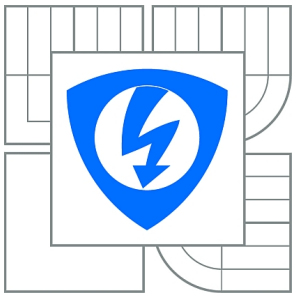
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF CONTROL AND INSTRUMENTATION

EXPERIMENT ROJOVÉ INTELIGENCE V RDS

SWARM INTELLIGENCE BASED EXPERIMENT IN RDS

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

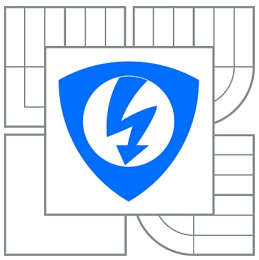
AUTOR PRÁCE — LADISLAV KOLÁŘ

AUTHOR

VEDOUCÍ PRÁCE — Ing. PETR HONZÍK, Ph.D.

SUPERVISOR

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav automatizace a měřicí techniky

Bakalářská práce

bakalářský studijní obor
Automatizační a měřicí technika

Student:	Ladislav Kolář	***ID:***	115201
Ročník:	3	***Akademický rok:***	2010/2011

NÁZEV TÉMATU:

Experiment rojové inteligence v RDS

POKYNY PRO VYPRACOVÁNÍ:

- Vypracujte přehled robotických simulátorů, ve kterých by bylo možné testovat různé interprety ovládání jednoosého diferenciálně řízeného robota pomocí joysticku;
- seznamte se s prostředím Robotics Developer Studio (RDS) a vypracujte manuál popisující komponenty RDS a postup vytvoření vlastního projektu;
- seznamte se s pojmem rojová inteligence;
- nastudujte a naprogramujte v RDS vybraný experiment ze článku zadaného školitelem;
- zopakujte a srovnejte výsledky vybraného experimentu realizovaného v RDS s výsledky publikovanými v článku.

DOPORUČENÁ LITERATURA:

Tsankova, D., Georgieva, V., Zezulka, F. and Bradac, Z. (2007), Immune network control for stigmergy based foraging behaviour of autonomous mobile robots. International Journal of Adaptive Control and Signal Processing, 21: 265–286. doi: 10.1002/acs.915

Termín zadání:	7.2.2011	***Termín odevzdání:***	30.5.2011

Vedoucí práce: Ing. Petr Honzík, Ph.D.

prof. Ing. Pavel Jura, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

ABSTRAKT

Práce je tvořena dvěma částmi. První část je věnována přehledu robotických simulátorů a pak blíže programu Microsoft Robotics Developer Studio (MRDS). Je zpracován manuál zaměřený na postup při tvorbě vlastního projektu kombinujícího jazyky C# a Visual Programming Language (VPL). Druhou část tvoří stručná definice pojmu rojová inteligence a popis experimentu, který má být implementován v prostředí MRDS. Následuje popis implementace a stručné zhodnocení dosažených výsledků.

KLÍČOVÁ SLOVA

Rojová inteligence,Microsoft Robotics Developer Studio,simulace,VPL

ABSTRACT

The thesis consists of two parts. In the first one the overview of robotic simulators is compiled with more detailed focus on the Microsoft Robotics Developer Studio (MRDS). The process of development of the new project including both programming languages C# and Visual Programming Language (VPL) is described in form of manual. The second part of the thesis is aimed to explain the term swarm intelligence, to describe the concrete experiment and to implement it in MRDS. Finally the achieved results are summarized and discussed.

KEYWORDS

Swarm inteligent,Microsoft Robotics Developer Studio,simulation,VPL

KOLÁŘ, Ladislav *EXPERIMENT ROJOVÉ INTELIGENCE V RDS*: bakalářská práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace, 2011. 50 s. Vedoucí práce byl Ing. Petr Honzík, CSc.

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „EXPERIMENT ROJOVÉ INTELIGENCE V RDS“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

Brno ....

(podpis autora)

PODĚKOVÁNÍ

Děkuji vedoucímu práce Ing. Petru Honzíkovi, Ph.D. za rady, trpělivost a ochotu vždy pomoct.

OBSAH

SEZNAM OBRÁZKŮ

ÚVOD

Simulace je dnes jednou z nejčastěji prováděných činností při vývoji. Umožňuje ověřit předem výsledek a ušetřit tak nemalé finance. Simulovat lze chemické reakce, vývoj podnebí nebo zatížení konstrukce. Často se provádí simulace pro potvrzení či vyvrácení hypotézy.

Cílem této práce je zopakovat vybraný experiment rojové inteligence. Původní experiment probíhal v softwaru MATLAB. Pokus spočíval v definování virtuálního robota, který přenášel kostky do místa s větší hustotou těchto kostek. Za pomoci jednoduchých pravidel tomu tak bylo, ovšem robot znal přesně pozice všech objektů v simulaci.

Pokus by měl být proveden znovu avšak přímo ve vývojovém softwaru Microsoft Robotics Developer Studio (MRDS). Na vývoji MRDS se podílelo celkem 11 programátorů a výsledkem je software, kde veškerá komunikace probíhá asynchronně a jednotlivé části programu jsou od sebe odděleny. Tento koncept sice přináší výhodu zpracování asynchronních signálů a možnost komunikace služeb přes síť, ale značně komplikuje samotnou tvorbu požadované aplikace a její ladění.

1 JOYSTICK A JINÁ POLOHOVACÍ ZAŘÍZENÍ

Microsoft Robotics Developer Studio umožňuje již v základu připojit veliké množství různých polohovacích zařízení. Pro účely práce nebudou využity. Základní rozdělení těch nejběžnějších s popisem je uvedeno níže.

1.1 Joystick

Joystick běžně používaná PC periferie, používaná převážně nadšenci pro letecké simulátory. Rozdělení joysticků do tří kategorií dle ceny:

Obr. 1.1: Ilustrační foto joysticku, převzato z `<www.logitech.com>`

- Levné – tyto joysticky mají většinou pohyb ve dvou osách a posuvník pro ovládání plynu. Běžně mají více tlačítek, avšak nejsou programovatelná. [13]
- Střední – tato třída joysticků podporuje navíc od horizontální výchylky X a vertikální Y také natáčení ve směru Z. Rovněž oproti nižší třídě stále častěji podporují i programovatelná tlačítka.
- Vyšší třída – nejdražší kategorie kdy periferie má všechny výše zmíněné výhody. Podporuje technologii force–feedback, kdy zařízení klade hráči odpor podle toho co se děje ve hře.

1.2 Gamepad

Gamepad je dnes běžně používaný u herních konzolí všech výrobců. Je ergonomicky tvarovaný, aby padl do ruky. Standardně obsahuje směrová tlačítka pro pohyb. Tlačítko `start` a `select` a 4 herní tlačítka. Moderní gamepady nahrazují směrová tlačítka analogovou páčkou, ta je převzata z joypadu. Mohou také obsahovat směrová

tlačítka, ale přemisťují je na méně důležitou pozici ke středu. Samozřejmostí jsou tlačítka přístupná pro ukazováčky a to jak ve formě stiskací, tak analogové.

Obr. 1.2: Ilustrační foto gamepadu, převzato z <www.logitech.com>

1.3 Volant

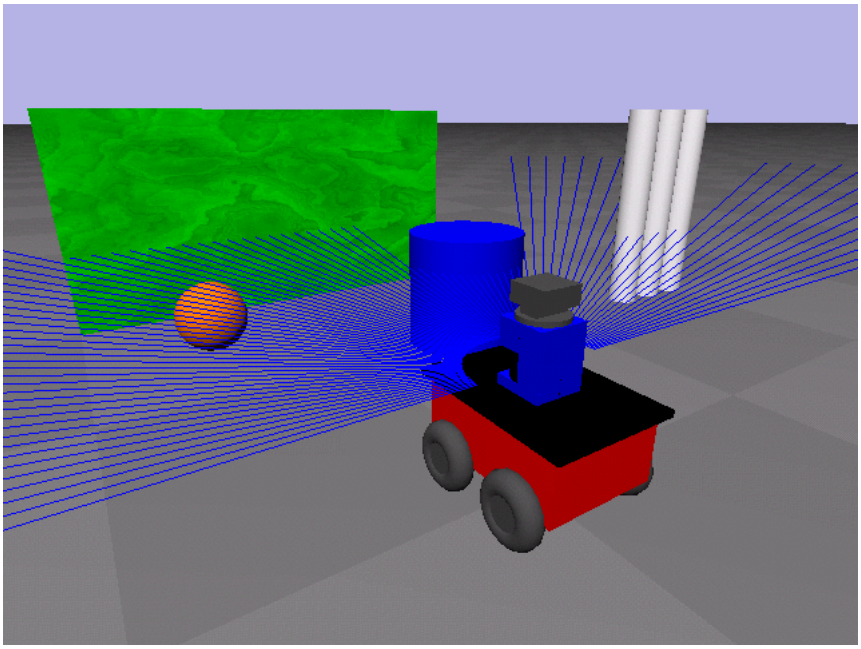
Volant je poslední z běžných zástupců herních periferií a většinou i nejdražší z nich. Primárním určením je přivést hraní závodních her trochu více k reálnému vozu, avšak mnohdy jeho použití zvyšuje obtížnost. Volanty mají většinou spoustu funkčních tlačítek, včetně speciálních za volantem pro řazení. V základu se vyskytuje plynový a brzdový pedál, ale existuje i klasická tří pedálová verze.

Obr. 1.3: Ilustrační foto volantu, převzato z <www.logitech.com>

2 SIMULÁTORY

2.1 Gazebo

Gazebo software vyvíjený pod licencí GNU[1] General Public License (GPL) pro 3D simulaci robotů s GUI v prostředí wxPython. Umožňuje simulovat většinu základních senzorů. Simulace obsahuje i fyziku, je možné zvedat předměty nebo do nich vrážet [9]. Na obrázku 2.2 je vidět náhled simulačního prostředí.



Obr. 2.1: Okno simulátoru Gazebo [9]

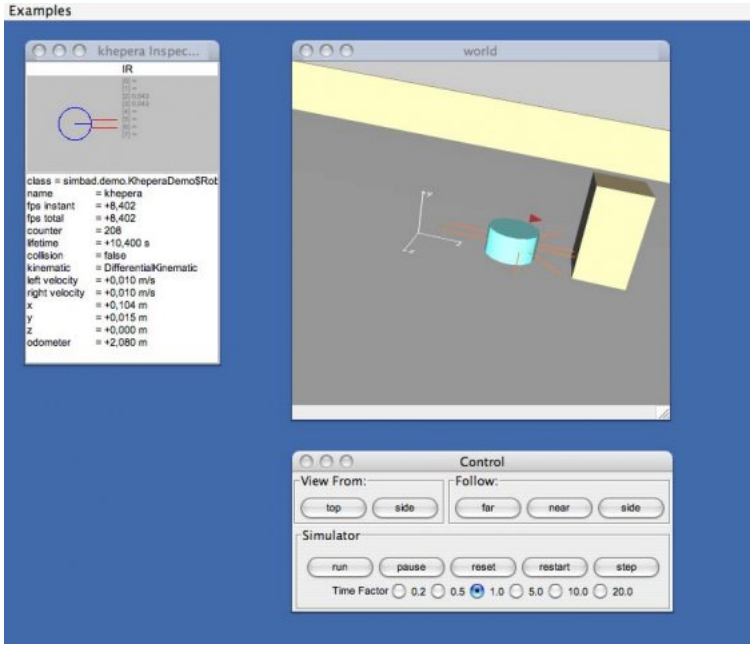
2.2 Microsoft Robotics Developer Studio

Vývojové studio z dílen Microsoftu, šířeno zdarma. Umožňuje prostorovou simulaci robotů s aplikací fyziky. Pro běh využívá technologie DirectX, PhysX, CCR a DSS. Více o tomto simulátoru v kapitole 3.

2.3 Simbad

Simbad je 3D simulátor psaný v Javě. Primárně určen pro výzkumné a studentské účely. Vzhled prostředí je vidět na obrázku 2.2 nebo na stránkách ze zdroje [5].

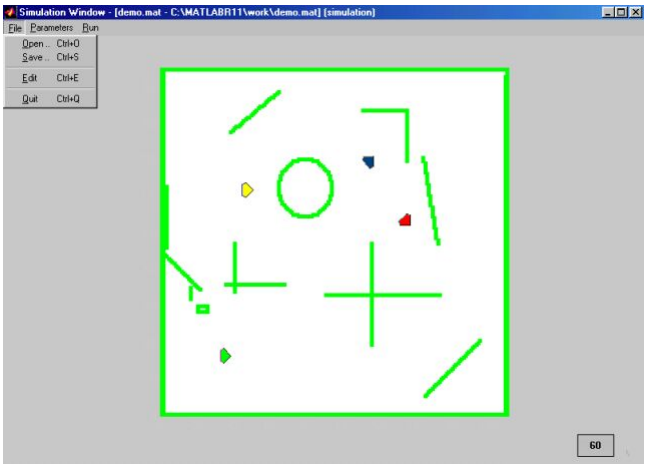
[1]GNU – akronym znamenající GNU is Not Unix



Obr. 2.2: Okno simulátoru Simbad [5]

2.4 SIMROBOT

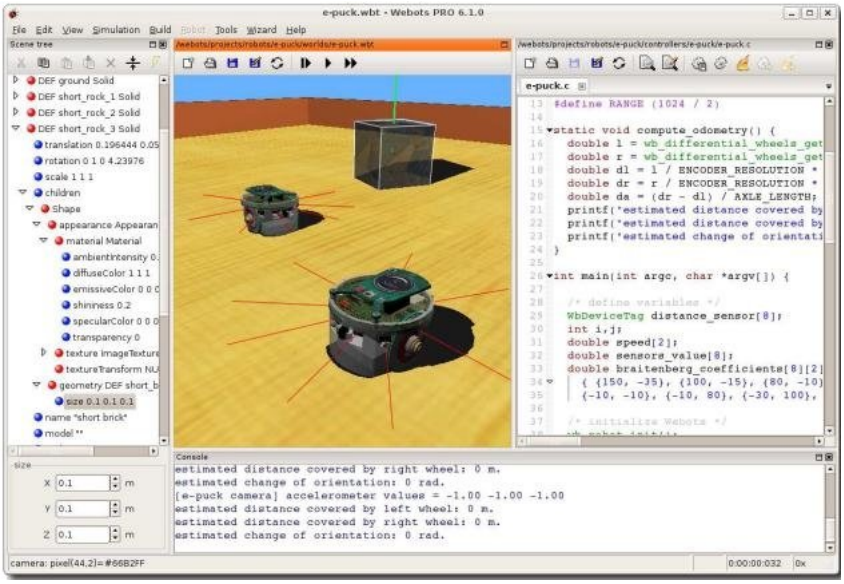
Toolkit pro Matlab, pro simulování autonomních robotů. Toolkit umožňuje každému robotu, přiřadit jiný algoritmus řízení. Vyvinutý na Ústavu automatizace VUT v Brně. Náhled simulátoru na obrázku 2.3 [14].



Obr. 2.3: Okno simulátoru SIMROBOT [14]

2.5 WeBots

WeBots je placený 3D simulátor robotů od firmy Cyberbotics. Podporuje asi nejvíce programovacích jazyků: C, C++, Java, Python, Matlab, URBI. Z náhledu na stránkách je vidět, že simulátor je na grafické úrovni velmi vysoko. Na přiloženém obrázku 2.4 je náhled [3].

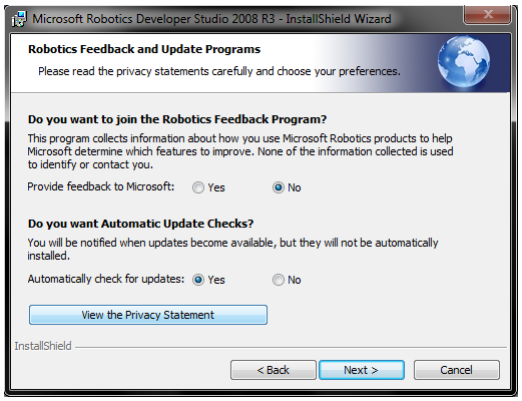


Obr. 2.4: Okno simulátoru Webots [3]

3 MICROSOFT ROBOTICS DEVELOPER STUDIO

3.1 Instalace MRDS

Před instalací je nutné stáhnout Microsoft Robotics Developer Studio přímo u tvůrce na webové stránce <www.microsoft.com/robotics/>. Po spuštění instalace naběhne základní okno, stiskem tlačítka `další` se zobrazí licence. Zde je potřeba potvrdit souhlas s licencí. Na obrázku 3.1 je následující okno instalace, kde jsou zobrazeny zvolené možnosti chování po instalaci.Tato nastavení jsou volena takto, protože většinu selhání MRDS zaviní chyby programátora, které vedou k nestabilitě aplikace. Tedy není nutné zasílat neustále logy o pádech MRDS.

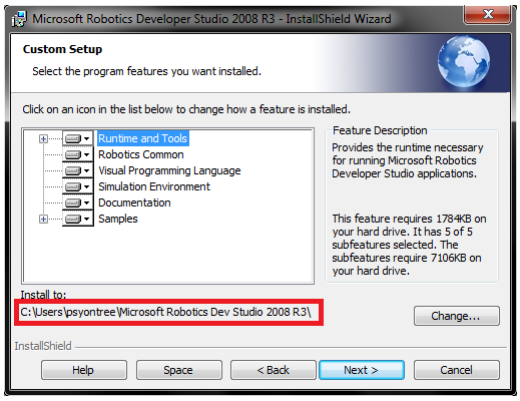


Obr. 3.1: Volba nastavení chování po instalaci

Na obrázku 3.2 je vyznačena předvolená cesta k instalačnímu adresáři v uživatelských dokumentech, tedy ne tak jak je obvyklé do složky `C:\Program files`. To je důležité, protože umístění adresáře MRDS je klíčové.

Po skončení instalace je potřeba doinstalovat součást CCR a DSS runtime, jejichž instalace je přiložena v podadresáři `redistributables`, který se nachází ve složce s instalací MRDS. Po nainstalování všech částí je doporučeno nechat systém aktualizovat. Protože některé balíky, které se instalují s MRDS obzvláště součásti platformy NET, mají již novější verze nebo vyšly záplaty.

Po provedení těchto kroků je potřeba spustit skript, který vygeneruje veškeré služby potřebné pro použití ve VPL. Tato možnost se provádí v konzoli, aby byl vidět výpis chybových hlášení. Tedy přes nabídku `Start` → `spustit` (vyhledat programy vista a win 7)→`cmd` a poté bude třeba pomocí příkazu `cd`(change directory), přejít do složky s MRDS do podadresáře `Samples`. Zde je potřeba spustit



Obr. 3.2: Cesta k instalaci

`buildallsamples.bat`, pár minut se bude kompilovat. Ideálně by výstup neměl obsahovat chyby (červená barva) ani varování (žlutá barva). Při výskytu chyby je třeba zkontrolovat, jestli je opravdu nainstalován balík s CCR a DSS, popř. jestli nechybí jiné části, jako například PhysX.

3.2 Základní vlastnosti MRDS

MRDS jako vývojové prostředí staví na moderních technologiích. Mezi tyto technologie patří zejména C#, jenž je součástí tzv. platformy NET. Dále knihovny DirectX obstarávající grafické vyobrazení simulace a v neposlední řadě PhysX obstarávající fyziku. První dva jmenovaní pocházejí z dílny Microsoftu, ovšem fyzikální část spadá pod společnost Nvidia, která je známá především tvorbou grafických karet.

3.2.1 C#

C# je základní stavební kámen MRDS a všechny zdrojové soubory jsou psané v něm. K dispozici jsou již hotové součásti, ihned použitelné k sestavení simulace. Rovněž je možné libovolně upravovat a používat upravené služby, vycházející z dodaných originálů. C# je jeden z mnoha vyšších programovacích jazyků a byl zvolen zcela logicky, neboť pochází z dílen Microsoftu a v dnešní době se stává mnohem více populární. Pro tvorbu zdrojových kódů je vhodné použít Microsoft Visual Studio, které podporuje vývoj aplikací jak v klasickém C, tak C++ a požadovaném C#. Zároveň se do Visual Studia přidají šablony pro tvorbu služeb. Po instalaci studia je vhodné zvolit možnost, že budeme programovat C#.

3.2.2 DirectX

Vykreslovací rozhraní je rovněž z dílen Microsoftu. Současná aktuální verze je DirectX 11, jenž přišel s nástupem operačního systému Windows 7. Nicméně MRDS využívá starší verzi a to DirectX 9.c podporovanou většinou grafických karet. Toto starší rozhraní splňuje všechny požadavky na grafické vyobrazení modelu. Zároveň umožňuje vyvíjet aplikace i pod starším operačním systémem, dnes stále ještě oblíbeným Windows XP. Novější verze přináší některá zásadní vylepšení, jako například přenos vybraných výpočtů z procesoru na grafickou kartu. To je dáno určením grafických karet pro zpracování obrazových dat, kdy se hardware hodí pro zpracování fyzikálních výpočtů mnohem více než procesor. Zároveň nemusí hardware splňovat zpětnou kompatibilitu, tu zajišťují ovladače v systému.

3.2.3 PhysX

PhysX je technologie spadající pod společnost Nvidia, původně vyvinutá společnosti Ageia. Společnost Ageia vyvinula akcelerační zásuvnou kartu, která byla určena pouze k fyzikálním výpočtům(výpočty s plovoucí čárkou). Karta neměla valný úspěch vzhledem k vysoké ceně a nezájmu vývojářů. Zároveň byla úzce specializovaná na konkrétní problém, který nebyl pro uživatele zásadní, bylo možno počítat fyziku procesorem. V důsledku úpadku byla společnost odkoupena firmou Nvidia. Ta získala veškeré technologie a měla před sebou již hotový výtvor. Bylo jen třeba jej dostat na trh. Od této chvíle začala Nvidia podporovat akceleraci fyzikálních výpočtů přímo na svých grafických kartách, kde tuto možnost zpřístupňovaly nové ovladače. Všechny grafiky od řady GeForce 8600 již podporují akceleraci pomocí PhysX. Tím vzniká problém, který nastává v situaci, kdy není v systému nainstalovaná grafická karta GeForce od Nvidie, ale jiná od firem AMD (ATI), Intel, VIA, Matrox nebo SiS. V tuto chvíli se musí PhysX počítat softwarově na CPU, což znatelně zatěžuje procesor a v případě her snižuje hratelnost. Navíc softwarová verze je záměrně omezena na jedno vlákno. V případě MRDS to není taková tragédie, ale nasazením jiné technologie by umožňovalo použití akcelerované fyziky na libovolném hardwaru. Mezi možné varianty by se dal zařadit engine Havok, spadající pod společnost Intel.

4 JEDNOTLIVÉ MODULY MRDS

4.1 CCR

Kapitola čerpá ze zdroje [6]. Concurrency and Coordination Runtime (CCR) řeší potřebu správy asynchronních událostí v aplikacích tvořených pomocí služeb. Řeší souběžnosti a použití paralelního hardwaru a vypořádává se s částečným selháním. To umožňuje uživateli navrhovat aplikace tak, že softwarové moduly nebo komponenty mohou být volně spojeny, což znamená, že mohou být vyvíjeny nezávisle a není třeba řešit konkrétní prostředí. Tento přístup umožňuje nový pohled na návrh programu s ohledem na souběžnost a korektní řešení chyb.

Problémové oblasti Oblasti, které při programování vícevláknových aplikací dělají největší potíže.

- **Asynchronnost** – Při komunikaci volně propojených programových celků, jako například programy běžící přes síť, nebo kód uživatelského rozhraní (UI) zajišťující vstup uživatele popřípadě obsluha souborového I/O subsystému. Asynchronní programování však výrazně snižuje čitelnost uživatelského kódu, protože logika je často rozdělena mezi zpětná volání a kód, který provádí operace. Je skoro nemožné správně zvládnout chyby mezi několika zvlášť spuštěnými bloky.

- **Souběžnost** – Aby bylo možné spouštět více kódu současně, musí být rozdělen na nezávislé logické části, které mohou běžet paralelně, a komunikovat v případě, když mají předat výsledky. Kód je strukturován do dlouhých sekvencí, které využívají blokování nebo synchronní volání, a řeší jenom jednu věc najednou. Při klasickém programování dochází mezi vlákny programu, ke komunikaci primárně pomocí sdílené paměti, což nutí programátora používat přesně, k chybám náchylné metody pro synchronizování přístupu k této sdílené paměti.

- **Koordinace a ošetření chyb** – Koordinace velkých projektů se stává nesnadnou. Různé možnosti řešení, jako je volání metod v objektech, používání signálů operačního systému nebo fronta a signály, to vše vede na nečitelný kód. Kdy v důsledku je třeba použít pro každou chybu, jinou metodu ošetření.

  CCR je vhodné pro použití v aplikaci, která dělí řešení na bloky, které mohou komunikovat pouze prostřednictvím zpráv. Přístup je velice podobný komu-

nikaci přes síť. Jenom vše může probíhat i lokálně. Concurrency and Coordination Runtime prostředí by mělo pomoct řešit problém, kdy jednotlivé bloky programu se vykonávají jinou rychlostí např. I/O k hardwaru a uživatelské rozhraní.

4.2 DSS

Kapitola čerpá z [7]. Decentralized Software Services (DSS), je jednoduché, na .NET platformě založené runtime prostředí, které využívá pro svoji činnost CCR. DSS poskytuje jednoduchý, stavově orientovaný model služeb, který spojuje představu reprezentačních stavu přenosu (REST[1]) s komunikací na systémové úrovni. V DSS jsou služby zpřístupněny jako zdroje, které jsou přístupné jak programově, tak přes UI rozhraní. Díky integrovanému izolování služeb, strukturované manipulaci se stavy, upozornění na události, a přesnému návrhu služeb, DSS splňuje potřeby pro napsání výkonné, trasovatelné aplikace, kde jednotlivé části běží na jednom lokálním uzlu, nebo se vše realizuje přes síť.

Pro komunikaci slouží Decentralized Software Services Protocol (DSSP). Je to jednoduchý SOAP(Simple Object Access Protocol) protokol, který poskytuje čistý aplikační model pro symetrický přenos stavů s podporou pro manipulaci se stavy a model událostí řízený změnou stavů. Tento protokol rozšiřuje možnosti HTTP protokolu.

DSS programové prostředí poskytuje vývojovou část s podporou pro tvorbu služeb, publikování/podepsání, spravování jejich životnosti, bezpečnost, monitorování, logování, a ještě mnohem více jak lokálně tak i přes síť. Služby mohou být vytvořeny ve Visual Studiu, nebo pomocí VPL.VPL je možné použít pro tvorbu aplikace složením jednoduchých služeb pouhým přetažením z nabídky a jejich spojením na základě datových závislostí. K dispozici je i DSS Manifest Editor (DSSME), který obsahuje grafické prostředí pro spojování, konfiguraci, a spuštění DSS aplikace lokálně, tak i přes síť.

Problémové oblasti v DSS

- **Robustnost**
  Odolnost vůči chybám. Ta je zajištěna, dvěma způsoby a to kopírování veškerých dat(i zpráv) mezi službami, čímž se zabrání použití neplatných dat a poté principem DSS a to, rozčleněním řešení na dílčí celky. Tento koncept práce s

[1]Representational State Transfer

daty má být rychlejší, než plné provázání.

- **Modulovatelnost**
  Klasické aplikace jsou psaný jako jeden celek. V MRDS jsou vše jen moduly které někde běží. DSS poskytuje možnosti pro běh a komunikaci, takto tvořených programů využívajíce přitom DSSP. Ten poskytuje služby pro zjištění závislostí a pro komunikaci mezi službami.

- **Pozorovatelnost** (trasování)
  Ladění aplikace, je velmi důležitá vlastnost každého prostředí. Tu poskytuje i DSS, ovšem v podobě webového rozhraní. Klasické trasovaní, je možné taky, ale nepodařilo se jej v rámci práce zprovoznit.

4.2.1 DSSP protokol

Protokol využívaný pro aplikace běžící na DSS, kompletní popis je přístupný na `download.microsoft.com` [4].

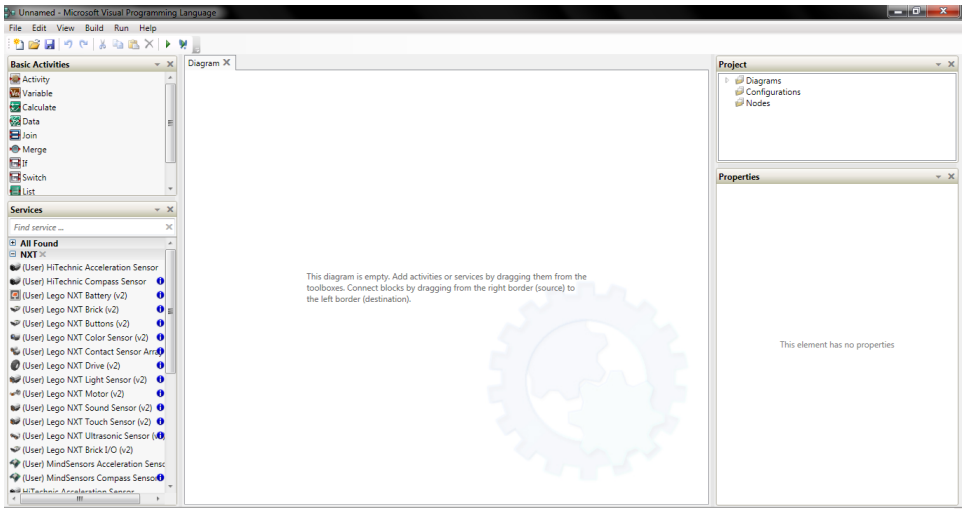
4.3 Microsoft VPL

VPL vytváří program pomocí diagramu, ve kterém se propojují hotové bloky(služby) do jednoho funkčního celku. Mezi službami je možné přenášet data a to jak ze služeb, tak přímo definované uživatelem. Například nastavení výkonu motorů robota napevno, provádí-li se daná část programu pošle se vždy stejná hodnota. Zpracování probíhá asynchronně a paralelně.

Na obrázku 4.1 je vidět hlavní okno VPL po prvním spuštění. V levé části okna se nachází výběr možných prvků a hotových služeb pro tvorbu diagramu aplikace. V pravé části je okno projektu společně s oknem vlastností, které vždy nabízí aktuální volby pro konkrétní výběr prvku nebo spojení. Kompilaci hotového programu lze vytvořit službu, kterou lze dále používat a lze ji najít také v nabídce. Při kompilaci vznikne také zdrojový kód v C#. Více o VPL v části tvorby jednoduché simulace na stránce 22.

4.4 Microsoft VSE

Microsoft Visual Simulation Enviroment (VSE) – Grafické simulační prostředí. Jedna z výhod balíku MRDS je grafické simulační prostředí postavené na technologii DirectX, jež je v oblasti grafiky zaběhlý standard a již zmiňované PhysX, která již sice není tak benevolentní k výběru hardwaru, ale stále umožňuje počítat fyziku



Obr. 4.1: Hlavní okno VPL

přes procesor. Tato stěžejní část simulátoru umožňuje zobrazit objekty, které musí být ve formátu `obj`, či `x`. Možný je také import a následné převedení objektů definovaných v `xml` souboru, nebo použít soubor typu `collada`. Poslední jmenovaný, je možné vytvořit v modelačním prostředí Blender [12]. Další možností je export modelu z prostředí Solid Works [11]. Možností tvorby modelu je mnohem více, tohle jsou jen některé.

- Minimální požadavky: Grafická karta podporující DirektX 9.0c a vyšší a minimálně shader model 2 s aspoň 64 MB grafické paměti.
- Doporučené požadavky: Grafická karta podporující DirectX 9.0c a vyšší, shader model 3+, 128 MB grafické paměti.

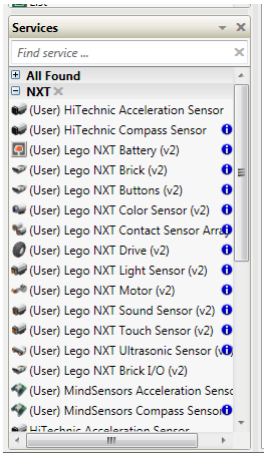
Pro fyziku počítanou přeš GPU je vyžadovaná grafická karta společnosti Nvidia GeForce 8 minimálně s 256 MB grafické paměti. Minimální požadavky získány z [8].

5 SIMULACE

Nejjednodušší variantou jak vytvořit simulaci je využití Microsoft Visual Programming Language. Tvorba simulace je jednoduchá a rychlá za pomoci již předpřipravených služeb od vývojářů MRDS. Ovšem pro pokročilou tvorbu či úpravu některých modulů je nutné editovat zdrojové kódy v C#.

5.1 Tvorba simulace pomoci Microsoft Visual Programming Language

Spuštění programu Microsoft Visual Programming Language lze provést přes nabídku `start` → `Microsoft Robotics Developer Studio` → `VPL`. Po spuštění naběhne okno vývojového prostředí jak je vidět na obrázku 4.1 . Při tvorbě jednoduché simulace jsou k použití dodané hotové služby.Tyto se nacházejí po levé straně okna, jak je vidět na obrázku 5.1.



Obr. 5.1: Nabidka sluzeb ve VPL

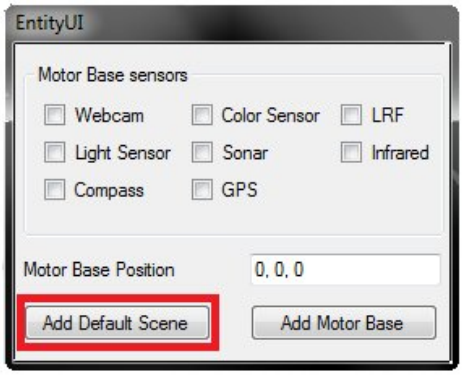
Prefix `(user)` u jednotlivých služeb znamená, že se jedná o služby zkompilované uživatelem. Tyto vznikly při spuštění příkazu `buildallsamples.bat`, spuštěného po instalaci MRDS v části 3.1. Toto rozlišení je tvůrci přidáno z důvodu odlišení původních dodaných služeb od upravených.

Postup tvorby ve VPL:

- Pro robota s differenciálním podvozkem je nutné do diagramu umístit službu `GenericDifferentialDrive`. Tato služba obstarává simulaci diferenciálně ří-

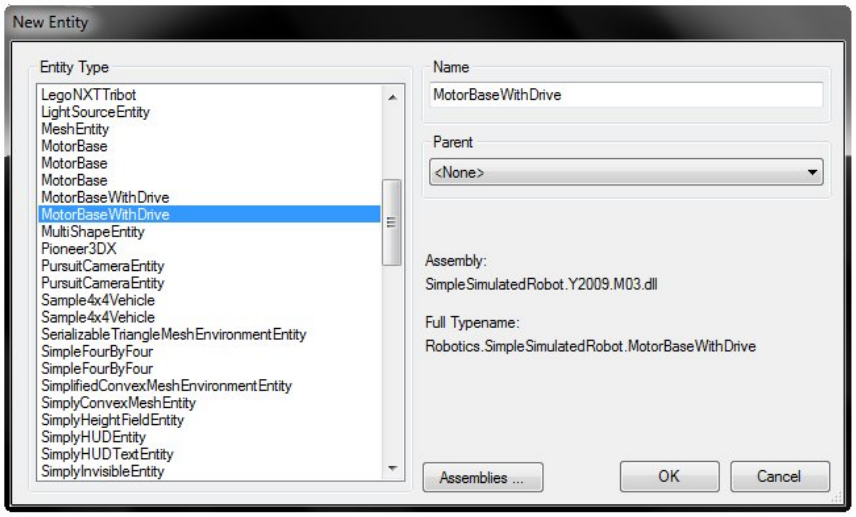
zeného robota. Zároveň je součastí i partnerská služba `SimulationEngine`, tedy není nutné ji pro funkčnost robota přidávat.

- Pro řízení robota jsou použitelné tyto služby: `DesktopJoystick`, `SimplyDashboard` nebo `GameController`. Poslední jmenovaná umožňuje pouze použití hardwarového joysticku, první zase čistě vytváří softwarový joystick. Služba `SimplyDashboard` kombinuje vlastnosti obou zmiňovaných a přidává možnost grafického zobrazení výstupu laserového snímače.
- Nakonec se vloží i služba `Simulation engine`. Pro běh není potřeba, ale pro úvodní tvorbu je nezbytná. Zajistí naběhnutí prostředí VSE, které obsahuje i editační nástroje pro tvorbu scény. Nyní je třeba vše uložit a spustit.
- Po spuštění naběhne okno VSE. Celé černé, neboť simulace neobsahuje žádné entity a to ani svět či světlo. Pro editaci je třeba přepnout VSE do editačního režimu a to kliknutím na `mode` → `edit`. Nyní je potřeba přidat do simulace svět. To se provede přes menu kliknutím na `Filo->Open manifest`. Hledaný manifest `entity.UI.manifest.xml` se nachází v podadresáři `Samples` → `config` v domovské složce MRDS. Po načtení manifestu se zobrazí okna na obrázku 5.2. V okně jsou volby, které umožňují vložit do simulačního prostředí scénu a diferenciálního robota se senzory dle nabídky. Pro vložení čistě simulační scény slouží tlačítko `Add Default Scene`. Po jeho stisku se do simulace přidá země, obloha a světlo.

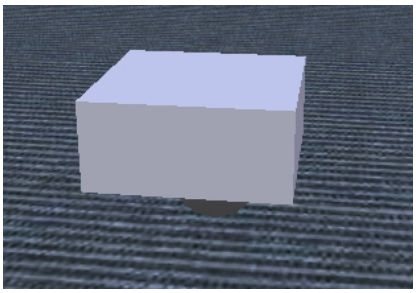


Obr. 5.2: EntityUI

- Nyní je v simulaci vytvořeno prostředí. Je tedy potřeba přidat robota. To se provede přes menu VSE, kde je potřeba zvolit `Entity` → `New`. Tím se otevře okno na obrázku 5.3 s nabídkou hotových entity, připravených k vložení do simulace.
  Pro volbu robota je možno zvolit vícero možností, ale pro demonstraci postačí základní `MotorBaseWithDrive`(osvědčila se druhá položka, první občas nefungovala). Jedná se o diferenciálního robota zobrazeného na obrázku 5.4.

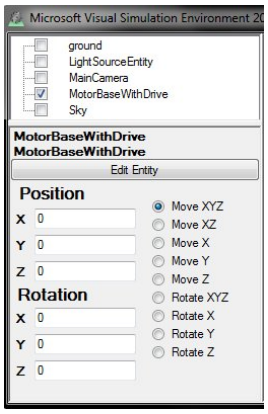


Obr. 5.3: okno New Entity

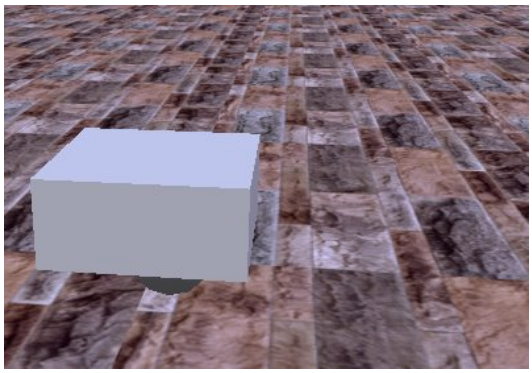


Obr. 5.4: RobotBaseWithDrive

- Nyní by v levé části okna měl být seznam vložených entit, jak je vidět na obrázku 5.5. Tyto vložené entity je možné dále editovat a to označením dané entity a kliknutím na volbu `Edit Entity`.
  V editačním okně je mnoho nastavení od barvy a tvaru až po fyzikální vlastnosti, jako např. učinit objekt kinematický. Je možné měnit i textury,kdy ve výsledku může simulace mít na podlaze kachličky nebo může vypadat jako na obrázku 5.6. Nabídka natavení entity, je pro každou z nich unikátní, stejné entity mají stejné volby(pokud jsou tvořeny identicky), ale vnořováním entit jedná do druhé, lze dosáhnout velmi nepřehledného menu.
- Důležitou možností je změnit pozici a natočení entit v simulaci, v zásadě existují dvě. Ta první a na jednoduché přesouvání snazší je po výběru entity a stisku tlačítka `Ctrl`, jednoduše entitu v simulaci přemístit myší. Ta druhá pro některé těžší avšak přesnější možnost, je měnit souřadnice zobrazené pod seznamem entit jak je vidět na obrázku 5.5. Druhou variantou lze měnit i



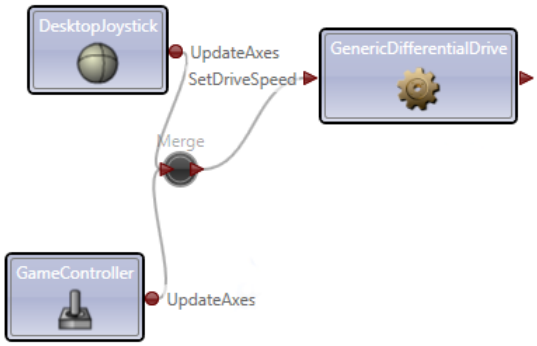
Obr. 5.5: Seznam entit



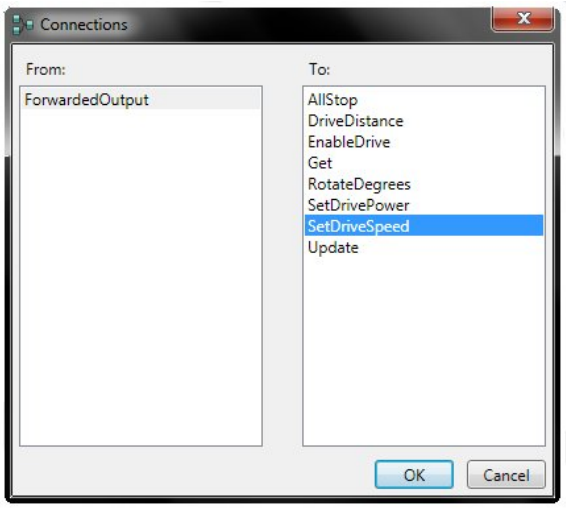
Obr. 5.6: změna textury podlahy

natočení ve všech osách.

- Tímto postupem vznikla jednoduchá simulace obsahující scénu a robota. Vše je třeba uložit a to následujícím způsobem `File` → `Save scene as` a jako složku pro uložení zvolit `config`. Tato složka je defaultní a manivest by měl být potom jednoduše nabídnut. Nyní se vše musí vypnout, aby došlo k načtení nového manifestu a bylo jej možné použít v nastavení služeb v diagramu.
- Po opětovném spuštění VPL je třeba otevřít uložený diagram `File` → `Open`. Po načtení již není potřeba služba `SimulationEngine`, takže bude odstraněna.
- Nyní je třeba služby mezi sebou vhodně propojit. A to způsobem jaký je uveden na obrázku 5.7. Jako první je třeba vložit do simulace prvek `merge`, ten se nachází v nabídce aktivit vlevo nahoře. V diagramu se nyní nachází tři služby a jedna aktivita. Jednotlivé služby mají výstupy na pravé straně bloku a jsou dva typy. Typ jedna je `ResponsePort` ten slouží k posílání odpovědí, jestli byla přijatá zpráva zpracovaná a jestli úspěšně nebo ne. Druhý typ je `notification` a slouží k posílaní dat ze služby. Data můžou mít jakýkoliv

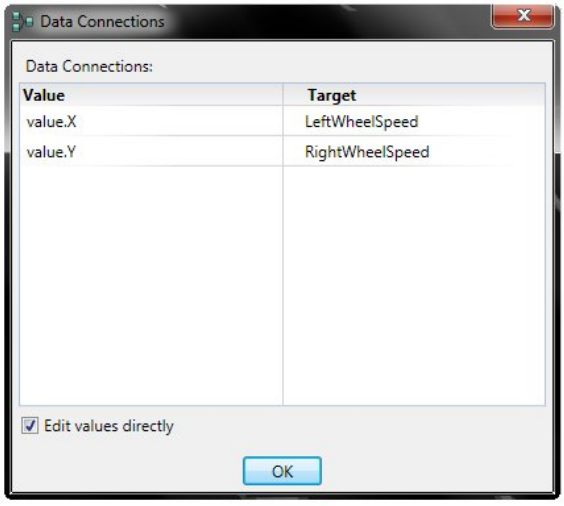


Obr. 5.7: Diagram propojení služeb ve VPL



Obr. 5.8: Volba datového propojení

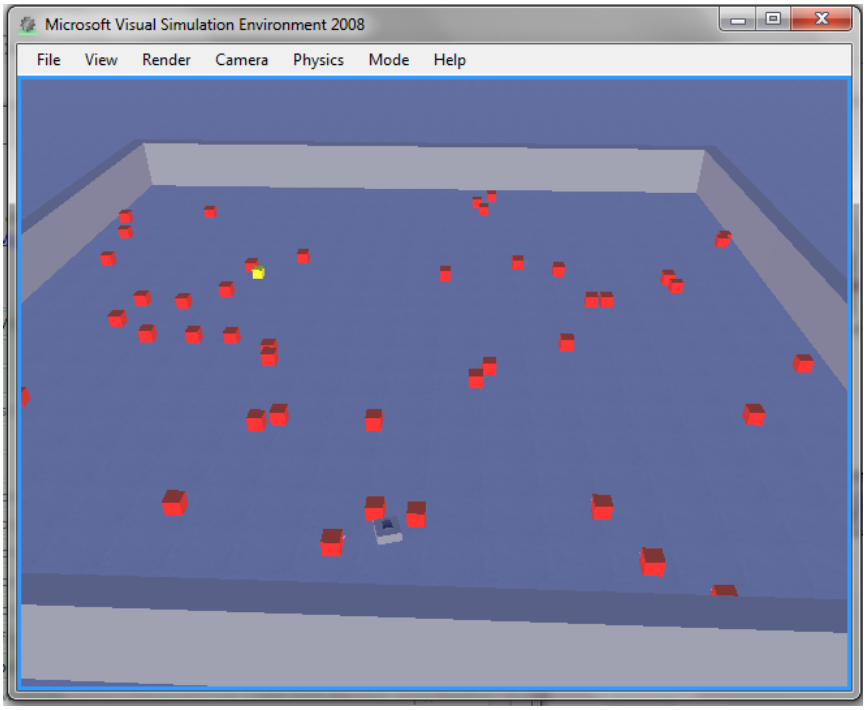
datový typ, ale většinou se jedná o jednoduché datové typy jako například `int`, `double`, `bool`, `float` nebo `string`. Ovšem je možné i narazit na pole těchto dat. Pro vyvedení dat ze služby `DesktopJoystick` a `GameController` slouží port `notification`. Klikneme tedy na něj a poté připojíme na uzel `merge`. Objeví se okno s volbou která data se budou posílat do uzlu `merge`. Je třeba zvolit `UpdateAxes` a potvrdit a rovněž toto provést i u druhé služby. Nyní je třeba spojit uzel `merge` se službou `GenericDifferentialDrive`, která se stará o motory robota. Po připojení výstupu z uzlu `merge` na vstup služby `GenericDifferentialDrive`, se opět objeví okno s volbou nastavení obrázek 5.8. Nyní ovšem je třeba se rozhodnout má-li být robot ovládán požadavky na výkon nebo na rychlost. Při požadavku na výkon je rozsah dat omezen na $\langle 0; 1 \rangle$ datového typu `float`. U požadavku na rychlost, sice tento limit není, ovšem nastavením velmi vysoké hodnoty se stane robot neovladatelným. Po připojení



Obr. 5.9: Volba přiřazení konkrétních dat na vstupy

na jeden z těchto portů se opět objeví okno s volbou, avšak už zde je již nutné přiřadit konkrétní položky(proměnné). Toto okno je zobrazeno na obrázku 5.9. Jelikož je třeba připojit data z joysticku, tedy údaje jednotlivých os, není možné je použít bez přepočtu. Osy jsou minimálně dvě `X` a `Y` a jsou přístupné jako surová data s rozsahem $\langle -1000; 1000 \rangle$. Zároveň vstupy do motoru jsou také rozděleny na dva. Levý a pravý kde vstup pro levý motor může vypadat například takto $\frac{Y+X}{1000}$, pro pravý motor například takto $fracY - X1000$. Tyto rovnice jsou platné za předpokladu, že osa `Y` nabývá kladných hodnot ve směru dopředu a osa `X` nabývá kladných hodnot ve směru doprava.

- Posledním krokem je přiřazení manifestu ke službám. V tomhle případě k jedné službě a to `GenericDifferentionalDrive`, kdy nejdříve kliknutím na službu se služba označí a na pravé straně dole v části `Properties` se objeví možnosti spouštění služby. Volbou `use manifest` se zvolí možnost načtení simulace ze souboru. Poté je potřeba kliknout na tlačítko `import manifest` kde v seznamu co se zobrazí je třeba vyhledat uloženou scénu, která byla vytvořena podle návodu výše. Nyní je třeba opět veškeré změny provedené v diagramu uložit. Po spuštění naběhne simulace s možností řízení robota joystickem, kdy simulace bude ve stavu v jakém byla scéna uložena a to i včetně nastavení kamery. Příklad hotové simulace je možné vidět na obrázku 5.10.



Obr. 5.10: Hotova simulace

6 ROJOVÁ INTELIGENCE

Jedná z mnoha kategorií umělé inteligence. U rojové inteligence(swarm inteligent), se vychází z biologických modelů v přírodě. Mezi vzory patří mravenci, termiti, husy,ryby, vosy, včely a mnoho dalších společensky žijících živočichů. Oblíbený je hmyz, protože je ho hodně, je malý a tak se s ním dobře pracuje. Základem rojové inteligence je myšlenka, že celek tvořen jednoduchými pravidly, se může ve výsledku chovat jako složitý, inteligentní systém. Jako vzor pro vysvětlení o co se jedná, dobře poslouží mravenci.

6.1 Optimalizace pomocí mravenčích kolonií

Ant Colony Optimization (ACO) je stochastický algoritmus pro řešení permutačních problémů. Klasickým příkladem chování mravenců je situace shromažďování potravy. Na začátku vyrazí mravenci do okolí mraveniště zcela chaotickým a náhodným způsobem, kdy hledají jídlo. V případě že jej najdou po cestě zpět do mraveniště vypouští feromon, který poslouží dalším jako vodítko, kde najít potravu. Mravenec, který hledá potravu a náhodně narazí na tento feromon se vydá po stopě až k potravě a po cestě zpět začne také vypouštět feromon, čímž se tato cesta stává stále více využívanou a je stále více značena. Jakmile dojde potrava, mravenci přestanou svými feromony označovat trasu a dojde přejdou opět do stavu náhodného hledání. Jednotliví mravenci tedy mají jen jednoduché pravidla, ale ve velkém měřítku dosahují velice dobrých výsledků.

U ACO je podobnost s mravenčí kolonií v těchto bodech [10]:

- kolonie kooperujících mravenců
- feromonová stopa
- nepřímá komunikace mravenců zprostředkovaná feromony (stihgmergy)
- pravděpodobnostní rozhodování, lokální strategie

Naopak v těchto bodech se algoritmus liší [10]:

- diskrétní svět
- vnitřní stav, paměť
- nejsou zcela slepí
- množství zanechaného feromonu může záviset na kvalitě nalezeného řešení
- možnost ukládání feromonu

ACO se používá pro následující úlohy ve kterých nalézá řešení například [10]:

- problém obchodního cestujícího
- směrování v sítích, navádění vozidel
- rozvrhování
- kvadratické přiřazování

- určení klasifikačních pravidel

6.2 Optimalizace Rojením částic

Particle swarm optimization (PSO) optimalizační algoritmus inspirovaný chováním ptáků. Je vhodný pro případy, kdy je potřeba nalézt nejvýhodnější polohu. Částice se pohybují v prostoru a hledají nejvýhodnější pozici. Zároveň komunikují s okolními částicemi s kterými si vyměňují informaci o nejlepší a zároveň mají k dispozici informaci o globální nejlepší pozici. Pokud je k dispozici lepší pozice, tak svou stávající opouští [1].

6.3 Vlastnosti původních experimentů

Původní experimenty probíhaly v matematickém prostředí MATLAB. Kdy cílem bylo přemístit puky z mnoha míst simulační scény, na jedno místo. Inspiraci hledali v mravenčím chování zvaném Stigmergy[1]. Pro tyto účely vypracovali čtyři možné metody, jak toho docílit [13].

Stigmergy s náhodným prohledáváním

Pravidlo 1.: **If** (robot nedrží puk)&(puk před robotem) **then** zvednout puk.

Pravidlo 2.: **If** (robot drží puk)&(puk před robotem) **then** polož puk, jeď zpět nějakou dobu a otoč se náhodným úhlem.

Pravidlo 3.: **If** Není žádný puk před robotem **then** jeď dopředu.

Pravidlo 4.: **If** Překážka v cestě **then** otočit pod náhodným úhlem a jet dopředu.

Robot jezdí po mapě dokud nenarazí na překážku nebo puk. Pokud narazí na překážku, aktivuje se 4. pravidlo a robot se začne otáčet dokud bude trvat stav detekce překážky, potom se rozjede dopředu. Pokud narazí na překážku, když drží puk, vyhne se překážce. Pokusí-li se zvednout více než jeden puk, přejde se do režimu pokládání, tedy pravidlo 2. Vyhýbání se objektům má větší prioritu než jejich pokládání.

[1]Stigmergy – zvláštní druh nepřímé komunikace používaný mravenci, kdy komunikace probíhá pomocí změn prostředí [2].

Stigmergy s vylepšeným zjišťováním koncentrace objektů

Pravidlo 1.: **If** (robot nedrží puk)&(puk před robotem) **then** zvednout puk.

Pravidlo 2.: **If** (robot drží puk)&(puk před robotem) **then** polož puk, jeď zpět nějakou dobu.

Pravidlo 3.: **If** (robot nedrží puk)&(žádný puk před robotem) **then** robot se vydá směrem, kde je nejmenší koncentrace puků.

Pravidlo 4.: **If** (robot drží puk)&(žádný puk před robotem) **then** robot se vydá směrem maximální koncentrace puků.

Pravidlo 5.: **If** Překážka v cestě **then** otočit pod náhodným úhlem a jet dopředu.

Stigmergy s navigací pomocí imunitních sítí

Pravidlo 1.: **If** (robot nedrží puk)&(puk před robotem) **then** zvednout puk.

Pravidlo 2.: **If** (robot drží puk)&(puk před robotem) **then** polož puk, jeď zpět nějakou dobu.

Pravidlo 3.: **If** (žádný puk před robotem)**OR**(překážka před robotem) **then** robot se vydá směrem, který určí umělá imunitní síť.

Imunitní síť obstarává směr kterým se robot vydá, když veze kostku směrem k maximální koncentraci a také vyhýbání se překážkám. Rozhoduje bude-li se provádět jeden nebo druhý úkon. Pokud robot drží kostku jede směrem k maximální koncentraci kostek. Pokud nedrží kostku jede směrem s minimální koncentraci.

Stigmergy s dvojicí nezávislých imunitních sítí

Tato metoda počítá s dvojicí na sobě nezávislých imunitních sítí IN1 a IN2. Jedna síť například IN1 má na starosti určení směru a druhá zvedání a pokládání puku.

Pravidlo 1.: Obě sítě pracují zároveň a nezávisle.

Pravidlo 2.: **If** (robot nedrží puk)&(puk před robotem)&(IN2 rozhodne, že se má zvedat puk) **then** robot zvedá puk.

Pravidlo 3.: **If** (robot drží puk)&(IN2 rozhodne, že se má položit puk)&(překážka před robotem)**OR**(puk před robotem) **then** polož puk, jeď zpět nějakou dobu.

Pravidlo 4.: **If** pokud nenastane pravidlo 2 nebo 3 **then** robot se vydá směrem, který určí IN1.

Cílem zakomponování dvou nezávislých imunitních sítí bylo, aby při jízdě směrem k minimální koncentraci puků byl robot schopen sebrat puk a při jízdě směrem k maximální koncentraci jej položit. Tento algoritmus umožňuje rozhodnout jestli se má puk podržet a pokračovat s ním dále nebo položit.

7 VYTVOŘENÝ PROGRAM

Program vytvořený pro tuto práci, kombinuje část vytvořenou v prostředí VPL a zároveň službu napsanou v jazyce C# vytvořenou ve vývojovém prostředí Microsoft Visual Studio 2010. Původním záměrem bylo vytvořit celý program čistě pomocí VPL, to se ovšem ukázalo jako velice problematické a tak došlo ke kombinaci obou metod. Tato varianta je schůdnější než pracovat čistě ve VPL, ale občas sebou přináší nepřehlednost.

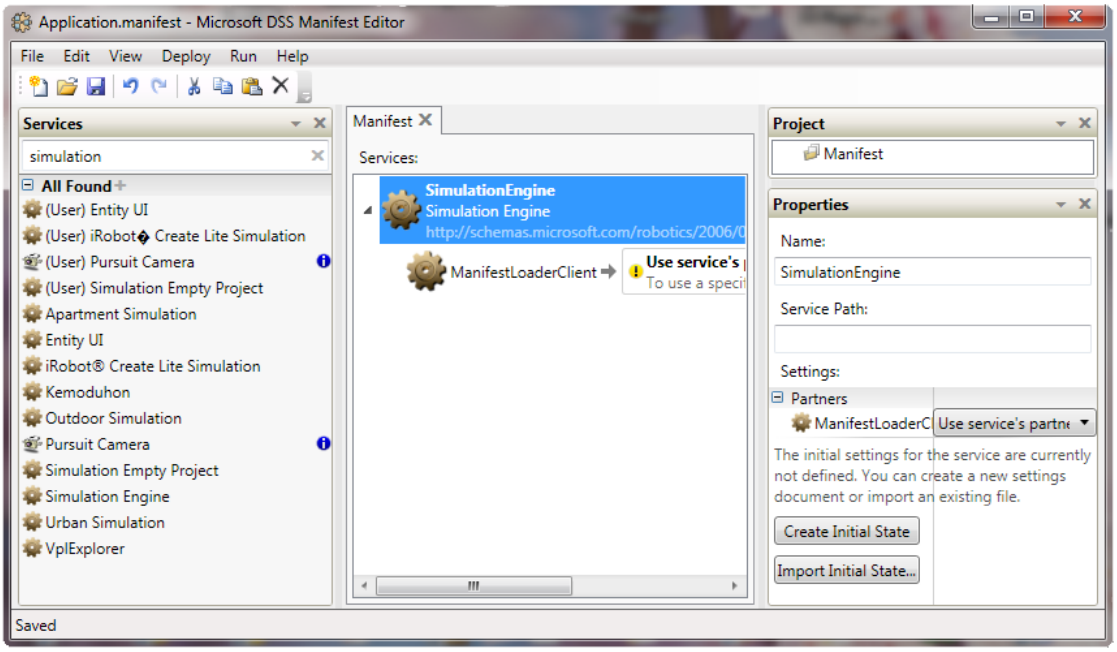
7.1 Součásti programu a jejich vazby

Program je tedy rozdělen do částí a to mezi VPL část a službu napsanou v C#. Tyto dva celky se dělí o obsluhu senzorů a řízení robota, kdy zpracování senzorových informací probíhá zcela ve VPL. O řízení robota se tyto dva celky dělí a to následujícím způsobem, kdy část kódu tvořená VPL obstarává náhodný pohyb při prohledávání simulace vytvořené pro robota, zároveň obstarává vyhýbání se objektům a náhodné natočení. Služba `Rizeni` vytvořená v C# obstarává řízení robota ve chvíli, kdy je zjištěna kostka před robotem a je potřeba aby se robot natočil na tuto kostku a poté ji zvednou. Zároveň obstarává i řízení robota ve chvíli kdy je třeba vyrazit směrem k maximu, drží-li robot kostku. Rovněž počítá maximum, které se využívá při řízení. Aby mohly tyto dva celky spolupracovat, je třeba vybavit vytvořenou službu potřebnými vstupy a také vytvořit port s výstupními daty, který je v diagramu VPL označován jako `notification`. Prostřednictvím těchto vstupů a výstupů je služba schopna komunikovat se službami které tvoří programovou část ve VPL. Nevýhodou je, že zásadní změny ve službě, jako například změny portů, vedou k nutnosti ji z diagramu odstranit, uložit diagram a vypnout prostředí VPL. Až po opětovném spuštění je možné vložit upravenou službu a začít ji používat. Tato nevýhoda spočívá hlavně v nutnosti znovu vytvořit spoje mezi službou a zbytkem programu tvořeném ve VPL. Z VPL se do vytvořené služby posílá informace o detekování kostky před robotem. Služba naopak posílá povely pro diferenciální podvozek robota.

7.2 Vytvoření simulace

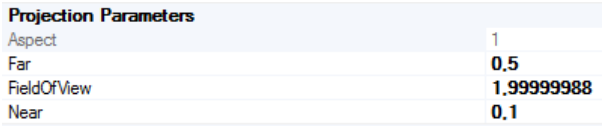
Základem programu je vytvoření simulační scény ve které se bude robot pohybovat. V téhle práci tvoří simulační scénu země, na které je vytvořena čtvercová plocha s hranou 20 m. V simulační scéně se nachází rovněž robot vybavený laserovým snímačem vzdálenosti a rovněž třemi snímači barev. O pohon se stará diferenciální

podvozek. Kostky které má robot přeskupovat se do simulační scény umisťují náhodně a tedy není možné je definovat v manifestu. Tvorba simulace se započne přes službu `SimulationEngine` a k tomu byl použit DSSME do kterého byla tato služba umístěna. Vzhled editoru je na obrázku 7.1.



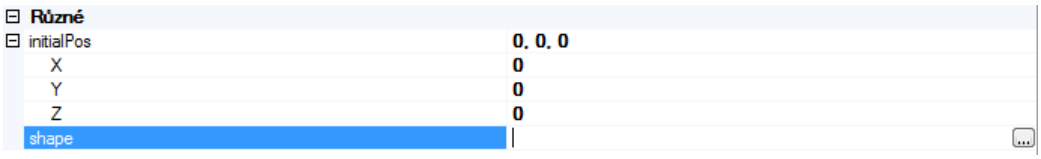
Obr. 7.1: okno DSSME

Simulace se spustí přes menu `Run`. Po naběhnutí VSE je třeba načíst manifest `entityUI.manifest.xml`, jak je popsáno v kapitole 5.1. Po spuštění `entityUI` je třeba nechat vložit do simulace defaultní scénu. A také robota s laserovým snímačem a snímačem barvy označovaným jako `SimulatedColorSensorEntity`, při pokusu o ruční vložení systém zahlásí špatný parametr a snímač barvy se nevloží. Tyto snímače jsou na robotu tři, aby bylo možné snímat přítomnost určité barvy ve směru vpřed, 45° nalevo a 45° napravo. Toho se docílí zkopírováním entity a to tak, že se označí entita `SimulatedColorEntity`, která je vnořena v entitě `MotorBaseWithDrive`. Po označení se klikne v menu na `Entity→copy` a poté je třeba označit znovu entitu `MotorBaseWithDrive` a zvolit `Entity→paste as Child`. Tím se vloží jedna entita do druhé. nastavení úhlu se provádí v levé spodní části `VSE` v kategorii `orientation`, kde se změní úhel na ose `Y` na -45°. To samé se udělá pro druhý snímač ale hodnota úhlu bude 45°. Dále je potřeba nastavit vzdálenost do které je snímač schopen rozpoznat barvu. Předdefinovaná je 5000 m, tu je potřeba změnit na 0,5 m. Toto lze provést v nastavení entity v sekci `Projection Parameters`, jak je vidět na obrázku



Obr. 7.2: Projection Parameters

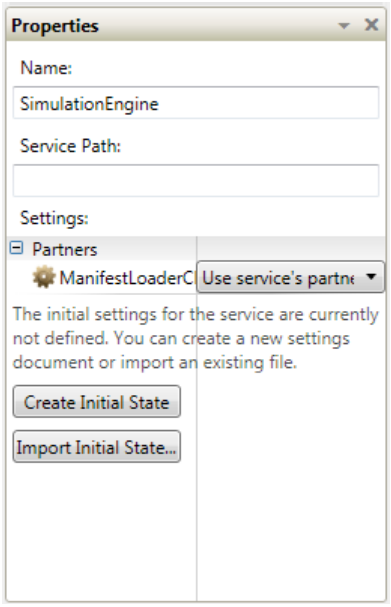
Je třeba také umístit laserový snímač níž. To z důvodu aby nemusely být detekované objekty tak velké. Toho se docílí změnou parametru `Y` v části `Position`. Do simulace je třeba ještě přidat ohraničení a tím vytvořit ohraničený prostor pro robota. Toto se provede přes `Entity`→`New`. Poté v seznamu entit vybrat `SingleShapeEntity`. Otevře se okno s volbami obrázek 7.3 kde je možné nastavit pozici, ale v části `shape` na obrázku modře vyznačená, se nastaví typ a velikost a jiné parametry, pokud není do tohoto menu nahlédnuto, automaticky se vytvoří koule.



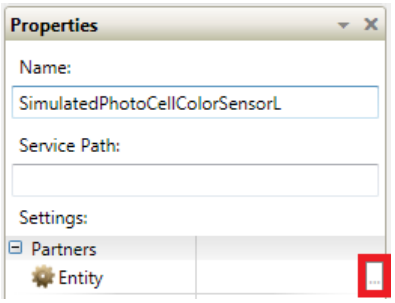
Obr. 7.3: NewSingleShapeEntity

Pomocí těchto jednoduchých entit se vytvoří ohraničení hřiště, kde změna velikosti se provede v nastavení entity v části `Dimension`, která je podmenu `State`. Po dokončení celé scény je potřeba zvolit možnost `File`→`Save Scene as` a uložit simulaci pod nějakým názvem. Po zavření VSE je potřeba v DSSME kliknout na `SimulationEngine` a poté zvolit v okně na obrázku 7.4 možnost `Import Initial State`, následně vybrat soubor s předvytvořenou scénou, kdy je třeba zvolit názevscény.manifest.xml. Tento soubor by měl mít větší velikost, než soubor podobného názvu.

Následně je třeba vložit veškeré služby do DSSME, tedy služba `GenericDifferentialDrive`, 3 × `SimulatedPhotoCellColorSensor`, `SimulatedLaserRangeFinder`. Po přidání služeb je třeba ještě nastavit, které entity simulace ke které službě patří. To se provede tlačítkem zvýrazněném na obrázku 7.5. Toto nastavení je třeba provést pro každou službu zvlášť. Pro rozlišení jednotlivých snímačů barev byly na téže místě změněny jejich názvy, tyto názvy jsou viditelné i v diagramu v programu VPL. Po dokončení je třeba takto vzniklý manifest uložit.



Obr. 7.4: Import nastavení do `SimulationEngine`

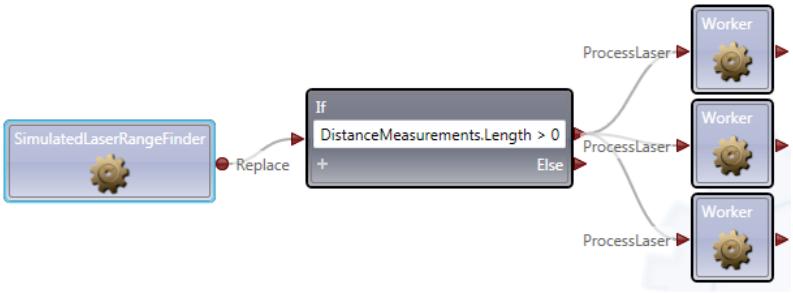


Obr. 7.5: Přiřazení entity k službě

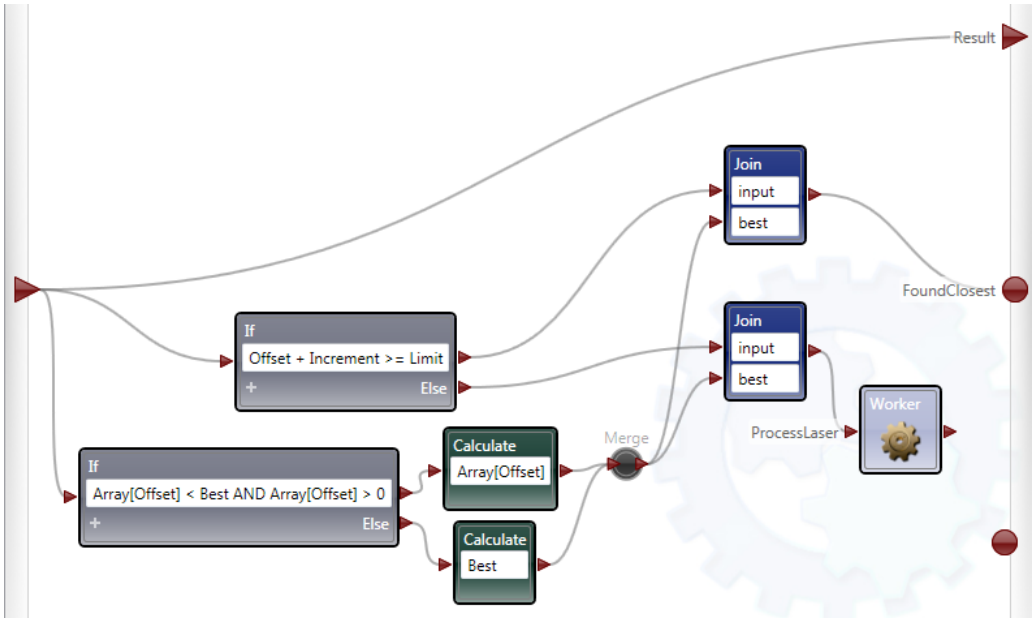
7.3 Vytvoření programu ve VPL

Program je z části vytvořen v VPL, kdy tato část obsluhuje vyhodnocení údajů ze senzorů, prohledávání mapy robotem a vyhýbání se překážkám. První částí je laserový snímač, který vyhodnocuje vzdálenost okolních objektů. Na obrázku 7.6 je vidět propojení služeb, kdy pro zpracování je vytvořena služba `worker`.

Data se posílají ze snímače přes rozhodovací blok if, kde se ověřuje, jestli snímač posílá data.Poté vše putuje do třikrát kopírované služby `worker`. Služba `worker` zpracovává vždy jen jednu třetinu rozsahu laserového snímače, tím se docílí rozčlenění na tři sektory nalevo, vepředu a napravo. Kdy jediná věc, co se liší je nastavení vstupních proměnných jednotlivých služeb, konkrétně offset, který službě řekne, odkud má začít. Vnitřní realizace služby `worker` je na obrázku 7.7.



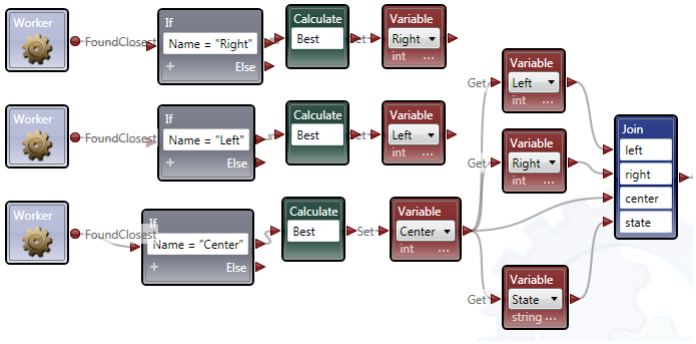
Obr. 7.6: Zpracování dat z laserového snímače



Obr. 7.7: vnitřní zapojení služby `worker`

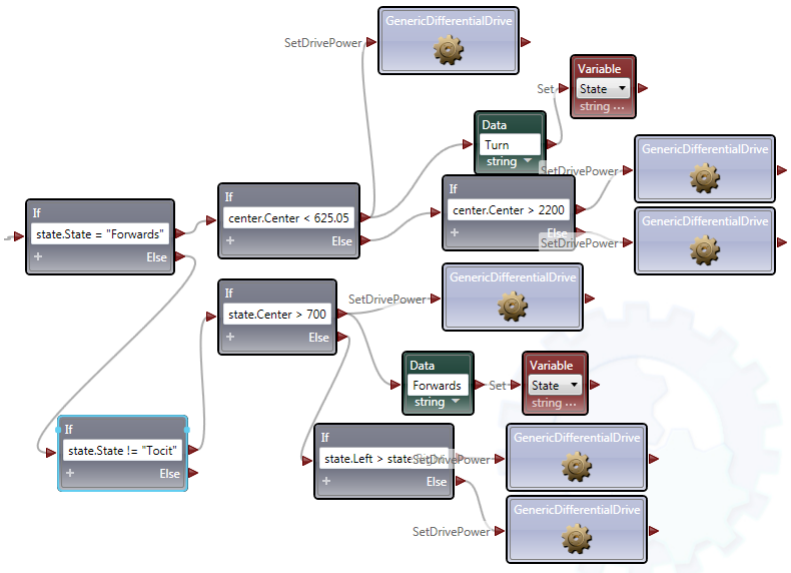
Jak je vidět na obrázku 7.7 data ze vstupu se okamžitě posílají na výstup, zároveň probíhá uvnitř služby zpracování. Kdy na vstupu jsou dva bloky if, jeden kontroluje, jestli příští iterace neposune službu do místa kde už zpracovává data jiná a pokud ano, tak pošle výsledky. V opačném případě se pošlou data s novým offsetem pro další zpracování. Druhý if prochází pole a hledá v něm nejmenší vzdálenost, která však nesmí být nulová. Pokud najde menší hodnotu než kterou zná, pošle novou, jinak pošle známou nejlepší. Bloky `Join` slouží pro spojení více zpráv v jednu, kdy se čeká až dorazí obě zprávy.

V předchozí části program vyhledal nejbližší vzdálenosti pro každý sektor. Tedy jsou dostupné celkově tři čísla nesoucí informaci o nejbližších objektech před robotem. Nyní se výsledky roztřídí do správných proměnných a nakonec se spojí do jedné zprávy v bloku `Join`, jak je vidět na obrázku 7.8. Tyto data potom slouží robotu, ke



Obr. 7.8: Zkompletování informací o vzdálenosti a stavu

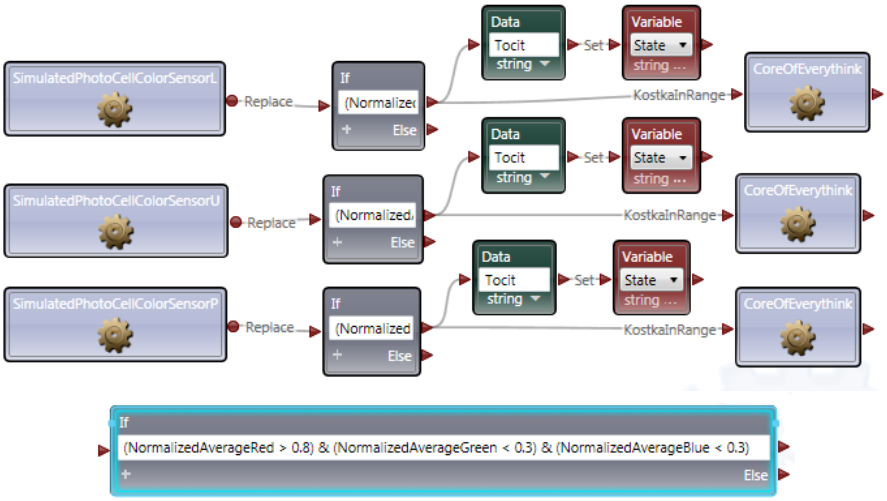
zjištění překážek a jejich vyhýbání se. Toho se docílí vytvořením vnitřního stavu, kde se uloží aktuální prováděný úkon. Pro realizaci jízdy po mapě náhodným způsobem slouží zapojení na obrázku 7.9 kde je vidět, jak pomocí stavu a jednoduchých `if` funkcí lze dosáhnout jízdy po mapě. Nejdříve se vyhodnotí v jakém stavu se nachází, potom se ověří centrální vzdálenost a na jejím základě se rozhodne, má-li se robot zastavit, otáčet nebo pokračovat kupředu. Kde otáčení probíhá ve chvíli, kdy robot přijede k překážce dostatečně blízko. Nejdříve se pošle do motoru povel zastavit, jinak by robot stále jel(vykonával by poslední zprávu) a zároveň se nastaví stav na `turn`. V dalším cyklu programu je simulace ve stavu `turn` a k vyhodnocení kterým směrem slouží porovnání dat z levého sektoru a pravého sektoru laserového snímače.



Obr. 7.9: Zapojení zajišťující jízdu po mapě

Další části je zpracování snímačů barvy, které slouží pro identifikaci, jeli objekt

před robotem kostka či nikoli. A jeli kostka pošle informaci do služby `Rizeni` na port `kostkainrange`. Toto zpracování probíhá na základě vyhodnocení barvy pomocí funkce `if` kdy ovšem je nutné vytvořit možný rozptyl barev, který je dán světelnými podmínkami v simulaci a parametry materiálu, z kterého je kostka tvořena. Na obrázku 7.10 je vidět toto zapojení, včetně obsahu bloku `if`.



Obr. 7.10: Zapojení zajišťující vyhodnocení barvy okolních objektů

Poslední součástí je výstup ze služby `Rizeni` který posílá zásahy do motorů robota. Počet zpráv byl radikálně snížen kontrolou jestli nový zásah bude jiný než starý, protože v opačném případě došlo k zahlcení, kdy se nestačily vybavovat pokyny do motoru a robot nebyl schopen splnit instrukce, které dostával na vstupu.

7.4 Vytvoření služby v C#

Služba vytvořená v C# řeší nejdůležitější úkony. Umožňuje vložit do simulace kostky určené k přemístění, rovněž získává entitu robota ze simulace, což je nutná podmínka k zjištění jeho pozice. Provádí se výpočty úhlů a samotné úkony jako je převezení kostky z jednoho místa na druhé.

Tvorba služby se započne pomocí Microsoft Visual Studia 2010, volbou nového projektu. Kdy šablony jsou dostupné jen pro psaní v C#, tedy je nutné zvolit jako programovací jazyk C# a poté vybrat volbu `Microsoft Robotics`. Po vytvoření základních zdrojových souborů je potřeba přidat do programu reference. To se provede v okně `Solution Explorer` kliknutím pravým tlačítkem myši na `Reference` a vyběrem volby `Add Reference`. Potřeba jsou tyto reference `PhysicsEngine` – umožňuje přístup k fyzikálnímu enginu, `RoboticsCommon`, `SimulationCommon` – definice typů,

`SimulationEngine` – přístup do simulačního enginu, `SimulationEngine.Proxy` – umožňuje načíst engine jako partnerskou službu.

Po přidání referencí je potřeba přidat deklarace oboru názvů

```
using physics = Microsoft.Robotics.Simulation.Physics;
using simtypes = Microsoft.Robotics.Simulation;
using simengine = Microsoft.Robotics.Simulation.Engine;
using Microsoft.Robotics.PhysicalModel;
using System.Numeric;
using Microsoft.Ccr.Adapters.WinForms;
```

Aby bylo možno získat entitu robota ze simulace je potřeba vložit simulační engine jako partnera.

```
[Partner("SimulationEngine", Contract = engine.Contract.Identifier,
CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]
private engine.SimulationEnginePort _engineServicePort =
    new engine.SimulationEnginePort();
```

Pro komunikaci se simulačním enginem slouží tyto proměnné:

```
simengine.SimulationEnginePort _simEngine;
simengine.SimulationEnginePort _notificationTarget;
```

Nyní je potřeba požádat simulační engine o entitu robota. Pro tento účel souží speciální objekt.

```
simengine.EntitySubscribeRequestType sub =
new simengine.EntitySubscribeRequestType();
sub.Name = "MotorBaseWithDrive1";
```

V proměnné `name` je uložen název entity v simulaci, jelikož simulace neumožňuje vložit dva objekty se stejným jménem, nemůže se stát, že by byl odeslán špatný objekt.

```
_simEngine.Subscribe(sub, _notificationTarget).Choice(resp => { },
       delegate(Fault f)
         {
         LogError(f.ToString());
         Console.WriteLine("nenalezena entita MotorBaseWithDrive1");
              }
        );
```

Požadovaná entity v tomhle případě robot se odešle na port `_notificationTarget`, pokud nebude nalezena odešle se do logu chybové hlášení. Aby entita byla správně spracovaná je třeba vytvořit Handler, který zpracuje příchozí data na portu.

```
Activate(new Interleave(new TeardownReceiverGroup (
     Arbiter.Receive<simengine.InsertSimulationEntity>(false,
      _notificationTarget, InsertEntityNotificationHandler)
      ),
      new ExclusiveReceiverGroup(),
        new ConcurrentReceiverGroup()
            ));
```

A v případě příchozích dat se volá funkce, která obslouží danou zprávu.

```
InsertEntityNotificationHandler(simengine.InsertSimulationEntity ins)
        {
         Robot = ins.Body;
         Robot.ServiceContract = Contract.Identifier;
        }
```

Proměnná robot v tomto algoritmu je typu `VisualEntity`. Po proběhnutí této části kódu je možné číst aktuální pozici robota příkazem

```
Robot.state.pose.position
```

Další důležitou součástí programu je vkládání entit do simulace

```
BoxShapeProperties cBoxShape = null;
SingleShapeEntity cBoxEntity = null;
cBoxShape = new BoxShapeProperties(0.1f, new Pose(),
new Vector3(velikost,velikost,velikost));
cBoxShape.Material =
new MaterialProperties("gbox", 0.5f, 0.4f, 0.5f);
cBoxShape.MassDensity.Mass = 1000;
cBoxShape.DiffuseColor = new Vector4(1.0f, 0.0f, 0.0f, 1.0f);
cBoxEntity = new SingleShapeEntity(new BoxShape(cBoxShape),
new Vector3(1.0f,2.0f,3.0));
cBoxEntity.State.Name = ‘‘Jmeno’’;
SimulationEngine.GlobalInstancePort.Insert(cBoxEntity);
```

Vytvoření entity, je složeno ze dvou částí, tvorba entity, kde se ukládá pozice ze simulačního engine a shape který obsahuje informace o vzhledu entity v simulaci.

Nejdříve se vytvoří vhled a velikost, potom se definuje materiál a jeho hustota. Následně barva. To vše se použije k tvorbě nové entity a přidá se souřadnice kde se entita bude nacházet, pojmenuje se a takto vytvořený objekt se odešle do simulačního enginu.

Aby bylo možno přijímat nebo odesílat data z VPL je potřeba vytvořit patřičné vstupní a výstupní porty. Tyto porty se definují v zdrojovém souboru `entityseznamTypes.cs`.

```
[ServicePort]
public class entityseznamOperations : PortSet<DsspDefaultLookup,
DsspDefaultDrop,
Subscribe, Replace, Uchop,Pozice,
KostkaInRange,Motory>
    {
    }
```

Každý port musí mít definované své datové typy, pro výstup `Motory` vypadá kód takto. Kdy první řádek je zobrazovaný název ve VPL, druhý je popis co se zobrazí jako nápověda. Port má data definované v třídě `MotoryData`.

```
[DisplayName("POWER TO ENGINE")]
[Description("Prenasi pokyny do motoru.")]
public class Motory : Update<MotoryData,
PortSet<DefaultReplaceResponseType, Fault>>
    {
    }

[DataContract]
public class MotoryData
{
 public double _motorleft;
 [DataMember]
 public double motorleft
 {
 get { return _motorleft; }
 set { _motorleft = value; }
 }

 public double _motorright;
 [DataMember]
```

```
 public double motorright
 {
 get { return _motorright; }
 set { _motorright = value; }
 }
}
```

Potom stačí ve službě vytvořit datový typ a odeslat vše na port.

```
MotoryData notifikace = new MotoryData();
SendNotification<Motory>(_subMgrPort, notifikace);
```

Aby bylo možné provádět vše periodicky je třeba vytvořit časovač, který se bude stále volat. Nejdříve se vytvoří objekt časovače.

```
private Port<DateTime> _timerPort = new Port<DateTime>();
```

Poté je třeba vytvořit handler který se spustí.

```
Activate(Arbiter.Interleave(
new TeardownReceiverGroup()),
new ExclusiveReceiverGroup
 (
 Arbiter.Receive(true, _timerPort, TimerHandler)
  ),
 new ConcurrentReceiverGroup()
  ));
```

Následuje vytvoření Služby která obstará zavolání ostatních služeb které chceme spouštět periodicky a zároveň vyšle zpožděnou zprávu na port.

```
  void TimerHandler(DateTime signal)
          {
SendPozice();
SendMotory();
Activate(
 Arbiter.Receive(false, TimeoutPort(50), delegate(DateTime time)
   {
   _timerPort.Post(time);
     })
  );
}
```

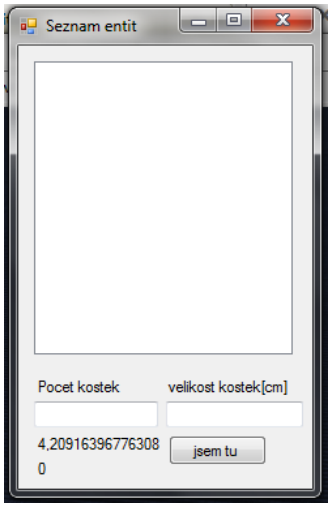
K prvnímu spuštění dojde zavoláním následujícího příkazu, poté se již bude cyklicky volat sám se zpožděním 50ms.

```
  _timerPort.Post(DateTime.Now);
```

V programu se používá výpočet úhlů, který směřuje ke kostce nebo virtuálnímu maximu, což je bod ke kterému mají okolní kostky nejblíže. Tento úhel je zvláštní tím, že je vztažen k ose z v záporném směru.

```
  public double angletocube()
{
 double smer = 0, udeltax = 0, udeltaz = 0,vdeltaz=0;
 udeltax = (chosencube.State.Pose.Position.X -
 Robot.State.Pose.Position.X);
 udeltaz = (chosencube.State.Pose.Position.Z -
 Robot.State.Pose.Position.Z);
 vdeltaz =-5;
 smer = Math.Acos((udeltaz * vdeltaz) /
 (Math.Sqrt(udeltax * udeltax + udeltaz * udeltaz) * 5));
   if (udeltax > 0)
   smer = 2 * Math.PI - smer;
               return smer;
          }
```

Uhel se přepočítává aby seděl s úhlem natočení, který je vracen robotem a přepočten na interval $\langle 0;2\pi \rangle$. Poslední důležitou částí programu je i vytvořené grafické rozhraní umožňující libovolně vkládat kostky do simulace s možností měnit počet a velikost. Zároveň poskytuje okno jednoduché kontrolní výpisy určené primárně pro kontrolu při vytváření služby. Toto okno je možné vidět na obrázku 7.11.



Obr. 7.11: Grafické rozhraní služby `Rizeni`

8 VÝSLEDKY PRÁCE

Experiment se nepodařilo zprovoznit až do úplného konce. Robot je schopen prohledávat mapu, detekovat kostky, natočit se na kostky, zvednout kostky, umí je i dovést a položit, ale už nezvládne po těchto úkonech otočit se náhodným směrem a jet pro další kostku, kterou by převezl. Dalším neduhem programu je vyčerpání paměťových zdrojů, kdy systém nedealokuje paměť, což může být způsobeno pozůstalými referencemi na data, popř nějakou neznámou chybou, kterou se mi nepodařilo odhalit. V případě že by robot dokázal v tuto chvíli splnit všechny úkony, nikdy by nedosáhl svého cíle, neboť by na to neměl dostatek času. K totálnímu vyčerpání paměti dochází přibližně po 20 minutách běhu, po té jsou programu odepřeny další systémové zdroje a padá.

9 ZÁVĚR

První část bakalářské práce se věnuje přehledu robotických simulátorů, které by bylo možné využít pro testování různých typů interpretů pákového ovladače (joysticku). Dále je vytvořen manuál popisující jednotlivé komponenty systému MRDS a postup při tvorbě nového projektu. Důraz je kladen na propojení dvou programovacích jazyků – jazyku C# a grafického jazyku Visual Programming Language (VPL). Jejich propojení výrazně zefektivňuje práci během vývoje.

Ve druhé části je nejdříve stručně definován pojem rojová inteligence. Na základě článku zadaného školitelem je popsán konkrétní experiment, jehož záměrem bylo simulovat a vyhodnotit chování multiagentního systému tvořeného agenty, které se řídí pouze několika jednoduchými pravidly. Během implementace této úlohy bylo nutné vyřešit řadu dílčích úkolů: vytvořit vhodné 3D prostředí pro simulaci, reprezentovat přenášené kostky a vyřešit jejich rozpoznání, uchopení a přenášení roboty, naprogramovat algoritmy chování robotů a definovat bod největší hustoty kostek.

Vzhledem k celé řadě komplikací způsobených nestabilitou MRDS a stále chybějící dokumentací nebyl experiment zcela dokončen. Robot je schopen jezdit v ohraničeném prostoru a vyhýbat se kolizím. Zároveň zvládne detekovat kostku jako cílový objekt a na tuto kostku se natočit tak, aby byl ke kostce čelem. Umí kostku zvednout a dopravit ji na místo k vypočtenému maximu, kde ji také odloží. Není implementovaná část, která by umožňovala robotovi dokončit tento úkol náhodným otočením a opět prohledávat scénu. V rámci řešení byla navržena metoda výpočtu bodu největší hustotou kostek, jejíž definice v předloženém článku chybí. Také byla navržena nová metoda detekce kostek, která je založena na odlišné barvě hledaných objektů. Zpracování barvy je snadnější než zpracování tvaru.

MRDS je volně šiřitelná aplikace od Microsoftu, což se projevuje velice obtížným sháněním informací potřebných pro rychlé vytvoření požadované aplikace. Většinu informací lze nalézt jen na stránkách komunity, která není nikterak velká a zbytek odvozením od kódů dodaných s MRDS. S lepší podporou ze strany výrobce by se MRDS mohlo stát velice oblíbené, protože dodává kompletní řešení zdarma.

LITERATURA

[1] Bc. TORAL, D.: *Skupinová spolupráce mobilních robotických strážních sytémů a jejich návaznost na činnost elektronických zabezpečovacích zařízení ve věznicích Vězeňské služby ČR*. Diplomová práce, Univerzita Tomáše Bati ve Zlíně,Fakulta aplikované informatiky, 2010, [online],[cit. 2011-05-24].
Dostupné z WWW: <http://dspace.knihovna.utb.cz/bitstream/handle/10563/12434/toral_2010_dp.pdf?sequence=1>.

[2] Bc. VACULÍKOVÁ, M.: *Ant Colony Optimization v prostředí Mathematica*. Diplomová práce, Univerzita Tomáše Bati ve Zlíně,Fakulta aplikované informatiky, 2008, [online],[2011-05-24].
Dostupné z WWW: <http://dspace.knihovna.utb.cz/bitstream/handle/10563/5817/vaculíková_2008_dp.pdf?sequence=1>.

[3] Cyberbotics: *Webots*. 2011, [online],[cit. 2011-05-23].
Dostupné z WWW: <http://www.cyberbotics.com>.

[4] Henrik Frystyk Nielsen: *Decentralized Software Services Protocol – DSSP/1.0* . 2007, [online],[cit. 2011-05-10].
Dostupné z WWW: <http://download.microsoft.com/download/5/6/B/56B49917-65E8-494A-BB8C-3D49850DAAC1/DSSP.pdf>.

[5] Louis Hugues, Nicolas Bredeche: *Simbad Project Home*. 2011, [online],[cit. 2011-05-23].
Dostupné z WWW: <http://simbad.sourceforge.net/>.

[6] Microsoft Corporation: *CCR Introduction*. 2010, [online],[cit. 2011-05-01].
Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/bb648752.aspx>.

[7] Microsoft Corporation: *DSS Introduction*. 2010, [online],[cit. 2011-05-01].
Dostupné z WWW: <http://msdn.microsoft.com/library/bb483056>.

[8] Microsoft Corporation: *VPL*. 2011, [online], [cit. 2011-04-20].
Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/bb483088.aspx>.

[9] Player/Stage: *Gazebo*. 2005, [online],[cit. 2011-05-23].
Dostupné z WWW: <http://playerstage.sourceforge.net/gazebo/gazebo.html>.

[10] Pošík P.: *Aplikace umělé inteligence*. 2010, [cit. 2011-05-24].

[11] SolidWorks Labs: *Collada export*. 2011, [online],[cit. 2011-05-24].
Dostupné z WWW: <http://www.blender.org/>.

[12] The Blender community: *Blender*. 2011, [online],[cit. 2011-05-24].
Dostupné z WWW: <http://www.blender.org/>.

[13] Tsankova D., Georgieva V., Zezulka F., Bradac Z.: *Immune network control for stigmergy based foraging behaviour of autonomous mobile robots. International Journal of Adaptive Control and Signal processing*, ročník 21, 2007: s. 265–286, doi:<10.1002/acs.915>.

[14] ÚAMT FEI VUT v Brně: *Simulátor autonomních mobilních robotů*. 2001, [online],[cit. 2011-05-23].
Dostupné z WWW: <http://www.uamt.feec.vutbr.cz/robotics/simulations/amrt/simrobot_cz.html#simulator>.

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

MRDS Microsoft Robotics Developer Studio

CCR Concurrency and Coordination Runtime

DSS Decentralized Software Services

DSSP Decentralized Software Services Protocol

VPL Microsoft Visual Programming Language

DSSME DSS Manifest Editor

VSE Microsoft Visual Simulation Enviroment

GPL General Public License

ACO Ant Colony Optimization

PSO Particle swarm optimization