



MATEMATICKO-FYZIKÁLNÍ FAKULTA Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Petr Geiger

Hra Dungeon Master pro platformu .NET

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D.

Studijní program: Informatika

Studijní obor: Programovaní a softwarové systémy

Praha 2016

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V dne

Podpis autora

Název práce: Hra Dungeon Master pro platformu .NET

Autor: Petr Geiger

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Cílem bakalářské práce je reimplementovat hru Dungeon Master. V současné době existuje již několik klonů této známé hry. Nicméně oproti nim se tato práce zaměřuje především na následující aspekty. Hra je naprogramována v jazyce C# s využitím platformy .NET. Dále celý engine je navrhnutý směrem k udržitelnosti a rozšířitelnosti. Tzn. s využitím tohoto enginu je možné naprogramovat i jinou hru založenou na podobných základech. Ale především je jednoduché přidávat do enginu nové funkce. Engine je také připravený na rozdílné vstupní formáty herních úrovní. Dále je také kompletně oddělena zobrazovací vrstva. Vzhledem k povaze projektu by engine měl sloužit jako ukázkový příklad použitelný pro vzdělávání.

Klíčová slova: RPG Dungeon Master Architektura softwaru vzdělávání

Title: Dungeon Master Game for the .NET Platform

Author: Petr Geiger

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract: The aim of this thesis is to reimplement the game Dungeon Master. There are currently several clones of this wellknown game. However, compared to them this work focuses on the following aspects. The game is programmed in C# using .NET platform. Furthermore, the entire engine is designed towards sustainability and scalability. That means that by using this engine it is possible to program slightly different game based on the same basics. Especially, it is easy to add new features to the engine. The engine is also prepared for different input formats of levels. Also the rendering layer of the game engine is completely separate. Due to nature of the project the engine should serve as prime example used for education.

Keywords: RPG Dungeon Master Software architecture education

Děkuji svému vedoucímu práce Mgr. Pavlu Ježkovi, Ph. D. jak za pomoc s výběrem práce, tak za cenné rady při její tvorbě.

Obsah

1	Úvod	3
1.1	Reimplementace Dungeon Masteru	4
1.2	Cíle	4
2	Analýza	6
2.1	Vstupní data pro herní úrovně – DUNGEON.DAT	7
2.1.1	Dlaždice (Tiles)	8
2.1.2	Textová data	10
2.1.3	Objekty	10
2.2	Data s vlastnostmi objektů – GRAPHICS.DAT	13
2.2.1	Akce a kombi	14
2.2.2	Předměty	14
2.2.3	Kouzla a symboly	15
2.2.4	Nepřátelské entity	15
2.3	Parsování vstupních dat	16
2.3.1	Herní úrovně – DUNGEON.DAT	16
2.3.2	Vlastnosti objektů – GRAPHICS.DAT	17
2.4	Výběr frameworku pro práci s grafickým výstupem	18
2.5	Reprezentace jádra enginu	19
2.6	Reprezentace dlaždic	19
2.7	Přepínače	21
2.7.1	Příklady	22
2.7.2	Reprezentace přepínačů	24
2.7.3	Reprezentace zprávy přepínače	27
2.8	Reprezentace entit	27
2.8.1	Vlastnosti	28
2.8.2	Dovednosti	28
2.8.3	Relace mezi entitami	28
2.8.4	Tělo a inventáře	29
2.9	Reprezentace předmětů	29
2.10	Reprezentace komb a akcí	30
2.11	Reprezentace kouzel	30
2.12	Builder herních úrovní	31
2.13	Inicializace objektů	32
2.14	Renderování a interakce	33
3	Vývojová dokumentace	35
3.1	Jádro enginu – DungeonBase	35
3.1.1	Asynchronní funkce	36
3.1.2	Správa herních úrovní a dlaždic	37
3.1.3	Aktualizace a rendering	38
3.2	Dlaždice – ITile	39
3.2.1	Inicializace dlaždic	41
3.2.2	Implementace dlaždic	41
3.2.3	Strany dlaždic – ITileSide	43

3.3	Přepínače – IActuatorX	44
3.3.1	Implementace obecných přepínačů	44
3.3.2	Implementace přepínačů se senzory	45
3.4	Herní entity	49
3.4.1	Implementace vlastností entit	49
3.4.2	Implementace dovedností entit	50
3.4.3	Tělo – IBody	50
3.4.4	Rozmístění entity na dlaždici	51
3.4.5	Relace s dalšími entitami	54
3.4.6	Implementace entit	55
3.5	Předměty	57
3.5.1	Implementace předmětů	57
3.6	Akce	57
3.7	Kouzla	58
3.8	Renderery	58
3.9	Builder map	59
3.9.1	Vlastní implementace builderu	59
3.9.2	Implementace builderu Dungeon Masteru	60
4	Uživatelská dokumentace	61
4.1	Mechaniky ve hře	61
4.2	Cíl hry	62
4.3	Ovládání	62
4.3.1	Aktivace přepínačů a sbírání předmětů	62
4.3.2	Souboj	63
4.3.3	Odpočinek	63
Závěr	64	
Seznam použité literatury	65	
Seznam obrázků	66	
Přílohy	67	

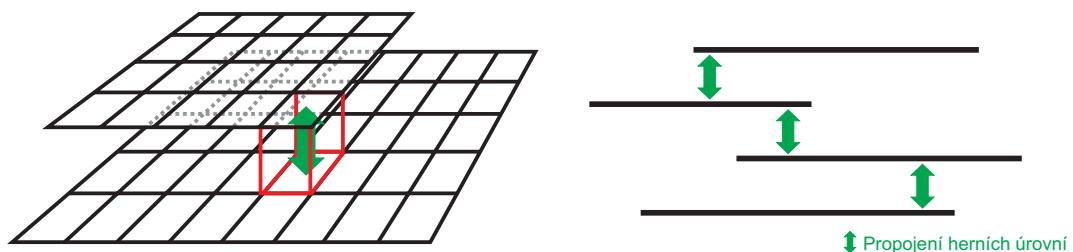
1. Úvod

Dungeon Master je počítačová hra žánru RPG (role playing game) vyvinuta firmou Faster Than Light v roce 1987. Byla to první real-time hra tohoto typu s pseudo 3D pohledem a ovládáním pomocí myši. Hráč má k dispozici skupinu až čtyř hrdinů, s kterými prochází podzemní bludiště a bojuje s nepřáteli (viz obr. 1.1). Tito hrdinové se ve hře nazývají šampioni a mohou se zdokonalovat v různých dovednostech.



Obrázek 1.1: Screenshot originální hry Dungeon Master

Bludiště se skládá z několika úrovní uspořádaných vertikálně pod sebou. Jednotlivé úrovně pak nemusí být stejně velké a mohou být od sebe různě horizontálně odsazené. Každá úroveň je tvořena obdélníkovou plochou s pravidelnou mřížkou (viz obr. 1.2). Pole vymezena mřížkou nazýváme dlaždice a je jich ve hře několik typů, které definují jejich vzhled a funkci. Některé dlaždice lze aktivovat tzv. přepínači. Takové typy dlaždic jsou například dveře nebo jáma (viz obr. 1.1), které lze takto otvírat či zavírat. Přepínače mohou být buď nášlapné na podlaze nebo aktivovatelné pomocí myši na zdech. Mezi úrovněmi lze sestupovat pomocí dlaždic typu schody. Dále je možné se teleportovat mezi dlaždicemi, a to i v různých úrovních, pomocí dlaždice typu teleport.

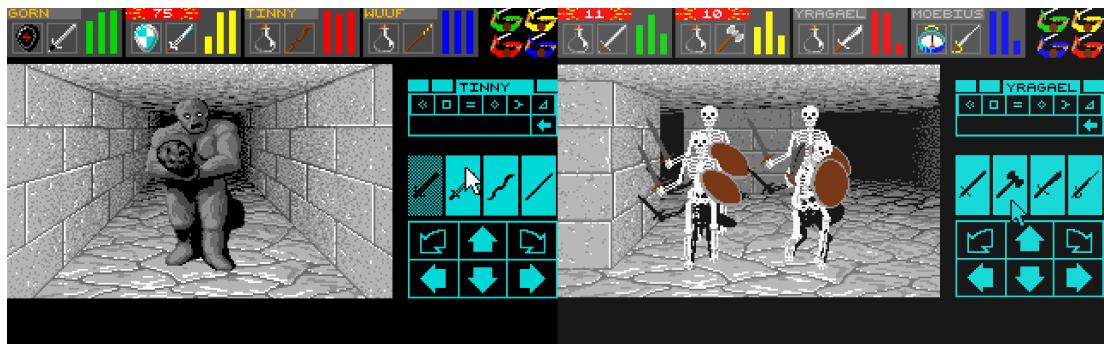


Obrázek 1.2: Ilustrace uspořádání herních úrovní.

Hráč je na začátku postaven do nejvýše položené úrovně, kde si vybere svoji skupinu šampionů. Pohyb skupiny mezi dlaždicemi je zcela diskrétní, to znamená, že se nelze se skupinou zastavit mezi dlaždicemi, ale pohyb je vždy dokončen až na vedlejší dlaždici. Se skupinou je tedy vždy asociována pouze jedna dlaždice. Na dlaždicích pak mohou být různé předměty, které je možné sbírat. Šampioni mohou s předměty provádět různé akce. Například se zbraněmi lze bojovat nebo

lektvary je možné pít a zlepšit si tak dočasně vlastnosti. Kromě těchto akcí může ještě šampion vyvolávat kouzla. Nicméně k tomu potřebuje dostatečnou úroveň odpovídající dovednosti. Tímto způsobem je možné vytvářet lektvary nebo vyvolávat útočná či obraná kouzla.

Ve hře je celá řada nepřátelských entit. Liší se vzhledem, útokem a pohybem. Pohyb některých entit probíhá ještě po hustší mrázce než jsou dlaždice. Každá entita má definovaný prostor dlaždice, který zaujímá. Je to buď prostor celé dlaždice, polovina či čtvrtina. (viz obr. 1.3) Pohyb entit je potom opět diskrétní jako v případě hráčovi skupiny, ale tentokrát mezi definovanými částmi dlaždic.



Obrázek 1.3: Ilustrace prostorů zaujímaných entitami.

1.1 Reimplementace Dungeon Masteru

Výsledkem reimplementace má být engine vhodný pro výuku jazyka C#. Z tohoto důvodu je nutné, aby byl jazyk C# použit pro implementaci enginu. Jelikož C# je objektově orientovaný jazyk, je třeba, aby výsledný engine měl dobrý objektový návrh, který bude sloužit jako vhodný materiál k jeho výuce. Díky tomu by mělo být možné vytvářet úlohy pro studenty, v kterých by si mohli doimplementovat další funkce enginu nad rámec této práce, a tak si vyzkoušet objektově orientované programování v praxi. Proto by také mělo být možné pro engine jednoduše dodělat komponentu, která by prováděla odlišný výstup enginu, díky kterému by mohli být úkoly kontrolované například formou CodExu (Code Examinator).

Pro využití enginu způsobem popsaném v předchozím odstavci, je tedy zejména nutné, aby engine obsahoval podporu pro herní mechaniky hry Dungeon Master – které by pak bylo možné rozšiřovat. Následkem toho musí engine obsahovat komponentu pro převedení vstupních dat z originální hry na odpovídající herní úrovně. Kromě toho je třeba poskytnout mechanismus, kterým bude možné dodávat vlastní implementace těchto komponent. V poslední řadě je nutné oddělit výstup enginu do zvláštní vrstvy, aby ji bylo možné také jednoduše změnit poskytnutím nové implementace.

1.2 Cíle

Cílem této práce je tedy vytvořit engine pro hru Dungeon Master, tak aby splňovala následující požadavky:

- C1** Engine bude naprogramovaný v jazyce C#.
- C2** Engine bude obsahovat podporu pro funkce a mechaniky vyskytující se ve hře Dungeon Master.
- C3** Bude kladen důraz na dobrý objektový návrh, tak aby byl engine co nejlépe rozšířitelný a bylo tak možné do enginu dodávat jednoduše nové funkce.
- C4** Engine bude schopný sestavit herní úrovně podle vstupních dat použitých v originální hře. Nicméně bude také poskytovat možnost, dodělat si podporu pro jiné formáty.
- C5** Engine bude obsahovat oddělenou zobrazovací vrstvu, tak aby mohl být její výstup jednoduše změněn.
- C6** Projekt bude cílený pro vzdělávaní, tak aby si studenti mohli vyzkoušet do enginu doprogramovat další funkce.

2. Analýza

Tato kapitola nejprve blíže popisuje formát a mechaniky originální hry a dále pojednává o postupech a řešeních použitých při implementaci nového enginu. Jsou zde popsány jak slepé a nevhodné návrhy, tak návrhy, které se ukázaly jako nevhodnější řešení.

Pro splnění práce bylo v prvé řadě zapotřebí získat vstupní data originální hry Dungeon Master. Nejdůležitější data jsou obsažená ve dvou souborech: **DUNGEON.DAT** a **GRAPHICS.DAT**. První z nich obsahuje definice herních úrovní a seznamy použitých předmětů, přepínačů a nepřátelských entit. Druhý z nich obsahuje konkrétní vlastnosti objektů a entit použitých v prvním souboru. Soubor **GRAPHICS.DAT** pak také obsahuje definicí akcí, vlastnosti kouzel, textury, atd. Později se ukázalo, že pouze data hry nebudou pro implementaci většiny funkcí dostačující – jedná se například o přesný způsob získávání dovedností šampionů, vyvolávání kouzel nebo provádění akcí. Dalším zdrojem jsou proto dekomplilované zdrojové kódy [9] originální hry v jazyce C s archaickou konvencí Kernighan & Ritchie.¹ Tyto zdrojové kódy jsou často náročné na porozumění, nicméně potřebné informace pro dokončení této práce se z nich podařilo získat.

¹ Brian Kernighan a Dennis Ritchie jsou autoři první verze jazyka C, jehož neformální specifikaci popsali v první edici knihy *The C Programming Language* [10]. Podle iniciálů autorů této knihy je tato první verze jazyka známa jako K&R C. Mezi největší specifika K&R C patří:

1. deklarace funkcí nespecifikuje parametry a není prováděna jejich kontrola,
2. funkce nemají návratový typ **void**,
3. při definici funkce jsou typy parametrů specifikovány na separátních řádcích za hlavičkou funkce,
4. lokální proměnné funkcí jsou deklarovány na začátku bloku funkce.

Ukázka kódu:

```
//declaration
typedef int VOID; //viz bod 2
VOID write_sorted(); //viz bod 1

//definition
VOID write_sorted(first, second) //viz bod 3
int first;
int second;
{
    int min; //viz bod 4
    int max;

    printf("We are about to find out sorted sequence\n");

    if(first < second){
        min = first;
        max = second;
    } else {
        min = second;
        max = first;
    }
    printf("Sorted values are (%d, %d)\n", min, max);
}
```

Pro detailnější popis dat Dungeon Masteru si bude dobré nejprve ujasnit vlastnosti a schopnosti šampionů. Každý šampion má následující vlastnosti:

- *Zdraví* (Health) – určuje kolik útoku šampion vydrží než zemře,
- *Výdrž* (Stamina) – určuje kolik akcí je schopen šampion vykonat než se unaví,
- *Mana* (Mana) – reprezentuje magickou energii pro vyvolávání kouzel,
- *Zatížení* (Load) – určuje maximální hmotnost, kterou je šampion schopen unést,
- *Síla* (Strength) – hodnota je používána pro výpočet zranění, síly hodu předmětů a maximální výše zatížení,
- *Obratnost* (Agility) – zvyšuje pravděpodobnost zásahu nepřítele a pomáhá se vyhnout nepřátelským útokům,
- *Moudrost* (Wisdom) – je to vlastnost důležitá pro kouzelníky a kněze, určuje rychlosť obnovy many,
- *Vitalita* (Vitality) – určuje rychlosť obnovy zdraví a výdrže,
- *Odolnost proti magii* (Magic Defense) – snižuje účinnost magických útoků,
- *Odolnost proti ohni* (Fire Defense) – snižuje účinnost ohnivých útoků,
- *Jídlo a Voda* (Food and Water) – pomocí těchto hodnot je obnovována výdrž a zdraví
- *Štěstí* (Luck) – zvyšuje či snižuje pravděpodobnost provedení akce.

Každý šampion má ve hře čtyři základní dovednosti a to bojovník, ninja, kněz a kouzelník. Kromě základních dovedností má šampion ještě šestnáct skrytých dovedností, které nejsou hráči zobrazeny. Každá z těchto skrytých dovedností náleží k nějaké ze základních dovedností. Jak moc daný šampion disponuje danou dovedností určuje úroveň dovednosti. Tato úroveň lze navýšovat pomocí zkušeností, které lze získat souboji nebo prováděním akcí. Každá akce pak může navýšovat zkušenosti pro jinou dovednost. Pokud jsou v některé ze skrytých dovedností získány zkušenosti, dostane odpovídající zkušenosti i její základní dovednost. Po získání dostatečného počtu zkušeností v nějaké dovednosti, získá šampion v této dovednosti novou úroveň. Nová úroveň potom navýší určité vlastnosti šampiona. Přesné hodnoty potřebných zkušeností a navýšovaných vlastností pro odpovídající dovednosti jsou pevně stanovené ve zdrojových kódech hry.

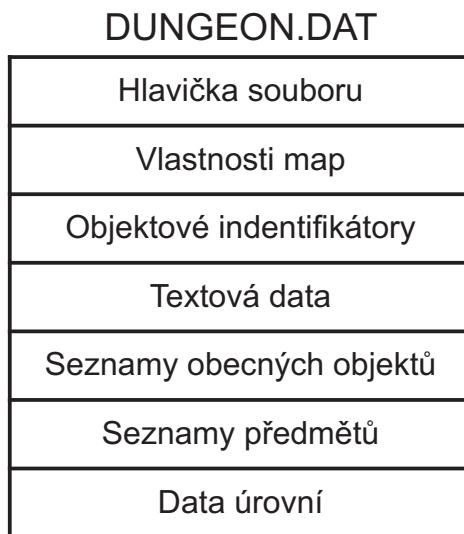
2.1 Vstupní data pro herní úrovně – DUNGEON.DAT

Originální hra má herní úrovně definované v binárním souboru **DUNGEON.DAT**. Formát tohoto souboru nebyl tvůrci nikdy zveřejněn, nicméně kolem hry se vy-

tvořila početná komunita, která k němu dokumentaci [6] vytvořila. Tuto dokumentaci jsme použili k porozumění obsahu souboru. Následuje stručný popis zmíněného souboru, pro kompletní dokumentaci navštívte přímo stránky dokumentace.

Soubor **DUNGEON.DAT** obsahuje jednotlivé herních úrovně a objekty v nich obsažené. Formát souboru lze rozdělit do následujících bloků: (viz obr. 2.1)

- Hlavička souboru – obsahuje velikost jednotlivých bloků souboru nebo velikost seznamů objektů.
- Vlastnosti herních úrovní – mezi tyto vlastnosti patří rozměry dané úrovně, tj. počet dlaždic na šířku a výšku. Dále definuje obtížnost úrovně, která je použita pro výpočet získaných zkušeností nebo zdraví nepřátelských entit. Každá úroveň má také určené použité podmnožiny dekorací na zdech, dveřích a přepínačích. Dekorace jsou v originální hře identifikovány čísly, která jsou napevno zabudovaná v kódu hry. Některé dekorace tak mohou mít ještě speciální význam, který je identifikován pouze podle zmíněného číselného identifikátoru. Jedná se například o dekorace s výklenky, které mohou být umístěny na zdech. V tomto případě originální herní engine rozpozná, že lze na dané místo vkládat předměty.
- Textová data – obsahuje všechny texty v hře.
- Seznamy obecných objektů – mezi tyto objekty patří, dveře, teleporty, popisky na zdech a přepínače.
- Seznamy předmětů – jsou to předměty, které se dají v herních úrovních sbírat a používat.
- Data úrovní – obsahují data pro každou dlaždici v každé úrovni.

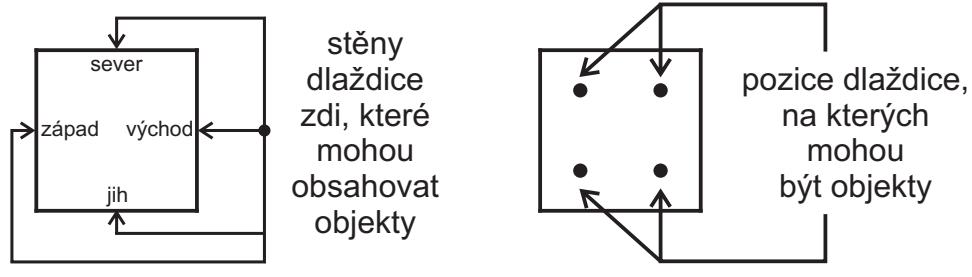


Obrázek 2.1: Ilustrace formátu souboru **DUNGEON.DAT**.

2.1.1 Dlaždice (Tiles)

Dlaždice se dělí na několik typů a pomocí přepínačů jím lze zaslat zprávu, na kterou mohou reagovat. Zpráva obsahuje akci a datovou položku. Akce dlaždici říká, zda se má aktivovat či deaktivovat. Co daná datová položka znamená si určuje každý typ cílové dlaždice sám. Originální hra pak pracuje s následujícími typy dlaždic:

- *Zed'* – zajišťuje zobrazování stěn a nelze na ni vstoupit. Pro každou stěnu (sever, východ, jih, západ – viz obr. 2.2a) dlaždice *zdi* je určeno, zda může mít tzv. náhodnou dekoraci. Pokud ji může mít, engine podle náhodného generátoru určí jaká to bude (případně žádná). Každá strana může obsahovat přepínač, který si může do *zdi* uložit předměty. Pokud má některá strana *zdi* dekoraci výklenku, jsou v něm zobrazeny předměty uložené ve *zdi*. Na stěnách *zdi* ještě mohou být popisky, které lze zobrazovat nebo skrývat pomocí zaslání zprávy. Aktivační zpráva popisek zviditelní a deaktivací skryje. Datová položka zprávy zde určuje, pro kterou ze stěn *zdi* je zpráva určena.
- *Podlaha* – Po těchto dlaždicích se hráč běžně pohybuje se svou skupinou. Obdobně jako u stran *zdi* může definovat, zda zobrazuje na podlaze náhodnou dekoraci. Podlaha dále může mít nášlapný přepínač. Dlaždice obsahuje čtyři pozice, na které lze pokládat předměty. Tyto pozice vzniknou rozdělením plochy dlaždice na čtvrtiny (viz obr. 2.2b).
- *Jáma* – *Jáma* může být buď otevřená nebo zavřená. Nicméně lze na ni vždy vstoupit a pokud je otevřená hráč spadne s svou skupinou o úroveň níže. Pád může být i přes několik úrovní. Živé objekty pak po dopadu obdrží zranění. Přijmutím aktivační zprávy se jáma otevře a deaktivací se zavře.
- *Schody* – Po této dlaždici se může hráč dostat o úroveň výše resp. níže.
- *Dveře* – Na těchto dlaždicích je vždy objekt dveře. Na dlaždici lze vstoupit pouze v případě, že jsou dveře otevřeny. Otevření je možné provést aktivační zprávou či pomocí tlačítka – pokud ho samotné dveře obsahují. Některé dveře lze také rozbít útokem, to si ale definuje již samotný objekt dveří.
- *Teleport* – Tyto dlaždice obsahují objekt teleport, který určuje, které objekty dokáže přenést. Mohou to být předměty, hráčova skupina nebo nepřátelské entity. Po vstupu resp. vložení objektu na dlaždici, je objekt teleportován, pokud je odpovídajícího typu. Příjem aktivační resp. deaktivacní zprávy aktivuje resp. deaktivuje teleport.
- *Iluze zdi* – Tato dlaždice může být iluze zdi a nebo otevíratelná zed'. V obou případech je vizuálně neodlišitelná od dlaždice typu *zdi*. V prvním případě je možné na dlaždici vstoupit vkročením do zdi. V druhém případě lze zed' odstranit pomocí zaslání odpovídající zprávy na dlaždici. Aktivační zpráva otevře zed', deaktivací zavře.



(a) Dlaždice zeď – stěny.

(b) Dlaždice podlaha – pozice.

Obrázek 2.2: Možné pozice pro uložení objektů na dlaždici.

Samotné úrovně jsou pak vždy definovány následujícími seznamy v tomto pořadí:

- seznam dlaždic – seřazeny po sloupcích dané úrovně,
- seznam dekorací příšer,
- seznam dekorací zdí,
- seznam dekorací podlah.

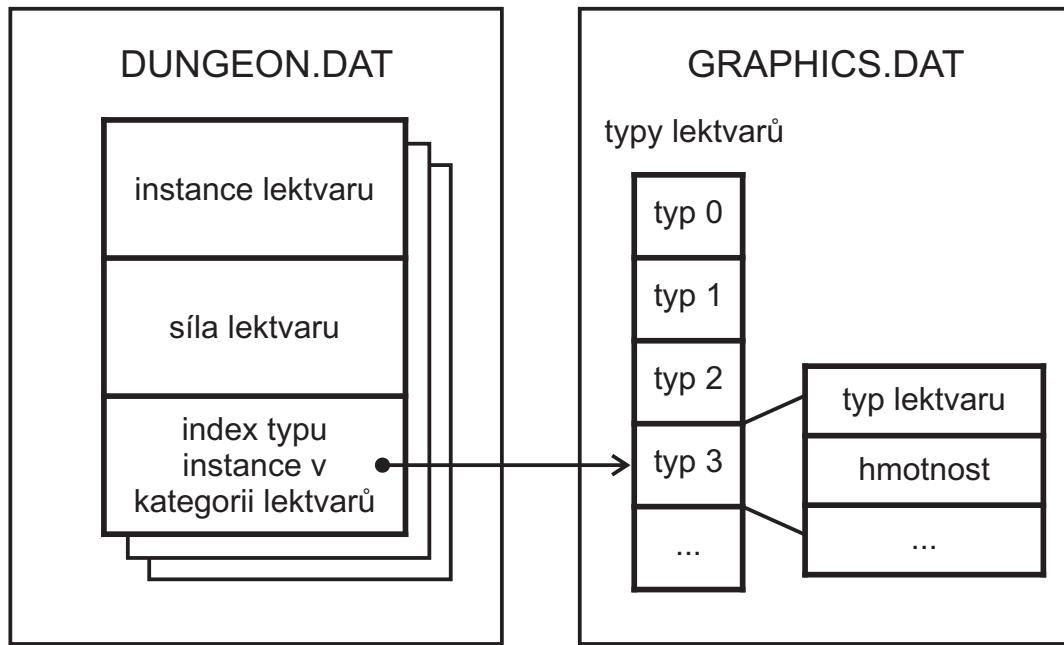
Počty jednotlivých dekorací a dlaždic jsou popsány ve vlastnostech úrovní.

2.1.2 Textová data

V binárním souboru jsou často data uložena po slovech procesoru, které v tomto případě mají velikost dvou bajtů. V této části souboru jsou uložena data s použitými texty ve hře. Tyto texty používají speciální kódování, kdy jednotlivé znaky jsou uloženy do slov (těch procesorových) a každé toto slovo obsahuje tři znaky. Všechny texty jsou pak uloženy za sebou v jednom bloku a ke konkrétním textům je přistupováno přes offsety počtu bajtů od začátku textových dat. Přesný popis může být opět nalezený v komunitní dokumentaci [6].

2.1.3 Objekty

Tyto objekty uložené v souboru **DUNGEON.DAT** tvoří jednotlivé instance. Odporovající typy k těmto instancím jsou potom spolu s jejich neměnnými vlastnostmi uloženy v souboru **GRAPHICS.DAT**, jehož obsah si rozebereme v následující sekci 2.2. Tyto typy se pak ještě dělí do kategorií popsaných dále v této sekci. Pro určení konkrétního typu má potom každá instance uložen index typu v dané kategorii (viz obr. 2.3). Dále pak objekty reprezentující instance obsahují pouze vlastnosti, které jsou rozdílné pro každou konkrétní instanci.



Obrázek 2.3: Ilustrace vztahu instancí a typů.

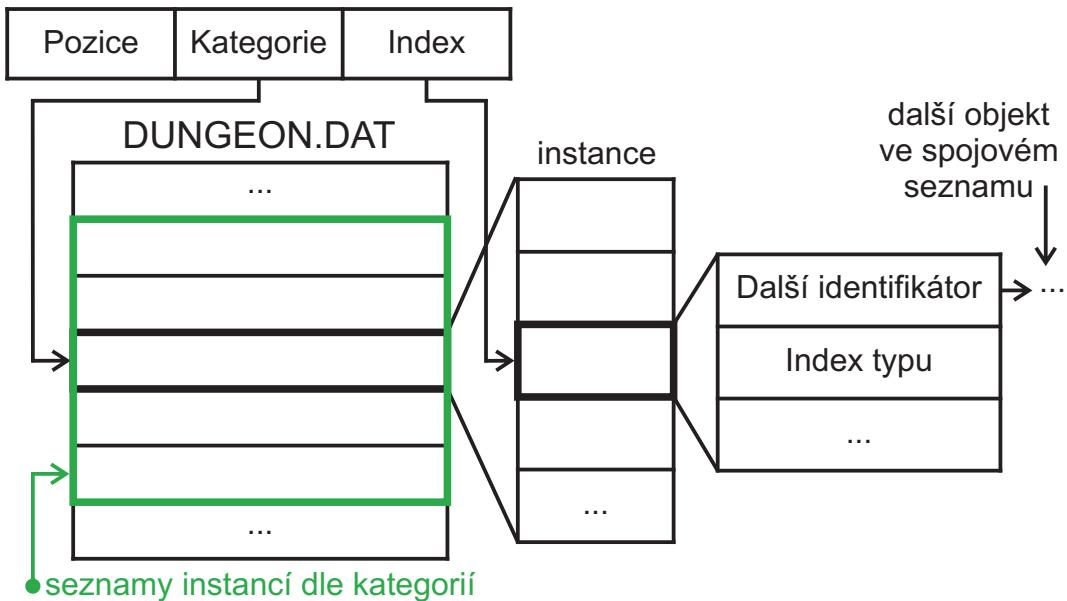
Identifikátory

Ke každé instanci objektu existuje unikátní identifikátor. Tento identifikátor se skládá z následujících částí:

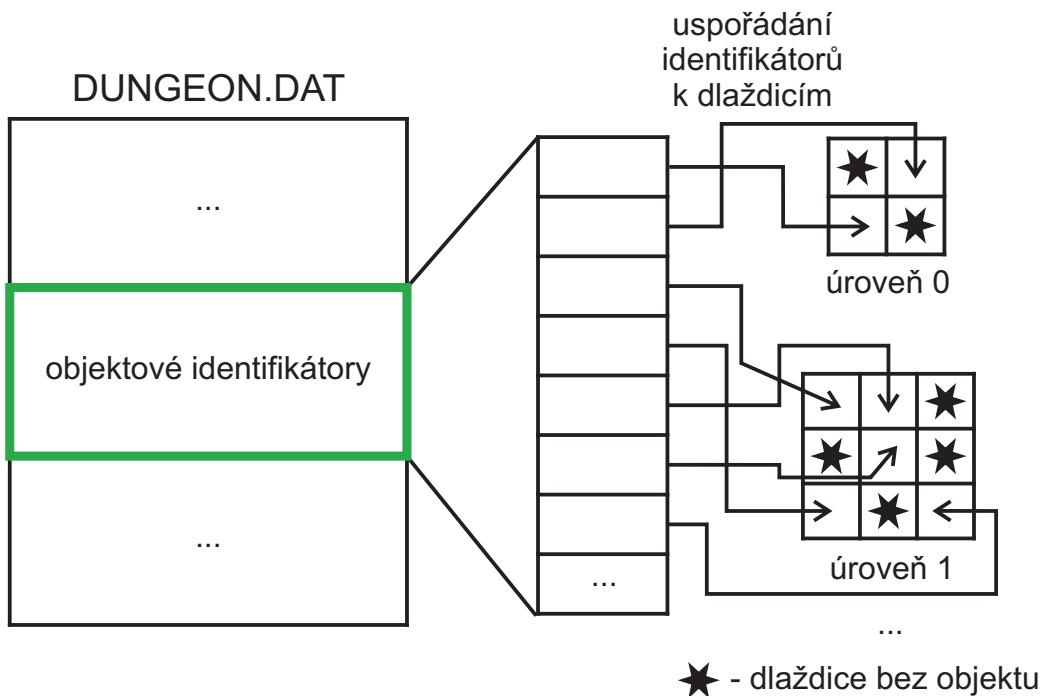
- Pozice – lze buď interpretovat jako světová strana nebo umístění na dlaždici, jedná-li se o podlahu.
- Kategorie typu objektu – pro každou kategorii existuje v souboru **DUNGEON.DAT** seznam jejich instancí.
- Index - určuje index objektu v seznamu instancí jeho kategorie.

Tyto identifikátory se dají chápout jako reference na konkrétní instance objektů (viz obr. 2.4) a za pomocí nich jsou vytvořeny spojové seznamy. Každá dlaždice pak specifikuje, zda může mít spojový seznam. Seznam objektových identifikátorů obsahuje první identifikátory spojových seznamů, na dlaždicích, které je mají. Seznam je uspořádaný pro všechny dlaždice od první do poslední úrovně, vždy po sloupcích odshora dolů (viz obr. 2.5).

Identifikátor:



Obrázek 2.4: Ilustrace obsahu identifikátoru a jeho využití ve spojových seznamech.



Obrázek 2.5: Ilustrace uspořádání prvních objektových identifikátorů dlaždic.

Předměty

Předměty jsou objekty, které lze sbírat v herních úrovních a ukládat je k šampionům nebo je používat. Dělí se do následujících kategorií:

- *zbraně* – lze je vložit šampionům do ruky a pak s nimi mohou bojovat,
- *oblečení* – lze jím obléknout šampiony, a tak zvýšit jejich odolnost,
- *svitky* – obsahují text, který může hráči usnadnit hru,
- *lektvary* – dělí se na lektvary modifikující šampionovi schopnosti a na lektvary provádějící nějaké akce – např.: výbuch. Každá instance má vlastnost určující sílu efektu.
- *kontejnery* – mohou obsahovat další předměty,
- *různé* – v této kategorii je například jídlo a pak různé předměty, s kterými nelze provádět žádné akce.

Ostatní objekty

Tyto objekty slouží jako pomocné objekty pro dlaždice a dělí se do následujících kategorií:

- *Dveře* – můžou být pouze na dlaždici typu *dveře* a obsahují informace, zda-li jsou dveře rozbitelné a jestli mají tlačítko, kterým je lze otevřít
- *Teleport* – může být pouze na dlaždici typu *teleport* a definuje cílovou dlaždici a kategorii objektů, které je schopen teleportovat.
- *Textové popisky* – mohou být pouze na dlaždici typu *zed'* a obsahují odkaz na konkrétní text do textových dat.
- *Nepřátelské entity* – definuje celou skupinu nepřátelských entity na dlaždici, jejich počet, rozmístění na dlaždici a předměty které lze získat jejich zabitem.
- *Senzory* – vytvářejí základní herní mechaniky bludiště. Senzory mají daný číselný typ, pomocí kterého hra určí jakým způsobem je možné senzor aktivovat. Po aktivaci senzor buď může provést lokální akci nebo odeslat zprávu s akcí na vzdálenou dlaždici. Přepínače se pak typicky skládají z několika senzorů. Více o přepínačích a senzorech bude zmíněno v sekci 2.7.

2.2 Data s vlastnostmi objektů – GRAPHICS.DAT

Soubor **GRAPHICS.DAT** obsahuje textury dekorací, vlastnosti předmětů a objektů, definici akcí a kouzel. Formát tohoto souboru nebyl tvůrci uveřejněn, avšak komunitě se opět podařilo data vyextrahovat i z tohoto souboru. K některým částem také existuje komunitní dokumentace [8], nicméně už není ucelená jako v případě dokumentace pro soubor **DUNGEON.DAT**. Zároveň jsou vyextrahovaná data zveřejněna na webu v HTML formátu ([2], [4], [3], [5], [7]). Z toho důvodů jsme se rozhodli pro využití již extrahovaných dat v HTML formátu. Dále následuje podrobnější popis využitého obsahu souboru.

2.2.1 Akce a komba

S předměty je možné provádět akce. Množina akcí pro daný předmět se nazývá kombo a obsahuje až tři akce. Ve hře je k dispozici 44 akcí a stejný počet komb. Kompletní seznam jednotlivých akcí a komb lze nalézt v komunitní dokumentaci [2]. Zde si popišme alespoň jejich vlastnosti.

Vlastnosti akcí jsou :

- *název* – textový popis akce ve hře,
- *zkušenosti* – počet zkušeností získaných po provedení akce,
- *dovednost* – identifikátor dovednosti, která získá zkušenosti,
- *obrana* – modifikátor obrany při používání akce,
- *výdrž* – modifikátor výdrže nutné pro provedení akce,
- pravděpodobnost provedení akce
- *zranění* – modifikátor pro výpočet konečného zranění nepřítele,
- *únava* – doba po kterou nelze provádět žádné akce.

Vlastnosti pro každou akci komba jsou:

- index akce ze seznamu akcí,
- úroveň dovednosti – minimální úroveň dovednosti pro úspěšné provedení akce.

2.2.2 Předměty

Ke každému typu předmětu existují v tomto souboru popisovače vlastností [3]. Pro každý předmět jsou definovány následující vlastnosti:

- globální identifikátor – unikátní identifikátor, který je použit k identifikaci konkrétního typu předmětu například v přepínačích,
- index útočného kombu,
- *lokace* – místo, na kterou část těla nebo, do kterého inventáře šampiona lze předmět vložit,
- index v kategorii – index typu předmětu v dané kategorii.

Globální identifikátor lze z instance předmětu získat pomocí jeho kategorie a indexu typu předmětu, který má každá instance předmětu. Každý předmět má také definovanou hmotnost. Zbraně a oblečení mají navíc definované následující vlastnosti jejichž hodnoty lze nalézt v komunitní dokumentaci [4].

Zbraně:

- poškození – hodnota zranění aplikována při útoku na nepřítele,
- kinetická energie – hodnota udává, jak daleko lze zbraň hodit,

- střelné poškození – hodnota dodatečného poškození u střelných zbraní.

Oblečení:

- síla brnění – obrana, kterou oblečení poskytuje,
- odolnost proti útoky ostrými předměty.

2.2.3 Kouzla a symboly

Další částí dat uložené v souboru **GRAPHICS.DAT** jsou vlastnosti kouzel. Pojdeme si nejprve přiblížit, jak se ve hře kouzla přesně vyvolávají. Každé kouzlo je složeno z několika symbolů, přičemž první z nich je speciální a jde o tzv. „power symbol“. Power symbol určuje, jak bude celkové kouzlo silné. Další symboly už určují konkrétní kouzlo. Každé vyvolání symbolu stojí šampiona *manu*. Po stanovení všech symbolů může šampion vyvolat samotné kouzlo. Vyvolání kouzla může selhat, pokud nemá šampion dostatečnou úroveň dovednosti vyžadované kouzlem. Datový soubor tedy obsahuje pro každý symbol při odpovídajícím power symbolu množství potřebné many pro jeho vyvolání. Dále obsahuje následující vlastnosti kouzel [5]:

- Obtížnost (Difficulty) – modifikátor obtížnosti vyvolání kouzla,
- Doba trvání (Duration) – po tuto dobu nemůže šampion vyvolávat další kouzla,
- Úroveň dovednosti (Skill level) – postačující úroveň určité dovednosti pro úspěšné vyvolání kouzla.

2.2.4 Nepřátelské entity

Poslední částí obsaženou v souboru **GRAPHICS.DAT**, kterou jsme využili, jsou vlastnosti nepřátelských entit. Následující výčet obsahuje vlastnosti, které implementuje nový engine.

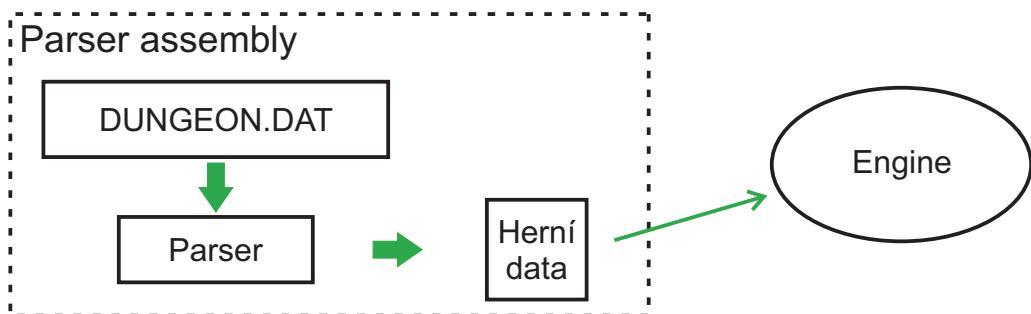
- *Doba pohybu* (Movement duration) – doba, po které se entita rozhodne změnit pozici.
- *Doba útoku* (Attack duration) – doba, po které může entita provést útok.
- *Síla brnění* (Armor) – určuje odolnost proti útokům.
- *Zdraví* (Base health) – hodnota použitá pro výpočet zdravý nových nepřátelských entit.
- *Síla útoku* (Attack power) – hodnota pro výpočet útku.
- *Otrava* (Poison) – určuje, jak velkou otáru způsobí útok.
- *Obraha* (Defense) – určuje, obtížnost pro šampiona, se kterou je entita zraněna.
- *Detekce* (Detection range) – určuje vzdálenost, na kterou detekuje nepřítele.

- *Odlnost proti ohni* (Fire resistance) – odolnost proti kouzlům způsobující ohnivý útok.
- *Odlnost proti jedu* (Poison resistance) – odolnost proti kouzlům způsobující jedový útok.
- *Pravděpodobnosti zranění částí těla* (Wounds) – pravděpodobnost pro každou část těla nepřítele, s kterou ji entita zraní.

Celý seznam vlastností je možné nalézt v komunitní dokumentaci [7].

2.3 Parsování vstupních dat

V době implementace této části práce nebylo jasné, zda nalezená dokumentace [6] bude dostatečná pro splnění práce. Parser je proto oddělen od zbytku enginu do zvláštní assembly, která nereferencuje žádnou knihovnu pro zobrazování grafiky. Cílem této assembly je tedy pouze převést struktury obsažené ve vstupních datech do odpovídajících datových tříd v jazyce C#. Tento balík dat se potom předá enginu, který z nich vytvoří herní úrovně. Toto schéma znázorňuje následující obrázek 2.6.



Obrázek 2.6: Ilustrace vztahu parseru vůči enginu.

2.3.1 Herní úrovně – DUNGEON.DAT

Parsování herních úrovní je rozděleno do dvou fází. V první fázi probíhá transformace dat objektů (viz sekce 2.1) ze souboru **DUNGEON.DAT** na odpovídající objekty v jazyce C#. V druhé fázi jsou pak objekty z první fáze upraveny, aby měly přehlednější strukturu.

První fáze

V této fázi je nejprve vytvořen datový objekt, který bude shromažďovat všechny ostatní data. Tento objekt obsahuje:

- metadata z hlavičky souboru – to jsou zejména velikosti všech seznamů s objekty (viz sekce 2.1) a počet map,
- seznamy objektů – to jsou seznamy objektů jazyka C# vzniklé rozparsováním odpovídajících dat objektů v souboru,

- seznam herních úrovní – to jsou objekty reprezentující data herních úrovní, které obsahují jejich vlastnosti (viz sekce 2.1) a seznam jejich dlaždic, dekorací, atd.

Dále parser rozparsuje všechny data vstupního souboru dle komunitní dokumentace [6] a vytvoří z nich adekvátní objekty, které uloží do zmíněného datového objektu. Ilustrace této fáze je na obrázku 2.7.

Druhá fáze

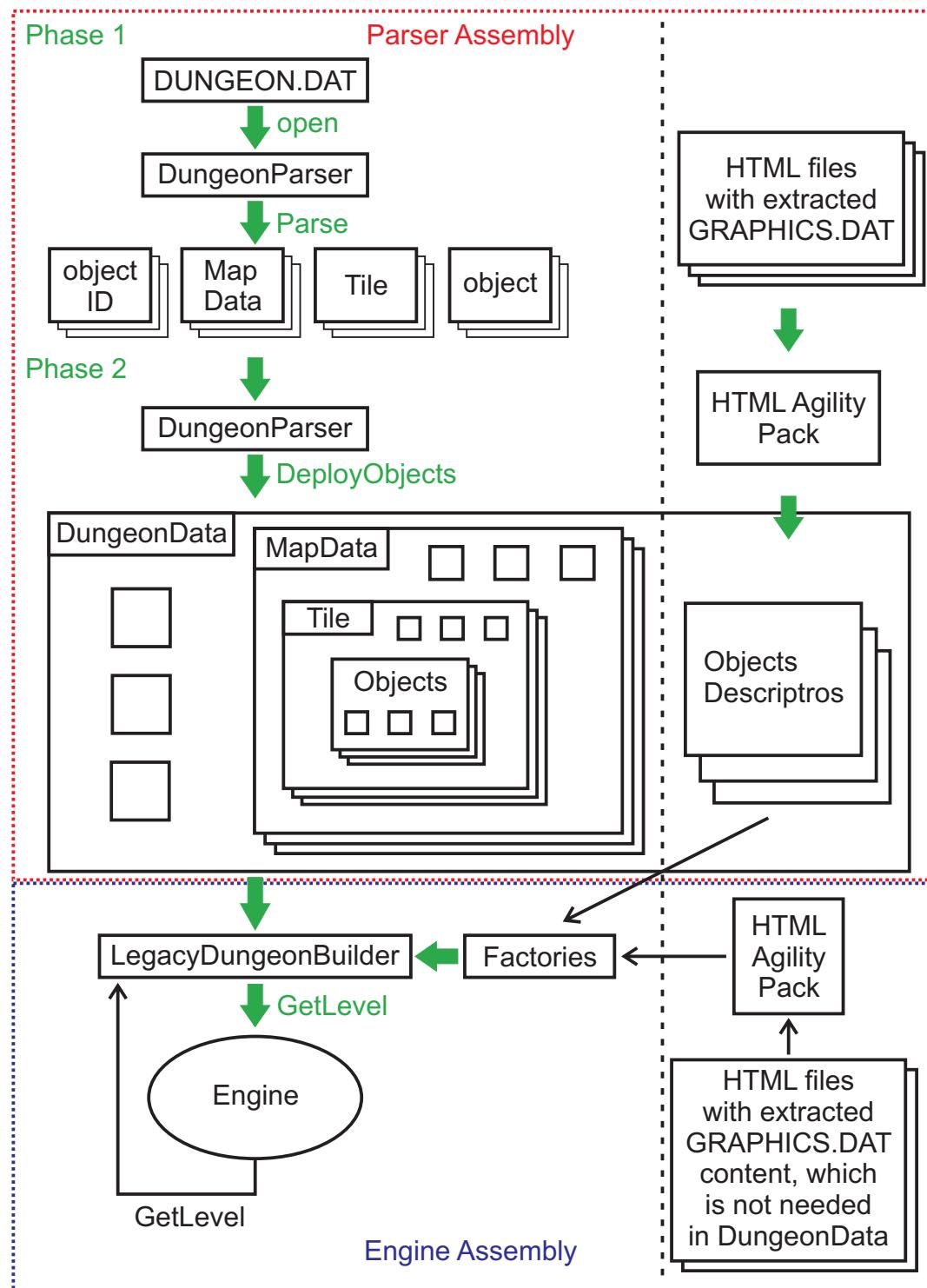
Po skončení první fáze je struktura vzniklého datového objektu podobná struktuře souboru **DUNGEON.DAT**. Tím máme namysli, že například spojové seznamy jsou reprezentovány pomocí objektových identifikátorů (viz sekce 2.1), což není příliš intuitivní a pro práci s těmito daty by engine musel rozumět jejich formátu. Navíc spojové seznamy obsahují objekty různých typů. Proto pro zjednodušení práce enginu s těmito daty jsou v datových objektech inicializovány ještě následující položky:

- seznam sbíratelných předmětů,
- seznam senzorů,
- seznam popisků,
- seznam nepřátelských entit,
- položka dveří pro dlaždici typu dveře,
- položka teleport pro dlaždici typu teleport,
- a pro dlaždici typu zeď a podlaha stanoví identifikátory náhodných dekorací.

Na počátku vývoje projekt neobsahoval druhou fázi, její vytvoření vedlo k zjednodušení kódu v enginu.

2.3.2 Vlastnosti objektů – **GRAPHICS.DAT**

Pro sestavení herních úrovní potřebuje engine také vlastnosti společné pro všechny konkrétní typy objektů ze souboru **DUNGEON.DAT**. Jak bylo zmíněno v sekci 2.2, tyto vlastnosti jsou v originální hře v souboru **GRAPHICS.DAT**. Nicméně tato data již existují vyextrahovaná v HTML formátu, a proto jsme je použili namísto původních dat. Ke zjednodušení parsování dat z HTML souborů byla použita knihovna HTML Agility Pack [11]. Pro každé typy vlastností jsou pak vytvořeny odpovídající datové objekty které jsou uloženy do seznamů ve výsledném datovém objektu zmíněného v předchozí sekci 2.3.1. V této části jsou získána data vlastností všech objektů (viz sekce 2.1) až na kouzla. Vlastnosti kouzel jsou rozparsovány v enginu přímo do objektů v něm použitých. Proto by případné vytváření datových souborů v této vrstvě bylo zbytečné. Celý proces zobrazuje diagram na obrázku 2.7.



Obrázek 2.7: Průběh parsování herních dat.

2.4 Výběr frameworku pro práci s grafickým výstupem

Po úspěšném rozparsování herní dat se bylo třeba rozhodnout, kterou knihovnou použít pro zobrazení grafického výstupu enginu. Aby se bylo možné soustře-

dit hlavně na samotný návrh enginu (viz cíl práce **C3**), použili jsme framework, s který už jsme měli nějaké zkušenosti. Jedná se o XNA Framework od firmy Microsoft. Nicméně tento framework již není firmou nadále podporován a vyvíjen, proto jsme využili nakonec klon MonoGame [1]. Výhodou této volby je také to, že framework je multiplatformní. Jeho tvůrci tvrdí, že podporuje platformy iOS, Android, MacOS, Linux, Windows, OUYA, PS4, PSVita, Xbox One.

2.5 Reprezentace jádra enginu

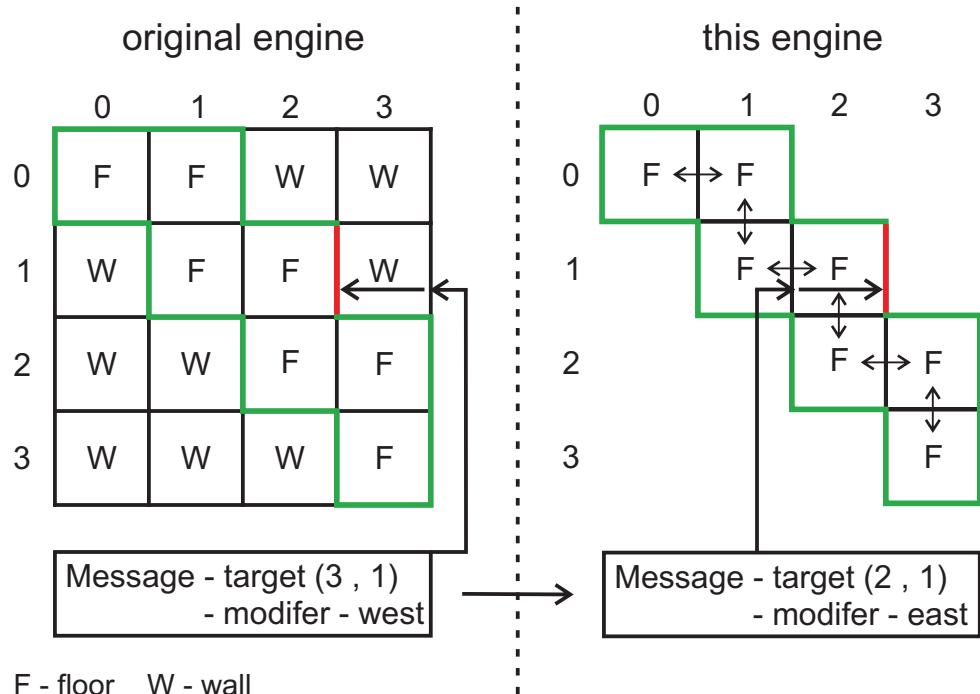
O samotnou herní smyčku se stará framework MonoGame, který na jádru enginu volá aktualizační a vykreslovaní obsluhy. Jádro enginu pak obsahuje následující tři důležité komponenty:

- Hráč – tato komponenta zajišťuje interakci s uživatelem. Skrze ni lze ovládat hráčovu skupinu šampionů, sbírat předměty nebo bojovat.
- Builder herních úrovní – tato komponenta zajišťuje sestavování herních úrovní.
- Objekt obsahující factories – pro každý typ popsaný v sekci 2.2 existuje odpovídající factory obsahující jeho vlastnosti. Factories pak mohou vytvářet objekty, jejichž typ reprezentují. Toto bude konkrétně specifikováno později (viz sekce 2.15), nicméně důležité je, že skrze jádro enginu lze k těmto factories přistupovat.

Podrobnosti o jádře enginu jsou ve vývojové dokumentaci (viz sekce 3.1).

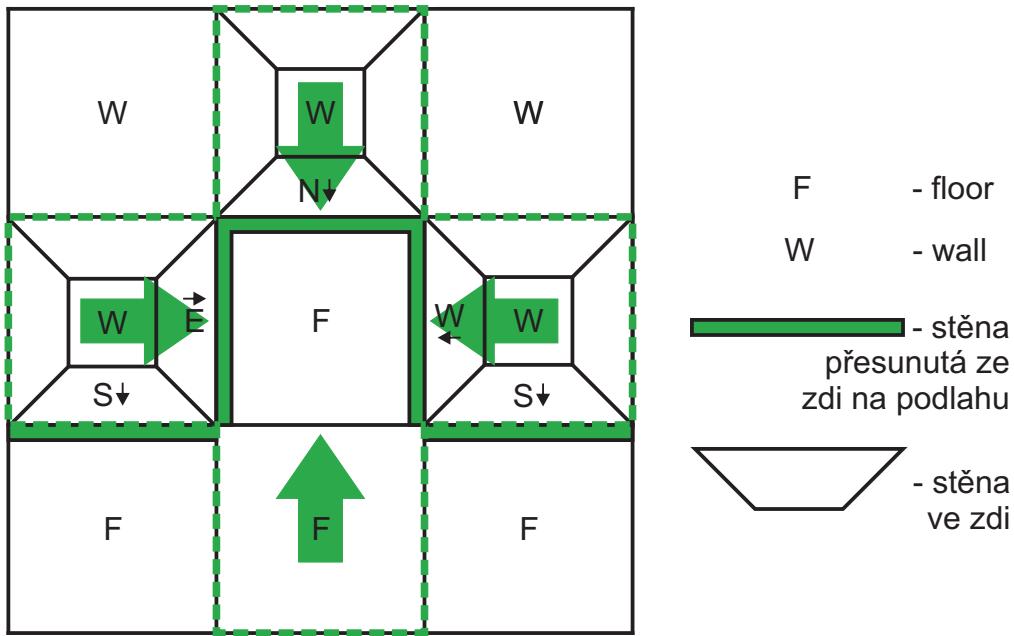
2.6 Reprezentace dlaždic

V originální hře jsou dlaždice vždy uloženy v obdélníkovém poli (viz sekce 2.1 a obr 2.8 vlevo). To znamená, že dlaždice, které nejsou využity pro chodby, jsou vždy vyplněny zdmi. Jelikož jedním z cílů této práce je udělat engine co nejrozšířitelnější (viz cíl práce **C3**), rozhodli jsme se tento problém vyřešit o něco obecněji (viz obr. 2.8 vpravo). Každá dlaždice je vrchol grafu a pokud jsou dvě dlaždice sousední jsou mezi nimi hrany oběma směry, to dohromady tvoří obousměrně orientovaný graf. Nicméně, pro dlaždice stále platí, že mají nanejvýš čtyři sousedy a jsou uspořádané do mřížky. Tato reprezentace pro hodně řídké mapy může ušetřit paměť. Avšak důležitější je, že při takové reprezentaci lze jednoduše získat sousedy pouze ze znalosti konkrétní dlaždice. V této podobě by bylo navíc jednodušší engine upravit tak, aby dlaždice nemusely být na mřížce a aby mohly mít více sousedů.



Obrázek 2.8: Uložení dlaždic v originálním vs tomto enginu.

Předchozí rozhodnutí také vede k tomu, že již nejsou potřeba dlaždice typu zeď (viz obr. 2.8), jelikož její stěny můžou být definovány ve dlaždicích typu podlaha (viz obr. 2.9). Při vytváření podlahy je pak nutné čist i její sousední dlaždice. Pokud je soused daným směrem zed', vytvoří se na této straně stěna a soused není nastaven. V opačném případě je soused nastaven na odpovídající dlaždice. Zprávy původně cílené zdí je také třeba posílat na odpovídající dlaždice podlahy (viz obr. 2.8). Tato reprezentace zdí opět vede k vyšší flexibilitě, jelikož případné místo využití původními dlaždicemi typu zdi lze využít jiným způsobem.



Obrázek 2.9: Ilustrace přesunu stěn do dlaždice *podlahy*

Pokud jsou tedy stěny součástí dlaždic, je třeba vyřešit jejich reprezentaci. Buděj možné, aby veškeré funkce stěn obsahovala přímo dlaždice, a nebo je možné tyto části delegovat do zvláštních objektů. Ze začátku byl v enginu použit první zmínovaný způsob. Nicméně tato reprezentace není příliš vhodná, protože potom samotná dlaždice řeší funkce, které k ní přímo nenáleží. Nakonec tato reprezentace byla změněna a v enginu byla použita druhá zmínovaná možnost. Vedlo k ní zejména oddělení zobrazovací vrstvy od logické. Výhoda tohoto přístupu byla kromě samotného oddělení kódu i možnost znova použití kódu za pomocí dědičnosti. Například třída reprezentující stěnu, která obsahuje přepínač, může dědit ze třídy reprezentující prázdné stěny. Avšak ukázala se i nevýhoda tohoto přístupu, a to, že komunikace směrem ze stěny k dlaždici je trochu těžkopádná. Buděj by bylo zapotřebí předat zdi referenci na jejich rodičovské dlaždice, nebo by se museli použít události. Pokud to bylo v některých případech nutné, přiklonili jsem se k použití událostí.

2.7 Přepínače

Přepínače tvoří základní mechaniky v herních úrovních Dungeon Masteru, mohou být buď na stěnách, nebo na podlaze. Na stěnách je lze aktivovat kliknutím myši a na podlaze potom přesunutím hráče na dlaždici s přepínačem. Aktivace ještě může být podmíněna dalšími faktory: obsah daného typu předmětu v ruce nebo směr kterým je hráč otočen.

Ve skutečnosti se každý přepínač skládá z řady senzorů a platí, že:

- poslední senzor určuje dekoraci přepínače,
- při pokusu o aktivaci přepínače, dojde postupně k pokusu aktivace každého senzoru,

- každý senzor může definovat akci, kterou provede, pokud byl aktivován.

Senzor má následující vlastnost:

- Typ – číselné označení, dle kterého originální herní engine určí způsob aktivace.
- Akce – může být buď lokální, nebo vzdálená. Pro lokální akce je to zarotování sekvencí senzorů přepínače nebo přidání zkušeností hráči. Pro vzdálené akce je to odeslání zprávy (viz sekce 2.1) na danou cílovou dlaždici.
- Data - pro každý typ senzoru se může obsah lišit,
- Zpoždění – doba, po které se provede akce.
- Opačný efekt – liší se dle typu senzoru, nicméně většinou otáčí podmínku aktivace pro daný typ senzoru.
- Opakovatelnost – určuje, zda lze senzor aktivovat neomezeně krát či pouze jednou.
- Dekorace – číselný identifikátor dekorace pro danou herní úroveň.

Výše popsaný systém přepínačů přináší hned několik problémů:

- P1** Z technické dokumentace [6] není vždy jasně zřejmé, jaká je aktivační podmínka senzoru pro určitý typ.
- P2** Jakým způsobem efektivně a přehledně rozparsovat data senzorů, když mají tolik nezávislých vlastností.
- P3** Jak reprezentovat přepínače v novém enginu.
- P4** Jakým způsobem provádět vzdálené akce.

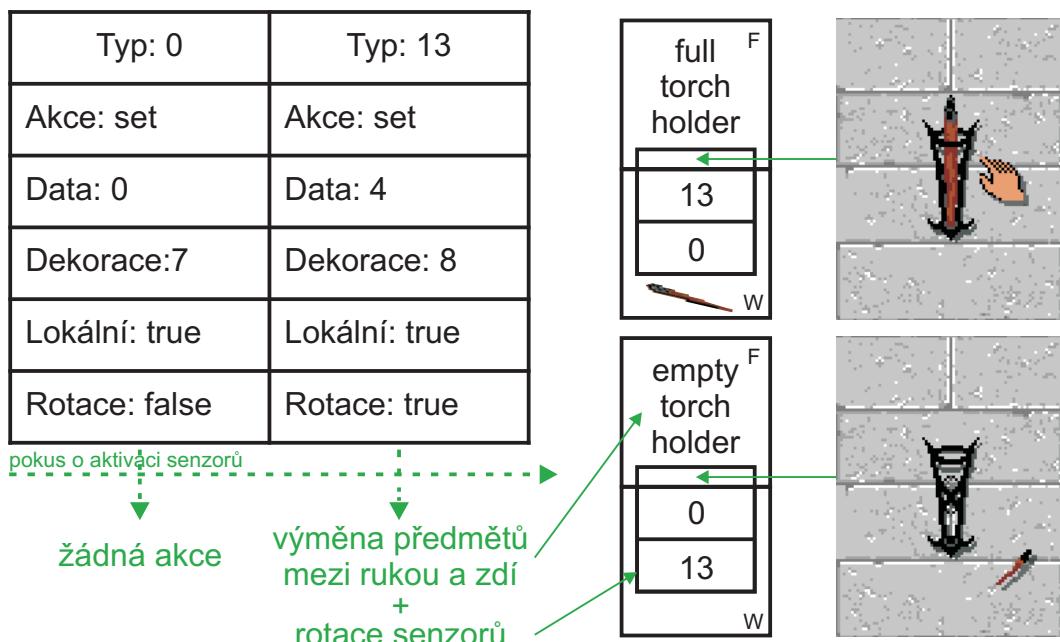
2.7.1 Příklady

Pro ujasnění celého konceptu přepínačů si pojďme ukázat několik příkladů. Textový popis aktivační podmínky senzorů pro jednotlivé jejich typy může být nalezen v komunitní dokumentaci [6]. Přesná funkce senzorů pak může být nalezna v dekompliovaných zdrojových kódech [9] v následujících funkcích, které jsou v souboru **SENSOR.C**:

- **F275_aszz_SENSOR_IsTriggeredByClickOnWall** – chování pro senzory na zdech,
- **F276_qzzz_SENSOR_ProcessThingAdditionOrRemoval** – chování pro senzory na podlaze.

Držák na pochodně

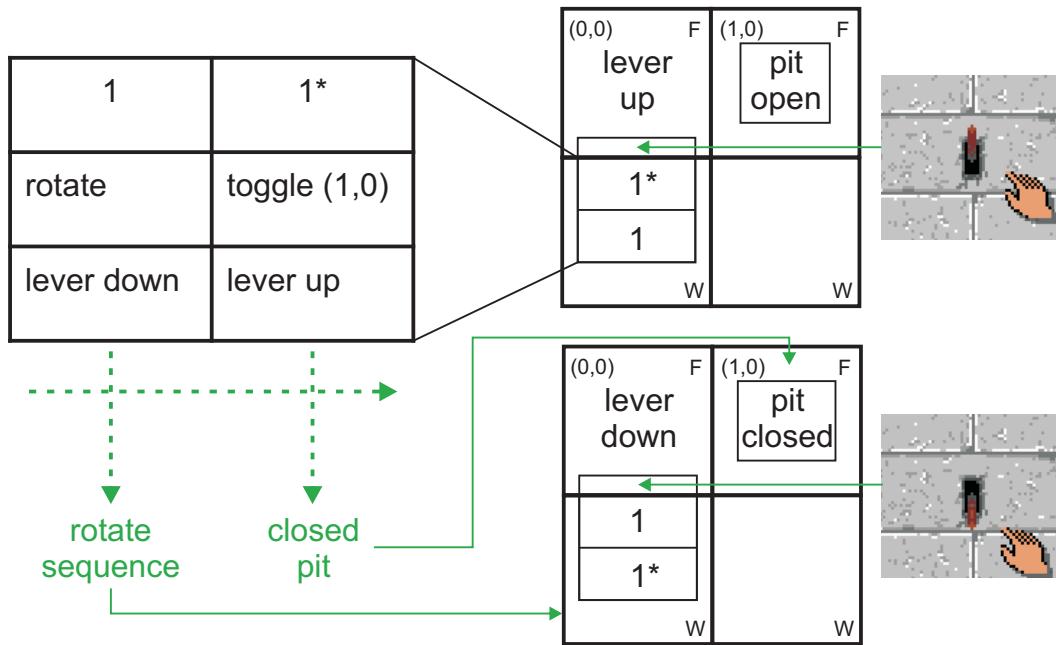
Držák na pochodně (viz obr. 2.10) se skládá ze dvou senzorů. Senzor na první pozici je typu 0, což znamená, že tento senzor nelze aktivovat. Tento senzor může pouze zobrazovat svoji dekoraci (v tomto případě prázdný držák na pochodně), pokud je na posledním místě sekvence, ostatní jeho vlastnosti nejsou využity. Senzor na druhé pozici je typu 13, což znamená, že tento senzor lze aktivovat, pouze pokud má hráč prázdnou ruku a pochodeň je uložena ve zdi přepínače (viz první část obr. 2.10) nebo naopak (viz druhá část obrázku 2.10). Dekorace této pochodně je držák s pochodní. Položka data pak určuje, pro jaký typ předmětu platí aktivační podmínka (v tomto případě tj. identifikátor pochodně). A efekt aktivace je rotace sekvence senzorů. Veškeré akce senzorů v tomto případě tedy dělá druhý senzor. Při aktivaci vezme pochodeň uloženou ve zdi, dá jí hráči do ruky (při druhé aktivaci naopak) a provede rotaci senzorů, čímž se zobrazí dekorace prvního senzoru (při druhé aktivaci dekorace druhého senzoru).



Obrázek 2.10: Ilustrace přepínače pro držák pochodní.

Páka

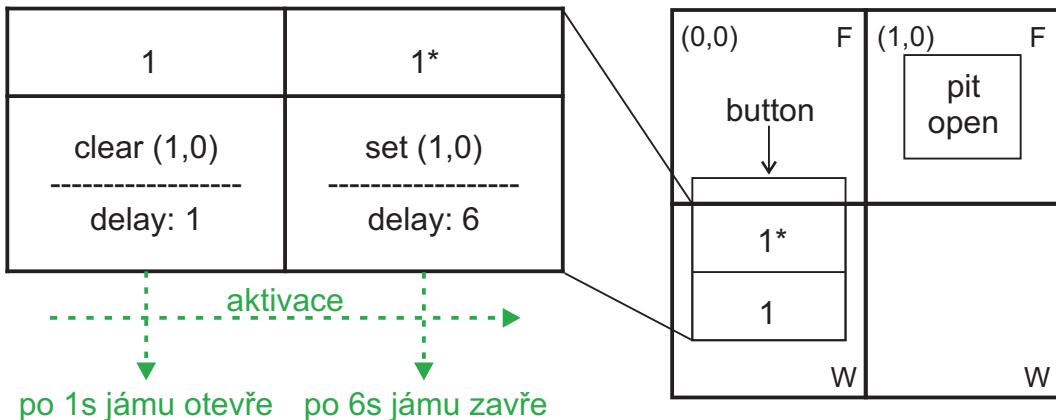
Páku tvoří dva senzory stejného typu 1, který znamená, že senzor lze aktivovat s čimkoliv v hráčově ruce (i s ničím). První senzor má potom lokální akci rotace senzorů a dekoraci páky dole (viz obr. 2.11). Druhý senzor má dekoraci s pákou nahoře a naopak vzdálenou akci – prohození stavu – dlaždice jámy na pozici (1,0). Po aktivaci prvního senzoru tedy dojde k rotaci senzorů a změně dekorace na páku dole. Druhý senzor potom odešle zprávu na cílovou dlaždici jámy, která se následně zavře resp. otevře. Opětovná aktivace přepínače vrátí vše do původního stavu.



Obrázek 2.11: Ilustrace přepínače reprezentující páku.

Tlačítko

Toto tlačítko se skládá opět ze dvou senzorů typu 1 (viz předchozí příklad). Oba senzory potom mají vzdálenou akci na dlaždici (1,0) - první senzor má deaktivaci zprávu a druhý aktivaci (viz obr. 2.12). První senzor má potom opoždění jednu sekundu a druhý šest. Z toho důvodu dojde 1 sekundu po aktivaci přepínače k otevření jámy na pět sekund.



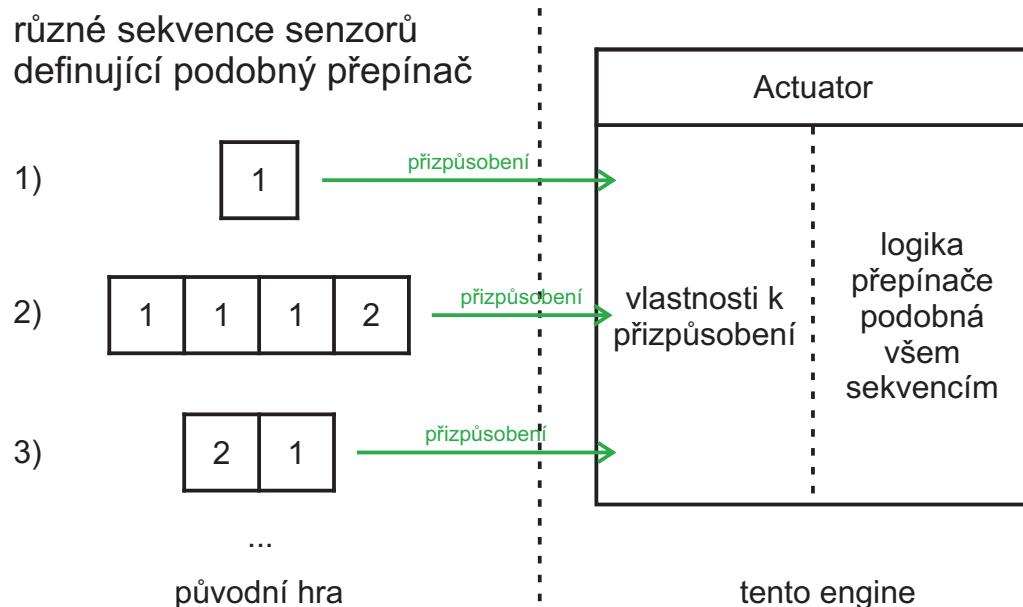
Obrázek 2.12: Ilustrace přepínače reprezentující dočasné tlačítko.

2.7.2 Reprezentace přepínačů

První tři body problémů (viz **P1**, **P2**, **P3**) řeší tato sekce.

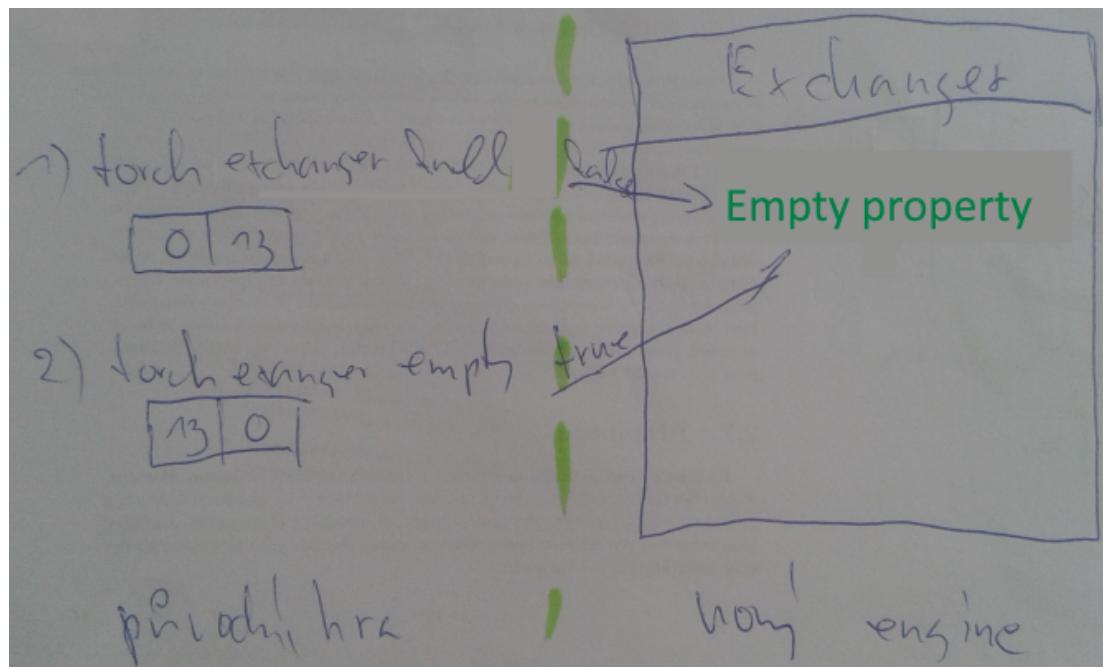
První reprezentace

Jako první způsob jsme se rozhodli vždy pro několik sekvencí senzorů vytvořit objekt, který měl stejné nebo o něco obecnější vlastnosti než dané sekvence. Z počátku se totiž zdálo, že jen výjimečně jsou sekvence senzorů větší či složitější. Proto jsme se rozhodli vytvořit v novém enginu objekty, které mají obecnější vlastnosti, a tak mohou být použité pro několik podobných sekvencí. Dle určité sekvence senzorů se poté objektu inicializují jeho vlastnosti. Celou situaci ilustruje obr. 2.13.



Obrázek 2.13: Ilustrace transformace sekvence senzorů na objekt přepínač.

Konkrétním příkladem může být objekt přepínače reprezentující držák na pochodně (viz obr. 2.14). Kdy první sekvence reprezentuje držák s pochodní a druhá držák bez ní – při vytváření objektu držáku se pak specifikuje, zda má či nemá pochodeň.



Obrázek 2.14: Ilustrace transformace sekvence senzorů na objekt přepínač.

Tento způsob sice řeší problém samotné reprezentace(viz bod **P3**), ale později se ukázalo, že velice nevhodně. Například kód převádějící sekvence na přepínače je složitý a nepřehledný. Bylo také složité vytvořit přepínač podle technické dokumentace [6] tak, aby fungoval správně. Popis v dokumentaci [6] také není dostačeně přesný pro vytvoření tohoto úkolu. I proto čím více přepínačů jsme tímto způsobem naimplementovali, tím více se ukazovalo problémů. Nakonec jsme dospěli k závěru, že tento způsob reprezentace je nemožný.

Druhá reprezentace

Tato reprezentace se snažila vyřešit problém nepřehlednosti kódu(viz bod **P2**) převádějícího sekvence na objekty přepínačů. Proto je i zde použit typ objektu přepínač z první reprezentace v sekci 2.7.2. Parsování sekvence senzorů jsme se rozhodli udělat pomocí konečného automatu. Tedy tak, že pro každý objekt přepínače existovaly předdefinované sekvence senzorů. Konečný automat potom pomocí vstupní sekvence identifikoval výsledný objekt přepínače. Při inicializaci objektu přepínače zde pak byla sekvence jasně definovaná, což řešilo problém s přehledností (viz bod **P2**). Později se však ukázalo, že i malá změna v sekvenci senzorů může generovat podobný objekt. Takže by bylo třeba spoustu předdefinovaných šablon, které by se lišily ve drobnostech. Tím se ukázalo, že tento přístup není dostatečně obecný a nelze jej tedy použít.

Třetí reprezentace – výsledná

Tato reprezentace se oproti předchozím snažila přiblížit co nejvíce k systému použitém v enginu originální hry. K dosažení tohoto cíle jsme se rozhodli použít dekomplikované zdrojové kódy [9] hry. Tím je vyřešen problém nedostatečné dokumentace ve věci přepínačů(viz bod **P1**).

Vytvořili jsme tedy objekt přepínač, který má v sobě sekvenci senzorů, tak jako tomu je v originální hře. Tím je vyřešen i problém nepřehlednosti převodního kódu (viz bod **P2**), jelikož jsou tyto reprezentace téměř identické. S využitím zdrojových kódů jsme vytvořili odpovídající objektově orientovaný kód v jazyce C#. Tato reprezentace tak zajišťuje maximální korektnost implementace a přitom poskytuje možnost rozšíření – což je jedním z cílů této práce (viz bod **C3**). Je tedy možné vytvořit nové senzory, které budou mít vlastní aktivační podmínky a ty následně použít v přepínačích. V originálním enginu je toto rozšiřovaní jen značně omezené, jednotlivé typy senzorů se tam odlišují jednobajtovým číselným identifikátorem. Ale jelikož jsme nechtěli rozšiřování enginu limitovat pouze na tento způsob vytváření senzorů, je možné si vytvořit přepínač, který bude fungovat na úplně jiné bázi. Takže případné nové přepínače nemusí vůbec senzory používat.

2.7.3 Reprezentace zprávy přepínače

Jak již bylo řečeno (viz sekce 2.1), celý systém mechanik herních úrovní je závislý na posílání zpráv. V originální hře je cíl odeslané zprávy vázán na její souřadnice dlaždice. Jelikož jsme se v enginu nechtěli vázat na tyto pevné souřadnice, rozhodli jsem se namísto nich cílovou dlaždici identifikovat její referencí. To by případně ulehčilo práci při transformaci enginu tak, aby dlaždice mohly mít více sousedů nebo aby nemusely být na pevné mřížce (viz sekce 2.6). Při takovéto reprezentaci však nastává problém s inicializací dlaždic a přepínačů (viz sekce 2.13).

V originální hře lze mezi dlaždicemi posílat pouze jeden typ zprávy. Jelikož cílem této práce je udělat engine co nejrozšířitelnější (viz bod **C3**), engine poskytuje za určitých podmínek používat i jiné typy zpráv. Těmito podmínkami jsou:

- Vlastní zprávy lze zasílat pouze vlastně vytvořeným dlaždicím, které na daný typ zprávy musí být připraveny.
- Všechny dlaždice musí umět přijímat originální typ zprávy – nicméně nemusí na ně reagovat.
- Třída reprezentující vlastní zprávu musí být potomkem originální zprávy v hierarchii dědičnosti jazyka C#.

2.8 Reprezentace entit

V originální hře existují pouze dva typy živých objektů, tj. nepřátelské entity a šampioni. Za účelem udělat engine co nejrozšířitelnější (viz cíl práce **C3**) jsme se rozhodli rozdělit entity na živé a neživé. Entitou je v tomto enginu každý objekt, s jehož vlastnostmi může interagovat nějaká akce (viz sekce 2.2.1). Například akce útok může entitu zranit nebo poškodit (pokud je neživá). Příkladem použité neživé entity jsou dveře, které je možné rozbít útokem. Živé entity disponují následujícími atributy:

- vlastnosti – jsou to vlastnosti jako zdraví, odolnost, atd. (viz sekce 2),

- dovednosti – to ve výsledku znamená, že jsou schopny používat akce,
- relaci vůči ostatním živým entitám - určuje, zda jsou vůči daným entitám přátelské či nepřátelské,
- části těla a inventáře,
- poloha na dlaždici – definují, jaké část dlaždic mohou zaujmít.

Každý z těchto bodů lze rozšířit, což může být užitečné v případě použití enginu pro implementaci jiné hry než Dungeon Master.

2.8.1 Vlastnosti

V původní hře jsou vlastnosti šampionů (viz sekce 2) a nepřátelských entit (viz sekce 2.2.4) pevně definované. Několik vlastností nepřátelských entit je odlišných od tě šampionových, nicméně ve většině se shodují. S těmito odlišnostmi musí počítat akce, které modifikují vlastnosti entit. Pokud entita nějakou vlastností nedisponuje, je vrácena vlastnost se základní hodnotou. Například pokud provádíme akci útok magickou ohnivou koulí a entita nemá žádnou z vlastností odolnost proti magii či odolnost proti ohni, výsledný útok akce nebude nijak modifikován.

2.8.2 Dovednosti

V originální hře mají šampioni pevně definované dovednosti (viz sekce 2), jako tomu v případě vlastností. I v tomto bodě se nový engine snaží o větší dynamičnost. Opět platí, že každá entita nemusí mít všechny dovednosti. Pokud je po entitě vyžadována dovednost, kterou entita nedisponuje, je její úroveň nulová. V originální hře jsou dva typy dovedností: základní a skryté. Pokud některá akce vylepšuje skrytu dovednost, rozdělí se získané zkušenosti mezi obě dovednosti (viz sekce 2). Tento koncept schopností není v enginu vyžadován a záleží potom na konkrétní implementaci dovednosti. Ta také specifikuje množství zkušeností nutné pro získání nových úrovní. Implementace dovedností pro hru Dungeon Master odpovídá té ve zdrojových kódech hry [9], konkrétně v následujících funkcích v souboru **CHAMPION.C**:

- **F304_apzz_CHAMPION_AddSkillExperience** – přidání zkušeností dané dovednosti.
- **F303_AA09_CHAMPION_GetSkillLevel** – získání úrovně dané dovednosti.

2.8.3 Relace mezi entitami

Další funkcí, kterou oproti originálu engine umožňuje, je definovat pro entity jejich nepřátelé. Každá živá entita má token identifikující skupinu. Entity v této skupině jsou mezi sebou přátelské. Každá entita si může nadefinovat svoje nepřátelské tokeny. Podle vzoru originálu jsou v hře pouze dva relační tokeny – první pro šampiony a druhý pro nepřátelské entity. Nicméně celý tento systém je navržen právě pro stanovení obecnějších vztahů mezi entitami a to může být aplikováno v libovolném rozšíření hry.

2.8.4 Tělo a inventáře

Každá živá entita má definované její části těla. Oproti originálu, kde se řeší pouze části těla šampionů, zde je možné definovat tělo pro každou entitu. Je tak možné pracovat s částmi těla entit, které nemají humanoidní formu. Některé části těla se dají použít jako úložiště předmětů, ty potom mají nadefinováno s jakým typem úložiště jsou kompatibilní. Tato reprezentace principiálně umožňuje podporu následujícího příkladu. Pokud máme lidské a medvědí nohy, tak lidské kalhoty by měli jít nasadit pouze na lidské nohy, nikoliv na medvědí. Naopak medvědí chrániče na nohy půjdou nasadit pouze na nohy medvědí. Takto pokročilejší mechanismy v originální hře nejsou použity, předměty tam mohou sbírat pouze šampioni. Nicméně tyto mechanismy jsou k dispozici při použití tohoto enginu k implementaci jiné hry.

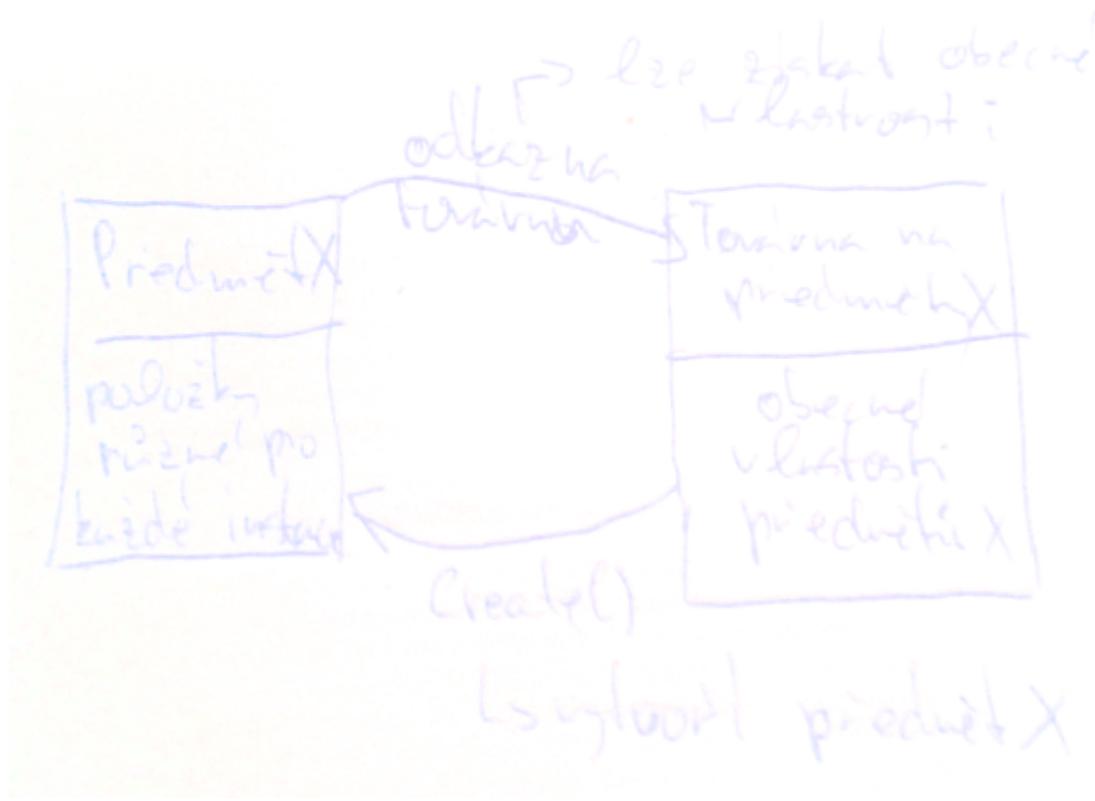
Předměty lze pak dále ukládat do dalších externích úložných prostorů. Mezi taktové úložiště patří například batoh, kapsa, truhla, toulec atd. O velikosti a typu těchto úložišť opět rozhoduje tvůrce konkrétních entit. Avšak stejně jako v originální hře, i tato instance má implementované úložiště pouze pro šampiony.

2.9 Reprezentace předmětů

V herních úrovních jsou různě rozmístěné předměty, které lze sbírat a ukládat si je do inventářů šampionů. Každý takový předmět náleží do nějaké kategorie (zbraň, lektvar, atd. – viz sekce 2.1.3). Všechny tyto kategorie obsahují několik typů předmětů. Pro každý typ předmětu pak existuje jednoznačný globální identifikátor (viz sekce 2.2.2). Identifikátory jsou použité například v přepínačích pro identifikaci typu předmětu, dle kterého se pak rozhodnou, zda se mají aktivovat či nikoliv. Platí tedy, že všechny instance daného typu předmětu mají stejný identifikátor.

Z počátku jsme pro stanovení identifikátorů zvolili stejnou strategii jako tvůrci originální hry, tedy každá instance měla daný číselný identifikátor. Nicméně takto zvolený způsob reprezentace identifikátorů by byl nepřehledný pro případné rozšířitele, jelikož by nemuselo být jasné, které identifikátory už jsou obsazeny a které nikoliv. Dále v počátcích každá instance předmětu měla všechny jeho vlastnosti a to i ty, které byly společné pro všechny instance daného typu (viz sekce 2.2.2).

Pro splnění cíle **C3** této práce bylo zapotřebí reprezentaci předmětů vylepšit. Řešením problému bylo delegování společných vlastností předmětů do zvláštních tříd (viz obr. 2.15), které odpovídají jednotlivým identifikátorům. Tyto třídy navíc slouží jako factories na předměty daného typu. Reference na konkrétní instanci třídy pak slouží jako identifikátor. To ale znamená, že reference na tyto factories jsou potřeba pro vytváření přepínačů. Z toho důvodu je součástí jádra enginu objekt obsahující všechny factories splňující tento vzor (viz sekce 3.1).



Obrázek 2.15: Ilustrace vztahu objektů a jejich factories.

Každá factory na předměty má také následující atributy:

- hmotnost,
- jméno,
- kombo – definuje akce, které lze s předměty provádět,
- definice míst v inventáři, kam lze předmět uložit.

2.10 Reprezentace komb a akcí

Každý typy akce definuje svoji factory, která je uložena v jádře enginu (viz sekce 3.1) – jako je tomu v případě předmětů. Tyto factories obsahují vlastnosti o dané akci (viz sekce 2.2.1), které také určují požadavky pro provedení dané akce. Pomocí factory lze pak vytvořit samotný objekt reprezentující konkrétní akci, kterou lze následně vyvolat určitým směrem (sever, východ, jih, západ). Jak podmínky vytvoření akce, tak její provedení je naimplementováno dle dekomplikovaných zdrojových kódů [9] originální hry. Každá factory na předměty má potom specifikovaný seznam typů akcí, které lze s předměty provádět.

2.11 Reprezentace kouzel

V jádře enginu jsou uloženy všechny symboly, včetně power symbolů (viz sekce 2.2.3). Tyto symboly mají definovány, kolik je třeba many pro jejich vyvolání při

daném power levelu. Každé kouzlo potom definuje svoji factory, která je rovněž uložena v jádře enginu (viz sekce ??) – stejně jako je tomu v případě předmětů a akcí. Objekt reprezentující kouzlo lze pak opět vytvořit pomocí jeho factory. Pro vyvolání kouzla je potom třeba určit směr a entitu, která kouzlo vyvolává. Vyvolávání a provádění kouzel je opět naimplementováno dle dekomplikovaných zdrojových kódů [9] originální hry. Modifikace vlastnosti entity při vyvolávání symbolů kouzla pak zajišťuje speciální manager.

2.12 Builder herních úrovní

Aby bylo možné sestavovat herní úrovně z různých vstupní formátů (viz cíl práce **C4**), bylo nutné tuto část vyčlenit do samostatného objektu, který je pak předán jádru enginu (viz sekce 3.1). Tento objekt budeme dále nazývat builder. Jádro enginu po builderu vyžaduje vytvoření herních úrovní, přičemž mu předá sadu factories – která je rovněž uložena v jádře. Builder by se měl také starat o kešování načtených úrovní, jelikož je engine po něm může požadovat několikrát. Objekt builderu je předán jádru enginu při jeho vytváření, tímto způsobem je možné jádru předat vlastní implementaci builderu.

Tato práce obsahuje pouze jeden builder, který je schopný sestavit herní úrovně z datového objektu popsaném v sekci 2.3. V sekci 2.6 se došlo k závěru, že dlaždice zdí nebudou v tomto enginu existovat. Z toho důvodu jsme se v prvních fázích projektu rozhodli při vytváření herní úrovně procházet pouze dlaždice, které jsou od pozice hráče přístupné. Toho bylo docíleno použitím algoritmu prohledávání do hloubky. Nicméně později se ukázalo, že některé nepřístupné části herní úrovně jsou používané teleporty nebo přepínači. Z toho důvodu se od tohoto způsobu procházení dat dlaždic upustilo a nyní se procházejí postupně všechny dlaždice.

Pro transformování dat na objekty srozumitelné enginu se používají ještě další pomocné subbuildery (viz obr. 2.16). Samotné sestavování herní úrovně potom probíhá následovně. V cyklu jsou vytvářeny všechny dlaždice – přitom se za pomocí factories z jádra enginu provádí vytváření veškerých objektů, které dlaždice obsahuje.



Obrázek 2.16: Ilustrace struktury builderu.

2.13 Inicializace objektů

Tento projekt si klade za cíl (viz cíl práce **C3**) vytvořit co nejlepší objektový návrh. Jedním z faktorů, který by měl takový návrh zahrnovat je, že veřejné budou jen nezbytně nutné položky tříd. Na to navazuje problém s inicializací takových tříd. Například přepínače potřebují referenci na cílovou dlaždici, ale zároveň jsou obsahem dlaždic, je tedy třeba oddělit fázi inicializace dlaždic od inicializace přepínačů. Nicméně potom nelze přepínače předat do dlaždic v konstruktoru a tím pádem je pro tuto akci třeba vytvořit veřejnou funkci nebo vlastnost, která tak provede. Tato funkce pak narušuje náš cíl, protože tuto funkci může kdokoliv zvoleť později, kdy už to není validní. Z toho důvodu jsme přišli s konceptem tzv. inicializátorů.

Inicializátor je datová třída obsahující vlastnosti, které by běžně byly parametry konstruktoru inicializovaného objektu. Místo těchto parametrů je do konstruktoru předán inicializátor, který má také události vyvolané při inicializaci a při dokončení inicializace. Třída beroucí inicializátor jako parametr konstruktoru se zaregistrouje na tyto události a tím pádem není nutná žádná přebytečná položka ve třídě pro inicializátor. Události jsou pak volané skrze metodu v inicializátoru. Inicializátorem inicializovaná třída si z něj nakopíruje parametry a odhlásí událost. Od té chvíle už není možné třídu modifikovat. Data inicializátoru se tedy mohou postupně naplňovat a po jejich naplnění se inicializace provede zavolením metody na inicializátoru. S využitím asynchronních metod je navíc možné vy-

volat inicializaci prvku (např. přepínačů, které potřebují reference na dlaždice) již při vytváření dlaždic. Což vede k přehlednějšímu kódu a celkově k inicializaci cyklických struktur bez nutnosti přebytečných veřejných položek.

S využitím dědičnosti inicializátorů je možné nasimulovat volání rutin konstruktorů, tak jak je v C# běžné. Na každé úrovní dědičnosti se využijí některé vlastnosti inicializátoru. Nechť inicializátor B je potomek inicializátoru A v hierarchii dědičnosti. Nechť C a D jsou třídy, které chceme inicializovat a zároveň D je potomkem C. Také platí, že konstruktor třídy D vyžaduje inicializátor třídy B a konstruktor třídy C vyžaduje inicializátor A. Potom inicializátor B můžeme použít pro oba konstruktory tříd C i D. Přičemž na každé úrovni dědičnosti inicializátoru může být zvláštní inicializaci oznamující událost. Pokud tedy inicializátor volá inicializační události ve správném pořadí, může to nasimulovat pořadí inicializace běžné u konstruktorů. Při inicializaci dlaždic je právě tento způsob používán. Celou situaci znázorňuje obrázek 2.17.



Obrázek 2.17: Ilustrace použití inicializátoru.

2.14 Renderování a interakce

Dalším cílem (viz cíl práce C5) tohoto projektu je oddělit v enginu zobrazovací vrstvu, tak aby ji bylo možné jednoduše nahradit za jinou. Se zobrazovací vrstvou nicméně úzce souvisí, jakým způsobem bude prováděna interakce s objekty, proto je v této vrstvě zahrnuta i ona. Každý objekt, který to vyžaduje má vytvořený vlastní renderer-interactor na míru. Jak renderer vypadá uvnitř je zpravidla na programátorovi. Nicméně obvyklý přístup je takový, že renderer má referenci na konkrétní renderovaný objekt. Podle jeho zpravidla readonly vlastnosti potom určuje chování vykreslování nebo interakce. Pokud se jedná o renderery pro sta-

tické objekty (např. dlaždice), tak se zpravidla pozicování určuje relativně vůči rodiči. Zobrazovací vrstva v tomto enginu nemá stejný grafický výstup jako originální hra, nicméně nic nebrání tomu napsat tuto vrstvu tak, aby jí odpovídala.

3. Vývojová dokumentace

Celá práce je rozdělena do dvou projektů Visual Studia. První z nich je **DungeonMasterParser**, který zajišťuje rozparsování herních dat (viz sekce 2.3). Celý tento projekt se skládá z datových tříd odpovídající formátu souboru **DUNGEON.DAT** resp. **GRAPHICS.DAT** (viz sekce 2.1 resp. 2.2), které pak sestaví dohromady třída **DungeonParser** za vzniku objektu třídy **DungeonData**. Při vytváření instance třídy **DungeonData** se pak provede načtení dat odpovídající soubor **DUNGEON.DAT**. Kvůli tomu tento projekt refereuje knihovnu HTML Agility Pack [11].

Druhý projekt **DungeonMasterEngine** obsahuje už samotný engine, kterému se bude hlavně věnovat tato kapitola. Projekt s enginem pak referencuje projekt **DungeonMasterParser** a framework MonoGame [1]. Data týkající se herních úrovní jsou pak ve složce **/Data** a data jako textury, fonty nebo modely jsou ve složce **/Content** jak je tomu MonoGame zvykem. Oba projekty se pak snaží dodržovat následující konvence:

- Každá třída nebo rozhraní jsou ve zvláštním souboru korespondujícím jejich názvu. (s výjimkou generických tříd se stejným názvem jako negenerické)
- Skupiny třídy, které patří logicky k sobě se separují do zvláštních složek.
- Jednotlivé složky odpovídají jemným prostorům.
- Názvy rozhraní začínají písmenem **I**.
- Třídy rozšiřující abstraktní třídy mají část jejich názvu.
- K abstraktním třídám většinou existuje odpovídající rozhraní se stejnými položkami.

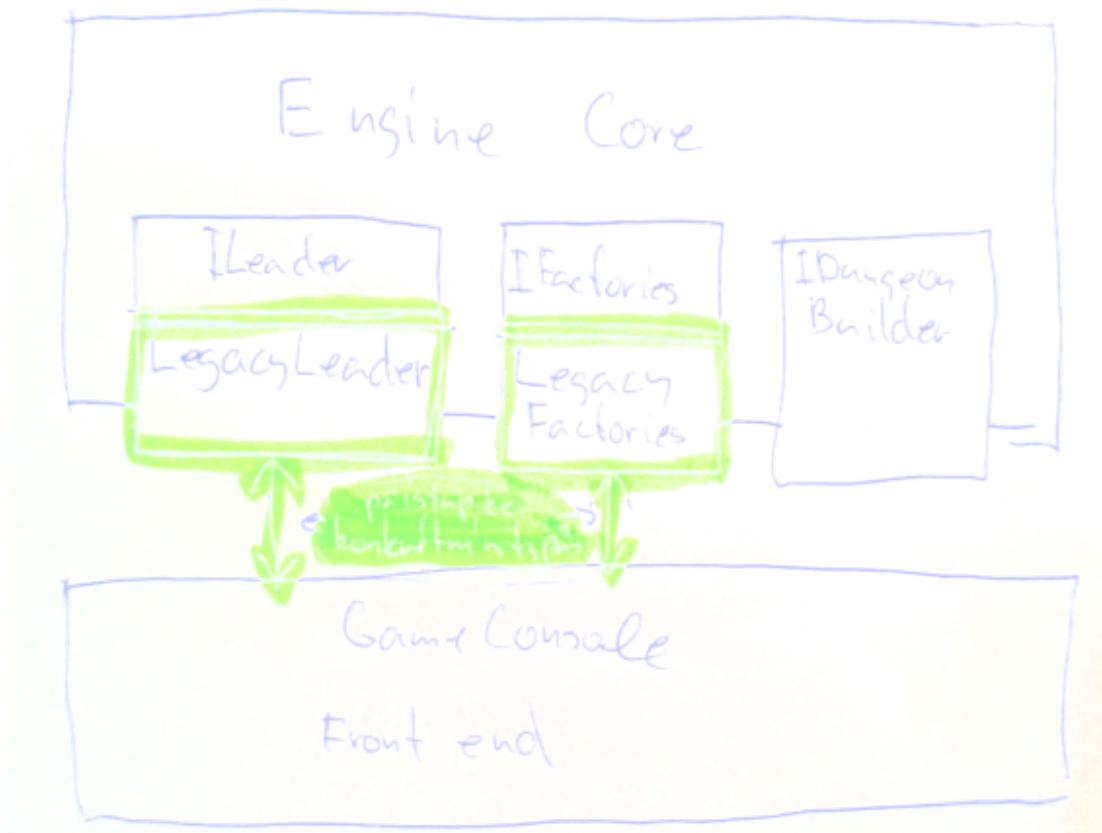
Následující sekce této kapitoly popisuje třídy reprezentující jednotlivé části enginu a vztahy mezi nimi.

3.1 Jádro enginu – **DungeonBase**

Jádro enginu reprezentuje třída **DungeonBase** a skládá ze tří důležitých komponent, které do něj lze předat při jeho vytváření. Za těmito komponentami stojí následující rozhraní:

- **ILeader** – reprezentuje hráče a jeho skupinu, kterou lze skrze něj ovládat.
- **IFactories** – obsahuje factories pro tvorbu předmětů, nepřátelských entit, kouzel, akcí a definuje úrovně osvětlení.
- **IDungeonBuilder** – vytváří pro engine herní úrovně.

Pro reimplementaci hry Dungeon Master jsou pak použity následující třídy **LegacyLeader**, **LegacyFactories**, **LegacyMapBuilder**. Konkrétní typy implementací rozhraní **ILeader** a **IDungeonBase** lze předat jádru jako typové parametry. Případný front end potom může skrze instanci jádra pracovat přímo s konkrétními typy (viz obr. 3.1). Tento projekt neobsahuje žádné uživatelské rozhraní, namísto něho je pro udávání některých příkazu použita konzole **GameConsole**.



Obrázek 3.1: Ilustrace komunikace frondendu s jádrem enginu.

3.1.1 Asynchronní funkce

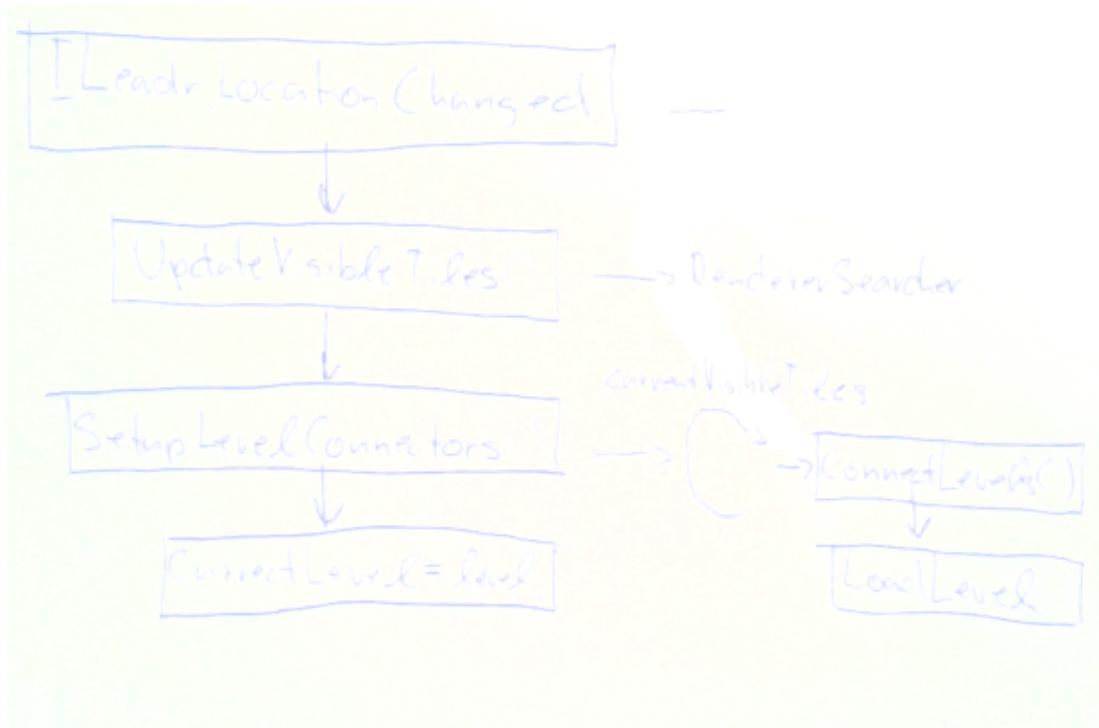
Samotné jádro enginu je vytvořeno a aktualizováno ze třídy **Engine**, která je potomkem třídy **Game** zajišťující herní smyčku. V řadě částí enginu se využívají asynchronní funkce ke zjednodušení implementace. Důležité je poznamenat, že tyto funkce předpokládají, že budou vykonávané v jednom vlákně - jinak může velmi jednoduše dojít k race condition. Z toho důvodu je nutné přepnout chování asynchronních funkcí tak, aby se prováděli v jednom vlákně. To lze udělat nastavením vhodného synchronizačního kontextu v hlavním vláknu skrze funkci **SynchronizationContext.SetSynchronizationContext**. Tato funkce má jako parametr instanci třídy **SynchronizationContext**. Správnou implementaci synchronizačního kontextu – tak aby se asynchronní funkce prováděly v jednom vlákně – zajišťuje třída **GameSynchronizationContext**. Tento synchronizační kontext je nastaven v konstruktoru třídy **Engine** a provádění asynchronních funkcí potom probíhá ve funkci **Engine.Update**. K synchronizaci je použita fronta, kterou je nutné synchronizovat pomocí zámků. V každém volání této

funkce je vyprázdněna celá fronta asynchronních funkcí, proto je třeba dát pozor na její přehlcení, které může v konečném důsledku vést až k neresponzivnosti aplikace.

3.1.2 Správa herních úrovní a dlaždic

Při inicializaci jádra je na objektu hráče zaregistrována událost **ILeader.LocationChanged**, která je vyvolána po přesunu hráče na novou dlaždici. Zaregistrování resp. odregistrování události se provede po přiřazení hodnoty do vlastnosti **Leader**. V obsluze této události se pak provede aktualizace viditelných dlaždic pomocí funkce **UpdateVisibleTiles** (viz obr. 3.2). K nalezení viditelných dlaždic tato funkce používá třídu **RendererSearcher**, která je potomkem třídy **BreathFirstSearch**. Nalezené dlaždice se uloží do protected proměnné **currentVisibleTiles**. V dalším kroku je zavolána funkce **SetupLevelConnectors**, jenž projde všechny viditelné dlaždice vedoucího do jiných úrovní. Pro každou tuto dlaždici dojde pak voláním funkce **ConnectLevels** k propojení dané dlaždice s další úrovní. Poslední akcí je přiřazení úrovně, v které se hráč vyskytuje, do proměnné **CurrentLevel**.

Funkce **ConnectLevels** načte danou úroveň pomocí funkce **LoadLevel**, která tak učiní skrze builder. Načtená úroveň je pak uložena do kolekce **ActiveLevels**, ve které jsou vždy tři poslední úrovně. Dlaždice, které propojují herní úrovně, by měli implementovat rozhraní **ILevelConnector**. Funkce **ConnectLevels** pak nastaví cílovou dlaždici do vlastnosti **ILevelConnector.NextLevelEnter**. Dlaždice implementující toto rozhraní by měla zajistit její propojení s dlaždicí **ILevelConnector.NextLevelEnter**. Všechny funkce zmíněné v této sekci jsou virtuální, a proto je možné jejich chování upravit přetížením.



Obrázek 3.2: Ilustrace funkce jádra.

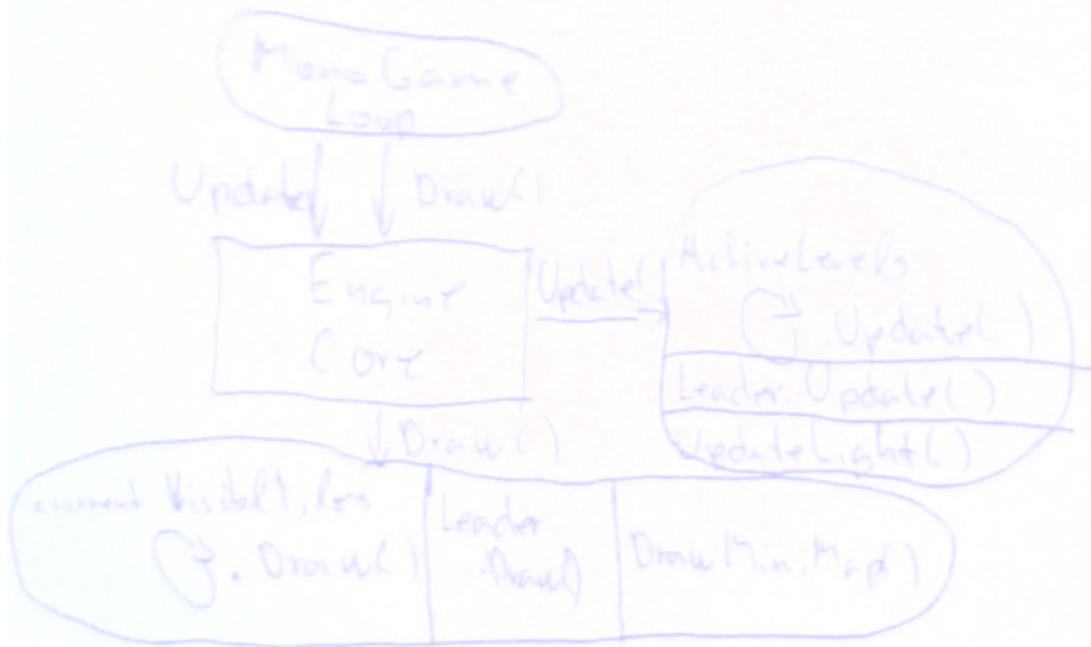
3.1.3 Aktualizace a rendering

MonoGame zajišťuje herní smyčku, skrze kterou je pak třeba na enginu volat funkce **Update** a **Draw**.

V každém zavolání funkce **Update** se provede aktualizace všech herních úrovní v kolekci **ActiveLevels**. Herní úroveň reprezentuje typ **DungeonLevel**, který má v sobě uložené dlaždice, obtížnost a objekty vyžadující aktualizaci – které se u něj zaregistrovaly (viz sekce 3.1). Funkce **DungeonLevel.Update** pak zavolá na všech zaregistrovaných objektech funkci **IUpdate.Update**. Všechny tyto objekty musí implementovat rozhraní **IUpdate** a musí se sami starat o odregistrování z kolekce, když už aktualizaci nevyžadují. Dále je volána funkce **ILeader.Update** na hráče. Při každém provedení aktualizace se ještě zavolá virtuální funkce **UpdateLight**. Proces reprezentuje obrázek 3.3.

Funkce **UpdateLight** přiřadí do proměné **Light** hodnotu určující úroveň světla, která je v setteru vlastnosti přepočítána na vzdálenost, po kterou se vykreslují dlaždice. Getter pak vrací již samotnou vzdálenost, nikoliv hodnotu světla. Hodnota světla je 0 až 5, kde 0 reprezentuje největší osvětlení. Takto je to nastaveno z důvodu kompatibility se zdrojovými kódy originální hry [9]. Funkce projde všechny šampiony v kolekci **ILeader.PartyGroup** a nalezne všechny objekty implementující rozhraní **ILightSource**. Na základě hodnot **ILightSource.LightPower** vypočte výslednou hodnotu osvětlení. Tento výpočet lze změnit přetížením funkce.

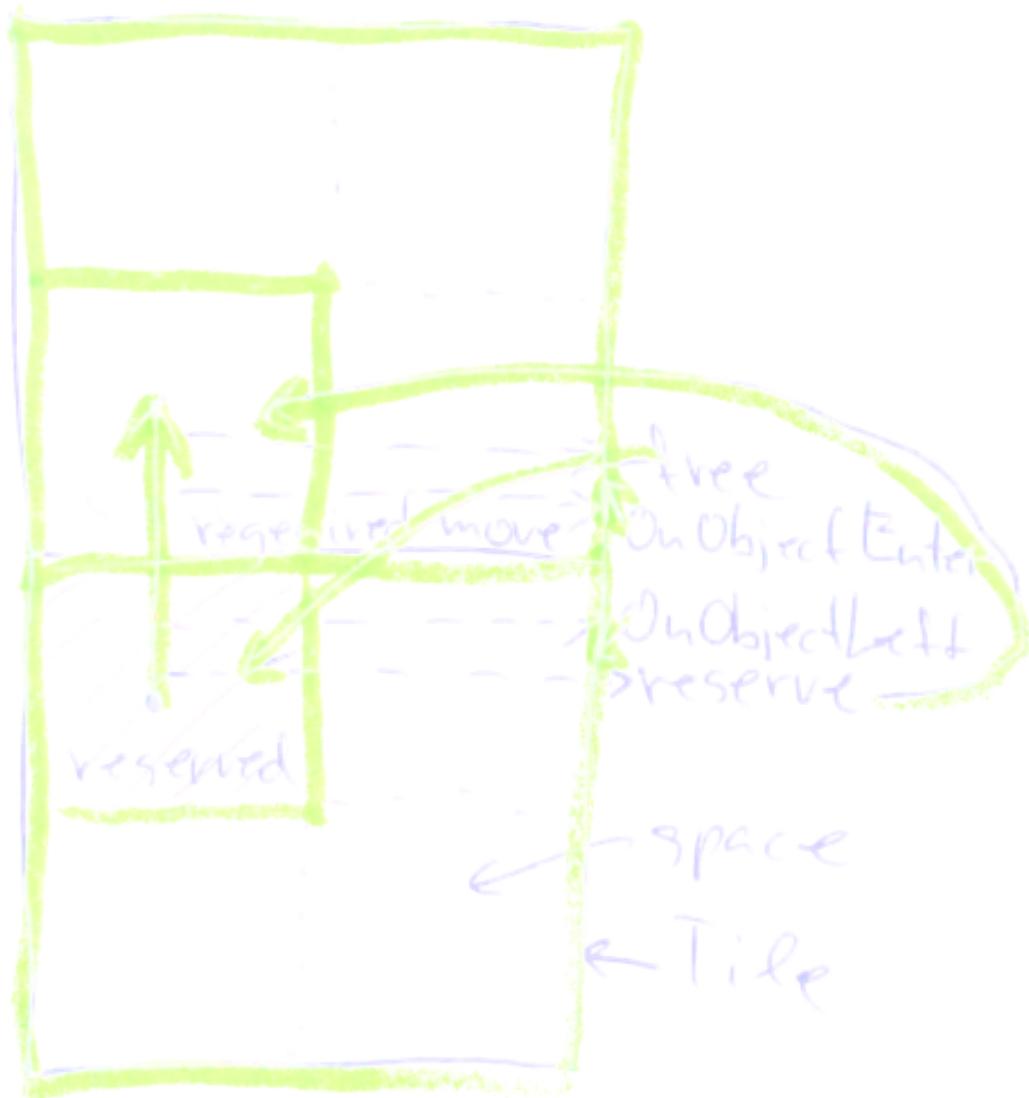
V každém zavolání funkce **Draw** se potom provede vykreslení všech dlaždic v kolekci **currentVisibleTiles** skrze jejich renderer (viz sekce 2.14). Dále jsou zavolány funkce **ILeader.Draw** pro vykreslení objektů ve správě hráče a virtuální funkce **DrawMiniMap**, která provede vykreslení minimapy.



Obrázek 3.3: Ilustrace aktualizace a vykreslování z pohledu jádra.

3.2 Dlaždice – ITile

Nejobecnější strukturu dlaždice definuje rozhraní **ITile**. Každá dlaždice musí mít definované sousedy, to zajišťuje položka **Neighbors** typu **TileNeighbors**. Tento typ implementuje rozhraní **INeighbors**, které vyžaduje pouze enumerátor dvojic se směrem a dlaždicí. Směr reprezentuje struktura **MapDirection**, kde se jednotlivé směry dají získat skrze její statické členy. Třída **TileNeighbor** potom implementuje ještě pomocné funkce a vlastnosti pro jednodušší práci se sousedy. Pomocí této vlastnosti se objekty mohou pohybovat mezi dlaždicemi (viz obr. 3.4). Po vstupu objektu na dlaždici je třeba zavolat na předchozí dlaždici funkci **OnObjectLeft** a na nové dlaždici funkci **OnObjectEntered**. Díky tomu může dlaždice reagovat na vstup resp. odchod objektů – což je použité například pro přepínače na podlaze. Další vlastností dlaždice je **LayoutManager**, který musí používat entity při pohybu po dlaždicích a mezi nimi. **LayoutManager** je třída poskytující správu prostoru na dlaždici, tak aby každý prostor zaujímal pouze jedna entita (viz sekce 3.4.4). Před pohybem entity je tedy ještě třeba zarezervovat skrze **LayoutManager** cílový prostor na dlaždici a po dokončení pohybu je naopak třeba uvolnit prostor předchozí. Poslední položkou potřebnou pro pohyb mezi dlaždicemi je vlastnost **IsAccessible**, která určuje, zda lze na dlaždici vstoupit.



Obrázek 3.4: Ilustrace pohybu mezi prostory a dlaždicemi.

Pro implementaci funkce přepínačů je třeba zajistit komunikaci mezi dlaždicemi (viz sekce 2.7.3). Za tímto účelem obsahuje dlaždice vlastnosti pro změnu jejího stavu. Buď lze obsah dlaždice aktivovat přímo pomocí funkce **ActivateTileContent** resp. **DeactivateTileContent** nebo pomocí zaslání zpráv, k jejichž příjetí slouží funkce **AcceptMessageBase**. Konkrétní stav je obsahuje pak vlastnost **ContentActivated**.

V poslední řadě rozhraní reprezentující dlaždice obsahuje prvky:

- **Position** – absolutní pozice od počátku souřadnic,
- **GridPosition** – souřadnice dlaždice na mřížce,
- **Drawables** – do této kolekce se mohou zaregistrovat objekty implementující rozhraní **IRenderable**, které se nechťejí registrovat u dlaždice pomocí funkcí pro vstup a výstup a zároveň se potřebují renderovat,
- **ObjectEntered** resp. **ObjectLeft** – události vyvolané při vstupu resp. odchodu objektu,

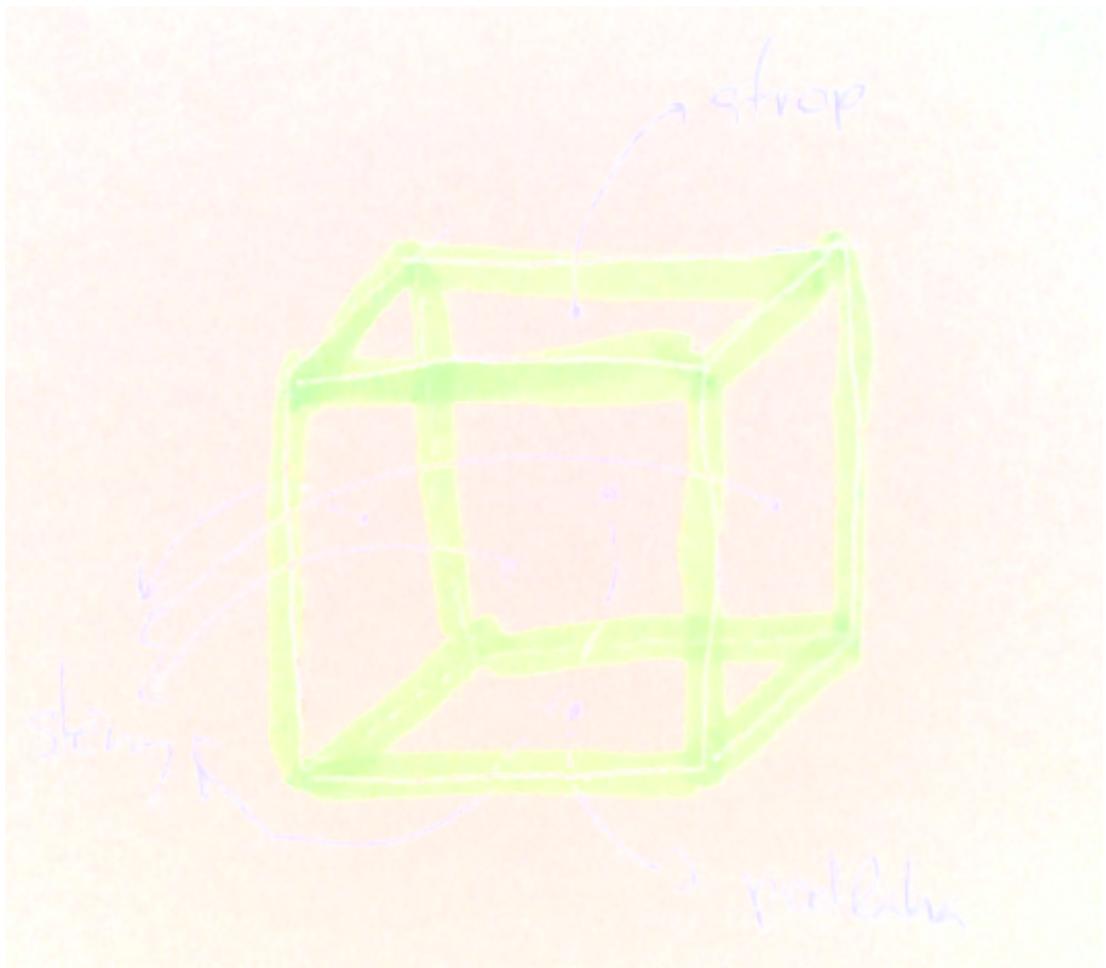
- **Level** – reference na level, ve kterém se dlaždice nachází,
- **IsInitialized** – určuje, zda je dlaždice plně inicializovaná (viz sekce 2.13),
- **Initialized** – tato událost je dlaždicí vyvolána pouze jednou a to v momentě, kdy jsou všechny její části inicializované. Tuto událost využívají živé entity k určení doby jejich obživnutí.

3.2.1 Inicializace dlaždic

Jak již bylo zmíněno v analýze (viz sekce 2.13), k inicializaci dlaždic jsou použity tzv. inicializátory. Inicializátor obsahuje vlastnosti, které by normálně byly předány jako parametry v konstruktoru. Ovšem v moment předávaní inicializátoru do konstruktoru, inicializátor ještě nemusí mít naplněné všechny hodnoty. Namísto toho má události **Initializing** resp. **Initialized**, které jsou vyvolány při resp. po inicializaci. Na tuto událost je třeba zaregistrovat inicializační funkci, která zkopiřuje data z inicializátoru do samotného objektu. Pro každou úroveň hierarchie dědičnosti jsou určeny zvláštní vlastnosti a zvláštní inicializační události. Rodičovské události inicializátoru jsou vždy po zdědění rodičovského inicializátoru zakryty novými inicializačními událostmi pro danou úroveň dědičnosti. Tento způsob je použit pouze pro inicializaci dlaždic. Pro jiné objekty, může inicializátor sloužit pouze jako objekt udržující parametry, které jsou hned v konstruktoru inicializované.

3.2.2 Implementace dlaždic

Částečnou implementaci rozhraní **ITile** zajišťuje abstraktní třída **Tile**, která definuje virtuálně **LayoutManager**. Dále abstraktně deklaruje kolekci **SubItems**, která obsahuje všechny objekty na dlaždici. Objekty, které implementují rozhraní **IRenderable**, jsou potom skrze tuto kolekci vykreslovány. Dlaždice také má tzv. strany, a jsou to buď stěny, strop nebo podlaha (viz obr. 3.5). Pro tyto strany pak abstraktně deklaruje kolekci **Sides**, jejímž stěnám pak přeposílá případné zprávy, které přišly na tuto dlaždici skrze metodu **AcceptMessageBase**. Dále implementuje také funkce **OnObjectEntered** resp. **OnObjectLeft** tak, že vyvolá jejich odpovídající událost. Proto je pří případném přetížení těchto funkcí v potomkovi nutné zavolat implementaci rodiče. Všechny předchozí popsané funkce je možné přetěžovat v potomkovi, a tak přizpůsobit prováděné akce. V poslední řadě se tento inicializátor stará o inicializaci pozic na mřížce, úrovně a sousedů skrze **TileInitializer**.



Obrázek 3.5: 3D ilustrace dlaždice a jejích stran.

Přímý generický potomek předchozí třídy `Tile<TMessage>`, kde `TMessage` je alespoň `Message`, poskytuje možnost přijímat v potomcích zprávy vlastního typu (viz sekce P3). V případných potomcích této třídy lze naimplementovat funkci `AcceptMessage`, která přijímá zprávy typu `TMessage`. Tato třída přetěžuje a zároveň uzavírá metodu `AcceptMessageBase` tak, že deleguje zprávy typu `TMessage` do metody `AcceptMessage`. Naopak základní implementace metody `AcceptMessage` je zavolání právě rodičovské implementace `AcceptMessageBase`. Při případném přetěžování této funkce je třeba rozhodnout, zda je nutné volat implementaci rodiče. Všechny dlaždice, které dědí ze třídy `Tile<TMessage>`, vytváří dva potomky. První z nich je pro případné rozšiřitele, a proto ponechává typový parametr, druhý pak specifikuje typový parametr na obecný typ `Message`.

Dlaždice podlahy – `FloorTile<TMessage>`

Tato dlaždice implementuje funkci všech jejích stran (viz obr. 3.5). Na stranách, kde nemá dlaždice sousedy jsou přidány stěny, které mají podporu pro přepínače (viz sekce 2.1) a dekorace (viz sekce 2.7). Dále je přidána strana na místo stropu a podlahy. Na stranu s podlahou pak může obsahovat přepínač a je na ni možné odkládat či z ni odebírat předměty. Dlaždice také deleguje veškeré vstupy resp. odchody objektů do samotné strany podlahy typu `FloorSide` (viz sekce 3.2.3) a to kvůli podpoře odkládání předmětů na zem.

Další implementace dlaždic

Další dlaždice převážně využívají dědičností dlaždici podlahy. Výjimkou jsou například schody, které jsou zároveň příkladem dlaždice implementující rozhraní **ILevelConnector**. Engine při načtení levelu nastaví vlastnost **ILevelConnector.NextLevelEnter** na odpovídající dlaždici (viz sekce 3.1.2). Při změně předchozí vlastnosti provede dlaždice nastavení sousedů vedoucích do dalšího levelu a používá k tomu vlastní implementaci sousedů přetížením vlastnosti **Neighbors**. Tato možnost je použita ještě u jámy, kvůli propadu do nižšího levelu. Rozhraní spojující leveley je ještě použito u teleportu, kde při vstupu správného objektu na teleportační dlaždici dojde k teleportaci objektu na dlaždici **ILevelConnector.NextLevelEnter**. Další dlaždici jsou dveře, které opět dědí z podlahy a navíc implementují rozhraní **IHasEntity**. Toto rozhraní obsahuje vlastnost typu **Entity**, která reprezentuje neživou entitu na dlaždici. S touto entitou můžou pak pracovat akce, kterými lze dveře rozbít. Poslední dlaždici reprezentuje třída **LogicTile**, na kterou nelze vstoupit a slouží pouze jako podpora pro přepínače (viz sekce 3.3).

3.2.3 Strany dlaždic – **ITileSide**

Jak již bylo naznačeno (viz obr. 3.5), dlaždice mohou mít strany, které jsou buď stěny, podlaha nebo strop. Každou tuto stranu reprezentuje samostatný objekt, který má svůj renderer-interactor (viz sekce 2.14). Toto rozhraní vyžaduje pouze položku pro renderer a funkci pro přijímání základních zpráv **Message**. Tato funkce je zde z kompatibilních důvodu s originální hrou, kdy texty na zdech mohou být témito zprávami skryty či zobrazeny.

Následuje seznam jednotlivých implementací:

- **TileSide** – reprezentuje pouze holou stěnu bez žádných funkcí, která obsahuje pouze vlastnost typu **MapDirection** určující její pozici.
- **ActuatorTileRenderer** – přímý potomkem předchozí strany, který navíc obsahuje přepínač a renderer, jenž se navíc stará o jeho vykreslování a interakci.
- **TextTileSide** – opět přímý potomek první strany, jejíž renderer navíc zobrazuje na zdi text. Viditelnost textu lze změnit zasláním zprávy této straně.
- **FloorTileSide** – tato strana obsahuje čtyři úložiště, na které může hráč pokládat předměty. Je to jedno z míst, kde je potřeba komunikovat s rodičovskou dlaždicí, k tomu slouží událost **SubItemsChanged**, která se vyvolá vždy, když byl nějaký předmět odebrán nebo přidán. Pomocí enumerátoru tohoto objektu je potom možné získat obsažené objekty. Předměty lze pak na podlahu přidávat dvěma způsoby, přičemž oba pomocí události informují rodičovskou dlaždici o změně jejího obsahu. První možností je položení předmětu přímo z ruky hráče pomocí renderer-interactoru. Druhý způsob je zavolání metody **OnObjectEntered** resp. **OnObjectLeft** objevující se například při teleportaci nebo hodu předmětu. Celou situaci zobrazuje obrázek 3.6.

- **ActuatorFloorTileSide** – přímý potomek předchozí strany podlahy, který navíc přidává na podlahu nášlapný přepínač.



Obrázek 3.6: Ilustrace komunikace podlahy s její dlaždicí při přidávání předmětu.

3.3 Přepínače – **IActuatorX**

Za přepínači stojí rozhraní **IActuatorX**, které vyžaduje pouze API pro příjem zpráv typu **Message** – pro dodržení kompatibility s původní hrou. Toto rozhraní pak implementuje třída **ActuatorX**, která reprezentuje přepínače používající pro jejich funkci senzory z originální hry (viz sekce 2.7). Kromě toho dále existují implementace přepínačů, které jsou provedeny jiným způsobem.

3.3.1 Implementace obecných přepínačů

Tato sekce pojednává o přepínačích, které ke své funkci nepoužívají senzory. V tomto enginu jsou použity pro dekorace, které provádějí nějakou funkci. K těmto dekoracím v originální hře nenáleží žádné senzory. Studií dekomplikovaného kódu

originální hry [9] se ukázalo, že některé akce jsou fixovány na konkrétní typy dekorací. Například pro dekorace s výklenky je naimplementována možnost vkládání předmětů. Jelikož tyto dekorace mohou být i součástí senzorů, bylo zapotřebí, aby objekty reprezentující dekorace implementovali rozhraní **IActuatorX**, a tak bylo možné naimplementovat funkce pro jejich interakci.

U těchto přepínačů lze zvolit jejich vnitřní formát. Existující implementace jsou:

- **DecorationItem** – stará se pouze o rendering dekorace, který zajišťuje třída **DecorationRenderer**.
- **Alcove** – reprezentuje dekoraci výklenku, o jehož rendering a interakci se stará třída **AlcoveRender**.
- **ViAltairAlcove** – potomek třídy **Alcove**, který se navíc stará o reinkarnaci šampionů.

3.3.2 Implementace přepínačů se senzory

Tyto přepínače se skládají z řady senzorů, které mohou provádět následující akce:

- změna stavu vzdálené dlaždice,
- zarovnání sekvencí senzorů přepínače (viz obr. ??),
- přidání zkušeností hráči.

Při pokusu o aktivaci přepínače dojde postupně, k aktivaci všech jeho senzorů. Senzory pak mohou být aktivovány následujícími způsoby:

- kliknutím myši na dekoraci senzoru, pokud je senzor na zdi,
- vstupem resp. odchodem hráčovi skupiny na dlaždici resp. z dlaždice,
- zprávou vyslanou jiným senzorem.

Více viz sekce 2.7.

Implementace senzorů

Každý typ senzoru má jinou podmínu aktivace. Senzory se dělí na senzory použité na podlaze, zdi a na speciální tzv. logické senzory.

Hlavní třídou reprezentující senzor je abstraktní třída **SensorX**. Tato třída obsahuje společné funkce, které jsou využívány jejími potomky. Dále také definiuje vlastnosti, které lze rozdělit do tří skupin. Tyto vlastnosti jsou inicializovány inicializátorem (viz sekce 2.13), který slouží pouze jako datový nosič. Jde o inicializátor, který reprezentuje třída **SensorInitializerX**. Pro inicializaci potomků jsou používány poděděné implementace tohoto inicializátoru.

Vlastnosti senzorů třídy **SensorX**:

- Obecné vlastnosti:

- **Delay** – opoždění akce v milisekundách, po kterém se provede akce senzoru.
- **LocalEffect** – informace, zda-li je akce určena lokálně pro rodičovskou dlaždici nebo pro vzdálenou.
- **OnceOnly** – informace, zda-li je senzor možné aktivovat pouze jednou.
- **RevertEffect** – u většiny přepínačů obrátí efekt výsledné akce, nicméně přesný význam si stanový vždy konkrétní senzor.
- **Audiable** – informace určující, zda po aktivaci dojde k přehrání zvuku. (v tomto enginu není použit)
- **GraphicsBase** – reprezentuje dekoraci a zároveň přepínač obecného typu implementující rozhraní **IActuatorX**. Pokus o aktivaci tohoto přepínače se provádí pouze pokud je zobrazena dekorace (viz obr. 3.7).
- Vlastnosti pro lokální akci:
 - **Rotate** – informace určující, zda-li se má seznam senzorů při aktivaci zarotovat (viz obr. ??).
 - **ExperienceGain** – informace určující, zda-li se mají hráči po aktivaci přidat zkušenosti.
- Vlastnosti pro vzdálenou akci:
 - **Effect** – určuje jaká zpráva je odeslaná na cílovou dlaždici. (Akce zprávy může být obrácená pokud je nastavena vlastnost **RevertEffect**). Hodnoty efektu jsou následující:
 - ◊ Aktivace
 - ◊ Deaktivace
 - ◊ Přepnutí stavu
 - ◊ Drž – tento stav nelze odeslat ve zprávě a je na senzoru, aby ho interpretoval pomocí aktivace či deaktivace.
 - **Specifier** – směr použitý v odeslané zprávě, který může být interpretován jako číslo (lze získat pomocí **MapDirection.Index**).
 - **TargetTile** – reference na cílovou dlaždici.

Kromě předchozích vlastností obsahuje tato třída funkce pro vykonání akcí senzorů. Pokud je efekt senzoru vzdálený, funkce **TriggerEffect** odešle zprávu na danou cílovou dlaždici. Jinak provede akci pro lokální efekt, který lze také vykonat přímo zavoláním funkce **TriggerLocalEffect**. Tyto funkce lze použít pouze v potomcích.

Následující třídy jsou přímými potomky třídy **SensorX**:

1. **FloorSensor** - tyto senzory lze vložit pouze do přepínačů, které mohou být na podlaze.
2. **WallSensor** - tyto senzory lze vložit pouze do přepínačů, které mohou být na stěně.

3. **LogicGateSensor** resp. **CounterSensor** - pro tyto senzory existuje speciální dlaždice **LogicGateTile**

Senzory z bodu 1 a 2 mají funkci **TryTrigger**, která se pokusí senzor aktivovat a pokud uspěje, stará se o provedení efektu. Zda-li se senzor aktivuje stanoví abstraktní funkce **TryInteract**, kterou implementují potomci. Pro každou variantu senzoru mají funkce odlišné parametry. Pokud je třeba modifikovat způsob provedení výsledného efektu, je možné funkce **TryTrigger** přetížit.

Kromě předchozích senzorů ještě existují tzv. logické senzory (viz 1). První z nich **LogicGateSensor** funguje jako logické hradlo, které má k dispozici osm bitů. Čtyři z nich jsou nastaveny při designu na hodnoty a ostatní na nuly. S druhou čtvericí lze pak manipulovat pomocí zpráv. Index směru zprávy určuje kolikátý bit se má ovlivnit. A tento bit je pak ovlivněn akcí zprávy. Pokud jsou obě čtverice stejné, je vyvolán efekt senzoru. Dalším senzorem je **CounterSensor**, který má dané číslo. Aktivační zprávy toto číslo poté navýšují a deaktivace zprávy ho snižují. Pokud je číslo nula, je vyvolán efekt.

Implementace přepínačů používající senzory

Rodiči těchto přepínačů je třída **Actuator**. Tato třída poskytuje funkci pro zarotování senzory přepínače, která lze vyvolat pouze z potomků. Zda-li se mají senzory zarotovat se určuje nastavením vlastnosti **Rotate**, která se po jeho provedení nastaví zpět na false. Samotné zarotování se pak provede zavolením funkce **ProcessRotationEffect**.

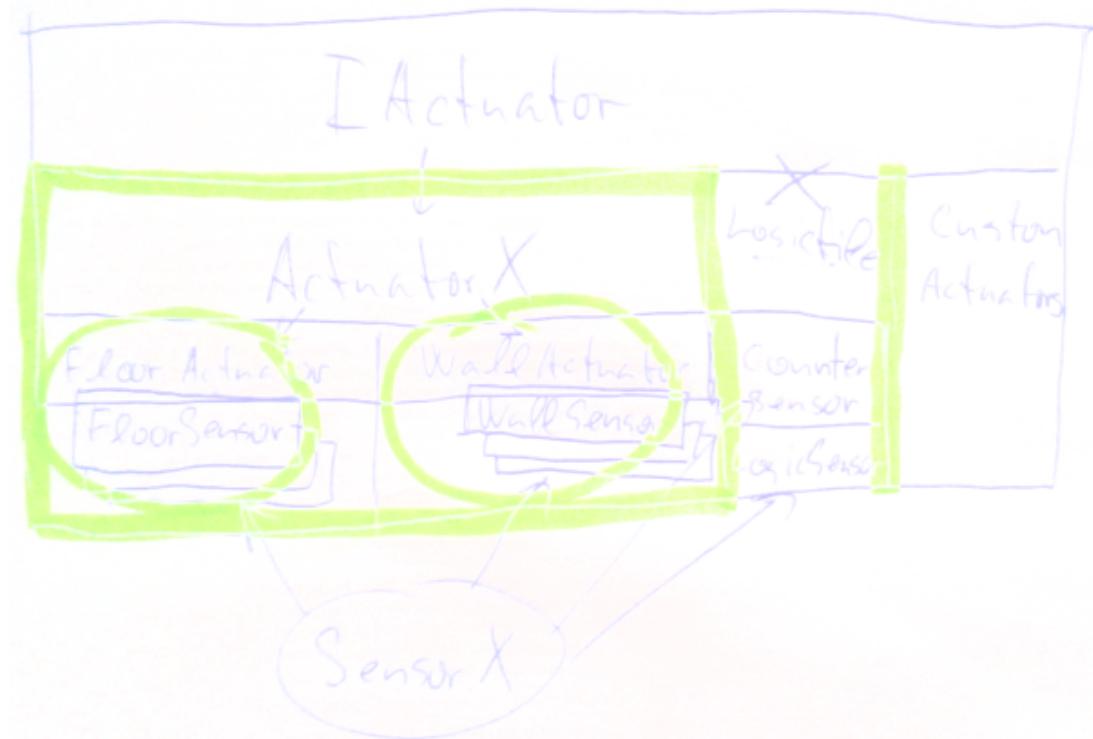
Prvním potomkem předchozí třídy je třída **FloorActuator**, která může obsahovat pouze senzory určené na podlahu tj. **FloorSensor**. Pokus o její aktivaci se provádí funkcí **Trigger**. Jako první parametr se jí předává objekt, který vstupuje resp. odchází z dlaždice, dále seznam všech objektů na dlaždici a naposled informace, zda objekt vstupuje či odchází. Pro přepínač je vytvořena speciální podlaha **ActuatorFloorSide**, která se stará o volání funkce **Trigger**.

Dalším potomkem je třída **WallActuator**, která může obdobně obsahovat pouze senzory určené na zeď tj. **WallSensor**. Pokus o její aktivaci se provádí opět funkcí **Trigger**, která má jediný parametr typu **ILeader**. Pro přepínač je určena strana **ActuatorWallTileSide**. Funkce **Trigger** je pak volána skrze renderer dané strany. Tato se pak pokusí postupně aktivovat všechny senzory a navíc se pokusí aktivovat přepínač zobrazěná dekorace (viz obr. 3.7).



Obrázek 3.7: Ilustrace struktury přepínače se senzory.

Poslední speciální implementací je třída **LogicActuator**, která může obsahovat logické senzory. Komunikace se senzory probíhá pomocí zasílání zpráv. Pro tento přepínač je vytvořena speciální dlaždice, na kterou nelze hráčem vstoupit. Tato dlaždice je reprezentována třídou **LogicTile**.



Obrázek 3.8: Ilustrace hierarchie přepínačů a senzorů.

3.4 Herní entity

Za deklarací neživých entit stojí rozhraní **IEntity**, které definuje funkci **GetProperty** pro získání vlastnosti dané entity. Za živými entitami stojí naopak rozhraní **ILiveEntity**, která navíc disponuje funkcí **GetSkill** pro získání schopností. Dále také deklaruje:

- tělo entity
- způsob rozmístění entity na dlaždici
- relace s dalšími entitami

3.4.1 Implementace vlastností entit

Vlastnost deklaruje rozhraní **IProperty**, které obsahuje následující atributy:

- **Value** – stanovuje aktuální hodnotu dané vlastnosti, v jejím getteru a setteru se typicky provádějí kontroly okrajových hodnot a případné reakce na ně.
- **BaseValue** – maximální hodnota, které může vlastnost nabýt, pokud není nějak modifikovaná.
- **MaxValue** – určuje maximální hodnotu včetně modifikací.
- **AdditionalValues** – je to kolekce typů **IEntityPropertyEffect** obsahující modifikace dané vlastnosti.
- **Type** – je to vlastnost typu **IPropertyFactory**, která reprezentuje typ vlastnosti. Entita na základě tohoto typu potom vrací odpovídající vlastnost. Pro usnadnění tvorby unikátních typů existuje generická třída **PropertyFactory**, která má jako typový parametr typ alespoň **IProperty**. Tato třída se řídí vzorem singleton, jejíž jediná instance lze získat přes statickou položkou **Instance**. Takto lze pak generovat typy pomocí třídy reprezentující vlastnost.

Pro usnadnění práce existuje implementace rozhraní **IProperty** a to třída **Property**. Tato třída definuje maximální hodnotu jako součet základních hodnoty a sumu modifikujících hodnot. Dále při nastavení konkrétní hodnoty ořezává hodnotu do povoleného intervalu, který je mezi nulou a maximální hodnotou. Také definuje událost vyvolanou při změně hodnoty. Všechny vlastnosti jsou deklarovány abstraktně nebo virtuálně.

Některé vlastnosti mohou požadovat přístup k jiným vlastnostem či přístup k rodičovské entitě. V takovém případě je na programátora, jak takového cíle dosáhne. Typicky se v takovém případě předá v konstruktoru potřebná vlastnost nebo se vytvoří událost, která danou závislost deleguje vně.

Seznam předdefinovaných vlastností je možné najít ve jemném prostoru **DungeonMasterEngine.DungeonContent.Entity.Properties**. Jak již bylo zmíněno (viz sekce 2.8.1), může nastat, že daná entita vlastností nedisponuje. V takovém případě je vrácena instance třídy **EmptyProperty**.

3.4.2 Implementace dovedností entit

Dovednosti deklaruje rozhraní **ISkill**, které obsahuje následující atributy:

- **SkillLevel** – udává úroveň, na které je daná dovednost včetně úrovní získaných kouzelnými předměty.
- **BaseSkillLevel** – udává úroveň bez úrovní získaných kouzelnými předměty.
- **Experience** – určuje celkovou dosavadní hodnotu zkušeností.
- **TemporaryExperience** – určuje dočasné zkušenosti, které můžou být jak přidávány tak odebírány.
- **BaseSkill** – je reference na základní dovednost (viz sekce 2). Pokud je daná schopnost již základní, je hodnota **null**.
- **AddExperience** – umožňuje přidat zkušenosti.
- **Type** – je to vlastnost typu **ISkillFactory**, která reprezentuje typ dovednosti. Entita na základě tohoto typu potom vrací odpovídající dovednost. Stejně jako u vlastností, je možné tyto reprezentanty vytvářet pomocí generické třídy **SkillFactory**.

Konkrétní algoritmus přepočítávání zkušeností na úrovní dovedností záleží vždy na konkrétní implementaci.

Třída **SkillBase** implementuje získávání zkušeností schopností jako v originální hře. Všechny schopnosti využívající tuto třídu jsou ve jemném prostoru **DungeonMasterEngine.DungeonContent.Entity.Skill**. Stejně jako u vlastností i zde entita nemusí mít dotazovanou schopnost. V takovém případě vrací instanci třídy **EmptySkill**.

3.4.3 Tělo – **IBody**

Následující API umožňuje pro entity vytvořit různé druhy těl. Tělo se skládá z částí, které mohou sloužit jako úložiště. Dále pak existují externí úložiště tzv. inventáře. API je navržené tak, aby dovolovalo definovat různé druhy úložišť a těl. Engine obsahuje pouze implementaci pro lidské tělo tj. třída **HumanBody**. Tělo entity je definováno rozhraním **IBody** a obsahuje následující atributy:

- **BodyParts** – obsahuje seznam částí těla,
- **Storages** – obsahuje seznam všech úložišť včetně částí těla,
- **GetStorage** – funkce pro vyhledání úložiště včetně částí těla,
- **GetBodyPart** – funkce pro vyhledání části těla.

Externí úložiště – **IInventory**

Každý inventář má definovanou vlastnost **Type**, což je typ úložiště, který reprezentuje instance třídy implementující rozhraní **IStorageType**. Dále pak definiuje readonly kolekci **Storage** pro ukládání předmětů. A v poslední řadě obsahuje funkce pro přidávaní a odebírání předmětů z úložného prostoru tj. **TakeItemFrom**, **AddItemTo**, **AddItem**, **AddRange**. Třída **Inventory** implementuje všechny funkce inventáře a je jí tedy možno přímo použít nebo rozšířit.

Část těla – **IBodyPart**

Část těla je definovaná rozhraním **IBodyPart**, které je potomkem rozhraní **IInventory**. Toto rozhraní navíc definuje následující vlastnosti:

- **IsWound** – definuje, zda je část těla zraněna,
- **InjuryMultiplier** – definuje pravděpodobnost zranění části těla..

Obecnou implementací rozhraní je třída **BodyPart**.

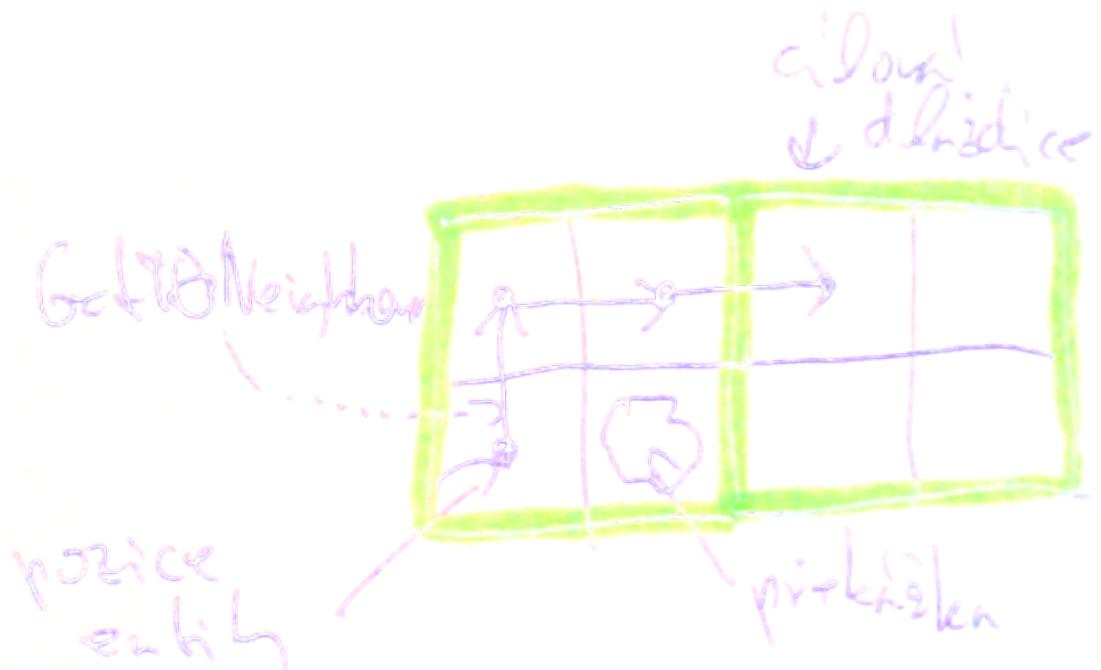
3.4.4 Rozmístění entity na dlaždici

Rozhraní **IGroupLayout** definuje obecně možné rozmístění na dlaždici. Jeho vlastnost **AllSpaces** obsahuje všechny možné prostory, které daná entita může využít. Toto prostory rozdělují prostor dlaždice na mřížku (viz obr. 3.9), která nemusí být rovnoměrná. Jednotlivé prostory jsou reprezentovány rozhraním **ISpace**. Dale API vyžaduje funkci **GetToNeighbour**, která naleze cestu skrze prostory na cílovou sousední dlaždice (viz obr. 3.10). Funkce **GetToSide** vrací cestu k libovolné ze stran dlaždice. Jednotlivé články cesty jsou reprezentovány rozhraním **ISpaceRouteElement**. Poslední funkce musí umět vytvořit z prostoru a dlaždice článek cesty, tedy **ISpaceRouteElement**.

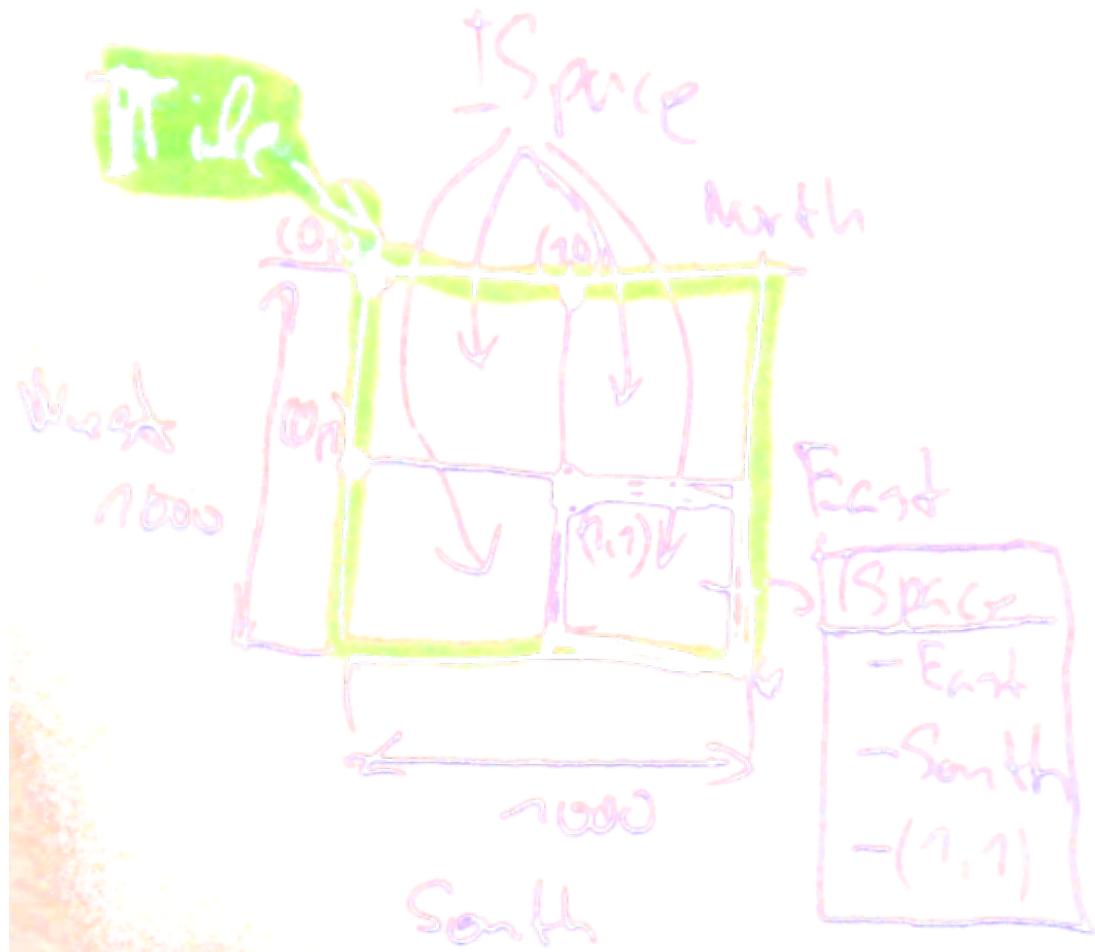


Obrázek 3.9: Ilustrace rozdelení dlaždice na prostory.

Rozhraní **ISpace** musí definovat, jakým stranám dlaždice je prostor přilehlý. Dále musí pomocí obdélníku definovat prostor, který na dlaždici využívá. Celý prostor dlaždice je pak definovaný na pole o velikosti 1000x1000. Přičemž souřadnice rostou shora dolů a zleva doprava. Nahoře je pak sever, vpravo východ, dole jih a vlevo západ. Kromě toho také musí definovat sousední prostory, k čemuž využívá generické rozhraní **INeighborable**. Ilustrace viz obrázek (viz obr. 3.11)



Obrázek 3.10: Ilustrace nalezení cesty na sousední dlaždici.



Obrázek 3.11: Ilustrace vlastností prostorů.

Rozhraní **ISpaceRouteElement** se potom skládá pouze z prostoru, dlaždice a absolutní pozice, na které má stát daná entita. Pro výpočet této pozice se může použít pozice dlaždice a prostor. Při při implementaci tohoto rozhraní je možné využít předem připravený hledač nejkratších cest pro prostory **GroupLayoutSearcher**. Pro reprezentaci sousedů prostorů mřížky je tu zase tříd **FourthSpaceNeighbors**.

Celé toto rozhraní je immutable objekt, který pouze definuje prostory na dlaždici a způsob pohybu mezi nimi. Z toho důvodu instance tříd implementující toto rozhraní mohou být singletony. Engine definuje layout pro prostor reprezentující celou dlaždici a pro prostory jako čtvrtiny dlaždice.

Řízení obsazeného prostoru na dlaždici

K předchozímu mechanismu je ještě třeba další část, která bude zaznamenávat samotné využité pozice na dlaždici. K tomuto účelu existuje třída **LayoutManager**. Lze pomocí ní získat seznam entity na dlaždici a seznamy využitých prostorů dlaždic. Dále poskytuje API pro přidání entity na daný prostor na dlaždici, odebrání prostoru a získání entit, které využívají alespoň část nějakého prostoru. Entity s různými rozmístěními mohou být na stejně dlaždici, pokud je na ní dostatek prostoru pro oba (viz obr. 3.12).



Obrázek 3.12: Ilustrace vlastností prostorů.

3.4.5 Relace s dalšími entitami

Každá živá entita musí definovat vlastnost typu **IRelationManager**. Toto rozhraní musí definovat relační token typu **RelationToken** pro danou entitu. Dále definuje funkce, která pro daný token vrátí, zda entity odpovídající danému tokenu je nepřátelská. Jednoduchou implementací tohoto rozhraní je třída **DefaultRelationManager**, která má si při svém vzniku definuje své neměnné nepřátele. Nicméně pokud daná implementace nevyhovuje je čistě na programátora, aby si vytvořil implementaci vlastní. K dispozici je ještě generátor uniká-

ních tokenů a to statická třída **RelationTokenFactory**.

3.4.6 Implementace entit

Engine obsahuje implementaci dvou entit a to pro šampiony jejich nepřátele.

Šampion

Šampion je první živá entita a reprezentuje jej třída **Champion**. Šampion neobsahuje žádnou umělou inteligenci, je ovládán hráčem skrze třídu **Leader**, která reprezentuje hráčovu skupinu šampionů. Inicializace vlastností a schopností probíhá přes datový initializer definovaný rozhraním **IChampionInitializer**. Resp. jeho předáním do konstruktoru, dále je nutné specifikovat relační token a seznam nepřátel. Posunout daného šampiona je možné přiřazením do vlastnosti **Location**.

Zde je dobré zmínit generickou třídu **Animator**, která má dva typové parametry. První z nich je typ posouvaného objektu, který musí implementovat alespoň rozhraní **IMovable**. Toto rozhraní definuje vlastnosti, pomocí kterých lze měnit pozici daného objektu. Dalším parametrem je typ prostorů, mezi kterými se objekt pohybuje. Ten musí implementovat alespoň rozhraní **IStopable**, které definuje pozici, na kterém se má objekt postavit. Animátor poskytuje API pro plynulý posun objektů mezi prostory.

U šampiona je tento animátor použit automaticky při změně lokace.

Nepřátelské entity

Všechny nepřátelské entity ve hře reprezentuje třída **Creature**. Vlastnosti jednotlivých typů příšer jsou ve třídách typu **CreatureFactory**. Každá konkrétní instance příšery má referenci na tuto třídu a její chování je ovlivněné vlastnostmi v ní obsažené.

Tato třída definuje pro nepřátelské enity jednoduchou umělou inteligenci. K zjednodušení implementace jsou zde využity asynchronní funkce (viz sekce 3.1.1). Ty jsou především využity pro vytváření zpoždění akcí pomocí **Task.Delay**, bez nutnosti počítání času a vracení se zpět do funkce po jeho uplynutí. Příkladem použití může být také čekání na dokončení pohybu mezi dlaždicemi viz následující kód. V tomto příkladě je navíc funkce **animator.MoveToAsync** implementována pomocí future a promise, která se splní po dokončení pohybu. Pohyb samotný je potom prováděn synchronně v metodě **animator.Update**.

```
async Task<bool> MoveToSpace(ISpaceRouteElement destination)
{
    bool EnemyAtTile = destination.Tile.LayoutManager.Entities
        .Any(x => RelationManager.IsEnemy(x.RelationManager.RelationToken));
    if (!EnemyAtTile &&
        destination.Tile.LayoutManager.TryGetSpace(this, destination.Space))
    {
        //free previous location
        location?.Tile.LayoutManager.FreeSpace(this, location.Space);
        await animator.MoveToAsync(this, destination, setLocation: true);
        return true;
    }
}
```

```

    }
else
    return false;
}

```

Následuje seznam funkcí a jejich popis použitých pro simulování inteligence. Všechny tyto funkce je možné přetěžovat.

- **Live** – V této funkci je nekonečný cyklus, který volá obslužné rutiny podle stavu příšery. Jsou to:
 - Lov – příšera spatřila nepřítele a pronásleduje ho
 - Cesta domů – příšera pronásledovala nepřítele, kterého následně ztratila, proto jde domů tj. na místo svého vzniku
 - Hlídkování – příšera hlídka v okolí oblasti svého vzniku
- **FindEnemies** – Pomocí prohledávání do šířky se pokusí najít entity s nepřátelským tokenem. Maximální hledanou oblast je určena dle vlastnosti příšery. Pokud je nepřítel nalezen nastaví proměnou **hountingPath**.
- **MoveToSpace** – Přesune příšeru mezi prostory pomocí animátoru a nastaví správně použité prostory pomocí **LayoutManageru**.
- **MoveThroughSpaces** – Přesune příšeru přes prostory dlaždice až na cílovou dlaždici pomocí funkce **MoveToSpace**. Při každém pohybu hledá nepřátele pomocí funkce **FindEnemies**, pokud je tak nastaveno parametrem funkce. Pokud nějaké najde, přeruší pohyb.
- **MoveToNeighbourTile** – Posune příšeru na určenou dlaždici přes prostory dlaždice pomocí funkce **MoveThroughSpaces**. Pokud je cesta nepřístupná vrátí **false**.
- **GoHome** – Příšera jde domů podle cesty uložené v proměnné **homeRoute** a poté ji nastav na **null**. Mezi dlaždicemi se posunuje pomocí funkce **MoveToNeighbourTile**.
- **FindNextWatchLocation** – Pomocí prohledávání do šířky z místa objevení příšery nalezne dlaždici, na které dlouho příšera nebyla a vrátí k ní cestu.
- **WatchAround** – Pomocí funkce **FindNextWatchLocation** nalezne cestu na další místo k hlídkování. Poté jde na dané místo pomocí funkce **MoveToNeighbourTile**.
- **Fight** – Provede útok na nepřítele.
- **PrepareForFight** – Dostane se co nejblíže k dlaždici, na který je nepřítel a zahájí útok pomocí funkce **Fight**.
- **GetPathHome** – Pokusí se nalézt cestu domů. Pokud ji nalezne nastaví **homeRoute**.
- **EstablishNewBase** – Nastaví výchozí dlaždici na nynější.

- **Hount** – Příšera pronásleduje nepřítele na poslední místo, kde ho spatřila. Pokud je nepřítel na sousední dlaždici připraví se k útoku pomocí funkce **PrepareForFight**. Pokud nepřítele ztratí, pokusí se najít cestu domů pomocí **GetPathHome**. Pokud cestu nenaleze založí si domov tam, kde je pomocí funkce **EstablishNewBase**.

K prohledávání do šířky se používá třída **BreathFirstSearch** nebo některý její potomek.

3.5 Předměty

Předměty reprezentuje rozhraní **IGrabableItem**. Jak již bylo zmíněno v analýze (viz sekce 2.15), každý předmět musí mít referenci na svoji factory minimálně typu **IGrabableItemFactoryBase**. Dále musí definovat vlastnost **Location** určující prostor dlaždice, na kterém se nachází. Přiřazení do této vlastnosti by mělo vyvolat přidání daného předmětu na danou dlaždici. Existují i případy, kdy toto není žádoucí a proto předměty musí ještě definovat funkci **SetLocation**, která pouze danou vlastnost nastaví. V poslední řadě musí definovat renderer.

Rozhraní factory na obecné předměty vyžaduje vlastnosti jako název, hmotnost, seznam možných akcí s předmětem a také definuje místa, kam lze předmět uložit.

3.5.1 Implementace předmětů

Při tvorbě vlastního předmětu je třeba vytvořit implementaci zmíněného rozhraní **IGrabableItem**. Tato implementace by kromě vyžadovaných položek měla obsahovat položky, které se můžou měnit pro každou instanci daného typu předmětu. Dále je třeba vytvořit samotnou factory na tyto předměty implementací rozhraní **IGrabableItemFactory**. Ta by měla obsahovat všechny společné vlastnosti pro daný typ předmětu. Z toho důvodu je dobré do vlastností přidat kromě rozhraním požadované reference na factory i přesně typovanou referenci, skrze kterou bude možné k vlastnostem přistupovat. Inspirace lze nalézt v již implementovaných třídách, které jsou v jemném prostoru **DungeonMasterEngine.DungeonContent.GrabableItem**.

3.6 Akce

Akce definuje rozhraní **IAction**, které má následující položky:

- **Factory** – reference na factory pro daný typ akcí.
- **Apply** – aplikuje akci, přičemž může použít směr, který je předaný jako parametr. Ostatní položky je třeba předat při inicializaci akce. Co akce provede na konkrétní implementaci. Implementované akce boje využívají směr k určení dlaždice, kam je útok aplikovaný.

Existuje ještě generický potomek, který může mít jako typový parametr potomka třídy **IActionFactory**. Toto rozhraní vyžaduje jedinou funkci **CreateAction** pro tvorbu akcí typu **IAction**. Každá factory je pak uložena ve třídě implementující **IFactories** v jádře enginu (viz sekce 3.1), tak aby mohl tyto akce kdokoliv používat.

Třída **AttackBase** obsahuje některé funkce, které používají jak útoky nepřátelský entit, tak útoky šampionů. Třída **CreatureAttack** je jejím prvním potomkem a implementuje útoky nepřátelský entit. Třída **HumanAttack** je jejím druhým potomkem a je rodičem pro všechny útoky šampionů, které lze nalézt ve jemném prostoru **DungeonMasterEngine.DungeonContent.Actions**.

3.7 Kouzla

Kouzlo definuje rozhraní **ISpell**, které vyžaduje implementovat jedinou funkci **Run**. Tato funkce má dva parametry a to vyvolávající entitu a směr vyvolání kouzla. Pro každé kouzlo pak existuje factory, kterou definuje rozhraní **ISpellFactory** s následujícími položkami:

- **CastingSequence** – sekvence symbolů typu **ISpellSymbol**, které vyvolají kouzlo.
- **Name** – název kouzla.
- **Difficulty** – modifikátor obtížnosti při vyvolávání kouzla.
- **Duration** – doba, po kterou šampion nemůže znova vyvolávat kouzla.
- **Skill** – dovednost typu **ISkillFactory** nutná pro vyvolání kouzla.
- **SkillLevel** – úroveň předchozí dovednosti nutná pro vyvolání kouzla.
- **CastSpell** – funkce která vytvoří kouzlo typu **ISpell**.

Za jednotlivými symboly potom stojí rozhraní **ISpellSymbol**, které definuje název symbolu a cenu many pro každý power level za použití symbolu. Pro samotné vyvolávání kouzel je pak použita třída **SpellCastingManager**, která implementuje rozhraní **ISpellCastingManager**.

3.8 Renderery

Jak bylo zmíněno v analýze engine odděluje zobrazovací vrstvu od zbytku enginu (viz sekce 2.14). Všechny objekty, které potřebují mít grafický výstup by měli implementovat rozhraní **IRenderable**, které vyžaduje položku typu **IRenderer**. Stěžejní funkce tohoto rozhraní jsou funkce **Render** a **Interact**.

Nejdůležitější z parametrů těchto funkcí je dosavadní transformace. Tj. obsahuje složeninu transformací složenou z jednotlivých transformací na cestě z kořene stromu reprezentující závislosti rendererů na sobě. Takže například kořen stromu bude renderer dlaždice, jeho syn bude strana dlaždice, její syn bude výklenek ve zdi a její syn bude předmět ve výklenku (list). Při takovéto reprezentaci se pak musí všechny renderované objekty posunout či jinak transformovat vůči jejich

rodiči. Tzn. každý renderer si musí zvolit pozicovací konvenci, kterou pak musí závislé renderery dodržovat. Tento způsob renderování je požíván u statických objektů jako jsou dlaždice, stěny, výklenky a podobně. Pro pohybující objekty je používaná absolutní pozice a renderery těchto objektů transformují objekty přímo na jejich pozici. Volba mezi těmito dvěma způsoby pozicování stojí vždy za zamýšlení.

Renderery jsou vždy dělány na míru objektu, který mají renderovat. Tzn. často se renderer v konstruktoru inicializuje instancí s konkretním typem. Třída této instance pak má zpravidla readonly vlastnosti, podle kterých renderer určuje, co má vykreslovat. Jelikož implementovaná grafická vrstva je pouze ve formě proof of concept, je co nejjednodušší a neobsahuje například animace. Nicméně ze zde popsané povahy rendererů je jasné, že toho může být dosaženo vytvořením událostí na renderovaných objektech, které si renderer zaregistrouje. Dovedeme si představit, že by šla s takovým návrhem velmi jednoduše udělat grafická vrstva, která bude velmi pěkná, bude mít kvalitní 3D modely a animace. Zabralo by to jen spoustu času a byla by k tomu potřeba spousta grafiků.

Při rozšíření nějakého rendereru je třeba zjistit dosavadní transformaci, která je normálně vypočítána v rodiči. K tomuto záměru slouží funkce **GetCurrentTransformation**, která bere jako parametr dosavadní transformaci. Rozšířením již existujícího rendereru si lze ušetřit mnoho práce a opakujícího se kódu. Proto je takový přístup v této implementaci často použit. Funkce popsané na začátku sekce jsou samozřejmě virtuální a jdou přetěžovat, obsahují také jeden parametr typu **object** pro případné předávaní dodatečných dat mezi renderery. Tohoto parametru není nikde v této implementaci využito.

Jelikož renderery zásadně mění vzhled a pozici všech věcí, je nutné tuto vrstvu propojit i s interakcemi. Interakční funkce má skrze parametr typu **ILeader** přístup k položce interactorku, která je typu **object**. Daný renderer musí vědět, pro jaký typ interactorku je a na ten si ho musí vhodně přetypovat. V této implementaci je použit paprsek (**Ray**). Celý tento koncept interactorku by šel sice udělat genericky, ale prolínal by se celou strukturou rendererů a z toho důvodu jsme se rozhodli v tomto místě ustoupit a udělat situaci jednoduší na úkor kontroly za překladu.

3.9 Builder map

Nyní když máme rozebrané všechny části enginu, které je potenciálně možné rozšířit, si můžeme vysvětlit, jak se vše sestaví dohromady v herní úrovni.

3.9.1 Vlastní implementace builderu

Jak již bylo zmíněno builder musí implementovat rozhraní **IDungeonBuilder** (viz sekce 3.1). Toto rozhraní je potom dotazované pro načtení herní úrovni jádrem enginu. Je úkolem builderu rozhodnout jak bude jednat v případě, že je podán několikrát dotaz na stejnou mapu. Typicky je tedy nutné rozhodnout se, jestli načtené mapy kešovat či nikoliv.

Správně implementovaný builder by měl načítat mapy zhruba tímto způsobem.

1. Vytvoření dlaždic a naplnění jejich inicializátorů. To znamená vytvořit strany dlaždic a přepínače či předměty v nich obsažené.
2. Nastavení sousedů dlaždic do inicializátorů.
3. Vytvoření samotného levelu **DungeonLevel**.
4. Dokončení inicializace inicializátoru a zavolení inicializaci dlaždic skrze inicializátory.
5. Případné kešování herní úrovně pro případný opakovaný požadavek.

3.9.2 Implementace builderu Dungeon Masteru

Builder pro samotnu hru Dungeon Master řeší třída **LegacyMapBuilder**. Tento builder využívá již rozparsovaná data reprezentována třídou **DungeonData**. Samotný builder map potom využívá ještě další buildery. Jsou to třídy pro vytváření dlaždic, přepínačů, předmětů a příšer. Při požadavku na určitou mapu pak následuje postup zmíněný v předchozí sekci.

4. Uživatelská dokumentace

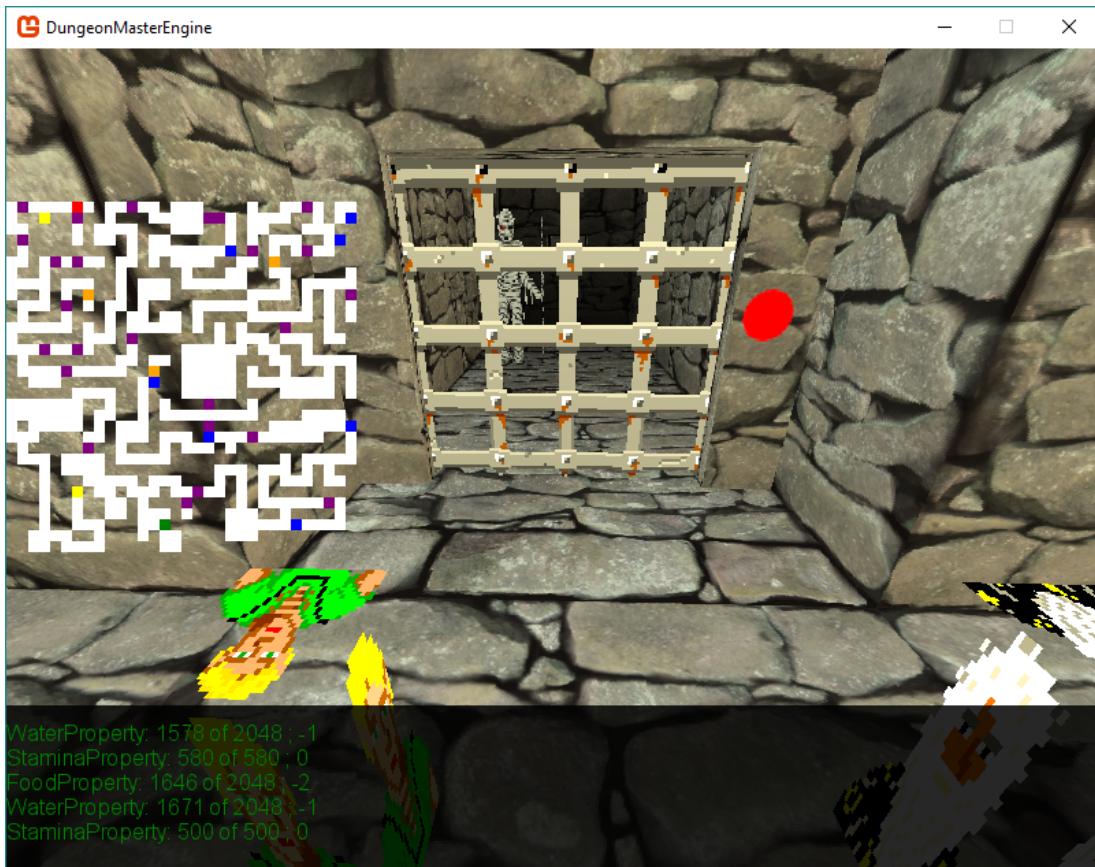
Pro spuštění reimplementace Dungeon Masteru je zapotřebí:

- Windows Vista a novější
- DirectX 9.0c runtime
- .NET 4.6

Tento projekt si neklade za cíl udělat zcela kompletní a dobré hratelnou reimplementaci Dungeon Master. Naopak se soustředí na dobrý návrh enginu jako takového. Z toho důvodu není herní zážitek nikterak oslňující, nicméně jako demonstrace funkčnosti enginu poslouží dobrě. Hru je možné spustit souborem `/DungeonMaster/DungeonMasterEngine.exe`.

4.1 Mechaniky ve hře

Hráč reprezentuje vůdce skupiny bojovníků známých jako šampióni. Může ovládat jejich pohyb, předávat jim předměty, donutit je k boji, ke kouzlení, ke konzumaci jedlých předmětů nebo lektvarů. Každý bojovník má sadu vlastností a dovedností, v kterých je schopný se zdokonalovat získáváním zkušeností. Zkušenosti lze získat bojem na prázdro, bojem proti nepřátelům, kouzlením, nebo jen použitím některých přepínačů. V dungeonu jsou přepínače na zdech, které lze aktivovat kliknutím myši. Dále jsou zde přepínače na podlaze, které lze aktivovat vstoupením na podlahu nebo hozením předmětu na daný spínač. Předměty lze pokládat na zem, do výklenků a nebo je uložit na tělo bojovníka či do jeho batohu. Přepínače pak mohou aktivovat nebo deaktivovat určité objekty ve hře jako jsou například dveře, teleporty, jámy, otevírací zdi, atd. Některé dveře je také možné rozbít útkem. Některé teleporty jsou pouze na konkrétní typy objektů. Do jam lze spadnout, ale lze se z nich většinou teleportem dostat ven. V chodbách je tma a proto je nutné používat pochodně – což lze vložením do jedné z ruk šampiona - nebo je nutné vyvolat magické pochodně.



Obrázek 4.1: Ilustrace reimplementace hry Dungeon Master

4.2 Cíl hry

Hra není úplně kompletní, proto ji není možné zcela dohrát. Z toho důvodu by se za cíl hry dalo považovat dostání se do poslední herní úrovně, kde nebudou neimplementované funkce bránit dalšímu postupu.

4.3 Ovládání

Pohybovat skupinou lze pomocí kláves W, S, A, D tj. dopředu, dozadu, doleva, doprava. Dále se je možné rozhlížet pomocí šipek na klávesnici.

4.3.1 Aktivace přepínačů a sbírání předmětů

Přepínače lze aktivovat namířením ukazatele myši na daný objekt a kliknutím nebo stiskem klávesy **ENTER**. Pokud se pod kurzorem vyskytuje předmět nebo přepínač, který může vložit nějaký předmět hráčovi do ruky, tak se tak provede, pokud je hráčova ruka prázdná. Pokud hráč již v ruce něco má, provede se opačná akce tj. předmět se položí na dané místo nebo je pohlcen přepínačem. Ostatní interakce probíhá přes konzoli. Konzole se aktivuje stiskem klávesy **TAB** a deaktivuje opětovným stisknutím tytéž klávesy. Příkaz **hand** zobrazí popis předmětu v hráčově ruce. Přidáním parametru **take** se provede uložení předmětu do dotažovaného inventáře daného bojovníka. Přidáním parametru **put** se naopak vloží

dotazovaný objekt z inventáře do ruky. Dalším možným parametrem příkazu je **takesub** resp. **putsub**, pomocí kterého lze uložit předmět z ruky do truhly v inventáři nějakého šampiona resp. vložit předmět z truhly šampiona do ruky. Posledním možným parametrem příkazu **hand** je parametr **use**, který se pokusí aplikovat předmět v ruce na zvoleného šampiona. Takovým způsobem lze použít jídlo, lektvary, nebo voda.

4.3.2 Souboj

Pro souboj je nejprve nutné vložit zbraň do akční ruky. Což je možné provést příkazy z minulé sekce. Samotný souboj probíhá potom skrze příkaz **fight**, kdy je interaktivně vybrán bojovník a způsob útoku. Příkaz lze také použít s dvěma číselnými parametry, kde první identifikuje pořadí šampiona a druhý pořadí akce. Kromě boje na blízko lze ještě vyvolávat útočná kouzla. K tomu slouží příkaz **spell**, kterému se jako první parametr předá index šampiona a pak buď jako parametry jména symbolů, nebo při nespecifikování parametrů příkazu lze šampiona a symboly zadat interaktivně.

4.3.3 Odpočinek

Posledním příkazem je **champion**, který může mít následující parametry:

- **list** – vypíše jména šampionů.
- **sleep** – uspí šampiony a tím dojde krychlejší regeneraci vlastností.
- **wake** – probudí šampiony ze spánku.

Pro více informací o příkazech je možné napsat příkaz **help** do konzole.

Závěr

V rámci této práce byl naimplementovaný engine pro Hru Dungeon Master. Engine obsahuje podporu pro všechny funkce z originální hry. Nicméně všechny konkrétní funkce nejsou naimplementované, což dává možnost navázat na práci případným následovníkům. Pro implementaci byl použit jazyk C#, platforma .NET a framework MonoGame. Většina částí enginu je jednoduše rozšířitelná či modifikovatelná, což vede na dobrou udržitelnost celého systému. Primárně je engine určený pro hru Dungeon Master a dokáže si herní úrovně načíst z originální binárních dat. Avšak načítání herních úrovní je v oddělené vrstvě, a proto je možné tuto vrstvu upravit kvůli případným rozšířením nebo ji je možné nově naimplementovat i pro jiné vstupní formáty. Z toho důvodu lze tento engine využít i pro implementaci jiných her na podobných principech. Engine má také oddělenou renderovací vrstvu, což umožňuje jednoduše doimplementovat hře lepší look and feel. Tento projekt tak může sloužit pro vzdělávání jazyka C# a objektově orientovaného programování. Studenti si tak můžu vyzkoušet do enginu dodělat nové komponenty a tak si vyzkoušet objektově orientované programování v praxi.

Future works

Na tuto práce je potenciálně možné navázat v následujících bodech:

- Dodělání všech funkcí Dungeon Masteru, tak aby hra byla hratelná až do konce.
- Dodělání zobrazovací vrstvy, tak aby odpovídala vzhledu původní hry Dungeon Master.
- Dodělání lepší 3D zobrazovací vrstvy.
- Udělat podporu pro textový výstup zobrazovací vrstvy pro možnost automatizovaného kontrolování například systémem CodEx.
- Vytvořit editor pro hru Dungeon Master.
- Vytvořit jinou hru na tomto enginu.
- Upravit engine tak, aby nebyl vázaný na stejně velké dlaždice uspořádané do mřížky, a umožňoval tak flexibilnější tvorbu herních úrovní.

Seznam použité literatury

- [1] J. A. L. de Farias. MonoGame. 2009. URL <http://www.monogame.net/>.
- [2] C. Fontanel. Technical Documentation - Dungeon Master and Chaos Strikes Back Actions and Combos. 2005. URL <http://dmweb.free.fr/?q=node/690>.
- [3] C. Fontanel. Technical Documentation - Dungeon Master and Chaos Strikes Back Items properties. 2005. URL <http://dmweb.free.fr/?q=node/886>.
- [4] C. Fontanel. Dungeon Master Items. 2005. URL <http://dmweb.free.fr/?q=node/259>.
- [5] C. Fontanel. Dungeon Master and Chaos Strikes Back Spells. 2005. URL <http://dmweb.free.fr/?q=node/195>.
- [6] C. Fontanel. Technical Documentation - File Formats - Dungeon File (DUNGEON.DAT). 2005. URL <http://dmweb.free.fr/?q=node/217>.
- [7] C. Fontanel. Technical Documentation - Dungeon Master and Chaos Strikes Back Creature Details. 2008. URL <http://dmweb.free.fr/?q=node/1363>.
- [8] C. Fontanel. Technical Documentation - Graphics.dat Item 559. 2008. URL <http://dmweb.free.fr/?q=node/1395>.
- [9] C. Fontanel. Back to the source: ReDMCSB. 2014. URL <http://www.dungeon-master.com/forum/viewtopic.php?f=25&t=29805>.
- [10] B. Kernighan and D. Ritchie. *The C Programming Language*. 1978.
- [11] S. Mourier. Html Agility Pack. 2012. URL <https://htmlagilitypack.codeplex.com/>.

Seznam obrázků

1.1	Screenshot originální hry Dungeon Master	3
1.2	Ilustrace uspořádání herních úrovní	3
1.3	Ilustrace prostorů zaujímaných entitami.	4
2.1	Ilustrace formátu souboru DUNGEON.DAT	8
2.2	Možné pozice pro uložení objektů na dlaždici.	10
2.3	Ilustrace vztahu instancí a typů.	11
2.4	Ilustrace obsahu identifikátoru a jeho využití ve spojových seznamech.	12
2.5	Ilustrace uspořádání prvních objektových identifikátorů dlaždic.	12
2.6	Ilustrace vztahu parseru vůči enginu.	16
2.7	Průběh parsování herních dat.	18
2.8	Uložení dlaždic v originálním vs tomto enginu.	20
2.9	Ilustrace přesunu stěn do dlaždice <i>podlahy</i>	21
2.10	Ilustrace přepínače pro držák pochodní.	23
2.11	Ilustrace přepínače reprezentující páku.	24
2.12	Ilustrace přepínače reprezentující dočasné tlačítko.	24
2.13	Ilustrace transformace sekvence senzorů na objekt přepínač.	25
2.14	Ilustrace transformace sekvence senzorů na objekt přepínač.	26
2.15	Ilustrace vztahu objektů a jejich factories.	30
2.16	Ilustrace struktury builderu.	32
2.17	Ilustrace použití inicializátoru.	33
3.1	Ilustrace komunikace frondendu s jádrem enginu.	36
3.2	Ilustrace funkce jádra.	37
3.3	Ilustrace aktualizace a vykreslování z pohledu jádra.	38
3.4	Ilustrace pohybu mezi prostory a dlaždicemi.	40
3.5	3D ilustrace dlaždice a jejích stran.	42
3.6	Ilustrace komunikace podlahy s její dlaždicí při přidávání předmětu.	44
3.7	Ilustrace struktury přepínače se senzory.	48
3.8	Ilustrace hierarchie přepínačů a senzorů.	48
3.9	Ilustrace rozdělení dlaždice na prostory.	51
3.10	Ilustrace nalezení cesty na sousední dlaždici.	52
3.11	Ilustrace vlastností prostorů.	53
3.12	Ilustrace vlastností prostorů.	54
4.1	Ilustrace reimplementace hry Dungeon Master	62

Přílohy