ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Spolupráce průmyslového kooperativního robota s pracovníkem s holografickými brýlemi
Student:	Bc. Adam Podroužek
Vedoucí:	Ing. Miroslav Skrbek, Ph.D.
Studijní program:	Informatika
Studijní obor:	Návrh a programování vestavných systémů
Katedra:	Katedra číslicového návrhu
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Seznamte se s průmyslovým robotem YuMi a jeho programováním. Dále se seznamte s holografickými brýlemi a vývojem aplikací pro ně. Navrhněte a realizujte programové vybavení, které dovolí komunikovat holografickým brýlím s řídící jednotkou robota, tak aby bylo možné promítat do zorného pole pracovníka vybaveného holografickými brýlemi stavové informace robota, varování pracovníkovi při přesunech materiálu nebo chapadel, instrukce pracovníkovi, jak má při kooperaci s robotem postupovat, apod. Spolupráci člověk - robot demonstrujte v demo aplikaci. Rozsah práce upřesněte po dohodě s vedoucím práce.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Hana Kubátová, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 29. ledna 2019

Diplomová práce

Spolupráce průmyslového kooperativního robota s pracovníkem s holografickými brýlemi

Bc. Adam Podroužek

Katedra číslicového návrhu
Vedoucí práce: Ing. Miroslav Skrbek, Ph.D.

7. května 2019

Poděkování

Tímto bych rád poděkoval svému vedoucímu práce Ing. Miroslavu Skrbkovi, Ph.D., že mi umožnil pracovat v laboratoři vestavných systému s technologiemi, ke kterým bych se jinak nedostal. Rovněž tímto děkuji své rodině, která mě podporovala v průběhu celého vysokoškolského studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 7. května 2019 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2019 Adam Podroužek. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Podroužek, Adam. *Spolupráce průmyslového kooperativního robota s pracovníkem s holografickými brýlemi.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato diplomová práce má za cíl seznámit čtenáře s problematikou vývoje řídícího softwaru pro kooperativního robota, který ke spolupráci s člověkem využívá rozhraní holografických brýlí. Rešeršní část práce se zabývá analýzou a popisem vývoje pro robota *YuMi* v prostředí RobotStudio a vývojem aplikace pro holografické brýle *HoloLens*. Praktická část práce se zaměřuje na vývoj jednoduché aplikace, která demonstruje možnosti komunikace mezi robotem *YuMi* a holografickými brýlemi *HoloLens*.

Z této práce může těžit především laboratoř inteligentních vestavných systémů na FIT ČVUT, která se bude se studenty zabývat vývojem aplikací na robota *YuMi*. Zároveň může práce posloužit jako výchozí bod pro případné zájemce o implementaci podobného kooperativního systému v průmyslu.

Klíčová slova Hololens, YuMi, kooperativní robot, rozšířená realita

Abstract

This diploma thesis aims to introduce the reader with the topic of development of a control software for a cooperative robot which interfaces with a human by holographic glasses. First part of the thesis deals with analysis and description

of the development process for *YuMi* robot in RobotStudio IDE as well as with development of applications for holographic glasses *HoloLens*.

The laboratory of intelligent embedded systems on FIT CTU can make use of this work when developing applications with students during classes. The thesis can also be a starting point for someone from the industry interested in implementing such cooperative system.

Keywords Hololens, YuMi, cooperative robot, mixed reality

Obsah

Odkaz na tuto práci vi

Úvod **1**

1 Cíle práce **3**

2 Analýza **5**

2.1 Robot YuMi 5

2.1.1 Základní charakteristika 6

2.1.2 RobotStudio 7

2.1.3 RAPID 11

2.1.4 FlexPendant 18

2.2 Holografické brýle HoloLens 21

2.2.1 Základní charakteristika 21

2.2.2 Unity 23

2.2.3 Visual Studio 25

2.3 Možnosti komunikace mezi YuMi a HoloLens 25

2.3.1 PC SDK 25

2.3.2 TCP sockety 26

2.3.3 Robot Web Services 26

3 Návrh řešení **29**

3.1 Volba technologií 30

3.1.1 Knihovna Mixed Reality Toolkit 30

3.1.2 Způsob komunikace 30

3.2 Demonstrační aplikace 31

3.2.1 Komunikační protokol 31

3.2.2 Řízení robota 32

3.2.3 Aplikace pro HoloLens 34

4 Implementace **37**

4.1 YuMi 37
4.1.1 Příprava robota a pracoviště 37
4.1.2 Nastavení RobotStudia 37
4.1.3 Implementace modulů 39
4.2 HoloLens 44
4.2.1 Vytvoření projektu v Unity 44
4.2.2 Tvorba scény 45
4.2.3 Implementace logiky 50
4.2.4 Export projektu a nahrání do HoloLens 53

5 Testování a budoucí práce **57**
5.1 Testování 57
5.2 Problémy při vývoji 58
5.2.1 Dlouhý proces testování aplikace na HoloLens 58
5.2.2 Přechod na MRTK v2 RC1 58
5.2.3 Nefunkční plovoucí klávesnice v HoloLens 59
5.3 Budoucí práce 59

Závěr **61**

Literatura **63**

A Seznam použitých zkratek **67**

B Obsah přiloženého CD **69**

Seznam obrázků

2.1 Robot YuMi 6
2.2 Prostředí RobotStudia 7
2.3 Karta Home 8
2.4 Karta Modelling 9
2.5 Karta Simulation 9
2.6 Karta Controller 10
2.7 Karta RAPID 10
2.8 Znázornění zóny [1] 14
2.9 FlexPendant 19
2.10 Holografické brýle HoloLens 21
2.11 Základní gesta HoloLens 22
2.12 Příklad komponent z MRTK 24
2.13 Příklad použití ARToolkit v HoloLens [2] 24
2.14 FlexPendant [3] 27

3.1 Základní diagram systému [4] [5] 29
3.2 Příklad komunikace 32
3.3 Konečný automat řízení robota YuMi 33
3.4 Souřadnicové systémy 36

4.1 YuMi s xylofonem 38
4.2 Finální scéna a hierarchie objektů 45
4.3 Vyčištěná example scéna 46
4.4 Tlačítka a hierarchie objektů 47
4.5 Panel pro připojení a jeho hierarchie objektů 48
4.6 Debugovací panel a jeho hierarchie objektů 48
4.7 Virtuální plocha s indikátorem a bariérou 49
4.8 Rozhraní komponenty skriptu `SceneManager` 52
4.9 Receiver událostí z tlačítek 54
4.10 Build settings 55

5.1 Snímek obrazovky z testování 58

Seznam tabulek

2.1 Základní datové typy RAPIDu 13
2.2 Operátory v RAPIDu 13
2.3 Vybrané rozšířené datové typy RAPIDu 13
2.4 Argumenty *move* instrukcí 17
2.5 SmartGripper instrukce 17

3.1 Zprávy, posílané mezi robotem a brýlemi 31

Úvod

V dnešní době stále častěji slýcháme pojmy jako automatizace, robotizace a tzv. *průmysl 4.0*. Vznikají obavy z možného budoucího úbytku pracovních míst v důsledku nahrazování člověka robotem. Přece však stále existují místa, kde se nejvyšší efektivity práce dosahuje nikoliv úplnou robotizací, ale určitou mírou kooperace – tedy spolupráce robota a člověka na jednom pracovišti.

Představme si následující situaci. V moderním podniku, který je vybaven průmyslovými roboty, kteří automatizují produkci, je stále třeba lidské pracovní síly. V tomto technologicky nabitém prostředí je poptávka po nových odborných profesích, které dříve na podobných pracovištích neexistovaly – od techniků obsluhujících a udržujících roboty až po programátory, kteří tyto roboty programují. Takzvaně u pásu je ale (dle typu výrobní linky) stále přítomno několik pracovníků.

Jak bude postupovat automatizace podniku, budou muset tito pracovníci, které nebude možné pořád úplně nahradit strojem, stále častěji kooperovat se složitými roboty na lince. Příkladem budiž montáž nějakého výrobku – robot sestaví část, kde je nutná vysoká přesnost a zároveň ji lze snadno automatizovat. Poté předá výrobek pracovníkovi, který provede sled úkonů pro robota netriviálních a následně jej opět předá robotovi. Touto spoluprací výrobek zhotoví.

Aby pracovník věděl, jak má s robotem spolupracovat, je nutné ho nejprve zaškolit. Je-li proces kooperace složitější, může docházet k chybám nebo ke zdržením, protože pracovník musí hledat v instrukcích, jak postupovat. Část těchto problému se můžeme pokusit vyřešit například použitím, dnes často znějící, rozšířené reality.

Problém je i na straně programátorů těchto kooperativních robotů. Technologicky jdou neustále dopředu a vývoj programů na ovládání a koordinaci pohybů není triviální. Těchto problémů se bude tato práce dotýkat.

Kapitola 1

Cíle práce

Cílem této práce je demonstrovat kooperaci mezi pracovníkem a robotem YuMi od společnosti ABB. Osoba kooperující s robotem bude mít holografické brýle HoloLens od firmy Microsoft, na kterých přímo uvidí ovládací prvky, pomocí kterých může s robotem komunikovat a zároveň uvidí na pracovní plochu před sebou. Robot bude moci do brýlí indikovat například výzvu k převzetí výrobku, žádost o umístění součásti na pracovní plochu nebo zvýraznění oblasti, ve které robot právě manipuluje s předmětem.

Dalším cílem je seznámit čtenáře o postupech při vývoji programů pro robota YuMi a holografické brýle HoloLens. Podle těchto popisů bude moci postupovat případný zájemce při implementaci vlastních programů.

Kapitola 2

Analýza

V této kapitole se seznámíme s platformami a vývojovými nástroji robota *YuMi* a holografických brýlí *HoloLens*. U platformy YuMi si popíšeme základní vlastnosti robota, analyzujeme základní funkce vývojového prostředí *Robot-Studio*, seznámíme se s programovacím jazykem RAPID a nakonec si ukážeme úvod do práce s ovládacím panelem *FlexPendant*.

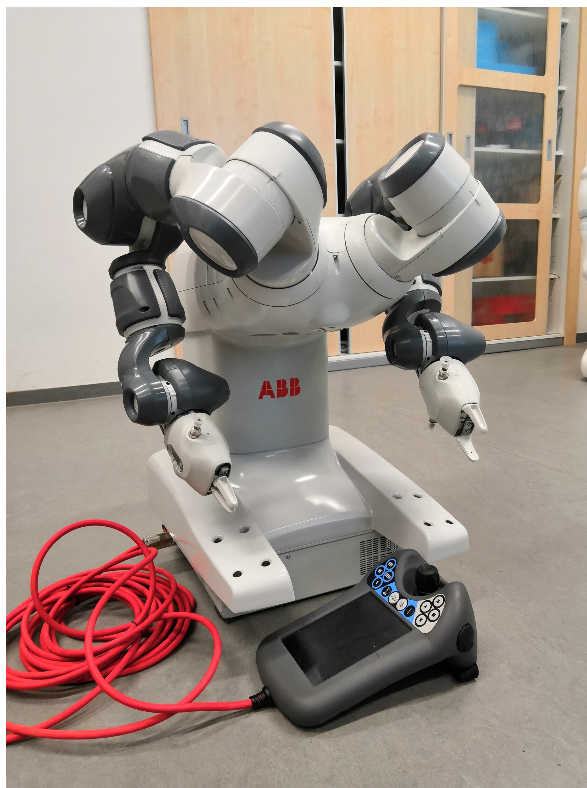
U holografických brýlí HoloLens si projdeme způsob vývoje ve vývojových prostředích *Unity* a *Visual Studio* a seznámíme se s platformou HoloLens jako takovou.

V poslední části analýzy budeme diskutovat o možnostech komunikace mezi výše zmíněnými platformami a analyzujeme jejich výhody a nevýhody.

2.1 Robot YuMi

YuMi (viz obr. 2.1) je relativně novým typem robota od spol. ABB. Tento stroj je prezentován jako zástupce nové éry kooperativní robotiky v průmyslu. Mezi jeho hlavní vlastnosti patří:

- kompaktnost a nízká hmotnost,
- vysoká přesnost pohybů,
- zabudovaný kontrolér přímo v těle robota,
- versatilní nástroj SmartGripper,
- učení pohybů pomocí ručního navádění,
- kamera pro strojové vidění.



Obrázek 2.1: Robot YuMi

2.1.1 Základní charakteristika

Jedná se o stacionárního robota, který je uchycen ke stolní desce. Jeho hmotnost je 38 kilogramů, šířka 339 mm, hloubka 497 mm. Ve srovnání s jinými průmyslovými roboty je YuMi velmi kompaktní a lehký – většina vnější konstrukce je plastová.

Má dvě na sobě nezávislá ramena schopná pracovat s malými součástkami. Ramena mají dohromady 14 stupňů volnosti – tedy os, kolem kterých se může otáčet a jsou schopna dosáhnout až do vzdálenosti 559 mm. Maximální zátěž, kterou může rameno zdvihnout, je 500 gramů. Maximální rychlost otáčení jednotlivých os se pohybuje mezi 180° a 400° za sekundu.

Na konce ramen lze uchytit nástroje, pomocí kterých robot následně manipuluje s objekty. Základním nástrojem je tzv. *SmartGripper*, jehož hlavním prvkem jsou svěrky, které slouží k přesnému úchopu předmětu. Dále může tento nástroj obsahovat aktivní přísavky pro manipulaci s plochými předměty a v neposlední řádě také kameru pro inteligentní vidění [6].

YuMi je programován v prostředí RobotStudio, které je podrobně popsané v podsekci 2.1.2. To co ovšem robota zajímá, je pouze program v jazyku RAPID (sekce 2.1.3). V kódu jsou jen jednoduché instrukce typu „přesuň se

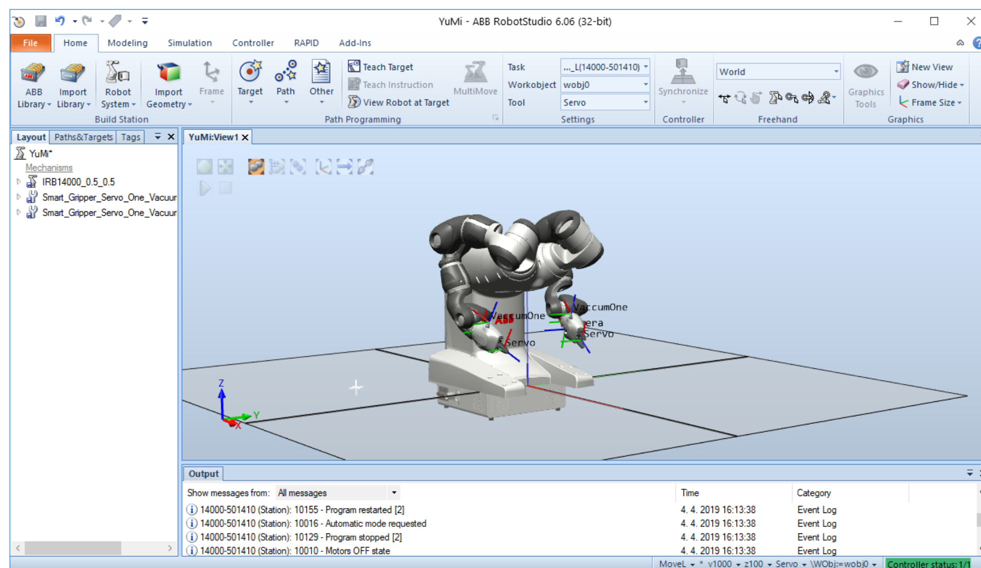
nástrojem na tyto souřadnice“, „čekej n milisekund“ nebo „sevři svěrky na SmartGripperu“. RobotStudio je komplexní nástroj, který nám zjednodušeně řečeno jen usnadňuje tvorbu RAPID kódu a umožňuje jej simulovat.

Bez RobotStudia se YuMi ovládá (a částečně i programuje) pomocí FlexPendantu (viz sekce 2.1.4), což je ovládací LCD panel připojený k robotovi.

2.1.2 RobotStudio

RobotStudio je vývojové prostředí od spol. ABB určené výhradně k vývoji řízení jejich robotů. Podporuje plně offline vývoj, což znamená, že lze projít celým procesem vývoje řízení robota bez přístupu k samotnému robotovi.

V následujících podsekcích jsou podrobněji popsány jednotlivé důležité funkce RobotStudia a postupy, jak s programem pracovat při vývoji řízení robota YuMi.



Obrázek 2.2: Prostředí RobotStudia

2.1.2.1 Základní pojmy

Před popisem prostředí RobotStudia je nutné nejprve vysvětlit několik terminologických pojmů, se kterými se v průběhu vývoje často pracuje.

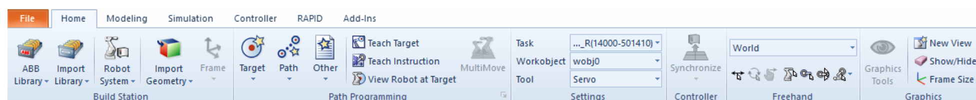
- **Tool** je pracovní nástroj, se kterým robot pracuje. V případě YuMiho jsou to prvky SmartGripperu na koncích jeho ramen (např. svěrky pro úchop předmětu, přísavky nebo kamera).

- **Target**, neboli cíl, je pozice v prostoru zadaná 3 souřadnicemi a vektorem otočení. V kódu RAPIDu je tento objekt definován klíčovým slovem `robtarget` (viz 2.1.3.1).

- **Workobject** je ve své podstatě báze souřadnicového systému. V rámci workobjectu se definují výše zmíněné targety. Můžeme si vytvořit více workobjectů a zadávat v nich pozice. Výhoda je v tom, že pokud si rozmyslíme, že chceme všechny pozice posunout nebo otočit, stačí pouze modifikovat workobject a pozice se rovněž transformují. Workobject s pozicemi je vztažen k jednomu konkrétnímu ramenu.

- **Path** je trajektorie pohybu procházející výše definovanými cíli. Pohyb k targetu se uskutečňuje vždy konkrétním nástrojem (*tool*). V kódu RAPIDu je trajektorie definována jako obyčejná procedura, která může i nemusí obsahovat instrukce typu *move*. Jazyk RAPID a jeho instrukce jsou popsány v sekci 2.1.3.

2.1.2.2 Karta Home



Obrázek 2.3: Karta Home

Na kartě *Home* (obr. 2.3) se nachází přehled základních funkcí. V sekci *Build Station* konfigurujeme stanici a zejména robota (v terminologii ABB se nazývá *System*). Ve vedlejší sekci lze vytvářet pozice v rámci stanice, které můžeme následně použít při tvorbě trajektorií pohybu.

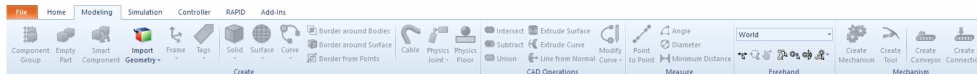
Z nabízených funkcí nás budou zajímat zejména tlačítka

- *Target*,
- *Path*,
- *Teach Target* a
- prvky v sekci *Freehand*.

Cíl (target) lze v prostoru vytvářet třemi způsoby. První způsob je pomocí tlačítka *Target* manuálním zadáváním souřadnic. To je v mnoha případech relativně zdlouhavé a nepraktické. Proto můžeme využít i druhý způsob, kdy napozicujeme aktivní rameno pomocí funkcí v sekci *Freehand* a poté pozici vytvoříme tlačítkem *Teach Target*. Třetí způsob zadávání souřadnic je s využitím FlexPendantu (ovládacího panelu připojenému k robotovi), což je blíže popsáno v sekci 2.1.4.4.

Funkcí *Path* vytvoříme trajektorii pohybu, do které přetáhneme v levém podokně již definované cíle a vytváříme *move* instrukce. Kterým nástrojem se pohyb uskuteční je zřejmé na horním panelu v sekci *Settings*, kde vidíme aktivní nástroj, workobject a task (rameno). Definicí path se v kódu RAPIDu vytvoří procedura se stejným jménem a v ní příslušné *move* instrukce.

2.1.2.3 Karta Modelling



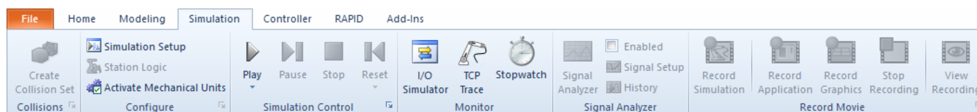
Obrázek 2.4: Karta Modelling

Karta *Modelling* slouží zejména k tvorbě a modelování pracovní stanice. Robot se v ostrém provozu bude pochopitelně nacházet na lince nebo pracovišti, které chceme mít v průběhu vývoje namodelované v RobotStudiu a rovnou s ním pracovat. Stanici lze komplexně namodelovat včetně pohyblivých součástí, jako jsou například pásové dopravníky.

V jednoduchých případech si vystačíme s několika málo funkcemi v sekci *Create*. Tlačítkem *Solid* můžeme vytvořit fyzický objekt (kostku, kouli, atd.) a umístit jej do souřadnicového systému (resp. do nějakého workobjectu). Budeme-li například chtít, aby robot manipuloval se dvěma kostkami, které bude mít před sebou na pracovní ploše, vytvoříme je funkcí *Solid*→*Cube*. V průběhu simulace pak budeme moci s těmito kostkami pracovat.

Mezi pokročilé funkce patří například tvorba mechanismu nebo nového nástroje, což je nad rámec této práce.

2.1.2.4 Karta Simulation

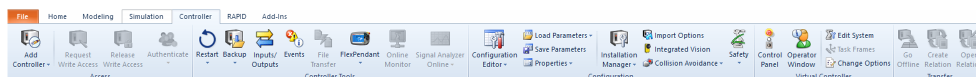


Obrázek 2.5: Karta Simulation

Na kartě *Simulation* ovládáme samotnou simulaci. Ovládací prvky jsou intuitivní (play, pause, stop, reset). Při spouštění simulace musíme mít na paměti, že se spouští pouze to, co je RAPIDu a ne nutně to, co vidíme v levém podokně v *Paths&Targets*. Mezi modelovým prostředím a RAPIDem proto musíme provádět synchronizaci, což je popsáno v podsekci 2.1.2.6.

Užitečná funkce je také nahrávání simulace jako videa do souboru. Lze nahrávat celé okno (*Record Application*) nebo pouze simulátor (*Record Graphics*).

2.1.2.5 Karta Controller



Obrázek 2.6: Karta Controller

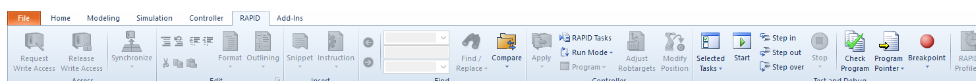
V kartě *Controller* najdeme všechny funkce související s kontrolérem robota. První funkcí je *Add Controller→One Click Connect*. Touto funkcí se připojíme k fyzickému kontroléru, který je zabudovaný v robotovi. Obvykle ale chceme využít možnost simulace, kterou nechceme spouštět, jen když máme připojený fyzický kontrolér. Proto máme také možnost vytvořit tzv. virtuální (offline) kontrolér. Ten vznikne, když použijeme funkci *Go Offline*. Zjednodušeně řečeno tím vytvoříme kopii připojeného kontroléru a můžeme pracovat na simulaci i když nemáme fyzický přístup k robotovi.

Z karty *Controller* můžeme dále aktivní (fyzický nebo virtuální) kontrolér restartovat nebo zálohovat jeho stav a souborový systém do backup souboru. Máme možnost také plně simulovat interakci s FlexPendantem (viz sekce 2.1.4).

Z této karty rovněž provádíme nahrání námi vytvořeného programu do fyzického kontroléru. Způsob výměny dat se může zdát na první pohled relativně neintuitivní. Musíme vytvořit relaci mezi fyzickým a virtuálním kontrolérem a přes tuto relaci provedeme synchronizaci. Jakmile máme připojeného robota a máme virtuální kontrolér, vytvoříme mezi nimi relaci pomocí *Create Relation*. Poté vytvořenou relaci otevřeme a můžeme synchronizovat – funguje to na podobném principu, jako když synchronizujeme mezi RAPIDem a modelovacím prostředím (nebo obráceně). Rozdíly mezi oběma zařízeními jsou označeny a můžeme zvolit směr synchronizace.

Před synchronizaci směrem do fyzického kontroléru je nutné si předem vyžádat právo k zápisu pomoci *Request Write Access*. Pokud máme FlexPendant v manuálním módu, objeví se na obrazovce dialog, který nás informuje o příchozí žádosti o zápis, kterou musíme potvrdit. V automatickém módu FlexPendant žádost schválí ihned.

2.1.2.6 Karta RAPID



Obrázek 2.7: Karta RAPID

Na kartě *RAPID* se budeme pohybovat především v samotném editoru kódu. Přesto je zde několik funkcí, které stojí za zmínku. Jsou to funkce:

- *Synchronize*,
- *Snippet* a
- *Apply*.

Funkce *Synchronize* slouží k přenosům změn mezi kódem RAPIDu a modelovacím prostředím (resp. stanicí). Představme si to na následujícím příkladu. V kartě *Home* přidáme ke každému ramenu 4 targety a vytvoříme z nich 2 trajektorie pohybu (pro každé rameno zvlášť). V hlavním okně se nám body i trajektorie zobrazí, avšak když spustíme simulaci, změny se neprojeví. To je způsobeno tím, že simulátor vykonává striktně jen to, co je v RAPIDu. Pokud tedy provedeme nějaké změny ve „WISIWYG[1]" prostředí, musíme je přenést do RAPIDu funkcí *Synchronize→Synchronize to RAPID*. Obráceně to platí stejně – změníme-li RAPID a chceme, aby se změny ukázaly i v modelovacím prostředí (podokně *Paths&Targets*), musíme je analogicky aplikovat funkcí *Synchronize→Synchronize to Station*.

Funkce *Snippet* slouží k zobrazení příkladů kódu řešící specifický problém. Jsou zde například kusy kódu pro základní pohyb ramen, kalibraci kamery nebo pohyb ramene na pozici detekovaného objektu.

Jednou z neintuitivních věcí při používání editoru RAPIDu je ukládání. Ve standardních nástrojích jsme zvyklí na používání klávesové zkratky `CTRL+S`. Zde se však zdrojové soubory ukládají do souborového systému kontroléru. Použijeme-li klasické `CTRL+S`, uloží se soubory do našeho lokálního systému a nikoliv do kontroléru. K tomu slouží výše jmenovaná funkce *Apply*, kterou z tohoto důvodu při vývoji používáme místo normálního ukládání.

2.1.3 RAPID

RAPID je vysoko-úrovňový jazyk určený k programování industriálních robotů od společnosti ABB. Vznikl v roce 1994 a nahradil starší jazyk ARLA[2]. Je to strukturovaný, procedurální jazyk, což z něj dělá jednoduchý jazyk na pochopení [7]. Je case-insensitive. Pro představu je v listingu 1 krátká ukázka kódu v RAPIDu. V následujících podsekcích budou popsány jeho základní vlastnosti a jak se používá při programování robota YuMi.

Mezi základní charakteristiky tohoto jazyka patří například

- procedury,
- funkce,
- „trap rutiny" – obsluhy přerušení,
- aritmetické a logické operace,

[1] What You See Is What You Get
[2] ASEA Programming Robot Language

- automatické ošetřování chyb,
- moduly,
- multi-tasking.

```
MODULE MainModule
  VAR num length;
  VAR num width;
  VAR num area;

  PROC main()
    length := 10;
    width := 5;
    area := length * width;
    TPWrite "Obsah obdélníku je " \Num:=area;
  END PROC
ENDMODULE
```

Listing 1: Ukázka RAPID kódu

2.1.3.1 Proměnné, datové typy a operátory

Jazyk RAPID zná tří typy proměnných:

- proměnná,
- persistentní proměnná,
- konstanta.

Standardní proměnná funguje obdobně jako v jiných jazycích. Po přiřazení hodnoty ji proměnná drží, i když program zastavíme a opětovně spustíme. Hodnota zanikne až po vyresetování program pointeru. Deklarace proměnné docílíme použitím klíčového slova `VAR`.

Persistentní proměnná se chová téměř stejně jako obyčejná proměnná. Jedinou změnou je, že hodnota nezaniká při resetování program pointeru. Deklarována je klíčovým slovem `PERS`.

Konstanta je proměnná, které nelze za běhu měnit hodnotu. Je deklarována klíčovým slovem `CONST`.

Při deklaraci jakékoli proměnné musíme definovat její datový typ. Základní tři datové typy jsou vyjmenovány v tabulce 2.1. Operátory, které pracují se

Datový typ	Popis
`num`	Číselná hodnota, reprezentuje celá i desetinná čísla (např. 10 nebo 3.14159)
`string`	Textový řetězec (např. ”Toto je text”), maximální délka je 80 znaků
`bool`	Booleovská logická hodnota nabývající hodnot `TRUE` nebo `FALSE`

Tabulka 2.1: Základní datové typy RAPIDu

Operátor(y)	Popis
`+ - * /`	*Numerické operátory* – sčítání, odčítání, násobení, dělení
`= <> > >= < <=`	*Relační operátory* – rovnost, nerovnost, vetší, vetší a rovno, atd.
`+`	Konkatenace řetězců

Tabulka 2.2: Operátory v RAPIDu

Datový typ	Popis
`robtarget`	Pozice v souřadnicovém systému
`speeddata`	Rychlost pohybu
`zonedata`	Zóna (volnost) kolem rohů pohybu
`tooldata`	Specifikace nástroje
`wobjdata`	Definice souřadnicového systému (workobjectu)

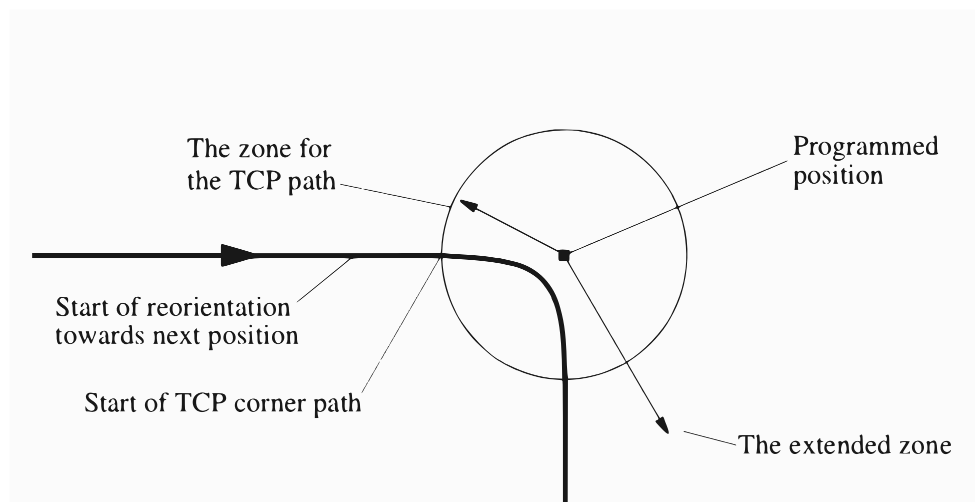
Tabulka 2.3: Vybrané rozšířené datové typy RAPIDu

základními datovými typy, jsou popsány v tabulce 2.2. Několik dalších datových typů, které jsou všechny založeny na zmíněných základních typech, jsou v tabulce 2.3. Příklady deklarací jsou v listingu 2.

Typ `robtarget` definuje cílový bod. Typ se skládá z pozice, rotace, osové konfigurace a externích os (viz listing 2). Osová konfigurace určuje, v jaké pozici bude rameno v daném bodě – často totiž může být v bodě několika způsoby (shora, ze strany, . . .). Pokud je robot například na otočném podstavci nebo na jeřábu, můžeme to vzít v potaz v poli popisující externí osy. V jednoduchých situacích to však nepotřebujeme.

Typ `speeddata` popisuje rychlost pohybu robota nebo externích os. Definice rychlosti se skládá ze tří komponent – rychlost pohybu, rychlost reorientace nástroje a rychlost externí osy. Pro YuMiho existuje několik předem definovaných rychlostí, např.: `v10, v50, v100, v500, ...`, kde číslo značí rychlost pohybu v mm/s. Rychlosti reorientace a externích os jsou u předem definovaných typů konstantní.

Typ `zonedata` specifikuje oblast kolem cílového bodu. Pokud není daný bod koncový (nekončí zde pohyb, ale následuje další *move* instrukce), tak zóna určuje jistou míru volnosti při průchodu skrz tento „rohový“ bod. Viz obrázek 2.8. Platí, že čím větší zóna, tím může být pohyb při průchodu rohovým bodem plynulejší.



Obrázek 2.8: Znázornění zóny [1]

K definici nástroje používáme datový typ `tooldata`. Při definici musíme zohlednit, zda-li je nástroj stacionární nebo jej robot drží. Dále specifikujeme TCP[3] pomocí souřadnic, rotaci pomocí kvaternionu (čtveřice), váhu nástroje a pozici těžiště.

Posledním ze zmíněných netriviálních typů je `wobjdata`. Tento typ popisuje workobject, tedy souřadnicový systém. U něj definujeme souřadnice v tzv. *world coordinates*, rotaci pomocí kvaternionu a zda-li robot workobject drží.

2.1.3.2 Podmínky a cykly

Podmínky i cykly fungují opět podobně jako v jiných jazycích. Příklad použití těchto konstrukcí je v listingu 3.

2.1.3.3 Move instrukce

Move instrukce jsou procedury, pomocí nichž říkáme robotovi, kam a jak se má pohnout. YuMi rozlišuje 3 typy pohybů:

- `MoveL`
- `MoveJ`

[3]Tool Centre Point

```
MODULE MainModule
  VAR bool flag; ! deklarace proměnné
  PERS num length; ! deklarace persistentní proměnné

  ! robtarget (pozice, rotace, konfigurace, externí osy)
  CONST robtarget p1 := [ [600, 500, 225.3],
                          [1, 0, 0, 0],
                          [1, 1, 0, 0],
                          [ 11, 12.3, 9E9, 9E9, 9E9, 9E9] ];
  ! workobject (pozice ve world coordinates, rotace, atd.)
  PERS wobjdata wobj2 :=[ FALSE, TRUE, "", [ [300, 600, 200],
                                             [1, 0, 0 ,0] ],
                                           [ [0, 200, 30],
                                             [1, 0, 0 ,0] ]];
  PROC main()
    flag := TRUE; ! přiřazení do proměnné
    length := 10;
  END PROC
ENDMODULE
```

Listing 2: Použití proměnných

- `MoveC`

Všechny zmíněné instrukce mají společné argumenty popsané v tabulce 2.4. Instrukce `MoveL` provádí lineární pohyb z aktuální pozice do cílového bodu. Hodí se na precizní přesuny. Nevýhodou je, že trajektorie může procházet tzv. *singularitami*, což jsou body, pro které neexistuje žádná konfigurace kloubů. V bodu singularity proto fyzicky nemůže rameno být. Tento problém se řeší buď vedením trajektorie zcela jinou trasou nebo jemným objetím samotné singularity.

Mnohdy ale nepotřebujeme stoprocentně přesný pohyb a tak použijeme instrukci `MoveJ`. Tento pohyb je pro robota nejjednodušší – nemusí počítat pohyb tak, aby jel po přímce, ale pouze nastaví klouby do příslušné cílové konfigurace. Následkem toho je, že rameno může opisovat celkem nepředvídatelnou trajektorii a může do něčeho narazit. Je nutné (experimentálně) zvolit vhodné konfigurace ramen v cílových pozicích, aby byl pohyb „ukázněný“.

Instrukce `MoveC` slouží k pohybu po oblouku, který specifikujeme body.

2.1.3.4 Ovládání SmartGripperu

Prvky SmartGripperu (servo svěrek a přísavky) se ovládají instrukcemi s prefixem `g_`. Než se zavolá jakákoliv z těchto instrukcí, je nutné SmartGripper

```
MODULE MainModule
  VAR string name;
  VAR bool flag;

  PROC main()
    ! příklad podmínky
    IF name = "a" THEN
      ! větev a
    ELSEIF name = "b" THEN
      ! větev b
    ELSE
      ! větev c
    ENDIF

    ! příklad FOR cyklu
    FOR i FROM 1 TO 10 DO
      ! 10x se opakující cyklus
    ENDFOR

    WHILE flag DO
      ! cyklí, dokud je flag=TRUE
    ENDWHILE
  END PROC
ENDMODULE
```

Listing 3: Použití podmínek a cyklů

inicializovat zavoláním instrukce `g_Init`. Rovněž je nutné SmartGripper po spuštění robota zkalibrovat. To uděláme buď přes FlexPendant nebo předáme instrukci `g_Init` volitelný argument `\Calibrate`. V tabulce 2.5 jsou vybrané funkce popsány.

Instrukcím pro sevření/rozevření svěrek lze předávat volitelné argumenty jako např. `\HoldForce`, neboli síla, se kterou se předmět uchopí. U přísavek se volitelně předává hraniční hodnota tlaku vzduchu v kilopascalech. Aby bylo možné přísavky použít, je nutné, aby byl robot připojen ke zdroji stlačeného vzduchu.

2.1.3.5 Synchronizace ramen

Jelikož každé rameno pracuje zcela odděleně s vlastním programem (i programovým čítačem), potřebujeme je obě čas od času synchronizovat. Příkladem budiž situace, kdy chceme, aby ramena nějakým způsobem kooperovala nebo

Argument	Popis
`ToPoint`	Cílový bod datového typu `robtarget`
`Speed`	Rychlost posuvu datového typu `speeddata` (například: `v5, v100, v1000, ...`)
`Zone`	Zóna (volnost) kolem rohů pohybu (datový typ `zonedata`), například: `z10, z50, ...`
`Tool`	Volba nástroje, který se zarovná s cílovým bodem (datový typ `tooldata`)

Tabulka 2.4: Argumenty *move* instrukcí

Instrukce	Popis
`g_Init [\Calibrate]`	Inicializace SmartGripperu s volitelnou kalibrací
`g_GripIn, g_GripOut`	Sevření, resp. rozevření čelistí svěrek
`g_VaccuumOn, g_VaccuumOff`	Aktivace/deaktivace přísavky

Tabulka 2.5: SmartGripper instrukce

když je akce jednoho ramena závislá na dokončení činnosti druhého. Samostatný program, který se vykonává nezávisle na ostatních, se v terminologii RAPIDu nazývá *task*. Zaměříme se na následující dva způsoby synchronizace tasků:

- synchronizace v jednom bodě v kódu (bariéra),
- synchronizovaný pohyb.

Prvním typem je synchronizace pomocí instrukce `WaitSyncTask`. Tato instrukce funguje na podobném principu jako bariéry ve vícevláknovém programování. Prvním argumentem instrukce je tzv. synchronizační identifikátor (datový typ `syncident`). Ten definujeme ve všech úlohách, které chceme synchronizovat, jako persistentní proměnnou se stejným jménem. Druhým argumentem je pole názvů tasků, které se synchronizují. Jednoduchý příklad synchronizace mezi dvěma tasky (`T_ROB_L` a `T_ROB_R`) je v listingu 4.

Pomocí bariéry dosáhneme synchronizace jen v jednom okamžiku na jednom místě v kódu. Pokud však potřebujeme provést nějaký složitější koordinovaný pohyb, museli bychom (pomocí této techniky) použít `WaitSyncTask` před každou *move* instrukcí. Výsledkem takového řešení by byl pouze semikoordinovaný pohyb, jelikož průběh jedné instrukce (rychlost) se nijak nekoordinuje. Řešením tohoto problému je instrukce `SyncMoveOn`, resp. `SyncMoveOff`, pomocí kterých docílíme plně koordinovaného pohybu.

Mezi voláním těchto dvou instrukcí vznikne blok kódu, kde se instrukce vykonávají synchronně napříč všemi tasky. To znamená, že *move* instrukce

T_ROB_L

```
PERS tasks task_list{2} := [
  ["T_ROB_L"], ["T_ROB_R"] ];
VAR syncident sync1;
...
WaitSyncTask sync1, task_list;
! sem vstoupí oba najednou
```

T_ROB_R

```
PERS tasks task_list{2} := [
  ["T_ROB_L"], ["T_ROB_R"] ];
VAR syncident sync1;
...
WaitSyncTask sync1, task_list;
! sem vstoupí oba najednou
```

Listing 4: Synchronizace úloh pomocí `WaitSyncTask`

začínají a končí ve stejný moment. Rychlost pohybu je vždy přizpůsobena nejdéle trvající *move* instrukci. Všem `MoveL` instrukcím, které jsou v synchronizovaném (tzv. *MultiMove*) bloku a mají být vykonány současně, musí být předán argument `\ID` se synchronizačním identifikátorem (číslem) se stejnou hodnotou [8]. Viz příklad v listingu 5.

T_ROB_L

```
PERS tasks task_list{2} := [
  ["T_ROB_L"], ["T_ROB_R"] ];
VAR syncident sync1;
VAR syncident sync2;
VAR robtarget bod_1 := ...;
...
SyncMoveOn sync1, task_list;
MoveL bod_1 \ID:=10, ...;
SyncMoveOff sync2;
```

T_ROB_R

```
PERS tasks task_list{2} := [
  ["T_ROB_L"], ["T_ROB_R"] ];
VAR syncident sync1;
VAR syncident sync2;
VAR robtarget bod_2 := ...;
...
SyncMoveOn sync1, task_list;
MoveL bod_2 \ID:=10, ...;
SyncMoveOff sync2;
```

Listing 5: Příklad koordinovaného pohybu

Další možnosti synchronizace a podrobnější popis jednotlivých synchronizačních instrukcí je v referenčním manuálu k jazyku RAPID [9].

2.1.4 FlexPendant

FlexPendant je ruční ovládací panel připojený k robotovi (viz obr. 2.9). Slouží k provádění mnoha úkonů souvisejícími s ovládáním robota – k spouštění nahraných programů, ručnímu ovládání ramen, upravování RAPID kódu, učení pohybů, atd. [10] V následujících podsekcích se podíváme na vybrané funkce FlexPendantu.

2.1.4.1 Jogging

Ruční ovládání ramen je ve FlexPendantu nazýváno *jogging*. Abychom mohli ramena ručně ovládat, musíme nejprve přepnout FlexPendant do manuálního módu kliknutím na ikonu v pravém dolním rohu a poté na první ikonu shora, kde následně v sekci *Operator Mode* zvolíme *Manual*.

Obrázek 2.9: FlexPendant

V hlavním menu (v levém horním rohu) zvolíme položku *Jogging*. Zde již můžeme zvolit:

1. *mechanickou jednotku* – rameno, které budeme ovládat (`ROB_L`, `ROB_R`),

2. *mód pohybu* – množina kloubů, kterou budeme ovládat, případně celkový mód (lineární pohyb, reorientace, ...),

3. *nástroj*, *workobject* a další.

Samotný pohyb pak v závislosti na zvoleném módu provádíme joystickem na pravé straně FlexPendantu. Pokud má v důsledku ručního posouvání dojít ke kolizi např. s tělem robota nebo druhým ramenem, pak je pohyb automaticky zastaven a objeví se chybová hláška. Ke kolizi však může dojít s objekty, o kterých robot neví (např. s nějakými pracovními předměty). Pokud robot detekuje odpor, také pohyb zastaví.

Za zmínku stojí rovněž resetování ramen do jejich „domovské" pozice. Pokud se nám průběh programu přeruší v nějakém neočekávaném okamžiku, chceme pravděpodobně uvést ramena zpět do výchozí pozice. Nejjednodušším způsobem resetu je manuální pohyb rameny po aktivaci funkce *lead-through* (viz sekce 2.1.4.4). Jakmile můžeme s ramenem volně pohybovat, nastavíme

všechny klouby tak, aby rysky, které jsou na obou stranách každého kloubu, spolu lícovaly.

2.1.4.2 SmartGripper

SmartGripper můžeme ovládat v hlavním menu po zvolení položky *SmartGripper*. Zde si zvolíme, chceme-li ovládat pravý nebo levý gripper. V boxu *Setup* musíme kliknout na *Calibrate* svítí-li u něj červená ikona.

Poté již můžeme pomocí tlačítek `Grip-` a `Grip+` ovládat sevření, resp. rozevření svěrky. Svěrka se svírá tak dlouho, dokud se nedetekuje uchopení předmětu nebo nedojede až na konec.

Potřebujeme-li hýbat se svěrkami ručně, použijeme tlačítka `Jog-` a `Jog+`.

2.1.4.3 Editor RAPIDu

Kód programu v jazyku RAPID lze editovat po otevření položky *Program Editor* v hlavním menu. Můžeme procházet existující tasky (`T_ROB_L`, `T_ROB_R`), jejich moduly a rutiny a přidávat či editovat instrukce.

Editor není příliš uživatelsky přívětivý na vývoj, ale hodí se na drobné úpravy a zejména na lead-through učení, které je popsáno v následující podsekci.

2.1.4.4 Lead-through učení

Při vývoji programu v RobotStudiu a v RAPIDu se můžeme setkat s dojmem, že nás velmi brzdí neustálé vytváření targetů v modelovacím prostředí, kde je nutné zadat přesné souřadnice, rotaci a přiřadit jej do správného workobjectu, konkrétnímu ramenu a nástroji.

FlexPendant nám umožňuje přepnout rameno do tzv. *Lead-Through* módu. Serva v kloubech se v tomto módu uvolní a my můžeme ramenem libovolně ručně pohybovat. Mód se aktivuje v menu *Jogging* pro aktivní rameno v dolní liště tlačítkem *Enable Lead-through*.

Kroky pohybu můžeme vytvářet tak, že umístíme rameno do kýžené pozice a v editoru RAPIDu přidáme *move* instrukci, která se automaticky vygeneruje s aktuální pozicí ramena jako cílovým bodem.

2.1.4.5 Spuštění programu

Nahraný program si můžeme prohlédnout buď v editoru RAPIDu (viz sekce výše) nebo v produkčním okně (v menu položka *Production Window*). Zde máme přehled o aktuální pozici programového čítače a o stavu programu (běžící nebo zastaven).

Před spuštěním si resetujeme programový čítač tlačítkem *PP to Main*. Následně program spustíme hardwarovým tlačítkem „play“ pod joystickem.

Tlačítky „next“ a „previous“ můžeme program krokovat. Tlačítkem „stop“ program zastavíme.

Běh programu je ovlivněn módem, ve kterém FlexPendant je (manuální nebo automatický). Máme-li zvolen manuální mód, jsou všechny *move* instrukce prováděny nízkou rychlostí nezávisle na argumentech typu `speeddata`, která jim předáváme. Tento mód proto slouží k testování a vývoji, jelikož můžeme pohyb pozorovat a případně bezpečně zastavit, je-li riziko, že nastane kolize.

Po odladění programu v manuálním módu jej můžeme spustit v automatickém módu, který již respektuje předávané rychlosti *move* instrukcím.

2.2 Holografické brýle HoloLens

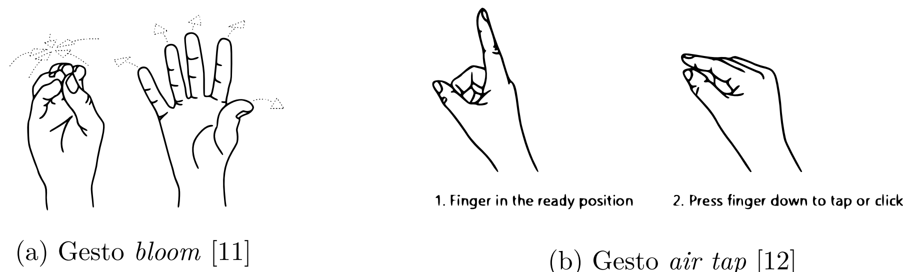
Holografické brýle *HoloLens* (obr. 2.10) od Microsoftu je zařízení pro tzv. rozšířenou realitu. Na rozdíl od virtuální reality máme s brýlemi HoloLens neustále přehled o reálném světě. Brýle mají průhledné sklo, na které je promítán obraz, čímž vytváří iluzi virtuálních objektů umístěných v reálném prostoru.

Obrázek 2.10: Holografické brýle HoloLens

2.2.1 Základní charakteristika

Brýle jsou vybaveny několika senzory a kamerami, které jim umožňují porozumět prostředí kolem. Mezi tyto prvky patří např.: 4 kamery pro rozpoznávání prostředí, 1 hloubková kamera, 4 mikrofóny, světelný sensor a další.

Z výstupů těchto senzorů a kamer jsou HoloLens schopny detekovat gesta, pohled a hlas člověka. Mezi základní gesta patří *bloom* a *air tap* (viz obr. 2.11). První gesto slouží k otevření hlavního menu nebo k ukončení aktuálně běžící aplikace. Gestem *air tap* interagujeme s tlačítky a podobnými prvky (obdoba levého tlačítka myši).



(a) Gesto *bloom* [11]

(b) Gesto *air tap* [12]

Obrázek 2.11: Základní gesta HoloLens

V brýlích je 32b Intel procesor s přídavnou holografickou procesní jednotkou (HPU). Operačním systémem je Windows 10 [13].

Aplikace pro holografické brýle HoloLens lze rozdělit na dva typy:

- 2D UWP[4] aplikace,
- 3D UWP aplikace.

2D UWP aplikace lze programovat v jazyce `C#`, `C++` a `Javascript`, k čemuž Microsoft doporučuje prostředí Visual Studio. Konceptem UWP je napsat jednu aplikaci, která poběží na všech zařízeních se systémem Windows 10 (PC, Xbox, HoloLens, Windows Phone, ...). V prostředí holografických brýlí se taková aplikace zobrazuje v plovoucím 2D okně. Tímto způsobem jsou v brýlích implementovány aplikace typu webový prohlížeč nebo průzkumník souborů.

3D UWP aplikace s prostorovými hologramy se vyvíjí v prostředí Unity a ve Visual Studiu. V Unity vytvoříme projekt, ve kterém je vymodelována 3D scéna s hologramy. Logiku implementujeme standardním způsobem skriptování v Unity – v jazyce C♯. Projekt pak exportujeme do Visual Studia, přičemž se provede transformace kódu dle zvoleného skriptovacího backendu: *IL2CPP* nebo *.NET Scripting* (viz dále). Ve Visual Studiu exportovaný projekt zkompilujeme a nahrajeme přímo do fyzických brýlí, nebo jej spustíme v emulátoru HoloLens.

Pro účely této práce budeme od začátku uvažovat 3D aplikaci, proto popíšeme proces vývoje pouze takovéto aplikace v následujících sekcích.

[4] Universal Windows Platform

2.2.2 Unity

Unity je multi-platformní vývojový nástroj a framework pro tvorbu 3D aplikací. Nejvíce je dnes používán k vývoji her, ale je také používán k vývoji aplikací pro rozšířenou realitu.

2.2.2.1 Základní prvky Unity

Po otevření Unity uvidíme 4 části, na které je prostředí rozděleno [14]:

- panel s hierarchií objektů,
- scéna,
- inspektor objektu,
- panel s adresářovou strukturou.

V panelu hierarchie máme zobrazeny objekty aktuálně otevřené scény. Objekt v Unity se nazývá `GameObject` – z této třídy dědí všechny objekty ve scéně. GameObject má jméno a může být aktivní nebo deaktivovaný. GameObject může být například kostka, model židle, zdroj světla, kamera, atd.

Aby GameObject něco dělal, přidáváme na něj tzv. *komponenty.* Unity poskytuje celou řadu komponent – např. *Transform* (komponenta definující transformaci objektu ve scéně), *BoxCollider* (kolizní „obal“ objektu pro detekci kolizí) nebo *Script* (umožňuje nám přidat na objekt skript, ve kterém definujeme vlastní logiku). Po kliknutí na příslušný objekt v panelu hierarchie můžeme komponenty přidávat v panelu inspektoru objektu.

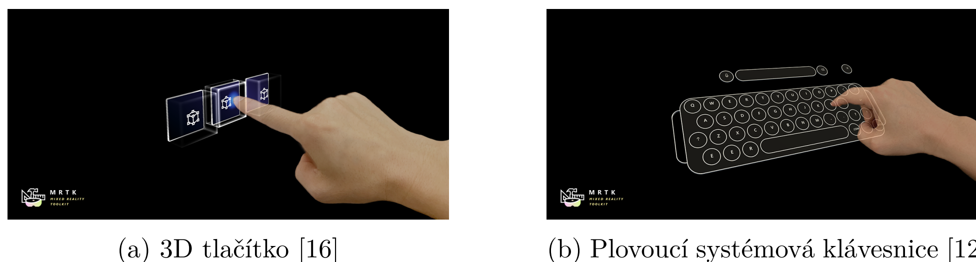
Mnoho typů objektů s předdefinovanými komponentami můžeme vyvářet v horní liště v menu *GameObject* (např. *Cube*, *Plane*, *TileMap*, . . .). Objekt s nakonfigurovanými komponentami si můžeme uložit jako tzv. *prefab*, který můžeme později recyklovat nebo z něj vytvářet instance objektů dynamicky.

Scénu vidíme v prostřední části editoru. V ní se pohybujeme kolečkem a prostředním tlačítkem myši a rozhlížíme se pravým tlačítkem myši. Aplikaci můžeme v Unity spustit tlačítkem *play* nad scénou.

2.2.2.2 Mixed Reality Toolkit

Mixed Reality Toolkit (zkráceně MRTK) [15] je projekt vyvíjený Microsoftem, který poskytuje základní komponenty pro Unity na vývoj aplikací rozšířené reality. Jejím cílem je usnadnit a urychlit vývoj těchto aplikací. Máme možnost pomocí ní vyvíjet pro platformy HoloLens, HoloLens 2, HTC Vive, Oculus Rift a další.

Součástí projektu je základní knihovní balíček *Foundation* ve formátu `.unitypackage`, který můžeme importovat do našeho projektu. Dále máme



(a) 3D tlačítko [16] (b) Plovoucí systémová klávesnice [12]

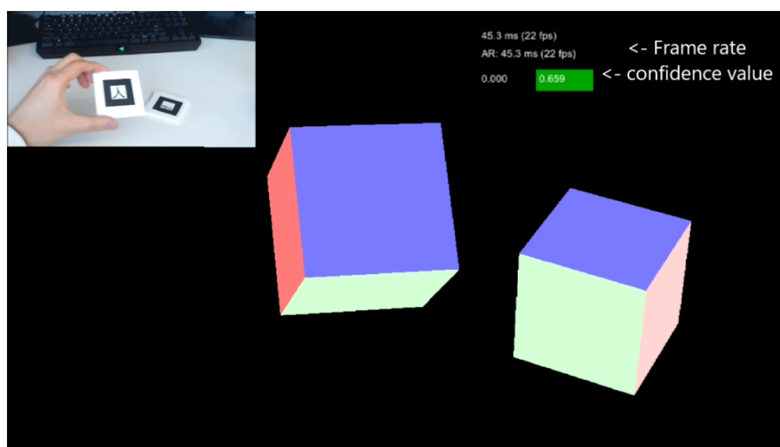
Obrázek 2.12: Příklad komponent z MRTK

k dispozici rozsáhlou dokumentaci použití jednotlivých komponent včetně balíčku *Examples* s demonstračními příklady scén.

V MRTK máme k dispozici například komponenty: *Button* (tlačítko), *System Keyboard* (plovoucí klávesnice), *Interactable* (množina „formulářových“ prvků – checkbox, radio button, . . .) a další. Viz obrázek 2.12. Výhoda této knihovny je skutečnost, že je vyvíjena Microsoftem, který ji přímo optimalizuje na svoji platformu HoloLens. Dle Microsoftu se jedná o jedinou doporučenou knihovnu pro vývoj 3D aplikací pro HoloLens [17].

2.2.2.3 HoloLens ARToolkit

HoloLens ARToolkit je experimentální knihovna, která má za cíl integrovat knihovnu *ARToolkit* a platformu HoloLens [18]. Knihovna *ARToolkit* vznikla již v roce 1999 a je určená k analýze obrazu, resp. k rozpoznávání a real-time „trackování“ určitých značek [2].



Obrázek 2.13: Příklad použití ARToolkit v HoloLens [2]

Pomocí knihovny můžeme z kamery na přední straně brýlí číst video a dle něho v prostoru lokalizovat *marker* (značku). Výsledkem může být situace znázorněná na obrázku 2.13. Tato knihovna by pro nás v této práci mohla

být užitečná – mohli bychom implementovat inteligentní rozpoznávání scény podle značek. Nevýhodou je, že se jedná čistě o experimentální open-source projekt, vyvíjený jedním člověkem.

2.2.3 Visual Studio

Visual Studio slouží pro finální kompilaci vyexportovaného projektu z Unity a nahrání do fyzického zařízení nebo do emulátoru. Projekt můžeme z Unity exportovat s jedním z následujících typů skriptovacích backendů:

- `.NET Scripting`,
- `IL2CPP`.

Skriptovací backend nám udává, v jakém formátu bude projekt z Unity exportován a ve výsledku i zkompilován ve Visual Studiu. Oba zmíněné způsoby mají své výhody i nevýhody. Backend *.NET Scripting* kompiluje skripty *Just In Time* (JiT), což znamená při runtime. Čas kompilace je kratší, ale výkon je nižší.

Naproti tomu *IL2CPP* kompiluje způsobem *Ahead Of Time* (AoT), tedy předem. *IL2CPP* vygeneruje kód v C++, který je poté přeložen do nativního kódu ve Visual Studiu. Výhodou je optimálnější kód na úkor delšího času kompilace [19].

Pro UWP aplikace na HoloLens můžeme použít jakýkoliv ze zmíněných skriptovacích backendů, nicméně Unity oznámilo, že od nových verzí považuje backend *.NET Scripting* za zastaralý a doporučuje používat *IL2CPP* [20].

2.3 Možnosti komunikace mezi YuMi a HoloLens

Zásadním problémem v této práci je vymyslet způsob, jakým bude robot komunikovat s holografickými brýlemi. Komunikace musí být plně obousměrná, tedy brýle budou odesílat data/příkazy robotovi a obráceně. V této sekci rozebereme 3 způsoby, jak lze komunikovat s robotem YuMi a zjistíme, který z nich lze implementovat v brýlích HoloLens.

Pozitivní je, že **při testování** komunikace **lze využít simulátor** v RobotStudiu, pokud v něm máme importovaný offline kontrolér robota.

2.3.1 PC SDK

PC SDK je dedikované API ke kontrolérům v robotech přímo od ABB [21]. Pomocí něj lze skenovat síť pro dostupné kontroléry a poté přistupovat k jejich funkcím (spuštění tasků v `RAPID`u, I/O systém, přístup k mechanickým jednotkám, apod.).

Pomocí tohoto přístupu máme k dispozici zřejmě nejrozsáhlejší možnosti ovládání a monitorování robota. API je zkompilované v dynamicky linkované

knihovně pod *.NET Framework*. Jakýkoliv program, schopný použít závislost v *.NET Frameworku*, může toto API použít.

V tom tkví základní problém tohoto řešení. Aplikace UWP v HoloLens není zkompilována pro *.NET Framework*, ale pro *.NET Core*. Dynamickou knihovnu PC SDK k ní tudíž nelze nalinkovat. Respektive existuje způsob s nejistým výsledkem, jak lze knihovnu v *.NET Framework* přeportovat na *.NET Core*, ale není toho možné dosáhnout žádným oficiálním nástrojem a nemáme žádnou záruku správné funkčnosti [22]. S těmito znalostmi se jeví jako jediné možné řešení použití nějakého prostředního zařízení (proxy), na kterém poběží aplikace s PC SDK a HoloLens s ní bude komunikovat nějakým běžným způsobem (např. přes sockety nebo HTTP).

2.3.2 TCP sockety

Další možností je využít klasickou síťovou komunikaci na úrovni socketů. Jazyk `RAPID` podporuje funkce na tvorbu socketů s podobnou sémantikou jako například v jazyce C (`SocketCreate`, `SocketSend`, `SocketBind`, atd.).

Výhodou tohoto přístupu je vysoká volnost – lze posílat jakákoliv data a implementovat jakoukoliv funkcionalitu. Zcela zásadní výhodou je také fakt, že ke komunikaci není třeba žádné další zařízení. Jak v HoloLens, tak v `RAPID`u lze používat standardní síťové sockety. Stačí jen, aby oba byli připojeni ke stejné síti.

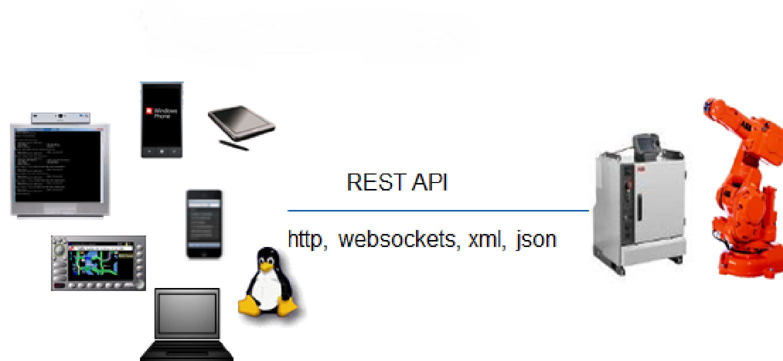
Nevýhodou je značná pracnost tohoto řešení – veškerá komunikace a protokol se musí implementovat (relativně nízkoúrovňově) na obou stranách.

2.3.3 Robot Web Services

Poslední možností je použít webové REST API, které je poskytováno samotným kontrolérem. Když kontrolér běží (nebo je načten virtuálně v offline režimu v RobotStudiu) jsou na jeho síťové adrese na portu 80 k dispozici tzv. *Robot Web Services* [23].

Pomocí tohoto REST API lze přistupovat podobným způsobem ke kontroléru robota jako při použití PC SDK. Přes HTTP metody `GET`, `POST`, `PUT` a `DELETE` můžeme přistupovat např. k taskům v `RAPID`u, k filesystému v kontroléru, k uživatelským profilům, I/O, atd.

Zajímavou a užitečnou součástí API jsou tzv. *subscriptions*. Tento mechanismus nám umožňuje asynchronně poslouchat změny na daném zdroji (např. na nějakém I/O portu). Subscriptions jsou implementovány přes *websockets*, což je standard definovaný v RFC 6455 [24]. Websocketem otevřeme samostatné připojení, které zůstává stále otevřené a chodí nám přes něj asynchronně události o změnách. Bez websocketů by se podobná funkcionalita musela implementovat pollováním daného endpointu REST API, což není příliš efektivní.



Obrázek 2.14: FlexPendant [3]

Výhodou této metody je opět žádná nutnost prostředního zařízení. Komunikace probíhá přes standardní HTTP, což z HoloLens můžeme používat přímo. Rozhraní je relativně jednoduché a dobře popsané v manuálu. Nevýhodou jsou určitá omezení oproti PC SDK, které je plnohodnotnější.

Kapitola 3

Návrh řešení

V této kapitole se podíváme na volbu technologií na základě předchozí analýzy a dále na návrh implementace demonstrační aplikace. Aplikaci navrhneme za účelem demonstrace komunikace mezi brýlemi HoloLens a robotem YuMi. Budeme postupovat podle základního diagramu propojení na obrázku 3.1.

Obrázek 3.1: Základní diagram systému [4] [5]

3.1 Volba technologií

V této sekci si odůvodníme výběr knihovny *Mixed Reality Toolkit*. Dále budeme argumentovat pro zvolení metody komunikace přes síťové sockety.

3.1.1 Knihovna Mixed Reality Toolkit

Jako hlavní framework pro práci s holografickými brýlemi zvolíme knihovnu *Mixed Reality Toolkit v2* (MRTK v2), která je vyvíjena Microsoftem pod licencí MIT jako otevřený software [15]. Pro vývoj zvolíme verzi *MRTK v2 RC1* (v čase psaní této práce se jednalo o nejnovější verzi).

Tuto knihovnu zvolíme z toho důvodu, že se jedná v podstatě o jediný exemplář kompletního frameworku pro vývoj 3D aplikací pro HoloLens. Další pozitivum je, že se jedná o oficiální knihovnu přímo od Microsoftu, který ji přímo doporučuje ve své dokumentaci [17]. Zcela největší výhodou verze *RC1* (oproti starším) je možnost simulovat běh programu přímo v editoru Unity. Z toho důvodu odpadne velmi zdržující proces neustálé kompilace Unity projektu, exportování do Visual Studia, další kompilace a nahrávání do HoloLens. Většinu celé aplikace tak budeme moci odladit přímo v editoru a můžeme věnovat vyšší úsilí například uživatelskému rozhraní.

Alternativou (nikoliv však rovnocennou) by byla knihovna *HoloLensARToolkit*, která poskytuje integraci knihovny *ARToolkit* s platformou HoloLens, respektive obecně s UWP aplikacemi [2]. Knihovna ovšem poskytuje odlišné funkce než *Mixed Reality Toolkit* (strojové vidění, analýza obrazu), proto je nedostačující pro kompletní vývoj. Je možné ji však použít v budoucí práci jako pomocnou knihovnu pro dynamické umisťování hologramů na určité pozice v prostoru, které jsou dány strojově čitelnými značkami.

3.1.2 Způsob komunikace

Jako komunikační metodu zvolíme protokol TCP, který implementujeme přes standardní síťové sockety. Důvod této volby je dán předpokladem, že lze tuto technologii použít na obou zařízeních (YuMi i HoloLens). Dalším důvodem je skutečnost, že ke komunikaci není třeba žádného prostředního prvku – robot komunikuje přímo s brýlemi.

Z alternativních způsobů se nám ještě nabízelo *PC SDK* (viz 2.3.1) a *Robot Web Services* (viz 2.3.3). První možnost by byla za jiných okolností zřejmě nejvhodnější volbou, ale skutečnost, že je tato knihovna v *.NET Frameworku* ji činí nepoužitelnou na platformě HoloLens (resp. UWP), která akceptuje závislosti v *.NET Core*. Portování knihovny mezi těmito verzemi *.NET* by bylo zbytečně složité, zdržující a s nejistým výsledkem.

Robot Web Services je příliš omezující a neflexibilní řešení. Implementovat jednoduchou a obousměrnou komunikaci pomocí standardního modelu

Request-Response přes HTTP + asynchronní události přes websockety je za daných okolností složitější, než totéž pomocí obyčejných TCP socketů.

3.2 Demonstrační aplikace

V rámci demonstrační aplikace bude robot YuMi hrát paličkou na dřevěný xylofon. Robot bude po síti dostávat jednoduché příkazy (např. typu „zahraj notu F") a zároveň bude odesílat souřadnice, kam se bude pohybovat. Pro jednoduchost demonstrace bude YuMi používat jen své **pravé** rameno.

Uživatel s holografickými brýlemi bude mít před očima rozhraní, pomocí kterého bude moci navázat síťové spojení s robotem. Po spojení bude moci klikat na tlačítka, kterými robotovi sdělí, na kterou klávesu xylofonu má zahrát. Rovněž bude dostávat odezvu o pohybu robota na miniatuře pracovní plochy (dále nazývána jako *virtuální plocha*). Nad touto plochou bude indikátor, který se bude pohybovat na základě přijatých souřadnic od robota. Při pohybu robota se kolem virtuální plochy zobrazí průhledná bariéra.

Propojení obou zařízení je znázorněno na diagramu 3.1. Do lokální sítě připojíme robota přes jeho WAN port. Lokální síť musí být nakonfigurována tak, aby DHCP přidělilo robotovi IP adresu. Brýle HoloLens připojíme do této sítě bezdrátově.

Návrh demonstrační aplikace jako celku lze rozdělit na tři části:

- návrh komunikačního protokolu,
- návrh řízení robota,
- návrh aplikace pro HoloLens.

3.2.1 Komunikační protokol

Spojení bude navázáno přes TCP socket jako standardní *client-server* spojení. Serverový socket vytvoříme na straně robota a klientský socket na straně HoloLens. Při komunikaci si budou obě strany vyměňovat zprávy v jednoduchém textovém formátu v kódování UTF-8. Jednotlivé zprávy oddělíme znakem nového řádku (`\n`). Přehled všech možných zpráv je v tabulce 3.1.

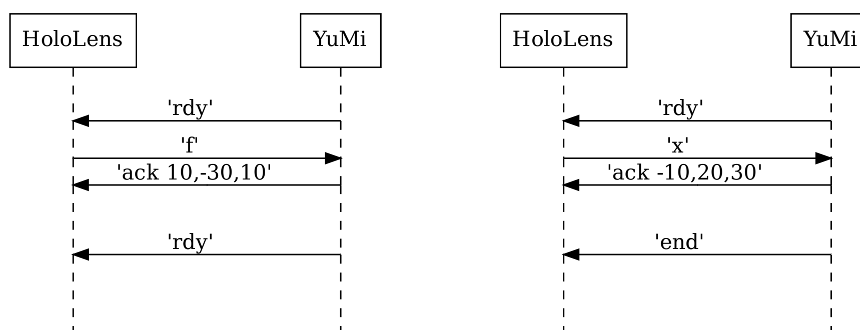
Obsah zprávy	Odesílatel	Popis
`rdy`	YuMi	Připraven k příjmu příkazu
`ack <pos>`	YuMi	Potvrzení příkazu s pozicí
`end`	YuMi	Oznámení před koncem spojení
`c, d, e, f, g, a, b`	HoloLens	Žádost o zahrání příslušné noty
`x`	HoloLens	Žádost o ukončení spojení

Tabulka 3.1: Zprávy, posílané mezi robotem a brýlemi

Po spojení oznámí robot, že je připraven pomocí zprávy „`rdy`“ a bude čekat na příkazy od HoloLens. Z brýlí budou poté chodit jednoznakové zprávy s písmenem příslušejícím klávese, kterou má robot na xylofon zahrát (např. „`f`“).

Po přijetí klávesy odpoví robot zprávou „ack <pos>“, kde <pos> je pozice v souřadnicovém systému robota (3 celá čísla oddělena čárkou). Tato pozice je souřadnice, kam se bude robot pohybovat. Jakmile je robot připraven přijímat další příkaz, odešle opět zprávu „`rdy`“. Příklad této komunikace je znázorněn na sekvenčním diagramu 3.2a.

Před ukončením spojení odešle HoloLens zprávu „`x`“, kterou požádá robota o uzavření socketu. Robot odpoví opět zprávou „ack <pos>“. Jakmile vykoná veškeré procedury nutné před ukončením svého programu, odešle na závěr zprávu „`end`“, čímž signalizuje brýlím ukončení své činnosti a uzavře socket. Příklad komunikace při ukončování spojení je znázorněn na sekvenčním diagramu 3.2b.



(a) Zahrání noty (b) Ukončení spojení

Obrázek 3.2: Příklad komunikace

3.2.2 Řízení robota

Řízení robota naprogramujeme za pomoci FlexPendantu a RobotStudia. Ve FlexPendantu „naučíme“ robota jednotlivým pohybovým pozicím přes *lead-through* učení (viz sekce 2.1.4.4). Tato metoda je v tomto případě nejrychlejší a nejjednodušší na rozdíl od ručního vytváření pozic v RobotStudiu.

V RobotStudiu následně vytvoříme kód v jazyku RAPID a zorganizujeme jej do následujících modulů:

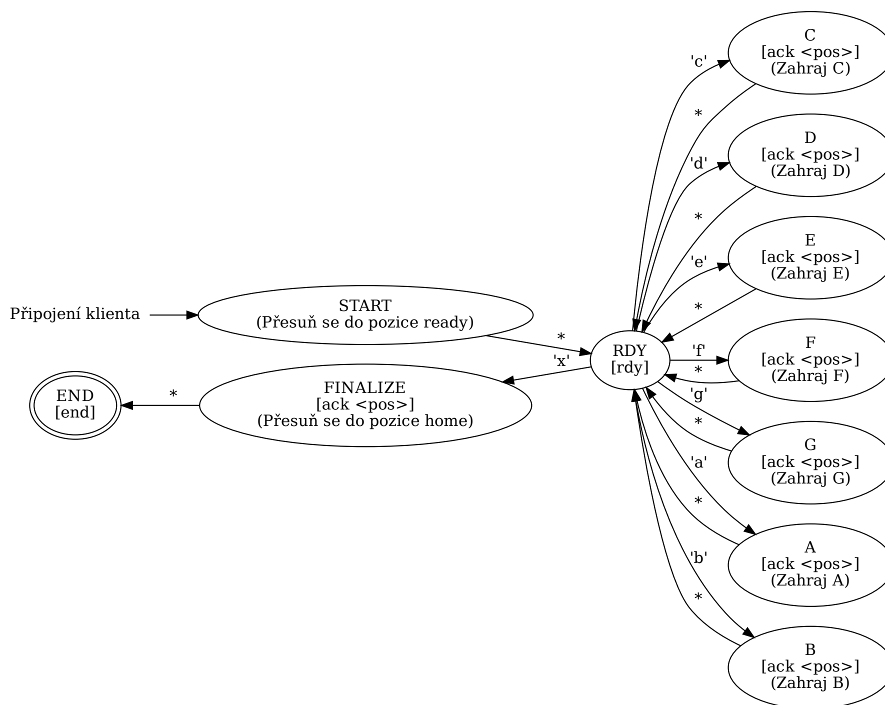
- **hlavní modul** – vstupní modul programu,
- **síťový modul** – implementace síťových funkcí,

- **modul s konečným automatem** – implementace řídícího konečného automatu,
- **pohybový modul** – implementace pohybových procedur.

3.2.2.1 Řízení konečným automatem

Chování robota budeme řídit konečným automatem o 11 stavech znázorněným na diagramu 3.3. Počáteční stav je `START` a koncový `END`. Pod názvem stavu může být navíc nepovinná *odezva* (v hranatých závorkách) a *akce* (v kulatých závorkách).

Po přechodu do nějakého stavu je nejprve připojeným brýlím odeslána *odezva* (pokud existuje). Pokud pak existuje i *akce*, tak je vykonána a po jejím dokončení se provede přechod do stavu po hraně označené symbolem `*`.



Obrázek 3.3: Konečný automat řízení robota YuMi

Do počátečního stavu `START` se automat přesune po úspěšném navázání spojení s HoloLens. V této chvíli přesune YuMi své pravé rameno s uchopenou paličkou z pozice *home* do pozice *ready*. Pozice *home* je výchozí umístění ramena tak, že rysky na všech jeho kloubech spolu lícují (viz sekce 2.1.4.1).

V pozici *ready* je rameno s paličkou ve vhodné vzdálenosti vpravo od xylofonu. Po dokončení pohybu se automat přesune do stavu `RDY`.

Při přechodu do stavu `RDY` odešle YuMi *odezvu* `rdy` a poté čeká na zprávu ze socketu. Přijde-li znak `a-g`, přesune se automat do jeho příslušícího stavu. V tomto stavu odešle *odezvu* `ack <pos>`, kde `<pos>` je pozice přímo nad danou klávesou na xylofonu. Následně zahraje paličkou na klávesu a přejde zpět do stavu `RDY`.

Přijde-li znak `x`, přejde automat do stavu `FINALIZE`. Zde se odešle *odezva* `ack <pos>` s pozicí *home*. Po dokončení pohybu se přejde do koncového stavu `END` a program se ukončí.

3.2.2.2 Síťový a pohybový modul

Síťový modul bude sloužit k separaci síťových funkcí. Vytvoříme zde funkce pro spuštění serverového socketu, přijmutí zprávy, odeslání zprávy, apod.

V pohybovém modulu budou funkce pro pohyby – např. *zahraj notu x*, *přesuň se do pozice ready*, apod. V tomto modulu rovněž zaregistrujeme pomocí *lead-through* učení jednotlivé targety (pozice paličky nad klávesami, pozice úhozu, ready pozice, atd.).

3.2.3 Aplikace pro HoloLens

Aplikaci pro HoloLens vytvoříme jako 3D UWP aplikaci. Jako základní framework použijeme knihovnu *MRTK v2 RC1* [15].

3.2.3.1 3D scéna

V 3D scéně v brýlích vytvoříme následující komponenty:

- panel s prvky pro připojení k robotovi,
- 7 tlačítek, každé příslušející jedné klávese na xylofonu,
- virtuální plocha s pohyblivým indikátorem pozice,
- debugovací panel.

Přes panel s prvky pro připojení k robotovi budeme zadávat IP adresu robota pomocí systémové klávesnice. Na panel dále umístíme tlačítko pro zahájení připojení. Panel budeme moci zobrazit a skrýt pomocí samostatného tlačítka ve scéně.

Ve scéně vytvoříme 7 tlačítek (každé pro jednu klávesu na xylofonu). Po jejich stisknutí v připojeném stavu odešleme socketem příslušný znak `a-g`. V nepřipojeném stavu neprovedeme nic.

V popředí scény vytvoříme čtvercovou plochu, která bude reprezentovat robotovo pracoviště. Nad plochu umístíme indikátor ve tvaru kvádru, kterým

budeme pohybovat na základě přijatých souřadnic v odezvě `ack <pos>`. Kolem plochy vytvoříme transparentní kvádr jako bariéru, kterou budeme zobrazovat pokaždé, když robot nebude ve stavu `RDY` (bude se pohybovat).

Poslední součástí scény bude panel s debugovacím logem, na který budeme v průběhu vývoje zapisovat např. události o kliknutí na tlačítko ve scéně, přijetí zprávy přes socket, apod.

3.2.3.2 Transformace souřadnic

Pro zobrazení indikátoru, kam se rameno robota bude pohybovat, musíme provést transformaci přijatých souřadnic z báze robota na bázi virtuální plochy ve scéně aplikace v HoloLens. Tato plocha má dané rozměry dvourozměrným vektorem $\vec{P}$, který reprezentuje její délku a šířku.

Výsledné souřadnice indikátoru jsou dány vektorem $\vec{o}$, který je lokální vzhledem k ploše – nezáleží proto, kde je plocha ve scéně umístěna globálně. Vektor je omezen podmínkami $o_x \in [-\frac{P_x}{2}, \frac{P_x}{2}]$ a $o_y \in [-\frac{P_y}{2}, \frac{P_y}{2}]$. Umístíme-li tedy indikátor na souřadnice $(0, 0)$, objeví se přímo nad středem virtuální plochy.

Nechť $\vec{c}$ je dvourozměrný vektor, který reprezentuje přijatou souřadnici z robota, která je v milimetrech od jeho počátku souřadnic. Dále nechť $\vec{m}$ (resp. $\vec{M}$), je dvourozměrný vektor minimálních (resp. maximálních) hodnot těchto souřadnic. Například v m_x je minimální možná hodnota souřadnice na ose x, v M_x maximální možná hodnota této souřadnice.

Pro výslednou transformaci si nejprve přijatý souřadnicový vektor $\vec{c}$ normalizujeme do intervalu $[0, 1]$ a výsledek přiřadíme do vektoru $\vec{n}$:

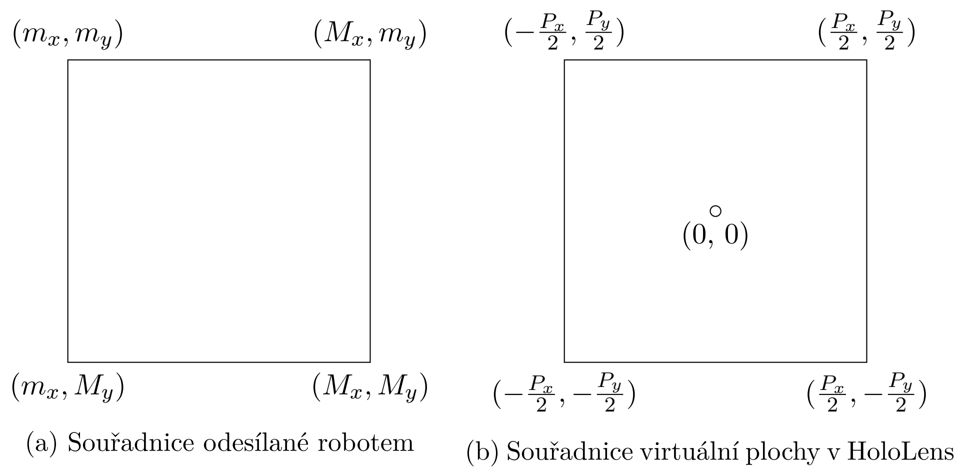
$$\vec{n} = \begin{pmatrix} \frac{c_x - m_x}{M_x - m_x} \\ \frac{c_y - m_y}{M_y - m_y} \end{pmatrix}$$

S normalizovanými vektorovými souřadnicemi $\vec{n}$ nyní provedeme samotnou transformaci tak, aby souřadnice v bodě $(0, 0)$ byla v levém horním rohu virtuální plochy a nikoliv ve středu.

$$o_x = (P_x * n_x) - \frac{P_x}{2},$$

$$o_y = (-P_y * n_y) + \frac{P_y}{2}$$

Výsledkem této transformace bude, že pokud nám robot pošle souřadnici (m_x, m_y), kde vektor $\vec{m}$ je definován stejně jako výše, objeví se indikátor nad levým horním rohem virtuální plochy (bude namapován na souřadnice $(-\frac{P_x}{2}, \frac{P_y}{2})$). Znázornění souřadnic odesílaných robotem je na diagramu 3.4a a systém souřadnic virtuální plochy v HoloLens je znázorněn na diagramu 3.4b.



(a) Souřadnice odesílané robotem

(b) Souřadnice virtuální plochy v HoloLens

Obrázek 3.4: Souřadnicové systémy

Kapitola 4

Implementace

V této kapitole si popíšeme postup implementace demonstrační aplikace popsané v předchozí kapitole.

4.1 YuMi

V této sekci si popíšeme přípravu pracoviště a vlastní implementaci řízení robota, kterou jsme navrhli v sekci 3.2.2.

4.1.1 Příprava robota a pracoviště

Před spuštěním robota si nejprve připravíme pracoviště. Jelikož se v průběhu implementace této práce nenachází v laboratoři pracovní plocha přímo určená pro YuMiho, použijeme provizorní prostředky. Robota umístíme na zem, aby měl kolem sebe dostatek místa a nehrozila kolize s okolím. V reálnější situaci bude robot fixován ke stolní desce.

Jako pracovní „stůl" robota nám poslouží obyčejná papírová krabice. Mohli bychom pracovat i přímo na podlaze, ale měkký materiál krabice nám poskytne malou rezervu, pokud by například hrozila kolize ramene s podlahou. Na krabici umístíme dřevěný xylofon.

Na servisní ethernetový port robota následně připojíme PC, na kterém běží RobotStudio (viz sekce 4.1.2). Robota poté spustíme černým otočným vypínačem na jeho boční straně. Po nabootování (které zabere zhruba minutu), nejprve zkalibrujeme SmartGrippery v menu *SmartGripper*. V okně ovládání pravého SmartGripperu klikneme na tlačítko `Grip+` a vložíme jednu dřevěnou paličku mezi svěrky. Výsledek bude vypadat jako na obrázku 4.1.

4.1.2 Nastavení RobotStudia

Po prvotním spuštění RobotStudia musíme nejprve:

1. nainstalovat balíky *RobotWare* a *SmartGripper*,

Obrázek 4.1: YuMi s xylofonem

2. vytvořit projekt s prázdnou stanicí,
3. přidat robota YuMi do stanice,
4. přidat oba SmartGrippery a napojit je na ramena YuMiho,
5. importovat kontrolér z fyzického robota a přejít do offline módu.

Tyto kroky jsou podrobněji rozepsané v následujících odstavcích. Alternativní způsob založení projektu je import vyexportovaného `.rspag` balíku, který je v příloze této práce. Importujeme jej v záložce *File→Share→Unpack And Work*. **Důležitý předpoklad:** veškeré uživatelské soubory použité RobotStudiem (ať už nainstalované balíky, projekt, stanice, apod.) musí být v cestě, která **neobsahuje mezeru**. Proto tyto soubory umístíme přímo do kořenové úrovně disku C.

Jako první nainstalujeme do RobotStudia balíky *RobotWare* a *SmartGripper* ještě před založením samotného projektu. To uděláme na záložce Add-Ins, kde tyto balíky vyhledáme a nainstalujeme přes funkci RobotApps. Nainstalované balíky jsou poté zobrazeny v levém podokně v *Installed Packages*.

Funkcí *File→New→Create with Empty Station* následně vytvoříme projekt s prázdnou stanicí (virtuální pracoviště s robotem). Do stanice poté vložíme robota přes *Home→ABB Library→IRB 14000*. Uvidíme, že robot zatím nemá na koncích ramen žádný nástroj. Použijeme proto dvakrát funkci *Home→Import Library→Browse for Library*, kde najdeme SmartGripper a zvolíme příslušnou formu dle reálné konfigurace robota (s jednou/dvěma přísavkami, s/bez kamery). Oba nástroje se objeví v levém podokně Layout, odkud je připojíme k ramenům přes pravé tlačítko myši a funkci *Attach to*.

4.1.2.1 Vytvoření kontroléru a relace

Abychom nemuseli ručně vytvářet celý kontrolér, naimportujeme jej z fyzického robota. Připojíme počítač s RobotStudiem síťovým kabelem na servisní port robota, který je umístěn vedle WAN portu. Jakmile robot nabootuje, připojíme se ke kontroléru přes *Controller→Add Controller→One Click Connect*. Tímto jsme připojili fyzický kontrolér v online módu. Abychom však mohli pracovat i bez dostupnosti robota a testovat chování pouze v simulátoru, musíme vytvořit kopii tohoto kontroléru. Toho docílíme funkci *Controller→Go Offline* máme-li v levém panelu vybraný připojený kontrolér.

Jakmile máme v RobotStudiu oba kontroléry (offline i online), vytvoříme mezi nimi synchronizační relaci. Na kartě *Controller* zvolíme *Create Relation*. Vzniklou relaci budeme následně používat, budeme-li chtít synchronizovat data a kód z virtuálního do fyzického kontroléru (a naopak).

4.1.3 Implementace modulů

V této sekci popíšeme implementaci jednotlivých modulů z návrhu. Veškeré ukázky kódu v této sekci jsou v jazyku RAPID (viz 2.1.3). Soubory s kompletním zdrojovým kódem jsou na přiloženém médiu (viz příloha B).

4.1.3.1 Síťový modul

Síťový modul nazveme *NetModule*. Na začátku modulu budou deklarace konstant a proměnných (objekt serverového a klientského socketu, buffer pro příjem zprávy, atd.) Dále bude modul obsahovat následující procedury:

- `StartSocketServer`,
- `ReceiveCharacter`,
- `SendMessage(string msg)`,
- `SendACK(robtarget target_pos)`,
- `CloseSocket`.

Procedura `StartSocketServer` (listing 6) slouží k vytvoření socketu na lokální adrese robota a k počkání na připojení klienta. K vytvoření serverového socketu využijeme systémové funkce `SocketCreate` (vytvoření socketu), dále `SocketBind` (navázání na IP adresu a port) a `SocketListen` (přepnutí do monitorovacího módu). Následně zavoláme funkci `SocketAccept`, která čeká na spojení a vytvoří klientský socket.

```
1 PROC StartSocketServer()
2   SocketCreate net_server; ! create the server socket
3   SocketBind net_server, "10.10.48.194", 35001; ! bind to local address
4   SocketListen net_server; ! start listening for incoming connections
5   SocketAccept net_server, net_client; ! accept an incoming connection
6 ENDPROC
```

Listing 6: Procedura `StartSocketServer`

Procedura `ReceiveCharacter` (listing 7) uloží jeden přijatý znak od klienta do proměnné `net_msg`. Pokud by jich klient odeslal více v jedné zprávě (textový řetězec), procedura celou zprávu uloží do bufferu (`net_msg_full`) a při dalších voláních čte po jednom znaku z tohoto bufferu dokud není opět prázdný. Volání je blokující.

Procedura `SendMessage` (listing 7) odešle připojenému klientovi řetězec daný argumentem `msg` s přidaným ukončovacím znakem nového řádku (`\n`). Procedura `SendACK` využívá funkci `SendMessage` a budeme pomocí ní odesílat `ack` odezvu s pozicí danou argumentem `target_pos`.

Poslední procedura `CloseSocket` uzavře spojení pomocí systémové funkce `SocketClose`.

4.1.3.2 Pohybový modul

Pohybový modul nazveme *LeadThroughModuleR*. Do deklarační části dáme konstanty typu `robtarget`, které vytvoříme s využitím *lead-through* učení. Ke každé klávese na xylofonu vytvoříme dva body. První bod bude umístěn tak, aby konec dřevěné paličky uchopené ve svěrkách byl několik centimetrů nad středem klávesy xylofonu (zde nám nezáleží na vysoké přesnosti). Druhý bod bude umístěn tak, aby se konec paličky dotýkal klávesy.

První bod pro notu `C` nazveme `c1u`, druhý bod `c1d`. Analogicky pojmenujeme body pro ostatní noty (`d1u` – `d1d`, `e1u` – `e1d`,...). Na závěr vytvoříme ještě tři body – *home* pozici (`home_pos`), *ready* pozici (`rdy_pos`) a bod na spojnici mezi nimi (`int_pos`).

Před vytvářením bodů se ujistíme, že je kontrolér v manuálním módu, jinak nepůjdou body vytvářet. V menu *Jogging* aktivujeme na pravém rameni *lead-through* mód. Potom v hlavním menu v levém horním rohu zvolíme položku *Program Data*. Následně zvolíme z nabídky datový typ `robtarget`.

```
PROC ReceiveCharacter()
  IF str_len > 0 THEN
    net_msg := StrPart(net_msg_full, str_idx, 1);
    IF str_idx = str_len THEN
      str_len := 0; ! entire string is processed
    ENDIF
    str_idx := str_idx + 1;
  ELSE
    SocketReceive net_client, \RawData:=net_data, \Time:=WAIT_MAX;
    UnpackRawBytes net_data, 1, net_msg_full, \ASCII:=15;
    str_len := StrLen(net_msg_full);
    IF str_len = 1 THEN
      net_msg := net_msg_full;
      str_len := 0;
    ELSE
      net_msg := StrPart(net_msg_full, 1, 1);
      str_idx := 2;
    ENDIF
  ENDIF
ENDPROC
```

```
PROC SendMessage(string msg)
  msg := msg + ByteToStr(10, \Char); ! append new line
  SocketSend net_client, \Str:=msg; ! send to client
ENDPROC
```

Listing 7: Funkce `ReceiveCharacter` a `SendMessage`

Přesuneme rameno do pozice a klikneme v dolním panelu na *New*. Přiřadíme vytvářený bod do pravého ramene do modulu *LeadThroughModuleR*. Tento proces zopakujeme pro každý bod.

V modulu dále vytvoříme tři jednoduché procedury:

- `MoveToHome`,
- `MoveToReady`,
- `HitNote`.

Procedura `MoveToHome` přesune rameno z *ready* pozice `rdy_pos`, přes bod `int_pos` do domovské pozice `home_pos`. Procedura `MoveToReady` udělá to samé, ale v opačném pořadí.

Procedura `HitNote` slouží k pohybu robota na pozici nad klávesou (argument `note_up`), úhozu do klávesy (argument `note_down`) a zpět nad klávesu. Všechny tyto funkce mají na svém konci instrukci `WaitRob \InPos`, na které se kurzor programu zastaví, dokud předchozí pohyb neskončí. Implementace zmíněných funkcí je v listingu 8.

```
PROC MoveToHome()
  MoveJ rdy_pos, v1000, z50, Servo;
  MoveJ int_pos, v1000, z50, Servo;
  MoveJ home_pos, v1000, z50, Servo;
  WaitRob \InPos;
ENDPROC

PROC MoveToReady()
  MoveJ home_pos, v1000, z50, Servo;
  MoveJ int_pos, v1000, z50, Servo;
  MoveJ rdy_pos, v1000, z50, Servo;
  WaitRob \InPos;
ENDPROC

PROC HitNote(robtarget note_up, robtarget note_down)
  MoveL note_up, v1000, z50, Servo;
  MoveL note_down, v1000, z50, Servo;
  MoveL note_up, v1000, z50, Servo;
  WaitRob \InPos;
ENDPROC
```

Listing 8: Procedury pohybového modulu

4.1.3.3 Modul s konečným automatem

Modul s implementací řídícího konečného automatu pojmenujeme *FSMModuleR*. Deklarační část obsahuje pouze jednu proměnnou `state` pro uchovávání aktuálního stavu. Zbytek modulu tvoří jedna procedura `ControllerFSM` se samotnou implementací přechodové a výstupní funkce podle diagramu 3.3 z předchozí kapitoly. Zkrácená implementace je v listingu 9.

Funkce `ControllerFSM` běží do té doby, dokud se nedostane do stavu `END`, neboli `state = 10`. Ve stavu `START` odešleme odezvu `ack` s pozicí `rdy_pos`. Do této pozice poté přesuneme rameno voláním `MoveToReady` a změníme stav na `RDY`.

Ve stavu `RDY` odešleme odezvu `rdy` a čekáme na znak od klienta. Voláním procedury `ReceiveCharacter` načteme do globální proměnné `net_msg` nový znak. Po přijetí znaku `a-g` přejdeme do příslušného stavu `A-G`. Na znak `x` přejdeme do stavu `FINALIZE`. Jiné znaky ignorujeme.

Ve stavech `A-G` odešleme odezvu `ack` s příslušnou pozicí (ve stavu `C` pozici `c1d`, ve stavu `D` `d1d`, . . .). Voláním `HitNote` uhodíme do klávesy a vrátíme se do stavu `RDY`.

Ve stavu `FINALIZE` odešleme odezvu `ack` s pozicí `home_pos`, přesuneme rameno voláním procedury `MoveToHome` a přejdeme do stavu `END`.

Ve stavu `END` odešleme odezvu `end` a poté příkazem `GOTO lbl_fsm_r_end` skočíme na konec procedury, čímž ji ukončíme.

```
MODULE FSMModuleR
  VAR num state := 0;

  PROC ControllerFSM()
  lbl_fsm_r:
    IF state = 0 THEN ! START state
      SendACK(rdy_pos);

      g_GripIn;
      MoveToReady;
      state := 1;
    ELSEIF state = 1 THEN ! RDY state
      SendMessage(RDY);

      ReceiveCharacter;
      IF net_msg = "c" THEN
        state := 2; ! change state to C
      ELSEIF net_msg = "d" THEN
        state := 3; ! change state to D
      ...
      ...
      ELSEIF net_msg = "x" THEN
        state := 9; ! change state to FINALIZE
      ENDIF
    ELSEIF state = 2 THEN ! C state
      SendACK(c1d); ! send ACK with position
      HitNote c1u, c1d; ! hit the note
      state := 1; ! change state to RDY
    ELSEIF state = 3 THEN ! D state
      ...
      ...
    ELSEIF state = 9 THEN ! FINALIZE state
      SendACK(home_pos);
      MoveToHome;
      state := 10;
    ELSEIF state = 10 THEN ! END state
      SendMessage(END);
      GOTO lbl_fsm_r_end; ! reached final state => end
    ENDIF

    GOTO lbl_fsm_r; ! repeat
    lbl_fsm_r_end:
  ENDPROC
ENDMODULE
```

Listing 9: Zkrácený modul s konečným automatem

4.1.3.4 Hlavní modul

V hlavním modulu, který pojmenujeme `MainModuleR`, vytvoříme jedinou proceduru `Main`, která bude vstupním bodem programu. Inicializujeme Smart-

Gripper, počkáme na spojení od klienta a spustíme konečný automat. Po jeho skončení uzavřeme socket a program ukončíme. Viz listing 10.

```
MODULE MainModuleR
  PROC Main()
    g_Init; ! initialize SmartGripper

    StartSocketServer; ! wait until a~client connects
    ControllerFSM; ! run FSM until END state
    CloseSocket; ! close connection
  ENDPROC
ENDMODULE
```

Listing 10: Procedury pohybového modulu

4.2 HoloLens

V této sekci si popíšeme postup při implementaci UWP aplikace pro HoloLens, kterou jsme navrhli v předchozí kapitole. Vyvíjenou aplikaci nazveme *YuMiLens*. Vývoj můžeme rozdělit do následujících kroků, které rozebereme v dalších sekcích:

- vytvoření projektu v Unity,
- tvorba 3D scény,
- implementace logiky v C♯ skriptech,
- export do Visual Studia a nahrání do HoloLens.

4.2.1 Vytvoření projektu v Unity

V této sekci projdeme postupem instalace závislostí a vytvoření projektu v Unity.

4.2.1.1 Instalace Unity a knihovny MRTK

Pro snadnou správu verzí Unity a projektů si nainstalujeme pomocný program *Unity Hub*. Stáhneme si v sekci *Installs* verzi Unity `2018.3.13f`. Poté vytvoříme nový projekt přes tlačítko *New*, kde vyplníme název, umístění a jako *Template* zvolíme *3D*.

Dále si musíme stáhnout z GitHubu knihovnu *MRTK v2 RC1* [15]. V sekci release si stáhneme oba balíčky – *Foundation* a *Examples*. Balíčky v tomto pořadí importujeme jako *asset* do otevřeného projektu přes hlavní menu *Assets* možností *Import Package*→*Custom Package*. Pokud v průběhu importu

vyskočí dialogové okno, potvrdíme jej tlačítkem *Apply*. Tímto se nám automaticky nakonfiguruje prostředí pro *Mixed Reality* vývoj včetně nastavení skriptovacího backendu na *IL2CPP*, což je s touto verzí MRTK preferovaná volba.

Nakonec musíme importovat závislosti *TextMesh Pro*, což uděláme přes menu *Window*, kde zvolíme *TextMeshPro→Import TMP Essential Resources*. Po importu **musíme** restartovat Unity.

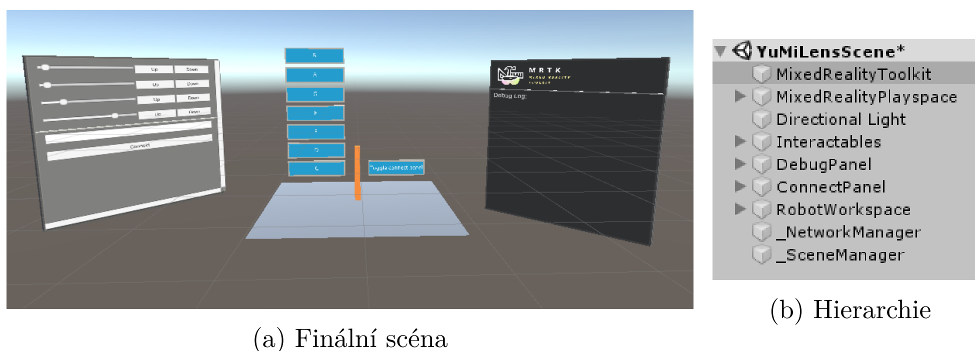
4.2.1.2 Instalace Windows SDK 18362

V čase vývoje této aplikace musíme nainstalovat speciálním postupem nejnovější verzi Windows SDK 18362. Tato verze stále není v oficiálních aktualizacích Windows a je dostupná pouze přes službu *Windows Insider Program*. Ke stažení SDK potřebujeme účet u Microsoftu.

Nenainstalujeme-li tuto verzi, nebude fungovat kompilace s verzí *MRTK RC1*.

4.2.2 Tvorba scény

V této podsekci si popíšeme hierarchii a tvorbu 3D objektů ve scéně. Po vytvoření všech objektů (podle následujících podsekcí) dostaneme scénu na obrázku 4.2a s hierarchií objektů dle obrázku 4.2b.



(a) Finální scéna

(b) Hierarchie

Obrázek 4.2: Finální scéna a hierarchie objektů

4.2.2.1 Naklonování scény z balíčku

V čerstvém projektu máme nyní otevřenou prázdnou scénu. Abychom mohli využívat možností knihovny MRTK, musíme mít ve scéně objekty[5] s těmito komponentami:

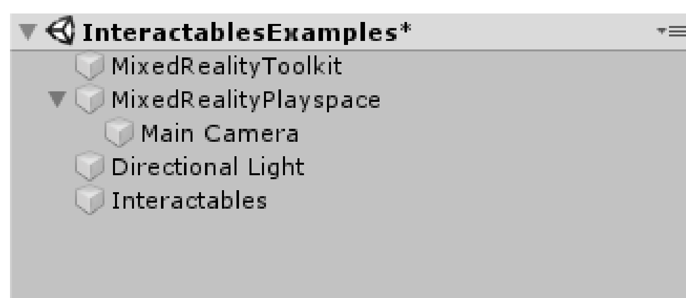
- `MixedRealityToolkit`,

[5] `GameObject` v Unity

- `MixedRealityInputModule`.

Jelikož jsme si importovali i balíček *Examples*, vytvoříme si naši scénu z existující scény *InteractablesExample*, abychom si ušetřili práci. Zkontrolujeme funkčnost otevřené demo scény kliknutím na tlačítko *Play* v horní části editoru. Pomocí kláves `W, S, A, D` se pohybujeme a při stisku pravého tlačítka myši otáčíme kamerou.

Ze scény smažeme nepotřebné prvky tak, abychom dostali stav jako na obrázku 4.3. Scénu následně uložíme pod názvem *YuMiLensScene*.



Obrázek 4.3: Vyčištěná example scéna

Jako poslední věc musíme změnit typ simulace gest. Protože se jedná o verzi *RC1*, která již v základu počítá s vývojem na platformu *HoloLens 2*, musíme změnit v objektu *MixedRealityToolkit* parametr *Hand Simulation* z typu *Articulated* na *Gestures*.

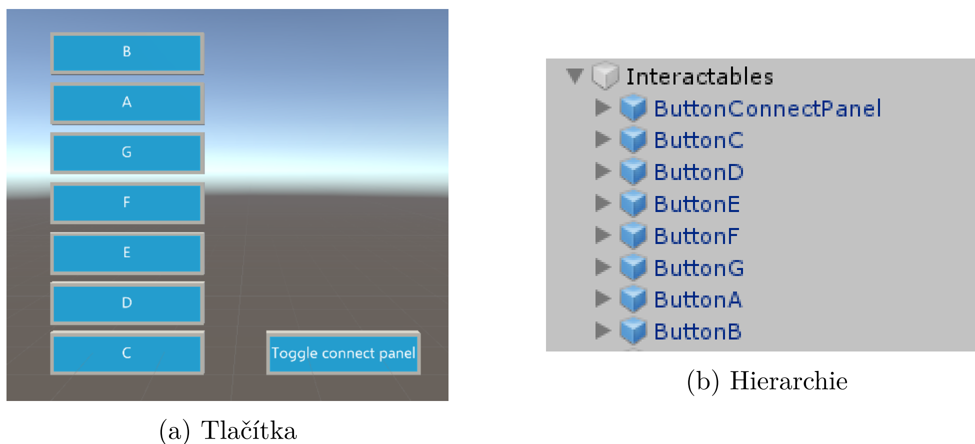
V takto nastavené scéně se do aplikace při spuštění navíc automaticky přidají 2 debugovací prvky. V dolní části zorného pole brýlí se zobrazí plovoucí panel s počítadlem FPS[6]. V prostoru se dále vykreslí trojúhelníková síť kolem hranic reálných objektů – můžeme si tak ověřit, jak HoloLens v daný okamžik vnímají prostor kolem sebe (viz sekce 5.1).

4.2.2.2 Tlačítka

Ve scéně nejprve vytvoříme 7 tlačítek pro odesílání příkazů k zahrání noty. Tlačítka vytvoříme z *prefabu* `Button` a vložíme jej do objektu *Interactables*. Na tlačítka dáme popisek s textem `A-G`. Vznikne hierarchie na obrázku 4.4b.

Stejným způsobem vytvoříme ještě jedno samostatné tlačítko a popíšeme jej textem `Toggle connect panel`. Tímto tlačítkem budeme zobrazovat nebo skrývat panel pro zadávání IP adresy a připojení k robotovi. Výsledkem bude stav jako na obrázku 4.4a.

[6]Frames Per Second



(a) Tlačítka

(b) Hierarchie

Obrázek 4.4: Tlačítka a hierarchie objektů

4.2.2.3 Panel pro připojení k robotovi

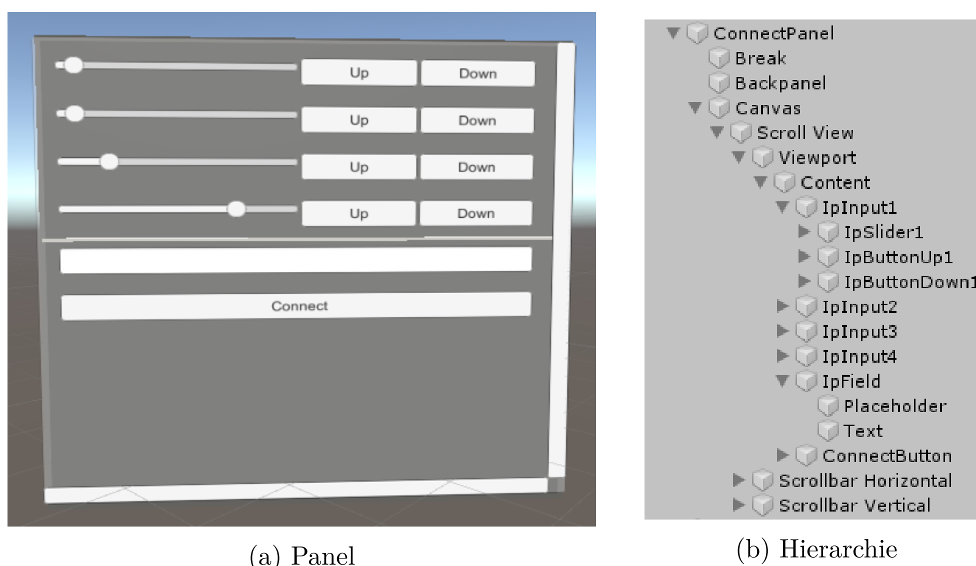
Panel (plovoucí okno) pro připojení k robotovi vytvoříme recyklací podobného panelu z dostupné example scény *HandInteractionExamples*. Objekt v této scéně se nazývá *UnityUIExamples*. V naší scéně tento objekt pojmenujeme *ConnectPanel*. V tomto panelu vytvoříme 4 posuvníky (komponenta *Slider*) pro zadávání jednotlivých oktetů IP adresy. Vedle všech posuvníků přidáme dvě tlačítka (komponenta *Button*) pro inkrementaci a dekrementaci daného oktetu adresy.

Pod posuvníky přidáme *Input Field*, ve kterém budeme zobrazovat zadanou IP adresu z posuvníků výše. Pod textové pole přidáme tlačítko s popiskem *Connect*, které bude iniciovat síťové spojení. Tuto (na první pohled obskurní) metodu zadávání adresy volíme z důvodů popsaných níže v sekci o problémech při implementaci (viz 5.2.3) a jsme nuceni se u tohoto prvku odchýlit od původního návrhu. Výsledkem bude panel na obrázku 4.5a s hierarchií na obrázku 4.5b.

4.2.2.4 Panel pro debugovací zprávy

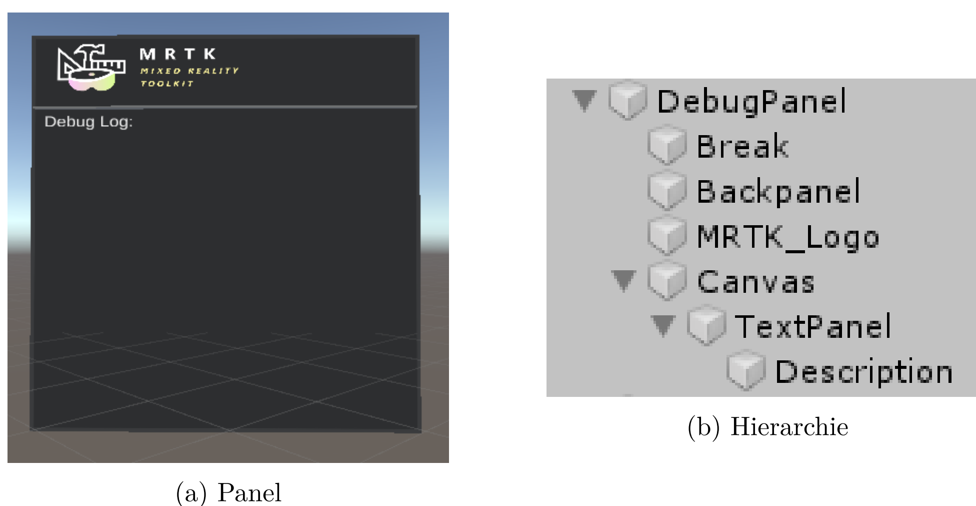
Panel pro zobrazení zpráv pro debugování vytvoříme recyklací objektu *SceneDescriptionPanel* ze stejné example scény jako v předchozí sekci. Objekt s názvem *Description*, přes který budeme zobrazovat logy, bude mít na sobě komponentu typu *Text Mesh Pro UGUI*. Pro zřejmost nastavíme její atribut *text* na „Debug Log:“. Zbytek objektů zachováme, slouží pouze k vizuálním účelům.

Výsledkem bude panel na obrázku 4.6a, který bude mít hierarchii objektů dle obrázku 4.6b.



(a) Panel (b) Hierarchie

Obrázek 4.5: Panel pro připojení a jeho hierarchie objektů



(a) Panel (b) Hierarchie

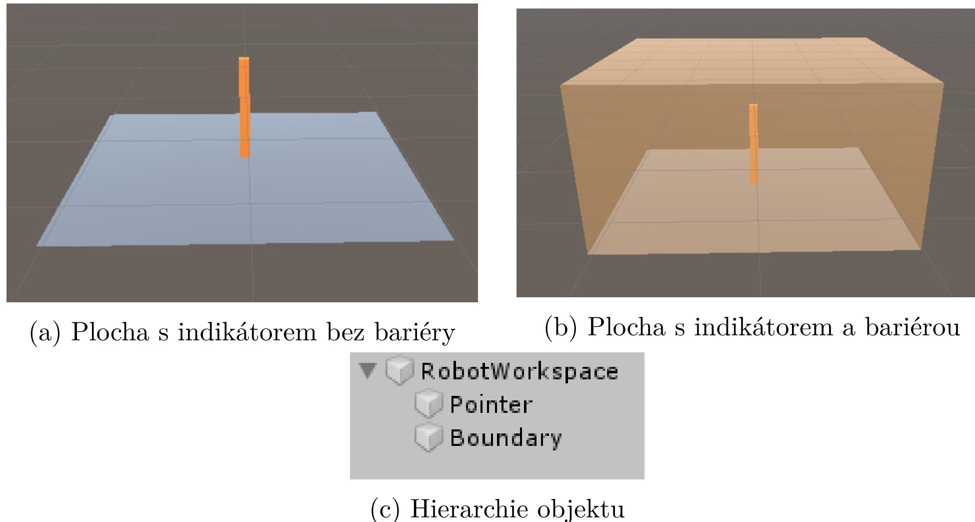
Obrázek 4.6: Debugovací panel a jeho hierarchie objektů

4.2.2.5 Virtuální plocha s indikátorem

Virtuální plocha s indikátorem a bariérou bude mít nadřazený `GameObject` s názvem *RobotWorkspace*. Tento objekt vytvoříme přes menu *GameObject* v hlavní liště, kde zvolíme možnost *3D Object→Plane*.

Dále vytvoříme podobným způsobem podobjekt pro indikátor pozice nazvaný *Pointer*, ale nikoliv jako *Plane*, ale jako *Cube*. Podobjekt pro bariéru vytvoříme zcela stejně. Bariéře navíc nastavíme transparentní materiál, aby bylo na indikátor stále vidět.

Ploše nastavíme škálování ve všech osách na hodnotu `0.05`. Škálování indikátoru nastavíme na `x=0.3`, `y=3` a `z=0.3`. Bariéře nastavíme škálování na hodnoty `x=10`, `y=5`, `z=10` a pozici na ose Y nastavíme na `2.5`. Bariéru v inspektoru objektu deaktivujeme, aby nebyla zobrazena ve výchozím stavu – to budeme dělat z kódu. Výsledkem bude virtuální plocha na obrázku 4.7a (resp. 4.7b s aktivovanou bariérou). Hierarchie objektu je na obrázku 4.7c.



(a) Plocha s indikátorem bez bariéry

(b) Plocha s indikátorem a bariérou

(c) Hierarchie objektu

Obrázek 4.7: Virtuální plocha s indikátorem a bariérou

4.2.2.6 Prázdné objekty pro skripty

K implementaci C♯ skriptů, které nebudou navázány na žádný zobrazený objekt ve scéně, potřebujeme vytvořit ve scéně tyto tři prázdné entity typu *GameObject*:

- *NetworkManager*,
- *SceneManager*,
- *ButtonReceiver*.

NetworkManager bude sloužit pro skript se síťovými funkcemi a *SceneManager* pro skript s funkcemi pro manipulaci s objekty ve scéně (viz další sekce 4.2.3). Názvy obou objektů prefixujeme znakem podtržítka, abychom je v hierarchii odlišili od objektů, které se reálně zobrazují ve scéně.

ButtonReceiver vložíme do objektu *Interactables* a bude sloužit k odchytávání událostí o stisku tlačítek, které jsme vytvořili v sekci 4.2.2.2 výše.

4.2.3 Implementace logiky

Logiku implementujeme ve skriptech v jazyce C♯. V projektové složce *Assets* si vytvoříme složku *Scripts* a v ní 4 soubory se stejnojmennými třídami:

- `DebugManager.cs`,
- `NetworkManager.cs`,
- `SceneManager.cs`,
- `YumiLensReceiver.cs`.

Třídy typu *Manager* implementujeme jako singletony. To znamená, že budeme mít přístup k jedné a té samé instanci ze všech míst v programu. Implementace obecného singletonu, ze které budeme vycházet v následujících podsekcích, je v listingu 11.

```
1  public class Manager : MonoBehaviour
2  {
3    public static Manager Instance = null;
4
5    void Awake()
6    {
7        if (Instance == null)
8            Instance = this;
9        else if (Instance != this)
10           Destroy(gameObject);
11   }
12 }
```

Listing 11: Implementace obecného singletonu

Další vlastnost knihovny *.NET*, kterou využijeme, jsou asynchronní úlohy (tasky). Dlouhé operace (navázání síťového spojení přes socket, příjem dat, ...) nemůžeme nechat blokovat hlavní vlákno programu. Proto je vytvoříme jako metody s klíčovým slovem `async`, které vrací objekt typu `Task` a budeme je volat pomocí klíčového slova `await`.

4.2.3.1 DebugManager

Skript *DebugManager* umístíme jako komponentu na objekt *_DebugManager*. Třída bude deklarována jako singleton podle kódu v listingu 11. Dále bude obsahovat jednu metodu `Log(string)`, která vypíše předaný argument na debugovací panel (viz sekce 4.2.2.4).

Zprávy budou umisťovány na panel každá na další řádek. Atribut, přes který obsah panelu nastavujeme, je v komponentě typu *TextMeshProUGUI*

na objektu *Description*. Jakmile počet zpráv přesáhne hodnotu 18 (maximální počet řádků před přetečením přes okraj panelu), budou zprávy posunuty o jeden řádek nahoru a první řádek odstraněn. Metodu můžeme použít z libovolného dalšího skriptu voláním `DebugManager.Instance.Log(...)`.

4.2.3.2 NetworkManager

NetworkManager implementujeme jako singleton v listingu 11. V NetworkManageru definujeme síťové funkce:

- `Connect(string ip, string port)` – připojení na adresu a port,
- `SendChar(char c)` – asynchronní odeslání jednoho znaku,
- `ReceiveMessage()` – asynchronní příjem zprávy,
- `Close(()` – zavření socketu.

Při implementaci těchto funkcí se setkáme s problémem nekompatibility *.NET* knihoven prostředí Unity a UWP aplikací. Pokud aplikaci simulujeme přímo v editoru, nemůžeme pro komunikaci použít třídy z jmenného prostoru *Windows.Networking* a musíme použít sockety ze *System.Net.Sockets*. Ty ovšem nemůžeme použít, jakmile projekt exportujeme do Visual Studia, abychom jej zkompilovali jako UWP aplikaci a nahráli do HoloLens. Tento problém vyřešíme preprocesorovými direktivami podmíněného překladu – viz implementace funkce `Connect` v listingu 12.

```
1 public async Task<int> Connect(string ip, string port)
2 {
3 #if !UNITY_EDITOR // we are in UWP (HoloLens)
4   return await ConnectUWP(ip, port);
5 #else // we are running directly in Unity
6   return await ConnectUnity(ip, port);
7 #endif
8 }
```

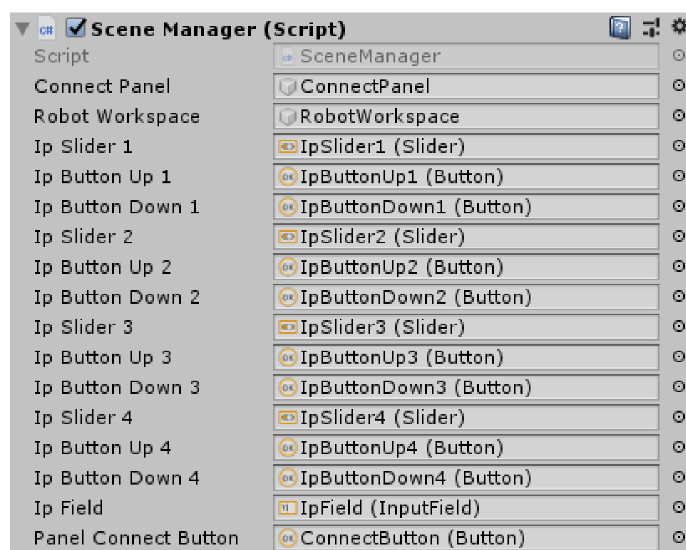
Listing 12: Implementace metody `Connect`

Spustíme-li kód ze zmíněného listingu v Unity, spustí se asynchronní metoda `ConnectUnity`, která používá sockety ze *System.Net.Sockets*. Po exportu a kompilaci pro HoloLens se bude spouštět metoda `ConnectUWP`, která využívá sockety z *Windows.Networking*.

4.2.3.3 SceneManager

Ve skriptu *SceneManager* implementujeme zbytek logiky, starající se o scénu – inicializaci UI v panelu pro připojení a obsluhu virtuální plochy (přepočet souřadnic, zobrazování bariéry, atd.).

Abychom mohli přistupovat k daným objektům ve scéně, předáme si jejich reference přes veřejné rozhraní skriptu (viz obr. 4.8).



Obrázek 4.8: Rozhraní komponenty skriptu `SceneManager`

Ve skriptu v metodě *Start* nejprve provedeme inicializaci UI prvků v panelu pro připojení. Nastavíme posuvníky adresy tak, abychom dostali výchozí adresu `10.10.48.194`, což je adresa přidělená robotovi v laboratoři. Z objektu *Robot Workspace* následně vypočítáme délku a šířku virtuální plochy s indikátorem.

Na konci metody `Start` inicializujeme obsluhu kliknutí na tlačítko *Connect* (viz listing 13). Tlačítku změníme text na *Disconnect* po úspěšném připojení (v metodě `ConnectToRobot`) a spustíme asynchronní příjem zpráv (v metodě `ReceiveMessages`). Po kliknutí na `Disconnect` odešleme znak `x`.

4.2.3.4 YumiLensReceiver

Události o kliknutí na tlačítka pro zahrání klávesy nebo zobrazení/skrytí panelu pro připojení budeme zachycovat ve skriptu *YumiLensReceiver*. Skript vytvoříme podle souboru `CustomInteractablesReceiver.cs`, který najdeme v importovaném *Example* balíku. Z toho skriptu odstraníme nepotřebné metody a využijeme pouze metodu `OnClick`, ve které obsloužíme kliknutí (viz listing 14).

```
panelConnectButton.onClick.AddListener(async delegate
{
  if (NetworkManager.Instance.IsConnected())
  {
    await NetworkManager.Instance.SendChar('x'); // send finish command
  } else
  {
    await ConnectToRobot(); // wait until connection
    ReceiveMessages(); // start receiving messages in async task
  }
});
```

Listing 13: Implementace obsluhy kliknutí na tlačítko *Connect*

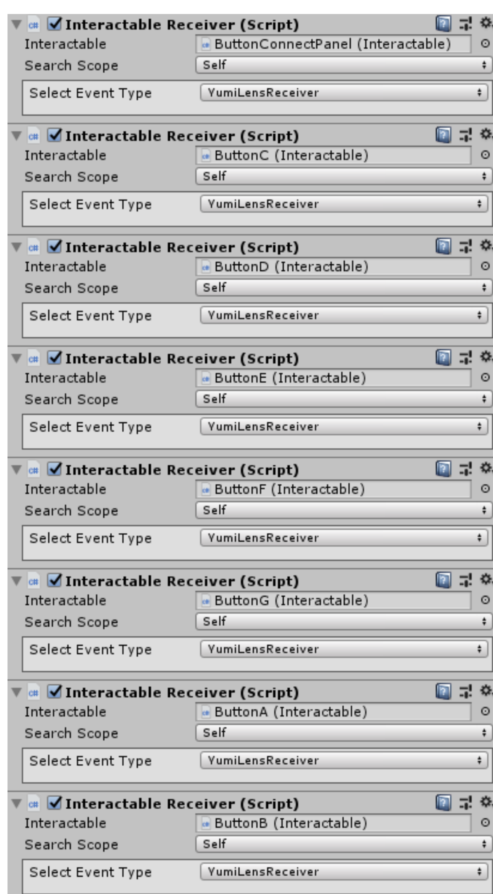
```
public async override void OnClick(InteractableStates state,
                                   Interactable source,
                                   IMixedRealityPointer pointer = null)
{
  base.OnClick(state, source);
  DebugManager.Instance.Log("Clicked " + source.name); // log click to panel
  if (source.name == "ButtonConnectPanel") // toggle-button was clicked
    SceneManager.Instance.ToggleConnectPanel();
  else if (source.name.StartsWith("Button")) // note button was clicked
  {
    if (NetworkManager.Instance.IsConnected()) // we are connected
    {
      char cmd = char.ToLower(source.name[source.name.Length - 1]);
      int ret = await NetworkManager.Instance.SendChar(cmd); // send note
      if (ret == 0)
        DebugManager.Instance.Log($"Command '{cmd}' sent");
    }
  }
}
```

Listing 14: Implementace metody *OnClick*

Abychom skript navázali na konkrétní události, musíme na objektu *ButtonReceiver* vytvořit pro každé tlačítko komponentu *Interactable Receiver*, která je dostupná z knihovny *MRTK*. Každé komponentě nastavíme prvek *Interactable* na daný objekt tlačítka a nastavíme *Event Type* na *YumiLensReceiver* (viz obr. 4.9).

4.2.4 Export projektu a nahrání do HoloLens

Zde se podíváme na proces exportu projektu do Visual Studia a následné kompilaci a nahrání UWP aplikace do HoloLens.



Obrázek 4.9: Receiver událostí z tlačítek

4.2.4.1 Export z Unity

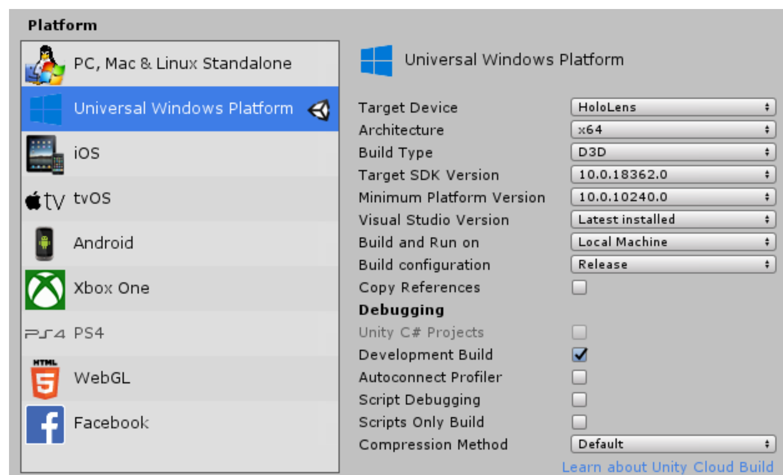
Před exportem UWP aplikace si nejprve otevřeme v Unity menu *File→Build Settings*, kde nastavíme možnosti podle obrázku 4.10. Dále se ujistíme, že máme přidanou scénu *YuMiLensScene*, případně ji přidáme tlačítkem *Add Open Scenes*.

Následně otevřeme menu *Mixed Reality Toolkit→Build Window*. Zde nastavíme cíl exportu na složku *UWP*, kterou umístíme do kořene Unity projektu a spustíme export tlačítkem *Build Unity Project*.

4.2.4.2 Kompilace a nahrání do HoloLens

Před kompilací si spustíme HoloLens. Po nabootování se přihlásíme, připojíme na síť a v aplikaci *Holographic Remoting Player* zjistíme IP adresu brýlí.

Exportovaný projekt otevřeme tlačítkem *Open in Visual Studio*. Ve Visual Studiu nastavíme v horní liště typ sestavení na *Release* a architekturu na



Obrázek 4.10: Build settings

x86. V menu *Debug→YuMiLens Properties* v záložce *Debugging* nastavíme typ debuggeru na *Remote Machine* a *Machine Name* na IP adresu HoloLens.

Následně spustíme kompilaci a nahrání v horní liště tlačítkem *Remote Machine*. Po nahrání se aplikace spustí v debugovacím režimu. Ukončíme ji gestem *bloom* a spustíme ji již standardně přes seznam aplikací v brýlích, kde se nachází pod názvem *YuMiLens*.

Kapitola 5

Testování a budoucí práce

V této kapitole zhodnotíme testování aplikace – ověříme její funkčnost, nedostatky a budeme diskutovat možnosti pro budoucí práci.

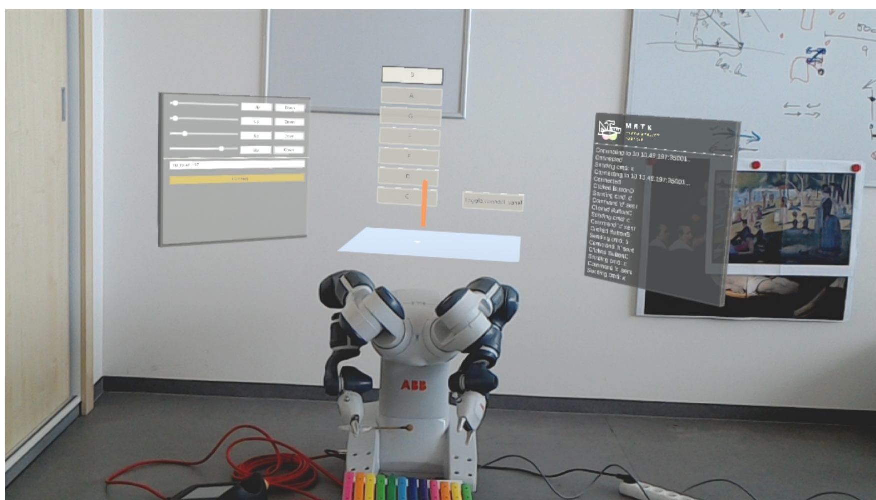
5.1 Testování

Demonstrační aplikaci jsme testovali v provizorním prostředí laboratoře vestavných systémů. Aplikaci pro HoloLens jsme nejprve testovali v podobě popsané v kapitole o implementaci. Poté jsme provedli stejný test znovu, ale aplikaci jsme zkompilovali bez debugovacích prvků (trojúhelníkové sítě a počítadla FPS) – tuto verzi jsme nahráli na záznam, který zmiňujeme dále.

Před testem aplikace jsme nejprve robota drátově připojili přes jeho WAN port do lokální sítě laboratoře. Poté jsme robota spustili a po nabootování zjistili, že jeho přidělená IP adresa je `10.10.48.194`. Zkalibrovali jsme oba SmartGrippery (viz sekce 2.1.3.4) a uchytli paličku do jeho pravé svěrky. Následně jsme spustili HoloLens, ujistili se, že jsme připojeni na stejnou lokální síť a otevřeli v menu aplikaci *YuMiLens*.

Poté jsme spustili přes FlexPendant řídící program robota. V *YuMiLens* jsme na panelu s připojením klikli na tlačítko *Connect*. Rameno se přesunulo do pozice *ready*. Následně jsme zkoušeli jednotlivá tlačítka kláves. Robot se korektně pohyboval a v brýlích byly indikovány změny pozic na *virtuální ploše*. Na závěr testu jsme klikli na tlačítko *Disconnect*. Rameno se přesunulo do pozice *home* a program řízení skončil. V průběhu celého testu byly zobrazovány na debugovacím panelu zprávy o kliknutích na tlačítka, o odeslaných znacích, apod.

Průběh testování jsme zaznamenali jako videozáznam, který je ve zdrojových souborech na přiloženém médiu. Pro ilustraci je na obrázku 5.1 snímek obrazovky z testování.



Obrázek 5.1: Snímek obrazovky z testování

5.2 Problémy při vývoji

V této sekci popíšeme problémy, které nastaly v průběhu vývoje a jakým způsobem ovlivnily výsledek práce.

5.2.1 Dlouhý proces testování aplikace na HoloLens

Jedním z hlavních problémů, které se při vývoji objevily, byl velmi dlouhý proces testování. Před verzí *MRTK v2 RC1* (viz další sekce) nebylo možné testovat aplikaci pro HoloLens jinak, než přímo ve fyzickém zařízení nebo v emulátoru.

V každém případě bylo nutné pokaždé exportovat projekt z Unity, otevřít jej ve Visual Studiu a zkompilovat jej. V případě spouštění přímo v brýlích byla nutná kontrola správného nastavení *deploy* konfigurace (IP adresy). Celý tento proces byl naměřen a při použití notebooku *ThinkPad T480* trval variabilně od 7 do 9 minut. Ve fázi seznamování se s danou platformou to byl velice omezující faktor. Tento nedostatek byl částečně vyřešen po přechodu na novou verzi knihovny *MRTK v2 RC1*.

5.2.2 Přechod na MRTK v2 RC1

V druhé polovině implementace této práce vyšla verze *RC1* knihovny *MRTK*. Za jiných okolností není výhodné měnit verzi hlavního frameworku v průběhu vývoje, ale verze *RC1* přišla s možností simulace přímo v editoru Unity. Tento přechod částečně vyřešil problém z minulé sekce – mohli jsme simulovat gesta rukou a pohyb přímo v editoru.

Přechod na novou verzi ve finále znamenalo kompletní vytvoření celého projektu od začátku (import balíčků, vytvoření scény podle example scény, atd.), jelikož byla nová verze příliš nekompatibilní se starou. Tento přechod byl nutný kvůli možnosti simulace v editoru, ale velmi zdržoval od užitečné práce.

I přes úspěšný přechod na verzi *RC1* se objevil další problém s nemožností použít stejné rozhraní pro sockety, které by fungovalo jak v simulaci v Unity, tak ve zkompilované UWP aplikaci v HoloLens. Museli jsme proto implementovat určité funkce dvakrát a vložit je do bloků podmíněného překladu (viz sekce 4.2.3.2).

5.2.3 Nefunkční plovoucí klávesnice v HoloLens

V původním návrhu aplikace pro HoloLens jsme počítali s plovoucí systémovou klávesnicí pro zadávání IP adresy robota. Avšak i přes přesné použití podle dokumentace [12] se klávesnice nezobrazila. Museli jsme proto přistoupit k alternativnímu řešení pomocí posuvníků a tlačítek, které je podstatně méně uživatelsky přívětivé.

5.3 Budoucí práce

Případná budoucí práce by mohla navazovat hlavně v oblasti vylepšování demonstrační aplikace. Je zde prostor pro využití „trackovací" knihovny *ARToolkit*, která by mohla umožnit rozpoznání pracoviště robota a zobrazovat např. indikátor pohybu přímo v prostoru a nikoliv pouze na *virtuální ploše* jako nyní.

U robota YuMi by mohlo být využito více jeho možností, např. kamera ve SmartGripperu, přísavky nebo kooperace obou ramen na nějakém složitějším úkolu. Namísto jednoduchých úderů paličkou na xylofon by mohl robot např. určitým způsobem skládat kostky dle vstupu od uživatele.

Závěr

Cílem této práce byla rešerše vývoje aplikací pro holografické brýle HoloLens a pro kooperativního robota YuMi. Na základě této rešerše jsme implementovali jednoduchou aplikaci, která demonstruje interakci uživatele s robotem přes rozšířenou realitu.

V analýze jsme se seznámili s robotem YuMi, vývojovým prostředím RobotStudio, programovacím jazykem *RAPID* a popsali si základní práci s těmito nástroji. Dále jsme si představili brýle HoloLens a prostředí Unity, ve kterém se programují. Na konci analýzy jsme si představili 3 možné způsoby, jak implementovat komunikaci mezi robotem a brýlemi.

Po zvážení všech výhod a nevýhod jsme v návrhu zvolili způsob komunikace přes TCP sockety. Na tomto základě jsme poté navrhli demonstrační aplikaci, ve které uživatel interaktivně posílá robotovi příkazy, na kterou klávesu xylofonu má zahrát.

V implementaci jsme vytvořili řídící program pro robota v jazyce RAPID a aplikaci v Unity pro brýle HoloLens. Přes 3D rozhraní může uživatel navázat síťové spojení s robotem a klikat na tlačítka, která volí klávesu xylofonu. Do brýlí uživatel dostává odezvu o pozici ramena robota, která se zobrazuje ve formě pohyblivého indikátoru.

V rámci poslední kapitoly jsme aplikaci úspěšně otestovali, vytvořili videozáznam s demonstrací a zhodnotili problémy, které vznikly v průběhu vývoje.

Literatura

[1] ABB: *RAPID Reference Manual [online]*. [cit. 2019-04-08]. Dostupné z: http://rab.ict.pwr.wroc.pl/irb1400/datasys_rev1.pdf

[2] Hirokazu Kato, Mark Billinghurst, Ivan Poupyrev: ARToolkit. [cit. 2019-04-28]. Dostupné z: https://github.com/artoolkit/artoolkit5

[3] ABB: RobotWare main resources [online]. [cit. 2019-04-11]. Dostupné z: http://developercenter.robotstudio.com/blobproxy/devcenter/Robot_Web_Services/html/images/rws.png

[4] RobotWorx: ABB YuMi - IRB 14000-0.5/0.5 [online]. [cit. 2019-04-11]. Dostupné z: https://www.robots.com/images/robots/ABB/Collaborative/ABB_YuMi_cutout.png

[5] Windows Central: Microsoft HoloLens [online]. [cit. 2019-04-11]. Dostupné z: https://www.windowscentral.com/sites/wpcentral.com/files/topic_images/2016/hololens-topic.png

[6] ABB: YuMi® – IRB 14000 [online]. [cit. 2019-04-11]. Dostupné z: http://search.abb.com/library/Download.aspx?DocumentID=9AKK106354A3256&LanguageCode=en&DocumentPartId=&Action=Launch

[7] ABB: *Introduction to RAPID [online]*. [cit. 2019-04-08]. Dostupné z: http://www.oamk.fi/~eeroko/Opetus/Tuotantoautomaatio/Robotiikka/Introduction_to_RAPID_3HAC029364-001_rev-_en.pdf

[8] ABB: *Application manual – MultiMove*. [cit. 2019-04-11]. Dostupné z: https://us.v-cdn.net/5020483/uploads/editor/9q/btyuwt8w867k.pdf

[9] ABB: *RAPID Instructions, Functions and Data Types [online]*. [cit. 2019-04-11]. Dostupné z: https://library.e.abb.com/public/

688894b98123f87bc1257cc50044e809/Technicalreferencemanual_RAPID_3HAC16581-1_revJ_en.pdf

[10] ABB: *Getting started, IRC5 and RobotStudio*. [cit. 2019-04-12]. Dostupné z: https://www.manualslib.com/manual/1256650/Abb-Irc5.html

[11] Lars Klint: *HoloLens Succintly*. Syncfusion, 2018.

[12] Microsoft Contributors: System Keyboard [online]. [cit. 2019-04-29]. Dostupné z: https://microsoft.github.io/MixedRealityToolkit-Unity/Documentation/README_SystemKeyboard.html

[13] Matt Zellen, Alex Turner and Brandon Bray: HoloLens (1st gen) hardware details. *Microsoft Docs [online]*, březen 2018, [cit. 2019-04-12]. Dostupné z: https://docs.microsoft.com/en-us/windows/mixed-reality/hololens-hardware-details

[14] Unity Technologies: Learning the interface [online]. [cit. 2019-04-28]. Dostupné z: https://docs.unity3d.com/Manual/LearningtheInterface.html

[15] Microsoft: Mixed Reality Toolkit. [cit. 2019-04-24]. Dostupné z: https://github.com/Microsoft/MixedRealityToolkit-Unity

[16] Microsoft Contributors: System Keyboard [online]. [cit. 2019-04-29]. Dostupné z: https://microsoft.github.io/MixedRealityToolkit-Unity/Documentation/README_Button.html

[17] Microsoft: Install the tools. [cit. 2019-04-29]. Dostupné z: https://docs.microsoft.com/en-us/windows/mixed-reality/install-the-tools

[18] Long Qian: HoloLens with ARToolkit v0.2. [cit. 2019-04-24]. Dostupné z: https://github.com/qian256/HoloLensARToolKit

[19] Unity Technologies: IL2CPP & Mono [online]. [cit. 2019-04-30]. Dostupné z: https://unity3d.com/learn/tutorials/topics/best-practices/il2cpp-mono

[20] Unity Technologies: Universal Windows Platform: .NET Scripting Backend [online]. [cit. 2019-04-30]. Dostupné z: https://docs.unity3d.com/2018.3/Documentation/Manual/windowsstore-dotnet.html

[21] ABB: *PC SDK*. [cit. 2019-04-11]. Dostupné z: http://developercenter.robotstudio.com/pcsdk

[22] Carter, Phillip: Port your code from .NET Framework to .NET Core. *Microsoft Docs [online]*, prosinec 2018, [cit. 2019-04-11]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/core/porting/

[23] ABB: *Robot Web Services*. [cit. 2019-04-11]. Dostupné z: http://developercenter.robotstudio.com/webservice

[24] Internet Engineering Task Force (IETF): *The Websocket Protocol*. 2011. Dostupné z: https://tools.ietf.org/html/rfc6455

Příloha **A**

Seznam použitých zkratek

TCP Transmission Control Protocol

SDK Software Development Kit

UWP Universal Windows Platform

C♯ Programovací jazyk (čti „sí šarp“)

IL2CPP Intermediate Language to C++

FPS Frames Per Second

Příloha B

Obsah přiloženého CD

```
README.md ........................ stručný popis obsahu CD
src
├── impl ......................... zdrojové kódy implementace
│   ├── RobotStudio .............. zdrojové soubory pro robota YuMi
│   │   ├── RAPID ................ zdrojové kódy v jazyku RAPID
│   │   └── YuMi_Lens.rspag ...... zabalená stanice pro funkci Unpack&Work
│   └── Unity .................... zdrojové kódy pro brýle HoloLens
└── thesis ....................... zdrojová forma práce ve formátu LaTeX
    ├── assets ................... obrázky, diagramy, grafy a video z testování
    └── cover .................... desky práce
text ............................. text práce
└── DP_Podrouzek_Adam_2019.pdf ... text práce ve formátu PDF
```