F3

Fakulta elektrotechnická
Katedra počítačů

Bakalářská práce

Návrh multiplayer hry využívající klasické plánování

Aneta Drahoňovská
Softwarové inženýrství a technologie

Květen 2024
Vedoucí práce: Ing. Michaela Urbanovská

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Drahoňovská** Jméno: **Aneta** Osobní číslo: **511136**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

Specializace: **Enterprise systémy**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Návrh multiplayer hry využívající klasické plánování

Název bakalářské práce anglicky:

Design of a multiplayer game using classical planning

Pokyny pro vypracování:

Cílem práce je navrhnout a naimplementovat multiplayer hru, ve které bude hráč spolupracovat s umělou inteligencí, která bude poháněna klasickým planováním [1].
Hráč bude operovat v rozšířené verzi hry Grid Mario [2] modelované v PDDL [3], která bude upravena pro režim se dvěma hráči. Jedním hráčem bude AI hráč, jehož rozhodnutí budou generována z optimálních řešení pomocí libovolného integrovaného PDDL plánovače.
Úkolem je seznámit se s jazykem PDDL, rozšířit hru o nové mechaniky a mód pro více hráčů. Dále převést vše do herního enginu Unity [4] a zaintegrovat PDDL plánovač tak, aby bylo možné ho použít pro generování tahů AI hráče. Dalším úkolem je vytvoření úrovní hry, na kterých bude možné otestovat všechny namodelované mechaniky a celkovou funkčnost hry.
1. Seznamte se s problémovou doménou Grid-Mario a jazykem PDDL, ve které je formulovaná a navrhněte její vhodné rozšíření pro multiplayer
2. Zprovozněte libovolný plánovač v PDDL, prozkoumejte a implementujte jeho napojení na prostředí Unity
3. Navrhněte strukturu a mechaniky plánovací domény Grid Mario pro herní engine Unity
4. Naimplementujte všechny herní mechaniky a AI hráče poháněného klasickým plánovaním
5. Navrhněte sadu úrovní, na kterých bude možné otestovat všechny herní mechaniky a funkčnost hry

Seznam doporučené literatury:

[1] - N. Lipovetzky (2014). Structure and Inference in Classical Planning. AI Access.
[2] - Urbanovská, M. (2023). PUI Assignment 1-1 PDDL . https://cw.fel.cvut.cz/b222/courses/pui/assignments/assignment1-1
[3] - Ghallab, M.; Howe, A.; Knoblock, C.; Mcdermott, D.; Ram, A.;
Veloso, M.; Weld, D. & Wilkins, D. (1998), 'PDDL---The Planning
Domain Definition Language' .
[4] - Haas, J. K. (2014). A history of the unity game engine.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Michaela Urbanovská katedra počítačů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **25.01.2024** Termín odevzdání bakalářské práce: **24.05.2024**

Platnost zadání bakalářské práce: **21.09.2025**

Ing. Michaela Urbanovská podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

Poděkování / Prohlášení

Ráda bych vyjádřila své upřímné díky vedoucí této práce, paní inženýrce Urbanovské, za její pomoc a vstřícnost, kterou mi poskytla během celého procesu tvorby této závěrečné práce. Rovněž bych chtěla poděkovat svému příteli za jeho neustálou podporu a víru v mé schopnosti.

Prohlašuji, že jsem předloženou práci vypracovala samostatně a že jsem uvedla veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 17. 5. 2024

.....

Abstrakt / Abstract

V rámci této práce je úkolem seznámit se s konceptem klasického plánování a modelovacím jazykem PDDL, s cílem navrhnout multiplayer tahovou počítačovou hru. V tomto kontextu je kladen požadavek na kooperativní interakci mezi hráčem a AI, kde AI pro své rozhodování a následný pohyb využívá klasické plánování. V úvodní části práce se zaměřujeme na modelovací jazyk PDDL a definici klasického plánování. Dále se seznamujeme s vyhledávacími algoritmy, včetně jejich optimalizace pomocí heuristických funkcí. Paralelně analyzujeme a selektujeme relevantní nástroje, které efektivně odpovídají specifikacím a kritériím této práce. V dalším kroku se zabýváme podrobným popisem konceptu hry, návrhem jednotlivých úrovní a detailní strukturou PDDL domény, reprezentující svět hry. Tyto komponenty následně implementujeme ve vývojovém prostředí Unity.

Klíčová slova: klasické plánování; PDDL; Unity;

The aim of this work is to familiarize oneself with the concept of classical planning and the PDDL modeling language, with the goal of designing a multiplayer turn-based computer game. In this context, there is a requirement for cooperative interaction between the player and AI, where the AI uses classical planning for its decision-making and subsequent movements. The introductory part of this work focuses on the PDDL modeling language and the definition of classical planning. Furthermore, we delve into search algorithms, including their optimization using heuristic functions. Concurrently, we analyze and select relevant tools that effectively meet the specifications and criteria of this work. In the next step, we provide a detailed description of the game concept, the design of individual levels, and a detailed structure of the PDDL domain representing the game world. These components are then implemented in the Unity development environment.

Keywords: classical planning; PDDL; Unity;

Title translation: Design of a multiplayer game using classical planning

Obsah /

/ Obrázky

Kapitola 1
Úvod

Cílem této práce je návrh a implementace kooperativní multiplayer tahové hry, ve které druhého hráče realizujeme jako umělou inteligenci ve formě klasického plánovače a plánovací systém následně integrujeme do herního enginu a herního prostředí Unity.

V prvním kroku se zaměříme na analýzu problematiky plánování prostřednictvím domény Grid-Mario [1] definované v jazyce PDDL [2]. Následně bude provedena integrace a konfigurace vybraného plánovače s herním prostředím Unity. Dále se zaměříme na návrh struktury a mechanik plánovací domény spolu s implementací hry v Unity. V závěrečné sekci práce otestujeme klíčové části vyvinutého systému pomocí jednotkového testování a průchodu hrou uživateli.

Kapitola 2
Analýza

V této kapitole se seznámíme s teoretickými základy problematiky plánování a její optimálnosti, za cílem zvolení si vhodného algoritmu pro nalezení řešení úrovní naší hry a pohybu umělé inteligence, plánovacího systému a prostředí pro vývoj hry.

2.1 PDDL

PDDL [2] neboli Planning Domain Definition Language, představuje klíčový nástroj v oblasti klasického plánování, jenž je určen pro formulaci plánovacích domén a problémů. Plánování samotné je proces uvažování, který vybírá a organizuje akce předjímáním jejich očekávaných výsledků. Cílem této úvahy je co nejlépe dosáhnout některých předem stanovených cílů [3]. PDDL tedy slouží jako jazykový standard, usnadňující modelování plánovacích problémů.

Jazyk PDDL je postaven na predikátové logice, která poskytuje základ pro specifikaci plánovacích problémů. Pokud bychom měli definovat pojem logika, mohli bychom říci, že je to studium argumentace. Logika se snaží kodifikovat správné postupy, pomocí nichž vyvozujeme platné závěry z daných informací. Existují dvě základní úrovně klasické logiky – výroková a predikátová. Pod pojmem výrok můžeme rozumět tvrzení, o němž lze v principu rozhodnout, zda je pravdivé nebo nepravdivé. Výroková logika tak poskytuje pouze pravdivostní hodnotu konkrétního tvrzení a postrádá větší výrazovou a formulační schopnost potřebnou nejen v matematice. Vyšší úroveň klasické logiky, predikátová logika, rozšiřuje výrokovou logiku o prostředky pro vyjádření objektů, jejich vlastností – predikátů a vztahů mezi objekty [4].

Pro zadefinování plánovacího problému jsou klíčové dvě složky – problém a doména. Tyto prvky společně umožňují popsat konkrétní stavy a cíle, kterých je třeba dosáhnout prostřednictvím vyhledávacích algoritmů.

2.1.1 Doména

PDDL doména reprezentuje svět a jeho pravidla. Obecně doména sestává ze tří hlavních částí. V první části `:requirements` stanovuje jakých vlastností PDDL jazyka doména využívá. Zpravidla tato sekce zahrnuje typování objektů, které lze vnímat jako obdobu tříd v objektově orientovaném programování. Pokud je typování objektů vyžádáno, sekce `:types` pak obsahuje hierarchii typů objektů.

Ukázku podoby PDDL domény můžeme vidět na Obrázku 2.1. Jedná se o Pednaultův příklad [5] přepravy předmětů mezi domovem a prací pomocí kufříku [2].

V rámci deklarace domény jsou vlastnosti objektů a jejich vzájemné vztahy formulovány prostřednictvím predikátů, které nalezneme v sekci `:predicates`. V poslední části domény specifikujeme akční schémata, která definují, jaké akce lze aplikovat na objekty v dané doméně. Schémata jsou vždy uvozeny výrazem `:action` následovaným názvem dané akce.

```
(define (domain briefcase-world)
  (:requirements :strips :equality :typing :conditional-effects)
  (:types location physob)
  (:constants (B - physob))
  (:predicates (at ?x - physob ?l - location)
               (in ?x ?y - physob))
```

Obrázek 2.1. Pednaultův příklad PDDL domény.

2.1.2 Problém

V procesu plánování slouží problém k vymezení iniciálního stavu jednotlivých účastněných objektů a cílových podmínek, kterých musí být dosaženo. Definice PDDL problému vychází z kontextu konkrétní PDDL domény a zahrnuje specifikaci následujících klíčových prvků:

- `:domain` - název domény, ke které náleží
- `:objects` - seznam konkrétních instancí objektů, které jsou v problému přítomny, včetně svých typů
- `:init` - počáteční stav problému
- `:goal` - logický výraz predikátů, který musí být pravdivý v každém cílovém stavu

Počáteční stav představuje výchozí stav prostředí, ve kterém plánovací systém začíná svou činnost. Tento stav obsahuje informace o prvotním nastavení účastněných objektů skrze predikáty, které jsou definované v dané náležící doméně. Predikáty nám tedy sdělují údaje o vlastnostech jednotlivých objektů a jejich vzájemných vztazích. Naopak cíl skrze predikáty jasně vymezuje, co musí být splněno v každém cílovém stavu. Plánovací systém se snaží dosáhnout těchto podmínek provedením sekvence deterministických akcí, jež jsou stanovené v rámci domény spjaté s tímto problémem. V této práci se PDDL problém využívá pro formulaci jednotlivých úrovní hry.

Jako příkladový problém PDDL problému je Pednaultův problém na Obrázku 2.2. Pednaultův problém předpokládá, že člověk má doma slovník a kufřík s výplatou uvnitř. Cílem je přepravit tyto předměty do práce, ale nechat si výplatu doma [2].

```
(define (problem get-paid)
    (:domain briefcase-world)
    (:init (place home) (place office)
           (object p) (object d) (object b)
           (at B home) (at P home) (at D home) (in P))
    (:goal (and (at B office) (at D office) (at P home))))
```

Obrázek 2.2. Pednaultův příkladový problém.

2.2 Klasické plánování

Podčástí automatizovaného plánování, jež je odvětvím umělé inteligence, je klasické plánování. Klasické plánování spočívá v hledání posloupnosti akcí, jež mapuje daný počáteční stav na stav cílový, přičemž akce a prostředí jsou deterministické [6]. Takováto prostředí jsou charakterizována jednoznačně definovanými akcemi a postrádají pravděpodobnostní prvky. Konkrétně se tyto vlastnosti projevují tak, že opakované provedení téže akce ve stejném stavu vždy vede ke stejnému výsledku.

Umělou inteligenci lze definovat jako studium racionálního jednání, což implikuje, že plánování, neboli navrhnutí plánu sestaveného z posloupnosti akcí k dosažení svých cílů, je kritickou součástí umělé inteligence [7].

Problematiku klasického plánování lze popsat jako stavový model:

$$M = \langle S, s_0, S_G, A, f \rangle$$

- S je konečná a diskrétní množina stavů
- $s_0 \in S$ je počáteční stav
- $S_G \subseteq S$ je neprázdná množina cílových stavů
- $A(s) \subseteq A$ představuje množinu akcí v A, které jsou aplikovatelné v každém stavu $s \in S$
- $f(a, s)$ je deterministická přechodová funkce, kde $s' = f(a, s)$ je stav který vznikne aplikací akce $a \in A_s$ ve stavu s

Řešením tohoto modelu, neboli takzvaný *plán*, je posloupnost aplikovatelných akcí $a_0, ..., a_n$, která generuje stavovovou sekvenci $s_0, s_1, ..., s_n, s_{n+1}$, kde s_{n+1} je cílový stav [8].

Klasické plánování lze tak formulovat jako problém hledání cesty přes orientovaný graf, jehož uzly reprezentují stavy daného prostředí a jehož hrany zachycují stavové přechody, které jsou umožněny prostřednictvím akcí. Výpočetní náročnost klasického plánování pak vyplývá z počtu stavů, který je exponenciální s počtem problémových proměnných [8]. Orientovaný graf je charakterizován hranami, které jsou definovány jako uspořádané dvojice uzlů, s jedním uzlem sloužícím jako počátečním a druhým jako koncovým uzlem dané hrany [9].

2.2.1 Prohledávací algoritmy

K nalezení řešení problému klasického plánování se využívají vyhledávací algoritmy, mezi které se řadí například algoritmy Breadth-first search (BFS) a Depth-first search (DFS). BFS a DFS jsou dvě standardní metody pro takzvané neinformované prohledávání grafů. BFS zkoumá stále širší okolí počátečního uzlu, naopak DFS sleduje jednu cestu tak dlouho dokud je to možné a při uvíznutí ve slepé uličce se vrací zpět na rozcestí [10]. Nicméně, jak bylo popsáno výše, tyto neinformované algoritmy s sebou přinášejí značnou výpočetní náročnost, a proto jsou vhodné pouze pro malé množiny stavů.

Dalším problémem je, že výstupy některých těchto algoritmů mohou často být suboptimální řešení [11]. Lze tak o těchto algoritmech konstatovat, že nám buď poskytují optimální řešení za cenu zvýšené časové náročnosti, nebo naopak řešení za nízkou výpočetní dobu nicméně s ne vždy optimálním výsledkem. To znamená, že výstupy neinformovaných algoritmů neodpovídají kritériím optimalizace, která je důležitým aspektem pro pohyb naší umělé inteligence a analýzu tahů hráče v rámci této práce.

Pro implementaci naší umělé inteligence je nezbytné, aby vyhledávací algoritmus konzistentně generoval optimální cestu. To konkrétně zahrnuje schopnost nalezení nejkratší možné cesty, která je v rámci této práce chápána jako optimální plán.

2.2.2 Heuristické funkce

V kontextu řešení problému hledání optimální cesty se často do vyhledávacích algoritmů integrují heuristické funkce, obyčejně označované jako $h()$, s cílem zefektivnění prohledávání rozsáhlých stavových prostorů. Heuristické funkce mapují jednotlivé stavy na reálné hodnoty, které vyjadřují odhad vzdálenosti z daného stavu k cíli, a pomáhají tak navigovat prohledávací algoritmus. Algoritmy, které inkorporují tyto heuristiky, se obvykle označují jako informované [11].

Heuristické funkce lze kategorizovat do dvou skupin: doménově nezávislé a doménově závislé. Zatímco doménově závislé heuristiky využívají informací specifických pro daný problém nebo doménu, doménově nezávislé heuristiky se zaměřují na obecné postupy, které nevyžadují jejich detailní znalost.

2.2.3 Greedy Best-First Search

Mezi algoritmy využívajícími heuristické funkce patří Greedy Best-First Search, který se řídí pouze hodnotou zvolené heuristické funkce. Odhadovaná délka cesty z daného stavu n do stavu cílového je pak definována jako $f(n) = h(n)$, kde $f(n)$ je funkce podle které se stavy řadí k prohledávání.

Z hlediska výkonu, Greedy Best-First Search může najít řešení mnohem rychleji než níže zmíněný prohledávací algoritmus A*. Greedy Best-First Search může rychlostně překonat A* přibližně o třicet sedm procent, jelikož A* má větší vyhledávací prostor kvůli zahrnutí nákladu $g(n)$ [12], který určuje počet kroků potřebných k dosažení daného uzlu. Nicméně, i přes svou obecně nižší časovou složitost, Greedy Best-First Search nezajišťuje nalezení optimální cesty. Existuje riziko, že při hledání cesty může uváznout v lokálním minimu nebo volit cesty, které se na počátku jeví jako příznivé, nicméně vedou k suboptimálním řešením [11].

2.2.4 A* Search

Ze stejného principu jako Greedy Best-First Search konceptuálně vychází algoritmus A*. Oproti Greedy Best-First Search algoritmu však v sobě zahrnuje navíc člen $g(n)$ charakterizující cenu za dosažení daného uzlu. A* systematicky prochází možnými stavy se záměrem každým dalším krokem minimalizovat hodnotu funkce danou vztahem

$$f(n) = g(n) + h(n)$$

kde $g(n)$ určuje délku cesty z počátečního uzlu do uzlu n, a tak se optimálně přiblížit cíli.

Pro zajištění optimálnosti tohoto algoritmu musí však použitá heuristická funkce splňovat dvě klíčové podmínky. První podmínka je, že heuristická funkce musí být přípustná. Přípustnost znamená, že musí být splněn vztah

$$h(n) \leq h^*(n)$$

kde h^* je takovou heuristikou, která se vždy rovná délce nejkratší cesty v každém stavu. Heuristická funkce tedy nesmí přecenit odhad vzdálenosti od cíle. Druhou podmínkou je zajištění konzistentnosti heuristické funkce, která je zajištěna tehdy, když

$$h(n) \leq h(n') + c$$

kde n představuje uzel, n' jeho následníka a c udává cenu za provedení kroku z n do n' [11]. Protože je A* standardní algoritmus pro hledání optimálních řešení, byl zvolen pro hledání cesty i v této práci.

2.2.5 h^{max}

Jednou z možností heuristik je pro svou konzistentnost, a tedy i pro potřeby optimálního plánování, max heuristika, označovaná jako h^{max}. Myšlenkou h^{max} je aproximovat náklady na sadu předpokladů podle ceny nejdražšího z nich. Tato heuristika se zaměřuje pouze na dílčí cíle, které jsou vnímány jako nejnákladnější. Jelikož náklady na dosažení sady předpokladů nemohou být nižší než náklady na dosažení každého z nich, max heuristika je zároveň přípustná [13].

2.3 Plánovací systémy

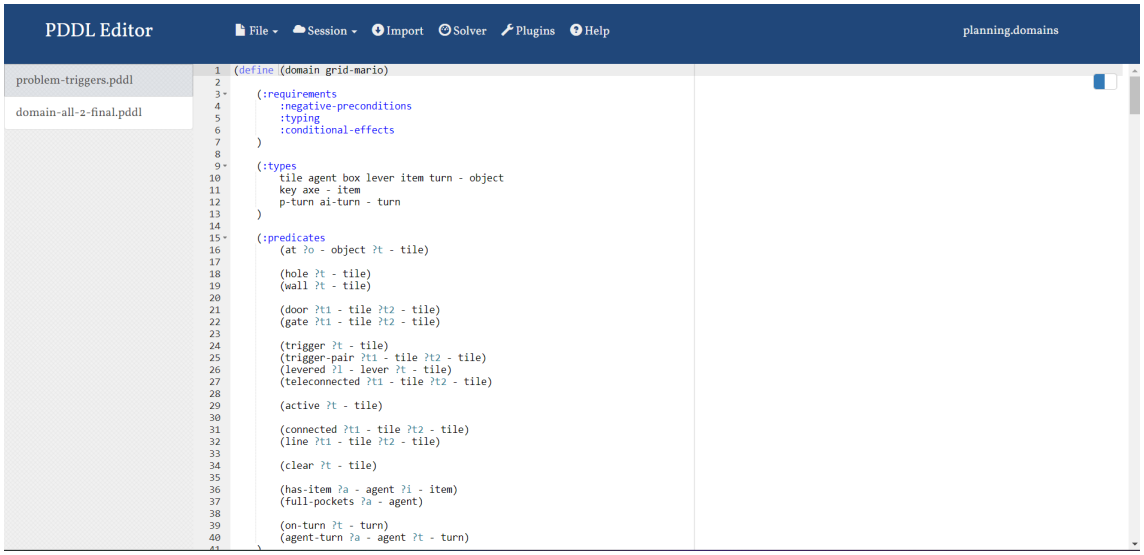
K automatickému generování plánů je nezbytné vybrat vhodný plánovací systém neboli plánovač. Pro potřeby této práce hledáme optimální a jednoduše integrovatelný plánovač. Současně vyžadujeme, aby plánovač byl doménově nezávislý, tedy řešil problémy bez specifických znalostí domény, na rozdíl od doménově závislých plánovačů [14].

V souladu s těmito požadavky jsme provedli rešerši na následující vybrané plánovací systémy.

2.3.1 Planning.Domains

Planning.Domains [15] je sada volně dostupných nástrojů pro usnadněnou práci s plánovacími doménami. Mezi tyto nástroje patří online Editor a Solver.

Editor, na Obrázku 2.3, je online nástroj pro tvorbu, úpravu a kontrolu korektnosti PDDL plánovacích domén a problémů. Poslouží tak jako vhodná pomůcka pro účely prvotního vývoje, debugu či vzdělávání. Online Editor byl v rámci této práce využit při modelování domény a jednotlivých úrovní hry.



Obrázek 2.3. Planning.Domains Editor.

Jako plánovací systém lze využít zmiňovaný Solver. Nicméně tato volba se ukázala jako nedostačující, zejména kvůli neoptimálnímu algoritmu, kterého tento systém využívá, a následně tak získávání neoptimálních řešení, která jsou nepřípustná pro správný chod naší umělé inteligence.

2.3.2 Fast Downward

Open-source plánovač Fast Downward [16], postavený na heuristickém vyhledávání, se stal preferovanou volbou pro účely a rozsah této práce z následujících důvodů. V první řadě, Fast Downward specializuje své plánování pro jazyk PDDL. Dále podporuje vyhledávací algoritmus A*, který v rámci plánovače garantuje spolu s využitím h^{max} heuristiky nalezení optimálního plánu, a to v krátkém čase. Současně je Fast Downward určen pro deterministické plánovací problémy.

2.4 Existující hry

Náš modelovaný problém plánování je situován na mřížce, která reprezentuje herní pole. Mezi logické hry odehrávající se na mřížce, které se používají jako benchmarky pro klasické plánování, se řadí tituly Sokoban [17] a Rush Hour [18]. Plánování je však využíváno i v rámci složitějších her, které jsou rovněž postaveny na systému mřížky, jako například série her GO. Tuto sérii her vyvinula společnost Square Enix [19].

2.4.1 Hitman GO

Hitman GO je mobilní verze oblíbeného holohlavého zabijáka z herní série Hitman pojaté ve stylu deskové hry. Každá úroveň je prezentována jako herní deska v podobě mřížky a vše od stráží až po samotného agenta se jeví jako malé herní figurky. Jedná se o tahovou hru, kde každý prvotní tah iniciuje hráč a poté se dostávají na řadu všichni nepřátelé. Každá mapa má jednoduchý úkol: dostat se na konec nebo zneškodnit cíl [20]. Snímek z této hry můžeme vidět na Obrázku 2.4.

Obrázek 2.4. Hitman GO.

2.4.2 Lara Croft GO

Jako následník Hitman GO byla vytvořena hra Lara Croft GO disponující obdobným konceptem, který je tentokrát zasazen do prostředí oblíbené herní série Tomb Raider. Oproti svému předchůdci si Lara Croft GO získala hráče nejen svou přívětivější estetikou připomínající kreslený film, ale zejména svou rozmanitostí v pohybu, který je rozšířen o vertikální rovinu [21], jak si můžeme všimnout na Obrázku 2.5.

Obrázek 2.5. Lara Croft GO.

Hádanky někdy vyžadují, aby Lara zatáhla za páku, která následně umožní změny v terénu. Stěny se tak kupříkladu posunou nahoru nebo dolů a plošiny se přesunou tak, aby Laře poskytly volnou cestu ven. Některé hádanky mají herní políčka posetá trhlinami; Lara může projít přes tyto prasklé prostory pouze jednou, protože druhý krok rozbije kámen a Lara se tak propadne do temnoty. Všechny tyto elementy přidávají další strategické prvky k již tak chytře navržené sadě hlavolamů, protože vyžaduje, aby si hráč své tahy opravdu pečlivě promyslel [21].

Lara Croft GO byla namodelována v podobě PDDL v rámci bakalářské práce [22], která se zabývala využitím klasického plánování na tento PDDL model hry.

2.4.3 Deus Ex Go

I přes své úspěchy sklidily oba předešlé tituly kritiku především za triviálnost jednotlivých úrovní. Společnost Square Enix tak přišla s další hrou této GO série, Deus Ex Go, jež je viditelná na Obrázku 2.6. Překonání úrovní Deus Ex Go vyžaduje chytrou kombinaci stealth schopností a pohybových vzorců nepřátel, aby mohl protagonista Adam Jensen, postava hráče, proklouznout bez povšimnutí [23].

Obrázek 2.6. Deus Ex Go.

2.5 Herní enginy

Pro implementaci naší hry je třeba zvolit herní engine a vývojové prostředí, které budou volně dostupné a umožní nám integraci plánovacího systému.

2.5.1 Unreal

K dnešním nejpopulárnějším herním enginům se řadí Unreal [24] engine od společnosti Epic Games, jehož jádro je psané v programovacím jazyce C++. Unreal je open-source a poskytuje specializované a celkově sofistikovanější nástroje, tudíž své zástupce má především z řad profesionálů v oblasti herního vývoje.

2.5.2 Unity

Herní engine a vývojové prostředí Unity [25] je oproti Unreal enginu populární volbou především pro svou přístupnost široké veřejnosti a rozsáhlou komunitu, která sdílí řadu tutoriálů a kompletních nástrojů pro volné použití. Tyto vlastnosti dnes činí z Unity jeden z nejoblíbenějších herních enginů, zejména pro prvotní seznámení se s problematikou za herním vývojem, a proto byl zvolen i pro účely této práce.

Kapitola 3
Návrh

V této kapitole se seznámíme s vizí herního prostředí, jeho objektů a pravidel a přiblížíme si průběh samotné hry.

3.1 Koncept

Hra nese název Adventures of Nico and Nita, ve které vystupuje dvojice kníratých hub - Nico (postava lidského hráče) a Nita (postava umělé inteligence), která se snaží překonat nástrahy lesa na své cestě domů. Hra se odehrává na libovolně velké mřížce na níž jsou rozprostřeny jednotlivé překážky, předměty a samotní hráči. Hlavním cílem hry je se svou figurkou dosáhnout vyznačeného políčka skrze interakci s objekty umístěnými na herním poli a spolupráci s postavou umělé inteligence. Dílčím cílem je tento proces provést s nejmenším možným počtem tahů.

3.2 Funkční a kvalitativní požadavky

Než se v procesu návrhu posuneme dále, musíme si nejprve stanovit, jaká očekávání neboli požadavky máme ohledně funkcionality naší hry.

Funkční požadavek je požadavek týkající se výsledku chování, který má zajistit funkce systému [26], na druhé straně nefunkční, neboli také kvalitativní požadavek, se týká aspektu kvality systému, která není pokryta funkčními požadavky. Tyto požadavky často ovlivňují architekturu systému více než funkční požadavky. Požadavky na kvalitu se obvykle týkají výkonu, dostupnosti nebo škálovatelnosti systému [26]. V následujících požadavcích naší hry je hráč chápán jako lidský hráč.

3.2.1 Funkční požadavky

- Hra umožní hráči se svou postavu interagovat s objekty umístěnými na herním políčku
- Hra umožní hráči svou postavu ovládat pomocí klávesnice
- Hra umožní hráči se se svou postavou pohybovat z jednoho políčka na druhé v jednu chvíli pouze v jednom směru (nahoru/dolů/doleva/doprava)
- Hra umožní hráči hru pozastavit
- Hra umožní hráči restartovat aktuální úroveň
- Hra umožní hráči úroveň opustit navigací do hlavního menu
- Hra umožní hráči si zvolit úroveň z nabídky
- Hra na konci každé úrovně zobrazí analýzu tahu hráče z aktuálního pokusu
- V menu pro selekci úrovně hra zobrazí nejvyšší dosažené skóre hráče každé úrovně
- Hra hráči poskytne návod pro ovládání postavy v hlavním menu
- Hra hráči umožní otáčet kamerou pro plnou viditelnost hracího pole

3.2.2 Kvalitativní požadavky

- Hra bude mít intuitivní klávesové ovládání
- Hra bude v angličtině

3.3 Pravidla, herní entity

V této podsekci si představíme jednotlivé entity hry spolu s jejich chováním.

3.3.1 Hráč

- může se pohybovat pouze ve čtyřech směrech
- může se pohybovat vždy maximálně o jedno políčko
- nemůže se přesunout na políčko, na kterém se nachází díra, zeď, druhý hráč nebo krabice, kterou v tomto směru nelze posunout
- může mít ve svém inventáři nanejvýš jeden předmět

3.3.2 Díra

- je vždy propojena s páčkou, která ji svou aktivací odstraní
- deaktivací páčky se navrátí
- hráč na ni nemůže vstoupit

3.3.3 Zeď

- nelze ji odstranit
- hráč na ni nemůže vstoupit

3.3.4 Krabice

- hráč ji může pohybem na stejné políčko posouvat, jestliže v tomto směru je políčko za ní volné - nenachází se zde díra, zeď, druhý hráč nebo další krabice
- lze ji odstranit pomocí sekery
- hráč nemůže vstoupit na stejné políčko, jestliže krabici v tomto směru nelze posunout nebo zničit

3.3.5 Teleport

- je vždy pojen s dalším teleportem ve dvojici
- hráč jej může aktivovat, jestliže se nachází na stejném políčku, a přesunout se tak na druhý teleport z této dvojice
- dvojice teleportů funguje obousměrně
- teleport lze aktivovat libovolněkrát

3.3.6 Páčka

- je vždy propojena s dírou
- hráč ji může aktivovat, jestliže se nachází na stejném políčku, a odstranit tak příslušnou díru
- po aktivaci ji lze opět deaktivovat

3.3.7 Dveře

- jsou umístěny na hraně dvou políček
- lze je otevřít pomocí jakéhokoliv klíče
- nelze projít skrze ně
- nelze je odstranit jinak než otevřením
- nelze je po otevření znovu zamknout

3.3.8 Brána

- je umístěna na hraně dvou políček
- otevře se, jestliže příslušná dvojice nášlapných desek je aktivována (stojí na nich hráči)
- nelze projít skrze ni
- nelze je odstranit jinak než otevřením

3.3.9 Nášlapná deska

- je pojena s další nášlapnou deskou ve dvojici
- je pojena ke konkrétní bráně
- hráč ji může aktivovat přesunutím se na stejné políčko
- deaktivuje se, jakmile hráč sestoupí z tohoto políčka
- po otevření brány jí již nelze aktivovat - stane se nefunkční

3.3.10 Klíč

- může být zvednut a umístěn do inventáře hráče
- jestliže je inventář hráče již plný, zamění se s předmětem, který hráč vlastní
- hráč jej může položit a uvolnit si tak místo v inventáři
- je univerzální - otevírá libovolné dveře
- po otevření dvěří zanikne

3.3.11 Sekera

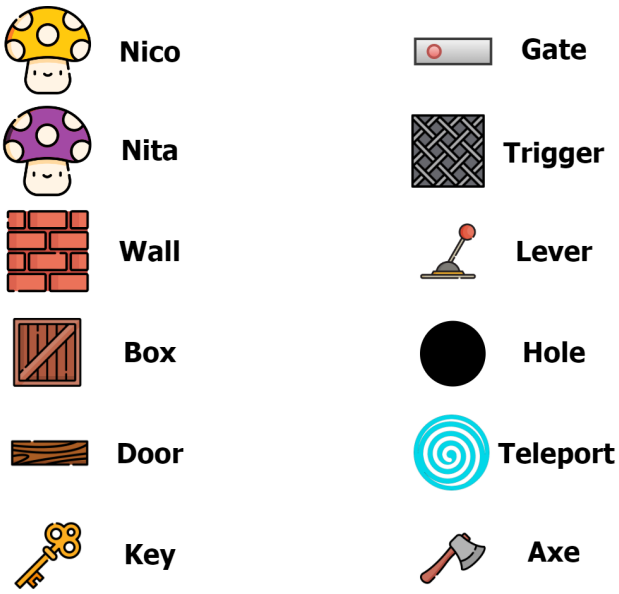
- může být zvednuta a umístěna do inventáře hráče
- jestliže je inventář hráče již plný, zamění se s tímto předmětem
- hráč ji může položit a uvolnit si tak místo v inventáři
- lze s ní odstranit jakoukoliv krabici
- po odstranění krabice nezaniká

3.4 Hratelnost

Hráč svou postavu ovládá pouze pomocí klávesnice. Postava hráče se může pohybovat z jednoho políčka na druhé v jednu chvíli pouze v jednom směru (nahoru/dolu/doleva/doprava), standardně buď pomocí šipek, nebo WASD. Interakce s objekty na herním poli je rozlišena na základě typu objektu. Na předměty (klíč/sekera) hráč působí prostřednictvím klávesy `I`. Pro uchopení předmětu do ruky, poté co daný předmět je již uložen v inventáři hráče, a tedy jeho použití musí hráč nejprve stisknout klávesu `E`. Konkrétní cíl, jehož se akce s předmětem dotýká (dveře/krabice), je pak určen pomocí pohybu hráče v tom samém tahu. Pro manipulaci s ostatními objekty (teleport/páčka) hráč využije klávesy `Space`. Zároveň hráč může klávesou `Q` svůj momentální tah kdykoliv přeskočit.

3.5 Level design

Celek hry tvoří jednotlivé úrovně hry, které postupně eskalují svou náročností vzhledem k optimální kombinaci tahů a synchronizaci s umělou inteligencí. Každá úroveň je hráči plně viditelná díky pohyblivé kameře, aby mu bylo umožněno hrát optimálně. Všechny úrovně byly modelovány v PDDL online Editoru, kde se zároveň testovala jejich správnost. Ačkoliv Editor negeneruje zpravidla optimální plány, pro ověření funkčnosti PDDL byl tento nástroj dostačující. Zároveň abychom měli garanci optimálních výsledků, plány jednotlivých úrovní byly získány z plánovacího systému Fast Downward.

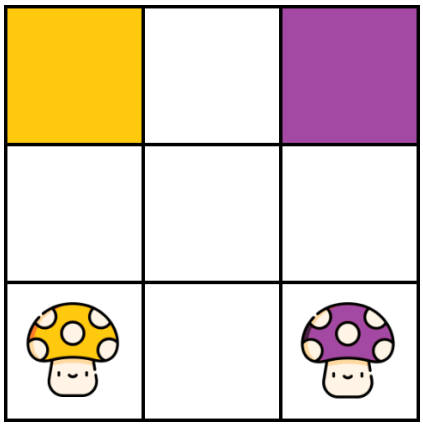


Obrázek 3.1. Legenda objektů v návrzích úrovní.

Ikonky pro reprezentaci jednotlivých entit na herních políčkách můžeme vidět na Obrázku 3.1. Tyto ikonky jsou použity v následujících návrzích úrovní, byly získány z webové stránky Flaticon [27] a samotné modely úrovní jsem vytvořila v softwaru diagrams.net [28].

3.5.1 Level 1

První úroveň sestává z pouhého pohybu hráčů po herním poli bez jediné překážky. Tento návrh zachycuje Obrázek 3.2.



Obrázek 3.2. Návrh první úrovně.

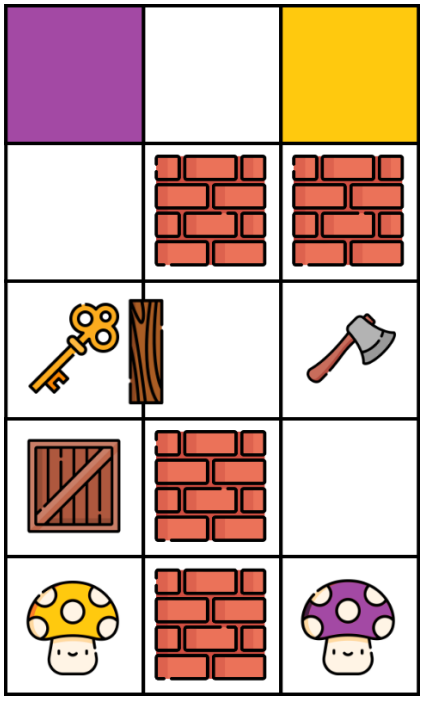
Problém tedy sestává z herního pole 3x3. Úkolem hráčů je se postupně dostat do horních rohů herní plochy. Optimální plán je vyobrazen na Obrázku 3.3

```
(move-agent nico tp tai t0x0 t0x1)
(move-agent-ai nita tp tai t2x0 t2x1)
(move-agent nico tp tai t0x1 t0x2)
(move-agent-ai nita tp tai t2x1 t2x2)
; cost = 4 (unit cost)
```

Obrázek 3.3. Optimální plán první úrovně.

3.5.2 Level 2

Druhá úroveň slouží k seznámení se s předměty, tedy sekerami a klíči. Úkolem lidského hráče je nejprve posouvat box, aby mohl dosáhnout políčka s klíčem, a otevřít dveře druhému hráči. Mezitím postava umělé inteligence se musí chopit sekery pro uvolnění cesty oběma hráčům k cíli. Úroveň ve formátu PDDL problém zachycuje Obrázek 3.4.



Obrázek 3.4. Návrh druhé úrovně.

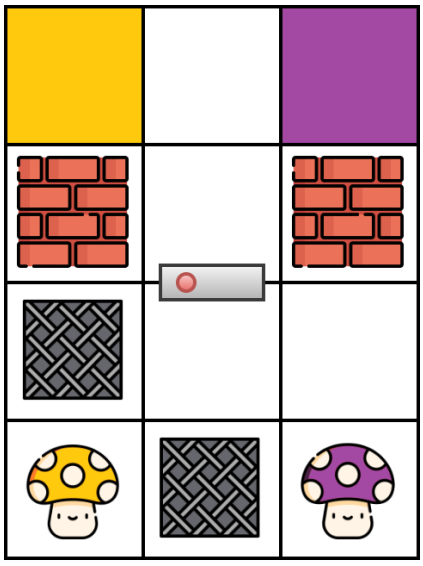
Narozdíl od první úrovně zde existuje více způsobů, jak tuto úroveň optimálně dokončit. Nicméně, Fast Downward nám jako první poskytne plán na Obrázku 3.5.

3.5.3 Level 3

Třetí úroveň obdobně jako předchozí je určena pro obeznámení se s novými objekty a to nášlapnou deskou a s ní spojenou bránou.

```
(push-box nico tp tai b1 t0x0 t0x1 t0x2)
(move-agent-ai nita tp tai t2x0 t2x1)
(push-box nico tp tai b1 t0x1 t0x2 t0x3)
(move-agent-ai nita tp tai t2x1 t2x2)
(pickup-item nico tp tai k1 t0x2)
(pickup-item-ai nita tp tai a1 t2x2)
(unlock-door nico tp tai k1 t0x2 t1x2)
(move-agent-ai nita tp tai t2x2 t1x2)
(push-box nico tp tai b1 t0x2 t0x3 t0x4)
(move-agent-ai nita tp tai t1x2 t0x2)
(skip nico tp tai)
(putdown-item-ai nita tp tai a1 t0x2)
(skip nico tp tai)
(move-agent-ai nita tp tai t0x2 t1x2)
(move-agent nico tp tai t0x3 t0x2)
(skip-ai nita tp tai)
(pickup-item nico tp tai a1 t0x2)
(skip-ai nita tp tai)
(move-agent nico tp tai t0x2 t0x3)
(move-agent-ai nita tp tai t1x2 t0x2)
(destroy-box nico tp tai a1 b1 t0x3 t0x4)
(move-agent-ai nita tp tai t0x2 t0x3)
(move-agent nico tp tai t0x4 t1x4)
(move-agent-ai nita tp tai t0x3 t0x4)
(move-agent nico tp tai t1x4 t2x4)
; cost = 25 (unit cost)
```

Obrázek 3.5. Optimální plán druhé úrovně.



Obrázek 3.6. Návrh třetí úrovně.

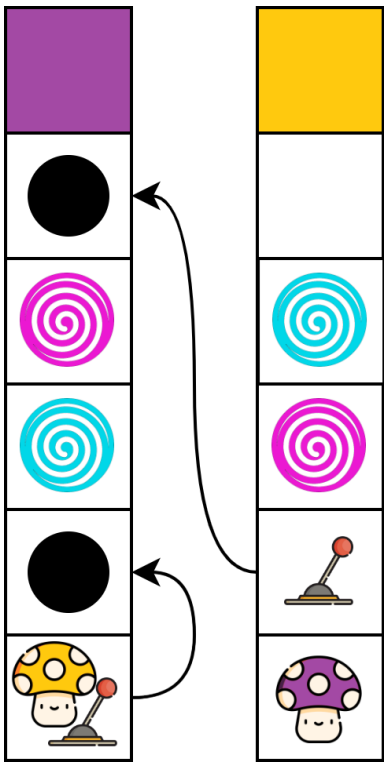
Pro otevření brány se musí oba hráči v jednom stavu hry nacházet na obou nášlapných deskách. PDDL problém této úrovně je ukázán na Obrázku 3.6 a získaný plán z plánovače pro tento problém pak na Obrázku 3.7.

```
(move-agent-on-trigger nico tp tai t0x0 t0x1 t1x0 t1x1 t1x2)
(move-agent-on-trigger-ai nita tp tai t2x0 t1x0 t0x1 t1x1 t1x2)
(move-agent nico tp tai t0x1 t1x1)
(skip-ai nita tp tai)
(move-agent nico tp tai t1x1 t1x2)
(move-agent-ai nita tp tai t1x0 t1x1)
(move-agent nico tp tai t1x2 t1x3)
(move-agent-ai nita tp tai t1x1 t1x2)
(move-agent nico tp tai t1x3 t0x3)
(move-agent-ai nita tp tai t1x2 t1x3)
(skip nico tp tai)
(move-agent-ai nita tp tai t1x3 t2x3)
; cost = 12 (unit cost)
```

Obrázek 3.7. Optimální plán třetí úrovně.

3.5.4 Level 4

Posledním úvodem do funkcionalit jednotlivých objektů je čtvrtá úroveň. Hráči zde musí pomocí páčky odstranit díru a následně se teleportovat do vedlejší části herního pole pro dosažení svého cílového políčka. Návrh úrovně lze vidět na Obrázku 3.8 a obsah obdrženého plánu na Obrázku 3.9.



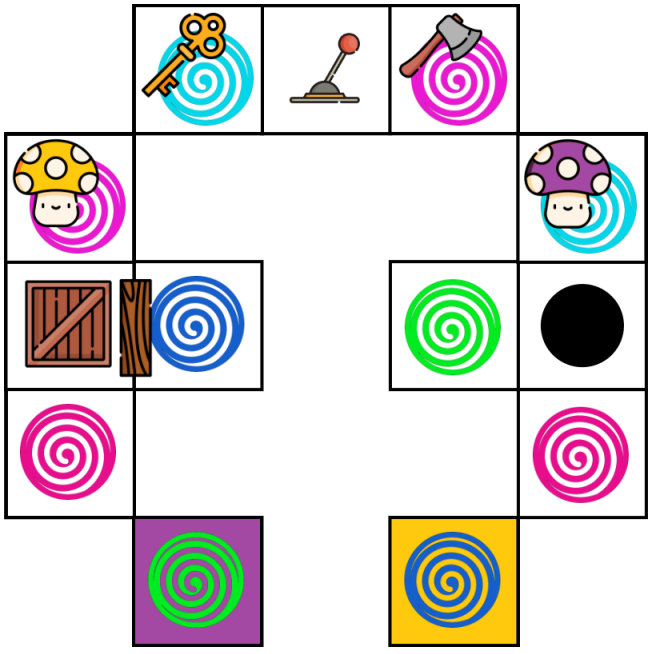
Obrázek 3.8. Návrh čtvrté úrovně.

```
(pull-lever nico tp tai t0x0 t0x1)
(move-agent-ai nita tp tai t2x0 t2x1)
(move-agent nico tp tai t0x0 t0x1)
(pull-lever-ai nita tp tai t2x1 t0x4)
(move-agent nico tp tai t0x1 t0x2)
(move-agent-ai nita tp tai t2x1 t2x2)
(use-teleport nico tp tai t0x2 t2x3)
(use-teleport-ai nita tp tai t2x2 t0x3)
(move-agent nico tp tai t2x3 t2x4)
(move-agent-ai nita tp tai t0x3 t0x4)
(move-agent nico tp tai t2x4 t2x5)
(move-agent-ai nita tp tai t0x4 t0x5)
; cost = 12 (unit cost)
```

Obrázek 3.9. Optimální plán čtvrté úrovně.

3.5.5 Level 5

První kombinací jednotlivých dvojic enit je pátá úroveň. Lidský hráč by se měl teleportovat a získat klíč. Postava umělé inteligence mezitím čeká na hráče. Lidský hráč dále uchopí klíč a otevře dveře, zatímco postava AI se teleportuje a zbaví se díry prostřednictvím páčky. Oba se následně skrze příslušné teleporty přemístí na cílová políčka. Kontrola správnosti tohoto problému, jehož návrh můžeme vidět na Obrázku 3.10, je oveřena získáním očekávaného plánu na Obrázku 3.11.



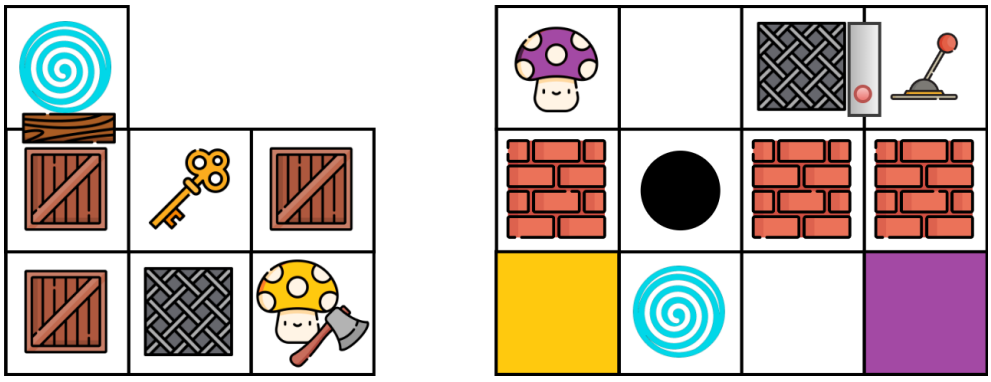
Obrázek 3.10. Návrh páté úrovně.

```
(use-teleport nico tp tai t0x3 t3x4)
(skip-ai nita tp tai)
(move-agent nico tp tai t3x4 t2x4)
(skip-ai nita tp tai)
(move-agent nico tp tai t2x4 t1x4)
(skip-ai nita tp tai)
(pickup-item nico tp tai k1 t1x4)
(skip-ai nita tp tai)
(move-agent nico tp tai t1x4 t2x4)
(use-teleport-ai nita tp tai t4x3 t1x4)
(pull-lever nico tp tai t2x4 t4x2)
(use-teleport-ai nita tp tai t1x4 t4x3)
(move-agent nico tp tai t2x4 t3x4)
(move-agent-ai nita tp tai t4x3 t4x2)
(use-teleport nico tp tai t3x4 t0x3)
(move-agent-ai nita tp tai t4x2 t3x2)
(push-box nico tp tai b2 t0x3 t0x2 t0x1)
(move-agent-ai nita tp tai t3x2 t4x2)
(unlock-door nico tp tai k1 t0x2 t1x2)
(move-agent-ai nita tp tai t4x2 t3x2)
(move-agent nico tp tai t0x2 t1x2)
(use-teleport-ai nita tp tai t3x2 t1x0)
(use-teleport nico tp tai t1x2 t3x0)
; cost = 23 (unit cost)
```

Obrázek 3.11. Optimální plán páté úrovně.

3.5.6 Level 6

V šesté úrovni by měl lidský hráč nejprve aktivovat nášlapnou desku a následně se ujmout sekery a zničit s ní krabici blokující dveře. Poté mu již zbývá pouze uvolněné dveře otevřít a teleportovat se ke svému cílovému políčku. Mezitím umělá inteligence taktéž nejprve vstoupí na políčko s nášlapnou deskou pro otevření brány, zatáhne za páčku a zprostředkuje si tak cestu k cíli.



Obrázek 3.12. Návrh šesté úrovně.

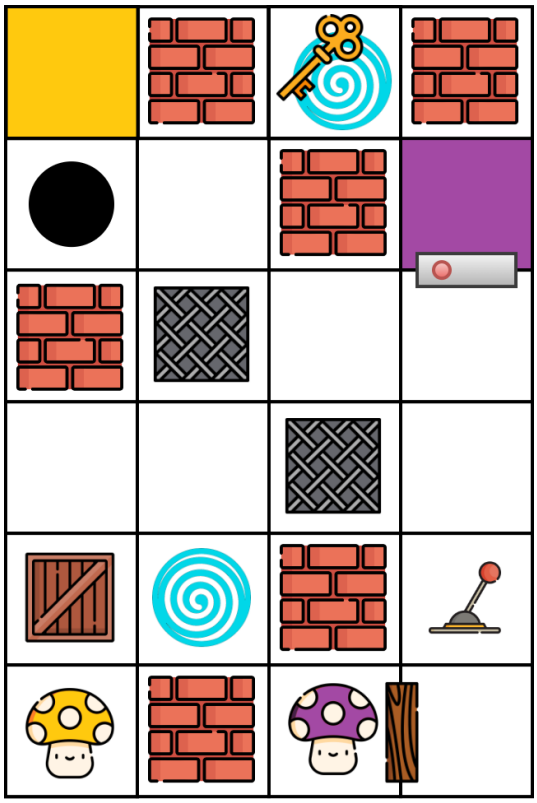
Podoba PDDL problému této úrovně je ukázána na Obrázku 3.12 a získaný plán z plánovače na Obrázku 3.13.

3.5.7 Level 7

Poslední úroveň sestává ze záchrany hráče umělé inteligence. Lidský hráč tedy musí získat klíč a vysvobodit svého spoluhráče odemknutím dveří. Následně spolu oba hráči

```
(pickup-item nico tp tai a1 t2x0)
(move-agent-ai nita tp tai t4x2 t5x2)
(move-agent-on-trigger nico tp tai t2x0 t1x0 t6x2 t6x2 t7x2)
(move-agent-on-trigger-ai nita tp tai t5x2 t6x2 t1x0 t6x2 t7x2)
(move-agent nico tp tai t1x0 t1x1)
(move-agent-ai nita tp tai t6x2 t7x2)
(destroy-box nico tp tai a1 b2 t1x1 t0x1)
(pull-lever-ai nita tp tai t7x2 t5x1)
(move-agent nico tp tai t0x1 t1x1)
(move-agent-ai nita tp tai t7x2 t6x2)
(swap-items nico tp tai a1 k1 t1x1)
(move-agent-ai nita tp tai t6x2 t5x2)
(move-agent nico tp tai t1x1 t0x1)
(move-agent-ai nita tp tai t5x2 t5x1)
(unlock-door nico tp tai k1 t0x1 t0x2)
(move-agent-ai nita tp tai t5x1 t5x0)
(move-agent nico tp tai t0x1 t0x2)
(move-agent-ai nita tp tai t5x0 t6x0)
(use-teleport nico tp tai t0x2 t5x0)
(move-agent-ai nita tp tai t6x0 t7x0)
(move-agent nico tp tai t5x0 t4x0)
; cost = 21 (unit cost)
```

Obrázek 3.13. Optimální plán šesté úrovně.



Obrázek 3.14. Návrh sedmé úrovně.

pokračují na nášlapné desky a dále ke svému cíli. Úroveň ve formátu PDDL problému zachycuje Obrázek 3.14. Fast Downward plánovač nám pak poskytne plán viditelný na Obrázku 3.15.

```
(push-box nico tp tai b1 t0x0 t0x1 t0x2)
(skip-ai nita tp tai)
(move-agent nico tp tai t0x1 t1x1)
(skip-ai nita tp tai)
(use-teleport nico tp tai t1x1 t2x5)
(skip-ai nita tp tai)
(pickup-item nico tp tai k1 t2x5)
(skip-ai nita tp tai)
(use-teleport nico tp tai t2x5 t1x1)
(skip-ai nita tp tai)
(move-agent nico tp tai t1x1 t1x2)
(skip-ai nita tp tai)
(move-agent-on-trigger nico tp tai t1x2 t2x2 t1x3 t3x3 t3x4)
(skip-ai nita tp tai)
(move-agent nico tp tai t2x2 t3x2)
(skip-ai nita tp tai)
(move-agent nico tp tai t3x2 t3x1)
(skip-ai nita tp tai)
(move-agent nico tp tai t3x1 t3x0)
(skip-ai nita tp tai)
(unlock-door nico tp tai k1 t3x0 t2x0)
(skip-ai nita tp tai)
(move-agent nico tp tai t3x0 t3x1)
(move-agent-ai nita tp tai t2x0 t3x0)
(move-agent nico tp tai t3x1 t3x2)
(move-agent-ai nita tp tai t3x0 t3x1)
(move-agent nico tp tai t3x2 t3x3)
(pull-lever-ai nita tp tai t3x1 t0x4)
(move-agent nico tp tai t3x3 t2x3)
(move-agent-ai nita tp tai t3x1 t3x2)
(move-agent-on-trigger nico tp tai t2x3 t1x3 t2x2 t3x3 t3x4)
(move-agent-on-trigger-ai nita tp tai t3x2 t2x2 t1x3 t3x4 t3x3)
(move-agent nico tp tai t1x3 t1x4)
(move-agent-ai nita tp tai t2x2 t3x2)
(move-agent nico tp tai t1x4 t0x4)
(move-agent-ai nita tp tai t3x2 t3x3)
(move-agent nico tp tai t0x4 t0x5)
(move-agent-ai nita tp tai t3x3 t3x4)
; cost = 38 (unit cost)
```

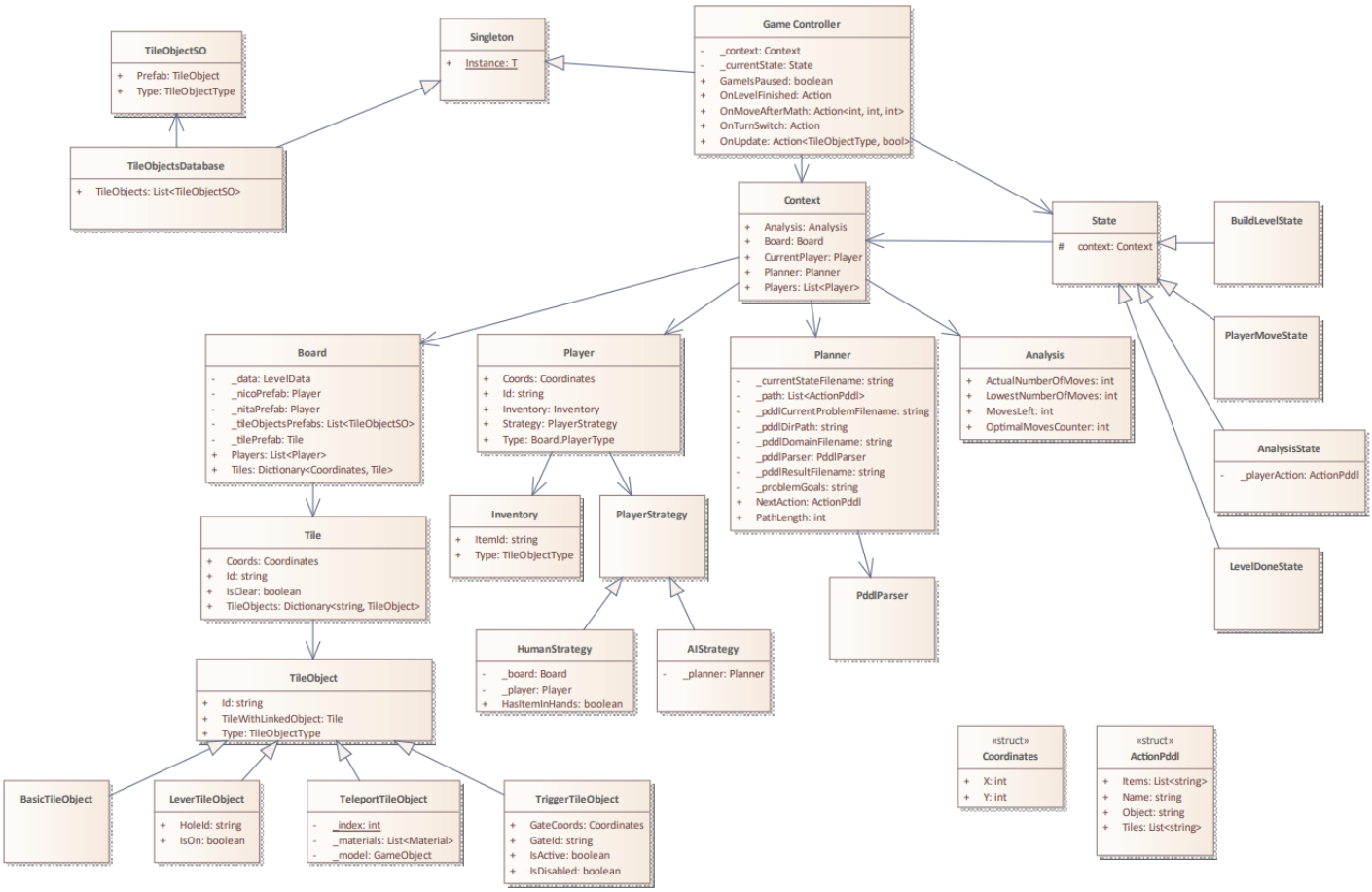
Obrázek 3.15. Optimální plán sedmé úrovně.

3.6 Diagram tříd

Diagram tříd je grafická reprezentace pohledu na systém z hlediska charakterizace jeho částí a jejich vzájemných statických vztahů. Zahrnuje jejich klasifikátory, obsah a vztahy. Jako klasifikátor se označuje modelový prvek, který popisuje behaviorální a strukturální rysy. Mezi druhy klasifikátorů se řadí například třída a rozhraní [29]. Na Obrázku 3.16 je vyobrazen diagram tříd naší vyvíjené hry.

3.6.1 Game Controller

Třída `GameController` spravuje stav hry a zastřešuje všechny ostatní třídy. Obsahem této třídy je pravdivostní hodnota o pozastavení hry a refernce na třídu `Context`, která má v sobě uloženy odkazy na herní pole, plánovač a analýzu. Pro naslouchání odkud-



Obrázek 3.16. Diagram tříd.

koliv na vyvolání údálostí, jež jsou definované na třídě `Game Controller`, tato třída implementuje návrhový vzor Singleton.

Singleton umožňuje zajistit, aby třída měla pouze jednu instanci, a zároveň poskytuje globální přístupový bod k této instanci [30].

Samotné události (event Action) v C# interně realizují návrhový vzor Observer. Observer nám umožňuje definovat mechanismus odběru pro upozornění více objektů na jakékoli události, které se stanou pozorovanému objektu [30]. `Game Controller` zde vystupuje jako `publisher` a poskytuje akce, které mohou být ostatními objekty (`subscribers`) odebírány:

- **OnTurnSwitch** - notifikace o změně tahu, vyměnit viditelnost UI textů `Your turn` a `AI turn`
- **OnLevelFinished** - notifikace o splnění cílových podmínek dané úrvoně, poskytnout `After Menu` UI
- **OnMoveAfterMath** - notifikace o dokončení analýzy pohybu hráče, aktualizovat UI texty reflektující statistické proměnné
- **OnUpdate** - notifikace o dokončení snímku, aktualizovat UI text, který poskytuje informaci o tom, zda se předmět z inventáře ocitá v rukou hráče

V našem systému je odběratelem třída `InGameUIManager`, která zapoudřuje veškeré UI herní scény. Jakmile publisher vyvolá akci, všichni odběratelé této události jsou notifikováni a automaticky se zavolá funkce odběratele, která je na tuto událost navě-

šena. Klíčové slovo event před třídou `Action` umožnuje akci vyvolat (.Invoke()) pouze ve třídě, ve které je akce deklarována.

3.6.2 Context

Jak již bylo uvedeno výše, třída `Context` slouží jakožto kontejner dat, jež spolu tvoří logiku hry. Tato třída existuje pro poskytování dat stavům. Jelikož přístup k třídě `Game Controller` je globální, `Context` tato data zapouzdřuje s cílem předejít jejich možnou modifikaci jinými třídami.

3.6.3 Board

Herní pole je modelováno jako mřížka, jejíž koncept je pro své vlastnosti hojně využíván při vývoji taktických her. Pohyb hráče je prostřednictvím mřížky jednoznačně vymezen, což přispívá deterministickému určení, v jakém stavu, tedy sadě pravdivých predikátu, se hra nachází. Takováto vlastnost je nedílnou součástí naší hry v rámci vyhovění předpokladů kladených na prostředí problému klasického plánování. Herní pole obsahuje seznam hráčů a svá veškerá políčka má uložené ve slovníku.

3.6.4 Tile

Celé herní pole je tvořeno jednotlivými herními políčky. Políčko je identifikováno pomocí svých souřadnic v rámci 2D herního pole. Pro potřeby určení konkrétní instance v kontextu PDDL problému je políčko navíc rozlišitelné svým ID. Každé herní políčko s sebou nese informaci o objektech, které se na něm aktuálně nachází. Počet přítomných objektů může nabývat nanejvýše sedmi instancí. Zároveň tato třída obsahuje pravdivostní hodnotu nesoucí informaci o tom, zda může hráč na políčko vstoupit.

3.6.5 Player

Postava hráče poskytuje informace o identifikátoru, typu (human/AI) a aktuální pozici hráče opět prostřednictvím souřadnic. Každý hráč si zároveň spravuje inventář v podobě třídy `Inventory`, jejíž obsahem je dvojice dat - ID drženého předmětu a typ tohoto předmětu, který je určen výčtovým typem `TileObjectType`. Pro odlišné chování obyčejného hráče a umělé inteligence v rámci stejného stavu hry jsem využila návrhový vzor Strategy.

Strategy navrhuje, abychom vzali třídu, která dělá něco konkrétního mnoha různými způsoby, a extrahovali všechny tyto algoritmy do samostatných tříd nazývaných strategie. Původní třída, neboli kontext, pak deleguje práci na propojený strategy objekt místo toho, aby ji provedla sama [30]. V našem případě jako kontext vystupuje stav `PlayerMoveState`, který reprezentuje fázi pohybu aktuálního hráče během chodu hry.

3.6.6 TileObject

Samotné objekty, které se můžou nacházet na herních políčkách, jsou souhrně označovány jako třída `TileObject`. Tato abstraktní třída v sobě obdobně jako předchozí třídy nese data o svém identifikátoru a typu objektu. Navíc však obsahuje referenci na políčko, na kterém se nachází související objekt, jestliže daný objekt nějaký má. Z `TileObject` dědí následující třídy:

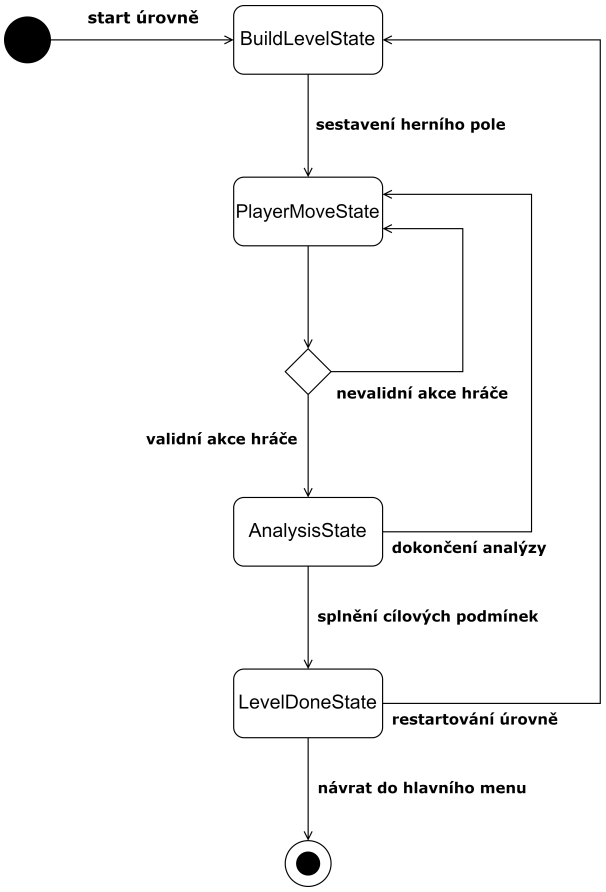
- **BasicTileObject** - vetšina objektů (díra, zeď, krabice, dveře, brána, klíč, sekera), neobsahuje další data
- **LeverTileObject** - rozšiřuje abstrakt o informaci, zda je páčka zaktivována a taktéž obsahuje ID pojené díry

- **TriggerTileObject** - rozšiřuje abstrakt o informaci, zda je nášlapná deska aktivní, zda již deska není funkční, ID pojené brány a její souřadnice
- **TeleportTileObject** - obsahuje v sobě seznam materiálů, které barevně rozlišují dvojici teleportů od ostatních dvojic instancí toho typu

3.6.7 State

Pro samotný chod hry využijeme návrhový vzor State, který úzce souvisí s konceptem Finite-State Machine. Hlavní myšlenkou tohoto konceptu je, že v každém daném okamžiku existuje konečný počet stavů, ve kterých se program může nacházet. V rámci každého tohoto stavu se program chová odlišně a může okamžitě přepnout z jednoho stavu do druhého. V závislosti na aktuálním stavu se však program může nebo nemusí přepnout do určitých jiných stavů. Tato pravidla přepínání, nazývaná přechody, jsou také konečná a předem určená [30].

State navrhuje, abychom vytvořili nové třídy pro všechny možné stavy objektu a do těchto tříd extrahovali všechna stavově specifická chování. Původní objekt, nazývaný kontext, pak ukládá referenci na jeden z objektů stavu, který představuje jeho aktuální stav, a deleguje veškerou práci související se stavem na tento objekt [30]. Jako kontext v našem případě vystupuje třída `GameController`.



Obrázek 3.17. Stavový diagram průběhu hry.

Celý průběh jedné úrovně hry tak sestává ze čtyř stavů, které mezi sebou přechází způsoby znázorněnými stavovým diagramem na Obrázku 3.17. Blíže jsou jednotlivé stavy specifikovány následovně:

- **BuildLevelState** – prvotní fáze hry pro vykreslení a inicializaci hráčů a herního pole spolu s odpovídajícími objekty
- **PlayerMoveState** – fáze tahu aktuálního hráče (proces tahu je mezi hráči odlišen pomocí zmíňovaného návrhového vzoru Strategy)
- **AnalysisState** – fáze zhodnocení optimálnosti hráčova tahu, v případě provedení neočekávané (neoptimální) akce ze strany lidského hráče zde proběhně komunikace s Docker pro získání nové optimální cesty z momentálního stavu hráčů a objektů na herním poli
- **LevelDoneState** – fáze ukončení, hráči je vyobrazena statistika jeho průchodu úrovní

3.6.8 Planner

Úlohou třídy `Planner` je komunikace s Docker a následné parsování získaného výstupu do formátu plánu. Plánu, uloženého v podobě seznamu skládajícího se z jednotlivých akcí, si můžeme povšimnout na Obrázku 3.18. Akce jsou předepsány strukturou `ActionPddl`, jejíž deklaraci vidíme na Obrázku 3.19. Tato struktura obsahuje informace o názvu a parametrech akce.

Obrázek 3.18. Přeparsovaný plán druhé úrovně v podobě seznamu.

Pokaždé, když hráč svým tahem nekopíruje očekávanou akci, na základě podnětu třídy `Planner` poskytne třída PddlParser nový PDDL problém, který reflektuje momentální stav herního pole. Tyto vytvořené PDDL problémy jsou následně `Planner`em odesílány do Docker image plánovacího systému Fast Downward. Odpovědí plánovače na tento příkaz je nový plán pro aktuální stav hry, který je opět zpracován způsobem popsaným výše.

```
public struct ActionPddl
{
    15 references
    public string Name { get; set; }
    18 references
    public List<string> Tiles { get; set; }
    9 references
    public List<string> Items { get; set; }
    10 references
    public string Object { get; set; }
```

Obrázek 3.19. Struktura `ActionPddl`.

3.6.9 PddlParser

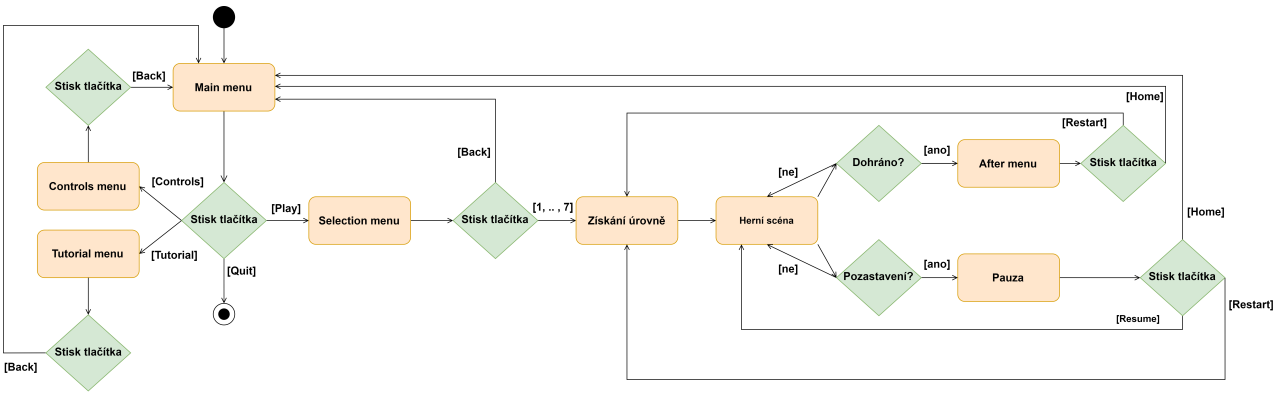
Pro převedení aktuálního stavu hry do syntaxu PDDL problému je určena třída `PddlParser`. Třída tedy vytvoří příslušné predikáty o každém políčku, jeho objektech a také jednotlivých hráčích.

3.6.10 Analysis

`Analysis` slouží jako data kontejner pro tři numerické proměnné, které společně poskytují vyhodnocení optimálnosti tahů lidského hráče. Jedná se o hodnoty určující počet nejméně možných kroků pro dokončení úrovně ze strany hráče, počet tahů hráče a poslední proměnná definuje kolik z jeho tahů bylo optimálních.

3.7 Navigace hrou

Pro vizuální zobrazení všech možných akcí a přechodů, které může hráč v průběhu hry učinit, převedeme naši hru do podoby UML diagramu aktivit, který je zachycen na Obrázku 3.20. Diagram aktivit je zvláštní případ stavového automatu, ve kterém jsou všechny stavy nebo většina z nich akcí nebo aktivitou. Stejně tak všechny nebo většina přechodů mezi stavy jsou spuštěny dokončením činnosti, ze které přechod vychází. Diagram tedy zachycuje workflow systému [29].



Obrázek 3.20. Diagram průchodu hrou.

Hra se tedy bude skládat ze dvou scén: hlavního menu a hry samotné. V hlavním menu bude mít hráč na výběr ze čtyř tlačítek. Tlačítko `Tutorial` zobrazí menu `Tutorial Menu`, které bude obsahovat seznam `TileObjects` spolu s krátkým popiskem jejich chování. Podobně tlačítko `Controls` přenese hráče do menu

`Controls Menu`, kde bude moci prozkoumat seznam akcí, které může ve hře se svou postavou aplikovat, spolu s přiřazenými klávesami. Tlačítkem `Quit` bude moct hru ihned ukončit, a nakonec prostřednictvím tlačítka `Play` se hráč přenese do menu `Selection Menu`, ve kterém si zvolí jednu z úrovní. Zde taktéž nalezne svá skóre z předchozích pokusů. Po výběru úrovně se získá a načte konfigurační soubor dané úrovně a hra následně přejde do samotné herní scény.

V herní scéně bude hráči neustále k dispozici tlačítko pro pozastavení hry. Jestliže dojde k přerušení, objeví se menu `Pause Menu`, jehož součástí budou tlačítka `Home`, `Restart` a `Resume`. Tlačítko `Home` přesměruje hráče zpět do první scény hlavního menu. Dále tlačítko `Restart` provede opětované načtení úrovně a tlačítkem `Resume` se hra navrátí do svého běhu.

Po úspěšném dokončení úrovně se zobrazí `After Menu`, které hráči poskytne analýzu jeho tahů a dvě tlačítka - `Home` a `Restart`, která budou mít obdobnou funkcionalitu jako ve fázi pozastavení hry.

3.8 Sekvenční diagram

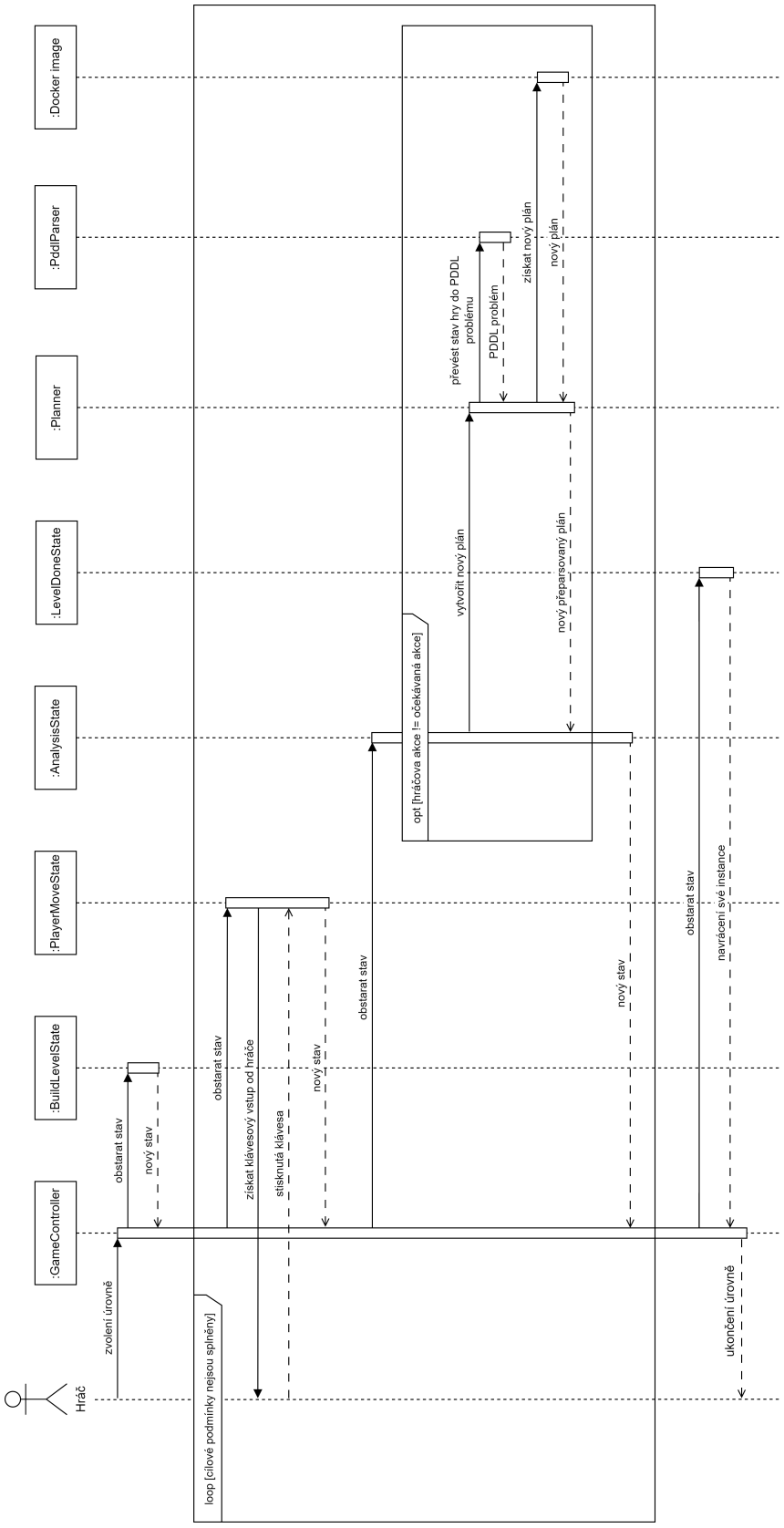
K modelaci sekvence komunikačních zpráv mezi klíčovými třídami naší hry jsem využila UML sekvenční diagram, jehož podoba je viděna na Obrázku 3.21. Tento diagram obecně vyobrazuje interakci objektů modelovaného systému jako dvourozměrný graf. Vertikální osa reprezentuje časovou osu, zatímco horizontální osa zobrazuje vzájemně vyměňované zprávy mezi objekty. Každý objekt je znázorněn klasifikační rolí, která definuje jeho úlohu v dané části systému, a doba jeho existence je vyjádřena svislým sloupcem, který se nazývá čára života. Zprávy jsou v diagramu uspořádány v časové posloupnosti [29].

Sekvenční diagram naší hry reflektuje sekvenci volání klíčových částí systému v průběhu úrovně. Konkrétně poukazuje na komunikaci mezi jednotlivými stavy. Po sestavení herního pole v `BuildLevelState` se již střídají pouze stavy `PlayerMoveState` a `AnalysisState`. Pokud hráč provede akci, která neodpovídá plánu, je stav hry převeden do podoby PDDL problému pro komunikaci s Docker image plánovače Fast Downward. Následně je z plánovače získán nový aktuální plán, který je pomocí třídy `Planner` přeparsován do podoby seznamu. Nakonec, jakmile `AnalysisState` stav vyhodnotí, že hráči splnili cílové podmínky dané úrovně, hra přechází do posledního stavu - `LevelDoneState`.

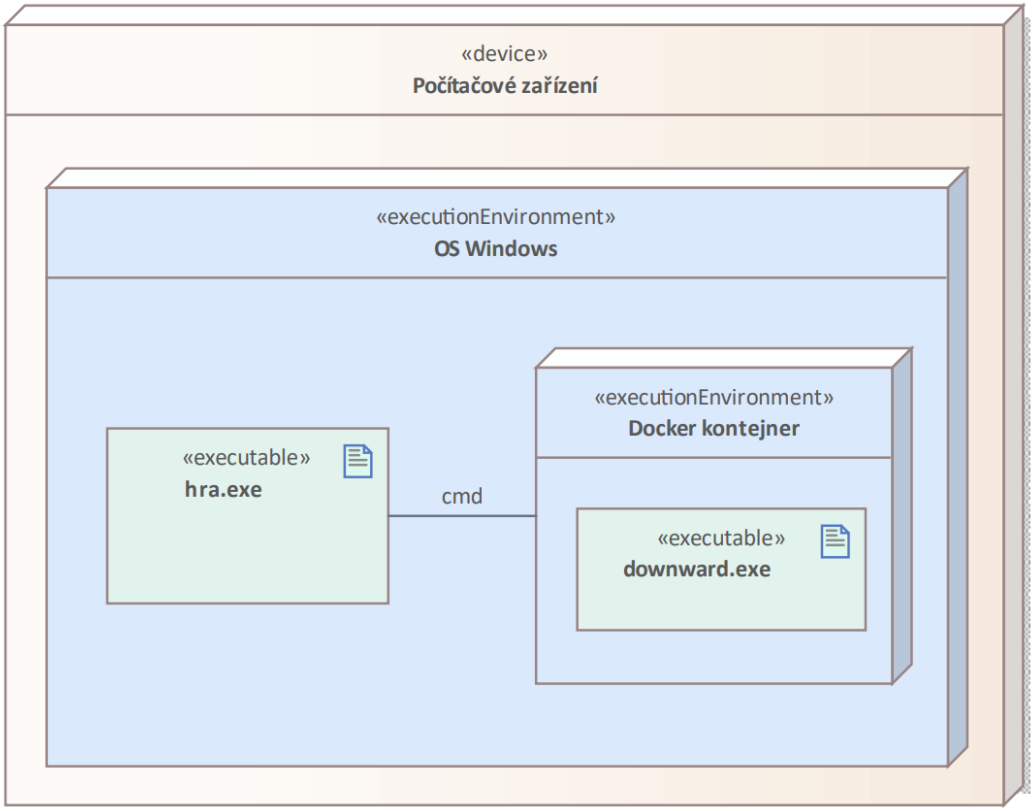
3.9 Diagram nasazení

Návrh naší hry uzavřeme popsáním konfigurace běžícího systému v prostředí reálného světa, tedy jeho nasazení na hardware. Diagram nasazení nám poskytuje pohled na systém jako na uzly, které zastupují fyzická zařízení modelovaného systému. Na každém uzlu jsou uloženy komponenty [29], jež reprezentují využívaný software a artifakty, které jsou obvykle fyzickým projevem jedné či více komponent. Mezi artifakty se řadí například fyzické a spustitelné soubory, knihovny či dokumenty [31].

Vyvíjená hra je určena pro zařízení s operačním systémem Windows. Vyexportovaný spustitelný soubor hry z Unity (hra.exe) komunikuje s Docker image plánovacího systému Fast Downward (downward.exe) skrze příkaz uvnitř skriptu `Planner` na tento image.



Obrázek 3.21. Sekvenční diagram.



Obrázek 3.22. Diagram nasazení.

Kapitola 4
Implementace

V této kapitole si popíšeme zvolené postupy pro implementaci jednotlivých prvků hry. Zároveň budou představeny klíčové nástroje a technologie, které byly vybrány s cílem splnit stanovená kritéria a požadavky této práce.

4.1 Nástroje a technologie

V této sekci se seznámíme s plánovacím systémem Fast Downward a jeho napojením do Unity prostřednictvím Docker image. Dále si ukážeme návrh PDDL domény a PDDL problému, včetně jejich následné implementace v herním enginu Unity.

4.1.1 Fast Downward

K automatickému generování plánů jsme vybrali plánovací systém Fast Downward. Tento volně dostupný plánovač nám spolu s využitím algoritmu A* a h^{max} heuristiky poskytuje optimální plány a řídí tak pohyb postavy umělé inteligence.

4.1.2 Docker

Docker je open-source platforma navržená s cílem podpory vývoje, nasazení a provozu aplikací v kontejnerech. Kontejner představuje izolovaný běžící proces na hostitelském počítači, oddělený od ostatních procesů tohoto systému. V kontextu Docker je klíčovým pojmem `image`, který reprezentuje základní souborový systém obsahující všechny potřebné komponenty pro spuštění aplikace – zahrnuje závislosti, konfigurace, skripty a tak dále. Hlavním cílem Docker je zajištění konzistentního spouštění aplikací bez ohledu na operační systém [32].

Zmíněný plánovač Fast Downward je k dispozici v různých distribučních formátech včetně Docker image, ve kterém je využíván i v rámci této práce.

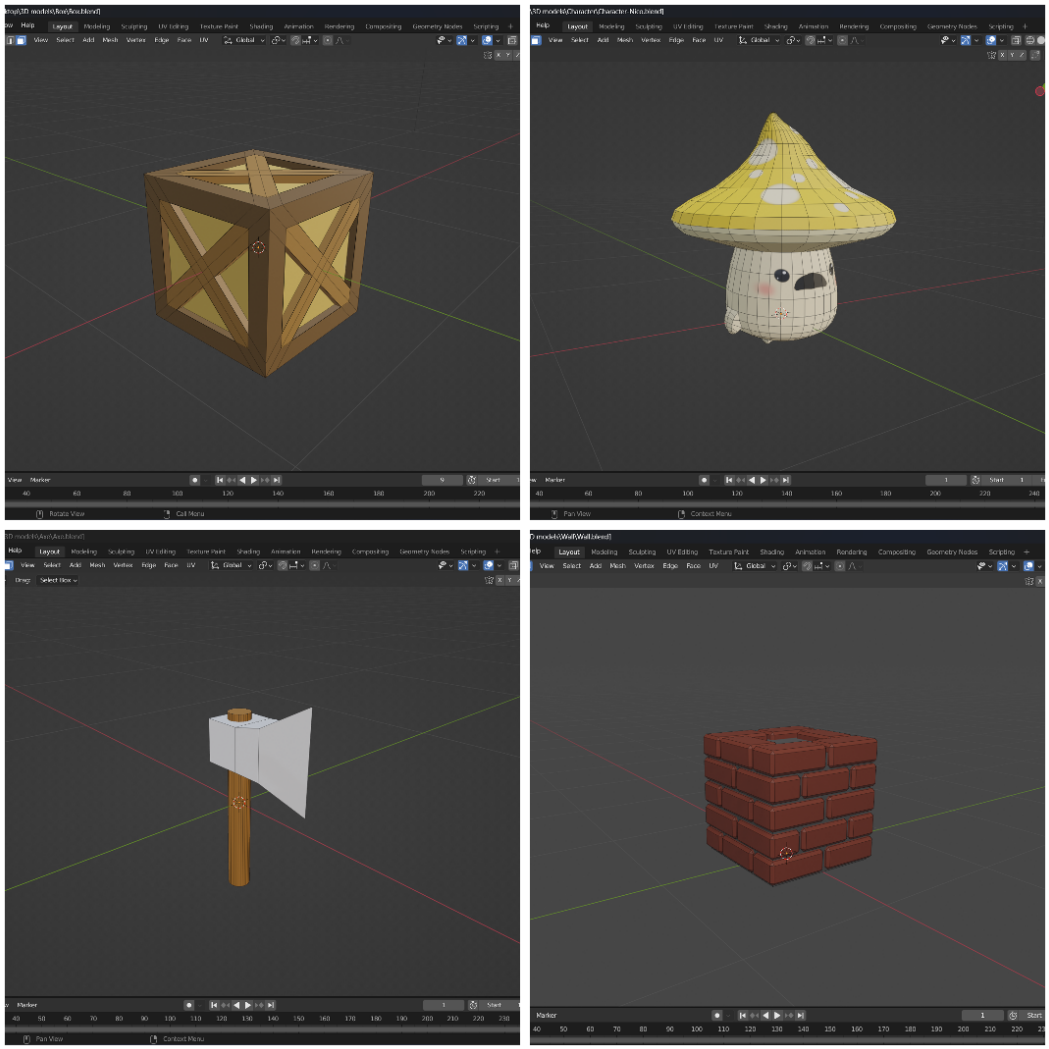
4.1.3 Blender

Oblíbeným nástrojem pro třírozměrnou počítačovou grafiku je bezplatný open-source software Blender. Funkcemi programu jsou především modelování, animace, motion capture a renderování [33]. Blender jsem využila k tvorbě 3D modelů postav hráčů a veškerých objektů na herních políčkách, které můžeme vidět na Obrázku 4.1.

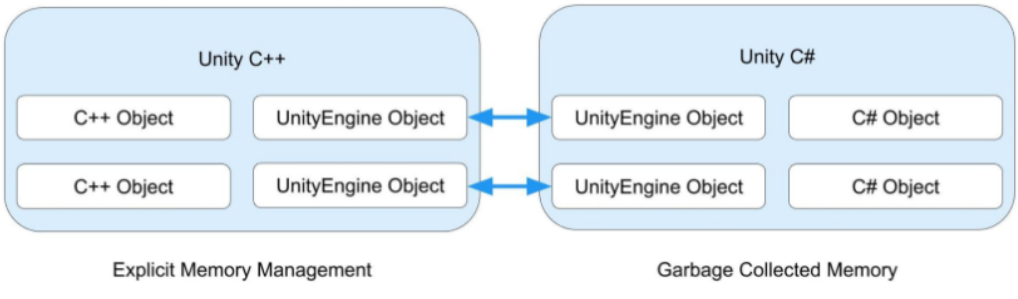
4.1.4 Unity

Pro implementaci herního prostředí byl využit engine a vývojové prostředí Unity. Tento nástroj je však využíván i pro neherní účely. V enginu například architekti mohou snadno prototypovat své nápady, umělci mohou vytvářet interaktivní umělecké instalace nebo jej výzkumníci mohou použít pro vizualizaci dat [34].

Unity Engine je interně postaven na nativním C/C++, avšak obsahuje také C# obal, který uživatelé využívají pro interakci s enginem [35]. Typ objektu



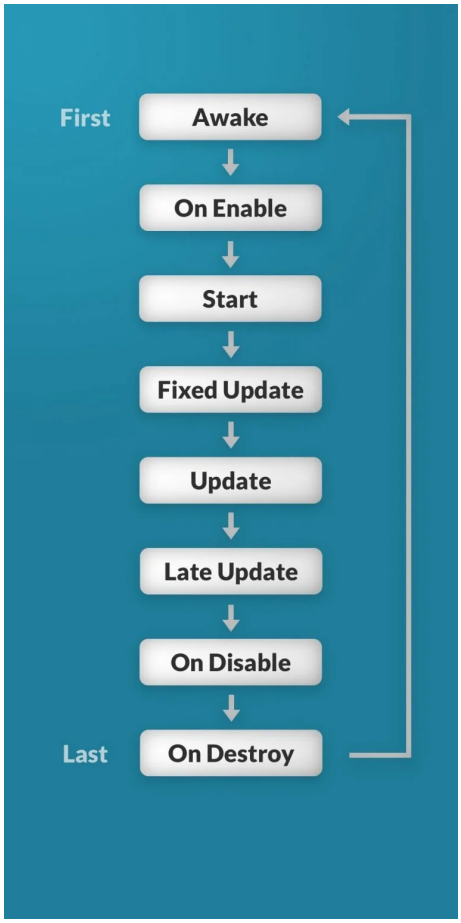
Obrázek 4.1. Blender modely herních entit.



Obrázek 4.2. Vazba mezi C++ a C# objekty v Unity.

`UnityEngine.Object` v jazyce C# je speciální, protože je propojen s odpovídajícím nativním C++ objektem. Tato vazba je znázorněna na Obrázku 4.2. Při použití například komponenty `Camera`, reprezentující pohled hráče, pak Unity ukládá stav objektu do odpovídajícího nativního objektu v C++, nikoliv do samotného objektu v jazyce C# [36].

Unity umožňuje ovládat svůj engine pomocí C# skriptů, které dědí z třídy `MonoBehaviour` [37]. Tato třída slouží jako základ pro každý skript v Unity a poskytuje metody pro řízení životního cyklu objekt [38]. Veškeré tyto akce vidíme na Obrázku 4.3. Mezi nejvýznamnější z nich patří metody `Start` a `Update`. Metoda `Start` je volána těsně před začátkem prvního snímku, zatímco metoda `Update` je volána s každým snímkem.

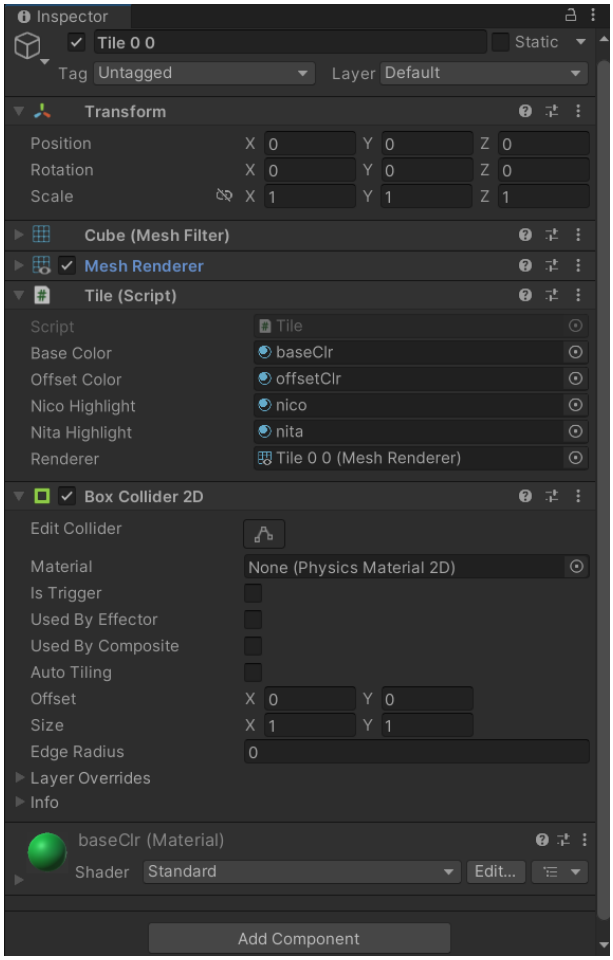


Obrázek 4.3. Cyklus funkcí třídy MonoBehaviour [39].

`MonoBehaviour` skripty jsou serializovány pomocí Unity a tato data jsou následně využita k vytvoření uživatelského rozhraní pro instanci naší třídy [37], které se při poklknutí na instanci zobrazí v okénku Inspectoru v rámci Editoru. Toto zobrazení, viditelné na Obrázku 4.4, nám umožňuje snadno referencovat informace ze scény do atributů našeho objektu pouhým přetažením myši a zároveň snadno přidávat, odebírat či modifikovat jeho jednotlivé komponenty.

Nejnovější verze Unity využívají datově orientovanou architekturu, která byla navržena s cílem zvýšit výkon a škálovatelnost. Tato architektura je často označována jako model Entity-Component-System (ECS) [40]. V rámci modelu ECS v Unity jsou herní objekty rozděleny do tří hlavních kategorií:

- **Entity** - tyto objekty představují základní stavební kameny v Unity a reprezentují základní jednotky hry, jako jsou postavy, předměty nebo překážky
- **Komponenty** - uchovávají data a chování spojené s entitami. Může se jednat například o definici polohy entity, vlastnosti vykreslování nebo chování entity



Obrázek 4.4. Uživatelské rozhraní objektu v Inspectoru.

- **Systémy** - prvky jsou zodpovědné za zpracování a aktualizaci entit a jejich komponent. Poskytují logiku interakce mezi komponentami a jejich chování v průběhu hry [40].

Scéna

Celá hra nebo její části jsou obsahem takzvané scény v Unity, ve které pracujeme s jednotlivými objekty (`GameObjects`) a tvoříme tak herní prostředí. Jednodušší hry většinou sestávají z jedné scény, zatímco pro složitější hry může například jednu úroveň představovat jedna scéna, z nichž každá má své vlastní prostředí, postavy, překážky, dekoraci a uživatelské rozhraní [41].

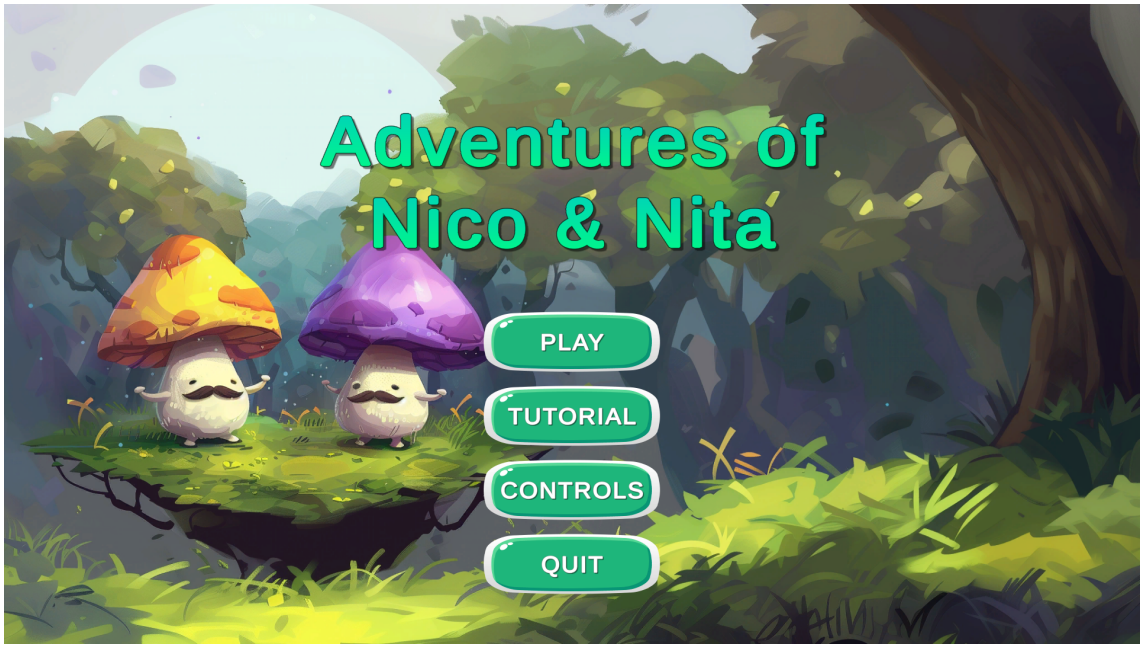
Při vytvoření nového projektu nám Unity otevře ukázkovou scénu, která vždy implicitně obsahuje kameru, `Directional Light` a `EventSystem` [41]. Kamera je zařízení, jehož prostřednictvím hráč pozoruje svět hry a `Directional Light` je světlo, které vyzařuje světlo pouze jedním směrem [42]. Nakonec `EventSystem` je zodpovědný za zpracování a obsluhu událostí v dané Unity scéně. Scéna by měla obsahovat vždy pouze jeden `EventSystem` [43]. Celkově se naše hra skládá ze dvou scén - menu a hry samotné. Vzhled tlačítek v obou scénách byl převzat z balíčku z Unity Asset Store [44].

Menu scéna

Scéna `Menu` v sobě zahrnuje dva herní objekty - `Setup` a `Canvas`. Objekt `Setup` jsem vytvořila pro účely zpřehlednění struktury scény. Tento herní ob-

jekt je instancí `Empty GameObject` a slouží pro zaobalení základních objektů scény - `Main Camera`, `Directional Light` a `EventSystem`.

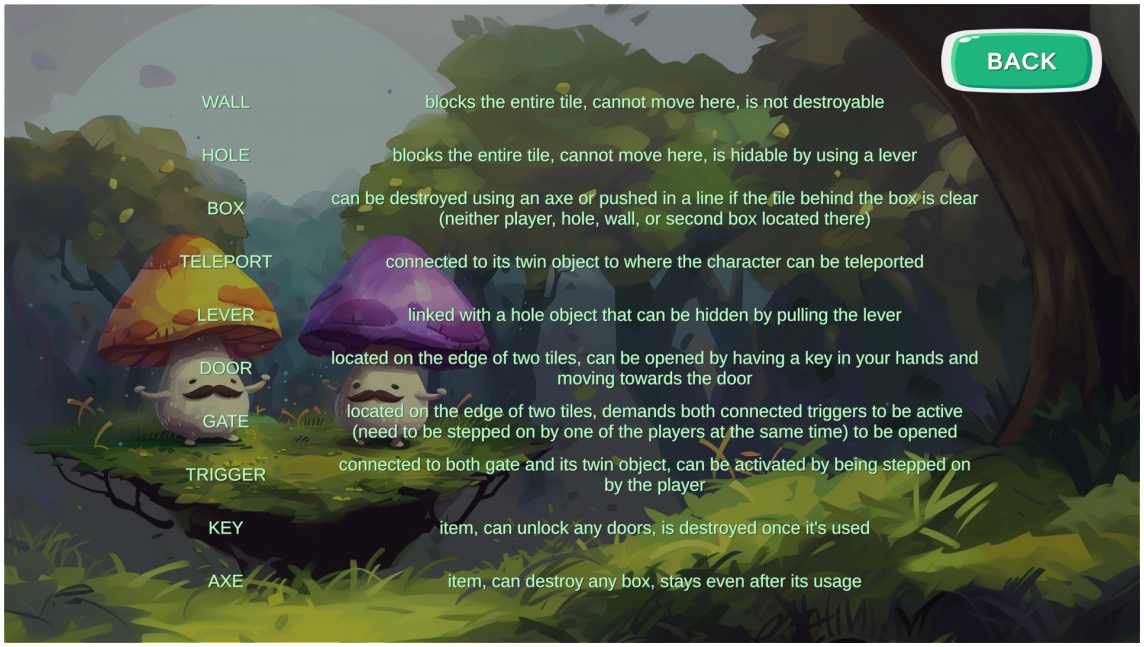
Druhý objekt `Canvas` je oblast, ve které by se měly nacházet všechny prvky uživatelského rozhraní. Jedná se o `GameObject` s komponentou `Canvas` a všechny prvky uživatelského rozhraní musí být jeho potomky [45].



Obrázek 4.5. Obrazovka `Main Menu`.

Součástí `Canvas` je obrázek pro vyplnění jeho oblasti, jenž byl vygenerován pomocí AI nástroje Midjourney [46]. Dále `Canvas` obsahuje tlačítko zpět a následující čtyři meny:

- **Main menu** - počátení vyobrazené menu, snímek tohoto menu zachycuje Obrázek 4.5, obsahuje tlačítka
  - `Play` - zaktivuje viditelnost `Selection Menu` a současně zneviditelní obsah Main Menu,
  - `Tutorial` - zaktivuje viditelnost `Tutorial Menu` a současně zneviditelní obsah `Main Menu`,
  - `Controls` - zaktivuje viditelnost `Controls Menu` a současně zneviditelní obsah `Main Menu`,
  - `Quit` - ukončí hru,
- **Selection menu** - v rámci stanovených omezení na počet elementů na řádek daných svou komponentou `Grid Layout` dynamicky vykreslí tlačítka reprezentující jednotlivé úrovně hry, menu lze vidět na Obrázku 4.8,
- **Tutorial menu** - Obrázek 4.6 poukazuje na dynamicky vykreslený obsah tohoto menu, jenž poskytne výpis všech objektů, které se mohou nacházet na herních políčkách, spolu s kratkým popiskem o jejich chování,
- **Controls menu** - poskytuje dynamicky vygenerovaný seznam akcí, které hráč může se svou postavou provést, a k nim přiřazené klávesnice. Jeho podobu vidíme na Obrázku 4.7.



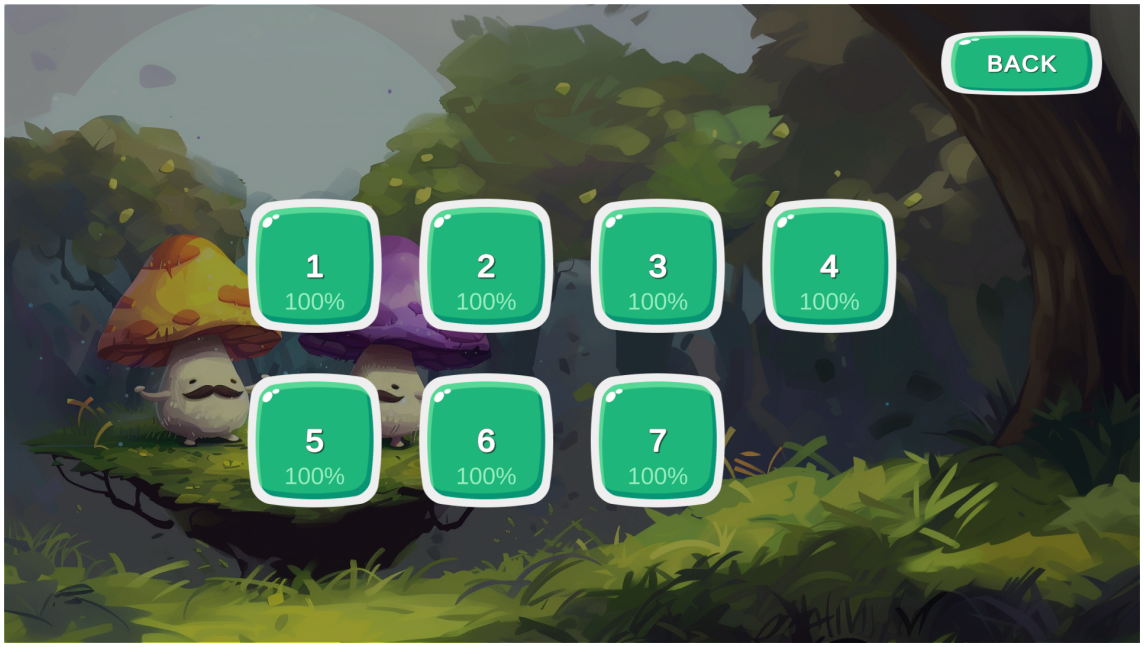
Obrázek 4.6. Obrazovka `Tutorial Menu`.

Obrázek 4.7. Obrazovka `Controls Menu`.

Tlačítko zpět je viditelné ve všech výše zmíněných meny s výjimkou `Main Menu`. Každé tlačítko úrovně v rámci `Selection Menu` má při svém instancování na svou událost `onClick`, která je vyvolána kliknutím hráče na dané tlačítko, navěšenou funkci. Funkce způsobí zápis absolutní cesty JSON souboru příslušné úrovně, jehož obsahem je struktura herního pole úrovně, do statické proměnné `chosenLevel` třídy `StaticData` a nařídí načtení následující scény `Game`.

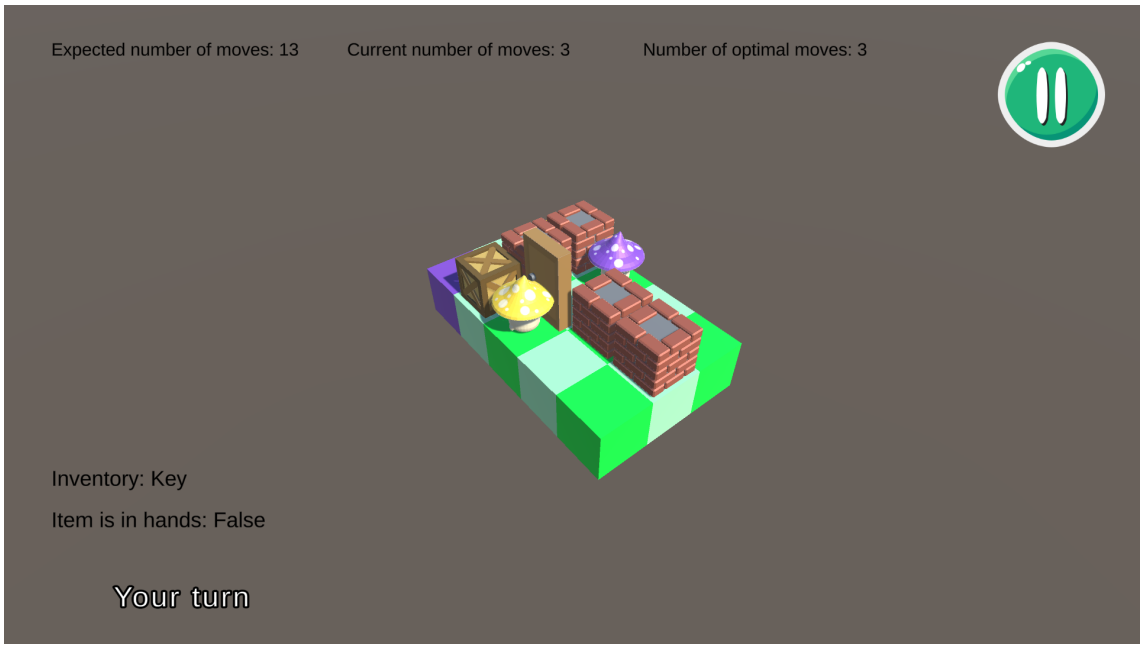
Game scéna

Scéna samotné hry, jejíž podobu zachycuje Obrázek 4.9, obsahuje identický `Setup` a obdobný `Canvas` objekt jako předchozí scéna. Součástí scény jsou navíc herní ob-



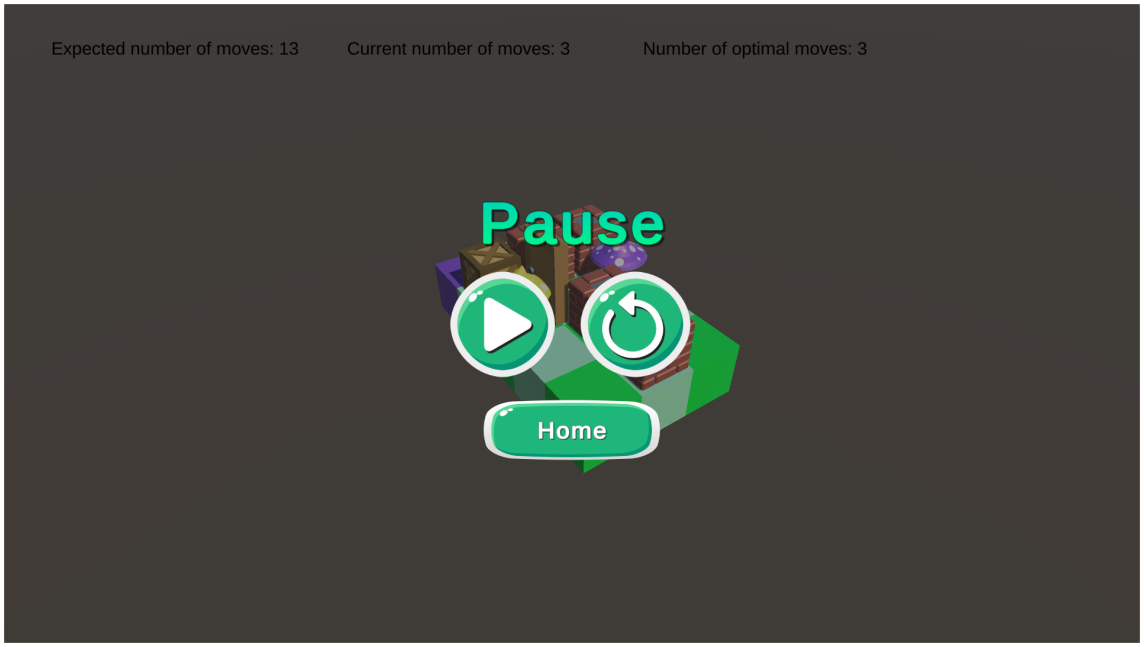
Obrázek 4.8. Obrazovka `Selection Menu`.

jekty `Game Controller`, `Board` a `Planner`, jejichž komponentami jsou stejnojmenné skripty/třídy. Objekt `Canvas` v sobě opět zahrnuje menu objekty:



Obrázek 4.9. Obrazovka `Game Scene`.

- **Pause menu** - vyobrazeno na Obrázku 4.10, zviditelní se při pozastavení hry, obsahuje tlačítka
  - `Resume` - uvede hru zpět do běhu předchozího stavu,
  - `Restart` - provede znovunačtení úrovně,
  - `Home` - navrátí hráče do `Menu Scene`,
- **After menu** - jak lze vidět z Obrázku 4.19, menu zobrazí statistiku tahů hráče a tlačítka `Restart` a `Home` s totožnou funkcionalitou jako výše zmíněná.



Obrázek 4.10. Obrazovka `Pause Menu`.

Posledními objekty spadajícími pod objekt `Canvas` jsou `UIManager` a `InGameUI`. `UIManager` obsahuje stejnojmenný skript, který na podnět událostí v `GameController` aktualizuje hodnoty proměnných ve scéně, či s úspěšným dosažením cíle dané úrovně zviditelní `After Menu`. Taktéž zahrnuje logiku pojenou s aktivací tlačítek. Na druhé straně `InGameUI` pouze agreguje prvky uživatelského rozhraní, jež jsou ve scéně přítomny.

Unity Editor

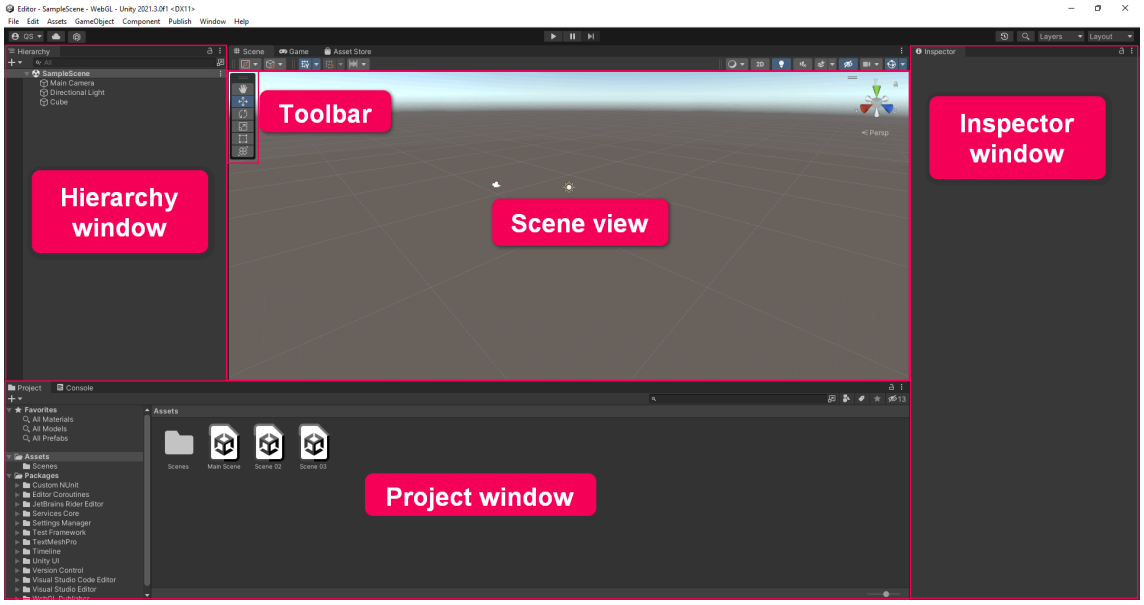
Základní rozhraní Unity Editoru, jehož podobu zachycuje Obrázek 4.11, má pět klíčových oblastí. Uprostřed výchozího rozvržení Unity Editoru je `Scene View`, které funguje jako interaktivní okno do světa, který vytváříme. `Scene View` se používá k manipulaci s objekty scény a jejich prohlížení z různých úhlů. Ve výchozím rozvržení se v této oblasti zobrazuje také `Game View`, které slouží k playtestu [47].

Další část Editoru je takzvané `Hierarchy Window`, skrze které můžeme organizovat všechny objekty ve svém projektu. Tyto objekty se v Unity nazývají `GameObjects`. Pokud do svého projektu přidáme `GameObject` v `Scene View`, bude objekt zalistován v `Hierarchy Window` [47].

`Project Window` je další nedílnou součástí Editoru, ve které můžeme najít všechny položky (`assety`), které jsou k dispozici pro použití v našem projektu. Okno projektu funguje jako průzkumník souborů organizovaný do složek. Chceme-li přidat asset do scény, stačí jej přetáhnout přímo z okna projektu do `Scene View`. `Hierarchy Window` tedy obsahuje všechny `GameObjects` v aktuální scéně, oproti tomu `Project Window` obsahuje všechny prostředky dostupné pro celý náš projekt [47].

`Inspector` je oblast, kde najdeme a nakonfigurujeme podrobné informace o jakémkoliv `GameObject`. Při pokliknutí na `GameObject` v `Scene View` nebo `Hierarchy Window` se nám součásti neboli takzvané komponenty herního objektu, objeví právě v inspektoru. Komponenty popisují vlastnosti a chování `GameObjects` [47].

Poslední výchozí částí Unity Editoru je `Toolbar`. Pomocí tlačítek na tomto panelu nástrojů můžeme měnit úhel pohledu na scénu a také spustit či zastavit režim hry,



Obrázek 4.11. Unity Editor.

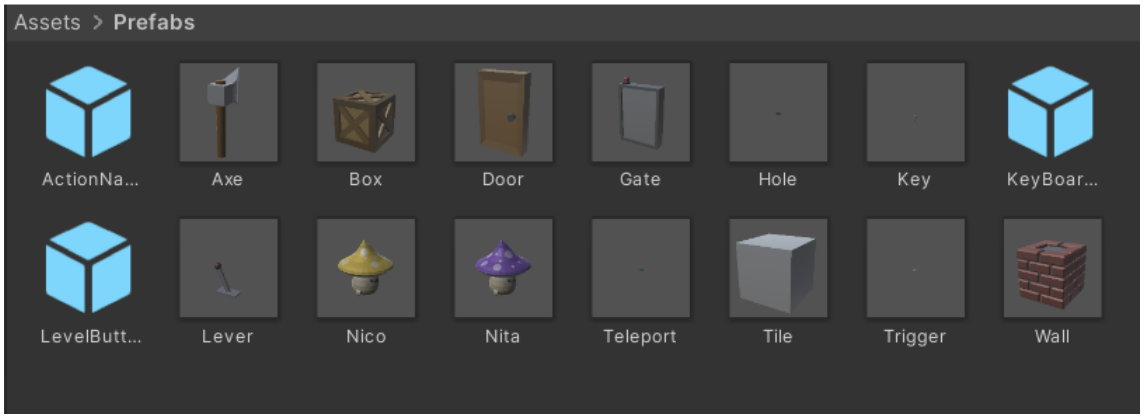
který se v Unity označuje slovem `PlayMode`. Funkce navigace ve scéně jsou umístěny v plovoucím panelu nástrojů v `Scene View`. Tyto funkce nám umožňují přesouvat, otáčet a měnit velikost vybraných objektů [47]. Hra byla vytvořena v Unity Editoru verze 2022.3.4f1.

GameObjects, Prefabs

Jak již bylo popsáno výše, Unity `GameObjects` jsou základní objekty v Unity, které představují postavy, rekvizity a scenérie. Fungují jako kontejnery pro komponenty, které implementují určitou funkcionalitu [45].

Unity `Prefabs` jsou úzce spojeny s `GameObjects`. `Prefab` systém totiž umožňuje vytvářet, konfigurovat a ukládat kompletní `GameObject` se všemi jeho komponentami jako opakovaně použitelný asset. `Prefab` funguje jako šablona, ze které můžeme vytvářet nové prefabrikované instance ve scéně [45].

`Prefabs`, vyobrazené na Obrázku 4.12, jsem využila pro implementaci herních objektů reprezentujících postavy hráčů, herní políčka a jednotlivé objekty, které se mohou na políčkách nacházet. Taktéž prvky uživatelského rozhraní, které najdeme v rámci každého menu, jsou tvořeny prostřednictvím prefabs. Třírozměrné modely, které jsem vytvořila v programu Blender, jsou pak zapouzdřeny v prefabrikátech jednotlivých objektů.

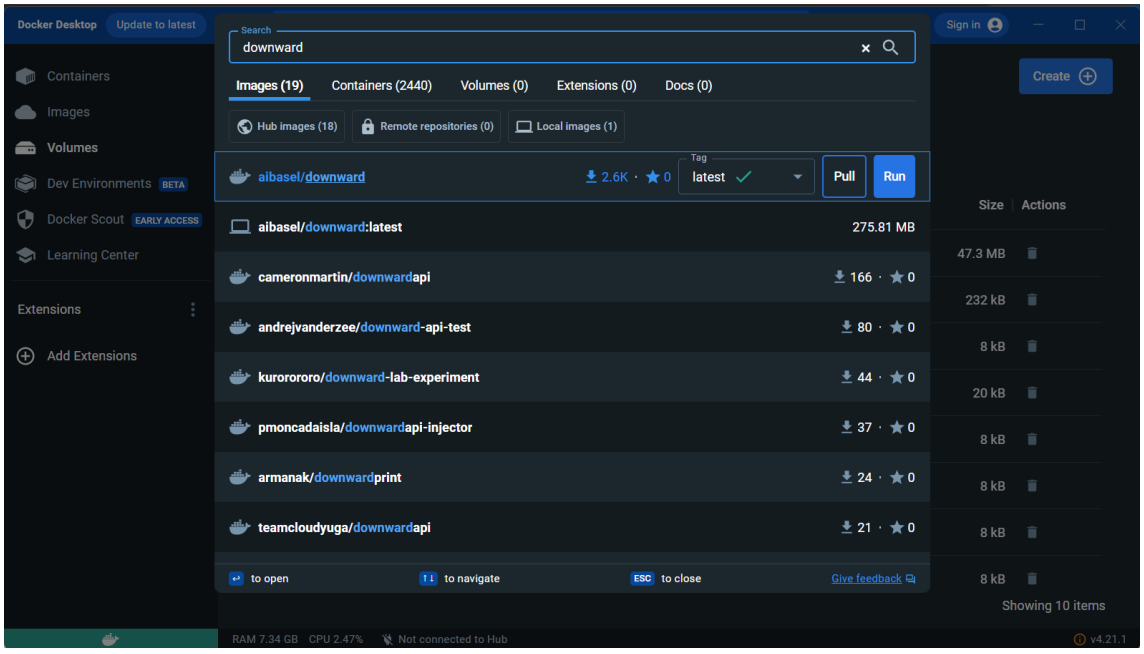


Obrázek 4.12. Prefabs hry.

Integrace plánovače do Unity

Abychom dosáhli napojení plánovače Fast Downward s prostředím Unity, využijeme jeho zmíněného Docker image. Nejprve si tedy nainstalujeme aplikaci Docker Desktop. Po instalaci si v horním panelu aplikace vyhledáme pod klíčovým slovem `downward` odpovídající image a stáhneme jej. Tento postup zachycuje Obrázek 4.13.

Naše hra si dokáže image plánovače stáhnout i sama skrze Docker příkaz, nicméně předběžným stažením se vyhneme časové prodlevě při načítání první úrovně, která je způsobená právě tímto stahováním. S využitím třídy `Process` vytvoříme příkaz, jehož syntax je definován specifikacemi plánovače, pro nalezení cesty v rámci níže vytvořené PDDL domény a PDDL problému. Z obdržené cesty si vyselektujeme jednotlivé akce a s nimi svázané parametry.



Obrázek 4.13. Nasazení Docker image.

4.2 PDDL model hry

Jak již bylo uvedeno v teoretickém úvodu této práce, jazyk PDDL představuje nástroj v oblasti klasického plánování, jež nám slouží pro formulaci plánovacích domén a problémů. Konkrétně v kontextu této práce je PDDL doména využívána k detailní charakterizaci prostředí a pravidel vyvíjené hry. Naopak PDDL problém představuje její jednotlivé úrovně.

4.2.1 Doména

Pro zadefinování jednotlivých entit, interakce s nimi, jejich vzájemných vazeb a celkových pravidel hry je třeba nejprve vytvořit PDDL doménu. PDDL doména se skládá z následujících částí:

- `define` - název domény
- `:requirements` - požadavky, formalizace skutečnosti, že ne všechny plánovače dokáží zpracovat všechny problémy formulované v notaci PDDL [2]
- `:types` – deklarace typů (tříd) entit včetně jejich hierarchie
- `:predicates` - predikáty
- `:action` - akce
  - `:parameters` - entity, které se účastní akce
  - `:precondition` - predikáty a vztahy mezi nimi, vyjádřeny logickými výrazy a operacemi, které musí být splněny před provedením akce
  - `:effect` - logický výraz definující jaké predikáty se stanou pravdivými po provedení akce

Část domény byla převzata od mé vedoucí, paní inženýrky Urbanovské, která ji vytvořila pro potřeby předmětu Plánování pro umělou inteligenci. Cílem bylo modelovat problém nazvaný Grid-Mario v jazyce PDDL. Modelovaný problém má představovat reprezentaci hry, kterou chceme řešit prostřednictvím klasického plánování. Úkolem bylo vytvořit reprezentaci světa zahrnující pohyb agenta na mřížce se všemi známými překážkami [1].

Tuto doménu jsem následně upravila do podoby tahového multiplayer a rozšířila o nové objekty a spolu s nimi pojenými akcemi. K implementaci střídání tahů mezi hráči jsem přistoupila deklarací každého tahu jako objektu příslušícímu vždy jednomu z hráčů. Parametrem každé akce je pak tento nový objekt typu `turn`. Uvedený přístup k implementaci tahového aspektu hry vyžaduje deklaraci každé akce ve dvou variantách: pro hráče a pro umělou inteligenci. Každá akce je tak v doméně duplikována s jediným rozdílem, a to se vstupním a výstupním objektem reprezentující aktuální tah (predikát `onTurn`). Všechny akce určené pro lidského hráče na svém začátku vyžadují pravdivost o svém přiřazeném `turn` objektu, tedy aby byly splněny dvě podmínky:

- **(on-turn tp)** - na řadě je lidský hráč
- **(agent-turn nico tp)** - objekt `tp` se váže na liského hráče (postavu Nico)

kde `tp` je instancí `p-turn`. Efektem akcí lidského hráče jsou vždy podmínky `(not (on-turn tp))` a `(on-turn tai)`. Obdobně tah umělé inteligence vyžaduje splnění podmínek:

- **(on-turn tai)** - na řadě je umělá inteligence
- **(agent-turn nita tai)** - objekt `tai` se váže na umělou inteligenci (postavu Nita)

kde `tai` je instancí typu `ai-turn`, a součástí výsledku provedení akce jsou podmínky `(not (on-turn tai))` a `(on-turn tp)`. S každým krokem tak dojde k simulaci výměny tahů v rámci procesu plánování.

Pokud jde o rozšíření objektů, byla entita páčky upravena tak, aby umožňovala obousměrnou funkcionalitu. První tahnutí za páčku odstraní příslušnou díru, avšak nově je možné opětovně táhnout za páčku a navrátit tak díru na původní místo. Dále jsem tuto entitu převedla z typu do pouhého predikátu, jelikož svými vlastnostmi na základě mého rozřazení objektů spadá pod nepohyblivé entity, tedy lze ji identifikovat skrze dané herní políčko. Specifikace nově přidaných objektů, mezi které se řadí nášlapné desky, brány a sekery, je popsána v následující sekci Typy.

Požadavky

Ještě před implementací jednotlivých entit, predikátů a akcí je třeba plánovači stanovit rozšíření PDDL jazyka, kterých bude doména využívat. Výčtem se jedná o následující prvky:

- `:typing` - umožňuje použití typování pro objekty, typování je podobné třídám a podtřídám v objektově orientovaném programování
- `:negative-preconditions` - umožňuje použití `not` v předpokladech
- `:conditional-effects` - umožňuje použití `when` pro vyjádření efektů akce, tedy pokud je něco pravda, aplikuj taktéž tento efekt [48].

Typy

Jak lze vidět na Obrázku 4.14, většina entit je zastřešena typem `object`. Pro vyjádření existence zbývajících (nepohyblivých) entit, které se můžou nacházet na herních políčkách, postačují predikáty. Třídu `object` lze dále kategorizovat do dvou podtypů. První skupina `item` definuje předměty, které hráč může vzít a následně tak použít pro provedení odpovídající akce. Druhý podtyp `turn` slouží pro umělou realizaci tahové hry a určuje tak, který hráč je v konkrétním stavu na tahu.

V oblasti umělé inteligence je hráč referován jako agent, a proto toto názvosloví bylo zvoleno pro pojmenování jeho entity. Agent je obecně entita, která vnímá své prostředí a jedná podle něj [11].

```
(:types
    tile agent box item turn - object
    key axe - item
    p-turn ai-turn - turn
)
```

Obrázek 4.14. Deklarace typů v rámci PDDL domény.

V rámci rozšíření byla doména doplněna o tři nové objekty – brány, nášlapné desky (`trigger`) a sekery.

S ohledem na omezení PDDL, které nezahrnuje implicitní podporu pro zadefinování kooperativního aspektu hry, jež je klíčový pro tuto práci, bylo nezbytné tento prvek do hry zasadit uměle. Tohoto jsem docílila prostřednictvím nášlapných desek, které se aktivují pouhým pohybem hráče na takovouto desku. Avšak aby tato akce měla reálný dopad na změnu stavu - způsobila otevření příslušné brány, obě nášlapné desky musí být aktivovány současně, což vyžaduje synchronizovanou spolupráci mezi hráči.

Další již zmíněná entita brána neboli `gate`, je vždy spojena s dvojící nášlapných desek. Svým umístěním je analogická entitě `door`, tedy nachází se vždy na hraně dvou herních políček, nicméně jak bylo řečeno výše, pro její otevření je nezbytné aktivovat obě nášlapné desky v jeden okamžik.

V souvislosti s bližší specifikací funkcionalit přidaných entit, poslední nová entita sekera (`axe`), ačkoliv obdobná ostatním předmětům, přináší odlišné možnosti interakce se stejnými objekty a narozdíl od klíče po svém prvním použití nezaniká, ale zůstává uložena v inventáři hráče.

Predikáty

Predikáty v rámci domény slouží jako vstupní podmínky pro realizaci akcí a jako důsledky po jejich uskutečnění. Jsou nedílnou součástí deterministického prostředí, ve kterém je hra situována.

V sekci :predicates nalezneme tyto predikáty:

- **(at ?o - object ?t - tile)** - zda se obejkt o nachází na políčku t
- **(hole ?t - tile)** - vyjádření existence díry a jejího umístění na políčku t
- **(wall ?t - tile)** - vyjádření existence zdi a jejího umístění na políčku t
- **(trigger ?t - tile)** - vyjádření existence nášlapné desky a jejího umístění na políčku t
- **(door ?t1 - tile ?t2 - tile)** - vyjádření existence dveří a jejich umístění na rozhraní políček $t1$ a $t2$
- **(gate ?t1 - tile ?t2 - tile)** - vyjádření existence brány a jejího umístění na rozhraní políček $t1$ a $t2$
- **(trigger-pair ?t1 - tile ?t2 - tile)** - provázání dvou nášlapných desek, které se nachází na políčkách $t1$ a $t2$
- **(tele-pair ?t1 - tile ?t2 - tile)** - provázání dvou teleportů, které se nachází na políčkách $t1$ a $t2$
- **(lever-hole-pair ?t1 - tile ?t2)** - provázání páčky na políčku $t1$ a díry na $t2$
- **(trigger-gate-paired ?t1 - tile ?t2 - tile ?t3 - tile)** - provázání nášlapné desky na políčku $t1$ a brány na rozhraní políček $t2$ a $t3$
- **(connected ?t1 - tile ?t2 - tile)** - hrana těchto dvou políček je průchozí
- **(line ?t1 - tile ?t2 - tile)** - uvádí rozsah (trojici) políček, které jsou v jedné rovině
- **(active ?t - tile)** - zda je nášlapná deska na daném políčku aktivní
- **(clear ?t - tile)** - zda se agent muže přemístit na políčko t
- **(hasItem ?a - agent ?i - item)** - zda agent vlastní předmět i
- **(fullPockets ?a - agent)** - zda je inventář agenta plný (již obsahuje jeden předmět)
- **(onTurn ?t - turn)** - který hráč je na tahu
- **(agentTurn ?a - agent ?t - turn)** - se kterým agentem je daný tah spjatý

Pro implementaci možnosti tlačení boxu je definován predikát `line`. Determinantem, zda se agent může přesunout na konkrétní políčko, je predikát `clear`. Políčko je považováno za `clear`, pokud na něm nejsou přítomny objekty typu `wall`, `hole`, druhý `agent` nebo `box`, který nelze daným směrem posunout. Dále platí, že dvě políčka nejsou `connected` jestliže se na jejich společné hraně nachází `door` nebo `gate` a že hráč může vlastnit maximálně jeden předmět (`axe`/`key`).

Akce

Poslední segment domény specifikuje seznam akcí, které může hráč v průběhu svého tahu aplikovat. Každá tato akce je charakterizována parametry, předpoklady v podobě predikátů, jež musí být splněny pro možnost vykonání dané akce, a takzvanými efekty, taktéž v podobě predikátů, které se po provedení akce stanou pravdivými.

V závislosti na zvolené strategii implementace tahového aspektu hry bylo třeba implementovat každou akci ve dvou podobách: jednu specifikovanou pro hráče a druhou pro AI, přestože v rámci PDDL domény mezi postavou hráče a postavou umělé inteligence neexistuje rozdíl.

Současně každá akce má tak v rámci svých parametrů tah hráče a tah AI. Tím je zajištěno, že na začátku volání akce je identifikován agent, který je aktuálně na tahu, a po jejím dokončení se predikát ohledně tahu druhého agenta pravdivostně zamění s předchozím, což implikuje výměnu tahů. Jednotlivé akce jsou pak definovány následovně:

- **skip(AI)** - k předejití uváznutí ve stavu, ve kterém nelze aplikovat žádnou akcí či volání přebytečných akcí pro pouhou synchronizaci aktivit
- **moveAgent(AI)** - pohyb hráče z jednoho políčka na druhé
- **moveAgentOnTrigger(AI)** - obdobné předchozí akci ale automaticky aktivuje danou nášlapnou desku, zároveň odstraní příslušnou bránu jestliže je i druhá nášlapná deska z dvojice aktivní
- **pushBox(AI)** - posunutí boxu v rámci přímky která je stanovena pohybem hráče
- **destroyBox(AI)** - odstranění boxu pomocí sekery
- **pullLever(AI)** - odstranění hole
- **unpullLever(AI)** - navrácení hole
- **unlockDoor(AI)** - otevření dveří pomocí klíče
- **useTeleport(AI)** - přemístění se z prvního teleportu na druhý
- **pickupItem(AI)** - zvednutí předmětu a jeho následné uložení do inventáře
- **putdownItem(AI)** - vyprázdnění inventáře
- **swapItems(AI)** - obdobné pickupItem, avšak původní předmět z inventáře se zamění s tím, jenž je umístěn na daném políčku, původně vlastněný předmět zůstane ležet na tomto políčku

4.3 Herní prostředí

V samotném prostředí Unity bylo stěžejní zvolit přístup k reprezentaci objektů na políčkách a konfiguračních souborů jednotlivých úrovní. Dále bylo nutné implementovat logiku akcí, které odpovídají výše uvedené PDDL doméně, a mechanismus analýzy pohybu lidského hráče.

4.3.1 Úrovně

Každá úroveň je uložena ve formátu JSON [49] souboru, který je následně pomocí utility `UnityEngine.JsonUtility` deserializován do podoby příslušné třídy. Tyto serializovatelné třídy jsou deklarovány ve skriptu `Board` a jejich struktura je vyobrazena na Obrázku 4.15 a Obrázku 4.16. Instance `Board` pak následně v rámci `BuildLevelState` stavu na základě získaných dat z JSON souboru vykreslí a nainicializuje jednolivá políčka, jejich objekty a také samotné hráče.

4.3.2 TileObjects

Třída `TileObject` reprezentuje všechny objekty, které se mohou nacházet na herních políčkách (vyjma postav hráčů). Jak již bylo navrženo v rámci diagramu tříd, tato třída je abstraktní a odvozené třídy dále rozšiřují konkrétní objekty o další atributy.

Pro zapouzdření `Prefabs`, které v sobě zahrnují 3D model z Blender a skript odpovídající danému typu objektu na políčku, jsem využila `ScriptableObjects`. Unity `ScriptableObject` je datový kontejner, který umožňuje ukládat rozsáhlá data nezávisle na instancích tříd. Jedním z hlavních využití `ScriptableObjects` je minimalizace

```
[Serializable]
21 references
public class LevelData
{
    22 references
    public List<TileData> tiles;
    16 references
    public List<PlayerData> players;
    21 references
    public string goals;
    1 reference
    public int width;
    1 reference
    public int height;
}

[Serializable]
81 references
public class TileData
{
    67 references
    public string id;
    69 references
    public CoordinatesData coords;
    67 references
    public bool isClear;
    67 references
    public List<TileObjectData> tileObjects;
    1 reference
    public string goalTile; // id (name) of the player whose goal this tile is
}

[Serializable]
97 references
public class TileObjectData
{
    40 references
    public string id;
    33 references
    public TileObjectType type;
    24 references
    public CoordinatesData connectedTile;   // for wall and box not present
    7 references
    public string gateId; // for triggers, for other tileObjects not present
    10 references
    public CoordinatesData gateCoords; // for triggers, for other tileObjects not present
}
```

Obrázek 4.15. První část struktury JSON dat.

paměťové náročnosti projektu tím, že se vyhneme duplikaci dat, jelikož vytvoříme znovupoužitelné a upravitelné datové objekty.

Tato metoda je užitečná, zejména pokud má náš projekt prefabrikáty uchovávající neměnná data v připojených `MonoBehaviour` skriptech [50]. V takovém scénáři, při vytvoření instance `Prefab`, obyčejně každá instance získá svou vlastní kopii dat. Avšak namísto opakovaného ukládání dat můžeme využít `ScriptableObject` a poté k datům přistupovat pomocí odkazu ze všech prefabrikátů. To znamená, že v paměti je pouze jedna kopie dat [50]. Jeden z těchto `ScriptableObject` našeho `TileObject` (`TileObjectSO`) je zobrazen na Obrázku 4.17.

TileObjectsDatabase

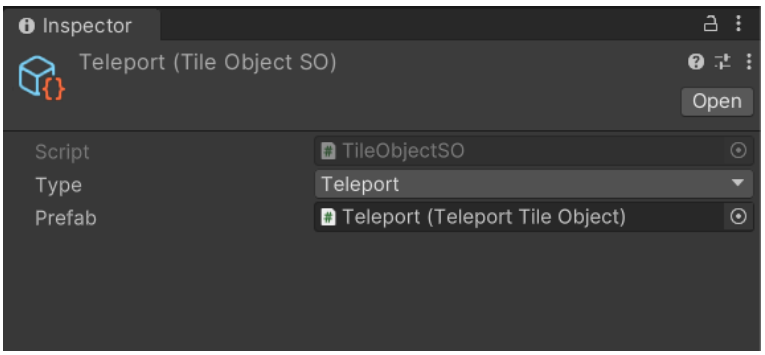
Za účelem uchování výše zmíněných `TileObjectSO` jsem vytvořila třídu napodobující databázi - `TileObjectsDatabase`. Tato databáze similuje strukturu slovníku, kde typ objektu (`Enumeration type`) je klíčem a `Prefab` objektu je hodnotou. Tyto páry jsou

```
[Serializable]
37 references
public class PlayerData
{
    22 references
    public string id;
    23 references
    public CoordinatesData coords;
    24 references
    public PlayerType type;
    22 references
    public string itemId;    // id of item in inventory
    22 references
    public TileObjectType itemType; // type of item in inventory
}

[Serializable]
99+ references
public class CoordinatesData
{
    99+ references
    public int x;
    99+ references
    public int y;
}

29 references
public enum PlayerType
{
    7 references
    Computer,
    19 references
    Human
}
```

Obrázek 4.16. Druhá část struktury JSON dat.



Obrázek 4.17. Konstrukce objektu `Teleport` pomocí ScriptableObject.

drženy prostřednictvím `TileObjectSO`. Databáze `TileObjectsDatabase` navíc dědí ze třídy `Singleton`, tudíž prefabrikáty dílčích objektů poskytuje kdekoli v kódu.

4.3.3 Akce

Pokud jde o realizaci akcí v prostředí Unity, akce jsou kategorizovány do čtyř skupin na základě kláves, které je aktivují. První skupina zahrnuje pouze jednu akci, a to `skip(AI)`, aktivovanou klávesou `Q`.

Do druhé skupiny se řadí akce spojené s pohybem agenta, konkrétně akce moveAgent(AI), moveAgentOnTrigger(AI), pushBox(AI), destroyBox(AI) a unlockDoor(AI). Akce `unlockDoor(AI)` patří do této skupiny kvůli potřebě identifikace konkrétní instance dveří v situacích, kdy se na jednom políčku nachází více objektů tohoto typu.

K identifikaci instance dvěří a rozlišení mezi akcemi pushBox(AI) a destroyBox(AI) byla inkorporována funkce `PullItemOutOfPockets`. Tato funkce se volá klávesou `E` a umístí předmět z inventáře hráče do jeho rukou, jestliže se zde nějaký předmět nachází. Následně tak hráč muže otevřít dveře či zničit krabici. Opětovaným stiskem klávesy `E` může případně předmět z rukou navrátit do svého inventáře. Tato interakce není považována za samostatný tah (je vyhodnocena jako null), aby byla zachována konzistence mezi možnostmi pohybu lidského hráče a umělé inteligence. Pro zachování intuitivního ovládání hry byly pro akce této skupiny použity standardně klávesy WASD a šipky.

Pod třetí podskupinu patří akce pullLever(AI), unpullLever(AI) a useTeleport(AI), které se provedou stiskem klávesy `Space`. Tyto akce se vztahují na páčku a teleport, které se od ostatních objektů liší tím, že jsou nepohyblivé (oproti krabici a předmětům), rozprostírají se na políčku (ne na hraně jako dveře a brána) a současně s nimi hráč může interagovat (narozdíl od díry a zdi), jestliže se nachází na stejném políčku. V každé úrovni je zajištěno, že na jednom políčku je pouze buď páčka nebo teleport. Zároveň mohou oba objekty své políčko sdílet s krabicí a předměty.

Poslední čtvrtá skupina akcí zahrnuje výčtem pickupItem(AI), putdownItem(AI) a swapItems(AI). Tyto interakce s předměty samotnými jsou zprostředkovány klávesou `I`. Na základě kontroly aktuálního stavu inventáře agenta a políčka, na kterém je hráč umístěn, se dále rozliší, o jakou akci z těchto tří se jedná podle tabulky na Obrázku 4.18.

	na políčku je předmět	inventář je plný	
null	0	0	
putdownItem	0	1	
pickupItem	1	0	
swapItems	1	1	

Obrázek 4.18. Pravdivostní tabulka akcí s předměty.

Pokud tedy je políčko s hráčem prázdné (nenalezneme zde jiný předmět) a zároveň inventář hráče obsahuje předmět, zaktivuje se akce `putdownItem`. Naopak, jestliže políčko není prázdné a inventář je opět plný, klávesa `I` se vyhodnotí jako akce `swapItems`. Pokud políčko je obsazeno dalším předmětem a hráčův inventář je prázdný, zavolá se akce `pickupItem`. Nakonec pokud neplatí ani jedno z předchozích tvrzení, klávesa se vyhodnotí jako neplatná akce.

Všechny neplatné akce (null) se počítají jako neúspěšný tah a hráč je tak vyzván k opětovanému pokusu o tah tak dlouho, dokud neprovede validní akci se svou postavou.

4.3.4 Analýza pohybu

Proces vyhodnocení optimálnosti kroků hráčů probíhá ve stavu `AnalysisState`. Zde na svém vstupu tento stav získá akci zahranou hráčem od předchozího stavu

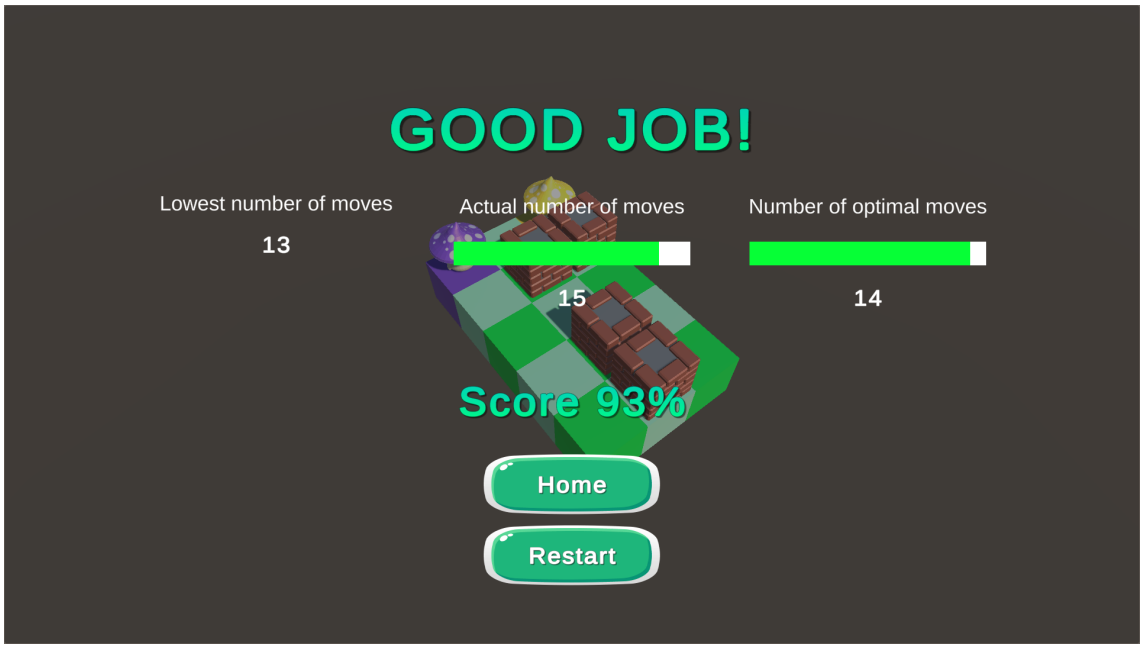
`PlayerMoveState`. Na svém začátku stav `AnalysisState` zkontroluje, zda pohybem hráče došlo ke splnění cílových podmínek a případně hru uvede do finálního stavu `LevelDoneState`. V opačném případě ověří, jestli tah byl optimální porovnáním tahu hráče s očekávaným tahem, jenž je uložen ve třídě `Planner`. Pro umělou inteligenci se samozřejmě očekávaná a skutečná akce vždy rovnají.

V případě neoptimálního tahu (lidského) hráče nejprve musíme prověřit, zda jeho tah není alternativním optimálním plánem, vzhledem k tomu, že ne všechny úrovně jsou navržené tak, aby existoval pouze jediný optimální plán. Pro tuto kontrolu je třeba vytvořit nový PDDL problém, který reflektuje stav hry vzniklý pohybem hráče. Nový plán nám poskytne třída `Planner`, která jej získá příkazem na Docker image plánovače. Následně můžeme porovnat délku nového plánu s naši proměnnou `movesLeft`, jejíž hodnota udává počet zbývajících kroků pro dosažení cíle následováním aktuálního plánu. Jestliže se hodnoty rovnají, hráčův tah vyhodnotíme jako optimální. V každém případě nově vzniklý plán se stane aktuálním a na řadě je druhý hráč.

4.3.5 Hráčovo skóre

Dosažený výsledek neboli skóre lidského hráče, se spočítá kombinací číselných proměnných, které jsou uloženy v data kontejneru `Analysis`. Jedná se o proměnné:

- `LowestNumberOfMoves` - počet kroků/akcí optimálního plánu,
- `ActualNumberOfMoves` - počet (dosavadních) tahů lidského hráče,
- `OptimalMovesCounter` - počet (dosavadních) optimálních tahů lidského hráče.



Obrázek 4.19. Obrazovka `After Menu`.

Hráči tedy poskytujeme dva ukazatele - jak blízko/daleko byl svým počtem tahů od optimálního průchodu úrovní a kolik z jeho kroků bylo optimálních. Tuto statistiku má hráč k dispozici při dokončení úrovně v rámci `After Menu` a svůj nejlepší výkon v jednotlivých úrovních má pak zobrazen při výběrů herní úrovně v `Selection Menu`.

Ukládání skóre jsem naimplementovala pomocí třídy `PlayerPrefs`, která ukládá předvolby hráče mezi herními relacemi. Může ukládat string, float a integer do re-

gistru platformy uživatele. Unity ukládá `PlayerPrefs` do místního registru bez šifrování, tudíž tato třída není vhodná pro citlivá data [51]. Ukládání skóre jsem naimplementovala pomocí třídy `PlayerPrefs`, která ukládá předvolby hráče mezi herními relacemi. Může ukládat string, float a integer do registru platformy uživatele. Unity ukládá `PlayerPrefs` do místního registru bez šifrování, tudíž tato třída není vhodná pro citlivá data [51].

Kapitola 5
Testování

S narůstající složitostí softwaru a rozšiřováním jeho funkcí je stále obtížnější vytvořit program bez závad [52]. V rámci každého systému je tak nezbytné jeho jednotlivé části i celek patřičně testovat jak již při návrhu, implementaci, tak i provozu systému.

Testování, které probíhá na nejnižší úrovni a bývá tak prvním krokem k ověření správného fungování systému, se nazývá testování jednotek neboli unit testing. Ve chvíli, kdy jsou otestovány jednotky systému a jsou nalezeny a opraveny nízkoúrovňové chyby, jsou tyto jednotky integrovány do takzvaných integračních testů. Tento proces inkrementálního testování pokračuje a spojuje dohromady stále více částí softwaru, dokud není celý produkt, nebo alespoň jeho hlavní část, otestována [52].

Za klíčové části naší hry, které je stěžejní otestovat, jsem uvážila konkrétně generování herního pole, provedení jednotlivých akcí, které jsou definovány v rámci PDDL domény, a nakonec generování PDDL problému z aktuálního stavu hry. Dále celý běh hry byl po své plné implementaci podroben uživatelskému testování.

5.1 Unity Test Framework

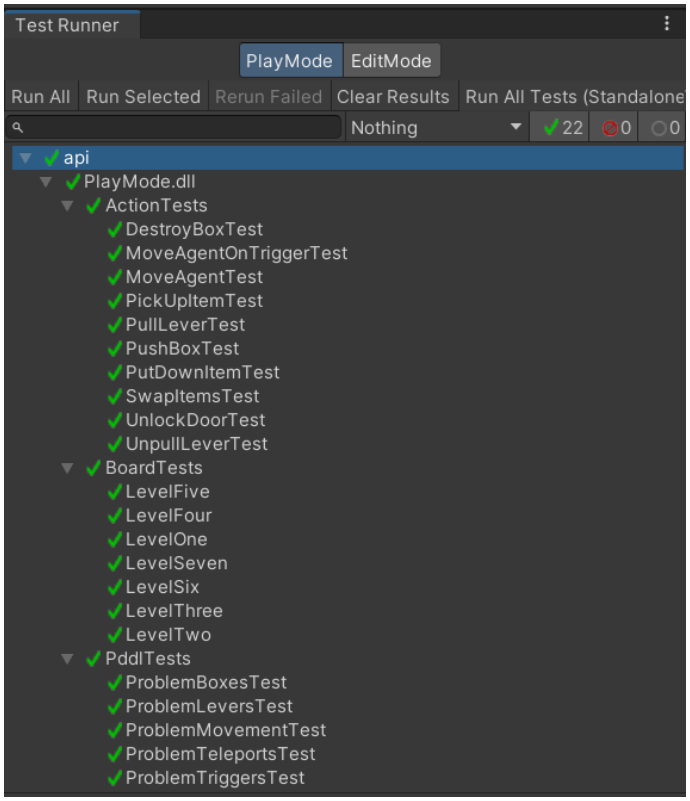
V prostředí Unity probíhá testování prostřednictvím takzvaného Unity Test Framework (UTF), dříve označován jako Unity Test Runner. Framework UTF umožňuje uživatelům testovat svůj kód jak v režimu úprav (`Edit Mode`), tak i režimu přehrávání (`Play Mode`). Zároveň zprostředkovává testovaní na cílových platformách, jako je Standalone, Android, iOS a další. UTF využívá integraci open-source knihovny NUnit, která byla vyvinuta pro účely jednotkového testování, určené všem jazykům `.Net`. UTF aktuálně používá NUnit verze 3.5 [53]. Podobu frameworku uvnitř Unity Editoru můžeme vidět na Obrázku 5.1.

Okénko testovacího frameworku získáme v horním menu Unity Editoru navigací zachycenou Obrázkem 5.2. Jak již bylo zmíněno, v Unity se setkáváme se dvěma typy testů na základě režimu, ve ktrém musí být testy spuštěny pro správné fungování: `Edit Mode` a `Play Mode`. V našem případně byly využity pouze `Play Mode` testy, jelikož objekty, které testujeme (`Board`, `Tile`, `TileObject` a `Player`), dědí ze třídy `Monobehaviour` a fungují tak za běhu programu.

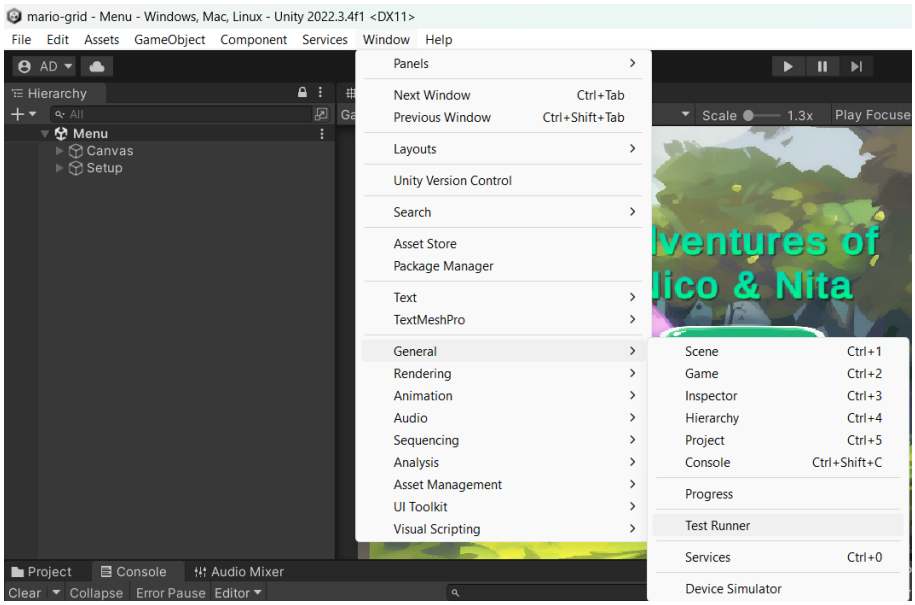
K porovnávání očekávaného a skutečného výsledku využíváme takzvané asserty. `Assert` je pravdivostní výraz, který kontroluje jistou podmínku. Tato třída obsahuje řadu metod, nicméně pro níže uvedené testy jsme si vystačili s metodou `AreEqual`. I tato metoda existuje s řadou přetížení, já jsem využila konkrétně podobu této funkce se dvěmi vstupními parametry třídy `Object`, která testuje, zda jsou zadané objekty stejné, a vyvolá výjimku, pokud se tyto dva objekty neshodují [54].

5.1.1 Generování herního pole

Pro každou úroveň jsem vytvořila jednotkový test, jenž kontroluje správné nastavení jednotlivých políček a jejich objektů korespondující s konfiguračním souborem dané



Obrázek 5.1. Unity Test Framework.



Obrázek 5.2. Navigace k Unity Test Framework.

úrovně. V `Setup` funkci, která je volána před každým jednotkovým testem v naší testovací sadě [55], inicializujeme prostředí získáním JSON konfiguračních souborů dílčích úrovní. Následně funkcí `GenerateBoard`, která na svém vstupu přijímá JSON soubor, testu nejprve zprostředkujeme prefabrikáty herního políčka a jednotlivých objektů a dále skrze asserty ověříme správnost jednotlivých atributů každé vytvořené instance. Tento proces opakujeme pro všechny úrovně.

5.1.2 Akce

Veškeré akce deklarované v PDDL doméně a následně naimplementované v Unity jsem taktéž testovala pomocí jednotkových testů. Pro každou akci jsem vytvořila konfigurační soubor herního pole, který vždy zachycuje pouze potřebné herní objekty včetně jednoho hráče, který akci provede. Obdobně jako předchozí i tato sada testů obsahuje funkci `Setup`, která našemu hernímu poli poskytne prefabrikáty jednotlivých objektů na herních políčkách. V testech samotných, ještě před aplikací akce, definujeme asserty o vlastnostech objektů pro ujištění se o jejich následné změně po zavolání akce. Poté každé akci vytvoříme odpovídající parametry a zavoláme ji. Nakonec opět pomocí assertů zkontrolujeme vlastnosti objektů a hráče, které by provedení akce mělo modifikovat.

5.1.3 PDDL parser

Poslední zásadní částí systému, kterou jsem otestovala, je `PDDLParser`. Úkolem této třídy je aktuální stav hry přesnést do podoby PDDL problému, aby následně tento vytvořený problém mohl být předán plánovacímu systému pro získání nového plánu.

Analogickým způsobem jako testování akcí jsem zkonstruovala jednotkové testy i pro parser. Prvně si tedy v iniciální funkci `Setup` získáme obsah souborů PDDL problémů, které společně pokrývají veškeré predikáty o herním poli a slouží nám jako očekávané výsledky. Dále každému testu poskytneme konfigurační JSON herního pole, který odpovídá příslušnému PDDL problému získanému z předešlého kroku. Postupně tyto jednotlivé PDDL problémy porovnáme s vytvořenými problémy z `PDDLParser`u pomocí assertu.

5.2 Testování použitelnosti

Nedílnou součástí testování systému je také testování s uživateli, kteří budou nakonec náš systém používat. Testování použitelnosti, známé také jako usability testing, se zaměřuje na kontrolu použitelnosti vyvíjeného systému, tedy na to, jak je interakce uživatele se systémem vhodná, funkční a efektivní [52].

Cílem uživatelského testování není pouze odhalit chyby v systému, ale také zjistit jeho uživatelskou přívětivost z perspektivy uživatelů z odlišných věkových skupin a s různými zkušenostmi s ovládáním počítačového zařízení.

Testování se zúčastnilo celkem dvanáct subjektů. Tuto skupinu tvořili muži i ženy ve věkovém rozmezí dvanáct až sedmdesát osm let. Všichni uživatelé měli za úkol absolvovat jednotlivé úrovně hry a během nich upozorňovat na případné nedostatky. Každému uživateli tedy byla předložena hra ve stavu po jejím spuštění a následně se uživatelé již sami navigovali jednotlivými herními meny pro naplnění své úlohy v pozici testerů.

Postava umělé inteligence zmizí ze hry

Problém V páté úrovni při hraní pouze akce `Skip` se postava umělé inteligence postupně přesune na teleport spjatý s políčkem, jež je počáteční pozicí postavy lidského hráče, a po jeho využití se vytratí ze scény.

Řešení V inicializačním JSON souboru pro pátou úroveň byla nalezena pravdivostní chyba v atributu `isClear` políčka s identifikátorem `0x3`, jehož hodnota byla chybně nastavena jako `true`. Jelikož na tomto poli začíná po-

stava lidského hráče, přemístění postavy umělé inteligence pomocí teleportu na toto samé políčko způsobilo, že postava byla zastíněna postavou druhého hráče a budila tak dojem zmizení. Následně logika plánování vyvolala správně error, protože se nepodaří z tohoto stavu najít plán pro dokončení úrovně. Nalezená chyba byla opravena nastavením atributu na očekávánou hodnotu `false`.

Krabici nelze přesunout na políčko, na kterém se nachází teleport / Nelze se přemístit na políčko s teleportem

Problém V páté úrovni na políčko s identifikátorem `0x1` se nelze pohnout.

Řešení Problém byl analogický předchozímu, tedy atribut `isClear` na tomto políčku byl opraven na hodnotu `true`.

Po znovupoužití páčky se hráč ocitá na díře

Problém Ve čtvrté úrovni po opětovaném táhnutí páčky lidským hráčem ve chvíli, kdy postava druhého hráče stojí na políčku, které je pojeno s páčkou, se postava ocitne na díře.

Řešení Tento problém byl vyřešen podmínkou v `HumanStrategy` skriptu, která zajišťuje, že akci `unpull-lever` nelze provést, jestliže se na souvisejícím políčku nachází hráč.

Přesunutím krabice na cílové políčko se hra zasekne

Problém V poslední sedmé úrovni lze krabici po přímce přesunout až na cílové políčko lidského hráče a zamezit tak jakémukoliv možnému dokočení úrovně.

Řešení V původním návrhu sedmé úrovně bylo hráči umožněno si nenávratně zablokovat cestu a způsobit tak error vyvolaný nenalezením řešení plánovačem. Struktura úrovně byla ošetřena předejitím této situace přidáním zdi před cílové políčko.

Není zřejmé, jaký předmět se nachází v mém inventáři

Problém Hráči není jasné, jestli má v inventáři nějaký předmět popřípadě jakého typu předmět je.

Řešení Do scény byl přidán text, který poskytuje informaci o obsahu inventáře.

Není zřejmé, zda se předmět z inventáře přesunul do rukou

Problém Hráči není jasně demonstrováno, zda stisknutí klávesy `E` pro umístění předmětu do rukou či z rukou funguje.

Řešení Do scény byl přidán text, který poskytuje informaci o této skutečnosti prostřednictvím pravdivostní hodnoty.

Nikde není řečeno, že mohu pohybovat s kamerou

Problém Hráč si není vědom, že má možnost pohybovat kamerou pro lepší přehled o rozmístění objektů na herním poli.

Řešení Do `Controls Menu` byl přidán řádek o této funkcionalitě.

V Controls Menu je matoucí popisek k akci pojené s klávesou E

Problém Hráč si není jist s dvojím významem funkcionality klávesy `E` (přesunutí předmětu z inventáře do rukou/z rukou).

Řešení Popisek akce byl pro lepší porozumění opraven podle představ hráče pomocí znaku lomítka.

Kapitola 6
Závěr

Cílem této práce bylo navrhnout a implementovat kooperativní multiplayer tahovou hru v prostředí Unity, kde druhým hráčem je umělá inteligence, řízena automatizovaným plánováním.

Prvním úkolem bylo obeznámení se s problematikou plánování. V této sekci jsme se seznámili s nástrojem PDDL, vyhledávacími algoritmy a heuristickými funkcemi, které tyto algoritmy optimalizují pro potřeby tvorby umělé inteligence.

Druhým úkolem bylo zprovoznění libovolného plánovače a jeho napojení na herní engine Unity, čehož jsme dosáhli využitím volně dostupného plánovače FastDownward, jehož napojení na Unity jsme zprostředkovali voláním příkazů uvnitř Unity na Docker image tohoto plánovače.

Třetím úkolem bylo navrhnout strukturu a mechaniky PDDL plánovací domény vycházející z PDDL domény problému Grid-Mario. Do domény jsme přidali objekty typu nášlapné desky a sekery a s nimi pojené akce `move-agent-on-trigger` a `destroy-box`. Zároveň jsme upravili typ `hole`, aby bylo možné tento objekt navrátit do scény pomocí akce `unpull-lever`. Nakonec jsme doménu upravili do podoby tahového multiplayer pomocí typu `turn` a zdvojení jednotlivých akcí.

Čtvrtým bodem byla implementace mechanik hry a hráče umělé inteligence poháněného klasickým plánováním. V rámci Unity jsme tak vytvořili třídy základních prvků hry – herních políček, herního pole, hráčů a entit umístěných na políčkách.

Následně jsme vytvořili funkce, které jsou svým obsahem analogické jednotlivým akcím definovaným v PDDL doméně. Jelikož u lidského hráče není jasně zřejmé, kterou z akcí by chtěl svou postavou aplikovat, vytvořili jsme pro tyto účely třídu `HumanStrategy`, která nám na základě blízkého okolí postavy hráče a stisknuté klávesy hráčem odvodí, o jakou akci se jedná.

Taktéž jsme vytvořili skript pro převedení aktuálního stavu hry z Unity do struktury PDDL problému, abychom pomocí něj mohli získat z plánovače optimální plán a řídit tak pohyb postavy umělé inteligence. Navíc jsme do hry přidali prvek vyhodnocování pohybu lidského hráče a poskytli výsledek této analýzy na konci úrovně.

Posledním požadavkem práce bylo navrhnout sadu úrovní, na kterých bude možné otestovat všechny herní mechaniky a funkčnost hry. Pro tyto účely jsme naimplementovali konfigurační JSON soubory sedmi úrovní, které postupně srdužují objekty herních políček, a jejich obsah pomocí utility `UnityEngine.JsonUtility` převedli do instancí tříd pro vykreslení v Unity scéně. Pro ověření správnosti rozvržení herního pole dané úrovně, chování akcí a převedení stavu hry do podoby PDDL problému jsme využili kombinaci online Editoru a jednotkových testů uvnitř samotného Unity. Nakonec jsme hru jako celek otestovali s uživateli prostřednictvím testování použitelnosti a jejich zpětnou vazbu a postřehy následně integrovali do naší hry.

Závěrem tak lze konstatovat, že stanovené cíle a požadavky závěrečné práce byly naplněny.

Literatura

[1] Michaela Urbanovská. *PUI Assignment 1-1 PDDL [CourseWare Wiki] — cw.fel.cvut.cz*.
https://cw.fel.cvut.cz/b222/courses/pui/assignments/assignment1-1. 2023. [Accessed 14-05-2024] .

[2] Malik Ghallab, Craig Knoblock, David Wilkins, Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, David Smith, Ying Sun a Daniel Weld. *PDDL - The Planning Domain Definition Language*. 1998.
https://planning.wiki/_citedpapers/pddl1998.pdf.

[3] Malik Ghallab, Dana Nau a Paolo. Traverso. *Automated Planning : Theory and Practice*. Elsevier Science Technology, 2004. ISBN 1-55860-856-7.
http://ebookcentral.proquest.com/lib/cvut/detail.action?docID=333985.

[4] Jaroslav Tišer. *Výroková logika.* 2021.
https://math.fel.cvut.cz/en/people/tiser/PropCalcul.pdf.

[5] Edwin Pednault. *Synthesizing plans that contain actions with context-dependent effects*. 1988.
https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8640.1988.tb00285.x.

[6] Nir Lipovetzky. *Structure and Inference in Classical Planning*. AI Access, 2014. ISBN 978-1312466210.
https://www.researchgate.net/profile/Nir-Lipovetzky/publication/272741496_Structure_and_inference_in_classical_planning/links/5b84ed3392851c1e1236d8fa/Structure-and-inference-in-classical-planning.pdf.

[7] Stuart Russell a Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 2009. ISBN 978-0136042594.
https://cw.fel.cvut.cz/old/_media/courses/a3b33kui/knihy/artificial_intelligence_a_modern_approach_3rd_edition_chapter_10.pdf?cache=nocache.

[8] Héctor Geffner a Blai Bonet. *Classical Planning: Full Information and Deterministic Actions*. 2013.
https://link.springer.com/chapter/10.1007/978-3-031-01564-9_2.

[9] Jiří Demel. *Grafy a jejich aplikace*. 2019.
https://kix.fsv.cvut.cz/~demel/grafy/gr.pdf.

[10] Bernhard Pfahringer a Jochen Renz. *AI 2015: Advances in Artificial Intelligence*. Springer, 2015. ISBN 978-3-319-26349-6.
https://link.springer.com/book/10.1007/978-3-319-26350-2.

[11] Harvard University. *Search - Lecture 0 - CS50's Introduction to Artificial Intelligence with Python 2020 — youtube.com*.

https://www.youtube.com/watch?v=WbzNRTTrX0g&list=PLhQjrBD2T381PopUTYtMSstgk-hsTGkVm&index=2&ab_channel=CS50. 2020. [Accessed 14-05-2024] .

[12] Charisma Tubagus Setyobudhi. *Comparison of A* algorithm and greedy best search in searching fifteen puzzle solution*. 2022. https://journalijisr.com/sites/default/files/issues-pdf/IJISRR-941.pdf.

[13] Blai Bonet a Héctor Geffner. *Planning as heuristic search*. 2001. https://www.cs.toronto.edu/~sheila/2542/s14/A1/bonetgeffner-heusearch-aij01.pdf.

[14] Juan Ramón Rabuñal Dopico, Julian Dorado a Alejandro Pazos. *Encyclopedia of Artificial Intelligence*. 2009. https://theswissbay.ch/pdf/Gentoomen%20Library/Artificial%20Intelligence/ISR.Encyclopedia.Of.Artificial.Intelligence.Aug.2008.eBook-ELOHiM.pdf.

[15] Chris Muise. *planning.domains*. http://planning.domains/. [Accessed 14-05-2024] .

[16] Malte Helmert a Silvia Richter. *HomePage - Fast Downward Homepage — fast-downward.org*. https://www.fast-downward.org. [Accessed 14-05-2024] .

[17] Hiroyukim Imabayshim. *Sokoban Official — sokoban.jp*. https://www.sokoban.jp. [Accessed 14-05-2024] .

[18] *Rush Hour - Online Play - ThinkFun — thinkfun.com*. https://www.thinkfun.com/rush-hour-online-play. [Accessed 14-05-2024] .

[19] Yasuhiro Fukushima a Masafumi Mijamoto. *SQUARE ENIX GLOBAL — square-enix.com*. https://www.square-enix.com. [Accessed 14-05-2024] .

[20] Richard Cobbett. *Hitman Go Review - IGN — ign.com*. https://www.ign.com/articles/2014/04/23/hitman-go-review. 2014. [Accessed 14-05-2024] .

[21] Alexa Ray Corriea. *Lara Croft Go Iterates on Hitman Go for a True Tomb Raider Adventure — gamespot.com*. https://www.gamespot.com/articles/lara-croft-go-iterates-on-hitman-go-for-a-true-tom/1100-6428337. 2015. [Accessed 14-05-2024] .

[22] Dinh Dinh Truong. *Using automated planning for intelligent player behaviour in a turn-based computer game*. 2023. https://dspace.cvut.cz/bitstream/handle/10467/108737/F3-BP-2023-Truong-Dinh%20Dinh-bachelor.pdf?sequence=-1&isAllowed=y.

[23] Jonathon Dornbush. *Deus Ex Go Review - IGN — ign.com*. https://www.ign.com/articles/2016/08/18/deus-ex-go-review. 2016. [Accessed 14-05-2024] .

[24] Epic Games. https://www.unrealengine.com. [Accessed 14-05-2024] .

[25] Unity Technologies. *Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine — unity.com*. https://unity.com. [Accessed 14-05-2024] .

[26] Klaus Pohl a Chris Rupp. *Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam - Foundation*

Level - IREB compliant. Rocky Nook, 2015. ISBN 978-1937538774. https://books.google.cz/books?hl=cs&lr=&id=1VsUDgAAQBAJ&oi=fnd&pg=PT38&dq=+Requirements+Engineering+Fundamentals:+A+Study+Guide+for+the+Certified+Professional+for+Requirements&ots=vA9Y2ROle9&sig=H2cW8WWbNox_UYnD8QbF8rTwJJU&redir_esc=y##v=onepage&q&f=false.

[27] Freepik, monkik a surang. https://www.flaticon.com.

[28] Gaudenz Alder. *Flowchart Maker & Online Diagram Software — app.diagrams.net*. https://app.diagrams.net. [Accessed 14-05-2024] .

[29] James Rumbaugh, Ivar Jacobson a Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999. ISBN 0-201-30998-X. https://idsi.md/files/file/referinte_utile_studenti/The%20Unified%20Modeling%20Language%20Reference%20Manual.pdf.

[30] Alexander Shvets. *Dive into Design Patterns*. Refactoring.Guru, 2018.

[31] Martin Komárek. *Přednáška 10 - Diagram nasazení*. https://moodle.fel.cvut.cz/pluginfile.php/338220/mod_resource/content/1/UML%20-%20Diagram%20nasazeni.pdf.

[32] Kamel Founadi, Solomon Hykes a and SebastienPahl. *Overview of the get started guide — docs.docker.com*. https://docs.docker.com/get-started. [Accessed 14-05-2024] .

[33] Blender Foundation. *About — blender.org — blender.org*. https://www.blender.org/about. [Accessed 14-05-2024] .

[34] John K Haas. *A History of the Unity Game Engine*. 2014. https://api.semanticscholar.org/CorpusID:86824974.

[35] Unity Technologies. *Unity - Manual: Unity architecture — docs.unity3d.com*. https://docs.unity3d.com/Manual/unity-architecture.html. [Accessed 14-05--2024] .

[36] Unity Technologies. *Unity - Manual: Overview of .NET in Unity — docs.unity3d.com*. https://docs.unity3d.com/Manual/overview-of-dot-net-in-unity.html. [Accessed 14-05-2024] .

[37] Alain Galvan. *Unity Engine Architecture — alain.xyz*. https://alain.xyz/blog/unity-engine-architecture. [Accessed 14-05-2024] .

[38] Unity Technologies. *Unity - Scripting API: MonoBehaviour — docs.unity3d.com*. https://docs.unity3d.com/ScriptReference/MonoBehaviour.html. [Accessed 14--05-2024] .

[39] John French. *Start vs Awake in Unity — gamedevbeginner.com*. https://gamedevbeginner.com/start-vs-awake-in-unity. [Accessed 14-05-2024] .

[40] Nahush Gowda. *Unity vs Unreal: Comparing Game Engine Architectures — nahush.gowda*. https://medium.com/@nahush.gowda/unity-vs-unreal-comparing-game-engine-architectures-55cc998db83f. [Accessed 14-05-2024] .

[41] Unity Technologies. *Unity - Manual: Scenes — docs.unity3d.com*. https://docs.unity3d.com/Manual/CreatingScenes.html. [Accessed 14-05-2024] .

[42] Unity Technologies. *Unity - Manual: Types of light — docs.unity3d.com*. https://docs.unity3d.com/Manual/Lighting.html. [Accessed 14-05-2024] .

[43] Unity Technologies. *Unity - Scripting API: EventSystem — docs.unity3d.com.* https://docs.unity3d.com/2018.2/Documentation/ScriptReference/EventSystems.EventSystem.html. [Accessed 14-05-2024] .

[44] Nayrissa. *371 Simple Buttons Pack | 2D Icons | Unity Asset Store — assetstore.unity.com.* https://assetstore.unity.com/packages/2d/gui/icons/371-simple-buttons-pack-97516. [Accessed 14-05-2024] .

[45] Unity Technologies. *Unity - Manual: Unity User Manual 2022.3 (LTS) — docs.unity3d.com.* https://docs.unity3d.com. [Accessed 14-05-2024] .

[46] David Holz. *Midjourney.* https://www.midjourney.com. [Accessed 14-05-2024] .

[47] Unity Technologies. *Explore the Unity Editor - Unity Learn — learn.unity.com.* https://learn.unity.com/tutorial/explore-the-unity-editor-1. [Accessed 14-05-2024] .

[48] Adam Green. *PDDL — planning.wiki.* https://planning.wiki/ref/pddl. [Accessed 14-05-2024] .

[49] Douglas Crockford. *JSON — json.org.* https://www.json.org/json-en.html. [Accessed 14-05-2024] .

[50] Unity Technologies. *Unity - Manual: ScriptableObject — docs.unity3d.com.* https://docs.unity3d.com/Manual/class-ScriptableObject.html. [Accessed 14--05-2024] .

[51] Unity Technologies. *Unity - Scripting API: PlayerPrefs — docs.unity3d.com.* https://docs.unity3d.com/ScriptReference/PlayerPrefs.html. [Accessed 14-05--2024] .

[52] Ron Patton. *Software testing.* Computer Press, 2002. ISBN 80-7226-636-5. https://dl.icdst.org/pdfs/files3/aede5f4a7bd951f08d6b4711beb59e42.pdf.

[53] Unity Technologies. *About Unity Test Framework | Test Framework | 1.4.4 — 1.4.* https://docs.unity3d.com/Packages/com.unity.test-framework@1.4/manual/index.html. [Accessed 14-05-2024] .

[54] Microsoft. *Assert.AreEqual Method (Microsoft.VisualStudio.TestTools.UnitTesting) — learn.microsoft.com.* https://learn.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert.areequal?view=visualstudiosdk-2022. [Accessed 14-05-2024] .

[55] John Reese, Theano Petersen, Okechukwu Somtochukwu, Pooja Poojari, Paul den Dulk, Tom Dykstra, Aleksei Mialkin, David Pine, Youssef Victor, Genevieve Warren, Andy De George, Nick Schonning, Maira Wenzel, Bruno Vinicius Figueiredo dos Santos a Sinan Kahveci. *Best practices for writing unit tests - .NET — learn.microsoft.com.* https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices. [Accessed 14-05-2024] .

Příloha A
Git repozitář projektu

Zde je popsána struktura přiloženého projektu. Repozitář projektu lze naleznout na následujícím URL: `https://gitlab.fel.cvut.cz/drahoane/mario-grid`

```
nico-nita.zip
   Assets
      3dModels.....................................Blender modely
      Prefabs.....................prefabrikáty objektů a UI prvků
      Resources..................materiály, textury, ScriptableObjects
      Scenes..............................Game Scene a Menu Scene
      Scripts.....................................C# skripty tříd
      Simple Buttons.........................package z Asset Store
      Streaming Assets
         Json.........................konfigurační soubory úrovní
         Pddl......soubory testovacích úrovní, domény, stavu hry a plánu
      Tests.....................................třídy s unit testy
      TextMesh Pro.....................................UI package
   exe.........................složka se spustitelným souborem hry
   Packages.....................................................dependencies
   ProjectSettings.....................Unity konfigurační soubory
   README.md.....................koncept, hratelnost a pravidla hry
   file.png.............................obrázek pozadí hlavního menu
```