VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Microsoft Robotic Studio a .NET Micro Framework

Microsoft Robotic Studio and .NET Micro Framework

2010

Václav Svatoň

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 5. května 2010 . . . .

Na tomto místě bych chtěl především poděkovat svému vedoucímu práce Ing. Janu Martinoviči Ph.D., za ochotu a pomoc při tvorbě této práce.

Abstrakt

Tato diplomová práce se zabývá dvěma nástroji, vyvinutými společností Microsoft a spadajícími do oblasti programování mikroprocesorů a robotiky. První nástroj je .NET Micro Framework, který je určen pro embedded systémy. Jeho hlavní předností je možnost psát programy v řízeném kódu s výhodami jež přináší platforma .NET Framework. V této části je cílem práce především vytvoření ukázkové úlohy, demonstrující některé pokročilejší funkce tohoto frameworku, které se objevily s jeho novou verzí. Druhý nástroj je Microsoft Robotics Developer Studio, což je nástroj pro kontrolu a simulaci robotů. Díky servisně orientované architektuře umožňuje tento nástroj práci jak s opravdovými roboty, tak i s roboty jež jsou pouze součástí grafického simulátoru. Zde je cílem podrobněji se seznámit s grafickými a fyzikálními možnostmi tohoto simulátoru. Na závěr popíši možný způsob, jakým by se v praxi dali oba zmíněné nástroje propojit. To znamená najít společný způsob komunikace obou nástrojů, pro jejich možné praktické využití.

Klíčová slova: .NET Micro Framework, embedded, Microsoft Robotics Developer Studio, roboti, simulace, fyzika, webové služby

Abstract

This thesis describes two tools developed by Microsoft and falling into the area of programming microprocessors and robotics. The first tools is .NET Micro Framework which is designed for embedded systems. Its main advantage is the ability to write programs in managed code with the benefits of .NET platform. This part of the thesis is demonstrating some of the advanced features of this framework which appeared with its new release. The second tool is the Microsoft Robotics Developer Studio which is a tool for controlling and simulating robots. Thanks to a service-oriented architecture this tool allows to work with real robots and also robots that are only part of the graphic simulator. Here the aim is to further acquaint themselves with the graphics and physics capabilities of the simulator. In conclusion, I will describe a possible way to connect these tools. It is meant to find a common way of communication between both tools for their potential practical applications.

Keywords: .NET Micro Framework, Embedded, Microsoft Robotics Developer Studio, Robots, Simulation, Physics, Web services

Seznam použitých zkratek a symbolů

ADC	–	Analog Digital Converter, převodník analogového signálu na signál digitální
ARM9	–	architektura ARM 32bit procesorů u níž byla von Neumannova koncepce nahrazena Harvardskou
CCR	–	Concurrency and Coordination Runtime, komponenta pro práci s asynchronními operacemi
DOF	–	Degree Of Freedom, označení pohybu/rotace objektu vzhledem k určitým osám
DPWS	–	Devices Profile for Web Services, množina implementací zajišťující výměnu zpráv, objevení zařízení a notifikaci událostí v prostředí webových služeb
DSS	–	Decentralized Software Services, aplikační model pro práci s webovými službami
Embedded systémy	–	většinou jednoúčelové počítače navrhnuté pro specifické funkce, mohou být také real-time systémy
FIFO	–	First In First Out, abstraktní datový typ realizující frontu
FPS	–	Frame Per Second, snímková frekvence, počet snímku zobrazených za sekundu
HTTP	–	Hypertext Transfer Protocol, internetový protokol, původně určený pro výměnu hypertextových dokumentů
I2C	–	Inter-Integrated Circuit, multi-masterová sériová sběrnice používaná pro připojování nízkorychlostních periferií
Managed code	–	řízený kód, který vykonává Virtual Machine, zajišťující větší bezpečnost a ochranu při jeho provádění
MMU	–	Memory Management Unit, hardware komponenta řídící přístup do paměti vyžadovaný CPU, překládá virtuální adresu na fyzickou, ochranu paměti, přidělování sběrnice ...
MRDS	–	Microsoft Robotics Developer Studio
Native code	–	nativní kód, procesorem vykonávaný kód
PDA	–	Personal Digital Assistant, malý kapesní počítač ovládaný nejčastěji stylusem a dotykovou obrazovkou
PWM	–	Pulse Width Modulation, diskrétní modulace používaná pro přenos analogového signálu pomocí dvouhodnotového signálu

RAM	–	Random Access Memory, rychlé paměti s náhodným přístupem, pro opakovaný zápis a čtení informace
SOAP	–	Simple Object Access Protocol, protokol pro výměnu zpráv založených na XML
SPI	–	Serial Peripheral Interface, sériová sběrnice pro komunikaci mezi mikroprocesory a integrovanými obvody
UART	–	Universal Asynchronous Receiver/Transmitter, hardware převádějící sériová data na paralelní
WCF	–	Windows Communication Foundation, aplikační uživatelské prostředí v .NET Frameworku, umožňující vytváření servisně orientovaných aplikací
WSDAPI	–	Web Services on Devices API, imlementace DPWS pro Windows Vista a Windows Server 2008
XML	–	Extensible Markup Language, značkovací jazyk používaný pro popis strukturovaných dat

Obsah

1 Úvod 4
1.1 Struktura práce 4

2 .NET Micro Framework 6
2.1 Tahoe-II Development Kit 8
2.2 Ukázková aplikace 9

3 Microsoft Robotics Developer Studio 20
3.1 Architektura 20
3.2 Simulace a simulační prostředí 26

4 Propojení .NET Micro Frameworku a Microsoft Robotics Developer Studia 48
4.1 .NET Micro Framework a DPWS 48
4.2 Microsoft Robotics Developer Studio a DSSP 49
4.3 Vzájemná komunikace 50

5 Závěr 52

6 Reference 53

Seznam obrázků

1 Architektura .NET Micro Frameworku - převzato z [2] 7
2 Tahoe-II Development Kit - převzato z [6] 8
3 Blokové schéma Meridian modulu - převzato z [11] 9
4 Blokové schéma Tahoe-II Development Kitu - převzato z [23] 10
5 Kit Tahoe-II s konstrukcí obsahující servomotorky a minikameru 11
6 Obrazovka aplikace TempWindow 13
7 Obrazovka aplikace ImageWindow 14
8 Ovládání servomotorků pomocí PWM 16
9 Příkaz SYNC - převzato z [3] 16
10 Obrazovka aplikace CameraWindow 17
11 Pauznutá obrazovka aplikace CameraWindow 17
12 Postup získání JPEG Preview obrázku - převzato z [3] 18
13 Architektura Robotics Developer Studia - převzato z [21] 21
14 Schéma webové služby - převzato z [14] 25
15 Schéma principu předplácení webových služeb - převzato z [14] 26
16 Okno simulace zobrazené ve VSE 27
17 Komponenty simulačního prostředí 28
18 Vykreslená 3D scéna s kuchyní 30
19 Základní scéna obsahující zem, oblohu, zdroj světla a kameru 31
20 Lego XNT Tribot, IRobot, Pioneer 3DX, KUKA LBR3 34
21 Typy fyzikálních modelů (Sphere, Capsule, Box, Convex, Triangle) 35
22 Nakloněná rovina s autíčky (zobrazen také fyzikální model) 39
23 Grafický a fyzikální model autíčka 40
24 Pohyb autíček 41
25 Vygenerovaný výškový terén 42
26 Výškový terén s množstvím vložených entit 43
27 Pracovní módy spojů 43
28 Kyvadla 46
29 Princip komunikace MRDS s fyzickým robotem 50

Seznam výpisů zdrojového kódu

1 Spustitelná metoda Main 11
2 Události vložení/vyjmutí paměťové karty 13
3 Konstruktor třídy AccControl 15
4 Vytvoření instance třídy Port a PortSet se zápisem na porty metodou Post 21
5 Přečtení zprávy z pasivního portu metodou Test 22
6 Přečtení zprávy z aktivního portu pomocí třídy Arbiter.Receive 22
7 Použití tříd Dispatcher a DispatcherQueue pro plánování úloh 23
8 Registrace partnerské služby SimulationEngine 28
9 Vykreslení 2D přímky do scény s použitím XNA Frameworku 29
10 Vytvoření SingleShapeEntity typu SphereShape 32
11 Vytvoření MultiShapeEntity (model činky) 32
12 Vytvoření TriangleMeshEnviromentalEntity s grafickým modelem devil.obj 36
13 BoxShape entita s materiálem a barvou 37
14 Metoda Push pro „postrčení“ autíčka 41
15 Vytvoření spoje pro rotaci kolem všech os 46

1 Úvod

V dnešní době dochází k stále většímu vývoji nástrojů v oblasti programování mikroprocesorů a robotiky. Mezi hlavní průkopníky obou těchto odvětví lze bezpochyby zařadit společnost Microsoft[12].

.NET Micro Framework[1] zastupuje oblast programování mikroprocesorů, přesněji tedy Embedded systémy. To jsou většinou malé jednoúčelové systémy se zabudovaným řídícím procesorem. Jedná se například o kalkulačky či PDA. .NET Micro Framework umožňuje psát programy v moderních programovacích jazycích (C#, Visual Basic, ...) a s využitím pokročilých vývojových nástrojů (Microsoft Visual Studio[15]). Aplikace napsané v .NET Micro Frameworku již nevyžadují žádný operační systém a jsou označovány jako samobootovací (bootable runtime).

Oblast robotiky zastupuje nástroj Microsoft Robotics Developer Studio[13], dříve také označován jako Microsoft Robotic Studio (MRS). Tento framework umožňuje kromě samotného vytváření a spouštění aplikací psaných pro roboty, také jejich monitorování či řízení vzájemné koordinace. Celý framework je postaven na servisně orientované architektuře, s čímž jsou spojeny dvě hlavní komponenty systému. První z nich je CCR[1][4], jež má za úkol zjednodušit psaní a řízení asynchronních aplikací které spolu komunikují prostřednictvím zasílaných zpráv. Druhá komponenta je DSS[2][7], což je aplikační model pro práci s webovými službami, postavený nad CCR. Obě tyto komponenty mají za úkol co nejvíce zjednodušit psaní aplikací pro roboty a dosáhnout větší škálovatelnosti při jejich návrhu.

Oba zmíněné nástroje mají především ulehčit práci programátorům a díky .NET platformě na níž jsou postaveny i celkově urychlit samotný vývoj aplikací, čímž se také zkracuje doba potřebná pro jejich uvedení na trh.

1.1 Struktura práce

Celá práce je rozdělena do tří hlavních částí. V první části se budeme věnovat .NET Micro Frameworku, který slouží k programování mikroprocesorů ve skupině embedded systémů. S příchodem nové verze tohoto frameworku se objevily i nové funkce, jako například práce s dotykovým displejem, senzorem teploty či accelerometrem. Tato část se zabývá právě těmito funkcemi a obsahuje také popis ukázkové úlohy, vytvořené na vývojovém kitu Tahoe-II[22], jež obsahuje komponenty, potřebné právě pro demonstraci těchto nových funkcí.

Druhá část se zabývá Microsoft Robotics Developer Studiem, jež kromě samotného řízení robotů umožnuje i simulaci jejich chování v reálném světě, tedy bez nutnosti mít robota fyzicky k dispozici. Vzhledem k množství funkcí a komplexnosti tohoto nástroje se podrobněji zaměříme právě na onu zmíněnou simulaci. Přesněji tedy na grafické a fyzikální možnosti simulátoru v Microsoft Robotics Developer Studiu a jejich využití.

[1]Concurrency and Coordination Runtime, komponenta pro práci s asynchronními operacemi
[2]Decentralized Software Services, aplikační model pro práci s webovými službami

Ve třetí části popíši možný způsob propojení mezi těmito dvěma druhy nástrojů. Tato část slouží jako náhled na možnost způsobu komunikace mezi .NET Micro Frameworkem a Microsoft Robotics Developer Studiem.

2 .NET Micro Framework

Tato kapitola slouží jako velmi stručný úvod k .NET Micro Frameworku. Protože jsem se .NET Micro Frameworkem zabýval již ve své bakalářské práci [20], je zde uveden pouze soupis několika základních vlastností tohoto frameworku. Pro získání podrobnějších informací o celé architektuře systému doporučuji nahlédnout buď do mé bakalářské práce, nebo do jiných dostupných zdrojů zabývajících se .NET Micro Frameworkem[8].

Jak již bylo několikrát zmíněno, .NET Micro Framework se řadí do kategorie Embedded systémů. Do této kategorie spadají také operační systémy Windows XP Embedded či Windows CE. Windows XP Embedded jsou postaveny na architektuře NT a vycházejí z Windows XP. Používají se ve větších zařízeních jako bankomaty či pokladny a umožňují uživateli poskládat si z komponent vlastní operační systém. Naopak Windows CE jsou postaveny na zcela rozdílném jádru a používají se převážně v Pocket PC. Oproti Windows XP Embedded mají i podstatně menší paměťové a energetické nároky, díky čemuž jsou uzpůsobeny právě pro menší zařízení.

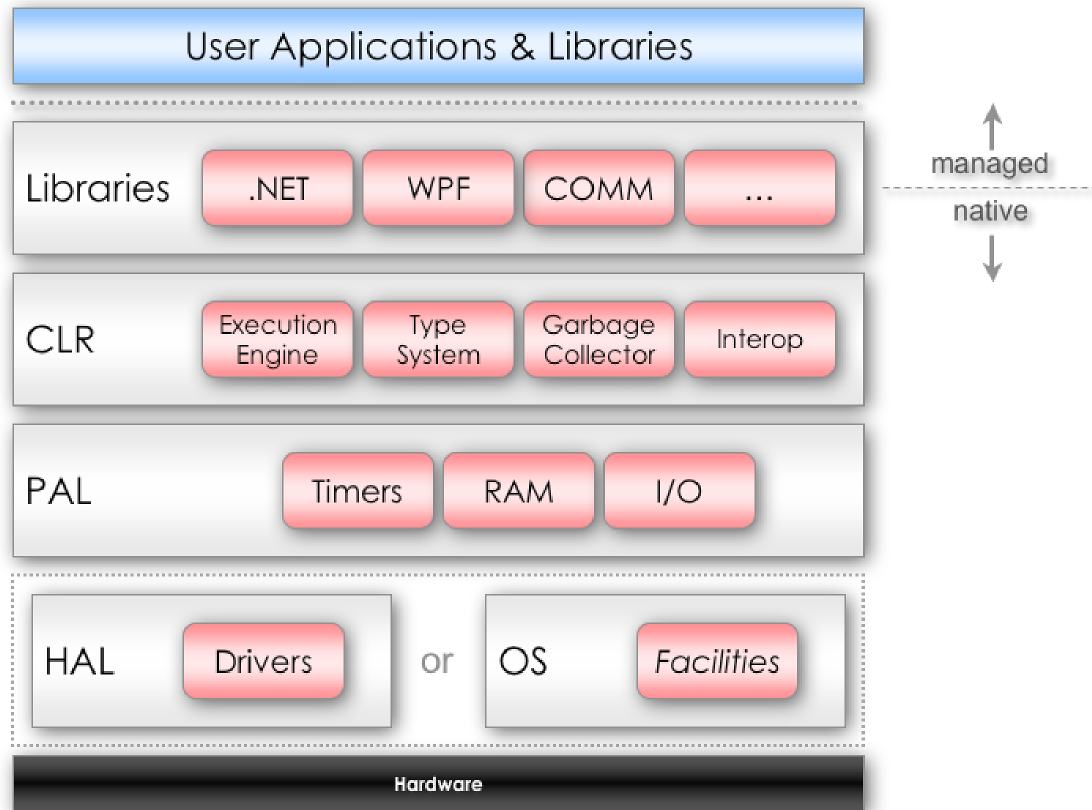
.NET Micro Framework se se svými paměťovými a energetickými nároky řadí ještě o stupínek níže, než zmiňované Windows CE. Pro svůj běh potřebuje pouze 300KB paměti RAM a nevyžaduje MMU, díky čemuž najde své uplatnění právě u těch nejmenších zařízení s mnohdy i levnějšími procesory, než jak je tomu u Windows XP Embedded či Windows CE. Díky tomu, že .NET Micro Framework obsahuje služby nezbytné k běhu aplikace (správa procesů, obsluha přerušení, ...), není zde již třeba operačního systému, který by tyto funkce zajišťoval. Proto nesou takovéto aplikace označení samobootovací.

Výhody platformy .NET Micro Framework:

- běh aplikací bez operačního systému,
- menší cena hardwaru (levnější procesory, menší nároky na paměť),
- aplikace v managed kódu,
- podpora Visual C# a .NET,
- vývojový nástroj Visual Studio s možností debugování na zařízení,
- menší napěťové nároky.

Architekturu .NET Micro Frameworku lze rozdělit do 4 základních vrstev. Jedná se vrstvy HAL, PAL, CLR a Libraries (obrázek 1). Níže je také ve stručnosti popsáno, k čemu každá z uvedených vrstev slouží.

HAL (Hardware Abstraction Layer)
Základní vrstva frameworku poskytující rozhraní pro práci s periferiemi umístěnými na hardwaru. V zásadě nahrazuje část operačního systému starající se o vstupy a výstupy, přerušení a časovače. Z pohledu vrstvy HAL zpracovává .NET Micro Framework jediné



Obrázek 1: Architektura .NET Micro Frameworku - převzato z [2]

vlákno aplikace, což má své výhody i nevýhody. Nedochází zde k problémům kritické sekce, ale naopak se zde musí provádět periodické přepínání zároveň běžících procesů.

PAL (Platform Abstraction Layer)
Účelem této vrstvy je zprostředkovat časovače, asynchronní komunikaci, struktury dat a další funkce vrstvě CLR a tím dále rozšířit funkcionalitu vrstvy HAL.

CLR (Common Language Runtime)
Jedná se o malé optimalizované prostředí sloužící k zavádění a provádění managed kódu, přičemž je považováno za jádro celé architektury frameworku. Přináší tedy .NET Micro Frameworku veškeré výhody managed kódu, jako je větší bezpečnost, validace, zotavení a další. Umožňuje tedy běh aplikací bez operačního systému, vykonávání programu z paměti ROM nebo Flash či zavádění funkčních knihoven přímo do zařízení. V neposlední řadě také používá vícevláknový plánovač, takže z pohledu CLR vrstvy může uživatel psát vícevláknové aplikace jako v klasickém .NET Frameworku.

Obrázek 2: Tahoe-II Development Kit - převzato z [6]

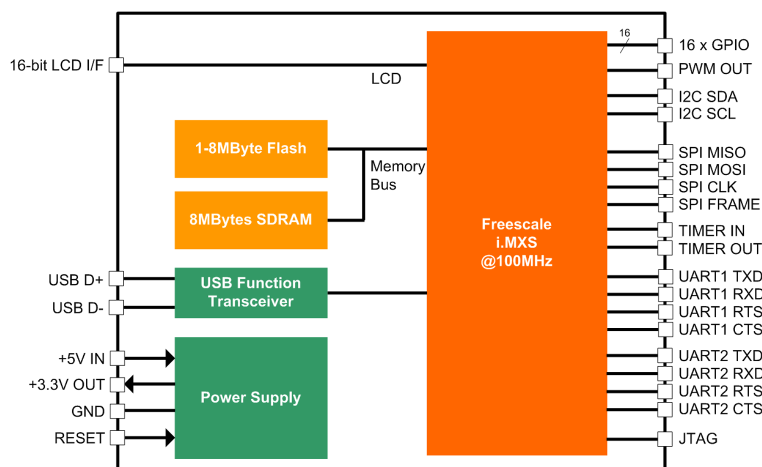
Libraries
.NET Micro Framework obsahuje značnou část standartní sady .NET knihoven, v klasickém .NET Frameworku také označovanou jako BCL (Base Class Libraries). Některé knihovny zůstaly nezměněny, některé byly přesunuty do nových namespace a některé nové knihovny, specifické právě pro .NET Micro Framework (Microsoft.SPOT), byly do sady knihoven přidány.

Aktuální verze .NET Micro Frameworku je 4.0. Od této verze je framework označen jako „open source“, takže je zbaven veškerých poplatků na zařízení a také jsou zdarma dostupné jeho zdrojové kódy. S touto verzí přichází i novinky jako verzování assembly, podpora pro HTTP, online synchronizace času a některé nové nativní funkce (watchdog, napájení, ...).

2.1 Tahoe-II Development Kit

Tahoe-II Development Kit (obrázek 2) je nástupcem populárního vývojového kitu Tahoe od společnosti Device Solutions[6]. Tahoe-II je stejně jako jeho předchůdce osazen Meridian modulem, mezi jehož hlavní komponenty patří ARM9 procesor, SDRAM a Flash paměť. Obsahuje také podporu pro komunikaci přes UART, I2C, SPI či USB. Blokové schéma modulu s jeho základními komponentami je vidět na obrázku 3.

Tento modul je základním prvkem celého Tahoe-II kitu, protože právě on sám je schopen zpracovávat zmiňovaný managed kód. Kromě tohoto modulu je deska osazena dalšími komponentami, které dále rozšiřují jeho funkcionalitu. Blokové schéma celého kitu je vyobrazeno na obrázku 4. Některé komponenty zůstali shodné s původním Tahoe kitem, jiné jsou zcela nové, umožnující tak větší variabilitu při návrhu aplikací pro embedded systémy.



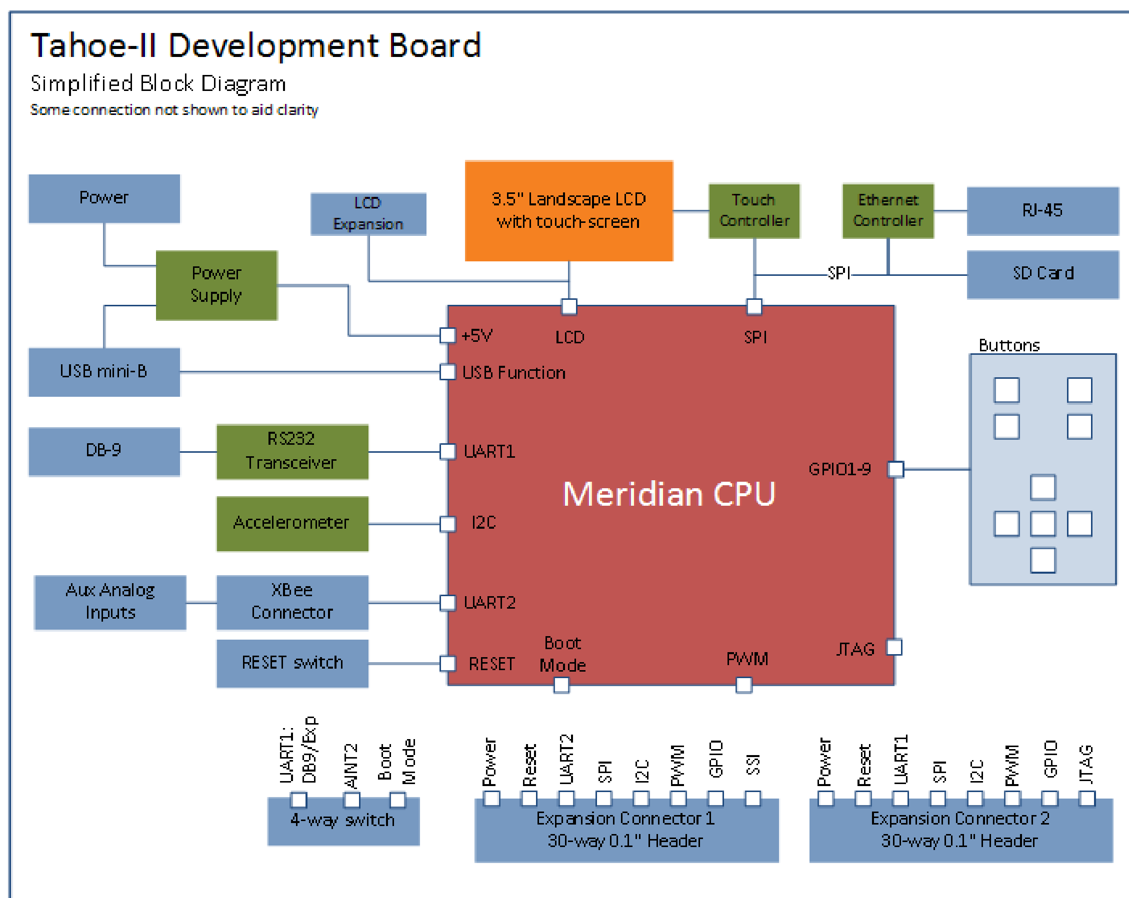
Obrázek 3: Blokové schéma Meridian modulu - převzato z [11]

Tahoe-II Development Kit:

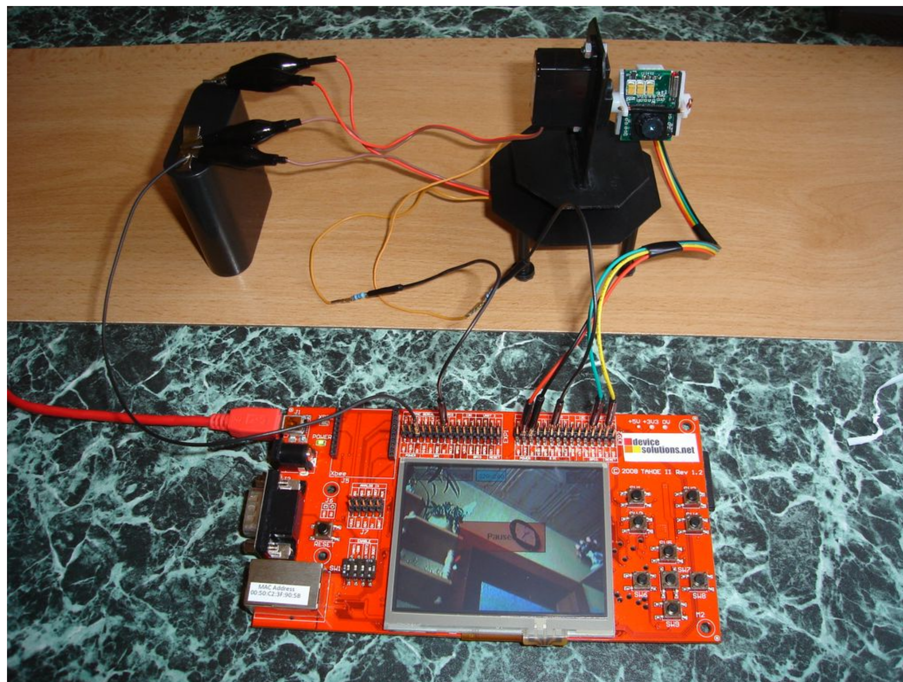
- Meridian modul (Freescale ARM920 @ 100MHz, 4Mbytes Flash, 8Mbytes SDRAM),
- 3.5" QVGA dotykový LCD displej (320x240),
- 9 uživatelských tlačítek,
- senzor teploty, 2x ADC kanály,
- čtečka SD paměťových karet,
- accelerometr - detekce naklonění a volného pádu,
- sériový port RS232 (DB9 konektor),
- Ethernet modul,
- miniUSB konektor,
- rozhraní pro připojení XBee modulu pro bezdrátovou komunikaci,
- PWM výstup,
- piny pro vyvedení GPIO, UART, I2C, SPI signálů.

2.2 Ukázková aplikace

Tato aplikace má demonstrovat některé nové funkce, které byly přidány do .NET Micro Frameworku v4.0. Přesněji se jedná o práci s dotykovým LCD, čtečkou SD karet, senzorem teploty a accelerometrem. Aplikace je realizována na výše zmiňovaném vývojovém



Obrázek 4: Blokové schéma Tahoe-II Development Kitu - převzato z [23]



Obrázek 5: Kit Tahoe-II s konstrukcí obsahující servomotorky a minikameru

kitu Tahoe-II, jež je více než vhodný pro demonstraci těchto funkcí. Ke kitu je dále připojena konstrukce vlastní výroby, obsahující dva servomotorky a minikameru, vše je vidět na obrázku 5. V následujících podkapitolách se nachází podrobnější popis hlavních tříd aplikace a funkcí v nich obsažených.

2.2.1 Třída MainApp

MainApp dědí z třídy Application a je hlavní třída celé aplikace. Obsahuje spustitelnou metodu Main (výpis 1), v níž jsou provedeny veškeré nutné počáteční nastavení, jako je inicializace dotykového displeje, gpio tlačítek, výběr hlavního zobrazovaného okna aplikace a v neposlední řadě její spuštění.

```
public static void Main()
{
    //vytvoreni instance aplikace, inicializace LCD a tlacitek
    MainApp myApp = new MainApp();
    Touch. Initialize (myApp);
    GPIO gpio = new GPIO(null);

    //nastaveni aktivniho okna aplikace a jeji spusteni
    MainApp.Current.MainWindow = new TempWindow();
    Buttons.Focus(MainApp.Current.MainWindow);
    myApp.Run();
```

```
}
```

Výpis 1: Spustitelná metoda Main

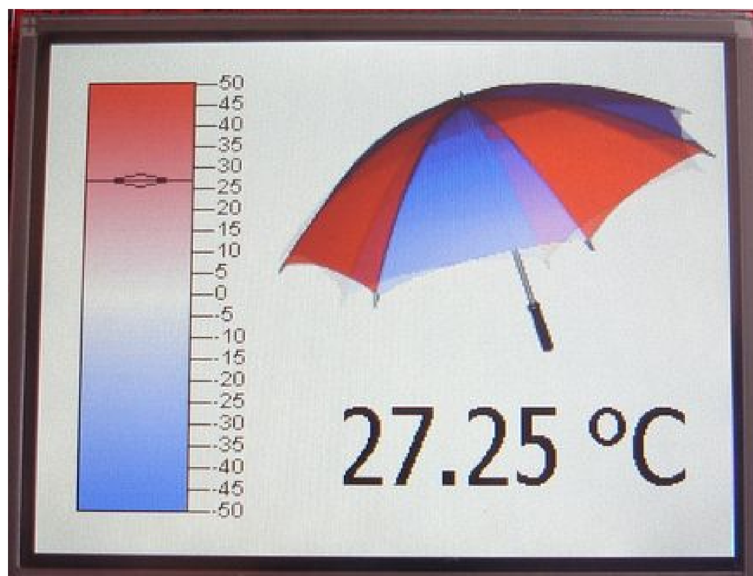
Třída MainApp dále obsahuje metodu SwitchWindow, která slouží k přepínání zobrazovaných oken aplikace. Dalo by se říci, že aplikace je rozdělena na tři dílčí aplikace (okna), z nichž každá demonstruje některé dříve zmíněné funkce. Tyto tři dílčí aplikace jsou reprezentovány třídami TempWindow, ImageWindow a CameraWindow, dědícími z třídy Window. Právě metoda SwitchWindow je zde použita pro přepínání mezi těmito dílčími aplikacemi.

Funkce tlačítek:

1. TempWindow
   - SW1 - další aplikace (ImageWindow)
2. ImageWindow
   - SW1 - další aplikace (CameraWindow)
   - SW6 - předchozí obrázek
   - SW7 - zapnout/vypnout slideshow
   - SW8 - následující obrázek
3. CameraWindow
   - SW1 - další aplikace (TempWindow)
   - SW3 - zapnutí/vypnutí pauzy
   - SW5 - pohyb kamery nahoru
   - SW6 - pohyb kamery doleva
   - SW7 - pohyb kamery na střed (centrování)
   - SW8 - pohyb kamery doprava
   - SW9 - pohyb kamery dolů

2.2.2 Třída TempWindow

Třída realizující jednoduchý teploměr. Vzhledem k objektové reprezentaci hardwaru je přístup k teplotnímu čidlu velmi snadný. Aktuální teplotu lze získat pouhým zavoláním metody ReadTemperature, nacházející se ve třídě TahoeII.Tsc2046. Třída obsahuje časovač, který v pravidelných intervalech volá právě tuto metodu a zobrazuje na displeji naměřenou teplotu. Vzhledem k možným odchylkám při měření, je zobrazovaná teplota počítána jako aritmetický průměr deseti rychle po sobě odečtených hodnot. Zobrazované okno aplikace teploměru je vidět na obrázku 6.



Obrázek 6: Obrazovka aplikace TempWindow

2.2.3 Třída ImageWindow

Tato třída realizuje jakýsi prohlížeč obrázků. Jinak řečeno, aplikace zobrazuje na displeji obrázky, načtené z vložené SD paměťové karty. Obrázky se dají přepínat po jednom vpřed či vzad, nebo je možné také spustit automatické přepínání obrázků tzv. slideshow. Tyto akce lze provést buď pomocí gpio tlačítek na kitu, nebo skrze dotykový displej, na němž jsou tyto ovládací tlačítka také vykresleny (obrázek 7).

Pro práci s paměťovou kartou slouží třída RemovableMedia, nacházející se v namespace Microsoft.SPOT.IO. Zde se také registrují události pro vložení nebo vyjmutí karty (výpis 2). Je třeba mít na paměti, že paměťovou kartu je nutno mít správně naformátovanou, jinak nemusí vůbec fungovat. K připojené paměťové kartě pak lze přistoupit jako k obyčejnému disku, přes standardní adresářovou strukturu.

```
RemovableMedia.Insert += new InsertEventHandler(MemoryCardAction);
RemovableMedia.Eject += new EjectEventHandler(MemoryCardAction);
```

Výpis 2: Události vložení/vyjmutí paměťové karty

Práci s dotykovým displejem má na starost namespace Microsof.SPOT.Touch. Pokud chceme využívat funkce dotykového displeje, je nutno u daného okna zaregistrovat handlery těchto událostí. Jedná se o metody TouchDown, TouchUp, TouchMove. V těchto metodách pak již lze zavoláním GetPosition, zjistit souřadnici bodu dotyku. Namespace Microsof.SPOT.Touch navíc také obsahuje funkce pro práci s gesty, což například umožňuje detekci písmen psaných stylusem ručně na displej.

Obrázek 7: Obrazovka aplikace ImageWindow

2.2.4 Třída CameraWindow

Třetí a poslední z těchto dílčích aplikací je i nejsložitější. Účelem této aplikace je demonstrovat práci s accelerometrem a možnosti připojení dalších externích zařízení ke kitu. Jak již bylo zmíněno, ke kitu je připojena konstrukce obsahující dva servomotorky a minikameru. Tato konstrukce je vytvořena tím způsobem, že jeden servomotorek je schopen otáčet minikamerkou v horizontálním a druhý naopak ve vertikálním směru. Díky integrovanému accelerometru je kit schopen měřit naklonění v osách X a Y, čehož se využívá právě u řízení servomotorků.

Aplikace tedy funguje tak, že obrázky snímané minikamerou jsou posílány do kitu a zobrazovány na displeji, přičemž natáčení minikamery je určeno nakláněním samotného kitu. Jednoduše řečeno, nakloněním kitu doleva se kamerka otočí doleva, nakloněním k sobě se otočí vzhůru atd.

Accelerometr

Kit Tahoe-II obsahuje accelerometr s označením MMA7455[17]. Accelerometr je vnitřně připojen k Meridian modulu přes sběrnici I2C[3], skrze kterou s ním lze pracovat. Práci s accelerometrem obstarává třída AccControl, jejíž konstruktor (výpis 3) obsahuje veškerou počáteční konfiguraci. Z konstruktoru je patrné, že accelerometr je reprezentován objektem typu I2CDevice. Také je zde nastaven pracovní mód na měření a provedena počáteční kalibrace. Tato kalibrace je nutná pro zvýšení přesnosti měření a provádí se zápisem hod-

[3]Inter-Integrated Circuit, multi-masterová sériová sběrnice používaná pro připojování nízkorychlostních periferií

not jednotlivých offsetů do příslušných registrů. Dále třída AccControl obsahuje metody pro získání X a Y souřadnice naklonění a metodu pro komunikaci s accelerometrem, zpracovávající I2C Read/Write transakce.

```
public AccControl()
{
    //vytvoreni instance accelerometru
    if (acc == null)
        acc = new I2CDevice(new I2CDevice.Configuration(ACC_ADDRESS, 100));

    //pocatecni nastaveni
    acc.Execute(
        new I2CDevice.I2CTransaction[] {
            //nastaveni modu na mereni
            acc.CreateWriteTransaction(new byte[] { COMMAND, MEASURE }),
            //pocatecni offset pro X
            acc.CreateWriteTransaction(new byte[] { XOFFSET, 0x17 }),
            //pocatecni offset pro Y
            acc.CreateWriteTransaction(new byte[] { YOFFSET, 0x39 }),
            //pocatecni offset pro X
            acc.CreateWriteTransaction(new byte[] { ZOFFSET, 0x84 })
        }
        , 1000);
}
```

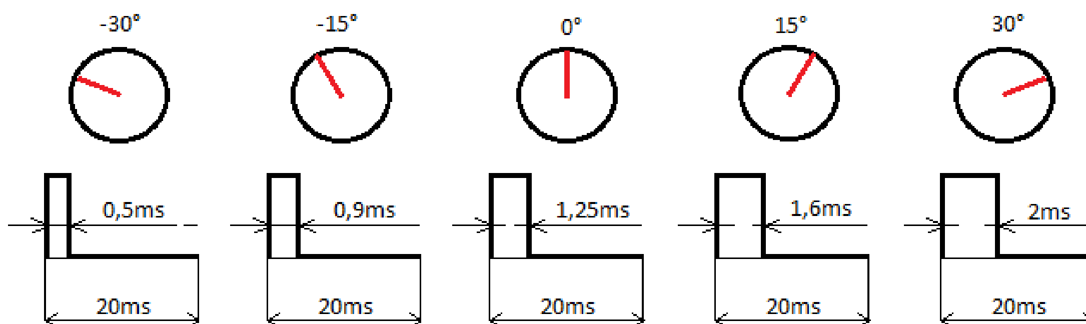
Výpis 3: Konstruktor třídy AccControl

Servomotorky

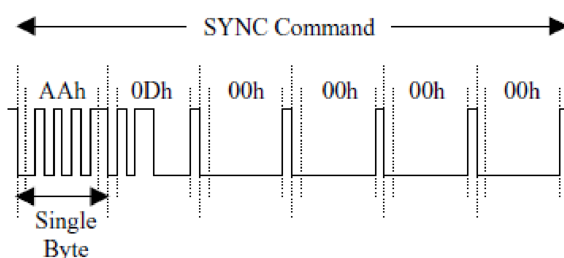
Jedná se o malé standardní servomotorky nesoucí označení Turnigy 1160A. Každý z nich obsahuje tři vodiče, přes které se připojují ke kitu (červený - napájení, hnědý - zem, žlutý - řízení). Dle výrobce má tento typ serva provozní napětí 4,8-6V a protože kit obsahuje dva piny pro vývod napájení 5V, neměl by být problém připojit oba servomotorky ke kitu. Zde jsem však narazil na problém zvýšeného odběru napájení při pohybu servomotorků, což mělo za výsledek resetnutí celého kitu. Řešením tedy nakonec bylo externí napájení servomotorků přes dodatečnou 4,5V plochou baterii.

Celou režii práce se servomotorky obstarává třída ServoControl. Otáčení probíhá v rozsahu od -30° do 30°. Servomotorky by nejspíše zvládly i větší rozsah, ale protože tento je dostačující, není třeba pokoušet štěstí a je lepší se vyvarovat jejich případnému poškození. Co se týče samotného ovládání servomotorků, to je prováděno pomocí PWM[4]. Kit generuje pro servomotorky pulzy o délce 20ms, přičemž na začátku pulzu je hodnota logická 1 (3,3V) a zbytek pulzu má logickou 0 (0V). Délka logické 1 se pohybuje v rozmezí 0,5-2ms a právě délka této logické 1 určuje pozici, na níž se má servo otočit. Vše je názorně ukázáno na obrázku 8.

[4] Pulse Width Modulation, diskrétní modulace používaná pro přenos analogového signálu pomocí dvouhodnotového signálu



Obrázek 8: Ovládání servomotorků pomocí PWM



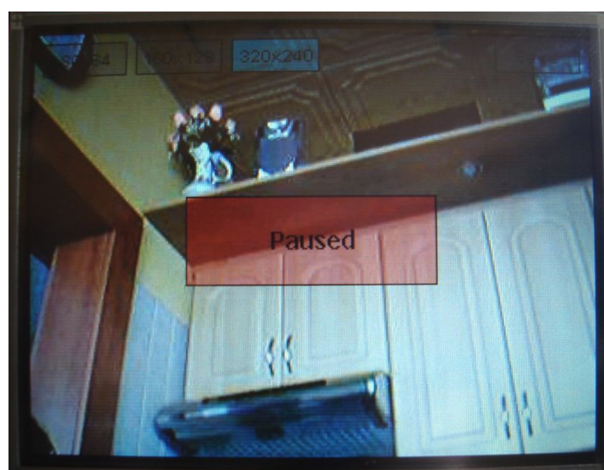
Obrázek 9: Příkaz SYNC - převzato z [3]

Pro jemnější pohyb obou servomotorků je však v samotné aplikaci celý rozsah zmíněné logické 1 (0,5-2ms) rozdělen na 10 mezních hodnot, vždy po kroku o velikosti 0,15ms. Tím je dosaženo větší přesnosti při nastavování pozice otočení a také se zabrání pohybu motorků za jejich fyzicky přípustnou hranici, v důsledku čehož by mohlo dojít k jejich nechtěnému zničení. Pozice pro natočení je vždy určena právě dle hodnoty získané z accelerometru.

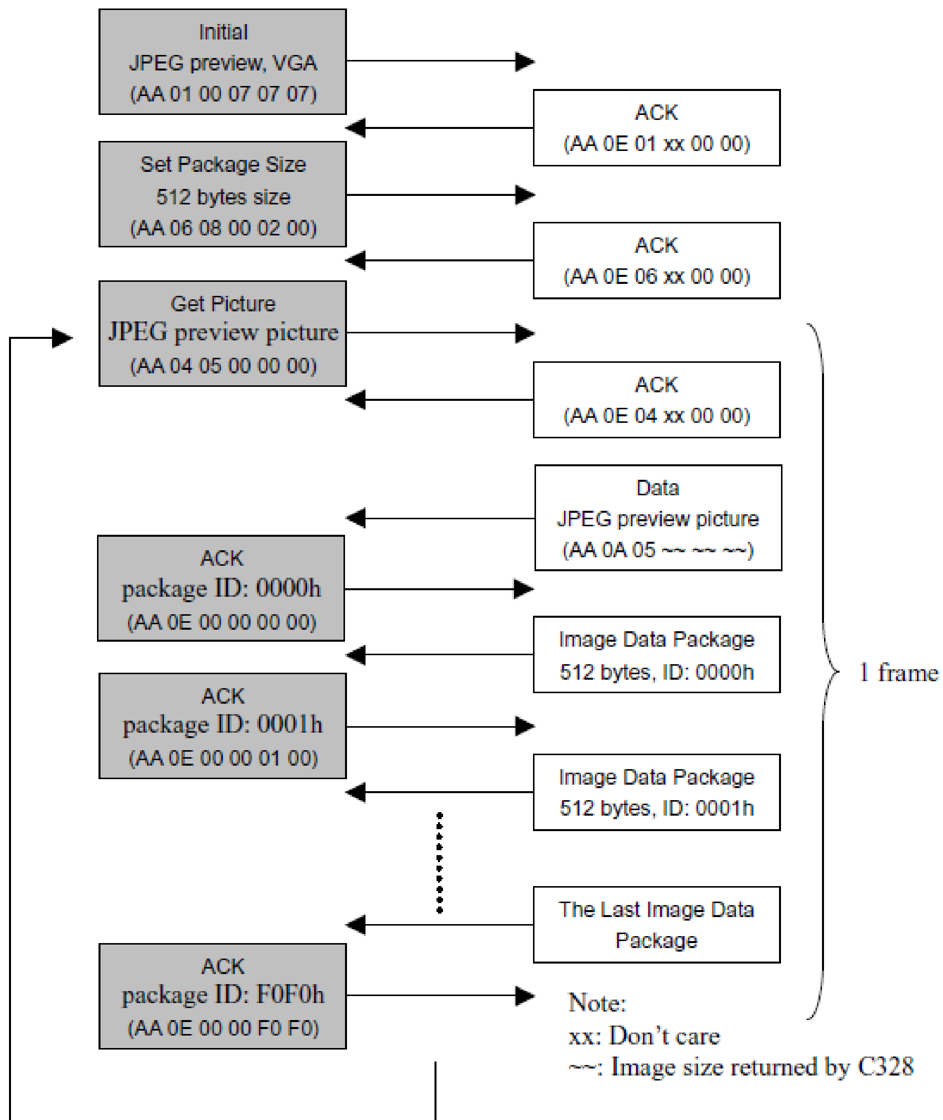
Minikamera

Poslední částí úlohy je minikamera s označením ITM-C-328. Samotná kamera je schopná provádět JPEG kompresi zachyceného obrazu, je vybavena sériovým rozhraním pro komunikaci (až 115,2Kbps) a díky napěťovým nárokům o velikosti pouze 3,3V, ji lze napájet přímo z kitu. Kamera umožňuje take získání obrázku ve formátu RAW, jak barevného tak i v několika odstínech šedi. .NET Micro Framework však není schopen takovýto obrázek zpracovat, proto se musíme spokojit s formátem JPEG. Díky vestavěnému JPEG kodeku je kamera schopna pracovat s rozlišeními 80x64, 160x128, 320x240 a 640x480. Zachycení obrazu lze provést ve dvou módech, Preview - zachycený obraz je ihned posílán sériovým rozhraním, nebo Snapshot - zachycený obraz je uložen do bufferu a odeslán až po potvrzení. Samotná aplikace využívá mód Preview pro co nejrychlejší získání obrázku z kamery.

Obrázek 10: Obrazovka aplikace CameraWindow



Obrázek 11: Pauznutá obrazovka aplikace CameraWindow



Obrázek 12: Postup získání JPEG Preview obrázku - převzato z [3]

Veškeré metody pro práci s minikamerou se nachází ve třídě CameraControl. Komunikace s kamerou probíhá prostřednictvím příkazů o velikosti 8B. Na obrázku 9 je znázorněn časovací diagram příkazu SYNC, jež se používá pro počáteční synchronizaci s kamerou. Synchronizace probíhá neustálým posíláním příkazu SYNC kameře, přičemž se čeká na potvrzení od kamery, která odpoví průměrně po 25ti obdržených příkazech SYNC. Pokud se tak nestane ani po 60ti odeslaných SYNC příkazech, kamera neodpovídá a je nutné ji resetovat odpojením od napájení a pokusit se o synchronizaci znovu. Po úspěšné sychronizaci s kamerou je nutno nastavit přenosovou rychlost, mód v kterém bude kamera pracovat (RAW/JPEG, Preview/Snapshot, rozlišení) a velikost balíčků, v nichž bude kamera po částech posílat zachycený obraz. Postup pro získání JPEG Preview obrázku i s veškerými příkazy, které se komunikace účastní, se nachází na obrázku 12. Detailnější popis veškeré komunikace s kamerou je podrobně popsán v jejím manuálu [3].

Na obrázcích 10 a 11 je vidět vzhled zobrazovaného okna celé aplikace. Obrázky získáné z minikamery jsou zobrazovány na displeji. V horní části obrazovky se nachází tlačítka pro změnu rozlišení obrázku a také tlačítko Save, které slouží pro uložení aktuálně zobrazeného obrázku na vloženou paměťovou kartu. Takto uložené obrázky si lze ihned prohlédnout v aplikaci která již byla popsána výše - třída ImageWindow. Rychlost zobrazování obrázků z kamery je při rozlišení 320x240 průměrně 1 FPS, přičemž se zmenšujícím se rozlišením tato rychlost mírně roste. Rychlost zobrazování tedy není nijak závratná, ale pro demostrační účely je vyhovující.

3 Microsoft Robotics Developer Studio

3.1 Architektura

Microsoft Robotics Developer Studio (dále jen MRDS[13]) je komplexní framework, zaměřený na oblast robotiky. Jako takový obsahuje služby a nástroje umožňující navrhovat, řídit a monitorovat aplikace vyvíjené pro roboty. Aplikace vytvořené pomocí MRDS jsou složené z vzájemně nezávislých služeb, běžících na webu nebo lokální intranetové síti. Mezi nespornou výhodu této architektury patří i možnost práce nejenom s fyzickými roboty, ale i s roboty nacházejícími se v simulačním prostředí. Díky platformě .NET, na níž je MRDS postaven, je i zde možnost programovat v pokročilých programovacích jazycích (C#) a využívat moderní vývojové nástroje (Visual Studio).

Na obrázku 13 jsou vyobrazeny klíčové komponenty MRDS. CCR je model založený na komunikaci prostřednictvím zasílání zpráv a slouží pro efektivní práci s asynchronními procesy. DSS je servisně orientovaný aplikační model, který je založený na architektuře webových služeb, tak jak jej známe z WWW. A v neposlední řadě CLR, jež umožňuje těmto dvěma klíčovým komponentám přístup k funkcím obsažených v .NET Frameworku.

3.1.1 Concurrency and Coordination Runtime (CCR)

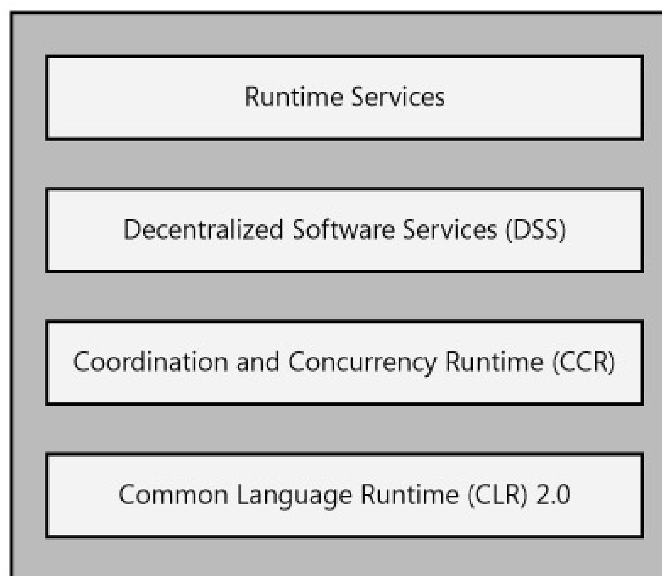
CCR je ta část MRDS, která se váže právě na práci se službami a umožňuje vytvářeným aplikacím provádět asynchronní operace. CCR je ve skutečnosti .NET knihovna, která se snaží o co největší zjednodušení při návrhu aplikací, jež vyžadují asynchronní zpracování. Umožňuje koordinaci zasílaných zpráv bez toho, abychom sami museli složitě řešit problémy typu kritické sekce, uzamknutí, semafory apod.

Asynchronní zpracování je důležité právě v oblasti robotiky, protože bez něj by se veškeré instrukce zpracovávané robotem vykonávaly sériově. To znamená, že pokud bychom robotovi zaslali instrukci s příkazem, aby dojel na určité místo a následně instrukci přikazující např. otočení robotické paže, příkaz na otočení paže by se vykonal až po dokončení všech předchozích instrukcí, tedy až po dojetí robota na určenou souřadnici.

CCR tedy řeší tři základní problémy spojené se servisně orientovanou architekturou. Jedná se tedy o samotné asynchronní zpracování, souběžnost procesů a koordinaci či obsluhu selhání. CCR API lze rozdělit do tří základních kategorií, jež jsou níže dále rozepsány.

CCR API:

- Ports, PortSets,
- Coordination,
- Task Scheduling.



Obrázek 13: Architektura Robotics Developer Studia - převzato z [21]

Ports, PortSets

Port a PortSet jsou generické třídy, realizující jednoduchou frontu typu FIFO. Prvek v této frontě může být jakýkoliv validní CLR typ (primitivní typy .NET Frameworku), nebo vlastní uživatelský kód označený pod názvem Receiver (přijímač). Jediný rozdíl mezi třidami Port a PortSet je ten, že třída Port obsahuje pouze jeden generický argument, kdežto třída PortSet jich může obsahovat více. Což je jistě praktičtější, pokud nechceme vytvářet více samostatných instancí třídy Port, pro příjem zpráv několika různých generických typů (výpis 4).

Pro zápis na port (vložení prvku do fronty) se používá metoda Post (výpis 4). Zápis hodnoty na port je asynchronní a neblokovací operace. To znamená, že metoda Post vrací řízení zpět programu jak nejdříve je to možné, jinými slovy nečeká na zpracování zprávy zapsané na port.

```
//vytvoreni instance tridy Port
Port<int> portInt = new Port<int>();

//zapsani Int cisla na port
portInt.Post(55);

//vytvoreni instance tridy PortSet
var portSet = new PortSet<int, string, double>();¨

//zapsani nekolika typu zprav na portSet
genericPortSet.Post(55);
genericPortSet.Post("zprava");
```

```
genericPortSet.Post(3.14);
```

Výpis 4: Vytvoření instance třídy Port a PortSet se zápisem na porty metodou Post

Samotný port může pracovat ve dvou módech, pasivním nebo aktivním. Při zápisu zprávy na pasivní port, se následně neprovede žádná operace pro zpracování zaslané zprávy. Zprávu z takovéhoto pasivního portu lze získat pomocí metody Test (výpis 5). Pokud se na portu nějaká zpráva nachází, je vrácena, pokud je port prázdný, metoda Test vrátí hodnotu False.

```
//zprava
int item;

//precteni portu
var hasItem = portInt.Test(out item);
if (hasItem)
    Console.WriteLine("Zprava na portu :" + item);
else Console.WriteLine("Port je prazdny");
```

Výpis 5: Přečtení zprávy z pasivního portu metodou Test

V případě že port pracuje v aktivním režimu, je pro něj registrován tzv. Arbiter. Ten slouží k plánování úkolů pro tento port. Pokud tedy na port přijde nějaká zpráva, paralelně se začně zpracovávat aktivovaným Arbiterem. Ve výpisu 6 je ukázka příjmu zprávy z aktivního portu pomocí Arbiteru typu Receive. Třída Arbiter je podrobněji popsána v následující části.

```
//prectení portu pomocí Arbiter.Receive
Arbiter.Activate(
    taskQueue,
    portInt.Receive(delegate (int item) //metoda ktera se ma provest
    {
        //paralelne provadeny kod
        Console.WriteLine("Zprava na portu :" + item);
    }
));
```

Výpis 6: Přečtení zprávy z aktivního portu pomocí třídy Arbiter.Receive

Coordination

U paralelně běžících asynchronních procesů vyvstává problém s jejich vzájemnou koordinací. Některé procesy mohou skončit úspěchěm, ale jiné mohou naopak selhat. U asychronního programování je tedy nutné počítat se všemi možnými způsoby skončení požadované operace. Právě do této kategorie spadá zmiňovaná třída Arbiter, jež se stará o koordinaci zpráv přicházejících na porty. Jedná se o statickou třídu implementující

návrhový vzor Factory (Továrna[19]). Dále je uveden popis základních typů Arbiterů, které je možné v MRDS použít. Uvedený popis je velmi stručný a pro podrobnější informace je nutno nastudovat příslušný CCR manuál.

- Arbiter.FromTask - přijímač vytváří instanci třídy Task.
- Arbiter.Choice - přijímač, který vykoná pouze jednu větev z daného rozhodovacího bloku (např. úspěch/selhání).
- Arbiter.Receive - přijímač, který po příchodu zprávy na port, vykoná např. kód přiřazeného delegáta.
- Arbiter.Interleave - řeší problém kritické sekce a vzájemného zablokování.
- Arbiter.JoinedReceive a Arbiter.MultipleItemReceive - příjem zpráv z více portů najednou a jejich zpracování (nezáleží na pořadí přijímaných zpráv).

Task Sheduling

Třetí část CCR určuje, jak jsou úlohy, generované při příchodu zprávy na port s registrovaným Arbiterem, vyvažovány mezi systémové prostředky daného stroje. Jinými slovy, tato kategorie má na starost plánování úloh a řadí se do ní třídy Task, DispatcherQueue a Dispatcher.

Třída Task implementuje rozhraní ITask, které určuje, že daná třída může být použita pro plánování (zpracována plánovačem). Třída Arbiter také implementuje rozhraní ITask, aby mohlo docházet k jejich správnémů zpracování. DispatcherQueue realizuje FIFO frontu pro vytvářené úlohy. Tato fronta může využívat buď CLR „thread-pool" pro samotné plánování, nebo instanci CCR třídy Dispatcher, která se stará o vlákna operačního systému a slouží k vyvažování zátěže. K tomu dochází pomocí úloh, jež Dispatcher vybírá z jedné nebo i více front typu DispatcherQueue.

```
//vytvoreni instance tridy Dispatcher
var dispatcher = new Dispatcher(
    0,                    //pouzije jedno vlakno na jedno CPU
    "MujDispatcher"  //nazev Dispatcheru
);

//vytvoreni instance fronty DispatcherQueue
var taskQueue = new DispatcherQueue(
    "MojeFronta",     // nazev fronty
    dispatcher              // instance Dispatcheru
);

//vytvoreni portu pro prijem Int a frontou pro planovani uloh
var portInt = new Port<int>();
Arbiter.Activate(taskQueue,
```

```
        Arbiter . Receive(
            true,
            portInt ,
            item => Console.WriteLine(item)
        )
    );

    // zaslani zpravy na port
    portInt . Post(55);
```

Výpis 7: Použití tříd Dispatcher a DispatcherQueue pro plánování úloh

Ve výpisu 7 je ukázka vytvoření portu pro příjem zprávy typu Int. Po zaslání zprávy na port je vygenerována úloha pro jeho obsloužení, která je přidělena prostředkům operačního systému, skrze námi vytvořenou instanci třídy Dispatcher.

Knihovna CCR dále obsahuje funkce pro práci s iterátory, díky nímž je schopen programátor napsat vícenásobnou asynchronní logiku do jedné iterační metody. To zvyšuje čitelnost kódu a zároveň uchovává asynchronní chování, protože při volání Yield nedochází k zablokování žádného vlákna operačního systému. Pro ošetření selhání, či správu výjimek, zůstala v CCR zachována standartní konstrukce bloků Try, Catch a Finaly.

3.1.2 Decentralized Software Services (DSS)

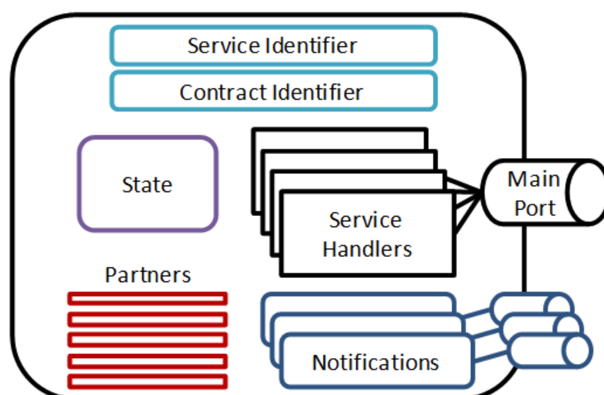
Práce s webovými službami je v MRDS realizována právě díky aplikačnímu modelu DSS. Tento model vychází ze standartní tzv. „Web-based“ architektury, označované také zkratkou REST. DSS je postavena nad již zmíněnou knihovnou CCR a umožňuje tedy vytvářet servisně orientované aplikace s využitím známé webové architektury. DSS tedy umožňuje vytvořit aplikaci, jako kolekci vzájemně komunikujících webových služeb, u nichž nezáleží na tom, jestli běží všechny na stejném stroji, nebo na několika různých místech v síti. Výsledkem takovéhoto přístupu je flexibilní a zároveň jednoduchý návrh servisně orientovaných aplikací.

Komunikace mezi webovými službami probíhá prostřednictvím zasílání zpráv. Základem pro realizaci této komunikace jsou protokoly DSSP[5] a HTTP. DSSP je jednoduchý protokol založený na SOAP, definující stavově orientované operace pro práci se strukturovanými daty a obsahující funkce např. pro notifikaci událostí u webových služeb. DSSP lze chápat jako rozšíření aplikačního modelu zprostředkovaného protokolem HTTP a používá se jako dodatek k stávající HTTP infrastruktuře.

Webové služby

Webová služba je základním stavebním prvkem DSS. Webové služby nemusí nutně reprezentovat pouze hardwarové komponenty, jako jsou různé senzory či snímače, ale i např. softwarové komponenty jako UI[5] nebo vlastní části programového kódu. Služby jsou prováděny v rámci DSS uzlů. DSS uzel je prostředí (kontext) pro podporu vytváření

[5] User Interface, uživatelské rozhraní

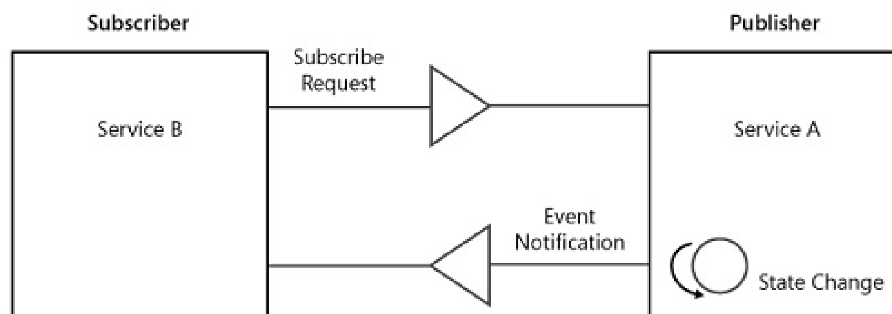


Obrázek 14: Schéma webové služby - převzato z [14]

a řízení webových služeb. Každá spuštěná služba se nachází v DSS uzlu dokud není smazána, nebo není zastaven celý uzel. Blokové schéma samotné webové služby je znázorněno na obrázku 14.

Komponenty webové služby:

- Service Identifier - dynamicky přiřazené, jednoznačné URI, umožňující její identifikaci. Také umožňuje přístup ke službě skrze webový prohlížeč. Označuje pouze název služby a neobsahuje žádnou informaci o jejím stavu či chování.
- Contract Identifier - obsahuje zkomprimovaný popis implementace dané webové služby. Poskytuje informaci o jejím chování a používá se při navazování DSS proxy spojení.
- Service State - obsahuje informaci o stavu služby. Lze také říct, že je to dokument obsahující popis aktuálního stavu služby (rychlost otáčení motorku, spotřeba paliva).
- Service Partners - seznam partnerů webové služby. Poskytuje informaci o tom, na kterých dalších službách tato aktuálně spuštěná služba závisí, aby bylo dosaženo správné funkčnosti při vzájemné komunikaci více služeb. Obsahuje také několik druhů partnerství.
- Main Port - hlavní port, instance CCR třídy PortSet, na který přicházejí zprávy od ostatních služeb. Bývá označován také jako port operací a je identifikován atributem ServicePort.
- Service Handlers - obsluha hlavního portu, která se stará o příchozí zprávy. Lze registrovat obsluhu např. pro operace Get, Replace, HttpGet a HttpPost.
- Event Notifications - události které generuje služba jako výsledek změny svého vlastního stavu. Způsob, jak dát jiné partnerské službě vědět o změně svého stavu.



Obrázek 15: Schéma principu předplácení webových služeb - převzato z [14]

Protože informace o stavu služby jsou realizovány XML dokumentem, MRDS obsahuje možnost uložení tohoto dokumentu. Takto uložený dokument může být později zase nahrán, čímž dojde k načtení předchozího uloženého stavu služby. Tohoto se hojně využívá např. v simulacích pro uložení právě zobrazované scény.

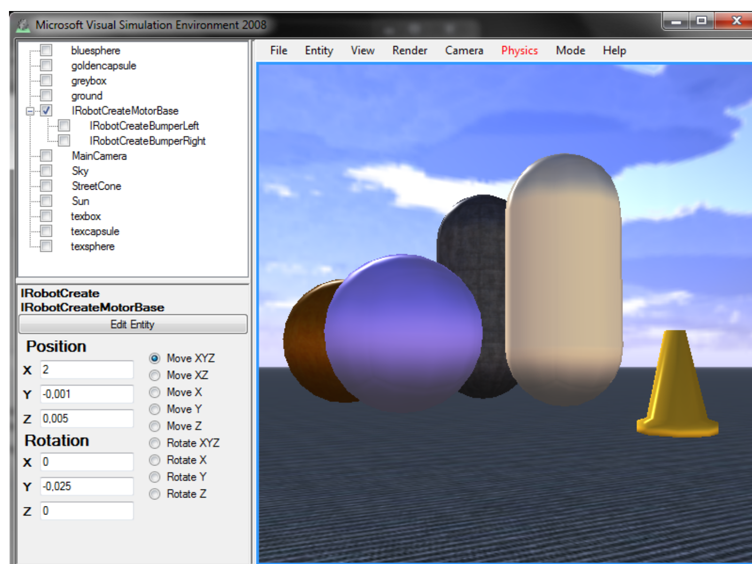
S dříve zmíněnou notifikací událostí úzce souvisí princip zvaný Subscription (předplácení). Princip je ukázán na obrázku 15. Zjednodušeně řečeno, pokud jistá služba chce dostávat informace o změně stavu jiné služby, zaregistruje se u ní jako předplatitel. Pokud pak tato služba změní svůj stav, oznámí to všem svým registrovaným předplatitelům. Služba nabízející možnost notifikace událostí se nazývá Publisher a služba registrovaná k odebírání těchto událostí je Subscriber. Pro práci s operacemi, sloužícími k předplácení služeb, je zde definován SubscriptionManager. Ten také obstarává samotné generování událostí.

Samotnou webovou službu lze spustit buď ručně z příkazové řádky pomocí aplikace DssHost.exe, nebo vytvořením nového projektu webové služby ve Visual Studiu. Ve Visual Studiu je také obsažen průvodce, v němž je možné hned na začátku nastavit základní vlastnosti nově vytvářené služby, jako je její název, jestli má obsahovat SubscriptionManager, nebo seznam partnerů a contractů, s nimiž bude tato služba spolupracovat. Následně je vytvořen defaultní template projektu webové služby a již nic nebrání tomu, začít psát vlastní kód.

MRDS obsahuje také aplikaci s názvem Visual Programming Language (VPL). VPL je vizuální programovací jazyk, který umožňuje jednoduše vytvořit výsledný program skládáním tzv. řídících bloků dohromady. Tyto bloky reprezentují další služby (např. služba ovládající pohyb robota) nebo části kódu, s nimiž má aplikace pracovat. VPL tak umožňuje vytvářet servisně orientované aplikace i lidem, kteří neumí programovat v některém standardním programovacím jazyce, jako je C# nebo Visual Basic.

3.2 Simulace a simulační prostředí

Co je to vůbec simulace? Simulace je pojem, označující matematický počítačový model, použitý k reprezentaci jednoho nebo více fyzických objektů. Do takovéhoto modelu jsou



Obrázek 16: Okno simulace zobrazené ve VSE

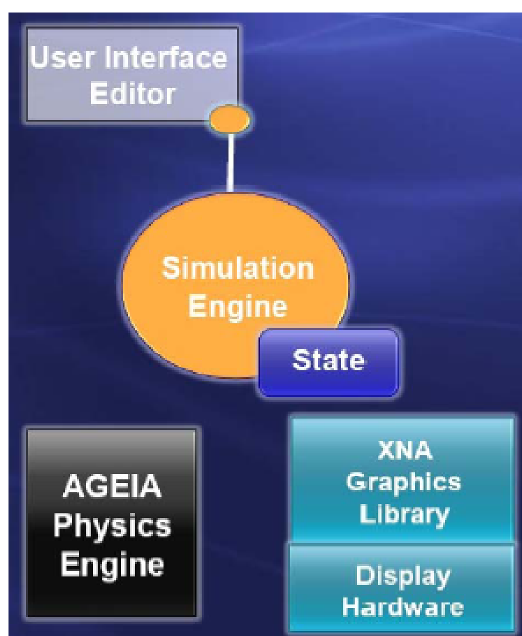
vložena data a jeho výsledkem může být například vyrenderovaný 3D snímek. V dnešní době se s těmito simulacemi setkáváme téměř neustále a to např. ve formě počítačových her.

Díky servisně orientované architektuře, umožňuje MRDS programátorovi poměrně snadno vytvářet simulace, jež mu mají pomoci při návrhu aplikací pro roboty. Použití těchto simulací má nespornou výhodu. Vše probíhá virtuálně v počítači, proto není třeba mít robota fyzicky k dispozici. To je obrovská výhoda např. při práci s drahými roboty, kdy se chceme vyvarovat jejich možnému poškození, během vytváření a následném testování dané aplikace. Nebo pokud tým programátorů pracuje na aplikaci obsluhující stejného robota, ale skutečný robot je pouze jeden. Bylo by neefektivní, kdyby každý z nich čekal, až se dostane na řadu s vypůjčkou robota, aby mohl otestovat svou část programu. Simulace umožňují i vytvoření vlastního prototypu robota a jeho otestování v simulačním prostředí, bez nutnosti jeho fyzické výroby.

Právě pro práci s těmito simulacemi obsahuje MRDS grafický editor s názvem Visual Simulation Enviroment (VSE[25]). Tento grafický editor dále urychluje práci a správu simulací, protože plní funkci tzv. grafické nádstavby samotného simulačního enginu. Jedná se o vlastní zobrazované okno simulace, umožňující správu scény simulace a všech entit, které se v ní nacházejí. Ukázku simulace ve VSE lze vidět na obrázku 16.

3.2.1 Vytvoření simulace

Jako většina komponent v MRDS, je i simulace reprezentována webovou službou. Pro vytvoření vlastní simulace je tedy nutné, nejprve si vytvořit vlastní DSS službu, v níž bude simulační služba s názvem SimulationEngine, definována jako služba partnerská (výpis 8). Toho je možné dosáhnout již přímo v průvodci, zobrazeném při vytváření nové



Obrázek 17: Komponenty simulačního prostředí

služby, nebo následným ručním přidáním kódu pro registraci nového partnera.

```
[Partner("SimulationEngine",
    Contract = engine.Contract. Identifier ,
    CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]

engine.SimulationEnginePort _simulationEnginePort = new engine.SimulationEnginePort();
```

Výpis 8: Registrace partnerské služby SimulationEngine

Celé simulační prostředí (obrázek 17) je tvořeno především dvěmi hlavními komponentami. Jedná se o simulační engine a fyzikální engine. Simulační engine se stará o servisně orientovaný přístup k objektům nacházejícím se v simulaci a je zodpovědný za renderování celé zobrazené scény. Jak již bylo zmíněno výše, je to služba, kterou je nutno přiřadit jako partnera k námi vytvářené službě.

Fyzikální engine se naopak stará o fyzikální výpočty v simulaci, jako je např. gravitace nebo kolize objektů. Tento engine plní funkci jakéhosi zjednodušeného rozhraní pro práci s fyzikou, protože odděluje uživatele od složité práce v nízkoúrovňovém fyzikálním enginu nad nímž je postaven.

3.2.2 Simulační engine

Tato kapitola se zabývá simulačním enginem MRDS. Kromě základních informací o simulačním enginu a použitém XNA Frameworku [16], se zde nachází podrobnější popis entitních typů, s nimiž lze v simulačním prostředí pracovat.

XNA Framework

Jak již bylo zmíněno, simulační engine se stará o zobrazení samotné simulace. Simulace je běžná 3D scéna, se standardním X, Y, Z souřadnicovým systémem. Simulační engine používá pro renderování výsledné 3D scény XNA Framework. Tento framework pochází také z dílny firmy Microsoft a obsahuje sadu .NET knihoven, postavených na DirectX API.

Knihovy XNA Frameworku se nachází v namespace Microsoft.XNA.Framework a obsahují funkce převážně pro práci s počítačovou grafikou, jako je například vykreslování 2D a 3D grafiky nebo stínování a osvětlování zobrazených modelů. Ve výpisu 9 je ukázka vykreslení 2D přímky do simulační scény.

```
//primka s krajnimy body (0,0,0) a (10,10,10)
VertexPositionColor [] points = new VertexPositionColor [] {
  new VertexPositionColor(new Microsoft.Xna.Framework.Vector3(0,0,0) , Color.Red),
  new VertexPositionColor(new Microsoft.Xna.Framework.Vector3(10,10,10) , Color.Red)
  };

int lineCount = 1;
int [] indices = new int [] { 0, 1 };

Device.DrawUserIndexedPrimitives<xna.Graphics.VertexPositionColor>(
   xna.Graphics.PrimitiveType.LineList,
   points,
   0,
   points.Length,
   indices,
   0,
   lineCount);
```

Výpis 9: Vykreslení 2D přímky do scény s použitím XNA Frameworku

V rámci MRDS však klasický uživatel nejspíše nebude zacházet do detailů XNA Frameworku, protože o veškeré základní funkce nutné pro vykreslení simulace se stará samotný simulační engine. Ukázku scény vykreslené simulačním enginem MRDS, obsahující zařízení kuchyně uvnitř bytu, lze vidět na obrázku 18.

Entity

Pod pojmem entita si lze představit jakýkoliv objekt, který se nachází v simulaci. Každá entita obsahuje informace o svém stavu, což můžou být například její rozměry.

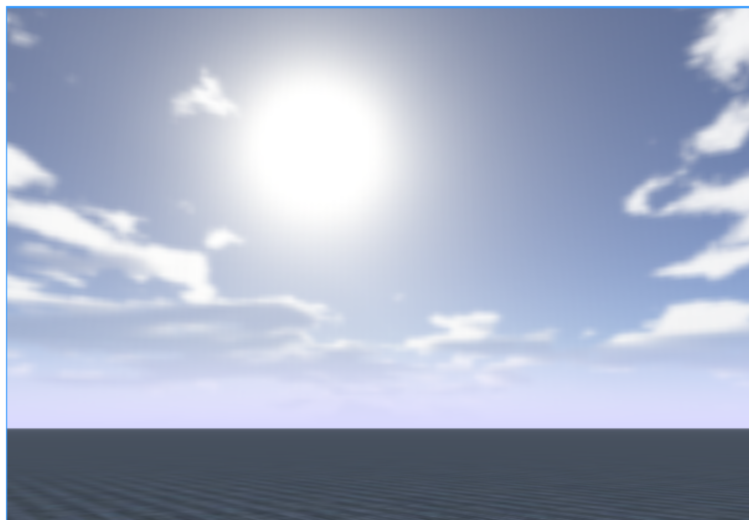
Obrázek 18: Vykreslená 3D scéna s kuchyní

Simulační engine umožňuje přístup k těmto informacím skrze grafický editor VSE nebo přímo z programového kódu. To dává uživateli možnost jednoduše měnit vlastnosti vybrané entity ve scéně. MRDS obsahuje několik základních předdefinovaných typů entit.

První kategorii lze označit jako Enviromentální entity. Do této kategorie spadají entity, které vytváří vzhled (prostředí) zobrazované scény. Ale nejedná se přímo o modely různých objektů (židle, robot apod), které lze následně do scény umístit. Pokud bychom do scény neumístily alespoň některé základní entity z této kategorie, scéna by měla podobu černého okna bez nějakého viditelného objektu. Na obrázku 19 je vidět, jak tato základní scéna vypadá.

- Enviromentální entity
  - HeightFieldEntity - nekonečně dlouhá plocha umístěna v horizontální poloze do scény, realizující jednoduchou podlahu (zem).
  - TerrainEntity - výšková plocha, vytvořená z bitmapy s odstíny šedi (odstín je v rozmezí 0-255 určující výšku terénu v daném místě). Oproti HeightFieldEntity má určenu velikost.
  - SkyDomeEntity - přidá do scény oblohu, umístěnou nekonečně daleko.
  - LightSourceEntity - zdroj světla pro nasvícení scény. Možnost vybrat ze dvou druhů světla, Omni (všesměrové - Slunce) a Directional (jednosměrové - baterka).
  - CameraView - entita realizující kameru snímající vytvářenou scénu. Defaultně snímá počátek souřadnicového systému.

Další kategorií jsou entity, které nám reprezentují již samotné objekty umisťované do scény. Ty lze navíc rozdělit do dvou dalších podkategorií a to dle jejich tvaru, nebo podle



Obrázek 19: Základní scéna obsahující zem, oblohu, zdroj světla a kameru

počtu objektů z níž jsou tvořeny. V praxi to je vcelku jednoduché. MRDS obsahuje entity reprezentující základní tvary, jako je koule (sphere), kapsule (capsule) a kvádr (box). Navíc obsahuje i další dvě speciální entity (ConvexShape a TriangleShape) pro reprezentaci složitějších objektů. K těmto dvěma entitám se vrátíme později, v kapitole zabývající se fyzikálním enginem.

- Rozdělení dle tvaru:
  - SphereShape,
  - CapsuleShape,
  - BoxShape,
  - ConvexShape,
  - TriangleShape.
- Rozdělení dle počtu objektů:
  - SingleShapeEntity,
  - MultiShapeEntity.

Je však jasné, že s těmito základními objekty (koule, kapsule a kvádr) si vystačí jen málokdo. Právě z tohoto důvodu obsahuje MRDS třídy SingleShapeEntity a MultiShapeEntity. Obě tyto třídy jsou zděděny z VisualEntity, což určuje že se bude jednat o grafické objekty vykreslované ve scéně. První zmiňovaná třída je, jak název napovídá, určena právě pro objekty jež jsou tvořeny pouze jedním ze zmiňovaných základních tvarů. Pokud tedy chceme vložit do scény např. pouze kouli, vytvoříme objekt typu SingleShapeEntity obsahující tvar SphereShape (výpis 10).

```
SingleShapeEntity m1 = new SingleShapeEntity(  //entita tvorena jednim objektem
   new SphereShape(                            //objekt tvaru koule
   new SphereShapeProperties(                  // vlastnosti koule
      5,                                       //vaha
      new Pose(),                              // relativni pozice koule
      1)),                                     //polomer koule
    new Vector3(0, 3, 0)                       //absolutni pozice ve scene
);
```

Výpis 10: Vytvoření SingleShapeEntity typu SphereShape

MultiShapeEntity naopak umožňuje vytváření složitějších objektů. Je to jakási kolekce entit základních tvarů, které jsou spojeny dohromady v jeden složitější objekt. Hlavní výhodou je, že s takovýmto objektem pak lze manipulovat jako s celkem. I v grafickém editoru VSE se pak daný objekt jeví jako jedna entita. Výpis 11 obsahuje ukázku vytvoření jednoduchého modelu činky (dvě koule spojené tyčí). Za zmíňku stojí parametr Pose u vytvářených dílčích objektů. Tento parametr obsahuje souřadnice (instance třídy Vector3), určující relativní pozici vůči ostatním dílčím objektům. Pomoci těchto souřadnic se tedy v tomto případě docílí rozmístění obou koulí na protilehlé kraje tyče.

Druhou možností vytváření složitějších tvarů, je spojování SingleShape entit do větších celků jako potomků, pomocí metody „Children.Add()“. U takto vytvořené entity pak lze přistupovat i k dílčím objektům, z nichž původně vznikla. Záleží tedy pouze na aktuální potřebě programátora, pro kterou z uvedených možností se rozhodne.

```
//pole obsahujici dilci entity
BoxShape[] boxes = new BoxShape[1];
SphereShape[] spheres = new SphereShape[2];

//vytvoreni tyce
BoxShape b = new BoxShape(
    new BoxShapeProperties(
        0, new Pose(), new Vector3(0.5f, 0.5f, 5)
    )
);
boxes[0] = b;

//vytvoreni leve koule
SphereShape s1 = new SphereShape(
    new SphereShapeProperties(
        0, new Pose(new Vector3(0, 0, -2.5f)), 1)
);
spheres[0] = s1;

//vytvoreni prave koule
SphereShape s2 = new SphereShape(
    new SphereShapeProperties(
        0, new Pose(new Vector3(0, 0, 2.5f)), 1)
);
spheres[1] = s2;
```

```
// vytvoreni vysledneho modelu cinky
MultiShapeEntity m = new MultiShapeEntity(boxes, spheres);
m.State.Name = "Cinka";

// vlozeni do sceny
SimulationEngine.GlobalInstancePort.Insert(m);
```

Výpis 11: Vytvoření MultiShapeEntity (model činky)

U každé visuální entity lze nastavit několik základních parametrů, jež ovlivňují její vzhled či chování v simulaci. Níže je uveden seznam nejpoužívanějších parametrů.

- Name - jméno entity, které musí být jednoznačné v rámci simulace.
- Shape - tvar entity, je zvolen během jejího vytváření (BoxShape, SphereShape, ...).
- Size - velikost, rozměry tvaru entity (specifické nastavení pro každý tvar).
- Mass, Density - hmotnost nebo hustota, nastavuje se pouze jeden uvedený parametr, druhý je automaticky dopočítán fyzikálním enginem.
- Pose - relativní pozice entity vzhledem k ostatním entitám ve stejné kolekci (využívá se u MultiShapeEntity).
- Position - souřadnice určující absolutní pozici entity ve scéně.
- Difuse Color - barva objektu (instance třídy Vector4).
- Material - materiál ovlivňující fyzikální vlastnosti entity.

MRDS obsahuje také předdefinované modely několika robotů, které lze v simulacích využívat. Jedná se o kompletní modely robotů Lego NXT Tribot, IRobot a Pioneer 3DX. Tito roboti dědí z třídy DifferentialDriveEntity, která umožňuje spuštění služby, ovládající pohyb robota. MRDS také obsahuje funkční model robotické paže KUKA LBR3. Modely robotů lze shlédnout na obrázku 20. Vlastní model robota si lze také „jenoduše" vytvořit pomocí MultiShapeEntity, poskládáním celkového modelu z jeho dílčích částí.

Další možností je vytvoření výsledného modelu v některém grafickém editoru (Blender, Cinema4D, ...). Takto vytvořený model pak již stačí pouze exportovat z grafického editoru jako soubor s příponou .obj (textová reprezentace grafického objektu, obsahující souřadnice vrcholů a stěn) a jeho následný import do MRDS. Právě u těchto importovaných .obj modelů se využívá entitních typů ConvexShape a TriangleShape, jež budou popsány v následujících kapitolách.

Tím však výčet druhů entit nekončí. Kromě práce s visuálními entitami umožňuje MRDS také pracovat se speciálním typem entit, jež reprezentují určitou funkcionalitu. V simulacích tedy tyto speciální entity zastupují různé sensory a měřiče, jimiž může být

Obrázek 20: Lego XNT Tribot, IRobot, Pioneer 3DX, KUKA LBR3

fyzický robot ve skutečnosti vybaven. MRDS obsahuje předdefinované entity např. pro senzor detekce vzdálenosti (LaserRangeFinder), infra-červené čidlo (IREntity) nebo sonar (SonarEntity). Díky těmto speciálním entitám je možné zcela přenést veškerou funkčnost fyzického robota do virtuálního simulačního prostředí.

3.2.3 Fyzikální engine

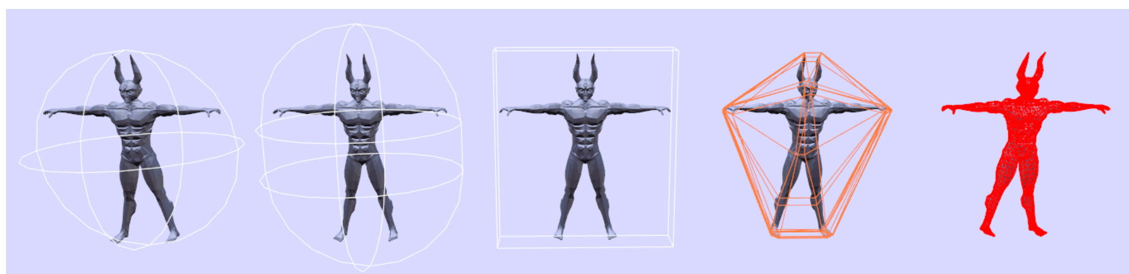
Obsahem této kapitoly je podrobnější popis fyzikálního enginu, jež MRDS využívá. Jsou zde popsány hlavní funkce enginu a jejich význam pro simulaci v MRDS. Kapitola také obsahuje popis ukázkových úloh, které tyto funkce demonstrují.

PhysX Engine

Fyzikální engine je poskytován firmou NVIDIA pod názvem NVIDIA PhysX[9] (dříve AGEIA PhysX). Tento engine umožňuje použití fyziky v simulačním prostředí MRDS. Díky němu lze v simulaci použít fyzikální vlastnosti reálného světa, jako například gravitaci, tření, sílu apod. Bez fyziky poskytované tímto enginem, by veškeré objekty pouze „visely ve vzduchu“, bez možnosti nějaké další funkčnosti.

Fyzikální model

Každá entita (objekt) vložená do simulačního prostředí je reprezentována dvěma modely. První z nich je tzv. vizuální či grafický model. Je to vizuální model entity, který je renderován dříve zmíněným XNA Frameworkem, skrze simulační engine. Jednoduše řečeno, je to vzhled entity, který vidíme v běžící simulaci (barva, odlesky, stínování, ...). Tyto modely byly popsány v předchozí kapitole, zabývající se simulačním enginem a en-



Obrázek 21: Typy fyzikálních modelů (Sphere, Capsule, Box, Convex, Triangle)

titnímy typy. Grafický model je také znám pod názvem Mesh a jak již bylo také zmíňeno, tyto Meshe lze do MRDS importovat z libovolného grafického editoru jako .obj soubor.

Druhým modelem je právě fyzikální model. Tento model je vytvořen fyzikálním enginem a reprezentuje tvar (vzhled) entity, s nímž bude pracovat právě fyzikální engine. Jinak řečeno, jedná se o model entity, který bude PhysX engine používat pro fyzikální výpočty. Jednou z hlavních funkcí tohoto modelu je např. detekce kolizí. Výčet typů fyzikálních modelů je uveden níže a odpovídá rozdělení entit dle tvaru, jež bylo zmíněno v kapitole pojednávající o simulačním enginu.

Typy fyzikálních modelů:

- SphereShape,
- CapsuleShape,
- BoxShape,
- ConvexShape,
- TriangleShape.

Právě tento „tvar" entit určoval jejich fyzikální reprezentaci. U těchto základních předdefinovaných typů entit (Sphere, Capsule, Box) v MRDS, je grafický a fyzikální model shodný. První tři uvedené typy jsou jednoduché, jedná se o fyzikální model reprezentovaný koulí, kapsulí a kvádrem. Zde se také vracíme k posledním dvěma typům, s názvy ConvexShape a TriangleShape. Tyto dva typy fyzikálních modelů naleznou uplatnění především u objektů se složitými tvary. ConvexShape je model, který fyzikální engine vypočítá jako co možná nejtěsnější obal daného objeku, ale tak aby nebyl příliš složitý. Naopak TriangleShape je nejpřesnější a tím i nejsložitější typ fyzikálního modelu. TriangleShape fyzikální model je tvořen polygony, které přesně odpovídají polygonům, z nichž je vytvořen samotný grafický model. Ten je tedy oproti ConvexShape modelu mnohem přesnější, ale také výpočetně mnohem složitější.

Prakticky lze tedy mít jeden stejný grafický model, ale použitý vícekrát s různými typy fyzikálních modelů. Toto lze vidět na obrázku 21. Jeden model postavy je použit se

všemi typy fyzikálních reprezentací. U prvních třech typů je vytvoření entity prakticky shodné s tím, co bylo ukázáno v kapitole pojednávající o entitách. Klasicky se vytvoří SingleShapeEntity s požadovaným tvarem fyzikálního modelu (SphereShape, CapsuleShape, BoxShape). Takto vytvořená entita má však nyní tvar daný zvoleným fyzikálním modelem. Pro zobrazení modelu postavy místo defaultního tvaru, tedy stačí vytvořené entitě pouze změnit její grafický model (přiřazení nového Meshe). Fyzikální modely ConvexShape a TriangleShape se tvoří lehce odlišným způsobem. A to skze vlastní definované typy entit s názvy SimplifiedConvexMeshEnvironmentEntity a TriangleMeshEnvironmentEntity. Výpis 12 ukazuje vytvoření TriangleMeshEnviromentalEntity.

```
TriangleMeshEnvironmentEntity m5 = new TriangleMeshEnvironmentEntity(
    new Vector3(9, 3, 0),                          //pozice v simulaci
    "devil.obj",                                   //graficky model
    null);                                         //tvar (null = bude vypocten)
m5.State.Name = "Devil_Triangle";                  //nazev
m5.State.MassDensity.Mass = 1;                     //vaha
SimulationEngine.GlobalInstancePort.Insert(m5);    //vlozeni do simulace
```

Výpis 12: Vytvoření TriangleMeshEnviromentalEntity s grafickým modelem devil.obj

Těchto pět uvedených typů fyzikálního modelu tedy určuje, jak se bude daná entita chovat v simulačním prostředí z pohledu fyzikálního enginu. Lze si tedy vytvořit entitu s grafickým meshem kostky a s fyzikálním modelem koule. V simulaci tedy uvidíme kostku, ale při jemném postrčení se začne kutálet jako koule.

Důležité je nezapomenout na to, že se složitějším fyzikálním modelem přichází i větší výpočetní náročnost. Při větším množství složitějších objektů ve scéně, se tedy můžeme na slabších počítačích setkat se snížením FPS (stejný princip jako u dnešních počítačových her).

Fyzikální vlastnosti a síla

Nedílnou součástí fyzikálního enginu jsou fyzikální vlastnosti. Ty lze nastavit entitě vkládané do simulace prostřednictvím materiálu, jež má danou entitu tvořit. Tyto vlastnosti ovlivňují fyzikální výpočty prováděné enginem s fyzikálním modelem této entity. Jejich seznam a následný popis je uveden níže.

Fyzikální vlastnosti:

- DynamicFriction - dynamické tření <0.0 - 1.0>,
- StaticFriction - statické tření <0.0 - n>,
- Restitution - odraz <0.0 - 1.0>.

Dynamické tření je aplikováno na entitu, pokud se dva objekty pohybují relativně vůči sobě, přičemž mezi nimi dochází ke kontaktu. Ke statickému tření dochází, pokud se dva objekty vůči sobě nepohybují ale pouze dotýkají. Pokud tedy máme v simulaci vloženou zem a na ní položíme například entitu ve tvaru kostky, tak pokud se kostka nehýbe, je na kostku aplikováno pouze statické tření. Pokud však kostku uvedeme do pohybu, přestává na ní působit statické tření a je aplikováno pouze dynamické tření.

Nastavení velikosti tření má však určitá pravidla. Oba typy tření se nastavují hodnotou z oboru reálných čísel. Velikost statického tření není omezena, ale dynamické tření je limitováno na rozsah <0.0 - 1.0> a s podmínkou, že musí být vždy menší než je velikost statického tření.

Restitution neboli odraz, ovlivňuje velikost síly, kterou bude entita při nárazu odmrštěna zpět. Jako příklad si lze představit entitu ve tvaru kostky, která bude vložena do scény např. 5m nad povrch země. Pokud bude mít tato entita nastaven nulový odraz, máme z ní doslova cihlu. Po dopadu na zem, zůstane ležet přesně na onom místě dopadu. Pokud jí však nastavíme odraz na maximum, máme z této entity skákací kuličku, která se po dopadu na zem odrazí zpět do vzduchu. Logicky je síla odrazu vždy menší než síla dopadu, proto i při nastavení maximálního odrazu není možné, aby entita „skákala donekonečna“. Sílu odrazu, stejně jako v reálném světě, také ovlivňuje nastavená hmotnost entity.

Jak již bylo zmíněno, tyto vlastnosti se nastavují jako materiál. Materiál je ve skutečnosti instance třídy MaterialProperties, která má jako vstupní parametry právě velikost tření a odrazu. Instance této třídy se následně přiřadí jako materiál vybrané entitě. Ve výpisu 13 je vidět ukázka vytvoření takovéto entity.

```
//vytvoreni entity typu BoxShape
BoxShape b = new BoxShape(
    new BoxShapeProperties(
        1,                                    //vaha
        new Pose(),                           // relativni pozice
        new Vector3(0.4f,0.075f,0.2f))        //rozmery
);

//vytvoreni materialu
b.BoxState.Material = new MaterialProperties(
    "MyMaterial",        //nazev materialu
    0.8f,                // Restitution (odraz)
    0.1f,                //DynamicFriction (dynamicke treni)
    0.2f                 // StaticFriction (staticke treni)
);

//cervena barva
b.State.DiffuseColor = new Vector3(1, 0, 0, 1);
//nazev entity
b.State.Name = "CervenaKostka";
```

Výpis 13: BoxShape entita s materiálem a barvou

Nyní víme, jak vytvořit entitu z materiálu s požadovanými vlastnostmi. Stačí tedy pouze vyzkoušet, jestli tyto vlastnosti opravdu má. Jak bylo zmíněno výše, můžeme do simulace vložit entitu a rozpohybovat ji. Zde však přichází problém, jak rozpohybovat entitu v simulačním prostředí. Poměrně jednoduše lze do simulace vložit model některého z připravených robotů a pomocí otevřeného ovládacího formuláře ho řídit. Jak je však tento pohyb robota řešen na „základní úrovni"?

Jako každý fyzikální engine, i NVIDIA PhysX engine obsahuje práci se silami. Síla je vektorová fyzikální veličina a je příčinou změny pohybového stavu tělesa. Umožňuje tedy uvést těleso do pohybu a také měnit jeho směr či rychlost. V MRDS je možnost použít dva druhy sil.

Druhy síly:

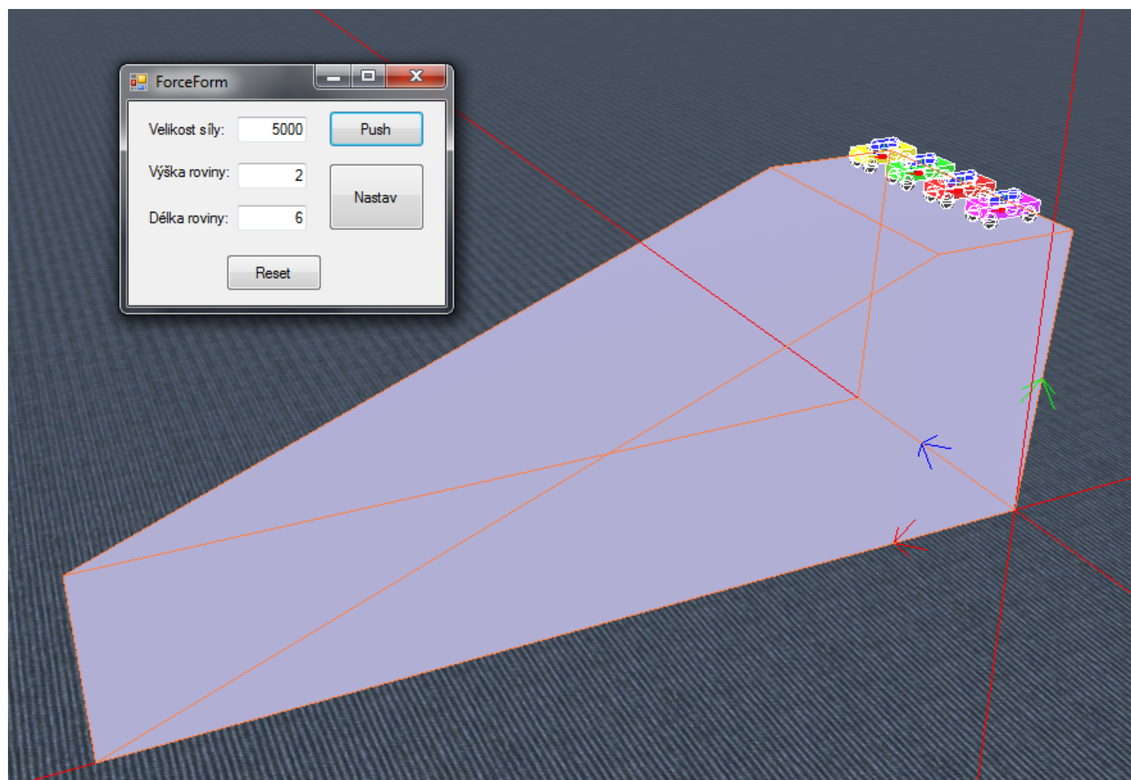
- Force - normální síla (pohyb ve směru),
- Torque - kroutivá síla (rotace kolem osy).

Force je standartní síla, tak jak ji známe. Jedná se tedy o sílu, umožňující uvedení tělesa do pohybu, ve směru určeném vektorem. Velikost vektoru udává velikost působení síly v daném směru. MRDS umožňuje aplikovat tuto sílu defaultně na střed tělesa (metoda ApplyForce), nebo na jeho libovolný bod určený souřadnicí (metoda ApplyForceAtPosition). Torque je naopak kroutivá síla. Ta vyjadřuje působení síly na bod, vzdálený od osy otáčení. Tato síla tedy slouží k rotaci objektu (metoda ApplyTorque) a její velikost je opět dána vektorem. Oba druhy sil lze navíc použít vzhledem k lokálnímu nebo globálnímu souřadnicovému systému. Lokální je vztažen k souřadnicím tělesa (entity) a globální vzhledem k osám simulační scény.

V MRDS působí tyto síly na fyzikální model entity. Je zde však problém, jak se k tomuto fyzikálnímu modelu vybrané entity dostat. Pro ujasnění, simulace je samostatně běžící služba, kterou pouze v námi nově vytvořeném projektu označíme za partnera (partnerskou službu). Když tedy vytvoříme entitu a vložíme jí do simulace, tak předáváme správu nad touto entitou simulační službě, tedy samotné simulaci. Nyní se však potřebujeme dostat zpátky k samotné entitě, přesněji k jejímu fyzikálnímu modelu, abychom na něj mohli použít sílu.

Toto je možné provést dvěma způsoby. První je skrze komunikaci mezi naší službou a simulační službou. Obsahuje vytvoření nového typu zprávy a veškeré obsluhy s tímto spojené. Principiálně by to fungovalo tak, že bychom zaslali simulační službě zprávu, pomoci které by jsme ji řekli, aby na námi vybranou entitu aplikovala sílu o dané velikosti. Tento přístup je však vcelku složitý, protože zahrnuje množství režie spojené s asynchronní komunikací (viz. kapitola pojednávající o CCR).

Snadnější řešení, je aplikace síly pomocí vnitřního stavu entity. Tím je myšleno to, že si vytvoříme vlastní třídu dědící z VisualEntity, která bude reprezentovat danou entitu. V ní si pak můžeme jednoduše napsat vlastní metodu, která bude obsahovat aplikaci síly na aktuální instanci této třídy (this.PhysicsEntity.ApplyForce). Takto můžeme z vnějšku simulace, zavolat funkci entity do ní vložené, která na tuto entitu sílu aplikuje.

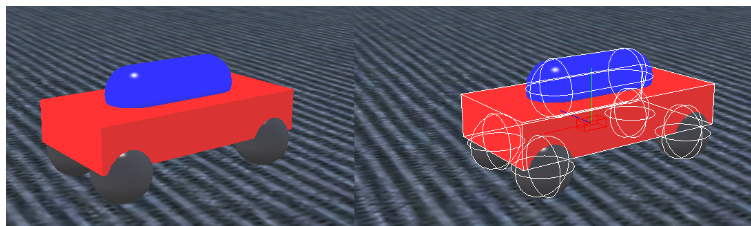


Obrázek 22: Nakloněná rovina s autíčky (zobrazen také fyzikální model)

Ukázka použití tření a síly

Ukázková úloha na obrázku 22 demonstruje použítí síly a materiálů s různými hodnotami tření. Do simulace je vložena nakloněná rovina a na její vrchol jsou umístěny čtyři modely autíček. Po spuštění simulace se mimo jiné zobrazí i ovládací formulář, který umožňuje nastavit výšku a délku nakloněné roviny a také velikost síly, kterou mají být autíčka „postrčeny" dolů.

Model nakloněné roviny byl vytvořen v grafickém editoru Blender[10] a následně exportován jako wavefront objekt, do souboru s příponou .obj. Jak již bylo zmíněno, jedná se o textový soubor, obsahující definici vrcholů (vertexes) a stěn (faces) daného objektu. Tento objekt je následně vložen do simulace jako nová entita typu SimplifiedConvexMeshEnvironmentEntity. Fyzikální engine se automaticky postará o vytvoření odpovídajícího fyzikálního modelu. Zobrazený formulář umožňuje měnit rozměry entity, což se děje skrze přístup do souboru .obj a změnou velikosti příslušných vrcholů tvořících objekt. Důležité je zmínit, že MRDS si během importu převede každý .obj soubor na soubor s příponou .bos, s nímž dále pracuje. Proto je nutné po změně .obj souboru, odebrat ze scény stávající model nakloněné roviny a na jeho místo vložit zcela nový model, vytvořený z tohoto upraveného .obj souboru. Bez tohoto nahrazení modelu, by simulace stále používala „starý" .bos soubor.



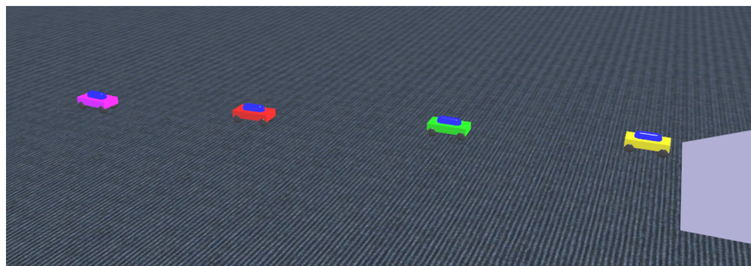
Obrázek 23: Grafický a fyzikální model autíčka

Na vrcholu nakloněné roviny se nachází čtyři autíčka. Model autíčka (obrázek 23) je realizován vlastní třídou MyEntity, dědící z třídy VisualEntity. Třída obsahuje vytvoření modelu autíčka jako jedinou entitu, která je složena z jednoho kvádru (karoserie), jedné kapsule (střecha) a čtyřmi kuličkami (kola). Fyzikální engine se opět postará o vytvořená fyzické reprezentace modelu. Každé této dílčí entitě je přiřazen materiál a barva. Protože se však autíčka pohybují „na kolech“, zajímají nás především fyzikální vlastnosti jejich materiálu. Pro demostrační účely je každé autíčko, přesněji tedy pouze jeho kola, tvořeny materiálem s různými fyzikálními vlastnostmi.

Použité materiály:

1. Fialové autíčko - žádné tření
   - StaticFriction = 0f
   - DynamicFriction = 0f
2. Červené autíčko - malé tření
   - StaticFriction = 0.1f
   - DynamicFriction = 0.1f
3. Zelené autíčko - střední tření
   - StaticFriction = 0.5f
   - DynamicFriction = 0.5f
4. Žluté autíčko - velké tření
   - StaticFriction = 1f
   - DynamicFriction = 1f

Použití síly na model autíčka je realizováno přesně podle principu, který jsem zmínil výše a to skrze vnitřní stav entity. Třída MyEntity, která reprezentuje entitu autíčka, obsahuje metodu Push (výpis 14) pro „postrčení“ modelu autíčka danou silou.



Obrázek 24: Pohyb autíček

```
//postrceni modelu auticka
public void Push(Vector3 force)
{
    //aplikovani sily na fyzikalni model teto entity
    this.PhysicsEntity.ApplyForce(
        force,    //velikost sily
        false     //true = lokalne, false = globalne
    );

    //update stavu entity
    this.PhysicsEntity.UpdateState(true);
}
```

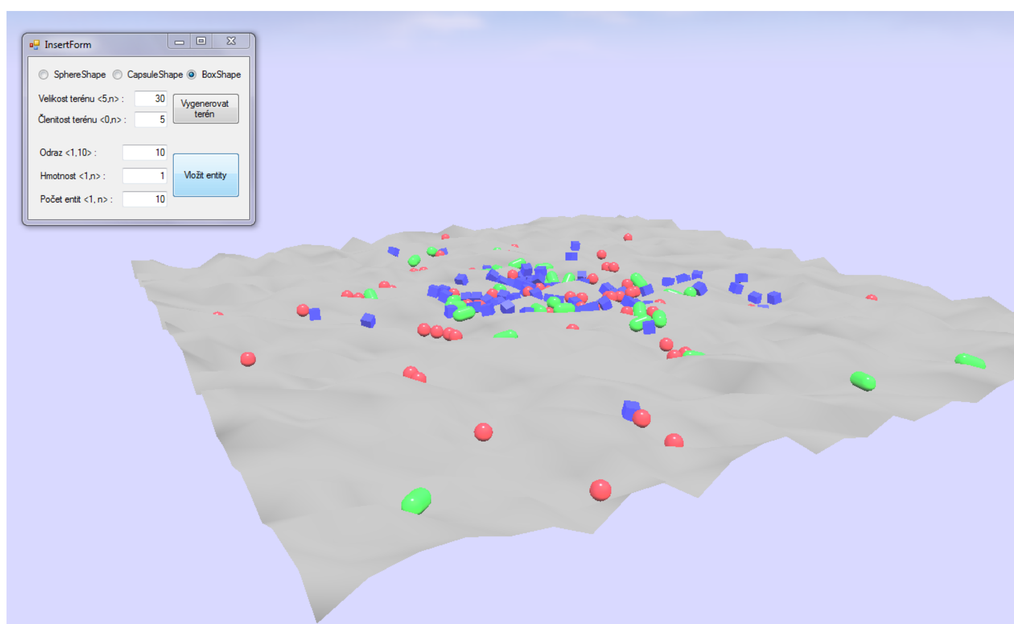
Výpis 14: Metoda Push pro „postrčení“ autíčka

Po stisknutí tlačítka Push na ovládacím formuláři, se tedy pro každé autíčko zavolá jeho vnitřní metoda Push. Všechny autíčka tedy budou postrčeny danou silou dolů z roviny. Rozdíl v materiálech, použitých u jednotlivých modelů, je patrný na první pohled (obrázek 24). Fialové autíčko má nastaveno nulové tření, proto se nikdy nezastaví a bude se pohybovat konstantní rychlostí po ploše stále do větší vzdálenosti (samozřejmě toto není vidět na obrázku). Zbylé tři autíčka se budou pohybovat dle nastaveného materiálu. Žluté autíčko s největším třením se zastaví nejdříve, červené s minimálním třením nejpozději a zelené, s poloviční hodnotou tření, někde mezi žlutým a červeným.

Ukázka použití entity terénu a vlastnosti odrazu

Tato ukázková úloha demonstruje práci s výškovým terénem a fyzikální vlastností Restitution (odraz). Po spuštění simulace je zobrazena defaultní scéna s oblohou a sluncem, která však místo standardní rovné země obsahuje vygenerovaný výškový terén. Součástí simulace je také ovládací formulář, kde je možno nastavit rozměr a členitost tohoto terénu. Vše je vidět na obrázku 25.

Vytvořený terén je ve skutečnosti entita typu HeightFieldShape (TerrainEntity), která byla dříve zmíněna v kategorii enviromentálních entit. Entita je vytvořena pomocí pole výškových bodů, které určují jak vysoký či nízký bude terén v daném místě. Tyto výškové

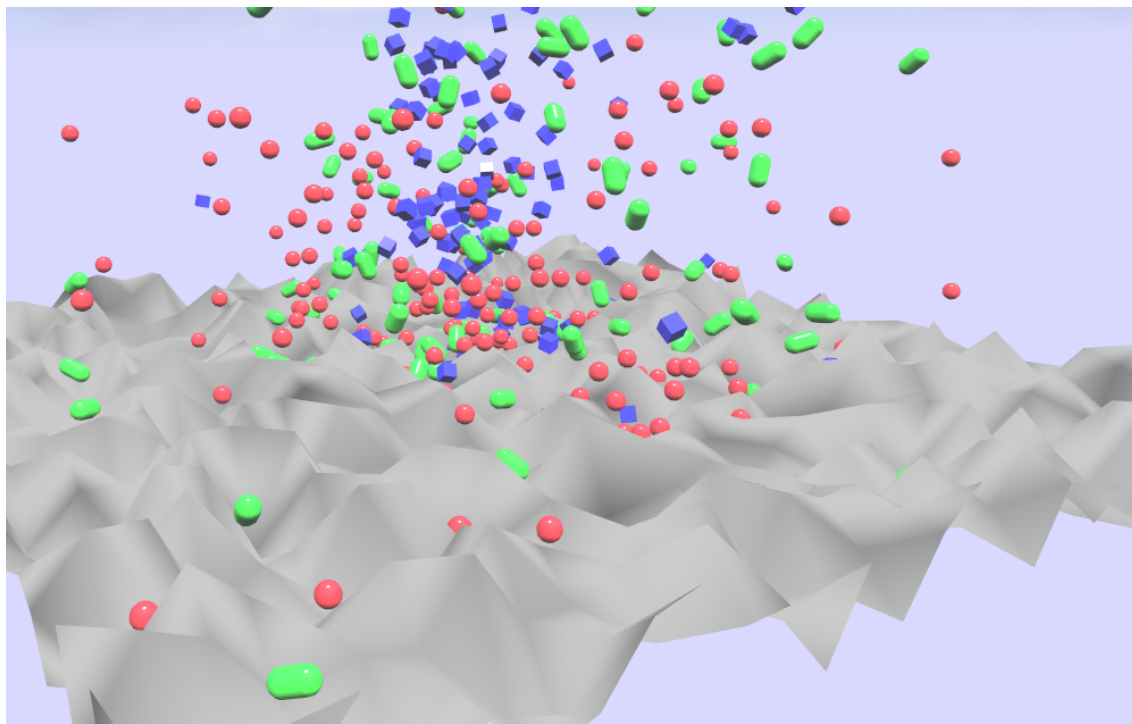


Obrázek 25: Vygenerovaný výškový terén

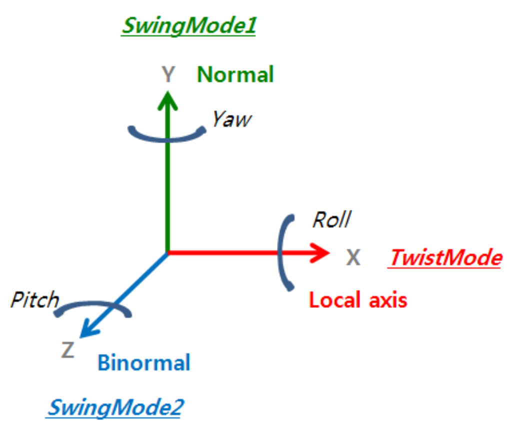
body jsou náhodně generovány z hodnoty členitosti terénu, zadané v zobrazeném formuláři. Při defaultní členitosti 5 (základní jednotka délky, v rámci simulace, je metr), jsou tedy náhodně generovány výškové body v rozmezí -5 až +5 a při nastavení členitosti rovné 0, je vygenerována rovná plocha. Terén lze také vytvořit např. pomocí bitmapového obrázku s odstíny šedi, definujícími jeho výšku. Rozměry terénu lze nahradit pixelovým rozměrem obrázku a výškové body jsou stupně šedi jednotlivých pixelů v rozmezí od 0 - 255.

Kromě standartního materiálu obsahuje tento terén i použití tzv. efektu. Efekt je soubor s příponou „fx“, který je přiřazen dané entitě. V tomto případě má tento soubor za následek výpočet stínu, aplikovaného jako texturu na model terénu. To ve výsledku umožňuje snadnější vizuální rozlišení jednotlivých výškových rozdílů. Čím vyšší je členitost terénu, tím výraznější je stín vrcholů.

Nyní se dostáváme k demostraci fyzikální vlastnosti odrazu. Pomocí zobrazeného formuláře lze totiž do scény vložit libovolný počet entit. U těchto entit je možnost nastavit právě velikost odrazu a jejich hmotnost. Lze volit entity ve tvaru koule, kapsule nebo krychle. Tyto objekty jsou postupně vkládány do simulace do určité výšky nad vygenerovaný terén, na nějž následně dopadají. Snad ani není třeba popisovat rozdíly v chování entit, při odlišném nastavení velikosti odrazu a hmotnosti. Na obrázku 26 je vidět cca 600 entit, poletujících a válejících se po vygenerovaném terénu. Za zmínku stojí, že Express edice MRDS obsahuje omezení na počet entit v simulaci, kterých může být maximálně 64. V mnou použité verzi MRDS s Academic licencí se toto omezení nenachází, proto jsem měl možnost vložit do simulace libovolný počet objektů.



Obrázek 26: Výškový terén s množstvím vložených entit



Obrázek 27: Pracovní módy spojů

Spoje

Spoje (Joints) jsou velice důležitou součástí fyzikálního enginu. Jsou nezbytnou součástí při návrhu pohyblivých částí, přičemž se nemusí vždy nutně jednat o robota nebo některou jeho část. Typickým příkladem využití spojů je např. robotická paže KUKA LBR3, jež je součástí MRDS. Obecně lze tedy říci, že jsou určeny pro spojování entit. Každý spoj je charakterizován svými fyzikálními vlastnostmi, které využívá právě fyzikální engine. Spoje lze rozdělit do několika kategorií dle jejich DOF [6].

Rozdělení spojů dle DOF:

- Angular (úhlové)
  - Twist (X)
  - Swing1 (Y)
  - Swing2 (Z)
- Linear (lineární, translační)
  - Local axis (X)
  - Normal (Y)
  - Binomal (Z)

Výše uvedené rozdělení spojů dle DOF lze chápat jako pracovní módy, které lze nastavit jednotlivým spojům. V základu se rozdělují na úhlové a translační. Úhlové spoje umožňují entitě rotaci vzhledem k vybrané ose, jež je určena zvoleným pracovním módem spoje (Twist, Swing1, Swing2). Lineární nebo také translační spoje, slouží k pohybu enity ve směru zvolené osy. Opět obsahuje tři pracovní módy (Local axis, Normal, Binormal), které rozlišují jednotlivé osy. Přehledné zobrazení všech os a jim odpovídajících pracovních módů se nachází na obrázku 27.

Tímto však možnosti spojů nekončí. Uvedené základní typy spojů lze různě kombinovat a vytvářet tak spoje nové, s pokročilejší funkcionalitou. Díky čemuž lze spoje rozdělit do dalších čtyř kategorií, rozlišující spoje z hlediska typu jejich použití. Tento takzvaný typ spoje tedy přímo souvisí s nastavenými pracovními módy spoje.

Rozdělení spojů dle typu použití:

- Revolute,
- Spherical,
- Prismatic,

[6]Degree Of Freedom, označení pohybu/rotace objektu vzhledem k určitým osám

- Cylindrical.

Revolute spoje umožňují rotaci entity vzhledem k jedné vybrané ose. Tento spoj lze vytvořit použitím jednoho z jmenovaných úhlových módů (Twist, Swing1 nebo Swing2), který určuje osu rotace. Pojmem Spherical jsou označovány spoje, které dovolují entitě rotaci vzhledem ke dvěma zvoleným osám. Tohoto lze dosáhnout kombinací dvou spojů, s různými úhlovými pracovními módy (Twist+Swing1, Swing1+Swing2, ...). Spoje typu Prismatic jsou čistě translační a umožňují entitě pohyb po ose X. Posledním typem je Cylindrical spoj, který slučuje vlastnosti spoje Revolute a Prismatic. Entitu lze tedy rotovat i s ní pohybovat, vzhledem k ose X. Uvedeným způsobem lze kupříkladu vytvořit spoj typu Spherical, který umožňuje entitě rotaci ve dvou osách a reprezentuje tedy funkci rameního kloubu.

Spoj je ve skutečnosti realizován jako instance třídy PhysicsJoint. Ta obsahuje vlastní nastavení, jako je použitý typ spoje (Angular nebo Linear) , odpovídající mód v němž bude pracovat a maximální velikost síly, kterou lze na spoj působit. Kromě těchto základních nastavení je nutno vytvořit tzv. Spring property (volně přeloženo jako vlastnost pružiny), která zajistí fixaci spoje vzhledem k nastaveným osám. Pomocí této třídy se také nastavují další vlastnosti spoje, jako je tuhost (SpringCoefficient), tlumení (DamperCoefficient) a klidová pozice spoje (EquilibriumPosition). Takto vytvořenému spoji pak již pouze stačí dodefinovat body dvou různých objektů, mezi nimiž se má vytvořit požadované spojení. Přesný postup pro vytvoření spoje se nachází níže, v popisu ukázkové úlohy.

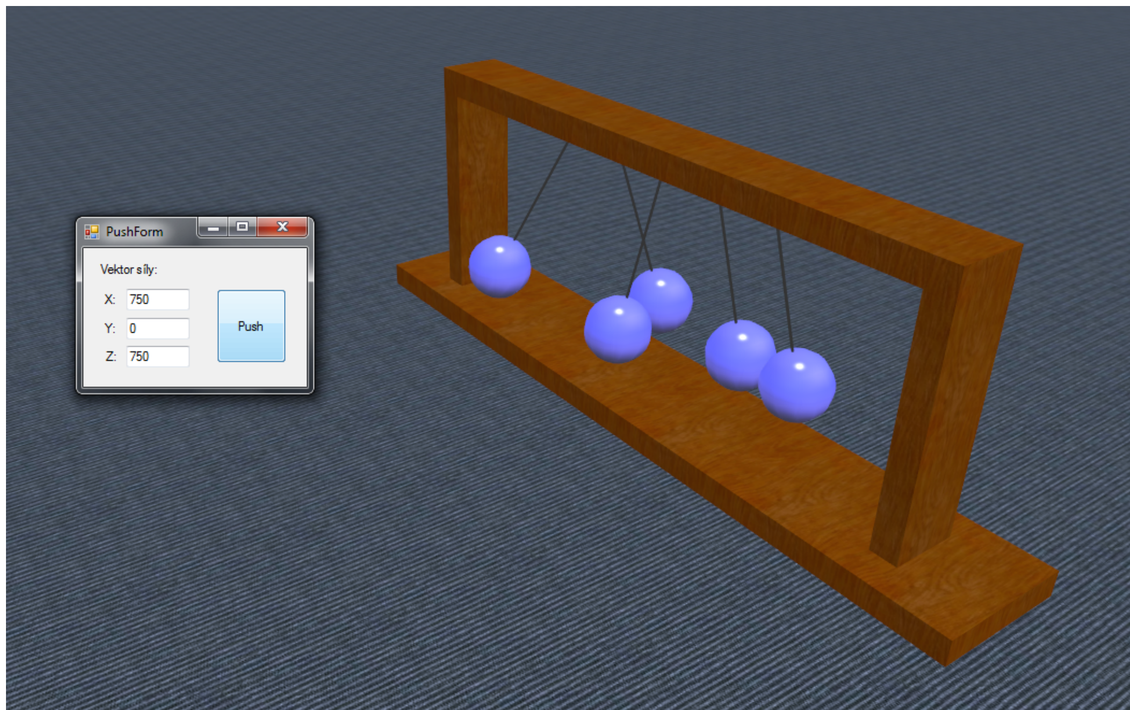
Pro rozpohybování spoje můžeme použít buď sílu působící na entitu ve spoji, stejný princip jako v případě modelů autíček v kapitole pojednávající o síle, nebo metody přímo obsažené ve třídě PhysicsJoint. Při použití úhlových spojů se jedná o metody SetAngularDriveOrientation pro nastavení úhlové pozice spoje a SetAngularDriveVelocity pro nastavení rychlosti otáčení. Analogicky u lineárních spojů lze použít metody SetLinearDrivePosition a SetLinearDriveVelocity.

Ukázka použití spojů pro vytvoření kyvadel

Po spuštění simulace se ve scéně nachází objekt (obrázek 28), připomínající známou fyzikální hříčku se zavěšenými kuličkami. Tento objekt je složen z různých částí dřevěného podstavce a z pěti kyvadel. Kyvadlem je myšlena entita, jež je složena z kuličky na tyčce, zavěšené na jednom dřevěném dílku.

Kyvadlo je reprezentováno vlastní třídou MyPendulum. Fyzicky je kyvadlo složeno z třech dílčích entit tvaru BoxShape, CapsuleShape a SphereShape. Na obdelník, tvořící část vrchního dřevěného úchytu, je uchycena tyčinka ve tvaru SpereShape, na níž visí samotná kulička.

Spoje jsou zde využity právě pro spojení těchto tří entit v jeden objekt, reprezentující kyvadlo. První spoj se nachází mezi spodní částí dřevěného úchytu a horním koncem tyčinky, druhý pak mezi spodním koncem tyčinky a vrchní částí kuličky. Oba použité spoje jsou úhlové. Pro dosažení co největšího realismu má horní spoj nastaven všechny tři



Obrázek 28: Kyvadla

pracovní módy, takže umožňuje rotaci ve všech třech osách X, Y, Z. Tento spoj umožňuje kuličce pohyb v každém směru a je obdobou tzv. Foucaultova kyvadla. Podrobný popis vytvoření tohoto spoje se nachází ve výpisu 15. Druhý spoj slouží pouze pro pevné uchycení kuličky k tyčince, je proto uzamčen (JointDOFMode.Locked) ve své klidové pozici.

```
//vytvoreni uhloveho spoje
JointAngularProperties commonAngular = new JointAngularProperties();

//nastaveni modu pro rotaci kolem os X, Y, Z
commonAngular.TwistMode = JointDOFMode.Free;
commonAngular.Swing1Mode = JointDOFMode.Free;
commonAngular.Swing2Mode = JointDOFMode.Free;

//nastaveni pohybu twist modu
commonAngular.TwistDrive = new JointDriveProperties(
    JointDriveMode.Position,          // rizeni dle pozice
    new SpringProperties(1, 1, 0),    //tuhost, tlumeni, klidova pozice spoje
    10000);                           // limit na silu

//nastaveni pohybu swing modu
commonAngular.SwingDrive = new JointDriveProperties(
    JointDriveMode.Position,          // rizeni dle pozice
    new SpringProperties(1, 1, 0),    //tuhost, tlumeni, klidova pozice spoje
    1000000);                         // limit na silu
```

```
// vlastnosti spoje
JointProperties jointProps = new JointProperties(commonAngular, null, null);
jointProps.EnableCollisions = true;

// vytvoreni a pojmenovani spoje
joint1 = PhysicsJoint.Create(jointProps);
joint1.State.Name = "Joint1";

// nastaveni konektoru pro spoj mezi boxem a kapsuli
joint1.State.Connectors[0] = new EntityJointConnector(
    box,                          // prvni entita
    new Vector3(0, 1, 0),         // normala
    new Vector3(1, 0, 0),         // osa
    new Vector3(0, -0.1f, 0));    // bod pro umisteni spoje

joint1.State.Connectors[1] = new EntityJointConnector(
    capsule,                      // druha entita
    new Vector3(0, 1, 0),         // normala
    new Vector3(1, 0, 0),         // osa
    new Vector3(0, 0.5f, 0));     // bod pro umisteni spoje

// vytvoreni fyzikalni reprezentace spoje
PhysicsEngine.InsertJoint(joint1);
```

Výpis 15: Vytvoření spoje pro rotaci kolem všech os

Celkový objekt ve scéně je tedy vytvořen z pěti instancí třídy MyPendulum. Součástí simulace je i formulář, kde lze nastavit velikosti síly, kterou má být postrčena poslední vytvořená kulička. Pomocí této síly, jež je určená vektorem, se dá lehce rozpohybovat všech pět kuliček. Snažil jsem se vytvořit model zmíněné fyzikální hříčky, kdy se jedním rozpohybováním neustále předává energie mezi krajními kuličkami. Nepodařilo se mi však najít žádný precizní návod, pro správné nastavení jednotlivých fyzikálních vlastností entit a pouhým tipováním velikosti tření a odrazu, jsem tohoto efektu nedocílil. I přesto však úloha slouží jako ukázka fyzikalních možností spojů.

4 Propojení .NET Micro Frameworku a Microsoft Robotics Developer Studia

Tato kapitola se zabývá možným způsobem vzájemné komunikace mezi .NET Micro Frameworkem a Microsoft Robotics Developer Studiem. Jejím obsahem je popis komunikačních protokolů, které tyto nástroje používají v prostředí webových služeb, přičemž princip vzájemné komunikace plyne právě z těchto komunikačních protokolů. Následně je zde uveden možný princip tohoto propojení a hlavně zdůvodnění, proč by bylo výhodné tuto komunikaci mezi oběma nástroji vůbec realizovat.

4.1 .NET Micro Framework a DPWS

Pod pojmem DPWS[7] si lze představit množinu implementací, vycházejících ze specifikací webových služeb. Microsoft poprvé představil DPWS[24] v květnu 2004. Jedná se tedy o implementaci mechanismů, jež umožňují klientovi jednoduché objevení zařízení, připojených v síti. Když klient takovéto zařízení objeví, může si od něj vyžádat popis na něm běžících služeb a následně je schopen tyto služby využívat. Windows Vista obsahuje nativně DPWS pod názvem WSDAPI[8], díky čemuž je schopen vyhledávat a využívat různá zařízení v síti (tiskárny, skenery, ...).

Funkcionalita DPWS:

- zjištění DPWS zařízení v síti a služeb které nabízí,
- zasílání zpráv DPWS zařízení a příjem odpovědí,
- popis služby prostřednictvím WSDL,
- interakce s vybranou službou dle WSDL,
- předplácení služeb a notifikace událostí.

.NET Micro Framework obsahuje podporu pro DPWS[18] již od verze 3.0 (aktuální verze je 4.0). Obsahuje tzv. DPWS Stack, který byl vytvořen speciálně pro .NET Micro Framework a nepoběží tedy na klasickém Windows počítači, s výjimkou emulátoru. DPWS knihovna pro .NET Micro Framework však neobsahuje veškerou funkcionalitu, kterou lze najít u plnohodnotného DPWS. Obsahuje pouze potřebnou podmnožinu funkcí pro podporu DPWS na malém embedded zařízení, přesto však nabízí množství různých konfigurací vzhledem k Host-Klient architektuře webových služeb.

Možné způsoby použití DPWS u .NET Micro Frameworku:

1. Host služba - .NET Micro Framework zařízení

[7] Device Profile for Web Services

[8] Web Services on Devices API, imlementace DPWS pro Windows Vista a Windows Server 2008

- Klient služba - .NET Micro Framework emulátor
- Klient služba - .NET Micro Framework druhé zařízení
- Klient služba - WSDAPI klient na Windows počítači
- Klient služba - WCF klient aplikace na Windows počítači

2. Klient služba - .NET Micro Framework zařízení
   - Host služba - .NET Micro Framework emulátor
   - Host služba - .NET Micro Framework druhé zařízení
   - Host služba - WSDAPI host na Windows počítači
   - Host služba - WCF host aplikace na Windows počítači

Protože DPWS stojí na specifikaci webových služeb, samotná komunikace probíhá skrze známý SOAP[9] protokol a zasílané zprávy jsou ve formátu XML. .NET Micro Framework obsahuje zmíněnou knihovnu pro práci s DPWS, jako i funkce pro práci s XML (XML parsery apod.). Pro použití WSDL u .NET Micro Frameworku již stačí jen dané zařízení připojit k síti, což s nabídkou dnešních kitů a modulů není problém. Například výše popsaný kit Tahoe-II obsahuje již v základu RJ-45 konektor pro TCP/IP komunikaci a patici pro volitelný wifi modul.

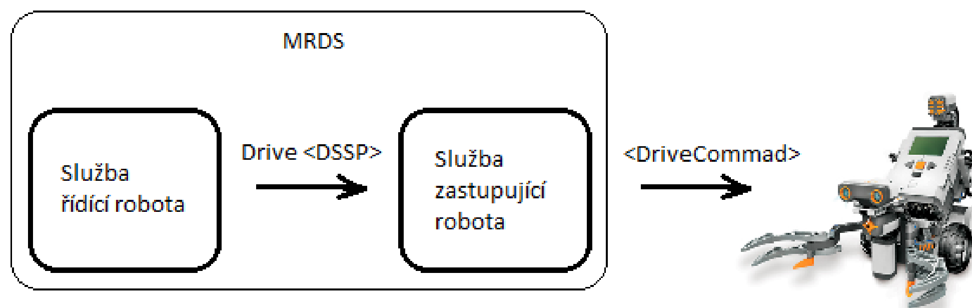
4.2 Microsoft Robotics Developer Studio a DSSP

Jak již bylo zmíněno, MRDS je postavené na servisně orientované architektuře a komuninikace mezi jednotlivými službami je realizována prostřednictvím zasílání zpráv. Protokol DSSP slouží právě pro komunikaci mezi těmito službami a je základním prvkem aplikačního modelu DSS. DSSP vychází z protokolu SOAP, což je všeobecně známý protokol, používaný pro výměnu dat mezi webovými službami. DSSP definuje jakýsi odlehčený servisní model, avšak se zachováním známých pojmů, jako je identita služby, stav služby a vztah mezi službami. DSSP definuje jednotný model pro vytváření, správu, předplácení a organizaci služeb. Níže jsou uvedeny operace, obsažené v tomto aplikačním modelu DSSP.

Aplikační model DSSP:

- vytvoření služby (operace CREATE),
- ukončení služby (operace DROP),
- zjištění kontextu služby (operace LOOKUP),
- zjištění stavu služby (operace GET, QUERY),

[9]Simple Object Access Protocol, protokol pro výměnu zpráv založených na XML



Obrázek 29: Princip komunikace MRDS s fyzickým robotem

- modifikace stavu služby (operace INSERT, UPDATE, UPSERT, DELETE, REPLACE),
- notifikace změny stavu služby (operace SUBSCRIBE).

Samotné zprávy mají formu strukturovaných dat ve formátu XML. Operace obsažené v DSSP jsou navrženy jako podpora pro manipulaci se strukturovanými daty, ale pro samotný přenos zprávy je využit protokol HTTP, stejně jako v případě protokolu SOAP. Pro výměnu zpráv mezi dvěma službami jsou definované dva vzory. První z nich je One-way a používá se u operací, kde není očekávána odpověď (např. operace INSERT). Druhý vzor s názvem Request-Response, se skládá z požadavku a odpovědi (např. operace SUBSCRIBE a QUERY). Protože celá struktura zpráv vychází z protokolu SOAP, lze u DSSP najít množství společných prvků, jako je například ošetření selhání, jež je u DSSP řešeno přesně dle specifikace protokolu SOAP.

4.3 Vzájemná komunikace

Je zřejmé, že vzájemnou komunikaci mezi .NET Micro Frameworkem a MRDS by teoreticky bylo možno realizovat prostřednictvím výše zmíněných protokolů. .NET Micro Framework obsahuje podporu pro DPWS, jež komunikuje s ostatními službami skrze protokol SOAP. Na druhou stranu u MRDS probíhá komunikace mezi službami prostřednictvím protokolu DSSP, který ale principiálně také vychází z protokolu SOAP.

K čemu je nám však takovéto propojení dobré? MRDS má jednu vcelku velkou nevýhodu, plynoucí právě z architektury webových služeb, na níž je postaven. Ta je patrná hlavně při práci s fyzickým robotem. Při psaní služby řídící robota, se její kód nevykonává přímo ve fyzickém robotovi, ale pouze komunikuje se službou, která daného robota zastupuje. Tato zástupná služba pak již přímo vysílá dané příkazy robotovi. Tento přenos však již neprobíhá prostřednictvím protokolu DSSP, ale pomocí nějakého nativního protokolu určeného pro komunikaci s robotem. Princip je vyobrazen na obrázku 29.

Například u robota Lego NXT Tribot jsou tyto nativní příkazy zasílány robotovi přes Bluetooth. Robot narazí na překážku, pošle skrze Bluetooth zprávu své zástupné službě,

že došlo ke kontaktu. Zástupná služba to pak již pomocí DSSP protokolu dá vědět naší řídící službě, která teprve danou situaci vyhodnotí. Tento typ komunikace je poměrně zdlouhavý a vede k problémům pomalého přenosu dat, ztráty dat, nízké úrovně signálu apod. Neumožňuje tedy tzv. real-time řízení robota.

Ideální řešení by tedy bylo, kdyby byl kód vytvořené DSS služby zpracováván přímo v robotovi, nebo alespoň v jeho „těsné blízkosti". To by umožnilo minimalizaci, či úplné odstranění výše zmíněných problémů. Zde přichází nápad použít .NET Micro Framework, protože mezi jeho přednosti patří především řízený kód a běh aplikace bez operačního systému. Komunikace mezi .NET Micro Frameworkem a MRDS by tedy mohla probíhat skrze aplikaci v roli prostředníka, která by byla schopna parsovat zprávy SOAP na DSSP (nebo naopak). Díky tomuto by bylo možné použít .NET Micro Framework pro samotné řízení robota, kdy by komunikace robota se službou běžící na vzdáleném počítači, byla nahrazena komunikací se službou běžící na malém .NET Micro Framework procesoru, umístěném např. přímo v robotovi.

5 Závěr

Přestože je .NET Micro Framework nejnovějším přírůstkem v rodině Windows Embedded, stále více si upevňuje svou pozici v oblasti programování těch nejmenších zařízení. Jeho popularita roste nejen díky jeho hlavním přednostem, mezi které patří především běh aplikací bez operačního systému a řízený kód, ale také proto, že s každou novou verzí tohoto frameworku se objevují i nové knihovny, dále rozšiřující stávající funkcionalitu frameworku. Výrobci samotných embedded zařízení drží krok s neustále se zvyšující podporou nového hardwaru, což ve výsledku umožňuje programátorům embedded aplikací využívat stále nové softwarové i hardwarové funkce. V této části jsem se pokusil navázat na mou bakalářskou práci[20], jež byla zaměřena na hlavní funkce .NET Micro Frameworku. Zde se však převážně zabývám právě novými funkcemi frameworku, které jsem názorně demonstroval na několika praktických úlohách, vytvořených na vývojovém kitu Tahoe-II.

Microsoft Robotics Developer Studio je všestranným nástrojem pro použití v oblasti robotiky. Vzhledem k jeho robustnosti, bylo cílem této části, podrobněji sezámit čtenáře s grafickými a fyzikálními možnostmi simulace. Kromě popisu základní architektury celého frameworku, obsahuje práce podrobnější popis simulačního enginu, jež se stará o správu entit v simulaci a vykreslování výsledné 3D scény, prostřednictvím XNA Frameworku. Podrobněji jsou zde popsány i hlavní funkce fyzikálního enginu, jež je poskytován firmou NVIDIA pod názvem NVIDIA PhysX engine. Ten má na starost výpočty týkající se fyzikálních vlastností a veličin, nebo také vytváření fyzikálních reprezentací modelů a spojů mezi entitami. Tyto popsané hlavní funkce simulačního a fyzikálního enginu jsou prakticky předvedeny na několika vytvořených ukázkových úlohách, jež demonstrují široké spektrum použití simulačního prostředí. Díky obsažené funkcionalitě se tedy simulační prostředí nejenže dokáže téměř vyrovnat s prostředím reálného světa, ale v mnoha případech je jeho použití téměř nutností. Kupříkladu v oblasti návrhu robotických prototypů a jejich testování ve specifickém prostředí (pohyb robota na povrchu planety Mars apod.).

Třetí část práce obsahuje stručný popis principu komunikace obou nástrojů s webovými službami, z čehož plyne i možný způsob vzájemné komunikace mezi .NET Micro Frameworkem a Microsoft Robotics Developer Studiem. Toto propojení se opírá o podobnost použitých komunikačních protokolů SOAP a DSSP. Netvrdím, že mnou popsaný způsob je jediný, nebo dokonce správný, ale má sloužit jako inspirace pro možné budoucí využití výhod .NET Micro Frameworku v oblasti robotiky.

Václav Svatoň

6 Reference

[1] *.NET Micro Framework*
URL: <http://www.microsoft.com/netmf/about/default.mspx> [cit. 2010-05-03]

[2] Microsoft, *.NET Micro Framework White Paper*

[3] Intertec Components, *Camera ITM-C-328 Manual*

[4] *CCR Introduction*
URL: <http://msdn.microsoft.com/en-us/library/bb648752.aspx>
[cit. 2010-05-03]

[5] Microsoft, *Decentralized Software Services Protocol specification*

[6] *Device Solutions*
URL: <http://www.devicesolutions.net> [cit. 2010-05-03]

[7] *DSS Introduction*
URL: <http://msdn.microsoft.com/en-us/library/bb483056.aspx>
[cit. 2010-05-03]

[8] Jens Kuhner, *Expert .NET Micro Framework*, Apress, 2008.

[9] *Fyzikální engine NVIDIA PhysX*
URL: <http://www.nvidia.com/object/physx_new.html> [cit. 2010-05-03]

[10] *Grafický editor Blender*
URL: <http://www.blender.org> [cit. 2010-05-03]

[11] Device Solutions Ltd, *Meridian Technical Reference Manual*

[12] *Microsoft*
URL: <http://www.microsoft.com> [cit. 2010-05-03]

[13] *Microsoft Robotics Developer Studio*
URL: <http://msdn.microsoft.com/en-us/robotics/default.aspx> [cit. 2010-05-03]

[14] Microsoft, *Microsoft Robotics Developer Studio 2008 R2 Documentation*

[15] *Microsoft Visual Studio*
URL: <http://msdn.microsoft.com/en-us/vstudio/default.aspx> [cit. 2010-05-03]

[16] *Microsoft XNA Framework*
URL: <http://msdn.microsoft.com/xna> [cit. 2010-05-03]

[17] *MMA7455 Accelerometer*
URL: <http://www.active-robots.com/products/parallax/mma7455-3axis-accelerometer-module.shtml> [cit. 2010-05-03]

[18] *MSDN Library, Introducing DPWS*
URL: <http://msdn.microsoft.com/en-us/library/dd170125.aspx>
[cit. 2010-05-03]

[19] *Návrhový vzor továrna*
URL: <http://en.wikipedia.org/wiki/Factory_method_pattern> [cit. 2010-05-03]

[20] Václav Svatoň, *Použití .NET Micro Frameworku*, Bakalářská práce.

[21] Sara Morgan, *Programming Microsoft Robotics Studio*, Microsoft Press, 2008.

[22] *Tahoe-II Development Board*
URL: <http://devicesolutions.net/Products/TahoeII.aspx> [cit. 2010-05-03]

[23] Device Solutions Ltd, *Tahoe-II Technical Reference Manual*

[24] Microsoft, *The Devices Profile for Web Service specification*

[25] *Visual Simulation Enviroment*
URL: <http://msdn.microsoft.com/en-us/library/bb896718.aspx>
[cit. 2010-05-03]

[26] *Windows Embedded, .NET Micro Framework*
URL: <http://msdn2.microsoft.com/cs-cz/embedded> [cit. 2010-05-03]