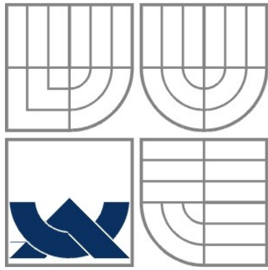
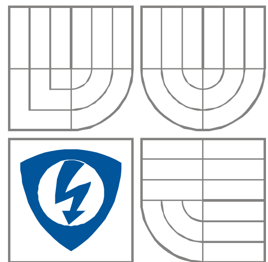


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF CONTROL AND INSTRUMENTATION

# ROBOTICKÝ FOTBAL V RDS

ROBOTIC SOCCER IN RDS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

TOMÁŠ KUPAROWITZ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETR HONZÍK, Ph.D.

BRNO 2011

# Zadání bakalářské práce

- 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 průběh simulace, způsob řešení událostí typu kolize, atd.;
- seznamte se s plánováním trajektorie pomocí vektorových polí (vector field path planning);
- implementujte plánování trajektorie do RDS a otestujte na jednom robotu.

# Abstrakt

Tato práce je zaměřena na tvorbu rešerše robotických simulátorů, vhodných pro simulaci hry robotický fotbal. Dále je v ní zvolen simulátor Microsoft Robotics Developer studio a v něm implementován algoritmus pro ovládání diferenciálně řízeného robota pomocí metody vektorových polí.

# Klíčová slova

Robotický fotbal, návrh trajektorie pomocí metody vektorových polí, RDS, MSRDS, Microsoft Robotic Developer Studio, robotické simulátory

# Abstract

The aim of this thesis is to create a list of robotic simulators suitable for simulating robotic soccer. Simulation environment included in Microsoft Robotic Developer Studio is also used to implement the vector field path planning algorithm for robotic soccer.

# Keywords

Robotic soccer, vector field path planning, Microsoft Robotic Developer Studio, MSRDS, RDS, robotic simulation environment

## Bibliografická citace:

KUPAROWITZ, T. *Robotický fotbal v RDS.* Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2011. 50 s. Vedoucí bakalářské práce Ing. Petr Honzík, Ph.D.

# Prohlášení

„Prohlašuji, že svoji bakalářskou práci na téma *Robotický fotbal v RDS* jsem vypracoval samostatně pod vedením vedoucího mé 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 části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne: **27. května 2011** ..............................

podpis autora

# Poděkování

Děkuji vedoucímu bakalářské práce Ing. Petru Honzíkovi, Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé bakalářské práce.
Dále děkuji Doc. Ing. Vlastě Sedlákové Ph.D. a Ing. Petru Sedlákovi Ph.D. za odbornou pomoc při řešení některých bodů mé bakalářské práce.
V neposlední řadě děkuji svému otci Ing. Václavu Velískovi za konečnou korekturu mé bakalářské práce, a děkuji též autorům všech v mé práci citovaných zdrojů.

V Brně dne: **27. května 2011** …………………………
podpis autora

# Obsah

1 Úvod.....9
2 Simulátory.....11
  2.1 anyKode Marilou.....11
  2.2 RoboLog.....13
  2.3 Microsoft RDS.....15
  2.4 Ostatní simulátory.....16
3 Vektorová pole.....18
  3.1 Problematika lokálních minim.....21
  3.2 Potenciálové Vektorové pole.....22
4 Implementace v MATLABU.....25
5 Implementace v RDS.....30
  5.1 Služba VectorFieldCalculate.....30
    5.1.1 Vlastnosti projektu VectorFieldCalculate.....33
    5.1.2 Třída VectorFieldCalculateState.....34
    5.1.3 Třída VectorFieldCalculateInput.....35
    5.1.4 CalculateHandler.....36
  5.2 Služba VectorFieldStart.....38
    5.2.1 Vlastnosti projektu VectorFieldStart.....41
    5.2.2 Použití TimeoutPort.....41
    5.2.3 Funkce TimerHandler.....42
    5.2.4 Funkce MainTask.....43
  5.3 Manifest VectorFieldStart.manifest.xml.....44
6 Závěr.....46

# 1 ÚVOD

Již od nepaměti se lidé zabývají hledáním lepších nástrojů k usnadnění prováděné práce. S příchodem druhé poloviny dvacátého století dostal tento zájem nový rozměr – člověk objevil možnost použití elektronického zařízení k autonomnímu provádění určené činnosti. Dnes je naprosto běžné setkat se s roboty pracujícími v továrnách a na místech nebezpečných člověku. A v posledních letech, spolu s poklesem ceny robotických součástí a obecným rozšířením výkonných počítačů, se návrh a výroba robotů stále více rozšiřuje i mezi běžnou populaci. Zvýšená poptávka po virtuálním prostředí, ve kterém je možno vyvíjet a testovat roboty, tak vede ke stále se rozšiřující základně robotických simulátorů, což má zpětně vliv na rozvoj robotického hardwaru.

Robotický fotbal je jedním ze způsobů výzkumu nových technik navádění robotů a jejich autonomního chování zábavnou, kolektivní formou. Na jeho vývoji se podílí stovky programátorů z celého světa, čímž přispívají k rozšíření vědomostí v oblasti autonomní robotiky a řadě jiných odvětví.

Stojíme na prahu doby, kdy například auta začínají jezdit bez řidičů. Pro takový projekt je ale třeba intenzivní práce na vývoji vylepšených algoritmů detekce srážky a práce v simulačním prostředí, což jsou body, kterými se budu ve své práci zabývat.

Cílem mé bakalářské práce tedy bude vypracovat krátký přehled simulátorů, které považuji z uvedených důvodů za předmětné, a ve kterých je možné implementovat algoritmus ovládání diferenciálně řízeného robota pomocí joysticku pro hru robotický fotbal. Navážu na svoji semestrální práci a popíši některé aspekty prostředi RDS (*Robotic Developer Studio*). Celý balíček RDS je možné stáhnout zdarma na stránkách firmy *Microsoft* [1].

V další části své práce představím plánování trajektorie pohybu robota pomocí vektorových polí. Použití algoritmu návrhu trajektorie vektorových polí je jedním z výpočetně nejjednodušších způsobů zjištění optimálního vektoru pohybu pro dosažení požadovaného cíle. Jedná se o metodu známou již několik desítek let, ale stále velmi elegantní, aktuální a užitečnou. Je možné ji využít ve velmi rychle se měnícím prostředí, jakým je například robotický fotbal, aniž by řízený robot přišel do kontaktu s některou překážkou.

Nakonec použiji tuto metodu v již zmiňovaném prostředí RDS. Pro její implementaci zvolím způsob matematicky odlišný od zavedeného standardu. Jedná se o několik usnadnění, spočívajících v zjednodušení kódu, který pracuje se simulačním prostředím. To povede na pomalejší, ale dynamicky stabilnější řešení vektorových polí. Výsledkem implementace algoritmu „hledání trajektorie pomocí vektorových polí" bude kód v jazyce C#, který přijímá všechna potřebná data a vrací optimální vektor pohybu pro naváděného robota.

Představené algoritmy jsou řešeny primárně pro použití v prostředí Microsoft Robotic Developer Studio (RDS) a jsou navrženy modulárně tak, aby bylo možné je upravovat přidáváním dalších rozšíření. Algoritmy jsou napsány jako služby (*service*), což jsou základní asynchronní stavební jednotky RDS. Je možné je použít pro ovládání více robotů současně, popřípadě pro výpočet trajektorie robota ze vzdáleného počítače.

V závěru práce shrnu možnosti jednotlivých představených simulátorů a implementovaný algoritmus. Krátce se zaměřím také na oblasti, které bude možné dále rozvíjet.

Pracuji s *Robotic Developer Studio 2008 R3*, což je k dnešnímu dni poslední vydaná verze, která obsahuje i podporu pro *Microsoft Visual Studio 2010* [2].

# 2 SIMULÁTORY

Důležitou součástí vývoje robotických aplikací je jejich testování, avšak testováním robotů v reálném prostředí může dojít k jejich poškození, a taktéž není vždy možné, aby každý vývojář měl k dispozici svého robota. Řešením těchto problémů je použití simulátorů. Výhody a nevýhody samotného užití simulátorů jsou zpracovány například v mé semestrální práci nebo ve zdrojích [3] a [10].

Předmětem této kapitoly bude seznam simulátorů, které by bylo možné použít pro implementaci hry robotický fotbal. V následujících kapitolách bude implementována metoda návrhu trajektorie pomocí vektorových polí pro jednoosého, diferenciálně řízeného robota. Z důvodů popsaných dále bude pro implementaci zvoleno simulační prostředí Microsoft Robotic Developer Studio.

Simulátory se od sebe liší v mnoha aspektech. Například úrovní a detailem výpočtu podmínek fyzikálního světa, možností následného použití kódu pro skutečné roboty, nebo množstvím existujících toolboxů pro práci se součástmi simulace. Důležitou vlastností simulátorů může být i jednoduchost implementace kódu a pochopení vnitřní struktury vývojového prostředí. Protože nemohu do hloubky posoudit poslední dvě zmiňovaná kritéria pro všechny představené simulátory, budu v této práci čerpat z příruček simulátorů, a představím ty názory přejaté z cizích zdrojů, které považuji za objektivní (převzaté z webových diskusí a od přátel není možné řádně citovat).

## 2.1 anyKode Marilou

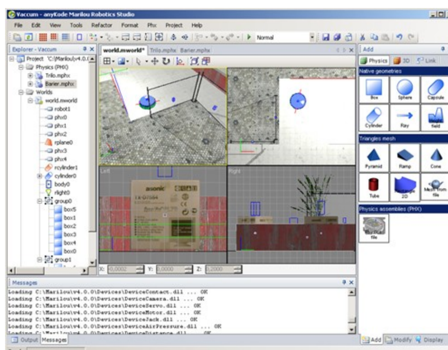
Jedním z nejrozvětvenějších robotických simulátorů je *Marilou*. Jde o prostředí, které je vyvíjeno již více než 15 let. Autorem je firma anyKode se sídlem ve Francii.

*Marilou* dává uživateli možnost implementovat robotické aplikace v široké škále programovacích jazyků jako jsou C/C++, VB#, J#, C#, C++ CLI na platformách Windows a Linux. [4]

*Marilou* využívá takzvané *MODA* SDK k zacházení se simulovanými roboty [5]. Jde o soubor knihoven pro použití na více platformách. V závislosti na programovacím jazyku je možné použít různé *lib* nebo *dll* knihovny s otevřeným zdrojovým kódem. V *MODA* je možné použít různé rozvržení *klient-server* komunikace. Aplikace může běžet na lokálním počítači nebo přes počítačovou síť.

Simulátor, jenž je součástí *Marilou*, má kompletní 3D grafický výstup a používá *PhysX* k akceleraci výpočtů fyzikálního prostředí. Pro simulátor je možné navrhovat prostředí pomocí vkládání a slučování tvarů odvozených ze základních tříd a terénních nerovností. Tyto základní tvary jsou hranol, koule, válec, jehlan, kužel a plocha.

Grafický přehled simulačního prostředí *Marilou*, přejatý z webových stránek projektu, je na Obr 2.1.



*Obr 2.1: Přehled grafického simulačního prostředí Marilou*

Simulátor má podporu křehkých objektů, což znamená, že je možné tyto entity v simulaci rozbít [5]. Děje se tak vhodným nastavením matic vztahů mezi jednotlivými prvky objektu. Tyto matice může uživatel zadat buď sám, nebo si je může nechat vygenerovat pomocí *Marilou* na základě vztahů mezi prvky, ze kterých se objekt skládá. Výhodou užití křehkých modelů je možnost jejich testování za nepříznivých podmínek nebo v kritických situacích. Objekty nastavené jako rozbitelné jsou vystaveny vlivům gravitace, oscilace, nárazům a dalším fyzikálním vlivům prostředí. Ale ne všechny prvky simulace je možné rozbít, entity nastavené jako statické není možné vlivem fyzikálních podmínek simulace změnit.

Podpora kloubů je v *Marilou* také samozřejmostí [5]. Kloubem se v prostředí simulace označuje spoj dvou nebo více jiných objektů. Kloubu je možné nastavit dané stupně volnosti pohybu vzhledem ke spojeným entitám. Možností pohybu kloubů je mnoho. Mezi základní patří fixované klouby, klouby na úrovni jedné či dvou os volnosti, klouby s univerzální možností pohybu ve všech osách, klouby typu kloubní jamka nebo klouby posuvné. Součástí definice kloubu jsou i parametry, které simulují mechanické chování mezi dvěma entitami. Mohou tedy být použity k modelování různých pružin, odporů a podobně. Pro dosažení lepšího přiblížení k realitě je dokonce možné klouby deformovat. K deformaci dochází na základě fyzikálních vlivů a je určena maticí vztahů mezi jednotlivými spojenými entitami.

Součástí *Marilou* je kompletní sada generických senzorů, které je možné použít pro získání dat ze simulace nebo skutečného prostředí [4]. Program je možné navrhnout s těmito generickými senzory nebo užít široké množství skutečných senzorů. V *Marilou* je možné psát knihovny i pro senzory, které podporu zatím nemají [5].

Jak již bylo nastíněno, je v *Marilou* možné používat několik vztahů mezi počítačem a robotem. Výsledný robot může být plně autonomní nebo může být ovládán pomocí

počítače. Při ovládání robota pomocí počítače je možné používat joystick, a je tedy vhodným prostředím pro implementaci robotického fotbalu s použitím joysticku.

Výhodou tohoto prostředí je bezesporu jeho naprostá technologická nadřazenost nad veškerou konkurencí. Grafické renderování simulátoru je na špičkové úrovni a simulace fyzikálního prostředí nikterak nezaostává. Možnost interakce s entitami v simulaci, zrychlení či zpomalení simulace, nahrávání grafických výstupů kamer v simulaci a podobně jsou dokonalé. Výhodou je i implementovaná podpora síťového propojení počítačů a možnost pracovat na dvou nejrozšířenějších platformách s podporou několika programovacích jazyků. Možnost dalšího rozšíření prostředí pomocí balíčků vytváří z *Marilou* silný nástroj k použití kódu třetích stran nebo znovupoužití vlastních knihoven. Poslední zde zmíněnou výhodou je přehledné použití *Marilou* a dostatek dokumentace s návody.

Hlavní nevýhodou pro spoustu vývojářů bude nejspíše to, že se nejedná o volně použitelný produkt. Licenční poplatky jsou nemalé. Dále zde chybí implicitní podpora vícevláknových aplikací a hardwaru, a protože není jasně dána struktura tvořeného kódu, musí si uživatel aplikaci vytvořit takzvaně od podlahy. Stráví tudíž daleko více času programováním vnitřního chodu programu než aplikace samotné.

## 2.2 RoboLog

*RoboLog* je jedním z projektů vyvíjených skupinou umělé inteligence při univerzitě v Koblenz. Jeho úkolem je vývoj tvorby a kontroly autonomních agentů a týmů. Cílem tohoto projektu je podpořit vývoj chytrých programů pro nástroje každodenní potřeby. Projekt se od roku 1999 účastní soutěže *RoboCup* v disciplíně robotický fotbal. V současné době je tento projekt aktivním účastníkem mnoha vědeckých výzkumných soutěží a pomáhá organizovat soutěž *RoboCup*. [6]

Název *RoboLog* je jméno projektu, který implementuje strategii pro hru robotický fotbal. Je to také název simulačního prostředí, vyvíjeného za tímtéž účelem stejnou skupinou lidí. Ti v současné době vyvíjí dva simulátory pro hru robotický fotbal.

2D simulátor pro hru robotický fotbal obsahuje fyziku hracího prostředí, možnost vkládat hráče a agenta typu rozhodčí. Hra je rozdělena na dva týmy, z nichž každý je ovládán z jiného počítače. Strategie hry je implementována pro každý tým zvlášť a je možné toto prostředí použít i pro simulaci hry pomocí joysticku.

Nově vyvíjenou součástí projektu *RoboLog* je již hotové 3D simulační prostředí. To se od původního dvourozměrného liší především lepší kvalitou grafického výstupu, a dále možností zpracovávat situace, kdy míč letí vzduchem.

Simulátor samotný je psán v programovacím jazyce C++ a s uživatelem je spojen na bázi *klient-server*. Klientské aplikace je možné psát v jakémkoliv jazyce, ale v jazyce C++ jsou již týmem z Univerzity v Koblenz napsány důležité knihovny pro interface.

Ovládání robotů v tomto simulovaném prostředí probíhá pomocí přímé změny souřadnic entit libovolným směrem. Výstupem simulace může být pohled na hrací plochu nebo přímo informace o daných entitách.

Problémem projektu *RoboLog*, diskutovaným na fóru, je obtížnost následné implementace kódu ve skutečných zařízeních. Navíc simulátory *RoboLogu* nenabízí možnost použít jiné, než předdefinované roboty.

Jak již bylo zmíněno, obsluhující program může být psán v kterémkoliv jazyce, nicméně je doporučeno používat jazyk C++. Implementace algoritmu vektorových polí za účelem návrhu trajektorie jednotlivých robotů je v tomto simulačním prostředí možná, stejně jako použití joysticku.

Výhodou prostředí *RoboLog* je bezesporu jeho jednoduchost a velikost. Vývoj simulátoru je od začátku zaměřen na hru robotický fotbal, proto je v něm implementován rozhodčí, který usnadňuje ovládání hry. Simulační prostředí není možno rozšiřovat jinak, než v souladu s pravidly hry robotický fotbal. Platformy pro použití jsou přirozeně *Linux* a *Windows*. Klienti a server mohou dokonce běžet na různých platformách.

*Obr 2.2: Grafický výstup 3D prostředí programu RoboLog*

Nevýhodou prostředí je například složité převedení napsaného kódu na skutečný hardware. Nebo téměř nulová možnost vývoje čehokoliv jiného, než hry robotický fotbal. Pro použití joysticku je třeba jeho kód implementovat takzvaně od podlahy. Na Obr 2.2 je ukázka grafického výstupu 3D rozhraní programu *RoboLog*.

## 2.3 Microsoft RDS

*Microsoft Robotics Developer Studio* je dnes objektivně nejlepší robotický simulátor k volnému stažení pro domácí a školní použití. Využívá *CCR* a *DSS* (*.NET CLR*), k běhu aplikací je použito řízení tokem dat. [7], [8]

*CCR* (Concurency and Coordination Runtime) umožňuje aplikacím provádět asynchronní operace, aniž by programátor musel psát složitá vlákna. RDS jej používá jako reakci na zvláštní potřeby robotických aplikací. [8]

*DSS* (Decentralized Sotware Services) je nástroj, který vývojáři umožňuje sledovat stav jednotlivých vláken (služeb) za běhu programu. *DssNode* je aplikace nadřazená všem službám. Shrnuje o nich informace a je nezbytná k jejich inicializaci i ukončení. Je v ní spouštěn soubor manifest, který inicializuje celou aplikaci. [8]

Základním stavebním blokem v RDS je takzvaná služba. Tu je možné považovat za vlákno programu, které obsahuje kód potřebný k provádění jedné nebo více funkcí. Tím se rozumí například čtení dat z jednoho senzoru, odesílání dat na výstup nebo komunikace s jiným vláknem nebo vnějším softwarem [7]. Robotické aplikace se zpravidla skládají z více služeb, které spolupracují při ovládání robota.

Základním blokem simulace je takzvaná entita. Entitou může být jakýkoliv prvek, obsažený v simulovaném prostředí, například obloha, terén nebo robot, ale třeba i kamera, která zprostředkovává grafický výstup simulace. Parametry entit je možné nastavovat (ovládat) pomocí manifestů. Každá entita je dceřinou třídou základní entity nebo entity z ní odvozené. Dědí tedy vždy vlastnosti všech nadřazených entit. [9]

RDS je omezeno na programovací jazyk C# a grafický programovací jazyk VPL (*Visual Programing Language*). Platforma, na které je možné RDS (*DssHost*) provozovat, je pouze operační systém *Windows*.

Simulační engine využívá AGEIA PhysX pro simulaci reálných fyzikálních podmínek. Pomocí přesměrování manifestů a událostí z reálných vstupů a výstupů robota tvoří obraz simulovaného prostředí. VSE (*Visual Simulation Environment*) je nadstavba nad simulačním prostředím a tvoří nástroj, sloužící ke grafickému zobrazení simulace.

Možnost přenosu aplikace ze simulace na skutečné zařízení je velice jednoduchá, zpravidla jde o změnu pár řádků kódu. Také přenos služby z jedné aplikace do druhé je díky systému řízení tokem dat velmi snadné. RDS navíc obsahuje knihovny pro obsluhu téměř všech druhů externích robotických a jiných zařízení, alespoň v generické podobě. Pro mnoho komerčních robotů navíc RDS obsahuje služby psané speciálně k jejich obsluze, což dává uživateli možnost využít všech jejich výhod oproti generickým zařízením. [7]

Výhodou RDS je především jeho naprostá přizpůsobivost a obrovská, stále se rozšiřující, základna knihoven a ovladačů k novým robotickým zařízením. Řízení tokem

dat a systém služeb dává uživateli možnost naprogramovat v zásadě jakoukoliv robotickou aplikaci. Například vytvoření aplikace s použitím joysticku je pro zkušeného uživatele ve VPL otázkou několika minut. [11] Navíc je možné vytvářet nové služby (servisy) v prostředí *Microsoft Visual Studio*. Je tedy například možné vytvořit službu pro obsluhu joysticku, která vyhovuje všem kladeným potřebám. [11] Také pro implementaci algoritmu vektorových polí poskytuje RDS přivětivé mechanismy, které je možné využít. Samotná implementace je součástí této práce. Nemalou výhodou je i to, že *Microsoft* v roce 2007 uvolnil RDS k bezplatnému domácímu a školnímu použití, aniž by přestal s jeho dalším vývojem.

Nevýhody jsou v zásadě jen dvě. Za prvé je to příliš velká složitost vnitřních dějů. Zejména jde o složité cyklické závislostí základních implementačních pravidel, do kterých začínající programátor jen obtížně vstupuje. Za druhé jde o fakt, že RDS není multiplatformní. Je tedy možné ho používat jen na operačních systémech firmy *Microsoft*. Proto může být mnohdy nepraktické jeho využití pro seriózní výzkum a vědeckou práci.

## 2.4 Ostatní simulátory

V předchozích kapitolách byly popsány tři robotické simulátory, které je možné nejlépe použít pro implementaci algoritmu vektorových polí a ovládání hry robotický fotbal pomocí joysticku. Z mnoha ostatních robotických simulátorů zde budou ve zkratce popsány pouze některé, v nichž je možné splnit tato zadání.

*Simrobot* je dvojrozměrné vývojové prostředí pro simulaci. Je vyvíjeno Ústavem Automatizace a Měřící Techniky při VUT v Brně (nezaměňovat se stejnojmeným projektem vyvíjeným na University of Bremen). Implementace tohoto simulátoru je psána v prostředí MATLAB, ale obsahuje kódy psané v jazyce C na obsluhu senzorů. Přestože se jedná o relativně jednoduchý simulátor, jeho síla je právě ve využití programu MATLAB, který pomocí standardního toolboxu implementuje použití joysticku a výpočet matic vektorových polí je pro něj nativní. *Simrobot* je možné volně distribuovat v nezměněné podobě. [12]

*PyRobbie* je simulační prostředí postavené na jazyce Python. Původním účelem tohoto simulátoru byla možnost učit se jazyk Python zábavnou formou, ale později byl rozšířen o další možnosti. V současné době sestává z velkého množství rozšiřujících toolboxů. Jelikož je Python konzolový interpretovaný jazyk, musí se uživatel spokojit s nehezky vypadajícím grafickým výstupem. Seriózní implementace robotického fotbalu ještě v tomto prostředí nebyla provedena. [13]

*Simbad* je 3D simulační prostředí psané v jazyce Java. Jde o otevřenou platformu pro simulaci několika základních senzorů a základních fyzikálních zákonů. Simulátor sám netvrdí, že je jakkoliv přesný v simulaci fyzikálního prostředí. Je zaměřen spíše na

testování správných postupů psaní agentů, a je koncipován co nejjednodušeji. Je v něm možné pomocí standardních toolboxů vytvořit robota ovládaného joystickem. K simulaci robotického fotbalu se ale nehodí díky benevolentnímu přístupu k simulaci fyzikálních podmínek. [14]

Přehled hlavních komerčních a otevřených simulátorů je možné nalézt na stránkách *WikiPedie*. [15] Množství simulátorů zde uvedené je jen zlomkem celkového množství všech v současné době vyvíjených simulátorů.

# 3 VEKTOROVÁ POLE

Prostředí pro pohyb robota si lze představit jako soustavu elektrických nábojů ve dvou či vícerozměrném prostředí. Z fyziky víme, že shodné náboje se odpuzují a opačné přitahují, a elektrické pole je orientováno směrem od kladných nábojů k záporným. Tohoto faktu využívá metoda vektorových polí. Dle zvolené konvence budou překážky kladně nabité a cíl, ke kterému budu směřovat, bude mít náboj záporný. V prostředí, kde překážky a cíl mají takto zvolenou polaritu nábojů, stačí z jakéhokoliv místa sledovat siločáry, které vedou do požadovaného cíle. Tyto náboje se liší svojí intenzitou a tudíž i intenzitou pole, které kolem sebe vytváří. Ve skutečném světě tato intenzita klesá se čtvercem vzdálenosti od pozice náboje (je předpokládán bodový náboj).

Vycházíme z elementárních vztahů elektrostatiky [16],[17]. Na základě Coulombova zákona:

$$\vec{F}=\frac{1}{4\pi\epsilon_0}\frac{q_1 q_2}{r^3}\vec{r} \tag{1}$$

a vztahu mezi intenzitou elektrického pole a elektrickou silou

$$\vec{E}=\frac{\vec{F}}{q} \tag{2}$$

*kde* $\vec{E}$ *je intenzita elektrického pole částice,*
$q$ *je náboj částice,*
$\vec{F}$ *je elektrická síla působící na částici.*

lze definovat vztah pro elektrické pole bodového náboje

$$\vec{E}=\frac{1}{4\pi\epsilon_0}\frac{q}{r^3}\vec{r} \tag{3}$$

*kde* $\epsilon_0$ *je elektrická konstanta*
$r$ *je vzdálenost od náboje*

Tato rovnice definuje elektrické pole kolem osamoceného náboje. Pokud v jeho okolí existuje jeden nebo více dalších bodových nábojů, využíváme principu superpozice pro stanovení intenzity elektrického pole pro určité místo. Výsledná intenzita elektrického pole je potom logicky jen vektorový součet intenzit jednotlivých nábojů.

$$\vec{E}=\sum\vec{E}_i \tag{4}$$

Coulombův zákon je pouze speciální případ obecnějšího Gaussova zákona. Pomocí Gaussova zákona je možno elektrické pole vypočítat jako souvislé rozložení náboje v prostoru.

$$\nabla \vec{E} = \frac{\rho}{\epsilon_0} \qquad (5)$$

*kde* $\rho$ *je objemová hustota náboje*

Gaussův zákon pro elektrické pole je jednou ze čtyř Maxwellových rovnic, které popisují makroskopickou teorii elektromagnetického pole.

Pro zjednodušení výpočtu je možno upravit rovnici (3) tak, že položíme konstantní část rovnice rovnu 1.

$$4\pi\epsilon_0 = 1 \qquad (6)$$

Vztah pro elektrickou intenzitu se zjednoduší nasledujícím způsobem:

$$\vec{E} = \frac{q}{r^3}\vec{r} \qquad (7)$$

Nyní je možné vypočítat pole elektrického náboje, jehož intenzita se zmenšuje se čtvercem vzdálenosti a začíná velikostí náboje $q$ . Kdyby tedy bylo třeba vypočítat, jakou silou na daný bod působí určitý náboj, stačí za $r$ dosadit vzdálenost bodu od náboje a za $q$ hodnotu náboje.

V souladu s předestřenou konvencí je tedy cíl označen záporným nábojem a překážky kladným nábojem. Ukazuje se však, že se stoupající vzdáleností od cíle se zvyšuje i negativní dopad překážek na požadovanou trajektorii. Tuto nevýhodu je nutné obejít použitím homogenního pole.

Homogenní elektrické pole je pole, které má ve všech svých bodech stejný směr a stejnou velikost intenzity elektrického pole. Typickým příkladem homogenního pole je elektrické pole mezi dvěma nekonečně velkými nabitými deskami. V případě použití homogenního pole pro robotický fotbal se musí jít ještě o něco dále. Nestačí totiž uvažovat homogenní pole mezi dvěma plochami, ale pole musí mít rovnoměrnou intenzitu paprskovitého tvaru, což je fyzikálně nemožné. Nyní nezbývá, než se oprostit od fyziky a modifikovat některé klíčové vztahy. Pro přiblížení se k tomuto požadavku ve skutečném světě je možné uvažovat dostatečně dlouhé nabité vlákno s rovnoměrně rozloženým nábojem, kdy je dán předpoklad, že intenzita elektrického pole neklesá se vzdáleností od osy vlákna. Dostáváme paprskovité elektrické pole, které má ve všech bodech stejnou velikost elektrické intenzity

$$E = \frac{q}{\epsilon_0} \approx q \; . \tag{8}$$

Ze vztahu (8) vyplývá, že intenzita elektrického pole je v tomto uvažovaném případě konstantní. Potom pro elektrický potenciál takového elektrického pole platí vztah:

$$\phi = \int E\, dr = q\, r \tag{9}$$

Dopad cílového náboje bude pak neměnný v poměru k ostatním nábojům nezávisle na vzdálenosti od něj. Takže vliv působení náboje, indikujícího pozici cíle, bude konstantní bez ohledu na vzdálenost $r$. V některých aplikacích je dokonce možné naopak vliv cílového náboje s rostoucí vzdáleností zvyšovat, čímž lze dosáhnout daleko „agresivnějšího" postupu, pokud se robot nachází dále od požadovaného bodu. [18]

Pokud je úkolem pouze navést robota na cíl, stačí vhodně umístit záporný pól. Pokud je ale třeba navést robota na cíl z určitého směru, je třeba zvolit složitější postup. Jedním ze způsobů dosažení toho, aby se robot přiblížil k cíli z daného směru, je vložit na polopřímku, začínající v cílovém bodě a mající směr požadovaného pohybu, bod s opačným potenciálem, než je bod cílový [19]. Pokud by se pole daných nábojů zmenšovalo se čtvercem vzdálenosti, jak je tomu ve skutečné fyzice, vznikne z těchto dvou nábojů dipól. Tento dipól má záporný náboj na pozici cíle a kladný náboj umístěný někde ve směru požadovaného pohybu. Jelikož je zde použit záporný pól, jehož intenzita se nezmenšuje se vzdáleností od náboje, je třeba kladnou část volit stejně. V takovém případě by však za umístěním těchto pólů vznikaly plochy s nulovou elektrickou intenzitou dipólu. Z tohoto důvodu je volen kladný pól dipólu menší, než je jeho záporný pól. Rozdíl těchto dvou intenzit je potom skutečná intenzita dopadu dipólu na robota a při výpočtech je nutno pracovat právě s touto hodnotou. Aby se u takto definovaného dipólu docílilo toho, že se robot skutečně přiblíží k cílovému bodu z daného směru, musí být na spojnici pólů, do těsné blízkosti cílového bodu, vložen ještě regulérní kladný pól v případě, kdy se robot nachází v pásmu mezi oběma póly. Tento kladný bod se chová jako překážka. Naváděný robot se bude snažit dostat kolem vloženého kladného bodu mimo středové pásmo dipólu. Jakmile je potom robot mimo toto středové pásmo, nemůže již na cílový bod najet ze špatného směru. Koncept dipólů a vkládané překážky je vidět na Obr 5.4 a Obr 5.5.

V odborných publikacích na stejné téma se vyskytuje způsob výpočtu vektoru pohybu pomocí sčítání vektorů působících překážek a cíle. Tato metoda je v současné době důkladně prostudována a byla již mnohokrát optimalizována [19], [20], [24], proto je v této bakalářské práci zvolen jiný přístup. Výpočet vektoru pohybu se děje pomocí výpočtu potenciálů v blízkosti naváděného robota. Jedná se o výpočet výsledného

vektoru pomocí záporného gradientu potenciálů, generovaných překážkami a cílem. [19], [20]

Gradient je v obecném smyslu slova směr růstu. V matematice je to diferenciální operátor, jehož výsledkem je vektorové pole, vyjadřující směr a velikost největší změny skalárního pole. Označuje se pomocí operátoru nabla $\nabla$ . Jedná se o invariantní veličinu, tudíž nezávisí na volbě souřadné soustavy. [21], [22]

K zjištění vlastního směru pohybu tedy stačí vypočítat záporný gradient součtu potenciálů všech překážek, působících na naváděného robota a cíle pohybu, čímž vznikne směr největšího spádu (od kladného k zápornému pólu).

## 3.1 Problematika lokálních minim

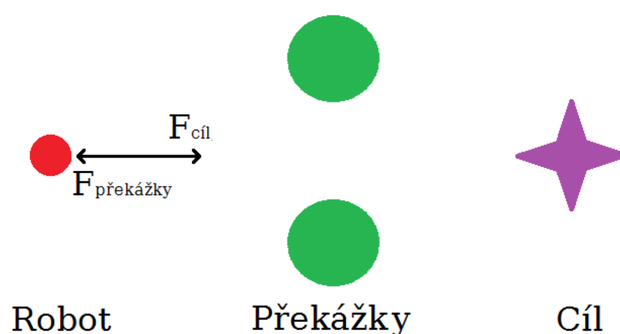
Jednou z nejdůležitějších nevýhod vektorových polí je to, že tato metoda nemůže být bez ztráty elegance a rychlosti použita pro plánování trasy v globálním měřítku. Blízce se tohoto postihu týká takzvaná problematika lokálních minim. Lokální minima jsou místa, kde vlivem překážek může robot uvíznout, a ze kterých již nevede žádná cesta směrem k nižší hodnotě potenciálu. [23]

Způsobů řešení problematiky lokálních minim existuje mnoho. V zadaném případě, kdy se robot nachází ve vysoce dynamickém prostředí s minimem překážek, které se pohybují o vlastní vůli, se jako optimální způsob řešení jeví jednoduše vyčkat na uvolnění trasy. Dojde-li k zaseknutí robota v lokálním minimu, zůstane robot v pohotové pozici, a vyčká na uvolnění cesty. Sám od sebe se při uvíznutí nesnaží z lokálního minima vyprostit. Na globálním plánovači pak závisí návrh trasy bez statických lokálních minim a průběžná kontrola, zda se robot v lokálním minimu nenachází.

*Obr 3.1: Lokální minimum s jednou překážkou*

Nejjednodušším druhem lokálního minima je případ, kdy se překážka nachází přímo mezi robotem a cílem. Lokální minimum tedy vznikne, pokud jsou tyto tři prvky na jedné spojnici a neexistuje žádná jiná překážka v dosahu, která by narušila tento rovnovážný stav. Odpudivá síla překážky se v tomto případě odečte s přitažlivou silou cíle a výsledná síla je nulová, což vede k zastavení robota. Tento druh lokálního minima je patrný z Obr 3.1.

Na Obr 3.2 je častěji se vyskytující druh lokálního minima. Totiž případ, kdy se odpudivé síly více překážek sčítají a vyváží tak přitažlivou sílu cíle. Robot se tedy zastaví v sedle, tvořeném kladným elektrickým polem dvou překážek. Z tohoto důvodu je třeba vhodně volit intenzitu překážek v poměru s intenzitou cíle tak, aby pro vznik tohoto druhu lokálního minima musela být mezera mezi dvěma překážkami menší než šířka robota. Tím se efektivně zmenší šířka a „prudkost" sedla, a tudíž je menší pravděpodobnost, že v něm robot uvízne.

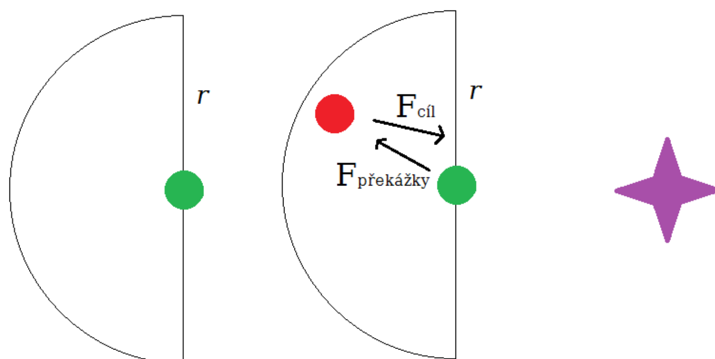



*Obr 3.2: Lokální minimum s více překážkami*

## 3.2 Potenciálové Vektorové pole

Teorie vektorových polí byla již mnohokrát implementována a optimalizována. V této práci je tedy navržen způsob odlišný od standardního přístupu k jejich výpočtu. Tato kapitola se bude věnovat jeho vysvětlení a budou v ní uvedena použitá vylepšení metody vektorových polí následně užité k jeho implementaci. Bude zde také navrženo zlepšení, které by mohlo být předmětem budoucích studií.

Nejdůležitějším krokem k urychlení výpočtu potenciálů z překážek je heuristika zjištění možnosti kontaktu s překážkou. Překážky, které neleží v cestě mezi robotem a cílem, proto nejsou při výpočtu brány v potaz. Dojde tak k efektivnímu vynechání výpočtu potenciálů překážek, které jsou vzdálenější od cíle než robot, nebo které leží v jiném kvadrantu, a tudíž nemohou ovlivnit trajektorii robota. Objekt překážky také obsahuje proměnnou *radius*, která udává, jak daleko se od překážky bere její vliv v potaz. Takže překážky, jež jsou od robota vzdáleny více než *radius* nebo neleží v cestě překážky, nejsou uvažovány ve výpočtu, jak je patrné z Obr 3.3.




*Obr 3.3: Poloměr vlivu překážek a jejich poloha*

Výpočet vektoru pohybu pomocí této metody probíhá výpočtem součtu potenciálů překážek a cíle pomocí rovnic (7) a (9). Suma potenciálů se bere v devíti bodech v těsné blízkosti robota samotného pro každý tento bod zvlášť. Výpočtem záporného gradientu o dvou stupních volnosti tohoto pole potenciálů pak vede na výsledný vektor pohybu. K drobnému urychlení by vedlo použití výpočtu výsledného vektoru pomocí gradientu pouze v pěti bodech v okolí robota.

| $X_1$<br>$Y_1$ | $X_1$<br>$Y_2$ | $X_1$<br>$Y_3$ |
|---|---|---|
| $X_2$<br>$Y_1$ | $X_2$<br>$Y_2$ | $X_2$<br>$Y_3$ |
| $X_3$<br>$Y_1$ | $X_3$<br>$Y_2$ | $X_3$<br>$Y_3$ |

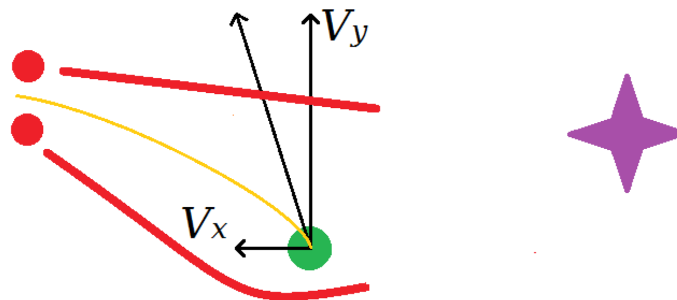
*Obr 3.4: Výpočet gradientu*

Na Obr 3.4 je znázorněna matice užitá pro výpočet gradientu v devíti bodech v okolí robota. Bod $X_2Y_2$ představuje pozici robota a zbývající pozice představují body v jeho těsné blízkosti. V každém bodě je možné zaznamenat různý dopad součtu potenciálů překážek a cíle. Výsledný vektor největšího poklesu potenciálu (směr kolem překážek k cíli) je možné vypočítat pro jeho dvě složky pomocí soustavy rovnic (10). [21]

$$\nabla f(x,y)=\left(\frac{df}{dx},\frac{df}{dy}\right)$$
$$\frac{df}{dx}=X_1Y_1+2X_1Y_2+X_1Y_3-X_3Y_1-2X_3Y_2-X_3Y_3 \tag{10}$$
$$\frac{df}{dy}=-X_1Y_3-2X_2Y_3-X_3Y_3+X_1Y_1+2X_2Y_1+X_3Y_1$$

Tato metoda zjištění vektoru pohybu je přirozeně výpočetně náročnější, než nejjednodušší originální implementace metod sčítání vektorů. Může ale ze své podstaty, totiž použitím gradientu k výpočtu, vést na stabilnější, méně kmitavou trajektorii při průjezdu například úzkou chodbou.

Posledním z navrhovaných vylepšení, které by si vyžádalo hlubší studii, je kalkulace s rychlostmi překážek. Vychází z předpokladu, že při známé rychlosti a vektoru pohybu překážky, a při známé požadované rychlosti robota, je možné volit ze dvou způsobů, jak se překážce vyhnout (viz Obr 3.5). Žlutá křivka dělí prostor v okolí překážky na plochu, ze které ještě lze projet před překážkou, a na plochu, ve které je již nutné počkat, než překážka projede, a zvolit tedy průjezd za ní. Tak jako na ostatní vylepšení, i na toto je myšleno v samotné implementaci algoritmu.




*Obr 3.5: Vyhýbání se překážce o známé rychlosti*

# 4 IMPLEMENTACE V MATLABU

Pro testování navrhované potenciálové metody je použit program MATLAB. Jsou využity jeho funkce pro tvorbu polí vektorů a pro práci s maticemi.

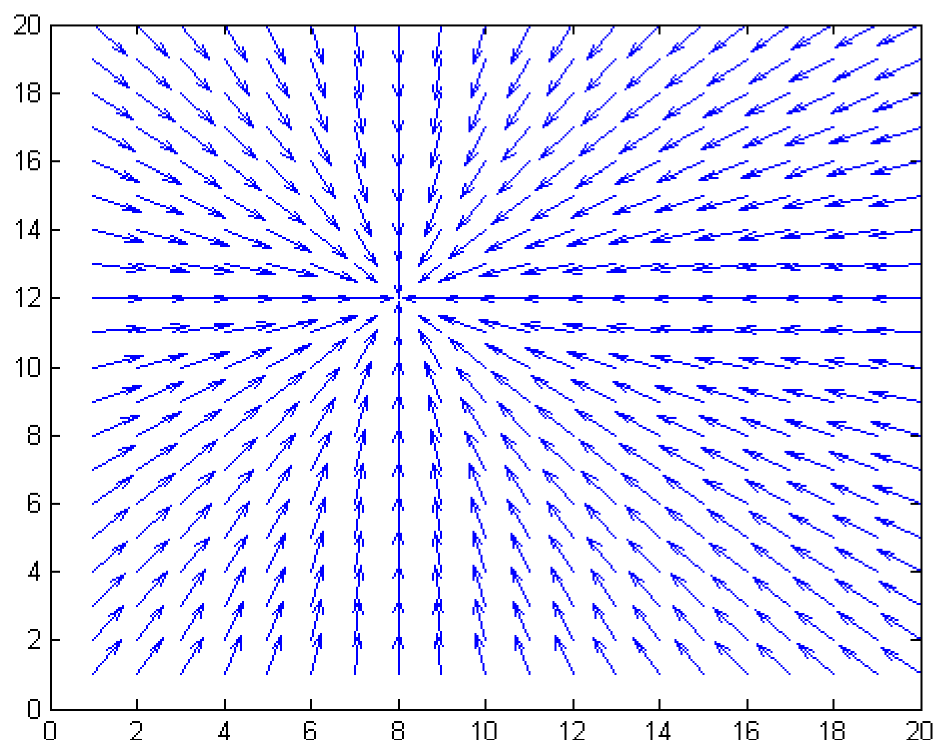
Nejprve je třeba naplnit dvě matice vektorů, s nimiž bude program pracovat, hodnotami indexu jejich sloupců a řádků. Byl použit příkaz *meshgrid*.

```
[X,Y] = meshgrid(1:1:sizeX, 1:1:sizeY);
```

Druhá pomocná proměnná je pole s názvem *radians*, které obsahuje vypočtené vzdálenosti od středu sebe sama. Takže v každém poli této matice bude číselná hodnota v plovoucí řádové čárce s hodnotou vzdálenosti od středu matice.

```
Radians = ((radiansX-radiansize+1).^2+(radiansY-
radiansize+1).^2).^.5;
```

Poté je třeba za pomoci matice *radians* naplnit pole potenciálů, a to tak, že bude matice *radians* využita při výpočtu vzdálenosti každého bodu prostředí od pozice překážky (na pozici překážky bude umístěn střed matice *radians*). Děje se tak podle vztahu (7).




*Obr 4.1: Vektorové pole v programu MATLAB bez překážek*

Jakmile je do matice potenciálů zaveden dopad všech překážek a cíle na všechny body v prostředí, je možné vypočíst kýžený vektor pohybu pro všechny pozice. Předpokládejme, že hodnoty potenciálů jsou uloženy v matici s názvem *potential* o rozměrech shodných s maticemi *X a Y*. Pak je možné pomocí funkce *gradient* vypočíst záporný vektor pohybu v obou osách souřadnicové soustavy pro každý bod matice *potential*.

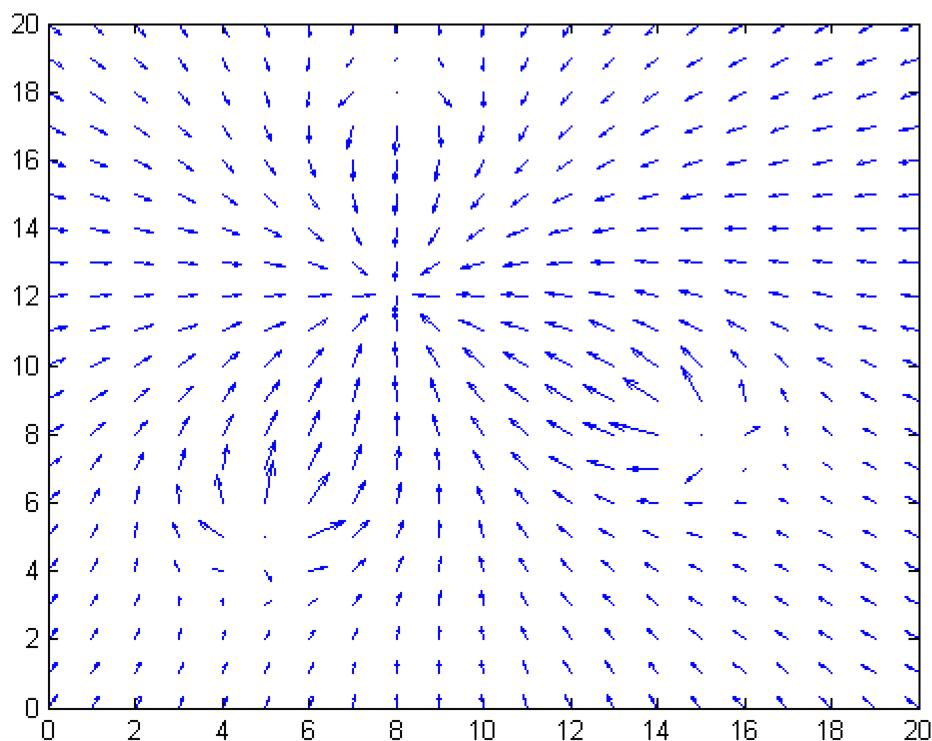
```
[Ex,Ey] = gradient(-potential, 1, 1);
```

Nakonec je výsledek zobrazen formou vektorového pole pomocí funkce *quiver*.

```
quiver(X,Y,Ex,Ey);
```

Obr 4.1 ilustruje grafický výstup při užití cíle v prostoru o rozměrech 20 x 20. Jak je patrné, vektory na všech pozicích směřují k cíli.

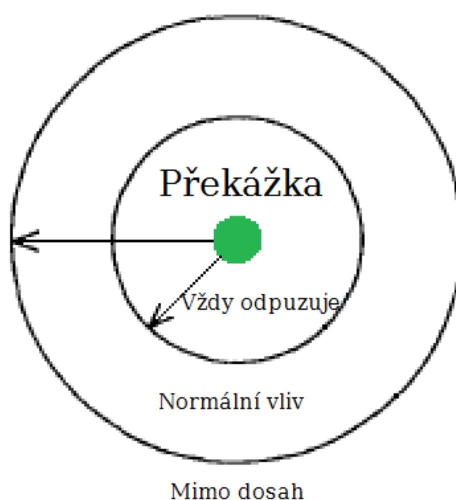
Grafický výstup skriptu v případě, kdy je do potenciálového pole přidán vliv několika překážek, je vidět na Obr 4.2.




*Obr 4.2: Vektorové pole v programu MATLAB s překážkami*

Na Obr 4.2 je jasně zřetelná tendence vyhýbat se překážkám. Aby bylo možné nastavit sílu této tendence, používá toto řešení dvou proměnných. U některých jiných projektů, které stojí před obdobným problémem, se vyskytuje následující řešení [19].

Prostor kolem překážek je rozdělen na tři pásma, viz Obr 4.3. Pásmo nejblíž k překážce je definované nejmenším poloměrem a značí prostor, kam se robot nikdy nesmí dostat. V okamžiku, kdy se robot v tomto pásmu ocitne, musí zvolit vždy jen trajektorii směrem od překážky, aby z pásma co nejdříve vyjel. Pásmo, které je vyčleněno poloměrem největším, značí prostor, ve kterém již překážka nemá na okolí žádný vliv. V prostředním pásmu je vliv překážky na robota zaznamenán a počítá se s ním standardním způsobem.



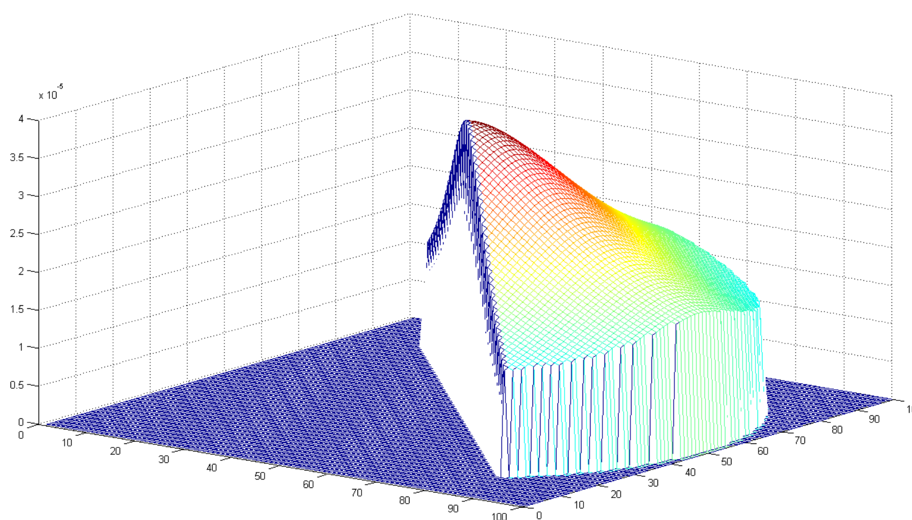

*Obr 4.3: Tři pásma překážky*

Naproti tomu zde předestřené řešení pracuje u překážek pouze s jedním poloměrem. Ten rozděluje prostor kolem překážek na místa, kde překážka je pro výpočet potenciálů brána v potaz, a na místa, kde již není patrná. Samotný kritický prostor těsně kolem překážky je pak vyčleněn pouze intenzitou překážky. Markant tohoto řešení spočívá v tom, že takto zavedený systém překážek může být použit i k simulaci zdí nebo terénních nerovností.

Pro realizaci úprav výpočtu potenciálů vektorového pole byl pro implementaci v MATLABu sepsán skript *pieslice.m*. Tento skript vrací kruhovou výseč s hodnotou logická jedna v místech náležících výseči, a logická nula v místech do výseče nenáležících. Překážky jsou pak při výpočtu násobeny takto získanou maticí hodnot a jejich vliv je tudíž patrný pouze ve směru za překážkou (bráno z pohledu od cílového bodu). Vstupem tohoto skriptu je poloměr kruhu výseče, počáteční úhel a koncový úhel výseče. Výstupem skriptu je čtvercová matice se stranou o délce dvojnásobku poloměru kruhové výseče. K výpočtu kruhové výseče je použit upravený Bresenhamův algoritmus pro vykreslení kružnice, který kreslí pouze výseč kružnice, a Bresenhamův algoritmus pro vykreslení přímky, jimiž jsou spojnice mezi středem a okrajem kruhové výseče [25]. Po aplikaci těchto dvou algoritmů je návratová matice skriptu *pieslice.m* vyplněna

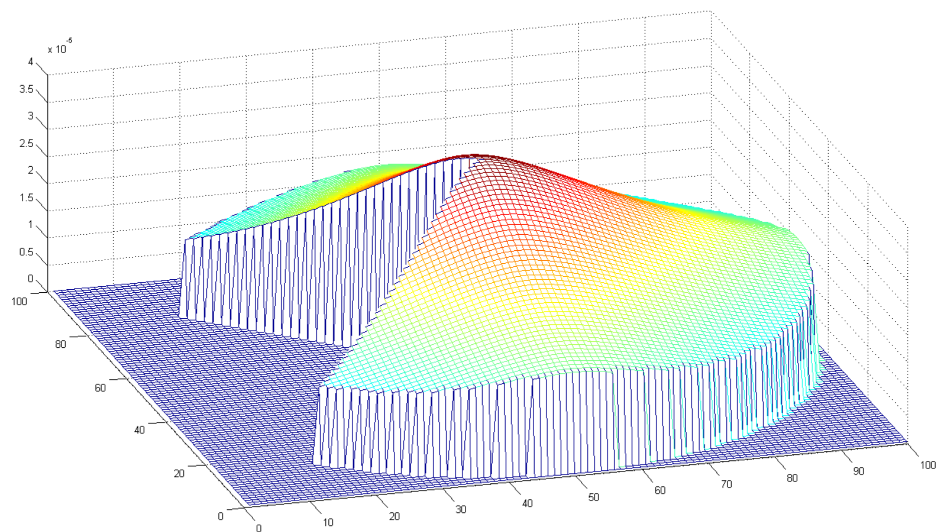
ohraničenou plochou kruhové výseče s hodnotou logická jedna v krajních bodech. Následně je obrazec vyplněn nízkoúrovňovou funkcí MATLABu *imfill*. Tento skript je využit k vykreslení kruhové výseče s úhlem 180°, která je použita tak, aby byla ve výstupním vektorovém poli jasně zřetelná oblast dosahu vlivu překážky. Překážka má tak dosah zvoleného poloměru a je započítána pouze pokud se nachází v cestě naváděného robota.

Jelikož je přechod potenciálů mezi pozicí překážky a místem s nulovým vlivem překážky pro vektorové zobrazení pomocí šipek v MATLABu příliš silný, a mimo tohoto přechodu by na vektorovém poli nebylo nic jiného patrné, je pro představu na Obr 4.4 zobrazen trojrozměrný vliv překážky reagující na cílový bod v počátku souřadné soustavy. Překážka sama je umístěna na pozici (50, 50) a její poloměr dosahu je nastaven na hodnotu 48 polí. Takto vypočtená matice potenciálů by byla v příslušném bodě přičtena k již existující matici potenciálů tak, aby střed této matice, reprezentující střed překážky, byl ve výsledné matici potenciálů na pozici překážky.




*Obr 4.4: Potenciál překážky s cílem v počátku souřadné soustavy*

Pro lepší představu použití skriptu *pieslice.m* je na Obr 4.5 ilustrována výseč s úhlem 270°. Takto vysoký úhel by bylo optimální využít pro výpočty s cílem, k němuž je třeba se přiblížit z určitého směru, ale realizace v jazyce C# by byla příliš náročná na výsledný strojový čas.




*Obr 4.5: Výseč potenciálu překážky s úhlem 270°*

# 5 IMPLEMENTACE V RDS

Návrh trajektorie diferenciálně řízeného robota pomocí vektorových polí byl realizován formou služby pro Microsoft Robotics Developer Studio. V této formě je možné program dále rozvíjet modulárně s jasně nastoleným protokolem komunikace s partnerskými službami. Ke komunikaci je použit SOAP protokol, vlastní aplikacím v RDS. Program je tak možné zkompilovat a dále distribuovat formou dynamické knihovny *dll*. Službu samotnou je poté možné změnou nastavení v programu *dsshost* spouštět buď z lokální paměti samotného robota, z externího počítače ovládajícího robota, nebo z jakéhokoliv počítače připojeného pomocí internetové sítě k ovládacímu počítači, případně k robotu samotnému. Aby program pracoval jak má, je nutné v RDS implementovat minimálně dvě služby. První služba bude realizovat samotný výpočet optimálního vektoru pohybu (*VectorFieldCalculate*) a druhá služba obstará spuštění simulačního prostředí, dále spuštění první služby, a bude implementovat strategii pohybu (*VectorFieldStart*). První službu je také možné spouštět pomocí manifestu načteného aplikací *dsshost*.

Způsob a použití manifestu v tomto projektu bude probráno v samostatné podkapitole.

Pro lepší pochopení struktury RDS a některých dějů popsaných v této kapitole je možné nahlédnout do zdrojů [7], [26], [27], [28].

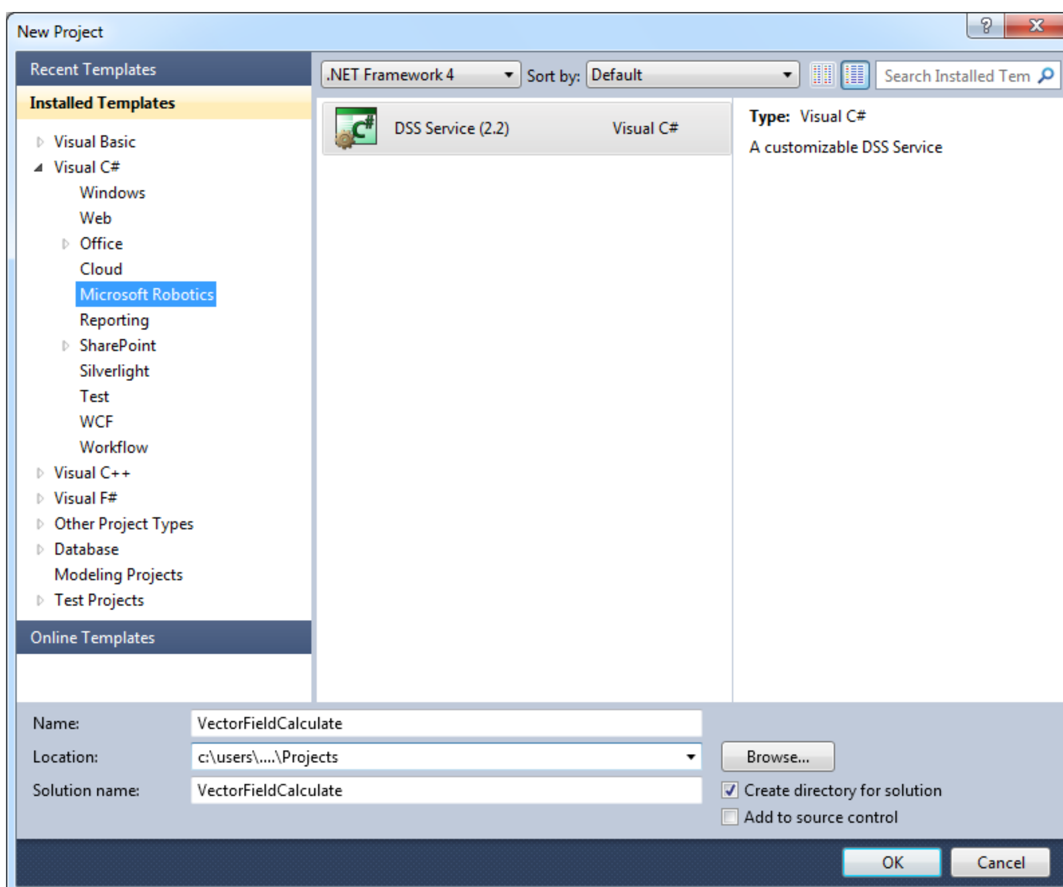
## 5.1 Služba VectorFieldCalculate

Důležité je správně pochopit roli služby *VectorFieldCalculate*. Ta po svém spuštění běží bez ohledu na činnost ostatních spuštěných služeb. Nemá přehled o ostatních službách a ani za ně nezodpovídá (mimo služby *arbiter*). U ostatních běžících služeb, které jej využívají, je tedy důležité správně definovat partnerské vztahy, aby se *arbiter* aplikace *dsshost* mohl správně postarat o doručování zpráv. Služba *VectorFieldCalculate* bude vždy formou *get* zprávy vyzvána k výpočtu optimálního vektoru pohybu pro stanovené rozložení překážek a cíle. V odpověď na tuto výzvu (*get*) bude služba *VectorFieldCalculate* vracet proměnnou typu *state* s hodnotou nového vektoru.

Služba *VectorFieldCalculate* také přirozeně zavádí do pracovního prostoru aplikace, která ji využívá, nové datové typy. Všechno jsou to třídy odvozené nebo využívající generické typy RDS a C#, díky čemuž se již nemusíme starat o správné parsování proměnných pro SOAP zprávy, které jsou zasílány většinou ve formě *xml* dokumentů. SOAP zprávy mohou být na základě vyhodnocení aplikace *dsshost* zasílany také formou bitové reprezentace obsažených proměnných. Na to uživatel nemá přímý vliv, ale může

správnou volbou zasílaných typů program *dsshost* k tomuto chování přimět (například i příslibem konstantnosti zasílaných proměnných).

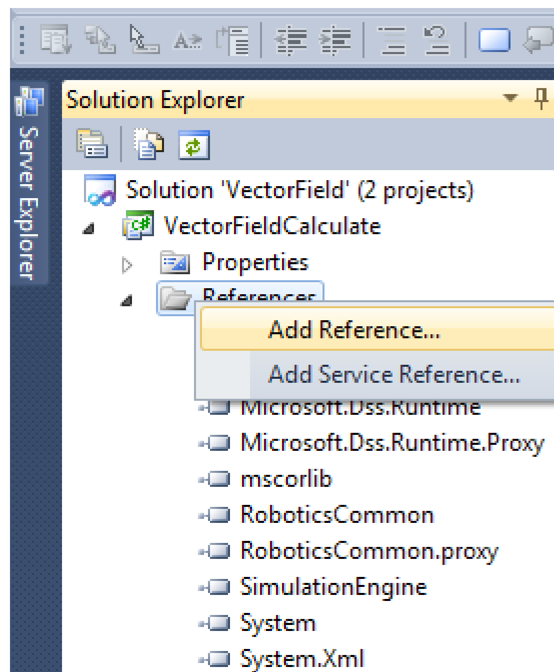
V prvním kroku implementace služby *VectorFieldCalculate* je nutné se ujistit, že na počítači je nainstalována nejnovější verze RDS (je použito Microsoft Robotics Developer Studio 2008 R3). Dále je nutné vytvořit nový projekt v aplikaci Microsoft Visual Studio (osobně jsem použil Microsoft Visual Studio 2010 Ultimate). V nabídce tvorby nového projektu je v odrážce C# zvolena možnost Microsoft Robotics a jako název projektu je zvolen *VectorFieldCalculate* (viz Obr 5.1).



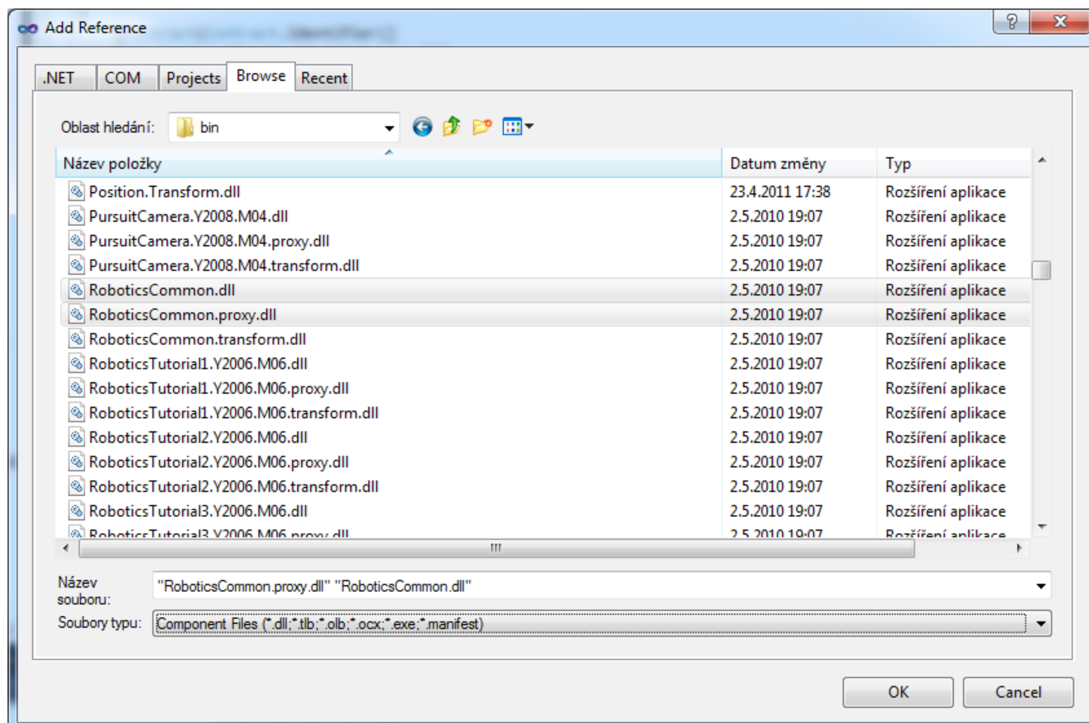

*Obr 5.1: Nová služba VectorFieldCalculate*

Po potvrzení těchto voleb je možné v následující nabídce utvořit počáteční nastavení této služby. Je nutno zde ponechat výchozí hodnoty. Všechny dodatečné vazby budou doplněny v nastavení projektu samotného.

V záložce *Preferences* ve stromu projektu je nutné přidat knihovny *RoboticsCommon* a *RoboticsCommon.proxy*. Přidání referencí je postupně vidět na obrázku Obr 5.2, a následně ze složky instalace *Microsoft Robotics Dev Studio 2008 R3/bin* na Obr 5.3.




*Obr 5.2: Přidání referencí*




*Obr 5.3: Přidání potřebných referencí*

Při tvorbě nového projektu pro RDS v Microsoft Visual Studiu jsou automaticky generovány dva soubory. Jedním z nich je *VectorFieldCalculate.cs,* druhým *VectorFieldCalculateTypes.cs*. Druhý zmiňovaný soubor slouží k definici tříd pro operace s daty pro novou službu. Jsou zde definovány důležité třídy pro chod služby jako například *contract*, *operations*, *subscribe* a *state*.

Služba *VectorFieldCalculate* přijímá zprávy volajících služeb s identifikátorem *Calculate* a daty typu *VectorFieldCalculateInput.* Přijatá zpráva má v souboru *VectorFieldCalculateTypes.cs* nastaven formát *get*. To znamená, že odesílatel bude čekat na přijetí odpovědi na svůj dotaz. Odpověď bude typu *VectorFieldCalculateState*. Oba zmíněné typy jsou ve stejném souboru také definovány. Jsou nastaveny jako datové třídy a používají generické datové typy nativní pro RDS, tudíž je jejich parsování pro zaslané zprávy velmi jednoduché.

Algoritmus výpočtu vektoru pohybu je v souboru *VectorFieldCalculate.cs*. V tomto souboru se nachází především definice statických dat služeb, otevření portů pro komunikaci a navázání komunikace se *Subscription Managerem* příslušného *dsshost*, metoda pro spuštění služby *VectorFieldCalculate* a *handlery* pro jednotlivé důležité funkce této služby. Nejdůležitější *handler* služby má název *CalculateHandler* a je volán při žádosti o výpočet nového vektoru pohybu.

### 5.1.1 Vlastnosti projektu VectorFieldCalculate

Výsledek kompilace služby v prostředí RDS je dynamická knihovna typu *dll*. Ta má všechny náležitosti potřebné k tomu, aby mohla být použita jako služba. Parametry tohoto výstupního souboru se nastavují ve volbě *Properties* ve stromu projektu zobrazeném na Obr 5.2. Součástí je i zdrojový soubor se standardním názvem *AssemblyInfo.cs*.

*AssemblyInfo.cs* obsahuje deklarace platné pro celý projekt a generovanou knihovnu. Soubor je zde uveden pro úplnost, není nutné v něm cokoliv měnit. Obsahuje atributy pro překladač, které mu oznamují, že výsledek překladu bude služba pro prostředí RDS. Dále je zde možné obsáhnout informace o názvech, autorech, licenci, popisu, či verzi knihovny. Syntaxe tohoto kódu je velmi jednoduchá a navíc je již většina tagů defaultně připravena při tvorbě nového projektu.

V následujících odstavcích budou probrány záložky volby *Preferences*, které bude třeba modifikovat.

V panelu *Application* je nastavena verze používaného *.NET*, která kvůli použitým typům musí být minimálně verze 3.5. Je zde zvoleno, že výstupem po kompilaci bude dynamická knihovna s názvem *VectorFieldCalculate.Y2011.M04*. Tento název bude posléze použit pro identifikaci služby při jejím volání jinými službami. Také se zde volí

obor názvů, pod kterým bude v ostatních službách implementován přistup k prvkům *VectorFieldCalculate*.

V záložce *Build* je nutné změnit adresář výstupu kompilace na složku *bin* v umístění instalace RDS. V této složce se nachází všechny zkompilované služby.

Aby bylo možné spolehlivě odlišit různé verze výstupních souborů, používá RDS takzvané podepisování. Je v něm možno nastavit, aby bylo pro každou novou verzi služby vyprodukováno globálně-unikátní označení. Tento klíč je nezbytně důležitý pro bezchybné fungování *Proxy* dané služby. K tomuto „podepsání“ RDS používá soubor s příponou snk.

Tento soubor je nutné pro službu *VectorFieldCalculate* zvolit v odrážce *Signing* v nastavení projektu.

### 5.1.2 Třída VectorFieldCalculateState

Třída *VectorFieldCalculateState* obsahuje návratovou hodnotu při žádosti o výpočet nového vektoru pro pohyb. Instance této třídy je také použita pro zaznamenání vnitřního stavu služby *VectorFieldCalculate*. Obsahuje datový člen *Direction*, který je typu *Vector2*. Defaultně je inicializován výchozím konstruktorem jeho typu. Tato proměnná obsahuje dva členy typu *float*, které mají význam výchylky ve dvou osách souřadného systému pozice robota. Typ *Vector2* je součástí dynamické knihovny *RoboticsCommon.dll*. Po volání operace *Calculate* reprezentuje návratová hodnota třídu obsahující vektor, kterým se má robot pohybovat. Pro jeho transformaci na úhel pohybu vzhledem k jednotlivým osám je možné použít například funkci *atan2*.

Druhým datovým prvkem třídy *VectorFieldCalculateState* je proměnná *Calculations* celočíselného, neznaménkového typu. Tato slouží jen pro evidenci množství vyřízených výpočtů, a nemá proto na běh samotný žádný vliv. Pomocí této proměnné proběhly například předběžné testy rychlosti použití této služby.

Rychlost výpočtu přirozeně kolísá s počtem překážek, jejich rozložením a vzhledem k faktu, že je použit *JIT* jazyka C#. Přesnějších měření by mohlo být dosaženo například použitím *garbage collectoru* před zahájením měření, nebo změnou priorit aplikace a podobně (některé prvky CCR RDS brání v použití některých metod měření rychlosti aplikace). Pro hrací hřiště robotického fotbalu se šesti hráči a jedním míčem, za použití operačního systému *Windows 7* a procesoru *intel core i3 370*, je délka jednoho výpočtu v řádu jednotek milisekund. Tato hodnota zahrnuje i latenci zasílání zpráv mezi službami a prodlevu vyhodnocování řídící služby, byť přirozeně běží na jiném vlákně. V případě testování této služby výpočtem vektorů pro náhodné rozložení překážek a cíle v okolí pozice robota se délka jednoho výpočtu pohybuje ve zlomcích milisekund.

### 5.1.3 Třída VectorFieldCalculateInput

Naopak třída *VectorFieldCalculateInput* je využita pro obsažení dat zaslaných službě *VectorFieldCalculate*. Tato třída obsahuje tři datové prvky. Člen *Position* je typu *Vector2* a znamená aktuální pozici robota. Proměnná *Dest* obsahuje informace o cíli, kterého má být dosaženo a je typu *GoalStruct*. Konečně člen s názvem *Obstruct* obsahuje informace o všech překážkách, se kterými se má počítat. Datový prvek *Obstruct* je mírně komplikovanější, protože definuje překážky, kterých může být libovolný počet; je proto typu zásobník prvků typu *ObstructStruct*. Zásobník sám je typu *DssStack*, který má již implementované metody pro parsování, které jsou nutné pro zasílání zásobníku jako součásti zprávy.

Je nutné poznamenat, že pozice robota, překážek a cíle je naprosto relativní pouze vůči sobě navzájem. Takže při výpočtu je zcela jedno, do kterých míst překážky zasadíte. Jediné, co je třeba brát v úvahu, je nastavení intenzity působení překážek vzhledem k měřítku prostředí a jejich rozměrům.

**Třída ObstructStruct**

Tato datová struktura obsahuje několik základních konstruktorů a čtyři datové členy. V budoucnu by mohlo být implementováno i několik členských metod této třídy pro provádění samotných výpočtů vektorového pole, aby byl využit plný potenciál OOP a dodržen standard slušnosti pro manipulaci s členskými daty.

Členská data jsou v první řadě pozice daného objektu (překážky) s názvem *Pos*. Členská proměnná pro zaznamenání rychlosti pohybu překážky se nazývá *Vel* a je typu *Vector2*. Je to směr a rychlost překážky pro výpočet s pohybujícími se překážkami. Tento prvek je předložen pro případné budoucí studium této problematiky. Pro dosažení možnosti zpětné kompatibility je s ním počítáno již nyní, i když nemá zatím na výpočet vektoru pohybu žádný vliv. Prvek s názvem *R* typu *float* představuje poloměr dosahu vlivu překážky. Za tímto poloměrem již překážka nikterak nepůsobí, a na rozhraní mezi plochou obsaženou poloměrem a neobsaženou poloměrem vzniká potenciálový schod, kterého lze využít. Nakonec třída *ObstructStruct* obsahuje člen *I*, který nese intenzitu dané překážky.

Práce s intenzitami překážek již byla vysvětlena v předchozích kapitolách, v zásadě ale platí, že pokud je intenzita cíle větší než intenzita překážky, může dojít ke kontaktu s překážkou. Tohoto faktu je možné například využít pro zakomponování terénní nerovnosti nebo plochy s problematickými podmínkami pohybu. Pokud je intenzita cíle nastavena na hodnotu větší než intenzita překážky, algoritmus nevyhodnotí trasu skrz tento objekt, pakliže stojí mezi robotem a překážkou. Díky vylepšením představeným v této práci nemůže být robot k najetí na překážku „dotlačen“ překážkou jinou.

**Třída GoalStruct**

Pomocí jednoho ze základních pravidel Objektově Orientovaného Programování, takzvaného dědění, kterému přirozeně rozumí i jazyk C#, byla třída *GoalStruct* odvozena od třídy *ObstructStruct*. To nabízí použití a případné přetížení navrhovaných metod třídy *ObstructStruct*, a přirozeně také všech jejích datových členů. Původní třída byla přetížena a byl do ní přidán datový člen *Goal*. Ten je typu *Vector2* a značí pozici, na kterou se má cíl eventuálně dostat. V předchozí podkapitole zmíněný člen *Pos* je pro objekt typu cíl místo, na které se snaží robot najet. Úprava tohoto místa je ještě prováděna pomocí členu *R*, který u objektu cíl nemá jiný význam, protože intenzita cíle je konstantní všude. Význam zbývajících datových proměnných se již oproti mateřské třídě nemění.

### 5.1.4 CalculateHandler

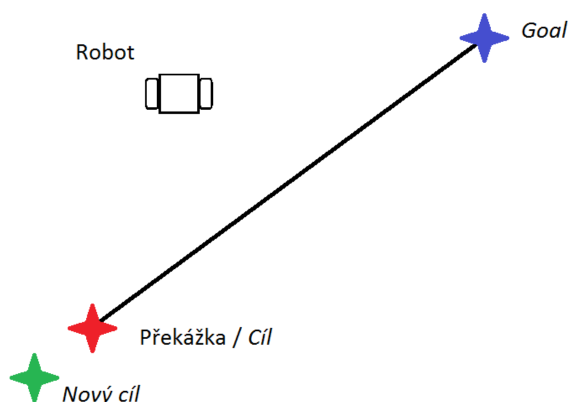
Prakticky nejdůležitější součástí služby *VectorFieldCalculate* je *callback* na přijetí zprávy třídy *Calculate* typu *get*. Jedná se o funkci *CalculateHandler* typu *IEnumerator<ITask>*, to znamená, že se pro provádění tohoto *callbacku* vyhrazuje nové vlákno služby. Chování této funkce je upraveno příznakem *ServiceHandlerBehavior.Exclusive*, což znamená, že v daný moment bude prováděna vždy jen jedna instance tohoto volání. Chování může být vzhledem k výkonnostním nárokům a optimálním potřebám volajících služeb upraveno mimo jiné například tak, aby došlo k rozvětvení služby *VectorFieldCalculate* pro každou volající službu jednou nebo vícekrát. Parametrem tohoto volání je instance typu *Calculate*, což je generický typ RDS, který v položce *body* obsahuje třídu s přijatou zprávou daného volání (jedná se o typ *VectorFieldCalculateInput*).

Výpočet započne inicializací základních proměnných, jako je například získání vstupních dat potřebných pro výpočet vektoru, tvorba proměnné pro návratovou hodnotu nebo pole potenciálů, které bude použito pro výpočet vektoru.

Pole potenciálů je čtvercová matice se třemi sloupci typu *float*. Tato matice vystihuje potenciály vektorového pole v těsné blízkosti robota. Všech devět prvků, z nichž středový je brán jako pozice samotného robota, reprezentuje potenciál dané pozice. S touto maticí potom stačí provést operaci gradient o dvou stupních volnosti, čímž získáme vektor největšího potenciálového spádu. Matematika použitá k výpočtu vektorového pole je popsána v kapitole věnující se samotným vektorovým polím. Je nutné poznamenat, že k tomuto výpočtu je možné použít kupříkladu i pole o pěti prvcích, zformovaných kolem pozice robota do kříže, nebo větší počet zkoumaných pozic. Příklad s devíti body byl vybrán proto, že dává dostatečně přesný výsledek s velmi malým požadavkem na strojový čas, neboť je zvolen efektivní výpočet gradientu.

Nejdříve je tedy matice potenciálů vyplněna působením cílového náboje. V případě, že je požadováno přiblížení se k cíli z určitého směru (vstupní vektory *Goal* a *Pos* nejsou shodné), je využita členská proměnná *R* k posunutí pozice cíle *Pos* o danou hodnotu směrem od pozice *Goal.*

Praktický dopad je takový, že například při hře robotický fotbal se robot nesnaží najet na střed míčku, který je jeho cílem, ale míří o *R* před něj s respektem k pozici *Goal*, která značí místo, kam má míč směřovat. Navíc v případě, kdy se robot nachází v pásmu za místem do kterého má najet, se v tomto kroku přidá do zásobníku *Obstruct* překážka na místo původní pozice *Goal*. Fakticky je tedy blíže k původní pozici *Goal*, než k pozici nové. Nově vložená překážka má opačnou polaritu, než cílový bod, a slouží k tomu, aby se robot nesnažil do nového cílového bodu přiblížit ze špatné strany. Na Obr 5.4 je vidět, kam se v případě, kdy se nachází robot v prostoru za cílem, přidá překážka. Obr 5.5 ilustruje případ, kdy je již robot před pozicí cíle, a není tedy nutno překážku přidat.

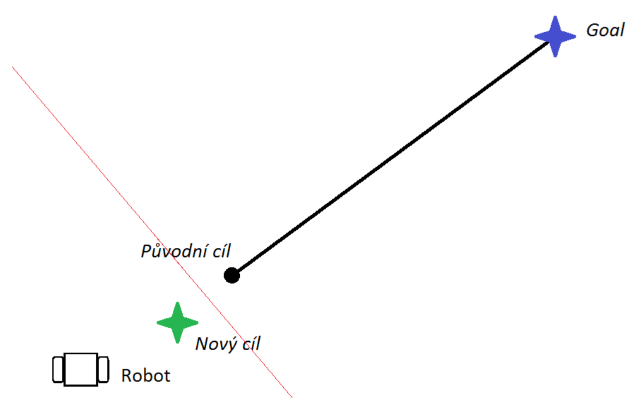



*Obr 5.4: Přidána překážka na pozici původního cíle*

Následně proběhne výpočet potenciálů překážek. Prvky jednotlivých překážek jsou načítány iterováním, prvek po prvku, ze zásobníku *Obstruct*. Nejprve je vyhodnocena platnost překážky. Pokud je překážka od cíle vzdálenější než robot, nebo se robot nenachází v jejím dosahu, nebo do sebe nemohou narazit, není s překážkou počítáno. V opačném případě je vliv překážky započítán na každou pozici matice potenciálů. Předpis samotného výpočtu je uveden v kapitole věnující se vektorovým polím.

Z vyplněné matice potenciálů je nyní možné pomocí záporu funkce gradient vypočíst vektor největšího spádu potenciálu. Tento vektor je tedy uložen do proměnné *Direction* typu *Vector2* obsažené v nové instanci třídy *VectorFieldCalculateState*.

Objekt s vypočteným vektorem je následně odeslán jako odpověď na původní zprávu na port udaný v parametru volání metody *CalculateHandler*. Pro dosažení naprosté exkluzivity volání této funkce počká ještě funkce na přijetí odpovědi volanou službou.




*Obr 5.5: Překážka není přidána na pozici původního cíle*

## 5.2 Služba VectorFieldStart

Tvorba služby pro samotný výpočet vektoru pohybu robota byla probrána v předchozí kapitole. Aby tuto službu bylo možné použít, je třeba vytvořit jinou službu, která jej bude volat.

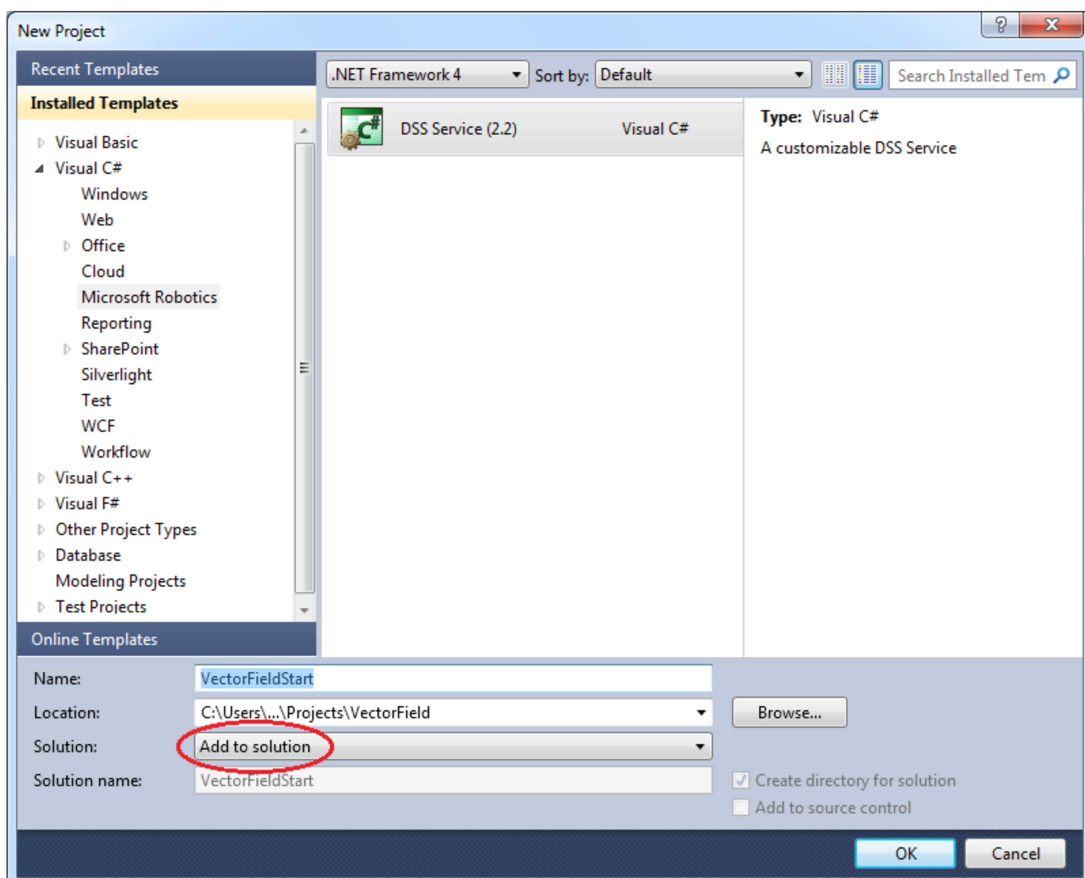
Služba *VectorFieldStart* bude tedy obsluhovat ostatní spuštěné služby, mezi nimi především *VectorFieldCalculate* a simulátor fyzikálního prostředí v RDS (*VSE*). Po načtení spouštěcího souboru a inicializaci sebe sama je spuštěna funkce *start* této služby. Ta následně spustí časovač, který bude v pravidelně zvolených intervalech volat funkci *TimerHandler*. Nejjednodušší způsob implementace časovače v RDS je pomocí utility *TimeoutPort*. *TimerHandler* je hlavní tělo služby *VectorFieldStart*, které obstarává běh celé aplikace.

Tuto službu je nutné vytvořit jako nový projekt v RDS. Optimální, zde popsané řešení, je vytvoření tohoto projektu jako součásti jedné aplikace zároveň se službou *VectorFieldCalculate*. Není to ovšem podmínkou. Ve většině případů práce se službou *VectorFieldCalculate* naopak programátor přichází do kontaktu pouze s knihovnou generovanou kompilací této služby. Samotná práce se ničím neliší, rozdíl je jen v pohodlnosti řešení.

K službě *VectorFieldStart* je opět nutné přiřadit reference. Vzhledem k tomu, že bude používat množství dalších služeb, bude počet referencí větší. Přidávané reference

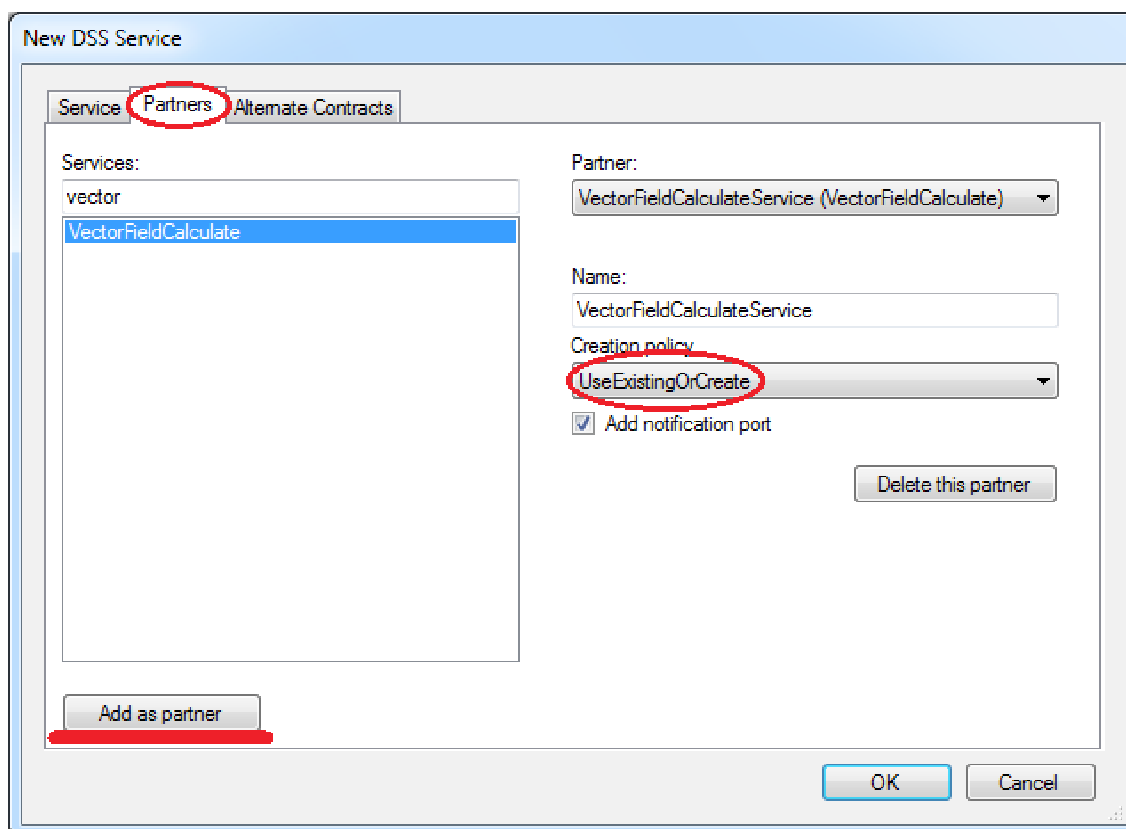
jsou *RoboticsCommon* a *RoboticsCommon.proxy* pro použití generických typů vlastních robotickým aplikacím v RDS; *Microsoft.Ccr.Adapters.WinForms*, *Microsoft.Xna.Framework*, *Simulation.Common*, *Simulation.Common.proxy*, *Simulation.Engine*, *Simulation.Engine.proxy*, *System.Windows.Forms*, *System.Data*, *System.Drawing* a *PhysicsEngine* pro obsluhu simulačního prostředí *VSE* v RDS; *SimulatedDifferentialDrive* a *SimulatedDifferentialDrive.proxy* pro vytvoření diferenciálně řízeného podvozku a konečně *VectorFieldCalculate* pro výpočet směru pohybu ovládaného robota. Názorný příklad vkládání referencí do projektu se nachází na začátku kapitoly **„Služba VectorFieldCalculate"**. Stává se, že při použití přiloženého projektu se nenačtou všechny reference správně. To je způsobeno jiným umístěním souborů referencí v počítači. Těmito soubory je především kořenová složka instalace RDS, *SDK* překladače, nebo jiná verze *.NET*. V tomto případě je nutné zkontrolovat adresy odpovídajících složek systému a RDS.

V aplikaci Microsoft Visual Studio je tedy přidána k projektu služba *VectorFieldCalculate* volbou *File > New > Project*. Opět je v záložce C# zvolen projekt typu Microsoft Robotics, nyní s názvem *VectorFieldStart*. V roletkovém menu *Solution* ale bude vybrána možnost *Add to solution*, jak je patrné z Obr 5.6.




*Obr 5.6: Přidání projektu do aplikace*

Po stisku tlačítka OK následuje výběr podrobných nastavení nové služby. Jako partnera je možné v této části přidat službu *VectorFieldCalculate*. V následujícím okně stačí dohledat službu v seznamu, a kliknutím *Add as partner* ji přidat. Na Obr 5.7 je vidět volba kritérií tvorby této partnerské služby. Je důležité navolit, aby byla tato partnerská služba spuštěna v případě, pokud tomu tak nebylo při inicializaci služby *VectorFieldStart*. Učiní se tak volbou *UseExistingOrCreate*. Obdobným způsobem je možno přidat jako partnerskou službu i simulační prostředí (*Simulation Engine*) a nezbytný blok diferenciálního pohonu (*SimulatedGenericDifferentialDrive*).




*Obr 5.7: Přidání partnerství*

Všechny tři výše zmíněné služby je také možno přehledně přidat v kódu. Jejich definice se musí nacházet v souboru *VectorFieldStart.cs*. Mimo tyto partnery bude mít nová služba navíc automaticky ještě partnera *SubscriptionManager*, který se stará o správný průběh propojení partnerských služeb, a službu *Arbiter*, která obhospodařuje zasílání zpráv. Pro každý z nich musí také být deklarován komunikační port, na který budou zasílány zprávy adresované dané službě.

Před podrobným výpisem jednotlivých součástí projektu *VectorFieldStart* je ještě v souladu se zde popisovanou implementací služby nutné nastavit tento projekt jako spouštěcí. To lze učinit v kontextovém menu názvu služby ve stromu projektů. Po

provedení je název projektu v tomto menu zvýrazněn. Tím bylo nastaveno, že soubor manifest této služby bude použit pro spuštění aplikace *DssHost*, je ale nutné, aby služba byla zkompilována pro nejnovější verzi služby *VectorFieldCalculate.* V nastavení projektových závislostí (v kontextovém menu názvu služby ve stromu projektů *Project Dependencies*) je zajištěno, že jako první proběhne kompilace služby *VectorFieldCalculate,* a teprve následně kompilace služby *VectorFieldStart*.

## 5.2.1 Vlastnosti projektu VectorFieldStart

Obdobně jako v kapitole **„Vlastnosti projektu VectorFieldCalculate“** bude tento text zaměřen na důležité prvky volby *Preferences* služby *VectorFieldStart*. Vyzdviženy jsou náležitosti, které je potřeba změnit oproti výchozímu nastavení nového projektu.

V záložce *Application* je uveden název generovaného souboru *dll*. Ten je použit v manifestu aplikace pro spuštění této služby.

Adresářem výstupu kompilace musí být adresář *bin* ve složce instalace RDS, který je nutno nastavit v záložce *Build*.

Záložka Build Events obsahuje možnost měnit příkazy provedené před a po kompilaci služby. V okně příkazu, který je spuštěn po kompilaci aplikace, je nutné rozvinout adresy parametrů programu DssHost tak, aby nedocházelo k chybám jejího běhu. Především je třeba zadat do parametru */referencepath* celou cestu k adresáři instalace RDS. V případě dalších chyb běhu *DssHost* je nutno obdobně upravit i zbytek řádku (někdy si Visual Studio zcela nerozumí s Robotic Developer Studiem a tato část projektu se jevila jako nejčastější zdroj problémů při kompilaci). Tento příkaz je zodpovědný za správné propojení nově vzniklé služby s již existujícími *proxy* jiných služeb.

V záložce *Signing* je specifikován klíč pro generování unikátního identifikátoru služby. Jeho funkčnost již byla vysvětlena v kapitole **Vlastnosti projektu VectorFieldCalculate**.

V záložce *Debug* je třeba se ujistit, že je spouštěn program *DssHost*, který jako jeden z potřebných parametrů přebírá název manifestu služby *VectorFieldStart*. V tomto projektu si manifest ponechává standardní název *VectorFieldStart.manifest.xml*. K povšimnutí je volba *Use remote machine*, která umožňuje spouštět služby na vzdálených počítačích.

## 5.2.2 Použití TimeoutPort

Při výpočtu vektoru pohybu se postupuje v takzvaných cyklech. Obdobně jako v průmyslu počítačových her, je i v případě navádění robota proveden jeden cyklus, následovaný čekáním. Tím je poskytnut prostor motorům k odeznění přechodových dějů, a nejsou na ně tedy vyvíjeny takové nároky. Obecně se v tomto intervalu,

nazývaném *idle*, dává prostor systému, který může ušetřený výkon využít jinak. Krok časovače, také nazýván *Heart Beat,* je volen dostatečně malý na to, aby se v tomto intervalu v zásadě nic nestalo, ale dostatečně velký k tomu, aby proběhly všechny výpočty s ním spojené. Dobrý *Timer* je navržen tak, aby byl jeden tik vynechán v případě, kdy cyklus výpočtu z jakéhokoliv důvodu nestihne proběhnout do konce vymezeného času. V RDS je vynechání jednoho *Heart Beat* docíleno užitím modifikátoru *exclusive* při volání *callback* timeru.

Časovač definovaný v této službě využívá zvláštního mechanismu, který mu umožňuje RDS. Nejprve je třeba definovat port, na který přichází zpráva v každém kroku timeru. Port je deklarován tak, aby přijímal zprávu s typem *DateTime*. Hodnota této zprávy není pro timer nijak závazná, je zde jen pro informaci při ladění. Následně je otevřen *handler* na tomto portu. Funkce *TimerHandler* je volána při přijetí každé zprávy a vzhledem k modifikátoru *Exclusive* probíhá vždy jen jedna instance tohoto volání. Přijímání na portu musí být nastaveno jako nepřetržité, aby se port po přijetí první zprávy neuzavřel. Poté je vytvořen přijímač volající metodu *IncrementTickHandler* s pomocí *TimeoutPort.* Ten po uplynutí každého *Heart Beat* přijme zprávu s určenou dobou prodlevy a není nastaven jako přetrvávající. V reakci na tuto přijatou zprávu proběhne zaslání zprávy, vyvolávající exkluzivně funkci *TimerHandler.* Funkce *TimerHandler* se provede a zašle další zprávu pomocí *TimeoutPort* na port timeru.

Nakonec je třeba timer nastartovat. To je provedeno zasláním zprávy pomocí *TimeoutPort* z funkce *start* služby.

### 5.2.3 Funkce TimerHandler

Tato funkce je volána pomocí timeru popsaného v předchozí kapitole. Jejím úkolem je zjistit data ze simulačního prostředí a spustit asynchronní metodu *MainTask*, která se postará o zavolání a vyhodnocení funkce *calculate*. Pro snížení latence a pro lepší přehled o přijatých datech se tato dvě volání dějí v opačném pořadí.

Zadáním této bakalářské práce je vytvoření již zmiňované služby *VectorFieldCalculate.* Při implementaci funkce *TimerHandler* služby *VectorFieldStart* je použit zjednodušený model shromažďování vstupních informací reálného světa, který ze simulátoru zjišťuje přesné údaje o všech entitách zasíláním dotazů. Ve skutečnosti by samozřejmě bylo nutné implementovat například službu laserového senzoru nebo kamery, a z nich data extrahovat. Právě fakt, že data o pozici a natočení entit je možné získat i pomocí jiných služeb, odlišuje *VSE* od většiny konkurenčních simulátorů.

Jako médium pro přenos dat o jednotlivých entitách v simulaci je zvolen zásobník, obsahující data typu *QuerySimulationEntityResponseType*, což je typ, který volí simulátor pro zaslání informací o zvolené entitě. V souboru manifest je totiž každý

předmět simulace nějak pojmenován. Požadovaná entita je tedy jednoznačně identifikována jménem.

Na začátku funkce se ověří, zda jsou ve zmiňovaném zásobníku nějaké prvky. Pokud ano, provede se zkopírování zásobníku voláním asynchronní funkce *MainTask*, která přijímá zásobník jako parametr. Následně je zásobník ve funkci *TimerHandler* vyprázdněn, a v průběhu stejného cyklu časovače *Heart Beat* je znovu naplněn daty ze simulace. Po zbytek cyklu se čeká na příchod asynchronní odpovědi simulátoru, takže simulátor aplikaci nebrzdí.

Zjišťování veškerých dat týkajících se fyzikálních vlivů jednotlivých entit a jejich atributů se děje pomocí známého názvu entity. Simulátoru je pro každou požadovanou entitu zaslána instance typu *VisualEntity*. V ní je specifikováno pouze jméno žádané entity. Jelikož je toto volání *Query* typu *get*, vrací odpověď. Odpovědi jednotlivých žádostí mohou přijít v různém pořadí, a proto jsou pouze vložena do zásobníku a tříděna až v metodě *MainTask*.

### 5.2.4 Funkce MainTask

V této exkluzivně běžící funkci typu *IEnumerator<ITask>* probíhá volání služby *VectorFieldCalculate*. Vstupním parametrem funkce je zásobník naplněný podrobnostmi o jednotlivých entitách simulace, které jsou brány v potaz. Zásobník má obor platnosti pouze pro funkci *MainTask* a zaniká tedy po jejím skončení.

Při volání *Calculate* je zasílán objekt typu *VectorFieldCalculateInput*. Nová instance tohoto typu je tedy vytvořena jako první a je postupně naplněna.

Iterací vstupního zásobníku jsou získana data o všech třech typech entit. Typem entity může být robot, pro který je vektor počítán, nebo pozice cíle. V případě cíle, na který má být najeto, je pozice *Goal* volena jako konstantní globální proměnná, a globální proměnná *KickTarget* je nastavena na hodnotu *true*. Pokud není entita ani jedním z uvedených typů, je brána jako překážka. Proměnná typu *VectorFieldCalculateInput* je postupně naplněna hodnotami všech entit. Jako hodnoty poloměru dosahu *R* a intenzity prvků *I*, potřebné pro chod služby *VectorFieldCalculate*, jsou zvoleny standardní hodnoty. Tyto hodnoty jsou vyčteny z rozměrů a důležitosti entit, a jsou voleny ve vzájemné harmonii tak, aby byl výpočet vektoru pohybu vyvážený.

Poté, co jsou datové členy *Dest*, *Obstruct* a *Position* proměnné typu *VectorFieldCalculateInput* naplněny, může dojít k poslání zprávy s požadavkem na výpočet vektoru pohybu. Ten je volán s modifikátorem *yield return*, což znamená, že do jejich navrácení a vyhodnocení neskončí exkluzivní funkce *MainTask*.

V těle volání zprávy je deklarován delegát, který provede vyhodnocení odezvy. V tomto místě aplikace je již známý původní úhel natočení robota (ze vstupního parametru

funkce). Přijatá zpráva od služby *VectorFieldCalculate* má typ *Vector2*. To znamená, že hodnota výchylky ve dvou osách souřadné soustavy nově požadovaného směru je známa ve formě vektoru. Tuto hodnotu je třeba nejprve převést na úhel (matematická funkce *Atan2*) a poté z ní zjistit úhel požadovaného pootočení robota. Proběhne vyhodnocení a nastavení rychlostí obou kol diferenciálního podvozku podrobněji popsané dále.

Optimálním řešením poměru rychlostí kol robota by bylo použití regulační smyčky. Některé metody jsou popsané například v [29], [30].

V tomto zjednodušeném řešení je ale v každém volání funkce *MainTask* možné nastavit výkony kol jen jednou. Z toho důvodu, a protože není možno opakovaně zjišťovat natočení robota, nelze použít zpětnovazební řízení podvozku. K výpočtu poměru výkonů je tedy použito rozložení žádaného úhlu natočení na ose $x$ pomocí funkce cosinus. Pokud je požadovaný úhel otočení robota 0 radiánů, je poměr výkonu kol roven 1. Obě kola se tedy točí se stejným výkonem. S rostoucím žádaným úhlem pootočení klesá poměr výkonů kol ve prospěch kola na vnější straně. Počítaná hodnota funkce cosinus je upravena tak, aby se kola při nutnosti pootočení robota o více než jednu třetinu radiánu točila různým směrem. Toto řešení někdy způsobuje mírnou oscilaci robota při rozjíždění, která ale při projíždění kružnic již patrná není.

Jinou možností pootočení robota o daný úhel je zavolání funkce služby *SimulatedDifferentialDrive*, která tuto činnost obstarává. Nevýhodou řešení je styl jízdy robota, který neustále zastavuje. Při průjezdu po eliptických drahách vlastních vektorovým polím je tento nedostatek velice patrný.

## 5.3 Manifest VectorFieldStart.manifest.xml

Projekty spuštěné pomocí aplikace *DssHost* používají takzvané manifesty k načtení spouštěných služeb a objektů pro *CCR*. Projekt *VectorFieldStart* byl zvolen jako spouštěcí, a proto je v jeho složce umístěn manifest celé aplikace. Nemusí tomu tak ovšem být. V nastavení tohoto projektu je možné zvolit jiný parametr pro spuštění *DssHost*.

Pokud byly všechny reference pro tento projekt správně přidány, bude soubor typu manifest, zahrnujíci všechny služby, vytvořen automaticky. Vygenerovaný manifest je poté možno upravit  například pomocí nástroje *ManifestEditor* [3].

Pro správné vykreslení entity míčku v simulaci je nutné zkopírovat jeho texturu do složky */store/media* v adresáři instalace RDS. Jedná se o obrázek s názvem *default.jpg* přiložený ke kódu.

V manifestu této robotické aplikace je nejdříve spuštěna služba *simulationengine*. Tomu je pomocí partnerské vazby přidán soubor formátu *xml,* obsahující rozložení entit v simulaci. Ve vytvořeném projektu jsou čtyři různé varianty upravených souborů entit.

Spustit aplikaci pro různá rozložení je možné zrušením jejich zakomentování. Každý z těchto souborů musí obsahovat vazbu na službu S*imulatedDifferentialDrive*. Další spouštěnou službou je *MotorBaseWithDrive*. Tato služba je použita k obsluze robotického podvozku *SimulatedDifferentialDrive*. Nakonec manifest spouští službu *VectorFieldStart*. Ta ve své inicializační části spouští i službu *VectorFieldCalculate*, protože je v definici partnerského vztahu použit modifikátor *UseExistingOrCreate*. Služba *VectorFieldCalculate* by taktéž mohla být spuštěna pomocí tohoto manifestu.

# 6 ZÁVĚR

Ve své bakalářské práci jsem udělal přehled robotických simulátorů, které mohou být použity pro testování různých interpretů ovládání jednoosého, diferenciálně řízeného robota pomocí joysticku, a které z uvedených důvodů považuji za zajímavé. Popsal jsem také výhody použití prostředí RDS k tomuto úkolu. Navázal jsem tím na svoji semestrální práci, ve které jsem sepsal manuál popisující průběh simulace a od toho odvozených událostí v RDS.

Ve své práci jsem představil metodu návrhu trajektorie jednoosého, diferenciálně řízeného robota pomocí vektorových polí a uvedl některé výhody a nevýhody tohoto řešení. Doplnil jsem několik vlastních vylepšení této metody, z nichž některá jsou v práci pouze předestřena, a jejich propracování a implementace by mohla být předmětem dalšího výzkumu.

Implementoval jsem upravenou metodu vektorových polí s potenciálovým řešením daného problému v prostředí Robotics Developer Environment v programovacím jazyce C#. Výsledkem práce je služba *VectorFieldCalculate*, která může být použita jako součást jiných projektů pro výpočet trajektorie pohybu.

Naprogramoval jsem také jednoduchou službu *VectorFieldStart*, která voláním služby *VectorFieldCalculate* získává optimální vektor pohybu a ovládá robota v simulaci robotického hřiště s míčem a překážkami. Ovládaný robot se snaží dostat míček na určitou pozici reprezentující branku.

Budoucí práce, rozvíjející toto téma, mohou být zaměřeny na implementaci strategie hry robotický fotbal nebo mohou implementovat globální plánovač trasy. Pro plánování trajektorie na lokální úrovni mohou použít službu *VectorFieldCalculate.*

# Literatura

[1] *Microsoft* [online]. 2010 [cit. 2010-11-2]. What's New. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/cc998522>.

[2] *Microsoft* [online]. 2010 [cit. 2010-11-2]. Welcome to Robotics Developer Studio. Dostupné z WWW: <http://msdn.microsoft.com/library/bb648760>.

[3] DÁVID, Muzika. *Inteligentní řízení robotického fotbalisty joystickem I* . Brno, 2010. 20 s. Semestrální práce. Vysoké Učení Technické v Brně, Ústav Automatizace a Měřící Techniky.

[4] *Anykode.com* [online]. 2011 [cit. 2011-05-22]. Marilou - keys features. Dostupné z WWW: <http://www.anykode.com/mariloukeyfeatures.php>.

[5] *Anykode.com* [online]. 2011 [cit. 2011-05-22]. Marilou - online documentation. Dostupné z WWW: <http://doc.anykode.com/frames.html?frmname=topic&frmfile=index.html>.

[6] *Uni-koblenz.de* [online]. 2006 [cit. 2011-05-22]. RoboLog. Dostupné z WWW: <http://www.uni-koblenz.de/FB4/Institutes/IFI/AGKI/Research/Current/Robolog.html>.

[7] MORGAN, Sara. *Programming Microsoft Robotics Studio*. Washington : Microsoft Press A Division of Microsoft Corporation, 2008-03-12. 288 s. ISBN 9780735624320.

[8] *Microsoft* [online]. 2010 [cit. 2011-01-10]. Using the DSS New Service Visual Studio Wizard. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/dd939185.aspx>.

[9] *Wiphala* [online]. c2010 [cit. 2011-01-12]. Microsoft Visual Simulation Environment. Dostupné z WWW: <http://www.wiphala.net/courses/robotics/MSROBOTICS/2008-I/msvse.html>.

[10] LOPEZ DE TORO, Carlos. *Microsoft Robotics Studio Walk-through* [online]. Barcelona : [s.n.], 2007 [cit. 2011-01-10]. Dostupné z WWW: <http://www.recercat.net/bitstream/2072/14089/2/PFC+Carlos+Lopez+de+Toro_Walk-through.pdf>.

[11] ERBEN, Vojtěch. *Inteligentní řízení robotického fotbalisty joystickem*. Brno, 2010. 22 s. Semestrální práce. Vysoké Učení Technické v Brně, Ústav Automatizace a Měřící Techniky.

[12] *Vutbr.cz* [online]. 2001 [cit. 2011-05-22]. SIMROBOT. Dostupné z WWW: <http://www.uamt.feec.vutbr.cz/robotics/simulations/amrt/simguide_cz.html>.

[13] *Sourceforge.net* [online]. 2003 [cit. 2011-05-22]. PyRobbie. Dostupné z WWW: <http://pyrobbie.sourceforge.net/home_en_1.html>.

[14] *Sourceforge.net* [online]. 2011 [cit. 2011-05-22]. Simbad 3D Robot Simulator. Dostupné z WWW: <http://simbad.sourceforge.net/>.

[15] Robotics simulator. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 31, 1, 2008, last modified on 9. 3. 2011 [cit. 2011-05-22]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Robotics_simulator>.

[16] HALLIDAY, David; RESNICK, Robert; WALKER, Jearl. *Fyzika : Část 3 Elektřina a magnetismus*. 1. vydání. Brno : VUTIUM, 2000. 314 s. ISBN 80-214-1868-0.

[17] Electric field intensity. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 18. 4. 2008, last modified on 16. 5. 2011 [cit. 2011-05-23]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Electric_field_intensity>.

[18] Vector Field Histogram. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 30. 6. 2008, last modified on 20. 6. 2010 [cit. 2011-05-23]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Vector_Field_Histogram>.

[19] Hrabec, J.; Honzik, B.: "Mobile robots playing soccer," Advanced Motion Control, 2002. 7th International Workshop on , vol., no., pp. 510- 513, 2002.

[20] J.C.Wolf,P.Robinson,J.M.Davies,"Vector field path planning and control of an autonomous robot in a dynamic environment." Proceedings of the 2004 FIRA World Congress, Busan, Korea.

[21] GREEN, Bill. *Pages.drexel.edu* [online]. 2002 [cit. 2011-05-23]. Edge Detection Tutorial. Dostupné z WWW: <http://www.pages.drexel.edu/~weg22/edge.html>.

[22] Gradient. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, [cit. 2011-05-23]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Gradient>.

[23] KOREN, Yoram; BORENSTEIN, Johann. Potential Field Methods and Their Inherent Limitations for Mobile Robot Navigation. *International Conference on Robotics and Automation*. 1991, s. 1398-1404.

[24] HWANG, Yong K.; AHUJA, Narendra. A Potential Field Approach to Path Planning. *IEEE Transactions on Robotics and Automation*. 1992, 8, 1, s. 23-32.

[25] SEČKAŘ, Jan. *Algoritmy a rasterizace 2D grafických objektů*. Zlín, 2007. 44 s. Bakalářská práce. Univerzita Tomášě Bati ve zlíně, Fakulta aplikované informatiky.

[26] *Microsoft* [online]. 2010 [cit. 2011-01-10]. Using the DSS New Service Visual Studio Wizard. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/dd939185.aspx>.

[27] *Microsoft* [online]. 2010 [cit. 2011-01-12]. Simulation Services Overview. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/dd146040.aspx>.

[28] AXELROD, Ben. *Benaxelrod* [online]. 2010 [cit. 2011-01-12]. Microsoft Robotics Developer Studio Extras. Dostupné z WWW: <http://www.benaxelrod.com/MSRS/>.

[29] SHOULING, He. Feedback Control Design of Differential-Drive Wheeled Mobile Robots. *IEEE*. 2005, 05, s. 135-140.

[30] EVGRAFOV, V. V.; PAVLOVSKY, V. V.; PAVLOVSKY, V. E. Dynamics, Control, and Simulation of Robots with Differential Drive. *Journal of Computer and Systems Sciences Internetional*. 2007, 5, 46, s. 836-841. ISSN 1064-2307.

# Seznam příloh

Příloha 1.
CD se zdrojovými kódy v jazyce C# pro Visual Studio 2010, soubory manifest, kódy pro program MATLAB a s elektronickou podobou bakalářské práce.