



**MATEMATICKO-FYZIKÁLNÍ  
FAKULTA  
Univerzita Karlova**

**BAKALÁŘSKÁ PRÁCE**

Jméno Příjmení

**Název práce**

Název katedry nebo ústavu

Vedoucí bakalářské práce: Vedoucí práce

Studijní program: studijní program

Studijní obor: studijní obor

Praha ROK

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

Poděkování.

Název práce: Název práce

Autor: Jméno Příjmení

Katedra: Název katedry nebo ústavu

Vedoucí bakalářské práce: Vedoucí práce, katedra

Abstrakt: Abstrakt.

Klíčová slova: klíčová slova

Title: Name of thesis

Author: Jméno Příjmení

Department: Name of the department

Supervisor: Vedoucí práce, department

Abstract: Abstract.

Keywords: key words

# Obsah

<b>1 Úvod</b>	<b>3</b>
1.1 RTS hry blíže . . . . .	4
1.1.1 Mapy . . . . .	6
1.1.2 Jednotky . . . . .	7
1.1.3 Budovy . . . . .	10
1.1.4 Suroviny . . . . .	13
1.1.5 Vývoj technologií . . . . .	14
1.2 Uživatelé . . . . .	16
1.2.1 Tvůrci her . . . . .	16
1.2.2 Hráči her . . . . .	18
1.3 Ukázková hra . . . . .	18
1.4 Cíle práce . . . . .	19
<b>2 Analýza</b>	<b>22</b>
2.1 Herní engine . . . . .	22
2.2 Rozdíly systémů . . . . .	22
2.2.1 Zobrazení a ovládání . . . . .	22
2.2.2 Kompilace . . . . .	24
2.2.3 Souborové systémy . . . . .	25
2.2.4 Shrnutí . . . . .	26
2.3 Formát a načítání dat . . . . .	26
2.3.1 Struktura balíčku . . . . .	26
2.3.2 Data entit . . . . .	27
2.3.3 Formáty assetů . . . . .	28
2.3.4 Formát uložených úrovní . . . . .	31
2.4 Poskytovaná funkcionality . . . . .	32
2.4.1 Mapa . . . . .	32
2.4.2 Hledání cesty . . . . .	36
2.4.3 Projektily . . . . .	39
<b>3 Programátorská dokumentace</b>	<b>42</b>
3.1 Struktura solution . . . . .	42
3.2 Spuštění aplikace . . . . .	43
3.2.1 Systémová část . . . . .	43
3.2.2 Přenositelná část . . . . .	44
3.3 Uživatelské rozhraní . . . . .	45
3.3.1 Definice vzhledu . . . . .	46
3.3.2 Přepínání obrazovek . . . . .	46
3.3.3 Automatizace uživatelského rozhraní . . . . .	47
3.4 Systém balíčků . . . . .	48
3.4.1 Adresář balíčků . . . . .	48
3.4.2 PackageManager . . . . .	49
3.4.3 GamePack . . . . .	50
3.4.4 Assety . . . . .	51
3.4.5 Načítání balíčku . . . . .	52

3.5	Logika hry . . . . .	53
3.5.1	Herní engine . . . . .	53
3.5.2	Pohled platformy . . . . .	55
3.5.3	Typy . . . . .	57
3.5.4	Pluginy . . . . .	57
3.6	Mapa . . . . .	60
3.6.1	Dlaždice . . . . .	61
3.6.2	Zobrazení . . . . .	63
3.7	Hledání cesty . . . . .	64
3.7.1	Reprezentace vrcholů a hran . . . . .	64
3.7.2	Dynamická část generování . . . . .	65
3.7.3	Reprezentace cesty . . . . .	65
3.7.4	Výběr algoritmu . . . . .	66
3.7.5	Implementace v platformě . . . . .	66
3.8	Kamera . . . . .	66
3.8.1	Typická RTS kamera . . . . .	66
3.8.2	Kamera sledující jednotku . . . . .	67
3.8.3	Volně létající kamera . . . . .	68
3.9	Vstup . . . . .	68
3.10	Editace mapy a ovládání hry . . . . .	69
3.11	Základní komponenty . . . . .	70
3.12	Ukládání a načítání úrovní . . . . .	72
3.12.1	Stav úrovně . . . . .	72
3.12.2	Pluginy . . . . .	73
<b>4</b>	<b>Tvorba balíčku</b>	<b>75</b>
4.1	Struktura XML . . . . .	75
4.2	Přidání jednotek . . . . .	77
4.3	Přidání budov . . . . .	79
4.4	Přidání projektilů . . . . .	79
4.5	Přidání dlaždic . . . . .	79
4.6	Přidání surovin . . . . .	80
4.7	Přidání AI hráče . . . . .	81
4.8	Přidání AI úrovně . . . . .	81
<b>5</b>	<b>Tvorba pluginů</b>	<b>82</b>
5.1	Vytvoření pluginu jednotky . . . . .	82
5.2	Plugin typu . . . . .	83
5.2.1	Vytvoření instance . . . . .	85
5.2.2	Metody pluginu . . . . .	88
<b>6</b>	<b>Ukázková hra</b>	<b>93</b>
<b>Závěr</b>		<b>94</b>
<b>Seznam použité literatury</b>		<b>95</b>
<b>A</b>	<b>Přílohy</b>	<b>98</b>
A.1	První příloha . . . . .	98

# 1. Úvod

Strategické hry jsou žánrem, ve kterém hráči využívají svých mentálních schopností, především taktického a strategického myšlení, pro porážku jednoho či více nepřátel. Ve většině případů se strategické hry zabývají tématem války. Takto popisuje strategické Ernest Adams ve své knize *Fundamentals of Game Design* [2, str. 419]

Žánr strategických her obsahuje mnoho poddruhů s velice rozdílnými nároky jak na hráče, tak na vývojové prostředí a na vývojáře samotného. Prvním kritériem pro rozdělení strategických her je, zda se hra odehrává jako posloupnost diskrétních tahů, nebo zda se hra odehrává v přímé závislosti na ubíhajícím reálném čase. Druhým kritériem je relativní četnost a důležitost strategických rozhodnutí vůči taktickým rozhodnutím. Tato dvě kritéria použil Mark H. Walker [31] pro jejich rozdělení na tyto poddruhy:

- *Real-time strategy* (RTS) - reálný čas, strategická rozhodnutí,
- *Real-time tactics* (RTT) - reálný čas, taktická rozhodnutí,
- *Turn-based strategy* (TBS) - diskrétní tahy, strategická rozhodnutí,
- *Turn-based tactics* (TBT) - diskrétní tahy, taktická rozhodnutí.

Cílem této práce je vytvořit platformu umožňující tvorbu Real-time strategy (RTS)<sup>1</sup> her. V následujících částech popíšeme rozdíly mezi poddruhy strategických her a vymezíme podmnožinu, jejíž vývoj bude naše platforma podporovat.

## Real-time strategy

RTS, v překladu strategické hry probíhající v reálném čase, jsou poddruhem strategických her ve kterém se změny stavu odehrávají v přímé závislosti na změně času v reálném světě. Reakce v reálném čase jsou náročnější jak pro hráče, který je často nucen použít suboptimální strategii, tak pro hru samotnou, která musí provádět výpočet dalšího stavu v omezeném čase. Stejně tak umělá inteligence, jakožto součást hry, musí reagovat na aktivity zbylých hráčů s omezeným časem, což limituje množství dat a složitost výpočtu, které může umělá inteligence použít. Z tohoto důvodu je vývoj RTS her složitým procesem, spojujícím mnoho oborů, který se pokusíme zjednodušit vytvořením naší platformy.

Zbylé poddruhy strategických her neplánujeme v naší platformě explicitně podporovat, ale nijak nevylučujeme, že bude možné do jisté míry využít naší platformu i pro tvorbu těchto poddruhů strategických her. Zároveň je pro tvorbu RTS her výhodné znát příbuzné žánry, z kterých je možno se inspirovat a přebírat některé z jejich mechanik. Z tohoto důvodu ve zkratce popíšeme i zbylé poddruhy.

---

<sup>1</sup>Název "real-time strategy", poprvé použitý při propagaci hry Dune II [33], je připisován prezidentu a spoluzakladateli Westwood Studios [39], Brettu Sperrymu. Toto studio následně využilo zkušenosti získané při tvorbě Dune II pro vývoj jedné z nejznámějších sérií RTS her, Command & Conquer [37].

## Real-time tactics

Prvním příbuzným žánrem jsou *Real-time tactics* (RTT) hry, někdy také nazývány fixed-unit real-time hry [30], neboli hry s pevným počtem jednotek probíhající v reálném čase. Hlavním rozdílem, odlišujícím RTT od RTS, je omezení strategických rozhodnutí a větším důrazem na taktická rozhodnutí a micromanagement jednotlivých jednotek. RTT hry nedovolují hráči tvorbu nových jednotek, stavbu budov či produkci surovin, hráč je tedy nucen s jednotkami, které má na začátku souboje, vyhrát celý souboj. Jedním z příkladů čistě RTT her je série Blitzkrieg [35]. Hráč začíná každou misi s jednotkami, které si vybral před misí. Tyto jednotky jsou jediné, které bude moci v průběhu mise využít, což nutí hráče použít svých taktických schopností a maximalizovat účinnost těchto jednotek.

## Turn-based strategy

*Turn-based strategy* jsou strategické hry, ve kterých změny stavu probíhají v diskrétních tazích. Doba mezi tahy často není nijak omezena, což hráči umožňuje vymyslet optimální strategii. Oproti RTS tyto hry často omezují taktickou část problémů a umožňují hráči soustředit se výhradně na strategickou část hry, tedy plánování budov, produkci surovin, vývoj technologií a celkovou strategii pro jeho ekonomiku a armádu. Příkladem tohoto žánru je série her Civilization [8]. Naše platforma nebude explicitně podporovat tahy, myslíme si však, že bude možné toto rozdelení do tahů vytvořit použitím platformou poskytovaných prostředků.

## Turn-based tactics

Turn-based tactics umožňují hráči přímé ovládání několika málo jednotek, které v každém tahu mohou provést jeden či více úkonů. Těmito úkony mohou být střelba na nepřátelskou jednotku, vyléčení přátelské jednotky, pohyb po mapě či například zničení terénu. Ukázkovým příkladem je série her X-COM [34]. Ve hře X-COM 2 hráč vlastní až malé desítky jednotek, z kterých vybírá malou skupinu a vysílá ji na jednotlivé mise. Při misi má každá jednotka v každém tahu možnost vykonat dvě akce. Akce může být přesun o omezenou vzdálenost, použití schopnosti či útok na nepřítele. Pro normální jednotky útok ukončuje tah dané jednotky a hráč může provést tah následující.

## Shrnutí

Jak můžeme vidět, žánr strategických her zahrnuje hry s velmi rozmanitými vlastnostmi a požadavky. Z tohoto důvodu se naše práce zaměří na podporu vývoje her jednoho konkrétního poddruhu, a to RTS. Přestože naše platforma bude cílena na tvorbu tohoto poddruhu, nevylučujeme, že bude možné využít ji i pro tvorbu her spadajících do jednoho ze zbylých poddruhů strategických her.

## 1.1 RTS hry blíže

Jak bylo řečeno výše, naše platforma bude navržena pro podporu vývoje RTS her. RTS hry jsou ovšem stále příliš velká množina s příliš rozmanitými mechanika-

kami na to, aby jedna platforma dokázala podporovat všechny možné RTS hry. Proto zde dále omezíme námi podporovanou podmnožinu RTS her.

Již od svého vzniku na konci osmdesátých let a začátku devadesátých let minulého století obsahovaly RTS hry několik konceptů, které lze nalézt v drtivé většině her tohoto žánru i dnes.

Těmito koncepty jsou:

- výroba a ovládání jednotek s cílem ovládnutí části mapy a zničení nepřátelských jednotek a budov;
- stavba budov pro umožnění stavby nových druhů budov, jednotek či získání surovin;
- získávání surovin pro stavbu jednotek a budov;
- výzkum nových technologií.

Tyto koncepty, jako základní kámen RTS her, se bude naše platforma snažit podporovat a zjednodušit tvůrcům her jejich implementaci.

Hlavní inspirací pro tvorbu platformy, a tím i pro typ her, které bude platforma nejjednodušejí podporovat, byla hra Stronghold Crusader. Na této a dalších hrách ukážeme v následujících částech blíže základní principy RTS her a námi podporované implementace těchto principů.

## Strategie vs. taktika

Jedním z hlavních problémů RTS her je pečlivé vyvážení kombinace strategie/macromanagementu a taktiky/micromanagementu. Tyto pojmy bývají často špatně chápány a někdy zaměňovány, pokusíme se je proto konkrétně definovat. Strategií míníme rozhodnutí týkající se globálního průběhu hry. Taktika naopak zahrnuje konkrétní pozice konkrétních jednotek, jejich pohyb po mapě a spolupráci v jedné bitvě.

Mezi strategická rozhodnutí v RTS hrách patří kupříkladu které budovy hráč postaví, v jakém pořadí dané budovy postaví, které suroviny bude produkovat, které suroviny vynechá, které jednotky bude rekrutovat a které vynechá. Tato 3 rozhodnutí jsou úzce propojena, protože výběr surovin určuje budovy, které bude hráč schopen postavit a typy jednotek, které bude moci zrekrutovat. Stejně tak výběr budov určuje suroviny, které hráč může produkovat a typy jednotek, které může zrekrutovat.

Taktika/micromanagement je v RTS hrách reprezentován ovládáním jednotek, jejich přesné pozice, pohybu, směru útoku, používání schopností atd. Micromanagement lze ale vidět i v ekonomické stránce hry, kdy hráč

Nejlepším příkladem pro rozlišení pojmu strategie a taktiky je série her Total War [38], které kombinuje mód tahové strategie s módem real-time tactics (RTT). V jedné části hry hráč přebírá kontrolu nad celým svým národem, rozhoduje, které budovy budou ve kterých městech postaveny, které jednotky budou rekrutovány a kde budou které armády umístěny. V druhé části, při souboji nepřátelských armád, hráč přebírá kontrolu nad konkrétní armádou a ovládá jednotlivé bataliony, jejich umístění, pohyb a útoky v reálném čase.

Naše platforma nemá za cíl nijak omezovat možnosti vytvářených her v rámci jejich zaměření na taktiku či strategii. Toto rozhodnutí chceme nechat čistě na uživateli naší platformy.

### 1.1.1 Mapy

Reprezentace herní mapy je jedním z hlavních rozhodnutí při tvorbě RTS her. Hlavní funkcí herní mapy je reprezentovat terén pro pohyb jednotek a stavbu budov. Tato funkce může zahrnovat neprostupné části mapy, několik různých druhů terénu prostupné různým druhům jednotek či části mapy ovlivňující rychlosť pohybu jednotek. Při stavbě budov je pak často omezen typ terénu, na kterém je hráč schopen danou budovu postavit. Vzhledem k uzavřenosti většiny RTS her je složité zjistit, jak je v každé z nich herní svět reprezentován, z pozorovatelného vnějšího chování lze ale odvodit několik základních druhů reprezentací.

Nejstarší a nejjednodušší reprezentací je rozdělení mapy na stejně velké dlaždice. Tyto dlaždice mohou být různých tvarů, nejčastěji jsou však čtvercové či hexagonální. Příkladem takového reprezentace je právě hra Stronghold Crusader [9], podle které chceme naši platformu modelovat. Jak je vidět na obrázku 1.1, herní mapa je viditelně rozdělena na stejně velké čtvercové dlaždice. Dále můžeme vidět, že je kamera orientována diagonálně vůči dlaždicím. Účelem této orientace je skrytí toho, že je mapa složena z dlaždic. Každá dlaždice je nějakého typu, který určuje její vzhled, což můžeme vidět v části zvýrazněné červenou barvou. Vidíme zde řady pěti dlaždic stejného typu, oddělené vždy jednou dlaždicí pouštěního typu. Můžeme si všimnout, že i dlaždice stejného typu mohou mít několik různých vzhledů. Typ dlaždice je dále využíván jako omezení při stavbě budov, kde například kamenolom lze postavit pouze na dostatečném počtu dlaždic typu kámen. Toto můžeme vidět v části označené modrou barvou, kde se hráč pokouší umístit budovu, která ovšem nemůže být postavena na dlaždicích typu mokřadu. Dlaždice tohoto typu jsou proto zvýrazněny červenou barvou v rámci půdorysu budovy. Dále můžeme v horní části ukázky vidět dlaždice s rozdílnou výškou od okolního terénu, tvořící nedostupnou oblast mapy. Dále můžeme vidět pro jednotky neprostupný typ dlaždic obsahujících vodu. Uprostřed obrázku poté vidíme velkou oblast kamení a železa, kde oblast kamení ukazuje texturu kvalitně skrývající složení mapy z dlaždic, naopak na oblasti železa jasně vidíme hranice jednotlivých dlaždic, především ve směru z levého dolního rohu do pravého horního rohu.



Obrázek 1.1: Ukázka dlaždic ve hře Stronghold Crusader

Na každé dlaždici může být postavena až na výjimky nejvýše jedna budova a stát nejvýše jedna jednotka. Při pohybu jednotek není tato vlastnost dodržována, lze tedy jednotky přesouvat přes dlaždice na kterých již jiná jednotka stojí.

Naše platforma bude podporovat rozšířenější verzi tohoto druhu mapy, ve které nebudeme vynucovat limity na počty jednotek a budov na jedné dlaždici. Tato omezení budou přenechána pro implementaci tvůrcem her využívajících naši platformu a bude vytvořeno rozhraní pro co nejjednodušší implementaci těchto limitů.

Naše platforma bude podporovat mapu s těmito vlastnostmi:

**M1:** terén rozdělený na čtvercové dlaždice,

**M2:** dlaždice s různou výškou,

**M3:** dlaždice různých typů,

**M4:** možnost dotazovat se na jednotky nacházející se na dlaždici,

**M5:** možnost dotazovat se na budovy postavené na dlaždici.

### 1.1.2 Jednotky

Jednotky jsou základním nástrojem hráče pro boj s nepřítelem. Pohybem po mapě, poškozováním ostatních jednotek a ničením budov jednotky umožňují hráči vést souboj s protivníkem, získat strategickou výhodu a následně vyhrát hru.

#### Pohyb

Hlavním odlišujícím prvkem jednotek od budov je jejich schopnost pohybu. Pohyb jednotky je určen jejím typem, kde každý typ jednotek může procházet jinými typy terénu, pohybovat se nad terénem, na vodě či pod vodou. Naše platforma bude podporovat pohyb jednotek kdekoli nad terénem, navíc bude tvůrcům poskytnuta komponenta umožňující chůzi po terénu, neboť je to nejčastější způsob pohybu.

Pohyb jednotek je nejčastěji řízen hráčem, ať už na úrovni příkazů jednotlivým jednotkám, tak na úrovni slučování jednotek do skupin a ovládání těchto skupin. Naše platforma bude umožňovat jak ovládání jednotlivých jednotek, tak celých skupin. Dále umožníme vývojáři přidat složitější ovládání, například slučování do permanentních formací a následné ovládání těchto formací.

V některých hrách existují také jednotky, které se mohou své schopnosti pohybu vzdát a stát se budovu, buď dočasně, nebo trvale. Příkladem takového jednotky/budovy mohou být budovy Nočních elfů ze hry Warcraft 3 [5]. Tyto entity jsou stavěny jako budovy, tedy jsou umístěny do světa a postaveny jednotkou, stejně jako u všech ostatních ras. Následně je ale hráči umožněno, pomocí speciální ability těchto budov, změnit je dočasně na jednotky a pohybovat s nimi či je dokonce použít pro boj. V tomto módu ale ztrácí všechny funkce budov, tedy není možné je použít pro sběr surovin či produkci jednotek. Následně je možné znovu je zakořenit, čímž získávají zpět své funkce budovy. Naše platforma by měla takové jednotky/budovy také podporovat.

## Umělá inteligence

Ve velké části RTS her jsou jednotky schopny do určité míry autonomního rozhodování bez zásahu hráče, od střelby na cíl, který se ocitne v jejich dostřelu, po vyhledání krytu, pokud jsou pod palbou.

Jako příklad jednoduché umělé inteligence jednotek můžeme vzít hry Starcraft [6] a Warcraft [5] od společnosti Blizzard [36]. Zde se jednotky chovají velice předvídatelně, splňují přesně hráčovi rozkazy a nedělají nic navíc, což James Lantz [16] označuje jako jeden z faktorů, které umožnili hře Starcraft II vytvořit jednu z prvních masivních e-sport scén na světě.

Dobrým příkladem jednotek s vysokou autonomií je série Company of Heroes [26], kde jednotky automaticky vyhledávají krytí, rozutečou se, pokud jsou pod palbou dělostřelectva, a v případě příliš velkých ztrát utečou z boje. Příklad tohoto chování můžeme vidět na obrázku 1.2. V levé části vidíme skupinu jednotek, útočících na nepřátelskou jednotku mimo obraz ve směru červené šipky. Na prostřední části můžeme vidět, že po zničení nepřátele se jednotka začala přesouvat ve směru modré šipky, nejspíše pro získání lepšího krytí a lepší palebné pozice vůči zbývajícím nepřátelským jednotkám. Tento přesun je vykonán bez zásahu hráče, jak můžeme vidět díky modré ikoně nad jednotkou, která značí, že jednotka není zvolena hráčem a není jím tedy ovládána. Oproti tomu v pravé části můžeme vidět, že jednotka hráčem označena je díky bílé barvě ikony nad jednotkou. Vidíme, že hráč vydal rozkaz pro přesun zpět, zvýrazněný zeleně. Tato reakce byla vynucena umělou inteligencí jednotky, která se sama od sebe rozeběhla ve směru nepřátelských linií a donutila hráče reagovat. Jak vidíme, autonomie má svou cenu, a to v nepředvídatelnosti chování jednotek. Při jednoduché umělé inteligenci jednotek je hráč schopen předvídat jejich chování a využít ho pro svůj prospěch. Naopak při složité umělé inteligenci, jako právě v případě Company of Heroes, je hráč často nucen provést více pokusů při vydávání rozkazu, či vydávat rozkazy pro negaci automatického chování jednotky, protože není schopen toto chování jednoduše odhadnout. Tato skutečnost činí hry často realističejší, protože simuluje chování reálných vojáků, kteří rozkaz interpretují a implementují podle svého, není ale vhodná pro souboje více hráčů, a už vůbec né více hráčů na profesionální úrovni.



Obrázek 1.2: Ukázka autonomního chování jednotek ve hře Company of Heroes [26]

Platforma bude umožňovat vykonat libovolný kód v rámci každého výpočtu stavu každé jednotky, bude tedy pouze na vývojáři, zda se budou jednotky chovat jednoduše a předvídatelně, nebo zda budou vykonávat složité, avšak nepředvídatelné úkony bez hráčova vědomí. Pro ulehčení vývoje bude platforma poskytovat předpřipravené komponenty, umožňující základní úkony jako střelbu na cíl, pohyb po mapě a útok na blízko.

## Produkce

Jednou z vlastností definujících RTS hry je možnost produkce nových jednotek. Existuje několik systémů produkce jednotek, úzce svázaných se systémem surovin v dané hře. (viz. 1.1.4) Od kontinuální produkce, kde hráč zvolí produkovány jednotky a suroviny jsou spotřebovávány v průběhu produkce, po diskrétní produkci, kde hráč musí vlastnit všechny suroviny potřebné pro výrobu dané jednotky při začátku produkce a všechny suroviny jsou odečteny v jeden okamžik. Kontinuální systém umožňuje hráči naplánovat produkci armády v předstihu, i když v daném okamžiku nevlastní dostatečné suroviny. Naopak při diskrétní produkci je hráč nucen čekat do chvíle, kdy má všechny suroviny, a až poté může začít s produkcí. Naše platforma se pokusí podporovat oba systémy. Bude záležet pouze na tvůrci hry, jak se k surovinám a produkci jednotek zachová a který z těchto systémů bude implementovat.

Počet jednotek je často limitován, jak pro účely vyvážení hry, tak pro omezení zátěže hardwaru. Z hlediska vyvážení síly jednotek umožňuje limit na počet jednotek předejít tzv. „Zergu“, kdy hráč vytvoří obrovské množství levných jednotek, které následně převálcují jakýkoli odpor. Z hlediska hardwarové náročnosti je účel limitu vcelku zřejmý, protože každá jednotka zabírá určité množství paměti a výpočetního výkonu. Stronghold Crusader omezuje počet jednotek na 1000 pro každého hráče. Toto omezení se jeví především jako limit na hardwarovou náročnost hry. Naše platforma žádné explicitní limity nestanovuje, avšak v uživatelské dokumentaci pro vývojáře budeme silně doporučovat stanovení limitů na počet budov, jednotek a projektů. Za tímto účelem umožníme vývojáři při vytvoření každé jednotky, budovy či projektu učinit rozhodnutí, zda je vytvoření možné a případně toto vytváření zrušit.

Hráč často začíná s malým počtem jednotek, jejichž účelem je zamezit tzv. „Rush“ strategii, ve které je cílem vytvořit co nejrychleji co možná nejvíce levných jednotek a zničit nepřitele ještě před tím, než je schopen začít produkovat své jednotky. Ve hře Stronghold Crusader hráč začíná každou hru s několika lučišníky a kopiníky, jejichž počet je určen v nastavení před začátkem hry. Toto bude v naší platformě umožněno přidáváním jednotek v rámci editace mapy, případně bude tvůrce hry schopen umožnit hráči určit počty jednotek před začátkem hry pomocí grafických elementů v uživatelském rozhraní a následně při začátku hry vytvořit požadované množství jednotek. Tyto jednotky bude poté hráč vlastnit již na počátku hry.

## Boj

V drtivé většině RTS her mají jednotky tzv. „hit pointy“, zkráceně *HP*, které určují počet zásahů, které může jednotka obdržet než bude zabita. S každým zásahem jsou poté tyto *HP* odečítány a v okamžiku, kdy je jednotka poškozena

na 0 *HP* je zabita. Naše platforma bude tento systém samozřejmě podporovat, ale nebude ho nijak explicitně vyžadovat, bude tedy tvůrci umožněno použít jakýkoli jím implementovaný systém.

Jednotky mohou obdržet poškození z mnoha zdrojů, nejčastěji však útokem z blízka (tzv. *melee*) či z dálky (tzv. *ranged*). Útok na blízko je omezen dosahem, rychlostí útoků a velikostí uděleného poškození. Naše platforma bude podporovat komponentu poskytující útoky na blízko právě s těmito parametry. Útok na dálku lze rozdělit do dvou typů, tzv. *hit-scan* a *projektily*. První typ je reprezentován například laserovými zbraněmi, které v okamžiku výstřelu urazí celou vzdálenost, dokud nenarazí na terén či nějakou entitu (budovu či jednotku). Druhý typ v okamžiku výstřelu vytvoří projektil, který se v průběhu času pohybuje herním světem, dokud také nenarazí na terén či nějakou entitu. Naše platforma bude podle předlohy Strongholdu Crusader podporovat především projektilové útoky. Za tímto účelem bude vytvořena komponenta umožňující střelbu projektilů, dále projektily samotné, simulace jejich letu a především možnost výpočtu pro střelbu na pohyblivý cíl. Hit-scan útoky nebudou přímo podporovány, mělo by však být umožněno tvůrci hry tento typ útoků implementovat manuálně.

### Shrnutí požadavků

Naše platforma bude podporovat jednotky s těmito vlastnostmi:

**J1:** pohyb jednotek volně kdekoliv nad terénem,

**J2:** podpora pohybu po terénu,

**J3:** ovládání jednotek a skupin jednotek,

**J4:** rozšířitelnost o složitější ovládání,

**J5:** podpora jednoduché i složité umělé inteligence v podobě vykonání libovolného kódu,

**J6:** podpora diskrétní i kontinuální produkce jednotek,

**J7:** přidávání jednotek při editaci mapy,

**J8:** přidávání jednotek při startu hry,

**J9:** podpora systému hit-pointů,

**J10:** útoky na blízko i na dálku,

**J11:** simulace projektilů.

### 1.1.3 Budovy

Stavba budov představuje jednu z hlavních prezentací hráčovi strategie. Podle postavených budov lze často vcelku přesně odhadnout, jakou strategii hráč zvolil, čímž je umožněno nepřátelům reagovat a adaptovat svou strategii odpovídajícím způsobem. Při volbě strategie lze ale narazit na problém, kdy je hráč nucen zvolit

svou strategii před tím, než naleze protivníky a tedy před tím, než může vidět jejich strategii. Tento problém je velmi výrazný při tzv. „rock-paper-scissors“ strategiích, popisovaných například v knize Fundamentals of Game Design [2, str. 329]. Hlavní myšlenkou tohoto návrhu her je situace, kdy strategie 1 poráží strategii 2, strategie 2 poráží strategii 3 a strategie 3 poráží strategii 1. Ve chvíli kdy naše hra umožňuje hráči zvolit pouze jednu z těchto strategií bez viditelnosti protivníkovy strategie, degeneruje hra v loterii, zda hráč náhodně vybere správnou strategii porážející tu vybranou nepřítelem. Jedním z řešení tohoto problému, navrhovaných v článku od studia Oxeye [24], je co nejmenší rozdíl v síle prvních úrovní technologie, což umožní hráčům reagovat a změnit svoji strategii před tím, než je rozdíl mezi jejich silami neúnosně velký. Dalším možným řešením, použitým ve Stronghold Crusader [9], je neskrývat před hráčem nepřítelovu strategii. Toto řešení lze použít v různé míře, od odhalení nepřátelských budov po úplné odkrytí celé mapy, tedy pozice všech jednotek i budov, ať už přátelských, nepřátelských či neutrálních. Naše platforma použije toto poslední řešení, kdy budeme hráč vidět všechny jednotky a budovy všech hráčů.

Budovy mají ve hrách mnoho funkcí, mezi které patří například:

- Produkce jednotek
- Vylepšování jednotek
- Produkce surovin
- Uskladnění surovin
- Obrana
- Stavba budov
- Zkoumání technologií

Naše platforma se pokusí co nejvíce zjednodušit implementaci těchto funkcí poskytnutím programátorského rozhraní pro umístění budov a jednotek do herního světa, přidání a odebrání surovin hráči či střelbu projektilů. Dále umožníme tvůrci her přístup ke grafickému rozhraní, do kterého bude možné umístit okna, tlačítka a další elementy, které následně použitím programátorské rozhraní budou schopné implementovat všechny zmíněné funkce budov.

Jak již bylo řečeno v sekci o jednotkách 1.1.2, naše platforma bude nechávat volbu mezi kontinuální a diskrétní produkci jednotek na tvůrci hry. Budeme se tedy snažit do co největší míry podporovat oba tyto způsoby.

## Obrana

Obrana bude podporována v podobě komponent, které umožní budovám jak útok na blízko, jako v případě pastí ve hře Stronghold Crusader [9], tak na dálku, jako například obranné věže ve hře Warcraft 3 [5].

Dále v rámci funkce obrany umožňují v některých hrách budovy jednotkám pohybovat se po nich. Jak můžeme vidět na obrázku 1.3 ze hry Stronghold Crusader [9], jednotky mohou být umístěny na hradbách, věžích, bránách či na tvrzi. Na obrázku můžeme vidět různé typy jednotek, umístěné po obvodu hradu pro

jeho obranu. Jak bylo řečeno v části o jednotkách, naše platforma bude umožňovat neomezený pohyb jednotkami, tedy i ve vzduchu, na budovách či skrz budovy. Protože se pohyb po budovách vyskytuje ve velkém množství RTS her, poskytne platforma možnost rozšíření terénu o plochu budov a komponentu umožňující chůzi po těchto plochách.

Umístění na budovách poskytuje jednotkám ochranu před nepřátelskými jednotkami útočícími na blízko, které se musí nejdříve dostat do blízkosti cíle, než zaútočí.



Obrázek 1.3: Jednotky umístěné na budovách ve hře Stronghold Crusader [9]

V některých hrách, kde bohužel Stronghold Crusader [9] není jednou z nich, poskytuje vyvýšení nad terén jednotkám větší dostrel. Tento efekt nemusí být omezen pouze na vyvýšení pomocí budov, ale lze ho dosáhnout už při rozdílných výškách terénu, na kterém je umístěn střelec a jeho cíl, obecněji na rozdílu výšky pozice střelce a cíle, pokud se střelec nemusí pohybovat přímo po terénu. Naše platforma bude podporovat realistické chování projektileů splňující tuto vlastnost.

Budovy, podobně jako jednotky, mají ve většině her určitý počet tzv. „hit pointů“, které určují počet zásahů, které může budova obdržet před tím, než bude zničena. Navíc oproti jednotkám mohou budovy často obdržet poškození pouze od omezené podmnožiny jednotek, nejčastěji pouze obléhacích strojů. Stejně jako u jednotek přenecháme systém poškození na tvůrci hry. Naše platforma pouze umožní budově reagovat na zásah projektilom či zbraní, ať už snížením svých *HP*, nebo ignorováním daného útoku v případě že přišel od jednotky či projektilu, který danou budovu nemůže poškodit.

## Stavba budov

Budovy mají často restrikce, které je hráč nucen splnit před stavbou budovy. Tyto restrikce mohou sahat od reliéfu a typu terénu, přes existenci jiných budov ve stejném místě, po vlastnictví určitého množství surovin či typu jednotek. Naše platforma umožní tvůrci před stavbou budovy zjistit stav všech těchto typů restrikcí a případně vetovat stavbu budovy.

Jednou z restrikcí je výzkum určité technologie či stavba určité předcházející budovy. Tyto možné závislosti jsou blíže popsány v sekci Vývoj technologií 1.1.5. Tímto druhem restrikcí jsou budovy uspořádány do postupně se zlepšujících úrovní, které hráč v průběhu času odemyká. Každá z úrovní obsahuje řadu rozdílných budov, umožňujících zvolit různé strategie. Platforma bude umožňovat tvůrce postupné zpřístupňování budov a jednotek, čímž bude tvůrce schopen implementovat postupné zkoumání nových technologií.

Existující hry využívají několik možností, jak hráči poskytnout zpětnou vazbu o splnění restrikcí při stavbě budovy. Jednoduší možnosti, použitou ve hře Stronghold Crusader [9], je zobrazení půdorysu v různých barvách podle splnění restrikcí. Další, složitější možností je zobrazení průhledného či jinak upraveného modelu budovy na místech, kde ji nelze postavit. Naše platforma bude podporovat jednoduší způsob, tedy zobrazení půdorysu budovy v různých barvách podle požadavků tvůrce hry.

### **Shrnutí požadavků**

Naše platforma bude podporovat tyto vlastnosti:

- B1:** stavbu budov v herním světě,
- B2:** komponenty pro útok budov na blízko i na dálku,
- B3:** rozšiřitelnost dostupného terénu v herní mapě o prostor na budovách,
- B4:** podpora zvýšení dostřelu při umístění jednotky na vyvýšený terén, například budovu,
- B5:** stavba budov při editaci mapy,
- B6:** stavba budov při startu hry,
- B7:** podpora systému hit-pointů,
- B8:** tvůrcem definované reakce na obdržení zásahu budovou,
- B9:** zobrazení půdorysu v různých barvách pro zpětnou vazbu splnění restrikcí,
- B10:** kontrolu požadavků při stavbě budovy,
- B11:** přístup ke grafickému rozhraní, možnost zobrazení elementů hráči.

#### **1.1.4 Suroviny**

„Resource management“ , tedy management surovin, je přítomný ve všech hrách žánru RTS již od jeho vzniku. Od koření v Dune II, přes zlato a dřevo ve Warcraft 3 [5], po všechny typy surovin ve hře Stronghold [9], získávání surovin je jednou z hlavních motivací konfliktu v RTS hrách.

Systémy surovin lze rozdělit podle způsobu získávání a počtu typů surovin. Podle způsobu získávání můžeme systém surovin rozdělit na

- 1) aktivní získávání surovin,

## 2) pasivní získávání surovin.

Při aktivním získávání surovin existuje ovladatelná herní entita, která svým pohybem mezi pozicemi na mapě přináší suroviny. Tento pohyb může být ovládán hráčem, ale nejčastěji dokáže pracovat jednotka samostatně. Příkladem může být Warcraft 3 [5], kde speciální jednotky získávají dřevo a zlato přenášením z lesů/dolů do hráčovy hlavní budovy. Při pasivním získávání přibývají suroviny bez akcí entit, pouze díky vlastnictví určité části mapy nebo druhu budovy. Zdroj bývá nekonečný nebo skoro nekonečný, poskytující suroviny do obsazení nebo zničení zdroje. Každý z těchto stylů podporuje jinou strategii kontroly mapy.

Naše platforma bude podporovat jak pasivní, tak aktivní získávání surovin. Bude pouze na tvůrci, v jakém okamžiku budou suroviny přičteny, ať už v závislosti na čase nebo na pohybu určitých jednotek.

### Shrnutí požadavků

Naše platforma bude podporovat tyto vlastnosti:

**S1:** aktivní i pasivní získávání surovin,

**S2:** rozhraní pro přidání a odebrání surovin hráči.

### 1.1.5 Vývoj technologií

Volba vyzkoumaných technologií důležitou součástí strategické části RTS her. Technologie jsou často uspořádány ve stromové struktuře či orientovaném acyklickém grafu (DAG), kde vyzkoumání technologie v rodičovském uzlu odemyká technologie následujících uzlech. Příklad takového uspořádání můžeme vidět v ukázce ze hry *Civilisation V* [8]1.4, kde vidíme počátek stromu technologií. V této hře jsou technologie uspořádány do DAGu, začínajícího v jednom kořeni. Můžeme vidět žluté vrcholy, značící vyzkoumané technologie, dále zelené vrcholy, značící technologie, které mají splněné všechny předky a mohou být vyzkoumány, černé technologie, které bude možné začít zkoumat po odemčení všech předků, a nakonec červené technologie, značící technologie v dalším věku. Dále můžeme vidět hrany spojující závislé technologie.

Další možné uspořádání je několik disjunktních stromů technologií, kdy je hráč nucen zvolit jeden z těchto stromů. Toto uspořádání můžeme vidět ve hře Company of Heroes [26], kde jsou hráči dostupné tři vzájemně výlučné cesty, každá zaměřená najinou oblast boje. Každá z těchto cest je dále rozdělena na dvě větve postupně se zlepšujících technologií.

Každé větvení v grafu technologií představuje možné rozhodnutí hráče, kterou z větví se hráč vydá a které technologie odemkne. Toto rozhodnutí jsou jedním z hlavních projevů hráčovi strategie.

Vyzkoumání technologie může mít mnoho různých efektů. Nejčastějším efektem bývá odemknutí nového typu jednotek nebo budov. Další možností je změna vlastností již vlastněných jednotek nebo budov. V neposlední řadě pak může vyzkoumání technologie odemknout nové schopnosti nebo kouzla, které hráč může následně použít při taktických soubojích.

Explicitní strom technologií, viditelný na 1.4, se v RTS hrách vyskytuje spíše výjimečně. Nejčastěji je odemykání nových typů jednotek a budov umožněno



Obrázek 1.4: Výřez stromu technologií ze hry Civilisation V [8]

stavbou určitého typu budovy nebo dosažení určitého stupně vylepšení již existující budovy. Jako příklad můžeme vzít Warcraft 3 [5], kde závislosti budov na stupních vylepšení a existenci jiných budov tvoří strom technologií. Graf tvořený těmito závislostmi můžeme vidět na obrázku 1.5. Zde vidíme stupně vylepšení budov, reprezentované šedými šipkami, a závislosti budov na ostatních budovách a stupních jejich vylepšení. Aby byl hráč schopen postavit budovu, musí vlastnit všechny budovy na kterých je tato budova závislá v požadovaném nebo lepším stupni vylepšení. Ve hře je tento graf reprezentován požadavky zobrazenými při najetí myší na ikonu zamčené budovy.



Obrázek 1.5: Budovy a závislosti mezi nimi tvořící obdobu stromu technologií ve hře Warcraft 3

Jak bylo řečeno v sekci o budovách 1.1.3, naše platforma bude umožňovat tvůrci postupné odemykání budov a jednotek, což umožní tvůrci her implementovat výzkum nových technologií. Pro technologie měnící vlastnosti typů jednotek bude pouze na tvůrci, aby změnil logiku hry a chování umělé inteligence v závislosti na vyzkoumaných technologiích.

### Shrnutí požadavků

Naše platforma bude podporovat tyto vlastnosti:

textbf{T1}: postupné odemykání budov a jednotek,

textbf{T2}: změny chování a vlastností jednotek v průběhu hry.

## 1.2 Uživatelé

V předešlé sekci jsme popsali druh her, který bude naše platforma podporovat. V této sekci popíšeme požadavky na chování platformy z pohledu všech druhů našich uživatelů.

### 1.2.1 Tvůrci her

Prvním druhem uživatele budou tvůrci her, používající naši platformu pro zjednodušení své práce a vyřešení problémů opakujících se ve většině RTS her. Tvůrce samozřejmě nemusí být pouze jeden člověk, vzhledem ke složitosti RTS her existuje při jejich tvorbě mnoho rolí, které požadují velmi různorodé schopnosti a mohou být plněny více lidmi. Příkladem rolí může být 3D umělec tvořící modely a animace, producent hudby, tvůrce umělé inteligence, programátor herní logiky a nakonec člověk, který toto vše integruje dohromady.

#### Programátor

Naším hlavním cílem je umožnit tvůrcům umělé inteligence a programátorům herní logiky použít všechny jazyky .NET Frameworku, především C#, pro jejich práci. Z programátorského pohledu bude naše platforma sloužit jako knihovna poskytující tyto funkce:

- 1) udržování aktuálního stavu hry a zjišťování tohoto stavu;
- 2) vytváření nových jednotek, budov a projektilů;
- 3) registrování metod, které budou zavolány při určitých událostech v průběhu hry;
- 4) úpravu terénu jak v editoru, tak při běhu hry;
- 5) vykreslování terénu, jednotek, budov, projektilů a grafického rozhraní;
- 6) ukládání a načítání stavu hry;
- 7) implementaci řešení pro základní problémy jako pohyb po terénu, let projektilů a výpočet úhlu střelby projektilů;
- 8) ovládání kamery.

Protože implementace vykreslování a grafického rozhraní by byla nad rámec jedné bakalářské práce, využije naše platforma pro tyto účely existující herní engine UrhoSharp. Pro programátory následně poskytneme přístup k relevantním částem UrhoSharp enginu.

Platforma bude poskytovat ovládání kamery v několika módech. Tato funkcionality bude potřebná již pro implementaci editoru, poskytneme ji tedy i tvůrcům her. Prvním módem bude klasická RTS top-down kamera, jakou můžete

vidět na ukázce ze hry Company of Heroes 1.6. Kamera je umístěna v dostatečné vzdálenosti pro zobrazení celých skupin jednotek, skloněna zhruba 45 stupňů od horizontální polohy. Ve hře Company of Heroes je ovšem oproti jiným starším RTS hrám jako Warcraft 3 nebo Age of Empires 3 s kamerou možné rotovat a přiblížovat, což nám přijde jako velice atraktivní. Příklad tohoto pohybu kamery můžete vidět na obrázku 1.7, kde můžeme vidět pohled z Německé strany na vylodění na pláži Omaha. Toto bychom rádi podporovali i v naší platformě. Druhým módem kamery bude tzv. „free-float“, kdy kamera volně „létá“ nad terénem. Třetím módem bude sledování jednotky, kdy se kamera bude chovat jako v prvním módu, její pohyb bude ale řízen pohybem jednotky.



Obrázek 1.6: Klasický RTS pohled kamery ve hře Company of Heroes



Obrázek 1.7: Co lze vidět s pouhým přiblížováním a otáčením kamery hře Company of Heroes

## Moddeři

Pro integraci vytvořených jednotek, budov, dlaždic a jejich modelů, textur a logiky umožní naše platforma vytvořit *balíček*, obsahující vše vytvořené programátory a umělci. Tento balíček poté platforma umožní přidat a načíst v libovolné instalaci platformy.

Platforma bude také sloužit jako editor úrovní, které budou využívat logiku, jednotky, budovy a typy terénu dodané pomocí balíčků. Vytvořené úrovně bude následně možné uložit zpět do balíčku a distribuovat spolu s balíčkem do dalších instancí naší platformy.

Pro editaci mapy poskytne platforma několik základních nástrojů a umožní tvůrcům modifikovat nástroje či přidávat své vlastní. Základní nástroje by měli umožnit editaci terénu mapy (změnu typu terénu na všechny tvůrci definované typy a editaci výšky dlaždic) a přidávání všech druhů jednotek a budov do mapy.

Formát ukládání úrovní bude definován otevřeně pomocí prostředků nezávislých na programovacím jazyce, címž chceme umožnit tvorbu separátních editorů map produkujících úrovně v námi používaném formátu.

### 1.2.2 Hráči her

Z pohledu hráče se bude platforma chovat jako instalovatelná aplikace, která hráči umožní za běhu přidávat balíčky a následně využít jejich obsah.

Při běhu platforma umožní plný přístup k nastavení UrhoSharp enginu, tedy k nastavení rozlišení, Vsync, triple buffer a dalších. Dále umožní správu balíčků, tedy přidávání, odebrání a spouštění. Při spuštění balíčku umožní platforma hráči výběr z existujících map, jak pro hraní, tak pro editaci. Dále platforma umožní vytvoření a editaci úplně nové mapy.

Při tvorbě nové mapy či editaci existující mapy mít bude hráč přístup ke všem jednotkám, budovám a typům terénu, ke kterým mu dají editační nástroje specifikované tvůrcem balíčku přístup. Následně bude hráči umožněno mapu uložit, a to přepsáním zdrojové mapy, kterou hráč načetl k editaci, nebo vytvořením nové mapy pod novým jménem.

## 1.3 Ukázková hra

Pro ukázkou bude vytvořen balíček s jednoduchou hrou demonstrující možnosti naší platformy. Ukázková hra bude zároveň sloužit jako referenční příklad použití naší platformy.

**Ukázková hra bude obsahovat několik jednotek, demonstrujících tyto vlastnosti:**

- 1) jednoduchou a složitější umělou inteligenci;
- 2) plně automatické jednotky, neovladatelné hráčem;
- 3) jednotky útočící na dálku;
- 4) jednotky útočící na blízko;
- 5) aktivní získávání surovin;

- 6) pohyb po terénu;
- 7) pohyb nad terénem (létání);
- 8) rozdílnou rychlosť pohybu různých jednotek;
- 9) rozdílnou přístupnost částí mapy pro různé jednotky;
- 10) pohyb po budovách.

**Demonstrovanými vlastnostmi budov budou:**

- 1) restrikce na umístění stavby,
- 2) neprostupnost budov pro některé jednotky;
- 3) produkce surovin budovami;
- 4) přidání plochy nebo části plochy budovy jako přístupný terén pro určité typy jednotek.

**Jako obecné vlastnosti bude ukázková hra demonstrovat:**

- 1) RTS mód kamery,
- 2) volný pohyb kamery,
- 3) sledování jednotky kamerou,
- 4) tvůrcem definované prvky v uživatelském rozhraní,
- 5) minimapu.

**Balíček obsahující ukázkovou hru bude demonstrovat tyto vlastnosti:**

- 1) tvorbu vlastních úrovní za použití jednotek, budov a typů terénu obsažených v balíčku;
- 2) ukládání a načítání hry.

## 1.4 Cíle práce

Cílem této práce je vytvořit platformu pro vývoj 3D RTS her pro jednoho hráče za použití herního enginu UrhoSharp, umožňující vývojářům vytvářet hry jako separátně distribuované balíčky, které bude poté koncový uživatel schopen přidat do naší platformy nainstalované na uživatelském počítači a použít je pro hraní dodaných úrovní či tvorbu svých vlastních.

Při tvorbě hry bude umožněno tvůrci použít jazyky frameworku .NET pro vytvoření Umělé inteligence jednotek, budov a nepřátelských hráčů, pro vytvoření další logiky hry a pro přidání nástrojů do editoru map.

**Požadované vlastnosti platformy:**

- 1) podporované vlastnosti jednotek:
  - a) pohyb jednotek (J1, J2)

- b) ovládání jednotek (J3, J4)
- c) umělá inteligence, definice chování (J5)
- d) produkce jednotek (J6)
- e) přidávání jednotek jako součást úrovně (J7, J8)
- f) podpora systému hit-pointů (J9)
- g) útoky na blízko i na dálku (J10, J11)

2) podporované vlastnosti budov:

- a) stavba budov v herním světě (B1, B9, B10)
- b) podpora obraných budov (B2, B4)
- c) rozšiřitelnost dostupného terénu o plochu budov (B3)
- d) přidávání budov jako součásti mapy (B5, B6)
- e) zničitelnost budov (B7, B8)
- f) produkce jednotek, surovin, stavba budov pomocí budovy (B11)

3) podporované vlastnosti surovin:

- a) přidávání a odebírání libovolného množství surovin (nejen celočíselných) (S1, S2)

4) podporované vlastnosti výzkumu technologií:

- a) postupné odemykání dostupných jednotek a budov, umožnit změny chování jednotek za běhu (T1, T2)

5) podporované vlastnosti mapy

- a) rozdelení mapy na dlaždice (M1)
- b) tvorbu terénu pomocí různých výšek dlaždic. (M2)
- c) různé typy dlaždic. (M3)
- d) rozhraní pro získání aktuálního stavu mapy (jednotek a budov na dlaždici, typu dlaždice ...) (M4, M5)

6) vlastnosti pro tvůrce balíčků:

- a) platforma musí umožňovat přidávání balíčků za běhu, obsahujících nové typy jednotek, budov, dlaždic, projektilů a hráčů spolu s jejich modely, texturami a AI
- b) platforma musí umožňovat použití přidaných balíčků pro tvorbu map a uložení vytvořených map do balíčku použitého pro jejich tvorbu
- c) editor map musí být rozšiřitelný o nástroje z balíčku
- d) herní grafické rozhraní musí umožňovat tvůrci přidávat vlastní okna, tlačítka a další prvky

7) vlastnosti pro koncového hráče:

- a) uživatelské rozhraní pro stolní počítače, umožňující vybírání balíčků, map a oponentů, dále načítání a ukládání her, a nastavování zobrazení hry
- b) herní uživatelské rozhraní musí obsahovat minimapu, poskytující hráči přehled o větší části mapy než kterou vidí vlastní kamerou
- c) ovládání kamery umožňující klasický top-down pohled, volné poleto-vání kamery po mapě a následování jednotky
- d) ukládání a načítání hry

## 2. Analýza

V první kapitole jsme specifikovali cíl naší práce, tedy implementaci platformy založené na herním enginu UrhoSharp umožňující tvorbu RTS her a jejich distribuci. V této kapitole popíšeme problémy při implementaci této platformy, námi zvolená řešení a jejich alternativy.

### 2.1 Herní engine

Jak jsme napsali v sekci 1.4 Cíle práce, naším cílem je vytvořit platformu pro tvorbu RTS her za použití herního enginu UrhoSharp. „*UrhoSharp je multiplatformní 3D a 2D engine který může být použit pro tvorbu animovaných 3D a 2D scén za použití modelů, materiálů, světel a kamer*“ [40], jak říká úvodní stránka dokumentace enginu. Jak už název napovídá, UrhoSharp je .NET binding pro Urho3D engine [28], což je opensource herní engine implementovaný v C++.

Tento engine a jeho binding do jazyka C# jsme si vybrali především kvůli tvůrcům .NET bindingu, společnosti Xamarin, která je také autorem implementace .NET Frameworku Mono. V rámci tohoto vztahu očekáváme největší podporu práce s managed kódem, například jeho načítáním za běhu, na čemž je postaven náš systém pluginů.

Jak uvidíme v následující části, přestože je engine UrhoSharp navržen pro platformu Mono, nebylo na všech systémech možné překonat jejich omezení.

### 2.2 Rozdíly systémů

Hlavním cílovým systémem naší práce jsme se rozhodli zvolit systém Windows, především kvůli nejrozsáhlejší podpoře frameworku .NET, dále kvůli zřejmým výhodám ovládání pomocí klávesnice a myši, a v neposlední řadě kvůli naší zkušenosti s tímto systémem. Pro podporu dalších plafotrem je naším cílem vytvořit návrh platformy umožňující co nejjednodušší rozšíření na tyto systémy. Herní engine a platforma .NET sice mnohé rozdíly systémů abstrahuje a umožňuje multiplatformní řešení, existují ovšem oblasti, které i při využití těchto abstrakcí vyžadují pro každý systém specifické řešení.

Implementace pro mobilní systémy (iOS, Android) naráží oproti PC systémům na několik problémů, vycházejících především z rozdílných operačních systémů, velikostí obrazovek a způsobu přijímání vstupu od uživatele. Rozdíly mezi PC systémy (Windows, různé distribuce Linuxu, macOS) nejsou tak rozsáhlé, přesto se mohou vyskytnout problémy především kvůli různým implementacím platformy .NET používaných mimo systém Windows.

Při řešení těchto problémů jsme narazili na rozdíly, specifika a omezení, které v následujících částech přiblížíme.

#### 2.2.1 Zobrazení a ovládání

Na mobilních systémech je vztah mezi GUI, tedy grafickým uživatelským rozhraním, a ovládáním mnohem bližší. Oproti PC systémům je zde nejčastěji jedi-

ným možným vstupem dotyková obrazovka. GUI musí tedy sloužit jak pro zobrazení informací hráči, tak pro získání převážné většiny vstupu od hráče. Dalším rozdílem je velikost obrazovky, která je obecně mnohem menší než u jiných systémů. Ze statistik zařízení od společnosti Google [12] můžeme vidět, že rozpětí velikostí obrazovek tzv. „smartphonů“ používajících systém Android sahá od úhlopříčky 3,5 palce po 6,4 palce, kde nejčastější velikostí je rozpětí od 5 palců po 6 palců. Oproti tomu nejčastější velikosti obrazovek laptopů sahají od 11 in po 17 in a více.

I přes to, že herní engine umožnuje tvorbu grafického rozhraní použitelného na všech systémech, různé druhy vstupu a velikostí obrazovek nutí nás i potencionální tvůrce her na naší platformě k výrazně odlišnému návrhu rozhraní pro mobilní systémy. Engine Urho3D poskytuje separátní vývojové prostředí pro návrh scén a uživatelského rozhraní, které je následně možné exportovat do XML souboru, který je poté možné načíst za běhu hry pro zobrazení specifikovaného uživatelského rozhraní. Tento postup lze použít k definici uživatelského rozhraní specifického pro cílový systém dané verze aplikace, tedy naší platformy. Oproti tomu by tvůrci balíčků byli nuceni distribuovat definice rozhraní pro všechny systémy, z kterých by poté za běhu vybírali podle aktuálního systému, nebo by byli nuceni vytvářet a distribuovat několik separátních balíčků, cílených vždy pro jediný systém.

Ukázku těchto problémů a jedno z možných řešení můžeme vidět na příkladu ze hry Hearthstone [7], kde 2.1 ukazuje mobilní verzi hry a 2.2 ukazuje PC verzi hry. Přestože návrh této hry je ideální pro přenos na mobilní zařízení, což můžeme vidět například ve velice podobném designu vlastní herní plochy v obou verzích, existují mezi PC a mobilní verzí viditelné rozdíly. Jedním z rozdílů je pozice kamery, která je v mobilní verzi umístěna mnohem blíže herní ploše. Toto je jedno z možných řešení problému menší velikosti obrazovek mobilních zařízení, díky kterému budou herní prvky na těchto obrazovkách zobrazeny ve větší velikosti za cenu zobrazení menší části herního světa. Další součástí řešení problému velikosti obrazovek je velikost fontu, která je v mobilní verzi mnohem větší než u PC verze. Zároveň s touto změnou musí být také provedena odpovídající změna designu karet a oblastí, kde se písmo a číslice vyskytují. Řešení problému vstupu pomocí dotykové obrazovky můžeme vidět na obrázku 2.1, který ukazuje dva různé stavy hry. První stav, který můžeme vidět na obrázku 2.1a, je navržen zobrazení celkového stavu hry a pozorování nepřátelských tahů. V tomto stavu jsou hráčovy karty zmenšené a schované v pravém dolním rohu obrazovky, kde nezakrývají žádnou část herní plochy. Při kliknutí na karty následně rozhraní přechází do druhého stavu, který můžeme vidět na obrázku 2.1b. V tomto stavu jsou karty hráče přemístěny do centrální pozice a zvětšeny. Tímto přemístěním sice zakryjí velkou část herní plochy, ale v danou chvíli jsou právě tyto karty cílem hráčovi pozornosti a jimi zakrytá plocha je pro něj irrelevantní. Tento systém vytahování, přemisťování a zvětšování ovládacích prvků aplikace je obecným trendem v mobilních zařízeních, umožňujícím větší velikost ovládacích prvků za cenu většího počtu interakcí uživatele se zařízením oproti počtu interakcí v PC verzi aplikace pro dosažení stejného cíle. Toto můžeme vidět na obrázku 2.2, který ukazuje stejnou situaci v PC verzi hry. Mezi 2.2a a 2.2b nejsou karty nijak přesouvány či zvětšovány a jsou viditelné neustále, hráč tedy může zahrát kartu bez předchozího kliknutí na schované karty, čím ušetří oproti mobilní verzi jednu



(a) Bez interakce hráče.



(b) Při interakci hráče.

Obrázek 2.1: Uživatelské rozhraní mobilní verze hry Hearthstone.



(a) Bez interakce hráče.



(b) Při interakci hráče.

Obrázek 2.2: Uživatelské rozhraní PC verze hry Hearthstone.

interakci.

Jak můžeme vidět na příkladu ze hry Hearthstone [7], vedou nás problémy s velikostí obrazovek a dotykovým ovládáním k separátnímu designu a implementaci uživatelského rozhraní a některých částí her. Pro tuto separátní implementaci jsme v naší práci připravili základní kostru, upustili jsme ovšem od konečné implementace z důvodu nedostatku času.

### 2.2.2 Kompilace

Dalším rozdílem, tentokrát s rozdílným chováním i mezi jednotlivými mobilními systémy, je jejich chování k spustitelným souborům aplikací. Jak píší Joseph a Ben Albahari [3, str. 3,4], jsou jazyky cílené na platformu .NET překládány do „Common Intermediate Language“ (CIL), z kterého jsou obvykle až za běhu aplikace kompilovány do instrukční sady stroje, na kterém právě běží. Tento způsob se označuje jako „Just-In-Time“ (JIT) komplilace, a je standardním způsobem spouštění .NET aplikací. V některých případech je ale použit jiný způsob, a to tzv. „Ahead-of-time“ (AOT) komplilace, kdy je CIL kód ještě před distribucí zákazníkovy zkompilován do instrukční sady cílového stroje a následně je distribuována tato již zkompilovaná verze. Tento způsob je používán pro zrychlení odezvy při větších velikostech assembly, čímž se předchází zpoždění v důsledku komplilace CIL kódu. Další využití, zde již ne pouze za účelem optimalizace, ale vynucené systémem samotným, je při distribuci na systém *iOS* [23]. Jak je napsáno v *iOS Security Guide pro iOS verze 12.1* [4, str. 27], není možné alokovat paměť zároveň jako „*writable*“, tedy s možností do ní zapisovat, a „*executable*“

, tedy s možností v ní uložená data vykonávat přímo jako instrukce procesoru. Tato skutečnost vylučuje jakékoli použití JIT komplikace, která používá právě takto namapované stránky jako výstup komplikace z *intermediate* jazyka, tedy například CIL, do instrukční sady procesoru. Výjimkou jsou aplikace společnosti Apple, podepsané jejich klíčem, kterým je umožněna jedna alokace (jedno zavolání funkce `mmap`) takto namapovaných stránek. Tato výjimka je použita ve webovém prohlížeči Safari pro implementaci Javascript JIT komplikátoru.

Tato skutečnost znemožňuje naší platformě jednoduché nahrání assembly pomocí reflexe a nutila by nás k složitějšímu řešení, které by se podle aktuálního systému muselo rozhodovat, kterou verzi assembly nahrát. Navíc by tento způsob nutil tvůrce balíčků přeložit svůj kód pro všechny možné architektury. Toto je důvod, proč jsme upustili od podpory systému iOS.

### 2.2.3 Souborové systémy

Každá aplikace má několik odlišných druhů souborů. Tyto druhy můžeme odvodit z pro ně navržených adresářů ve Windows API [17], či z implementace tohoto API v platformě .NET [18]. Těmito druhy souborů jsou:

- *Roaming user data* - data uživatele přítomná na všech počítačích v síti, na které se může uživatel přihlásit;
- *Local user data* - data uživatele lokální pro aktuální počítač;
- *Private app data* - data aplikace přístupná pouze aplikaci;
- *Public app data* - data aplikace přístupná všemi aplikacemi;
- *Static app data* - neměnná data aplikace distribuovaná spolu s aplikací.

Některé z těchto druhů souborů jsou na různých systémech sloučeny. Systém Android například nepodporuje více uživatelů, čímž ztrácí smysl rozdělovat uživatelská data a data aplikace. Přístup aplikací k souborovému systému je na tomto systému dále omezen. Jsou definována čtyři místa, kam může aplikace ukládat data [11]. Těmito místy jsou:

- 1) *internal file storage*,
- 2) *external file storage*,
- 3) *shared preferences*,
- 4) *databases*.

*Internal file storage* je interní úložný prostor zařízení. Každá aplikace má zde systémem vytvořenou složku, do které má přístup pouze aplikace samotná. Tato složka je odstraněna při odinstalování aplikace.

*External file storage* je „externí“ úložný prostor, není tedy garantováno, že bude vždy přítomný. Tento prostor může být tvořen jak vestavěnou pamětí zařízení, tak fyzicky vyjmateLNým prvkem, jako například SD kartou. Soubory na tomto úložišti jsou veřejně přístupné a nejsou odstraňovány při odinstalování aplikace.

*Shared preferences* a *Databases* slouží pro ukládání dat bez explicitního využití souborových systémů. Tento přístup k datům nemá na PC systémech přímou podporu.

Dalším rozdílem mezi systémy je přístup k souborům distribuovaným spolu s aplikací, v našem případě s naší platformou. Na systému Android je každá aplikace distribuována jako .apk soubor. Formát Apk je zip archiv, obsahující všechny soubory naší aplikace, od kódu, přes assety, po preference. Tento archiv je v duchu Linuxového VFS přímo namapován do stromu souborového systému viditelného z naší aplikace. Bohužel .NET filesystem API nedokáže s tímto mapováním pracovat, tedy není možné ho využít pro čtení těchto souborů. Řešením je využití Xamarin.Android zabalujícího Android Java API, které s tímto archivem pracovat dokáže.

Přístup k adresáři aplikace má další rozdíl, a to v zápisu do souborů. Na systému Android je zápis do těchto souborů úplně zakázán. Na PC systémech může být pro některé uživatele možné zapisovat do těchto souborů, ale aplikace by s tímto přístupem neměla počítat.

Všechny tyto rozdíly při přístupu k souborům nás nutí k implementaci separátní komponenty pro práci se soubory, která je implementována pro každý systém zvlášt a následně poskytována přenositelné části platformy.

#### 2.2.4 Shrnutí

I když je naším cílem pouze implementace pro systém Windows, ukázali jsme, že rozšíření na některé další systémy by nebyl problém a nastínili jsme řešení možných problémů, které by při tomto rozšíření vznikly. Návrh naší implementace zohledňuje tato řešení a umožňuje jejich budoucí implementaci. Vzhledem k velikosti implementace pro systém Windows nebudou ovšem tato řešení pro zbylé systémy součástí naší práce. Dále jsme zjistili, že systém iOS je neslučitelný s požadavky naší aplikace a rozšíření na tento systém tedy nebude možné.

### 2.3 Formát a načítání dat

Důležitou součástí implementace naší platformy je systém balíčků pro distribuci vytvořených her. Tyto balíčky obsahují všechny součásti hry, od modelů a textur, přes logiku a umělou inteligenci, po mapy a úrovně vytvořené tvůrcem hry. Všechny tyto součásti musí naše platforma být schopna načíst za běhu a použít jak pro tvorbu nových map, tak pro hraní již existujících.

#### 2.3.1 Struktura balíčku

Pro implementaci načítání balíčků musíme definovat strukturu, kterou budou balíčky splňovat, a podle které bude platforma určovat typy souborů a jejich závislosti.

První možností je založit strukturu balíčku na pevné adresářové struktuře, kde každý balíček bude tvořen jedním adresářem obsahujícím další pevně specifikovanou podadresářovou strukturu. Jednotlivé typy zdrojů, tedy 3D modely, textury, popis jednotek nebo skripty, by následně byly rozděleny a identifikovány touto adresářovou strukturou.

Pro popis typů jednotek, budov, projektilů, dlaždic a nepřátele jsme se inspirovali v existujících hrách, ať už Civilization V [8] nebo Kerbal Space Program [21], a využili jsme XML soubor pro definici závislostí. Dalšími možnostmi pro popis typů entit bylo využít formát JSON nebo dokonce definovat vlastní formát. Oproti ostatním zmíněným formátům má formát XML vestavěnou podporu přímo v platformě .NET. Navíc tento formát umožňuje automatickou validaci vůči schématu, což nám ulehčí validaci načtených dat. Možnou výhodou formátu JSON je jeho expresivita, umožňující minimalizovat velikost souborů. Vzhledem k velikosti ostatních druhů dat jsme ovšem usoudili, že tato výhoda není dostačující pro volbu tohoto formátu.

Implementace pomocí pevné adresářové struktury a XML souborů pro popis typů entit ovšem vedla ke dvěma problémům. Prvním bylo velké množství malých XML souborů, jejichž správa by mohla vést k častým omylům a následným chybně pojmenovaným souborům, špatným cestám a chybějícím souborům. Druhým problémem bylo přidávání balíčku do běžící hry. Hráč by mohl sice specifikovat adresář reprezentující balíček, následné ověření, zda je tento balíček korektní a lze ho nahrát by nás ale nutilo k procházení adresářové struktury, k pokusům o jejich načtení a validaci. Takováto implementace by byla pomalejší, náročnější na správu a náchylnější k chybám.

Řešením bylo vytvořit centrální XML soubor, definující celý balíček. Všechny typy jednotek, budov, projektilů, nepřátele, logik úrovní, všechny úrovně obsažené v balíčku a další jsou popsány v tomto souboru pomocí stejného XML, jako byly v separátních souborech. Následně všechny assety, tedy modely, textury či assembly mohou být specifikovány relativní cestou vůči adresáři obsahujícímu tento jeden XML soubor. Tímto způsobem lze replikovat předchozí uspořádání, kde je každý typ assetů rozdělen do vlastního adresáře, ale navíc tento způsob umožňuje tvůrce balíčku specifikovat vlastní rozdělení a umístění assetů. Zároveň toto uspořádání ulehčuje přidání balíčku a ověření jeho korektnosti, kde stačí, aby uživatel zadal cestu k tomuto XML souboru, a pouhou validací podle schématu lze ověřit jeho správnost.

Naším finálním řešením je tedy reprezentovat každý balíček jedním XML souborem, který dále obsahuje relativní cesty odkazující na zbylý obsah balíčku. Tento soubor má formát daný pevným schématem a tento formát je kontrolován při každém načítání.

### 2.3.2 Data entit

Nyní již víme, jak data rozdělit a odkazovat na ně. Následně musíme určit, jaká data budeme u jednotlivých typů entit požadovat a jaká vlastní data umožníme tvůrcům her si zde uložit. Naším cílem je umožnit specifikování typů entit a k nim náležejících dat, definujících vlastnosti těchto typů entit.

Naše platforma bude poskytovat funkcionality společnou většině her typu specifikovaného v úvodní části 1. Tato funkcionality zahrnuje:

- 1) načítání pluginů a volání jejich metod,
- 2) uživatelské rozhraní,
- 3) správa balíčků,

- 4) vykreslování herního světa.

Naše platforma bude vynucovat specifikaci nezbytných dat pro implementaci poskytované funkcionality. Těmito daty jsou:

- 1) assembly;
- 2) ikony, barvy pro zobrazení na minimapě;
- 3) identifikátory a jména balíčků, typů entit, úrovní, logik hráčů;
- 4) 3D modely, textury, animace.

Protože jsou tyto vlastnosti společné většině entit v námi podporovaných typech her, nahrává naše platforma automaticky zde specifikované vlastnosti při vytvoření entity v herním světě a následně je využívá pro implementaci poskytovaných herních prvků.

Dále umožníme přidat libovolná další data do XML elementu reprezentujícího typ entity pro použití tvůrcem hry v jeho implementaci logiky. Tato data nebude naše platforma nikak validovat, pouze je při vytvoření entity v herním světě předá načítanému pluginu implementujícímu logiku vytvářené entity. Bude pouze na tvůrce této logiky, aby získaná data validoval a následně z nich inicializoval logiku či použil systém výjimek k oznámení chyby v datech. Tento způsob implementace umožní tvůrcům vytvářet universálnější logiku, kterou budou schopni odlišit právě načítanými daty ze souboru. Bez této funkcionality by byli tvůrci her nutenci všechna data přesunout z XML souboru přímo do kódu logiky.

### 2.3.3 Formáty assetů

V předešlé části jsme popsali, jak náš systém balíčků umožňuje zaznamenat umístění dat potřebných pro spuštění úrovně. V této části dále upřesníme, jakých formátů můžou tato data být a jaká jsou omezení při použití určitých formátů dat. Slovem „asset“ označujeme jakákoli data, které může tvůrce hry poskytnout naší platformě a použít je při tvorbě hry. Tato data lze rozdělit do několika kategorií:

- grafická data,
- logika a pluginy,
- ostatní.

V následujících částech popíšeme jednotlivé kategorie dat, jejich formáty a využití.

#### Grafická data

Mezi grafická data patří především 3D modely a k nim náležící textury a animace. Podporované formáty těchto dat jsou omezeny námi používaným herním enginem Urho3D. Tento engine interně používá knihovnu Open Asset Import Library (Assimp) [1], která umožňuje načítání mnoha formátů 3D modelů do uniformní reprezentace, která je následně používána enginem samotným.

Mezi další grafická data používaná naší platformou jsou textury dlaždic. Každý typ dlaždic, jak je popsáno v části 2.3.1 o struktuře balíčku, je definován XML elementem. Tento element povinně specifikuje texturu, představující vzhled dlaždic tohoto typu. Kvůli naší implementaci zobrazení mapy, blíže popsaného v části 2.4.1, musí být tato textura manipulovatelná z C# kódu naší platformy. Bohužel engine Urhosharp v aktuální verzi neposkytuje přístup k dekompresi textur, implementované enginem Urho3D. Z tohoto důvodu není možné využít komprimované formáty textur pro vzhled dlaždic.

Dalšími povinnými grafickými daty, které naše platforma požaduje, jsou textury ikon pro jednotky a budovy. Pro každý z těchto dvou druhů entit požadujeme texturu, která je následně použita při implementaci nástrojů poskytovaných naší platformou, umožňujících editaci a hraní úrovní. Tyto nástroje umožňují umisťovat jednotky a budovy do editované úrovně a dále při jejím hraní vybírat skupiny jednotek a vydávat jim rozkazy. Definice každého typu jednotky nebo budovy následně obsahuje specifikaci části těchto textur, která má být využita pro reprezentaci tohoto konkrétního typu. Přestože platforma umožňuje tvůrcům her implementovat vlastní nástroje pro editaci a hraní her, předpokládáme, že i tyto nástroje budou potřebovat reprezentovat jednotlivé druhy jednotek a budov a využijí k tomu právě tyto ikony.

Grafická data obsažená v balíčku nejsou omezena pouze na platformou požadovaná a používaná data, ale je umožněno tvůrci přibalit libovolná data podporovaná enginem UrhoSharp a následně je využít při hře. Díky tomu, že mají tvůrci her plný přístup k schopnostem enginu UrhoSharp, mohou, stejně jako platforma, za běhu použít tato data pro úpravu vzhledu jednotek, budov či projektů, případně pro rozšíření grafického uživatelského rozhraní. Mezi tato data patří například *Shadery* a animace.

## Logika a pluginy

Jak jsme uvedli v části 1.1.5, jedním z hlavních cílů naší práce bylo umožnit využití jazyka C# pro skriptování logiky hry a umělé inteligence nepřátel. Díky využití herního enginu UrhoSharp, postaveného nad platformou .NET, máme zajištěno, že v naší implementaci budeme schopni použít systém *Reflection* pro načítání a používání pluginů.

I v případě, že by naše práce nevyužívala engine UrhoSharp, či jiný engine postavený na platformě .NET, bylo by možné použít jazyk C# pro skriptování. Taková implementace, využívající jiný jazyk pro tvorbu platformy, by vyla nucena využít C++/CLI pro vytvoření „mostu“ mezi kódem tvořícím platformu a managed kódem, umožňujícím nahrávání pluginů za běhu a jejich využití z kódu platformy. Tato alternativa je ovšem mnohem složitější pro implementaci, než je tomu při použití enginu založeného na platformě .NET. Při použití herního enginu připraveného pro .NET, jako je právě UrhoSharp, je možné tvůrcům pluginů navíc poskytnout plnou sílu tohoto enginu a umožnit jim využít všechny jeho schopnosti bez jakéhokoli zásahu či manuálního zprostředkovávání. V případě využití enginu v jiném jazyce by musel „most“ explicitně implementovat všechny schopnosti enginu, které by chtěl poskytnout tvůrcům pluginů a přeposílat každý požadavek vlastnímu hernímu enginu.

Obě předešlá řešení využívají systém „Reflection“ pro načítání tvůrcem dodaných assembly, nalezení tříd odpovídajících jednotlivým jednotkám, vytvoření in-

stancí těchto tříd a jejich následné použití. Pojmem „Reflection“ označujeme sadu tříd .NET frameworku, umožňující introspekcí .NET assembly, poskytující informace o třídách obsažených v těchto assemblies, jako například implementovaná rozhraní, poskytované metody či obsažené atributy. „Reflection“ dále umožňuje načítání existujících assembly za běhu programu, či dokonce generování nových assembly.

Assembly jsou identifikovány jménem, verzí, *culture* a veřejným klíčem. Při načítání je assembly nahrána do tzv. kontextu. Existují čtyři kontexty, do kterých jsou assembly nahrávány. Těmito kontexty jsou [32]:

- *Default Load Context*
- *Load-From Context*
- *Reflection-only Context*
- *No Context*

**Reflection-only** context slouží pro zkoumání metadat assembly pomocí reflection a znemožňuje vykonání kódu nahraného do tohoto kontextu, proto je pro nás nezajímavý a dále ho nebudeme rozebírat.

**Default Load Context** je kontext, ve kterém je nahrána assembly naší aplikace a všechny její závislosti. Do tohoto kontextu lze manuálně nahrávat další assembly, pokud se tyto assembly nachází v *Global assembly cache (GAC)*, v adresáři aplikace (*applicationBase*) nebo v aplikací specifikovaných podadresářích *applicationBase (PrivateBinPath)*. Pokud je identická assembly již nahrazena, nenahrává se znova ale vrátí se reference na již nahranou assembly. Závislosti nahrávaných assembly jsou automaticky vyhledávány na těchto třech místech.

**Load-From Context** je kontext, do kterého jsou nahrávány assembly pomocí metody `Assembly.LoadFrom`. Do tohoto kontextu lze nahrát assembly specifikovaným cestu spolu s výše zmíněnými vlastnostmi identifikujícími assembly, a tato cesta je přidána jako pátá identifikační vlastnost. Tímto způsobem lze nahrávat assembly ležící mimo *GAC*, *applicationBase* a *PrivateBinPath*. Závislosti jsou hledány nejdříve mezi již nahranými assembly v *Default Load Contextu*, následně v adresáři, ze kterého byla assembly nahrazena a nakonec na cestách pro nahrávání assembly do *Default Load Contextu*.

**No Context** je využíván při načítání assemblies vygenerovaných pomocí `Reflection.Emit` a `Assembly.LoadFile`. Navíc je toto jediný způsob, jak načíst dvě verze té samé assembly. Pod pokličkou je vytvořen každé nahrazené assembly zvláštní privátní kontext. Problémem tohoto kontextu je, že nejsou automaticky nahrávány závislosti. Tedy nezbývá nic jiného než závislosti nahrát manuálně, buď před načtením assembly nebo odchycením `AssemblyResolve` události.

V naší platformě používáme `Assembly.LoadFrom`. Tento způsob nám umožňuje nahrávat assembly z libovolných podadresářů uvnitř tvůrce tvořených balíčků, bez omezení na jejich adresářovou strukturu. Díky načítání závislostí ze zdrojového adresáře assembly mohou tvůrci her přibalit jimi používané knihovny do balíčku a ty budou následně při použití automaticky načteny.

Alternativou by bylo vynutit tvůrce balíčků specifikovat cesty, ve kterých se mohou vyskytovat assembly, a tyto cesty následně přidat do *PrivateBinPath*. Tento přístup by ovšem omezil místa, kde může naše platforma mít umístěné

balíčky, na podstom adresáře, ve kterém je umístěna naše platforma. Jak jsme psali v části 2.2.3 o rozdílech souborových systémů, některé ze systémů nepodporují změny v adresářovém podstomu aplikace a není tedy možné do tohoto podstomu umístit balíčky. I na systémech, na kterých je možné zapisovat do podstomu aplikace, není tento přístup k souborům považován za dobrý design, jak říká tento, i když poněkud zastaralý, návod pro umístování souborů na systému Windows [19]. Hlavním důvodem je omezení přístupových práv, které systém uvaluje na standardní cíl instalace aplikací.

Naše aktuální implementace nijak neřeší uvolňování assemblies ve chvíli, kdy hra končí a hráč načítá jiný balíček. Vzhledem k velikosti assembly v paměti by ale toto neměl být při běžných počtech používaných balíčků problém. Vzhledem k budoucí unifikaci .NET Frameworku, .NET Core a Mono pod implementací .NET 5 [15], který je následníkem .NET Core, není v této chvíli jednoduché zvolit způsob uvolňování nepoužívaných assembly. Aktuální způsob v námi používaném .NET Framework je použití `AppDomain`, pomocí které by bylo možné oddělit balíčky od sebe a umožnit jejich uvolňování. Toto řešení ovšem není podporované v .NET Core, který funkcionality uvolňování assembly ze systému `AppDomain`, který není v této verzi Frameworku podporován, přemisťuje do třídy `AssemblyLoadContext` [20]. Protože .NET 5 vychází z .NET Core, předpokládáme, že by v budoucnu bylo možné využít právě `AssemblyLoadContext` pro řešení tohoto problému.

### 2.3.4 Formát uložených úrovní

Ukládáním úrovně rozumíme serializaci aktuálního stavu hry a uložení takto serializovaných dat do souboru. Pro serializaci jsme měli několik požadavků:

- 1) otevřenosť schématu serializovaných dat,
- 2) minimalizace velikosti serializovaných dat,
- 3) rychlosť serializace a deserializace.

Účelem prvního požadavku je umožnit tvorbu nezávislých editorů úrovní, importujících a exportujících náš formát dat. Tento požadavek splňují serializace do formátu XML, definovaného pomocí *XSD* schématu, a binární serializace popsaná pomocí *interface description language (IDL)*. Příkladem takového binární serializace je formát Protocol buffers [13] od společnosti Google. Z binárních serializačních frameworků je pro jazyk C# nejlépe podporován právě formát Protocol buffers, ať už použit sám o sobě či za podpory knihovny protobuf-net, umožňující automatickou generaci IDL souborů z anotací ve zdrojovém kódu.

Druhý a třetí požadavek nás vedl k volbě binární serializace, která minimálnízuje velikost dat a má nejrychlejší zpracování. Tuto skutečnost sděluje jak dokumentace Protocol buffers [10], tak ji můžeme vidět experimentálně podloženou v benchmarku Maxima Novaka [22]. Tímto jsme vybrali formát Protocol buffers, který lze v jazyce C# použít buď manuální specifikací „*message*“ pomocí IDL, vygenerováním zdrojového kódu tříd v jazyce C# z této specifikace a následnou manuální serializací, nebo využitím knihovny Protobuf-net, která z anotací ve zdrojovém kódu generuje specifikaci „*message*“ a metody pro serializaci a deserializaci dat.

Naším cílem bylo využití knihovny Protobuf-net, bohužel tato knihovna vyžaduje v případě použití dědičnosti, aby *base* typy, tedy typy, od kterých se dědí, musí být označeny atributem `ProtoInclude` [14]. Vzhledem k tomu, že všechny naše třídy, reprezentující jednotky, budovy, projekty a hráče, dědí od UrhoSharp třídy `Component`, není možné tyto třídy serializovat pomocí Protobuf-net. Tento problém lze řešit dvěma způsoby:

- 1) Separovat reprezentaci entit do dvou tříd, první, která dědí od třídy `Component` a reprezentuje entitu v grafu herní scény, a druhou, která reprezentuje entitu z pohledu logiky, s kterou by pracovala naše platforma i implementace pluginů, a která by vlastnila a ovládala první třídu. V této reprezentaci by bylo možné využít Protobuf-net, znamenalo by to ovšem složitou delegaci volání mezi třídami a možné problémy při využití grafu scény pro nalezení entity.
- 2) Využít pro serializaci přímo technologii protocol buffers, která je na pozadí používána knihovnou Protobuf-net, a manuálně implementovat serializaci hry. Tato možnost se nám zdála jako lepší, protože nám také umožňovala manuálně specifikovat `.proto` soubory, z kterých protocol buffers generují třídy umožňující serializaci a deserializaci v různých jazycích. Navíc nám tento postup umožňuje specifikovat přesnou posloupnost akcí při deserializaci.

## 2.4 Poskytovaná funkcionality

V úvodní kapitole jsme vymezili druh her, které bude naše platforma podporovat, pomocí společných rysů a funkcionality. Cílem naší platformy je poskytovat implementaci této společné funkcionality a tím zjednodušit tvorbu her. Tato kapitola rozebera naši implementaci, její výhody a omezení, a popíše možnosti modifikace chování těchto implementací z uživatelského kódu.

### 2.4.1 Mapa

Jednou z hlavních funkcionalit poskytovaných naší platformou tvůrcům her je implementace herní mapy. Tato implementace poskytuje rozhraní pro dotazování na aktuální stav, manipulaci s mapou a grafickou reprezentaci zobrazenou hráči. Naše implementace odděluje logickou reprezentaci, která je poskytována zbylému kódu platformy a kódu uživatelských pluginů, a grafickou reprezentaci, která je vytvářena z logické reprezentace a zobrazována hráči. Toto rozdelení je určeno pro možné změny grafické reprezentace bez ovlivnění logické reprezentace a na ní závislého kódu.

#### Logické rozdělení

Mapa je ve hrách využívána pro zjišťování typu a výšky terénu či přítomnosti entit, jako například budov a jednotek, nejčastěji za účelem vyhledání cesty mezi dvěma pozicemi v herním světě. Dalším možným důvodem pro zjišťování mapou poskytovaných informací může být zjištění viditelnosti mezi dvěma entitami, možnosti střelby mezi dvěma entitami či interakce hráče se skupinami entit. Existuje několik způsobů, jak implementovat tyto funkce herní mapy.

V úvodní kapitole jsme specifikovaly, jaké hry bude naše platforma podporovat. Z vlastností těchto her nám vyplynula obdélníková mapa, rozdělená na čtvercové dlaždice. Toto ovšem není jediná možná implementace. V této části popíšeme další možné typy implementací map, jejich výhody, nevýhody a použití.

**Mapa bez rozdělení** vede při každém dotazu na přítomnost budov či jednotek v dané oblasti mapy k výpočtu průsečíků ploch zabíraných všemi jednotkami a budovami s danou oblastí mapy. Asymptotická složitost této operace je tedy  $O(N)$ , kde  $N$  je počet budov či jednotek v naší hře.

**Mapa s rozdělením** na části umožňuje při dotazu na přítomnost jednotek či budov v oblasti mapy prozkoumat pouze části tvořící tuto oblast mapy. Části, označovány jako „*Tiles*“ , v překladu dlaždice, mohou být různých tvarů. Nejčastějšími tvary jsou čtverec či šestiúhelník.

Při rozdělení na čtverce lze mapu jednoduše uložit do dvourozměrného pole, což dále usnadňuje dotazy na prohledání oblasti mapy. Nevýhodou čtvercového tvaru dlaždic je různá vzdálenost sousedních dlaždic, pokud umožníme pohyb mezi všemi osmi sousedními dlaždicemi. Kde vzdálenost středů dlaždic sousedících celou hranou je rovna délce hrany, vzdálenost středů dlaždic sousedících pouze rohem je rovna  $\sqrt{2a^2}$ , kde  $a$  je délka hrany čtverce. Oproti tomu při použití šestiúhelníkových dlaždic je vzdálenost mezi všemi sousedy rovna.

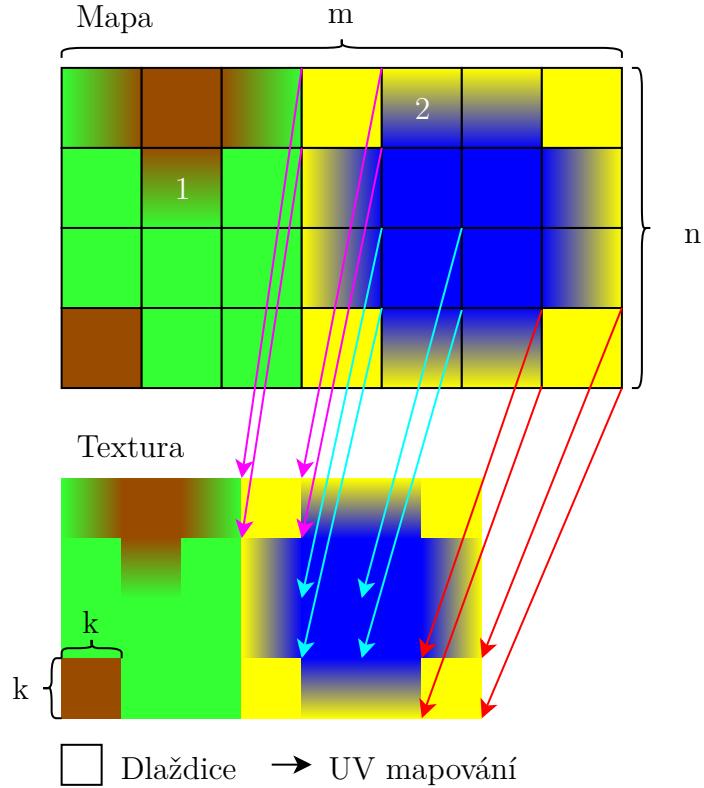
## Grafická reprezentace

Jak jsme popsali v předešlé části, mapa je obdélníkového tvaru, rozdělena na čtvercové dlaždice o velikosti 1x1. Existují dva základní druhy možných implementací grafického zobrazení takového mapy.

První možností, ilustrovanou obrázkem 2.3, je vytvoření textury, ve které bude každé dlaždici odpovídat separátní část textury. Tuto implementaci nazýváme „Textura dlaždic“. Velikost textury je závislá jak na velikosti mapy, v obrázku značené  $m, n$  a udávané v počtu dlaždic v daném rozměru, tak na velikosti textury jednotlivých typů dlaždic, zde značené  $k$ . S ohledem na čtvercový tvar dlaždic požadujeme i po texturách jednotlivých typů čtvercový tvar. Velikost výsledné textury bude rovna  $k^2 * m * n$ . Tuto texturu můžeme vidět v dolní části obrázku 2.3. Na objekt reprezentující terén je následně tato textura promítnuta pomocí standardního *UV* mapování, což je na ukázce reprezentováno šípkami mapujícími vrcholy geometrie na texturu. V takovéto textuře má každá dlaždice přiřazenu disjunktní část textury, lze tedy vzhled jednotlivých dlaždic upravovat nezávisle na vzhledu ostatních dlaždic. Tímto způsobem lze vytvořit různé přechody mezi typy dlaždic a provést další úpravy na celkovém vzhledu herní mapy. Ukázku takovýchto přechodů můžeme vidět na dlaždicích 1 a 2.

Nevýhodou této implementace je především velikost textury, která v závislosti na velikostech textur jednotlivých typů dlaždic může dosáhnout stovek MiB až jednotek GiB. Takováto velikost textury by byla neúnosná i pro slabší počítače, nemluvě o mobilních zařízeních. Další nevýhodou této implementace je složitost změny typu dlaždice. Při změně typu je v této reprezentaci nutné najít část textury odpovídající měněné dlaždici a do této části nakopírovat texturu nového typu. Vzhledem k velikosti textur dlaždic by tato operace mohla být časově velice náročná, především při změně typu velkého počtu dlaždic. Možnou výhodou této implementace je schopnost reprezentovat každý z vrcholů, ve kterém se vyskytují rohy sousedních dlaždic, pomocí jednoho „vertexu“, tedy grafického vrcholu. Díky

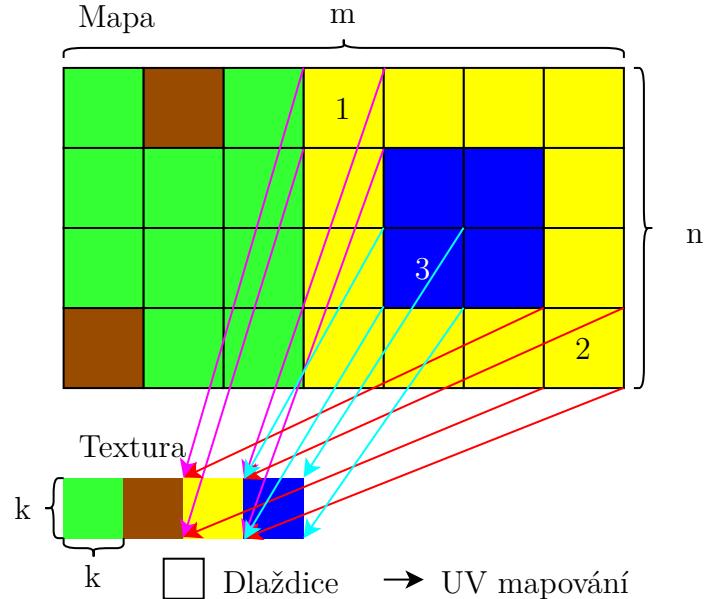
tomu je omezena velikost *vertex bufferů* a zaručena spojitost mapy. Vzhledem k velikosti textury je ovšem úspora paměti nedostatečná.



Obrázek 2.3: Implementace pomocí textury dlaždic.

Druhou možností, ilustrovanou obrázkem 2.4, je vytvořit texturu, ve které bude textura každého typu dlaždic právě jednou. Tuto texturu, obsahující vzhled typů dlaždic, můžeme vidět v levé dolní části obrázku. Velikost takového textury závisí pouze na počtu typů dlaždic, oproti předchozí implementaci, ve které závisela velikost textury na velikosti mapy. I když typů dlaždic může být neomezeně, nepředpokládáme, že by tento počet přesáhl malé stovky. Díky tomu je tato implementace paměťově mnohem úspornější. Při takovéto implementaci je poté každá dlaždice určitého typu namapována na stejnou část textury, což je na obrázku ukázáno na příkladu dlaždic 1 a 2. Vzhledem k tomuto mapování všech dlaždic stejného typu na stejnou část textury nelze jednoduše vytvořit přechody mezi typy dlaždic, jak tomu bylo v první implementaci. Jedinou možností je vytvořit separátní typ, který bude reprezentovat přechod mezi dvěma typy dlaždic. Další možnou nevýhodou této implementace je nutnost reprezentovat každou dlaždici separátní čtvericí tzv. „vertexů“, tedy grafických vrcholů. Oproti první implementaci tedy bude pro reprezentaci mapy stejné velikosti potřeba čtyřnásobný počet vrcholů. Tato možná nevýhoda je ale vyvážena úsporou paměti v rámci textury typů, dále díky jednoduší implementaci změny typů dlaždic, a v neposlední řadě jednodušším rozdělením mapy na části, popsané dále. Změna typu dlaždice v této reprezentaci spočívá ve změně UV souřadnic čtyř vrcholů, odpovídajících měněné dlaždici. Jedná se tedy o zapsání osmi čísel, které oproti kopírování částí textury v první implementaci mění mnohem méně dat. V obrázku si můžeme toto přemapování představit jako přesunutí všech čtyř šipek dané dlaždice na jinou část

textury. Vzhledem k těmto výrazným úsporám času a paměti jsme zvolili právě tuto možnost pro naši implementaci.



Obrázek 2.4: Implementace pomocí textury typů.

Další součástí naší implementace je rozdelení mapy na části, kterým my, podle hry Minecraft, která nás k tomuto rozdelení inspirovala, říkáme „Chunk“ . Hlavním důvodem pro toto rozdelení je velikost vertex bufferů, která je v enginu Urho3D, a tedy i enginu UrhoSharp, limitovaná na 64 tisíc vertexů. To při naší implementaci odpovídá mapě o velikost 127 krát 127 dlaždic, což je z našeho pohledu příliš malá velikost pro typ her, který chceme podporovat. Zřejmou výhodou tohoto rozdelení je rozšíření množiny možných velikostí map, která je při této implementaci z vrchu omezena pouze výkoností hardwaru a velikostí paměti. Na druhou stranu toto rozdelení klade na možné velikosti map jiné omezení, a to nutnost, aby každá velikost map byla celočíselným násobkem velikosti chunků v obou rozměrech. V aktuální verzi platformy je velikost chunku nastavena na 50 krát 50 dlaždic, což určuje krok mezi velikostmi map. Každá mapa tedy musí být velikosti  $50^x \times 50^y$ . Tuto velikost jsme zvolili podle velikostí mapy ve hře Stronghold Crusader, kterou jsme v úvodní kapitole označili jako zástupce typu her, které chceme v naší platformě podporovat.

Další výhodou tohoto rozdelení je možnost tzv. „cullingu“ , kdy engine nevykresluje modely vzdálené od kamery více než nastavitelný limit. Účelem této vlastnosti enginu je zmenšení hardwarových nároku, který plní i v tomto případě. Dále můžeme při úpravě terénu využít tohoto rozdelení pro zamknutí pouze upravovaných chunků, čímž zmenšujeme velikost dat, které je potřeba přenést do paměti grafické karty, a dále zmenšujeme výpočetní náročnost reprezentace mapy.

## Deformace terénu

Jak jsme v předchozích částech popsali, je naše mapa logicky rozdělena na čtvercové dlaždice, kde každá z dlaždic je graficky reprezentována čtyřmi vrcholy v modelu mapy. Tyto dvě reprezentace, logickou a grafickou, musí naše imple-

mentace držet synchronizované, aby hráči byl zobrazován skutečný stav hry, jak ho vidí jednotky a nepřátelé.

Logickou myšlenkou je zvolit si jednu z reprezentací, kterou budeme upravovat manuálně, a následně druhou reprezentaci generovat z reprezentace první. Pro manuální upravování jsme určili logickou reprezentaci, která je tvořena C# objekty a lze ji tedy jednoduše upravit bez použití složitějších nástrojů jazyka C#. Tímto máme zajištěno, že logiky hráčů, jednotek a budov budou pracovat s aktuálním stav mapy. Nyní musíme zajistit, aby i hráči byl zobrazen tento stav.

Oproti úpravě logické reprezentace je úprava grafické reprezentace složitější. Z předchozí části víme, že chceme využít manuální práce s vertexy, především pro nízkou výpočetní složitost takovýchto úprav. Pro práci s vertexy poskytuje engine UrhoSharp dva nástroje.

To do  
(7)

Prvním nástrojem je třída `CustomGeometry`, která překrývá `VertexBuffer` a `IndexBuffer` a poskytuje API pro zjednodušení práce s těmito strukturami. Bohužel zde narázíme na aktuální implementaci UrhoSharp, která neumožňuje přístup k metodám, která v Urho3D pracují s třídou `PODVector`. Jak říká dokumentace [29], tato třída slouží jako vektor (ve smyslu jazyka C++) pro uložení tzv. *POD (plain old data)* typů, což jsou typy „*nevolají konstruktor nebo destruktor a používají block move*“. Bohužel tuto třídu nebyli zatím autoři UrhoSharp schopni poskytnout v jazyce C#, a tím pádem ani jakoukoli metodu, která v enginu Urho3D tuto třídu používá. V případě `CustomGeometry` je to metoda `GetVertices()`, která umožňuje přístup ke všem vertexům `CustomGeometry` najednou, bez režie metod poskytujících přístup k vertexům jednotlivě, jako například `GetVertex`. Tato skutečnost nám znemožňuje použití `CustomGeometry` s dostatečnou výkoností, a proto jsme byli nuteni použít druhý nástroj.

Druhým nástrojem jsou třídy `VertexBuffer` a `IndexBuffer`. Tyto třídy umožňují přímý přístup k datům geometrií, umožňující měnit jednotlivé vrcholy a indexy. Za použití těchto tříd dokážeme generovat geometrii či ji upravovat přímo za běhu, a to s minimální možnou režií. Tato minimální režie je umožněna poskytnutím přímého přístupu k datům v paměti a jejich následnou manipulací pomocí `unsafe` C# kódu. K této minimální režii nám oproti použití `CustomGeometry` také tento způsob umožňuje do paměti zamýkat pouze části geometrií, kterých se změny týkají, oproti `CustomGeometry`, která, pokud by UrhoSharp tuto funkcionality implementoval, by nás nutila zamknout do paměti veškerá data geometrie.

## 2.4.2 Hledání cesty

Hledání cesty, neboli *Pathfinding*, je jednou z nejuniversálněji používaných funkcionalit ve všech typech her, nejen RTS. Každá hra obsahující jednotky pohybující se po mapě používá nějakou formu pathfindingu. Nejčastěji ve hrách hledáme nejkratší cestu mezi dvěma body, odpovídajícími aktuální pozici jednotky a cílové pozici jednotky. Standardní algoritmy používané pro hledání cesty pracují nad váženým grafem. Tato reprezentace zmenšuje prohledávaný prostor a umožňuje rychlejší nalezení cesty. Graf je složen z vrcholů, reprezentujících pozice v herním světě, a hran, reprezentujících možnost průchodu mezi spojenými vrcholy. Tyto hrany mohou být orientované, umožňující průchod pouze jedním směrem.

V této části popíšeme různé možnosti převodu logické reprezentace mapy na

graf pro vyhledávání cesty, implementaci těchto převodů v algoritmu pro hledání cesty poskytovaném naší platformou a možnost použití tvůrcem implementovaného algoritmu pro hledání cesty naší platformou.

## Převod mapy

Převod mapy na graf lze dělat dvěma způsoby, dynamicky, kdy je graf generován při výpočtu cesty, nebo staticky, kdy je graf generován na počátku hry, případně je generován znovu při změně mapy.

Dynamické generování grafu umožňuje reprezentovat aktuální stav mapy, který se může v průběhu času měnit. Tímto způsobem lze do hledání cesty zahrnout aktuální pozice jednotek, postavené budovy či změny terénu. Problémem tohoto typu generování grafu je výpočetní náročnost.

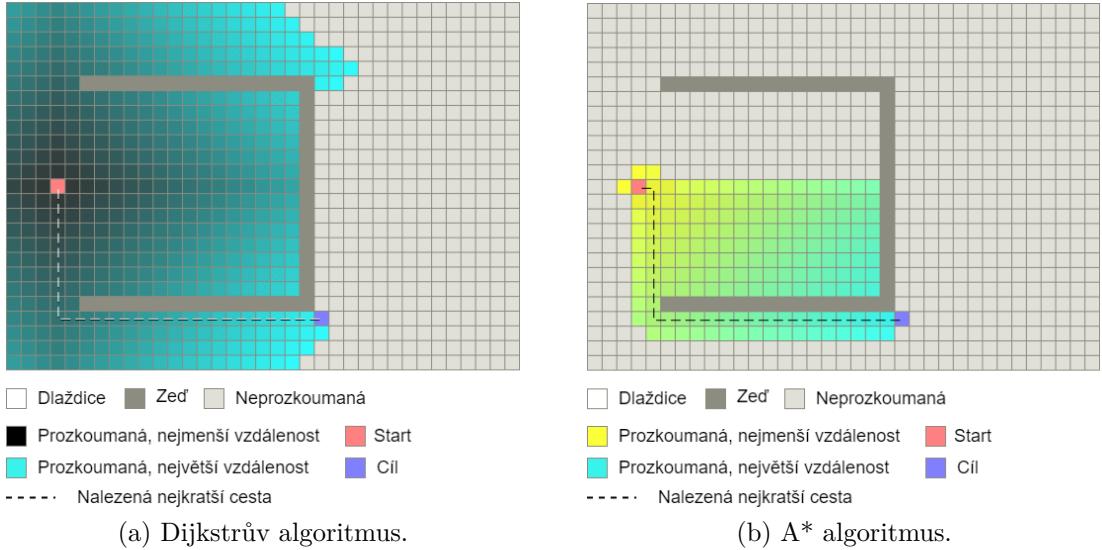
Statické generování grafu umožňuje jedno vygenerování grafu, které předzpracuje logickou reprezentaci mapy a maximálně optimalizuje graf pro zmenšení náročnosti hledání cesty. Nevýhodou je neschopnost odrážet změny v terénu, pozice entit či stavbu budov. Dalším problémem je rozdílná průchodnost částí mapy pro různé typy jednotek. Pro řešení tohoto problému pouze statickým generováním musí být pro každý typ jednotky vygenerován zvláštní graf, což se negativně projeví na paměťové složitosti programu.

## V naší platformě

Naše platforma definuje rozhraní `IPathfindAlg`, které reprezentuje algoritmus pro hledání cesty používaný součástmi platformy a poskytovaný pro použití tvůrci hry. Zároveň platforma poskytuje tvůrcům hry jednu možnou implementaci tohoto rozhraní, a to pomocí algoritmu A\*.

Algoritmus A\* je jedním z nejpoužívanějších algoritmů pro hledání cest ve hrách, jak říká Amit Patel ve svém článku [25]. Z tohoto důvodu jsme se tento algoritmus rozhodli použít jako základní implementaci hledání cesty v naší platformě. A\* je grafový algoritmus pracující nad váženým orientovaným grafem. Algoritmus dostává jako zadání graf, počáteční vrchol a koncový vrchol. Oproti Dijkstrou algoritmu, který prochází vrcholy od startovního vrcholu v pořadí vzrůstající ceny cesty do vrcholu, modifikuje A\* toto usporádání vrcholů pomocí heuristiky, kterou přičítá k ceně cesty do vrcholu. Toto porovnání můžeme vidět na obrázku 2.5, které ukazují pořadí prohledaných vrcholů v grafu. Můžeme vidět, že Dijkstrův algoritmus vytváří okolo startovního vrcholu rovnoramenný kruh prohledaných vrcholů (pokud vzdálenost vrcholu bereme jako cenu cesty ze startovního vrcholu). Oproti tomu algoritmus A\* prohledává vrcholy nejen podle jejich vzdálenosti od počátečního vrcholu, ale za použití heuristiky zohledňuje i jejich vzdálenost od cílového vrcholu. Díky tomu algoritmus A\* ve většině případů zmenšuje počet prohledaných vrcholů.

Problémem algoritmu A\* může být vlastní heuristika. Algoritmus A\* požaduje, aby heuristika splňovala dvě vlastnosti, a to přípustnost a monotonost. Přípustnost označuje, že hodnota heuristiky pro každý vrchol je menší než minimální délka cesty do tohoto vrcholu. Monotonost poté označuje tento vztah  $h(x) \leq d(x,y) + h(y)$ , kde  $h(x)$  je hodnota heuristiky vrcholu x a  $d(x,y)$  délka hrany z vrcholu x do vrcholu y. Splnění těchto vlastností zajišťuje, že nalezená



Obrázek 2.5: Porovnání Dijkstrova algoritmu s algoritmem A\*. Ilustrace převzaty z článku Amita Patela [25].

cesta bude optimální (přípustnost) a každý vrchol prohledáme nejvýše jednou (monotonnost).

Naše implementace algoritmu A\* používá spojení statického a dynamického generování grafu, spolu s tvůrcem dodanou heuristikou. Graf, reprezentující terén úrovně, je vygenerován staticky. Tedy pro každou dlaždiči je vytvořen vrchol a každý vrchol je spojen se všemi vrcholy reprezentujícími sousední dlaždice. Dále je umožněno do grafu za běhu přidávat vrcholy a spojovat je s již existujícími. To je použito pro umožnění pohybu po budovách, u kterých jsou při jejich stavbě do grafu dodány vrcholy reprezentující schůdné plochy budovy a tyto vrcholy jsou spojeny jak mezi sebou, tak s existujícími vrcholy grafu.

Dynamická část generování grafu se projevuje při výpočtu vah hran. Hrany nemají pevně stanovenou váhu, místo toho je pro výpočet cesty požadováno po tvůrci hry, aby každá jednotka hledající cestu poskytla tzv. kalkulátor, který dynamicky ohodnocuje hrany a definuje jejich průchodnost. Tento přístup nám umožňuje zachovávat pouze jeden graf, který se díky využití kalkulátorů chová jako rozdílný graf pro každý kalkulátor. Vzhledem k požadavkům algoritmu A\* na vztah heuristiky a ceny hran, popsaných výše, požadujeme po tvůrci hran poskytnutí heuristiky v rámci kalkulátoru. Zároveň tento přístup umožňuje tvůrci místo vlastní implementace algoritmu pro hledání cesty pouze změnit poskytované hodnoty hran a/nebo heuristiky, čímž může tvůrce změnit chování naší implementace algoritmu A\* pro svoji potřebu.

Naše implementace A\* ovšem není jediným použitelným algoritmem pro hledání cesty v naší platformě. Pokud tvůrci nestačí úprava váhy hran a heuristiky, umožňuje platforma implementaci vlastního algoritmu splňujícího rozhraní `IPathfindAlg`, který bude následně používán všemi součástmi platformy. Používaný algoritmus je specifikován při startu úrovně, je tedy možné, aby i v jednom balíčku různě úrovně používaly různé algoritmy pro hledání cesty.

### 2.4.3 Projektily

Simulace projektilů je častým problémem v RTS hrách, tedy v typu her, který naše platforma chce podporovat. Rozhodli jsme se proto implementovat tuto funkcionality, umožňující výpočet počátečního směru projektilu při střelbě na pohyblivý cíl, simulaci letu se stálou gravitací a detekci zásahů.

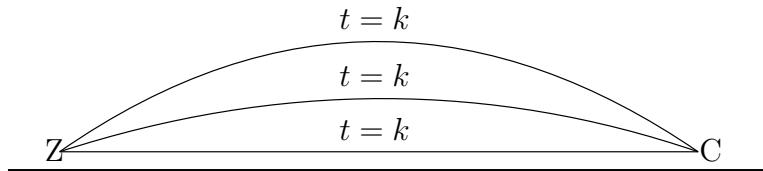
#### Typy projektilů

Existuje několik typů simulací projektilů, rozdelených podle stupně přesnosti simulace reálného světa. Těmito typy, seřazenými podle vzrůstající realističnosti simulace, jsou:

- *Hitscan* (bez simulace letu),
- s konstantní horizontální rychlostí (proměnlivá gravitace, žádný odpor vzduchu),
- s konstantní dopřednou rychlostí (konstantní gravitace, žádný odpor vzduchu),
- s plně realistickým chováním (konstantní gravitace, simulace odporu vzduchu).

**Hitscan projektily** jsou zvláštním typem projektilů, který má nulovou dobu letu. V okamžiku výstřelu je proveden výpočet, zda se ve směru, kterým byl výstrel mířen, vyskytuje cíl. Pokud ano, je tomuto cíli okamžitě uděleno poškození. Hitscan projektily lze dále rozdělit podle chování po zasáhnutí prvního cíle. Projektil může být zastaven prvním cílem, nebo pokračovat dále skrz první cíl a případně zasáhnout další cíle.

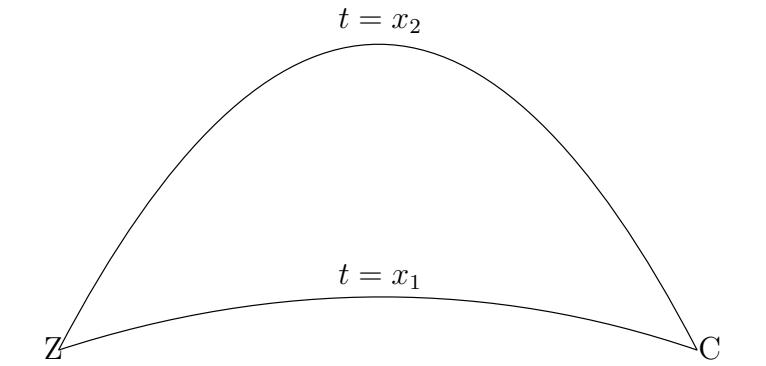
**Projektily s konstantní horizontální rychlostí** umožňují tvůrce hry z pohledu logiky uvažovat pouze pohyb v rovině terénu. Jak můžeme vidět na obrázku 2.6, výška oblouku nemá u tohoto druhu projektilu vliv na dobu letu a je závislá pouze na vzdálenosti zdroje a cíle v rovině. Tato vlastnost umožňuje zvolit výšku oblouku tak, aby graficky vypadala dobře. Z pohledu vyvážení síly jednotek a budov umožňuje tento typ projektilů jednoduše spočítat dobu letu jako  $t = \text{vzdálenost(střelec, cíl)} / v$ , kde  $v$  značí určenou rychlosť projektilu. Tento typ projektilů je především vhodný pro hry odehrávají se v jedné rovině, jako například Warcraft nebo Starcraft.



Obrázek 2.6: Trajektorie a čas letu projektilu s konstantní horizontální rychlostí.

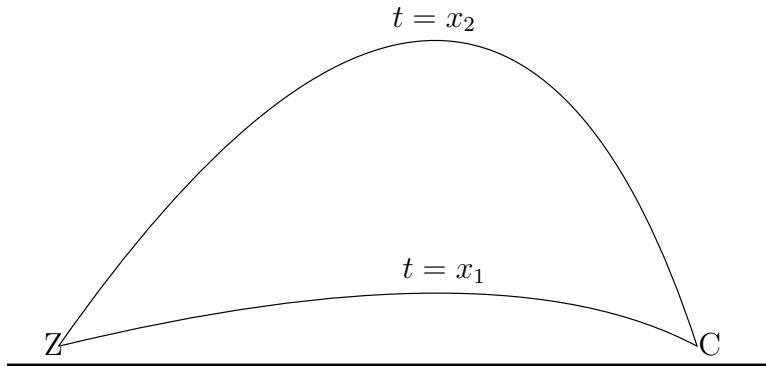
**Projektily s konstantní dopřednou rychlostí** se pohybují po balistické křivce, určené počáteční rychlostí, směrem výstřelu a sílou gravitace. Jak můžeme vidět na obrázku 2.7, existují vždy nejvýše dvě trajektorie, kterými lze zasáhnout při daných hodnotách výše vyjmenovaných vlastností. Díky konstantní dopředné

rychlosti a různým délkám trajektorií vždy platí vztah  $x_1 \leq x_2$ . Oproti předešlým typům projektilů vyžaduje tento typ složitější výpočty pro zasažení cíle, především pohybujícího se cíle. Tyto výpočty byly v naší platformě převzaty z článku Forresta Smitha [27]. Tento článek poskytuje vzorec, podle kterého lze vypočítat směr střelby projektilu, pokud známe pozice strelce, cíle, gravitaci a počáteční rychlosť projektilu. Pohyb cíle musí být konstantní rychlosť konstantním směrem. Vzhledem k implementaci pohybu v naší platformě, který se odehrává po spojnicích středů dlaždic, je potřeba najít spojnicu, na které se bude cíl právě vyskytovat. Tento postup je iterativní, kdy procházíme cestu cíle a v každém bodě změny směru počítáme dobu letu projektilu. Ve chvíli, kdy najdeme dva body, kde v prvním bude jednotka dříve než projektil, a v druhém později než projektil, našli jsme požadovanou spojnicu. Následně využijeme převzatý vzoreček pro spočtení směru výstřelu. Tento postup by bylo možné celý provést iterativně, kde místo použití vzorečku by jsme znova prohledávali spojnice dvou bodů a přibližovali se k cíli.



Obrázek 2.7: Trajektorie a čas letu projektilu s konstantní dopřednou rychlosťí.

**Projektily s realistickým chováním** přidávají oproti předešlému typu ještě jednu vlastnost, a to odpor vzduchu. Jak můžeme při porovnání trajektorií na obrázcích 2.7 a 2.8, má trajektorie tohoto typu projektilu složitější tvar. Výpočet tohoto typu projektilů je nejčastěji prováděn iterativně, kdy je odhadnuta doba letu, podle této doby letu je predikován pohyb cíle, a následně spočítána opravdová délka letu. Tento postup je opakován do dosažení potřebné přesnosti.



Obrázek 2.8: Trajektorie a čas letu projektilu s plně realistickým chováním.

Podle priorit a zasazení hry jsou často voleny méně realistické modely, které z pohledu hry mohou mít výhody oproti plně realistickému chování. Těmito vý-

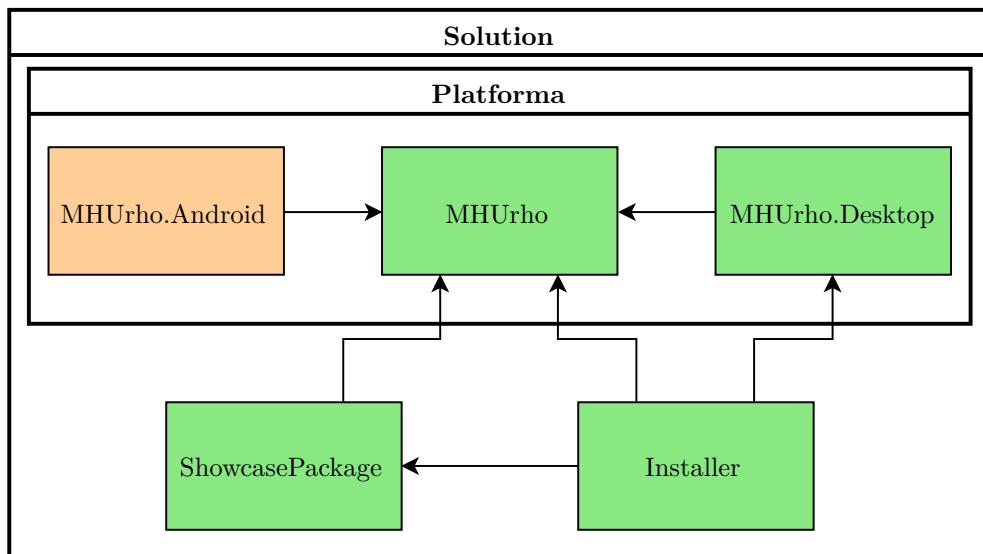
hodami může být lepší zasazení do tématu hry, což můžeme vidět u laserových zbraní v sci-fi hrách jako Starcraft 2 [6], které jsou typu *Hitscan*. I když by se zdálo, že je vždy nejlepší co nejvíce se blížit realitě a tedy vybrat si plně realistické chování, v některých hrách je důležitější možnost vyvážení síly jednotek, předvídatelnosti chování a/nebo vzhledu.

Naše platforma poskytuje implementaci výpočtu směru střelby pro projektily s konstantní dopřednou rychlostí a simulaci letu tohoto typu projektilů. Naším cílem ovšem bylo nijak neomezovat tvůrce her, kvůli čemuž jsou tyto prvky poskytovány jako komponenty, které lze připojit k projektilu a které následně řídí jeho chování. Nijak ale nejsou provázány s vlastní třídou projektilu, čímž je tvůrcům hry umožněno implementovat kterýkoli z typů projektilů.

# 3. Programátorská dokumentace

V této kapitole popíšeme naší implementaci platformy MHUrho a ukázkové hry. K implementaci platformy bylo použito Visual Studio 2017 Education a .NET Framework 4.7.2, který je zároveň cílovým frameworkem naší platformy. Celá implementace je obsažena v jediném *solution*, MHUrho.

## 3.1 Struktura solution



Obrázek 3.1: Struktura solution. Zelené značí součásti naší práce, oranžové místo budoucí rozšiřitelnosti.

Hlavním cílem naší práce byla tvorba platformy pro tvorbu RTS her. Jak můžeme vidět v diagramu 3.1, vlastní implementace platformy je realizována několika projekty. Účelem tohoto rozdělení je separace přenositelně implementovatelných částí platformy, které díky využití enginu UrhoSharp a celkovému návrhu práce tvoří velkou většinu funkcionality, od částí závislých na cílovém systému.

Přenositelné části jsou obsaženy v projektu **MHUrho**. Výstupem tohoto projektu je knihovna využívaná jak implementacemi pro specifické systémy, tak balíčky tvořícími jednotlivé hry.

Implementace pro specifické systémy obsahují především inicializační kód, který dostává spuštěný program do konzistentního stavu bez ohledu na platformu, a dále implementace rozhraní specifikovaných v přenositelné části, které umožňují této části pracovat s oblastmi, pro které ani platforma .NET ani používaný herní engine neposkytuje přenositelnou implementaci. Tyto oblasti byly popsány v části 2.2.

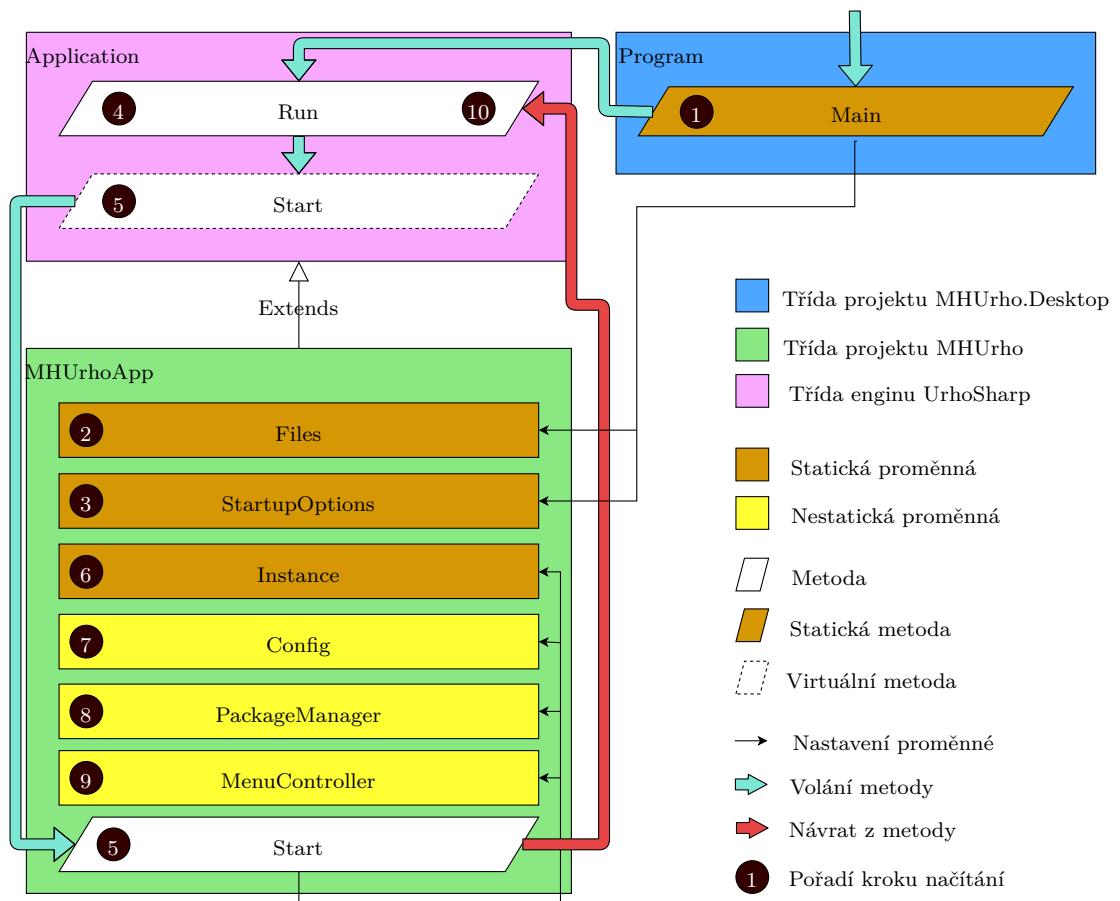
Dále diagram ukazuje dvě součásti solution, které nejsou přímou součástí funkcionality platformy. První z těchto součástí je projekt **ShowcasePackage**, který obsahuje implementaci ukázkové hry sloužící pro demonstraci schopností platformy a jako referenční implementace pro tvůrce budoucích balíčku. Druhou součástí je **Installer**, jehož výstupem je **.msi** instalátor umožňující jednoduchou

instalaci platformy na systém Windows spolu s instalací všech závislostí a vložení *Ukázkového balíčku* do nainstalované instance platformy.

## 3.2 Spuštění aplikace

Spuštění aplikace je specifické pro každý ze systémů. Vzhledem k tomu, že cílem naší práce byla implementace platformy pro systém Windows, popíšeme v této části spuštění aplikace v rámci tohoto systému.

### 3.2.1 Systémová část



Obrázek 3.2: Spuštění aplikace.

Spuštění začíná v implementaci specifické pro daný systém. Hlavní třídy účastnící se spuštění aplikace a posloupnost volání metod můžeme vidět na obrázku 3.2. U každé ze tříd jsou ilustrovány pouze prvky účastnící se inicializace.

Pro systém Windows začíná celý proces zavolením metody `Program.Main`. Tato metoda jako první inicializuje `FileManager`, implementující práci se soubory na systému Windows, a uloží ho do statické položky třídy `MHUrhoApp`. Tento systém je nezbytné inicializovat před spuštěním herního enginu, protože je používán při inicializaci třídy `MHUrhoApp`.

Jak ukazuje diagram, třída `MHUrhoApp` je potomkem třídy `Application` z enginu UrhoSharp. Tato třída, již podle názvu, reprezentuje celou aplikaci v enginu

UrhoSharp a poskytuje přístup ke všem součástem enginu.

Dalším krokem v metodě `Main` je zpracování parametrů aplikace z příkazového řádku do platformou definované systémově nezávislé reprezentace. Instance třídy `StartupOptions` obsahující tuto reprezentaci je následně také uložena do statické položky třídy `MHUrhoApp`, protože je stejně jako `FileManager` používána při inicializaci `MHUrhoApp`.

Položky `FileManager` a `StartupArgs` jsou statické pro budoucí rozšířitelnost na platformu Android. Důvodem je nutnost spuštění enginu pomocí následujícího kódu:

```
await surface.Show<MHUrhoApp>(new ApplicationOptions("Data"));
```

Metoda `Show` bohužel nedovoluje přidat konstruktoru `MHUrhoApp` další parametry, kterými by bylo možné předat `FileManager` a `StartupOptions`.

Na systému Windows následuje vytvoření instance `MHUrhoApp` a zavolání metody `Run`. Tímto voláním je předána kontrola nad procesem enginu UrhoSharp, který je tímto inicializován a je v něm spuštěna herní smyčka.

V rámci své inicializace zavolá třída `Application`, reprezentující herní engine, virtuální metodu `Start`, jejíž přetížením nám umožňuje inicializovat naši část aplikace. Tímto se dostáváme do přenositelné části inicializace.

### 3.2.2 Přenositelná část

Jak je řečeno v předchozí části, přenositelná část inicializace začíná v okamžiku volání metody `Start` herním enginem.

V rámci této metody provede naše platforma inicializaci všech svých součástí a přejde do režimu reagování na interakce uživatele s grafickým rozhraním.

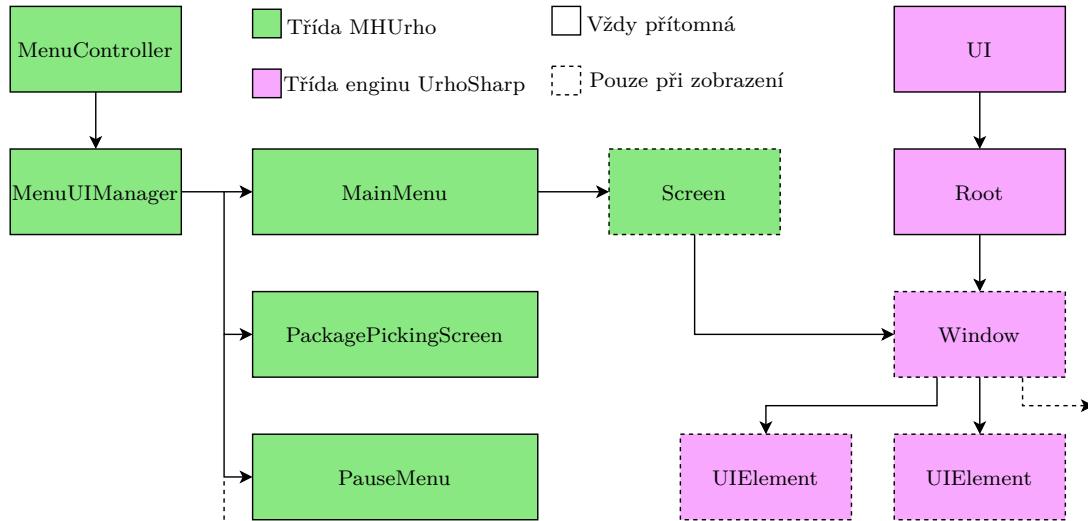
Důležitým předpokladem naší platformy i enginu UrhoSharp je přítomnost pouze jedné instance třídy `MHUrhoApp`, respektive jejího předka `Application` v dané `AppDomain`. Vzhledem k tomuto předpokladu je tato jediná instance třídy zpřístupněna pomocí statické proměnné `Instance`.

Dalšími kroky inicializace jsou:

- 1) Vytvoření `SynchronizationContext`, který umožňuje implementaci načítání popsaného v částech 3.4.5 a 3.12.
- 2) Načtení a nastavení konfigurace aplikace za využití třídy `FileManager` inicializované v systémové části. Toto nastavení ovlivňuje například vzhled okna aplikace či vykreslovanou vzdálenost ve hře.
- 3) Načtení seznamu přítomných balíčků, popsané blíže v části 3.7. Při chybě v načítání kteréhokoli z balíčků je po spuštění uživatelského rozhraní uživateli zobrazeno hlášení o této chybě.
- 4) Inicializace vstupu a výstupu. Tato inicializace je implementována podle návrhového vzoru *Factory* pro zjednodušení přepínání mezi různými schématy vstupu a výstupu. V rámci naší práce je implementováno pouze schéma myši a klávesnice, ale v implementaci platformy je připravena základní struktura pro implementaci schématu pro dotykové obrazovky.

Tímto je inicializace kompletní. Volání metody `Start` se vrací do metody `Run`, ve které engine předchází do smyčky obsluhující události uživatelského vstupu.

### 3.3 Uživatelské rozhraní



Obrázek 3.3: Datová struktura uživatelského rozhraní.

Jak můžeme vidět na diagramu 3.3, centrálními třídami uživatelského rozhraní v rámci menu jsou třídy `MenuController` a `MenuUIManager`. Dále můžeme vidět, že z pohledu herního enginu je grafické rozhraní reprezentováno třídou `UI` s kořenovým prvkem `Root`.

Účelem třídy `MenuController` je odstínění implementace uživatelského rozhraní od detailů implementace zbytku platformy. Pokud je tedy výsledkem uživateli akce změna stavu jiné části platformy než UI, je tato akce delegována právě třídě `MenuController`. Zároveň se také tato třída snaží odstrňovat zbytek platformy od implementace uživatelského rozhraní, tedy pokud část hry požaduje změnu uživatelského rozhraní, měla by tuto akci delegovat třídě `MenuController`.

Třída `MenuUIManager` je, jak můžeme vidět, centrálním bodem implementace uživatelského rozhraní menu. Její hlavní funkcí je management obrazovek menu, implementace přepínání mezi nimi a poskytnutí přístupu k třídám reprezentujícím tyto obrazovky. V rámci přepínání obrazovek umožňuje tato třída jednoduše přepínat zpět na předchozí obrazovky, protože při každém přepnutí na novou obrazovku je stará obrazovka přidána do zásobníku předchozích obrazovek.

Každá z obrazovek menu je reprezentována samostatnou třídou implementující chování obrazovky a zajišťující správu prostředků herního enginu příslušících dané obrazovce. Jak můžeme vidět na diagramu 3.3, slouží každá ze tříd reprezentujících obrazovky menu jako *proxy* pro třídu, která implementuje chování obrazovky a je vlastníkem prostředků enginu pro její zobrazení uživateli.

Z pohledu enginu tvoří uživatelské rozhraní strom s kořenem `Root`. Každá obrazovka je tedy dítětem tohoto kořenu, s dalšími prvky jako tlačítka a popisky jako svými dětmi. Tyto prostředky, reprezentující danou obrazovku, jsou spravovány právě implementační třídou a mají stejnou životnost jako tato třída.

Zobrazení obrazovky je uskutečněno v reakci na zavolání metody `Show` na *proxy* třídě obrazovky. Metoda `Show` vytvoří novou instanci implementační třídy, která v rámci konstruktoru použije engine a `UI.Root` pro nahrání obrazovky. Při následném skrytí obrazovky v reakci na volání metody `Hide` na proxy třídě je implementační třída zahozena pomocí volání `Dispose` a nastavení reference na null. V rámci volání `Dispose` jsou z reprezentace uživatelského rozhraní uvnitř enginu odstraněny všechny prvky příslušící této obrazovce. Díky této implementaci nedochází k hromadění naalokovaných obrazovek i při cyklickém přepínání mezi několika obrazovkami.

### 3.3.1 Definice vzhledu

Engine UrhoSharp umožňuje definici rozložení a vzhledu uživatelského rozhraní jak v kódu, tak pomocí XML souborů. Podle složitosti jednotlivých obrazovek používáme kombinaci obou těchto možností. Některé jednodušší obrazovky, jako například hlavní menu, mají celý svůj vzhled definován pomocí XML souborů. Složitější obrazovky, měnící svůj vzhled v reakci na uživatelské akce, mají základní vzhled definován pomocí XML a následně měněn pomocí kódu dané obrazovky.

Použité XML soubory se rozdělují na dva druhy:

- 1) definující vzhled,
- 2) definující rozložení.

Soubory definující uživatelské rozhraní platformy pro systém Windows jsou uloženy v adresáři `MHUrho/Desktop/Data/UI/`. Vzhled prvků rozhraní je definován především v souboru `MainMenuStyle.xml`, spolu s několika menšími soubory pojmenovanými „`...Style.xml`“, definujícími vzhled dynamicky přidávaných prvků. Hlavní styl uživatelského rozhraní `MainMenuStyle.xml` je nastaven v kořeni `UI.Root` jako *defaultní*, je tedy aplikován na všechny prvky v celém stromu, pokud jim není explicitně přiřazen jiný vzhled.

Rozložení prvků je pro každou obrazovku definováno v separátním souboru, pojmenovaném [Jméno obrazovky]`Layout.xml`. Tyto soubory byli vytvořeny za použití editoru enginu Urho3D, který je distribuován spolu s enginem UrhoSharp.

### 3.3.2 Přepínání obrazovek

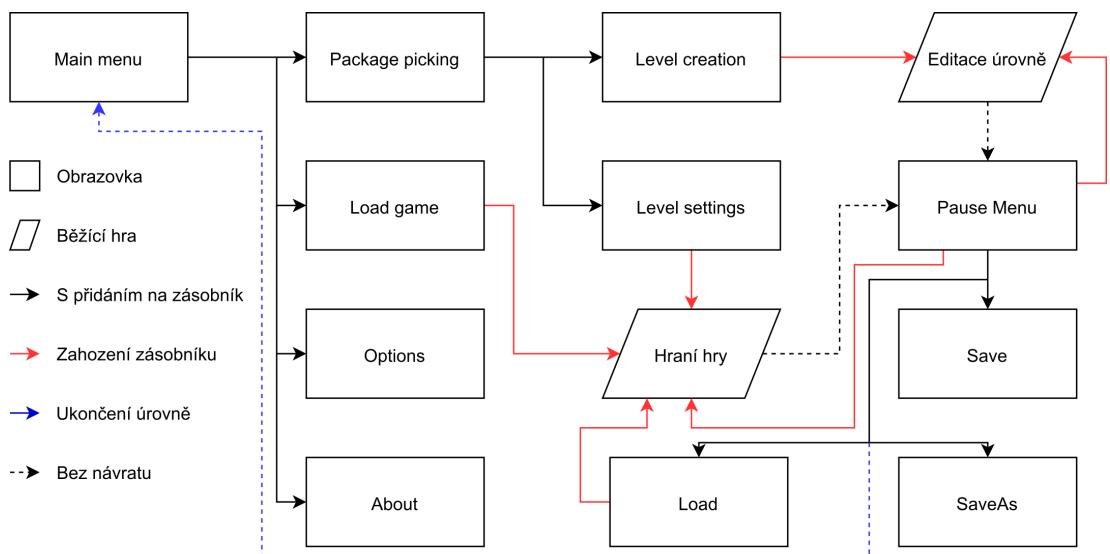
Přepínání obrazovek je jedním z hlavních úkolů třídy `MenuUIManager`. Tato třída poskytuje metody pro přepnutí na každou z obrazovek menu, spolu s uložením aktuální obrazovky na zásobník předchozích obrazovek pro možnost návratu. Každá z metod pro přepnutí obrazovek deklaruje data potřebná pro zobrazení nové obrazovky, která následně poskytne třídě reprezentující danou obrazovku.

Posloupnost možných přechodů obrazovek můžeme vidět na diagramu 3.4. Při spuštění aplikace je počáteční obrazovkou hlavní menu. Jak můžeme vidět na diagramu, z tohoto menu se lze dostat na obrazovky *Package picking*, *Load game*, *Options* a *About*. Dále lze z tohoto menu ukončit celou aplikaci. Každý z černých přechodů přidává starou obrazovku na zásobník obrazovek a nastavuje novou

obrazovku jako `CurrentScreen`, tedy aktuální obrazovku. Tímto způsobem poskytuje třída `MenuUIManager` implementaci tlačítka zpět, které je schopno vracet se v protisměru černých šipek.

Červené šipky v diagramu značí přechod do hry, který vyčistí zásobník obrazovek a zahodí aktuální obrazovku. Tímto je uvolněna veškerá nepotřebná paměť zabraná uživatelským rozhraním menu a je přenechána pro běh hry.

Dále můžeme vidět přerušované šipky, které značí přechody bez možnosti návratu touto cestou. Při pozastavení úrovně je hra převedena do obrazovky *Pauza*. Tato obrazovka se mírně liší podle toho, zda jsme se do ní dostali dostali z editované či hrané úrovně. Tento rozdíl je implementován podle návrhového vzoru *Strategy*.



Obrázek 3.4: Obrazovky menu a přechody mezi nimi.

Vlastní akce přepnutí obrazovky je implementována vytvořením instance implementační třídy, která v rámci svého konstruktoru vytvoří danou obrazovku. Toto vytvoření probíhá za použití souborů definujících rozložení prvků, které jsou voláním `UI.LoadLayoutToElement` použity pro vytvoření objektové reprezentace uživatelského rozhraní uvnitř enginu. Tato objektová reprezentace je následně přístupná i z našeho kódu, a je použitá pro další modifikace, například v reakci na akci uživatele.

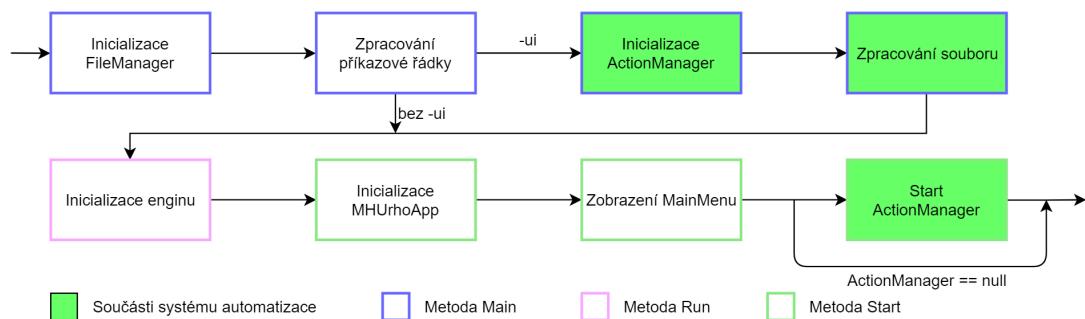
Herní data potřebná pro zobrazení některých obrazovek, například herní balíček při zobrazení výběru úrovní, jsou zpřístupněna jako veřejná *property proxy* tříd. Při zavolání metody `Show` je kontrolována přítomnost a validita poskytnutých dat. Implementační třída při své alokaci dostává referenci na proxy třídu, čímž jí je umožněn přístup k poskytnutým herním datům a je schopna tato data použít pro zobrazení obrazovky.

### 3.3.3 Automatizace uživatelského rozhraní

Pro účely testování byl vytvořen systém pro automatické přepínání obrazovek bez zásahu uživatele. Tento systém je možné použít vytvořením XML souboru a specifikací parametrů příkazové řádky.

Parametrem spouštějícím tento systém je „-ui [d|s] xmlFilePath“, který z kombinace poskytnuté cesty a přepínačů „d|s“ vytvoří cestu, z které nahraje XML soubor. Přepínače „d|s“ určují, zda je cesta „xmlFilePath“ relativní vůči adresáři aplikace či adresáři dynamických dat, blíže popsaném v části 3.4.1. Soubor XML musí být validní vůči schématu `MenuActions.xsd`. Z tohoto souboru jsou nahrány třídy specifikující akce pro jednotlivé obrazovky. Tyto třídy jsou potomkem `MenuScreenAction` a jsou vždy určeny pro konkrétní obrazovku. Implementace každé z obrazovek zná sobě příslušnou třídu a umí v ní zakódovanou akci vykonat. Dále je vytvořena instance `ActionManager`, která slouží jako úložiště těchto nahrávaných tříd a řídí jejich předávání jednotlivým obrazovkám.

Diagram 3.5 je zjednodušením diagramu 3.2 a ukazuje části inicializace platformy, ve kterých je prováděn tento systém. Jak můžeme vidět, při startu aplikace bez parametrů příkazové řádky je tento systém úplně přeskočen.



Obrázek 3.5: Nahrávání akcí v průběhu inicializace platformy.

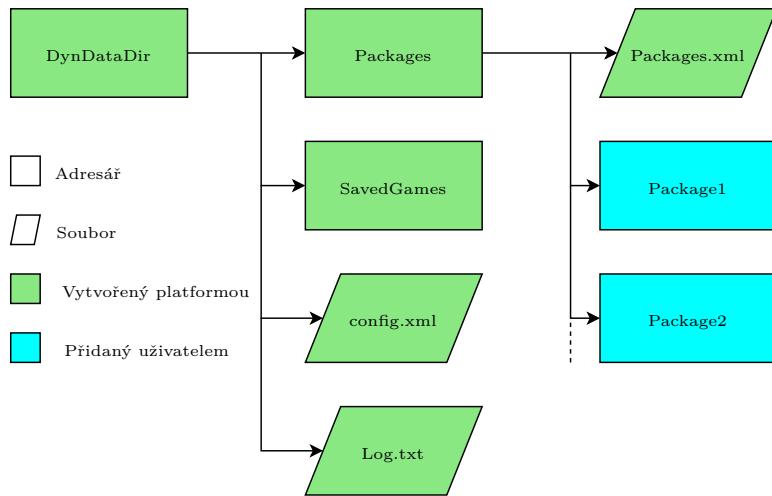
## 3.4 Systém balíčků

V části 2.3.1 jsme popsali základní strukturu systému balíčků, jejich popis pomocí XML souboru a podporované formáty dat pro nahrávání do hry. V této části popíšeme implementaci této funkcionality.

### 3.4.1 Adresář balíčků

Při instalaci hry je vytvořen adresář, ve kterém jsou uložena data generovaná aplikací či přidávaná uživatelem do aplikace. Tento adresář je v kódu označován jako `DynDataDir`, tedy adresář pro dynamická (měnící se) data. Jako součást tohoto adresáře je při instalaci vytvořen podadresář `Packages`, jehož účelem je centralizované umístění balíčků.

Obsah tohoto podadresáře můžeme vidět na diagramu 3.6. Z pohledu platformy nejdůležitějším obsahem je soubor `Packages.xml`, ve kterém jsou uloženy záznamy o všech balíčcích dostupných ve hře. Tento soubor je ve formátu XML, splňujícím schéma `MHUrho/Desktop/Data/Schemas/GamePack.xsd`. Záznamy je možné do tohoto souboru přidávat dvěma způsoby. Prvním je pomocí grafického rozhraní při běhu platformy, konkrétně stisknutím tlačítka `Add` na obrazovce pro výběr balíčků `PackagePickingScreen`. Druhým je manuální editace souboru `Packages.xml`, při které musí být dodrženo schéma souboru. Tímto způsobem je tedy možné přidat balíčky mimo běh platformy.

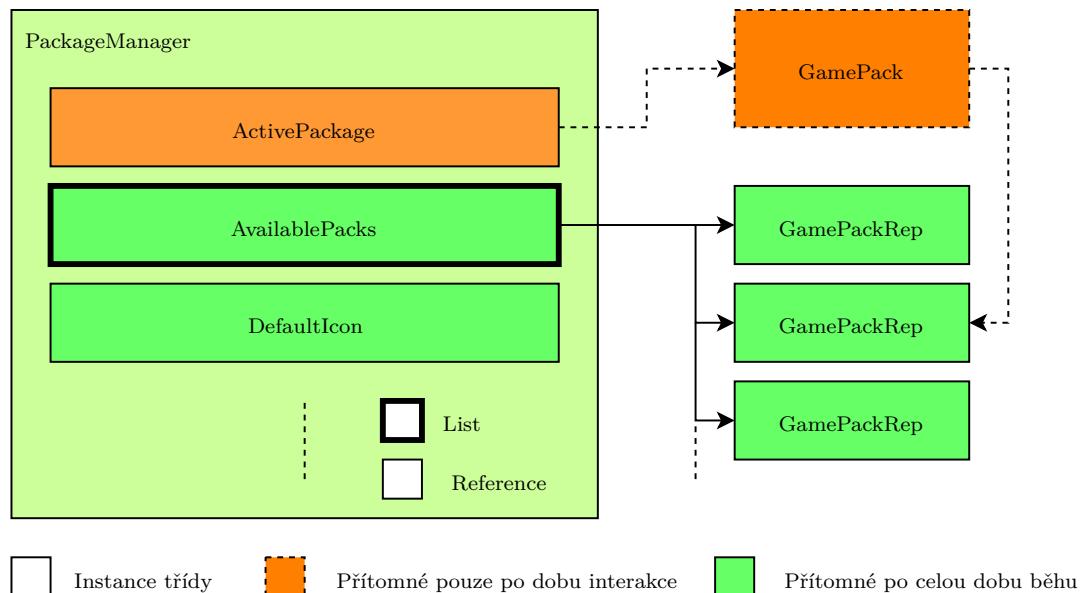


Obrázek 3.6: Typická struktura adresáře DynDataDir.

Tato struktura ukládání balíčků je vystavěna s myšlenkou uložení všech balíčků jako podadresářů adresáře `Packages`. Tato vlastnost ale není nikak vynucována či kontrolována, je tedy možné přidat i balíčky mimo tento adresář. Jedinou překážkou je nutnost použití relativní cesty vůči adresáři `Packages`.

### 3.4.2 PackageManager

Třída `PackageManager` je základem celého systému balíčků. Jak můžeme vidět v části 3.2 na obrázku 3.2, vytvoření instance `PackageManager` je součástí inicializace platformy. První akcí na takto vytvořené instanci je načtení souboru `Packages.xml`, popsaného v předchozí části 3.4.1.



Obrázek 3.7: Datová struktura balíčků.

Na diagramu 3.7 můžeme vidět datovou strukturu systému balíčků. Při načtení souboru `Packages.xml`, případně při přidání balíčku do běžící hry, je pro každý

balíček vytvořena jedna instance třídy `GamePackRep`, která je použita pro prezentaci tohoto balíčku uživateli při výběru pro načtení na obrazovce `PackagePickingScreen`. Instance třídy `GamePackRep` obsahuje pouze data potřebná pro její použití, tedy pro prezentaci balíčku a případné následné načtení balíčku. Těmito daty jsou:

- jméno,
- popis,
- ikona,
- cesta k XML souboru.

Tato data jsou z XML souboru balíčku načtena při startu hry či přidání balíčku a zůstávají přítomna v paměti až do ukončení hry. Zbylá data balíčku jsou načítána až po zvolení jednoho konkrétního balíčku hráčem. Ve hře je vždy nejvíše jeden načtený balíček, dostupný jako `ActivePackage` na třídě `PackageManager`.

Druhou funkcí třídy `PackageManager` je obalení rozhraní třídy `ResourceCache`, poskytované enginem UrhoSharp. Tato třída umožňuje načítání všech druhů assetů podporovaných enginem a jejich cachování. Bohužel přístup k této třídě je omezen na hlavní vlákno aplikace, tedy při přístupu z jiného vlákna, ve kterém implementujeme načítání hry, dochází k pádu aplikace. Pro zamezení chyb při programování načítání jsme vytvořili identické rozhraní na třídě `PackageManager`, které automaticky přesměrovává všechna volání na hlavní vlákno aplikace, kde následně zavolá odpovídající metodu `ResourceCache`.

### 3.4.3 GamePack

Třída `GamePack` představuje kompletně načtený balíček. Jak můžeme vidět na diagramu 3.7, obsahuje tato třída referenci na instanci `GamePackRep` reprezentující daný balíček. Dále obsahuje tyto data:

- typy dlaždic,
- typy jednotek,
- typy budov,
- typy projektilů,
- typy surovin,
- typy hráčů,
- typy logik úrovní,
- seznam úrovní,
- defaultní typ dlaždice,
- textury ikon.

Tato data jsou načítána podle jejich popisu v XML souboru, splňujícího schéma `MHUrho/Desktop/Data/Schemas/GamePack.xsd`. Soubor je vůči tomuto schématu validován jak při načítání reprezentanta balíčku (`GamePackRep`), tak při načítání celého balíčku. Pro zjednodušení manipulace s tímto XML souborem v naší platformě jsme vytvořili třídy v souboru `GamePackXml.cs`, které tvoří objektovou reprezentaci tohoto schématu. Pokud je tedy v elementu `TileType` požadovaný potomek `texturePath`, pak *singleton* instance třídy `TileTypeXml` bude obsahovat položku `TexturePath`, kterou bude možno použít v metodě `XElement.Element()` jako argument pro získání tohoto potomka. Tímto způsobem je izolován zbytek kódu platformy od změn ve formátu či jménech elementů ve schématu. Další výhodou tohoto přístupu, a důvodem použití návrhového vzoru *Singleton*, je reprezentace dědičnosti ve schématu pomocí dědičnosti v jeho objektové reprezentaci.

Při hře slouží třída `GamePack` jako databáze typů, tedy obsahuje metody pro získání reference na typy z výčtu výše podle jména či podle číselného identifikátoru. Jméno i identifikátor jsou načítány z XML souboru, ve kterém je jejich přítomnost vyžadována schématem. Systém typů je blíže popsán v části 3.5.3.

Poslední funkcí je přístup k úrovním obsaženým v balíčku. Úrovně jsou, podobně jako balíčky samotné, před vlastním načtením reprezentovány separátní třídou `LevelRep`. Tato třída, podobně jako `GamePackRep`, obsahuje data potřebná pro výběr a nastavení úrovně před jejím spuštěním. Dále tato třída umožňuje manipulaci s úrovní z pohledu assetu, tedy její ukládání při editaci, vytvoření kopie pro editaci, načtení pro hru atd. Toto chování je implementováno podle návrhového vzoru *State*.

### 3.4.4 Assety

Pojmem assety označujeme v naší práci veškerá data použitelná pro implementaci hry. Patří mezi ně například:

- 3D modely,
- textury,
- XML data,
- části logiky poskytované enginem.

Každý typ entity, tedy budovy, jednotky či projektilu, má definovanou svou množinu assetů, které ji reprezentují při zobrazení či při výpočtu kolizí. Pro dodání assetů instancím těchto typů poskytujeme tři způsoby definice.

Prvním způsobem je explicitní specifikace assetů přímo v XML souboru balíčku. Tento způsob je implementován třídou `ItemsAssetContainer`, která z XML elementu vyčte všechna potřebná data a při požadavku vytvoří reprezentaci entity v herním enginu. Bližší popis reprezentace hry v herním enginu můžete nalézt v části 3.5.1.

Druhým způsobem je využití funkcionality poskytované herním enginem a s ním distribuovaným editorem. V tomto editoru lze definovat reprezentaci entity a následně tuto definici serializovat do XML či binárního souboru, do takzvaného *prefab*, neboli prefabrikátu. Pro tento způsob poskytujeme dvě třídy,

`XmlPrefabAssetContainer` a `BinaryPrefabAssetContainer`, které obalí soubor definující reprezentaci entity a při žádosti o vytvoření instance entity použijí engine a tyto soubory pro její vytvoření.

Použitý způsob vytváření je specifikován v XML popisu typu entity, a v závislosti na zvoleném způsobu jsou dále požadována další data, specifická pro daný způsob. Použití XML či Binary prefab umožňuje využití všech možností UrhoSharp enginu. Námi implementovaný `ItemsAssetContainer` umožňuje zjednodušenou specifikaci bez externích nástrojů, jakým je editor Urho3D, zato ale omezuje použitelné součásti enginu na tuto podmnožinu:

- `Static model`,
- `Animated model`,
- `Collision shape`,
- nastavení `Scale`.

Ke každému z modelů navíc umožňuje definovat jeho textury. Tato podmnožina je podle nás dostatečná pro tvorbu graficky jednodušších her,

### 3.4.5 Načítání balíčku

Načítání balíčku je implementováno metodou `Load`, jejíž prototyp je:

```
public static async Task<GamePack> Load(string pathToXml,  
GamePackRep gamePackRep,  
XmlSchemaSet schemas,  
IProgressEventWatcher loadingProgress = null)
```

Už podle deklarace metody vidíme, že se implementace pokouší o asynchronní načítání balíčku. Nejdříve se o úplné asynchronní načítání v pravém slova smyslu, protože drtivou většinu dat je nutno načítat na hlavním vlákně z důvodu restrikcí herního enginu, blíže popsáným v části 3.4.2. Hlavním cílem je tedy rozdělit načítání na části, mezi kterými je platforma schopna nadále zpracovávat vstup od hráče. Tímto se snažíme zamezit tzv. „zamrzání“ aplikace, kdy aplikace z pohledu uživatele nic nedělá a odmítá reagovat na jakýkoli vstup od hráče. V prvních verzích bez tohoto přístupu jsme narázeli na problém, kdy systém Windows pořádoval náš proces za mrtvý a navrhoval nám jej zabít. Touto implementací jsme tomuto problému zabránili a navíc jsme umožnili notifikovat hráče o průběhu načítání.

Z pohledu implementace vlastního načítání je princip vcelku jednoduchý. Po-sloupnost akcí je následovná:

- 1) načíst a validovat XML soubor;
- 2) načíst typy dlaždic, jednotek, budov, projektů, surovin;
- 3) načíst typy logik hráčů a úrovní;
- 4) načíst textury ikon;

5) načíst reprezentanty úrovní,

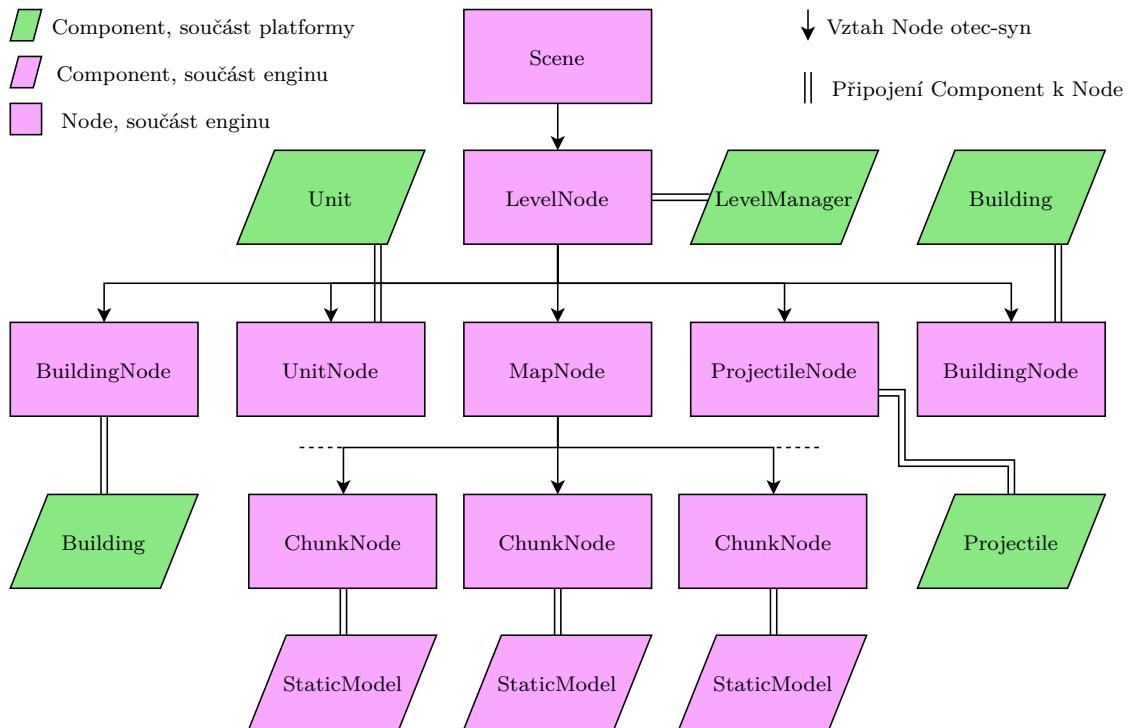
6) zavřít XML soubor.

Při jakékoli chybě načítání, ať už nevalidnímu XML či chybějícímu souboru některého assetu, je načítání zastaveno, dosud načtená data znovu odalokována, chyba zalogována a zpět vypuštěna výjimka typu `PackageLoadingException`, signalizující chybu při načítání balíčku.

## 3.5 Logika hry

V této sekci popíšeme implementaci vlastního průběhu hry, nejdříve z pohledu herního enginu a následně z pohledu naší platformy. Oba tyto pohledy jsou poskytnuty tvůrcům balíčků pro tvorbu pluginů.

### 3.5.1 Herní engine



Obrázek 3.8: Reprezentace hry z pohledu herního enginu.

Diagram 3.8 ukazuje reprezentaci hry z pohledu herního enginu. Aktuálně spuštěná úroveň je v herním enginu reprezentována tzv. *Scene graph* strukturou, tedy grafem scény. Tento graf je strom s vrcholy typu `Node`. Kořenem tohoto stromu je instance třídy `Scene`, která je specializací `Node`.

Diagram 3.8 ukazuje část grafu scény, ve které můžeme vidět jeho základní strukturu. Názvy v instancích třídy `Node` ukazují, který herní objekt daná instance reprezentuje. Všechny jsou ale instancí stejné třídy `Node`. Oproti tomu názvy komponentů ukazují jméno třídy, jejíž jsou instancí. Tyto třídy jsou pouze potomkem třídy `Component`. Tento rozdíl vychází z návrhu enginu, ve kterém je k

přidávání uživatelské logiky zamýšlena třída **Component**. Třída **Node** obecně není určena pro vytváření potomků.

Na diagramu můžeme vidět stav hry, ve kterém jsou v herním světě umístěny dvě budovy, jednotka a projektil. Vztah **Component** a **Node** v naší platformě je blíže popsán v části 3.5.2. Dále můžeme vidět mapu a její reprezentaci. Tato reprezentace je blíže popsána v části 2.4.1 a 3.6. Jak můžeme vidět na diagramu, lze **Node** použít k reprezentaci jedné entity, částí entit nebo skupiny entit. Příkladem reprezentace částí může být právě rozdělení mapy na „**Chunky**“. Budovy, jednotky či projektily mohou mít pod touto jednou **Node** také přidány podstrom, reprezentující jejich součást. Za reprezentaci skupiny entit jednou **Node** můžeme považovat situaci, kdy kamera sleduje jednotku. V tu chvíli **Node** dané jednotky reprezentuje jak jednotku samotnou, tak kamery, a obě současně přesouvá v herním světě.

Každá instance třídy **Node** má několik atributů, které umožňují její umístění v herním světě a další manipulaci. Nejdůležitějšími z těchto atributů jsou:

- **Position** (pozice),
- **Rotation** (rotace),
- **Scale** (velikost),
- **Enabled**.

Již podle názvu slouží **Position** a **Rotation** pro umístění dané **Node** do herního světa a určení jejího otočení. Obě tyto vlastnosti jsou počítány vůči rodičovské **Node**, jak můžeme vidět na obrázku 3.9. Atribut **Scale** určuje závislost souřadnic v podstromu této **Node** na souřadnicích vzhledem k souřadnicím v podstromu otce této **Node**. Na ukázce můžeme vidět dvě **Node**, kdy levá horní **Node** je otcem pravé spodní **Node**. Otec je umístěn na pozici (1,0,1) a má nastaven **Scale** na (2,2,2). Potomek má vlastní atribut **Position** nastaven na (1,0,1). Výslednou pozici potomka v herním světě, označovanou jako **WorldPosition**, pak získáme tímto výpočtem:

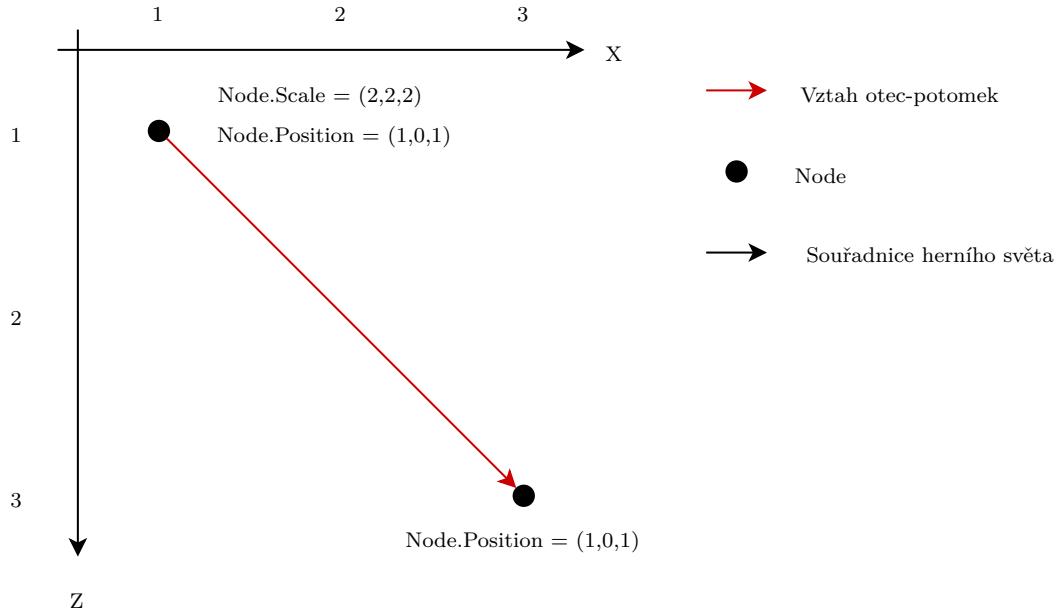
$$\text{Potomek.WorldPosition} = \text{Otec.WorldPosition} + \text{Potomek.Position} * \text{Předek.Scale}$$

Díky tomu bude výsledná pozice potomka v herním světě (3,0,3). Použitá **WorldPosition** otce je vypočítána stejným způsobem.

Pro implementaci chování hry lze na každou instanci třídy **Node** připojit potomky třídy **Component**. Hlavním způsobem implementace logiky při použití herního enginu UrhoSharp je vytvoření vlastních specializací třídy **Component**, implementace virtuálních metod a přidání této naší komponenty k instanci **Node** v grafu scény. Tímto způsobem je vytvořena také implementace naší platformy, blíže popsána v části 3.5.2.

Engine sám poskytuje několik základních typů komponent, implementujících nejčastější potřeby her. Těmito typy jsou:

- **StaticModel** pro vykreslování bez animací,
- **AnimatedModel** pro vykreslování s animacemi,



Obrázek 3.9: Ukázka výpočtu pozice `Node` v herním světě.

- `RigidBody` pro simulaci fyziky,
- `CollisionShape` pro výpočet kolizí ve spolupráci s `RigidBody`,
- `Camera` pro vykreslení na obrazovku,
- `Light` pro přidání světla.

Jak bylo řečeno v části 3.4.4, umožňuje platforma tvůrcům balíčků specifikovat assety příslušící jednotlivým typům jednotek mnoha způsoby. Všechny tyto způsoby ale využívají tento popis právě k vytvoření instancí `Node` a `Component`, které jsou spojeny a inicializovány podle uložených dat.

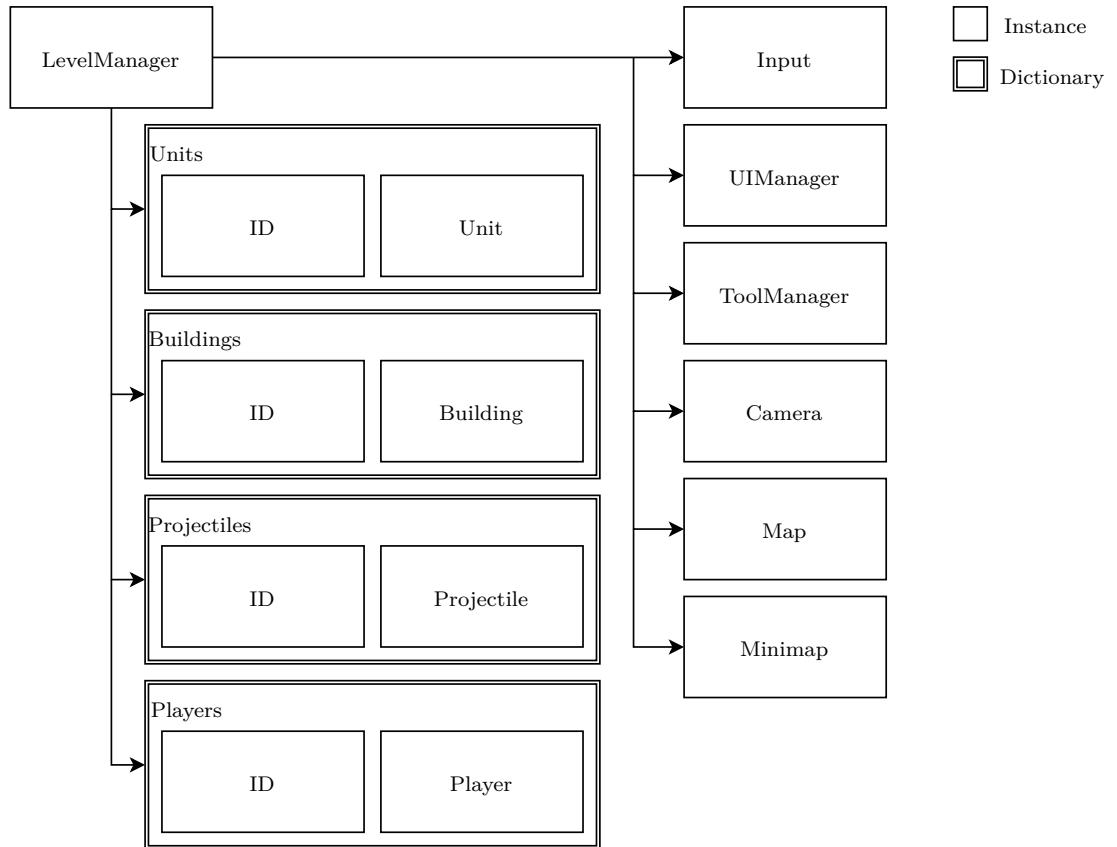
Vlastní výpočet stavu hry je iniciován enginem, který ve všech prvcích grafu scény, tedy `Node` i `Component`, zavolá jejich virtuální metodu `OnUpdate`. Tato metoda je volána s parametrem `timeStep`, který představuje čas uběhlý od předchozího výpočtu stavu. V rámci této metody provádí standardní i uživatelské komponenty výpočet následujícího stavu.

Zmíněný atribut `Enabled` třídy `Node` ovlivňuje šíření volání metody `OnUpdate`. Pokud je tento atribut nastaven na hodnotu `false`, není na dané instanci `Node` ani na jejím podstromu proveden „update“. Tento atribut je přítomný i na třídě `Component`, na které ovládá volání `OnUpdate` na dané instanci `Component`.

### 3.5.2 Pohled platformy

Z pohledu platformy se hra skládá z několika částí, ilustrovaných na diagramu 3.10. Vztah některých těchto částí s grafem scény popsáným v části 3.5.1 můžeme vidět na diagramu 3.8.

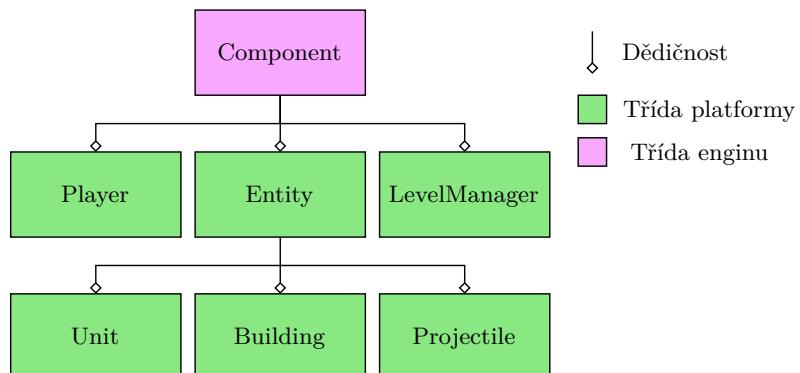
Centrální částí je instance třídy `LevelManager`, která slouží jako přístupový bod ke všem součástem tvořících implementaci platformy. Dále tato třída slouží jako databáze všech budov, jednotek, projektilů a hráčů přítomných ve hře., Pro



Obrázek 3.10: Třída LevelManager.

budovy, jednotky a projektily umožňuje dále jejich vytvoření či zničení. Jak můžeme vidět na diagramu 3.8, je tato třída potomkem třídy Component a je připojena k Node obsahující jako podstrom celý zbytek herního světa. Tímto způsobem je pohled platformy spojen s pohledem herního enginu.

Každá entita, tedy každá budova, jednotka či projektil je reprezentována instancí odpovídající třídy implementující chování tohoto druhu. Těmito třídami jsou Building, Unit a Projectile. Jak můžeme vidět na diagramu 3.11, jsou všechny tyto třídy potomkem třídy Entity, která je sama potomkem třídy Component. Tímto způsobem je provázána reprezentace entity z pohledu herního enginu a z pohledu platformy.



Obrázek 3.11: Potomci třídy Component v platformě.

### 3.5.3 Typy

Platforma zavádí systém typů herních prvků. Tento systém je separován od typového systému jazyka a je aplikován pro tyto prvky:

- logiky úrovně (`LevelLogicType`),
- hráče (`PlayerType`),
- dlaždice (`TileType`),
- budovy (`BuildingType`),
- jednotky (`UnitType`),
- projektily (`ProjectileType`),
- suroviny (`ResourceType`).

Pro každý z druhů prvků definuje typ jejich vzhled a/nebo chování. Vzhled je definován pomocí množiny assetů specifikovaných v XML elementu definujícím tento typ, které jsou následně použity při vytváření herních prvků tohoto typu. Chování je definováno pomocí systému pluginů, popsaných v části 3.5.4.

Výjimkou z tohoto pravidla jsou typy surovin, které ve hře nemají vlastní instance a jsou používány pouze jako klíč pro přístup k množství daného typu suroviny vlastněného hráčem.

Každá instance výše vyjmenovaných druhů herních prvků obsahuje referenci na svůj typ. Příklad můžeme vidět na diagramu 3.14, kde vidíme instance `Building`, tedy budov, které obsahují reference na instance `BuildingType` reprezentující typ budov `Gate`, tedy brána, nebo `Wall`, tedy zed.

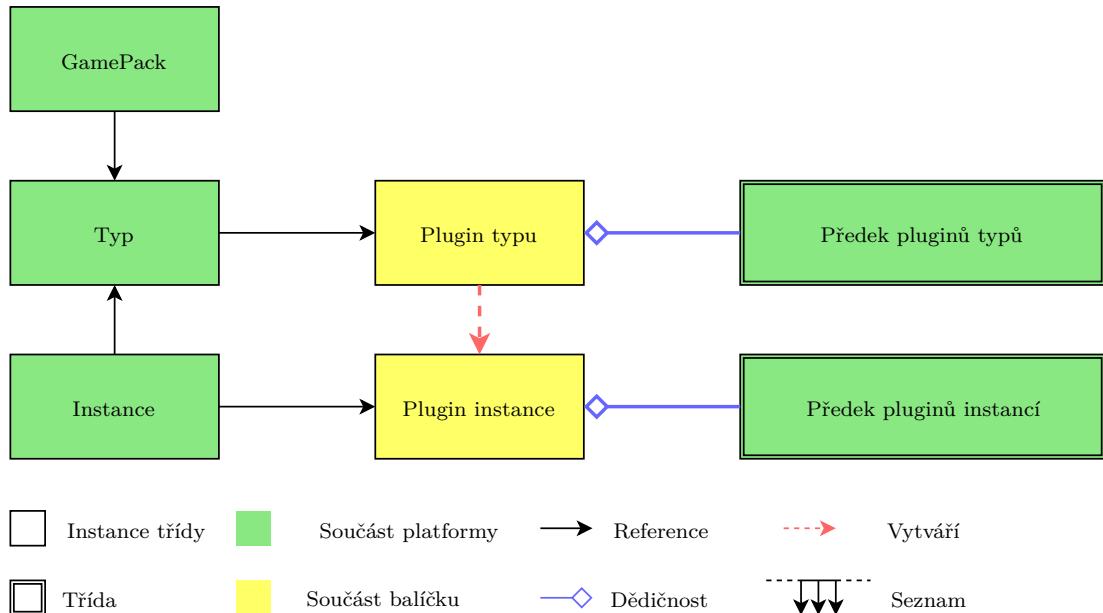
Typy jsou jednou z hlavních částí obsahu balíčku. Instance reprezentující jednotlivé typy jsou vytvořeny při načtení balíčku a zanikají při odalokování balíčku.

### 3.5.4 Pluginy

Hlavním cílem naší práce bylo umožnit tvůrcům balíčků použít jazyk C# pro tvorbu logiky hry ve formě pluginů. Návrh tohoto systému a použité technologie byl popsán v části 2.3.3. Platforma definuje dva druhy pluginů:

- 1) pluginy typů,
- 2) pluginy instancí.

Ukázku zapojení těchto pluginů do struktury hry můžeme vidět na diagramech 3.12 a 3.14. Diagram 3.12 ukazuje základní princip propojení pluginů, instancí herních prvků a typů těchto herních prvků. Tyto vztahy jsou dále popsány v následující části 3.5.4. Diagram 3.14 obsahuje ukázku několika budov v herním světě různých typů s jejich pluginy.



Obrázek 3.12: Princip zapojení pluginů do struktury hry.

### Pluginy typů

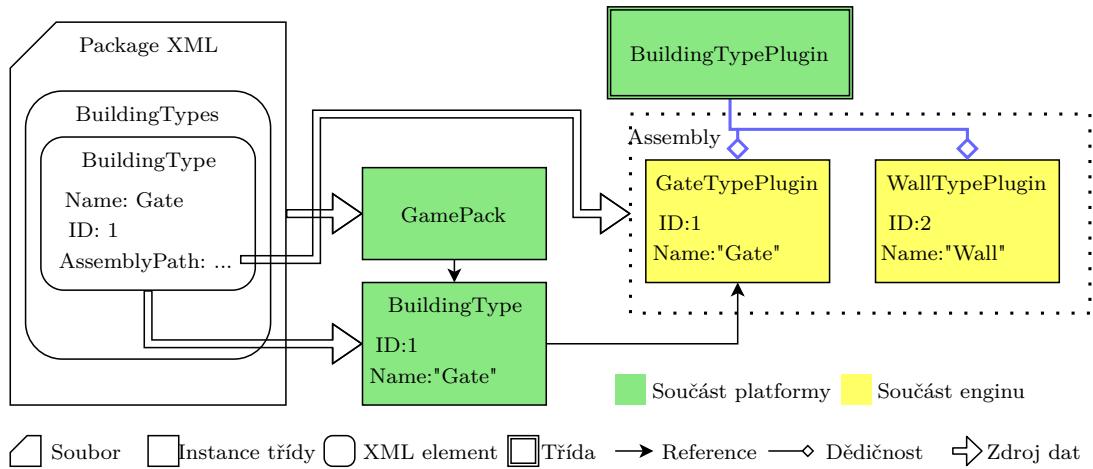
Jak jsme popsali v části 3.5.3, velká část herních prvků má určený svůj typ. XML definice každého z těchto typů podle schématu vyžaduje specifikaci assembly, která je při načítání balíčku pomocí systému *Reflection* nahrána do procesu platformy. Tato akce je blíže popsána v části 2.3.3. V této assembly je následně podle *ID* a jména typu nalezena třída, která slouží jako typový plugin tohoto typu.

Tato třída musí být potomkem jedné z těchto tříd:

- `LevelLogicTypePlugin`,
- `PlayerAITypePlugin`,
- `BuildingTypePlugin`,
- `UnitTypePlugin`,
- `ProjectileTypePlugin`.

Dále je vyžadováno, aby tato třída měla bezparametrický konstruktor. Inicializace je implementována separátní metodou `Initialize`, které je předán XML element `<extension>` z definice typu. Tento element slouží pro uložení tvůrcem balíčku definovaných dat a jeho obsah není nijak omezen či validován.

Příklad můžeme vidět na diagramu 3.14 a 3.13, kde jsou v balíčku definovány dva typy budov, *Gate*, neboli brána, a *Wall*, neboli zed. Při vytváření instance `BuildingType` pro každý z těchto typů je z cesty uvedené v XML elementu `assemblyPath` v záznamu daného typu nahrána odpovídající assembly. V této assembly jsou nalezeny všechny typy odvozené od jedné z výše uvedených tříd platformy, v naší ukázce odvozené od třídy `BuildingTypePlugin`. Od každé z těchto tříd je vytvořena instance mající stejné hodnoty v položkách `Name` a `ID`



Obrázek 3.13: Načítání pluginu typu.

jako hodnoty načtené z XML typu. Instance této třídy je následně připojena do atributu `Plugin` instance `BuildingType` reprezentující daný typ.

Hlavním použitím pluginu typu jsou jeho metody `CreateNewInstance` a `GetInstanceForLoc`, které vytváří pluginy pro instance herních prvků daného typu. Tento systém byl navržen podle návrhových vzorů *Factory method* a *Factory*.

Dále je např. u `BuildingTypePlugin` definována metoda `CanBuild`, která dostává aktuální úroveň a definovanou pozici v mapě a je používána pro zjištění, zda je možné v této pozici vytvořit budovu.

### Pluginy instancí

Každá instance následujících tříd má při svém vytváření přiřazenu svoji prioritní instanci třídy z balíčku jako plugin. Těmito třídami jsou:

- 1) `LevelManager`,
- 2) `Player`,
- 3) `Building`,
- 4) `Unit`,
- 5) `Projectile`.

Třída z balíčku sloužící jako plugin instance některé z předcházejících tříd musí být potomkem odpovídající z těchto tříd:

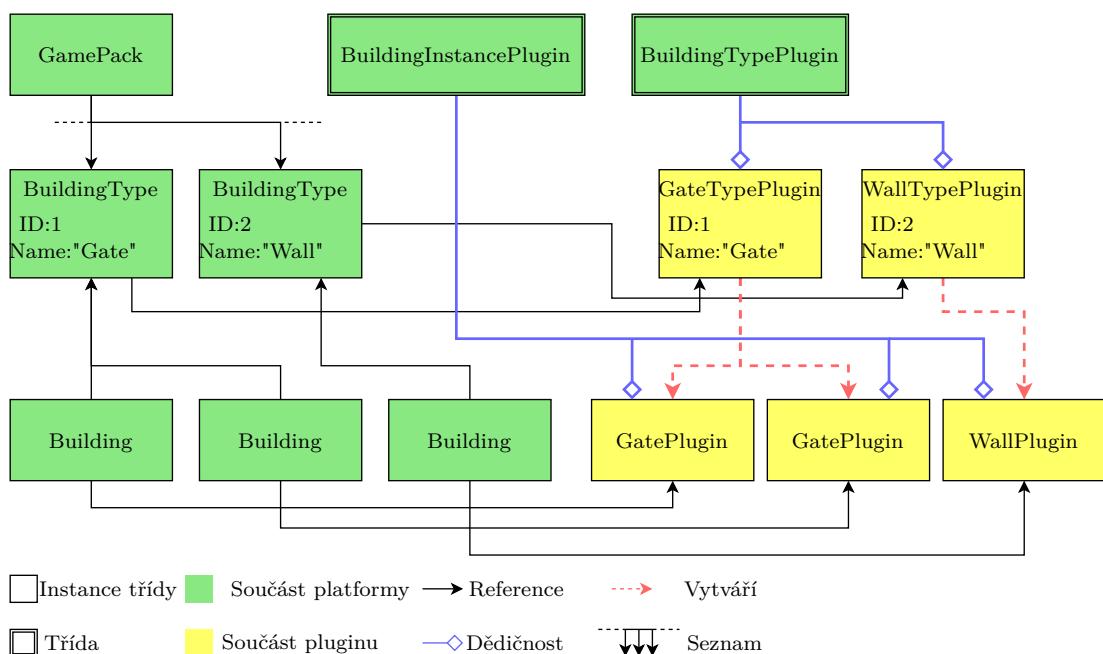
- 1) `LevelLogicInstancePlugin`,
- 2) `PlayerAIInstancePlugin`,
- 3) `BuildingInstancePlugin`,
- 4) `UnitInstancePlugin`,
- 5) `ProjectileInstancePlugin`.

Tyto třídy mají definovány virtuální metody, které jsou volány při určitých událostech v herním světě, které se týkají daného herního prvku. Implementací těchto metod mohou tvůrci balíčků reagovat na tyto události a tím implementovat chování těchto herních prvků.

Jednou z hlavních metod pro implementaci logiky v pluginech je metoda `OnUpdate`. Jak můžeme vidět na diagramu ??, je tato metoda volána z metod `OnUpdate` výše vyjmenovaných potomků třídy `Component`. Třída `Component` a její metoda `OnUpdate` jsou blíže popsány v části 3.5.1. Metoda `OnUpdate` u pluginů má stejnou sémantiku, je tedy volána při každém výpočtu stavu hry, kde jako parametr dostává čas uběhlý od předcházejícího výpočtu stavu.

Množina metod je různá pro různé typy herních prvků, příkladem ale může být událost přidání jednotky hráče, odstranění jednotky hráče či změna objemu surovin vlastněných hráčem.

Jak bylo popsáno v části 3.5.3, obsahuje každá instance herního prvku referenci na svůj typ. Tento typ dále obsahuje referenci na svůj typový plugin, jak bylo popsáno v části 3.5.4 a jak můžeme vidět na diagramech 3.14 a 3.13. Pro vytvoření pluginu pro vytvářenou instanci herního prvku je použita jedna z `Factory` metod pluginu typu.

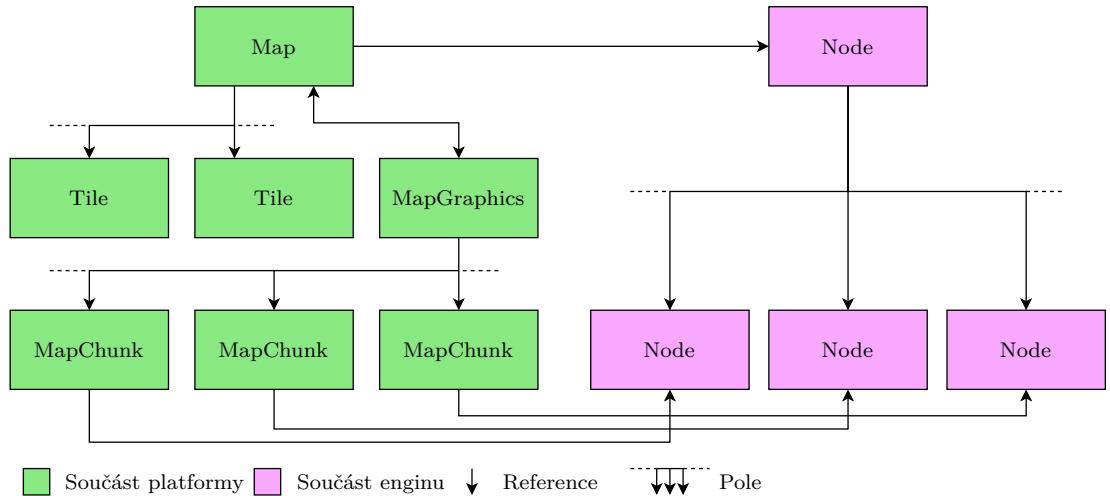


Obrázek 3.14: Ukázka zapojení pluginů budov do struktury hry.

## 3.6 Mapa

Jak můžeme vidět na diagramu 3.15, mapa je v naší platformě reprezentována instancí třídy `Map`. Tato instance je přístupná všem součástem platformy i pluginům jako položka třídy `LevelManager`, jak můžeme vidět na diagramu 3.10.

Jak jsme popsali v části 2.4.1, je naše implementace mapy obdélníkového tvaru, rozdelená na čtvercové dlaždice. Tyto dlaždice můžeme na diagramu 3.15 vidět jako privátní položku třídy `Map`. Dlaždice jsou uloženy v jednorozměrném



Obrázek 3.15: Implementace mapy.

poli, k jehož indexaci poskytuje třída `Map` několik pomocných metod, které převádějí pozici v souřadnicích herního světa do indexu dlaždice v tomto poli.

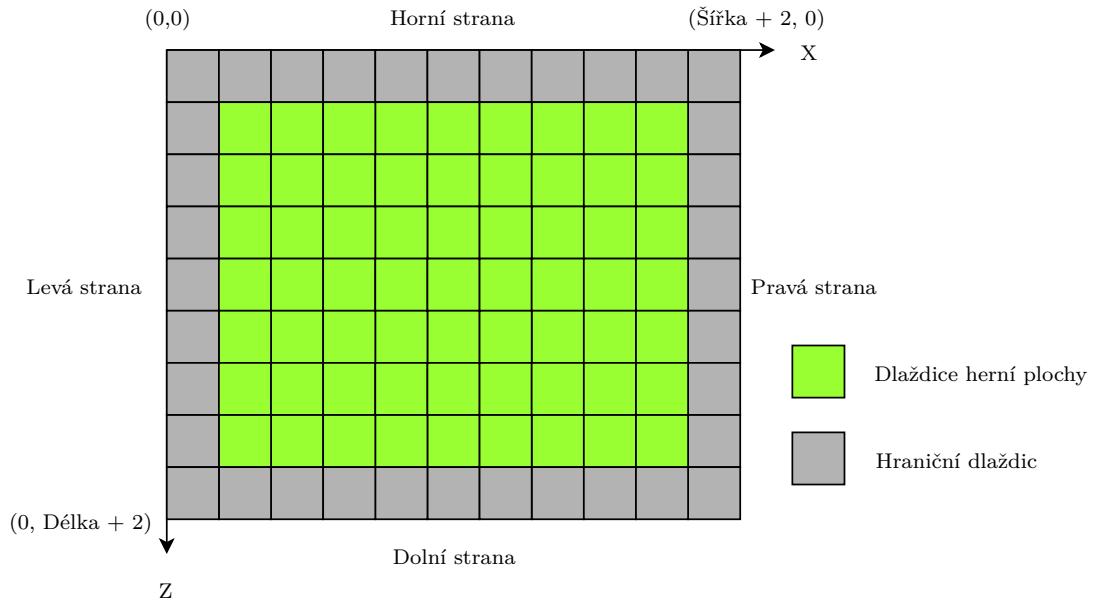
Při popisování a implementaci mapy jsou využívány termíny jako horní levý roh, pravý spodní roh, vršek dlaždice a podobné. Tyto termíny vycházejí z představy ilustrované obrázkem 3.16, kdy rovinu mapy položíme do roviny monitoru a levý horní roh monitoru určíme jako počátek, tedy bod (0,0). Tato představa vychází z jedné z prvních implementací naší platformy, kdy byla mapa reprezentována pouze dvojrozměrně a přišlo nám logické využívat stejný systém souřadnic jako je používán při definici prvků v uživatelském rozhraní. Bohužel tuto představu nelze jednoduše reprezentovat ve 3D prostoru, kdy pro pohled světa odpovídající výše popsané představě, tedy levý horní roh v levém horním rohu obrazovky, je nutné dívat se na mapu ze spodní strany, tedy s kamerou pod úrovní terénu. Dalším problémem ve 3D je poloha herní roviny, která se nachází v rovině definované osami X a Z, kde osa Y udává výšku nad rovinou. Toto se projevuje u některých metod při pojmenování jejich parametrů a jejich volání, kdy souřadnice Y dvourozměrného vrcholu je předávána do parametru se jménem Z metody operující v rovině mapy. Pro úplnost tedy:

- levá strana je strana s nižší souřadnicí X,
- pravá strana je strana s vyšší souřadnicí X,
- horní strana je strana s nižší souřadnicí Z,
- dolní strana je strana s vyšší souřadnicí Z.

Dále můžeme na obrázku 3.16 vidět dva různé druhy dlaždic. Toto rozdělené bude popsáno v následující části 3.6.1.

### 3.6.1 Dlaždice

Jednotlivé dlaždice mapy jsou reprezentovány instancí třídy `Tile`. Tyto instance obsahují několik atributů popisujících danou dlaždici:



Obrázek 3.16: Pohled třídy `Map` na reprezentaci mapy.

- 1) seznam jednotek přítomných na dlaždici,
- 2) referenci na budovu přítomnou na dlaždici,
- 3) referenci na typ dlaždice,
- 4) pozici v herním světě,
- 5) výšku každého ze svých rohů.

Jak můžeme vidět na obrázku 3.16, jsou všechny dlaždice čtvercového tvaru s velikostí hrany 1. Pro indexaci do pole obsahujícím instance třídy `Tile` používáme souřadnice levého horního rohu dlaždice. Protože je tento výběr rohu v podstatě náhodný, poskytuje třída `Tile` položku `MapLocation`, odstínující nás od výběru rohu reprezentujícího dlaždice.

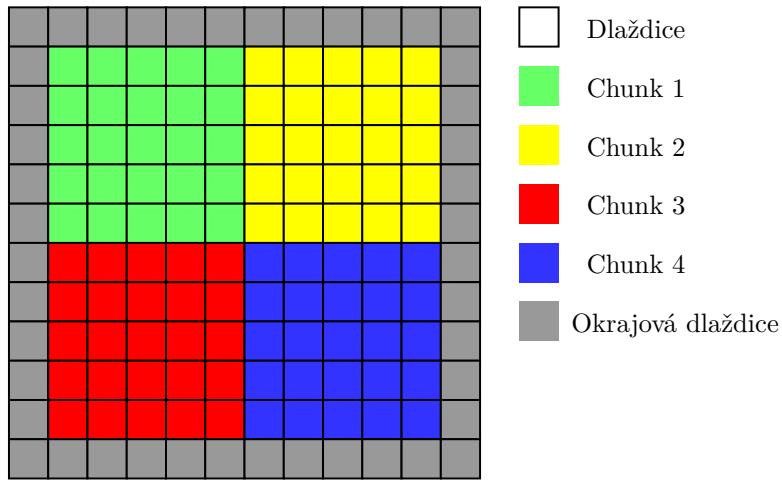
Dále nám umístění rohů dlaždic na celočíselné souřadnice a jejich jednotková velikost umožňují jednoduché zjištění dlaždice obsahující libovolný bod v herním světě. Použitím operace dolní celé části na souřadnice  $X$  a  $Z$  daného bodu získáme horní levý roh dlaždice, která tento bod obsahuje.

Pro redukci duplikace informací obsahuje každá dlaždice informaci pouze o výšce svého levého horního rohu. Ostatní rohy jsou získávány od sousedních souřadnic, jejichž levé rohy se nachází ve stejné pozici jako zkoumaný roh dané dlaždice. Tímto způsobem máme zajištěnu celistvost terénu a předcházíme tak možným programátorským chybám. Nevýhodou této implementace je nutnost speciálních dlaždic na okraji mapy, které můžeme vidět na obrázku 3.16 označené šedou barvou. V naší implementaci jsou tyto dlaždice reprezentovány instancemi třídy `BorderTile`. Tyto dlaždice nejsou viditelné z pohledu hráče či pluginů, jejich jedinou funkcí je udržování informace o výšce všech svých rohů a tím rohů sousedních herních dlaždic.

### 3.6.2 Zobrazení

Zobrazení mapy je implementováno třídou `MapGraphics`, která je vnitřní privátní třídou třídy `Map`. Tento vztah je použit pro přístup k privátním položkám a zjištění celkového stavu `Map` pro jeho zobrazení. Obrázek 3.17 ilustruje systém zobrazení mapy. Jak můžeme vidět, mapa je rozdělena na určitý počet částí stejné velikosti. Tyto části označujeme jako *Chunk* a v kódu jsou reprezentovány instancemi třídy `MapChunk`. Důvod rozdělení mapy a omezení z toho plynoucí jsou rozebrána v části 2.4.1.

Rozdělení na chunky je pouze grafickou záležitostí, proto se netýká okrajových dlaždic mapy, jak můžeme vidět na diagramu 3.17. Díky tomu se omezení velikosti mapy, popsáne v části 2.4.1 týká rozměrů samotné hrací plochy.



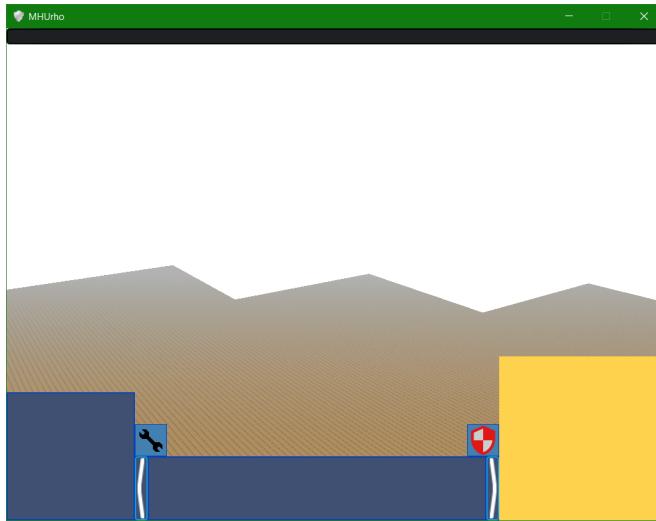
Obrázek 3.17: Princip přiřazení dlaždic do chunků.

Každý z chunků odpovídá jedné instanci `Node`, která je potomkem `Node` mapy. Toto rozdělení můžeme vidět na obrázku 3.8. Každá z těchto `Node` obsahuje právě jeden `StaticModel`, použitý pro zobrazení odpovídajícího chunku. Tento model je vytvořen v průběhu volání konstruktoru třídy `MapChunk`, za využití dynamické generace vertex a index bufferů. Jak bylo popsáno v části 2.4.1, používá naše implementace pro vygenerování a úpravy grafické reprezentace mapy přímý *unsafe* přístup k vertex a index bufferům, v kódu reprezentovaných instancemi třídy `VertexBuffer` a `IndexBuffer`. Při generování či načítání mapy jsou tyto buffery vytvořeny ve velikosti odpovídající potřebnému počtu vrcholů potřebných pro zobrazení všech dlaždic chunku.

Ukázkou chunků ve hře můžeme vidět na obrázku 3.18, kde se v uživatelem nastavené vzdálenosti od kamery přestávají vykreslovat, čímž šetří zdroje při vykreslování.

Jak jsme popsali v části 2.4.1, je každá dlaždice reprezentována čtyřmi vertexy. Z těchto čtyřech vertexů jsou následně za použití indexů v `IndexBufferu` vytvořeny dva trojúhelníky. Rozdělení na trojúhelníky je provedeno podle vyšší diagonály, tedy podle diagonály, která je uprostřed dlaždice výše.

Chunky dále umožňují zamknout jejich vertex a index buffer do paměti, pro umožnění modifikace výšek dlaždic. Při každé změně výšky vertexů dlaždice je výše zmíněné rozdělení dlaždic na trojúhelníky přepočítáváno.



Obrázek 3.18: Ukázka rozdělení mapy na *Chunky*.

## 3.7 Hledání cesty

Systém hledání cest je v naší platformě implementován s důrazem na rozšiřitelnost tvůrci balíčků, jak bylo popsáno v části 2.4.2. Základem této implementace je rozhraní `IPathFindAlg`, které musí být implementováno kterýmkoli algoritmem, který chce tvůrce balíčku použít jako algoritmus pro hledání cesty ve svém balíčku.

Rozhraní `IPathFindAlg` je navrženo pro podporu mnoha algoritmů pro hledání nejkratší cesty. Z tohoto důvodu je k rozhraní `IPathFindAlg` vytvořeno rozhraní pro reprezentaci grafu, které bude implementováno každým z možných algoritmů. V rámci jednoho běhu úrovně je podporován pouze jediný algoritmus, což umožňuje implementaci algoritmu využít znalosti opravdových typů objektů, které získává jako reference na rozhraní. Následně je možné využít přetypování pro přístup ke konkrétním typům objektů specifických pro daný algoritmus.

Jak bylo popsáno v části 2.4.2, reprezentaci grafu je možné generovat z logické reprezentace mapy dvěma způsoby, a to staticky či dynamicky. Návrh rozhraní v naší platformě je cílen spíše pro dynamické generování grafu, předpokládáme ovšem, že by bylo možné jeho využití i pro statické generování. Rozhraní pro reprezentaci grafu se skládá ze tří částí. Těmito částmi jsou:

- 1) reprezentace vrcholů a hran,
- 2) ohodnocení hran,
- 3) reprezentace cesty v grafu.

### 3.7.1 Reprezentace vrcholů a hran

Reprezentace vrcholů a hran tvoří statickou část generování grafu. Vrcholy grafu jsou dvou typů. Prvním je `ITileNode`, reprezentující dlaždice mapy a druhým je `IBuildingNode`, reprezentující dostupné části budov. Vzhledem k tomu, že velikost mapy, a tedy počet dlaždic není možné měnit, předpokládáme, že tyto vrcholy bude každý algoritmus generovat jednou, při své konstrukci. Existence

vrcholů typu `IBuildingNode` závisí na životnosti jimi reprezentovaných budov. Předpokládáme tedy, že budou vytvářeny při stavbě budov a mazány při zničení těchto budov.

Posledním typem vrcholu je `ITempNode`, využívaná pro přesnější reprezentaci hrany v herním světě. Konkrétní sémantika a životnost vrcholů je ale v rukou tvůrce konkrétního algoritmu. Naše platforma nijak tyto vlastnosti nekontroluje.

Každá hrana mezi dvěma vrcholy má přiřazen způsob pohybu. V aktuální implementaci podporujeme explicitně pouze dva typy pohybu, a to pohyb lineární a teleportaci. Význam těchto typů pohybu není daný algoritmem vyhledávání cesty, ale koncovou implementací pohybu jednotek. Platforma poskytuje jednu možnou implementaci pohybu jednotek pomocí komponenty `WorldWalker`, popsáne v části 3.11.

Pro přidávání akcí nad hranami grafu, tedy dvojicemi vrcholů, je definováno rozhraní podle návrhového vzoru `Visitor`. Dále je podpora tohoto návrhového vzoru požadována rozhraním jednotlivých `INode`. Jedním z možných využití rozhraní `Visitor` je tzv. *double dispatch*, který umožňuje jednodušší implementaci `INodeDistCalculator`.

### 3.7.2 Dynamická část generování

Dynamická část generování grafu je tvořena rozhraním `INodeDistCalculator`. Instance třídy implementující toto rozhraní je vyžadována pro každé spuštění výpočtu cesty. Konkrétní využití a vlastnosti třídy implementující `INodeDistCalculator` závisí především na vlastním algoritmu.

Protože naše platforma požaduje právě jeden `IPathFindAlg` v průběhu úrovně, může tento algoritmus využít přetypování na svůj konkrétní typ implementující `INodeDistCalculator` a využít jeho plného rozhraní.

Jedinou požadovanou metodou v rozhraní `INodeDistCalculator` je metoda `GetTime`, která je využívána pro zjištění existence hrany mezi dvěma vrcholy a v případě existence pak hodnotu této hrany, v rozhraní označovanou jako `time`, tedy čas potřebný pro přesun z prvního vrcholu na druhý. Tato metoda je využívána ve třetí části, popsáné dále.

### 3.7.3 Reprezentace cesty

Třetí částí reprezentace grafu je reprezentace cest. Tuto reprezentaci tvoří dvě třídy, `Waypoint`, tedy bod v cestě, a z nich složená `Path`, tedy cesta. Při použití v platformě jsou instance `Waypoint` chápány jako body, mezi kterými se jednotka pohybuje konstantní rychlostí po jejich spojnici. Cesta je potom posloupností těchto úseček. Každý z `Waypoint` bodů by měl odpovídat jedné instanci `INode`, tedy vrcholu z reprezentace grafu. V platformě je tato reprezentace použita pro implementaci komponent `WorldWalker`, poskytující pohyb jednotek po mapě, a `MovingRangeTarget`, umožňující střelbu na pohyblivý cíl. V případě, že tvůrce balíčku nepoužívá tyto dvě komponenty, může se jeho interpretace cesty vracené jeho algoritmem lišit.

### 3.7.4 Výběr algoritmu

Používaná instance algoritmu je přístupná pomocí property `Pathfinding` na instanci mapy aktuální úrovně. Získávání instance třídy implementující `IPathFindAlg` při načítání úrovně je implementováno podle návrhového vzoru `Abstract Factory`. Instance `IPathFindAlgFactory` je získána od pluginu logiky aktuální úrovně. Tato *factory* je následně předána mapě, která jako poslední krok své inicializace vytvoří instanci algoritmu. Tento způsob jsme vybrali pro umožnění přístupu k načtené mapě v konstruktoru algoritmu.

### 3.7.5 Implementace v platformě

Platforma poskytuje základní použitelnou implementaci rozhraní `IPathFindAlg` pomocí algoritmu A\*, popsaného v části 2.4.2.

Graf vytvářený tímto algoritmem obsahuje všechny možné hrany mezi sousedními dlaždicemi. Dále při připojení vrcholů budov předpokládá vytvoření všech hran, které by mohla kterákoli z jednotek použít. Následně jsou tyto hrany filtrovány a ohodnocovány v průběhu výpočtu nejkratší cesty za použití potomka třídy `NodeDistCalculator`, definovaného v balíčku.

Tato třída dále vyžaduje implementaci metody pro výpočet heuristiky. Hodnoty heuristiky závisí pouze na tvůrci balíčku, není tedy nijak kontrolována podmínka přípustnosti a monotonie.

## 3.8 Kamera

Kamera je v enginu UrhoSharp reprezentována komponentou `Camera`. V naší platformě je pro tuto komponentu vytvořena separátní `Node`, která je následně přesouvána po herním světě. Tento pohyb kamery implementuje platforma pomocí vlastní komponenty `CameraMover`, která je přidána na stejnou `Node` jako `Camera`.

Kamerou lze pohybovat třemi způsoby. Tyto způsoby jsou implementovány pomocí návrhového vzoru `State` a jsou jimi:

- 1) typická RTS kamera (state `FixedCamera`),
- 2) kamera sledující jednotku (state `EntityFollowingCamera`),
- 3) volně létající kamera (state `FreeFloatCamera`).

V této části popisujeme pouze API poskytované zbytku platformy a tvůrcům balíčku pro pohyb kamery. Vlastní řízení pohybu kamery hráčem je popsáno v části ??.

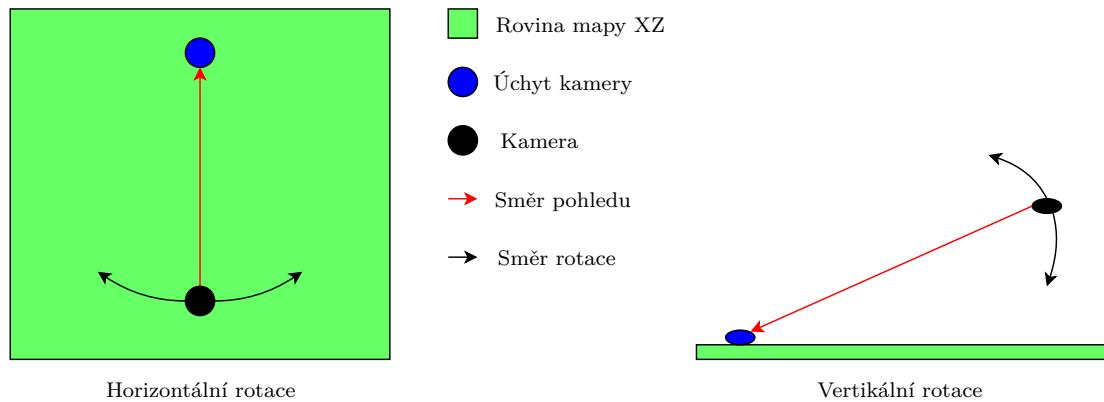
### 3.8.1 Typická RTS kamera

Pohyb typické RTS kamery je implementován třídou `FixedCamera`. Typická RTS kamera je namířena na pevný bod v herním světě, od kterého je v konstantní vzdálenosti. Toto je implementováno pomocí hierarchie instancí `Node`, do které mezi vlastní `Node` obsahující kameru a `Node` reprezentující celou úroveň přidáváme třetí `Node`, kterou nazýváme `CameraHolder`, tedy úchyt kamery.

Kamera je umístěna v základní pozici vůči úchytu a pro pohyb kamery po herním světě není pohybováno přímo kamerou, ale pouze úchytom.

V naší implementaci úchyt při pohybu kopíruje terén mapy, což zjednodušuje hráči pohyb kamery po mapě, kdy se nemusí starat o oddalování a přibližování kamery se změnami výšky terénu.

Při rotaci je naopak měněna pozice kamery vůči úchytu, tedy je měněn atribut `Position` na instanci `Node` obsahující kameru. Rotace jsou ilustrovány na obrázku 3.19. Při horizontálním otáčení se kamera otáčí okolo osy Y v rodičovském prostoru. Protože rodičem kamery je úchyt, otáčí se kamera okolo vertikální osy procházející úchytom. Při vertikálním otáčení je kamera také otáčena okolo úchytu, ale tentokrát okolo osy mířící vpravo z pohledu kamery. Po každém otáčení je směr pohledu upraven tak, aby znova mířil na úchyt.



Obrázek 3.19: Ukázka rotací s pohledem ze směru osy rotace.

Dále lze kameru přiblížovat a oddalovat od úchytu ve směru pohledu. Tato akce je pouhou změnou velikosti atributu `Position` kamery. Tedy pro přiblížení vynásobíme velikost tohoto vektoru číslem menším než jedna, pro oddálení větším než jedna. Kamera má limit na nejbližší možné přiblížení pro zamezení pohledu skrz terén.

Pohyb `CameraHolderu` je omezen na herní plochu, čímž omezujeme možnost pohledu kamery mimo herní plochu. Dále je kontrolována vlastní pozice kamery, které není dovoleno dostat se pod úroveň terénu.

### 3.8.2 Kamera sledující jednotku

Sledování jednotky kamery je z pohledu platformy pouhé sledování jiné `Node` než `CameraHolder`. Proto také jsou tyto dva stavy kamery potomkem jedné třídy, a to `PointFollowingCamera`.

Jedinou změnou oproti sledování `CameraHolderu` je odstínění kamery od otáčení jednotky. Oproti `CameraHolderu`, u kterého zachováváme po celou dobu neměníme rotaci a pouze s ním pohybujeme po herním světě, mohou jednotky měnit jak svoji pozici, tak rotaci. Dále může mít jednotka nastaven jiný `Scale` na své `Node` než je `Scale` u úchytu kamery.

Pro řešení nechtěných rotací kamery s jednotkou si ukládáme separátně chtěný směr pohledu. Následně při každém výpočtu stavu hry otočíme směr pohledu kamery tak, aby mířil tímto směrem. Toto řešení nám umožní ignorovat rotace jednotky, kterou sledujeme, a udržovat konstantní směr pohledu.

Problém s **Scale** nastává při změně přepnutí sledování mezi dvěma různými jednotkami či jednotkou a **CameraHolderem**, a dále při přibližování a oddalování kamery. Řešením změny mezi dvěma různými sledovanými **Node** je vynásobení poměrem jejich **Scale**. Při přibližování či oddalování stačí potom vydělit chtěnou změnu pozice **Scale** sledované **Node**, čímž následně změníme pozici kamery v herním světě nezávisle na **Scale** sledované **Node**.

Při sledování jednotky neumožňujeme hráči ovládat pohyb kamery po herním světě. Při pokusu o pohyb dochází k přepnutí zpět na sledování **CameraHolderu**.

### 3.8.3 Volně létající kamera

V tomto módu se stává **Node** obsahující kameru přímým potomkem **LevelNode** a je jí umožněn nezávislý pohyb po celé úrovni. Otáčení je v tomto módu prováděno okolo vlastní pozice kamery, jak v horizontálním, tak ve vertikálním směru.

Stejně jako při předchozích způsobech pohyb je i zde kamera omezena na herní plochu mapy. Není tedy možné mapu opustit, ani se dostat pod úroveň terénu.

## 3.9 Vstup

Vstup je implementován separátně pro ovládání menu a hry. Každá z těchto částí je dále rozdělena na definici přenositelného rozhraní a implementace pro dotykový display či myš a klávesnici. Toto rozdělení můžeme vidět na diagramu 3.20. Jak bylo psáno v části 2.2, cílem naší práce je implementace pro systém Windows a ovládání pomocí myši a klávesnice. Proto implementace tříd dotykového rozhraní je pouhá kostra pro budoucí implementaci.

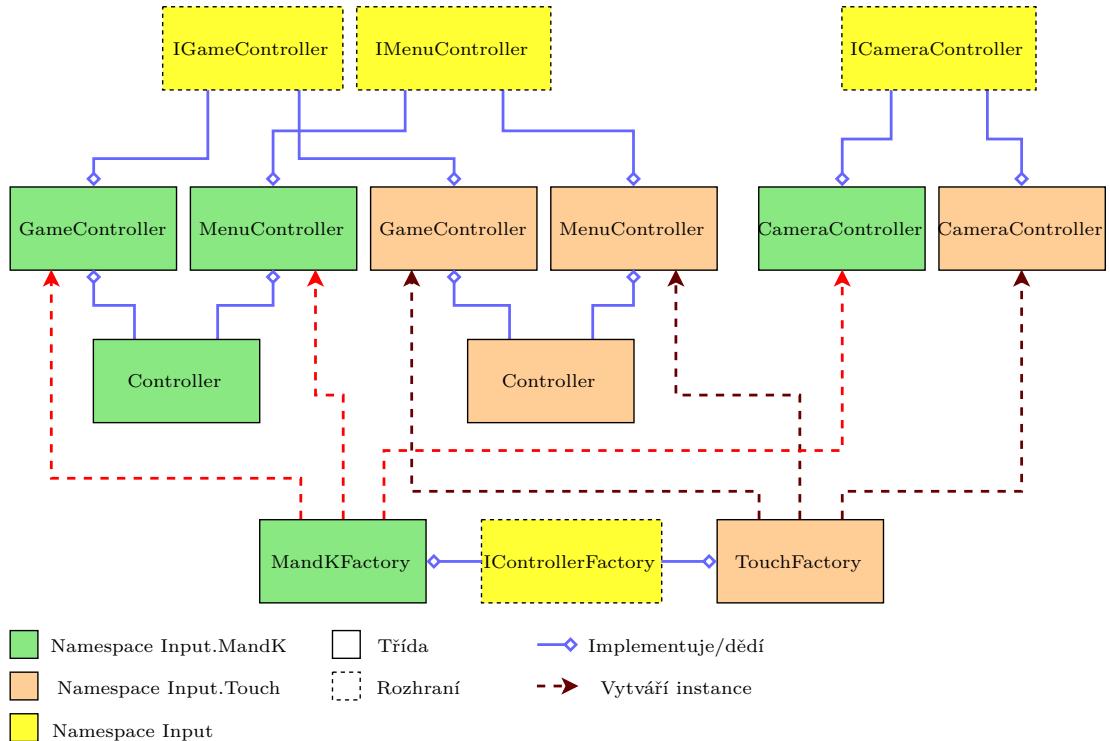
Vzhledem k úzkému provázání uživatelského rozhraní se vstupem, především v menu, jsou třídy ovládající uživatelské rozhraní vytvářeny, spravovány a uvolňovány třídami kontrolujícími vstup. Tuto hierarchii můžeme vidět na diagramu 3.20. Spolu s vlastnictvím uživatelského rozhraní slouží třída vstupu také jako odstínění zbytku platformy od implementace grafického rozhraní poskytnutím metod ovládajících uživatelské rozhraní.

V menu je vstup ovládán třídou **MenuController**. Vzhledem k tomu, že veškerý vstup uživatele je v menu získáván pomocí grafického rozhraní, slouží třída **MenuController** především jako fasáda z návrhového vzoru **Facade**, překrývající složitost grafického rozhraní z pohledu zbytku platformy.

Při hře je uživatelský vstup zprostředkováván třídou **GameController**. Účelem této třídy je, stejně jako u **MenuController**, sloužit jako fasáda, tentokrát jak nad grafický uživatelským rozhraním, tak nad systémem herního enginu pro zpracovávání uživatelského vstupu.

Vstup ve hře je přiřazen právě jednomu hráči, kterého je možné zjistit pomocí **Player** property třídy **GameController**. Ve zbytku platformy lze potom při zpracování vstupu změnit svoje chování podle aktuálního hráče, kterému patří vstup. Příkladem takového změny chování může být přiřazení aktuálního hráče se vstupem jako vlastníka právě postavené budovy či jednotky. Další změny chování jsou popsány v části ??.

Pro ovládání kamery je vytvořena separátní třída, využívající třídy **GameController** a grafického rozhraní pro přijímání vstupu od uživatele a v části 3.8 popsanou



Obrázek 3.20: Hierarchie tříd tvořících systém vstupu.

třídu `CameraMover` pro vykonávání akcí podle přijatého vstupu. Třída implementuje mapování stisků klávesnice a pohybů myši na pohyb kamery.

Pro zjednodušení přepínání mezi schématy ovládání implementuje každá ze tříd přenositelné rozhraní, což můžeme vidět na diagram 3.20. Implementace těchto rozhraní nám umožňuje vytváření instancí tříd implementovat pomocí návrhového vzoru `Factory`. Díky tomu existuje jediné místo, a to v metodě `Start` třídy `MHUrhoApp`, kde se platforma při inicializaci explicitně rozhoduje, jaké schéma ovládání použít. Po vytvoření `Factory` jsou pak využívány pouze metody přenositelného rozhraní, případně ve třídách specifických pro dané ovládací schéma dochází k přetypování zpět na konkrétní typy specifické pro dané schéma.

### 3.10 Editace mapy a ovládání hry

Editování a ovládání úrovně je v platformě spojeno do systému, který nazýváme „Tools“. Tools, neboli nástroje, umožňují jak platformě, tak tvůrci pluginu, definovat třídy, které přijímají vstup od uživatele a převádí jej do modifikací mapy, vytváření jednotek či budov, ovládání jednotek či jinou manipulaci s herním světem. Protože implementace jednotlivých nástrojů je závislá na použití schématu ovládání, má systém nástrojů podobnou strukturu jako systém vstupu, ukázanou na diagramu 3.20. Struktura nástrojů také obsahuje přenositelnou část, zde definovanou ve jmenném prostoru `MHUrho.EditorTools.Base`, a následně odvozené implementace pro jednotlivá schémata ovládání ve jmenných prostorech `MHUrho.EditorTools.MandK` pro myš a klávesnici a `MHUrho.EditorTools.Touch` pro dotykový display. Stejně jako u vstupu je i zde aktuální implementace pro

dotykový display pouhou kostrou pro budoucí implementaci.

Platforma poskytuje několik základních nástrojů, především pro editaci terénu. Těmito nástroji jsou:

- `TileTypeTool`, umožňující změnu typů dlaždic;
- `TerrainManipulatorTool`, poskytující několik způsobů změny reliéfu.

Dále pak platforma poskytuje nástroje pro vytváření jednotek a budov v podobě těchto tříd:

- `BuildingBuilderTool` pro stavbu budov;
- `UnitSpawningTool` pro tvorbu jednotek.

Tyto nástroje umožňují stavbu všech budov a tvorbu všech jednotek přítomných v balíčku. Předpokládáme ovšem, že při vlastní hře, případně i při editaci, bude hráči omezena množina dostupných jednotek a budov, případně bude při pokusu o vytvoření kontrolováno a měněno množství surovin vlastněné hráčem. Z tohoto důvodu předpokládáme, že tvůrci balíčků tyto nástroje nahradí svou vlastní implementací.

Posledním nástrojem je `UnitSelectorTool`, umožňující označení skupiny jednotek a vydávání rozkazů této skupině. Tento nástroj implementuje jednoduché schéma ovládání, které je dostačující pro jednodušší hry, ale stejně jako u předchozích nástrojů předpokládáme, že tvůrci balíčků tento nástroj vymění za svoji vlastní implementaci.

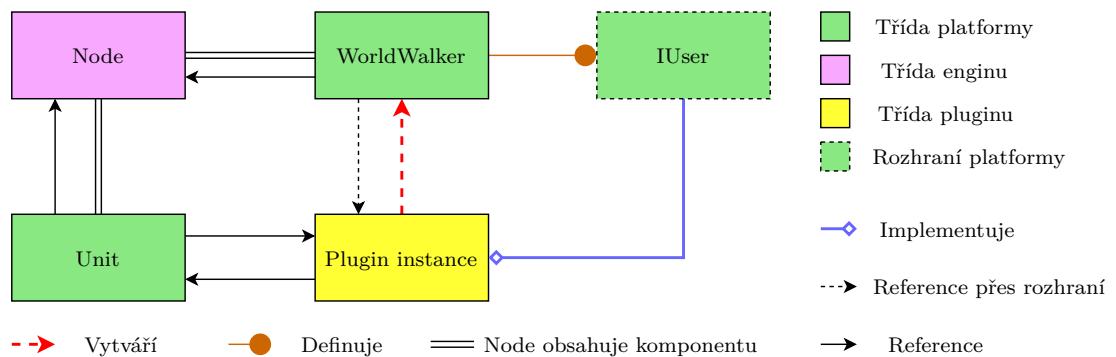
Pro výběr poskytovaných nástrojů používá platforma podobný systém jako u výběru algoritmu pro hledání nejkratší cesty. Po pluginu logiky úrovně je požadována definice metody `GetToolManager`, vracející tvůrcem pluginu definovaného potomka třídy `ToolManager`. Z této třídy je poté získán seznam potomků třídy `Tool`, tedy všech nástrojů, které budou dostupné hráči v aktuální úrovni.

### 3.11 Základní komponenty

`DefaultComponents`, neboli základní komponenty, jsou komponenty poskytované platformou, implementující funkcionalitu společnou velké části podporovaných typů her. Tato funkcionalita zahrnuje:

- pohyb po terénu (`WorldWalker`),
- označování jednotek a vydávání rozkazů (`UnitSelector`),
- střelba na statický a pohyblivý cíl (`Shooter`, `StaticRangeTarget`, `MovingRangeTarget`),
- útok na blízko (`StaticMeeleAttacker`, `MovingMeeleAttacker`),
- reakce na kliknutí (`Clicker`),
- simulace balistického projektlu (`BallisticProjectile`).

Platforma sama základní komponenty jednotkám nepřiděluje. Schéma vztahů `DefaultComponent` k ostatním částem programu můžeme vidět na diagramu 3.21. Pro použití těchto komponent musí tvůrce pluginu při vytvoření instance entity, tedy jednotky, budovy či projektilu, vytvořit požadovanou základní komponentu a připojit ji k této instanci entity. Implementace základních komponent této skutečnosti využívá, a v metodách vytvářejících instance základních komponent požaduje instanční plugin, který navíc musí implementovat rozhraní `IUser` specifikované danou komponentou. Tímto způsobem je umožněna implementace základních komponent, která není závislá na konkrétních implementacích ostatních částí platformy, nijak neomezuje chování entit používajících tuto komponentu a umožňuje tvůrci balíčku modifikovat chování této komponenty implementací požadovaných metod.



Obrázek 3.21: Vztah `DefaultComponent` k ostatním částem platformy.

Příkladem může být třída `WorldWalker`, která po uživateli požaduje metodu vracející `INodeDistCalculator`, který je následně použit pro výpočet nejkratší cesty. Tímto způsobem je implementace `WorldWalker` oproštěna od závislosti na použitém algoritmu pro hledání nejkratší cesty. Druhým příkladem může být `MovingMeeleAttacker`, který po uživateli požaduje tři metody, a to `IsInRange`, `PickTarget`, `MoveTo`. Metody `IsInRange` a `PickTarget` slouží pro specifikaci chování komponenty tvůrcem balíčku, tedy umožňují mu vybrat cíl a rozhodovat, zda je cíl v dosahu, podle vlastních kritérií. Metoda `MoveTo` naopak slouží pro izolaci komponenty od implementace pohybu jednotky. Jednotka tedy není omezena na pohyb pomocí komponenty `WorldWalker`, ale umožňuje tvůrci vytvořit vlastní implementaci pohybu jednotky po mapě.

Další události, na které by mohl tvůrce pluginu chtít reagovat, jsou poskytovány jako **eventy**, jejichž obsluhu si může tvůrce pluginu zaregistrovat. Příkladem těchto událostí může být začátek pohybu, dokončení pohybu či zrušení pohybu u komponenty `WorldWalker`.

Důležitou vlastností těchto komponent je jejich schopnost automatického ukládání a načítání spolu s entitou, ke které jsou přidány. Z pohledu tvůrce pluginu tedy stačí přidat tyto komponenty k entitě při jejím vytvoření. Jediným omezením jsou **eventy**, které si musí tvůrce pluginu zaregistrovat po načtení úrovně znova, protože nelze jednoduše implementovat jejich automatické ukládání a načítání bez nutnosti jejich implementace každým uživatelem komponenty.

## 3.12 Ukládání a načítání úrovní

Načítání úrovní lze iniciovat třemi způsoby:

- 1) vytvoření nové úrovně,
- 2) načtení úrovně existující v balíčku,
- 3) načtení uložené hrané úrovně.

Prvním je vytvoření nové úrovně, která je načtena do základní podoby a je umožněna její editace.

Druhým je vybráním již existující úrovně z balíčku. Tyto úrovně jsou ve stavu, do kterého byly uvedeny v editoru úrovní a ještě nebyly spuštěny pro hru. Takovéto úrovně označujeme jako „Prototype“ úrovně.

Třetí možností je vybrání hry, která byla uložena již v průběhu hraní. Tyto již nelze nahrát pro editaci, což umožňuje zjednodušení logiky pluginů.

Pro reprezentaci těchto různých stavů je podle návrhového vzoru *State* implementována třída `LevelRep`. Všechny možné stavy a přechody mezi nimi můžeme vidět na diagramu 3.22. K

aždá úroveň začíná ve stavu `NewlyCreated`. Z toho stavu je vygenerována podle uživatelem zadaných parametrů do editoru. Parametry vytvářené úrovně, jako velikost, ikona a plugin, jsou vybrány v obrazovce grafického rozhraní `LevelCreationScreen`. Dalším parametrem je počáteční typ dlaždice, specifikovaný v balíčku. Editace úrovně je blíže popsána v části ??.

Z editoru lze úroveň uložit jako `Prototyp`, čímž je zapsána do balíčku a je následně možné ji z grafického rozhraní spouštět v herním módu. Dále lze úroveň uložit pod novým jménem, implementující standardní akci `SaveAs`, čímž přechází do `ClonedEditing` stavu.

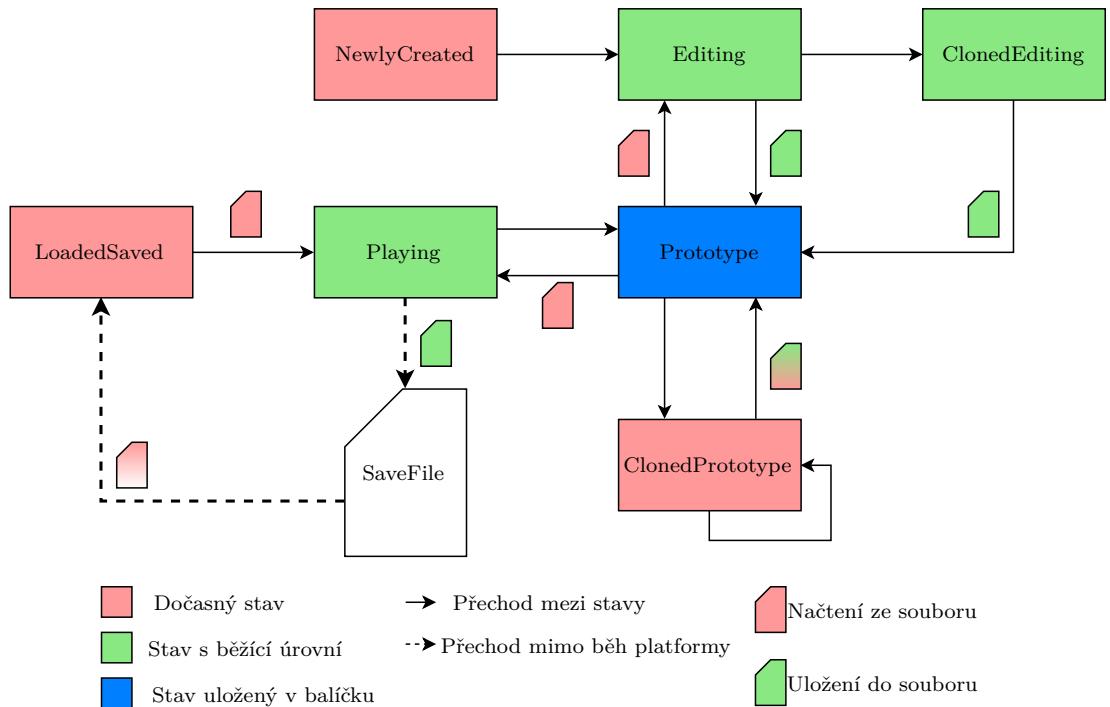
Ze stavu `Prototype` lze úroveň nahrát pro editace, čímž lze upravit aktuální prototyp nebo pomocí `SaveAs` vytvořit odvozený. Dále lze prototyp nahrát pro hru. Při tomto nahrání hráč specifikuje počáteční parametry hry, definované pluginem logiky úrovně, a pluginy, které mají být přiřazeny jednotlivým hráčům. Následně je úroveň nahrána a přechází do stavu `Playing`. Poslední možnou akcí je duplikace prototypu, umožňující nahrání prototypu pod novým jménem a vytvoření nového prototypu. Tato funkcionality z jisté míry duplikuje `SaveAs`, ale přišlo nám vhodné umožnit hráči duplikaci úrovně již před načtením pro zamezení nechtěné změny prototypu.

Při hraní lze aktuální stav úrovně uložit do tzv. Save file, tedy vytvořit úložku. Tato úložka je uložena do adresáře specifikovaného platformou a může následně být nahrána pro pokračování z uloženého stavu hry. Stav hrané úrovně nelze uložit zpět do prototypu, při skončení hry je tedy aktuální stav ztracen a úroveň přechází zpět do stavu prototyp se stavem úrovně před spuštěním hry.

Uložené hry lze nahrát do stavu `LoadedSaved`, kdy je úložky načtena pouze potřebná část informace. Následně může být hra nahrána do plného stavu `Playing`, kde pokračuje z uloženého stavu.

### 3.12.1 Stav úrovně

Pro ukládání a načítání aktuálního stavu úrovně používá platforma serializaci pomocí *protocol buffers*, jak bylo popsáno v části 2.3.4.



Obrázek 3.22: Stavy LevelRep a přechody mezi nimi.

Systém *protocol buffers* obsahuje interface definition language (IDL), tedy jazyk pro popis ukládaných struktur nezávislý na cílovém programovacím jazyce, a kompilátor souborů v IDL do cílových programovacích jazyků.

Platforma definuje čtyři `.proto` soubory, obsahující definici struktur a kompliovaných do `.cs` souborů, které jsou používány v platformě. Těmito čtyřmi `.proto` soubory jsou:

- 1) `UrhoTypes.proto`, definující serializaci typů enginu UrhoSharp;
- 2) `MHUrhoTypes.proto`, definující serializaci potomků `DefaultComponent` a `Path`;
- 3) `PluginStorage.proto`, definující strukturu ukládání poskytovanou pluginům;
- 4) `GameState.proto`, definující serializaci celkového stavu úrovně.

Uvnitř těchto souborů je použit systém importování, kdy lze obsah jednoho `.proto` souboru importovat a použít v jiném souboru.

Ze struktur definovaných v těchto souborech potom `protoc` kompilátor vygeneruje zdrojové soubory v jazyce C#, které obsahují definici tříd odpovídajících těmto strukturám a umožňujícím serializaci.

### 3.12.2 Plugins

Pro uložení stavu pluginů jsme vytvořili datovou strukturu umožňující ukládat libovolnou posloupnost podporovaných typů. Podporovanými typy jsou jak základní typy jazyka C#, tak typy enginu UrhoSharp a pole těchto typů.

Ukládaná data lze ukládat a načítat podle pořadí, indexovat čísla či pojmenovávat textovými jmény. Těmto třem způsobům odpovídají tyto třídy:

- 1) `SequentialPluginData`
- 2) `IndexedPluginData`
- 3) `NamedPluginData`

Uložení pomocí `SequentialPluginData` produkuje nejmenší soubory, protože nemusí ukládat mimo vlastní data a identifikaci typu dat žádná data navíc. `IndexedPluginData` musí navíc ukládat číselný index, `NamedPluginData` potom celý text jména.

Při ukládání je spolu s vlastní hodnotou dat ukládán také jejich typ. Toto je implementováno pomocí systému `oneof` protocol buffers, který umožňuje uložit jeden z vyjmenovaných typů a poté při načtení obsahuje identifikaci, který z typů v něm byl uložen. Tento uložený typ je pak při načítání dat používán pro kontrolu, zda se tvůrce pluginu opravdu snaží číst typ, který byl uložen. Pro specifikaci čteného typu je vytvořeno generické rozhraní, které umožňuje číst libovolný typ a provádí kontrolu vůči uloženému typu. Tím je zajištěna typová bezpečnost tohoto systému za cenu zvětšení ukládaných dat.

# 4. Tvorba balíčku

Každý balíček je tvořen jedním hlavním XML souborem, reprezentujícím vlastní balíček, spolu s dalšími soubory, představujícími assety hry jako 3D modely, textury a definice uživatelského rozhraní. Tato část dokumentace se bude zaobírat pouze tvorbou hlavního XML souboru. Tvorba 3D modelů, textur a definice uživatelského rozhraní je mimo cíle této práce. Pro získání 3D modelů a textur použitelných v enginu UrhoSharp a naší platformě je možné postupovat podle tohoto tutoriálu [? ]. Pro tvorbu definic uživatelského rozhraní a prefabrikátů jednotek, budov a projektilů je dále možné použít editor distribuovaný spolu s enginem UrhoSharp [? ].

Při tvorbě balíčku je prvním krokem umístění všech assetů, tedy 3D modelů, textur, definic uživatelského rozhraní, assembly pluginů a dalších do adresářového podstromu libovolného adresáře. Konkrétní struktura závisí pouze na tvůrci balíčku, protože platforma pouze načítá relativní cesty z XML souboru definujícího balíček, a používá adresář, ve kterém je tento XML soubor umístěn, jako kořen těchto relativních cest.

V následujících částech popíšeme tvorbu XML souboru a význam jednotlivých elementů a atributů tohoto souboru.

## 4.1 Struktura XML

XML soubor představující balíček je popsán schématem `Data/Schemas/GamePack.xsd`. Spolu s tímto schématem obsahuje distribuce platformy šablonu XML souboru balíčku, umístěnou v adresáři instalace na cestě `Data/Templates/PackageDefinition.xml`. Příklad XML souboru funkčního balíčku můžete vidět v instalaci platformy na cestě `%AppData%\\MHUrho\\Packages\\ShowcasePackage\\PackageDef.xml`, na které se nachází ukázkový balíček.

Kořenovým elementem XML souboru balíčku je element `gamePack`. Tento element musí specifikovat jmenný prostor `xmlns="http://www.MobileHold.cz/GamePack.xsd"` pro možnost validace balíčku a jeho nahrání do platformy. Dále zde definujeme jméno balíčku, které bude uživateli při výběru balíčků.

```
<gamePack xmlns="http://www.MobileHold.cz/GamePack.xsd"
           name="[Package name]">
```

Prvním elementem uvnitř elementu `gamePack` je element `description`. Textový obsah tohoto elementu se zobrazí uživateli na obrazovce výběru balíčků při označení našeho balíčku.

```
<description>[Description text]</description>
```

Dalším elementem je element `levels`, jehož jediný potomek `dataDirPath` definuje cestu k adresáři, ve kterém budou ukládány datové soubory uložených prototypů úrovní. Všechny cesty uvedené používané kdekoli v balíčku jsou brány jako relativní vůči adresáři, ve kterém je umístěn XML soubor definující balíček.

```
<levels>
```

```

<dataDirPath>[Path to directory where level data will be
stored]</dataDirPath>
</levels>

```

Následuje posloupnost elementů umožňujících definice typů herních prvků. Těmito elementy jsou:

- 1) `levelLogicTypes` (typy logik úrovní),
- 2) `playerAITypes` (typy logik hráčů),
- 3) `resourceTypes` (typy surovin),
- 4) `tileTypes` (typy dlaždic),
- 5) `unitTypes` (typy jednotek),
- 6) `projectileTypes` (typy projektilů),
- 7) `buildingTypes` (typy budov).

Obsah těchto prvků bude popsán v následujících částech.

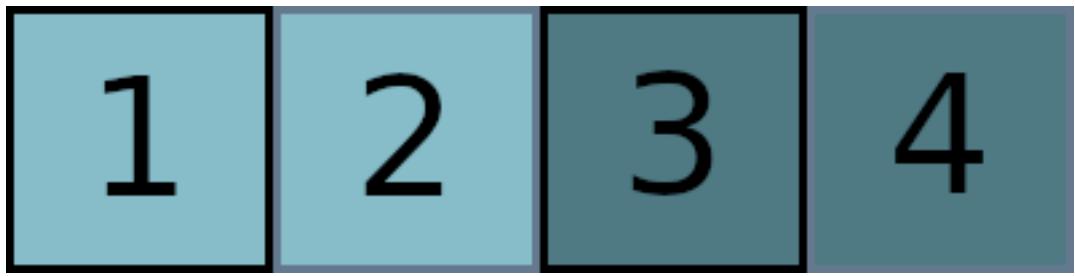
Jako poslední obsahuje element `gamePack` šestici elementů obsahujících cesty k texturám ikon. Těmito elementy jsou:

- 1) `resourceIconTexturePath` (textura ikon surovin),
- 2) `tileIconTexturePath` (textura ikon dlaždic),
- 3) `unitIconTexturePath` (textura ikon jednotek),
- 4) `buildingIconTexturePath` (textura ikon budov),
- 5) `playerIconTexturePath` (textura ikon hráčů),
- 6) `toolIconTexturePath` (textura ikon nástrojů),

Tyto cesty jsou, stejně jako všechny cesty v celém balíčku, relativní vůči adresáři, ve kterém je umístěn XML soubor definující balíček. Uvedené cesty mohou odkazovat na různé soubory, jeden soubor nebo libovolnou množinu souborů textur. Při použití editačních nástrojů platformy je předpokládán podoba textury ikon zobrazená na obrázku 4.1. Textura se skládá ze čtyř čtverců o stejné velikosti, které jsou používány pro zobrazení v grafickém prvku `Checkbox`, tedy zaškrťávací box. Čtverce jsou zobrazeny v následující situaci:

- 1) prvek není stlačen, myš není nad prvkem,
- 2) prvek není stlačen, myš je nad prvkem,
- 3) prvek je stlačen, myš není nad prvkem,
- 4) prvek je stlačen, myš je nad prvkem.

V obrázku 4.1 můžeme vidět čtverce označené odpovídajícími čísly.



Obrázek 4.1: Požadovaná podoba textury ikon.

## 4.2 Přidání jednotek

Přidání typu jednotky do balíčku znamená přidání elementu `unitType` jako potomka elementu `unitTypes`, uvedeného v předchozí části. Element `unitType` má tuto základní strukturu:

```
<unitType name="[Name]" ID="[Number]">
    <assets type="[xmlprefab|binaryprefab|items]">[content
        dependent on type]</assets>
    <assemblyPath>[Path to assembly of the
        plugin]</assemblyPath>
    <extension>[User defined part of XML]</extension>
    <iconTextureRectangle left="[Number]" top="[Number]"
        right="[Number]" bottom="[Number]" />
</unitType>
```

Atributy `name`, tedy jméno, a `ID`, tedy identifikátor, identifikují daný typ jednotky. Jméno i identifikátor musí být unikátní mezi typy jednotek, ale můžou se shodovat se jménem a identifikátorem nějakého typu jiného druhu herního prvku. Jméno může být libovolný neprázdný textový řetězec, ID může být libovolné celé nenulové číslo.

Element `assets` definuje assety, například model, texturu či tvar pro výpočet kolizí, nahrávané při vytváření instance tohoto typu jednotky. Tyto assety mohou být definovány třemi způsoby, rozlišenými hodnotou atributu `type`:

- 1) `xmlprefab`,
- 2) `binaryprefab`,
- 3) `items`

XML a binary prefab jsou soubory vytvářené editorem Urho3D. Obsahem těchto elementů je jediný potomek `path`, jehož obsahem je cesta k souboru obsahujícímu prefabrikát. Tato cesta je, jako všechny v balíčku, relativní vůči adresáři obsahujícímu soubor XML definující balíček.

```
<path>[Path to prefab file]</path>
```

Pro `Items` obsahuje element `assets` několik potomků, kupříkladu následující:

```

<scale x="0.2" y="0.2" z="0.8"/>
<model type="static">
    <modelPath>Assets/Box.mdl</modelPath>
    <material>
        <simpleMaterialPath>Assets/StoneMaterial.xml</simpleMaterialPath>
    </material>
</model>

```

Element **scale**, neboli škálování, určuje nastavení hodnoty Scale vytvořené instance **Node** reprezentující jednotku. Tato hodnota škáluje pozice a rozměry potomků a komponent této **Node**, tedy například rozměr 3D modelu.

Element **model** určuje 3D model použitý pro zobrazení této jednotky. Tento model je přidán jako komponenta na vytvořenou **Node**. Střed modelu je tedy vždy umístěn na pozici **Node**. Atribut **type** určuje, že je model statický (hodnota *static*), či animovaný (hodnota *animated*). Statický model je úspornější z pohledu výkonu, ale jak už název napovídá, nelze na něj použít animace. Oproti tomu animovaný model použití animací podporuje. Animace jsou ovládány v pluginech, proto popis využití animací naleznete v části ??.

Součástí definice modelu je definice jeho textur. Textury jsou v enginu UrhoSharp reprezentovány materiálem, který definuje vlastní texturu, použitý **shader** a jeho parametry. Systém materiálů je popsán v dokumentaci enginu UrhoSharp a Urho3D, nebudeme proto v této práci tento systém rozebírat. Model může být složen z více geometrií, kde každé z nich je přiřazen separátní materiál. Přiřazení materiálů lze provést dvěma způsoby:

- 1) souborem **MaterialList.txt**,
- 2) explicitním vyjmenováním materiálů.

Geometrie modelu jsou indexovány od nuly, čehož využívá soubor **MaterialList.txt**. Tento soubor, generovaný editory 3D modelů, lze využít pro připojení modelů...

Následuje element **assemblyPath**, který obsahuje cestu k assembly obsahující plugin tohoto typu jednotek. Cesta je, jako všechny cesty v balíčku, relativní vůči adresáři obsahující XML soubor. Tento plugin je následně využit pro tvorbu instančních pluginů a definici chování jednotek ve hře. Blíže je význam a funkce pluginů popsán v části ???. Plugin typu je identifikován podle hodnoty **name** a **ID**, které se musí shodovat s hodnotami uvedenými zde v XML.

Element **extension** slouží tvůrci pluginů pro uložení vlastních dat. Obsah tohoto elementu není platformou nijak omezen či validován, a je předáván pluginu typu při jeho načítání do platformy. V ukázkovém balíčku se tento element používá například pro specifikaci ceny jednotky, vymezení druhů dlaždic, kterými může jednotka procházet či specifikaci používaného projektu.

Poslední element, **iconTextureRectangle**, vymezuje část textury, které je specifikována elementem **unitIconTexture**, popsáným v předchozí části. Obdélník, vymezený hodnotou atributů tohoto elementu, definuje pozici textury první situace, tedy případu, kdy je **Checkbox** nezmáčknutý a myš není umístěna nad ním. Pozice dalších stavů jsou potom předpokládány postupně s posunem o šířku obdélníku vpravo, jak můžeme vidět na obrázku 4.1.

## 4.3 Přidání budov

Přidání typu budov do balíčku je z velké části identické přidání typu jednotky, popsaném v předchozí části 4.2. Z tohoto důvodu zde zmíníme identické části a blíže popíšeme pouze části rozdílné.

Přidání typu budov znamená přidání elementu `buildingType` jako potomka elementu `buildingTypes`. Základní struktura obsahu elementu `buildingType` je následující:

```
<buildingType name="[Name]" ID="[Number]">
    <assets type="[xmlprefab|binaryprefab|items]">[content
        dependent on type]</assets>
    <assemblyPath>[Path to assembly of the
        plugin]</assemblyPath>
    <extension>[User defined part of XML]</extension>
    <iconTextureRectangle left="[Number]" top="[Number]"
        right="[Number]" bottom="[Number]" />
    <size x="[Number of tiles]" y="[Number of tiles]" />
</buildingType>
```

Jak můžeme vidět, jediným rozdílem oproti elementu definujícímu typ jednotek je element `size`. Tento element definuje velikost obdélníku mapy zabíraného tímto typem budov. Rozměry obdélníku jsou uváděny v počtu dlaždic.

Stejně jako u jednotek i zde musí být hodnoty atributů `name` a `ID` unikátní mezi všemi typy budov. `iconTextureRectangle` oproti jednotkám určuje pozici v textuře ikon budov, tedy určené elementem `buildingIconTexture`.

## 4.4 Přidání projektilů

Přidání projektilů je, stejně jako přidání budov, také z velké části identické přidání jednotek, popsaném v části 4.2. Přidání typu projektilů je provedeno přidáním elementu `projectileType` jako potomka elementu `projectileTypes`. Struktura obsahu elementu `projectileType` je následující:

```
<projectileType name="[Name]" ID="[Number]">
    <assets type="[xmlprefab|binaryprefab|items]">[content
        dependent on type]</assets>
    <assemblyPath>[Path to assembly of the
        plugin]</assemblyPath>
    <extension>[User defined part of XML]</extension>
</projectileType>
```

Jak můžeme vidět, obsahuje `projectileType` elementy identické obsahu `unitType`, se stejným významem.

## 4.5 Přidání dlaždic

Typy dlaždic mají oproti jednotkám, budovám a projektilům jiný účel. Kde jednotky, budovy a projektily jsou přidávány jako nové herní prvky do herního

světa, slouží typy dlaždic pouze pro dodání vzhledu prvkům již existujícím v úrovni, tedy dlaždicím mapy. Dále jsou typy dlaždic využívány pro implementaci různého chování v různých částech mapy.

Typ dlaždice je reprezentován elementem `tileType`, který musí podle schématu XML být potomkem `tileTypes`. Obsah elementu `tileType` je následující:

```
<tileType name="[Name]" ID="[Number]">
    <iconTextureRectangle left="[Number]" top="[Number]"
        right="[Number]" bottom="[Number]" />
    <texturePath>[Path to texture file]</texturePath>
    <minimapColor R="[0-255]" G="[0-255]" B="[0-255]" />
</tileType>
```

Atributy `name` a `id` mají stejnou sémantiku jako v předchozích částech, musí tedy být unikátní mezi všemi druhy dlaždic. Stejně tak `iconTextureRectangle` má stejný význam jako v předcházejících částech s tím rozdílem, že je textura specifikována elementem `tileIconTexturePath`.

Element `minimapColor` určuje barvu dlaždice při zobrazení na minimapě. Jak vidíme z ukázky XML, barva je specifikována pomocí červené, zelené a modré složky s hodnotami od 0 do 255.

Specifikem typů dlaždic je nutnost definice základního typu dlaždic, který bude určen jako typ dlaždic nově vygenerované úrovně. Tento typ je specifikován elementem `defaultTileType`. Obsah tohoto elementu je identický s obsahem elementu `tileType`, jediným rozdílem je tedy jméno tohoto elementu, které platforma využívá pro jeho odlišení. Element `defaultTileType` musí být první definovaný typ dlaždice, a stejně jako ostatní `tileType` musí i jeho jméno a identifikátor být unikátní mezi všemi typy dlaždic.

## 4.6 Přidání surovin

Suroviny jsou v platformě používány pouze pro označení čísla, které určuje jejich množství. Z tohoto důvodu není potřeba u tohoto druhu herního prvku specifikovat velké množství informací.

Každý typ surovin je specifikován elementem `resourceType`, který je musí být potomkem elementu `resourceTypes`. Obsah elementu `resourceType` je následující:

```
<resourceType name="[Name]" ID="[Number]">
    <iconTextureRectangle left="[Number]" top="[Number]"
        right="[Number]" bottom="[Number]" />
</resourceType>
```

Stejně jako u všech předchozích i zde musí jméno a identifikátor být unikátní mezi typy surovin. Element `iconTextureRectangle` potom určuje část textury určené elementem `resourceIconTexturePath`, která reprezentuje ikonu suroviny. Tato ikona není v tuto chvíli platformou v žádné části využívána, nejsou na její vzhled tedy kladený žádná omezení.

## 4.7 Přidání AI hráče

Hlavní součástí umělé inteligence hráče je plugin, jehož vytváření popíšeme v následující části ???. Z pohledu balíčku je každý typ umělé inteligence hráče reprezentována elementem `playerAIType` v podstromu elementu `playerAITypes`. Element `playerAIType` má tuto strukturu:

```
<playerAIType name="[Name]" ID="[Number]"  
category="ai|human|neutral">  
    <iconTextureRectangle left="[Number]" top="[Number]"  
    right="[Number]" bottom="[Number]" />  
    <assemblyPath>[Path to assembly of the  
    plugin]</assemblyPath>  
    <extension>[User defined part of XML]</extension>  
</playerAIType>
```

Oproti předchozím součástem balíčku se umělé inteligence hráčů dělí do tří skupin:

- 1) umělé inteligence oponentů (ai),
- 2) umělá inteligence pomáhající hráčovy (human),
- 3) umělá inteligence neutrálního hráče (neutral).

Umělé inteligence oponentů řídí akce hráčů, kteří jsou součástí některého z týmů ale nejsou ovládání uživatelem. Umělá inteligence pomáhající hráčovy řídí stejného hráče, jakého řídí uživatel svým vstupem. Tato umělá inteligence může být použita pro implementaci herních událostí či automatizaci některých součástí hry. Neutrální hráč potom ovládá prvky mapy, které jsou součástí scenérie a neúčastní se přímo souboje hráčů. Těmito prvky mohou být zvířata pohybující se v úrovni, stromy či součásti terénu.

## 4.8 Přidání AI úrovně

Definice typu umělé inteligence úrovně je reprezentována eementem `levelLogicType`, který je potomkem `levelLogicTypes`. Obsah elementu `levelLogicType` je následující:

```
<levelLogicType name="[Name]" ID="[Number]">  
    <assemblyPath>[Path to assembly of the  
    plugin]</assemblyPath>  
    <extension>[User defined part of XML]</extension>  
</levelLogicType>
```

Jak můžeme vidět, obsahuje definice typu umělé inteligence úrovně elementy po- psané v předcházejících částech se stejnou sémantikou.

# 5. Tvorba pluginů

Pluginy rozumíme .NET assembly nahrávané platformou MHUrho a implementující chování úrovní, hráčů, jednotek, budov a projektilů. Pluginy jsou v balíčku uloženy jako .dll knihovny, cílené pro .NET Framework 4.7.2.

V této části dokumentace popíšeme postup vytvoření takového knihovny. Její následné připojení do balíčku je popsáno v předchozí části 4.

Platforma nijak neomezuje rozdělení pluginů pro různé herní prvky do různých knihoven. Lze tedy všechny pluginy shromáždit v jedné, jak je to provedeno v ukázkové hře, nebo je možné vytvořit více různých knihoven s libovolným rozdělením pluginů.

Stejně jako v předchozí části i zde je velká část tvorby shodná pro různé druhy prvků. Celý proces proto popíšeme pouze v části tvorby pluginu jednotky a v následujících částech uvedeme pouze rozdíly oproti této části.

## 5.1 Vytvoření pluginu jednotky

V této části popíšeme tvorbu typu jednotky. Tato jednotka bude mít tyto vlastnosti:

- 3D model s animacemi,
- pohyb po terénu přes omezenou množinu typů dlaždic,
- střelba na dálku za použití ypu projektilu specifikovaného v XML.

Jako první vytvoříme definici jednotky v XML. Tento krok je popsán v předchozí části 4, nebudeme ho zde proto opakovat. Výsledné XML typu jednotky tedy bude takovéto:

```
<unitType name="Chicken" ID="3">
<assets type="xmlprefab">
<path>Assets/Units/Chicken/prefab.xml</path>
</assets>
<assemblyPath>ShowcasePackage.dll</assemblyPath>
<extension>
<cost>
<Wood>0.2</Wood>
<Gold>0.5</Gold>
</cost>
<canPass>
<Sand/>
<Xamarin/>
<Grass/>
<Water/>
</canPass>
</extension>
<iconTextureRectangle left="0" top="200" right="100"
 bottom="300"/>
```

```
</unitType>
```

Hotovou jednotku můžete vidět v ukázkové hře pod názvem **Chicken**.

## 5.2 Plugin typu

Prvním krokem je vytvoření dvou veřejných tříd, jedné reprezentující plugin typu a druhé reprezentující plugin instance.

Jako první vytvoříme třídu reprezentující plugin. Tato třída musí být veřejná a dědit od třídy **UnitTypePlugin**, definované platformou. Takováto třída poté bude nalezena pomocí **Reflection** a za běhu platformy načtena jako plugin daného typu.

```
public class ChickenType : UnitTypePlugin {  
}
```

Následuje implementace požadovaných vlastností a metod třídy. Jako první přidáme vlastnosti **Name** a **ID**.

```
public override string Name => "Chicken";  
public override int ID => 3;
```

Tyto vlastnosti jsou používány pro nalezení pluginu pro daný typ. Hodnoty **Name** a **ID** se tedy musí shodovat s hodnotami uvedenými v atrributech **name** a **ID** v XML definici typu.

Dále platforma po všech typových pluginech požaduje implementaci tří hlavních metod:

- 1) **Initialize**,
- 2) **GetInstanceForLoading**,
- 3) **CreateNewInstance**.

První metodou společnou všem pluginům typů je metoda **Initialize**. Tato metoda je poskytnuta z důvodu použití **Reflection** pro vytváření instancí typových pluginů, což vynucuje použití konstruktoru bez parametrů. Metoda **Initialize** tedy do jisté míry nahrazuje konstruktor. Tato metoda je volána pouze jednou, při načtení balíčku do hry. Typická implementace načte data z **extension** elementu v XML definici typu. Tato data často budou odkazovat na jiné typy herních prvků, metoda **Initialize** tedy dostává referenci na balíček, kterou může použít pro získání referencí na ostatní typy z balíčku. Implementace pro naši jednotku bude vypadat takto:

```
protected override void Initialize(XElement extensionElement,  
    GamePack package) {  
    XElement costElem =  
        extensionElement.Element(package.PackageManager.GetQualifiedXName(CostEleme
```

```

cost = Cost.FromXml(costElem, package);

 XElement canPass =
extensionElement.Element(package.PackageManager.GetQualifiedXName(PassableTileTypes));
PassableTileTypes = ViableTileTypes.FromXml(canPass, package);

myType = package.GetUnitType(ID);
ProjectileType = package.GetProjectileType("EggProjectile");
}

```

`Cost` a `ViableTileTypes` jsou pomocné třídy, které zpracují části XML a podle načtených dat získají reference na typy. Ukázku manuálního získání typu můžeme vidět při inicializaci `ProjectileType`, kde používáme balíček pro získání typu projektilu.

Druhou metodou je metoda `CreateNewInstance`, která je volána při vytvoření nové instance jednotky tohoto typu. Jejím účelem je získání pluginu pro nově vytvářenou instanci jednotky. Tvorba pluginu instance bude popsána v následující části ??, zde pouze uvedeme že plugin instance bude představován třídou `Chicken`.

```

public override UnitInstancePlugin
CreateNewInstance(ILevelManager level, IUnit unit){
return Chicken.CreateNew(level, unit, this);
}

```

Metoda získává jako první argument instanci `ILevelManager`, který reprezentuje aktuální úroveň a slouží jako přístupový bod ke všem službám platformy. Druhým argumentem je potom `IUnit`, která je instancí reprezentující nově vytvářenou jednotku v platformě. Právě pro tuto jednotku vytváříme instanční plugin.

Poslední požadovanou metodou je metoda `GetInstanceForLoading`. Tato metoda má podobný účel jako předchozí metoda `CreateNewInstance`, a to získání instančního pluginu jednotky. Rozdíl je ale v tom, že tato jednotka není nově vytvářena v průběhu hry úrovně, ale právě dochází k načtení úrovně z úložky a tedy počáteční stav jednotky i pluginu bude načten z uloženého souboru. Z tohoto důvodu musí být instanční plugin inicializován takovým způsobem, aby mohl následně načíst uložený stav. Implementace této metody bude tedy vypadat takto:

```

public override UnitInstancePlugin
GetInstanceForLoading(ILevelManager level, IUnit unit) {
return Chicken.CreateForLoading(level, unit, this);
}

```

Typy jednotek mají jedinou metodu odlišnou od ostatních pluginů typů, a to:

```

public override bool CanSpawnAt(ITile tile);

```

Tato metoda je volána před vytvořením nové instance jednotky a jejím účelem je zjistit, zda lze jednotku vytvořit na dlaždici `tile`. Typická implementace ověří, zda se na dané dlaždici vyskytuje budova, zda je dlaždice správného typu a případně zda se na dané dlaždici vyskytují další jednotky. Naše implementace

využije pomocnou třídu `ViableTileTypes`, která slouží právě jako seznam správných typů dlaždic.

```
public override bool CanSpawnAt(ITile tile) {
    return PassableTileTypes.IsViable(tile) &&
    (tile.Building == null);
}
```

Tímto jsme vytvořili funkční plugin typu, který nám umožní vytvářet nové instance jednotek tohoto typu, kontrolovat místa v mapě, na kterých jsou vytvářeny a načítat uložené instance jednotky do hry.

### Plugin instance

Každá instance jednotky má přiřazenu jednu instanci instančního pluginu. Tato instance pluginu je získávána voláním jedné z funkcí `CreateNewInstance` a `GetInstanceForLoading`, popsaných v předchozí části 5.1. Metoda `CreateNewInstance` je volána při tvorbě nové instance jednotky v běžící úrovni, metoda `GetInstanceForLoading` je pak volána při načítání jednotky v průběhu načítání uložené úrovni.

Plugin instance je tvořen třídou, která je potomkem `UnitInstancePlugin`. Pro naši jednotku tedy vytvoříme takovouto třídu:

```
class Chicken : UnitInstancePlugin {  
}
```

Tuto definici budeme ještě v průběhu tvorby upravovat, ale pro začátek takováto definice stačí.

#### 5.2.1 Vytvoření instance

Pro implementaci metod `CreateNewInstance` a `GetInstanceForLoading` musíme poskytnout dva různé způsoby inicializace třídy. Toho lze docílit několika způsoby, my jsme si zvolili vytvoření dvou statických metod, které provedou inicializaci třídy.

Pro vytvoření nové nové instance v běžící úrovni vytvoříme metodu `CreateNew`. Volání této metody jsme mohli vidět v části 5.1 při implementaci `CreateNewInstance`. Pro získání instance při načítání pak vytvoříme metodu `CreateForLoading`, která je použita pro implementaci `GetInstanceForLoading`:

```
public static Chicken CreateNew(ILevelManager level, IUnit unit,
    ChickenType type);  
  
public static Chicken CreateForLoading(ILevelManager level, IUnit
    unit, ChickenType type);
```

## Pro načítání uloženého stavu

Metoda `CreateForLoading` bude pouze jednoduché volání konstruktoru, tedy celá implementace bude vypadat následovně:

```
public static Chicken CreateForLoading(ILevelManager level, IUnit
    unit, ChickenType type) {
    return new Chicken(level, unit, type);
}
```

Konstruktor bude také jednoduchý, protože drtivou většinu dat budeme v tomto případě načítat z uloženého stavu. Implementace konstruktoru tedy bude takováto:

```
public Chicken(ILevelManager level,
    IUnit unit,
    ChickenType type)
:base(level,unit)
{
    this.myType = type;
    this.distCalc = new ChickenDistCalc(this);
    unit.AlwaysVertical = true;
}
```

Položka `myType` ukládá referenci na `ChickenType`, který obsahuje data o schůdných typech dlaždic, používaném typu projektilu a další data načtená z XML, společná všem jednotkám tohoto typu.

Pro implementaci pohybu po mapě bude naše jednotka využívat algoritmus pro hledání cesty, který bude požadovat instanci kalkulačoru ohodnocující hrany grafu. Touto instancí je právě instance `ChickenDistCalc`, popsaná v následujících částech.

`AlwaysVertical` položka instance jednotky určuje, zda se jednotka při pohybu otočí čelem přímo do směru pohybu, nebo zda se otočí pouze okolo osy Y a tedy její hlava bude stále kolmo nad terénem.

## Vytvoření v běžící hře

Implementace metody `CreateNew` specifikujeme, které součásti platformy budeme využívat. Tyto součásti, které platforma nazývá `DefaultComponent`, jsou schopny samostatného ukládání a načítání, stačí je tedy přidat na nově vytvářenou či existující jednotku. Vlastní implementace `CreateNew` bude začínat následovně:

```
public static Chicken CreateNew(
    ILevelManager level,
    IUnit unit,
    ChickenType type)
{
    Chicken newChicken =

```

```
new Chicken(level, unit, type);
newChicken.health = 100;
```

Jak můžeme vidět, jako první vytvoříme instanci pluginu. Důvodem jsou **DefaultComponenty**, které ve při svém vytváření potřebují referenci na instanční plugin. Navíc na tento plugin mají další požadavky, které uvedeme později. Dále inicializujeme počet životů jednotky na 100. Při načítání jednotky je počet životů načten z uloženého stavu, není proto inicializován v předchozí metodě **CreateForLoading**.

Následuje vytvoření a nastavení **DefaultComponentů** spolu s uložením referencí na tyto komponenty pro budoucí ovládání:

```
newChicken.Walker = WorldWalker.CreateNew(newChicken, level);
newChicken.Shooter = Shooter.CreateNew(newChicken, level,
type.ProjectileType, new Vector3(0, 0.7f, -0.7f), 20);
newChicken.Shooter.SearchForTarget = true;
newChicken.Shooter.TargetSearchDelay = 2;

MovingRangeTarget.CreateNew(newChicken, level, new Vector3(0,
0.5f, 0));

var selector = UnitSelector.CreateNew(newChicken, level);
```

Jak jsme specifikovali na počátku tohoto tutoriálu, naším cílem je jednotka, která je schopna pohybovat se po terénu, střílet po nepřátelských jednotkách, sama může být cílem nepřátelských jednotek a může být označena a ovládána hráčem. Tyto vlastnosti jsou popořadě implementovány komponentami **WorldWalker**, **Shooter**, **MovingRangeTarget** a **UnitSelector**. Jak můžeme vidět, instance komponent **WorldWalker** a **Shooter** si ukládáme pro budoucí ovládání v dalších metodách. Oproti tomu **MovingRangeTarget** neplánujeme měnit, proto pouze vytváříme jeho instanci na naší jednotce. Instanci **UnitSelctor** pak pouze nastavíme v této metodě a poté ji již nebudeme měnit.

Poslední požadovanou vlastností byl 3D animovaný model. Model samotný je k jednotce přidán platformou automaticky podle popisu v XML. Jedinou funkcí pluginu je následně spouštět a zastavovat animace modelu. K tomuto účelu poskytuje engine UrhoSharp komponentu **AnimationController**, kterou připojíme k naší jednotce. Také si uložíme referenci na tuto komponentu pro ovládání modelu v dále definovaných metodách.

```
newChicken.animationController =
unit.CreateComponent<AnimationController>();
```

Posledním krokem je registrace obsluh událostí. Platformou definované **DefaultComponenty** poskytují **eventy**, ke kterým si může plugin zaregistrovat obsluhu. V našem případě provedeme následující registraci:

```
walker.MovementStarted += OnMovementStarted;
walker.MovementFinished += OnMovementFinished;
walker.MovementFailed += OnMovementFailed;
walker.MovementCanceled += OnMovementCanceled;
```

```

shooter.BeforeShotFired += BeforeShotFired;
shooter.TargetAutoAcquired += OnTargetAutoAcquired;
shooter.TargetDestroyed += OnTargetDestroyed;

selector.UnitSelected += OnUnitSelected;

```

Události komponenty **WorldWalker** nastávají v těchto situacích:

- 1) **MovementStarted** event nastává při začátku pohybu;
- 2) **MovementFailed** event nastává při úspěšném dokončení pohybu do cíle;
- 3) **MovementFailed** event nastává při neúspěšném ukončení pohybu, například při zablokování cesty;
- 4) **MovementCanceled** event nastává při explicitním přerušení pohybu.

Události komponenty **Shooter** nastávají v těchto situacích:

- 1) **BeforeShotFired** event nastává těsně před vystřelením projektilu;
- 2) **TargetAutoAcquired** event nastává při nalezení cíle;
- 3) **TargetDestroyed** event nastává při zničení cíle.

Poslední událost **UnitSelected** nastává ve chvíli, kdy je jednotka označena hráčem.

Tímto je inicializace hotova. Nyní nám zbývá implementovat metody požadované předkem **UnitInstancePlugin**, požadované DefaultComponenty a obsluhující události.

### 5.2.2 Metody pluginu

Třída **UnitInstancePlugin** požaduje po svých potomcích implementaci sedmi metod. Těmito metodami jsou:

```

public abstract void SaveState(PluginDataWrapper pluginData);
public abstract void LoadState(PluginDataWrapper pluginData);

public abstract void Dispose();

public abstract void TileHeightChanged(ITile tile);
public abstract void BuildingBuilt(IBuilding building, ITile
    tile);
public abstract void BuildingDestroyed(IBuilding building, ITile
    tile);
public abstract void OnHit(IEntity other, object userData);

```

## Ukládání a načítání

Dvojice metod `SaveState` a `LoadState` umožňuje pluginu uložit aktuální stav v metodě `SaveState` a následně tento stav při načítání úrovně zpět načíst v metodě `LoadState`.

V naší jednotce je jedinou informací počet životů, které jednotce zbývají. Uložení této informace provedeme takto:

```
public override void SaveState(
PluginDataWrapper pluginDataStorage)
{
var writer =
pluginDataStorage.GetWriterForWrappedSequentialData();

writer.StoreNext(health);
}
```

Zpětné načtení jednotky bude potom vypadat následovně:

```
public override void LoadState(PluginDataWrapper pluginData) {
Unit.AlwaysVertical = true;

Walker = Unit.GetComponent<WorldWalker>();
Shooter = Unit.GetComponent<Shooter>();
var selector = Unit.GetComponent<UnitSelector>();

RegisterEvents(Walker, Shooter, selector);

animationController =
Unit.CreateComponent<AnimationController>();

var reader = pluginData.GetReaderForWrappedSequentialData();
reader.GetNext(out health);
}
```

Jak můžeme vidět, při načítání jednotky je potřeba vykonat více práce než při jejím ukládání. První znovu nastavíme držení vertikální pozice jednotky. Tato vlastnost se během hry nemění, proto ji nepotřebujeme ukládat a pouze ji nastavíme při vytváření a načítání jednotky.

Následuje získání referencí na `DefaultComponenty`. Jak bylo řečeno u vytváření nové instance, dokáží se tyto komponenty samy ukládat a načítat. Z tohoto důvodu nám zde stačí získat reference na již načtené komponenty.

Dále zaregistrujeme obsluhy událostí stejně jako při načítání. Následuje vytvoření komponentu `AnimationController`. Jak bylo zmíněno v části o vytváření instance, není tato komponenta poskytována platformou MHUrho ale samotným enginem UrhoSharp, není tedy schopna sama se uložit. Plugin tedy musí jak při vytváření nové jednotky tak při načítání existující tuto komponentu vytvořit znova. Jako poslední načteme uložený počet životů.

## Uvolnění zdrojů

Uvolnění dat je prováděn pomocí metody `Dispose`. V naší implementaci pluginu nevytváříme žádné zdroje, které by bylo nutné uvolňovat pomocí `Dispose`, implementace této metody tedy bude prázdná.

## Události platformy

Poslední čtyři metody požadované třídou `UnitInstancePlugin` jsou volány v těchto situacích:

- `TileHeightChanged` při změně výšky dlaždice, na které se jednotka právě nachází,
- `BuildingBuilt` při stavbě budovy na dlaždici, na které se jednotka právě nachází
- `BuildingDestroyed` při zničení budovy na dlaždici, na které se jednotka právě nachází,
- `OnHit` pokud je jednotka zasažena projektilom či útokem jiné jednotky či budovy.

Pro implementaci `TileHeightChanged` je důležité vědět, že platforma sama udržuje všechny součásti nad úrovní terénu. Metoda tedy bude zavolána, ale při tomto volání bude vždy jednotka nad úrovní terénu. Naopak pokud je výška terénu snížena, platforma s jednotkou neprovádí žádné akce. Naše implementace `TileHeightChanged` tedy při změně výšky dlaždice přesune jednotku v ose `Y`, tedy ve vertikální ose, na výšku terénu v daném bodě.

```
public override void TileHeightChanged(ITile tile)
{
    var newPosition = Unit.Position;
    newPosition.Y = Level.Map
        .GetHeightAt(newPosition.X,
    newPosition.Z);
    Unit.MoveTo(newPosition);
}
```

Property `Unit` a `Level` jsou poskytována předkem `UnitInstancePlugin`, kteřemu jsou předávány v konstruktoru, jak můžeme vidět v naší implementaci.

Metoda `BuildingBuilt` by neměla v naší ukázce nikdy být zavolána, protože vytvořené budovy nebudou dovolovat stavbu na dlaždicích obsahujících jednotky. Pro oznamení chyby při zavolání tedy bude metoda vyhazovat výjimku. Implementace tedy bude vypadat takto:

```
public override void BuildingBuilt(IBuilding building, ITile tile)
{
    throw new InvalidOperationException("Building building on top of
        units is not supported.");
}
```

Metoda `BuildingDestroyed` bude oproti předchozí metodě naší implementací využívána. Naše jednotka bude schopná chůze po budovách, při zničení budov bude tedy nutné jednotku přemístit zpět na úroveň terénu. Tuto funkcionality jsme již implementovali v metodě `TileHeightChanged`, implementace této metody bude tedy obdobná. Navíc při zničení budovy zastavíme pohyb jednotky. Implementace tedy bude vypadat takto:

```
public override void BuildingDestroyed(IBuilding building,
ITile tile)
{
    var newPosition = Unit.Position;
    newPosition.Y = Level.Map
        .GetHeightAt(newPosition.X,
    newPosition.Z);
    Unit.MoveTo(newPosition);
    Walker.Stop();
}
```

Poslední požadovaná metoda `OnHit` informuje jednotku o tom, že byla zasažena. Argumenty metody je `IEntity`, která nás zasáhla, tedy jednotka, budova či projektil, a `object`, který tato entita předala metodě `HitBy`, kterou zavolala na naši `Unit`. Tento `object` v naší hře představuje udělené poškození. Dále v naší hře nedovolujeme poškození přátelských jednotek. Implementace `OnHit` bude tedy následující:

```
public override void OnHit(IEntity other,
                           object userData)
{
    if (Unit.Player.IsFriend(other.Player)) {
        return;
    }

    int damage = (int)userData;
    health -= damage;

    if (health < 0) {
        animationController.PlayExclusive("Assets/Units/Chicken/Models/D
            0, false);
        dying = true;
        Shooter.Enabled = false;
        Walker.Enabled = false;
    }
}
```

Pokud je udílející jednotka přátelská, k žádnému udělení poškození nedojde. Poté získáme udělené poškození z argumentu `userData` a odečteme ho od aktuálního počtu životů jednotky. Pokud počet životů klesne pod nulu, pak spustíme animaci umírání a zastavíme komponenty `Walker` a `Shooter`, čímž se jednotka

přestane pohybovat a přestane střílet. Proměnná `dying` je použita v následující metodě pro změnu chování během sekvence umírání.

## OnUpdate

Předek `UnitInstancePlugin` poskytuje k přetížení ještě jednu metodu, kterou je `OnUpdate`. Tato metoda je volána při každém výpočtu stavu hry a umožňuje periodicky kontrolovat aktuální stav a podle tohoto stavu provádět úkony.

Naše jednotka bude v této metodě provádět několik úkonů. Těmito úkony budou:

- 1) odstranění jednotky z úrovně ve chvíli, kdy je dokončena animace umírání;
- 2) střelbu na cíl explicitně zvolený hráčem;
- 3) otočení proti aktuálnímu cíli střelby.

Pro kontrolu, zda byla dokončena animace umírání, použijeme komponentu `AnimationController`, kterou jsme vytvořili při vytvoření či načtení pluginu a uložili si na ni referenci v proměnné `animationController`. Zároveň využijeme proměnnou `dying`, kterou jsme nastavili v metodě `OnHit`, pospané v předešlé části. Kód bude tedy vypadat následovně:

```
if (dying &&
    animationController.IsAtEnd("Assets/Units/Chicken/Models/Dying.ani"))
{
    Unit.RemoveFromLevel();
    return;
}
```

## Metody požadované potomky DefaultComponent

### Implementace obsluh událostí

Umělá inteligence úrovně je určena pro kontrolu globálního stavu hry, jako například omezení na možnosti reliéfu mapy, omezení stavby budov v určitých částech mapy či změny vlastností některých jednotek. Rozdělení umělé inteligence mezi úroveň, hráče a entity je možné mnoha způsoby. Platforma žádný z těchto způsobů explicitně nepodporuje ani nezakazuje.

## 6. Ukázková hra

# Závěr

# Seznam použité literatury

- [1] Open-Asset-Importer-Lib. <http://www.assimp.org/index.php>. [Online; accessed 2019-05-17].
- [2] ADAMS, E. (2009). *Fundamentals of Game Design*. New Riders, 2 edition. ISBN 0-321-64337-2.
- [3] ALBAHARI, J. (2017). *C# 7.0 in a Nutshell*. O'Reilly UK Ltd. ISBN 978-1-491-98765-0. URL [https://www.ebook.de/de/product/29084295/joseph\\_albahari\\_c\\_7\\_0\\_in\\_a\\_nutshell.html](https://www.ebook.de/de/product/29084295/joseph_albahari_c_7_0_in_a_nutshell.html).
- [4] APPLE INC. (2019). iOS Security. [https://www.apple.com/business/site/docs/iOS\\_Security\\_Guide.pdf](https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf). [Online; accessed 2019-05-17].
- [5] BLIZZARD ENTERTAINMENT, INC. (2002). Warcraft 3. <https://playwarcraft3.com/en-us/>. [Online; accessed 2019-03-18].
- [6] BLIZZARD ENTERTAINMENT, INC. (2010). Starcraft 2. <https://starcraft2.com/en-us/>. [Online; accessed 2019-03-18].
- [7] BLIZZARD ENTERTAINMENT, INC. (2014). Hearthstone. <https://playhearthstone.com/en-us/>. [Online; accessed 2019-05-17].
- [8] FIRAXIS GAMES, INC. (2010). Civilization V. <https://civilization.com/civilization-5/>. [Online; accessed 2019-03-18].
- [9] FIREFLY STUDIOS (2002). Stronghold Crusader. <https://fireflyworlds.com/games/strongholdcrusader/>. [Online; accessed 2019-03-18].
- [10] GOOGLE (2018). Developer Guide. <https://developers.google.com/protocol-buffers/docs/overview>. [Online; accessed 2019-05-17].
- [11] GOOGLE (2019). Data and file storage overview. <https://developer.android.com/guide/topics/data/data-storage>. [Online; accessed 2019-05-18].
- [12] GOOGLE (2019). Device metrics. <https://material.io/tools/devices/>. [Online; accessed 2019-06-06].
- [13] GOOGLE (2019). Protocol Buffers. <https://developers.google.com/protocol-buffers/>. [Online; accessed 2019-05-17].
- [14] GRAVELL, M. protobuf-net. <https://github.com/mgravell/protobuf-net>. [Online; accessed 2019-06-06].
- [15] LANDER, R. (2019). Introducing .NET 5. <https://devblogs.microsoft.com/dotnet/introducing-net-5/>. [Online; accessed 2019-06-03].
- [16] LANTZ, J. (2008). Opinion: The Evolution of the Modern RTS. [http://www.gamasutra.com/php-bin/news\\_index.php?story=18326](http://www.gamasutra.com/php-bin/news_index.php?story=18326). [Online; accessed 2019-04-01].

- [17] MICROSOFT. KNOWNFOLDERID. <https://docs.microsoft.com/en-us/windows/desktop/shell/knownfolderid>. [Online; accessed 2019-05-17].
- [18] MICROSOFT. Environment.SpecialFolder Enum. <https://docs.microsoft.com/en-us/dotnet/api/system.environment.specialfolder?view=netframework-4.8>. [Online; accessed 2019-05-17].
- [19] MICROSOFT (2006). Chapter 4: Data and Settings Management. [https://docs.microsoft.com/en-us/previous-versions/ms995853\(v=msdn.10\)](https://docs.microsoft.com/en-us/previous-versions/ms995853(v=msdn.10)). [Online; accessed 2019-05-17].
- [20] MICROSOFT (2019). AssemblyLoadContext Class. <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.loader.assemblyloadcontext?view=netcore-2.2>. [Online; accessed 2019-06-03].
- [21] MONKEY SQUAD, S.A. DE C.V. (2015). Kerbal Space Program. <https://www.kerbalspaceprogram.com/>. [Online; accessed 2019-05-17].
- [22] NOVAK, M. (2014). Serialization Performance comparison (C#/NET) – Formats & Frameworks (XML–DataContractSerializer & XmlSerializer, BinaryFormatter, JSON–Newtonsoft & ServiceStack.Text, Protobuf, MsgPack). <https://maxondev.com/serialization-performance-comparison-c-net-formats-frameworks-xmldatacontr> [Online; accessed 2019-05-15].
- [23] O'BRIEN, L., SCHONNING, N., DUNN, C., UMBAUGH, B. a PAKALA, Y. (2017). iOS App Architecture. <https://docs.microsoft.com/en-us/xamarin/ios/internals/architecture>. [Online; accessed 2019-04-08].
- [24] OXEYE GAME STUDION (2008). RTS Game-play Part 3: Build Options. <http://www.oxeyegames.com/rts-game-play-part-3-build-options/>. [Online; accessed 2019-04-01].
- [25] PATEL, A. (2009). Introduction to A\*. [IntroductiontoA\\*](#). [Online; accessed 2019-05-17].
- [26] RELIC ENTERTAINMENT (2006). Company of Heroes. <http://www.companyofheroes.com/>. [Online; accessed 2019-03-18].
- [27] SMITH, F. (2016). Solving Ballistic Trajectories. [https://www.forrestthewoods.com/blog/solving\\_ballistic\\_trajectories/](https://www.forrestthewoods.com/blog/solving_ballistic_trajectories/). [Online; accessed 2019-05-17].
- [28] URHO3D. Urho3d. <https://urho3d.github.io/>. [Online; accessed 2019-05-17].
- [29] URHO3D. PODVector Documentation. [https://urho3d.github.io/documentation/1.4/class\\_urho3\\_d\\_1\\_1\\_p\\_o\\_d\\_vector.html#details](https://urho3d.github.io/documentation/1.4/class_urho3_d_1_1_p_o_d_vector.html#details). [Online; accessed 2019-05-17].

- [30] WALKER, M. H. (2004). Strategy Gaming: Part II. <https://web.archive.org/web/20070129065355/http://archive.gamespy.com/articles/february02/strategy02/>. [Online; accessed 2019-04-01].
- [31] WALKER, M. H. (2004). Strategy Gaming: Part V – Real-Time vs. Turn-Based. <https://web.archive.org/web/20081201113359/http://archive.gamespy.com/articles/february02/strategygames05/index.shtml>. [Online; accessed 2019-04-01].
- [32] WENZEL, M., YISHENGJIN, LATHAN, L., PRATT, T., PETRUSHA, R., HOFFMAN, M., JONES, M. a DEV, A. (2017). Best Practices for Assembly Loading. <https://docs.microsoft.com/en-us/dotnet/framework/deployment/best-practices-for-assembly-loading>. [Online; accessed 2019-04-08].
- [33] WESTWOOD STUDIOS (1992). Dune II. [https://en.wikipedia.org/wiki/Dune\\_II](https://en.wikipedia.org/wiki/Dune_II). [Online; accessed 2019-03-18].
- [34] WIKIPEDIA CONTRIBUTORS (2019). X-COM. <https://en.wikipedia.org/wiki/X-COM>. [Online; accessed 2019-05-17].
- [35] WIKIPEDIA CONTRIBUTORS (2019). Blitzkrieg (video game series). [https://en.wikipedia.org/wiki/Blitzkrieg\\_\(video\\_game\\_series\)](https://en.wikipedia.org/wiki/Blitzkrieg_(video_game_series)). [Online; accessed 2019-05-17].
- [36] WIKIPEDIA CONTRIBUTORS (2019). Blizzard Entertainment. [https://en.wikipedia.org/w/index.php?title=Blizzard\\_Entertainment&oldid=892549860](https://en.wikipedia.org/w/index.php?title=Blizzard_Entertainment&oldid=892549860). [Online; accessed 2019-05-17].
- [37] WIKIPEDIA CONTRIBUTORS (2019). Command & Conquer. [https://en.wikipedia.org/wiki/Command\\_%26\\_Conquer](https://en.wikipedia.org/wiki/Command_%26_Conquer). [Online; accessed 2019-05-17].
- [38] WIKIPEDIA CONTRIBUTORS (2019). Total War (series). [https://en.wikipedia.org/wiki/Total\\_War\\_\(series\)](https://en.wikipedia.org/wiki/Total_War_(series)). [Online; accessed 2019-05-17].
- [39] WIKIPEDIA CONTRIBUTORS (2019). Westwood Studios. [https://en.wikipedia.org/wiki/Westwood\\_Studios](https://en.wikipedia.org/wiki/Westwood_Studios). [Online; accessed 2019-05-17].
- [40] XAMARIN (2015). UrhoSharp. <https://docs.microsoft.com/en-us/xamarin/graphics-games/urhosharp/>. [Online; accessed 2019-05-15].

# A. Přílohy

## A.1 První příloha

## To do . . .

- 1 (p. 5): Možná vynechat Možná vynechat
- 2 (p. 10): Přeuspořádat Lépe uspořádat sekci o budovách
- 3 (p. 12): zničitelnost
- 4 (p. 12): různé druhy poškození
- 5 (p. 12): restrikce na umístění
- 6 (p. 35): Kde sem to nesel?
- 7 (p. 36): oproti čemu nízkou