidla kód nějakého jazyka, který popisuje dané rozhraní, které vývojář v rámci editoru navrhl. Většina editorů umožňuje návrh rozhraní pomocí postupného vkládání elementů na kreslicí plochu ze seznamu poskytovaných prvků a následně přiřazovat funkce či objekty pro obsluhu událostí v rámci daného GUI.

Některé editory bývají integrovány přímo do prostředí pro vývoj aplikací, například editor wxSmith pro vývojové prostředí *Code::Blocks* pro jazyk C++, či editor GUI ve vývojovém *NetBeans* pro jazyk Java. V takových případech bývá vývojové prostředí použito pro současný vývoj aplikační logiky i grafického uživatelského rozhraní. Zpravidla je pak výstupem GUI editoru kód, který je obdobný tomu, pokud by programátor naprogramoval při použití imperativního přístupu pro tvorbu GUI.

wxSmith – *wxSmith* je zásuvný modul pro vývojové prostředí *Code::Blocks*. Slouží k vývoji grafického rozhraní s použitím grafických elementů poskytovaných knihovnou *wxWidgets* pro jazyk C++ [14].

Swing GUI Builder – *Swing GUI Builder*, také označovaný jako *Project Matisse*, je integrován do vývojového prostředí *NetBeans* pro jazyk Java. GUI vytvořené tímto editorem využívá funkčnost a elementy poskytované knihovnou *Swing*. Jeho součástí jsou i vizuální nástroje pro ladění, které umožňují během ladění aplikace uložit aktuální stav GUI v daném momentě, přičemž uživatel pak může prozkoumávat GUI v uchovaném stavu. Je umožněno zobrazit vlastnosti jednotlivých GUI elementů, jejich zdrojový kód i pracovat s událostmi a třídami pro jejich obsluhu [15].

Ultimate++ – *Ultimate++* je vývojové prostředí pro vývoj GUI aplikací v jazyce C++. Jeho součástí je editor pro návrh GUI. Toto prostředí poskytuje vlastní knihovnu se sadou několika desítek použitelných elementů [16].

2.4 Deklarativní GUI

Pro tvorbu GUI se také používají deklarativní jazyky, tyto jazyky se zpravidla označují jako *User interface mark-up languages*. Tyto jazyky slouží pro popis uživatelského rozhraní podobně, jako například jazyk *HTML* slouží pro popis dokumentu. V jazyce je zapsán popis uživatelského rozhraní, podle kterého je dané uživatelské rozhraní vytvořeno. Ve velké části případů je kód v daném jazyce interpretován za běhu aplikace systémem pro správu GUI. Některé jazyky, mezi něž patří například *QML*, lze přeložit do byte-kódu, který je v rámci cílové aplikace interpretován.

Následují příklady deklarativních jazyků pro tvorbu GUI:

- **UIML** – Jedná se o jeden z prvních jazyků pro deklarativní popis uživatelského rozhraní. *User Interface Markup Language* je odvozena z jazyka *XML*. Jedná se o jazyk vyvinutý za účelem vytvoření univerzálního jazyka, který by mohl být použit pro tvorbu GUI nezávisle na platformě [17].

- XAML – XAML vychází z jazyka XML. Používá se pro tvorbu GUI aplikací v prostředí .NET. V XAML deklarované elementy GUI přímo reprezentují instance jednotlivých objektů. Tento jazyk je tak silně provázán s typovým systémem prostředí .NET [18].
- FXML – FXML skriptovací jazyk vycházející z XML. Umožňuje tvořit objektové grafy v jazyce Java. Používá se primárně pro deklaraci GUI pro JavaFX aplikace [19].
- MXML – Jedná se o další XML jazyk. Je inspirován jazykem HTML. Umožňuje zkompileování souborů do SWF souborů. Vytvořené GUI pak může být používáno s pomocí přehrávače jazyka Flash. Tento jazyk umožňuje rozšíření souboru poskytovaných komponent o další, ke kterým může být přistupováno stejným způsobem [20].

2.5 Jazyk QML

Součástí této práce je i tvorba deklarativního jazyka. Tento jazyk se inspiruje jazykem QML, který je součástí rozsáhlého vývojového prostředí poskytovaného knihovnou Qt.

2.5.1 Qt

Qt je multi-platformní open-source knihovna sloužící k vývoji především GUI aplikací v jazyce C++. Aplikace se vyznačují tzv. nativním vzhledem GUI, což znamená, že se pro grafické rozhraní využívá standardní vzhled GUI poskytovaný operačním systémem. Tímto je dosaženo obdobného vzhledu, jaký mají standardní aplikace pro daný operační systém. Součástí Qt je Qt Quick, což je soubor technologií sloužících k tvorbě deklarativního GUI.

Qt poskytuje vývojové prostředí Qt Creator, které lze použít pro vývoj C++ aplikací využívající prvky Qt nebo aplikací Qt Quick. Součástí Qt creator je vizuální debugger C++, editor kódu, grafický editor pro uživatelské rozhraní, interpret pro jazyk QML a debugger pro jazyk QML [21].

2.5.2 Qt Quick

Qt Quick je soubor technologií sloužících k tvorbě deklarativního GUI. Poskytuje sadu GUI elementů, deklarativní jazyk QML a modul Qt declarative. Qt declarative slouží jako modul pro využití QML v C/C++ aplikacích, který za běhu rozparsuje vstup ve formátu QML a sestaví podle něj GUI s využitím funkčnosti poskytovanou Qt. Jeho součástí je interpret jazyka JavaScript, který umožňuje evaluaci JavaScript kódu ve funkcích definovaných v QML souboru. Qt declarative odděluje správu GUI od aplikační logiky C++. K datům vytvořených v QML lze přistupovat v C++ části aplikace a naopak prostřednictvím tohoto modulu [21].

2.5.3 QML

QML je jazyk vycházející z jazyka JavaScript. Jedná se o jazyk pro deklaraci hierarchického GUI, na vrcholu hierarchie je vždy jediný element a všechny ostatní elementy

Ukázka kódu 2.1: Tvorba dvou jednoduchých elementů pomocí jazyka *QML*.

```
Rectangle
{
    width : 100; height : 100
    Image
    {
        source : "img.jpg"
    }
}
```

jsou na něj navázány ve stromové struktuře. Každý element může mít teoreticky neomezené množství potomků (elementy, jež se nacházejí v hierarchii přímo pod ním), avšak může mít nanejvýš jednoho rodiče (element, který se nachází v hierarchii přímo nad ním). Pouze kořenový element na vrcholu hierarchie nemá žádného rodiče [22].

Syntaxe

Základní syntaxe je ilustrována viz. fragment kódu 2.1. Soubor s tímto kódem vytvoří dva *GUI* elementy. Kořenovým prvkem je element typu *Rectangle*, jehož potomkem je element typu *Image*. Element typu *Rectangle* má přiřazenu hodnotu dvěma atributům *width* a *height*, zatímco element typu *Image* má nastaven atribut *source* řetězcem „img.jpg“.

Přidávání nových atributů

QML umožňuje rozšířit existující typ elementu o přídavné atributy pomocí klíčového slova *property* a zadáním datového typu atributu viz. fragment kódu 2.2.

Ukázka kódu 2.2: Ukázka deklarace dvou nových atributů.

```
Rectangle
{
    property int newProp01
    property int newProp02 : 5
}
```

Tento kód vytvoří dva nové atributy typu *int* jménem *newProp01* a *newProp02*, přičemž *newProp02* inicializuje na hodnotu 5.

Výrazy

Hodnoty jednotlivých atributů lze definovat pomocí výrazů. Hodnoty výrazů se přepočítávají za běhu aplikace, pokud se jejich hodnota může změnit, jak je ilustrováno viz. fragmenty kódu 2.3, 2.4 a 2.5.

Ukázka kódu 2.3: Výraz ilustrující přiřazení konstanty.

```
width : 5
```

Ukázka kódu 2.4: Výraz ilustrující přiřazení hodnoty jiného atributu.

```
width : height / 2
```

Ukázka kódu 2.5: Výraz ilustrující přiřazení hodnoty atributu jiného elementu.

```
width : parent . width / 3
```

V prvním výrazu (2.3) je hodnota nastavena na konstantní hodnotu 5.

V druhém výrazu (2.3) je atribut nastaven pomocí jiného atributu *height*, kdykoli se tudíž za běhu programu změní hodnota atributu *height*, pak se změní i hodnota atributu *width*.

V posledním výrazu (2.3) je použito klíčové slovo *parent*, pomocí něž lze přistupovat k atributům nadřazeného elementu v hierarchii (tzv. rodiče). Pokud se v takovémto případě změní hodnota rodičova atributu *width*, dojde i k opětovnému spočtení výrazu a tak i k úpravě hodnoty *width* potomka. Tímto způsobem je umožněno dynamicky přizpůsobovat velikost i jiné atributy elementů v hierarchii v závislosti na změnách atributů jiných elementů.

Toto je jednou z důležitých vlastností při použití *QML* k definici uživatelského rozhraní. Je tak možné provázat mezi sebou jednotlivé atributy (tzv. „property binding“) tak, že se za běhu programu jejich hodnoty dynamicky přepočítávají kdykoli, se změní hodnoty atributů, na nichž jsou závislé.

Identifikátory objektů

Každý objekt může mít přiřazen unikátní identifikátor, pomocí něž může být k němu přistupováno. k tomu slouží speciální atribut *id*, do něž může být přiřazen daný identifikátor. Identifikátor musí být unikátní v dané komponentě. V Fragment kódu 2.6 je ukázka použití tohoto atributu.

Ukázka kódu 2.6: Ukázka použití atributu *id*.

```
Rectangle
{
    id : identifier01
    width : 100
}
Rectangle
{
    width : identifier01 . width
}
```

V tomto případě je hornímu elementu nastaven atribut *width* na hodnotu 100 a atribut *width* spodního elementu je nastaven na totožnou hodnotu.

Komponenty

GUI definované uvnitř jednoho *QML* souboru se označuje jako komponenta. Každou komponentu je možné znovu použít jako stavební elementy jiných komponent, obdobně jako je tomu u standardních typů elementů. Komponenty se importují automaticky, při použití jména souboru jako typu *GUI* elementu.

Ukázka kódu 2.7: Ukázka použití komponenty z jiného souboru.

```
Rectangle
{
    CustomButton {}
}
```

Například existuje-li soubor „CustomButton.qml“, pak kód Fragment kódu 2.7 vytvoří uvnitř obdélníku komponentu definovanou v importovaném souboru „CustomButton.qml“.

Funkce

V *QML* lze pomocí klíčového slova *function* definovat funkci se zdrojovým kódem v jazyce *JavaScript*, při vyhodnocování výrazu v takovýchto funkcích platí totožná pravidla s těmi pro vyhodnocování výrazů přiřazených atributům (viz. ukázka kódu 2.8).

Pokud místo výrazu je atributu přiřazen kód těla funkce v jazyce *JavaScript* ohraničený složenými závorky, je pro vyhodnocení hodnoty atributu použita tato funkce, výsledná hodnota atributu se v takovém případě vrací pomocí klíčového slova *return* (viz. ukázka kódu 2.9).

Ukázka kódu 2.8: Ukázka použití funkce definované pomocí klíčového slova *function* pro určení hodnoty atributu.

```
function someFunction ()
{
    if (someCondition)
        return 500;
    else
        return 200;
}
Rectangle
{
    v : someFunction ()
}
```

Ukázka kódu 2.9: Ukázka použití funkce definované v rámci elementu pro určení hodnoty atributu.

```
Rectangle
{
    v : {
        if (someCondition)
            return 500;
        else
            return 200;
    }
}
```

Kontrola atributů

Kontrola existence atributů nebo kontrola datových typů je částečně řešena při zpracování *QML* souboru a částečně až za běhu výsledného programu. Pokud je nějakému atributu přiřazena hodnota, je kontrolována existence daného atributu již při zpracování *QML* souboru. V případech, že nelze předem určit existenci nějakého atributu (například při přístupu k atributům elementu prostřednictvím obecné reference), se daná kontrola provádí až za běhu programu.

Ukázka kódu 2.10: Ukázka přístupu k atributu pomocí reference na element.

```
{
    v : w.x + d
}
```

Uvažujme příklad Fragment kódu 2.10. Pokud element nemá atribut se jménem *v*, nastane chyba už při zpracování *QML* souboru. V opačném případě mohou nastat chyby až během spuštění programu. Aby byl kód validní, musí existovat buď element identifikátorem *w*, mající atribut *x* typu kompatibilním s *v* nebo musí element mít atribut *w*, který obsahuje objekt, jež má atribut *x* typově kompatibilní s *v*. Zároveň musí existovat atribut *d*, který je typově kompatibilní s *v*, přičemž musí tento typ podporovat operaci sčítání.

V případě, že existuje zároveň atribut *w* a element s identifikátorem *w*, bere se v potaz pouze element s identifikátorem *w*. Tudiž, pokud existuje atribut *w* obsahující atribut *x*, ale zároveň existuje nějaký element s identifikátorem *w*, který neobsahuje atribut *x*, dojde k chybě, kvůli neexistenci atributu *x* v elementu *w*.

Použití externí logiky

Je umožněno v rámci *QML* souboru použít logiku načtenou z externího souboru v jazyce *JavaScript*, což lze učinit prostřednictvím klíčového slova *import*. Toto je ilustrováno v kódu 2.11, kde je na prvním řádku načten soubor „externalLogic.js“ a jeho funkčnost je zapouzdřena do objektu přístupného pod jménem *Logic* z libovolného místa v daném *QML* souboru. Dále je v ukázce demonstrován přístup k funkci *GetNewWidth*, která se nachází uvnitř načteného souboru, přičemž hodnota vracená touto funkcí je přiřazena atributu *width*.

Ukázka kódu 2.11: Použití externí logiky v jazyce *QML*.

```
import "externalLogic.js" as Logic
MouseArea
{
    onClick : {
        width = Logic.GetNewWidth();
    }
}
```

Výhody a Nevýhody

Výhody:

- Umožňuje rychlé prototypování a rychlé iterace návrh-zobrazení při tvorbě *GUI*.
- Poskytuje velké množství *GUI* elementů.
- Umožňuje tvorbu nových *GUI* elementů, které mohou i nemusí vycházet z již poskytnutých.
- Multiplatformní podpora.
- Umožňuje jednoduše dynamicky provázat vlastnosti různých elementů pomocí tzv. property binding.

Nevýhody:

- Jedná se o interpretovaný jazyk, což znamená, že dosahuje nižšího výkonu oproti neinterpretovaným jazykům, které se kompilují do nativního kódu.
- Jedná se o slabě typovaný jazyk, tudíž se typy proměnných a hodnot určují až za běhu aplikace, což představuje výkonostní nevýhodu oproti silně typovaným jazykům. Jazyk je tak také náchylný k chybám, které se projeví až za běhu aplikace.

Kapitola 3

Jazyk *CQML* - Návrh a vlastnosti

Cílem při návrhu jazyka *CQML* bylo zachovat co největší podobnost s *QML*, co se týče jeho funkčnosti a možností, ale zároveň umožnit jednoduchou integraci do aplikací v jazyce *C++*. Pro výstupní jazyk překladače byl vybrán jazyk *C++*. Pro jednodušší převod do výstupního formátu je pro kódy funkcí pro obsluhu událostí a výpočtů hodnot atributů, oproti jazyku *QML* (využívající *JavaScript*), použita syntaxe jazyka *C++*. Což umožní ve výsledném kódu vygenerovaným překladačem například zavolání funkcí či přistupovat ke globálním proměnným.

3.1 Návrh syntaxe

Syntaxe byla zvolena podobná jazyku *QML*. Rozdílem je ukončující znak středníku na konci každého výrazu. Ačkoli ukončovací znak není nutný (viz. *QML* a *JavaScript*), je tímto dosaženo toho, že výsledná gramatika je nevypouštěcí, což má za následek jednodušší zpracování *CQML* kódu a také přesnější chybová hlášení při syntaktické analýze. Základní syntaxe je ilustrována viz. kód 3.1.

Ukázka kódu 3.1: Tvorba dvou jednoduchých elementů pomocí jazyka *CQML*.

```
Rectangle
{
    id : identifier01;
    width : 100;
};
Rectangle
{
    width : identifier01.width;
};
```

Tímto způsobem se vytvoří stejné *GUI*, jako v případě jazyka *QML* viz. Fragment kódu 2.1. Podobně jako v *QML*, *GUI* komponenty definované uživatelem mohou být uloženy v oddělených souborech a následně importovány a opětovně používány jako samostatné elementy v jiných *GUI* komponentech.

3.1.1 Komponenty

Na rozdíl od *QML* se soubory neimportují automaticky, ale začátku každého *CQML* souboru jsou definovány soubory, z nichž jsou importovány *GUI* hierarchie a také identifikátor, jehož použitím jako typ elementu je na dané místo umístěna importovaná komponenta. Fragment kódu 3.2 ilustruje import komponenty v jazyce *CQML* ze souboru „CustomButton.cqml“, ke které se následně přistupuje pomocí klíčového slova *Button*.

Ukázka kódu 3.2: Ukázka importu komponenty v jazyce *CQML*.

```
import "CustomButton.cqml" as Button;
Rectangle
{
    Button {};
```

3.1.2 Použití externí logiky

V rámci *CQML* souboru lze použít logiku implementovanou uvnitř uživatelské aplikace. Pomocí klíčového slova *include* lze do výstupního *C++* kódu zahrnout hlavičkový soubor a následně uvnitř kódu funkcí v jazyce *CQML* volat funkce, jejichž hlavička byla deklarována v daném hlavičkovém souboru. Toto je ilustrováno v ukázce kódu 3.3, kde je na prvním řádku načten hlavičkový soubor „externalLogic.h“ a zároveň se použije hodnota vracená funkcí *GetNewWidth* k nastavení hodnoty atributu *width*, přičemž funkce *GetNewWidth* je deklarována v načteném hlavičkovém souboru.

Ukázka kódu 3.3: Ukázka použití funkce deklarované v hlavičkovém souboru.

```
include "externalLogic.h"
MouseArea
{
    MouseClick : {
        width = GetNewWidth();
    }
}
```

Programovací jazyky zpravidla nebývají bezkontextové, nicméně jejich syntaktickou analýzu (nikoli však sémantická) lze provést pomocí bezkontextové gramatiky. Toto platí například pro jazyk *C*, z jehož gramatiky se vychází pro syntaktickou analýzu kódu funkcí a výrazů v *CQML*. Z tohoto hlediska je i gramatika *CQML* bezkontextová, protože typová kontrola a kontrola existence atributů či identifikátorů není prováděna během syntaktické analýzy. To má za výhodu, že není během syntaktické analýzy potřeba použít vyhledávací tabulku pro kontrolu symbolů.

Ukázka kódu 3.6: Možnosti interpretace víceznačného výroku *IF-ELSE*

```

1.
if (a){
    if (b)
        s ;
    else
        s2 ;
}

2.
if (a){
    if (b)
        s ;
}
else
    s2 ;

```

3.2 Volba gramatiky

Pro gramatiku *CQML* byla zvolena *YACC* notace, vzhledem k dostupnosti technologií pro generování parseru a existenci gramatiky pro jazyk *C* [28] ve formátu *YACC* [23]. Gramatika ve formátu *YACC* se nachází v příloze A.

Gramatika jazyka *CQML* je víceznačná, protože pro definici funkcí a výrazů používá syntaxi jazyka *C*, jehož gramatika je víceznačná. Víceznačnost je způsobena pravidlem pro podmínku *IF-ELSE* (viz. Fragment kódu 3.4), jedná se o tzv. „dangling else problem“.

Ukázka kódu 3.4: Víceznačné *IF-ELSE* pravidlo gramatiky.

```

selection_statement
: IF '(' expression ')' statement
| IF '(' expression ')' statement ELSE statement

```

Výrok uvedený ve Fragmentu kódu 3.5 může být podle daného pravidla rozparsován dvěma způsoby viz. Fragment kódu 3.6.

Ukázka kódu 3.5: Příklad víceznačného výroku *IF-ELSE*

```

if (a) if (b) s ; else s2 ;

```

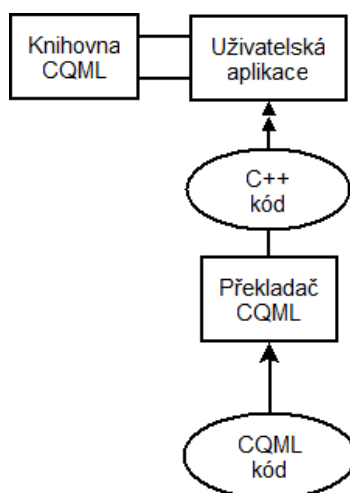
Podle konvencí se preferuje první způsob interpretace, který lze chápat tak, že se *ELSE* přiřadí k nejnivnějšímu *IF*.

Kapitola 4

Návrh nástrojů pro tvorbu *GUI* jazyce *CQML*

Dalším cílem této práce je vytvořit nástroje, které umožní z deklarace uživatelského rozhraní v jazyce *CQML* vytvořit zdrojový kód v jazyce *C++*, jenž může být následně využit uvnitř grafické aplikace, která deklarované uživatelské rozhraní zobrazí a umožní uživateli s ním interagovat. Vytvoření zdrojového kódu *C++* je realizováno pomocí překladače. Pro co nejjednodušší integraci vygenerovaného zdrojového kódu do uživatelské aplikace je vytvořena knihovna, která se má starat o správu a vykreslování struktur vygenerované hierarchie uživatelského rozhraní. Na obrázku 4.1 je náčrtek, ukazující jednotlivé bloky, systému pro vytvoření *GUI* pomocí *CQML*, včetně uživatelské aplikace.

Jedním z cílů této práce je umožnit jednoduchou rozšiřitelnost základní funkčnosti této



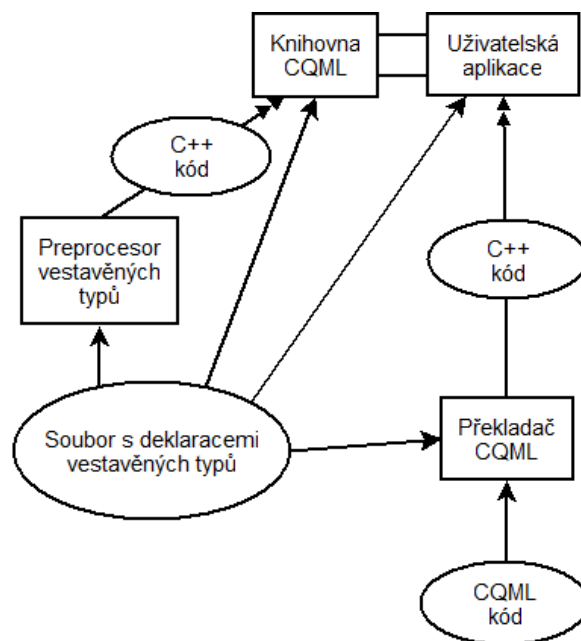
Obrázek 4.1: Návrh struktury aplikace.

knihovny, například přidáním dalších *GUI* elementů, které nevychází z funkčnosti již existujících, tudíž by to nebylo možné pouze jejich tvorbou pomocí *CQML* souborů. Prvky, které nejsou vytvořeny pomocí *CQML* souborů z již existujících prvků, se nazývají vesta-

věné prvky.

Použití nových vestavěných typů vyžaduje jejich deklaraci v hlavičkovém souboru používaném jak v aplikaci používající knihovnu, tak i v knihovně samotné, přičemž je také nutné, aby o jejich existenci a členech věděla i aplikace překladače *CQML*. Aby byla rozšiřitelnost co nejjednodušší, je potřeba, aby deklarace nových prvků byla na co nejmenším počtu míst. Z tohoto důvodu se deklarace nových vestavěných prvků spolu s jejich registrací pro potřeby překladače realizuje pomocí maker v jednom souboru, přičemž funkčnost těchto maker je v jednotlivých aplikacích rozlišena pomocí konstruktů `#ifdef`. Pro usnadnění rozšiřitelnosti jsou některé metody nových vestavěných prvků vygenerovány, jako například metody pro inicializaci nebo update hodnot atributů daného typu. Generování některých takovýchto metod nemusí být triviální problém, který by šlo vyřešit pomocí maker v rámci *C/C++* preprocesoru. Z tohoto důvodu je nutné přidat aplikaci, která předzpracuje vestavěné typy a vypíše zdrojové kódy vygenerovaných metod, které jsou použity v rámci knihovny.

Na obrázku 4.2 je ilustrován finální návrh struktury systému pro vývoj *GUI* v jazyce



Obrázek 4.2: Návrh struktury aplikace rozšířený o části pro předzpracování datových typů.

CQML. Dvojitě šipky představují vztah mezi jednotlivými bloky, kde na počátku šipky je blok, na jehož výstupu je zdrojový kód, zatímco na konci šipky je blok, do kterého je zdrojový kód zahrnut. Jednoduché šipky ilustrují, do kterých bloků je zahrnut soubor s deklaracemi vestavěných typů nebo vstup vstup *CQML* souborů.

4.1 Deklarace vestavěných typů

Jak již bylo zmíněno, deklarace nových vestavěných typů je prováděna pomocí maker v jednom souboru. Funkčnost se liší v závislosti na aplikaci, do které je soubor zahrnut. V pří-

Ukázka kódu 4.1: Makro registrující datový typ v případě, že je definováno jiné makro.

```
#ifndef REGISTRATION_APP
#define REGISTER_PRIMTYPE(x) RegisterPrimitive(x)
#else
#define REGISTER_PRIMTYPE(x)
#endif
```

padě knihovny a uživatelské aplikace daná makra vygenerují pouze kód definující struktury nových datových typů. V případě překladače a preprocesoru vestavěných typů jsou pomocí těchto maker volány funkce pro registraci nových datových typů spolu s funkcemi pro registraci jejich atributů. Ovšem k vygenerování kódu pomocí makra je nutné nejen makro vytvořit, ale i zajistit, že je někdy provedeno, proto se zde nachází i jednoduchý seznam maker, která se mají provést. Pro účely preprocesoru a překladače je potřeba i zaregistrovat primitivní typy, které mohou být používány v rámci jazyka *CQML*, proto se v seznamu prováděných maker nachází i makra pro registraci primitivních datových typů. Tato makra neprádí nic v případě knihovny *CQML* či uživatelské aplikace, avšak v případě preprocesoru a překladače zavolají funkce pro registraci primitivního datového typu.

Ve fragmentu kódu 4.1 je ukázka makra, které by zavolalo funkci *RegisterPrimitive* s parametrem *x*, když je *REGISTRATION_APP* definováno, přičemž neprovede nic, když *REGISTRATION_APP* definováno není.

Ukázka kódu 4.2: Makro které registruje strukturu nebo vypíše její deklaraci v závislosti na jiném makru.

```
#ifndef REGISTRATION_APP
#define REGISTER_STRUCTURE(macro , name) \
struct name { macro(DECL_ATTRIB) };
#else
#define REGISTER_STRUCTURE(macro , name) \
RegisterStructure(name);\
macro(REG_ATTRIB)
#endif

#define DECL_ATTRIB(type , name) type name;

#define REG_ATTRIB(type , name) RegisterAttribute(type , name);
```

Fragment kódu 4.2 ilustruje makra, která by mohla být použita pro registraci struktury v jedné aplikaci, kde je *REGISTRATION_APP* definováno, a zároveň by definovali strukturu tam, kde *REGISTRATION_APP* definováno není. Fragment kódu 4.3 pak ukazuje příklad použití maker, zatímco fragment 4.4 ukazuje vygenerovaný makry kód pro případ, že je *REGISTRATION_APP* definováno, a fragment 4.5 pro případ opačný. Uživatel tak může takovýmto způsobem přidat nový datový typ vytvořením makra pro vygenerování dané datové struktury a vložením názvu makra do seznamu prováděných maker.

Ukázka kódu 4.3: Ukázka makra, které vytvoří/zaregistruje určitou strukturu.

```
#define STRUCT_CUSTOM(OPERATION) \
OPERATION(int , first) \
OPERATION(char , second) \

REGISTER_PRIMTYPE(char)
REGISTER_PRIMTYPE(int)
REGISTER_STRUCTURE(STRUCT_CUSTOM, Custom)
```

Ukázka kódu 4.4: Kód vygenerovaný makrem který zaregistruje strukturu.

```
RegisterPrimitive(char);
RegisterPrimitive(int);

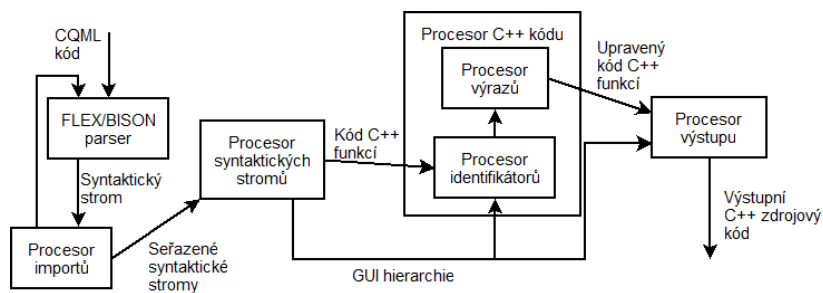
RegisterStructure(Custom);
RegisterAttribute(int, first);
RegisterAttribute(char, second);
```

4.2 Překladač

Cílem je vytvořit aplikaci překladače, která přemění kód z jazyka *CQML* pro deklarativní *GUI*, kód importovatelný do aplikace v jazyce *C++*. Tudíž na vstupu je soubor se zdrojovým kódem v jazyce *CQML* a výstupem je zdrojový kód v jazyce *C++*, ve formě definic datových struktur pro *GUI* a funkcí.

Výsledný překladač je navržen tak, že lze rozdělit do několika bloků, viz. Obrázek 4.3.

Cílem prvního bloku (FLEX/BISON Parser) je přečíst zdrojový kód z *CQML* souboru a ově-



Obrázek 4.3: Návrh struktury překladače.

Ukázka kódu 4.5: Deklarace struktury vygenerovaná pomocí makra.

```
struct Custom { int first; char second; };
```


řit jeho syntaktickou správnost. V případě, že je kód syntakticky v pořádku, je výstupem tohoto bloku syntaktický strom. Pro tento blok jsou použity zdrojové kódy vygenerované pomocí aplikací *Bison* a *Flex* (viz. příloha B), z *YACC* gramatiky pro *CQML*.

V syntaktickém stromu jsou nalezeny příkazy pro import souborů (blok Procesor importů). Tyto soubory jsou následně otevřeny a postupně zpracovány totožným způsobem.

Jakmile jsou všechny importované soubory zpracovány a jsou jejich zpracováním vytvořeny jejich syntaktické stromy, jede zjištěno, zda jsou některé soubory cyklicky závislé. To se realizuje tak, že se nejprve zkonstruuje orientovaný graf, kde každý vrchol představuje soubor a závislost mezi soubory je znázorněna hranou (směřující k importovanému souboru). Následně se ověří, zda v grafu existuje orientovaný cyklus. Pokud v grafu takový cyklus existuje, pak jsou soubory cyklicky závislé, což vyústí v chybu a ukončení programu. Je-li graf acyklický, jsou podle něho topologicky seřazeny syntaktické stromy jednotlivých souborů (vrcholy) pro následné zpracování.

V dalším bloku (Procesor syntaktických stromů) se zpracuje syntaktický strom a podle něj se vytvoří *GUI* hierarchie. Pro každý uzel představující *GUI* element, atribut, funkci či nový atribut se vytvoří instance příslušné třídy. Každému elementu jsou podle jeho potomků v syntaktickém stromu přiřazeny potomci v *GUI* hierarchii a jeho atributy a funkce. Následně se zpracují všechny elementy s definovaným atributem *id*, a tyto identifikátory se použijí ke tvorbě mapy, pomocí níž je umožněno přistupovat k danému elementu podle definovaného identifikátoru.

V další části programu (Procesor *C++* kódu) se zpracují kódy funkcí do formy potřebné pro závěrečný výstup v jazyce *C++*.

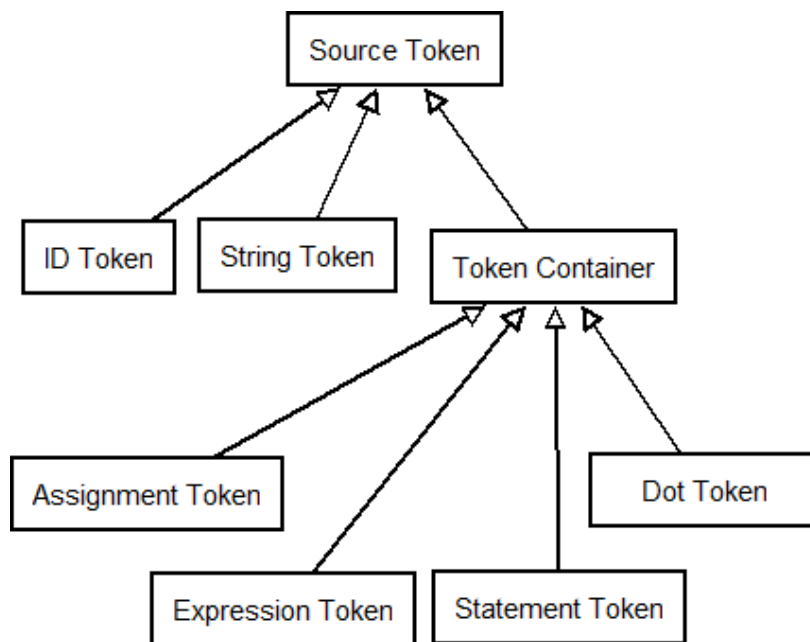
4.2.1 Zpracování kódu v jazyce *C++*

Syntaktické stromy jednotlivých funkcí (případně výrazů) se projdou a převedou se na pole samostatných symbolů (tokenů) takovým způsobem, že jeden prvek v poli představuje list stromu. Vzhledem k nutnosti kontroly existence atributů, jsou pro snadnější zpracování některé symboly v poli seskupeny do jednoho, který má v sobě uloženo pole symbolů, které seskupil. Vzhledem k tomu, že každý symbol může obsahovat další symboly, se stále jedná o stromovou strukturu, nicméně, má nižší hloubku.

Na obrázku Obrázek 4.4 je znázorněn diagram tříd symbolů. Všechny třídy představující symboly dědí z jedné abstraktní třídy *SourceToken*. Symboly, jež sdružují skupinu symbolů do jednoho, dědí z třídy *TokenContainer*, která dědí z třídy *SourceToken*. *ID Token* představuje nějaký identifikátor uvnitř kódu, zatímco *String Token* představuje jakýkoli jiný řetězec. *Dot Token* sdružuje symboly pro výraz, v němž se pomocí operátoru „.“ přistupuje k atributům. *Statement Token* představuje libovolný výrok, *Assignment Token* představuje přiřazovací výraz a třída *Expression Token* libovolný jiný výraz.

Zpracování identifikátorů

Kódy (resp. syntaktické stromy) jednotlivých funkcí se projdou, a naleznou se identifikátory, jež se nachází v mapě identifikátorů elementů (id atributy elementů), a tyto identifikátory se v kódu nahradí skutečným názvem elementu uvnitř výsledné *GUI* aplikace.



Obrázek 4.4: Diagram tříd symbolů.

Dynamická kontrola typů a přiřazování hodnot

Jelikož je do atributů některých typů umožněno ukládat reference na objekty různého typu a pomocí této reference je možné přistupovat k atributu daného objektu, nemůže být během kompilace určeno, že atribut, ke kterému se aplikace snaží přistoupit, je v daném objektu přítomen. Proto je potřeba za běhu výsledné *GUI* aplikace (využívající kód na výstupu překladače) kontrolovat přítomnost takového atributu.

Uvažujme kód 4.6. Z kódu je zřejmé, že u atributu *ref* je potřeba zkontrolovat, zda má objekt referovaný tímto atributem atribut *ref2* a objekt referovaný *ref2* obsahuje *ref3*.

Ukázka kódu 4.6: Pseudokód problematického použití operátoru „.“ v přiřazovacím výroku.

```
width = ref1.ref2.ref3
```

Oproti nahrazení identifikátoru názvem elementu nestačí pouze nahradit část textového řetězce ve výrazu a k získání hodnoty atributu je použita metoda *Get*, která jako parametr přijímá název atributu. Kontrola existence atributu je provedena uvnitř této metody a v případě, že atribut neexistuje vyhodí výjimku. Výsledný kód tak vypadá jako v ukázce kódu 4.7.

Ukázka kódu 4.7: Řešení v pseudokódu problematického použití operátoru „.“ v přiřazovacím výroku.

```
width = ref1.Get("ref2").Get("ref3");
```

4.2.2 Výstup

Poslední blok se postará o výpis do výstupních souborů. Pro každý zpracovaný vstupní soubor se vytvoří hlavičkový soubor s deklarací struktur a hlaviček funkcí. Každý soubor má vlastní inicializační funkci pro inicializaci hierarchie a počáteční přiřazení hodnot a ukazatelů na potřebné funkce.

Pro každý *GUI* element, kterému je pomocí příkazu přidán nový atribut, se definuje nový typ struktury, která dědí z původního typu struktury. Každý přidávaný atribut se deklaruje v nové struktuře, spolu s ukazatelem na funkci, která slouží k jeho updatování. Pro každou takovou strukturu se také vytvoří funkce, které takovou strukturu alokují a inicializují, přičemž se v dané funkci zavolá funkce pro inicializaci struktury, ze které se dědí.

Pro všechny elementy se vytvoří funkce pro jejich update, která slouží k updatování hodnot v attributech elementů. Pro každý atribut, jemuž je přiřazena nějaká hodnota v *CQML* souborech, je definována funkce pro jeho update zadanou hodnotou. Ukazatel na danou funkci je přiřazen do členské proměnné uvnitř struktury určené pro daný atribut.

4.3 Preprocesor datových typů

Tato aplikace využívá výše zmíněný soubor s deklaracemi vestavěných typů, které jsou realizovány pomocí maker (viz. sekce 4.1). Pro účely předzpracování datových typů v rámci preprocesoru se pomocí maker z daného souboru zaregistrují jednotlivé vestavěné typy spolu se seznamem jejich atributů. Také se zvlášť zaregistrují primitivní typy. Ze seznamu atributů se vygenerují hašovací funkce pro všechny vestavěné typy pomocí algoritmu popsáního v sekci 4.3.2.

4.3.1 Generování výstupního kódu

Účelem preprocesoru je vygenerovat zdrojové kódy pro použití v runtime knihovně *CQML*. Vygenerují se zdrojové kódy metod pro přístup k obecnému atributu pomocí jeho jména. To, k jakému atributu se přistupuje, je rozlišeno pomocí hodnoty, která je výstupem příslušné hašovací funkce. Také se vygenerují metody pro update hodnot atributů daného vestavěného typu pomocí updatovacích funkcí, které mohly být přiřazeny v rámci překladače. Dále se vygeneruje kód konstruktorů, které inicializují hodnoty atributů na jejich výchozí hodnotu, přičemž se v jejich rámci také inicializuje hodnota identifikátoru typu. Hodnota tohoto identifikátoru, který je pro každý typ unikátní, slouží k výběru příslušné hašovací funkce pro daný typ v metodě pro přístup k obecnému atributu pomocí jeho jména. Kód, který je výstupem tohoto preprocesoru, musí být zahrnut do kódu pro runtime knihovnu *CQML*.

4.3.2 Hašování

Hašovací funkce je zobrazení. Je-li dána množina W slov o konečné délce skládajících se ze symbolů abecedy, pak je hašovací funkce takové zobrazení, které přiřadí každému slovu z

množiny W , právě jednu celočíselnou hodnotu z nějakého intervalu $[0, k - 1]$. Pokud nějakým dvěma různým slovům z množiny W je hašovací funkcí přiřazena stejná celočíselná hodnota, pak se hovoří o tzv. kolizi. Dokonalá hašovací funkce, je taková funkce, která pro každé slovo z dané množiny W vygeneruje unikátní celočíselnou hodnotu v nějakém intervalu $[0, k - 1]$. V takovém případě se hovoří o tzv. bezkolizní nebo dokonalé hašovací funkci. Jsou-li slova z množiny W klíči datových prvků nějaké množiny dat, pak lze mít taková data uložená v poli tak, že lze s pomocí bezkolizní hašovací funkce přímo přistupovat k jednotlivým prvkům, jelikož výsledkem takové funkce může být index do pole dat. Dané pole však by muselo mít naalokováno k prvků a to i v případech, že pro počet m skutečných datových záznamů platí $m < k$. Tudíž je jasné, že by pro ideální hašovací funkci pro tento účel platilo, že $m = k$, přičemž by tato funkce musela zůstat bezkolizní. Taková funkce se nazývá minimální dokonalá hašovací funkce. Výhodou minimální dokonalé hašovací funkce je, že se dá použít pro efektivní uskladnění dat označených klíči. Nevýhodou je výpočetně náročné hledání takové funkce, což má za následek to, že se v praxi nedá použít pro dynamicky se měnící soubory klíčů. Tudíž se používá pouze pro statické množiny klíčů. Za další nevýhodu lze považovat, že taková funkce může mít vyšší paměťovou náročnost pro uchování svých parametrů a také může mít náročnější výpočet, i když ten je stále asymptoticky konstantní pro konečný počet klíčů. Aplikace využívá algoritmus pro nalezení dokonalé hašovací funkce z [24], jehož výsledná hašovací funkce má tvar 4.1. Kde f_1 a f_2 jsou funkce mapující řetězec na celočíselnou hodnotu, zatímco g je funkce mapující libovolné celočíselné hodnoty do rozsahu $[0, m - 1]$, přičemž m je počet klíčů.

$$h(w) = (g(f_1(w)) + g(f_2(w))) \bmod m \quad (4.1)$$

Tento algoritmus je založen na metodě generování náhodných neorientovaných grafů. Na počátku je graf o N vrcholech, přičemž jeho hrany jsou definovány tak, že každá hrana představuje jedno slovo z množiny klíčů. Výsledky funkcí f_1 a f_2 , na jejichž vstupu je příslušné slovo, určují počáteční a koncový vrchol takové hrany. Je-li výsledný graf acyklický, lze najít takové ohodnocení jeho vrcholů, že součet hodnot koncových vrcholů pro každou hranu vygeneruje unikátní číslo v rozsahu $[0, m - 1]$. Cílem je tedy najít funkce f_1 a f_2 . Tyto funkce na vstupu přijímají slovo z množiny klíčů a na výstupu je celočíselná hodnota v rozsahu $[0, N - 1]$. Vzorec 4.2 ukazuje takovou funkci. T_k je v tomto případě jednorozměrná tabulka, která obsahuje hodnotu pro každou možnou pozici ve slově (tudíž její velikost je rovna maximální délce slova v množině klíčů), a $w[i]$ je číselná hodnota znaku slova. Tabulky T_1 a T_2 se hledají generováním náhodných čísel v rozsahu $[0, N - 1]$. N je nějaké zvolené číslo, které musí být větší než m , jinak by nebyl vygenerovaný graf nikdy acyklický. Pro příliš nízké hodnoty hrozí, že algoritmus trvá příliš dlouho, či nikdy neskončí, zatímco pro příliš vysoké hodnoty zabere uchování grafu větší místo v paměti.

$$f_k(w) = \left(\sum_{i=1}^{|w|} T_k(i) \times w[i] \right) \bmod n \quad (4.2)$$

$$g(v) := (h(e = (u, v)) - g(u)) \bmod m \quad (4.3)$$

Algoritmus se skládá ze dvou fází, mapování a přiřazení. V první fázi se vygenerují náhodně tabulky T_1 a T_2 . Pomocí funkcí f_1 a f_2 se vytvoří graf výše popsaným způsobem a následně se zjistí, zda je graf acyklický, je-li tomu tak, pak se fáze opakuje, jinak se pokračuje fází

přiřazení. Ve fázi přiřazení se každému vrcholu grafu přiřadí hodnota, jež je použita pro výpočet hašovací funkce. Postupně se prochází jednotlivé vrcholy. Nebyl-li nějaký vrchol u ještě navštíven, je ohodnocen hodnotou nula a zpracují se jeho sousedé a to tak, že se jim přiřadí hodnota vypočtená podle vzorce 4.3, kde h vrací index slova v množině klíčů, které přísluší hraně spojující souseda v s vrcholem u , a g je ohodnocení vrcholu. Soused je pak rekurzivně zpracován, jako byl zpracován vrchol u .

$$h(w) = (((\sum_{i=1}^{|w|} T_1(i) \times w[i]) \bmod n) + ((\sum_{i=1}^{|w|} T_2(i) \times w[i]) \bmod n)) \bmod m \quad (4.4)$$

Výsledná hašovací funkce je ilustrována vzorcem 4.4. Pro její uchování tedy stačí uchovat ohodnocení všech vrcholů grafu g a hodnoty tabulek T_1 a T_2 .

4.4 Knihovna

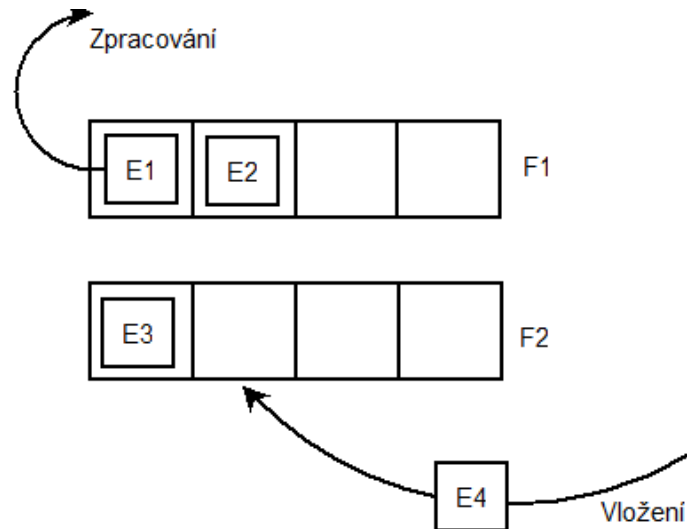
K zajištění co největší multiplatformní podpory *CQML* aplikací, aplikace podporuje vyměnitelné externí rozhraní pro vykreslování, vstup a správu zdrojů (např. obrázky, fonty). Proto tato funkčnost není implementována v rámci knihovny, ale musí být poskytnuta v rámci uživatelské aplikace. Jelikož knihovna je předkompilovaná nezávisle na uživatelské aplikaci, není možné, aby z knihovny byly přímo volány funkce v uživatelské aplikaci, protože nejsou v době kompilace knihovny ještě známy. Tudíž není možné volat například vykreslovací funkce zevnitř knihovny. Tento problém lze vyřešit předáním ukazatele na funkci v uživatelské aplikaci prostřednictvím funkce poskytované knihovnou. Vzhledem k velkému počtu vykreslovacích funkcí jsou funkce sdruženy pod abstraktní třídu (interface) jako virtuální metody, které musí implementovat uživatelská aplikace.

Ve většině realtime aplikací jsou aplikační výpočty a vykreslování rozděleny do dvou oddělených fází, které se neprotínají. Fáze s aplikačními výpočty se zpravidla nazývá *Update*. Tyto fáze jsou tudíž odděleny i v případě knihovny. Během vykreslovací fáze je vykreslována aktuální *GUI* hierarchie, zatímco během fáze *Update* se přepočítávají hodnoty uvnitř jednotlivých elementů.

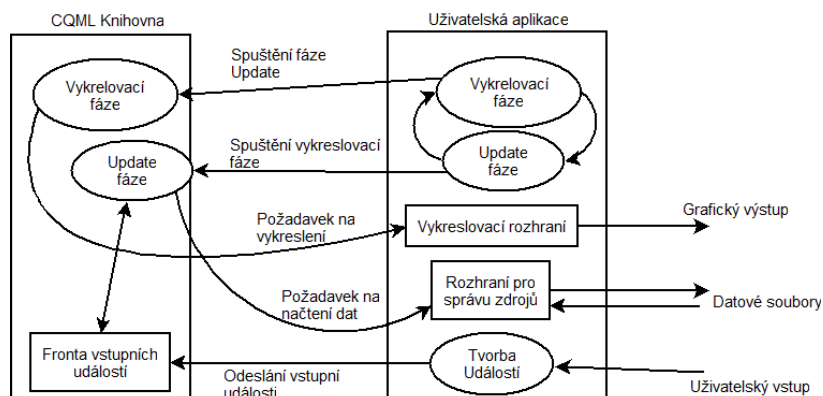
4.4.1 Vstup

V realtime aplikacích se v některých případech bere vstup ze vstupních zařízení v odděleném vláknu, než v jakém probíhá hlavní vlákno aplikace, to má za úkol zajistit, že aplikace nezamrzne vůči uživatelově vstupu v případě, že právě provádí složitý výpočet či vykresluje složitý obraz. Proto knihovna podporuje tento přístup. Vstup se zpracovává pomocí událostí, události musí generovat uživatelská aplikace, která je prostřednictvím funkce ukládá do fronty pro jejich zpracování. Vkládat události do fronty je umožněno i v odděleném vláknu, proto knihovna využívá threadsafe frontu, která je realizována pomocí dvou oddělených front, přičemž do jedné fronty je možno vkládat události, zatímco se druhá zpracovává (viz. obrázek 4.5). Zpracování fronty probíhá na počátku fáze *Update*, jakmile se fronta zpracuje, prohodí se fronta pro vkládání událostí a pro zpracování. Fronta pomocí vláknových zámků zajišťuje, že při vkládání zároveň s prohozením front, nebo při vkládání událostí ze dvou

vláken v tentýž okamžik, nedojde k porušení atomicity jednotlivých operací. Komunikace mezi knihovnou a uživatelskou aplikací je ilustrována na obrázku 4.6.



Obrázek 4.5: Návrh realizace fronty.



Obrázek 4.6: Návrh komunikace mezi knihovnou a uživatelskou aplikací.

4.4.2 Správa GUI

Hierarchie vygenerovaného GUI je uchovávána v podobě stromu. Na vrcholu stromu se tudíž nachází kořenový element, jež může mít libovolný počet potomků, přičemž každý takovýto potomek je element, který může mít opět neomezený počet potomků. Každý vestavěný datový typ, který je přímo součástí hierarchie a ne pouze atributem nějakého elementu, dědí z obecné struktury *Element*. Každý uzel v rámci stromu GUI, tak je typu *Element* nebo

typu, který je z něj odvozen. Typ *Element* je struktura, která obsahuje pole jeho potomků, atributy souřadnice a velikost. Struktura *Element* vlastní virtuální metody *Draw* a *Update*, které mohou struktury z nich odvozené přetěžovat.

Každý element může mít atributy, které jsou přístupné zevnitř kódu *CQML*. Takové atributy uvnitř struktury reprezentující daný element nejsou představovány pouze členem pro uchování jejich hodnoty, ale také ukazatelem na funkci, která může sloužit k přepočítání hodnoty daného atributu v rámci fáze *Update*. Funkce, na kterou daný ukazatel ukazuje, je vygenerována pomocí překladače a je danému atributu přiřazena. Jelikož se jedná o ukazatel na funkci a nikoliv o metodu samotného elementu, je uchováván i tzv. kontext funkce. Kontextem funkce je v tomto případě myšlen *CQML* soubor, v jehož rámci byla daná funkce specifikována, což znamená, že tato funkce by měla být schopna přistupovat ke všem elementům v hierarchii specifikované v daném souboru. Jelikož každý *CQML* soubor představuje samostatnou komponentu, kontext je tak uložen ve formě struktury, která obsahuje ukazatel na kořenový prvek v rámci dané komponenty (souboru) a také ukazatel na element jemuž byla funkce s daným kontextem přiřazena.

Každý element v *GUI* hierarchii má metodu *Update*, kterou dědí z typu *Element*. Tato metoda má za úkol přepočítat hodnoty atributů, jsou-li jim přiřazeny funkce pro jejich přepočítání. Dále tato metoda zavolá metodu *Update* všech potomků daného elementu.

4.4.3 Vykreslování

Aby bylo zajištěno, že se element nacházející výše v hierarchii vždy vykreslí před elementy, jež jsou v hierarchii níže, tak se pro vykreslování použije fronta. Na počátku vykreslovací fáze se do ní zařadí pouze prvek na vrcholu hierarchie. Na prvek ve frontě se zavolá jeho vykreslovací metoda, která jej vykreslí a zároveň přidá jeho potomky na konec fronty. Po vyprázdnění fronty se ukončí fáze vykreslování.

Jak již bylo zmíněno, samotné vykreslovací funkce jsou z knihovny pouze volány pomocí vykreslovacího rozhraní poskytovaného uživatelskou aplikací. Tyto funkce jsou volány zevnitř přetížených metod *Draw*, jednotlivých elementů. Každý vestavěný typ, který dědí z typu *Element* může přetěžovat vykreslovací metodu *Draw*.

Jelikož by knihovna *CQML* měla umožnit použití externích zdrojů, jako jsou například obrázky či fonty, v obecném formátu. Správa těchto dat je obsluhována uvnitř uživatelské aplikace podobně, jako je tomu u vykreslování, pomocí speciálního rozhraní. Každý element, který taková data vyžaduje může v rámci své přetížené metody *Update* dát do fronty požadavek pro načtení dat.

Tyto požadavky jsou zpracovány na konci fáze *Update* zavoláním příslušné metody rozhraní pro načtení dat. Metody rozhraní vrací hodnotu typu *void**, což může představovat ukazatel na libovolná data. Tato hodnota pak je uložena do mapy pod specifickým identifikátorem, v případě obrázku je takovým identifikátorem název souboru, v případě fontu je tento identifikátor název fontu, který má za sebou připojeny parametry fontu ve formě textového řetězce. Kdykoliv pak knihovna vyžaduje manipulaci s externím zdrojem (například předáním funkci pro vykreslení), učiní tak pomocí dané hodnoty typu *void**.

Ukázka kódu 4.8: Ukázka *CQML* kódu ilustrující nemožnost staticky určit existenci specifického atributu.

```
Element
{
    id: firstElement;
    property Element ref:secondElement;
    width :    ref.status;
    MouseClick :
    {
        ref=thirdElement;
    };
};
Element
{
    id: secondElement;
    property int status : 0;
};
Element
{
    id: thirdElement;
};
```

4.4.4 Inicializace

Před použitím *CQML* knihovny v rámci uživatelské aplikace je nutné knihovnu nejprve inicializovat. Během inicializace je nutné knihovně předat ukazatele na instance objektů implementující rozhraní, jež slouží ke správě externích zdrojů a vykreslování. Také je potřeba předat knihovně ukazatel na funkci pro inicializaci hašovacích funkcí.

4.4.5 Variant

Každý element může mít atribut v němž se uchovává reference na jiný element. Každý element může být rozšířen o atribut libovolného typu a názvu. Jelikož se za běhu programu může změnit, na který element reference ukazuje, nelze při přístupu k atributům referencovaného elementu staticky určit jejich typ ani to, zda daný element takový atribut vůbec má. Toto je problém, jelikož jazyk *C/C++* je staticky typovaným jazykem, což znamená, že není možné manipulovat s hodnotou, aniž by byl v době kompilace znám její typ. Toto je ilustrováno v kódu 4.8, kde se po kliknutí na první element změní, na který element reference ukazuje. Z tohoto důvodu je k atributům takového elementu nutno přistupovat pomocí virtuální metody, která musí vracet nějaký obecný typ, do kterého může být uložena hodnota libovolného typu, se kterou je následně možno manipulovat.

Pro manipulaci s hodnotami různého datového typu uniformním způsobem lze v jazyce *C/C++* použít konstrukt *union*. Konstrukt *union* však nelze použít v objektově orientova-

ném prostředí, jelikož nepodporuje použití datových typů s netriviálním konstruktorem či destruktorem, datové typy s triviálním konstruktorem se označují zkratkou *POD* („Plain old data“), také se označují jako pasivní datové struktury.

Pro jazyk *C/C++* existuje knihovna *Boost* [25], která poskytuje řešení v podobě šablony *variant*. Šablona *variant* umožňuje uložení libovolného datového typu, ať už se jedná o typ *POD* či nikoliv.

V jazyce *QML* tuto funkci má datový typ *var*. Tento typ má obdobnou funkci jako proměnná v jazyce *JavaScript*, tudíž je do něj možné uložit data libovolného datového typu. Proměnný typ *var* tak umožňuje nejen ukládání primitivních a složených datových typů, ale lze jej také použít k uložení referencí na grafické elementy. Tento typ také podporuje provádění aritmetických operací, uložení do jiné proměnné libovolného typu, přičemž pokud dojde k použití aritmetické operace mezi nekompatibilními typy (například mezi referencí na element a číselným typem), či pokusu o uložení do proměnné nekompatibilního typu, pak *QML* aplikace vyhodí výjimku.

Pro účely knihovny *CQML* je tedy vhodné vytvořit datový typ, který by se svou funkčností co nejvíce podobal typu *var* jazyka *QML*. Tento typ je nazýván *Variant*. Pro odlišení mezi šablonou *Variant* poskytovanou knihovnou *boost* má název tohoto typu velké počáteční písmeno.

Typ *Variant* v knihovně *CQML* přetěžuje operátory aritmetických operací a operátor přiřazení. Typ *Variant* podporuje konverzi na libovolný datový typ pomocí metody *As*, která je definována pomocí šablony, jež přijímá požadovaný datový typ jako parametr. Typ *Variant* uchovává identifikátor svého typu a svou hodnotu. Za účelem, aby se každá operace prováděla jen mezi totožnými typy, je zavedena konvence, že při použití binárního operátoru je vždy pravý operand přetypován na typ levého operandu. Každá konvence mezi typy, které nejsou kompatibilní (například mezi nečíselným a číselným typem), vyústí v chybu.

Typ *Variant* implementuje metodu *Get*, která v případě, že *Variant* obsahuje referenci na nějaký element, umožní získat hodnotu atributu jehož název byl metodě předán jako parametr (podobně jako je tomu u složených typů používaných v rámci knihovny *CQML*). V případě, že je v typu *Variant* uložen primitivní typ, který nemá žádné atributy, pak dojde k ohlášení chyby, podobně jako by tomu bylo v případě žádosti o získání neexistujícího atributu.

Ukázka kódu 4.10: Přeložený kód funkce, který ilustruje nutnost vrácení reference metodou *Get*.

```
ref .Get (" status ")=1;
```

Při manipulaci s elementem pomocí reference může dojít k situaci, kdy je potřeba změnit hodnotu nějakého z jeho atributů. Tato situace je ilustrována v ukázce kódu 4.9, kde po kliknutí na první element se má pomocí reference změnit hodnota atributu *status* druhého elementu. Po zpracování se vnitřek funkce pro obsluhu události kliknutí převede na kód v 4.10. Problémem, který zde nastává je, že metoda *Get* vrací hodnotu typu *Variant*, která není nijak vázaná na adresu, kde byla původní hodnota uložena. Proto je potřeba vytvořit další datový typ, který narozdíl od typu *Variant* namísto hodnoty uchovává adresu atributu, kde byla původní hodnota uložena. Tento datový typ budiž nazván *VariantRef*. *VariantRef* implementuje metodu *Get*, stejně jako ostatní typy používané v rámci *CQML*. *VariantRef* také přetěžuje operátor přiřazení, přičemž při použití tohoto operátoru se změní hodnota na adrese, na níž *VariantRef* ukazuje. Kvůli možnosti změnit hodnotu referencovaného atri-

Ukázka kódu 4.9: *CQML* kód měnící hodnotu atributu elementu prostřednictvím reference na daný element.

```

Element
{
    id: firstElement;
    property Element ref:secondElement;
    MouseClick :
    {
        ref.status = 1;
    };
};
Element
{
    id: secondElement;
    property int status : 0;
};

```

Ukázka kódu 4.12: Kód ilustrující operace s typy *Variant* a *VariantRef*.

```

reference1.Get("att1") = reference2.Get("att2")
    = reference3.Get("att3") + 3;

```

butu metody *Get* všech typů v rámci *CQML* (včetně *VariantRef* a *Variant*) tedy vrací typ *VariantRef* namísto *Variant*. Jelikož typ *Variant* podporuje narozdíl od *VariantRef* i aritmetické operace, *VariantRef* podporuje implicitní přetypování na typ *Variant* zkopírováním hodnoty na adrese, na níž ukazuje *VariantRef*, do nové instance typu *Variant*.

Ukázka kódu 4.11: Řádek *CQML* kódu ilustrující přístup k atributům elementů uložených v referenci.

```

reference1.att1 = reference2.att2 = reference3.att3 + 3;

```

Fragment 4.12 ukazuje, jak by vypadal kód funkce z fragmentu 4.11 po přeložení do *C++*. Výsledky metod *Get* jsou hodnoty typu *VariantRef*, přičemž výsledek operace sčítání v daném kódu je typu *Variant*. Tato hodnota je následně uložena do atributu *att1* v prvku referencovaném v *reference1* a hodnota atributu *att2* v prvku referencovaném v *reference2*.

Kapitola 5

Implementace

5.1 Přidávání vestavěných typů

Vestavěné struktury lze rozdělit na dvě různé skupiny. První skupina jsou struktury, ke kterým se v rámci aplikace přistupuje jako k referenčním typům, tudíž při manipulaci s nimi se manipuluje s ukazatelem ukazující na jejich umístění v paměti. Ve druhé skupině jsou struktury, které fungují jako hodnotové typy, takže se manipuluje přímo s jejich obsahem (hodnotou).

Zdrojový kód je sdílený mezi několika aplikacemi. Pomocí preprocesorového konstruktu `#ifdef CQML_PARSER` je rozlišeno, jaké definice maker se v dané aplikaci používají. Přičemž makro `CQML_PARSER` je definováno pro aplikaci překladače a preprocesoru vestavěných typů, zatímco pro grafickou knihovnu a uživatelskou aplikaci definováno není. Výstupem jsou tedy dvě verze kódu. Verze s definovaným makrem `CQML_PARSER` je dále označována jako parser verze. Verze bez definice makra `CQML_PARSER` je dále označována jako runtime verze.

Prvním makrem je makro `REGISTRATION`, toto makro je jedinné voláno přímo, zatímco ostatní makra pro definici vestavěných datových typů jsou volána prostřednictvím tohoto makra ať už přímo nebo transitivity. Makro `REGISTRATION` se v parser verzi nachází uvnitř funkce `RegisterBasicTypes`, která je volána na začátku aplikací překladače a preprocesoru vestavěných typ. V runtime verzi se toto makro nachází uvnitř namespace `CQML_GUI`. Makro samotné v obou verzích má totožnou funkčnost. `REGISTRATION` přijímá tři parametry, jimiž jsou makra. První je reprezentováno názvem `MACRO2`, které slouží pro registraci nové struktury hodnotového typu, zatímco druhým parametrem je `MACRO2REF`, které slouží pro registraci struktury referenčního typu. Třetím parametrem je `MACRO3`, což slouží pro registraci struktury dědicí z jiného již registrovaného datového typu. `MACRO2` a `MACRO2REF` přijímá jako první parametr název makra pro generování datové struktury, jako druhý parametr přijímá název datové struktury. `MACRO3` má totožné první dva parametry, ale má i třetí parametr, který je názvem datového typu, ze kterého generovaná struktura dědí.

Na počátku makra `REGISTRATION` se volají makra `REGPRIMITIVE` pro primitivní datové typy, jako jsou například `int`, `char` či `float`. V parser verzi se pomocí makra `REGPRIMITIVE` zavolá funkce `RegisterPrimitive`, která jako parametr přijímá název primitivního datového typu, který byl makru předán jako parametr. Makro `REGPRIMITIVE` v případě

runtime verze neprovede nic. Ve zbytku makra *REGISTRATION* se volají makra, která mu byla předána v parametrech, s parametry pro registraci jednotlivých datových typů.

Makru *REGISTRATION* jsou v runtime verzi předána makra *MAKE_STRUCTURE2* (toto makro je předáno jako první dva parametry) a makro *MAKE_STRUCTURE3*. Zatímco v parser verzi jsou předána makra *PARSER_DECLARE2*, *PARSER_DECLARE2-REF* a *PARSER_DECLARE3*.

Makra *MAKE_STRUCTURE2* a *MAKE_STRUCTURE3*, mají téměř totožnou funkčnost. Vytvoří tělo struktury s definovaným názvem, konstruktorem a metodou *Get* pro získání libovolného atributu. Jedinným rozdílem je, že v případě makra *MAKE_STRUCTURE3* dědí vygenerovaná struktura z typu, jehož název byl makru předán pomocí třetího parametru, zatímco struktura vygenerovaná makrem *MAKE_STRUCTURE2* dědí ze struktury *CQMLObject*. Makro, které bylo předáno jako první parametr, slouží k vygenerování kódu pro deklaraci atributů v těle dané struktury, pomocí maker několika maker, jež mu jsou předány jako parametry.

Makra *PARSER_DECLARE2*, *PARSER_DECLARE2REF* a *PARSER_DECLARE3*, volají funkci, které jako první parametr předají jméno struktury (druhý parametr předaný makru) a jako druhý parametr předají nulu, v případě prvních dvou maker, a v případě třetího makra se předá jméno rodiče (třetí parametr předaný makru). První a třetí makro takto volá funkci *RegisterStruct* a druhé makro takto volá funkci *RegisterStructRef*. Všechny tři makra na závěr zavolají makro, jenž jim bylo předáno prostřednictvím prvního parametru. Volané makro slouží k zavolání funkcí pro registraci jednotlivých atributů, pomocí několika maker, jež mu jsou předány jako parametry.

Zmíněná makra *PARSER_DECLARE2*, *PARSER_DECLARE2REF*, *PARSER_DECLARE3*, *MAKE_STRUCTURE2* a *MAKE_STRUCTURE3* přijímají jako první parametr stejný druh makra, respektive makra vycházející ze stejného vzoru. Toto makro slouží pro vygenerování deklarace atributů v runtime verzi a pro registraci atributů v parser verzi. Pro každý vestavěný typ je jedno takové makro. Makro přijímá pět maker jako a svoje parametry v následujícím pořadí:

- MF** slouží pro vytvoření atributu, který je přístupný pouze ve vnitřním *C++* kódu knihovny. V parser verzi toto makro zaregistruje daný atribut. V runtime verzi toto makro deklaruje atribut uvnitř struktury.
- F** slouží pro vytvoření atributu, který je přístupný uvnitř *CQML* souborů i ve vnitřním *C++* kódu knihovny. V parser verzi toto makro zaregistruje daný atribut. V runtime verzi toto makro deklaruje atribut uvnitř knihovny, spolu s ukazatelem na funkci pro jeho výpočet a ukazatelem na kontext, v jakém daná funkce operuje.
- M** vytvoří ukazatel na funkci, která může sloužit například k obsluze událostí. V parser verzi toto makro zaregistruje daný ukazatel. V runtime verzi toto makro deklaruje ukazatel na funkci spolu s ukazatelem na kontext, v jakém daná funkce pracuje.
- ME** deklaruje metodu, kterou je nutno implementovat v rámci zdrojového kódu knihovny. V parser verzi toto makro neprovede nic. V runtime verzi deklaruje hlavičku metody.
- MEV** deklaruje virtuální metodu, kterou je nutno implementovat v rámci zdrojového kódu knihovny. V parser verzi toto makro neprovede nic. V runtime verzi deklaruje hlavičku virtuální metody.

Všechna uvedená makra přijímají jako první parametr typ (v případě metod a ukazatelů na funkce je to návratový typ), jako druhý parametr název, jako třetí parametr výchozí hodnotu (tento parametr je irelevantní pro metody). Makra *M*, *ME* a *MEV* přijímají jako zbylé parametry seznam parametrů daných funkcí (resp. metod).

5.2 Překladač

5.2.1 Zpracování vstupu

Pomocí generátoru *Bison* (viz. příloha B) byl podle gramatiky jazyka *CQML* vytvořen parser. Pro vygenerování lexikálního analyzátoru byl použit generátor *FLEX* (viz. příloha B). Zdrojový kód parseru a lexikálního analyzátoru je pak použit v překladači.

Po spuštění program načte výchozí typy *GUI* elementů (pomocí funkcí vygenerovaných prostřednictvím maker v sekci 5.1), uloží si jejich názvy a seznam jejich atributů, včetně jejich typů a výchozích hodnot, do instancí třídy *ClassContainer*.

Program na vstupu přijímá jméno souboru, se zdrojovým kódem v jazyce *CQML*. Pokud daný soubor existuje, je otevřen a jeho obsah je předán parseru. Pokud parser detekuje syntaktickou chybu ve zdrojovém souboru, vypíše chybu a program se ukončí. Výstupem parseru je syntaktický strom. V první fázi se ve stromu vyhledají příkazy pro import elementu z jiných souborů. Totéž program cyklicky provede pro všechny soubory, z nichž se elementy importují.

5.2.2 Konstrukce a zpracování pomocných struktur

Následně je vytvořen orientovaný graf vzájemných závislostí mezi jednotlivými soubory, kde každý vrchol představuje soubor a každá hrana představuje vazbu mezi soubory, zatímco počátečním vrcholem hrany je vrchol představující soubor, do něhož je element importován, a koncovým vrcholem je vrchol představující soubor, z něž byl element importován. Pomocí algoritmu prohledávání do hloubky je zjištěno, zda se v grafu nachází cyklus. Pokud se v grafu nachází cyklus, je vypísána chybová hláška a program se ukončí.

Pokud je graf acyklický, pak každý vrchol představuje komponentu a tak lze jednotlivé vrcholy grafu a tudíž i soubory topologicky seřadit. Soubory (resp. vrcholy v grafu) jsou seřazeny pomocí Tarjanova algoritmu [26]. Syntaktické stromy jednotlivých seřazených souborů jsou postupně zpracovány následujícím způsobem.

Během průchodu stromem se postupně alokují instance třídy *Element*, přičemž každá instance představuje určitý element v hierarchii *GUI*. Každému elementu je přiřazen jeho typ a seznam jeho potomků resp. elementů, které se v hierarchii nachází níže. Dále je každému elementu, u něž jsou definované změny některých z jejich atributů, přiřazena množina dvojic názvů atributů a jejich hodnot. Hodnotou v tomto případě nemusí být konstanta, ale může jí být i výraz v jazyce *C/C++*, jehož výpočtem se získá daná hodnota. Atributu namísto toho může být přiřazen kód celé funkce, v tomto případě by daná funkce vracela hodnotu atributu. Kód v jazyce *C/C++* je ve formě syntaktického stromu předán funkci *SourceToHandler* (jedná-li se o celou funkci) nebo *ExpressionToHandler* (jedná-li se pouze o výraz). Výstupem obou funkcí je instance třídy *SourceHandler*, která uchovává zdrojový kód

v podobě posloupnosti symbolů způsobem popsaným v sekci 4.2.1. V tomto případě není symbolem myšlen pouze znak či řetězec v syntaktickém stromu, jenž je výstupem parseru, ale i struktura obsahující pomocná data (např. informace o tom zda je symbol nahrazen něčím jiným).

Dále program vytvoří mapu identifikátorů, pro všechny elementy, u nichž byl nastaven atribut *id*. Tento atribut musí mít unikátní hodnotu, pokud je v souboru více než jeden element nastaven na stejnou hodnotu atributu *id*, dojde k chybě. Pro každý soubor se uchovává jedna mapa identifikátorů.

Každému elementu je podle jeho specifikovaného typu přiřazen ukazatel na instanci třídy *ClassContainer*. Pokud byly nějakému elementu přiřazeny nové atributy (pomocí klíčového slova *property*), musí být ve výstupním kódu daný element reprezentován speciální třídou, tudíž se vytvoří nová instance *ClassContainer*, do které jsou mimo výchozích atributů přidány i atributy nové. Všechny nové atributy musí být buď nějakého primitivního typu, nebo musí typem nějakého výchozího elementu, jinak dojde k chybě. Následně dojde k vygenerování hašovacích funkcí pomocí stejně, jako je tomu u vestavěných datových typů v sekci 5.3.2.

5.2.3 Zpracování kódu v jazyce C++

Vzhledem k tomu, že se lze v částech, které jsou napsané v jazyce C++, odkazovat na jiné elementy pomocí jejich identifikátorů *id*, je nutné analyzovat výrazy a nahradit ve výsledném kódu identifikátory referencemi na dané elementy. To je vyřešeno projitím všech symbolů, jež obsahují identifikátor a vyhledáním příslušného identifikátoru v mapě identifikátorů (pro právě zpracovávaný soubor). Je-li identifikátor nalezen, pak se k příslušnému symbolu zapíše, že má být ve výstupu nahrazen referencí na konkrétní identifikovaný element.

Následně se v kódu zpracují všechny operátory „.“ pro přístup k atributu, jehož pravým operandem je vždy jméno získávaného atributu. Tento operátor lze v obecném případě zřetězit několikrát za sebou. V případě, že operand nejvíce vlevo není identifikátorem nějakého elementu ani existujícím atributem, pak se může jednat o přístup ke globální proměnné, a tudíž se přestane dané zřetězení operátoru „.“ zpracovávat a na výstup se odešle ve stávající formě. V opačném případě se ověří, zda se nalevo od nějakého operátoru „.“ nachází reference na obecný element (což se netýká použití identifikátorů *id* popsaného výše). To probíhá tak, že se nejprve zjistí typ operandu nejvíce vlevo (to lze vždy učinit staticky) a postupně se postupuje směrem doprava. Jakmile se narazí na typ (operand), který je referencí na nějaký element, znamená to, že nelze staticky ověřit typy či existenci atributů napravo od něj, tudíž je nutné všechny operátory „.“ napravo od daného operandu nahradit voláním metody *Get*, která za běhu uživatelské aplikace ověří existenci atributu a případně jej získá. U všech operandů (atributů) nalevo od prvního nahrazeného operátoru lze staticky určit jejich existenci a typ. Pokud se v rámci těchto atributů narazí na neexistující atribut, pak aplikace vypíše chybovou hlášku a ukončí se.

5.2.4 Výstup

Na závěr jsou pro každý zpracovaný *CQML* soubor (resp. komponentu) vytvořeny dva výstupní soubory, jejichž názvy jsou zakončeny „.h“, „.c“.

Do prvního souboru („.h“) je zapsána definice příslušné komponenty, respektive struktury,

kteřá ji reprezentuje, tudíž se do těla dané struktury vypíše členské proměnné (reprezentující *GUI* elementy v komponentě). Pro každý nový typ definovaný ve zpracovávaném souboru (reprezentovaný instancí třídy *ClassContainer*) se také zapíše definice struktury, která jej reprezentuje, přičemž do těla struktury se jako členské proměnné vypíše atributy daného typu. Každý atribut je zde reprezentován nejen proměnnou pro uložení své hodnoty, ale také ukazatelem na funkci pro její přepočítání a ukazatelem na kontext dané funkce.

Druhý soubor („*.c“) slouží k zapsání definic vygenerovaných funkcí používaných v dané komponentě. Každý atribut má přidělenou funkci, která se stará o jeho update. Do těla takové funkce je zapsán kód výrazu nebo funkce, který byl zpracován a nyní je uložen v instanci třídy *SourceHandler* příslušící danému atributu.

Pro každou strukturu definovanou v příslušném hlavičkovém souboru je vygenerován konstruktor, který inicializuje instanci struktury, přiřadí každému atributu počáteční hodnotu, všem ukazatelům na funkce příslušnou funkci a ukazatelům na kontexty funkcí příslušný kontext. Také je vygenerována funkce, která alokuje paměť pro danou strukturu a zavolá příslušný konstruktor. Pro všechny nově definované typy je vygenerován i kód metod *Update* a *Get* stejným způsobem jako je tomu u vestavěných typů v sekci 5.3.3. Jakmile jsou takto zpracovány všechny vstupní soubory, vytvoří se poslední výstupní soubory „parser_output.h“ a „parser_output.c“. V souboru „parser_output.h“ jsou hlavičky hlavních funkcí pro obsluhu *GUI*. V souboru „parser_output.c“ jsou tyto funkce definovány. Pro každý nový datový typ se zde vypíše i kód pro inicializaci hašovacích funkcí, jež byly pro daný typ vygenerovány.

5.3 Preprocesor vestavěných typů

5.3.1 Inicializace

Na počátku programu jsou spuštěny funkce pro registraci vestavěných typů a podporovaných primitivních typů, volání těchto funkcí jsou vygenerovány prostřednictvím maker (viz. sekce 5.1).

Registrované typy jsou uloženy do dvou polí, přičemž jedno pole obsahuje seznam jmen primitivních datových typů, a druhé pole obsahuje struktury uchováující data o vestavěných typech. Každému vestavěnému typu jsou přiřazeny atributy, které jsou přístupné z *CQML* kódu, atributy, které jsou přístupné pouze z *C++* kódu, a také ukazatele na funkce, jež slouží k obsluze událostí. Jednotlivým atributům je přiřazen jejich typ, jméno a výchozí hodnota. Pro každý zaregistrovaný vestavěný typ je postupně vygenerována hašovací funkce viz. sekce 5.3.2. A následně se vygeneruje výstupní kód.

5.3.2 Generování hašovacích funkcí

Každý vestavěný datový typ, či vygenerovaný datový typ musí umožnit dynamicky přístup ke svým atributům. Pro každý datový typ *CQML* je potřeba vytvořit hašovací funkci, která vrací unikátní hodnotu pro každý jeho atribut. K tomu je využit algoritmus popsáný v sekci 4.3.2. Na vstupu tohoto algoritmu je seznam jmen všech atributů, kde každé jméno

představuje jeden klíč. Na počátku se zvolí N třikrát větší než je počet klíčů m , N udává počet vrcholů ve vygenerovaném grafu. Následně se zjistí délka nejdelšího klíče, která určí velikost tabulek T_1 a T_2 . Algoritmus pro vygenerování grafu pracuje v cyklu, kde při každém projití cyklu se pokusí vygenerovat graf následujícím způsobem. Nejprve se tabulky T_1 a T_2 zaplní náhodnými čísly v rozmezí $[0, N - 1]$. Následně se pro každý klíč funkcí používající tabulky T_1 a T_2 pomocí vzorce 4.2 vypočítají indexy koncových vrcholů hrany v grafu, která reprezentuje daný klíč.

Jakmile jsou takto vygenerovány všechny hrany ověří se acyklicita daného grafu. Pokud graf není acyklický spustí cyklus znovu, jinak se cyklus přeruší. Acyklicita grafu se kontroluje postupným procházením grafu do hloubky s označováním navštívených vrcholů při průchodu dolů a jejich odznačením při průchodu zpět, přičemž navštíví-li se při průchodu dolů již označený vrchol, pak graf není acyklický. V případě, že se v určitém počtu iterací tohoto cyklu nedosáhne nalezení bezkonfliktní funkce, pak se N zvýší o 1, což zvýší šanci na nalezení acyklického grafu, a pokračuje se od začátku.

Jakmile je nalezen acyklický graf provede se ohodnocení jednotlivých vrcholů grafu. Hodnoty všech vrcholů jsou nejprve inicializovány na hodnotu 0. Dále se postupně zpracovávají jednotlivé vrcholy, každý zpracovaný vrchol se označí. Při zpracování vrcholu se projdou všichni jeho sousedé a každému z nich je přiřazena hodnota hrany mezi jím a právě zpracovávaným vrcholem, která je snížena o hodnotu právě zpracovávaného vrcholu. Následně se daný soused zpracuje totožným způsobem. Jakmile je graf ohodnocen jsou jeho data vložena do struktury *PerfectHashData* (viz. ukázka kódu 5.1) pro uchování vygenerované hašovací funkce. Tato data obsahují tabulky T_1 a T_2 , seznam klíčů (resp. názvů atributů), ohodnocení vrcholů grafu g , hodnotu m udávající počet klíčů a hodnotu N udávající počet vrcholů grafu.

Ukázka kódu 5.1: Struktura pro uchování hašovací funkce.

```
struct PerfectHashData
{
    string * keys;
    int * T1;
    int * T2;
    int * g;
    int m;
    int n;
};
```

5.3.3 Generování výstupního kódu

V další části preprocesorové aplikace jsou vygenerovány zdrojové kódy pro použití v runtime knihovně *CQML*. Nejprve se pro každý vestavěný typ vygeneruje zdrojový kód konstruktoru. Ve kterém se každému z jeho atributů přiřadí jeho výchozí hodnota. Každý vestavěný prvek má atribut typu integer s názvem *classID*, do něž se ve vygenerovaném kódu přiřadí identifikátor třídy, v tomto případě je identifikátorem třídy číslo označující pořadí, v jakém byl daný vestavěný typ zaregistrován.

Pro každý vestavěný typ se vypíše zdrojový kód sloužící k inicializaci hašovacích funkcí po-

užívaných za běhu uvnitř knihovny. Jedná se o výpis dat vygenerovaných v sekci 5.3.2. Následně je zapsán kód metody *Get*, pro získání libovolného atributu daného elementu, která na vstupu přijímá řetězec a jejíž návratovou hodnotou je typ *VariantRef*. Na počátku této vygenerované metody se zavolá funkce *GetHash*, které je předána hodnota parametru *classID* a vstupní řetězec. Tato funkce vybere z pole funkcí pomocí hodnoty *classID* hašovací funkci, pomocí níž spočte hodnotu haš pro vstupní řetězec. Hodnota haš je předána konstrukturu *switch*. V konstrukturu *switch* je každé větvi *case* dána hodnota představující určitý atribut (spočtená předáním jména atributu příslušné vygenerované hašovací funkci), v takovém případě daná funkce vrátí referenci na daný atribut ve formě datového typu *VariantRef*. Pokud hodnota předaná konstrukturu *switch* nesouhlasí s hodnotou hašovací funkce pro žádný atribut, pak mocí větve *default* vyhodí program výjimku. V poslední části generování zdrojových kódů v rámci preprocesoru jsou vygenerovány zdrojové kódy metody *Update* pro každý vestavěný prvek. Metoda *Update* je vygenerována pouze pro vestavěné referenční typy, pro hodnotové typy tomu tak není. Pro každý atribut, který je buď primitivního typu, nebo se jedná o vestavěný referenční typ, je do kódu této metody vložen konstrukt *if*, který se táže, zda má daný atribut přiřazen ukazatel na funkci pro update daného atributu, která jeho hodnotu znovu přepočítá, pokud ano, pak se daná funkce zavolá. Pokud se jedná o atribut vestavěného hodnotového typu, pak se provede totéž, s tím rozdílem, že za *if* je vložen konstrukt *else*, v němž se nachází další konstrukty *if*, které se dotazují na atributy uvnitř daného atributu stejným způsobem. Tudíž v případě, že není danému atributu přiřazen ukazatel na funkci pro jeho update, pak se funkce dotáže, zda totéž platí pro jednotlivé podatributy daného atributu a případně (jsou-li přiřazeny) je zavolá.

5.4 Knihovna

Do kódu knihovny je zahrnut kód, který je výstupem z preprocesoru. Tento kód obsahuje především implementace konstruktorů a metod vestavěných typů. Také se zde nachází funkce *InitDefaultHashTabs*, která inicializuje hašovací funkce, které jsou nutné pro dynamický přístup k elementům.

5.4.1 Inicializace

Před započítím užívání knihovny je potřeba knihovnu nejprve inicializovat z uživatelské aplikace. Inicializace má několik fází. Nejprve je nutné zavolat funkci *_CQML_Init*, která zavoláním funkcí *MakeQueue* a *InitQueueThreadsafe* vytvoří a inicializuje frontu pro vstupní události.

V další fázi inicializace je potřeba knihovně předat instance tříd implementující rozhraní pro vykreslování a správu externích zdrojů. Což je učiněno zavoláním funkcí *SetDrawIFace*, pro vykreslování, a *SetResourceManager*, pro správu externích zdrojů. Na závěr inicializace se zavolá funkce *_CQML_Start*, která je vygenerována v překladači a provede následující úkony.

Pomocí funkce *SetInitHashTabs*, je uložen do proměnné ukazatel na funkci pro inicializaci hašovacích funkcí. Inicializace hašovacích tabulek je následně provedena, když funkce *CQMLI-*

nitHashes zavolá funkci *InitDefaultHashTabs* pro inicializaci hašovacích funkcí vestavěných typů a následně zavolá funkci uloženou v dané proměnné. Pomocí funkce *SetRoot* je knihovně předán kořenový prvek hierarchie. Na závěr inicializace se zavolá funkce *_CQML_Update()*, pro inicializaci atributů *GUI* na validní hodnoty.

5.4.2 Update

Funkce *_CQML_Update* zavolá nejprve funkci *PreUpdate*, která spustí zpracování fronty vstupních událostí (viz. 5.4.4). Následně zavolá metodu *Update* kořenového prvku a na závěr je zavolána funkce *PostUpdate*, která se postará o načtení externích zdrojů prostřednictvím rozhraní pro jejich správu (viz. 5.4.5).

Každý element v *GUI* hierarchii dědí ze základního typu *Element* ať již přímo či transitivity, přičemž překrývá jeho metodu *Update*. Metoda *Update* každého odvozeného elementu je vygenerována buď v rámci preprocesoru vestavěných typů (pro vestavěné typy), a je tak integrována přímo do zdrojového kódu knihovny, nebo v rámci překladače (pro typy definované pomocí *CQML*). Každá vygenerovaná metoda volá metodu *DefaultUpdate*, která může, ale nemusí, být překryta, přičemž ta slouží k implementaci další funkčnosti v rámci *Update* fáze, kterou vygenerovaný kód neposkytuje. Na závěr metody *Update* se zavolá metoda *Update* nadřazené třídy (resp. struktury), s výjimkou metody *Update* poskytované v rámci struktury *Element*.

5.4.3 Vykreslování

Během vykreslovací fáze se zavolá metoda kořenového elementu, která vykreslí daný element a vloží všechny jeho potomky na konec vykreslovací fronty. Následně se odebere přední element z fronty a zavolá se jeho vykreslovací metoda, která jej vykreslí a vloží jeho potomky na konec fronty. To pokračuje, dokud se nevykreslí všechny *GUI* elementy ve stromové struktuře a fronta není prázdná.

Veškerá funkčnost pro vykreslování na obrazovku je realizována prostřednictvím metod vykreslovacího rozhraní *DrawIFace*, jehož instance (resp. instance třídy z něj odvozené) byla knihovně předána během inicializace. Každý druh *CQML* elementu je představován třídou dědicící ze třídy *Element* a může překrýt jeho vykreslovací metodu. Následuje seznam *CQML* elementů a popis jejich vykreslovacích metod.

Element - nevykresluje nic, jen zavolá vykreslovací metody všech potomků.

Rectangle - vykreslí jednobarevný obdélník pomocí metody *DrawFilledRectangle*, v případě že má definovaný okraj nulové šířky. V opačném případě vykreslí obdélník s okrajem pomocí metody *DrawFiledBorderedRectangle*.

Image - vykreslí obrázek o daných rozměrech pomocí metody *DrawImage*

ScaledImage - vykreslí obrázek rozřezaný podle parametrů na devět částí (viz. sekce D.1.5), tak aby se při změně rozměrů obrázku některé části nedeformovali. Pro vykreslení se tak devětkrát metodu *DrawImageSegment*.

Text - vykreslí text daného písma a velikosti na zadaných souřadnicích pomocí metody *DrawText*.

5.4.4 Zpracování vstupu

Knihovna přijímá vstup ve formě událostí, které jsou vkládány do fronty pro zpracování. Fronta pro zpracování je realizována pomocí vláknových zámků tak, že pro případ použití ve vícevláknové aplikaci nehrozí, že dojde k problémům, způsobených tím, že více vláken přistupuje k této frontě najednou. Na počátku fáze *Update* se zavolá funkce *ProcessEvents*, která má za úkol zpracovat události ve vstupní frontě. Během zpracování jednotlivých událostí se každá událost vybere z fronty a následně se pošle kořenovému elementu ke zpracování. Události jsou realizovány strukturou *CQMLEvent*, která obsahuje identifikátor druhu události, a konstrukt *union*, v němž se nachází proměnné typu *CQMLMouseEvent* a *CQMLKeyboardEvent*. To která z těchto dvou proměnných se využije je určeno typem události. Struktura *CQMLMouseEvent* obsahuje data vstupní události vygenerované pomocí myši, konkrétně identifikátor události, hodnotu případného použitého tlačítka, souřadnice myši a případně souřadnice pohybu myši. *CQMLKeyboardEvent* obsahuje identifikátor události vygenerované klávesnicí a hodnotu stisknuté klávesy. Následující seznam vypisuje všechny podporované vstupní události.

MousePressed - stisknutí tlačítka myši

MouseReleased - puštění tlačítka myši

MouseMove - pohyb tlačítka myši

MouseScrolled - pohyb tlačítka myši

KeyPressed - stisknutí klávesy

KeyReleased - puštění klávesy

Události se zpracovávají totožným způsobem. Element, kterému je předána, ji nejdříve předá svým potomkům ke zpracování a následně se ji pokusí zpracovat sám. Samotné zpracování události však závisí na druhu elementu, neboť ty mohou metody pro zpracování překrývat. Události generované klávesnicí se zpracovávají tak, že v případě, že je pro danou událost přiřazena elementu nějaká funkce pro její zpracování (ukazatel na ni není nulový), pak se prostřednictvím ukazatele zavolá.

Události generované myší jsou zpracovávány pouze v elementech typu *MouseArea*. U události pro stisk tlačítka myši se pomocí souřadnic a rozměrů elementu a souřadnic myši ověří, zda je ukazatel myši uvnitř daného elementu. Pokud ano, vloží se element do pole elementů *pressedElement*. Pokud je elementu přiřazena funkce pro obsluhu této události, pak se zavolá. V případě události pro puštění tlačítka myši se pro všechny elementy v poli *pressedElement* zavolá funkce pro obsluhu této události (byla-li přiřazena), zároveň se u daných elementů zkontroluje, zda se v nich nachází ukazatel myši, pokud tomu tak je, zavolá se funkce pro obsluhu události kliknutí.

U události pro pohyb myši se zavolá funkce pro obsluhu, pokud však myš pohybem právě najela na nebo vyjela z oblasti elementu, zavolá se i funkce pro obsluhu jedné z těchto událostí. Při otočení tlačítka myši se u dané události zkontroluje, zda se stala uvnitř elementu, je-li tomu tak zavolá se funkce pro jeho obsluhu.

Událostí, které lze přiřadit v rámci *CQML* je tak více než těch, které lze knihovně předat. Následuje seznam vstupních událostí, které lze obsloužit v rámci jazyka *CQML*.

MousePressed - stisknutí tlačítka myši (*MouseArea*)

MouseReleased - puštění tlačítka myši (*MouseArea*)

MouseClicked - kliknutí tlačítkem myši (*MouseArea*)

MouseMove - pohyb tlačítka myši (*MouseArea*)

MouseEntered - pohyb tlačítka myši dovnitř elementu (*MouseArea*)

MouseExited - pohyb tlačítka myši vně elementu (*MouseArea*)

MouseScrolled - pohyb tlačítka myši (*MouseArea*)

KeyPressed - stisknutí klávesy (všechny elementy)

KeyReleased - puštění klávesy (všechny elementy)

5.4.5 Správa externích zdrojů

Ve fázi *Update* jednotlivých elementů se mohly změnit adresy nebo parametry některých elementů používajících externí zdroje, tudíž může být nutné některé zdroje načíst znovu. Proto se u všech elementů, u kterých k tomuto mohlo dojít, uloží do fronty požadavek pro načtení daného externího zdroje. Kokrnětné *CQML* elementy *Text* a *Image* (stejně tak i elementy z nich odvozené), v rámci metody *DefaultUpdate*, předávají tyto požadavky pomocí funkcí *TryLoadFont* a *TryLoadImage*. Knihovna momentálně podporuje pouze dva druhy externích zdrojů a tím jsou písmo a obrázky. V případě obrázku je takový požadavek definován řetězcem, kterým je cesta k souboru obrázku. V případě písma se jedná o název písma a jeho velikost.

Na konci fáze *Update* se zpracují všechny požadavky ve frontě. Načtená data jsou v rámci kódu knihovny uchovávána v mapě, která data každého záznamu uchovává ve formě obecného ukazatele typu *void** ukazující na data načtená data externího zdroje. Jako klíč pro záznam se používá textový řetězec, který je odvozen z požadavku pro načtení. Pro obrázek je jím opět cesta k souboru a pro písmo je jím jméno písma, za které je do řetězce přidána jeho velikost. V případě, že již byl načten totožný požadavek v minulosti, pak se nenačítá znovu. To je rozpoznáno tak, že se v mapě již nachází záznam pod daným klíčem. V opačném případě se prostřednictvím objektu pro obsluhu externích zdrojů, jenž byl ve fázi inicializace knihovně předán, zavolá funkce pro načtení daného externího zdroje, přičemž data následně jsou pod příslušným klíčem uložena do mapy.

Kdykoli pak knihovna potřebuje předat uživatelské aplikaci data načtená z externího zdroje, učiní tak předáním *void** ukazatele uloženého v mapě.

Objekt pro obsluhu externí zdrojů, který je knihovně předán v inicializační fázi, musí dědit z abstraktní třídy *ResourceManagerIFace* a implementovat její metody. Abstraktní třída *ResourceManagerIFace* vlastní metodu pro načtení písma *LoadFont* a metodu *LoadImage* pro načtení dat obrázku.

5.4.6 Datový typ *Variant* a *VariantRef*

Variant je datový typ umožňující manipulaci s hodnotou libovolného datového typu. Účelem typu *VariantRef* je umožnit obdobnou manipulaci i s referencí na hodnotu libovolného typu.

Součástí typu *Variant* je identifikátor typu hodnoty, jaká je uvnitř dané instance uchovávána. Samotnou hodnotu uchovává pomocí konstrukturu *union*, jež obsahuje jeden atribut pro každý datový typ, který je možné v typu *Variant* uchovávat. Jelikož v rámci konstrukturu *union* nelze uchovávat datové typy s netriviálním konstruktorem, jsou takové typy uchovávány pomocí ukazatele, přičemž jejich hodnota je do typu *Variant* uložena zkopírováním na naalokované místo na haldě a uložení ukazatele na dané místo v paměti pomocí metody *Copy*. Mezi takové typy patří standardní *C++* řetězec a také libovolný datový typ *CQML*. Jelikož všechny *CQML* datové typy dědí ze třídy *CQMLObject*, je ukazatel na takové typy typu ukazatele na obecný *CQMLObject*.

Typ *Variant* obsahuje metodu *As*, která je realizována pomocí čtyř šablon. Tato metoda vrací hodnotu stejného typu jako je typ, který byl předán jako parametr šablony. Správná šablona je vybrána během kompilace podle jejího parametru. První šablona platí pro číselné typy, metoda *As* pak v případě, že daná instance typu *Variant* uchovává číselná data, vrátí číselnou hodnotu a v opačném případě vyhodí výjimku. Druhá šablona provádí totéž pro řetězcová data. Třetí šablona činí totéž pro ukazatele na objekt typu *CQMLObject*. Čtvrtá šablona je pro ostatní datové typy a ta vždy vyhodí výjimku. Typ *Variant* implementuje několik přetížených konstruktorů, každý z nich inicializuje identifikátor typu hodnoty a následně uloží hodnotu, která mu byla v parametrech předána.

Typ *Variant* přetěžuje základní aritmetické operátory, všechny operátory nejprve zjistí zda typ levého operandu podporuje danou operaci a následně se pokusí pravý operand přetypovat pomocí metody *As* na stejný typ, jakým je levý operand. V případě úspěchu se daná operace provede. Pro většinu nečíselných typů nejsou podporovány aritmetické operace. Výjimkou je operace sčítání, která je implementována i pro řetězce. Pokud některý typ nepodporuje určitou aritmetickou operaci, pak při zavolání daného operátoru dojde k vyhození výjimky.

Typ *Variant* implementuje metodu *Get*, která v případě, že je v něm uchovávána hodnota typu *CQMLObject*, zavolá tutéž metodu na daný *CQMLObject* a vrátí vrácenou hodnotu. Pro hodnoty ostatních typů vyhodí výjimku. Součástí datového typu *VariantRef* je identifikátor typu hodnoty, na kterou ukazuje daná instance. Samotný ukazatel se uchovává pomocí konstrukturu *union*, jež obsahuje jeden atribut pro každý typ ukazatele. Narozdíl od typu *Variant* je zde pro *CQML* typy rozlišováno zda se jedná o datové či hodnotové typy. Také je zde uchováván ukazatel na ukazatel funkce pro přepočítání atributu.

VariantRef také implementuje metodu *As*, avšak narozdíl od typu *Variant*, je realizována pomocí jedné šablony, která přijímá návratový typ jako parametr. Daná metoda pouze přetypuje daný objekt na typ *Variant* a zavolá jeho metodu *As*.

VariantRef přetěžuje podobně jako typ *Variant* operátory. V případě operátorů pro přiřazení, však dojde ke změně hodnoty, na kterou typ *Variant* ukazoval. Jelikož se typ *VariantRef* používá pro přímý přístup k atributu nějakého elementu, dojde při změně hodnoty k vynulování ukazatele na funkci, jež slouží pro přepočítávání hodnoty daného atributu. V případě ostatních operátorů se levý operand přetypuje na typ *Variant* a provede se ope-

race, která byla implementována v rámci přetíženého operátoru ve třídě *Variant*. Metoda *Get* je zde realizována totožným způsobem jako u typu *Variant*.

Typy *Variant* a *VariantRef* podporují uložení následujících datových typů:

- číselné typy - *int*, *long*, *long long* (*signed* i *unsigned*), *float*, *double*, *long double*
- řetězec - *string*, *char**
- libovolný datový *CQML* typ - *CQMLObject**
- obecný ukazatel - *void**

5.4.7 Datové typy *CQML*

Všechny datové typy přístupné v rámci jazyka *CQML*, které nejsou primitivními typy, dědí z datového typu *CQMLObject*. Třída *CQMLObject* obsahuje člen *classID*, který v sobě uchovává identifikátor třídy. Dále obsahuje virtuální metodu *Copy* pro zkopírování aktuální instance, a také virtuální metodu *Get*, která slouží pro přístup k libovolnému atributu. Metoda *Get* jako parametr přijímá název atributu, který by měl být členem dané struktury. Tato metoda podle hodnoty vypočtené pomocí hašovací funkce z názvu vybere příslušný atribut, přičemž hašovací funkce je vybrána podle identifikátoru *classID*. Tato metoda vrací hodnotu typu *VariantRef*, do kterého se uloží adresa daného atributu a také adresa ukazatele na funkci pro jeho zpracování.

5.5 Uživatelská aplikace

Pro účely demonstrace funkčnosti knihovny jím generovaného *GUI* bylo nutné vytvořit testovací aplikaci, která by umožnila použití *CQML* knihovny. Pro účely vstupu a vykreslování je použita knihovna *SDL2*. Pro použití knihovny je nutno implementovat třídy, jež implementují rozhraní pro vykreslování a správu externích zdrojů. Pro účely vykreslování je vytvořena třída *SDL_Drawer*, která dědí z abstraktní třídy *DrawIFace*. Pro správu externích zdrojů slouží třída *SDL_Manager* dědicí z abstraktní třídy *ResourceManagerIFace*. Na počátku programu se inicializuje knihovna *SDL*. Následně se spustí inicializace *CQML* knihovny metodou *__CQML_Init*. Poté se vytvoří instance třídy *SDL_Drawer* a *SDL_Manager*, jež se předají runtime *CQML* knihovně. A na závěr se zavoláním metody *__CQML_Start* ukončí inicializace. Pomocí metody *GetCQMLWindow* knihovny *CQML* získají rozměry kořenového elementu v *GUI* hierarchii. Následně se vytvoří okno a spustí se hlavní programová smyčka. Hlavní programová smyčka se skládá ze tří fází – zpracování vstupu, update, vykreslování. Knihovna *SDL* zpracovává vstup jako frontu vstupních událostí. Proto se ve fázi pro zpracování vstupu vyjmou z fronty všechny vstupní události a postupně se přeformují do formátu přijímaného knihovnou *CQML* a pomocí metody *PushEvent* se knihovně *CQML* odešlou. Ve fázi update se spustí funkce *__CQML_Update* čímž se aktualizují hodnoty elementů uvnitř *GUI* hierarchie. Ve fázi vykreslení se na počátku vyčistí obrazovka a následně se vykreslí *GUI* hierarchie pomocí funkce *__CQML_Draw*. Hlavní programová smyčka běží, dokud uživatel nestiskne křížek pro zavření okna, který vyvolá vstupní událost *SDL_QUIT*,

která smyčku ukončí, což vede k uvolnění zdrojů a ukončení programu.

SDL_Manager implementuje metodu pro načtení písma *LoadFont*, která pomocí funkce *TTF_OpenFont* knihovny *SDL_TTF* načte požadovaný font a vrátí na něj ukazatel. Také implementuje metodu pro načtení obrázků pomocí knihovny *SDL_image*, která pomocí funkce *IMG_Load* načte data obrázku. Načtená data obrázku jsou uložena spolu s rozměry obrázku do struktury *ImageData*, která je následně metodou vrácena. *SDL_Drawer* implementuje vykreslovací metody abstraktní třídy *DrawIface*. Tyto metody jsou realizovány pomocí vykreslovacích funkcí poskytovaných knihovnou *SDL*.

Kapitola 6

Výsledky

6.1 Srovnání tvorby jednoduchého rozhraní pomocí *CQML* a *QML*

Srovnání funkčnosti je ilustrováno na jednoduché *GUI* aplikaci. Aplikace má několik částí. Obsahuje hlavní nabídku s několika tlačítky, kalkulačku a jednoduchou hru. V nabídce uživatel může kliknutím na příslušné tlačítko vybrat část aplikace, která se následně spustí.

Jednotlivé části aplikace jsou realizovány pomocí samostatných komponent. Základní funkčnost *QML* ani *CQML* nepodporuje tvorbu více oken, tudíž není možné rozdělit komponenty do samostatných oken. Z tohoto důvodu je přepínání mezi částmi aplikace řešeno jejich skrytím a odkrytím prostřednictvím nastavení hodnoty parametru pro viditelnost.

6.1.1 Hlavní nabídka

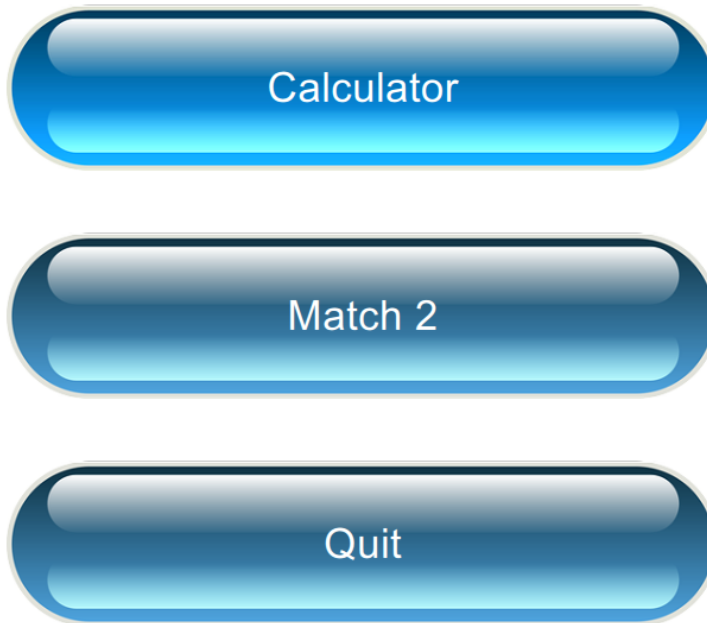
Jednoduchá nabídka, jenž obsahuje tři tlačítka. První tlačítko zobrazí rozhraní kalkulačky, druhé zobrazí rozhraní hry a třetí ukončí program.

QML Realizace

Tlačítko v nabídce je realizováno jako samostatná komponenta *TextButton*. Kořenovým elementem v této komponentě je *MouseArea*. Tlačítku jsou přiřazeny metody pro zpracování událostí, pro najetí ukazatelem myši na plochu vymezenou tlačítkem ukazatele myši a také na její opuštění, dále jsou přiřazeny metody pro stisknutí a puštění tlačítka myši na daném elementu. Tyto metody slouží pro přepínání obrázku tlačítka. Kořenový element má dva potomky - jeden typu *Text* a druhý typu *Image*. Element *Text* zde slouží pro zobrazení textu na tlačítko a *image* slouží k zobrazení obrázku tlačítka. Cesty k obrázkům jsou uloženy do přidáných řetězcových atributů a jsou celkem tři – základní obrázek, obrázek po najetí myši na tlačítko a obrázek po jeho stisknutí.

Při manipulaci s komponentou zvenčí je možné přistupovat pouze k atributům, které vlastní kořenový prvek komponenty. Aby tak bylo možné vytvářet pomocí této komponenty tlačítka s různým popiskem, je do kořenového prvku pomocí příkazu *alias* přidán odkaz na atribut

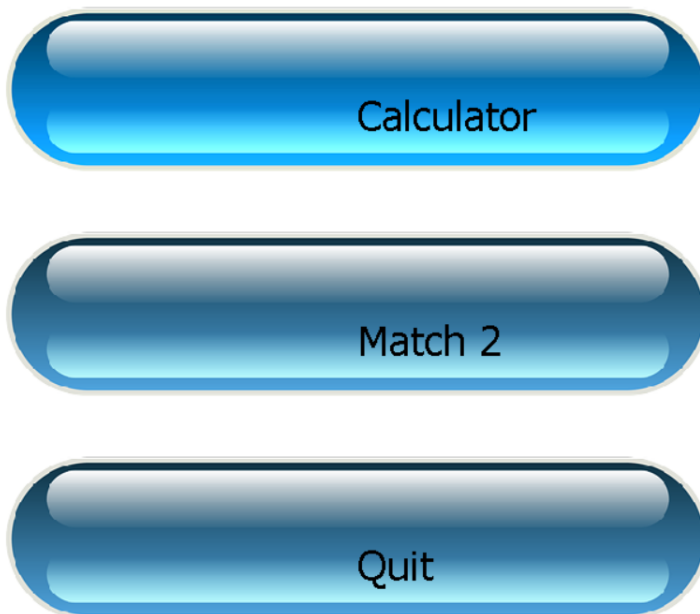
text elementu *Text* pro zobrazení textu. Nabídka se nachází na vrcholu celé *GUI* hierarchie, tudíž jejím kořenovým elementem je element typu *Window*. Ten obsahuje dvě hlavní komponenty (kalkulačku a hru), jejichž viditelnost je vypnuta, dále obsahuje tři tlačítka vytvořená z komponenty *TextButton*. Každé tlačítko má přiřazeno funkci pro obsluhu události kliknutí. V případě prvních dvou tlačítek tato funkce vypne viditelnost všech tlačítek a zapne viditelnost jedné z hlavních komponent. Třetí tlačítko v případě kliknutí vyšle signál k ukončení aplikace. Na obrázku 6.1 je ukázáno vygenerované *GUI*.



Obrázek 6.1: Obrázek jednoduché nabídky realizované v jazyce *QML*.

CQML Realizace

Realizace nabídky v rámci *CQML* je téměř totožná s realizací pomocí *QML*, zanedbají-li se rozdíly v sintaxi obou jazyků. Stejně jako je tomu u *QML*, *CQML* umožňuje z vnější komponenty přímo manipulovat pouze s atributy kořenového prvku dané komponenty. *CQML* na rozdíl od *QML* ale neumožňuje vytvoření odkazu na atribut elementu. Tudíž k němu není možné přímo přistupovat. v tomto případě je to však vyřešeno tak, že kořenovému prvku je přiřazen identifikátor *thisButton* a zároveň je do něj přidán nový atribut jménem *buttonLabel* typu řetězec. V příkazu pro určení hodnoty atributu text elementu *Text* je následně výrazem *thisButton.buttonLabel* určeno, že se má použít hodnota *buttonLabel* kořenového elementu. Tímto způsobem tak lze nepřímě měnit hodnotu zobrazovaného textu z vnější komponenty. Na obrázku 6.2 se nachází rozhraní vygenerované z *CQML* kódu.



Obrázek 6.2: Obrázek jednoduché nabídky realizované v jazyce CQML.

6.1.2 Kalkulačka

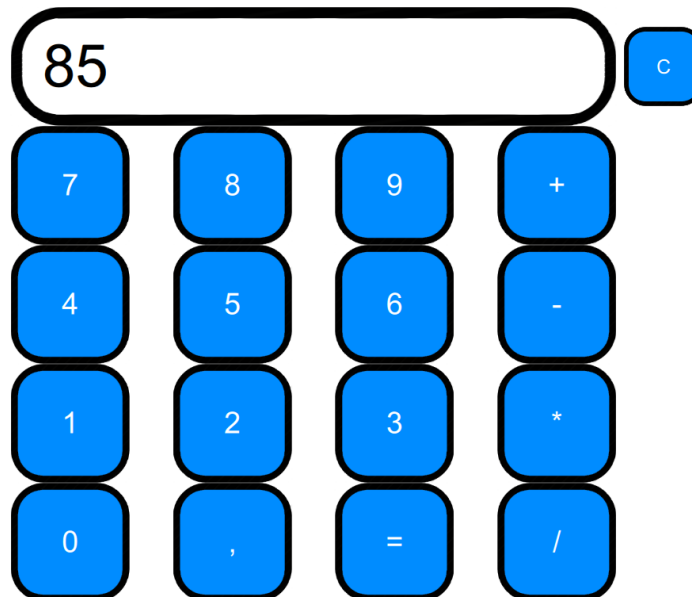
Cílem této části je vytvořit jednoduchou kalkulačku. Kalkulačka umožní uživateli zadávat čísla a provádět základní aritmetické operace. Kalkulačka obsahuje zobrazovací řádek pro zobrazení zadávaného čísla, který ale také slouží pro zobrazení výsledku operace. Dále aplikace obsahuje deset tlačítek pro zadání jednotlivých číslic, tlačítko pro vynulování, tlačítko pro spuštění zadávání desetinné části a pět tlačítek reprezentující aritmetické operátory plus, mínus, dělení, krát a rovná se. V této části je využita externí funkčnost a to tak, že každé tlačítko volá příslušnou funkci z externího souboru.

QML Realizace

Řádek pro výstup je realizován pomocí elementu *BorderImage*, jež zobrazí jako obrázek okraje řádku, na který se vypisuje výstup kalkulačky. Tento element v sobě obsahuje jednoho potomka typu *Text* označeného identifikátorem *thisText*, který slouží k zobrazení textového výstupu kalkulačky. Všechna tlačítka jsou realizována pomocí komponenty *CalculatorButton*. Komponenta *CalculatorButton* se skládá z pouze jednoho elementu, kterým je komponenta *TextButton*, které je nastavena šířka a výška a odlišné cesty ke zdrojovým obrázkům, než má nastavené výchozí *TextButton*.

Každé tlačítko má přiřazenou funkci pro obsluhu událostí, která volá funkce externí logiky importované ze souboru „CalcLogic.js“. Každá externě volaná funkce v tomto případě představuje operaci kalkulačky, kterou má provést po stisknutí tlačítka, přičemž vrací hodnotu,

která se má zobrazit na výstupním řádku kalkulačky, tato hodnota se přiřadí elementu označenému *thisText*, který jí zobrazí. Obrázek 6.3 ukazuje výsledné rozhraní.



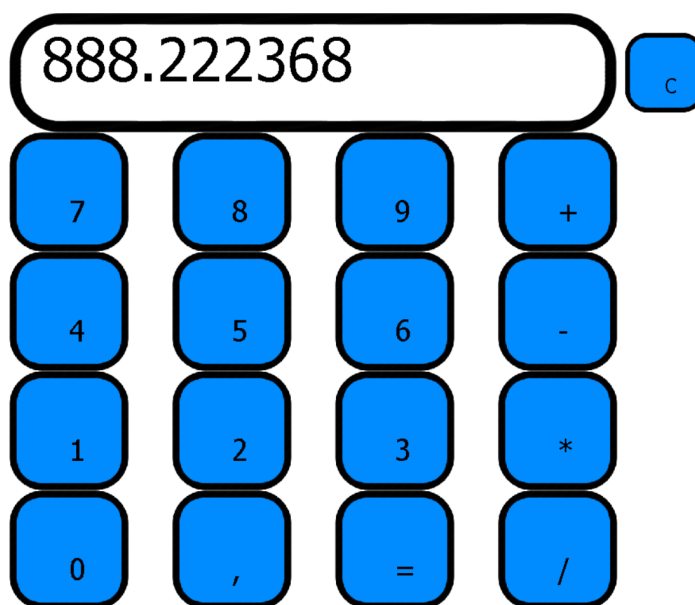
Obrázek 6.3: Obrázek kalkulačky realizované pomocí deklarace v jazyce QML.

CQML Realizace

CQML realizace je v mnoha směrech podobná, avšak s několika rozdíly. Veškerá logika kalkulačky je implementována v rámci uživatelské aplikace v jazyce C++. Pomocí příkazu `include` je do CQML zahrnut hlavičkový soubor „`calc_logic.h`“. V tomto souboru je definovaná struktura *CalculatorLogic*, která slouží pro obsluhu aplikační logiky kalkulačky a je zde také deklarovaná proměnná typu této struktury s názvem *CalcLogic*. V rámci funkcí uvnitř CQML kódu pak lze k dané proměnné přistupovat a volat její metody. Jednotlivá tlačítka mají přiřazena funkci pro obsluhu události kliknutí, z nichž každá volá příslušnou metodu struktury *CalculatorLogic*. Obrázek 6.4 ukazuje výsledné rozhraní realizované v CQML.

6.1.3 Jednoduchá hra

Pro ilustraci možností používání dynamických referencí na atributy poslouží hra s velmi jednoduchými pravidly. Na herní ploše je rozmístěno několik různě barevných hracích kamenů, pro účely aplikace jsou ve formě tlačítek. Úkolem hráče je označit dvě stejně barevná tlačítka tak, že na ně klikne pomocí myši. Pokud hráč označí tlačítka s různou barvou, pak vybírá znovu. Při správném označení dvou tlačítek stejné barvy jsou obě tlačítka odstraněna a hráči jsou přičteny body k jeho celkovému skóre. Hráčovo skóre se zobrazuje v zobrazovacím řádku.



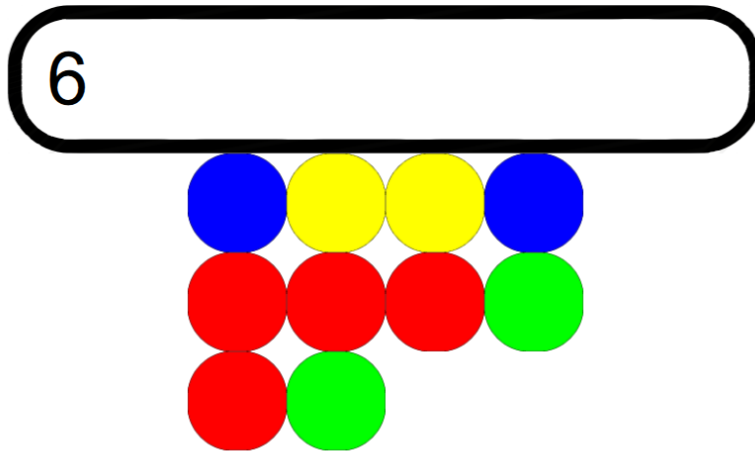
Obrázek 6.4: Obrázek kalkulačky realizované pomocí deklarace v jazyce CQML.

QML Realizace

Jednotlivá tlačítka představující hrací kameny jsou realizována pomocí komponent *GameCircle*, který je odvozen z komponenty *NormalButton*, podobně jako *CalculatorButton* je odvozen z *TextButton*. *NormalButton* je komponenta s totožnou funkčností jako *TextButton*, jen neobsahuje funkčnost pro zobrazení popisku tlačítka. Hrací plocha je reprezentována elementem typu *Rectangle*, který je rozšířen o několik atributů. Hrací plocha obsahuje element *Text* pro zobrazení vítězné zprávy, šestnáct elementů typu *GameCircle*, které jsou umístěny v elementu typu *Grid*, který je všechny rozmístí do zadaného počtu sloupců a řádků. Element představující hrací plochu je rozšířen o několik celočíselných atributů pro uchování hodnoty hráčova skóre, aktuálního stavu a hodnoty předchozího vybraného hracího kamene, další dva přidávané atributy slouží k uchování reference na předchozí hrací kámen a na element pro zobrazení vítězné zprávy.

Kořenovým elementem komponenty *GameCircle* je element typu *NormalButton*, který je rozšířen o dva atributy – *value* a *matcher*. Atribut *value* udává hodnotu herního kamene a *matcher* uchovává referenci na element představující hrací plochu. Obrázky, které určují vzhled tlačítka, jsou vybírány podle hodnoty herního kamene. Dále je kořenovému elementu této komponenty přiřazena funkce pro obsluhu události kliknutí. Daná funkce prostřednictvím reference *matcher* přistupuje k atributům hrací plochy, pomocí nichž zjistí, zda je vybrán nějaký hrací kámen. Není-li tomu tak, je následně v hrací ploše uložena reference na hrací kámen, na nějž bylo kliknuto. Byl-li již nějaký hrací kámen vybrán, je zkontrolováno, zda má stejnou hodnotu s kamenem, na který bylo právě kliknuto. Pokud mají stejnou hodnotu, je přičtena hodnota kamene k hráčovu skóre a oba hrací kameny jsou smazány. V opačném případě je smazána reference na předchozí vybraný kámen a hráč musí vybrat oba kameny znovu. Jsou-li všechny kameny smazány, což se pozná kontrolou ma-

ximální hodnoty skóre, je prostřednictvím reference matcher na hrací plochu přistoupeno k elementu pro zobrazení vítězné zprávy a zapne se její viditelnost. Obrázek 6.5 ilustruje vzhled rozhraní po odebrání několika herních kamenů.



Obrázek 6.5: Obrázek jednoduché hry realizované pomocí deklarace v jazyce *QML*.

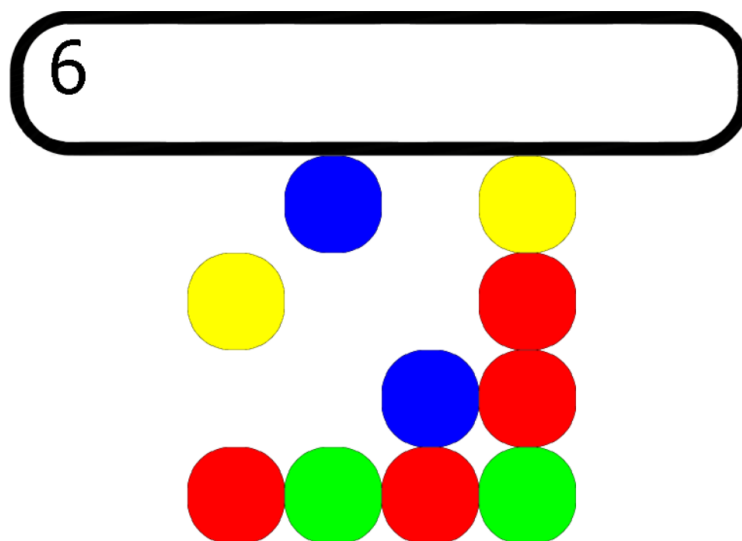
CQML Realizace

Na rozdíl od *QML* aktuální verze *CQML* neumožňuje užívání elementů pro automatické rozvržení, jakými jsou například *Grid*, *Row*, *Column*. Z tohoto důvodu je nutné v *CQML* souboru ručně zapsat souřadnice jednotlivých hracích kamenů resp. tlačítek. *CQML* neumožňuje dynamickou tvorbu a mazání elementů, proto nejsou při správném výběru hrací kameny smazány a pouze se skryjí. Na obrázku 6.6 je vidět vygenerované rozhraní po odebrání několika herních kamenů, oproti výsledku zobrazeném v *QML* realizaci se po odebrání kamene ostatní kameny neseřadí (právě toto v *QML* obstarává element *Grid*).

6.2 Porovnání *QML* a *CQML*

Základní syntaxe a funkčnost jazyků *CQML* a *QML* je velmi podobná. Hlavním rozdílem je, že zatímco jazyk *QML* je interpretován (ať už ve své textové podobě či po překladu do byte-kódu) v rámci uživatelské aplikace, tak jazyk *CQML* slouží pro vygenerování zdrojových kódů v jazyce *C++*, které jsou přímo do uživatelské aplikace vloženy a jsou překládány spolu s ní.

Knihovna, která za běhu obsluhuje hierarchii vygenerovanou z jazyka *CQML* umožňuje uživateli použít vlastní funkce pro vykreslování a správu zdrojů (obrázků a písem), ovšem na rozdíl od knihovny *QML* neposkytuje vlastní funkčnost pro tento účel. Oba jazyky mimo jiné umožňují dynamicky provázat atributy elementů, přidávání nových atributů, přiřazování funkcí pro obsluhu vstupních událostí a tvorbu nových komponent. Oba jazyky podporují



Obrázek 6.6: Obrázek jednoduché hry realizované pomocí deklarace v jazyce *CQML*.

zapouzdření funkčnosti *GUI* do samostatných souborů (komponent), které mohou být následně využívány, jako samostatné elementy v rámci jiné *GUI* hierarchie.

Nevýhodou *CQML* oproti *QML* je výrazně menší počet podporovaných grafických elementů. *CQML* momentálně poskytuje jen několik základních elementů, zatímco *QML* poskytuje několik stovek elementů s odlišnou funkčností. Dalším rozdílem mezi oběma jazyky, který lze považovat za nevýhodu jazyka *CQML* je, že ačkoli knihovna pro *CQML* umožňuje použití externího zobrazovacího rozhraní implementovaného v rámci uživatelské aplikace, narozdíl od *QML* neposkytuje žádné vlastní vykreslovací funkce.

Za výhodu knihovny *CQML* lze považovat, že se nachází pod volnější open-source licencí než prostředí *Qt*, jehož součástí je knihovna *QML*. Zatímco *CQML* je dostupné pod licencí *FreeBSD*, *Qt* spadá pod licenci *LGPL* [27]. Zdrojové kódy knihovny *QML* tak nelze například použít v rámci komerčních aplikací, které nejsou open-source, jinak než ve formě dynamické knihovny. Následuje popis hlavních rozdílů týkajících se možnosti realizace funkčnosti *GUI*.

6.2.1 Tvorba více oken

Oba jazyky nepodporují tvorbu více oken uvnitř *GUI* Hierarchie, jazyk *QML* však umožňuje v rámci uživatelské aplikace v jazyce *C++* spustit několik samostatných instancí *GUI*, mezi kterými je možno přepínat, to pro knihovnu *CQML* není možné.

6.2.2 Reference na atributy

Jazyk *QML* umožňuje v rámci elementu pomocí klíčového slova *alias* vytvářet reference na atributy jiných elementů. V jazyce *CQML* toto není možné, avšak tento problém se dá v některých případech obejít, jak je ilustrováno v 6.1.1.

6.2.3 Dynamická změna hierarchie elementů

QML jazyk umožňuje dynamicky měnit hierarchii elementů pomocí mazání a přidávání elementů či komponent za běhu programu pomocí vyhrazených funkcí a metod. *CQML* jazyk aktuálně takovou možnost nemá. *QML* knihovna dále umožňuje za běhu načítat nové soubory jazyka *QML* a používat uvnitř definované GUI. Knihovna pro jazyk *CQML* toto už z principu neumožňuje, jelikož se jazyk překládá do nativního kódu *C++* a tudíž musí být všechny používané *CQML* soubory zpracovány a přeloženy ještě před spuštěním uživatelské aplikace.

6.2.4 Rozšiřitelnost

Jazyk *QML* může být rozšířen o nové elementy pomocí jejich implementace v uživatelské *C++* aplikaci. Nový element je představován třídou, která dědí z třídy *QObject*, a její vlastnosti, které mají být přístupné z jazyka *QML*, musí být deklarovány pomocí speciálních maker. Jazyk *CQML* může být také rozšířen o nové elementy. Pro vytvoření nového elementu je potřeba definovat název, atributy a metody daného elementu pomocí speciálních maker v rámci hlavičkového souboru, který je pro to určen (viz. sekce 5.1). Po definici nových elementů pomocí těchto maker musí být spuštěn preprocesor vestavěných datových typů, který vygeneruje kód, který se zahrne do runtime knihovny. V rámci knihovny je nutno implementovat funkčnost nových elementů a následně knihovnu zkompileovat. Teprve poté je možno nový vestavěný element použít.

Kapitola 7

Závěr

Během této práce byl pečlivě prozkoumán jazyk *QML* pro deklarativní definici *GUI*. Na jeho základě byl navržen jazyk *CQML* s podobnou syntaxí a byla vytvořena jeho gramatika. Jazyk *CQML* může být rozšířen o další grafické elementy, které mohou být definovány v samostatných *CQML* souborech, takové elementy musí vycházet z již existujících. V základu však *CQML* neposkytuje takové množství grafických elementů jako jazyk *QML*.

Podářilo se vytvořit překladač, který vygeneruje zdrojový kód pro *GUI* definované pomocí jazyka *CQML*. Vygenerovaný kód je v jazyce *C++* a lze jej přímo zahrnout do kódu uživatelské aplikace, která jej zobrazí.

Pro správu *GUI* hierarchie v rámci uživatelské aplikace byla vytvořena runtime knihovna, která tuto funkčnost obstará. Knihovna podporuje vyměnitelné externí rozhraní pro vstup a výstup, který musí být poskytnut uživatelskou aplikací. Komunikace s uživatelskou aplikací je prováděna pomocí rozhraní pro vstup a výstup. Rozšíření základní funkčnosti knihovny, které nelze realizovat tvorbou nových komponent v rámci *CQML* souborů, vyžaduje zásah do zdrojových kódů knihovny. Nové typy elementů se deklarují v souboru „struct_definition_macros.h“ pomocí speciálních maker, který je sdílen mezi všemi částmi *CQML* aplikace s tím, že nová funkčnost přidaných typů elementů musí být implementována pouze v kódu runtime knihovny. Kód pro samotnou integraci nového elementu do jazyka *CQML* je generován automaticky. Je však stále nutné všechny části aplikace znovu přeložit.

Pro testovací účely byla naprogramována uživatelská aplikace, která využívá runtime knihovnu *CQML* a implementuje externí rozhraní pro vstup a výstup pomocí knihovny *SDL2*. Dále bylo vytvořeno uživatelské rozhraní v jazyce *CQML* a *QML*, na kterém byla demonstrována funkčnost tvorby *GUI* v jazyce *CQML* a také srovnání definice *GUI* v obou jazycích. Knihovna *CQML* je poskytována jako open-source pod *FreeBSD* licenci. Knihovnu je možné nadále rozvíjet a rozšiřovat její funkčnost, včetně přidávání dalších grafických elementů.

Literatura

- [1] Douglas C. Engelbart. AUGMENTING HUMAN INTELLECT: a Conceptual Framework. Menlo Park: Stanford Research Institute. Říjen 1962. AFOSR-3233
- [2] Douglas C. Engelbart, William K. English. A Research Center for Augmenting Human Intellect. AFIPS Conference Proceedings of the 1968 Fall Joint Computer Conference. Prosinec 1968, vol. 33, s. 395-410.
- [3] Xerox Alto. Fotografie. (cit. 8. květen 2015) Dostupné z <http://toastytech.com/guis/alto.html>
- [4] Jeremy Reimer. A History of the GUI. 2005. (cit. 8. květen 2015) Dostupné z <http://www.cdpa.co.uk/UoP/Found/Downloads/reading6.pdf>
- [5] Xerox Parc History. (cit. 8. květen 2015) Dostupné z <http://www.parc.com/about/>
- [6] Apple Macintosh. Fotografie. (cit. 8. květen 2015) Dostupné z <http://history-computer.com/ModernComputer/Personal/Macintosh.html>
- [7] Microsoft Windows. Fotografie. (cit. 8. květen 2015) Dostupné z <http://history-computer.com/ModernComputer/Software/Windows.html>
- [8] Workbench Nostalgia: The history of the AmigaOS Graphic User Interface (GUI). Fotografie. (cit. 8. květen 2015) Dostupné z http://www.gregdonner.org/workbench/wb_13.html
- [9] Graphic User Interface Timeline. Fotografie. (cit. 8. květen 2015) Dostupné z <http://toastytech.com/guis/guitimeline2.html>
- [10] Egbert Schlungbaum. Model-based User Interface Software Tools. Georgia Institute of Technology. Graphics, Visualization & Usability Center, Georgia Institute of Technology. Listopad 1996.
- [11] Angel R. Puerta. A Model-Based Interface Development Environment. Software, IEEE. Červenec 1997, vol. 14, no. 4, s. 40-47. ISSN: 0740-7459
- [12] António M. Rosado Da Cruz, Joao P. Faria. Automatic Generation of User Interface Models and Prototypes from Domain and Use Case Models. User Interfaces. InTech 2010. ISBN-13: 9789533070841

- [13] Egbert Schlungbaum, Thomas Elwert. Automatic User Interface Generation from Declarative Models. CADUI. 1996, s. 3-17.
- [14] WxSmith plugin. (cit. 10. květen 2015) Dostupné z http://wiki.codeblocks.org/index.php?title=WxSmith_plugin
- [15] Swing GUI Builder. (cit. 10. květen 2015) Dostupné z <https://netbeans.org/features/java/swing.html>
- [16] Ultimate++ Overview. (cit. 10. květen 2015) Dostupné z [http://ultimatepp.org/www\\$suppweb\\$overview\\$en-us.html](http://ultimatepp.org/www$suppweb$overview$en-us.html)
- [17] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, Jonathan E. Shuste. UIML: an appliance-independent XML user interface language. Computer Networks. 17. květen 1999, vol. 31, no. 11-16, s. 1695–1708. doi:10.1016/S1389-1286(99)00044-4
- [18] XAML Overview (WPF). (cit. 13. květen 2015) Dostupné z <https://msdn.microsoft.com/en-us/library/ms752059.aspx>
- [19] Greg Brown. Introducing FXML: a Markup Language for JavaFX. 2011. (cit. 13. květen 2015) Dostupné z <http://fxexperience.com/wp-content/uploads/2011/08/Introducing-FXML.pdf>
- [20] Developing applications in MXML. (cit. 13. květen 2015) Dostupné z http://help.adobe.com/en_US/flex/using/WS2db454920e96a9e51e63e3d11c0bf5f39f-7fff.html
- [21] Ray Rischpater. Application Development with Qt Creator - Second Edition. Packt Publishing. Listopad 2014. ISBN: 1784398675 / 978-1784398675
- [22] Qt QML. (cit. 10. květen 2015) Dostupné z <http://doc.qt.io/qt-5/qtqml-index.html>
- [23] John R. Levine, Tony Mason, Doug Brown. Lex & Yacc. 2nd/updated edition. O'Reilly & Associates. Říjen 1992. ISBN: 1565920007
- [24] Zbigniew J. Czech, George Havas, Bohdan S. Majewski. An optimal algorithm for generating minimal perfect hash functions. Information Processing Letters. Květen 1992, vol. 43, no. 5, s. 257-264.
- [25] Eric Friedman, Itay Maman. Boost Variant. 2003. (cit. 10. květen 2015) Dostupné z http://www.boost.org/doc/libs/1_54_0/doc/html/variant.html
- [26] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. Society for Industrial and Applied Mathematics. 1972. ISSN: 0097-5397. DOI:10.1137/0201010
- [27] Qt Licensing. (cit. 8. květen 2015) Dostupné z <http://www.qt.io/qt-licensing-terms/>
- [28] Jutta Degener. ANSI C Yacc grammar. 1995. Dostupné z <http://www.lysator.liu.se/c/ANSI-C-grammar-y.html>

- [29] BorderImage QML Element. (cit. 8. květen 2015) Dostupné z <http://doc.qt.io/qt-4.8/qml-borderimage.html>

Příloha A

Gramatika

Následuje soubor pravidel pro gramatiku ve formátu *YACC* pro jazyk *CQML*. Terminály jsou psané velkým písmem a neterminální symboly malým. Neterminály *compound_statement*, *type_specifier* a *conditiona_expression* jsou převzaty z gramatiky pro jazyk *C*, a jejich zpracování je provedeno podle pravidel gramatiky jazyka *C* z [28].

Ukázka kódu A.1: Gramatika jazyka *CQML*

```
start_point
    : element_or_import_list
;
element_or_import_list
    : import_list element
    | element
;
import_list
    : import import_list
    | import
;
import
    : IMPORT STRING_LITERAL AS IDENTIFIER
;
element
    : IDENTIFIER '{' attribute_or_subelement_list '}'
    | IDENTIFIER '{' '}'
;
attribute_or_subelement_list
    : attribute_or_element ';' attribute_or_subelement_list
    | attribute_or_element ';'
;
attribute_or_element
    : element
    | event_handler
    | attribute
    | property
;
event_handler
    : IDENTIFIER ':' compound_statement
;
property
    : PROPERTY IDENTIFIER attribute
    | PROPERTY type_specifier attribute
    | PROPERTY IDENTIFIER IDENTIFIER
    | PROPERTY type_specifier IDENTIFIER
;
attribute
    : IDENTIFIER ':' conditional_expression
    | IDENTIFIER '.' IDENTIFIER ':' conditional_expression
;
```


Příloha B

Použité technologie

Flex

Flex je nástroj pro tvorbu skenerů pro lexikální analýzu. Jeho cílem je rozpoznat lexikální šablony v textovém vstupu a převést je na výstup ve formě posloupnosti symbolů (tokenů), které mohou být zpracovány dále například pomocí parseru.

Bison

Bison je víceúčelový generátor parserů, který z dané bezkontextové gramatiky vygeneruje deterministický LR parser nebo generalizovaný LR parser ve formě zdrojového kódu k programu v jazyce *C* nebo *C++*.

SDL2

SDL2 je multiplatformní knihovna umožňující přístup ke vstupním a výstupním zařízením systému. Podporuje platformy *Windows*, *Mac OS X*, *Linux*, *iOS* a *Android*. Je implementován v kódu jazyka *C* a tak jí lze použít i v rámci *C++* aplikací.

Příloha C

Syntaxe jazyka *CQML*

V této kapitole je na příkadech vysvětlena syntaxe jazyka *CQML* pro tvorbu *GUI*.

C.1 Tvorba elementů

Kód C.1 vytvoří hierarchii elementů, kde se na vrcholu nachází element základního typu *Element* a v něm se nachází dva elementy typu *Rectangle*, z nichž jeden obsahuje element typu *MouseArea*.

Ukázka kódu C.1: Syntaxe pro tvorbu hierarchie elementů.

```
Element
{
    Rectangle
    {
    };
    Rectangle
    {
        MouseArea
        {
        };
    };
}
```

C.2 Přiřazení hodnoty atributu

Kód C.2 ukazuje nastavení hodnoty atributu *width* pomocí výrazu a hodnotu atributu *height* pomocí funkce vracející požadovanou hodnotu.

Ukázka kódu C.2: Syntaxe přiřazení hodnoty elementu pomocí výrazu nebo funkce.

```
Element
{
    width : 500;
    height : {
        return 500;
    };
}
```

C.3 Provázání atributů

Kód C.3 demonstruje možnost provázat mezi sebou jednotlivé atributy. Vrchní element má nastavenou hodnotu atributu *width* na 500, s tímto atributem je následně provázán atribut *height*, který se nastaví na hodnotu o 100 větší. Parametry elementu *Rectangle* jsou pak provázány s hodnotami atributů nadřazeného elementu pomocí klíčového slova *parent*. Tudíž oba elementy mají schodné hodnoty atributů *width* a *height*.

Ukázka kódu C.3: Ukázka provázání atributů mezi sebou.

```
Element
{
    width : 500;
    height : width+100;
    Rectangle
    {
        width : parent.width;
        height : parent.height;
    };
}
```

C.4 Použití identifikátoru pro přístup k elementu

Prostřednictvím atributu *id* je možné označit element identifikátorem, pomocí něhož je možné k elementu přistupovat v rámci daného *CQML* souboru. V kódu C.4 je jednomu z elementů přiřazen identifikátor *firstRectangle*. Prostřednictvím tohoto identifikátoru je pak hodnota atributu *width* označeného elementu přiřazena jinému.

C.5 Rozšíření elementu o nový atribut

Kód C.5 ukazuje rozšíření elementu o nový atribut jménem *alpha* typu *int* pomocí klíčového slova *property*, přičemž je mu přiřazena hodnota 100.

Ukázka kódu C.4: Ukázka přístupu k elementu a jeho atributům pomocí identifikátoru.

```
Element
{
    Rectangle
    {
        id : firstRectangle;
        width : 500;
    };
    Rectangle
    {
        width : firstRectangle.width;
    };
}
```

Ukázka kódu C.5: Syntaxe rozšíření elementu o nový atribut.

```
Element
{
    property int alpha : 100;
}
```

C.6 Reference na element

V kódu C.6 je kořenový element rozšířen o atribut *refElement*, v němž se uchovává reference na atribut označený identifikátorem *firstRectangle*. Následně je přistupováno k atributu *width* referencovaného elementu prostřednictvím dané reference.

Ukázka kódu C.6: Ukázka přístupu k atributům elementu pomocí reference.

```
Element
{
    property Element refElement : firstRectangle;
    width : refElement.width;
    Rectangle
    {
        id : firstRectangle;
        width: 500;
    };
}
```

Ukázka kódu C.7: Syntaxe přiřazení funkce pro obsluhu události.

```
Element
{
    MouseArea
    {
        width : 100;
        height : 100;
        MouseClicked : {
            width = 500;
            height = 500;
        };
    };
}
```

C.7 Zpracování událostí

Kód C.7 ilustruje přiřazení funkce pro obsluhu události kliknutí myši. V tomto případě se po kliknutí myši na element *MouseArea* změní hodnoty jeho atributů *width* a *height* ze 100 na 500.

C.8 Použití externí logiky

Pro použití externí logiky definované v rámci uživatelské aplikace lze do kódu vygenerovaného překladačem zahrnout hlavičkový soubor pomocí příkazu *include*. Kód C.8 ukazuje použití funkce *GetWidth*, která je deklarovaná uvnitř importovaného souboru „logic.h“.

Ukázka kódu C.8: Ukázka kódu používajícího externí funkci deklarovanou v hlavičkovém souboru.

```
include "logic.h"
Element
{
    Rectangle
    {
        width : {
            return GetWidth();
        };
    };
}
```

C.9 Použití komponenty

Samostatné soubory v jazyce *CQML* lze používat jako samostatné komponenty v rámci jiných *CQML* souborů, s nimiž je možno manipulovat jako s jakýmkoli jiným elementem.

Ukázka kódu C.9: Zdrojový soubor komponenty.

```
Rectangle
{
    property int status : 0;
    width: 500;
    height: 500;
}
```

Toto je demonstrováno v ukázce kódu [C.10](#), kde je pomocí příkazu *import* načtena komponenta definovaná v souboru „component1.h“ (viz. kód [C.9](#)), která je pro použití označena klíčovým slovem *Square*. Pomocí daného klíčového slova jsou pak vytvořeny dvě instance dané komponenty a to stejným způsobem, jako by tomu bylo u běžných elementů.

Ukázka kódu C.10: Ukázka použití komponenty z jiného souboru.

```
import "component1.h" as Square
Element
{
    width : 500;
    height : 500;
    Square
    {
        width : 100;
        height : 100;
    };
    Square
    {
        width : 200;
        height : 200;
    };
}
```


Příloha D

Datové typy poskytované jazykem *CQML*

Knihovna *CQML* poskytuje několik hodnotových a referenčních datových typů pro použití při tvorbě *GUI*. Pro reprezentaci grafických elementů slouží referenční typy, aby se například při přiřazení elementu do nějaké proměnné element nekopíroval a bylo možné s ním manipulovat i prostřednictvím dané proměnné.

D.1 *Grafické elementy*

Grafické elementy jsou základními stavebními součástmi *GUI* a každý z nich může být umístěn v hierarchii. Ke všem z nich je v rámci jazyka *CQML* přístupováno jako k referenčním typům.

D.1.1 *Element*

Jedná se o základní element pro použití uvnitř hierarchie *GUI*. Sám nemá žádnou grafickou podobu a využívá se zejména jako kontejner pro umístění dalších grafických elementů.

Seznam atributů:

x - horizontální souřadnice elementu. (typ *int*)

y - vertikální souřadnice elementu. (typ *int*)

width - šířka elementu. (typ *int*)

height - výška elementu. (typ *int*)

visible - hodnota 1 znamená, že se element vykreslí. Hodnota 0 zamezí jeho vykreslování. (typ *int*)

enabled - hodnota 1 udává, že je element zapnutý a bude reagovat. Hodnota 0 element vypne. (typ *int*)

Seznam funkcí pro obsluhu událostí:

KeyPressed - funkce pro obsluhu stisknutí klávesy.

KeyReleased - funkce pro obsluhu puštění klávesy.

D.1.2 *Rectangle*

Jednoduchý grafický element sloužící pro vykreslení jednoduchého obdélníku o specifikovaných rozměrech. Lze mu nastavit barvu a šířku okraje a také barvu výplně.

Seznam atributů:

border - šířka okraje. (typ *float*)

borderColor - barva okraje. (typ *Color*)

color - barva výplně. (typ *Color*)

všechny atributy z typu *Element*

Seznam funkcí pro obsluhu událostí:

všechny funkce z typu *Element*

D.1.3 *MouseArea*

Podobně jako základní typ *Element*, nemá tento element žádnou grafickou podobu. Slouží zejména k obsluze vstupních událostí vyvolaných pomocí myši.

Seznam atributů:

všechny atributy z typu *Element*

Seznam funkcí pro obsluhu událostí:

MousePressed - funkce pro obsluhu stisknutí tlačítka myši.

MouseReleased - funkce pro obsluhu puštění tlačítka myši.

MouseClicked - funkce pro obsluhu kliknutí tlačítkem myši.

MouseEntered - funkce pro obsluhu najetí myši na plochu vymezenou elementem.

MouseExited - funkce pro obsluhu vyjetí myši z plochy vymezené elementem.

MouseMoved - funkce pro obsluhu pohybu myši.

MouseScrolled - funkce pro obsluhu otočení kolečkem myši.

všechny funkce z typu *Element*

D.1.4 *Image*

Grafický element sloužící pro zobrazení obrázku načteného ze souboru. V případě, že rozměry obrázku v souboru jsou odlišné od rozměrů elementu, pak se obrázek roztáhne či smrští do definované plochy.

Seznam atributů:

img - data obrázku. (typ *Img*)

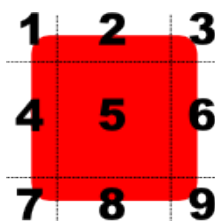
všechny atributy z typu *Element*

Seznam funkcí pro obsluhu událostí:

všechny funkce z typu *Element*

D.1.5 *ScaledImage*

Tento element je inspirován elementem *BorderImage* poskytovaný knihovnou *QML*. Načte obrázek, který následně rozřeže na devět částí (viz. obrázek D.1), přičemž rohy (1, 3, 7 a 9) zůstávají stejné velikosti jako v původním obrázku, ostatní okrajové části (2, 4, 6 a 8) mohou změnit velikost jen v jednom rozměru a prostřední část (5) může velikost měnit v obou rozměrech.



Obrázek D.1: Diagram ilustrující rozřezání obrázku pomocí elementu *ScaledImage* [29].

Seznam atributů:

leftBorder - hodnota udávající vzdálenost pravého řezu od pravého okraje v souřadnicích obrázku. (typ *int*)

rightBorder - hodnota udávající vzdálenost levého řezu od levého okraje v souřadnicích obrázku. (typ *int*)

topBorder - hodnota udávající vzdálenost horního řezu od horního okraje v souřadnicích obrázku. (typ *int*)

bottomBorder - hodnota udávající vzdálenost spodního řezu od spodního okraje v souřadnicích obrázku. (typ *int*)

všechny atributy z typu *Image*

Seznam funkcí pro obsluhu událostí:

všechny funkce z typu *Image*

D.1.6 Text

Grafický element sloužící pro vykreslení textu o definovaných parametrech.

Seznam atributů:

text - vykreslovaný řetězec. (typ *string*)

textColor - barva písma. (typ *Color*)

font - styl písma. (typ *Font*)

všechny atributy z typu *Element*

Seznam funkcí pro obsluhu událostí:

všechny funkce z typu *Element*

D.2 Hodnotové typy

Některá data používaná elementy jsou sloučena do struktur, aby bylo možné s nimi manipulovat najednou. K tomuto účelu je vytvořeno několik hodnotových typů.

D.2.1 Font

Datový typ, jenž slouží pro uchování údajů o stylu písma.

Seznam atributů:

family - název písma. (typ *string*)

size - velikost. (typ *int*)

D.2.2 Img

Tato datová struktura slouží k uchování dat cesty pro načtení obrázku ze souboru.

Seznam atributů:

src - cesta k obrázku. (typ *string*)

D.2.3 *Color*

Tato struktura slouží pro uchování údajů o barvě prostřednictvím hodnot pro červený, zelený a modrý kanál.

Seznam atributů:

red - hodnota červeného barevného kanálu. (typ *float*)

green - hodnota zeleného barevného kanálu. (typ *float*)

blue - hodnota modrého barevného kanálu. (typ *float*)

Příloha E

Návod na použití knihovny pro tvorbu *GUI* pomocí jazyka *CQML*

Uživatel musí definovat uživatelské rozhraní v jazyce *CQML* a uložit jej do souboru. Následně musí daný soubor předat překladači, který z něj vygeneruje zdrojové kódy pro použití v uživatelské aplikaci, která pro obsluhu *GUI* používá runtime knihovnu pro *CQML*.

E.1 Použití překladače

E.1.1 Spuštění překladu

Překladač v příkazové řádce přijímá jeden parametr, kterým je jméno souboru, v němž je definována vrchní komponenta *GUI* hierarchie. Ukázka kódu E.1 ilustruje spuštění překladače pro překlad souboru „src.cqml“ prostřednictvím příkazové řádky. V případě, že jsou v rámci daného souboru využívány komponenty z jiných souborů, tyto soubory se načtou automaticky.

Ukázka kódu E.1: Spuštění překladače pomocí příkazové řádky.

```
parser.exe "src.cqml"
```

E.1.2 Výstupní soubory

Výstupem překladače je několik souborů (*X* je pořadové číslo zpracovaného souboru):

parser_output.cpp - zdrojový soubor s vygenerovanými funkcemi pro inicializaci a obsluhu rozhraní.

parser_output.h - hlavičkový soubor, v němž jsou deklarovány hlavičky funkcí pro inicializaci a obsluhu rozhraní.

outputX.cpp - zdrojový soubor komponenty.

outputX.h - hlavičkový soubor komponenty.

outputfiles.txt - seznam všech vygenerovaných zdrojových souborů.

outputheaders.txt - seznam všech vygenerovaných hlavičkových souborů.

E.2 Použití knihovny v uživatelské aplikaci

E.2.1 Kompilace uživatelské aplikace

Pro překlad uživatelské aplikace používající zdrojové kódy vygenerované překladačem je nutné použít dynamickou knihovnu, která byla zkompileována do souborů „CQML_DLL.dll“ a „CQML_DLL.dll“ (projekt pro kompilaci knihovny je přiložen na CD viz. sekce G). Do projektu pro uživatelskou aplikaci je potřeba zahrnout soubory vygenerované překladačem a také hlavičkové soubory pro použití knihovny ze složky „cqml_include“.

E.2.2 Inicializace rozhraní

Před spuštěním hlavní programové smyčky, která komunikuje s *GUI*, musí být provedena inicializace. Nejprve program musí zavolat funkci `__CQML_Init`. Následně je nutné přiřadit instance rozhraní pro vykreslování a správu externích zdrojů pomocí funkcí `SetDrawIFace` (viz. sekce E.2.5) a `SetResourceManager` (viz. sekce E.2.4). Na závěr je potřeba inicializaci ukončit zavoláním funkce `__CQML_Start`, která spustí logiku pro obsluhu *GUI*. Pro volání všech funkcí pro obsluhu *GUI* musí být do zdrojového souboru zahrnut soubor „qml_includes.h“ ze složky „cqml_include“ a také soubor „parser_output.h“ vygenerovaný překladačem.

E.2.3 Zpracování vstupu

Vstup se zpracovává prostřednictvím událostí, které musí být ve specifickém formátu předány knihovně prostřednictvím funkce `PushEvent`. Tato funkce přijímá jeden parametr typu `CQMLEvent`, ve kterém jsou uložena specifika dané vstupní události. Struktura datového typu `CQMLEvent` je uvedena v kódu E.2. Kód E.3 ilustruje předání události stisknutí klávesy knihovně.

E.2.4 Rozhraní pro zpracování externích zdrojů

Pomocí funkce `SetResourceManager` se nastaví rozhraní pro správu externích zdrojů, to je jí předáno prostřednictvím ukazatele na instanci daného rozhraní. Rozhraní musí dědit z abstraktní třídy `ResourceManagerIFace` a implementovat metody ukázané v kódu F.1.

E.2.5 Vykreslovací rozhraní

Pomocí funkce `SetDrawIFace` se nastaví rozhraní pro vykreslování, to je jí předáno prostřednictvím ukazatele na instanci daného rozhraní. Rozhraní musí dědit z abstraktní třídy `DrawIFace` a implementovat metody ukázané v kódu E.5.

Ukázka kódu E.2: Struktura datového typu *CQMLEvent*.

```

int EventType; //hodnota CQML_KEY_EVENT nebo CQML_MOUSE_EVENT
// následující hodnoty platí jen pro CQML_KEY_EVENT
    int keyEvent.action; // KEY_RELEASED nebo KEY_PRESSED
    int keyEvent.key; // kód stisknuté klávesy
// následující hodnoty platí jen pro CQML_MOUSE_EVENT
    int mouseEvent.action; // MOUSE_WHEEL_SCROLLED,
        // MOUSE_MOVEMENT, MOUSE_BUTTON_RELEASED
        // nebo MOUSE_BUTTON_PRESSED
    int mouseEvent.button; // kód stisknutého tlačítka
    int mouseEvent.x; // souřadnice x ukazatele myši
    int mouseEvent.y; // souřadnice y ukazatele myši
    int mouseEvent.relativeX; // relativní pohyb v ose x
    int mouseEvent.relativeY; // relativní pohyb v ose y

```

Ukázka kódu E.3: Předání vstupu z klávesnice knihovně.

```

CQMLEvent e;
e.EventType = CQML_KEY_EVENT;
e.keyEvent.action = KEY_PRESSED;
e.keyEvent.key = c; // c je hodnota označující klávesu
PushEvent(e);

```

Ukázka kódu E.4: Metody rozhraní pro správu externích zdrojů.

```

void* LoadFont(const char * fontStr, int fontSize);
// načte data písma a vrátí na ně ukazatel
// fontStr je název písma
// fontSize je velikost písma

ImageData LoadImage(const char * path);
// načte data obrázku a vrátí je ve formě struktury popsané níže
// path je cesta k souboru s obrázkem

struct ImageData
{
    void * img; // ukazatel na načtená data
    int w; // šířka načteného obrázku
    int h; // výška načteného obrázku
};

```

Ukázka kódu E.5: Metody vykreslovacího rozhraní.

```
void DrawFilledRectangle(int x,int y, int w, int h,
    float r, float g, float b);
// vykreslí jednobarevný obdélník
// x, y jsou souřadnice obdélníku
// w, h jsou rozměry obdélníku
// r, g, b určují barevné složky barvy výplně obdélníku

void DrawFilledBorderedRectangle(int x, int y, int w, int h,
    float r, float g, float b, float border,
    float br, float bg, float bb);
// vykreslí obdélník s okrajem
// x, y jsou souřadnice obdélníku
// w, h jsou rozměry obdélníku
// r, g, b určují barevné složky barvy výplně obdélníku
// border určuje šířku okraje
// br, bg, bb určují barevné složky barvy okraje obdélníku

void DrawText(int x, int y, int w, int h,
    const char* text, void* font, float r, float g, float b);
// vykreslí text
// x, y jsou souřadnice textu
// text je textový řetězec, který se má vykreslit
// font je ukazatel na data písma
// r, g, b určují barevné složky barvy písma

void DrawImage(int x, int y, int w, int h, void* image);
// vykreslí obrázek
// x, y jsou souřadnice obrázku
// w, h jsou rozměry, do kterých se obrázek roztáhne
// image je ukazatel na data obrázku

void DrawImageSegment(int x, int y, int w, int h,
    void* image, int iX, int iY, int iW, int iH);
// vykreslí určitý výřez obrázku
// x, y jsou souřadnice obrázku
// w, h jsou rozměry, do kterých se obrázek roztáhne
// image je ukazatel na data obrázku
// iX, iY jsou souřadnice výřezu ve zdrojovém obrázku
// iW, iH jsou rozměry výřezu ve zdrojovém obrázku
```

E.2.6 Funkce volané z hlavní programové smyčky

Pro vykreslování *GUI* uživatelská aplikace musí zavolat funkci `__CQML_Draw`. Pro přepočítání hodnot jednotlivých elementů v hierarchii uživatelská aplikace musí zavolat funkci `__CQML_Update`.

Příloha F

Rošiřování jazyka *CQML* o nové vestavěné typy a elementy.

Jazyk *CQML* lze rozšířit o další vestavěné typy a to pomocí speciálních maker v souboru „struct_definition_macros.h“. Nejprve programátor vytvoří makro pro definici struktury daného typu a následně dané makro zaregistruje. Toto je ilustrováno v kódu [F.1](#), kde je přidán nový element jménem *EnhancedElement*, který je odvozen z typu *Element*. Veškerou funkčnost, která není poskytována již u ostatních elementů, musí však programátor implementovat v rámci zdrojového kódu knihovny.

Upravený soubor „struct_definition_macros.h“ musí programátor vložit do projektů všech částí knihovny *CQML*, tj. překladače, preprocesoru datových typů i runtime knihovny a také do projektu uživatelské aplikace. Následně musí zkompileovat a spustit preprocesor datových typů, který vygeneruje zdrojový soubor „preparser_output.cpp“, který musí být zahrnut do zdrojových kódů knihovny, resp. je jím nutno přepsat ten již existující. Po té je nutné znovu zkompileovat překladač a runtime knihovnu. Poté už je možné nový datový typ použít.

Ukázka kódu F.1: Registrace nového vestavěného datového typu.

```
// makro STRUCT_ENHANCED vytvoří strukturu s následujícími členy:
// 1. atribut typu int jménem Attribute s výchozí hodnotou 0
// 2. atribut typu float jménem HiddenAttribute, který není
//    přístupný z CQML, s výchozí hodnotou 0
// 3. virtuální metoda jménem VirtualMethod s návratovým
//    typem void a žádnými a parametry
// 4. metoda jménem Method s návratovým typem int
//    a dvěma parametry typu int
// 5. ukazatel na funkci pro obsluhu nějaké události, který je
//    z CQML přístupný pod jménem EventHandler
//    a je inicializován na nulovou hodnotu.
//    v tomto případě přijímá parametr typu QMLMouseEvent,
//    který je v CQML dostupný pod názvem EVENT
#define STRUCT_ENHANCED(MF, F, M, ME, MEV) \
    F(int, Attribute, 0) \
    MF(float, HiddenAttribute, 0) \
    MEV(void, VirtualMethod) \
    ME(int, Method, int, int) \
    M(void, CustomEventHandler, 0, QMLMouseEvent EVENT) \

// následně se dané makro zaregistruje přidáním nového řádku
// do makra REGISTRATION (viz. jeho poslední řádek).
// první argument na posledním řádku určuje název registrovaného
// makra
// druhý argument na posledním řádku určuje jméno elementu
// poslední argument na posledním řádku určuje, z jaké struktury
// nový prvek dědí – pro případ, že chceme vytvořit
// hodnotový datový typ, který z ničeho nedědí, odstraní se
// poslední argument a makro MACRO3 se změní na MACRO2
#define REGISTRATION(MACRO2, MACRO2REF, MACRO3) \
    REGPRIMITIVE(int) \
    MACRO2(STRUCT_COLOR, Color) \
    MACRO2REF(STRUCT_ELEMENT, Element) \
    MACRO3(STRUCT_RECTANGLE, Rectangle, Element) \
    MACRO3(STRUCT_ENHANCED, EnhancedElement, Element) \
```

Příloha G

Obsah CD

složka/soubor	obsah
index.html	výchozí stránka
readme.txt	popis významu adresářů a souborů
install.txt	postup instalace a spuštění
text/DP.pdf	tištitelná forma diplomové práce
text/latex	složka se zdrojovými soubory pro vygenerování tištitelné formy
src/Solution	složka obsahující zdrojové kódy C++ aplikací
src/cqml_includes	složka obsahující hlavičkové soubory pro použití v uživatelských aplikacích
src/cqml_sources	složka obsahující zdrojové soubory CQML vytvořené pro ukázkou v rámci diplomové práce
src/qml_sources	složka obsahující zdrojové soubory QML vytvořené pro ukázkou v rámci diplomové práce
exe/translator	složka obsahující spustitelnou verzi překladače
exe/CQML_run_app	složka obsahující projekt pro zobrazení GUI, přeloženého pomocí překladače
exe/Executable_test_app	složka obsahující spustitelnou verzi GUI vytvořenou pro ukázkou v rámci diplomové práce
docs/	adresář se soubory dokumentace