VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

HRA S AGENTY NA BÁZI UMĚLÉ INTELIGENCE

GAME WITH NON-PLAYER CHARACTERS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. LUKÁŠ TOMEČKO**
AUTHOR

VEDOUCÍ PRÁCE **doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Tomečko Lukáš, Bc.**
Program: Informační technologie Obor: Inteligentní systémy
Název: **Hra s agenty na bázi umělé inteligence**
Game with Non-Player Characters
Kategorie: Umělá inteligence

Zadání:

1. Prostudujte doporučenou literaturu zabývající se problematikou konstrukce počítačových her.
2. Navrhněte akční strategickou hru v dvoudimenzionálním světě (pohled shora), kde hraje člověk proti inteligentním nepřátelům (NPC - non-player character). NPC jsou stroje, resp. roboti, vybavení zbraněmi působícími na dálku. Hru organizujte do misí, kde bude hráč plnit zadané úkoly (přesun, průzkum, vyhledání objektu), aniž by byl zničen. Mise by měly být propojeny příběhem. Navrhněte několik typů NPC, vybavených různými typy senzorů (např. dálkoměr, detektor pohybu, mikrofon), aktuátorů (např. směrové pohony, zbraně atd.) a inteligentního řízení. Umožněte též navigaci a vzájemnou komunikaci a kooperaci NPC. Prostředí by mělo zahrnovat různé typy terénu i městské oblasti. Při kolizích v prostředí se uplatní fyzika.
3. Hru implementujte s využitím vhodných knihoven pro herní grafiku a fyziku. Umožněte hru snadno upravovat a rozšiřovat o nové typy agentů. Pro testování připravte několik typů agentů s různými variantami inteligentního řízení.
4. Hru otestujte a vyhodnoťte dosažené výsledky. Při testování a vyhodnocování výsledků se zaměřte jak na na subjektivní pocity hráče, tak na objektivní hodnocení kvality NPC pomocí vhodně navržených metrik (výsledek a trvání hry, stav/poškození robotů na konci hry, případně procento splněných podúloh).

Literatura:

- Yannakakis, Georgios N., and Julian Togelius. Artificial Intelligence and Games. Springer, 2018. URL: https://link.springer.com/book/10.1007/978-3-319-63519-4
- Waveren, J.M.P.. (2001). (Master of Science thesis) The Quake III Arena Bot. URL: https://www.researchgate.net/publication/240430519_The_Quake_III_Arena_Bot

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Janoušek Vladimír, doc. Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 20. května 2020
Datum schválení: 31. října 2019

Abstrakt

Cieľom práce je navrhnúť a implementovať 2D akčnú počítačovú hru so strategickými prvkami, v ktorej sa hrá proti inteligentným agentom. Návrh architektúry vychádza z rokmi overených techník a vzorov. Hra je písaná v jazyku C++, na grafickú stránku a vstupy sa využíva knižnica SFML, na simuláciu kolízií knižnica Box2D. Umelá inteligencia je založená na kombinácii štandardných prístupov používaných v hernom priemysle. Výsledná hra a inteligencia agentov je testovaná a zhodnotená hráčmi a metrikami.

Abstract

The goal of this project is to create a 2D action strategy videogame, featuring intelligent enemies. The architecture design is based on techniques and patterns used in game industry. Game is written in C++, SFML library is used for graphics and inputs, Box2D library takes care of physics. Enemies' artificial intelligence applies standard algorithms used in videogame industry. Human players and metrics are used for evaluation of final game and enemies' intelligence.

Kľúčové slová

hra, engine, umělá inteligence, SFML, Box2D

Keywords

game, engine, artificial intelligence, SFML, Box2D

Citácia

TOMEČKO, Lukáš. *Hra s agenty na bázi umělé inteligence*. Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Vladimír Janoušek, Ph.D.

Hra s agenty na bázi umělé inteligence

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána docenta Janouška. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Lukáš Tomečko
3. júna 2020

Poďakovanie

Chcel by som sa poďakovať pánovi docentovi Janouškovi za užitočné rady a voľnosť.

Obsah

Kapitola 1

Úvod

To, čo sa v minulosti považovalo len za zábavu pre deti, je dnes multimiliardový biznis s odhadovanými svetovými tržbami vyše 80 miliárd dolárov (v 2014) a priemerným vekom hráča 35 rokov. Finále svetového poháru v 2018 jednej z najhranejších hier sveta, League of Legends, sledovalo online takmer 100 miliónov unikátnych divákov. V turnaji hry Dota 2 sa v roku 2019 zase hralo o vyše 35 miliónov dolárov. [26] Na jednej strane veľké herné štúdiá tlačia na vývoj stále výkonnejšieho hardware a efektívnejších algoritmov. Na druhej strane stále viac a viac nezávislých vývojárov si môžu dovoliť experimentovať s netradičnými prvkami hier, čím sa obe strany obohacujú.[14] [5]

Táto práca sa preto zaoberá štruktúrou typickej komplexnej počítačovej hry a jej jednotlivých podsystémov prítomných takmer v každej hre (kap. 2). V ďalšej časti poskytuje prehľad prístupov umelej inteligencie využívanej v hrách. Cieľom práce je vytvoriť 2D počítačovú hru, kde by hráč hral proti inteligentným počítačom riadeným nepriateľom, s využitím viacerých typov umelej inteligencie. Pomocou subjektívnych a objektívnych metrík sa potom zhodnotí vytvorené dielo.

Motiváciou k práci bola snaha pochopiť návrh zložitejších hier, a využiť teoretické školské učivo z rôznych oblastí na praktickom projekte.

Kapitola 2

Architektúra videohier

Vzhľadom na to, že vo väčšine hier sa riešia podobné problémy, aj ich vnútorná štruktúra sa podobá. Jedná sa o komplexné modulárne systémy, často využívajúce komponenty tretích strán, z ekonomických aj časových dôvodov.

Po odbornom pohľade na videohru sa v tejto kapitole vymedzí pojem herný engine, spolu s príkladmi existujúcich enginov. Ďalšie podkapitoly predstavujú prehľad jeho jednotlivých podsystémov, často používané vzory, koncepty a pojmy používané naprieč konkrétnymi implementáciami. Vzhľadom na zamýšľanú výslednú hru a obmedzený rozsah práce budú niektoré časti rozobrané viac než iné. Táto práca sa vôbec nezaoberá herným designom, ktorý je pre úspešnosť každej hry kritickým. Jedná sa napríklad o pravidlá a ciele hry, možné akcie (tzv. mechaniky), návrh prostredia, postáv a prvkov, vyvažovanie obtiažnosti, ekonomiky atď.

Ak nie je uvedené inak, tak kapitola primárne vychádza z knihy [24].

2.1 Videohry

Z informatického hľadiska by sa dala väčšina videohier charakterizovať ako soft real-time interaktívna agentne založená počítačová simulácia. Ide o zjednodušený model reálneho či vymysleného sveta, reprezentovaný a simulovaný v počítači. Vystupujú a interagujú v ňom entity (agenti), ktoré môžu mať podobu postáv, vozidiel, či rôznych abstraktných tvarov. Používateľ ovplyvňuje stav hry interakciou cez vstupné zariadenia (klávesnica, volant a pedále, mobilná kamera, dotyková obrazovka, myš atď.), a hra reaguje zmenou na obrazovke, zvukom apod. Simulácia a interakcia prebieha v reálnom čase, pričom obrazovka sa zvyčajne prekresľuje 24 až 60krát za sekundu. Ak sa má teda užívateľský vstup prejaviť okamžite, hra by to mala stihnúť do 1/60 až 1/24 sekundy. Keďže sa ale jedná o soft real-time systém a tento čas sa nestihne, nič životohrozujúce sa nedeje (narozdiel od hard real-time systémov). Užívateľ si vtedy všimne zmenu plynulosti hry, čo pôsobí rušivo a znižuje zážitok z hry.

2.2 Herný engine

Keď v sedemdesiatych a osemdesiatych rokoch vznikali softvérom riadené arkádové a konzolové hry, ich softvér bol vysoko špecifický pre danú hru a konkrétny hardvér. Postupom času sa však začali v softvéri hier objavovať určité spoločné vzory, pretože väčšina hier riešila podobné problémy – vykresľovanie grafiky, kolízie objektov, audio atď.

Keď v devädesiatych rokoch vyšla celosvetovo populárna hra Doom (od id Software), vývojári a hráči si všimli oddelenosť jadra hry od herného sveta. Využili to a s novými assets a len minimálnymi zásahmi do jadra – tzv. engine – upravovali a vytvárali nové hry.

Na tejto myšlienke prispôsobovania a znovu použiteľnosti začali vznikať hry ako Unreal a Quake III Arena, ktoré boli okrem dátových súborov (textúry, geometria sveta, konfigurácie NPC a herných položiek) upraviteľné aj cez skriptovací jazyk [34]. Licencovanie enginov sa tak stalo pre vývojárov druhým zdrojom príjmov. V súčasnosti sa okrem celých enginov dajú opätovne používať a licencovať aj jednotlivé komponenty (viac v kap. 2.3), čo je rýchlejšie a často ekonomickejšie, než vyvíjať vlastnú komponentu.

Ani súčasné enginy však nie sú dostatočne univerzálne na vytvorenie akejkoľvek hry. Buď sú vyvíjané pre určitý žáner, a vytvorenie hry iného žánru môže byť v nich krkolomné. Alebo aj keď sa snažia byť univerzálnymi, tak je to na úkor výkonu. Jedná sa teda o spojité spektrum, kde na jednom konci je engine špeciálne vyvinutý a optimalizovaný pre jedinú hru pre konkrétny hardvér. Na druhom konci spektra je (idealizovaný) univerzálny engine vhodný pre akúkoľvek hru na akejkoľvek platforme. To tiež znamená, že hranica medzi enginom a herným obsahom často nie je jasná.

2.2.1 Existujúce enginy

Všetky z nižšie vymenovaných enginov obsahujú okrem runtime komponent aj offline nástroje s grafickým rozhraním na vytváranie herných assetov a kódu hry. Taktiež umožňujú vytvárať hry pre viac platforiem. Vzhľadom na ich neustály vývoj nemá moc zmysel rozpisovať ich konkrétne schopnosti, ktoré sú dohľadateľné na ich webových stránkach.

Medzi najpopulárnejšie 3D enginy patrí Unreal Engine, ktorý sa od roku 1998 dostal už do verzie 4. Okrem C++ (resp. v minulých verziách skriptovacieho jazyku UnrealScript) sa dá použiť aj vizuálny skriptovací systém. Pre nekomerčné účely je bezplatný, pri komerčnom využití sú k dispozícii aj jeho zdrojové kódy. Vyčerpávajúci zoznam hier [15] v ňom vytvorených, obsahujúci množstvo komerčne úspešných titulov, hovorí sám za seba.

Ďalším populárnym enginom na 2D a 3D hry je Unity. Programovanie umožňuje v C# (a v starších verziách v UnityScripte – založený na JavaScripte). Existuje v bezplatnej a platenej verzii.

Ďalej existuje Source engine. Herná spoločnosť Valve, autor enginu, ho použila vo viacerých svojich úspešných 3D hrách. Komerčné využitie inými spoločnosťami je spoplatnené. Využíva C++.

Autori CryEngine tiež ukázali schopnosti svojho enginu na 3D komerčných tituloch, ktorý je pre ostatných zdarma. Bezplatnou alternatívou na 2D a 3D hry od Microsoftu je XNA Game Studio, ktoré využíva C#. Na 2D a 3D hry sa taktiež dá použiť bezplatný Godot.[3] Medzi open source enginy sa radia OGRE 3D či Torque.

Z 2D enginov treba spomenúť Game Maker Studio (v minulosti Game Maker), ktorý si obľúbili mnohí nezávislí vývojári a dal za vznik viacerým úspešným hrám. Na skriptovanie hier využíva vlastný interpretovaný jazyk GML, alebo sa dá využiť „drag and drop“ prístup. Existuje v obmedzenejšej bezplatnej lite verzii, ale aj plnohodnotnej platenej, s cenou podľa požadovaných platforiem. [27]

V prostredí webových prehliadačov sa s rozšírením HTML5 objavilo množstvo JavaScriptových enginov (hlavne 2D), ktoré zjednodušujú prácu so vstavanými Canvas API a WebGL API. Tie sa nevyužívajú len na hry, ale aj na interaktívne animované webové stránky. Okrem riešenia problémovej kompatibility webových prehliadačov (grafiky a vstupov) poskytujú aj prostriedky dostupné v ne-webových enginoch, ako detekcia kolízií a

fyzika, načítavanie assets atď. Niektoré z aktuálne populárnych sú: Phaser, GDevelop, PixiJS, Babylon.js.[4]

2.3 Jednotlivé podsystémy videohry

Na obrázku 2.1 je ukázaná obecná architektúra modernej komplexnej 3D hry, ktorej jednotlivé komponenty budú rozpísané v nasledujúcich podkapitolách. Vzhľadom na obmedzený rozsah tejto práce sa bude jednať len o stručný popis jednotlivých častí a vysvetlenie často používaných konceptov a pojmov, ktoré sa opakujú v rôznych implementáciách. Taktiež budú spomenuté existujúce implementácie, ktorých využitie zjednodušuje a urýchľuje vývoj hier.

Čím nižšie vrstvy na obrázku [img-arch], tým obecnejšie býva využitie daných komponent, nielen na hry, väčšinou vyvíjané treťou stranou. Čím vyššie vrstvy, tým špecializovanejšie komponenty pre danú hru. Hranica univerzálneho znovupoužiteľného enginu a konkrétnej hry sa často nedá presne určiť.

2.3.1 Cieľová platforma

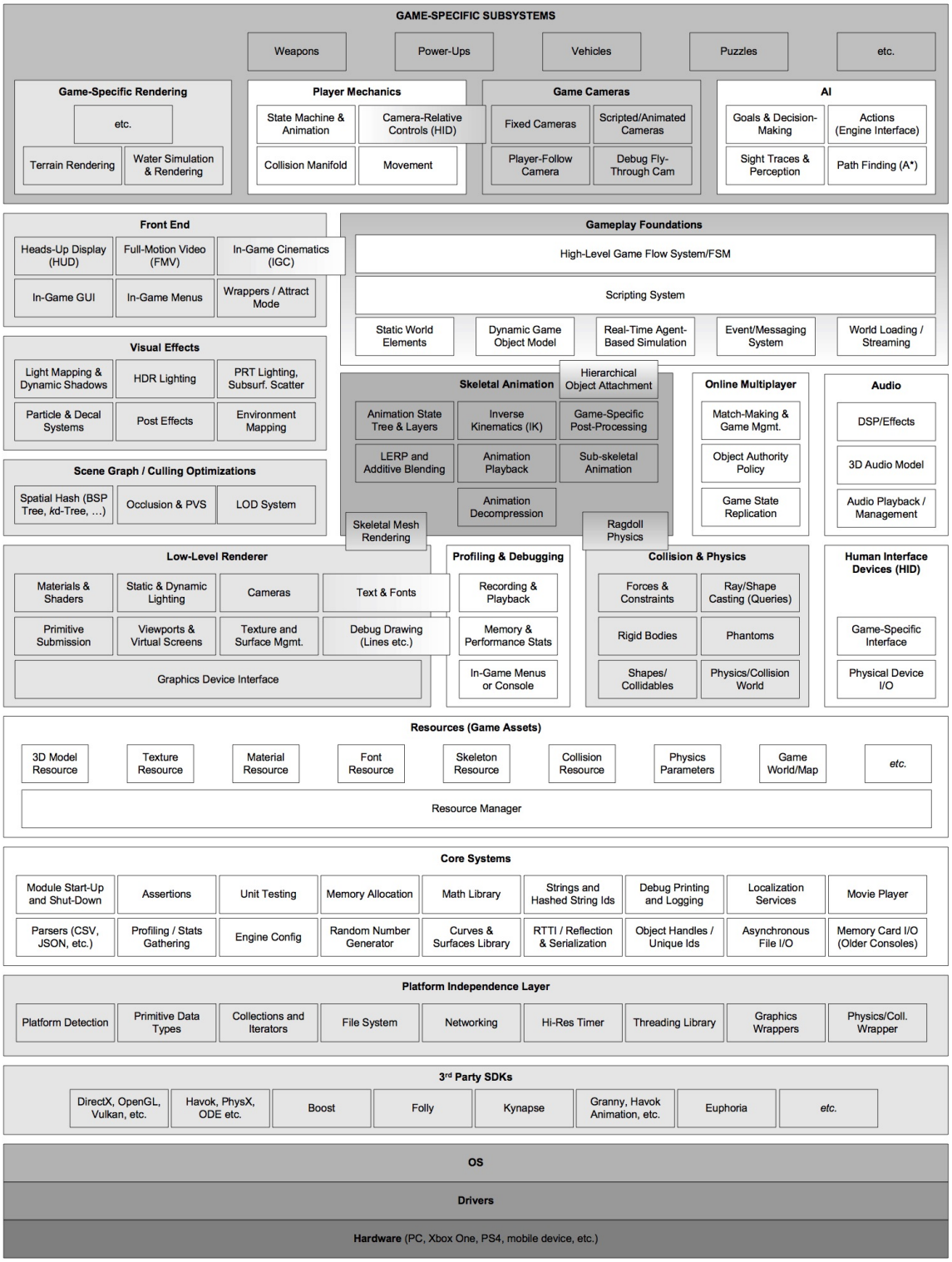
Najnižšou vrstvou, ktorú musí herný engine zohľadňovať, je cieľový hardware. V súčasnosti sa môže jednať o kancelárske notebooky až herné stolné počítače, herné konzoly, tablety či mobily. Rozdiely nie sú len v dostupnom hardware, ktorému sa musí hra prispôsobiť (napr. úrovne grafických detailov kvôli zachovaniu plynulosti a náročnosti), ale aj rozdielnymi inštrukčnými sadami a operačnými systémami. Dokonca aj v hardwarovej architektúre herných konzol rovnakej triedy rovnakej generácie sú podstatné rozdiely. Napríklad Xbox 360 okrem GPU poskytuje 3 rovnaké obecné PowerPC jadrá a 512 MB RAM pre kód, data aj grafiku. Naproti tomu jeho konkurent PlayStation 3, okrem GPU, poskytuje 1 obecné CPU a 6 špecializovaných jadier, 256 MB obecnej RAM a 256 MB video pamäte. V druhom prípade sa teda hrám vyplatí paralelizácia, ktorá však kvôli špecializácii jadier je o to náročnejšia. Taktiež je treba premyslenejšia správa pamäte.

2.3.2 Knižnice tretích strán

Na ušetrenie času a financií väčšina enginov využíva software tretích strán, napr. na vykresľovanie grafiky (DirectX, OpenGL), na kolízie a fyziku (Havok, PhysX) a na animácie (Granny, Havok Animation). Okrem týchto herne špecifických komponent často vývojári siahajú aj po alternatívach k štandardným knižniciam na dátové štruktúry a algoritmy. Pre C/C++ to môžu byť Boost či STLport. Štandardné implementácie im nemusia vyhovovať z pohľadu rýchlosti alebo využívania pamäte.

2.3.3 Platformovo nezávislá vrstva

Väčšina herných enginov je multiplatformná, aby ich hry boli dostupné pre viac hráčov. Pre vývojárov a designérov hry je potom výhodné nezaťažovať sa rozdielmi v platformách. Príkladom môže byť používanie vláken nezávislé na operačnom systéme, či práca s cestami k súborom, kde vývojár nemusí riešiť konkrétny oddeľovač (lomítko, spätné lomítko a iné). Ďalej môže byť výhodné vytvoriť abstraktnú vrstvu nad konkrétnou grafickou implementáciou či komponentou na fyziku (napr. PAL – Physics Abstraction Layer).



Obr. 2.1: Behová architektúra typického 3D herného enginu; prebrané z [24]

2.3.4 Core systems

Správa pamäte

Vzhľadom na ich dynamickú a real-time povahu, hry alebo komponenty si často sami spravujú pamäť, s cieľom efektívnejšieho spracovania. Je na to viac dôvodov: [25]

- Dynamická alokácia na hromade (heap) (funkcia alebo operátor new) je relatívne pomalá. Jednak je to kvôli tomu, že pri potrebe novej pamäte môže dôjsť k prechodu do režimu jadra a naspäť, čo je obecne časovo náročná operácia. Po druhé, štandardné implementácie musia byť dostatočne obecné, aby dokázali alokovať od niekoľkých bajtov až po gigabajty.

- Umiestnenie v pamäti ovplyvňuje rýchlosť. Prvky spracovávané spolu by mali byť ideálne blízko pri sebe v pamäti, čím sa dosiahne lepšieho využitia medzipamätí. Napríklad spracovávanie prvkov zoznamu, kde každý prvok môže byť umiestnený v inej stránke v pamäti, bude pomalšie, než keby boli prvky uložené v poli. To môže byť aj dôvod na vytvorenie vlastnej efektívnejšej dátovej štruktúry.

- Pri častej alokácii a dealokácii objektov nastáva fragmentácia pamäte, ktorá môže vyústiť do predčasného nedostatku pamäte, kedy nie je možné nájsť požadovane veľký spojitý blok pamäte. Oproti starším systémom, na moderných systémoch s podporou virtualizácie pamäte to už nepredstavuje taký problém.

Rýchlosť je teda hlavným dôvodom vzniku vlastných alokátorov. Najprv sa alokuje štandardným spôsobom (malloc(), new) veľký súvislý blok pamäte. Z tohto bloku sa potom vhodným spôsobom priraďuje pamäť na požiadanie. Alokácia a dealokácia teda potom prebieha bez prechodu do režimu jadra.

Typicky sa v engine využíva viac alokátorov súčasne, každý z nich môže byť optimalizovaný na iný typ požiadavkov, napr. podľa veľkosti požadovaných objektov, podľa doby životnosti objektov, rôzne zarovnané v pamäti apod. Takýto alokátor nemusí byť viditeľný programátorovi priamo, ale môže byť skrytý napríklad za factory metódu komponenty tretej strany, ako v knižnici Box2D na efektívnejšie vytváranie telies.[1]

Konkrétne typy alokátorov použiteľné v hrách:

- stack-based alokátor – funguje ako zásobník, kde dealokácie musia byť v presne opačnom poradí ako alokácie. Obe operácie predstavujú len posun ukazateľa vrcholu zásobníka. Prakticky sa dá použiť napríklad na prvotné alokovanie dát, ktoré žijú po celý čas hry. Následne, aktuálne hraná úroveň hry (level) sa na začiatku úrovne prialokuje, a na konci úrovne sa jedinou operáciou uvoľní. Takýto alokátor sa dá taktiež použiť na dočasné prvky používané vo výpočtoch v rámci jediného snímku hry, a pred ďalším snímkom sa môžu jedinou operáciou uvoľniť (single-frame allocator). Single-frame allocator je napríklad použitý v knižnici Box2D.

- double-ended stack – v jednom veľkom bloku pamäte z jednej strany rastie jeden zásobník a z druhej strany druhý. V hre môže jeden z nich predstavovať herné data aktuálnej úrovne, a druhý dočasné data aktuálne počítanej snímky. Ďalším príkladom použitia je, keď zásobníky obsahujú dočasné dáta aktuálnej a nasledujúcej snímky, a takto si medzi snímkami predávajú výsledky asynchrónnych operácií (napríklad dotazy na fyzikálnu komponentu). Na konci snímku sa role zásobníkov vymenia (zmenou príznaku) a dáta staršieho snímku sa uvoľnia jedinou zmenou ukazateľa vrcholu.

- object pool – jedná sa o veľké predalokované pole malých rovnako veľkých objektov, z ktorého sa pri požiadavke berie a pri uvoľnení sa vracia. Alokátor si môže viesť zoznam voľných objektov, v tom prípade má alokácia konštantnú zložitosť $O(1)$. Zoznam môže byť uložený mimo bloku, alebo ako súčasť (nepoužívaných) objektov, tzv. free list. Ak sa používané prvky dajú odlíšiť od nepoužívaných, cez prvky sa dá iterovať. Takto sa dajú implementovať napríklad časticové systémy, kde sa v jednej iterácii všetky častice aktualizujú a v ďalšej vykreslia. [25] V knižnici Box2D sa využíva varianta object pool, ktorá si udržuje viac menších pools s rôzne veľkými položkami.

Textové reťazce

V hrách sa vyskytujú dva typy textových reťazcov: interné pre vývojárov a tie, ktoré sú viditeľné užívateľovi. Každý z nich rieši iné problémy.

Medzi interné textové reťazce patria názvy súborov, identifikátory herných assetov, herných objektov, identifikátory textových reťazcov apod. Oproti identifikátorom celočíselného typu, vymenovaných hodnôt (enum) a GUID majú texty obrovskú výhodu zrozumiteľnosti pre človeka. Ich veľkou nevýhodou je rýchlosť práce s nimi za behu, ktorá má zložitosť $O(n)$ kde n je dĺžka reťazca. Nie je nevidené, aby sa vo funkciách na porovnávanie a vytváranie reťazcov trávilo najviac času v celom engine. Možným riešením je tzv. hashed string id, kde sa reťazce prepočítajú na (ideálne) unikátne celé číslo, s ktorým sa už dá pracovať rýchlo. Prepočet sa môže vykonať predspracovaním zdrojových kódov a/alebo za behu jednorázovo, za použitia hashovacej alebo CRC funkcie. Napríklad Unreal engine zapúzdruje túto funkcionalitu do triedy FName a označuje to ako name.

Pri textových reťazcoch viditeľných používateľom je hlavným problémom lokalizácia. Okrem vhodného kódovania textov je potreba, aby písmo použité v hre vedelo vykresliť všetky jeho znaky, aby grafické rozhranie počítalo s rôzne dlhými prekladmi, či dokonca rôznymi orientáciami textov (čínština – zhora dolu). Možnými riešeniami je použiť kódovanie UTF-8 alebo tzv. široké znaky. UTF-8 je spätne kompatibilné s reťazcami v ASCII, obecne môže používať premenlivý počet bajtov na znak a preto sa označuje ako multibyte character set (MBCS). Zástupcom širokých znakov je napríklad UTF-16, kde každý znak zaberá presne 2 alebo 4 bajty, označuje sa wide character set (WCS). Podľa označení MBCS a WCS sa potom dajú odlíšiť tie správne funkcie na prácu s nimi (porovnávanie, kopírovanie, konverzie...). Pre účely lokalizácie by sa nikdy nemali v zdrojových textoch vyskytovať tieto texty priamo, ale pomocou dohodnutej funkcie sa dynamicky zistia podľa identifikátoru reťazca, napr. String getLocalizedString(StringId id). Takáto funkcia získa odpoveď z databázy reťazcov podľa aktuálneho jazyka.

Ladenie počas vývoja

Vzhľadom na komplexnosť hier je nutnosťou, aby hra umožňovala rozumnými prostriedkami lokalizovať chyby a nedostatky. Okrem použitia debuggeru môže engine poskytovať tieto globálne dostupné prostriedky:

- textové výpisy – formátovaný výstup do konzol/y alebo súborov, viac úrovní detailov (verbosity), osobitné kanály výpisov pre rôzne komponenty vrátane filtrovania zobrazovaných kanálov, crash report s aktuálnym stavom enginu
- vykresľovanie – Môže byť názornejšie problém graficky vykresliť než analyzovať potenciálne tisíce riadkov výpisov. Vykresľovať sa dajú primitívne tvary, kolízne tvary

či drôtené modely objektov a texty, všetko na ľubovolnej pozícii v priestore, prípadne so zadanou životnosťou (viac než jedna snímka).

- vývojárske menu a konzola – umožňuje konfiguráciu hry za behu, zmenu hodnôt, zapnutie a vypnutie vlastností, ad hoc operácie
- pozastavenie herného sveta (ale nie enginu), snímka a záznam obrazovky
- profilovanie výkonu – vlastným meraním (napr. využitím QueryPerformanceCounter na Windowse), knižnicami (napr. PAPI) alebo profilovacími nástrojmi (napr. Intel VTune). Namerané hodnoty sa môžu v reálnom čase zobrazovať na obrazovke ako čísla či grafy, následne sa exportovať do súboru.

Dobrou praktikou na odchytávanie programátorských chýb je kontrola s využitím assert. Niekedy však ukončenie celej hry kvôli jedinému nedostatku nie je žiadané, napríklad keď je jediný 3D model chybný, tak by to zbytočne mohlo pozastaviť celý tím. V tom prípade môže byť užitočnejšie na nedostatok jasne poukázať, ale nechať hru bežať, napr. namiesto chýbajúceho modelu zobraziť iný, jasne nesprávny model alebo na danom mieste zobrazovať varovný text.

Ďalšie užitočné funkcie

Prakticky žiadna hra sa nezaobíde bez matematickej knižnice na vektory, matice, generátory náhodných čísel, kvaternióny a iné. Ďalej sú v enginoch prítomné parsery na rôzne formáty (XML, INI...) či nástroje na prácu s konfiguráciou enginu a hry (textové a binárne súbory, Windows registre...).

2.3.5 Resource manager

Každá väčšia hra si ukladá svoje herné dáta v externých súboroch, mimo hlavný spustiteľný súbor. Medzi tieto herné dáta, označované v angličtine ako resources alebo assets, patria grafické textúry, 2D a 3D geometria objektov (tzv. mesh), animácie, zvuky a hudba, kolízne a fyzikálne vlastnosti objektov, rozloženie herného sveta a ďalšie. Na jednoduchšie využívanie a efektívnu správu týchto dynamicky načítavaných súborov každá väčšia hra využíva v určitej forme tzv. resource manager. Ten môže pozostávať z off-line nástrojov využívaných počas vývoja hry, a on-line komponenty fungujúcej za behu hry.

Hlavnou úlohou týchto off-line nástrojov je správa a spracovanie zdrojových assets do podoby vhodnej pre engine. Konkrétne sa môže jednať o exportéry, ktoré z natívnych formátov rôznych editorov získajú súbory vo formátoch, s ktorými sa bude lepšie pracovať. Ak je to potreba, tak kompilátory môžu ďalej vhodne transformovať exportnuté súbory, ktoré sa nakoniec môžu zlinkovať alebo zbaliť s ostatnými spracovanými súbormi tvoriacimi logický celok. Pre komplexnejšie hry sa často využíva databáza assetov, ku ktorým sú naviazané metadáta, napríklad o spôsobe ich predspracovania. Extrémnym príkladom je UnrealEd, čo je editor pre Unreal Engine, v ktorom sa dajú assety rôznych typov prezerať, vytvárať a upravovať, taktiež predstavuje ich databázu v ktorej sa dá vyhľadávať, kontroluje integritu zložených assetov, a vytvára finálne archívy.

On-line resource manager komponenta okrem blokujúcich operácií so súbormi sú častými úlohami resource managera:

- jednotný prístup k súborom – na rôznych pamäťových médiách (pevný disk, DVD, online úložisko, pamäťové karty), na rozdielnych operačných systémoch rieši rozdielne oddeľovače priečinkov a zväzky
- podpora archívov – zložené assets môžu byť zbalené do jediného archívu, čo urýchli načítavanie a ušetrí úložisko, napr. v Ogre3D engine sa používajú ZIP archívy, v Unreal Engine PAK súbory
- asynchrónne / neblokujúce načítavanie súborov (tzv. streaming) – požiadavka na načítanie (zápis) súboru sa okamžite vráti do volajúceho kódu, ktorý môže pokračovať inou užitočnou prácou. Po dokončení požiadavky sa buď zavolá špecifikovaný callback, alebo ak volajúci má len limitované množstvo užitočnej práce, tak si počká (blokujúco) na dokončenie operácie. Ak sa na to nevyužíva existujúca komponenta a ani systém to nepodporuje, je to možné implementovať pomocou ďalšieho vlákna, ktoré vykonáva určené operácie blokujúco, zatiaľ čo volajúce vlákno pokračuje. Ak volajúci nemá čo robiť a potrebuje výsledok diskovej operácie, tak sa môže využiť semafór, na ktorom počká. Takto sa dajú implementovať hry takmer bez využitia nutnosti tzv. načítavacích obrazoviek, a vytvára sa tak dojem spojitého herného zážitku. Asynchrónne načítavanie dát sa v hrách využíva napríklad pri prehrávaní hudby v pozadí, videí, alebo počas hrania objemných levelov, ktoré by sa naraz nezmestili do pamäte.
- zaisťuje, že každý asset je načítaný v pamäti maximálne jedenkrát – aj keď existuje súčasne viac herných objektov využívajúcich daný súbor, tak všetky sa odkazujú na jedinú inštanciu v pamäti, tzv. Flyweight návrhový vzor [25]
- zaisťuje, že nepotrebné načítané súbory sa uvoľňujú z pamäte – ak je to však možné, tak aby sa minimalizovalo opätovné načítavanie v budúcnosti
- načítavanie zložených assets – napríklad ak sa načítavaný model odkazuje na svoje textúry a kolízne dáta, ktoré sú v iných súboroch, tak sa načítajú aj tie
- správne umiestnenie v pamäti – napríklad textúry a 3D mesh sa môžu ukladať do videopamäte namiesto RAM, fyzikálna komponenta môže vyžadovať zarovnané umiestnenie dát v pamäti kvôli efektívnejšiemu výpočtu
- predspracovanie načítaných objektov – napríklad výpočet potrebných hodnôt objektu kvôli spôsobu fungovania enginu, ktoré nemohli byť uložené v súbore na disku, pretože daný štandardizovaný formát ich nepodporuje

2.3.6 Fyzika v hrách

To čo sa v hrách označuje pojmom fyzika, sú v skutočnosti dva spolupracujúce systémy: systém na detekciu kolízií a simulácia fyzikálnych interakcií. Prvý z nich geometrickými operáciami zisťuje či a ako sa kolízne tvary objektov prekrývajú (v 2D) alebo zdieľajú spoločný objem (v 3D). Druhý zo systémov s pomocou numerickej integrácie pohybuje objektami podľa fyzikálnych zákonov s ohľadom na ich vlastnosti, a rieši ich prípadné kolízie.

Primárne riešeným problémom fyzikálnej komponenty býva dynamika pevných telies, ktorá pracuje s predpokladmi, že objekty sú pevné, nezničiteľné a nedeformujú sa pod vplyvom síl. Taktiež väčšinou zanedbáva relativistické deje, premenu energie na teplo a zvuk, či aerodynamiku. Výnimkou môžu byť simulačné hry, kde sa vývojári snažia zvyšovať realistickosť s ohľadom na obmedzený dostupný výkon.

So zvyšujúcim sa výkonom hardware a výskumom efektívnejších algoritmov sa okrem dynamiky pevných telies v hrách postupne objavujú aj ďalšie fyzikálne vlastnosti, ako deformovateľné telesá, realistická simulácia látok (oblečenia), vlasov a chlpov, vodného povrchu a dynamiky kvapalín.

Problémy s fyzikou

Fyzika v hrách nezvyšuje len vierohodnosť, ale umožňuje vznik aj novým mechanizmom hry, ako hlavolamy, a umožňuje kreatívne riešenia problémov pomocou objektov umiestnených vo svete. Zároveň tým, že kontrolu nad objektami preberajú fyzikálne zákony, vývojári a designéry hry o kontrolu prichádzajú. Nevhodne umiestnený objekt hráčom v hre môže spôsobiť, že naplánovaná explózia dopadne inak než vývojári čakali, a znemožní mu ďalší postup v hre. Nevhodne odstavené hráčovo vozidlo zase môže prekážať v naplánovanej ceste počítačom riadenej postave, ktorej pohyb je dôležitý pri rozprávaní herného príbehu. Možnými riešeniami takýchto prípadov sú:

- presunúť objekt programovo – napr. `setPosition(vectorPos)` – to však môže spôsobiť nestabilitu simulácie
- aplikovať impulz, silu alebo točivý moment – napr. `applyForce(vectorPoint, vectorPos)` – pokúsi sa o posun objektu rešpektujúc zákony
- filtrovanie kolízií – dočasne alebo natrvalo ignorovať kolízie vybraných objektov, teda im umožní cez seba prechádzať

Praktické používanie fyzikálneho podsystému

Väčšina komponent na simuláciu fyziky sa používa podobným spôsobom. Najprv sa vytvorí objekt zapúzdrujúci simuláciu, ktorý Havok a Box2D nazýva *world*, PhysX používa *scene* a ODE zase *space*. Cez tento objekt sa vytvoria telesá (*body*), ktoré sa skladajú z jedného alebo viacerých kolíznych tvarov (*shape*), ich transformácie (pozícia, natočenie) a fyzikálnych vlastností (hustota, trenie, pružnosť...).

Kolíznym tvarom býva primitívny konvexný tvar (kruh, obdĺžnik, n-uholnik, kapsula...). Konkávne a zložitejšie tvary sa vždy dajú rozložiť na viac jednoduchých konvexných. Požiadavka na jednoduchosť a konvexnosť je kvôli efektívnejšiemu výpočtu kolízií. Kolízny tvar sa zvyčajne nevykresľuje užívateľovi (len počas vývoja), ale mal by vhodne aproximovať vykresľovaný tvar, aby pre užívateľa nevznikali nečakané situácie.

Keďže sa jedná o numerickú integráciu, tak treba opakovane prepočítavať stav sveta, typicky volaním určenej metódy nad fyzikálnym svetom, často nazývaná update alebo step. Toto aktualizovanie treba podľa konkrétnej knižnice vykonávať 30 až 120krát za sekundu, ideálne s pevným krokom kvôli presnosti a stabilite numerickej integrácie. Po vypočítaní nového stavu sveta je potrebné, aby si tzv. herné objekty od fyzikálneho podsystému zistili nové transformácie objektov a podľa toho aktualizovali napríklad vykresľovanú reprezentáciu objektu.

Časté funkcionality

Kolízna komponenta väčšinou umožňuje dotazovanie na tzv. ray cast a shape cast kedy nás zaujíma, či by hypotetický paprsok (priamka) resp. tvar (kruh, kocka...) kolidoval s nejakými a akými objektami. Tieto objekty sa však do sveta nevytvárajú a taktiež nepôsobia žiadnymi

silami na dotýkajúce sa objekty. Táto funkcia sa využíva napríklad na zisťovanie viditeľnosti medzi agentami – zrak Ďalším špeciálnym typom objektu sú tzv. fantómy (angl. phantom; niekde aj senzor), čo sú dlhodobo umiestnené objekty s kolíznym tvarom (často pre hráča neviditeľné), ktoré však nemajú vplyv na dynamiku ostatných telies. Tie sa používajú na detekovanie objektov (hráča, NPC) v určenom priestore, čo môže následne vyvojať nejakú akciu.

Ďalšou z mnohých funkcionalít fyzikálnych komponent je filtrovanie a maskovanie kolízii: buď pevne určené (napr. hráčove časti tela spoločne nekolidujú pretože sa čiastočne prekrývajú), alebo dynamicky cez callbacky (pri kontakte sa zavolá funkcia, ktorá rozhodne, či sa kolízia bude ignorovať). Nemenej dôležitou je podpora rýchlo pohybujúcich sa telies (continuous collision detection (CCD) tiež známe ako time of impact (TOI)), napríklad guľky zo zbraní, pri ktorých by inak mohol nastávať neželaný jav známy ako tunelovanie.

Existujúce knižnice

Tak ako pri iných podsystémoch existujú aj fyzikálne komponenty od tretích strán, ktoré bývajú odladené a optimalizované nielen na viac platforiem, ale aj s podporou GPU a viacerých jadier.

Z univerzitného prostredia existujú knižnice čisto na kolízie SWIFT, V-Collide a RAPID. Plnohodnotnými open source knižnicami aj na dynamiku telies je ODE a Bullet. Bezplatne od spoločnosti NVIDIA je poskytovaná PhysX, ktorá podporuje aj GPU na urýchlenie a odľahčenie CPU. Dlhoročným komerčným štandardom je Havok.

Bezplatnými 2D alternatívami sú Box2D a Chipmunk2D.

2.3.7 Grafický podsystém (rendering engine)

Grafický podsystém býva jednou z najväčších a najkomplexnejších komponent každého herného enginu. Poskytuje jednotné rozhranie nad ľubovolným vykresľovacím hardvérom. Jeho základom je nízkoúrovňový renderer, ktorého úlohou je čo najrýchlejšie vykresliť kolekciu geometrických primitív, ako sú trojuholníky, čiary, body, textové reťazce, 3D mesh atď. Pritom zohľadňuje im priradené farby, textúry, transformačné matice, shadery, osvetlenie, tieňovanie atď. Vyššie vrstvy rozhodujú, ktoré primitíva sa vykreslia (angl. tzv. culling optimization), prípadne s akou úrovňou detailov (level of detail – LOD). Podľa [img-arch] sú nad týmto nízkoúrovňovým vykresľovaním vizuálne efekty, medzi ktoré patria časticové systémy (napr. na iskry, dym, oheň, explózie, ...), dynamické tiene, či post efekty (ako vyhladzovanie hrán, HDR, úpravy farieb...). Do poslednej vrstvy na vykreslenie, v literatúre označenou ako front end, patria (zväčša) 2D objekty staticky pozicované voči obrazovke nezávisle na polohe hráča v 2D či 3D svete, medzi ktoré patria grafické užívateľské rozhranie (GUI; napr. dočasné menu a dialógy), HUD (Heads-Up Display; trvalo zobrazené informácie pre hráča na obrazovke, napr. rýchlosť vozidla v závodných hrách), vývojárske informácie (grafy, vektory, vlastnosti objektov, ...) a prehrávanie dopredu natočených videí (tzv. FMV – Full-Motion Video).

2.3.8 Skriptovanie v hrách

Okrem jazyku, v ktorom je napísaný herný engine, a jazyku konfiguračných súborov, niektoré časti chovania hier bývajú v externých súboroch formou skriptov. Jednoznačne najpopulárnejším jazykom na skriptovanie v hrách je LUA, ktorá sa používa už vyše 20 rokov. Medzi výhody skriptovania v hrách sa uváza rýchlejší vývoj bez nutnosti prekladu zdrojo-

vých kódov v spojení s častou automatickou správou pamäte umožňuje aj navrhovať hru aj ľuďom bez technického vzdelania. Často tiež otvára možnosti komunite hráčov vytvárať vlastné komponenty a tzv. mody. Príkladom môže byť hra World of Warcraft, kde je len na stránke https://www.curseforge.com/wow/addons zhromaždených vyše 8 000 užívateľmi vytvorených addonov.

2.4 Herný cyklus (game loop)

Ako bolo v kap. 2.1 vysvetlené, väčšina videohier v reálnom čase reaguje na užívateľove vstupy grafickým a iným výstupom. V engine sa to prejavuje tak, že hra beží v nekonečnej slučke, kde v každej iterácii spracuje vstupy, podľa nich spraví krok simulácie a aktuálny stav hry vykreslí na obrazovku. Architektúra prakticky každej videohry vypadá nasledovne:

```
init();
while (! quit)
{
    processInputs();
    updateGame(dt);
    drawGame();
}
teardown();
```

V prípade využitia už existujúceho herného enginu samotný herný cyklus býva pred programátorom často skrytý a má nad ním len minimálnu kontrolu, napríklad len cez nastaviteľnú frekvenciu aktualizácie a/alebo prekresľovania. V hlavnom cykle sa trávi takmer 100 % času hry, zväčša s frekvenciou 30-120 iterácií za sekundu. Prvou časťou cyklu – spracovanie vstupov – býva často tzv. message pump, čo je obslúženie udalostí od operačného systému. Ide o ďalší cyklus, ktorý podľa typu prijatej správy, napr. stlačenie klávesy, vhodne zareaguje, správu prepošle do herného kódu, alebo správu ignoruje. Parametrom druhej časti iterácie - aktualizácie hry – býva čas dt (delta time), o ktorý sa má hra aktualizovať. Po správnosti by sa malo jednať o čas, ktorý potrvá spracovanie danej iterácie. To sa však dopredu nedá presne vedieť, takže sa buď používa konštanta alebo čas odvodený z predchádzajúcich snímok, napr. trvanie poslednej snímky/iterácie alebo priemer z posledných n snímok. Podľa spôsobu výpočtu dt a vzťahu aktualizácie a vykreslenia hry sú/boli najčastejšie tieto typu herného cyklu [21][25][24]:

- **typ 1** – bežať najrýchlejšie ako dokáže – Tento typ bol rozšírený v minulosti, kedy boli hry vytvárané na konkrétne hardvérové patformy, kde autor vedel, koľko času má na výpočet. Delta time tak bolo implicitné. Problém nastal, ak sa hra spustila na inak rýchlom hardvéri. Potom hra (simulácia) bežala pomalšie alebo rýchlejšie než bolo zamýšľané.

- **typ 2** – krok s uspaním – Pri tomto type herného cyklu môže byť hodnota dt konštantná a to v prípade, kedy je známa cieľová frekvencia hry, vyjadrovaná v snímkoch za sekundu – FPS – frames per second. Potom $dt = \frac{1}{FPS}$. Na dosiahnutie požadovanej frekvencie sa hra na konci snímky/iterácie uspí na taký čas, aby sa dosiahlo správne trvanie snímky. Ak je zapnuté VSync, teda hra má byť synchonizovaná s obnovovacoou frekvenciou monitoru, tak sa dá použiť trvanie predchádzajúcej snímky. Pri VSync sa o dosiahnutie požadovanej frekvencie postará blokujúce volanie funkcie, ktorá má na starosti zobrazenie snímky. Pri knižnici SFML je tou funkciou

`sf::RenderWindow::display()`. Naopak, ak jedna snímka hry trvá dlhšie než bol vyhradený čas, tak pri FPS režime sa hra spomalí (nedochádza k žiadnemu uspaniu) a pri VSync režime sa musí počkať až do zobrazenia nasledujúcej snímky. Možnosť prepínať medzi režimami FPS a VSync býva v hrách bežnou. Tento typ herného cyklu je použitý napríklad v engine Game Maker.

- **typ 3** – oddelenie update a draw – V súčasných hrách sa často najviac času z celej snímky strávi vo vykresľovacom kóde. Ak hardvér nestíha, tak z pohľadu hrateľnosti je prijateľnejšie, ak sa zníži frekvencia prekresľovania a zachová sa rýchlosť simulácie, než keby sa spomalila celá hra. Udržanie správnej rýchlosti aktualizácie je tiež dôležité pri online hrách. To sa dosiahne oddelením frekvencie aktualizácie a prekresľovania. Dalo by sa tiež na to dívať tak, že vykresľovanie je producent času a aktualizovanie sveta je jeho konzument po pevných krokoch. Čas si vymieňajú cez akumulátor, v nasledujúcom kóde nazývaný premennou lag. Konštantný krok zabezpečí stabilnú fyzikálnu simuláciu a deterministické chovanie, čo sa využíva napr. pri záznamoch hier alebo online hraní. Tento typ cyklu, rovnako ako aj typ 4, predpokladá, že aktualizácia sveta o *dt* modelového času zaberie omnoho menej než *dt* reálneho času. Inak hrozí tzv. spiral of death, kedy hra nakoniec prestane reagovať, pretože nestíha „dobehnúť“ čas. Proti tomu sa dá brániť obmedzením vnútorného cyklu na maximálny počet iterácii, po ktorého prekročení sa hra len spomalí. Ďalším užitočným vylepšením je obmedziť prírastky akumulátoru na určitú maximálnu hodnotu, čím sa zabráni neželaným javom pri pozastavení procesu pri ladení cez debugger.

```
double previous = getCurrentTime();
double lag = 0.0; // akumulator
while (! quit)
{
    double current = getCurrentTime();
    double elapsed = current - previous;
    previous = current;
    lag += elapsed;

    processInputs();

    while (lag >= STEP)
    {
        updateGame(STEP);
        lag -= STEP;
    }

    drawGame();
}
```

Výpis 2.1: Kód herného cyklu typu 3, kvôli prehľadnosti bez vylepšení proti spiral of death a debuggeru; prebrané a upravené z [25].

- **typ 4** – interpolovať medzi stavmi – V súčasnosti sa rozširujú herné monitory s frekvenciou 120Hz a viac. V prípade, ak by sa hra aktualizovala na nižšej frekvencii než monitor, napr. 60x pri 120Hz monitore, tak pri cykle typu 3 by boli každé 2 po sebe nasledujúce snímky vizuálne rovnaké. Riešením je interpolovať medzi dvoma poslednými

snímkami podľa hodnoty v akumulátore. Aby však bola interpolácia stavov možná, tak si hra/engine musí byť schopná pamätať predchádzajúci stav hry, a vykresľovacie rutiny musia vedieť interpolovať.

2.4.1 Paralelizácia herného cyklu

Paralelizácia herného cyklu kvôli lepšiemu využitiu hardvéru a vyššiemu výkonu je možná viacerými spôsobmi. Na najnižšej úrovni je možné využiť vektorové inštrukcie – SIMD – single instruction multiple data, kedy sa rovnaká inštrukcia aplikuje na viac dát súčasne. Na umožnenie vektorizácie kódu treba brať ohľad na správne umiestnenie dát v pamäti, s čím často súvisí aj úprava samotného algoritmu. Ďalej je možné využiť princíp fork & join, kedy sa určité časti iterácie, napríklad fyzikálna simulácia, rozdelí na menšie časti, ktoré sa dajú počítať paralelne. Konkrétne pri fyzikálnej simulácii sa využíva koncept tzv. ostrovov telies, kedy existuje viac zhlukov telies, pričom každý zhluk sa dá počítať nezávisle od ostatných zhlukov a len s minimálnymi dátovými závislosťami medzi zhlukmi, čo otvára priestor na paralelizáciu. Ďalšou možnosťou je vykonávať každý podsystém enginu vo vlastnom vlákne. To však často nie je jednoduché, pretože podsystémy spolu komunikujú, čo vyžaduje synchronizáciu. Nakoniec je možné prácu rozdeliť na menšie úlohy, v literatúre [24] označované ako jobs. Tieto miniúlohy, s dĺžkou výpočtu v mikrosekundách či až milisekundách, sa najprv zoradia podľa ich vzájomných závislostí a potom vykonávajú na dostupných procesoroch/jadrách.

2.4.2 Hry po sieti s viac hráčmi

Podpora viacerých hráčov súčasne (multiplayer) výrazne ovplyvňuje architektúru hry a hra musí riešiť ďalšie problémy. Hlavným problémom pri hrách po sieti býva rýchlosť odozvy, ktorá je najhoršia pri akčných a závodných hrách, kde latencia na úrovni desiatok milisekúnd už zhoršuje hrateľnosť. Okrem problémov s odozvou je veľký problém podvádzanie hráčov. Ak by simulácia herného sveta prebiehala u klienta, tak by si existujúcimi nástrojmi mohol hráč napríklad pridať nekonečné množstvo munície, byť nesmrteľný alebo sa pohybovať neprirodzene rýchlo. Medzi najčastejšie prístupy patrí klient-server a peer-to-peer. [24][18] [22].

Pri prvom prístupe býva klient relatívne „hlúpy“ – len posiela vstupy a prijíma nový stav, ktorý vykreslí a prehrá audio. Kvôli lepšej odozve sa hry snažia predikovať budúce stavy. V prípade zlej predikcie, teda keď príde zo serveru iný stav než predvídala hra, tak svoj stav zahodí a použije nový, pričom takéto skokové zmeny pôsobia na hráčov rušivo. Server podľa prijatých vstupov od hráčov simuluje herný svet a pripojeným hráčom posiela stav hry. Server môže bežať buď na dedikovanom počítači alebo na počítači jedného z hráčov. V druhom prípade môže bežať v osobitnom procese, osobitnom vlákne, alebo kvôli ušetreniu výkonu kvôli synchronizácii može bežať v rovnakom vlákne ako klient – v hernom cykle sa najprv vykoná serverový kód, potom kód klienta. Tzv. *client-on-top-of-server* označuje prípad, kedy dokonca aj lokálna hra pre jedného hráča je implementovaná ako klient-server s jediným hráčom. Server všeobecne môže bežať na inej frekvencii než hra u klienta, napr. pri hre Quake bežal na 20 FPS a pri súčasnej hre Valorant beží na 128 FPS. [24][18][21].

Pri peer-to-peer prístupe každý hráč vystupuje čiastočne v role serveru aj klienta. Každý objekt v hre je vlastnený práve jedným hráčom. Takže pre objekty, nad ktorými má hráč autoritu, tak sa chová ako server, a nad ostatnými ako klient, ktorých stav prijíma od ostatných hráčov, ktorí ich vlastnia. Z toho vyvstávajú nové komplikácie, napríklad ak

hráči počas hry odchádzajú alebo sa pripájajú noví, tak musia objekty migrovať medzi strojmi.[24][18] [21].

Kapitola 3

Umelá inteligencia v hrách

Aj napriek obrovským pokrokom v oblasti umelej inteligencie (ďalej len AI – artificial intelligence) videohry využívajú len časť dostupných techník a algoritmov. Je to dané rozdielnymi cieľmi a požiadavkami herného priemyslu a akademickej sféry. Rozdiely v týchto paralelne sa vyvíjajúcich odvetviach budú popísané ďalej. Potom sa načrtnú 3 základné oblasti využitia AI v hrách. Nasledovať budú charakteristiky AI problémov a algoritmov v 3.4. Nakoniec budú spomenuté techniky v hrách používané, všetko spolu s reálnymi príkladmi. Ak nie je spomenuté inak, tak kapitola primárne vychádza z knihy [35].

3.1 AI v akademickej sfére

Umelá inteligencia v hrách bola spočiatku vyvíjaná na hranie doskových hier, konkrétne piškvorky, dáma, šach či backgammon. Jednalo sa teda o ťahové hry, kde obe strany poznali kompletný stav hry, a na rozhodovanie mali relatívne dostatok času (aspoň jednotky až desiatky sekúnd). Po úspechoch sa postupne výskumníci presúvali na ťažšie hry, v súčasnosti napríklad Go, kde algoritmus AlphaGo od DeepMind (vlastnené Google/Alphabet) v roku 2017 dokázal poraziť svetového majstra, bežiac na jedinom počítači (narozdiel od superpočítača).

Okrem doskových hier bol v roku 2011 dosiahnutý aj úspech v Jeopardy!, čo je televízna vedomostná súťaž s otázkami formulovanými v ľudskej reči. Software Watson od IBM, bežiaci na superpočítači s naštudovanými miliónmi dokumentov, dokázal poraziť viacerých vrcholových súťažiacich.

Za väčšou výzvou sa výskumníci presúvajú na videohry iné než doskové hry, kde sa už často nejedná o ťahové hry, časť informácií je skrytá, reagovať treba v reálnom čase a stav hry je reprezentovaný rádovo rozmernejšími vektormi (viac o týchto charakteristikách v kap. 3.4). Príkladom sú hry Starcraft II (real-time strategická hra založená na budovaní a ovládaní bojových jednotiek) a TORCS (závodný simulátor automobilov s realistickým fyzikálnym modelom).

Cieľom hernej AI z akademického výskumu je teda vyhrať hru, a to buď tak efektívne ako to len ide, alebo pod určitými obmedzeniami (napríklad s obmedzeným počtom akcií za čas, aby pripomínali ľudského hráča). Naproti tomu v hernom priemysle nejde o to vždy vyhrať, dokonca to môže byť neželané alebo z princípu hry nemožné.

3.2 AI v hernom priemysle

Hlavnou požiadavkou na hernú AI v hernom priemysle nie je výhra, ale zábavnosť pre (ľudského) hráča. Hráč by mal byť schopný AI poraziť, čo by nemalo byť ani príliš náročné ani príliš jednoduché. Podľa literatúry by sa dala hra definovať ako „interaktívny zážitok poskytujúci hráčovi stále ťažšie a ťažšie výzvy, ktoré sa učí až kým ich nezvládne“.[24] S tým korešponduje aj požiadavka z praxe, aby správanie AI bolo aspoň čiastočne opakujúce sa, a pre hráča predvídateľné. Ďalej, aby bolo správanie AI pre hráča prirodzené a uveriteľné, tak zväčša nemôže byť striktne optimálne. V praxi sa to môže prejaviť tak, že AI ovládaný protivník sa nebude pohybovať po najkratšej trase z miesta A do B a výber ďalších akcií mu nepotrvá len nevyhnutne krátku dobu, ale že sa bude pohybovať po krivke bez ostrých zatočení, s prípadným náhodným blúdením, a premýšľanie mu zaberie určitý čas. Tým sa dosiahne nielen lepšej uveriteľnosti, ale AI sa stáva zaujímavejším a vyrovnaným súperom.

Ďalšou požiadavkou hier na AI je reagovanie v reálnom čase, často v rámci niekoľkých jednotiek až desiatok milisekúnd, pretože okrem AI sa hra musí starať aj o ďalšie svoje podsystémy, ako je grafika, fyzikálna simulácia, herná logika... Našťastie, nie všetky výpočty AI je potrebné opakovať v každej snímke hry, a teda sa môžu počítať v paralelnom vlákne, prípadne ich výpočet môže byť prekrytý animáciou (napr. prepočet dynamického navmesh počas animácie explózie steny a usadnutia trosiek [33]).

V komerčnom hernom priemysle bývajú často tesné termíny, a teda málo času na vývoj. Okrem AI je treba vyvinúť aj mnoho ďalších obsiahlejších súčasti hry, ako je návrh herného sveta, jeho grafiku, tisíce predmetov, či hodiny hudby a zvukov; na AI teda nezostáva moc zdrojov. Preto vývojári zostávajú pri jednoduchých, overených a dobre testovateľných AI technikách, ktoré by sa na akademickej pôde za umelú inteligenciu ani nepovažovali. Veľmi častými sú konečné stavové automaty a rozhodovacie stromy (implementované ako vnorené if-else príkazy). Ďalej sa používajú behaviorálne stromy či fuzzy logika. Tie sú zrozumiteľné aj pre netechnických návrhárov a majú deterministické chovanie.

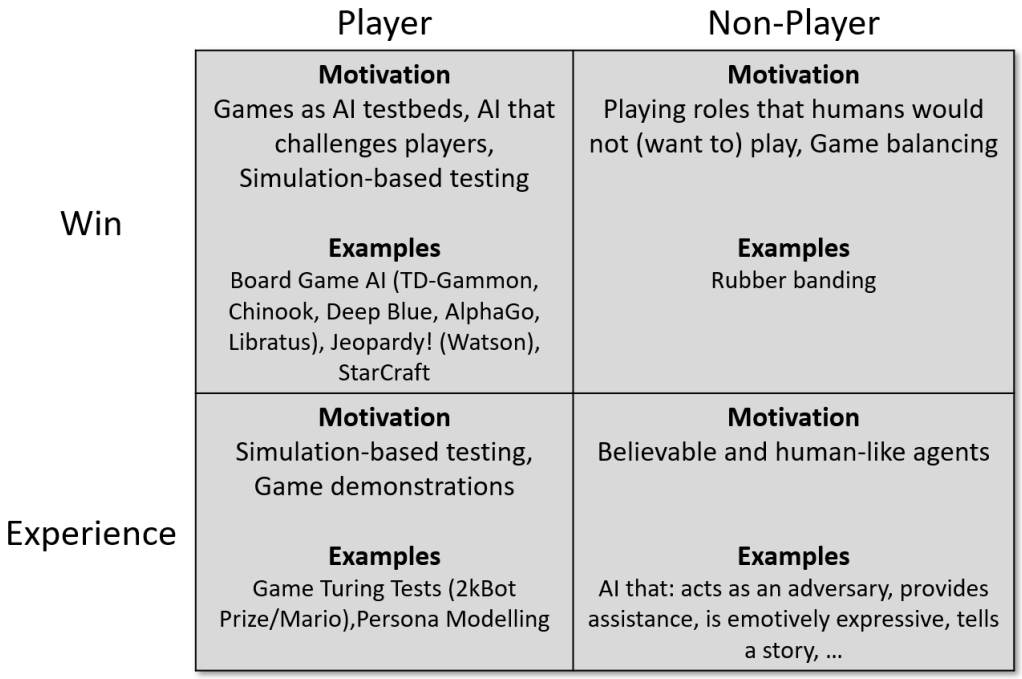
Ak sa v hre používa pokročilejšia technika umelej inteligencie, tak sa často predáva ako hlavná vlastnosť hry, napr. neurónové siete v hre Creatures (1996) na správanie evolučne vyvinutých tvorov. Ďalším príkladom je <reinforcement-learning> v hre Black and White (2000) na učenie stvorení cez odmeny a tresty od hráča, spolu s BDI modelom na ich rozhodovanie. Nevýhodami pokročilejších AI techník je dlhší vývoj, často vyššia výpočtová a/alebo pamäťová náročnosť, a nedeterminizmus možných výsledkov u hráčov. Vývojári tým riskujú, že sa algoritmus u hráča naučí neočakávané správania s negatívnym dopadom na hrateľnosť.

3.3 Využitie AI v hrách

Techniky z oblasti umelej inteligencie si v hrách nájdu rôzne využitia: hľadanie cesty priestorom a jeho navigácia, procedurálne generovanie sveta, analýza hráčových reakcií, skupinové správanie NPC, učenie a plánovanie NPC... Obecne by sa dali určiť 3 oblasti využitia AI v hrách: hranie hier, generovanie obsahu, a modelovanie hráčov. Tieto oblasti sa však často používajú spolu, napríklad dynamická zmena náročnosti hry podľa schopností hráča.

3.3.1 AI na hranie hier

Túto oblasť by bolo možné rozdeliť podľa cieľa AI na výhru a zážitok, a podľa role AI na hráča a nehráča (NPC - non-player character - počítačom riadená postava). Tým vzniknú 4 kategórie využitia AI na hranie hier, viz obrázok 5.3.



Obr. 3.1: Hranie hier podľa cieľa (výhra, zážitok) a role (hráč, nehráč). Prevzaté z [35]

Prvú kategóriu (hráč, výhra) predstavujú náročné hry, ktoré posúvajú vedecký výskum dopredu. Väčšina vývoja sa sústredí na samotnú výhru, pričom testovacie prostredie je len jednoduché (napr. doskové hry), alebo bolo pôvodne vytvorené pre ľudských hráčov (napr. hra StarCraft). Vzhľadom na väčší priestor sa tu uplatňujú pokročilejšie technógie, ako Monte Carlo tree search a hlboké neurónové siete.

Do skupiny hráč + zážitok by sa dali zaradiť hlavne online počítačom riadení protivníci s cieľom, aby počítačový protivník bol pre ľudského kráča na nerozoznanie od človeka, tzn. prejde Turingovým testom. Príkladom je svetová súťaž 2K BotPrize, kde Turingov test predstavuje hra Unreal Tournament 2004 (strieľačka z pohľadu z prvej osoby) a úlohou rozhodcov je odlíšiť ľudských a počítačom riadeným hráčov.

Kategória nehráč + výhra by sa dala brať ako základ pre najčastejšiu kategóriu nehráč + zážitok. Až keď je počítač schopný vyhrať proti ľudskému hráčovi, až potom sa môžu naň klásť ďalšie obmedzenia za účelom vyvažovania obtiažnosti hry. Napríklad technika rubber banding v závodných hrách umožňuje NPC (non-player character) nepozorovane prispôsobiť svoju rýchlosť ľudskému hráčovi, aby nebola príliš ďaleko pred ani za hráčom.

Pod pojmom umelá inteligencia v hrách sa najčastejšie myslí, že AI hrá v role nehráča za účelom vyvolania dobrého herného zážitku, na obrázku 5.3 časť vpravo dole. Ako bolo vysvetlené v kap. 3.2, využívajú sa na to jednoduchšie, predvídateľné, človekom ručne navrhnuté modely správania (behaviorálne stromy, . . .).

3.3.2 AI na generovanie obsahu

Z podľadu AI by sa dalo na procedurálne generovanie obsahu (PCG – procedural content generation) dívať ako na hľadanie riešenia, ktoré spĺňa určité kritériá a/alebo maximalizuje určitú metriku. Riešením sa v tomto prípade myslí najčastejšie herný svet (prostredie s umiestnenými hernými objektami), ale môže sa jednať aj o generovanie hudby, úloh pre hráča, herných objektov, nepriateľov apod. Kritériami a metrikami môžu byť: prístupnosť všetkých častí vygenerovaného prostredia aj s jeho objektami, pestrosť prostredia bez viditeľne opakujúcich sa prvkov, požadovaná veľkosť prostredia, počet umiestnených nepriateľov... Riešenia sa teda skladajú z vopred ľuďmi navrhnutých častí, ako sú bloky prostredia, atribúty a schopnosti nepriateľov. Netriviálne metriky, ako pestrosť/zaujímavosť prostredia, sa môžu vyhodnocovať modelom natrénovaným na človekom vytvorených a anotovaných dátach. Takéto generovanie a overovanie riešenia sa najčastejšie odohráva u hráča na začiatku hry. Generovanie môže byť ďalej deterministické (a teda zopakovateľné), alebo stochastické.

Ako výhoda PCG namiesto ručného vytvárania herného obsahu sa v literatúra uvádza zníženie nákladov na výrobu, unikátne prostredie pri každom spustení hry, a redukcia veľkosť hry na disku alebo pri prenose po sieti.

Historicky ako jedna z najstarších hier využívajúcich PCG sa uvádza hra Rogue (1980), za ňou nasledujúca populárna séria Diablo, v ktorých sa procedurálne generuje časť herného sveta. V online hre No Man's Sky (2016) sa deterministicky generuje vesmír až s možnými 2^{64} planétami, pričom na serveroch sa namiesto celých planét ukladajú len ich počiatočné hodnoty, ktoré ich vygenerovali, spolu so zásadnejšími zásahmi od hráčov. V jednej z najpredávanejších hier sveta Minecraft (2011) sa dokonca od hráča pri vytváraní hry explicitne požaduje počiatočná hodnota (tzv. seed) v podobe ľubovoľného reťazca. Extrémnym prípadom je 3D strieľačka .kkrieger s veľkosťou 96 kB, ktorá si pri načítaní vygeneruje herný svet, textúry, 3D objekty a dokonca aj hudbu.

3.3.3 AI na modelovanie hráčov

Do tretice sa pomocou AI analyzujú hráčove emócie a správanie v hre.

To môže prebiehať pred/počas vývoja hry, napríklad počas alfa a beta testovania, a podľa toho sa redesignuje a balancuje herný obsah, napríklad obtiažnosť konkrétnych protivníkov. Do tejto kategórie patrí okrem iného aj vypĺňanie dotazníkov po hraní.

Modelovanie môže prebiehať aj počas hrania u konečného zákazníka, hlavne za účelom vyvažovania náročnosti, tzv. dynamic difficulty adjustment alebo dynamic game balancing. Jedným z príkladov je rubber banding v závodných hrách z kap. 3.3.1. V hre Left 4 Dead (2008) systém nazvaný AI Director sleduje stav hráčov a striedavo stochasticky generuje vlny nepriateľov.[19] Extrémnym prípadom je thriller hra Nevermind (2015), ktorá podľa fyziologickej a emočnej odozvy hráča viditeľne upravuje hru. Na meranie fyziologickej reakcie podporuje niekoľko bežne predávaných senzorov srdečnej aktivity. Emočnú reakciu zisťuje z výrazov tváre snímanej pomocou bežnej webkamery. Hra však funguje aj bez týchto technológií. [6]

3.4 Vlastnosti riešených problémov a algoritmov

Na výber vhodných algoritmov je treba najprv dobre pochopiť riešený problém. Ten sa dá charakterizovať viacerými vlastnosťami popísanými nižšie.

3.4.1 Dostupnosť informácií (observability)

Podľa toho, koľko zo stavu hry je známych hráčovi, tak môžu byť problémy (hry) s úplnou informáciou (doskové hry, Pac-Man) a s neúplnou informáciou (poker, väčšina videohier).

V strategických videohrách sa takáto nemožnosť všetko vidieť označuje ako fog of war – často hráč môže vidieť len územia (statické prostredie) ktoré sám už niekedy navštívil svojimi jednotkami, a aby videl aj nepriateľské jednotky (dynamické prvky) na danom území, musí tam mať v danom čase umiestnené vlastné jednotky.

Najjednoduchším riešením je tieto skryté informácie z pohľadu AI ignorovať. Niektoré hry je však väčšinou nemožné vyhrať len s dostupnými informáciami, ako poker a strategické videohry. Preto musí AI modelovať aj tieto skryté informácie, čo vo výsledku podstatne zvyšuje náročnosť riešenia.

3.4.2 Determinizmus

Podľa toho, či je výsledok hry daný jedine hráčovými akciami, alebo aj niečím iným, tak hry môžu byť deterministické a nedeterministické (stochastické). V nedeterministickej hre ak hráč hrá opakovane rovnakú hru s rovnakými akciami v rovnakom čase, tak hra môže dopadnúť inak. Tento prvok náhody môže byť v hre zastúpený náhodným pohybom nepriateľov, hádzaním kockou či ťahaním (zamiešaných) kariet, teda niečím čo znemožňuje alebo sťažuje plánovanie. Algoritmy na hranie stochastických hier preto musia byť o to robustnejšie.

3.4.3 Faktor vetvenia

Pojmom faktor vetvenia (branching factor) sa označuje počet akcií, medzi ktorými sa hráč rozhoduje v každom kroku hry. V mobilnej hre Flappy Bird (2013) sa hráč v každej chvíli hrania môže rozhodnúť, či dotykom obrazovky zamáva krídlami aby trochu vzlietol, alebo nie a ostane padať – má teda branching factor 2. V šachu sa hráč v každom kole môže rozhodovať priemerne medzi 35 možnými ťahmi.

Faktor vetvenia sa môže počas hry meniť. Napríklad v hre Go sa postupne znižuje (pri prvom ťahu má hráč 400 možností, potom klesá), v šachu je najvyšší v strede hry, a vo videohrách, kde je možné učiť sa nové akcie a zbierať predmety, tak rastie. V tom prípade sa uvádza jeho priemerná hodnota, napr. pre šach 35, pre Go 250.

V prípade videohier má hráč často na výber nie len z niekoľkých diskrétnych možností, ale môže zadávať takmer spojité vstupy. Napríklad v 3D strieľačkách si plynule myšou môže vybrať ako presne sa pred výstrelom natočí, a v strategických hrách si vyberie ktorú podmnožinu z desiatok jednotiek pošle na kurzorom vybrané miesto na rozľahlej mape. V takýchto prípadoch sa toto číslo môže pohybovať aj v miliónoch a viac.

Ak sa teda v každom z d kôl hry môže hráč rozhodovať medzi b rôznymi akciami, takto exponenciálne narastajúci stavový priestor o veľkosti b^d nie je možné prehľadávať hlbšie na viac než pár úrovní, niekedy ani len prvú.

3.4.4 Granularita času

Podľa plynutia času je možné hry rozdeliť na ťahové a real-time hry. V ťahových hrách, ako šach, sa hráči zväčša striedajú po kolách, pričom trvanie jedného kola nemá vplyv na hru (s výnimkou vypršania časového limitu a „straty ťahu“). Real-time hry, napríklad stratégia

StarCraft alebo závodné simulátory, sa snažia o spojitý čas, čo sa na technickej úrovni dosahuje veľmi rýchlym aktualizovaním stavu hry (cca 30-120krát za sekundu).

Dôležité je však poznamenať, že jednak v real-time hrách hráč nevykonáva akciu v každom možnom ťahu (aktualizácii/snímke hry), tak ako v ťahových hrách. Taktiež v real-time hrách akcie len veľmi pomaly menia stav hry – 10 ťahov v šachu môže rozhodnúť celú hru, ale 10 akcií v real-time hre môže predstavovať len presun jednotiek alebo kamery naprieč hracím poľom.

Nezávisle na frekvencii alebo významnosti akcií v real-time hrách stále nie je možné prehľadávanie stavového priestoru ďalej do budúcnosti. Riešením môže byť plánovanie na úrovní makroakcií/stratégií, ktoré by sa potom skladali z viacerých jednoduchších akcií.

3.4.5 Forward model a učenie

Dôležitou vlastnosťou hry je, či je možné počas hrania použiť <forward model> t.j. model, ktorý zo stavu s pri akcii a dosiahne rovnakého stavu s', ktorého by dosiahla hra zo stavu s pri akcii a. Algoritmus si teda môže počas hry odsimulovať viaceré akcie alebo postupnosti akcií, a až tú najlepšiu akciu/postupnosť z nich vykoná v hre. Nutnou podmienkou forward modelu je jeho rýchlosť, aby bol prehľadávací algoritmus schopný odsimulovať množstvo stavov a akcií. Bez forward modelu nie je možné prehľadávanie stavoveho priestoru.

Bohužiaľ, väčšina videohier je tak komplexná a jednotlivé podsystémy tak previazané, že je veľmi ťažké separovať forward model na účely AI. Riešením tak ostáva aproximovať forward model, pričom kritickou požiadavkou zostáva jeho rýchlosť.

Okrem forward modelu je ďalším riešením naučenie modelu dopredu. Keď sa potom od AI počas hry očakáva rozhodnutie, tak nemusí simulovať možné akcie, ale len relatívne rýchlo vyberie tú najlepšiu podľa toho, ako bol model predtým natrénovaný.

Oba prístupy – forward model a naučený model – je možné aj kombinovať.

3.4.6 Stav hry ako vstup pre AI (Observation Space)

Tak ako človek vníma stav hry cez jej grafický, zvukový, prípadne iný výstup, tak AI tiež potrebuje hru nejako vnímať, aby mohla vhodne reagovať. Človeku prirodzený grafický rastrový výstup je pre počítač relatívne redundantným a náročným na spracovanie. Avšak aj s týmto prístupom sa dosahujú úspechy, napríklad na závodnej hre TORCS, alebo na starších hrách ako Super Mario. Dôvodom na hranie podľa pixelov býva snaha o všeobecnú inteligenciu (general intelligence), kedy by sa AI naučila hrať ľubovoľnú hru tak ako človek. Ďalším dôvodom môže byť nedostupnosť API pre AI. Vo väčšine prípadov však AI má k dispozícii stav hry v zjednodušenej a predspracovanej forme, niekedy označované aj ako pozorovanie (observation), napríklad extrahované číselné pozície prekážok, pozíciu a rýchlosť hráča a ostatných nepriateľov, zoznam viditeľných objektov do určitej vzdialenosti, vzdialenosť k prekážke v určitom smere, diskretizovaná prístupnosť prostredia [26], či textové správy od hráča (v hre Quate III Arena)[34]. Redukcia množstva vstupných informácií a výber len tých relevantných obecne zrýchľuje učenie tých metód, ktoré vyžadujú trénovanie. Určiť, ktoré informácie sú relevantné na daný AI problém, môže intuitívne človek, pomáhať si môže metódami z oblasti data mining (korelačná analýza, ...), a hlavne praktickým testovaním a experimentovaním, aby dosiahol čo najlepšie výsledky.[35][26]

3.5 Prehľad používaných AI metód v hrách

Táto kapitola predstavuje prehľad najčastejšie používaných techník umelej inteligencie v hrách, podľa [35]. Nesnaží sa spomínané algoritmy podrobne vysvetľovať, ale len načrtnúť ich princíp a základné rysy, aby sa prípadný čitateľ trochu zorientoval v hernej AI. Podrobnejšie sú vysvetlené v zdrojovej literatúre, avšak aj tá ďalej odkazuje na podrobnejšie výklady. Niektoré zo spomenutých algoritmov patria do viacerých kategórií súčasne, napr. neurónové siete sa primárne zaraďujú do „učenia s učiteľom“, ale ich varianta SOM a autoenkodéry sa dajú zaradiť pod „učenie bez učiteľa“.

Väčšina z týchto algoritmov vyžaduje od používateľa určitú formu ohodnotenia riešenia, ktorú sa snaží maximalizovať (nazývané ako užitočnosť, odmena, ohodnotenie či fitness) alebo minimalizovať (cena, chyba).

Na zlepšenie výsledku sú samotné algoritmy často obohatené o znalosti domény (napr. osvedčené makroakcie) a/alebo sa kombinujú s ďalšími AI technikami. Dobrým príkladom môže byť AlphaGo (2015–...), čo je svetovo úspešný softvér na hranie hry Go, ktorý v sebe kombinuje stromové prehľadávanie algoritmom MCTS, spolu s hlbokými neurónovými sieťami trénovanými na zápasoch profesionálnych ľudských hráčov, spolu s posilovaným učením hraním proti sebe samej.[30]

3.5.1 Ad-hoc chovanie

Najčastejším typom umelej inteligencie v hrách je človekom ručne navrhnuté chovanie, v zdrojovej literatúre označované pojmom ad-hoc behaviour authoring, jedná sa vlastne o expertný znalostný systém. Typickými príkladmi z tejto skupiny algoritmov sú populárne konečné stavové automaty (FSM), behaviorálne stromy (BT) a utility-based AI. Tieto prístupy teda nevyžadujú trénovanie ani prehľadávanie stavového priestoru za behu (teda ani forward model). Ich výhodami sú relatívne jednoduchý návrh aj pre netechnických tvorcov (často prostredníctom grafického rozhrania namiesto programovania), a predvídateľnosť chovania za behu, teda možnosť odhaliť možné chyby.

Veľmi často využívané, aj v minulosti aj súčasnosti, sú konečné stavové automaty (Finite State Machine – FSM). Jedná sa o orientované grafy, pozostávajúce z uzlov predstavujúcich stav (stav hry, objektu, animácie...), ktoré sú spojené hranami reprezentujúcimi podmienené atomické prechody medzi stavmi. V stavoch alebo pri prechodoch medzi nimi sa potom vykonávajú určené akcie. Jednoducho sa implementujú, ale pre komplexné systémy môžu byť neprehľadné a zložité na návrh.

Pri utility-based AI má každá možná akcia priradenú tzv. funkciu užitočnosti (utility function), ktorá za behu vypočíta dôležitosť či užitočnosť danej akcie v aktuálnej situácii. Nakoniec sa vyberie akcia s najvyššou hodnotou. Tento prepočet stačí v praxi vykonať každých n snímkov hry. Touto funkciou môže byť akýkoľvek lineárny či nelineárny vzťah, dokonca môže využívať aj užitočnosť iných akcií. Napríklad užitočnosť akcie „útek“ lineárne vzrastá s klesajúcou vzdialenosťou od nepriateľa; užitočnosť granátu je najvyššia proti cieľu v strednej vzdialenosti, aby hráč neohrozil seba, ale zároveň zvládol dohodiť. Oproti konečným automatom a behaviorálnym stromom sú modulárnejšie, jednoduchšie rozšíriteľné a dajú sa znovu použiť. Táto metóda bola úspešne použitá napr. v hre Red Dead Redemption (2010) na výber zbrane a v dialógoch, či v hre F.E.A.R. (2005) na dynamické taktické rozhodovanie.

3.5.2 Behaviorálne stromy

Stali sa veľmi populárnymi v hernej AI po ich úspešnom použití v hrách Halo 2 (2004) a Bioshock (2007). Dokonca sú natívne podporované v Unreal engine a CRYENGINE. Použité boli aj v celosvetovo známej pôvodom českej hre Kingdom Come: Deliverance (2018), kde ovládajú súčasne priemerne 600 NPC (ľudia aj zvieratá).[32][2] Výhodou behaviorálnych stromov oproti konečným automatom je modularita a možnosť vytvárať komplexné chovania. [35][31]

Výpočet prebieha opakovaným prechádzaním stromu od koreňa k listom na tzv. tick signále, pričom v praxi sa to nevykonáva v každej snímke hry, ale napr. len 10krát za sekundu. Rodičovský prvok sa dotáže potomka na jeho stav. Potomok môže vrátiť jednu z 3 hodnôt: success alebo failure keď dokončil svoju úlohu, alebo running ak ešte nedokončil. Rodičovský prvok potom podľa svojho typu buď pokračuje rovnakým spôsobom s ďalším potomkom, alebo vracia svojmu rodičovi nejaký výsledok, taktiež success, failure alebo running. Potomkovia uzlu majú navzájom určené poradie/priority, čo sa graficky znázorňuje zoradením zľava (najvyššia priorita) doprava. Uzly jednej inštancie BT môžu navzájom komunikovať cez zdieľaný kontext, napr. v Unreal engine nazývaný Blackboard. [35] [31] [11] Napríklad, uzol „Najdi najbližšieho nepriateľa“ uloží svoje zistenia do zdieľaného kontextu, a uzol „Zaútoč na nájdeného nepriateľa“ ich využije. Ich návrh prebieha vizuálnou formou, vďaka čomu sú vhodné aj pre dizajnérov bez znalosti programovania.

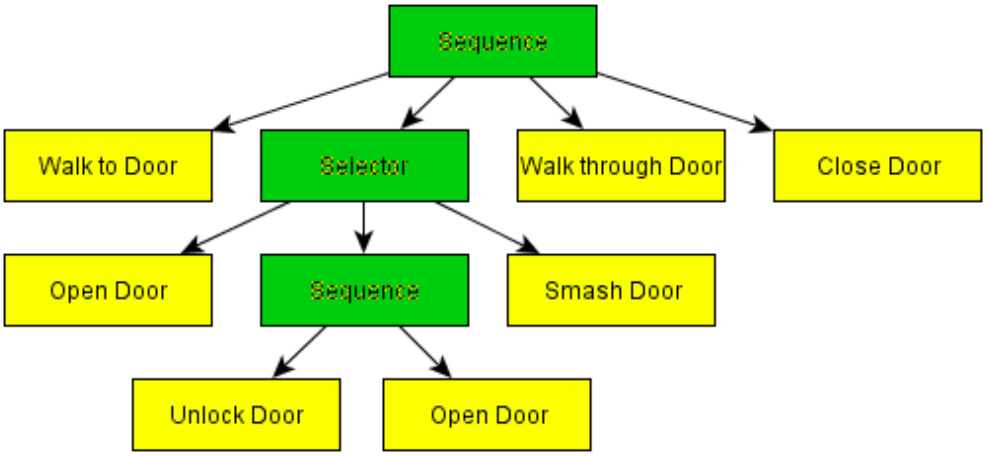
Behaviorálny strom je okrem koreňa zložený z týchto typov uzlov: [35] [31] [11]

- Úloha (task) – koncový uzol / list – predstavuje samotné chovanie. Môže sa taktiež jednať o ďalší BT. Ak sa úloha vykonáva naprieč viacerými prechádzaniami stromu (tick), tak uzol vracia hodnotu running.
- Zložený riadiaci uzol (composite, control flow) – vnútorný uzol stromu – 2 hlavné podtypy sú:
  - Postupnosť (sequence) – potomkov vykonáva podľa priority (zľava doprava). Ak potomok vráti výsledok failure, tak celý uzol vráti failure. Inak sa pokračuje vykonávaním ďalšieho potomka podľa priorít, a po úspešnom vykonaní všetkých potomkov uzol tiež vracia success.
  - Výber (selection) – v základnom variante sa potomkovia volajú podľa priorít dovtedy, kým nejaký nevráti success. Potom aj uzol okamžite vracia success, bez potreby spúšťať ďalších potomkov.
- Dekorátor (decorator) – vnútorný uzol stromu s jediným potomkom. Upravuje návratovú hodnotu svojho potomka svojmu rodičovi (napr. negácia), alebo podmienene alebo opakovane volá potomka.

Okrem tejto základnej varianty existujú aj ďalšie rozšírenia, napríklad paralelne vykonávané úlohy, prerušenia bežiacich podstromov, či stochastický výber potomkov namiesto priorít.

3.5.3 Prehľadávanie stromu (tree search)

Túto skupinu algoritmov predstavujú tie, ktoré počas behu hry prehľadávajú stavový priestor možných akcií, reprezentovaný stromom. Uzly stromu predstavujú stavy hry, pričom koreňom stromu je aktuálny stav. Hrany vychádzajúce z uzlov predstavujú možné akcie,



Obr. 3.2: **Ukážka behaviorálneho stromu.** Žlté uzly sú úlohy (task), zelené sú zložené riadiace uzly (composite). Interpretácia: Ak uzlom „Walk to Door“ úspešne pristúpi k dverám, tak sa najprv pokúsi otvoriť dvere („Open Door“). Ak sa mu to nepodarí, tak sa pokúsi odomknúť dvere („Unlock Door“). Ak sa mu to poradí, tak ich otvorí („Open Door“). Ak sa mu ich nepodarí odomknúť alebo otvoriť, tak sa ich pokúsi rozbiť („Smash Door“). Ak sa mu ich teda nejakým spôsobom podarí otvoriť, tak nimi prejde („Walk through Door“), atď. Prevzaté a prefarbené z [31]

ktoré hráč môže vykonať v danom stave, čím sa dostáva do ďalšieho stavu – uzlu. Plánovanie je možné aj nad makroakciami a symbolickými stavmi, napr. akcia „presunúť sa na bezpečné miesto“ sa v praxi bude skladať z množstva menších akcií. Tieto algoritmy teda nepotrebujú trénovanie, ale vyžadujú od hry rýchly forward model (okrem plánovania cesty vo fyzickom priestore). Stromové algoritmy je možné použiť na deterministické hry s menším faktorom vetvenia a úplnou informáciou. Výsledkom vyhľadávania je postupnosť akcií, ktoré vedú k najlepšiemu výsledku. Podľa spôsobu prechádzania stromu tak vznikajú rôzne rodiny algoritmov.

Najjednoduchšou skupinou stromových vyhľadávacích algoritmov sú neinformované metódy prehľadávania, ktoré nepracujú so žiadnym ohodnotením stavov či hrán. Základnými variantami sú prehľadávanie do hĺbky (depth-first search) a prehľadávanie do šírky (breadth-first search). Kvôli ich neefektivite sa však v hrách využívajú len zriedka.

Informované metódy prehľadávania už pracujú s ohodnotením stavov/hrán. Typickým zástupcom je algoritmus A*, v ktorom tzv. heuristická funkcia určuje odhad vzdialenosti zo stavu do cieľa, čo sa využíva na efektívnejšie prehľadávanie stromu. Najčastejšie sa využíva na hľadanie cesty v priestore. Na efektívnejšie vyhľadávanie v rozsiahlych priestoroch, a/alebo priestoroch so špecifickými vlastnosťami (napr. kde všetky hrany predstavujú rovnakú vzdialenosť) vznikli početné varianty. Okrem hľadania cesty vo fyzickom priestore sa môže A* využiť aj na plánovanie ľubovolných akcií.

V ťahových hrách s úplnou informáciou, kde hrajú dvaja (a viac) hráči proti sebe, ako šach a dáma, je možné použiť algoritmus Minimax. Ten predpokladá, že obaja hráči sa snažia vyhrať. Vyžaduje funkciu na ohodnotenie stavu. Pri simulácii hry (teda prehľadávaní priestoru) v každom kroku ohodnotí všetky stavy, do ktorých je možné dostať sa z aktuálneho stavu. Striedavo potom pokračuje do stavu, ktorý je najlepší pre prvého hráča,

a v nasledujúcom kroku najlepší pre druhého hráča. Často je prehľadávanie obmedzené na maximálnu hĺbku. Existuje viac jeho variant, napr. pre hry s viac než 2 hráčmi.

V roku 2007 bol objavený algoritmus Monte Carlo Tree Search (MCTS), ktorý narozdiel od predchádzajúcich stromových algoritmov dokáže pracovať s nedeterministickými hrami s neúplnou informáciou s veľkým faktorom vetvenia, teda napr. na hranie Pokeru alebo hry Go. Dokonca nevyžaduje ani heuristickú funkciu. Vyžaduje len znalosť pravidiel hry a ohodnotenie *koncového* stavu (napr. výhra a prehra, alebo koncové skóre). Jeho základnou myšlienkou je, že neprehľadáva všetky uzly. Keď si už nejaký uzol/stav vyberie (na základe predchádzajúceho hrania, nie heuristiky), tak z tohto stavu náhodne hrá až do konca hry. Výsledok hry sa potom spätne propaguje až do koreňa stromu, podľa čoho sa potom vyberá v nasledujúcich krokoch (využitím ľubovolnej funkcie na výber akcie, napr. UCB1, ϵ-greedy...). Algoritmus MCTS bol použitý napríklad v softvéri AlphaGo na hranie hry Go, a dokáže poraziť aj svetových profesionálov.[30]

3.5.4 Učenie s učiteľom (supervised learning)

Do tejto kategórie patria algoritmy, ktoré sa procesom učenia snažia aproximovať funkciu $f : X \to Y$, kde X je obecne vektor vstupov (atribúty/príznaky/rysy) a Y je požadovaná výstupná hodnota. Podľa typu požadovaného výstupu sa môže použiť buď na odhadovanie spojitej veličiny (regresia), alebo na zaraďovanie vstupov do kategórií (klasifikácia). Proces učenia prebieha (okrem algoritmu k-NN) na množine trénovacích dát (dataset) $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, pozostávajúcej z n dvojíc – vektoru vstupných príznakov/atribútov x_i a ich požadovaným výstupom y_i. Cieľom učenia na trénovacích dátach je teda vytvoriť dostatočne všeobecný model, ktorý by dokázal správne priraďovať výstupy pre nové nevidené vstupy. Učenie zväčša prebieha pred použitím modelu, teda počas vývoja hry, pretože na zobecnenie vyžaduje dostatok trénovacíh dát, a je to relatívne výpočetne (a teda aj časovo) náročný proces. Trénovacie dáta bývajú v prípade hier založené na hrách odohraných ľudskými hráčmi, kde vstupom býva stav prostredia a výstupom akcia vykonaná hráčom. Napríklad, vstupom môže byť trojica údajov: zostávajúci život hráča a protivníka, spolu so vzdialenosťou medzi nimi; výstupom môže byť výber z 3 možných akcií: útok, útek, nečinnosť.

Podľa [35] sa v hrách typicky využívajú neurónové siete, support vector machines a rozhodovacie stromy. Ďalší príkladom algoritmu z tejto kategórie AI metód je algoritmus k-najbližších susedov (k-nearest neighbors - k-NN). Ten je, aspoň v základnej variante, neporovnateľne jednoduchší na implementáciu, avšak oproti ostatným vymenovaným sa netrénuje dopredu, ale výsledok sa relatívne výpočetne náročne zisťuje až pri použití.[16]

Učenie s učiteľom bolo použité napríklad v závodnej hre Forza Motorsport (2005). Všetky trate boli zložené z obmedzenej množiny segmentov. Trénovacie dáta boli tvorené ľuďmi odjazdenými závodmi na rôznych tratiach. Počas hrania sa potom počítačom riadení protivníci snažili podľa aktuálneho segmentu napodobniť trasu, po ktorej daný segment hral človek.

3.5.5 Posilované učenie (reinforcement learning – RL)

Výhodou RL môže byť, že nevyžaduje trénovacie dáta. Namiesto toho potrebuje dostatočne rýchle prostredie na učenie, v ktorom podľa jeho akcií (jeho výstupy) získa odmenu/trest a nový stav prostredia/hry. Podľa odmeny/trestu sa učí, akú akciu má vybrať v danom stave hry, teda učí sa, aké akcie má vyberať, aby maximalizoval celkový súčet odmien resp. minimalizoval tresty. Jadrom RL je teda správna rovnováha medzi tzv. exploration

a exploitation – preskúmavaním stavového priestoru a využívaním naučených znalostí. Na dosiahnutie uspokojivých výsledkov je treba tisícky až milióny odohraných hier. V prípade hier s jedným hráčom ich hrá proti prostrediu, v prípade hier s protivníkmi ich môže hrať proti svojim predchádzajúcim verziám (napr. hry backgammon a Go). Príkladom svetového úspechu RL môže byť systém OpenAI Five, ktorý dokázal v strategickej počítačovej tímovej hre Dota 2 poraziť aj profesionálnych jednotlivcov aj tímy.[26]

3.5.6 Učenie bez učiteľa (unsupervised learning)

Pri tejto skupine algoritmov ide o hľadanie skrytých vzorov vo vstupných dátach, bez znalosti požadovaného výstupu. Teda narozdiel od „učenia s učiteľom“ je množina vstupných dát $D = \{x_1, \ldots, x_n\}$ tvorená len n vzorkami, reprezentovaných vektorom m vstupných príznakov. Pre hry sú podľa [35] najviac zaujímavé zhlukovacie metódy a hľadanie častých vzorov.

Úlohou zhlukovania (clustering) je hľadanie zhlukov v dátach, teda podmnožín množiny vstupných vzorkov, kedy vzorky vnútri jedného zhluku sú si podobné, a zároveň vzorky z odlišných zhlukov si podobné nie sú. Kvalita výsledkov záleží na vstupných atribútoch, výbere metódy a jej parametrov. Jednotlivé algoritmy sa odlišujú napríklad tým, čo považujú za zhluk. Konkrétne metóda k-means dokáže nájsť zhluky guľového tvaru (obecne hyperguľa v m-dimenzionálnom priestore m atribútov vstupných vzorkov), zatiaľ čo hierarchická metóda Chameleon zvláda zhluky rôznych tvarov. Ďalej, rôzne algorimy vyžadujú rôzne parametre, napr. k-means vyžaduje zadať počet zhlukov, a u hierarchických metód sa zadáva ukončujúca podmienka. Príkladom využitia zhlukovania v hre by mohlo byť automatické zaradenie hráča do preddefinovaných zhlukov podľa jeho hracieho štýlu, a podľa toho by sa prispôsobilo chovanie počítačom riadených protivníkov, aby bola hra pre daného hráča primeranou výzvou, ani nie príliš jednoduchá a nudná, ani nie príliš frustrujúco náročná.

Pri hľadaní častých vzorov (frequent pattern mining) je z pohľadu hier najviac zaujímavé hľadanie, tzv. dolovanie, 2 typov častých vzorov: časté postupnosti udalostí a časté podmnožiny položiek (frequent itemsets). V prípade dolovania častých postupností sa vyhľadáva nad vzorkami tvorenými zoradenými položkami (napr. udalosťami v čase) a hľadá sa podpostupnosť opakujúca sa naprieč vzorkami. Napríklad, ak každá vzorka, tzv. transakcia, predstavuje hráčom splnené úlohy v čase, tak zistenie, že hráči preferujú plniť niektoré úlohy v určitom poradí bez toho, aby to hra nejako vynucovala, môže byť pre vývojárov zaujímavé zistenie. V prípade dolovania častých podmnožín položiek prehľadávané dáta sú tvorené nezoradenými množinami položiek, a hľadajú sa často vyskytujúce sa podmnožiny položiek naprieč vzorkami. Napríklad, ak jednotlivé vzorky predstavujú herné objekty zakúpené jednotlivými hráčmi, tak týmto spôsobom sa dá zistiť, aké kombinácie objektov si hráči často kupujú, čo môže poslúžiť ako informácia na lepšie balancovanie obsahu hry.

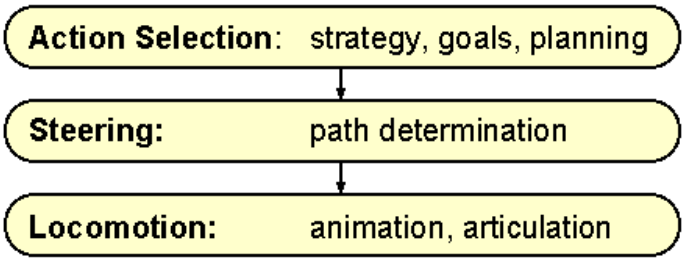
3.5.7 Ďalšie prístupy

Ďalšími populárnymi prístupmi používanými v hrách bývajú lokálne a globálne optimalizačné metódy, ktoré tiež využívajú funkciu na ohodnotenie kvality riešenia. Z lokálnych medzi jednoduchšie patrí metóda hill climbing. Lokálne prehľadávania však môžu skončiť v lokálnom extréme, čo sa snaží riešiť napríklad metóda simulated annealing (v češtine simulované žíhání). Globálne optimalizačné/prehľadávacie metódy, z ktorých sú v hernej AI najobľúbenejšie evolučné algoritmy, si udržiavajú súčasne niekoľko riešení, tzv. populáciu. Tieto riešenia s využitím operátorov modifikujú, napr. krížením či mutáciou. Riešenia ohodnotia, tzv. fitness funkciou, a určeným spôsobom vyberú jej podmnožinu na ďalšiu ite-

ráciu výpočtu, tzv. generáciu. Výpočet prebieha, kým sa nenájde riešenie s požadovaným ohodnotením alebo sa nevyčerpá výpočetný čas.

3.6 Pohyb v prostredí

Pohyb v prostredí by sa dal popísať na 3 úrovniach, tak ako je naznačené na obrázku 3.3. Do najvyššej úrovne patrí plánovanie cesty prostredím 3.6.1, ktorého výsledok je realizovaný nižšími vrstvami. Do druhej vrstvy patria tzv. steering behaviours, popísané v 3.6.2, ktoré v hrách vykonávajú samotný pohyb. Od najnižšej úrovne locomotion sa v hrách väčšinou „abstrahuje preč“, býva realizovaná len formou vizuálnych animácií. Patrí tu napríklad otáčanie kolies na vozidle a pohybovanie chodidlami pri chôdzi.



Obr. 3.3: Hierarchia pohybových chovaní. Prevzaté z [28]

3.6.1 Navigácia

Základnou požiadavkou na inteligentných agentov v hrách býva navigácia, teda pohyb po prostredí. Medzi najpoužívanejšie prístupy patria tzv. waypoint system a navigation mesh. V oboch prípadoch okrem geometrie herného sveta existuje aj špeciálna paralelná reprezentácia sveta pozostávajúca z miest prístupných pre agentov.[20] Táto reprezantácia predstavuje graf, v ktorom sa vyhľadáva požadovaná cesta, napríklad pomocou algoritmu A*, v prípade rozľahlejších prostredí niektorá z pokročilejších techník ako hierarchický A* alebo jump point search.[35]

Waypoint system

Waypoint system, historicky starší, využíva body v priestore (waypoints) prepojené zväčša priamočiarymi spojeniami, po ktorých sa môže agent pohybovať bez obáv o kolízie so statickou geometriou prostredia. Keďže sa vo výsledku jedná o orientovaný graf, na vyhľadávanie výslednej cesty medzi waypointami sa dajú použiť algoritmy ako A* alebo Dijkstrov algoritmus. Tieto waypointy, prípadne aj spojenia medzi nimi, často umiestňuje človek ručne. Naopak, pri automatickom vytváraní waypoint systému, napríklad preložením sveta jemnou mriežkou bodov, sa vytvára viac bodov než je potrebné, čo vo výsledku zvyšuje zložitosť výpočtu cesty počas behu hry. Pri príliš riedkom umiestňovaní bodov sa môžu stať niektoré časti prostredia nedostupné. Preto sa takmer vždy vyžaduje aj zásah človeka. [20][12][34]

Špeciálnym prípadom je pravidelná pravouhlá mriežka bodov, tzv. grid, na ktorej sa často vysvetľuje princíp algoritmu A*, a využitá je napríklad na plánovanie pohybu v engine Game Maker. V ňom sa prekážky, teda nedostupné časti prostredia, označia ako forbidden cells, a pohyb sa plánuje len vo free cells. V Game Makeri sa dá jemnosť mriežky, teda

veľkosť buniek, ľubovolne nastaviť, avšak hľadanie cesty v jemnejšej mriežke je všeobecne výpočetne náročnejšie. Jemnejšia mriežka môže byť zase presnejšia v okolí nepravidelných prekážok. [20][27]

Waypoint systém bol používaný napr. v starších hrách z Source engine, ako Counter Strike 1.6, a označoval sa ako Nodegraph. V Source engine človek ručne umiestňoval body (tzv. nodes) a engine dopočítal medzi nimi spojenia (tzv. links) ak boli dostatočne blízko seba a nebola medzi nimi prekážka. [12][13]

K waypointom a spojeniam môžu byť priradené ďalšie vlastnosti, ako napr. v Source engine radius waypointu, ktorý sa využíval na pokrytie väčších plôch menším počtom waypointov, ďalej pokyn na skrčenie postavy alebo preskočenie postavy medzi dvoma waypointami. [12]

Navigation mesh

Vývojovo mladší systém na navigáciu v priestore je navigation mesh, skrátene navmesh, kde namiesto bodov v priestore sú prístupné časti prostredia reprezentované navzájom prepojenými mnohouholníkmi (polygónmi). Ide o veľmi populárny prístup s natívnou podporou v moderných enginoch ako Unity, Unreal, Source či Godot, použitý napr. v Counter Strike GO, Left 4 Dead (1,2) či Team Fortress 2. [10][9][13][3]

Navmesh sa počítačovo generuje z geometrie herného sveta, primárne počas vývoja hry a nie počas hrania (existuje aj dynamický navmesh [33][3]). Tento časovo náročný proces najprv vytvorí spojenia medzi susednými (walkable) polygónmi, ďalej môže vypočítať pokročilejšie typy spojení (napr. skok medzi platformami, vylezenie po rebríku, skrčenie sa...), nakoniec to ešte môže dokončiť človek ručne.[10][13]

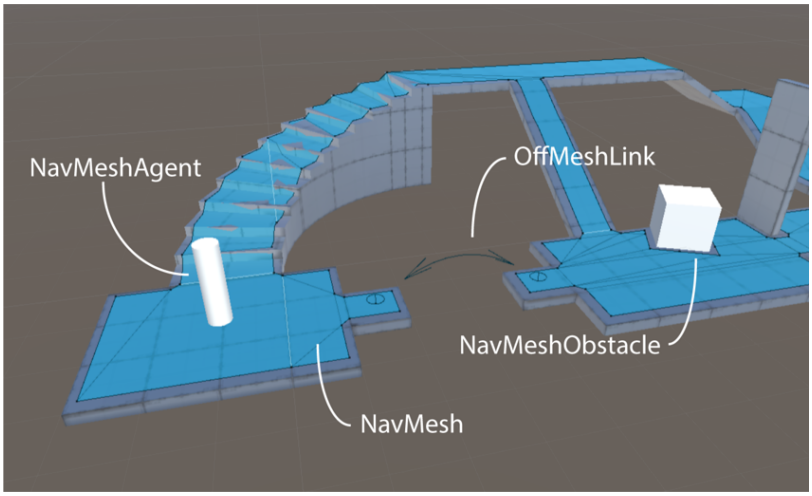
Keďže výsledkom vyhľadanej cesty (napr. pomocou A*) je postupnosť polygónov a nie priamo bodov, výsledná trasa sa dá vypočítať napríklad pomocou intuitívneho Simple Stupid Funnel Algorithm, alebo iným spôsobom určiť vhodné body z polygónov. [20]

Oproti waypoint systému má navmesh viac výhod, konkrétne má lepšie pokrytie dostupného priestoru, a vďaka tomu, že na pokrytie väčšinou treba menej polygónov než waypointov, tak aj hľadanie cesty za behu býva rýchlejšie, a môžu mať menšie pamäťové nároky.[9][20] Navmesh, okrem vyhľadávania cesty, sa dá využiť napríklad aj na hľadanie vhodnej streleckej pozície alebo hľadanie miesta na skrytie.[33]

3.6.2 Steering behaviours

Nižšie nasledujú príklady typických chovaní z pohľadu pohybu voči statickým a dynamickým objektom, ktoré môžu byť v role cieľa, prekážky alebo ostatných pohyblivých objektov. Konkrétne algoritmy výpočtu sú dostupné v zdrojovej literatúre [28][29]. Ich vstupom sú pozície a rýchlosti relevantných objektov, a výsledkom je vektor sily, ktorý sa má v aplikovať na ovládaný objekt. Dokonca ani na skupinové chovania sa nevyžaduje žiadna forma komunikácie medzi objektami. Pracujú s jednoduchým fyzikálnym modelom založeným na doprednej integrácii Eulerovou metódou, teda sa opakovane prepočítavajú (napríklad v každom snímku hry) a zanedbávajú pokročilejšie fyzikálne javy ako aerodynamiku, ktoré aj tak nie sú pre hry väčšinou podstatné.

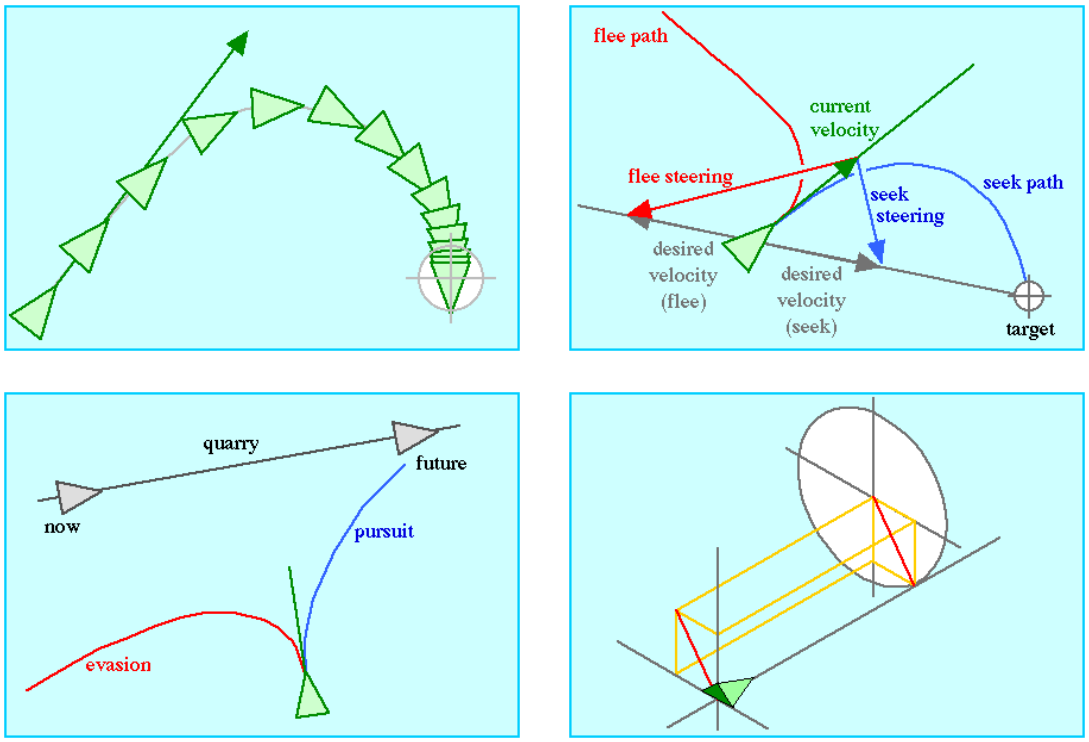
- *Seek* – smeruje za statickým cieľom (jeho pozíciou) konštantnou rýchlosťou
- *Arrival* – smeruje za statickým cieľom, so zmenšujúcou sa vzdialenosťou k cieľu spomaľuje, až zastaví v cieli



Obr. 3.4: **Ukážka navigation mesh.** Modrý povrch predstavuje navmesh vytvorený zo šedej geometrie; *OffMeshLink* predstavuje typ spojenia medzi 2 polygónmi; *NavMeshAgent* je pohybujúci sa objekt využívajúci navigáciu; *NavMeshObstacle* je statická prekážka, ktorá spôsobila rozdelenie priľahlej plochy na mnoho menších polygónov. Prevzaté z [10]

- *Flee* – smeruje opačným smerom než je statický cieľ
- *Pursuit* – aplikovanie *seek* na pohybujúci sa objekt, pričom na dobré výsledky musí nasledovať budúcu odhadovanú pozíciu objektu o T krokov dopredu, kde kľúčové je určiť správnu hodnotu T v danom čase
- *Evasion* – ako *pursuit*, ale namiesto *seek* sa použije *flee*, teda ide v opačnom smere než je predpovedaná pozícia objektu
- *Offset pursuit* – aplikovanie *seek* na bod posunutý od cieľa kolmo na smer pohybu
- *Obstacle avoidance* – pomocou detekcie potenciálnych kolízií s imaginárnych obdĺžnikom (v 2D) alebo valcom (v 3D; obecne označované ako shape cast) v smere pohybu môže včas zmeniť smer pohybu aby nedošlo ku kolízii
- Ďalšie: *wander* – náhodná plynulá prechádzka; *explore* – preskúmavanie oblasti; *path following* – sledovanie cesty s možnou odchylkou; *containment* – držanie sa v určitej oblasti; *separation* – udržiavanie určitého odstupu od ostatných pohyblivých objektov; *cohesion* – pripojenie a urdžiavanie sa v skupine pohybujúcich sa objektov; *leader following* – množina objektov nasleduje vybraný objekt (*offset arrival*), navzájom si medzi sebou udržujú odstupy (*separation*) a udržujú sa pokope (*cohesion*).

Je teda možné identifikovať určité základné spôsoby chovania, ktorých kombináciou vznikajú ďalšie pokročilejšie chovania. Ich kombinovanie je možné jednak na vyššej vrstve (viď obrázok 3.3), napríklad najprv sa herná postava náhodne pohybuje (*wander*) a pri spozorovaní hrozby začne utekať (*flee*). Druhou možnosťou ako ich kombinovať je váhovaním všetkých chovaní súčasne, napríklad *leader following*, alebo pri úteku (*evasion*) sa musí súčasne vyhýbať prekážkam (*obstacle avoidance*). [28][29]



Obr. 3.5: **Ukážky pohybových chovaní.** Na prvom obrázku je *arrival*, a na poslednom *offset pursuit*. Prevzaté z [28]

Kapitola 4

Návrh hry

Najskôr bude popísaný návrh z pohľadu užívateľa. Ďalšia časť rieši navrhované riešenia z pohľadu použitých knižníc a algoritmov. Nakoniec sa rieši spôsob testovania a vyhodnotenia.

4.1 Hra z pohľadu hráča

Výsledkom by mala byť 2D akčná strategická hra, kde by hral človek proti inteligentným nepriateľom (NPC – non-player character). NPC by predstavovali stroje riadené umelou inteligenciou. Hráč by mal k dispozícii lietajúce vozidlo vybavené zbraňami, a potom kontrolou by mohol mať menšiu letku podobných strojov. Cieľom NPC je zabrániť hráčovi v dosiahnutí úloh, alebo ho zničiť. Ničenie by prebiehalo zbraňami na diaľku.

Hra by bola z pohľadu hráča organizovaná do misií, v ktorých by sa mal napr.:

- dostať z bodu A do bodu B, pričom by sa mal vyhnúť NPC, alebo sa cez ne prebojovať
- preskúmať nepriateľské prostredie, niečo vyhľadať
- ubrániť sa útoku NPC

Misie by boli prepojené príbehom.

4.1.1 Umelá inteligencia NPC

NPC by boli agenti podľa definície – mali by senzory (diaľkomer, detektor pohybu, mikrofón...), aktuátory (smerové pohony, zbrane...) a inteligentné riadenie. Bolo by ich viac typov, s rôznou štruktúrou, vlastnosťami, odolnosťou. Vyskytovali by sa v skupinách, a mali by byť schopní navzájom komunikovať, spolupracovať, predávať si znalosti, reagovať na zmenu v prostredí, pohybovať sa v priestore. Podľa aktuálneho cieľa by vhodne útočili, obraňovali, skúmali prostredie, unikali pred hráčom.

4.1.2 Grafický štýl

Grafický štýl by bol len jednoduchý, dvojrozmerný pohľad zhora, bez textúr, niečo ako v hre Reassembly. Aby to nepôsobilo na hráča až tak nudne, použilo by sa pár grafických efektov, napríklad ako v hre Geometry Wars.

4.1.3 Prostredie

Prostredie by bolo čiastočne v jaskyniach / kaňonoch (t. j. nepravidelné, napr. ako v hre SketchFighter 4000 Alpha) a čiastočne v mestských oblastiach. V spojení s grafikou popísanou vyššie by hra vypadala, ako by sa odohrávala na mape. Celý svet by mohol mať rozmery zhruba 10 x 10 obrazoviek.

Svet by bol teda tvorený nedostupnými časťami (steny jaskýň, budovy) a priestorom umožňujúcim voľný pohyb. S tým by súvisela obmedzená viditeľnosť pre NPC, možnosti skrývať sa a potreba navigácie v (ne)známom prostredí pre hráča aj NPC. Pre obe strany by bolo prostredie, aspoň zo začiatku, neznáme.

Pri kolíziách so stenami a robotmi navzájom, by sa uplatňovali fyzikálne zákony. Kolízie by mali dopad na životnosť robotov.

4.2 Návrh implementácie

4.2.1 Použitie existujúcich komponent

Hra by bola implementovaná v C++, bez využitia existujúceho herného enginu. Využívala by bezplatnú multiplatformnú knižnicu SFML 2.5, ktorá okrem 2D grafického vykresľovania (načítavanie obrázkov, vykresľovanie tvarov a textu, podpora shaderov atď.) dokáže pracovať aj so zvukom a vstupmi (myš, klávesnica). Na simuláciu dynamiky pevných telies by sa použila voľne dostupná knižnica Box2D. Hra by bola určená pre počítače s operačným systémom Windows (ako platforma obľúbená hráčmi), prípadne pre Linuxové systémy.

4.2.2 Paralelizmus

Hlavná herná slučka by bežala v jednom vlákne. Neočakáva sa, že by bolo treba využívať nejaký spôsob paralelizmu (ani fork-join ani task-based). Ďalšie vlákno by sa použilo na asynchrónne načítavanie dátových súborov. V treťom vlákne by sa počítali krátke úlohy ohľadom umelej inteligencie, zadávané hlavne od NPC a hromadené vo fronte, napríklad hľadanie cesty alebo iné prehľadávanie stavového priestoru. Okrem toho si SFML vytvára vlastné vlákna, napríklad na prehrávané zvuky.

4.2.3 Umelá inteligencia

Na rôznych typoch NPC by sa vyskúšali rôzne prístupy k inteligencii. Konkrétne by sa jednalo o použitie štandardu v hernom priemysle - behaviorálnych stromov. Vrstvená architektúra by sa vyskúšala aspoň na jednom type nepriateľov. Ďalej by sa využili neurónové siete na pohyb v nepravidelnom svete pomocou kombinácie smerových pohonov. Na rozhodovanie sa o najvhodnejšej akcii by sa mohlo využiť posilované učenie alebo nejaká forma prehľadávania stavového priestoru.

V engine by bola komponenta sprostredkujúca dostupné vnemy. Hráč aj NPC by sa jej dotazovali, napríklad na vzdialenosť k najbližšiemu objektu z bodu A v danom smere. Úlohou hráča aj NPC by už bolo si túto informáciu vhodne interpretovať, čo by otvorilo možnosti použitiu algoritmov z oblasti robotiky, prípadne testovaniu nových prístupov. Toto by predstavovalo rozdiel oproti bežným hrám, kde NPC bývajú často vševediace, napr. vidia cez steny. Taktiež by bola snaha o čo najmenšie predspracovanie sveta v prospech NPC, aby mali rovnaké podmienky ako hráč (čo teda v bežných hrách tiež býva inak).

Na prístupy, ktoré vyžadajú trénovanie, by pripadali do úvahy tieto možnosti:

- trénovanie proti ľudskému hráčovi – asi najmenej preferovaná možnosť, pretože obecne by bolo treba veľa odohraných hier
- umelo (vývojárom) vymyslené úlohy, na ktorých by sa hľadalo najlepšie riešenie – napríklad trénovanie neurónových sietí dostať sa z miesta A do miesta B cez nepravidelný svet, kde metrikou riešenia by bol čas cesty, počet nárazov po ceste alebo najkratšia cesta
- hranie sám proti sebe – na trénovanie obrany a útoku, avšak bude treba dať pozor, aby sa algoritmus nenaučil síce efektívnu, ale neprirodzenú alebo nudnú stratégiu pre hráča

4.2.4 Assets

Počas vývoja by bolo vhodné vytvoriť grafický nástroj na tvorbu herného sveta, prípadne NPC, hráčovu letku a iné herné objekty. Vzniknuté modely a ďalšie dátové súbory herného sveta by sa ukladali v textovom formáte, pre budúce možnosti úprav aj bez špecializovaného editoru.

4.3 Testovanie a vyhodnotenie

Cieľom práce je vytvoriť hru, ktorú by bolo možné upravovať a rozširovať o nové typy inteligentných agentov. Nejednalo by sa teda len o akademické vývojové prostredie, ale šlo by aj o reálnu hru určenú bežným hráčom. Na demonštráciu funkčnosti by hra implementovala niekoľko typov agentov využívajúcich rôznych typov umelej inteligencie.

Vyhodnotenie by bolo možné 2 spôsobmi:

- ľudskými hráčmi – ich pocity ohľadom náročnosti, zábavnosti, prirodzenosti atď.
- namiesto človeka by boli napevno nasimulované scenáre, v ktorých by sa sledovali reakcie NPC, napr. nasledovanie hráča v neznámom prostredí, hráč v presile, nečakaná zmena situácie... Chovanie NPC by potom mohol zhodnotiť človek, alebo objektívne metriky, ako výsledok a trvanie hry, stav/poškodenie robotov na konci hry, prípadne percento splnených podúloh.

Kapitola 5

Implementácia

Analogicky k obrázku 2.1 by sa vytvorená hra dala popísať obrázkom 5.1. Zväčša sa nejedná o konkrétne triedy ale o subsystémy, komponenty či skupiny objektov. Analogicky ku kapitole 2 sa v tejto kapitole popíše vzniknutá hra od nižších vrstiev (platforma a využité knižnice), cez znovupoužiteľné komponenty tvoriace engine, plynule prechádzajúc do vyšších vrstiev komponent špecifických pre vzniknutú hru.

5.1 Fungovanie enginu

Táto sekcia popisuje ako zvyšok tejto kapitoly do seba zapadá. Vcelku tak popisuje obsah funkcie main ako vstupného bodu programu.

Po spustení hry sa ako prvá vec načíta globálny konfiguračný súbor game.data v koreňovom adresári hry. Podľa neho sa inicializujú tie podsystémy 5.4, ktoré to potrebujú. Konkrétne: vývojárske nástroje (logovanie 5.5.1, profilovanie 5.5.5, debug vykresľovanie 5.5.2, grafy 5.5.3), lokalizácia reťazcov 5.7, čas 5.9, správca assets 5.8, grafické nastavenia a vytvorenie okna 5.12, a parametre herného cyklu 5.13. Podľa konfigurácie sa taktiež môžu prednačítať všetky potrebné herné dátové súbory, ktorých zoznam je v súbore preload.txt. Nakoniec sa inicializuje prvá herná scéna 5.10, konkrétne scéne s hlavným menu.

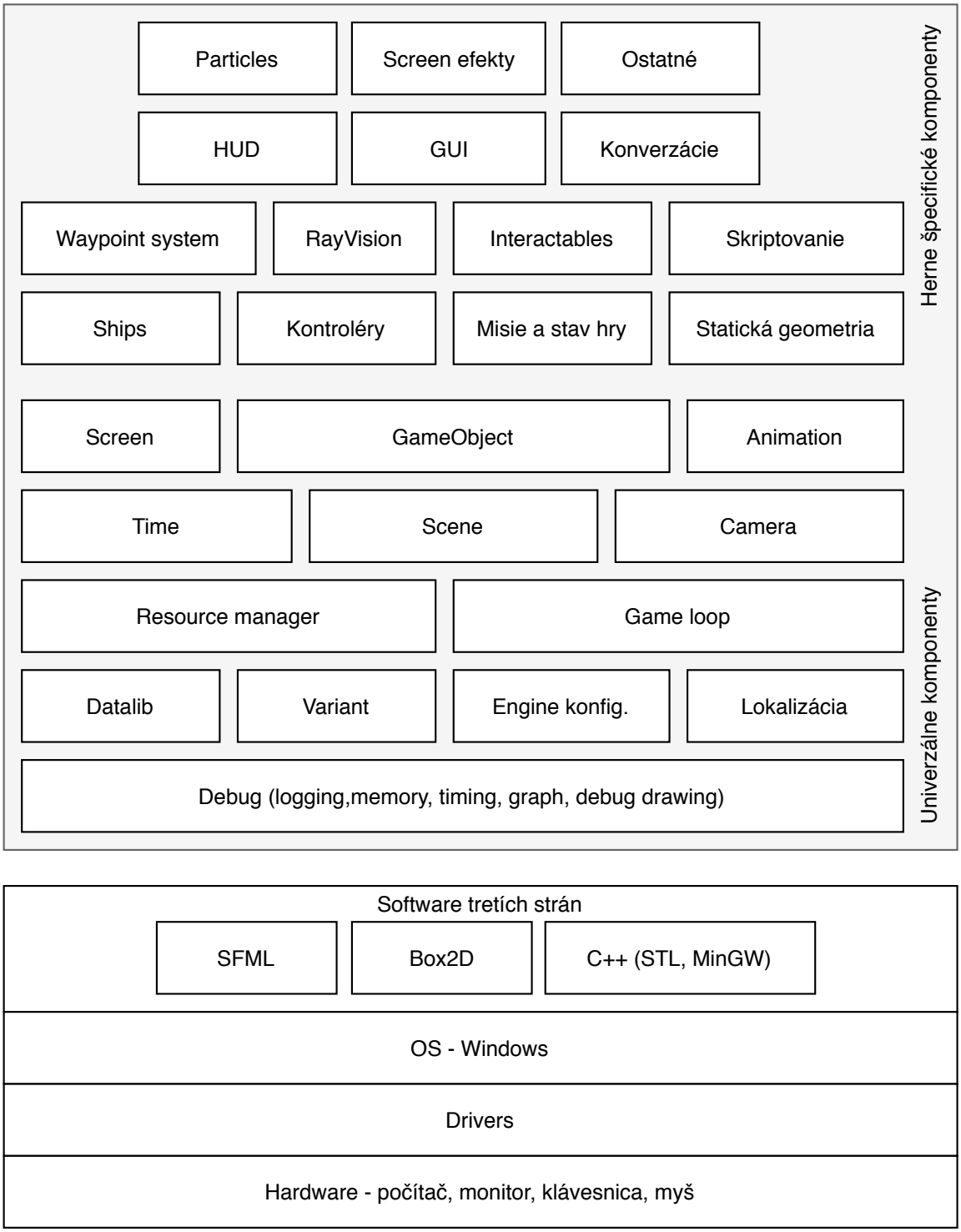
Následne, konečne začne hlavný herný cyklus 5.13. Ten v každej iterácii najprv obslúži udalosti od operačného systému resp. knižnice SFML. Herné objekty však namiesto udalostí používajú v druhej časti cyklu na vstupy polling – od statických tried sf::Mouse a sf::Keyboard. Druhou časťou herného cyklu je aktualizácia všetkých herných objektov 5.15 o jednu snímku hry, ktorá sa vykoná zavolaním metódy update nad aktuálnou scénou. Poslednou časťou každej iterácie cyklu je vykreslenie scény a vývojárskych informácií, ak sú zapnuté. Cyklus beží dovtedy, kým sa hra neukončí.

Po ukončení hry sa zapíšu nazbierané vývojárske informácie do súborov podľa globálnej konfigurácie, napr. memory leaks, informácie o časovaní atď.

Nakoniec sa zničí scéna so všetkými hernými objektami, a podsystémy enginu sa zničia v opačnom poradí než sa inicializovali.

5.2 Platforma

Hra je určená pre osobný počítač (PC), nie pre konzoly ani tablety či telefóny. Z periférií predpokladá prítomnosť klasického rastrového farebného monitoru, klávesnicu (s latinskými znakmi) a myš (s aspoň dvoma tlačítkami a funkciou kolieska).



Obr. 5.1: Schéma podsystémov a komponent implementovanej hry/enginu.

Hra bola primárne vyvíjaná na systéme a pre systém Windows, pretože tak pokrýva majoritu hráčov. S minimom úprav by však nemal byť problém so sprístupnením aj na iných operačných systémoch. To preto, že využívané knižnice boli zámerne vybrané ako prenositeľné. Kód hry využíva WinAPI jedine na profilovanie počas vývoja (funkcie QueryPerformanceCounter) a na farebné výpisy do konzoly, čo by sa dalo relatívne jednoducho buď preniesť na iné platformy, alebo úplne vypnúť bez dopadu na hrateľnosť.

Hlavným programovacím jazykom je C++11, pretože kvôli nízkoúrovňovému prístupu núti programátora rozumieť počítaču, umožňuje písaného vysoko výkonného kódu, je v ňom napísaná väčšina herných enginov, a tiež zdrojová literatúra ohľadom enginov využíva tento jazyk. [24] Aj napriek možným nevýhodám popísaným v kapitole 5.3 hra využíva vstavanú knižnicu STL, pretože na vyvíjanú hru bohate postačovala.

Vývoj prebiehal v prostredí MinGW, z ktorého sa využíval prekladač GCC a program make. Na ladenie niektorých chýb, hlavne chybných prístupov do pamäte, slúžil program gdb.

5.3 Využité knižnice

Základom na prácu s grafikou bola knižnica **SFML** vo verzii 2.5. Jedná sa o multiplatformnú bezplatnú C++ knižnicu určenú na zjednodušenie vývoja (dvojdimenzionálnych) hier a multimediálnych aplikácií. Okrem grafického API poskytuje aj funkcie na prácu s oknom (a vstupmi od užívateľa), audiom a sieťou. Umožňuje jednoduché načítavanie rastrových obrázkov, prácu s pravidelnými n-uholníkmi aj kolekciami vertexov, vykresľovanie textov, tzv. sprites, a shadery. Predpokladá, že základom aplikácie bude tzv. message pump obsluhujúca prichádzajúce udalosti od systému, čo zapadá do typickej hernej slučky podľa kap 2.4. Udalosti prichádzajú pri zmenách na vstupných perifériách (klávesnica, myš, joystick, ...) a zmenách veľkosti okna. Okrem asynchrónnych udalostí je možné používať aj tzv. polling, teda dotazovanie sa na aktuálny stav vstupných zariadení. Okrem množstva ďalších funkcionalít treba spomenúť triedu na meranie uplynulého času a jednotnú triedu na Unicode reťazce (sf::String), ktorá umožňuje jednoduchú prácu s diakritikou. Grafické API pracuje v pixeloch, uhly sú v stupňoch, a bodom (0,0) sa označuje ľavý horný roh obrazovky. Všetky funkcionality z knižnice SFML budú ďalej prefixované menným priestorom sf, napríklad sf::ConvexShape.[7]

Druhou použitou knižnicou tretej strany je **Box2D**, čo je knižnica na simuláciu fyziky pevných telies v 2D priestore určená pre hry, napísaná v prenositeľnom C++. Podporuje rýchlo pohybujúce sa telesá (continuous collision detection), senzory, ray casting, a ukladanie užívateľských dát k objektom. Kvôli rýchlosti a fragmentácii pamäte sa objekty vytvárajú prostredníctvom factory metód (interne využíva variantu Object pool a zásobníkového alokátoru, viď 2.3.4). Každé teleso sa môže skladať z viacerých tvarov, pričom tvar musí byť konvexný, a v prípade n-uholníkov maximálne s 8 vrcholmi. Na každý tvar je naviazané tzv. fixture, ktoré udržuje jeho fyzikálne vlastnosti (hustotu, trenie, či sa jedná o senzor, ...). Ako bolo popísané v kapitole 2.3.6, hra využívajúca fyziku sa musí v každej snímke hry dotazovať na aktuálne vlastnosti objektov a podľa toho ich vykreslí – simulácia teda prebieha oddelene od zvyšku hry a Box2D nič nevykresľuje. Knižnica pracuje v jednotkách KMS, teda kilogramy-metre-sekundy, uhly sú v radiánoch, a svet je prevrátený podľa osy x. Knižnica bola vyladená na dynamické objekty o veľkosti 0,1 až 10 metrov a statické objekty do zhruba 50 metrov. Všetky funkcionality z knižnice Box2D budú ďalej prefixované reťazcom b2, napr. b2World.[1]

5.4 Podsystémy enginu – štart, koniec a konfigurácia

Triedy, u ktorých sa predpokladá existencia jedinej inštancie, niekde nazývané aj ako manager triedy, namiesto osobitných tried využívajú statické členy a metódy tried, ktoré by inak spravovali, čím sa tiež dosiahne obmedzenia na jedinú inštanciu. Vzhľadom na to, že statické premenné v jazyku C++ majú pri štarte programu nedefinované poradie inicializácie, tak všetky podsystémy/triedy, ktoré to potrebujú, implementujú statické metódy init a destroy. Tým sa zabezpečí explicitné poradie inicializácie a tiež je možné tieto triedy parametrizovať. Tento jednoduchý a efektívny spôsob je podporený aj literatúrou [24]. Parametrizácia, hlavne metódy init, prebieha formou jediného parametru typu asociatívny slovník (viac o variantnom type v 5.6). Toto sa vo zvyšku kapitoly označuje spojením „globálna konfigurácia“, pretože konfiguračné informácie pre všetky podsystémy sa načítavajú z jediného súboru game.data v koreňovom adresári hry.

5.5 Vývojárske nástroje

Literatúra zaoberajúca sa hernými enginmi [24] zdôrazňuje vhodné vývojárske nástroje ako nevyhnutnú súčasť každého herného enginu, takže podľa potreby bolo implementovaných viac funkcionalít na zjednodušenie ladenia. Všetky sú umiestnené v adresári debug. Nasleduje ich popis podľa frekvencie používania.

5.5.1 Trieda Logger

Vypisovanie do konzoly sa ukázalo ako najviac používaná funkcionalita, aj na bežné kontrolné výpisy, aj na riešenie problémov, a to omnoho častejšie než debugger.

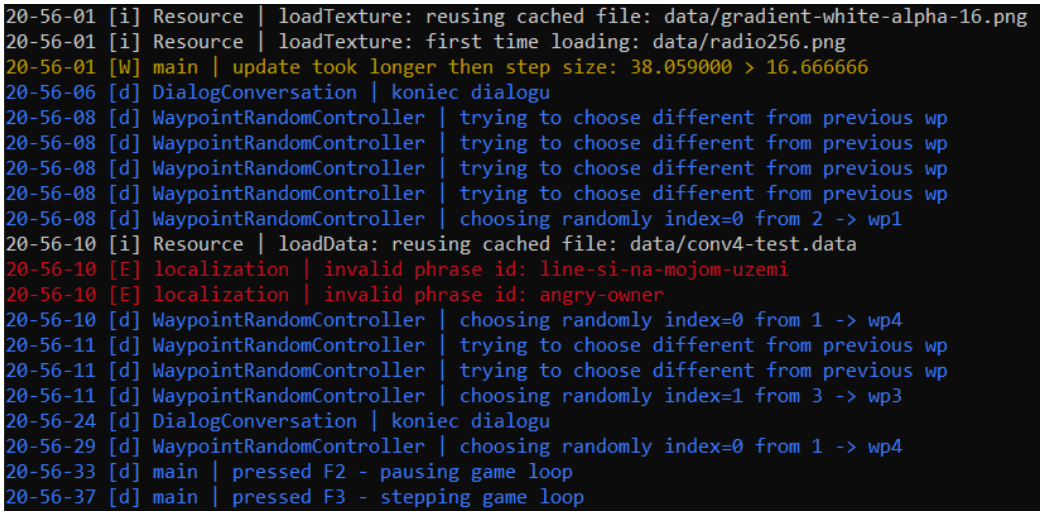
Trieda Logger pracuje na základe myšlienky, že každý zdroj výpisov (najčastejšie trieda) má názov, a môže vypisovať na viac úrovniach, konkrétne v poradí debug, info, warning a error.

Výpisy smerujú do konzoly a/alebo do súboru podľa určenej úrovne, čo sa dá kontrolovať cez globalnu konfiguráciu. Každý zdroj môže vypisovať do osobitného súboru a do jedného spoločného globálneho pre všetky zdroje. Metódy na výpis využívajú formátovací reťazec s variabilným počtom argumentov, napr. `void warning(const char *fmt, ...)`. Na systéme Windows sa do konzoly vypisuje podľa úrovne odlišnými farbami (s pomocou funkcie SetConsoleTextAttribute z WinAPI). Súbory sú pomenované dátumom a časom inicializovania triedy, teda pri spustení hry. Logger inštancia vzniká pri prvom dotázaní podľa názvu (metóda `static Logger *get(const std::string &name)`) a existuje až kým sa neukončí hra, čo je pri zavolaní statickej metódy destroy. Trieda si v statickej premennej udržuje mapovanie medzi názvom a inštanciou.

Ako možné vylepšenia sa ukázalo zapisovanie do súborov asynchrónne, aby nespomaľovalo volajúci kód. Taktiež by metóda get mohla namiesto ukazateľa vracať referenciu, aby sa tým naznačilo používateľovi, že sa nemá snažiť objekt uvoľňovať.

5.5.2 Trieda DebugDraw

Táto trieda pôvodne vznikla na umožnenie debug vykresľovania pre knižnicu Box2D. To sa dosiahlo implementáciou rozhrania b2Draw, ktorej inštancia sa jednak predáva knižnici, a tiež je dostupná ako globálna premenná na využitie v hre. Rozhranie b2Draw slúži na vykresľovanie základných primitív ako kruh a čiara, a tým umožňuje knižnici Box2D jed-



Obr. 5.2: Ukážka reálnych výpisov do konzoly počas vývoja s využitím triedy Logger.

noducho vykresľovať všetky telesá, vektory síl či ohraničujúce obdĺžniky (tzv. axis-aligned bounding box – AABB).

Okrem požadovanej funkcionality pre rozhranie b2Draw boli pridané aj ďalšie možnosti, napríklad vykresľovať vývojárske texty a mriežku (každý meter, so súradnými osami sveta). Navyše trieda umožňuje aj vykresľovať debug informácie kdekoľvek z nevykresľovacieho kódu (napr. z konštruktorov), so životnosťou viac než jednu snímku, a nad všetkou ostatnou grafikou. Podľa návrhového vzoru Command [25] si udržuje zoznam vykresľovacích príkazov, ktoré predstavujú reifikované volania funkcií. Tieto objekty majú nastaviteľnú životnosť v sekundách, čo sa hodí napr. pri udalostiach čo trvajú len 1 snímku. Do tohto globálne dostupného zoznamu sa môžu pridávať príkazy z ľubovolného miesta v kóde, a potom sa po vykreslení všetkej „bežnej“ grafiky nakoniec vykreslí aj zoznam. S týmto designom sa taktiež dá vykresľovanie vývojárskych informácií na jedinom mieste zapnúť a vypnúť. V implementovanej hre/engine sa dajú vývojárskej informácie zobraziť podržaním klávesy Q.

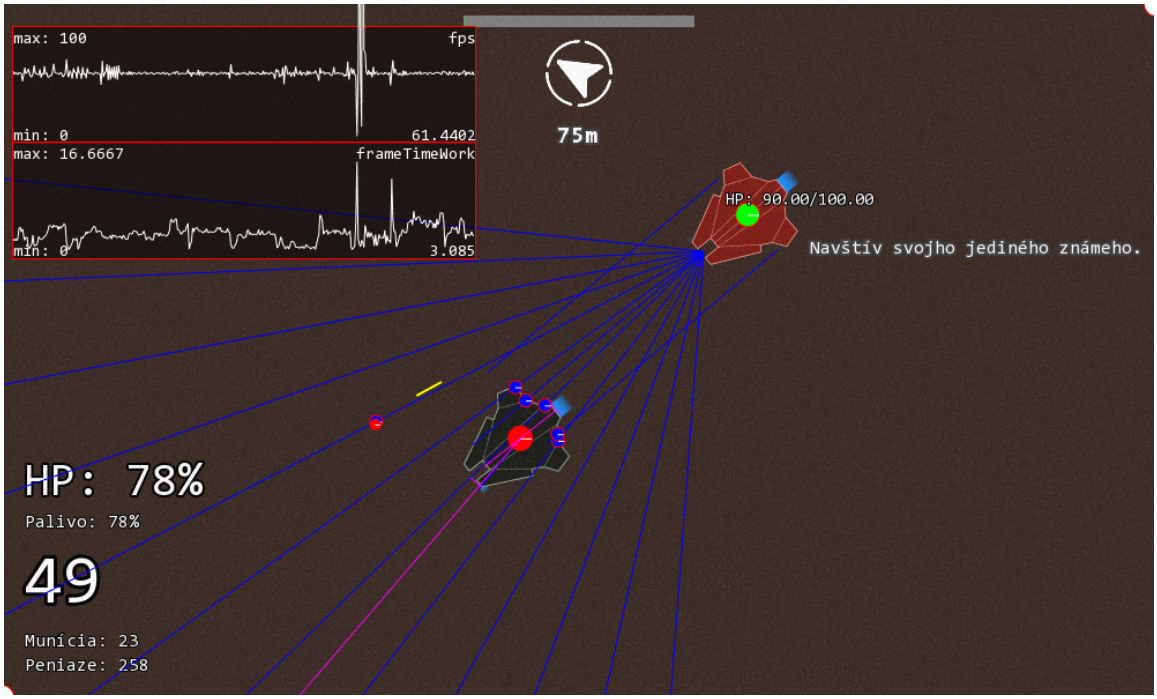
5.5.3 Trieda Graph

Vývojárska trieda Graph si pamätá buď všetky alebo posledných N hodnôt, ktoré boli do grafu pridané. Graf sa v reálnom čase vykresľuje na obrazovku a nazbierané hodnoty je možné exportovať vo formáte CSV. Udržuje si minimálnu a maximálnu hodnotu, buď od začiatku alebo z posledných N hodnôt, prípadne nastavenú manuálne, čo sa tiež využíva na užívateľsky prívetivejšie vykresľovanie.

5.5.4 Monitorovanie alokácií

Trieda Mem slúži na monitorovanie alokácií. Do deklarácie triedy, ktorej alokácie sa majú sledovať, sa pridá makro MONITOR_ALLOCATIONS, ktoré prepíše operátor new a delete. Nové operátory fungujú štandardným spôsobom s využitím C funkcií malloc a free, a navyše registrujú resp. odregistrujú objekty v globálnom zozname alokovaných objektov.

Tento zoznam sa udržuje v statickej premennej triedy Mem. Kvôli rýchlosti je implementovaný ako std::unordered_map, a mapuje ukazatele na štruktúry, obsahujúce okrem veľkosti objektu aj jeho triedu a voliteľne dodatočnú informáciu. Dodatočnou textovou in-



Obr. 5.3: Snímka z hry zachycujúca prenasledovanie hráča počítačom riadeným protivníkom. Na obrázku je naľavo dolu vidno HUD, v strede hore je GPS a PursuitManager, napravo je jediná aktívna úloha. Ďalej je zapnuté vykresľovanie vývojárskych informácií: 2 grafy (na FPS a čas snímky, s manuálne nastavenými extrémami kvôli prehľadnosti), modrými čiarami je vizualizovaná trieda RayVision [kap-rayvision], kde modré krúžky predstavujú to čo vidí NC, a pri červenej lodi je vypísané HP prenasledovateľa. Tenká žltá čiarka pod grafmi je vystrelený projektil od NPC.

formáciou môže byť napríklad pôvod daného objektu, čo môže pomôcť pri zisťovaní, prečo sa neuvoľnil.

Okrem tohto globálneho zoznamu si trieda Mem tiež vedie štatistiky o počte de/alokácií a veľkosti celkovo alokovaných a neuvoľnených dát. Štatistiky a zoznam všetkých alokovaných objektov je možné vypísať buď do konzoly alebo do súboru vo formáte CSV – podľa globálnej konfigurácie.

Pri preklade s makrom NDEBUG sa monitorovacie makro namiesto operátorov rozvinie na prázdny príkaz a teda nemá žiadny dopad na rýchlosť.

Všetky triedy vytvorené autorom tejto práce majú takýmto spôsobom monitorované alokácie.

5.5.5 Trvanie kódu

Na profilovanie blokov kódu z pohľadu času ich trvania je možné daný kód zabaliť do bloku, na začiatku ktorého sa pridá makro PROFILE s názvom merania, napríklad názov meranej funkcie. Makro sa rozvinie na objekt triedy HighResTiming, ktorého konštruktor si zapamätá počiatočný čas a deštruktor ukončí meranie. Nameraný čas sa uloží do globálneho zoznamu meraní podľa názvu merania. Zoznam meraní je implementovaný ako statický std::unordered_map. Predpokladá sa opakované meranie s rovnakým názvom, takže okrem

kumulatívneho času sa ukladá aj počet meraní. Meranie na systéme Windows prebieha pomocou 64bitovej hotnoty získanej z funkcie QueryPerformanceCounter z WinAPI.

Namerané hodnoty je možné vypísať do konzoly alebo uložiť do súboru vo formáte CSV – podľa globálnej konfigurácie.

Pri preklade s makrom NDEBUG sa makro PROFILE namiesto operátorov rozvinie na prázdny príkaz.

5.5.6 Trieda Trail

Doplnkovou funkcionalitou je trieda Trail, ktorá zbiera posledných N pozícií v 2D priestore a vykresľuje ich ako krivku. Dá sa využiť na sledovanie trajektórie pohybujúcich sa objektov.

5.6 Formát dátových súborov a variantný typ

Konfiguračné a dátové súbory pre engine aj hru sú zapísané v špeciálnom textovom formáte v súboroch s príponou `.data`, ktorý pripomína formát JSON. Po načítaní takéhoto súboru do pamäte sú dáta uložené vo variantnom type [17]. Práca s dátovými súbormi a variantným typom je implementovaná v súboroch `datalib.h/.cc`.

Hlavnou vlastnosťou nového formátu je jeho jednoduchosť, je bez „nepotrebných“ symbolov, vhodný na manuálne vytváranie človekom. Literatúra [24] však hovorí, že z ekonomického pohľadu by bolo lepšie využiť štandardný existujúci formát, ktorý by ušetril čas programovania a mal by odladené knižnice a viac možností (napr. hexadecimálny zápis binárnych dát).

Formát vychádza z formátu JSON, z ktorého sa odstránili čiarky a dvojbodky. Ďalej, ak textový literál (kľúč v slovníku alebo hodnota) pozostáva len z vybranej podmnožiny ASCII znakov, tak ho netreba ohraničovať úvodzovkami. Podporuje riadkové a blokové komentáre, ale s inými oddeľovačmi. Pozná 4 dátové typy: čísla, textové reťazce, heterogénny zoznam a asociatívny slovník. Po vzore jazyku herného štúdia Naughty Dog (spomínané v [24] a [23]) a jazyku Lisp je možné rovnakou syntaxou zapisovať dáta aj programy, čo sa v obmedzenej forme využíva v triede ActionExecutioner, viac v kap. 5.22.

Spracovanie dátových súborov je implementované ako spojenie konečného automatu na lexikálnu analýzu a rekurzívneho zostupu na syntaktickú analýzu. Variantný typ je implementovaný triedou AData a triedami od neho odvodenými využitím dedičnosti a polymorfizmu. Okrem 4 typov použítých aj v jeho textovej podobe podporuje aj typ na páry (hlavne na interné účely) a typ na ukazatele.

Okrem práce s dátovými súbormi sa variantný typ používa aj na predávanie dát medzi podsystémami a objektami, napríklad na inicializáciu podsystémov a na zakódovanie informácii o vzniknutej udalosti medzi hernými objektami (metóda `GameObject::onEvent(DataDict *eventInfo)`).

5.7 Lokalizácia textov

Po vzore [24] je lokalizácia textových reťazcov implementovaná ako jednoduchá globálna funkcia getLocalizedString, ktorá očakáva ako jediný parameter textový identifikátor reťazca (napr. „press-space-to-interact“). Tá, podľa aktuálne zvoleného jazyka z globálnej konfigurácie, vráti reťazec typu sf::String, ktorý predstavuje ideálny prenositeľný typ na uchovanie textov zachovávajúci diakritiku. Pri nenájdení požadovaného reťazca sa vráti chybový reťazec, napr. *[ERROR:press-space-to-interact:cz]*, cez triedu Logger (viď 5.5.1) sa

vypíše chyba, a hra pokračuje ďalej, čo umožňuje vyvíjať hru aj bez kompletných assets. Lokalizačné dáta sú uložené v súbore `loc.data` v UTF-8 kódovaní v koreňovom adresári, vo textovom formáte popísanom v kapitole 5.6. Jedná sa o jeden veľký asociatívny slovník indexovaný podľa identifikátora reťazca. Jednotlivé položky tohto slovníka sú tiež slovníky, obsahujúce reťazec pre každý podporovaný jazyk kľúčované dvojpísmenným kódom jazyka.

5.8 Trieda Resource

Jedná sa o statickú triedu, ktorá poskytuje metódy pre načítavanie každého potrebného typu dát. Aktuálne podporuje textové súbory s príponou `.data` (viď 5.6), rastrové obrázky s príponou `.png`, alebo načítanie ľubovolného typu súboru (používa sa na shadery).

Podľa návrhového vzoru Flyweight [25] si udržuje len jednu reprezentáciu z každého požadovaného súboru, čím šetrí miesto v pamäti a urýchľuje opakované načítavanie rovnakého súboru.[24] Vzhľadom na malú výslednú hru načítané dáta ostávajú v pamäti až do ukončenia hry, teda do zavolania statickej metódy destroy.

Ako sa spomína v kap. 2.3.5, načítané súbory najprv spracuje, a až túto inicializovanú formu predáva ako výsledok. Konkrétne, súbory s príponou `.data` prevedie na variantnú štruktúru (popísanú v 5.6), pričom sa vykonáva lexikálna a syntaktická kontrola. Načítané PNG obrázky vytvoria sf::Texture, čo je dekódovaný obrázok umiestnený vo video pamäti. Inšpirované literatúrou [24], pri chybe načítavania obrázku sa vráti obrázok symbolizujúci chybu (červeno-zelená šachovnica) a hra pokračuje ďalej, teda vývoj môže pokračovať aj s nekompletnými assets. Ostatné (i binárne) súbory sa vracajú ako typ std::string. Taktiež, podľa súboru `preload.txt` a globálnej konfigurácie je možné prednačítať všetky potrebné súbory a netreba tak čakať, kým si ich hra explicitne vyžiada, čo je užitočné aj na kontrolu integrity všetkých súborov. Tento súbor sa dá vygenerovať spustením hry s prepínačom `--generatePreloader`.

Počas vývoja sa ukázalo, že v budúcnosti by mohla pribudnúť možnosť označiť variantný typ len na čítanie (tzv. ho zmraziť), aby si ho rôzni užívatelia nemohli navzájom, hoci omylom, upravovať. Ďalej, ako bolo tiež naznačené v literatúre [24], užitočná by bola kontrola načítaných dát cez schémy, napríklad na správny dátový typ, povinné atribúty a doplnenie východzích hodnôt.

5.9 Čas

Statická trieda Time udržiava herný čas (modelový čas simulácie), ktorý môže bežať zrýchlene alebo spomalene, čo sa často využíva v hrách na rýchlejšie plynutie dní. Okrem herného času udržiava aj rýchlosť simulácie, čo umožňuje tzv. slow motion efekt pri zachovaní rovnakej prekresľovacej frekvencie. Taktiež umožňuje hru pozastaviť a krokovať. Pri takomto pozastavení herný cyklus klasicky beží a vykresľuje scénu, ale herný svet sa neaktualizuje, čo sa dobre využíva na ladenie rýchlych procesov, ako udalosti súvisiace s kolíziami fyzikálnych objektov.

V implementovanej hre sa spomalenie modelového času využíva pri interaktívnych konverzáciách. Pozastavenie, krokovanie a pokračovanie v hre je možné klávesami F2, F3 resp. F4. Taktiež sa scéna neaktualizuje (len vykresľuje), keď okno hry nie je na popredí – nemá focus.

5.10 Scény

Hru je možné organizovať do tzv. scén, pričom hra je vždy práve v jednej scéne. Obdobný koncept sa v iných enginoch nazýva rôzne, napr. pre Game Maker je to room, pre Unreal Engine je to Level, pre Unity a Godot tiež Scene.[27][9][10][3] Príkladom jednej scény môže byť napríklad počiatočné menu a druhou scénou by bol jediný herný svet alebo jednotlivé úrovne hry (levely). Scéna v tejto práci implementuje metódy init a destroy tak ako iné podsystémy, a tiež update a draw ako herné objekty. Aktuálna scéna je jediný herne špecifický objekt, o ktorom vie herný cyklus, predstavuje teda akési rozhranie medzi enginom a hrou. Volania metód update a draw z herného cyklu deleguje na herné objekty.

5.11 Kamera

Úlohou triedy Camera je prepočet medzi súradným systémom fyzikálnej simulácie a obrazovky monitoru, ktorých rozdiely sú popísané v kap. 5.3. Jednak na to poskytuje funkcie na prepočet bodov/pozície a tiež transformačnú maticu, ktorá sa priamo predáva ako parameter do vykresľovacích rutín objektov herného sveta. Vstupmi pre výpočet transformačnej matice M sú rozmery obrazovky v pixeloch p_x a p_y, rozmery viditeľnej časti simulovaného sveta v metroch m_x a m_y, a pozícia kamery v metroch (v strede viditeľnej časti) c_x, c_y.

$$M = \begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix}, \text{ kde}$$

$$t_x = (-c_x + \frac{m_x}{2}) \cdot \frac{p_x}{m_x}$$
$$t_y = (-c_y - \frac{m_y}{2}) \cdot \left(- \frac{p_y}{m_y} \right)$$
$$s_x = \frac{p_x}{m_x}$$
$$s_y = -\frac{p_y}{m_y}$$

5.12 Trieda Screen

Táto statická trieda podľa globálneho konfiguračného súboru vytvára okno, do ktorého sa hra vykresľuje. Medzi jeho nastavenia patria napríklad rozlíšenie okna, roztiahnutie na celú obrazovku alebo úroveň vyhladzovania hrán (anti-aliasing).

V spojení s hlavným herným cyklom podporuje 4 tzv. render módy:

- FPS – frames per second - knižnica SFML sa snaží prekresľovať okno pri rýchlosti zadaného počtu snímkov za sekundu, napr. 60. Ak sa hra zvláda aktualizovať a vykresliť v požadovanom čase, tak sa hra uspí (na volaní sf::Window::display) na dosiahnutie daného FPS.
- bez obmedzenia – hra beží najrýchlejšie ako môže. Neželaným efektom môže byť trhanie obrazu, pretože rôzne časti obrazovky sa prekreslia v rôznych snímkach. Vzhľadom na nenáročnosť hry sa však tak zrýchli, že sa môže stať nehrateľnou. Riešením je použitie herného cyklu typu 3.

- VSync – prekresľovanie hry je synchronizované s obnovovacou frekvenciou monitoru, čo by malo zabrániť trhaniu obrazu.
- bez vykresľovania – herný cyklus nevolá vykresľovacie rutiny a beží najrýchlejšie ako môže. Tento režim môže byť vhodný na strojové učenie, kedy algoritmus umelej inteligencie nepotrebuje obraz (s výnimkou učenia podľa pixelov), čím sa môže ešte viac zrýchliť beh hry a trénovania.

5.13 Hlavný herný cyklus (game loop)

Cez globálnu konfiguráciu sa dá meniť typ herného cyklu medzi 2 možnosťami: `LAG_FIXED` (označuje typ 3) a `UNIFORM`.

Vo vzťahu k 4 render módom (RM) popísaným pri triede Screen 5.12 sa dajú dosiahnuť tieto 3 typy herného cyklu podľa 2.4, nastaviteľné bez potreby prekladu, len cez konfiguračný súbor `game.data`:

- typ 1 – bežať najrýchlejšie ako dokáže – pri použití RM „bez obmedzenia“ (hodnota NOLIMIT) alebo „bez vykresľovania“ (hodnota NORENDER)
- typ 2 – „framerate governing“ / uspanie – pri použití typu cyklu `UNIFORM`, a zároveň RM FPS alebo VSync
- typ 3 – oddelenie aktualizovania od vykresľovania – pri použití typu cyklu `LAG_FIXED`, a zároveň RM FPS alebo VSync

Praktický rozdiel medzi týmito 3 typmi je pozorovateľný len pri náročnejších scénach. V tom prípade sa typ 2 spomalí (aj prekresľovanie aj simulácia), a typ 3 bude prekresľovať menej často, avšak svet sa bude snažiť aktualizovať správnou rýchlosťou. Všeobecne, ak aktualizácia sveta (update) v danej snímke zaberie viac než vyhradený čas na snímku, tak sa vypíše varovanie. Proti „spiral of death“ je to ošetrené cez maximálny počet iterácií, po ktorého prekročení sa simulácia proste spomalí.

Hlavný herný cyklus sa nachádza v súbore `game-main.cc`, vo funkcii `main`. Ukážka reálneho kódu tu nie je možná, pretože kvôli 4 render režimom, 2 typom herného cyklu, krokovaniu, profilovaniu a vývojárskemu vykresľovaniu, je kód príliš dlhý.

5.14 Animácia hodnôt klúčovými snímkami

Trieda Animation implementuje interpoláciu hodnôt v čase cez kľúčové snímky (keyframes), inšpirovanú enginom Three.js.[8] Objekty tejto triedy si udržujú vektor dvojíc čas-hodnota a aktuálny lokálny čas, ktorý je aktualizovaný spolu s ostatnými hernými objektami. Podľa príslušnosti do časového intervalu v rámci vektoru sa hodnota interpoluje použitím vybranej funkcie. Interpolačné funkcie sú reprezentované triedami, ktoré implementujú rozhranie AnimationEasing. Aktuálne sú okrem lineárnej interpolácie podporované a využité ďalšie 3 funkcie, podľa stránky https://easings.net. Podľa spôsobu ukončenia animácie je možné: ukončiť animáciu (REPEAT_NONE), opakovať od začiatku (REPEAT_LOOP), pokračovať od konca na začiatok jedenkrát (REPEAT_YOYO_ONCE) alebo donekonečna (REPEAT_YOYO).

5.15 Herné objekty – trieda GameObject

Takmer všetky herne špecifické objekty (na schéme 5.1 umiestnené *nad* položkou GameObject) dedia od triedy GameObject. To umožňuje ich jednotnú správu a komunikáciu.

5.15.1 Spoločné atribúty

Každý z herných objektov má priradený unikátny konštantný identifikátor id – monotónne rastúce celé číslo, s periódou 2^{31}, čo by malo byť na hry dostačujúce.[1] Táto hodnota sa dá využiť namiesto ukazateľov na vyhľadanie a testovanie existencie objektu, na čo sa bežné ukazatele využiť nedajú.

Herné objekty majú ďalej className ako textový reťazec názvu triedy (len posledná trieda v hierarchi dedičnosti). Ten sa dá využiť na dynamické zisťovanie typu inštancie, napríklad pri kolíziách.

Ďalej majú objekty vlastnosť name typu std::string. Zo strany enginu je táto hodnota bez akýchkoľvek požiadavok na obsah alebo unikátnosť, takže je na autorovi hry či a ako ich využije. Spolu s id a className sa dá využiť na dynamické vyhľadávanie objektov počas hry. V implementovanej hre sa tento atribút využíva na unikátne názvy objektov.

Celočíselná hodnota depth typu int slúži na určenie poradia, v akom sa objekty vykresľujú. Vykresľuje sa od najmenších hodnôt po najväčšie. Na zjednodušenie určenia hodnoty poskytuje trieda GameObject konštanty pre rôzne úrovne (11), napr. na pozadie, statickú geometriu, lode či dialógy. Zmena tejto hodnoty spôsobí opätovné zoradenie všetkých objektov pred ďalším vykresľovaním.

Nakoniec všetky herné objekty majú boolean vlastnosti autoUpdate a autoDraw. Prvá z nich určuje, či sa má objekt aktualizovať spolu s ostatnými objektami pri volaní GameObject::updateAll a zničiť pri volaní GameObject::destroyAll. Pri nastavení na hodnotu false sa predpokladá volanie update a destroy iným objektom, ktorý ich logicky vlastní, čím sa dá explicitne vynútiť poradie aktualizovania a ničenia objektov. Druhá z vlastností sa využíva analogicky pri volaní GameObject::drawAll. Na podobné využitie existuje napríklad v engine Game Maker funkcia set_automatic_draw.[27]

5.15.2 Spoločné metódy

Konštruktor bázovej triedy herných objektov novovzniknutý objekt pridá do globálneho zoznamu objektov a nastaví príznak, že treba objekty pred vykreslením znovu zoradiť podľa depth. Deštruktor objekt z tohto zoznamu odstráni. Ďalej, všetky herné objekty implementujú metódy update a draw. Jedná sa návrhový vzor Update Method [25], pri ktorom sa nad objektami rôznych typov rekurzívne volajú dané metódy.

Metóda `void update(float dt)` slúži na aktualizovanie objektu o jednu snímku hry. Jej parametrom je delta time, ktorý môže nadobúdať ľubovolné hodnoty podľa aktuálneho typu herného cyklu a režimu vykresľovania (viď kap. 5.13). Aktualizácia všetkých objektov o delta time modelového času by nemala trvať viac než je delta time reálneho času, inak sa hra spomalí.

Metóda `void draw(sf::RenderTarget &window)` slúži na vykreslenie grafickej reprezentácie daného objektu na cieľ daný parametrom typu sf::RenderTarget, čo môže byť buď sf::RenderWindow alebo sf::RenderTexture. Typ sf::RenderTexture predstavuje rastrový povrch v pamäti, na ktorý sa dá vykresľovať, a ktorý sa automaticky nikde nevykresľuje.

[1] Častice s krátkou životnosťou nie sú implementované ako GameObject, ale cez ParticlePool, viac v 5.28.

Dá sa využiť napríklad na optimalizácie (jednorázové predkreslenie komplexnejšej grafiky) alebo na rôzne efekty (shadery, tiene...). Bohužiaľ, podľa testov a oficiálneho fóra, knižnica SFML ešte nepodporuje vykresľovanie na sf::RenderTexture s vyhladzovaním hrán (anti-aliasingom), a preto sa v celej hre posiela do metódy draw ako parameter okno aplikácie, teda typu sf::RenderWindow. [7].

Metóda `bool onEvent(DataDict *event)` je z pohľadu vývoja hry relatívne nová takže niektoré objekty komunikujú iným spôsobom (napríklad dialóg s jeho tlačítkami využívajú polling). Jedná sa o dynamicky typovanú väzbu. Parametrom je asociatívny slovník obsahujúci relevantné informácie o udalosti. Kód implicitne predpokladá existenciu aspoň jedinej hodnoty s kľúčom „type“, ktorá určuje typ udalosti. Žiadna kontrola správnosti obsahu slovníka nie je implementovaná. Návratová hodnota metódy hovorí, či sa má v propagácii udalosti pokračovať alebo už bola udalosť obslúžená. Tento spôsob komunikácie sa využíva napríklad pri fyzikálnych kolíziách herných objektov, kedy sa obom zúčastneným objektom posiela informácia, kto je druhým kolidujúcim objektom (kľúč „other“), spolu so silou nárazu (kľúč „damage“). Ďalšie využitie našli udalosti pri komunikácií lodí, kde správy sú prefixované reťazcom „message:“, napr. „message:destroyed“ ak je daná NPC loď zničená hráčom.

Metóda destroy objekt hneď nezničí, ale pridá ho do zoznamu objektov pripravených na zničenie (viac v 5.15.3).

5.15.3 Správa herných objektov

Ako bolo naznačené v kap. 5.4, na správu objektov triedy GameObject sa používajú statické členy tiež triedy GameObject.

Všetky existujúce inštancie herných objektov sú registrované ako ukazale v std::list s názvom instances_. Tento zoznam sa udržuje zoradený podľa atribútu depth kvôli vykresľovaniu. Programátor hry by však nemal predpokladať žiadne poradie objektov pri aktualizácií (metóda update).

Ďalej trieda obsahuje statické metódy na vyhľadanie prvej inštancie podľa id, triedy alebo atribútu name.

Statické metódy updateAll, destroyAll a drawAll zavolajú nad inštanciami dané 3 metódy, rešpektujúc atribúty autoUpdate resp. autoDraw. V updateAll sa najprv zničia všetky objekty čakajúce na zničenie a nad zvyšnými objektami sa zavolá update. V drawAll sa pred volaním draw objekty zoradia podľa depth, teda ak je to potrebné (boolean statická vlastnosť reorderByDepth sa nastaví pri zmene depth alebo pri vytvorení objektu).

V statickom toDestroy_ typu std::list sú pripravené objekty na odstránenie z instances_, čo sa vykoná v statickej metóde GameObject::destroyPending(). Keby sa objekty mazali okamžite, tak by to mohlo spôsobovať problémy. Reálnym príkladom problému je, ak sa pri kolízii zničí guľka (trieda Bullet), tak v jej deštruktore chce zničiť aj fyzikálne teleso guľky (typu b2Body), čo sa nedá, pretože svet je uprostred aktualizácie (je uzamknutý presne kvôli takýmto prípadom). Výhodou typu std::list je, že taktiež umožňuje pridávať položky počas iterovania nad rovnakým zoznamom. Tejto vlastnosti sa využíva pri odstraňovaní objektov, ktoré už sú v tomto zozname, pričom tieto objekty môžu vo svojich deštruktoroch zavolať destroy nad objektami, ktoré sami vlastnia, a teda sa ďalšie objekty pridajú na koniec tohto zoznamu.

5.16 Lode

Ako bolo popísané v kapitole 4 hráč v hre ovláda akési vozidlá na pomedzí vznášadla a vesmírnej lode, ďalej nazývané ako „lode“ – ships. Všetky lode v hre dedia od spoločnej triedy Ship, ktorá sama dedí od GameObject.

Každá loď vlastní jedno fyzikálne teleso reprezentujúce loď (typu b2Body), vizuálnu reprezentáciu typu MultishapeWithBorder a kontrolér lode. Pohony a zbrane sú uložené ako dva std::vector-y obsahujúce objekty tried Thruster a Weapon. Zbrane a pohony sú ovládané instančnými premennými weaponTrigger resp. wantMoveDIR a wantRotateDIR. DIR v prvom prípade zastupuje: forward, backward, left, right. DIR v druhom prípade zastupuje left a right. Nakoniec má loď ešte stavové premenné na zostávajúci život (označované ako HP – health points), palivo a muníciu.

Fyzikálne teleso lode sa skladá z viacerých podtvarov, konkrétne n-úholníkov, kvôli požiadavkam na tvary popísané v kap 5.3. V prvotných fázach vývoja boli modely lodí vytvárané ručne na papieri, rozmery boli merané v milimetroch a cez parameter scale vhodne vynásobené.

Pri konštrukcii lode sa rozlišuje medzi statickými a dynamickými informáciami, ktoré sú predávané ako parametre konštruktoru. Statickými sú informácie obsiahnuté v súbore s modelom lode, dynamickými informáciami sa myslia pozícia lode vo svete a hodnoty jej stavových členov, ako život či palivo. Model každej lode je uložený v osobitnom súbore, ktorý obsahuje polygóny fyzikálneho a vizuálneho telesa, parametre pohonov a zbraní.

Trieda Ship, podľa navrhového vzoru Subclass Sandbox [25], poskytuje metódy na inicializáciu lode podľa súboru s modelom, konkrétne initBody, initThrusters a initWeapons. Od triedy Ship dedia všetky lode:

- Player – predstavuje jedinú hráčom ovládanú loď. Ako jediná loď kontroluje dostupnosť svojho paliva. Hra končí, keď je hráč buď bez paliva, alebo keď jeho život (HP) klesne na nulu.

- DummyShip – jedná sa o väčšinu lodí viditeľnú počas hry, ktoré slúžia na oživenie herného sveta. Tieto lode využívajú kontrolér WaypointRandomController na náhodné pohybovanie sa mestom pomocou waypoint systému.

- AnnoyingShip – loď použitá v jednej misii, ktorá jazdí tak pomaly, aby vyprovokovala hráča k „agresívnejšej“ interakcii – nabúraniu alebo strieľaniu. Ak hráč do tejto lode nabúra, tak AnnoyingShip pošle AngryOwnerShip správu o nabúraní, pri streľbe správu o streľbe, a pri zničení (kedy $HP \leq 0$ správu o zničení. Komunikácia medzi loďami prebieha ako zavolanie metódy onEvent, a animovane sa naznačí komunikácia. Po zničení loď zmení farbu na šedú a jej kontrolér sa zmení na NullController, ktorý nijak loďou nepohybuje.

- AngryOwnerShip – loď použitá v jednej misii. Prijíma správy od AnnoyingShip, pričom „najhoršiu“ hodnotu si zapamätá a pri interakcii s hráčom podľa toho reaguje. Interakcia s hráčom nastáva, keď sa hráč k tejto lodi priblíži dostatočne blízko – automaticky sa spustí dialóg-konverzácia. Výsledkom konverzácie je buď zaplatenie výkupného hráčom, alebo začatie prenasledovania, viac v kapitole [kap-pribeh]. Po zničení sa loď chová ako AnnoyingShip.

5.17 Pohony a zbrane

Pohony aj zbrane, ako triedy Thruster a Weapon, dedia od triedy GameObject. Ich spoločnými vlastnosťami je, že loď ich môže všeobecne vlastniť niekoľko súčasne. Taktiež, pri ich konštrukcii dostávajú od lode: referenciu na loď (aby vedeli, kde aplikovať impulz resp. vytvoriť projektil); referenciu na premennú, ktorá ich ovláda (množstvo paliva resp. dostupných nábojov); ich umiestnenie v lokálnom súradnom systéme lode; a sila (sila pohonu resp. poškodenie spôsobované projektilmi).

Pohony majú navyše parameter určujúci smer relatívne voči lodi, podľa kľúča mapping zo súboru s modelom lode. Cez premennú, ktorá ich ovláda, je možné pohon využiť na 0 až 100 % resp. na 0–1 násobok maximálneho výkonu. Tým sa umožňuje plynulé a precízne ovládanie, špeciálne pre NPC ovládané počítačom. Spotreba paliva sa počíta podľa percent využitia výkonu, veľkost pohonu (parameter width v modely lode) a maximálneho výkonu daného pohonu.

Zbraň má navyše parameter cooldown určujúci minimálny čas medzi jednotlivými výstrelmi. Spotreba munície je vždy 1. Pri výstrele sa vytvára herný objekt typu Bullet. Tento projektil má vlastné fyzikálne teleso s nastaveným príznakom bullet, čo preň aktivuje tzv. Continuous Collision Detection, aby sa predišlo tunelovaniu.

5.18 Kontroléry

Všetky kontroléry pre lode implementujú rozhranie AShipController. Toto rozhranie vyžaduje jedinú metódu update, s parametrom lode ktorú ovláda a delta time. V hre sa využívajú tieto kontroléry:

- NullController – podľa návrhového vzoru Null Object [25], ktorý sa loďou nijak nesnaží nepohybovať.
- PlayerMouseWSADInputController – slúži na ovládanie lode hráčom spôsobom popísaným v kapitole 6.1. V zdrojových kódoch existuje aj starší spôsob, v kap. 6.1 označený ako „divný“, ale nevyužíva sa.
- WaypointPathController – pohyb po zadanom zozname waypointov. Využíva sa len pri jednej misii, kde nehrozí kolízia s inými loďami. Preto môže ovládaná loď prechádzať priamo cez stred waypointu. Dalo by sa povedať, že sa jedná o chovanie *arrival* z kap. 3.6.2, pretože so zmenšujúcou sa vzdialenosťou k waypointu loď vypne pohony a využije svoju zotrvačnosť.
- WaypointRandomController – loď si vždy vyberá ďalší waypoint ako náhodný zo zoznamu možných spojení aktuálneho waypointu, viac v kap. 5.19.
- FollowAndShootTargetController – ako názov napovedá, jedná sa o prenasledovanie dynamického cieľu, v tomto prípade hráča, so súčasným strieľaním. Bola tu snaha aplikovať nejakú z AI metód popísaných v kap. 3, hlavne behaviorálne stromy, prípadne (hierarchické) stavové automaty. Avšak najintuitívnejším, síce nie práve elegantným, bolo ad-hoc riešenie, ale s dobrými výsledkami.

  Tento kontrolér využíva triedy RayVision ako zrak pre loď. Ak je hráč v zornom poli prenasledovateľa, tak si loď zapamätá jeho pozíciu a smeruje za ním. Z posledných dvoch videných pozícií si prenasledovateľ priebežne počíta hráčovu rýchlosť. Ak sa

hráč skryje za prekážkou, tak loď smeruje na posledné miesto, kde ho videla. Ak sa prenasledovateľ priblížil už dostatočne blízko k danému miestu ale stále hráča nevidí, tak sa podľa vypočítanej hráčovej rýchlosti snaží predvídať, kde by sa mohol nachádzať, a tam pôjde. Počas celého prenasledovania platí, že ak je hráč videný na prostrednom paprsku RayVision, tak prenasledovateľ vystrelí zo zbrane.

Počas prenasledovania komunikuje kontrolér s triedou PursuitManager. Vždy, keď prenasledovateľ uvidí hráča, tak to signalizuje PursuitManageru, ktorý následne vynuluje tzv. evadeCounter. Bez tohto signálu evadeCounter narastá rýchlosťou reálneho času a keď dosiahne 10 sekúnd, teda na 10 sekúnd stratil prenasledovateľ z dohľadu hráča, tak sa prenasledovanie končí. Stav evadeCounter sa graficky znázorňuje hráčovi ako HUD prvok.

5.19 Waypoint systém

V súbore s herným svetom sú po svete umiestnené zástupné objekty reprezentujúce waypointy, v literatúre [24] tiež označované pojmom spawner. Tieto objekty majú pozíciu (x, y), názov waypointu, a zoznam jeho spojení (názvy ďalších waypointov oddelených medzerou). Podľa týchto spawnerov sa vytvoria herné objekty triedy Waypoint.

Podľa toho, či medzi dvoma waypointami existuje spojenie len jedným alebo oboma smermi (je možné zistiť metódou `bool Waypoint::isTwoWayWith(Waypoint* other)`, podľa toho sa lode inak pohybujú. Pri jednosmernom spojení môže loď smerovať priamo do stredu waypointu. Pri obojsmernom spojení treba počítať s loďou pohybujúcou sa v opačnom smere. Tu sa využíva konvencia, že sa jazdí po pravej strane. Loď teda na obojsmerných „cestách“ mieri do bodu o kúsok posunutom od waypoitnu, jedná sa o pohybové chovanie *offset seek/pursuit* z kap. 3.6.2.

5.20 RayVision

Táto trieda funguje pre NPC lode buď ako diaľkomer alebo ako kamera. Do fyzikálneho sveta vyšle paprsky, tzv. ray cast, a vráti, ktoré objekty, ako ďaleko a kde presne zaznamenala. Simuluje tak obmedzené zorné pole s malou frekveciou vzorkovania. Parametrizovaná je zorným uhlom, počtom paprskov a maximálnou viditeľnosťou. Ray casting má na starosti trieda implementujúca rozhranie b2RayCastCallback z knižnice Box2D, ktorá si naprieč všetkými callbackmi zapamätá len najbližší videný objekt.

5.21 Konverzácie

Konverzácie slúžia v hre na rozprávanie a posúvanie príbehu dopredu, prípadne na robenie rozhodnutí. Najčastejšie bývajú spúšťané tým, že sa hráč postaví na označené miesto (trieda StaticPlatform) a po vyzvaní stlačí medzerník. Formou konverzácie je tiež riešený úvodný tutorial, ktorý hráčovi ukáže dôležité prvky a nastaví mu prvú úlohu.

Konverzácia je implementovaná ako herný objekt triedy DialogConversation. Jej konštruktor je parametrizovaný súborom s konverzáciou (načítaný ako DataDict). Konverzácie boli inšpirované dialógovým systémom hry The Last of Us [23].

Celá konverzácia predstavuje konečný stavový automat, ktorý sa skladá zo segmentov, pričom na obrazovke je vždy vykreslený práve jeden segment. Pred začatím konverzácie

sa vykonajú príkazy uložené pod kľúčom „preActions“ (viac o príkazoch v kap. 5.22). Počiatočným stavom/segmentom je implicitne prvý segment súboru. Segment bez vetvenia dialógu obsahuje svoje id (unikátny reťazec vrámci konverzácie), rečníka, text segmentu (jednu alebo viac viet) a id nasledujúceho segmentu. Takýto segment vykresľuje jediné tlačítko na prechod na ďalší segment. Ak segment predstavuje vetvenie, tak obsahuje pole options, kde každá položka má text položky a id ďalšieho segmentu; rečník je implicitne hráč. Takýto segment obsahuje pre každú možnosť z options jedno tlačítko s daným textom. Každý segment môže obsahovať ešte kľúč actions, čo je pole príkazov (viac v 5.22), ktoré sa vykonajú na prechode do ďalšieho segmentu. Dialóg prebieha dovtedy, kým id nasledujúceho stavu nie je null.

Konverzácia sa vykresľuje nad všetkými hernými objektami vrátane HUD, a hra pod ňou je rozmazaná shaderom (viac v kap. 5.27), zosvetlená, a fyzikálna simulácia je spomalená na 20 % bežnej rýchlosti. Podľa rečníka sa líši farba textu a jeho meno je uvedené na začiatku správy. Texty sa zobrazujú animovane po písmenách rýchlosťou čítania, aby hráč nemohol konverzáciu rýchlo preklikať, nabáda ho to na čítanie.

5.22 Skriptovanie

Pri vývoji hry sa od začiatku nepočítalo s využitím externých skriptov, napr. v typickom hernom jazyku LUA, ale len s C++ zdrojovými kódmi. Počas vývoja sa to však ukázalo ako vhodné, tak sa na to v obmedzenej miere využil existujúci formát (kap. 5.6). Príkazy pre hru sú reprezentované ako pole akcií, kde akcia je pole obsahujúce názov akcie s jej parametrami. Pripomína tak volanie funkcie. O vykonávanie týchto príkazov/akcií sa stará trieda ActionExecutioner s metódou executeActionList. S využitím týchto akcií sa pracuje s aktuálnymi misiami (akcie questAdd, questComplete, questRemoveAll), GPS, peniazmi (akcie moneyAdd, moneySubtract) atď. Tieto akcie sa využívajú v konverzáciách. Spustenie kódu pre akciu je implementované ako jednoduchá sekvencia if-else príkazov, čo má lineárnu zložitosť. Pri vyššom počte podporovaných príkazov by sa mohlo vyplatiť určitá forma hashovania (s konštantnou zložitosťou), napríklad využitím std::unordered_map.

5.23 Stav hry

Pri každom vykonaní akcie z kap. 5.22 sa súčasne celá akcia ukladá do statickej triedy GameState. Vzniká tak zoznam akcií, ktoré predstavujú aktuálny stav hry. Ak by sa tento zoznam uložil na disk, hra by mohla podporovať ukladanie a načítanie rozohranej hry. K tomuto zoznamu však treba ešte pridať stav lodí, ako je ich pozícia, poškodenie, či stav kontroléru.

5.24 Herný svet

Na vytváranie herného sveta a modelov lodí sa pôvodne vyvíjal vlastný editor. Po týždni vývoja sa to ukázalo ako nečakane časovo náročné. Následne bol objavený existujúci editor vytvorený pre knižnicu Box2D s názvom R.U.B.E. (skratka od Really Useful Box2D Editor), dostupný na stránke https://www.iforce2d.net/rube/. V bezplatnej verzii umožňuje prehliadať a vytvárať scény, v platenej verzii za 35 dolárov umožňuje aj scény ukladať a exportovať vo formáte JSON.

Exportovaný súbor obsahuje množstvo nepotrebných informácií pre hru. Do herného sveta je taktiež treba pridávať a pracovať s hernými objektami, tzv. spawners. Práca s objektami môže byť v editore miestami nepríjemná, takže objekty sú vytvárané len symbolicky. Preto bol vytvorený program na konverziu z JSONU do súboru s príponou .data vo formáte z kap. 5.6. Program sa nazýva RJ2GD, skratka od R.U.B.E. JSON to game DATA, a je implementovaný v jedinom súbore `rj2gd.cc`. Jedná sa o konzolový program s dvoma parametrami – vstupný a výstupný súbor.

Herný svet obsahuje statickú geometriu (steny a budovy), waypointy, tzv. spawn-points, a geometriu interaktívnych platforiem. Steny a budovy majú ako atrubút príznak, či sa jedná o budovu alebo stenu, podľa čoho sa mení ich farba a poradie ich vykresľovania. Waypointy majú svoj názov a atribút obsahujúci zoznam názvov ďalších waypointov, s ktorými tvoria spojenie. Spawn-pointy predstavujú neviditeľné objekty s unikátnym názvom, podľa ktorých sa pozicujú herné objekty počas hry, napríklad miesta, kde sa majú vytvoriť NPC lode. Nakoniec, platformy predstavujú akési interaktívne plošiny, na ktoré keď sa hráč postaví svojou loďou, tak umožňujú spustiť konverzáciu. Platformy sú previazané s ich interaktívnymi vlastnosťami cez ich unikátne názvy.

Načítanie sveta počas hry má na starosti trieda LevelBuilder, ktorú sa volá pri vytvorení scéna s hrou (trieda SceneGameplay).

5.25 2D geometria

Vizuálna reprezentácia stien, budov, lodí a interaktívnych platforiem je implementovaná cez triedy ShapeWithSharedBorder a MultishapeWithBorder. Objekty oboch tried využívajú na inicializáciu informácie zhodné s geometriou pre fyziku, a navyše majú farbu výplne, hrán a hrúbku hrán.

ShapeWithSharedBorder sa využíva na steny a budovy. Predpokladá, že statická geometria celého sveta bude tvorená množstvom menších objektov a nie z jediného veľkého objektu. S tak obrovským objektom by sa nie len zle pracovalo, ale ani by nespĺňal požiadavky fyzikálnej knižnice. Na dosiahnutie celistvého dojmu jednotlivé „podsteny“ na seba tesne naväzujú a sú vykresľované bez ich hrán. Vykresľujú sa až hrany s prostredím a nie medzi jednotlivými objektami. Výpočet týchto hrán je možný až po vytvorení všetkých podobjektov. Potom sa jednoduchým algoritmom (s kvadratickou zložitosťou) zistí, ktoré hrany nie sú zdieľané so žiadnymi inými objektami, a tie sa považujú za výslednú hranu.

Trieda MultishapeWithBorder sa požíva na lode a platformy, ktoré predstavujú jediný objekt ale z viacerých podtvarov (viac n-úholníkov). Tieto objekty sú vykresľované tiež bez vnútorných hrán, ale výpočet prebieha len vrámci daného objektu.

5.26 HUD a GUI

HUD (Head-up dispay), teda vrstva informácií vykresľovaná nad všetkými objektami sveta, je vždy tvorená údajmi o hráčovi (zostávajúci život, palivo, rýchlosť, munícia a peniaze), GPS (ukazuje smer a vzdialenosť k cieľu) a zoznamom aktívnych úloh. V niektorých situáciách je tiež zobrazená výzva na interakciu (keď sa hráč dotýka interaktívnej platformy), notifikácie o zmene (napr. pri začatí novej úlohy alebo obdržaní peňazí za úlohu), popisky pri kurzore myši alebo ukazateľ priebehu prenasledovania. Na začiatku a konci kapitoly je celá obrazovka (dokonca aj HUD) zakrytá nadpisom kapitoly.

Medzi interaktívne prvky zobrazené nad svetom patria konverzácie s tlačítkami (viac popísané v 5.21) a správa o konci hry.

Všetky tieto prvky sú implementované ako osobitné triedy dediace od GameObject, aby boli centralizovane aktualizované, vykresľované, a boli dynamicky dohľadateľné.

5.27 Efekty obrazovky

V hre sú využité 3 druhy celoobrazovkového efektu: zosvetlenie, zrnenie a rozmazanie. Zosvetlenie je viditeľné počas konverzácií aby viac vynikol dialóg, a je implementované ako jednoduchá polopriesvitná vrstva svetlej farby. Nenápadné zrnenie počas celej hry vynahradzuje textúry na veľkých jednofarebných plochách. Jedná sa o textúru s náhodnými pixelmi v odtieňoch šedej, mierne väčšiu než je obrazovka, a táto textúra sa vykresľuje s náhodným posunom, čím sa dosahuje lepší efekt, než keby bola textúra statická. Efekt rozmazania je prítomný počas konverzácií, aby odpútal pozornosť od sveta za prebiehajúcim dialógom. Tento efekt najprv podvzorkuje vykreslenú hru, na túto textúru sa aplikuje shader s rozmazaním s kernelom o veľkosti 5x5, a výsledok sa opať roztiahne na celú obrazovku s vyhladzovaním, čím sa dosiahne ekvivalentný výsledok ako pri použití omnoho väčšieho kernelu v shadery.

5.28 Časticové efekty

Zatiaľ jediným časticovým efektom v hre sú iskry pri kolíziách lodí. Iskry majú krátku životnosť (maximálne 0,2 sekundy) a pri kolízii ich vznikajú desiatky až stovky. Vzhľadom na dôvody popísané v kap. 2.3.4 nie sú implementované ako plnohodnotné herné objekty (trieda GameObject) ale prostredníctvom návrhového vzoru Object Pool [25].

V hre existuje rozhranie AParticlePool. Tým že dedí od GameObject, tak sa automaticky aktualizuje, vykresľuje a je vyhľadateľné za behu hry. Ako rozhranie pre prípadné budúce časticové systémy stanovuje, že častice sa vytvárajú len interne a externý kód tak nedostane ukazatele/referencie na častice, čím sa zabezpečí spoľahlivá správa pamäte. Žiadne rozhranie pre častice neexistuje, aby boli možné ľubovolné optimalizácie, napríklad aktualizovanie častíc vektorovými inštrukciami.

Pre iskry existuje v hre ParticleSparkPool. Na správu iskier formou poolu používa std::vector pevnej veľkosti, odhadnutej počas vývoja. Ak by však bol vektor plný, tak sa pri požiadavke na vytvorenie častice požiadavka jednoducho ignoruje a vypíše sa varovanie. Na rýchlejšie vytváranie častíc si pool udržuje tzv. free list, viď [25]. Jedná sa o zoznam nepoužitých častíc, pričom tento zoznam sa pomocou C prvku union udržuje v samotných nepoužitých časticiach. Vykresľovanie častíc bolo zefektívnené. Namiesto rekurzívneho vykresľovania každej častice osobitne (tak ako v prípade GameObject) sa najprv všetky častice pridajú ako sf::Vertex-y do spoločného sf::VertexArray, ktorý sa len jedenkrát vykreslí. Pri malých počtoch častíc to nepredstavuje veľký rozdiel, ale pri veľkých počtoch sa vykresľovanie zrýchlilo aj päťnásobne.[2]

[2]Pri 630 časticiach nastalo zrýchlenie z 1,4 ms na 1 ms, pri 64 000 časticiach z 51 ms na 10,5 ms, čo už predstavuje výrazné zlepšenie FPS hry.

Kapitola 6

Testovanie

Testovanie a vyhodnotenie umelej inteligencie prebiehalo formou užívateľského testovania s podporou objektívnych metrík. Objektívnymi metrikami na posúdenie inteligencie NPC boli počet výstrelov a počet úspešných zásahov, aj zo strany hráča aj NPC, spolu so zostávajúcim životom (HP) všetkých lodí na konci hry. Namerané hodnoty sa vypisujú po ukončení hry do konzoly alebo súboru.

Za prvý typ inteligencie by sa dalo považovať náhodné pohybovanie mierumilovných NPC cez waypoint systém. Tento systém bol navrhovaný tak, aby nedochádzalo ku kolíziám lodí ani s ďalšími loďami ani so statickou geometriou sveta, čo sa aj podarilo. Vo veľmi zriedkavých situáciách, ak šli lode *do* a *z* rovnakého waypointu, tak kvôli implementovanému chovaniu *offset seek* neprešli okolo seba paralelne, ale pod určitým uhlom. Vzhľadom na to, že proti kolíziám implementujú ešte aj chovanie *separation*, tak sa zastavili kúsok naproti sebe. V niektorých situáciách sa postupnými pohybmi do strán (chovanie *collision avoidance*) nakoniec obišli bez kolízie. Ale veľmi zriedka zostali nehybne naproti sebe, čo mohlo zablokovať dané spojenia aj pre ďalšie lode a tvorili sa tak kolóny.

Druhým typoch umelej inteligencie bolo prenasledovanie hráča podľa toho, kde ho NPC videlo naposledy (trieda RayVision), pričom po hráčovi strieľalo zo zbrane. Tieto NPC označili hráči ako nečakane zábavné a bola to pre nich výzva. Tento pocit však trval len dovtedy, kým sa buď omylom alebo skúšaním nenaučili taktiku proti NPC. Ukázalo sa, že hneď ako sa hráčom sprístupní zbraň, tak začnú okolo seba náhodne strieľať. Naproti tomu NPC vystrelia len vtedy, keď majú hráča priamo pred sebou. Tento rozdiel v chovaní vychýlil v štatistikách presnosť hráčov. Slabinou prenasledujúcich NPC je, že sa otáčajú pomalšie než hráč, a to kvôli jednoduchosti a stabilite kontroléru. Ak sa hráč rýchlo dostane za NPC, tak NPC ho stratí z dohľadu, a aj napriek predikci hráčovej pozície to umožní hráčovi uniknúť. Ďalšia slabina NPC je, ak hráč za prekážkou zahne, tak je veľká šanca, že ho NPC už potom nenájde. S narastajúcimi skúsenosťami sa vyčlenili 2 typy hráčov: tí čo náhodne strieľajú v okolí NPC a dúfajú že ho trafia; a tí, čo využijú slabiny NPC, potom sa priplížia mimo zorného poľa NPC a jednoducho NPC zničia zo strany alebo zozadu, pretože na to obranu voči takýmto útokom programované nie sú. Tu sa už však prejavuje rozdiel vo vnímaní hráča a NPC. Hráč vidí celú obrazovku a NPC len určitý výsek. Na udržanie zábavnosti pre hráča by mohla pomôcť dynamická zmena obtiažnosti, napríklad upravením zorného poľa NPC alebo parametrov ich lodí.

6.1 Ovládanie lode

Kedže ovládanie lode bolo pre hru kľúčové, už v ranných fázach vývoja sa testovali s používateľmi rôzne prístupy k ovládaniu lode. Prvým z nich bolo otáčanie lode šípkami spolu s pohonom podpredu. Všetci testeri vrátane autora sa zhodovali, že je to „síce logické, ale v praxi divné“ a prakticky sa s tým nedalo hrať, alebo možno len po dlhom tréningu a v priestrannom prostredí. Ďalej navrhovali, že možnosť „cúvať“ by tomu pomohla. Po implementácii pohonu dozadu bolo ovládanie stále „divné“. Inšpirácia na to správne ovládanie lode prišla z hry Reassembly, kde sa ovláda vesmírna loď a síce v omnoho rozmernejšom prostredí a takmer úplne bez prekážok. Hra Reassembly poskytuje dynamické prepínanie 3 režimov ovládania. Prvý je zhodný s implementovaným, ktorý bol „divný“. V druhom prípade existuje pohon dopredu a dozadu relatívne k natočeniu lode, tiež pohon do strán (takmer) bez zmeny natočenia lode (tzv. „strafe“), pričom loď sa otáča vždy za myšou. Tento druhý spôsob bol implementovaný do hry a pre používateľov vyšiel ako omnoho lepší než prvý spôsob. Tretí spôsob ovládania v hre Reassembly je natočenie lode podľa myši, a absolútny pohyb lode klávesnicou (hore, dole, doprava, doľava) nezávisle na natočení lode. Takéto ovládanie je pre vytváranú hru nepraktické, pretože narozdiel od hry Reassembly sa odohráva v ohraničených priestoroch a uličkách, pričom týmto spôsobom by sa hráč dokázal pohybovať len po násobkoch 45° (4 smery a ich kombinácie).

Kapitola 7

Záver

Zadaním práce bolo naštudovať literatúru zaoberajúcu sa tvorbou počítačových hier a umelou inteligenciou v hrách. Literatúra hovorí, že hlavnou požiadavkou na inteligenciu v hrách je zábavnosť a primeraná výzva pre hráča, na čo sa v hernom priemysle často využívajú jednoduché techniky. Počas vývoja tejto práce sa potvrdilo, že na vyvolanie požadovaného zážitku stačia, možno nie práve elegantné, ale relatívne jednoduché a efektívne techniky. Hráči hru označili kvôli inteligencii nepriateľov za zábavnú a ako výzvu. To trvalo, až kým neodhalil taktiku proti NPC.

Počas vývoja sa ukázalo, že vzhľadom na autorove skúsenosti a schopnosti, pôvodne navrhovaná hra bola až príliš funkčne pokročilá a obsahovo rozsiahla, a preto nakoniec vznikla zjednodušená verzia hry. Málo obsahu negatívne vplývalo aj na hráčov, nehovoriac o absencii zvuku ako inak bežnej súčasti počítačových hier.

Vzniknutý herný engine je dostatočne výkonný, všestranný a znovupoužiteľný aj na iné typy hier.

V budúcnosti by sa teda mohla hra rozšíriť o viac obsahu a zvuky. Užitočná by mohla byť aj dynamické prispôsobenie obtiažnosti NPC podľa schopností hráča.

Literatúra

[1] *Box2D | A 2D Physics Engine for Games*. [cit. 2018-10-07]. Dostupné z: https://box2d.org/.

[2] Modular Behavior Tree. *CRYENGINE Programming – Documentation*. [cit. 2020-05-02]. Dostupné z: https://docs.cryengine.com/display/CEPROG/Modular+Behavior+Tree.

[3] Navigation. *Godot engine latest documentation*. [cit. 2020-03-27]. Dostupné z: https://docs.godotengine.org/en/3.2/classes/class_navigation.html.

[4] *HTML5 Game Engines*. [cit. 2019-10-05]. Dostupné z: https://html5gameengine.com.

[5] 2018 Events By the Numbers. *LoL Esports*. [cit. 2019-01-06]. Dostupné z: https://www.lolesports.com/en_US/articles/2018-events-numbers.

[6] Tech. *Nevermind*. [cit. 2020-03-01]. Dostupné z: https://nevermindgame.com/tech/.

[7] *SFML*. [cit. 2018-09-28]. Dostupné z: https://www.sfml-dev.org/.

[8] NumberKeyframeTrack. *Three.js docs*. [cit. 2020-04-15]. Dostupné z: https://threejs.org/docs/#api/en/animation/tracks/NumberKeyframeTrack.

[9] Navigation Mesh Reference. *UDK*. [cit. 2020-03-27]. Dostupné z: https://docs.unrealengine.com/udk/Three/NavigationMeshReference.html.

[10] Manual: Navigation System in Unity. *Unity*. [cit. 2020-03-27]. Dostupné z: https://docs.unity3d.com/Manual/nav-NavigationSystem.html.

[11] Behavior Trees. *Unreal Engine Documentation*. [cit. 2020-04-14]. Pre verziu Unreal Engine 4. Dostupné z: https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/BehaviorTrees/index.html.

[12] Nodegraph. *Valve Developer Community*. [cit. 2020-03-14]. Dostupné z: https://developer.valvesoftware.com/wiki/Nodegraph.

[13] Navigation Meshes. *Valve Developer Community*. [cit. 2018-11-08]. Dostupné z: https://developer.valvesoftware.com/wiki/Navigation_Meshes.

[14] Video game industry. *Wikipedia*. [cit. 2019-01-06]. Dostupné z: https://en.wikipedia.org/wiki/Video_game_industry.

[15] List of Unreal Engine games. *Wikipedia*. [cit. 2019-02-12]. Dostupné z: https://en.wikipedia.org/wiki/List_of_Unreal_Engine_games.

[16] k-nearest neighbors algorithm. *Wikipedia*. [cit. 2020-04-17]. Dostupné z: https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm.

[17] Variant type. *Wikipedia*. [cit. 2020-05-02]. Dostupné z: https://en.wikipedia.org/wiki/Variant_type.

[18] Netcode & 128-Servers // Dev Diaries - VALORANT. *YouTube*. [cit. 2020-04-15]. Dostupné z: https://www.youtube.com/watch?v=_Cu97mr7zcM.

[19] Booth, M. *The AI Systems of Left 4 Dead*. [cit. 2018-10-14]. Dostupné z: https://steamcdn-a.akamaihd.net/apps/valve/2009/ai_systems_of_l4d_mike_booth.pdf.

[20] Ceipek, J. *Game Path Planning*. [cit. 2018-10-14]. Dostupné z: http://jceipek.com/Olin-Coding-Tutorials/pathing.html.

[21] Fiedler, G. Fix Your Timestep! *Gaffer On Games*. [cit. 2019-02-01]. Dostupné z: https://gafferongames.com/post/fix_your_timestep/.

[22] Fiedler, G. What Every Programmer Needs To Know About Game Networking. *Gaffer On Games*. [cit. 2020-05-12]. Dostupné z: https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/.

[23] Gregory, J. *Context-Aware Character Dialog inThe Last of Us*. [cit. 2018-11-21]. Dostupné z: https://www.gameenginebook.com/resources/Gregory_Jason_ContextAwareDialog.pdf.

[24] Gregory, J. *Game Engine Architecture*. 1. vyd. A K Peters, 2009. ISBN 978-1-56881-413-1.

[25] Nystrom, R. *Game Programming Patterns*. Genever Benning [cit. 2018-09-20]. ISBN 978-0-9905829-0-8. Dostupné z: http://gameprogrammingpatterns.com/contents.html.

[26] OpenAI, Berner, C., Brockman, G., Chan, B., Cheung, V. et al. Dota 2 with Large Scale Deep Reinforcement Learning. 2019. Dostupné z: https://arxiv.org/abs/1912.06680.

[27] Overmars, M. *Game Maker Documentation*. Version 8.0; Microsoft Compiled HTML Help file, štandardne distribuované s programom Game Maker 8 ako súbor Game_Maker.chm.

[28] Reynolds, C. W. *Steering Behaviors For Autonomous Characters*. [cit. 2018-10-14]. Dostupné z: http://www.red3d.com/cwr/steer/gdc99/.

[29] Shiffman, D. *The Nature of Code*. 2012 [cit. 2018-10-10]. ISBN 978-0985930806. Dostupné z: https://natureofcode.com/book/.

[30] Silver, D., Huang, A. a Maddison, C. Mastering the game of Go with deep neural networks and tree search. *Nature*. 2016, zv. 529, s. 484–489, [cit. 2020-05-09]. Dostupné z: https://deepmind.com/research/publications/mastering-game-go-deep-neural-networks-tree-search.

[31] Simpson, C. Behavior trees for AI: How they work. *Gamasutra*. [cit. 2020-04-14]. Dostupné z: https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php.

[32] Smrček, P. Jak řídit živoucí svět v počítačové hře. *Youtube*. [cit. 2020-04-27]. Dostupné z: https://www.youtube.com/watch?v=oux1p0R7Fm0.

[33] Thompson, T. How A Navigation Mesh Works in 3D Games | AI 101. *Youtube*. [cit. 2019-06-13]. Dostupné z: https://www.youtube.com/watch?v=U5MTIh_KyBc.

[34] Waveren, J. *The Quake III Arena Bot*. 118 s. Diplomová práca. Dostupné z: https://www.researchgate.net/publication/240430519_The_Quake_III_Arena_Bot.

[35] Yannakakis, G. N. a Togelius, J. *Artificial Intelligence and Games*. Springer, 2018. ISBN 978-3-319-63518-7. Dostupné z: http://gameaibook.org.