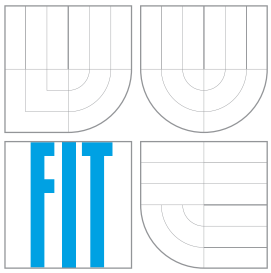
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ZÁSUVNÝ MODUL PRO UNITY IMPLEMENTUJÍCÍ FUZZY STAVOVÝ STROJ

FUZZY STATE MACHINE PLUGIN FOR UNITY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — PATRIK HRONSKÝ
AUTHOR

VEDOUCÍ PRÁCE — Ing. RUDOLF KAJAN
SUPERVISOR

BRNO 2014

Abstrakt

Tato bakalářská práce se zabývá implementací fuzzy stavového stroje v podobě zásuvného modulu pro Unity engine. Přibližuje čtenáři problematiku fuzzy stavových strojů a poukazuje na spůsoby řešení některých problémů. Seznamuje čtenáře s prostředím Unity enginu a vysvětluje možnosti rozšiřitelnosti enginu pomocí zásuvních modulů a knihoven. Popisuje návrh a implementaci knihovny za použití jednoduchých a srozumitelných obrázků. Použitelnost knihovny je demonstrována ukázkovou scénou *Trolls* vytvořenou v Unity engine s využitím vytvořené knihovny.

Abstract

This bachelor's thesis addresses the implementation of a fuzzy state machine via a plugin for the Unity engine. It describes the subject of fuzzy state machines and shows the solutions to a number of problems. It introduces the reader to the Unity engine and explains how it can be augmented by plugins and libraries. It deals with the design and implementation of our library with the help of simple and comprehensible pictures. The usability of our library is demonstrated by the *Trolls* scene, which was created in the Unity engine using our library.

Klíčová slova

fuzzy logika, fuzzy stavové stroje, zásuvný modul, Unity

Keywords

fuzzy logic, fuzzy state machines, plugin, Unity

Citace

Patrik Hronský: Zásuvný modul pro Unity implementující fuzzy stavový stroj, bakalářská práce, Brno, FIT VUT v Brně, 2014

Zásuvný modul pro Unity implementující fuzzy stavový stroj

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Rudolfa Kajana a že jsem uvedl všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Patrik Hronský
19. května 2014

Poděkování

Ďakujem pánovi Ing. Rudolfovi Kajanovi za odbornú pomoc pri riešení bakalárskej práce a rady pri písaní textu bakalárskej práce.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Tradičná Booleova logika nachádza svoje uplatnenie v nespočetných odvetviach sveta informačných technológií. Vo väčšine týchto odvetví plne postačuje pri riešení problémov danej oblasti, výnimku však tvorí oblasť umelej inteligencie.

Pri riešení jednoduchých problémov a simulácii jednoduchej umelej inteligencie sú prostriedky poskytnuté Booleovou logikou postačujúce, na opis zložitejšej a reálnejšej umelej inteligencie však nestačia. Pre simuláciu zložitejších myšlienkových pochodov, ako napríklad strachu, dôvery alebo hnevu, sú dva stavy poskytnuté klasickou logikou nepostačujúce. Z tohto dôvodu vznikla v oblasti umelej inteligencie technika zvaná *fuzzy logika*.

Potreba modelovať vierohodnú umelú inteligenciu vzniká okrem iného aj v oblasti počítačových hier. Vzhľadom na neuveriteľne rýchly a veľký posun v oblasti hernej grafiky sa do popredia dostali nové faktory, ktoré rozhodujú o úspechu či neúspechu. A práve umelá inteligencia je jedným z nich.[9] Dnešným hráčom už nestačí, že ich oponent reaguje na ich akcie, vyžadujú, aby tieto reakcie boli reálne a vierohodné. Konečným cieľom je, aby hráč necítil, že súperí s počítačom, ale mal pocit, že proti nemu stojí reálna ľudská inteligencia. Niekedy je dokonca žiadúce, aby počítačový oponent podvádzal a mal k dispozícii prostriedky, ktoré ľudský hráč poruke nemá, pretože sa tým vyrovnáva fakt, že počítačový oponent nie je schopný skutočnej improvizácie a iných výhod ľudskej inteligencie.[8]

Táto práca sa zaoberá herným enginom *Unity* a jeho vstavaným vývojovým prostredím (angl. *integrated development environment – IDE*). IDE sa dá rozšíriť použitím *zásuvných modulov* (angl. *plugin*), ktoré tvoria nadstavbu nad základnými možnosťami enginu. Na trhu momentálne neexistuje zásuvný modul, ktorý by zjednodušoval prácu s fuzzy logikou pri tvorbe umelej inteligencie v hrách, a práve jeho návrh a implementácia je cieľom tejto práce.

Kapitola 2 predstavuje fuzzy logiku, opisuje jej základné princípy a možnosti využitia pri modelovaní umelej inteligencie v počítačových hrách. Nasleduje kapitola 3, ktorá predstavuje herný engine Unity, jeho možnosti pri tvorbe počitačových hier a koncept zásuvných modulov pre Unity. V kapitole 4 predstavujeme samotný návrh zásuvného modulu, princípy jeho funkčnosti a opis implementácie. Nakoniec sa v kapitole 5 venujeme stručnému predstaveniu ukážkovej scény *Trolls* a prínosu implementovaného zásuvného modulu pri tvorbe tejto scény.

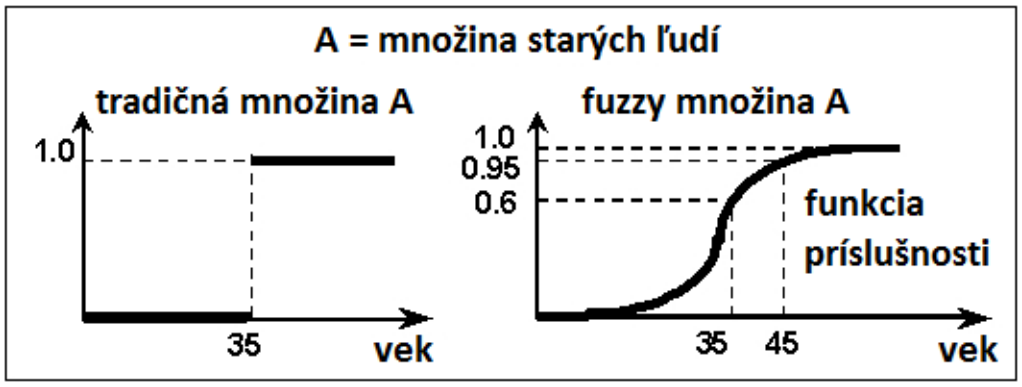
Kapitola 2

Fuzzy logika

Pre pochopenie fuzzy logiky je vhodné vysvetliť jej rozdiely v porovnani s tradičnou logikou. Tá pracuje len s dvomi hodnotami, *true* (*pravda*) a *false* (*nepravda*). Každý výrok takejto logiky je ohodnotený len jedným z dvoch možných stavov. Na rozdiel od toho umožňuje fuzzy logika ohodnotiť logické výroky ľubovoľnou hodnotou zo stanoveného rozsahu, väčšinou 0 až 1. Vďaka tomu sme pomocou fuzzy logiky schopní modelovať koncept čiastočnej pravdy, kde sa miera pravdivosti pohybuje od absolútnej nepravdy až po absolútnu pravdu. [5]

Pokiaľ nie sme schopní stanoviť presné hranice triedy vymedzené neurčitým pojmom (napr. vysoký, starý alebo šťastný), nahradíme toto rozhodnutie mierou vybranou z určitej škály. Táto miera bude vyjadrovať mieru príslušnosti do danej triedy. Trieda, v ktorej je každý prvok charakterizovaný stupňom príslušnosti k danej triede, sa označuje pojmom *fuzzy množina*. [7]

Nasledujúci obrázok 2.1 jasne zobrazuje rozdiel medzi tradičným (Booleovským) a fuzzy vnímaním množín.

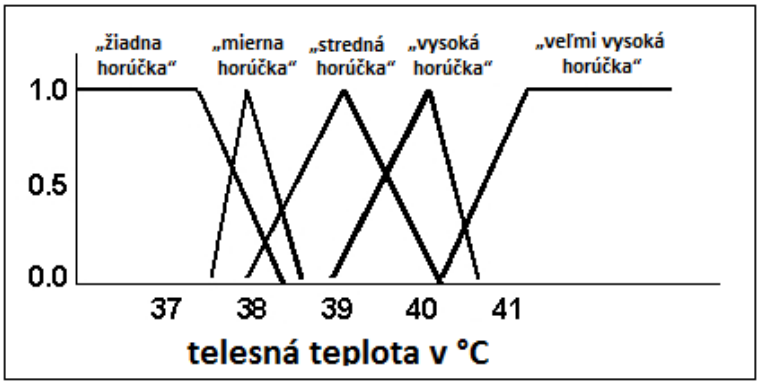


Obrázek 2.1: Tradičná a fuzzy množina.

Na obrázku vidíme, že v tradičnej logike by človek vo veku 34 rokov nespadal do kategórie starých ľudí, zatiaľ čo človek vo veku 35 rokov už plne áno, rovnako ako napríklad človek vo veku 45 rokov. Tento ostrý rozdiel na stanovenej hranici neposkytuje dostatočnú flexibilitu a nevystihuje vágny pojem veku. Fuzzy množiny využívajú namiesto jednej pevne stanovenej hranice funkciu príslušnosti s plynulejším priebehom. V dôsledku toho bude človek vo veku 35 rokov patriť medzi starých ľudí s mierou príslušnosti 0,6, zatiaľ čo človek vo veku 45 rokov už s mierou príslušnosti 0,95. Tento model omnoho výstižnejšie zachytáva

koncept veku a ľudské vnímanie a radenie v takejto vágnej skutočnosti.

Fuzzy množiny vystihujú nepresnosti prirodzeného jazyka a dokážu s nimi pracovať v podobe *lingvistických premenných*. Zatiaľ čo číselné premenné pracujú s konkrétnymi číselnými hodnotami, lingvistické premenné pracujú s jazykovými hodnotami, teda slovami, ktorým je priradená miera príslušnosti k množine.



Obrázek 2.2: Lingvistická premenná.

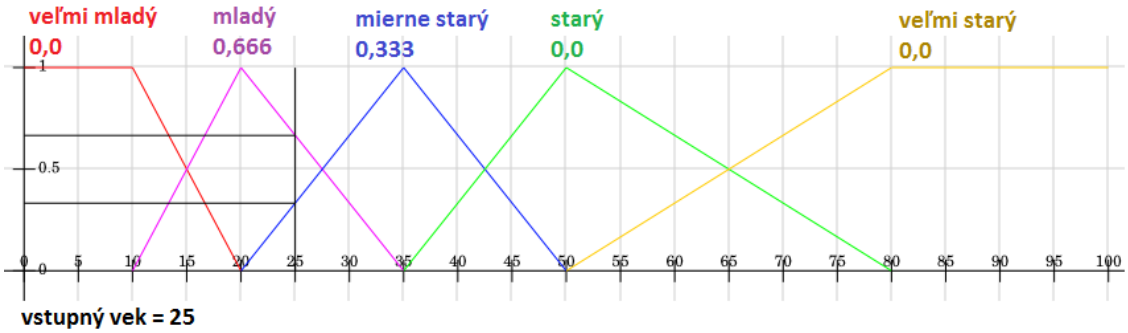
Na obrázku 2.2 je ukážka lingvistickej premennej „telesná teplota“, ktorá môže nadobúdať päť jazykových hodnôt: „žiadna horúčka“, „mierna horúčka“, „stredná horúčka“, „vysoká horúčka“ a „veľmi vysoká horúčka“. Každá z týchto jazykových hodnôt má priradenú fuzzy množinu a je definovaná funkciou príslušnosti. Napríklad pre hodnotu 38,2 °C nadobúda premenná „telesná teplota“ hodnotu „žiadna horúčka“ s mierou 0,1, zároveň hodnotu „mierna horúčka“ s mierou 0,5 a zároveň hodnotu „stredná horúčka“ s mierou 0,3.

2.1 Práca expertného systému

Expertné systémy sú počítačové programy, ktoré simulujú rozhodovanie experta v určitých úzko zameraných situáciach. Základná myšlienka je použitie znalostí experta na vytvorenie systému, ktorý bude riešiť daný problém rovnako, ako by ho riešil samotný expert. [11] Problém nastáva vo chvíli, keď sa pokúšame previesť expertov opis do formy matematických formulí, pretože expertov použitý prirodzený jazyk je charakterizovaný svojou vágnosťou. V tejto situácii nám pomôže fuzzy logika, ktorá umožňuje modelovať význam slov prirodzeného jazyka. [7]

Expertný systém pozostáva z nasledujúcich súčastí: *fuzzifikátor*, *inferenčný mechanizmus*, *defuzzifikátor* a *báza znalostí*. Fuzzifikátor dostáva na vstup konkrétne číselné hodnoty, ktoré na základe funkcií príslušnosti definovaných pre každú fuzzy množinu pretvára na mieru príslušnosti k jednotlivým fuzzy množinám.

Obrázok 2.3 ukazuje príklad fuzzifikácie. Číselná premenná „vek“ s hodnotou 25 je fuzzifikovaná použitím funkcií príslušnosti definovaných pre jednotlivé fuzzy množiny lingvistickej premennej „vek“. Výsledkom fuzzifikácie je, že lingvistická premenná „vek“ nadobúda hodnoty „mladý“ s mierou príslušnosti 0,666 a „mierne starý“ s mierou príslušnosti 0,333. Hodnoty príslušnosti pri ostatných lingvistických hodnotách nadobúdajú mieru príslušnosti 0. Analogicky sú pri fuzzifikácii spracované všetky vstupné premenné, pre ktoré existujú zodpovedajúce lingvistické premenné. [17]



Obrázek 2.3: Fuzzifikácia.

Takto vypočítané lingvistické premenné sú následne spracované inferenčným mechanizmom, ktorý využíva bázu znalostí. Báza znalostí obsahuje *fuzzy IF-THEN pravidlá*, ktoré majú nasledujúci tvar:

$$\text{IF } X_1 = A_1 \wedge X_2 = A_2 \wedge \ldots \wedge X_n = A_n \text{ THEN } Y = B,$$

kde X_i a Y sú lingvistické premenné a A_i a B sú lingvistické hodnoty. IF časť pravidla sa nazýva *premisa*, zatiaľ čo THEN časť sa nazýva *konsekvent*. Konkrétne fuzzy pravidlo môže vyzerať nasledovne:

$$\text{IF vek} = \text{„mladý“} \wedge \text{majetok} = \text{„veľký“ THEN spokojnosť} = \text{„veľká“}.$$

Inferenčný mechanizmus teda využije bázu znalostí a aplikuje fuzzy pravidlá na zodpovedajúce lingvistické premenné. Využíva sa tu fuzzy logické AND. Výsledkom inferencie budú hodnoty priradené lingvistickým premenným konsekventov pravidiel. Pri použití vyššie uvedeného fuzzy pravidla by mohla inferencia vyzerať nasledovne:

$$\text{vek} = 25, \text{majetok} = \$\,10\,000 \Rightarrow \text{vek} = \text{„mladý“ (0,666), majetok} = \text{„veľký“ (0,8)} \Rightarrow \text{spokojnosť} = \text{„veľká“ (0,666)}.$$

Keďže však môže existovať väčší počet pravidiel s rovnakou premennou v konsekvente, musia byť jednotlivé fuzzy množiny týchto premenných zlúčené použitím fuzzy logického OR. To znamená, že pokiaľ by sme v našom príklade mali ešte jedno fuzzy pravidlo, ktoré by taktiež malo v konsekvente premennú „spokojnosť“ s hodnotou „veľká“, a výsledkom inferencie pre túto premennú by bola napríklad hodnota 0,2, konečným výsledkom pre túto hodnotu by bola hodnota 0,666.

Posledným krokom je defuzzifikácia, teda prevod fuzzy výsledkov inferenčného mechanizmu do podoby konkrétnych číselných údajov. Tu sa často využíva metóda ťažiska. Všetky fuzzy množiny rovnakých lingvistických premenných konsekventov sa zlúčia do jedinej fuzzy množiny, následne sa vypočíta ťažisko novovzniknutého útvaru a jeho x-ová súradnica sa použije ako výstup defuzzifikácie. Obrázok 2.4 ukazuje celý proces práce expertného systému pre konkrétny príklad.

Je vhodné podotknúť, že pri zvyšovaní počtu fuzzy premenných narastá počet pravidiel exponenciálne a pre veľké množstvo fuzzy premenných sa stáva neúnosným. Tento jav sa nazýva *kombinačná explózia* a aby sme mu predišli, je vhodné využiť napríklad *Combsovu metódu*, ktorú v tejto práci však nebudeme rozoberať. [18]

2.2 Fuzzy stavové stroje

Fuzzy stavové stroje (alebo *automaty*) vychádzajú z konceptu *konečných stavových automatov* (angl. *finite state machines*), avšak prispôsobujú tento koncept využitiu fuzzy logiky.

Základný rozdiel medzi tradičným konečným stavovým automatom a fuzzy stavovým automatom je, že prechody fuzzy automatu sa spúšťajú prostredníctvom fuzzy premenných a nie tradičných (angl. *crisp*) udalostí a samotné prechody fuzzy automatu sú tiež fuzzy. V dôsledku toho sa v ľubovoľnom čase môže celý automat nachádzať nielen v jednom, ale v mnohých stavoch súčasne, pričom každý z týchto stavov má priradenú hodnotu označenú ako *miera členstva*. [6]

Fuzzy stavový automat, rovnako ako konečný stavový automat, je reprezentovaný množinou stavov prepojených prechodmi. Každý prechod je označený logickým fuzzy výrazom ľubovoľnej zložitosti. Vzhľadom na to, že fuzzy stavový automat sa môže nachádzať vo viacerých stavoch súčasne, musí každému stavu prislúchať miera členstva, ktorá je reprezentovaná reálnym číslom z rozsahu 0 až 1. Miera členstva vyjadruje, akou mierou sa fuzzy automat nachádza v danom stave.

Pre tradičné automaty okrem iného platí, že súčet mier členstva sa v každom momente rovná 1. To je v tradičných automatoch zrejmé, keďže sa nachádzajú vždy práve v jednom stave s mierou členstva 1. Táto podmienka však platí aj pre fuzzy stavové automaty. Hoci sa teda automat môže nachádzať vo viacerých stavoch súčasne a miera členstva sa medzi jednotlivými stavmi môže *presúvať*, súčet všetkých mier členstva sa musí v každej chvíli rovnať 1.

Z praktického hľadiska je prechod od tradičného konečného stavového automatu k fuzzy stavovému automatu otázkou niekoľkých zmien. Jednou z hlavných je, že nový automat musí podporovať viac než jeden momentálne aktívny stav. Druhou dôležitou úpravou je pridanie podpory rôznych úrovní členstva stavu. Poslednou zmenou je prispôsobenie systému prechodov tak, aby podporoval možnosť viacerých aktuálnych stavov a rôzne miery členstva jednotlivých stavov. [2]

Jedným z problémov, ktoré vznikajú pri fuzzy stavových strojoch a ktoré v prípade tradičných automatov nebolo nutné riešiť, je prenášanie aktivity medzi stavmi. V tradičnom automate je vždy aktívny práve jeden stav a práve jeden prechod, takže sa celá aktivita presunie po jednom prechode z jedného do druhého stavu. Vo fuzzy stavových strojoch však môže byť aktívnych súčasne viac stavov a viac prechodov, a preto je nutné riešiť, aké množstvo aktivity sa bude presúvať.

Jedným zo zaužívaných spôsobov je presun na základe hodnoty platnosti prechodu, teda na základe vyhodnotenia fuzzy množiny zodpovedajúcej podmienke prechodu. Pri tomto spôsobe sa ignoruje miera členstva aktuálneho stavu, teda stavu, z ktorého aktívny prechod pochádza. Tento prístup je však nevhodný pre niektoré prípady, napríklad pre použitie v počítačových hrách. Predstavme si časť fuzzy stavového automatu pozostávajúcu z dvoch stavov a jedného prechodu medzi nimi. Jeden z týchto stavov má mieru členstva 0, druhý 0,01 a podmienka na prechode medzi nimi je ohodnotená hodnotou 1. V prípade ignorácie miery členstva rodičovského stavu prechodu by došlo k tomu, že stav, ktorý bol aktívny len v miere 0,01, uvedie iný stav do aktivity v miere 1.

Predstavme si praktickú situáciu, kde stav s hodnotou 0,01 zodpovedá stavu „hľadám nepriateľa“ v umelej inteligencii strážnika v počítačovej hre, stav s hodnotou 0 zodpovedá stavu „strieľam“ a podmienka na prechode odpovedá fuzzy výroku „cieľ je jasne viditeľný“. V takejto situácii by nastalo to, že hoci sa strážnik len velmi nepatrnou mierou nachádza v stave „hľadám nepriateľa“, a teda väčšinu svojej pozornosti venuje inej aktivite (napríklad

konverzácii s iným strážnikom), pri vkročení nepriateľa do zorného poľa strážnika by okamžite a s absolútnou silou prešiel do stavu „strieľam“ a zahájil paľbu na nepriateľa s plnou pozornosťou a presnosťou sústredenou do tohto stavu. Takéto správanie je nereálne a pre potreby umelej inteligencie v počítačových hrách neakceptovateľné.

Preto musí byť pri určovaní novej miery členstva stavu použitá funkcia, ktorá berie do úvahy okrem sily prechodu aj mieru členstva rodičovského stavu. Najlepšie riešenie závisí od konkrétnej aplikácie, použiteľné je napríklad využitie minima, maxima alebo priemeru, prípadne inej vhodnej matematickej funkcie. Funkcia priradenia členstva však každopádne musí spĺňať dve základné pravidlá:

$$0 \leq F(\mu, \delta) \leq 1,$$

kde F je funkcia priradenia členstva, μ je miera členstva rodičovského stavu a δ je sila prechodu, a

$$F(0,0) = 0 \wedge F(1,1) = 1.$$

Tieto pravidlá stanovujú, že výsledok funkcie priradenia členstva sa musí nachádzať v rozsahu 0 až 1 a že funkcia musí pre nulové parametre vracať nulovú hodnotu a pre jednotkové parametre jednotkovú hodnotu.

Ďalšia záležitosť, ktorú treba mať pri tvorbe fuzzy stavového automatu na pamäti, je, že pri určitých typoch funkcie priradenia členstva existuje riziko, že súčet výsledkov všetkých týchto funkcií v konkrétnom momente presahuje hodnotu 1. Takýto prípad by mohol nastať, ak by funkcia priradenia členstva pracovala napríklad na princípe maxima, teda

$$F(\mu, \delta) = max(\mu, \delta).$$

V takomto prípade by postačovalo, aby aspoň dva prechody automatu boli vyhodnotené na hodnotu vyššiu než 0,5 (čo nie je vôbec neobvyklé) a celková suma by hneď prekračovala limitnú hodnotu. Je teda nutné buď vytvoriť funkciu priradenia členstva, pri ktorej nehrozí takáto situácia, alebo dodatočne kontrolovať výsledky týchto funkcií v automate a prípadne výsledky korigovať, aby nedošlo k prekročeniu maximálnej hodnoty aktivity v automate.

Posledným problémom, ktorý sa vzťahuje na fuzzy stavové automaty, je problém *mnohočlenstva* (angl. *multi-membership*). Ide o jav, ktorý nastáva, keď sa minimálne dva rodičovské stavy pokúšajú cez prechod nastaviť novú mieru členstva tomu istému cieľovému stavu. Tento problém je opäť možné riešiť viacerými spôsobmi. Jedným z nich je metóda *odstránenia mnohočlenstva*, ktorá pre každý prípad, kde sa viaceré stavy snažia zmeniť mieru členstva rovnakého stavu, vytvára nové, pomocné stavy, a tým mnohočlenstvo odstraňuje. Táto metóda však zvyšuje počet stavov automatu a mení celkovú štruktúru automatu, čo nie je vždy prípustné. Lepším riešením je teda použitie funkcie na riešenie mnohočlenstva. Táto funkcia prijíma ako parametre všetky hodnoty miery členstva, ktoré by mali byť novému stavu priradené, vykoná nad nimi určité operácie a vo výsledku poskytne jednu konečnú mieru členstva, ktorá bude priradená novému stavu. Konkrétne operácie, ktoré daná funkcia vykonáva, je opäť vhodné stanoviť podľa konkrétneho prípadu, použiteľnými príkladmi sú napriklad funkcia maxima alebo aritmetického priemeru. [1]

2.3 Zhrnutie fuzzy logiky

Hlavnou výhodou fuzzy logiky ako takej je jej blízkosť k prirodzenému jazyku. V dôsledku toho sú systémy založené na fuzzy logike jednoduchšie na návrh a ich zložitosť je prakticky

nezávislá na zložitosti procesu, ktorý má byť regulovaný. Fuzzy logika a systémy na nej založené je teda vhodné využívať najmä vtedy, pokiaľ je proces, ktorý máme regulovať, veľmi zložitý. [10]

Praktické použitie našla fuzzy logika ako v domácnosti, tak aj v rôznych priemyselných odvetviach a doprave. V domácnosti je využívaná napríklad v práčkach, vysávačoch, holiacich strojčekoch alebo fotoaparátoch. V doprave je použiteľná napríklad pre riadenie podzemnej dráhy, v automobiloch v protišmykovom systéme ABS alebo automatických prevodovkách, alebo napríklad v helikoptérach na ovládanie výkonov rotorov. Rovnako sú používané aj pri bezpečnosti nukleárnych reaktorov alebo regulácii rôznych priemyselných procesov, od pecí cez chemické procesy až po pohyb žeriavov. [7]

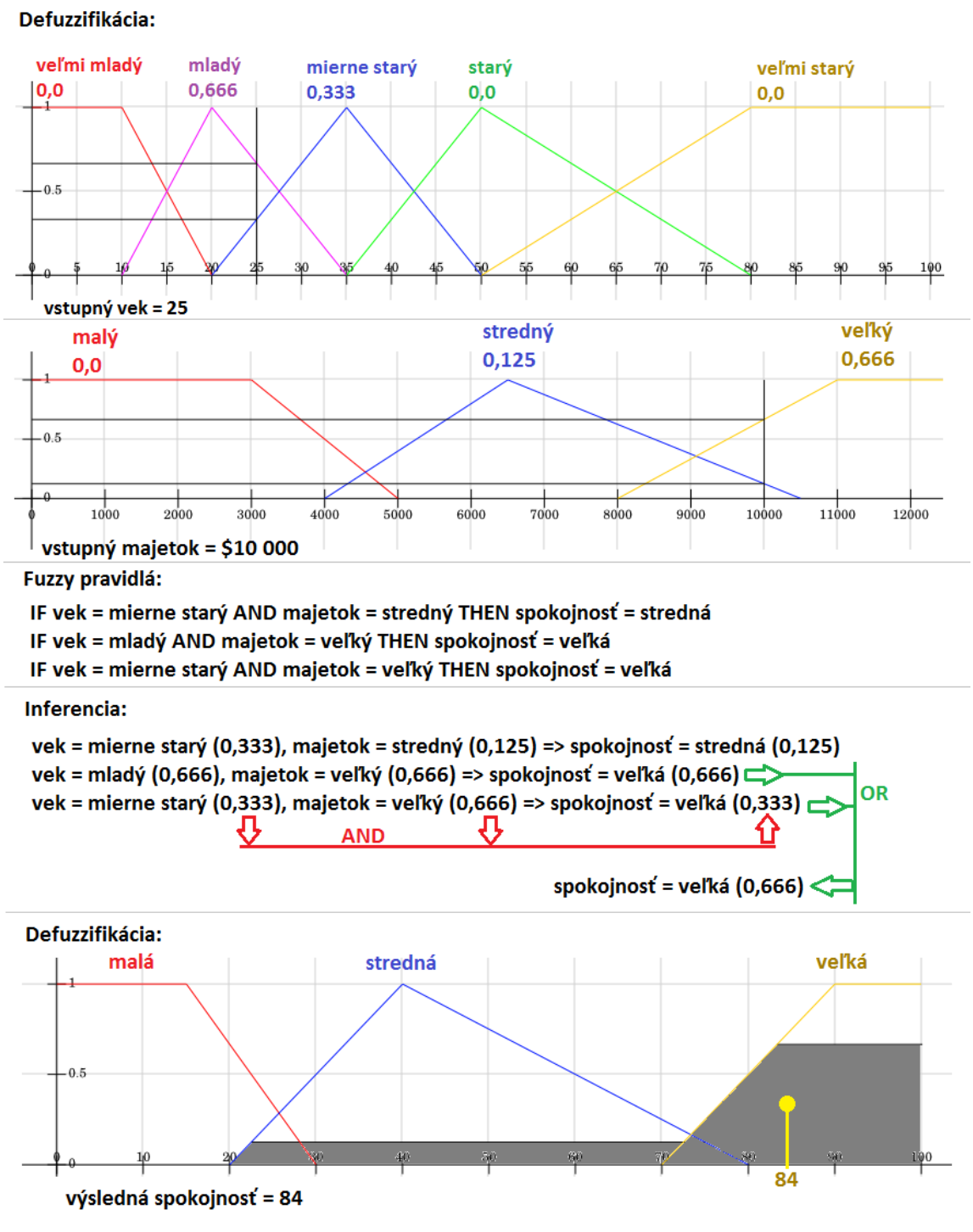
Svoje využitie našla fuzzy logika aj v hernom priemysle, a to v umelej inteligencii. Na tomto mieste považujeme za vhodné podotknúť, že pokiaľ ide o umelú inteligenciu v počítačových hrách, nehovoríme v praxi o skutočnej simulácii ľudského myslenia, ale len o dostatočne vierohodnej náhrade. Ide len o imitáciu schopnú napodobňovať reakcie ľudskej mysle pre danú konkrétnu oblasť, ktorou sa hra zaoberá.[4]

Fuzzy logiku môžeme využiť pre simuláciu emócií počítačom ovládaných postáv, kde by tradičná logika neposkytovala dostatočne širokú škálu možností. Či už ide o simuláciu toho, do akej miery sa protivník bojí hráča, alebo ako veľmi obchodník hráčovi dôveruje. Prípadné využitie je aj pri simulácii zhlukovania, kde jednotliví členovia skupiny môžu riešiť, ako ďaleko sa nachádzajú od zvyšku skupiny. Fuzzy logika môže však byť použitá aj na simuláciu správania neživých predmetov, napríklad pohybu mrakov na základe rýchlosti a smeru vetra. [3]

Hoci si to často neuvedomujeme, fuzzy logika je v hrách využívaná už od dávnych čias napríklad v systéme zdravia alebo bodov zásahu (angl. *hit points*). Namiesto toho, aby bola postava živá alebo mŕtva, čo by mohlo byť reprezentované klasickou Booleovskou logikou, predstavuje použitie bodov zásahu širokú škálu stavov, v ktorých sa postava môže nachádzať, od „absolútne zdravý“ cez „čiastočne zranený“ až po „takmer mŕtvy“. To všetko už sú fuzzy stavy.

Ďalším použitím v hrách môže byť napríklad ovládanie akcelerácie a brzdenia automobilov ovládaných počítačom. Namiesto využitia tradičných Booleovských stavov typu „plynový pedál stlačený“ a „plynový pedál voľný“ môžeme pomocou fuzzy logiky reprezentovať rôzne úrovne stlačenia plynového pedálu a teda rôzne úrovne akcelerácie, analogicky brzdenia.

Vo všeobecnosti je vhodné využiť fuzzy logiku vo všetkých prípadoch, keď daný rozhodovací proces môže mať viac než dva výsledky. Použitie fuzzy logiky má potenciál výrazne zlepšiť pocit z hry, pretože umožňuje modelovať omnoho širšiu a reálnejšiu škálu reakcií počítačom ovládaných postáv. Tým sa zároveň zvyšuje znovuhrateľnosť hier, keďže sa rozširuje rozpätie reakcií a podmienok, na ktoré môže hráč v hrách naraziť, vďaka čomu môže hráč zažiť rôzne výsledky v navzájom podobných situáciách. [2]



Obrázek 2.4: Celý proces práce expertného systému.

Kapitola 3

Unity

Unity je multiplatformový herný engine pre tvorbu 2D a 3D hier a interaktívneho obsahu. Jeho súčasťou je vstavané IDE (*Integrated development environment*, slovensky *vývojové prostredie*) naprogramované v jazyku C#. Samotné jadro Unity enginu je napísané v jazyku C++. Umožnuje vyvíjať hry a interaktívny obsah ako pre stolové platformy (zahŕňajúce PC, Mac a Linux), tak aj pre web (prostredníctvom Unity Web Player), herné konzoly (PS3 a Xbox 360 s plánovanou podporou Xbox One, Wii U, PS4, PS Vita, PS Mobile a Tizen) a mobilné zariadenia (Android, iOS, Windows Phone 8 a Blackberry 10). [16]

Prvá verzia Unity bola predstavená v roku 2005 na konferencii spoločnosti Apple. V tomto čase išlo o engine schopný pracovať a vytvárať programy pre počítače typu Mac. V roku 2010 bola predstavená už tretia verzia enginu Unity. V tomto čase počet vývojárov využívajúcich Unity engine presahoval 200 000 a engine podporoval širokú škálu platforiem. Koncom toho istého roku predstavila spoločnosť Unity Technologies systém *Unity Asset Store*, ktorý predstavuje trh, kde vývojári pracujúci s Unity enginom môžu obchodovať so zdrojmi (angl. *assets*), či už ide o 3D modely, animácie, rozšírenia IDE, skripty, textúry alebo iné položky. V súčasnosti je Unity engine dostupný vo verzii 4.3 a na Game Developers Conference 2014 bola ohlásená verzia 5. [15] Počet registrovaných vývojárov využívajúcich Unity engine sa aktuálne pohybuje na 2,5 miliónoch.

Unity engine ponúka vývojárom dve licencie. Na rozdiel od Unity Free, ktorá je zadarmo, pokiaľ používateľ nie je komerčná organizácia s ročným obratom presahujúcim $ 100 000, prípadne vzdelávacia, nezisková alebo vládna organizácia s ročným rozpočtom presahujúcim $ 100 000, je licencia Unity Pro dostupná za cenu $ 1 500. [13] Unity Pro verzia ponúka oproti Free verzii podporu LOD (*level of detail*, slovensky *úroveň detailnosti*), niektoré dodatočné schopnosti systému *Mecanim*, podporu 3D textúr, výpočet tieňov pre bodové a kužeľové svetlá v reálnom čase, svetelné sondy a mnohé ďalšie vymoženosti.

Od verzie 3.x ponúka Unity *Mono Develop*, vývojové prostredie umožnujúce písať skripty v jazykoch C#, JavaScript a Boo. Unity taktiež poskytuje manuál pozostávajúci z viacerých častí: *sprievodca používateľa* (angl. *user guide*), *často kladené otázky* (angl. *FAQ – frequently asked questions*) a *pokročilé* (angl. *advanced*). Sprievodca používateľa obsahuje úvod do vývojového prostredia Unity, prehľad jednotlivých súčastí a základné návody pre vytvorenie interaktívnej scény. Sekcia „často kladené otázky“ obsahuje zoznam odpovedí na často kladené otázky súvisiace s jednoduchými úkonmi v Unity. Posledná časť „pokročilé“ sa zaoberá zložitejšími témami, ako napríklad optimalizáciou, shadermi alebo nasadzovaním vytvorených scén. Okrem toho ponúka Unity aj referenčný manuál, ktorý poskytuje informácie o konkrétnych súčastiach Unity enginu, a skriptovací referenčný manuál, ktorý poskytuje detailné informácie ohľadom *API* (*application programming interface*, slovensky *rozhra-*

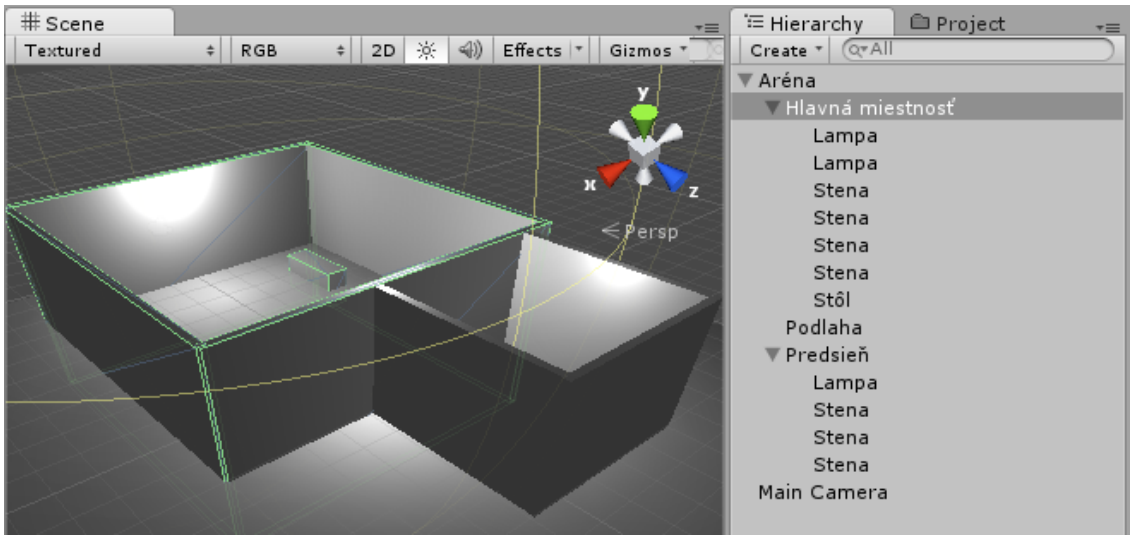
nie pre programovanie aplikácií) pre skriptovanie v Unity. Tento manuál obsahuje opis tried a metód dostupných v Unity API spoločne s praktickými ukážkami v jazykoch C#, JavaScript a Boo.

3.1 Editor Unity

Majoritná časť tvorby hry v Unity engine prebieha prostredníctvom Unity editoru. Tento vstavaný editor poskytuje súhrnný prístup ku zdrojom, rovnako ako pohľad na vytváranú scénu a zoznam všetkých herných objektov a ich vlastností. Takisto umožňuje prístup ku všetkým vývojovým nástrojom, ako napríklad k tvorbe materiálov, spracovaniu svetiel alebo úprave animácií.

Hry v Unity sú postavené na princípe *scén*. Jeden projekt obsahuje niekoľko scén, pričom pod pojmom *scéna* rozumieme uzavretý celok oddelený od ostatných scén, ktorý obsahuje konkrétne herné objekty, nastavenia vykresľovania, nastavenia svetiel a podobne. Prakticky si pod scénou môžeme predstaviť časť herného sveta, ktorá je celá aktuálne načítaná, napríklad jeden level akčnej hry. Jediné, čo sa medzi scénami prenáša a čo je prístupné v rámci celého projektu, sú dostupné zdroje, ako napríklad modely, skripty, textúry a podobne.

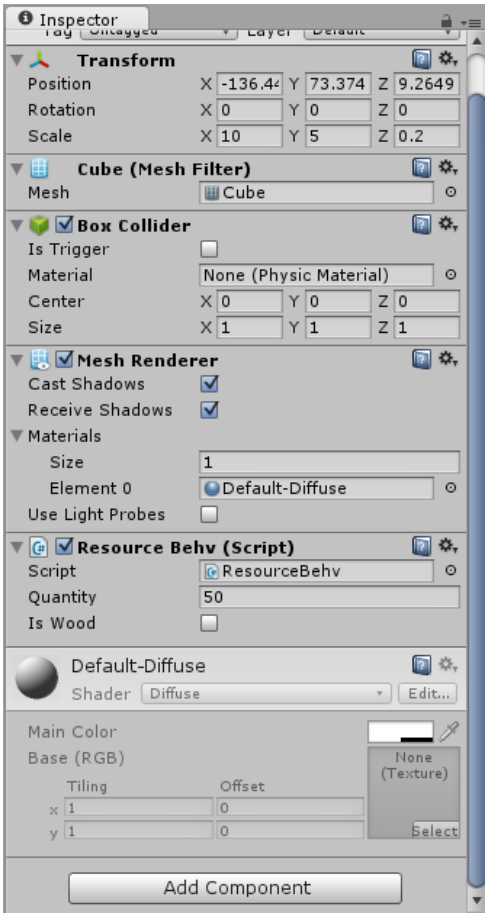
Scéna je vybudovaná na základe *herných objektov*. Je nutné podotknúť, že nejde o *objekty* v ponímaní objektovo orientovaného programovania, ale o predstaviteľov objektov reálneho sveta, napríklad „stôl“, „hlava“ alebo „guľa“. Tieto objekty predstavujú každú časť scény, či už ide o viditeľné záležitosti ako steny, postavy, predmety a svetlá, alebo o neviditeľné súčasti, takzvané *prázdne herné objekty*. Všetky herné objekty je možné do seba hierarchicky zanárať a vytvárať tak logické celky poskladané z menších častí. Typickým príkladom môže byť prázdny herný objekt „aréna“, ktorý sa skladá napríklad z prázdnych herných objektov „predsieň“ a „hlavná miestnosť“, ktoré už pozostávajú z herných objektov reprezentujúcich steny, dvere, okná a podobne. To znázorňuje obrázok 3.1.



Obrázek 3.1: Hierarchia scény v Unity.

Herné objekty obsahujú *komponenty*, ktoré predstavujú určité vlastnosti objektu. Každý herný objekt v scéne musí obsahovať minimálne jeden komponent, a tým je komponent „Transform“, ktorý obsahuje informácie o polohe, rotácii a veľkosti daného objektu v rámci

scény. Komponenty predstavujú jediný spôsob manipulácie s hernými objektami, čo sa týka ich vzhľadu, polohy a správania. Komponenty reprezentujú každý aspekt existencie herného objektu: od umiestnenia objektu cez jeho animácie, polygonovú mriežku, zdroje zvukov, súčasti zodpovedné za fyzikálne správanie, svetelné vlastnosti, detekciu kolízií a mnohé ďalšie. Ukážka komponentov je na obrázku 3.2.



Obrázek 3.2: Komponenty herného objektu.

Jedným z najvýznamnejších komponentov je komponent „Script“. Ten nám umožňuje prideliť danému objektu neobmedzený počet skriptov v nami zvolenom jazyku, a tým nám poskytuje veľmi široké možnosti úprav správania herného objektu počas behu scény. V skriptoch môžeme totiž upravovať a dokonca pridávať a odoberať jednotlivé komponenty objektu a rovnako aj pristupovať k cudzím objektom a ich komponentom. Práve tu máme možnosť naprogramovať objektom správanie a umelú inteligenciu a vytvoriť tak pre naše objekty ich vnímanie scény a spôsoby interakcie s ňou. Dôležitou pomôckou pri testovaní je možnosť upravovať premenné skriptu počas behu scény, rovnako ako aj možnosť scénu kedykoľvek pozastaviť a upraviť ktorýkoľvek komponent objektu. Editor takisto umožňuje priamy prístup na Unity Asset Store, takže nemusíme opúšťať vývojové prostredie, ale môžeme potrebné zdroje vyhľadať a importovať do projektu priamo cez editor.

Základný Unity editor poskytuje intuitívny prístup ku všetkým aspektom scény, v prípade potreby je však dodatočne rozšíriteľný prostredníctvom *zásuvných modulov* (angl. *plugin*).

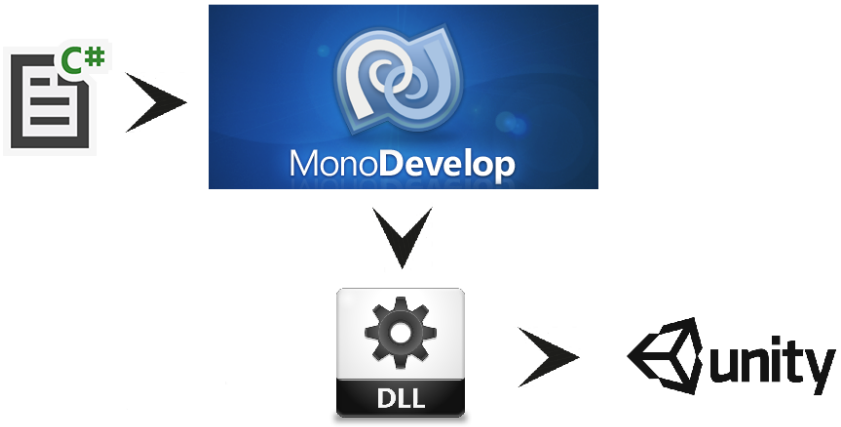
3.2 Zásuvné moduly pre Unity

V prvom rade je nutné rozlíšiť, na čo sa vlastne vzťahuje pojem Unity plugin, keďže v tomto vznikajú rozpory a nezrovnalosti.

Zásuvné moduly pre Unity sú knižnice natívneho kódu napísané v C, C++, Objective-C a podobne. Zásuvné moduly umožňujú v rámci herného kódu, ktorý je napísaný v JavaScripte, C# alebo Boo, volať funkcie z týchto knižníc. Pre stolové platformy platí, že zásuvné moduly sú vyhradené len pre Pro licenciu Unity enginu. [12]

Pojem „zásuvný modul“ sa však často používa aj na označenie Mono DLL knižníc využívaných v Unity. V tomto prípade ide o dynamicky spájanú knižnicu, ktorá je výsledkom kompilácie C# kódu pomocou externého prekladača. Takáto knižnica môže byť následne pridaná do Unity projektu a môže byť použitá, akoby išlo o klasický skript. Dôvodom pre použitie knižnice namiesto klasického skriptu môže byť napríklad fakt, že sme sa ku knižnici dostali z externého zdroja (napríklad sme ju našli na Unity Asset Store). Dôvodom pre samotné vytvorenie knižnice namiesto vytvorenia klasického skriptu môže byť, že sa chceme podeliť o vytvorený Unity kód s inými vývojármi, ale nechceme im umožniť vidieť alebo upravovať zdrojový kód. [14]

Vzhľadom na fakt, že táto bakalárska práca pracuje s Free licenciou Unity enginu, nemáme možnosť využívať zásuvné moduly v ich pravom zmysle. V dôsledku toho sa táto práca zaoberá návrhom a implementáciou DLL v jazyku C#, ktorá bude použiteľná v Unity. Z dôvodu jednoduchosti však budeme naďalej používať aj pre túto C# knižnicu označenie „plugin“.

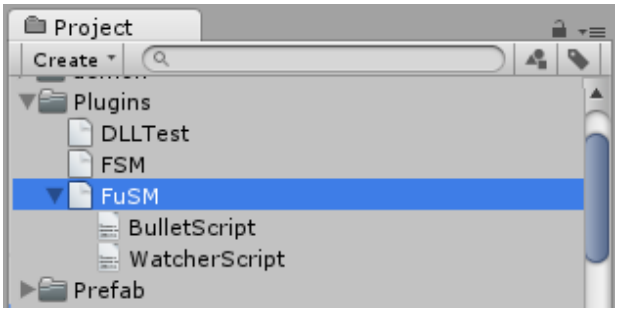


Obrázek 3.3: Postup tvorby DLL pre Unity.

Obrázok 3.3 znázorňuje postup tvorby zásuvného modulu pre Unity. Základom je kód v jazyku C#, v ktorom môžeme využívať triedy prístupné cez Unity API. Tento kód následne preložíme pomocou vhodného prekladača. Je nutné poznamenať, že nie každý prekladač schopný produkovať .NET kód musí byť kompatibilný s Unity.

V prípade, že náš C# kód využíva Unity API, je nutné odpovedajúce Unity DLL sprístupniť prekladaču. Oficiálne podporovanými IDE, ktoré sú schopné generovať .NET knižnice a spolupracovať s Unity, sú MonoDevelop a Visual Studio. Konkrétny postup závisí od prekladača a nebudeme ho v tejto práci uvádzať. Výslednú DLL je následne možné pridať do Unity projektu rovnako ako ktorýkoľvek iný zdroj, napríklad skopírova-

ním do priečinka „Assets“.



Obrázek 3.4: Práca s DLL v Unity.

Na obrázku 3.4 vidíme zobrazenie DLL v editore Unity. Túto knižnicu je následne možné využiť v skriptoch Unity rovnakým spôsobom, ako by sme využívali napríklad Unity API. Pokiaľ vytvorená knižnica obsahuje triedy dediace od triedy *MonoBehaviour*, editor Unity nám ich zobrazí ako hierarchicky zanorené prvky našej DLL. To vidno na obrázku 3.4, kde knižnica „FuSM“ obsahuje dve triedy dediace od triedy *MonoBehaviour*, a to triedy „BulletScript“ a „WatcherScript“. Tieto triedy je možné jednoduchým *drag-and-drop* systémom priradiť herným objektom rovnakým spôsobom ako klasické skripty.

Ako sme ukázali, engine Unity poskytuje veľmi príjemné podmienky pre vývoj hier. Licenčná politika umožňuje malým, začínajúcim spoločnostiam a vývojárom využívať zdarma mocný nástroj schopný tvorby vizuálne príťažlivých, multiplatformových a detailne prepracovaných hier a interaktívnych scén. Hoci ponúka Free licencia obmedzené možnosti oproti platenej Pro licencii, tieto možnosti stále bohato postačujú na vytváranie kvalitných hier. Unity editor navyše poskytuje jednoduché, prehľadné a intuitívne rozhranie vhodné pre začínajúcich vývojárov s množstvom pomôcok, ktoré zjednodušujú a spríjemňujú proces vývoja, ako aj jednoduchú možnosť rozširovania IDE a vytvárania zásuvných modulov. Aj toto sú dôvody, prečo sme si pre túto prácu zvolili práve engine spoločnosti Unity Technologies.

Kapitola 4

Návrh a implementácia

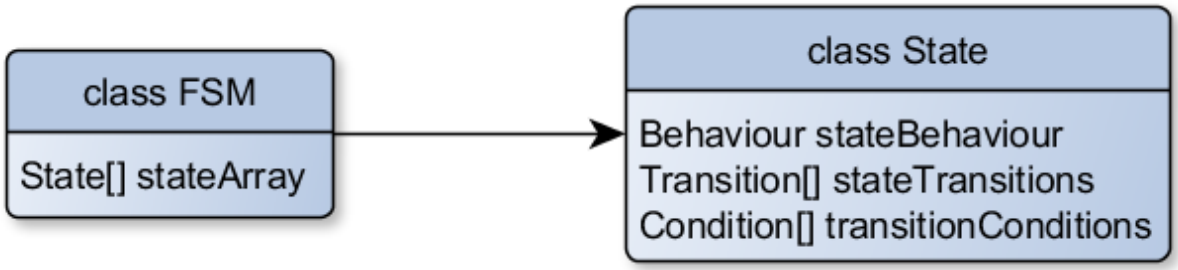
V tejto kapitole sa budeme venovať samotnému návrhu a implementácii nášho zásuvného modulu. Prvá časť, zaoberajúca sa návrhom, sleduje vývoj nápadov a myšlienok vznikajúcich pri tvorbe návrhu, problémy, ktoré z jednotlivých nápadov vyplynuli, a riešenia, ktoré boli v týchto prípadoch zvolené. Návrh sa samozrejme dynamicky menil v závislosti od zistených problémov a nedostatkov, ktoré vyplynuli až v priebehu implementácie. Druhá, implementačná časť opisuje konkrétne aspekty implementácie práce, použité postupy a využité technológie. Táto časť obsahuje aj schematické obrázky znázorňujúce štruktúru a fungovanie implementovaného systému, ako aj konkrétne príklady princípov funkčnosti systému.

4.1 Návrh

Prvou fázou projektu bolo vytvorenie *konečného stavového automatu* (angl. *FSM – finite state machine*) v podobe zásuvného modulu do Unity, ktorý mal byť následne upravený do podoby *fuzzy stavového automatu* (angl. FuSM - *fuzzy state machine*).

4.1.1 Konečný stavový automat

Počiatočná podoba nášej knižnice vyzerala nasledovne:



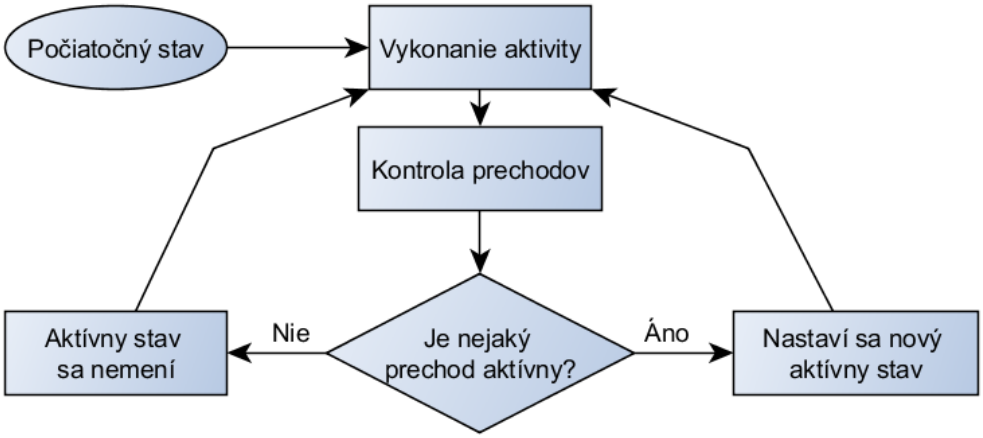
Obrázek 4.1: Prvotný návrh FSM.

Mala existovať trieda *FSM*, ktorá by obsahovala pole stavov. Stav by reprezentovala trieda *State*, ktorá by mala definované svoje správanie (triedou *Behaviour*), prechody do iných stavov (triedou *Transition*) a podmienky týchto prechodov (triedou *Condition*).

Z tohto návrhu následne vyplynula nutnosť definovať triedy *Behaviour*, *Transition* a *Condition*. Pod správaním stavu by sa dalo predstaviť konkrétne pevne definované správanie, ktoré by bolo možné modifikovať použitím parametrov. Príkladom takéhoto správania by mohlo byť napríklad *nasledovanie cieľa*, *hliadkovanie* alebo *presun na miesto*. Parametrami modifikujúcimi správanie by v týchto príkladoch mohla byť poloha cieľa, smerové body alebo rýchlosť pohybu. Pod podmienkami by sa dala predstaviť *vzdialenosť k cieľu*, *kolízia s objektom* alebo napríklad *časovaná podmienka*.

Z praktického hľadiska by išlo o pevný počet konkrétne stanovených správaní, resp. podmienok, z ktorých by si používateľ mohol vybrať a vybrané správanie (resp. podmienku) modifikovať pomocou parametrov.

Konečná myšlienka automatu bola nasledovná: trieda *FSM* musí obsahovať zoznam stavov a odkaz na aktuálny aktívny stav. Automat sa musí vždy nachádzať práve v jednom stave, preto je nutná inicializačná funkcia, ktorá automat uvedie do počiatočného stavu. Jadro funkcie automatu bude spočívať v dvoch aktivitách, a to vo vykonaní aktivity prislúchajúcej aktívnemu stavu a následne kontrole prechodov aktívneho stavu a prípadnému presunu aktivity do nového stavu. Toto znázorňuje obrázok 4.2.



Obrázek 4.2: Princíp práce FSM.

Používateľ by si z týchto pevne daných stavebných blokov poskladal samotný stavový automat opisujúci správanie jeho herného objektu. Vytvoril by v skripte objekt triedy *FSM*, následne by vytvoril požadované stavy ako objekty triedy *State* a priradil im správania prostredníctvom triedy *Behaviour*. Takisto by musel vytvoriť prechody pomocou triedy *Transition*, prechodom nastaviť podmienky pomocou triedy *Condition* a definovať, ktorý prechod smeruje do ktorého stavu. Následne by už len stačilo pridať jednotlivé pripravené stavy do automatu a nastaviť počiatočný stav.

Implementácia tohto automatu je zdokumentovaná v časti 4.2.

4.1.2 Fuzzy stavový automat

Po úspešnej implementácii a otestovaní konečného stavového automatu bolo nutné upraviť jeho návrh do podoby fuzzy stavového automatu. Hlavné zmeny mali spočívať v pridaní podpory viacerých aktuálne aktívnych stavov a podpory miery členstva jednotlivých stavov.

V tom čase vyplynul problém. Predstavme si napríklad dva rôzne stavy, jeden so správaním typu *hliadkovanie* a druhý so správaním typu *nasledovanie cieľa*. Každý z týchto

dvoch stavov má mieru členstva 0,5, pričom „hliadkovanie“ má rýchlosť 2 a „nasledovanie“ má rýchlosť 4. Akou konečnou rýchlosťou sa bude herný objekt pohybovať?

Prvým riešením bolo spojenie rýchlostí daných stavov pri použití mier členstva ako váh. Zoberme si príklad, ktorý sme predstavili o pár riadkov vyššie. V tomto prípade by sa na rýchlosť 2 stavu „hliadkovanie“ aplikovala hodnota 0,5, ktorá je mierou členstva tohto stavu. Na rýchlosť 4 stavu „nasledovanie“ by bola taktiež aplikovaná hodnota 0,5. Výsledné rýchlosti, teda 1 a 2, by sa sčítali na hodnotu 3 a táto hodnota by predstavovala konečnú rýchlosť pohybu herného objektu.

Neskôr sa však ukázalo, že toto riešenie nie je postačujúce, pretože nie všetky aktívne stavy musia mať nutne rovnaké premenné. Ako príklad si môžeme predstaviť, že máme v automate aktívny stav typu *hliadkovanie* a stav typu *streľba*, obidva s mierou členstva 0,5. Stav typu *streľba* neobsahuje premennú „rýchlosť“ vzhľadom na to, že nejde o pohybový stav. Priemerovať rýchlosť len jedného stavu nedáva v takejto situácii zmysel. Upustili sme teda od tohto riešenia a situáciu vyriešili tak, že sa pri vykonávacej časti kódu vykonajú postupne všetky aktívne stavy bez vzájomnej interakcie.

Prakticky to napríklad znamená, že sa najprv vykoná pohyb spôsobený stavom „hliadkovanie“, následne sa vykoná pohyb spôsobený stavom „nasledovanie cieľa“, a potom sa vykoná streľba spôsobená stavom „streľba“ za predpokladu, že práve tieto tri stavy sú momentálne aktívne. Automat teda nebude kontrolovať vzájomné súvislosti medzi stavmi a ich aktivitami, ale jednoducho ich vykoná jednu za druhou. Toto by mohlo spôsobiť problém napríklad v prípade, že by sa dva stavy pokúšali pohybovať v tom istom smere. V takejto situácii by došlo k dvom posunom v danom smere a javilo by sa teda, že sa herný objekt pohybuje dvojnásobnou rýchlosťou.

Tento problém by však zmizol vo chvíli, keď by sme využívali hodnotu miery príslušnosti stavu na ovplyvnenie parametrov jeho aktivity. Prakticky by sa to prejavilo tak, že každé definované správanie by malo určené určité premenné, ktoré by mala ovplyvňovať hodnota miery príslušnosti. Nazvime si tieto premenné *fuzzy faktory*. Typickým kandidátom na fuzzy faktor je napríklad rýchlosť pohybu pri stave „hliadkovanie“. Používateľ by danému stavu nastavil hraničné hodnoty rýchlosti hliadkovania a automat by na základe hodnoty miery príslušnosti nastavoval aktuálnu rýchlosť na hodnotu v týchto hraniciach. Predstavme si, že stav „hliadkovanie“ má ako minimálnu rýchlosť nastavenú hodnotu 1 a ako maximálnu rýchlosť hodnotu 5. Nie je podstatné, o aké jednotky ide, môžeme ich chápať napríklad ako metre za sekundu. Pokiaľ by bol automat v stave „hliadkovanie“ s mierou 0,5, nastavil by aktuálnu rýchlosť hliadkovania na hodnotu 3 použitím vzorca

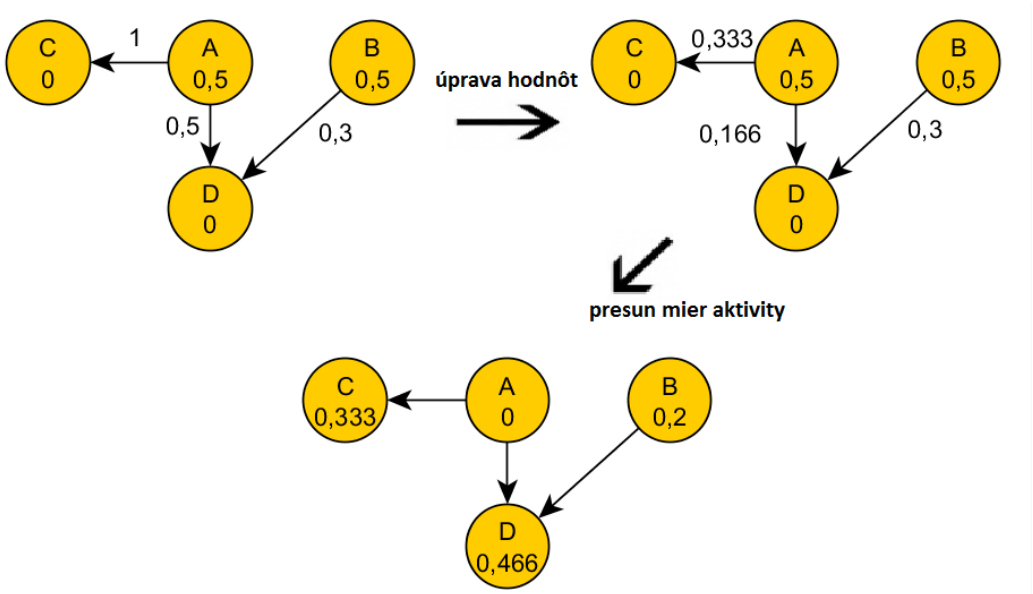
$$v = min + (max - min)/f,$$

kde v je výsledná rýchlosť, min a max predstavujú hranice rýchlosti a f je hodnota miery členstva stavu.

Ďalšou záležitosťou, ktorú sme riešili, bola realizácia fuzzy množín pri fuzzy podmienkach automatu. Pôvodný návrh počítal s tým, že každá podmienka bude mať päť fuzzy množín, a to „extremely low“, „low“, „medium“, „high“ a „extremely high“. Používateľ by si pre každú množinu mohol nastaviť konkrétne hodnoty, napríklad by určil, že „low“ bude v konkrétnom prípade znamenať hodnotu nižšiu než 3 jednotky. Tento koncept sa neskôr zmenil, pretrvával však v návrhu relatívne dlho.

Následne sa vynorila otázka prenosu aktivity medzi stavmi automatu. Prvý návrh pracoval na nasledujúcom princípe: hodnoty platnosti prechodov vychádzajúcich z jedného stavu by bolo nutné skontrolovať a pokiaľ by spoločne presahovali hodnotu miery aktivity rodičovského stavu, boli by znížené tak, aby zodpovedali maximálne tejto hodnote. Výsledné

hodnoty platnosti prechodov by sa odčítali od rodičovského stavu a pričítali k stavom cieľovým. V prípade, že by do jedného stavu prechádzalo viacero aktívnych prechodov, miera členstva by sa v cieľovom stave sčítala. Tento návrh je znázornený na obrázku 4.3.

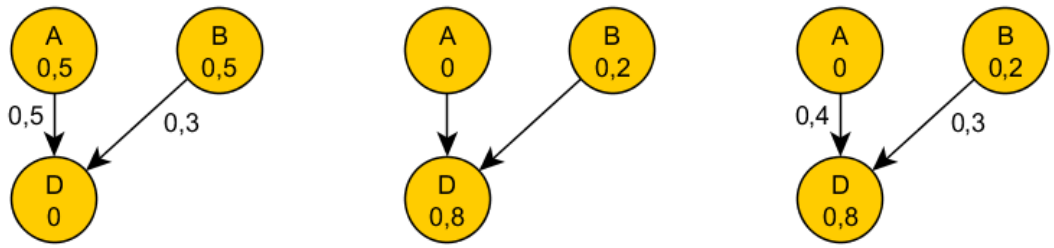


Obrázek 4.3: Princíp presunu aktivity FuSM.

Vidno, že hoci stav A oplýva mierou členstva 0,5, suma platností jeho výstupných prechodov má hodnotu 1,5, a teda sa bude musieť upraviť. Výsledok úprav je znázornený v druhej časti schémy, kde boli platnosti prechodov znížené na takú úroveň, aby dosahovali spoločne maximálne 0,5, a to v zodpovedajúcom pomere. Zároveň vidíme, že pri prechode zo stavu B do stavu D nebolo nutné hodnotu upravovať, pretože neprekračovala mieru aktivity rodičovského stavu B. V tretej časti schémy vidíme výsledok presunu mier aktivity medzi stavmi. Vidno, že hodnoty vstupujúce do stavu D sa sčítali. Takisto vidno, že týmto spôsobom sa zachovala celková suma mier aktivity v automate na požadovanej hodnote 1.

Po implementácii a otestovaní tohto návrhu sa však ukázalo, že je nepoužiteľný. Predstavme si jednoduchý príklad: náš automat obsahuje dva stavy, a to „nerobím nič“ a „strieľam“. Prvý menovaný stav má na počiatku mieru členstva 1, druhý 0. Podmienka na prechode medzi týmito stavmi má podobu vzdialenosti k cieľu a je platná na 0,5 (princíp vyhodnocovania fuzzy množín bol vysvetlený v kapitole 2). V takomto prípade sa v prvom snímku (angl. *frame*) presunie zo stavu „nerobím nič“ do stavu „strieľam“ miera aktivity s hodnotou 0,5, výsledné hodnoty mier aktivity v stavoch budú 0,5 v „nerobím nič“ a 0,5 v „strieľam“. Vzhľadom na to, že sa za čas jedného snímku nestihnú v scéne udiať výrazné zmeny, bude aj v ďalšom snímku platiť naša podmienka zhruba na 0,5. V dôsledku toho sa opäť presunie hodnota 0,5 zo stavu „nerobím nič“ do stavu „strieľam“ a tým pádom sa v priebehu dvoch snímkov presunie automat plne z jedného stavu do druhého, hoci podmienka na ich prechode je splnená len na mieru 0,5.

Jedným z možných riešení tohto problému by bolo obmedzenie maximálnej hodnoty cieľového stavu prechodu na hodnotu platnosti prechodu. To však so sebou prinieslo ďalší problém: kam presunúť mieru členstva pri poklese platnosti podmienky? Pri jedinom vstupnom prechode je riešenie triviálne, predstavme si však situáciu znázornenú na obrázku 4.4.



Obrázek 4.4: Príklad problému s návratom.

Stav D obdrží od stavu A mieru 0,5 a od stavu B mieru 0,3, čím sa dostane na mieru 0,8. V poslednej časti schémy poklesne platnosť podmienky na hrane medzi A a D na úroveň 0,4. Tu nastávaju otázky: kam presunúť aktivitu? Koľko aktivity presunúť? Na riešenie týchto problémov vznikol nový koncept automatu, ktorý fungoval na značne inom princípe.

Išlo by o automat, ktorý by pri výpočte mier aktivity jednotlivých stavov ignoroval aktuálne aktívne stavy. Na začiatku každého snímku by sa začínalo s prázdnou množinou aktívnych stavov a voľnou aktivitou v miere 1. Následne by sa vyhodnotili všetky fuzzy podmienky, ktoré danému automatu prislúchajú. Podmienky by mali tvar

IF splnená podmienka prechodu THEN cieľový stav prechodu bude aktívny.

Ako príklad môžeme uviesť nasledujúci automat: existujú stavy „nerobím nič“ a „strieľam“. Existujú takéto podmienky:

IF nepriateľ je blízko THEN prejdi do stavu „strieľam“,

IF nepriateľ nie je blízko THEN prejdi do stavu „nerobím nič“.

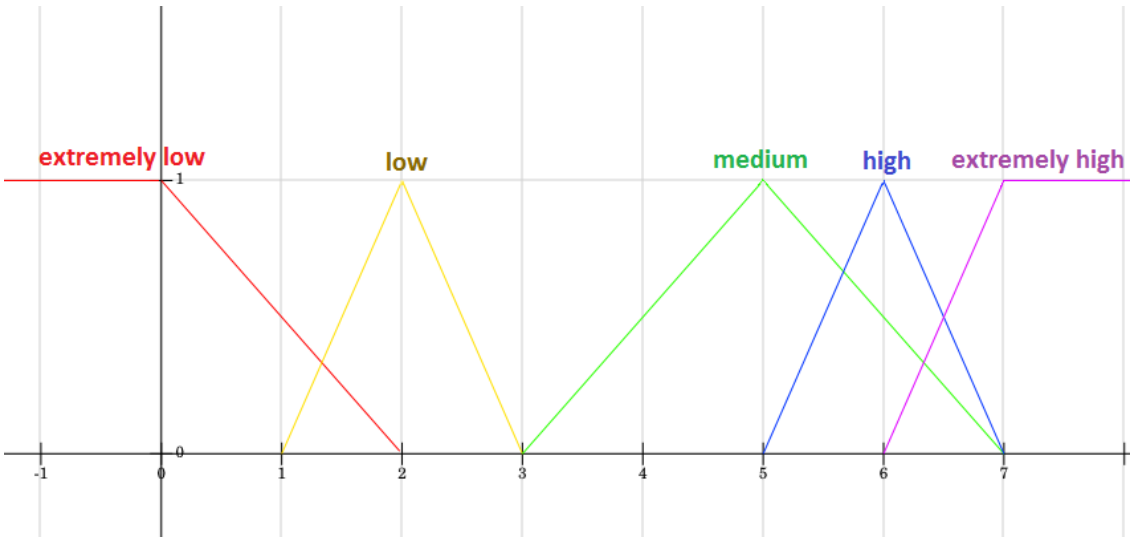
Povedzme, že podmienka „nepriateľ je blízko“ sa vyhodnotí na mieru 0,2. V tomto prípade sa aktivita presunie do stavu „strieľam“ s hodnotou 0,2 a do stavu „nerobím nič“ s hodnotou 0,8. Pokiaľ v ďalšom snímku napríklad poklesne platnosť podmienky na 0,1, automat sa prepočíta nanovo a rozdelí aktivitu v pomere 0,9 pre „nerobím nič“ ku 0,1 pre „strieľam“. V tejto koncepcii by z princípu nebolo nutné riešiť spätné presuny aktivity pri poklese platnosti prechodov, pretože by sa automat v každom snímku vyhodnocoval nanovo.

Uvedomili sme si však, že táto koncepcia sa príliš vzďaľuje od koncepcie stavového automatu, keďže neberie do úvahy aktuálne aktívne stavy automatu. Po konzultácii s pánom Ing. Kajanom sme sa zhodli, že tento princíp nezodpovedá zadaniu práce, ktorá požaduje využitie fuzzy stavového stroja, a bude lepšie riešiť automat spôsobom bližším konceptu stavového automatu. Takisto nám bolo odporučené vyriešiť problém spätného návratu aktivity tak, že si prechody budú pamätať bývalú hodnotu platnosti podmienky a pri jej poklese dôjde k spätnému presunu aktivity.

Nový návrh teda počítal s tým, že si jednotlivé stavy budú kontrolovať nielen svoje výstupné prechody, ale aj svoje vstupné prechody, a v prípade poklesu platnosti podmienky na vstupnom prechode vrátia časť svojej aktivity rodičovskému stavu prechodu. Otázkou však stále zostávalo, koľko aktivity presúvať a akými pravidlami sa má tento presun riadiť.

Medzitým sa v priebehu návrhu často menila aj samotná funkcia presunu aktivity medzi stavmi, presnejšie povedané funkcia, ktorá by určovala, koľko aktivity sa má presunúť. Niektoré návrhy počítali s mierou členstva rodičovských stavov a platnosťou podmienok prechodov, či už išlo o ich násobok, priemer alebo iné matematické funkcie, zatiaľ čo iné experimentovali len s využitím platností podmienok, prípadne rozdielov medzi bývalými a aktuálnymi mierami platností podmienok.

Po testovaní implementácie vyplynula zbytočná komplikovanosť systému zadávania fuzzy podmienok. Do určitej doby fungoval tento systém tak, že každá fuzzy podmienka mala päť množín, od „extremely low“ až po „extremely high“. V praxi teda užívateľ musel pri každej vytvorenej podmienke zadať jej priebeh. To názorne ukazuje obrázok 4.5.



Obrázek 4.5: Príklad vytvorenia podmienky.

Používateľ by teda musel nastaviť pre všetkých päť množín ich stred a rozptyl, na obrázku vidíme napríklad zelenú množinu „medium“ so stredom 5 a rozptylom 2. Vyplynulo však, že v našom automate sa vždy využíva len jedna množina pre podmienku, a teda je zbytočné zadávať všetky. V novom návrhu teda používateľ zadáva jediný stred a jediný rozptyl, ako aj informáciu, či ide o *ľavý*, *stredný* alebo *pravý* typ množiny.

Ľavý typ znamená, že naľavo od stredu množiny zostáva hodnota množiny na 1. Príkladom tohto typu je na obrázku 4.5 červená množina. Analogicky *pravý* typ znamená hodnotu množiny 1 pri vstupných hodnotách vpravo od stredu (na obrázku fialová množina). Typ stredný má trojuholníkový priebeh, teda hodnota množiny klesá naľavo aj napravo od stredu a hodnotu 1 nadobúda množina len vo svojom strede. Na obrázku je *stredného* typu žltá, zelená a modrá množina.

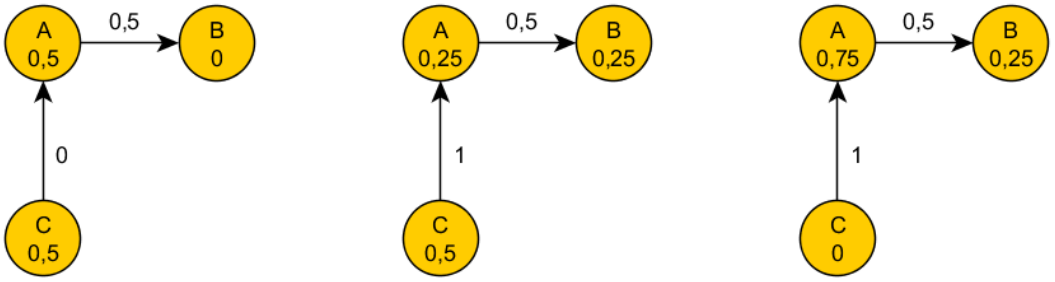
Vo vtedajšom návrhu existovali pevne stanovené podmienky, napr. „vzdialenosť od cieľa“ alebo „časovaná podmienka“, to však neumožňovalo dostatočnú flexibilitu systému. Do návrhu bola teda pridaná generická fuzzy podmienka, ktorá by od používateľa dostala konkrétnu premennú, s ktorou má pracovať. Rozdiel voči pevným podmienkam by bol, že zatiaľ čo pevnej podmienke by zadal používateľ ako parameter napríklad herný objekt, voči ktorému by následne automat automaticky zisťoval vzdialenosť, v generickej podmienke by úloha získať konkrétne číslo pripadala užívateľovi a automatu by musel pravidelne poskytovať aktuálne čísla. Toto síce presúva réžiu mimo automat, na druhú stranu však poskytuje možnosť

vytvoriť podmienku na sledovanie ľubovolnej vlastnosti scény.

Ako príklad si uveďme situáciu, kde používateľ chce ako podmienku prechodu nastaviť plnosť batohu nejakej postavy. Žiadna zo vstavaných podmienok takúto špecifickú situáciu nepokrýva, preto si používateľ vytvorí generickú podmienku, ktorej zadá fuzzy množinu, ako bolo vysvetlené o pár riadkov vyššie. Následne je v jeho réžii, aby pravidelne dodával automatu informáciu o aktuálnej plnosti batohu, a automat sa už sám postará o vyhodnotenie tejto podmienky na základe dodaných informácií.

V priebehu testovania sa ukázalo, že možnosť spätného presúvania aktivity pri poklese platnosti podmienky nie je vždy žiadúca. Z tohto dôvodu bola pridaná možnosť každý prechod označiť ako *jednosmerný* alebo *obojsmerný*, pričom spätné návraty sa berú do úvahy len pri obojsmerných prechodoch.

V tom čase pracovalo presúvanie aktivity na princípe merania rozdielu platnosti prechodov medzi snímkami. Pokiaľ by v jednom snímku platnosť prechodu bola nulová, zatiaľ čo v ďalšom už nadobúdala hodnotu 0,5, rozdiel platností by bol 0,5, a preto by sa presunulo 50 % aktivity z rodičovského stavu do nového. Pokial by v ďalšom snímku platnosť podmienky zostávala na 0,5, rozdiel by bol nulový a ďalšia aktivita by sa neprenášala. Pokiaľ by platnosť podmienky poklesla, vznikol by záporný rozdiel a došlo by k spätnému presunu aktivity. Tento systém však nebol postačujúci, ako znázorňuje situácia na obrázku 4.6.

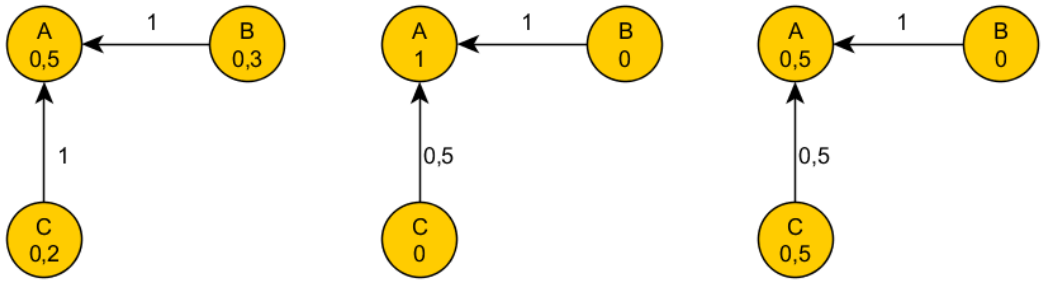


Obrázek 4.6: Problém s princípom presunu.

Povedzme, že na počiatku majú všetky prechody platnosť podmienky 0. V prvej časti schémy narastie platnosť podmienky na prechode medzi stavom A a B na hodnotu 0,5. Rozdiel hodnôt je teda 0,5 a presunie sa 50% aktivity zo stavu A do stavu B, ako znázorňuje druhá časť schémy. V tejto chvíli sa do stavu A presunie aktivita zo stavu C a uvedie tak stav A do miery členstva 0,75. Vzhľadom na to, že podmienka medzi A a B má platnosť 0,5, bolo by logické očakávať, že sa aktivita presunie zo stavu A do B tak, aby reflektovala túto platnosť. Keďže sa však platnosť podmienky nemení, automat nepresúva po danom prechode žiadnu aktivitu. Tento systém nie je teda postačujúci.

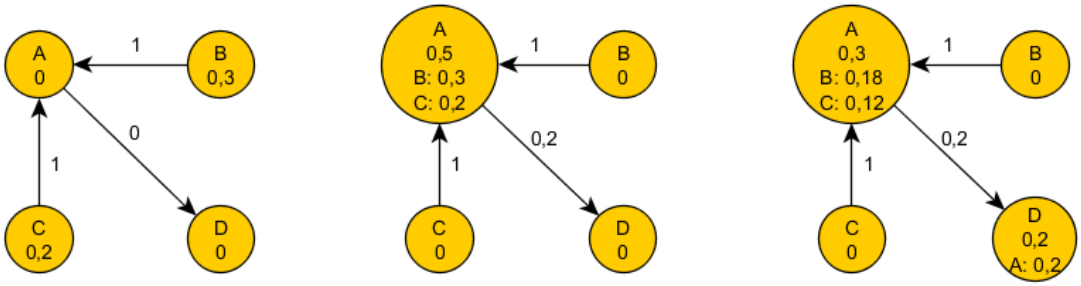
Nový systém, ktorý pretrval až do aktuálnej verzie, pracuje tak, že množstvo aktivity, ktorá sa má po prechode presunúť, je vypočítané ako rozdiel platnosti prechodu a aktuálnej miery členstva stavu. Napríklad pokiaľ stav má mieru členstva 0,5 a vstupný prechod tohto stavu má platnosť 0,6, rozdiel medzi nimi je 0,1 a práve takéto množstvo aktivity by sa malo presunúť z rodičovského do nového stavu. Tento systém počíta aj so spätnými prechodmi, pretože pokiaľ má stav vačšiu mieru členstva než je platnosť prechodu, vypočítaný rozdiel je záporný a automat vykoná spätný presun aktivity. Konkrétne množstvo presunutej aktivity sa samozrejme prepočíta vzhľadom na mieru členstva rodiča, aby nedošlo k situácii, kde sa automat snaží presunúť väčšie množstvo aktivity, než akým daný rodič disponuje.

V súvislosti s týmto systémom musel byť zavedený aj systém *prispievateľov* (angl. *contributors*), ktorý pri každom stave monitoruje, od koho obdržal aké množstvo aktivity. Tento systém je nutný, aby nedošlo k situácii, kde napríklad stav A obdrží aktivitu od stavu B, ale následne pri poklese platnosti podmienky na prechode zo stavu C do stavu B vykoná spätný presun aktivity zo stavu B do stavu C, hoci mu stav C neprispel hodnotou, ktorú stavu C vracia. Pre jednoduchosť je tento príklad znázornený na obrázku 4.7.



Obrázek 4.7: Problém so spätným presunom.

Vidíme, že na počiatku sa platnosť oboch prechodov rovná 1. Presunú sa teda plné aktivity rodičov a stav A nadobudne mieru členstva 1. Predstavme si, že v ďalšej chvíli poklesne platnosť prechodu medzi C a A na hodnotu 0,5. Rozdiel medzi platnosťou prechodu a mierou členstva stavu A je záporných 0,5, preto dôjde k spätnému prechodu v hodnote 0,5 z A do C. Toto správanie je nespravodlivé vzhľadom na to, že stav A obdržal od stavu C len 0,2 miery členstva, a C teda nemá nárok obdržať od A až 0,5. Systém prispievateľov tento problém eliminuje, vytvára však nový problém. Pozrime sa ešte raz na obrázok 4.7 a predstavme si, že by existoval ešte jeden prechod, a to zo stavu A do nového stavu D. Pri splnení podmienky na tomto novom prechode je stav A povinný presunúť aktivitu do stavu D, presúva však aj aktivitu, ktorú obdržal od stavov B a C, a vďaka tomu im ju v prípade potreby už nebude schopný vrátiť. Na vyriešenie tohto problému sme zaviedli systém, kde v takejto situácii, keď stav posúva obdržanú mieru členstva ďalej, zároveň znižuje hodnotu príspevku od všetkých prispievateľov v pomere, akým prispeli. Náš príklad demonštruje obrázok 4.8.



Obrázek 4.8: Riešenie systému prispievateľov.

Vidíme, že v druhej časti schémy obdržal stav A mieru členstva 0,3 od stavu B a 0,2 od stavu C. Následne v tretej časti presúva aktivitu do stavu D, a to s hodnotou 0,2. V tejto chvíli sú znížené hodnoty príspevkov od stavov B a C o hodnotu, ktorá odišla do stavu D, a to v pomere, v akom stavy B a C prispeli. Tento návrh pretrval až do konečnej fázy.

Rovnako ako vyplynula nutnosť vytvoriť generickú podmienku na pokrytie všetkých možných situácií, uvedomili sme si potrebu vzniku generického správania, ktoré postihuje prípady mimo pevne vstavaných modelov správania. Vznikol teda návrh generického správania, ktoré dostane od používateľa ľubovoľný počet premenných, ktoré má modifikovať, spolu s hranicami, v ktorých sa majú premenné pohybovať, a ďalšími potrebnými informáciami. Generické správanie následne počas behu automatu modifikuje tieto premenné na základe miery členstva stavu a používateľ si môže prečítať aktuálne hodnoty premenných a vo vlastnej réžii ich použiť na parametrizovanie vlastného správania herného objektu.

Ako priklad si môžeme predstavit situáciu, kde používateľ chce svojmu hernému objektu priradiť v určitom stave správanie „točím sa“. Predpokladajme, že toto správanie je parametrizované premennou „rýchlosť točenia“. Takéto správanie nie je definované medzi zabudovanými modelmi správania automatu, preto používateľ použije generické správanie, ktorému nastaví, akú premennú má v sebe modifikovať a v akých hraniciach sa hodnota tejto premennej má pohybovať. Následne je na používateľovi, aby pravidelne čítal aktuálnu hodnotu danej premennej a následne ju vo vlastnej réžii použil na úpravu jeho špeciálneho správania „točím sa“.

Finálny návrh má teda podobu fuzzy stavového stroja, kde presúvanie miery členstva medzi stavmi závisí od hodnoty miery členstva aktívnych stavov, ako aj od hodnoty platnosti podmienok jednotlivých prechodov. Prechody existujú v dvoch variantoch, a to jednosmerné prechody a obojsmerné prechody, pričom druhé menované podporujú spätný presun mier členstva na základe systému prispievateľov. Automat má niekoľko pevne vstavaných modelov správania a typov podmienok, avšak obsahuje aj generické správanie a generickú podmienku. Tieto dva prvky obohacujú automat o flexibilitu a umožňujú jeho využitie pre ľubovoľný prípad. Systém funguje tak, že používateľ vytvorí požadované stavy, týmto stavom pridelí modely správania a prechody, pričom prechodom musí priradiť podmienky. Následne používateľ musí automat inicializovať a zariadiť, aby sa automat pravidelne aktualizoval, a ten už vo vlastnej réžii vykonáva všetky nutné operácie. Používateľ má prístup ku všetkým prakticky využiteľným súčastiam automatu a môže tak jednoducho čítať jeho stav.

4.2 Implementácia

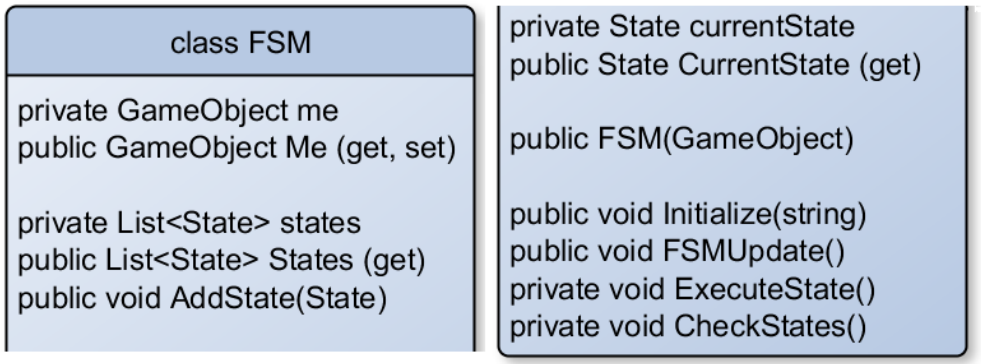
V tejto časti sa zameriame na konkrétne implementačné detaily a postupy použité pri pretváraní návrhu do reálnej podoby. Opíšeme štruktúru jednotlivých častí automatu a niektoré zaujímavé myšlienky, ktoré pri procese implementácie vznikli. Stručne predstavíme podobu konečného stavového automatu, z ktorého neskorší fuzzy stavový automat vychádza, detailnejšie sa však zameriame na samotný fuzzy stavový automat.

4.2.1 Konečný stavový automat

Náš konečný stavový automat je implementovaný triedou *FSM* (obrázok 4.9).

Trieda obsahuje okrem iného odkaz na vlastníka daného automatu, teda na herný objekt, ktorému je tento automat priradený. Metóda *Initialize* na základe názvu stavu uvedie automat do počiatočného stavu. Metódy *ExecuteState* a *CheckStates* sú volané z metódy *FSMUpdate* a sú zodpovedné za vykonanie aktivity aktívneho stavu a kontroly prechodov daného stavu.

Metóda *ExecuteState* obsahuje príkaz *switch*, kde rozozná, o ktorý model správania ide, a na základe toho vykoná špecifickú činnosť. Napríklad pre model správania *patrol* metóda

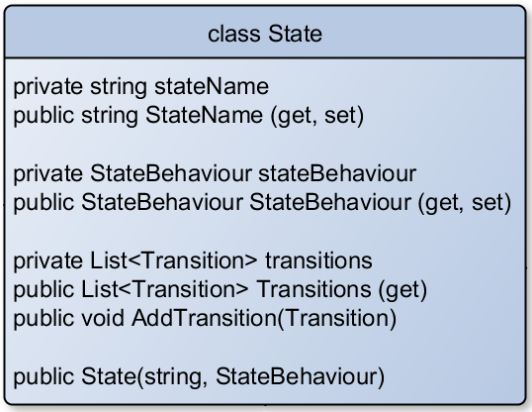


Obrázek 4.9: Trieda FSM.

skontroluje vzdialenosť k aktuálnemu bodu cesty (angl. *waypoint*) a na základe nej buď vyžiada ďalší bod cesty, alebo pohne herným objektom v požadovanom smere s využitím Unity API, konkrétne metódou *transform.LookAt* a priamou úpravou *transform.position* pomocou metódy *Vector3.Lerp*.

Metóda *CheckStates* v cykle prejde cez všetky prechody aktuálneho aktívneho stavu, načíta informácie o danom prechode, konkrétne cieľový stav prechodu, typ podmienky na prechode, typ porovnania podmienky (napríklad *lessThen* alebo *greaterEqual*) a hodnotu, voči ktorej má byť podmienka porovnaná. Opäť sa tu nachádza príkaz *switch*, ktorý rozozná typ podmienky (napríklad *distanceToTarget* alebo *targetVisible*) a na jej základe a na základe konkrétneho typu porovnania vykoná porovnanie a prípadne vykoná zmenu aktívneho stavu. Napríklad pre podmienku typu *distanceToTarget* a typ porovnania *lessThen* porovná metóda aktuálnu vzdialenosť herného objektu od cieľa s cieľovou hodnotou. Pokiaľ je porovnávacia podmienka splnená, nastaví metóda nový aktuálny stav a nastaví príslušný príznak (angl. *flag*). Na konci hlavného príkazu *switch* tento príznak skontroluje a pokiaľ došlo k zmene aktívneho stavu, neprehľadáva už ďalšie prechody stavu, ale ukončí cyklus.

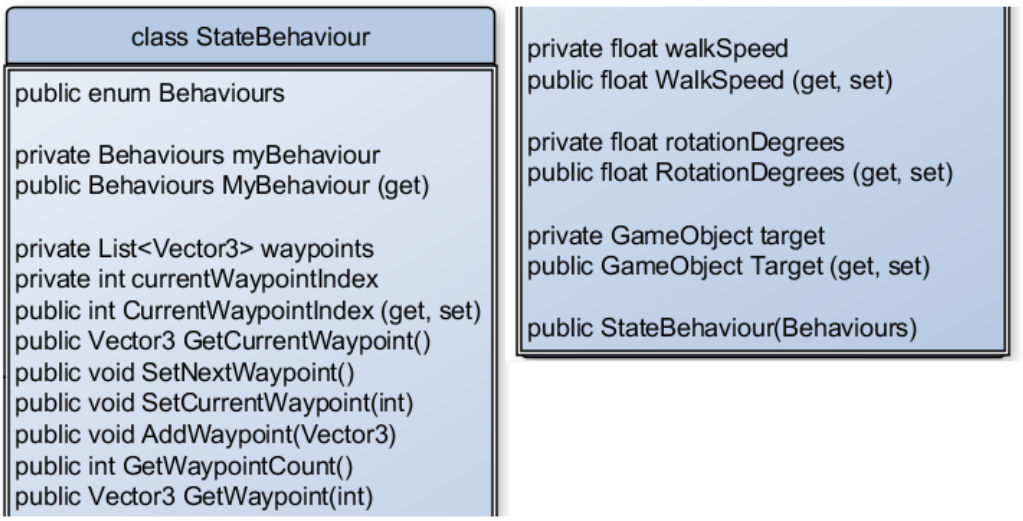
Stavy automatu sú implementované triedou *State* (obrázok 4.10).



Obrázek 4.10: Trieda State vo FSM.

Trieda obsahuje okrem iného odkaz na model správania daného stavu a zoznam prechodov daného stavu.

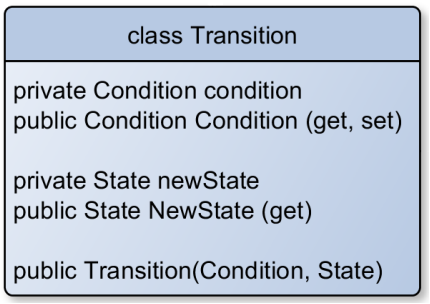
Správanie v stave je implementované triedou *StateBehaviour* (obrázok 4.11).



Obrázek 4.11: Trieda StateBehaviour vo FSM.

Trieda obsahuje *enum* na rôzne modely správania, vybrané správanie pre konkrétny prípad a následne premenné a metódy týkajúce sa konkrétnych modelov správania. Napríklad obsahuje zoznam *waypoints* využitý pri modeli správania *patrol*, nasledovaný premennými a metódami súvisiacimi s modifikáciou tohto zoznamu. Nie všetky premenné musia byť definované, záleží od konkrétneho modelu správania, napríklad model správania *idle* nepotrebuje využívať žiadnu z týchto premenných.

Prechody automatu sú implementované triedou *Transition* (obrázok 4.12).



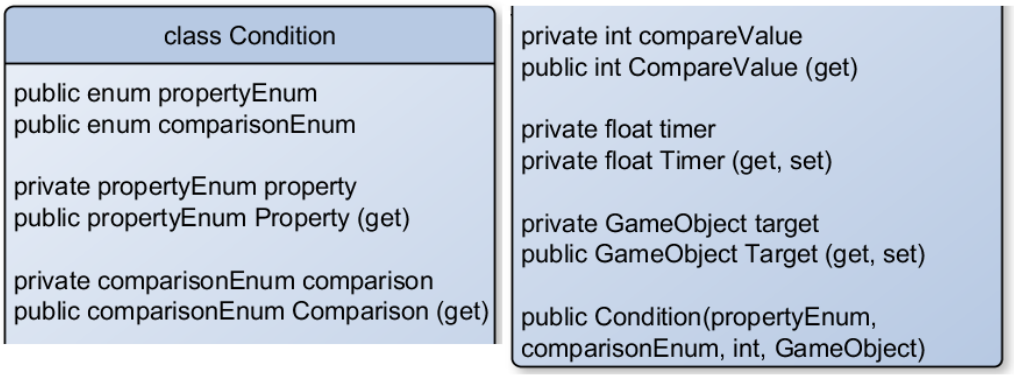
Obrázek 4.12: Trieda Transition vo FSM.

Trieda obsahuje podmienku daného prechodu a odkaz na cieľový stav prechodu.

Podmienky prechodov sú implementované triedou *Condition* (obrázok 4.13).

Trieda obsahuje *enum* na porovnávanú vlastnosť (napríklad *distanceToTarget*) a na porovnávací typ (napríklad *lessThan*). Ďalej obsahuje vlastnosť a porovnávací typ pre konkrétnu podmienku, hodnotu, oproti ktorej bude špecifikovaná vlastnosť porovnávaná, a odkazy na objekty, ktoré sú v niektorých typoch podmienok využívané (napr. *Timer* alebo *Target*). Konštruktor podmienky obsahuje kontroly na správne vyplnenie potrebných hodnôt v závislosti od typu podmienky.

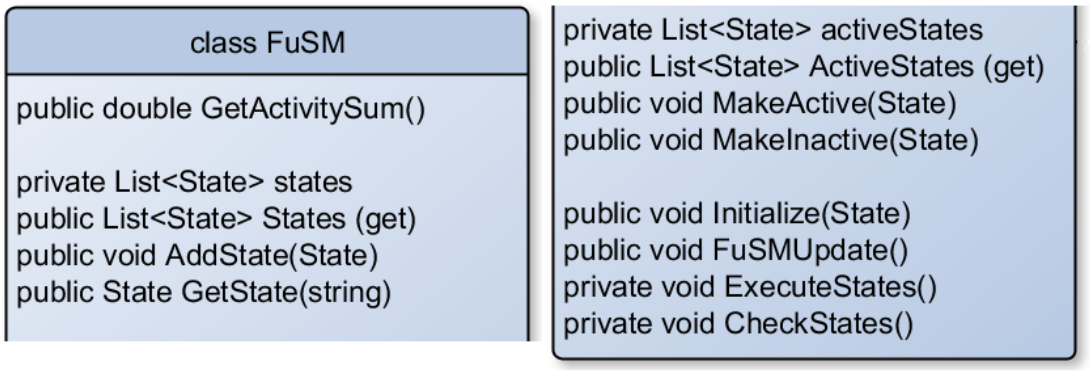
Princíp použitia automatu bol vysvetlený v časti 4.1.



Obrázek 4.13: Trieda Condition vo FSM.

4.2.2 Fuzzy stavový automat

Fuzzy stavový automat je implementovaný triedou *FuSM* (obrázok 4.14).



Obrázek 4.14: Trieda FuSM.

Trieda obsahuje okrem iného zoznam všetkých stavov automatu (*states*), zoznam všetkých aktívnych stavov automatu (*activeStates*) a metódy na úpravy týchto zoznamov. Takisto obsahuje inicializačnú metódu *Initialize*, ktorú je nutné raz zavolať pred spustením aktualizácie automatu.

Inicializačná funkcia jednak nastaví počiatočný stav automatu, ale aj skontroluje, či sú správne nastavené premenné jednotlivých modelov správania stavov automatu. Takisto naplní zoznam vstupných prechodov vo všetkých stavoch a nastaví všetkým prechodom ich rodičovský stav.

Trieda ďalej obsahuje metódu, ktorá vráti aktuálny súčet mier členstva jednotlivých stavov automatu. Toto je neskôr využívané pre kontrolu, či sa súčet mier členstva príliš nevzdialil od požadovanej hodnoty 1.

Metóda *FuSMUpdate* je zodpovedná za aktualizáciu celého automatu a musí byť pravidelne volaná v používateľskom skripte. Táto metóda pozostáva z volania metódy *ExecuteStates* a metódy *CheckStates*. Metóda *ExecuteStates* je jednoduchá, jej činnosťou je cyklický prechod cez všetky aktívne stavy automatu a zavolanie dvoch metód každého aktívneho stavu, a to metódy *ApplyFuzzy*, ktorá upraví fuzzy faktory modelu správania stavu na základe

miery členstva, a metódy *Execute*, ktorá vykoná činnosť prislúchajúcu modelu správania.

Metóda *CheckStates* je zložitejšia a obsahuje hlavné jadro systému automatu. Táto metóda je zodpovedná za vyhodnocovanie podmienok, výpočet reálnych hodnôt na presun medzi stavmi, ako aj samotný presun aktivít. Pri tomto využíva mechanizmy spomenuté v časti 4.1.

Metóda prechádza cyklicky cez všetky aktívne stavy a pre každý si počíta sumu hodnôt prechodov, teda hodnôt, ktoré by mali opustiť daný stav a prejsť do iných stavov. Táto suma je neskôr použitá na výpočet reálnych presúvaných hodnôt. Pre každý aktívny stav prejde funkcia cez všetky výstupné a následne aj vstupné prechody stavu, vypočíta platnosť prechodu a vypočíta, koľko aktivity a v ktorom smere bude daný prechod presúvať. Pri tomto využíva aj systém *prispievateľov*, aby zaručil spravodlivé presuny aktivity pri obojsmerných prechodoch. V prípade, že suma hodnôt, ktoré by mali opustiť stav, presahuje mieru členstva daného stavu, je nutné patrične znížiť tieto hodnoty. Toto sa vykoná na základe vzorca

$$T_{new} = T_{old}/(S/A),$$

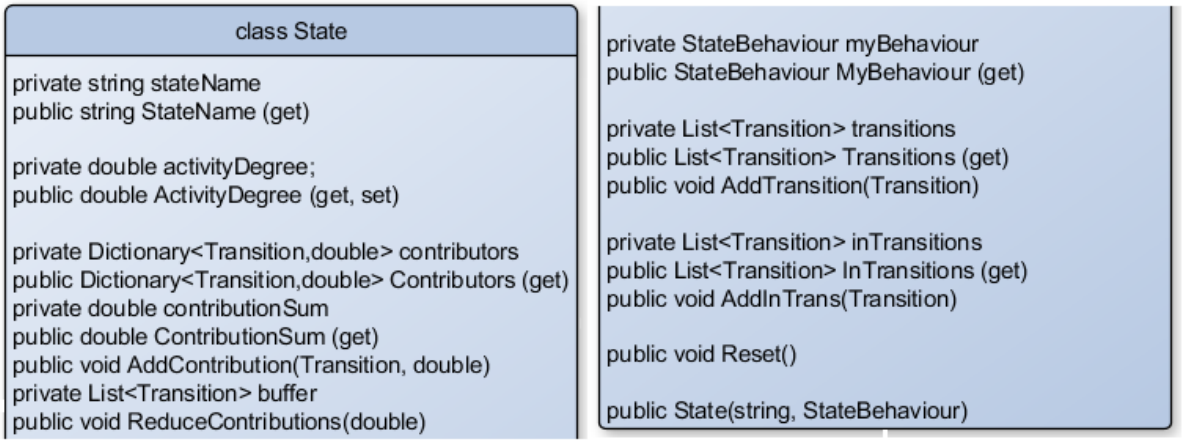
kde T_{new} predstavuje novú hodnotu, ktorá sa bude presúvať po prechode, T_{old} pôvodnú, neupravenú hodnotu, S je suma hodnôt, ktoré sa majú presunúť von zo stavu, a A je miera členstva daného stavu. Týmto výpočtom sú upravené všetky vstupné aj výstupné prechody, po ktorých miera členstva opúšťa daný stav. Po skončení tejto úpravy už je zaručené, že zo stavu odíde maximálne toľko miery členstva, koľko daný stav reálne obsahuje.

Následne sa ešte skontroluje, či sa daný stav nesnaží prevziať aktivitu od stavu, ktorý už má nulovú mieru členstva, a v takomto prípade sa nastaví na daný prechod nulová hodnota, aby nedošlo k uvedeniu stavu do zápornej miery členstva. Po prepočítaní prechodov všetkých aktívnych stavov vykoná automat ešte jeden priechod cez všetky aktívne stavy, v ktorom presunie miery členstva. Po presune skontroluje všetky stavy a pokiaľ sa nejaký stal aktívnym, teda stúpla jeho miera členstva nad 0, pridá ho do zoznamu aktívnych stavov. Analogicky, pokiaľ nejakému stavu klesla miera členstva pod 0, vyradí ho automat zo zoznamu aktívnych stavov. Posledným úkonom tejto metódy je kontrola, či sa celková miera členstva v automate príliš nevzdialila od ideálnej hodnoty 1. Ak áno, vykoná korekciu pomocou vzorca

$$A_{new} = A_{old} * (1/S),$$

kde A_{new} je nová miera členstva stavu, A_{old} je pôvodná miera a S je suma mier členstva jednotlivých stavov. Číslo 1 vo vzorci reprezentuje ideálnu hodnotu, ktorú chce automat udržiavať. Po tejto kontrole je činnosť automatu ukončená.

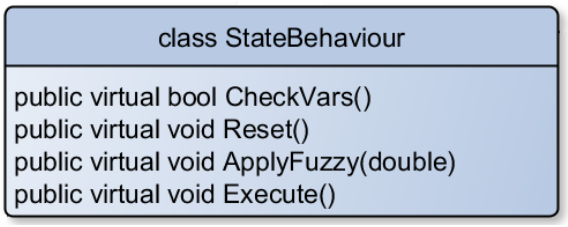
Jednotlivé stavy automatu sú implementované triedou *State* (obrázok 4.15).



Obrázek 4.15: Trieda State vo FuSM.

Trieda obsahuje mieru členstva stavu, zoznam prispievateľov, model správania stavu a zoznamy vstupných a výstupných prechodov stavu. Princíp práce systému prispievateľov bol vysvetlený v časti 4.1.

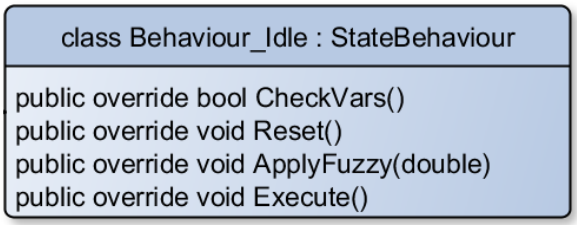
Model správania stavu je implementovaný triedou *StateBehaviour* (obrázok 4.16).



Obrázek 4.16: Trieda StateBehaviour vo FuSM.

Táto trieda sama o sebe poskytuje len virtuálne metódy na kontrolu premenných (*CheckVars*), nastavenie premenných na implicitnú hodnotu (*Reset*), aplikáciu miery členstva na fuzzy faktory (*ApplyFuzzy*) a vykonanie aktivity zodpovedajúcej modelu správania (*Execute*). Konkrétne modely správania dedia z tejto triedy a implementujú spomenuté metódy tak, aby zodpovedali konkrétnym modelom správania.

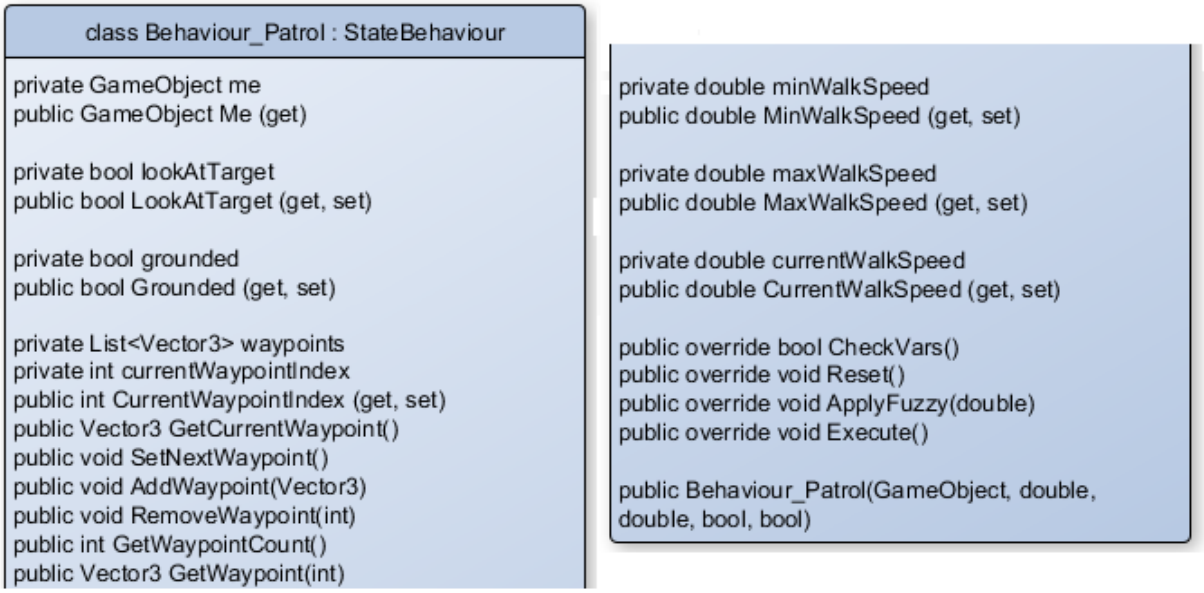
Správanie typu *nerobím nič* implementuje trieda *Behaviour_Idle* (obrázok 4.17).



Obrázek 4.17: Trieda Behaviour_Idle vo FuSM.

Toto správanie nič nevykonáva a telá jeho metód sú teda prázdne, až na metódu *CheckVars*, ktorá vracia hodnotu *true*.

Správanie typu *hliadkujem* implementuje trieda *Behaviour_Patrol* (obrázok 4.18).



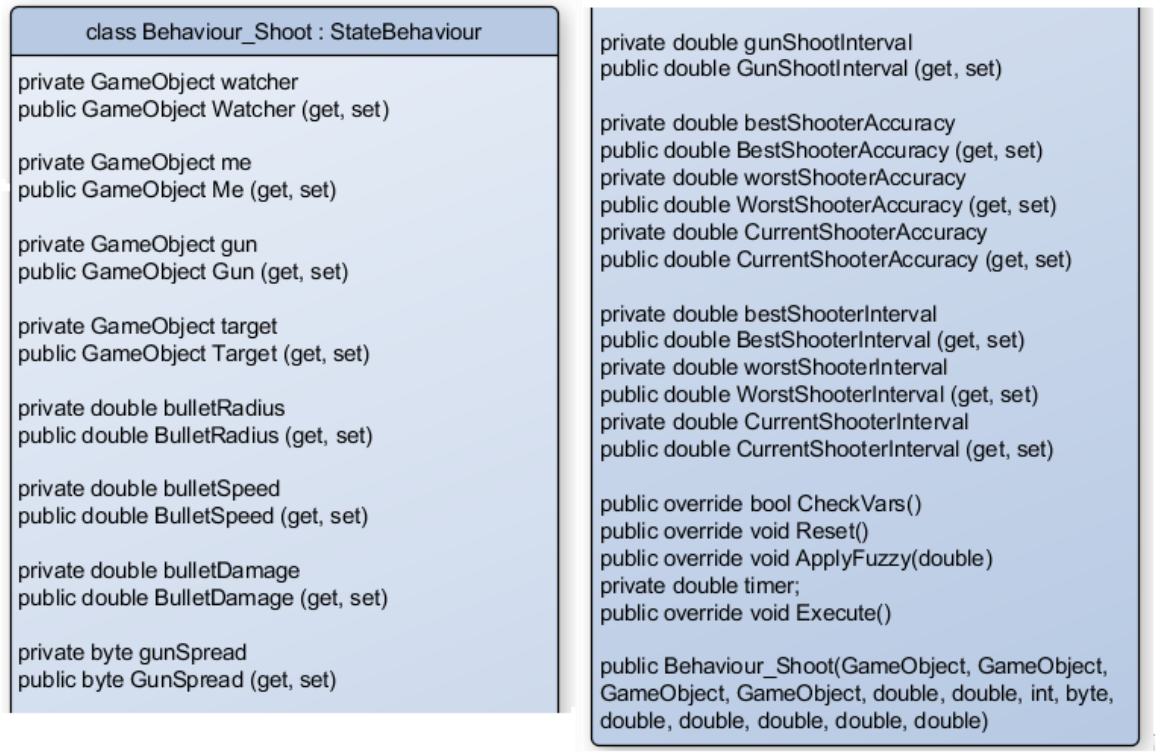
Obrázek 4.18: Trieda Behaviour_Patrol vo FuSM.

Trieda obsahuje okrem iného odkaz na vlastníka automatu, zoznam smerových bodov hliadkovacej cesty a rýchlosť pohybu herného objektu. Rýchlosť pohybu je fuzzy faktor, používateľ pri tvorbe stavu zadáva len hraničné hodnoty a automat sám na základe miery členstva upravuje aktuálnu rýchlosť. Metóda *Execute* následne využíva aktuálnu rýchlosť, aby cez Unity API pohla herným objektom v požadovanom smere, a to buď horizontálne, teda bez pohybu v y osi, alebo ľubovoľne, to znamená prípadne aj v y osi.

V priebehu vývoja sme experimentovali s rôznymi spôsobmi, ako hernými objektami pohnúť. Spočiatku sme využívali metódu *Vector3.Lerp* a upravovali *transform.Position* priamo, neskôr sme skúšali využiť fyzikálne vlastnosti Unity enginu a pracovať s *Rigidbody* a jeho metódou *AddForce*, vo výsledku sme však skončili pri metóde *Translate* komponentu *transform*. Hlavnou nevýhodou tohto postupu je fakt, že z technického hľadiska nejde o plynulý pohyb, ale o diskrétny posun hoci o malú jednotku, a v dôsledku toho nedochádza

k automatickej detekcii kolízií. Vhodnejším by mohlo byť využitie *Character controller*, prípadne ďalšie experimenty s *Rigidbody*, v práci je však momentálne využívaný spôsob *Translate*. Toto sa vzťahuje nielen na *Behaviour_Patrol*, ale na všetky pohybové modely správania.

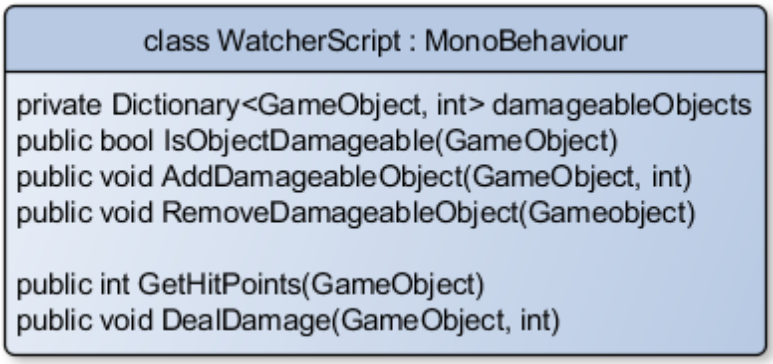
Správanie typu *strieľam* implementuje trieda *Behaviour_Shoot* (obrázok 4.19).



Obrázek 4.19: Trieda Behaviour_Shoot vo FuSM.

Táto trieda obsahuje odkaz na objekt typu *Watcher*, odkaz na vlastníka stavu, na zbraň a na cieľ. Ďalej obsahuje informácie o projektile (polomer, rýchlosť alebo spôsobené zranenie), informácie o zbrani (rozptyl a frekvenciu streľby) a informácie o strelcovi (presnosť a rýchlosť streľby). Typ *Watcher* bude objasnený neskôr. Presnosť a rýchlosť streľby sú fuzzy faktory. Za zmienku stojí telo metódy *Execute*. V prípade tohto modelu správania vytvorí metóda nový herný objekt, a to guľu (*PrimitiveType.Sphere*), ktorej následne nastaví vlastnosti ako rozmer, polomer *collideru* a smer letu. Smer letu je ovplyvnený okrem polôh strelca a cieľa aj rozptylom zbrane a presnosťou strelca. Tento rozptyl sa započítava pomocou metódy *Quaternion.AngleAxis*, ktorej výsledkom je násobený aktuálny smer letu projektilu a ktorá ako parameter dostáva náhodnú hodnotu zo stanoveného rozsahu. Tento rozsah zodpovedá rozptylu zbrane (respektíve presnosti strelca) zadanému používateľom. Projektilu je následne pridaný skript *BulletScript*, ktorý bude objasnený o chvíľu.

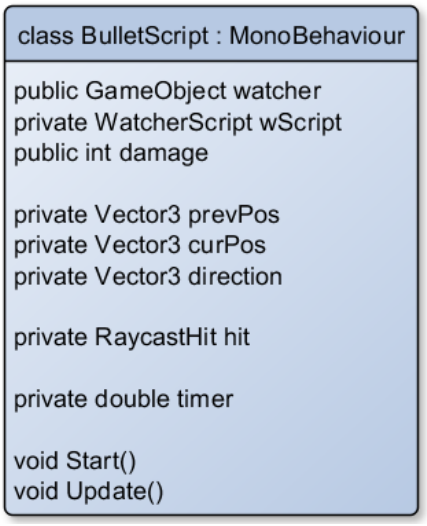
Watcher je unikátny herný objekt scény, ktorý je vybavený skriptom *WatcherScript*. Tento skript, rovnako ako skript *BulletScript*, je súčasťou vytvorenej knižnice. Watcher zastáva funkciu dozorcu nad dianím v scéne a je využívaný modelom správania *Behaviour_Shoot*. Skript *WatcherScript* obsahuje triedu *WatcherScript* (obrázok 4.20).



Obrázek 4.20: Trieda WatcherScript.

WatcherScript obsahuje zoznam zraniteľných objektov a metódy na registráciu a odregistráciu herných objektov do tohto zoznamu, ako aj metódy na zistenie stavu bodov zásahu a zmenu bodov zásahu objektov. Pokiaľ chce používateľ využívať *Watchera*, musí každý herný objekt, ktorý má byť zraniteľný, zaregistrovať u *Watchera* a pri zraneniach (resp. liečeniach) informovať *Watchera* o tejto udalosti.

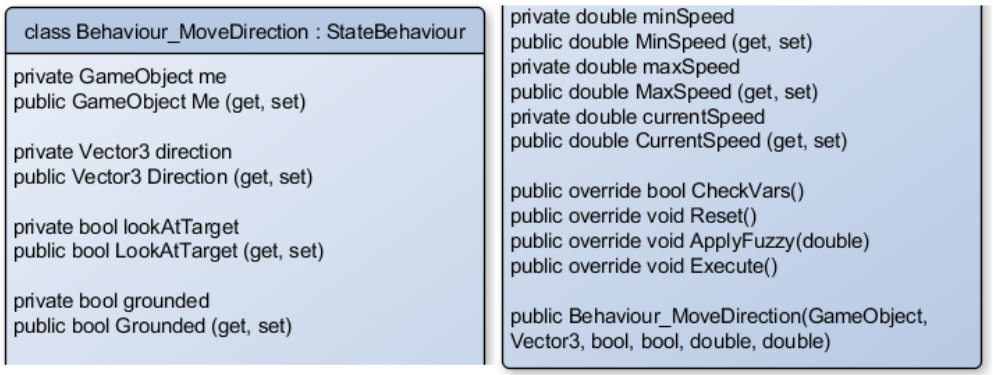
Skript *BulletScript* obsahuje triedu *BulletScript* (obrázok 4.21).



Obrázek 4.21: Trieda BulletScript.

Táto trieda obsahuje odkaz na *Watchera* a jeho skript, ďalej zranenie spôsobiteľné projektilom a premenné pre výpočet kolízie vo vysokej rýchlosti. Pri vysokých rýchlostiach, ktoré projektil bežne dosahuje, je nutné počítať kolízie s objektami vo vlastnej réžii pomocou *Raycastu*. Projektil si uchováva svoju bývalú pozíciu a v každom snímku vykoná *Raycast* od bývalej pozície po aktuálnu pozíciu. Tu si skontroluje, či nedošlo ku kolízii lúča s nejakým objektom, a pokiaľ áno, zistí od *Watchera*, či je daný herný objekt zraniteľný. V prípade, že áno, udelí danému objektu zranenie a následne zanikne. Inak zranenie neudelí, zanikne ale každopádne.

Správanie typu *pohybujem sa určitým smerom* je implementované triedou *Behaviour_MoveDirection* (obrázok 4.22).



Obrázek 4.22: Trieda Behaviour_MoveDirection vo FuSM.

Táto trieda obsahuje odkaz na vlastníka automatu, smer pohybu, príznaky špecifikujúce či sa má herný objekt pozerať, ktorým smerom sa pohybuje, a či sa má objekt pohybovať aj vo vertikálnej osi. Ďalej obsahuje fuzzy faktor rýchlosť pohybu. Veľmi podobné sú aj triedy *Behaviour_MovePosition* a *Behaviour_ChaseTarget* (obrázok 4.23).

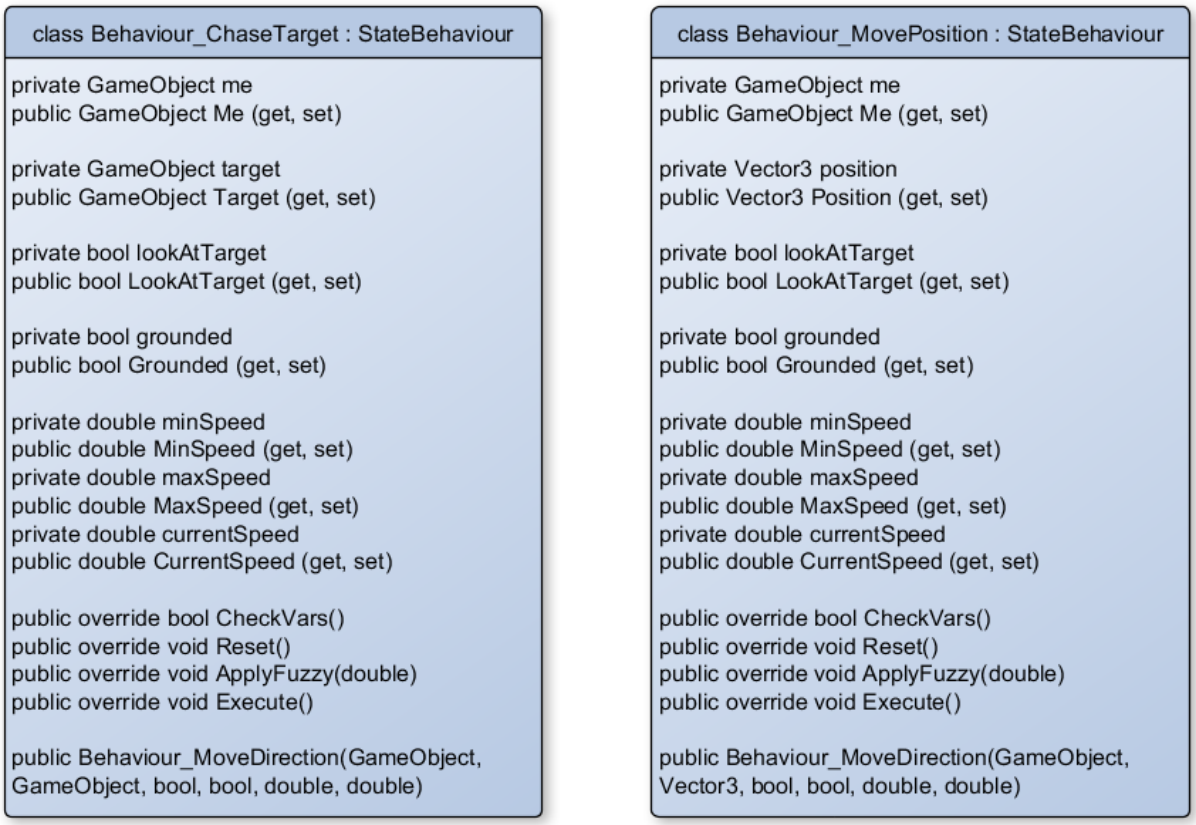
Tieto triedy sa líšia od triedy *Behaviour_MoveDirection* tým, že trieda *Behaviour_MovePosition* namiesto premennej *direction* obsahuje premennú *position*, keďže sa nepohybuje v stanovenom smere, ale na stanovené miesto. Podobne trieda *Behaviour_ChaseTarget* obsahuje premennú *target*, keďže ide o pohyb na miesto určené polohou cieľa.

Posledným modelom správania je typ *generické správanie*, implementované triedou *Behaviour_Generic* (obrázok 4.24).

Táto trieda obsahuje zoznam premenných, pričom premenné sú implementované triedou *FuzzyVar*, ku ktorej sa dostaneme o chvíľu. Ďalej obsahuje metódy na prácu s týmto zoznamom. Spôsob použitia tohto správania bol vysvetlený v časti 4.1.

Ako vyzerá trieda *FuzzyVar* znázorňuje obrázok 4.25.

Obsahuje meno premennej, ktoré je potrebné kvôli neskoršiemu prístupu k nej. Ďalej obsahuje hodnotu premennej, implicitnú hodnotu, na ktorú sa hodnota nastaví v prípade resetovania, spodnú a vrchnú hranicu premennej a príznaky, ktoré určujú, ku ktorej hranici sa má hodnota posúvať pri veľkej miere členstva a či sa má premenná pri resete nastavovať na implicitnú hodnotu. Príkladom premennej, ktorá by sa v prípade vysokej miery členstva mala pohybovať k hornej hranici, je napríklad „rýchlosť“. Pokiaľ sa objekt nachádza veľkou mierou v nejakom stave, chceme, aby sa pohyboval rýchlejšie. Opačným prípadom je napríklad premenná „rozptyl“ pri streľbe. Čím viac sa nachádzame v danom stave, tým presnejšie chceme strieľať, preto rozptyl musí klesať k spodnej hodnote.



Obrázek 4.23: Triedy Behaviour_MovePosition a Behaviour_ChaseTarget vo FuSM.

Týmto sme pokryli všetky modely správania vstavané v knižnici. Ďalej sa budeme zaoberať prechodmi, ktoré implementuje trieda *Transition* (obrázok 4.26).

Táto trieda obsahuje zoznam podmienok daného prechodu spojených cez logické *AND*, ďalej premennú *OneWay*, ktorá určuje, či ide o jednosmerný alebo obojsmerný prechod, odkaz na rodiča a cieľ prechodu, ako aj hodnotu, ktorá sa po danom prechode bude prenášať. Pokiaľ je táto hodnota kladná, ide o presun v smere prechodu (z rodiča do cieľového stavu), pri zápornej hodnote ide o spätný presun (z cieľového stavu do rodiča). Takisto obsahuje metódu na vyhodnotenie platnosti prechodu. Tá vyhodnocuje postupne všetky podmienky na prechode a vráti hodnotu najmenej platnej z nich, keďže ide o spojenie cez *AND*.

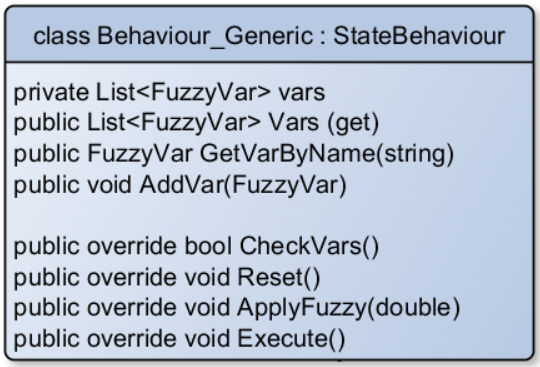
Podmienky sú implementované triedou *Condition* (obrázok 4.27).

Táto trieda tvorí bázovú triedu pre konkrétne podmienky. Obsahuje odkaz na vlastníka automatu, virtuálnu metódu *Evaluate*, ktorú si implementujú jednotlivé podmienky, *enumy* na cieľ *crisp* podmienok a na typ fuzzy množiny fuzzy podmienok, hodnoty stredu a rozptylu fuzzy množiny a metódu *DeFuzzy*, ktorá slúži na výpočet konkrétnej hodnoty na základe hodnôt fuzzy množiny a reálnej hodnoty na vstupe.

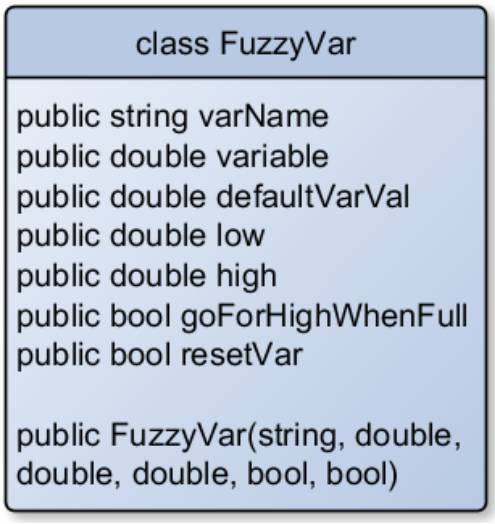
Metóda rozozná, či ide o fuzzy množinu typu *ľavá*, *stredná* alebo *pravá* (tieto pojmy boli vysvetlené v časti 4.1) a na základe toho a vstupných parametrov vypočíta hodnotu. To znázorňuje obrázok 4.28.

Na obrázku má fuzzy množina stred 5, rozptyl 2, je typu *stredná* a vstupná hodnota je 4. Metóda vyhodnotí tieto parametre a vráti ako výsledok hodnotu 0,5.

Z tejto triedy dedia triedy implementujúce konkrétne podmienky. Podmienku typu



Obrázek 4.24: Trieda Behaviour_Generic vo FuSM.



Obrázek 4.25: Trieda FuzzyVar vo FuSM.

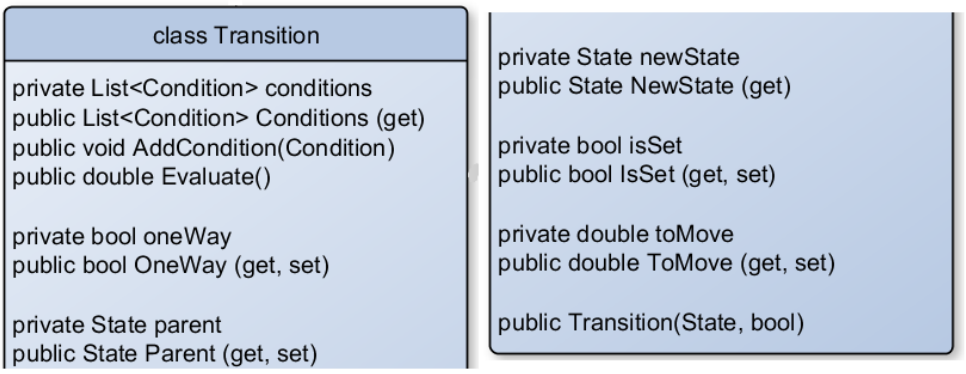
vzdialenosť k cieľu implementuje trieda *Condition_DistanceToTarget* (obrázok 4.29).

Táto trieda k prvkom definovaným v rodičovskej triede pridáva odkaz na herný objekt, voči ktorému bude porovnávať vzdialenosť a príznak *Grounded*, ktorý určuje, či má merať vzdialenosť vo všetkých troch dimenziách, alebo má ignorovať výškové rozdiely. Metóda *Evaluate* vypočíta vzdialenosť s použitím Unity API a metódy *Distance* triedy *Vector2* alebo *Vector3* a vypočítané informácie predá metóde *DeFuzzy*. Následne výsledok tejto metódy vráti ako výsledok vyhodnotenia platnosti podmienky.

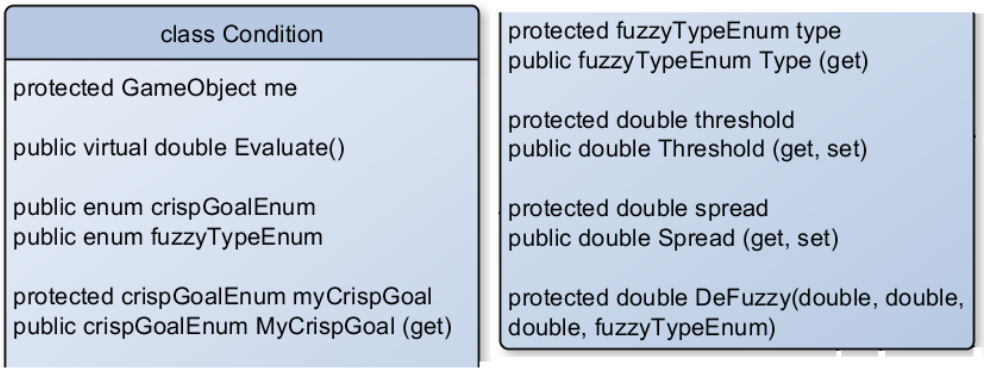
Podmienku typu *vzdialenosť k bodu* implementuje trieda *Condition_DistanceToPoint* (obrázok 4.30).

Táto trieda je prakticky totožná s predchádzajúcou triedou *Condition_DistanceToTarget*, avšak namiesto herného objektu, voči ktorému by merala vzdialenosť, sa tu nachádzajú súradnice bodu, voči ktorému bude vzdialenosť meraná. Zvyšok triedy je prakticky rovnaký.

Podmienku typu *viditeľnosť nepriateľa* implementuje trieda *Condition_TargetVisibility* (obrázok 4.31).



Obrázek 4.26: Trieda Transition vo FuSM.



Obrázek 4.27: Trieda Condition vo FuSM.

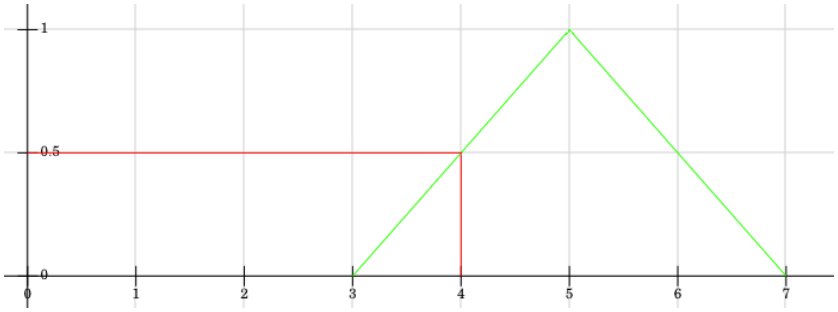
Táto trieda sa od ostatných líši tým, že implementuje *crisp* podmienku, zatiaľ čo ostatné implementujú fuzzy podmienky. Obsahuje odkaz na herný objekt, voči ktorému zisťuje viditeľnosť. V tele metódy *Evaluate* využíva metódu *Physics.Linecast* na zistenie, či medzi vlastníkom automatu a cieľom leží nejaký herný objekt. Pokiaľ narazí *Linecast* na nejaký herný objekt a nejde o náš cieľ, považujeme cieľ za skrytý. Pokiaľ *Linecast* zasiahol náš cieľ, považujeme ho za viditeľný.

Podmienka typu *časovaná podmienka* je implementovaná triedou *Condition_Timer* (obrázok 4.32).

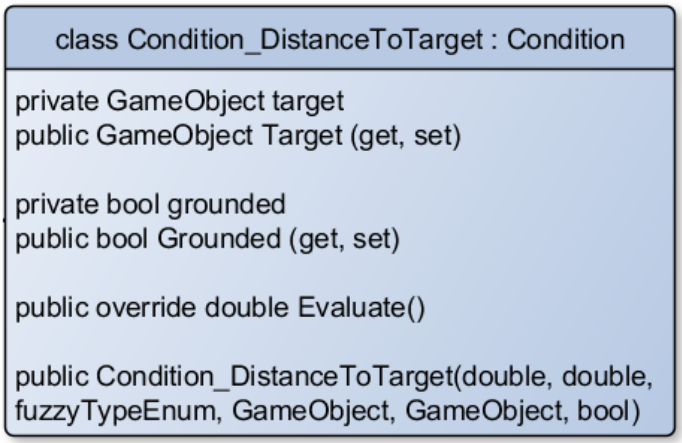
Trieda obsahuje odkaz na časovač implementovaný triedou *Timer*. Používateľ si musí vo vlastnej réžii vytvoriť objekt typu *Timer* a tento nastaviť a následne predať podmienke typu *Condition_Timer*. Štruktúru triedy *Timer* znázorňuje obrázok 4.33.

Trieda obsahuje *enum* na typ časovača (buď *časovač*, teda odpočet od nejakej kladnej hodnoty nadol, alebo *stopky*, teda beh času od 0 nahor), metódy na spustenie a zastavenie časovača, čas v sekundách, metódy na reštart časovača, metódu na aktualizáciu času pomocou *Time.deltaTime* a metódy na zistenie ubehnutého času v hodinách, minútach a sekundách.

Poslednou podmienkou je *generická podmienka* implementovaná triedou *Condition_GenericFuzzy* (obrázok 4.34).



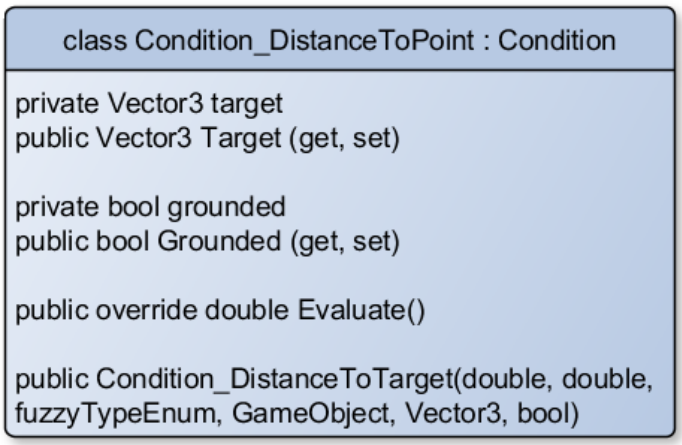
Obrázek 4.28: Vyhodnotenie fuzzy množiny vo FuSM.



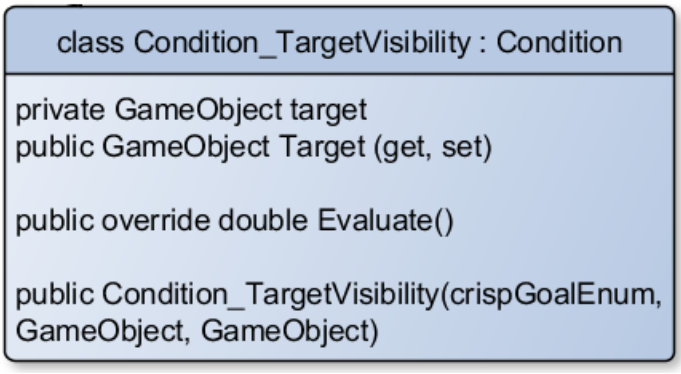
Obrázek 4.29: Trieda Condition_DistanceToTarget vo FuSM.

Táto trieda obsahuje premenné pre typ *double*, *float* a *int* a *enum* na typ premennej v danej podmienke. Metóda *Evaluate* na základe typu premennej predá konkrétnu premennú funkcii *DeFuzzy* a výsledok vráti ako platnosť podmienky. Trieda obsahuje konštruktory pre každý typ premennej.

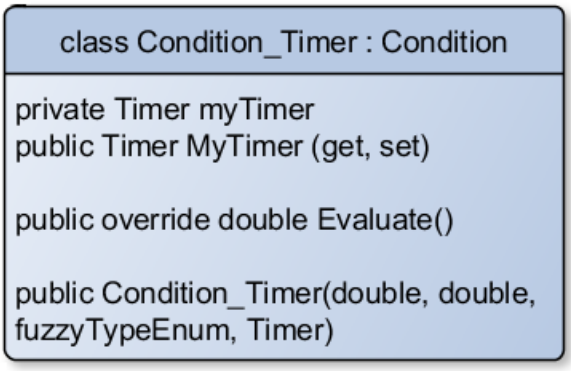
Na záver kapitoly o návrhu a implementácii uvedieme stručné štatistiky kódu. Hlavný zdrojový kód s jadrom práce obsahuje 2088 riadkov kódu a 1272 riadkov komentárov. Pomocný zdrojový kód „WatcherScript.cs“ obsahuje 58 riadkov kódu a 42 riadkov komentárov. Pomocný zdrojový kód „BulletScript.cs“ obsahuje 45 riadkov kódu a 29 riadkov komentárov.



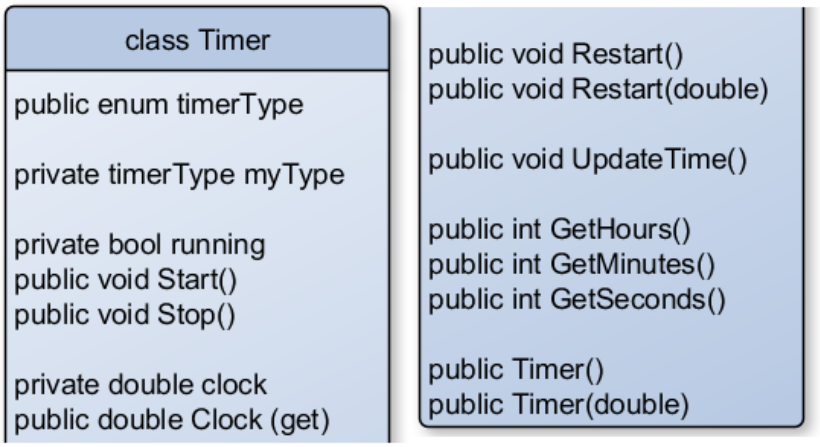
Obrázek 4.30: Trieda Condition_DistanceToPoint vo FuSM.



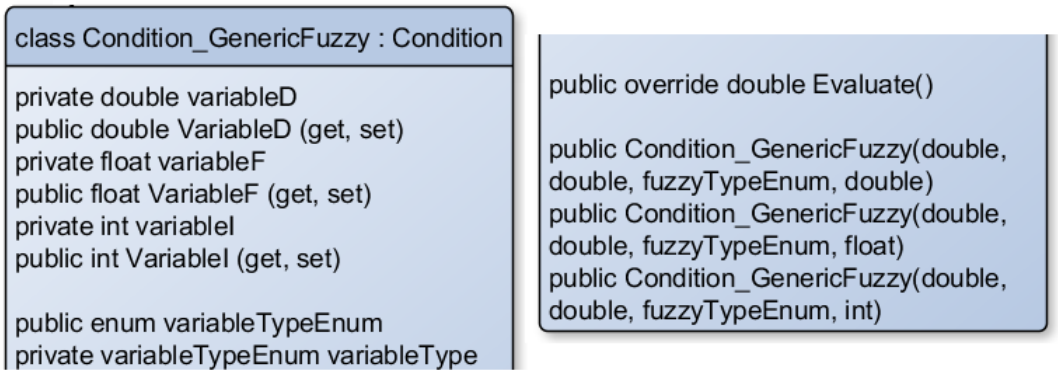
Obrázek 4.31: Trieda Condition_TargetVisibility vo FuSM.



Obrázek 4.32: Trieda Condition_Timer vo FuSM.



Obrázek 4.33: Trieda Timer vo FuSM.



Obrázek 4.34: Trieda Condition_GenericFuzzy vo FuSM.

Kapitola 5

Demonštračná scéna *Trolls*

Trolls je názov ukážkovej scény, ktorú sme v Unity engine vytvorili na demonštráciu schopností vytvoreného zásuvného modulu. Ide o relatívne veľký svet, v ktorom existujú tvory známe ako *trollovia*. Títo trollovia sa pohybujú po svete a zbierajú dva druhy surovín: vodu a zlato. Suroviny následne odkladajú na konkrétne úložné miesta. Keď už sú trollovia z práce unavení, nájdu si miesto na oddych pri ohnisku, aby načerpali energiu. Vo svete sa však môžu zjaviť aj *démoni*, ktorí našich trollov naháňajú, našťastie je im však náš svet proti srsti, a preto tu nikdy nevydržia dlhšie než pár sekúnd. Takisto sa niekedy z hlbín môže vynoriť *pekelná kosa* a vytvoriť okolo seba ohnivé pole. Kosa sa síce nepohybuje, trollovia z nej však majú náležitý strach a oblúkom sa jej snažia vyhnúť.

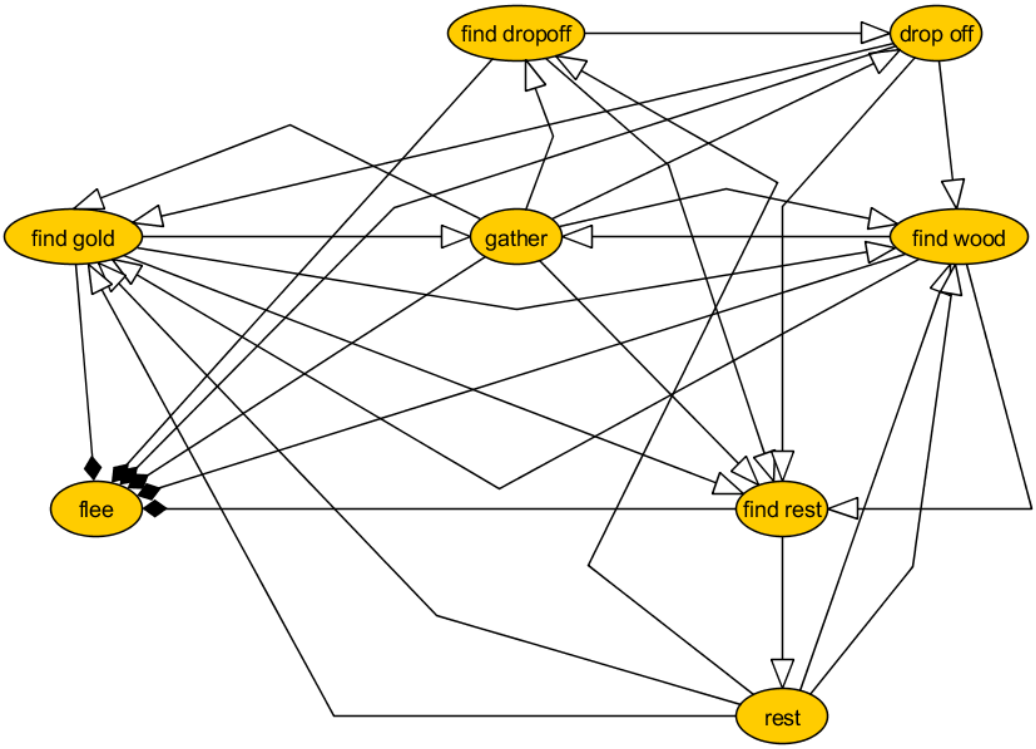
Prvá záležitosť, ktorou sme sa zaoberali, bolo vytvorenie fuzzy stavového stroja pre trollov. Tento sa v priebehu tvorby scény vyvíjal a menil, jeho finálnu podobu ukazuje obrázok 5.1.

Prechody s bielou šípkou znázorňujú jednosmerné prechody, zatiaľ čo prechody s čiernym kosoštvorcom predstavujú obojsmerné prechody. Bežná rutina trolla spočíva v hľadaní suroviny a v následnom zbere, pri zaplnení batohu potom hľadá úložisko, kde vyloží svoj náklad. Následne sa opäť pustí do hľadania suroviny. Pokiaľ je troll príliš unavený, rozhodne sa zanechať aktuálnu činnosť a začne hľadať miesto na oddych, kde si následne oddýchne. Keď ho počas aktivity zaskočí démon alebo pekelná kosa, snaží sa troll utiecť do bezpečia. Keď následne nebezpečenstvo pominie, vráti sa troll späť k aktivite, ktorú vykonával.

Po zostavení automatu sme si uvedomili nutnosť *pathfindingu*, teda schopnosti trolla nájsť vhodnú cestu k stanovenému cieľu. Nejakú dobu sme experimentovali s implementáciou tejto schopnosti, nakoniec sme však usúdili, že pathfinding nie je cieľom tejto práce a z dôvodu nedostatku času sme sa rozhodli využiť externý systém. Konkrétne ide o *A* Pathfinding Project*[1]. Tento systém využíva sieť bodov, v rámci ktorej vyhľadá optimálnu cestu k cieľu. Táto sieť sa vypočíta raz pri spustení scény a ďalej nie je nutné ju prepočítavať, až kým do scény nie je pridaná pekelná kosa. V tomto prípade je nutné sieť prepočítať tak, aby okolie kosy považovala za nepriechodné a trollovia sa tak tomuto územiu vyhýbali. Tento prepočet mriežky je však lokálny a neprepočítava sa teda celá mriežka. Ukážka práce *A* Pathfinding Project* je na obrázku 5.2.

Čo sa týka hľadania vhodnej suroviny, troll jednoducho skontroluje spoločné zásoby surovín a rozhodne sa hľadať tú, ktorej je menej. Spôsob hľadania najbližšej suroviny sme pôvodne riešili prepočítaním ciest ku všetkým surovinám a následným výberom najkratšej z nich. Toto riešenie poskytovalo cestu k reálne najbližšej surovine, avšak dĺžka výpočtov sa pri veľkom počte surovín a trollov stala prakticky nepoužiteľnou, keďže scéna takmer

[1] http://arongranberg.com/astar/



Obrázek 5.1: Automat umelej inteligencie trolla.

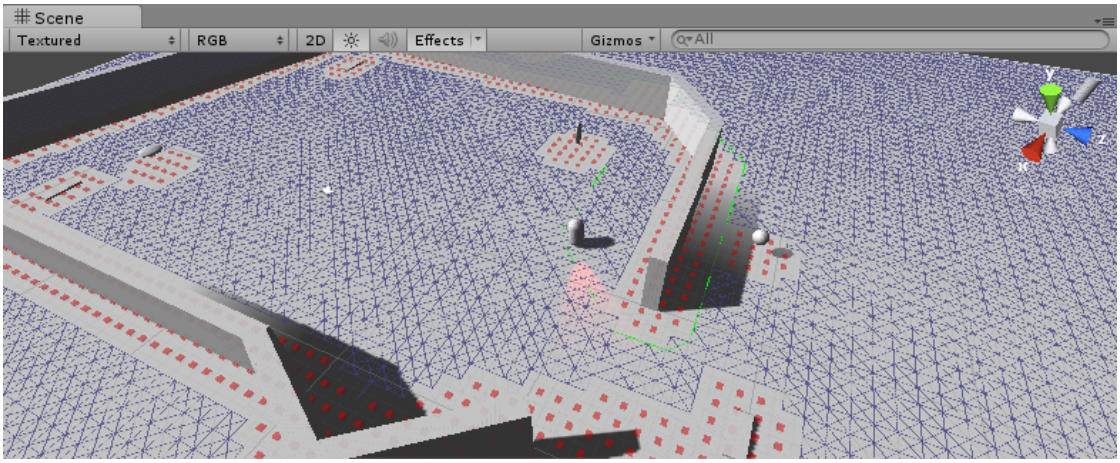
všetok čas trávila počítaním ciest a počet snímkov za sekundu rapídne klesal. Vo výsledku sme teda využili výber najbližšej suroviny na základe vzdialenosti cez vzdušnú čiaru. Tento prístup teoreticky neodráža reálnu vzdialenosť k cieľu, prakticky sa však ukázal ako dostačujúci.

Z dôvodu posunu herných objektov cez metódu *Translate* bolo nutné riešiť kolízie osobitne. Na to sme využili algoritmus, ktorý pri kolízii trolla s objektom využíva metódu *Raycast* na preskúmanie svojho okolia a zistenie, v ktorom smere leží prekážka a ktoré smery sú voľné, pričom sa následne pohne v smere vypočítanom ako priemer voľných smerov so započítaním miernej náhodnosti.

Počas testovania scény dochádzalo niekedy k javu, kde sa viac trollov snažilo zaujať rovnaké miesto pri surovine alebo úložisku. Toto bolo spôsobené relatívne veľkým krokom medzi bodmi mriežky. Vymysleli sme teda systém *slotov*, kde každá surovina, úložisko a oddychové stanovisko majú určitý počet miest, ktoré sa nachádzajú po obvode objektu. Vždy keď nejaký troll začne hľadať cestu ku konkrétnej surovine (resp. úložisku alebo oddychovému stanovisku), požiada o pridelenie slotu, ktorý bude následne vyhradený len pre tohto konkrétneho trolla. Keď sa neskôr troll rozhodne, že o danú surovinu (resp. úložisko alebo oddychové stanovisko) už nemá záujem, svoj slot uvoľní. Týmto sme eliminovali nežiadúci jav obsadzovania totožného miesta viacerými trollmi.

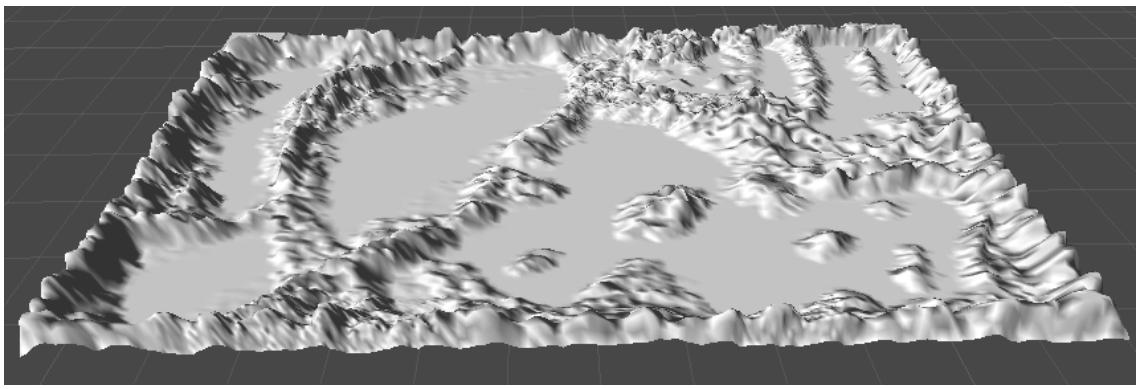
Každý troll si taktiež prepočítava vzdialenosť k najbližšiemu ďalšiemu trollovi a v prípade nízkej vzdialenosti využíva na vyhýbanie podobný algoritmus, aký sme použili pri riešení kolízií s objektami.

Vo chvíli, keď všetky základné mechanizmy fungovali podľa očakávaní, sme začali s tvorbou sveta, ktorý budú trollovia obývať. Na toto sme využili herný objekt typu *Terrain*,



Obrázek 5.2: Mriežka *A* Pathfinding Project*.

ktorý umožňuje na princípe *heightmapy* vytvoriť plochu s vyvýšeninami a priehlbinami, ktorú je následne možné otextúrovať a zaplniť vegetáciou. Keďže sme chceli dosiahnuť príjemný a rôznorodý terén, využili sme ako inšpiráciu existujúci terén, konkrétne terén oblasti *Dun Morogh* z počítačovej hry *World of Warcraft* od spoločnosti *Blizzard Entertainment*. Pri tvorbe sme využívali stránku *World of MapCraft*[2]. Vymodelovaný terén zobrazuje obrázok 5.3.

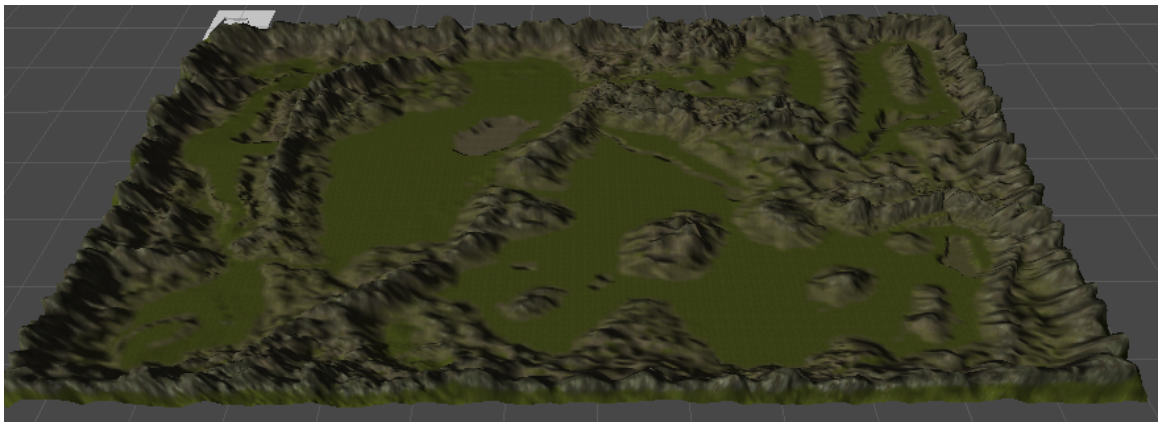


Obrázek 5.3: Prvá podoba terénu.

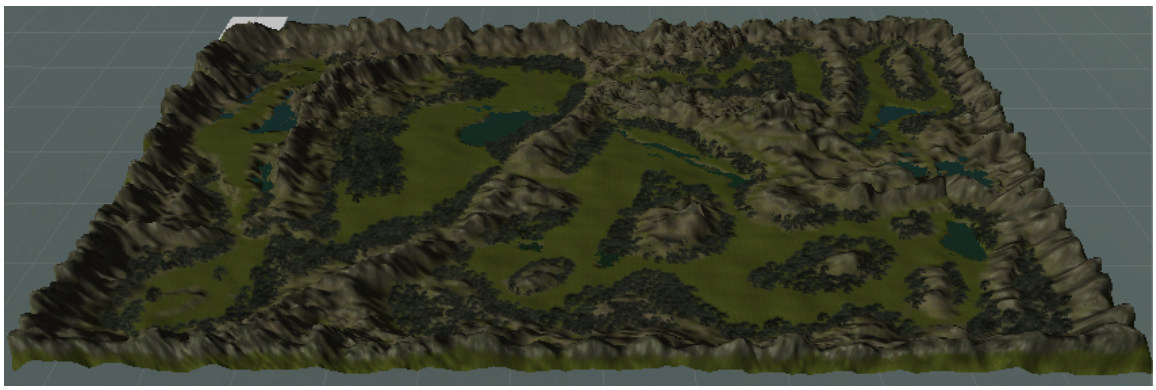
Na následné otextúrovanie terénu sme využili textúry zo štandardných Unity zdrojov (*standard assets*), rovnako ako na vytvorenie vody, stromov a časticových efektov. Otextúrovaný terén ukazuje obrázok 5.4, terén po pridaní vody a stromov obrázok 5.5.

Na ostatné záležitosti sme využili Unity Asset Store, a to nasledovne: ako *skybox* sme využili *Sky5X One*[3], na mosty sme využili *Simple Wooden Bridge*[4], na trollov sme využili *The Earthborn Troll*[5], na oddychové stanoviská a úložiská sme použili *Campfire Pack*[6], na suro-

[2] http://worldofmapcraft.com/#/4546/2525/4/
[3] https://www.assetstore.unity3d.com/#/content/6332
[4] https://www.assetstore.unity3d.com/#/content/819
[5] https://www.assetstore.unity3d.com/#/content/13541
[6] https://www.assetstore.unity3d.com/#/content/11256



Obrázek 5.4: Terén s textúrami.



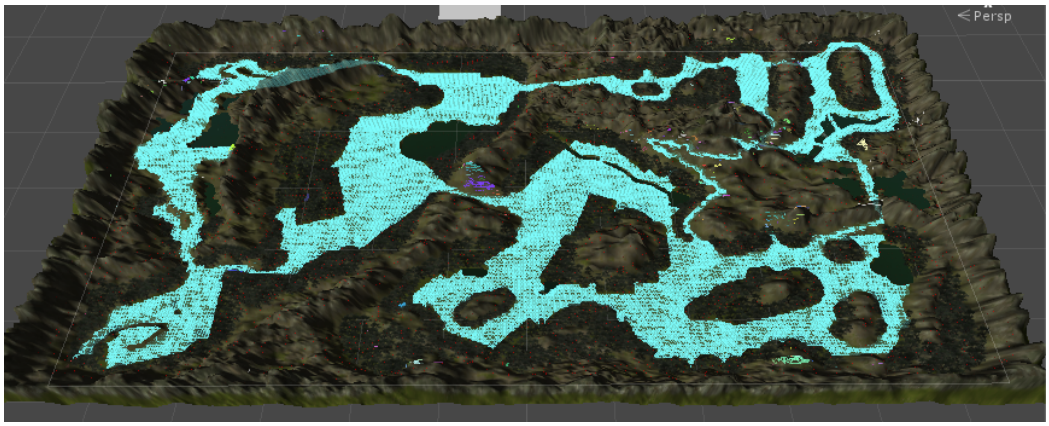
Obrázek 5.5: Finálny terén.

viny sme použili *Traditional water well*[7] pre vodu a *Medieval Gold*[8] pre zlato, na démonov sme použili *Maze Element Demon*[9] a pre pekelné kosy sme použili *Skull Scythe*.[10] Na zvuky chôdze, plameňov, vodopádov, démonov a zberu surovín sme použili *8 Bit Retro Rampage: Free Edition*[11] a *8-bit Sounds Free Package*[12]. Ambientnú hudbu tvorí pieseň *Spring Charm* od autora *Adrian von Ziegler*.[13]

Ďalej bolo nutné nastaviť vhodné parametre výpočtu pathfinding mriežky. Pôvodná vzdialenosť 1 hernej jednotky ako kroku medzi jednotlivými bodmi mriežky bola pre daný terén nepoužiteľná vzhľadom na jeho rozmery 600 krát 1200. Graf nemohol byť vypočítaný, pretože počet generovaných bodov mriežky presahoval maximálne limity. Po niekoľkých experimentoch sme skončili s hodnotou vzdialenosti medzi bodmi mriežky 2,5 hernej jednotky. Dôsledkom zväčšenia rozostupov bolo, že naplánované trasy už nepôsobili tak hladko, v praxi však boli dostatočne použiteľné a iné riešenie nám aj tak nezostávalo vzhľadom

[7] https://www.assetstore.unity3d.com/#/content/4477
[8] https://www.assetstore.unity3d.com/#/content/14162
[9] https://www.assetstore.unity3d.com/#/content/2971
[10] https://www.assetstore.unity3d.com/#/content/11982
[11] https://www.assetstore.unity3d.com/#/content/7946
[12] https://www.assetstore.unity3d.com/#/content/3766
[13] http://adrianvonziegler.bandcamp.com/

na rozmery terénu. Ako druhý parameter sme museli nastaviť *max climb*, ktorý určoval, aký je maximálny výškový rozdiel medzi bodmi, ktorý má byť považovaný za priechodný. Vzhľadom na väčší rozptyl bodov mriežky než je štandardné sme museli túto výškovú hodnotu zvýšiť. Po nastavení parametrov sa dĺžka výpočtu mriežky pohybovala okolo 3 sekúnd, čo bolo relatívne dlho. Následné pridanie *colliderov* na miesta, ktoré majú byť neprístupné (ako kopce, voda, zrázy a lesy), však znížilo dĺžku výpočtu až na približne 700 milisekúnd. Obrázok 5.6 ukazuje výslednú mriežku prístupných oblastí (tyrkysová farba).

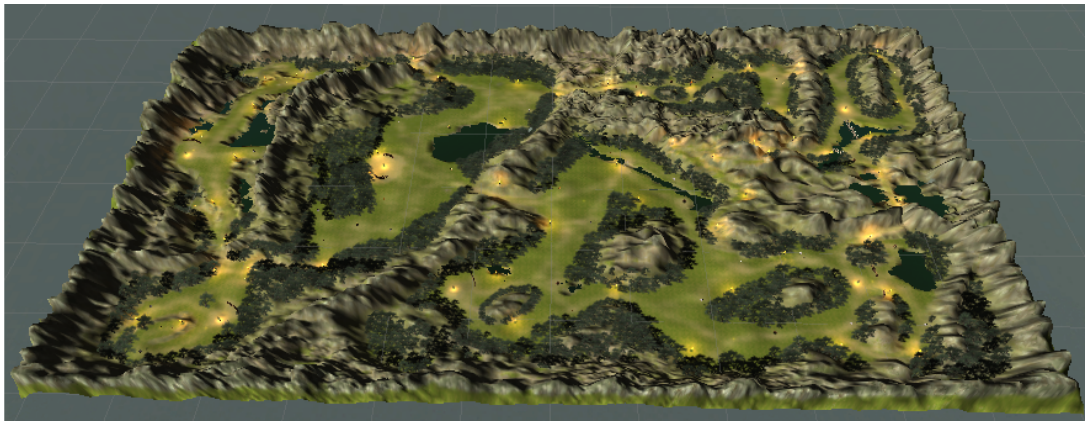


Obrázek 5.6: Mriežka prístupných oblastí.

Nasledovalo pridanie *GUI* (*graphical user interface*, slovensky *grafické používateľské rozhranie*) na pridávanie démonov, pekelných kôs a trollov. Toto pridávanie využíva funkciu, ktorá na základe raycastu od kamery smerom k polohe kurzoru nájde konkrétny bod na povrchu. Potom sme vytvorili systém pohybu kamery, ktorý je podobný systému z hry *Black & White* od spoločnosti *Lionhead Studios*. Pravým tlačidlom myši používateľ voľne otáča kamerou, zatiaľ čo ľavým sa posúva po teréne a koliečkom ovláda vzdialenosť od povrchu. Je samozrejmé, že pohyb kamery závisí od vzdialenosti od povrchu a vďaka tomu si zachováva pocit konštantnej rýchlosti, či už sa nachádza pár metrov nad povrchom, alebo vysoko vo vzduchu.

Takisto sme implementovali striedanie dňa a noci, s čím súvisela otázka osvetlenia a tieňov. Free licencia Unity engine nepodporuje počítanie tieňov od *point* a *spot* svetiel v reálnom čase, museli sme teda využiť možnosť vytvorenia *lightmapy*. Nasledovalo niekoľko hodín experimentovania s lightmapami, z ktorých vyplynuli dve možnosti: buď sa lightmapa vytvárať nebude, objekty budú mať *realtime* tiene od slnka a osvetlenie (ale nie tiene) od ostatných zdrojov svetla, alebo sa vytvorí lightmapa, avšak terén bude statický, čo znamená, že nebudú existovať realtime tiene od slnka, ale budú vytvorené svetlá a tiene od ostatných zdrojov svetla. Možnosť vytvoriť lightmapu s tým, že by terén nebol statický, bola nereálna, pretože v tom prípade by sa na tento terén nenaniesli ani tiene, ani svetlá z *baked-only* svetelných zdrojov. Možnosť vytvoriť lightmapu tak, že by terén bol statický a aj svetlo zo slnka by bolo považované za *baked-only*, bola taktiež nereálna, pretože by odporovala systému striedania dňa a noci. V konečnom výsledku sme sa rozhodli ponechať terén statický, takže na ňom budú svetlá a tiene od všetkých svetelných zdrojov okrem slnka, a realtime tiene od slnka simulovať použitím *shadow blob projectorov* nad každým objektom, s rotáciou týchto projektorov na základe rotácie slnka. Ako doplnok k striedaniu dňa a noci sme implementovali aj zmenu skyboxu v rôznych častiach dňa. Výsledný svet

so všetkými hernými objektami a s použitím lightmapy zobrazuje obrázok 5.7.

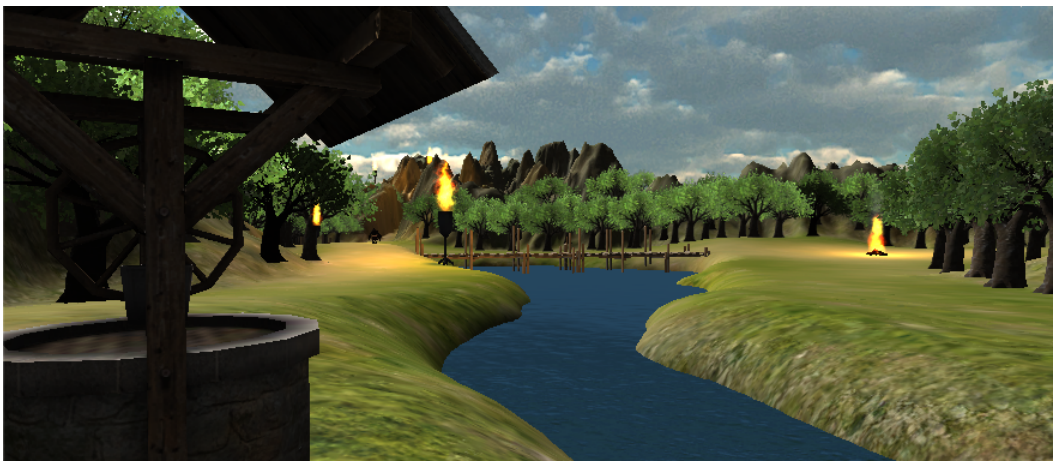


Obrázek 5.7: Celý herný svet.

Využitie našej knižnice zjednodušilo tvorbu umelej inteligencie na niekoľko krokov: naplánovanie fuzzy stavového stroja a definíciu stavov, modelov správania, podmienok a prechodov. Následne sme už museli len čítať stav automatu a implementovať detaily, ktoré zásuvný modul s tak všeobecným zameraním nemohol riešiť vo vlastnej réžii, ako napríklad hľadanie surovín.

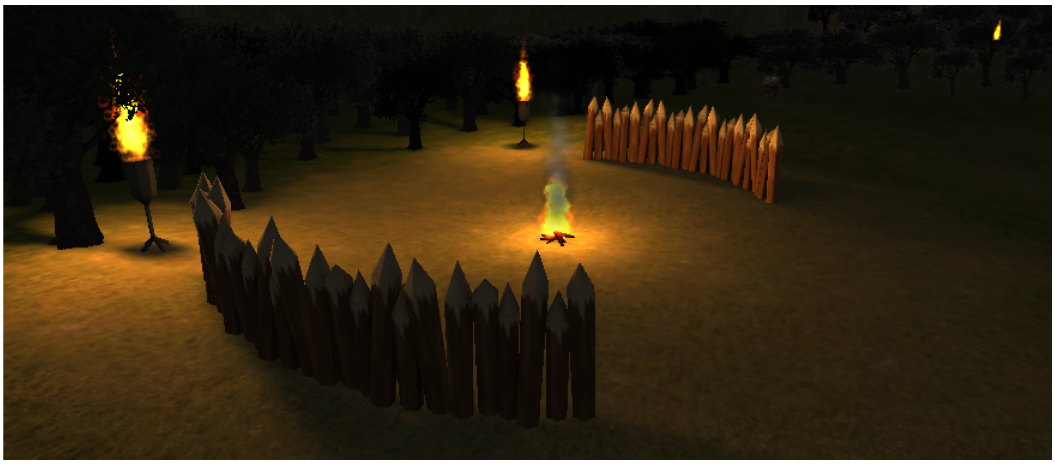
Aktuálne je v scéne použitých okrem pathfindingu 10 skriptov. Hlavný skript zodpovedajúci umelej inteligencii trollov obsahuje 1168 riadkov kódu a 197 riadkov komentárov. Zvyšné skripty, ktoré sú zodpovedné za správanie ostatných objektov scény, majú spoločne 608 riadkov kódu a 103 riadkov komentárov.

Nasleduje niekoľko snímkov obrazovky z tejto scény.



Obrázek 5.8: Studňa a premostená rieka.

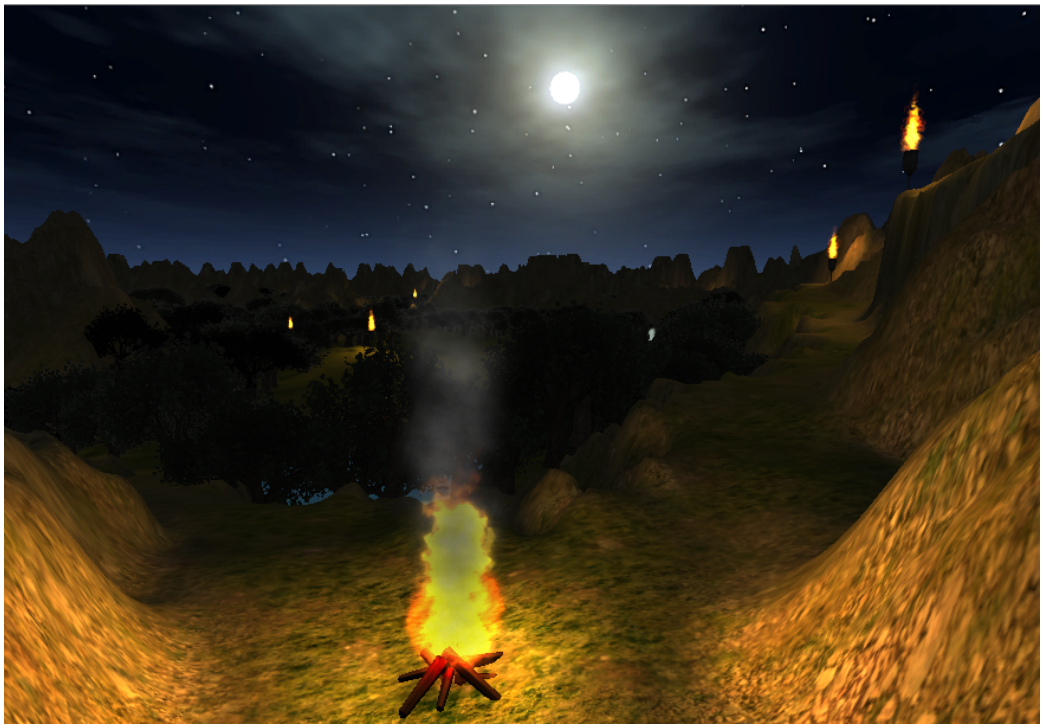
Obrázek 5.9: Jazero s vodopádom.



Obrázek 5.10: Oddychový tábor trollov.

Obrázek 5.11: Večerná cesta.

Obrázek 5.12: Nočný svet.



Obrázek 5.13: Ohnisko pod nočnou oblohou.

Kapitola 6

Záver

Cieľom práce bolo vytvoriť zásuvný modul pre Unity implementujúci fuzzy stavový stroj. Až na nejasnosť okolo pojmu *zásuvný modul* vysvetlenú v časti 3.2 bol cieľ práce splnený. Vytvorená knižnica sa dá jednoducho vložiť do Unity projektu a využívať s dobrými výsledkami, ako ukazuje ukážková scéna *Trolls*. V knižnici využívame Unity API. Celý systém funkčnosti fuzzy stavového automatu v knižnici bol vytvorený bez prevzatia akejkoľvek existujúcej súčasti podobného typu, hoci, prirodzene, na základe všeobecných pravidiel a princípov fuzzy logiky a fuzzy stavových automatov.

Počas práce sme hlbšie prenikli do princípov fuzzy logiky a fuzzy stavových automatov, ako aj do ovládania Unity enginu. Prakticky sme narazili na problémy a ťažkosti spojené s fuzzy logikou a vyskúšali rôzne riešenia.

Projekt ponúka viaceré možnosti na budúci vývoj, a to ako v oblasti vnútorných mechanizmov, tak aj v oblasti používateľského rozhrania. Systém pohybu herných objektov by mohol byť riešený novými spôsobmi, napríklad použitím *Character controlleru* alebo *Rigidbody*. Mohli by byť pridané bohatšie vstavané modely správania a podmienky. Rovnako by mohli byť rozšírené existujúce vstavané modely správania a podmienky, aby ponúkali širšiu možnosť parametrizácie. Takisto by sa dalo uvažovať o spolupráci automatu so systémom animácií *Mecanim*. V oblasti používateľského rozhrania by bola vhodná integrácia s prostredím Unity editora a využitie grafického prístupu ako doplnku k textovému prístupu cez skripty.

Literatura

[1] Doostfatemeh, M.; Kremer, S. C.: New directions in fuzzy automata. *International Journal of Approximate Reasoning*, ročník 38, č. 2, 2005, ISSN 0888-613X.

[2] Dybsand, E.: A Generic Fuzzy State Machine in C++. In *Game Programming Gems 2*, kapitola 3.12, Charles River Media, 2001, ISBN 978-1-58450-054-4.

[3] McCuskey, M.: Fuzzy Logic for Video Games. In *Game Programming Gems 1*, kapitola 3.8, Charles River Media, 2000, ISBN 978-1-58450-049-0.

[4] Millington, I.: *Artificial Intelligence for Games*. Taylor & Francis Group, 2006, ISBN 978-0-12-497782-2.

[5] Novák, V.; Močkoř, J.; Perfilieva, I.: *Mathematical Principles of Fuzzy Logic*. Springer, 1999, ISBN 978-0-7923-8595-0.

[6] Reyneri, L. M.: An introduction to Fuzzy State Automata. In *Biological and Artificial Computation: From Neuroscience to Technology*, Springer Berlin Heidelberg, 1997, ISBN 978-3-540-63047-0.

[7] Schwarz, J.: Fuzzy - principy, mikrokontroléry, aplikace. Prednáška predmetu IMP.

[8] Scott, B.: The Illusion of Intelligence. In *AI Game Programming Wisdom*, kapitola 1.2, Charles River Media, 2002, ISBN 978-1-58450-077-3.

[9] Tozour, P.: The Evolution of Game AI. In *AI Game Programming Wisdom*, kapitola 1.1, Charles River Media, 2002, ISBN 978-1-58450-077-3.

[10] Wong, G.; Wang, J.: A Fuzzy-Control Approach to Managing Scene Complexity. In *Game Programming Gems 6*, kapitola 3.9, Charles River Media, 2006, ISBN 978-1-58450-450-4.

[11] WWW stránky: Fuzzy expertné systémy [online]. 2009 [cit. 2014-5-4].
URL
http://www2.fiit.stuba.sk/~kapustik/ZS/Clanky0910/farkas/index.html

[12] WWW stránky: Unity - Plugins (Pro/Mobile-Only Feature). 2012-2-2 [cit. 2014-5-5].
URL http://docs.unity3d.com/Documentation/Manual/Plugins.html

[13] WWW stránky: Unity Software License Agreement 4.x. 2013-7-12 [cit. 2014-5-5].
URL http://unity3d.com/company/legal/eula

[14] WWW stránky: Unity - Using Mono DLLs in a Unity Project. 2013-7-8 [cit. 2014-5-5].
URL http://docs.unity3d.com/Documentation/Manual/UsingDLL.html

[15] WWW stránky: Unity 5 Announced at GDC 2014, Pre-order Begins. 2014-3-18 [cit. 2014-5-5].
URL
http://unity3d.com/company/public-relations/news/unity-announces-unity5

[16] WWW stránky: Unity - Game Engine [online]. 2014 [cit. 2014-5-5].
URL http://unity3d.com/

[17] WWW stránky: Fuzzy Logic [online]. [cit. 2014-5-4].
URL http://wing.comp.nus.edu.sg/pris/FuzzyLogic/FuzzyLogicContent.html

[18] Zarozinski, M.: Imploding Combinatorial Explosion in a Fuzzy System. In *Game Programming Gems 2*, kapitola 3.13, Charles River Media, 2001, ISBN 978-1-58450-054-4.