VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

STRATEGICKÁ 3D HRA V UNITY

3D STRATEGY GAME IN UNITY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

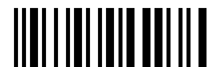
AUTOR PRÁCE **DAVID MACHŮ**
AUTHOR

VEDOUCÍ PRÁCE **Ing. MICHAL VLNAS**
SUPERVISOR

BRNO 2025

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce



164134

Ústav: Ústav počítačové grafiky a multimédií (UPGM)
Student: **Machů David**
Program: Informační technologie
Název: **Strategická 3D hra v Unity**
Kategorie: Počítačová grafika
Akademický rok: 2024/25

Zadání:

1. Nastudujte techniky herního vývoje, AI a vhodná vývojová prostředí.
2. Navrhněte 3D hru žánru realtime-strategy obsahující automaticky generované herní mapy, protihráče formou jednoduché AI a vhodné mechaniky.
3. Hru implementujte v prostředí Unity.
4. Proveďte uživatelské testování a zhodnoťte dosažené výsledky.
5. Vytvořte demonstrační video.

Literatura:

- Gregory, Jason. *Game engine architecture*. crc Press, 2018. ISBN 1351974289, 9781351974288
- Bishop, Lars, et al. "Designing a PC game engine." IEEE Computer Graphics and Applications 18.1 (1998): 46-53.
- Adams, Ernest, and Joris Dormans. Game mechanics: advanced game design. New Riders, 2012. ISBN 0321820274, 9780321820273
- Koster, Raph. Theory of fun for game design. O'Reilly Media, Inc., 2013.
- Schell, Jesse. The Art of Game Design: A book of lenses. CRC press, 2008.
- Unity Learn. Unity, https://learn.unity.com/.

Při obhajobě semestrální části projektu je požadováno:
Body 1, 2 a pokročilý prototyp bodu 3.

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

Vedoucí práce: **Vlnas Michal, Ing.**
Vedoucí ústavu: Černocký Jan, prof. Dr. Ing.
Datum zadání: 1.11.2024
Termín pro odevzdání: 14.5.2025
Datum schválení: 12.11.2024

Abstrakt
Tato práce se věnuje oblasti herního vývoje, procedurálnímu generování a umělé inteligence v herním designu. Cílem je realizovat hru žánru real-time strategie v Unity s automaticky generovanou herní mapou a protivníkem ve formě umělé inteligence. Herní mapa je generována za pomoci šumové funkce s tím, že je reprezentována šestiúhelníkovou mřížkou obsahující různé biomy. Součástí hry je sada typů jednotek s danými atributy a různými vlastnostmi chování. Jednotky mohou být nasazovány, vykonávat pohyb a útočit na nepřátelské jednotky, kde cílem je zničit nepřátelskou základnu. Protivník je realizován formou umělé inteligence schopným strategického rozhodování.

Abstract
This thesis focuses on the field of game development, procedural generation, and artificial intelligence in game design. The goal is to implement a real-time strategy game in Unity with an automatically generated game map and an opponent in the form of artificial intelligence. The game map is generated using a noise function and is represented by a hexagonal grid containing various biomes. The game includes a set of unit types with defined attributes and different behavioral characteristics. Units can be deployed, move, and attack enemy units, with the objective being to destroy the enemy base. The opponent is implemented in the form of artificial intelligence capable of strategic decision-making.

Klíčová slova
počítačová hra, herní vývoj, procedurální generování, umělá inteligence, Unity, real-time strategie, šestiúhelníková mřížka

Keywords
computer game, game development, procedural generation, artificial intelligence, Unity, real-time strategy, hexagonal grid

Citace
MACHŮ, David. *Strategická 3D hra v Unity.* Brno, 2025. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Michal Vlnas

Strategická 3D hra v Unity

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Michala Vlnase. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

....
David Machů
12. května 2025

Poděkování

Děkuji panu Ing. Michalu Vlnasovi za vedení bakalářské práce, odborné rady a vstřícný přístup v průběhu konzultací. Dále děkuji své rodině za podporu.

Obsah

1 Úvod **2**

2 Herní vývoj **3**
2.1 Základy herního designu . . . 3
2.2 Herní enginy . . . 6
2.3 Existující strategické hry . . . 10

3 Procedurální generování a umělá inteligence **12**
3.1 Procedurální generování . . . 12
3.2 Umělá inteligence . . . 17

4 Návrh **23**
4.1 Herní mapa . . . 23
4.2 Přehled jednotek . . . 28
4.3 Akce jednotek . . . 30
4.4 Umělá inteligence . . . 34

5 Implementace **37**
5.1 Herní mapa . . . 37
5.2 Správa herního jádra a uživatelského rozhraní . . . 42
5.3 Architektura a životní cyklus jednotek . . . 44
5.4 Pohyb jednotky . . . 49
5.5 Útok jednotky . . . 53
5.6 Umělá inteligence . . . 58
5.7 Testování . . . 65

6 Závěr **67**

Literatura **68**

A Obsah přiloženého paměťového média **70**

Kapitola 1

Úvod

Vývoj počítačových her patří mezi nejmladší a nejrychleji se rozvíjející oblasti zábavního průmyslu, které výrazně ovlivňují moderní kulturu. Hry dnes nejsou jen prostředkem odpočinku a relaxace, ale také nástrojem pro rozvoj strategického myšlení a již dávno nejsou jen volnočasovou zábavou, ale staly se i plnohodnotnou součástí profesionálního sportu. Vývoj her představuje unikátní propojení technické disciplíny a umělecké kreativity. Na jedné straně je potřeba zvládnout programovací techniky, zatímco na straně druhé se klade důraz na estetický design.

Téma této práce spadá do oblasti herního vývoje a zaměřuje se na tvorbu real-time strategie s procedurálně generovanou mapou založenou na šestiúhelníkovém rozvržení herní plochy, využívající šumovou funkci, a implementací protivníka řízeného umělou inteligencí se strategickými prvky rozhodování.

Hry žánru real-time strategie jsou schopny nabídnout hráčům výzvy spojené s plánováním, řízením zdrojů a taktickým myšlením v reálném čase, přičemž jsou často náročnější na pochopení a známé svou strmější učící křivkou. Nedílnou součástí snahy o zajištění znovuhratelnosti her je procedurální generování obsahu, a to včetně herních map. Existuje celá řada technik, přičemž jednou z nejpoužívanějších jsou šumové funkce, které se hojně využívají k reprezentaci terénu. Významnou roli v moderních hrách hraje také umělá inteligence, která umožňuje vytvářet přesvědčivější chování nepřátel a přizpůsobovat se hráčově strategii.

Cílem této práce je navrhnout a implementovat hru žánru real-time strategie s přístupnější učící křivkou. Návrh zahrnuje definici různých typů jednotek a herních mechanik jako je pohyb, útok a nasazení jednotek. Hra se odehrává na mapě, která je procedurálně generována s cílem zajistit variabilitu jednotlivých herních instancí. Mapa je tvořena šestiúhelníkovou mřížkou s různými biomy mající dopad na vlastnosti některých jednotek. Součástí je také realizace protivníka řízeného umělou inteligencí, který zohledňuje strategicky výhodnou volbu dostupných akcí.

Kapitola 2 objasňuje koncept hry samotný a následně popisuje okolnosti spjaté s jejich vývojem včetně herních enginů. V kapitole 3 jsou rozebrány koncepty procedurálního generování a umělé inteligence primárně zaměřeny v oblasti herního vývoje včetně vybraných technik v daném odvětví a využitých v rámci této práce. Návrh hry v kapitole 4 rozebírá matematické vlastnosti šestiúhelníkové mřížky s definováním postupu při generování herní mapy. Také je zde obsažen přehled o navržených jednotkách a jejich jednotlivých vlastnostech. Na závěr je zde popsán návrh rozhodování umělé inteligence. V kapitole 5 je detailně popsána implementační realizace hry.

Kapitola 2

Herní vývoj

Hry jsou přirozeným důsledkem lidské touhy objevovat, tvořit a překračovat hranice imaginace. Jako jeden z nejstarších způsobů zábavy a interakce umožňují lidem nejen bavit se, ale také se učit a rozvíjet. Pochopení obecného pojmu „hra“ je klíčové pro její vývoj, protože právě tento pojem definuje základní rámec, ve kterém se herní design odehrává.

> „*Hry vycházejí z lidské touhy po hraní a z naší schopnosti předstírat. Hraní představuje širokou kategorii činností, které nejsou nezbytné, jsou zpravidla rekreační a často mají i společenský význam. Předstírání je mentální schopnost vytvářet imaginární realitu, kterou člověk ví, že se liší od skutečného světa, a kterou může podle své vůle vytvářet, opouštět nebo měnit. Hraní a předstírání jsou základními prvky her. Oba tyto fenomény byly rozsáhle zkoumány jako kulturní a psychologické jevy [1].*“

2.1 Základy herního designu

Tato kapitola se zaměřuje na hlavní aspekty a principy, které tvoří základ při návrhu a tvorbě her.

2.1.1 Elementy hry

Hru utváří sada základních a rozšiřujících elementů, které Ernest Adams uvedl v knize *Fundamentals of Game Design, 3rd edition* [1]:

Hraní představuje aktivní zapojení hráče do dění hry, čímž se odlišuje od pasivních forem zábavy, jako je sledování filmů nebo čtení knih. Hráč prostřednictvím svých rozhodnutí a interakcí ovlivňuje průběh hry.

Předstírání je mentální proces, při kterém si hráči vytvářejí a přijímají imaginární realitu. Tento koncept umožňuje hráčům ponořit se do fiktivního světa hry a vnímat jej jako dočasně platnou realitu.

Cíl Každá hra má svůj cíl, který určuje její směr a poskytuje hráči motivaci. Tento cíl může mít různou podobu, od dosažení vítězství přes splnění úkolu až po dosažení určitého stavu ve hře.

Pravidla jsou základním toho, co je hře možné a co nikoliv. Definují hranice, v nichž se hráči mohou pohybovat, a poskytují herním akcím logiku a smysl. Díky pravidlům se hra stává organizovanou a konzistentní zkušeností.

Rozšiřující elementy hry Základní elementy hry nezahrnují aspekt konfliktu. Přesto je konflikt často jedním z klíčových prvků mnoha her, zejména těch, které kladou důraz na soutěžení nebo dramatický děj. Konflikt hráčům poskytuje motivaci, výzvy a příležitost k překonání překážek, čímž zvyšuje napětí a zapojení do hry. Hry s konfliktem, například strategie nebo akční tituly, umožňují hráčům rozvíjet taktické schopnosti, zlepšovat reflexy nebo prožívat emocionální drama. Konflikt také podporuje variabilitu a dynamiku, protože hráči mohou hledat různé způsoby jeho řešení.

Existují hry, které se obejdou bez konfliktu, například kreativní hry jako Minecraft[1], které nemají pevně stanovený soutěžní cíl. Podobně definice hry neobsahuje aspekt zábavy, protože ne všechny hry jsou navrženy primárně pro zábavu. Například vzdělávací hry, jako Duolingo[2], jsou určeny ke zlepšení jazykových dovedností, nikoli k poskytnutí zábavy nebo ke konkurenci.

Hry mohou mít různé formy a účely, ale mají společné základní elementy jako hraní, předstírání, cíle a pravidla, které poskytují dostatečně flexibilní rámec, aby zahrnuly široké spektrum herních zážitků, a dále mohou obsahovat i rozšiřující elementy.

2.1.2 Jádro herního designu

Jádrem herního designu je spojení konceptů výzev, kterým hráč musí čelit, aby dosáhl cíle hry a akcí, které má hráč povoleno provádět k překonání těchto výzev [1].

Výzvy by měly být přiměřené hráčovým schopnostem, aby hra zůstala zábavná, a zároveň dostatečně motivující, aby hráče vybízely k překonání překážek. Akce by měly být intuitivní a přímo reagovat na hráčovy záměry.

V rámci vývoje herní strategie je důležité vyvážit výzvy a akce tak, aby podporovaly hráčovu kreativitu při řešení problémů a zároveň nabízely různé způsoby dosažení cílů. To zahrnuje například zvládnutí návrhu hry tak, aby umožňovala různé styly přístupů hráče v daném okamžiku, jako je rychlá ofenziva nebo pečlivá správa zdrojů.

2.1.3 Proces herního designu

Proces herního designu zahrnuje následující kroky [1]:

1. **Představení si hry.** Prvním krokem je vytvoření představy o hře a její celkové koncepci.
2. **Definování toho, jak hra funguje.** Určení herních mechanik, pravidel a základních funkcí hry.
3. **Popsání prvků, které tvoří hru.** To zahrnuje konceptuální, funkční, umělecké a další prvky hry.
4. **Předání informací o hře týmu.** Zajištění jasné komunikace herní vize týmu, který bude hru vytvářet.

[1]Minecraft: https://www.minecraft.net/.
[2]Duolingo: https://www.duolingo.com/.

5. **Zdokonalování a ladění hry.** Proces iterace a testování během vývoje k dosažení kvalitního výsledku.

2.1.4 Herní mechaniky

Herní mechaniky jsou základní pravidla a systémy, které definují, jak hra funguje a jak hráč interaguje s herním světem. Jsou základem toho, co dělá hru hrou, a mohou zahrnovat různé prvky v závislosti na typu hry. Představují klíčový prvek každé hry, protože určují nejen herní zážitek, ale i vzájemnou interakci mezi hráčem a herním prostředím. Herní mechaniky určují, jaké výzvy hra hráči předkládá, jakými prostředky je může překonávat a jak jeho akce ovlivňují herní svět. Stanovují pravidla pro dosažení herních cílů a důsledky úspěchů i neúspěchů hráče [1].

Zajišťují také systém odměn, který motivuje hráče k dalšímu postupu. Tento systém může mít podobu materiálních odměn, jako jsou body, virtuální měna nebo nové herní prvky, či nemateriálních odměn, například pocitu uspokojení z dosažení cíle nebo zvládnutí výzvy. Odměny nejen udržují zájem hráče, ale také podporují jeho zapojení do hry, protože mu poskytují pozitivní zpětnou vazbu za jeho snahu a rozhodnutí.

Zároveň je zásadní, aby mechaniky harmonicky spolupracovaly s dalšími aspekty hry, jako je příběh, prostředí a uživatelské rozhraní.

Míra abstrakce

Herní mechaniky je nutné jasně definovat a převést do podoby, která umožní jejich jednoznačnou implementaci, přičemž je klíčové zohlednit míru realismu v závislosti na zaměření hry. Tento proces často zahrnuje zjednodušení nebo abstrahování reálných prvků, aby byla hra zábavná. Například v mnoha hrách, kde se vyskytují postavy, se běžně nezohledňují některé každodenní potřeby, jako je spánek nebo jídlo, které by mohly hráče zbytečně zatěžovat a odvádět od hlavního cíle hry [1].

Podobně se často zjednodušují environmentální a logistické faktory, jako jsou vlivy počasí, degradace materiálů nebo dlouhodobé dopady rozhodnutí. I když tyto prvky mohou zvýšit realističnost, jejich složitost by mohla narušit hratelnost a odradit širší publikum.

Mechaniky ekonomiky v real-time strategiích

V knize *Game Mechanics: Advanced Game Design* [2] autoři popisují, jak frekventované a zásadní je použití interní ekonomiky v real-time strategiích. Tento prvek herního designu přináší do her nejen strategickou hloubku, ale také klade důraz na dlouhodobé plánování a rozhodování. Ekonomika se stává hlavním prvkem, který ovlivňuje tempo a dynamiku hry [2].

Zároveň autoři upozorňují, že prvek ekonomiky se nemusí vždy vázat na tradiční koncepty měny nebo produkce. Například v šachu se správa zdrojů týká figurek, kdy hráči pečlivě zvažují, kdy své zdroje obětovat, využít nebo investovat pro dosažení strategické výhody. Podobný princip se objevuje například ve hře Prince of Persia: The Sands of Time[3], kde hráči spravují písek umožňující manipulaci s časem. Tato omezená mechanika přidává vrstvu strategie, protože hráči musí rozhodovat, kdy a jak tento zdroj využít, čímž se hra stává taktickou i na úrovni správy zdrojů [2].

[3] Prince of Persia: The Sands of Time: https://en.wikipedia.org/wiki/Prince_of_Persia:_The_Sands_of_Time

Symetrie v herním designu

Pro obzvlášť strategické hry s prvkem konfliktu je často žádanou vlastností to, co autor popisuje jako *symetrii*:

> *„V symetrické hře hrají všichni hráči podle stejných pravidel a snaží se dosáhnout stejné podmínky pro vítězství [1].“*

Například lední hokej díky úvodnímu vhazování (buly) poskytuje zcela vyvážené podmínky, protože oba týmy mají stejnou šanci získat kontrolu nad pukem. To činí hokej příkladem symetrické hry, kde jsou pravidla a herní prostředky pro oba týmy identické.

2.1.5 Uživatelské rozhraní ve hrách

Hráčovy akce ve hrách nejsou vždy navrženy tak, aby byly co nejjednodušší nebo nejpřímočařejší. Naopak jsou často záměrně ztíženy výzvami, které od hráče vyžadují strategické myšlení a kreativní řešení. Například v sérii Dark Souls[4] hráči čelí obtížným protivníkům a složitým prostředím, která vyžadují nejen přesnou koordinaci pohybů, ale také hluboké pochopení mechanik hry, jako je správné načasování útoků, výběr vybavení nebo využití slabin nepřátel. Na rozdíl od uživatelského rozhraní v běžném softwaru, kde je cílem co nejefektivnější a nejjednodušší interakce, je herní uživatelské rozhraní navrženo tak, aby podporovalo výzvy. Mnoho her navíc hráčům neodhaluje všechny informace ihned, ale postupně je zpřístupňuje v průběhu hraní, čímž podporuje objevování a prohlubuje hráčský zážitek [1].

Uživatelské rozhraní ve hrách se výrazně liší podle typu hry a jejího žánru. V některých hrách, zejména v těch zaměřených na atmosféru a hluboké ponoření do herního světa (například adventury), je minimalismus v uživatelském rozhraní zásadní. Čím méně prvků hráč vidí, tím více se může ponořit do herního světa a lépe prožít jeho atmosféru.

Na druhé straně existují hry, které vyžadují neustálé informování hráče o stavu hry. Typickým příkladem jsou strategické hry nebo hry zaměřené na správu zdrojů, kde uživatelské rozhraní musí poskytovat detailní informace, jako jsou statistiky, časové údaje nebo mapy. V těchto případech je přehledné a informativní uživatelské rozhraní zásadní pro úspěch hráče.

2.2 Herní enginy

Herní engine je softwarová platforma navržená pro zjednodušení a zrychlení vývoje her. Poskytuje řadu nástrojů ve formě knihoven a subsystémů, které vývojářům umožňují efektivně vytvářet hry bez nutnosti vyvíjet základní infrastrukturu od nuly. Herní enginy fungují jako znovupoužitelný základ, který lze přizpůsobit specifickým potřebám projektu. Výběr konkrétního enginu závisí na typu a zaměření hry. Mezi populární patří Unity[5], Unreal Engine[6] a Godot[7]. Unreal Engine je často volbou pro AAA tituly, které kladou důraz na realistickou grafiku a fyziku, zatímco Unity a Godot jsou oblíbené v oblasti indie her díky své flexibilitě a přístupnosti. Herní enginy rovněž nabízejí podporu pro různé cílové platformy, což umožňuje vytváření buildů pro PC, konzole, mobilní zařízení i webové aplikace [4].

[4] Dark Souls: https://en.wikipedia.org/wiki/Dark_Souls
[5] Unity: https://unity.com/
[6] Unreal Engine: https://www.unrealengine.com/en-US
[7] Godot: https://godotengine.org/

2.2.1 Základní subsystémy herních enginů

Následující subsystémy poskytují abstrakce nad běžně se vyskytujícími komponentami potřebnými pro vývoj her. Jejich existence v herních enginech umožňuje vývojářům se plně soustředit přímo na tvorbu herního obsahu [4]:

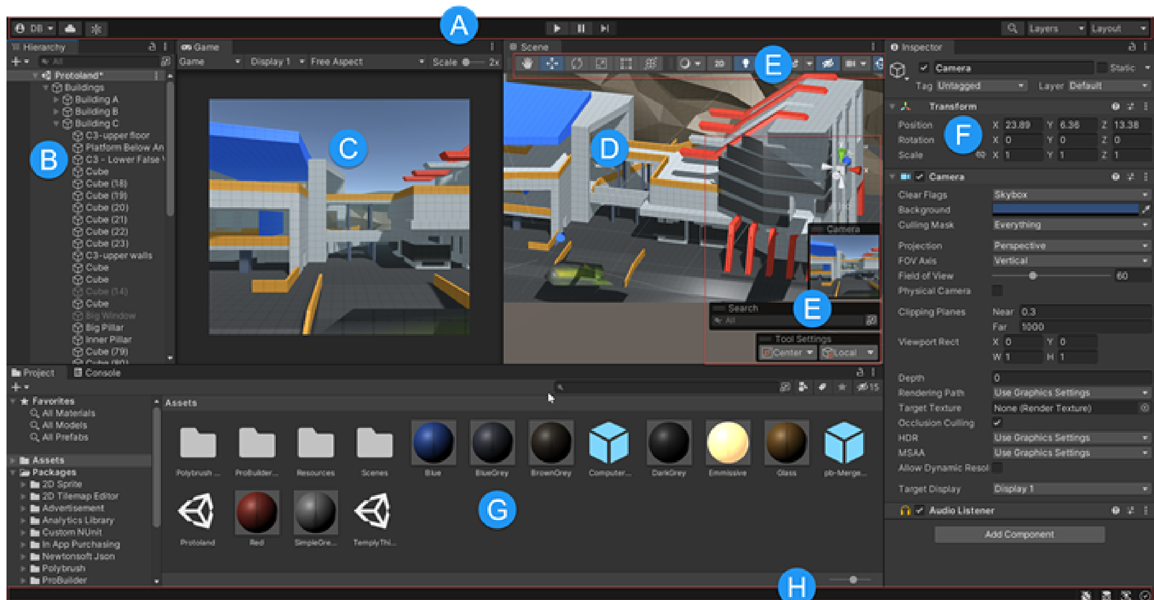
- **Renderovací engine:** Starost o vizuální zpracování hry, vykreslování objektů, materiálů, světel a efektů. Využívá technologie například OpenGL[8] nebo DirectX[9].
- **Fyzikální engine:** Řeší simulaci fyzikálních jevů, jako jsou kolize, gravitace, pohyby a interakce objektů. Příklady zahrnují Havok[10] nebo PhysX[11].
- **Zvukový systém:** Práce se zvukovými efekty, hudbou a prostorovým zvukem.
- **Síťový systém:** Podpora pro online hraní, synchronizace mezi hráči a serverem.
- **Scéna:** Hierarchická struktura, která organizuje objekty ve hře a jejich vztahy (entity a komponenty).
- **Vstupní a výstupní zařízení:** Správa interakce hráče s hrou pomocí ovladačů, klávesnic, myší, dotykových obrazovek nebo dalších zařízení.
- **Správa assetů:** Nástroje pro import a organizaci zdrojů (modely, textury, zvuky).
- **Nástroje pro debugging:** Sledování výkonu, logování a diagnostika problémů pro odhalování a opravy chyb.
- **Uživatelské rozhraní:** Tvorba vizuálních prvků, jako jsou menu, tlačítka, panely a ukazatele s využitím data bindingu (například zdraví nebo skóre).

Při vývoji her se často využívají i externí nástroje, které rozšiřují možnosti tvorby herního obsahu. Pro vytváření 3D modelů, animací a dalších vizuálních prvků je oblíbeným nástrojem Blender[12], zatímco pro práci s 2D grafikou, texturami či návrhem uživatelského rozhraní slouží například GIMP[13].

2.2.2 Herní engine Unity

Unity [19] je jedním z nejpopulárnějších herních enginů na světě, který umožňuje vývoj her pro různé platformy, včetně PC, mobilních zařízení, konzolí a webu. Tento engine nabízí širokou škálu nástrojů pro tvorbu 2D i 3D her, včetně grafického rozhraní, fyzikálních simulací a podpory pro skriptování. Náhled hlavního okna je možné vidět na obrázku 2.1.

[8] OpenGL: https://www.opengl.org/
[9] DirectX: https://learn.microsoft.com/en-us/windows/win32/directx
[10] Havok: https://www.havok.com/
[11] PhysX: https://developer.nvidia.com/physx-sdk
[12] Blender: https://www.blender.org/
[13] GIMP: https://www.gimp.org/



Obrázek 2.1: Náhled hlavního okna editoru Unity převzat z oficiální dokumentace Unity[14]. Popis jednotlivých částí: (A) Toolbar, (B) Hierarchy window, (C) Game view, (D) Scene view, (E) Overlays, (F) Inspector window, (G) Project window, (H) Status bar.

Prefaby

Prefaby jsou šablony herních objektů, které lze opakovaně používat. Umožňují efektivní správu složitých prvků, například nepřátel, střel nebo terénu. Změny provedené v prefabu se automaticky aplikují na všechny jeho instance.

Scény a hierarchie objektů

Unity organizuje hru do **scén**, které obsahují hierarchii herních objektů:

- **Hierarchie objektů:** Umožňuje organizaci vztahů mezi objekty.
- **Komponenty:** Připojené ke každému hernímu objektu určují jeho funkčnost (např. fyzika, zvuk, skripty).

Skriptování v C#

Herní logika se v Unity implementuje pomocí jazyka C#:

- **MonoBehaviour:** Základní třída, která poskytuje přístup k metodám, jako `Start()` nebo `Update()`.
- Skripty lze snadno připojit jako komponenty k herním objektům.

[14] Zdroj obrázku: https://docs.unity3d.com/2022.3/Documentation/Manual/UsingTheEditor.html.

Vývoj uživatelského rozhraní (UI)

Unity podporuje několik nástrojů pro tvorbu uživatelského rozhraní:

- **UI Toolkit:** Nejnovější systém pro vývoj uživatelských rozhraní v Unity, inspirovaný webovými technologiemi. Používá *UXML* pro strukturování rozhraní (podobně jako HTML) a *USS* pro stylování prvků (podobně jako CSS).
- **Canvas System:** Tradiční nástroj pro návrh uživatelských rozhraní přímo ve scéně. Je ideální pro 2D prvky, které jsou renderovány na *Canvasu*. Každý prvek je organizován v hierarchii objektů, což usnadňuje správu složitějších rozvržení.
- **TextMeshPro:** Nástroj pro pokročilou práci s textem.

Tvorba animací

Unity obsahuje vestavěný nástroj pro tvorbu animací:

- **Animator Controller:** Řídí přechody mezi animacemi na základě podmínek.
- **Animation Clips:** Krátké segmenty animací, které lze použít například pro pohyb.
- **Timeline:** Umožňuje synchronizaci animací s dalšími událostmi, jako je přehrávání zvuku.

Správa projektu a balíčků

Unity poskytuje nástroje pro správu obsahu a balíčků:

- **Package Manager:** Instalace a aktualizace oficiálních i komunitních balíčků.
- **Assets:** Struktura adresářů pro organizaci všech souborů projektu včetně herních modelů, textur, skriptů, materiálů a dalších zdrojů. Složky jsou obvykle pojmenovávány podle konvencí, jako například `Scripts` pro skripty, `Materials` pro materiály nebo `Prefabs` pro prefaby.

Shader Graph

Shader Graph je vizuální editor pro tvorbu shaderů, který umožňuje návrh materiálů pomocí jednoduchého propojení uzlů.

Mapování ovládání a práce s ovladači

Unity podporuje širokou škálu vstupních zařízení prostřednictvím dvou hlavních systémů:

- **Legacy Input Manager:** Starší systém pro základní zpracování vstupů, vhodný pro jednodušší projekty.
- **Input System:** Moderní systém nabízející flexibilnější mapování akcí, podporu různých zařízení a možnost dynamického přepínání mezi nimi.
- **Podpora zařízení:** Gamepady, joysticky, klávesnice a další vstupní zařízení s možností přizpůsobení ovládání preferencím hráče.

2.3 Existující strategické hry

V herním světě existuje mnoho žánrů, které oslovují různé typy hráčů. Obvykle má hra primární žánr, jenž určuje její hlavní charakter, a sekundární žánr, který tento primární žánr podporuje. Například akční hry s temnými hororovými prvky oslovují hráče hledající napětí a atmosféru strachu, zatímco akční hry s barevným a hravým stylem mohou cílit na kompetitivní publikum. Kombinace žánrů a stylů tak umožňuje přizpůsobit hru specifickým preferencím různých hráčských skupin.

Například tahové strategie často přitahují hráče, kteří preferují klidnější a promyšlenější přístup, zatímco real-time strategie vyžadují nejen strategické uvažování, ale také rychlé reakce, čímž oslovují dynamičtější povahy.

Strategický herní žánr zaujímá významné postavení v herním světě díky své schopnosti oslovit široké spektrum hráčů a spoléhat na přirozené lidské přemýšlení, plánování a taktizování. Schopnost strategicky uvažovat je hluboce zakořeněná v lidské přirozenosti, a proto mnoho her, i těch, které nespadají čistě do strategického žánru, vyžaduje od hráčů důvtip a rozhodovací schopnosti.

Například i v populárních hrách, jako je League of Legends[15], je klíčovým prvkem strategické myšlení, ať už jde o výběr správného šampiona nebo koordinaci s týmem. I když se tato strategie odehrává v reálném čase, probíhá hlavně v mysli hráče, který analyzuje situace, odhaduje kroky soupeře a přizpůsobuje svou hru.

Cílem této sekce je nahlédnout na některé z existujících strategických her, které mají podobné mechaniky nebo koncepty jako projekt vyvíjený v této práci. Mezi hry tohoto typu lze zařadit Civilization VI[16] (viz obrázek 2.2), Age of Empires IV[17] (viz obrázek 2.3) a Teamfight Tactics (TFT)[18] (viz obrázek 2.4).

Obrázek 2.2: **Civilization VI**[19] je tahová strategická hra, kde hráči budují civilizace od starověku až po moderní dobu. Hra nabízí široké možnosti správy zdrojů, diplomacie, vědeckého pokroku a vojenských strategií, což umožňuje hráčům zvolit různé cesty k vítězství.

[15] League of Legends: https://www.leagueoflegends.com/cs-cz/
[16] Civilization VI: https://store.steampowered.com/app/289070/Sid_Meiers_Civilization_VI/
[17] Age of Empires IV: https://www.ageofempires.com/games/age-of-empires-iv/
[18] Teamfight Tactics: https://teamfighttactics.leagueoflegends.com/cs-cz/
[19] Zdroj obrázku: https://store.steampowered.com/app/289070/Sid_Meiers_Civilization_VI/.

Obrázek 2.3: **Age of Empires IV**[20] je real-time strategická hra, která hráčům umožňuje vést civilizace napříč historickými érami. Hra kombinuje správu ekonomiky, vojenské strategie a historicky inspirované kampaně, čímž nabízí dynamický herní zážitek, kde hráči musí balancovat mezi expanzí, obranou a inovacemi.

Obrázek 2.4: **Teamfight Tactics (TFT)**[21] je automatizovaná strategická hra, kde hráči sestavují týmy šampionů, kteří následně bojují na hexagonální mřížce. Hra kombinuje strategické plánování, správu zdrojů (zlato) a adaptaci na herní situaci.

[20] Zdroj obrázku: Steam – Age of Empires IV Anniversary Edition.

[21] Zdroj obrázku: `https://teamfighttactics.leagueoflegends.com/cs-cz/news/game-updates/teamfight-tactics-reckoning-ii-gameplay-overview/`.

Kapitola 3

Procedurální generování a umělá inteligence

Tato kapitola se zaměřuje na dvě významné oblasti vývoje her – procedurální generování a umělou inteligenci. Představuje základní principy obou oblastí a vybrané techniky a algoritmy.

3.1 Procedurální generování

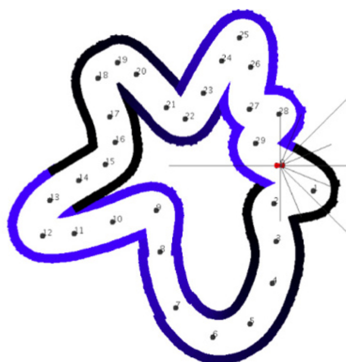
Procedurální generování obsahu představuje oblast herního vývoje, která se zabývá automatizovaným vytvářením herního obsahu pomocí algoritmů [9]. Tento přístup umožňuje vytvářet herní světy a situace, které by bylo obtížné nebo časově náročné navrhovat ručně. Uplatnění nachází například při tvorbě opakovaně hratelných úrovní, dynamické hudby, unikátních příběhů řízených samotnými hráči [17], ale také při generování map, textur, předmětů a dalších herních prvků [9].

I když je procedurální generování silným nástrojem, jeho využití s sebou nese určitá rizika. Výsledný obsah nemusí být vždy kvalitnější nebo efektivnější než ručně vytvořený, a jeho tvorba může být náročná na zdroje i ladění. Proto je klíčové zvolit vhodný přístup s ohledem na konkrétní problém, herní žánr a cíle projektu [17].

3.1.1 Přístup založený na vyhledávání

Přístup založený na vyhledávání využívá evoluční algoritmy nebo jiné stochastické optimalizační metody k vytváření herního obsahu, který splňuje předem definovaná kritéria, jako je hratelnost, vyváženost či estetická kvalita. Návrh obsahu je zde vnímán jako problém hledání v rozsáhlém prostoru možných řešení, přičemž každé řešení představuje jednu variantu úrovně, mapy, nepřítele nebo herního úkolu [9].

Základ tohoto přístupu tvoří vyhledávací algoritmus, který generuje nové návrhy a postupně je vylepšuje podle jejich hodnocení. Důležitou roli hraje i způsob reprezentace obsahu, který může mít formu například číselných polí, grafů nebo matic, a dále hodnoticí funkce, která jednotlivým návrhům přiřazuje číselnou hodnotu na základě zvolených kvalitativních kritérií [9]. Ukázky praktického využití tohoto přístupu při návrhu závodní tratě a strategické mapy znázorňuje obrázek 3.1.



(a) Trať vytvořená evolucí na základě posloupnosti Bézierových křivek [9].

(b) Vyvinutá mapa pro strategickou hru StarCraft [18].

Obrázek 3.1: Příklady využití přístupu založeného na vyhledávání při generování herního obsahu.

3.1.2 Formální gramatiky a L-systémy

Formální gramatiky (viz definice 1) nacházejí uplatnění v oblasti procedurálního generování, zejména při vytváření struktur jako jsou bludiště, jeskyně nebo rostliny a porosty [9].

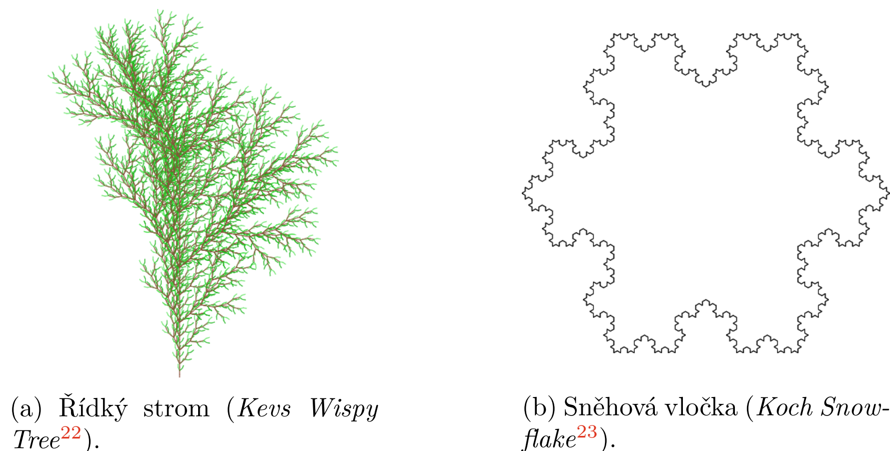
Definice 1. Gramatika ve zpětné-Naurově formě (BNF) [9] je čtveřice $G = (N, T, P, S)$, kde:

- N je konečná množina neterminálů,
- T je konečná množina terminálů, přičemž $N \cap T = \emptyset$,
- P je konečná množina pravidel přepisování tvaru $A \to \alpha$, kde $A \in N$ a $\alpha \in (N \cup T)^*$,
- $S \in N$ je počáteční symbol.

Formální gramatiky představují soubor pravidel, která určují, jak lze přepisovat řetězce znaků. Každé pravidlo definuje, jak se určitý symbol nebo skupina symbolů nahradí jinými. Při zpracování se řetězec prochází a vždy, když se v něm objeví levá strana některého pravidla, dojde k jejímu nahrazení pravou stranou. Obvykle se rozlišují neterminální symboly (velká písmena), které se dále přepisují, a terminální symboly (malá písmena), které již zůstávají beze změny. V kontextu procedurálního generování je důležité, zda je gramatika *deterministická* (každý symbol má právě jedno přepisovací pravidlo), nebo *nedeterministická*, kde může existovat více možností. Dále hraje roli i způsob aplikace pravidel – například sekvenční či paralelní přepis [9].

Speciálním typem formální gramatiky jsou *L-systémy*, které se vyznačují **paralelním přepisem**, kde na rozdíl od tradičních gramatik jsou při každé iteraci aplikována všechna pravidla současně. Tato vlastnost umožňuje i s velmi jednoduchými pravidly generovat složité a pravidelné struktury. Ukázkovým příkladem může být L-systém definovaný pravidly $A \to AB$, $B \to A$, jenž generuje posloupnosti A, AB, ABA, $ABAAB$, ... [9].

Tento proces lze vizuálně interpretovat pomocí tzv. *želví grafiky*, kde jednotlivé symboly určují pohyb a směr kresby. Pomocí této metody lze generovat dvourozměrné i trojrozměrné obrazce, jako jsou větvící se rostliny nebo fraktální tvary viz obrázek 3.2 [9].



(a) Řídký strom (*Kevs Wispy Tree*[22]).

(b) Sněhová vločka (*Koch Snowflake*[23]).

Obrázek 3.2: Ukázky obrazců vygenerovaných pomocí L-systémů.

L-systémy se tak osvědčují jako silný nástroj pro procedurální generování, který může být dále kombinován například s evolučními algoritmy pro dosažení vyšší variability a kontroly nad výstupem [9].

3.1.3 Šumové funkce

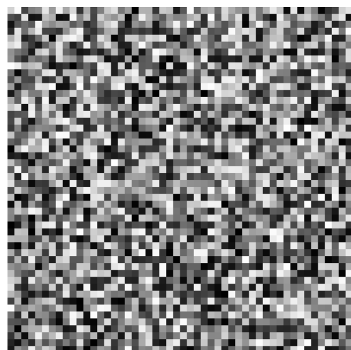
Šumovou funkci lze chápat jako matematické vyjádření, které pro zadaný vstup x vrací hodnotu typicky v rozsahu $\langle 0; 1 \rangle$. Vstupem nemusí být pouze reálné číslo, ale také bod ve dvou, třech či čtyřech dimenzích. Hodnota x může být kladná i záporná a teoreticky se může pohybovat v nekonečném rozsahu v obou směrech, tedy $x \in (-\infty, +\infty)$ [15].

Při generování výstupních hodnot pomocí generátoru náhodných čísel by byl získán **bílý šum**, jehož vlastnosti nejsou považovány za vhodné pro grafické účely – hodnoty jsou generovány zcela náhodně a mezi sousedními body nevzniká žádná kontinuita. Takový výstup je vnímán jako neuspořádaný a postrádající přirozenou plynulost, která je typická pro vzory pozorované v přírodě [15].

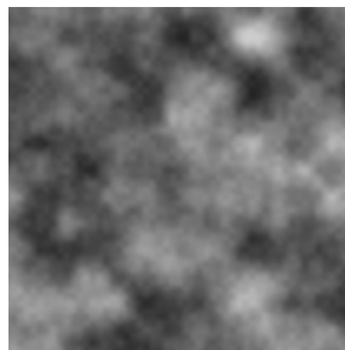
Pro dosažení vizuálně plynulejších struktur je při generování **hodnotového šumu** využívána interpolace. Náhodné hodnoty jsou přiřazovány pouze vrcholům pravidelné mřížky (například v 2D prostoru) a výsledná hodnota pro libovolný bod mezi těmito vrcholy je následně dopočítávána pomocí interpolační funkce. Nejčastěji je používána lineární interpolace v 1D prostoru a bilineární interpolace v 2D prostoru [15, 16]. Dalším krokem ve vývoji šumových funkcí bylo představení **Perlinova šumu**, který dále zlepšuje kontinuitu a vizuální kvalitu generovaných dat viz obr. 3.3.

V oblasti počítačové grafiky slouží šum jako efektivní nástroj pro obohacení vizuálního vzhledu scén, a to především prostřednictvím textur [6]. Významné uplatnění nachází také ve hrách, jejichž prostředí často zahrnuje terén, který neplní pouze roli vizuální kulisy, ale ovlivňuje i samotnou hratelnost. Pro generování takového prostředí se běžně využívají šumové funkce, jež umožňují vytvářet přirozeně působící krajinu s výškovou členitostí a variabilním povrchem [9].

[22] Zdroj obrázku: `https://www.kevs3d.co.uk/dev/lsystems/`.
[23] Zdroj obrázku: `https://www.kevs3d.co.uk/dev/lsystems/`.



(a) Náhodně generované hodnoty[24] (bílý šum).



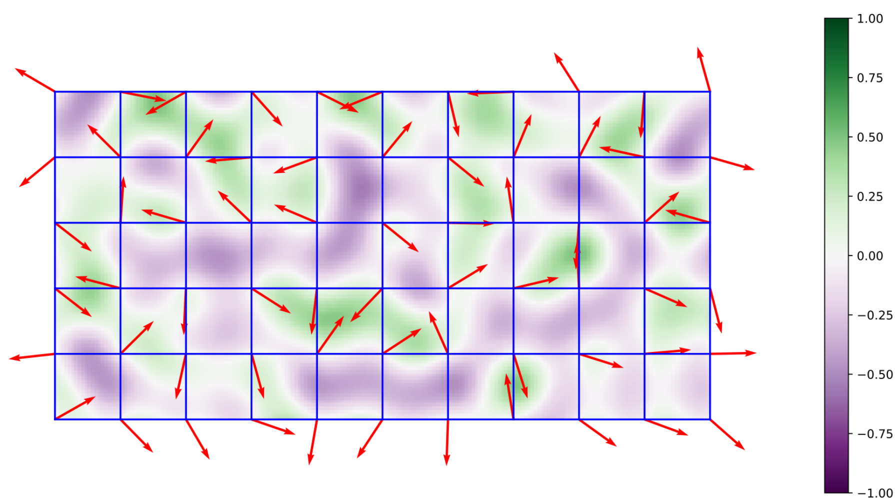
(b) Perlinův šum[25].

Obrázek 3.3: Porovnání náhodného šumu a Perlinova šumu.

Perlinův šum

Perlinův šum je typ gradientového šumu, který navrhl Ken Perlin v roce 1985 pro potřeby počítačové grafiky [11]. Stejně jako u hodnotového šumu se Perlinův šum opírá o systém pravidelné mřížky, ačkoli místo přiřazování náhodných čísel do vrcholů se používají gradienty nebo-li náhodné normalizované vektory [14] viz obrázek 3.4.

Výstupní hodnota se vypočítá tak, že se nejprve určí buňka mřížky, ve které se daný bod nachází. Pro každý vrchol této buňky se následně vytvoří vektor směřující z vrcholu do daného bodu a provede se skalární součin tohoto vektoru s gradientem ve vrcholu, čímž vznikne reálná hodnota [14]. Tyto hodnoty se poté interpolují, například pomocí bilineární interpolace v případě 2D prostoru, nebo pomocí běžně používané vyhlazovací funkce tvaru $f(t) = 3t^2 - 2t^3$ [15], kterou Ken Perlin později vylepšil do podoby funkce v následujícím tvaru $f(t) = 6t^5 - 15t^4 + 10t^3$ [12].

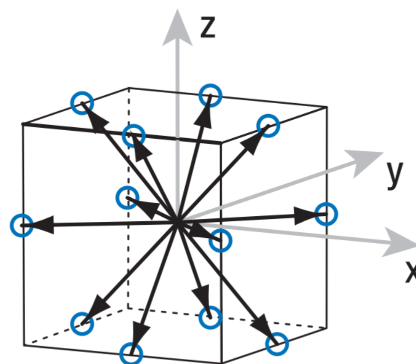


Obrázek 3.4: Gradientní vektory na pravidelné mřížce a odpovídající interpolovaná šumová hodnota v pozadí[26].

[24] Zdroj obrázku: https://www.zazow.com/info/perlin-noise.php.
[25] Zdroj obrázku: https://www.zazow.com/info/perlin-noise.php.

$$\begin{array}{cccc} (1,1,0) & (-1,1,0) & (1,-1,0) & (-1,-1,0) \\ (1,0,1) & (-1,0,1) & (1,0,-1) & (-1,0,-1) \\ (0,1,1) & (0,-1,1) & (0,1,-1) & (0,-1,-1) \end{array}$$

(a) Matematicky



(b) 12 gradientů v 3D [5].

Obrázek 3.5: Srovnání matematické reprezentace a vizualizace 12 gradientů.

Dalším vylepšením je použití dvanácti předem definovaných gradientových vektorů, které jsou záměrně orientovány mimo hlavní osy a dlouhé diagonály. Tím dochází k odstranění směrové zaujatosti a k rovnoměrnějšímu rozložení šumových vzorů. Ken Perlin zvolil vektory (viz obr. 3.5), které odpovídají směrům od středu krychle ke středům jejích hran [12]. Gradienty lze získat pomocí souřadnic buňky, ve které se daný bod nachází, a jejich použitím v hashovací funkci, která vrací index do tabulky předem definovaných směrů [5].

Simplexový šum

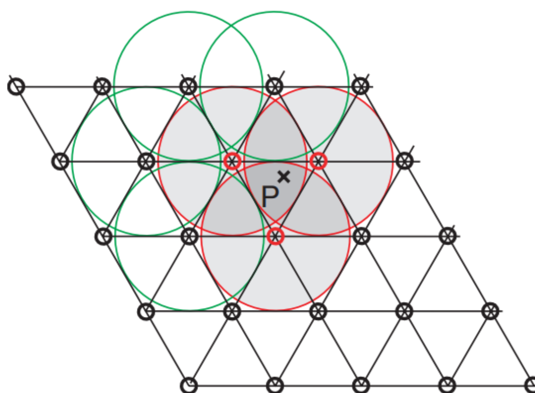
Výpočet vychází z odlišného uspořádání mřížky než u Perlinova šumu a místo interpolace v každé dimenzi používá přímé sčítání příspěvků z jednotlivých vrcholů *simplexu* [5].

Simplexová mřížka představuje způsob rozdělení vícerozměrného prostoru na co nejjednodušší možné prvky, tzv. **simplexy**. Ve 2D prostoru jde o rovnostranné trojúhelníky, ve 3D o zkosené čtyřstěny a obecně v N dimenzích o tvary s $N + 1$ vrcholy. Oproti běžné mřížce tvořené hyperkrychlemi, které mají 2^N vrcholů, nabízí simplexová mřížka menší počet vrcholů, což přináší nižší výpočetní náročnost [5].

Pro bod P uvnitř jednoho simplexu se z každého jeho vrcholu získá gradientní vektor a provede se skalární součin s vektorem směřujícím z vrcholu do bodu P (podobně jako u Perlinova šumu). Výsledný příspěvek je dále násoben útlumovou funkcí závislou na relativní vzdálenosti od vrcholu, například $t = 0{,}5 - x^2 - y^2$. Pokud je $t < 0$, příspěvek je nulový, jinak se tlumí pomocí t^4 viz obr. 3.6. Hodnota šumu v bodě se pak získá jako součet těchto tří příspěvků [5].

Simplexový šum přináší oproti Perlinovu šumu řadu výhod včetně vyšší rychlosti generování a vizuální rovnoměrnosti ve všech směrech, čímž eliminuje směrové artefakty. Jeho širší využití však omezuje patentová ochrana. Jako alternativu proto vznikl algoritmus **OpenSimplex**, který nabízí srovnatelnou kvalitu bez právních omezení a je dostupný jako open-source řešení [17].

[26] Zdroj obrázku: https://en.wikipedia.org/wiki/Perlin_noise.



Obrázek 3.6: Bod je ovlivněn pouze třemi blízkými vrcholy svého simplexu, zatímco vzdálenější příspěvky zanikají před dosažením hranice [5].

3.2 Umělá inteligence

„Umělá inteligence se zabývá tím, jak přimět počítače, aby zvládaly myšlenkové úkoly, které jinak zvládají lidé a zvířata“ [8]. Počítače dnes zvládají úlohy jako třídění nebo vyhledávání lépe než lidé, i když tyto schopnosti byly dříve považovány za umělou inteligenci. Naopak úkoly, které lidé zvládají snadno jako rozpoznávání tváří nebo kreativní činnost, zůstávají pro stroje náročné. Co dnes označujeme za umělou inteligenci, často závisí jen na obtížnosti problému [8]. V rámci akademického pojetí umělé inteligence se dělí na silnou, která usiluje o napodobení lidského myšlení, a slabou, která řeší specifické úkoly v aplikovaných oborech [7].

Vývoj akademické umělé inteligence lze podle Iana Millingtona rozdělit do tří období: raného období, symbolické éry a éry přirozeného počítání a statistiky. Ačkoliv se tato období částečně překrývají, představují důležité směry, ze kterých herní vývojáři čerpají osvědčené principy a techniky [8].

Rané období Předcházelo vzniku počítačů a bylo ovlivněno filozofií o mysli. Ve 40. letech motivovala válka vývoj prvních programovatelných počítačů, přičemž průkopníci jako Turing nebo von Neumann zároveň položili základy rané umělé inteligence [8].

Symbolická éra Zhruba od 50. do 80. let dominovaly symbolické systémy založené na pravidlech a znalostech. Typické techniky zahrnovaly expertní systémy, stavové automaty nebo rozhodovací stromy. Principem bylo, že čím více má umělá inteligence znalostí, tím méně musí hledat a naopak [8].

Éra přirozeného počítání a statistiky Od 80. let rostla nespokojenost se symbolickými metodami kvůli jejich omezené škálovatelnosti a biologické nevěrohodnosti. Výzkum se přesunul k technikám inspirovaným přírodou, jako jsou například genetické algoritmy nebo neuronové sítě. Přirozeně inspirované techniky tak otevřely cestu k moderním přístupům, jako je hluboké učení [8].

Ve vývoji umělé inteligence přetrvává základní rovnováha mezi hledáním a znalostmi. Moderní přístupy jako hluboké učení spoléhají na rozsáhlé výpočetní hledání s minimem vstupních znalostí, zatímco jednodušší úkoly lze často řešit přímo na základě předem daných pravidel. Hry jsou však většinou navržené pro běh na běžném spotřebitelském hardwaru,

kde je výpočetní kapacita omezená a většinu výkonu zabírá grafika. Proto bývají ve hrách efektivnější metody s nízkými nároky na výpočet a větším podílem znalostí často právě symbolické techniky, které se v herní umělé inteligenci uplatňují dodnes. Ačkoli jsou statistické výpočetní metody v určitých případech užitečné, v herní praxi lze obdobných výsledků často dosáhnout jednoduššími a lépe kontrolovatelnými postupy [8].

Systém umělé inteligence lze chápat jako kombinaci učící se a rozhodovací složky, avšak herní umělá inteligence často žádný učící subsystém neobsahuje. Místo adaptivního chování na základě dat spoléhá spíše na předem definovaná pravidla a deterministické postupy. Tyto techniky sice neodpovídají klasickému pojetí inteligence, ale v kontextu her plně postačují. Jejich cílem není skutečné porozumění či učení, ale vytváření iluze inteligentního chování, které podpoří plynulý a uvěřitelný herní zážitek [7].

3.2.1 Model herní umělé inteligence

Na obrázku 3.7 je znázorněn obecný model herní umělé inteligence, který navrhl Ian Millington. Tento model rozděluje herní umělou inteligenci do tří hlavních úrovní [8]:

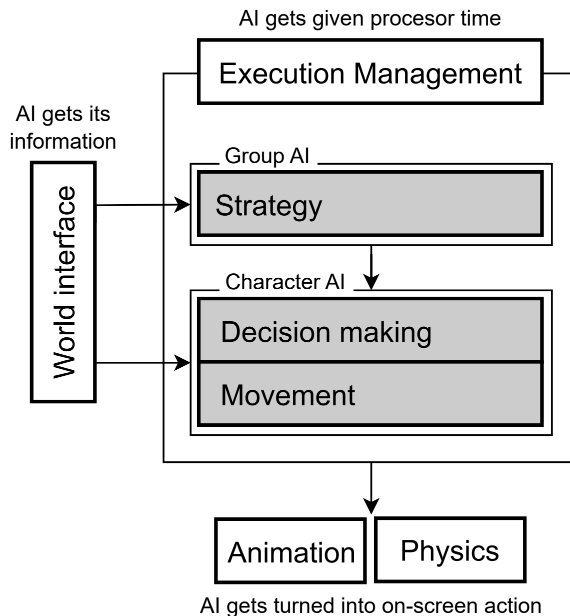
- **Pohybová vrstva** zajišťuje převod rozhodnutí postavy do konkrétní akce v herním světě. Může se jednat o jednoduché směřování k cíli, ale také o složitou navigaci zahrnující vyhýbání se překážkám nebo průchod složitým prostředím.

- **Rozhodovací vrstva** určuje, jaké chování má postava v dané situaci zvolit. Na základě informací z herního světa (vnějších znalostí) a vlastního vnitřního stavu (například zdraví, cílů nebo minulých akcí) vybírá vhodnou akci, kterou má vykonat. Výstupem rozhodnutí je obvykle požadavek na konkrétní činnost, kterou následně provede pohybová vrstva nebo animační systém.

- **Strategická vrstva** se zaměřuje na koordinaci chování více postav současně. Zatímco pohyb a rozhodování probíhá na úrovni jednotlivců, strategie určuje celkový postup skupiny. Například plánování útoku, obklíčení hráče nebo rozdělení úkolů.

Model nepředstavuje jediný možný způsob návrhu herní umělé inteligence, ale poskytuje obecný rámec odpovídající běžným potřebám her. Jeho jednotlivé části nemusí být vždy přítomné, což závisí na konkrétním žánru a stylu hry. Například pohybová vrstva není nutná ve hrách s diskrétním nebo předem definovaným pohybem (např. tahové strategie), kde je samotný přesun součástí rozhodnutí. Podobně strategická vrstva může být vynechána ve hrách, kde postavy jednají izolovaně bez nutnosti týmové spolupráce (např. plošinové hry) [8].

Kromě použití vhodných algoritmů ve třech hlavních vrstvách umělé inteligence (pohyb, rozhodování, strategie) je zásadní také správné navržení rozhraní mezi umělou inteligencí a ostatními systémy hry. Umělá inteligence potřebuje přístup k informacím z herního světa, aby mohla činit smysluplná rozhodnutí, a zároveň musí být schopna převádět výstupy (například pohybové příkazy) do konkrétních animačních nebo fyzikálních akcí. Toto propojení s herním světem představuje významný prvek nejen při získávání a předávání informací, ale také z hlediska řízení výpočetních nároků umělé inteligence systému [8].



Obrázek 3.7: Obecný model herní umělé inteligence [8].

3.2.2 Konečné stavové automaty (Finite State Machines)

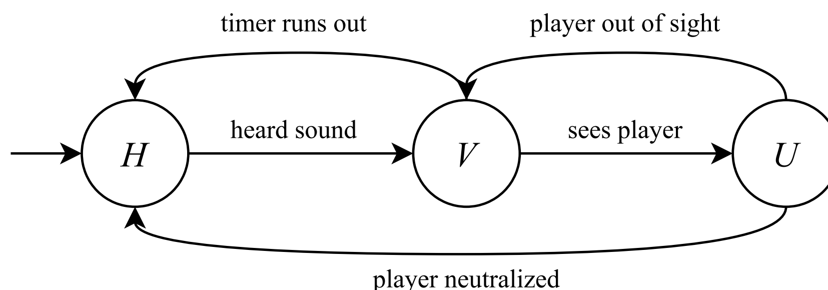
Herní postavy se často chovají podle omezené sady vzorců, přičemž v daném chování setrvávají, dokud nenastane událost, která jejich jednání změní. Pro modelování tohoto typu rozhodování se nejčastěji využívají stavové automaty, které byly navrženy právě pro tento účel a jejich použití je zpravidla jednodušší než u alternativních metod [8]. Konečný stavový automat je definován v definici 2.

Definice 2. Konečný automat (FSM) [7] lze z pohledu teorie grafů definovat jako uspořádanou dvojici $G = (S, E)$, kde:

- $S = \{s_1, \ldots, s_n\}$ je konečná množina vrcholů (stavů automatu),
- $E \subseteq \{(s_i, s_j) \mid s_i, s_j \in S\}$ je množina hran (přechodů), které jsou uspořádanými dvojicemi prvků z S (tj. přechod ze stavu s_i do s_j).

Přechody nemusí být nutně mezi různými stavy, jinými slovy je povolena i smyčka $s_i \to s_i$. Každý přechod může být navíc označen událostí nebo podmínkou, která jeho aktivaci umožňuje [7].

Obrázek 3.8 schématicky znázorňuje možné chování strážného ve hře se skrytým postupem jako konečný stavový automat. Výchozím stavem je hlídka (H), kdy se pohybuje po předem definované trase. Jakmile uslyší podezřelý zvuk, přejde do stavu vyšetřování (V). Pokud při vyšetřování uvidí hráče, přejde do stavu útoku (U). Pokud hráče ztratí z dohledu, vrací se nejprve zpět do vyšetřování, a pak do patroly. V případě, že je hráč zneškodněn, NPC se rovnou vrací zpět do hlídky.



Obrázek 3.8: Názorný příklad využití konečného automatu (FSM) pro řízení chování nepřátelského NPC. Diagram ukazuje možné stavy (hlídka, vyšetřování, útok) a podmínky, za kterých mezi nimi dochází k přechodům.

3.2.3 Systémy založené na pravidlech (Rule-Based Systems)

Pravidlové systémy obvykle sestávají ze dvou hlavních částí: **znalostní báze**, která obsahuje informace dostupné umělé inteligenci, a **sady pravidel** ve formátu `if-then`. Každé pravidlo porovnává svou podmínku s aktuálním stavem znalostní báze, a v případě, že je tato podmínka splněna, dané pravidlo se označí jako aktivní. Aktivní pravidla pak mohou být vybrána k provedení, což znamená, že se vykoná jejich akční část. Některé implementace pravidlových systémů obsahují i rozhodovací mechanismus, který v případě vícero aktivních pravidel určuje, které z nich bude skutečně provedeno [8].

Jednoduchý pravidlový systém s nízkou škálovatelností může být užitečný v kombinaci s jinými rozhodovacími algoritmy. Nechť $C = \{c_1, c_2, \ldots, c_n\}$ je množina podmínek a $A = \{a_1, a_2, \ldots, a_n\}$ množina akcí. Pravidlový systém pak může být implementován tak, že každé splněné pravidlo z množiny C spouští odpovídající akci z množiny A. Na obrázku 3.9 je znázorněn příklad, jak lze takový systém reprezentovat pomocí podmíněného výrazu a odpovídající tabulky rozhodnutí [7].

Podmínky v pravidlech mohou být složené a zohledňovat více aspektů současně. Například pravidlo může testovat stav několika jednotek a přítomnost určitého objektu [8]:

```
if soldier1.health <= 0 && soldier1.has(bomb) then
    soldier2.pickup(bomb)
```

Tímto pravidlem se definuje, že pokud jednotka `soldier1` zemřela a má u sebe bombu, jednotka `soldier2` by měla bombu sebrat. Je však důležité poznamenat, že výsledek pravidla v tomto případě pouze určuje záměr nebo cíl akce, který se bude dále realizovat v rámci jiného subsystému [8].

```
if c_1 then
  a_1
else if c_2 then
  a_2
  ⋮
else if c_n then
  a_n
end if
```

(a) Implementace pravidlového systému jako podmíněný výraz [7].

c_1	$\cdots$	c_n	**Akce**
TRUE	$\cdots$	$*$	a_1
$\vdots$	$\ddots$	$\vdots$	$\vdots$
FALSE	$\cdots$	**TRUE**	a_n

(b) Rozhodovací tabulka pro výběr akce na základě splněné podmínky [7].

Obrázek 3.9: Srovnání zápisu pravidlového systému.

Iterativní způsob implementace, viz Algoritmus 1 [8], pravidlového systému začíná s databází obsahující relevantní data. Externí sada funkcí do ní nejprve přenese informace o aktuálním stavu hry, přičemž databáze může zároveň uchovávat i další údaje, jako je například vnitřní stav řízené postavy. Pravidla se poté aplikují opakovaně v jednotlivých iteracích, během nichž se každé pravidlo porovná se stavem databáze. Jakmile je nalezeno první splňující pravidlo, je provedena s ním spojená akce a iterace končí. Tento přístup odpovídá strategii *first applicable*, kdy se vždy provede pouze první aktivované pravidlo v daném pořadí [8].

Algoritmus 1 Iterace pravidlového systému

```
1: function RULEBASEDITERATION(database: DataNode, rules: Rule[])   ▷ Projdi všechna pravidla
2:     for all rule in rules do
3:         bindings ← []                                            ▷ Vytvoř prázdné vazby
4:         if rule.ifClause.matches(database, bindings) then        ▷ Pravidlo se spustí
5:             rule.getActions(bindings)                            ▷ Proveď akce
6:             break                                                ▷ Ukonči iteraci
7:         end if
8:     end for    ▷ Pokud se žádné pravidlo nespustilo, lze použít náhradní akci nebo nic
9: end function
```

3.2.4 Systémy založené na užitečnosti (Utility-Based AI)

Přístup založený na užitečnosti představuje flexibilní a často využívanou metodu rozhodování agentů ve hrách [3]. Základní princip spočívá v tom, že každé možné akci je přiřazena číselná hodnota pomocí tzv. **užitkové funkce**, která reprezentuje její přínos nebo vhodnost v daném kontextu. Na základě těchto hodnot označovaných jako **skóre užitku** agent vybírá tu akci, která je v dané chvíli nejvýhodnější [3, 13].

Aby bylo možné jednotlivé akce mezi sebou objektivně porovnávat, je nutné, aby jejich ohodnocení probíhalo na jednotné škále. Nejčastěji se využívají normalizované hodnoty v intervalu 0 až 1, které se snadno kombinují (například průměrováním) a umožňují efektivní porovnání různých rozhodovacích faktorů. Obecně však lze použít jakýkoli číselný rozsah, pokud je dodržena konzistence napříč všemi proměnnými [13].

Jedním z rozšířených přístupů je výpočet tzv. maximálního očekávaného užitku, který zohledňuje pravděpodobnosti jednotlivých výsledků dané akce. Nejprve se každému možnému výsledku přiřadí jeho užitek a pravděpodobnost výskytu, a poté se jejich součin sečte. Výsledkem je očekávaný užitek dané akce [13]:

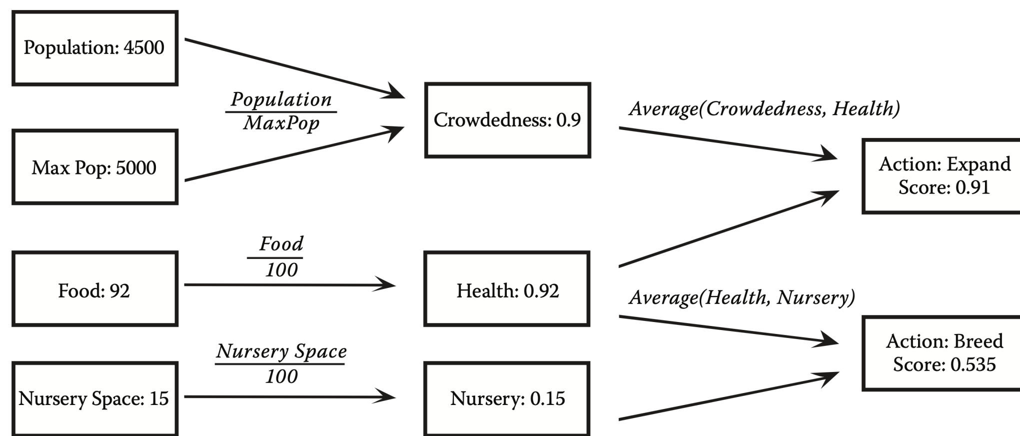
$$EU = \sum_{i=1}^{n} D_i \cdot P_i, \tag{3.1}$$

kde D_i je užitek daného výsledku a P_i pravděpodobnost, že tento výsledek nastane. Pravděpodobnosti musí být normalizovány tak, aby jejich součet byl roven 1 [13].

Typicky se rozhodnutí obvykle nezakládá pouze na jednom jediném údaji. Každé rozhodnutí je ovlivněno více různými aspekty, které lze chápat jako **rozhodovací faktory**. Tyto faktory reprezentují jednotlivé pohledy nebo kritéria, která je třeba při volbě zohlednit. Jejich význam lze dále upravit pomocí vah, které určují relativní důležitost každého faktoru [13].

Jedním z možných přístupů je aplikovat výpočet očekávaného užitku samostatně na každý faktor a tím získat jeho dílčí příspěvek k rozhodnutí. Pokud jsou výsledná skóre jednotlivých faktorů normalizována, je možné je zprůměrovat a tím získat celkové užitkové skóre dané akce. Rozhodnutí pak lze chápat jako kombinaci těchto dílčích vstupů [13].

Tento princip rozhodování umělé inteligence lze například uplatnit v simulaci mraveniště (viz schématický obrázek 3.10). Systém vyhodnocuje tři rozhodovací faktory: přeplněnost kolonie, zásoby potravy a dostupné místo v chovné oblasti. Každý faktor je vyjádřen jako normalizované skóre v rozsahu 0 až 1. Pro akce rozšíření kolonie a rozmnožování jsou použity různé kombinace faktorů, jejichž hodnoty jsou zprůměrovány. Skóre pro rozšíření vychází z přeplněnosti a zdraví kolonie, zatímco skóre pro rozmnožování zohledňuje dostupné místo v chovné oblasti a zdraví [13].



Obrázek 3.10: Příklad výpočtu užitkového skóre pro dvě akce (rozmnožování a rozšíření kolonie) v simulaci mraveniště na základě rozhodovacích faktorů. Výsledná skóre slouží k určení, která akce je v dané situaci výhodnější [13].

Kapitola 4

Návrh

Navržená hra spadá do žánru real-time strategie s válečnou tematikou, zpracovanou v minimalistickém a abstraktním vizuálním stylu. Hra se odehrává se na mapě tvořenou šestiúhelníky s různými biomy. Hráč ovládá jednotky (pozemní, námořní a vzdušné), spravuje ekonomiku, kde cílem je zničit nepřátelskou základnu.

Herní mapa je procedurálně generována a obsahuje různé typy terénu jako jsou pláně, lesy, hory a vodní plochy, přičemž každý typ ovlivňuje pohyb nebo efektivitu jednotek. Součástí návrhu je také systém umělé inteligence, který řídí chování protivníka. Hra je zpracována ve 3D prostředí s použitím izometrického pohledu, který zajišťuje dobrou přehlednost dění na mapě.

Real-time strategie jsou obecně známé svou vysokou komplexitou a nároky na hráčovu pozornost, plánování a schopnost rychle reagovat na změny v herní situaci. Osvojení základních herních mechanik zpravidla vyžaduje delší čas a značnou míru trpělivosti, což přispívá ke strmé učící křivce. Návrh této hry proto klade důraz na intuitivní uživatelské rozhraní a sníženou komplexitu, která však stále zachovává herní prvky a nabídne zážitky typické pro žánr real-time strategií.

4.1 Herní mapa

Cílem herní mapy je zajistit variabilitu, tedy přispět k znovuhratelnosti, a zároveň zachovat určitou míru kontroly nad výslednou strukturou herního světa. V případě symetrických strategických her (viz podsekce 2.1.4) je tato kontrola obzvláště důležitá, neboť je nezbytné zajistit naprosto rovné podmínky pro obě zúčastněné strany.

Mapa je tvořena šestiúhelníky označenými biomy, které slouží jako základní prvky pro pohyb jednotek a další interakce. Jelikož je hra určena pro dvě oponující strany, celkové rozvržení mapy do jisté míry sleduje eliptický tvar se základnami umístěnými na protilehlých stranách.

Struktura mapy je rozdělena do dvou výškových vrstev, které jsou reprezentovány samostatnými šestiúhelníkovými mřížkami: *pozemní vrstva* a *vzdušná vrstva*.

Každému šestiúhelníku je přiřazen jeden z následujících biomů, které ovlivňují herní mechaniky, jako je pohyb, výhoda v boji nebo dostupnost určitých jednotek: *pláně*, *les*, *kopce*, *voda* a *vzduch*.

4.1.1 Generování šestiúhelníkové mřížky

Tato sekce popisuje způsob rozmístění šestiúhelníků, které dohromady tvoří záměrně nedokonale eliptický tvar. Toho je dosaženo kombinací deterministického výpočtu a šumové funkce, jež zajišťuje variabilitu mezi jednotlivými herními instancemi. Výsledkem je definice šestiúhelníkové mřížky pro pozemní i vzdušnou vrstvu mapy v rámci jedné konkrétní hry.

Určení indexace a světové pozice šestiúhelníků

Tato podsekce ukazuje konkrétní aplikaci geometrických vlastností šestiúhelníků a indexační techniky [10].

Šestiúhelníky v šestiúhelníkové mřížce lze indexovat pomocí offsetových souřadnic (x, y), případně přehlednější kubickou reprezentací (q, r, s), která je výhodná zejména pro výpočty vzdáleností či určování sousedních šestiúhelníků.

Pro výpočet světové pozice středu šestiúhelníku ve *flat-top* uspořádání (tj. šestiúhelníky s vodorovnou základnou) je třeba znát rozměry jednoho šestiúhelníku a rozestupy mezi jeho sousedy.

Za předpokladu, že s je poloměr opsané kružnice jednoho šestiúhelníku (vzdálenost od středu k vrcholu), pak:

- horizontální rozestup mezi středy sousedních šestiúhelníků je $h = \frac{3}{2} \cdot s$,
- vertikální rozestup je $v = \sqrt{3} \cdot s$.

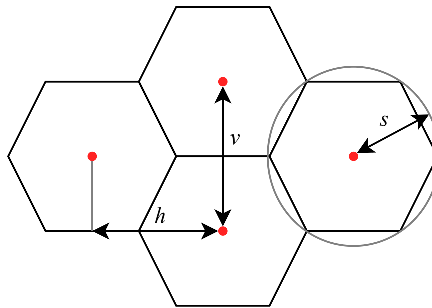
Pro dané souřadnice (x, y) lze světovou pozici šestiúhelníku získat následovně:

$$X = x \cdot h, \quad Z = -\left(y \cdot v + \delta_y\right),$$

kde δ_y je svislý posun aplikovaný pouze v lichých sloupcích a odpovídá výrazu:

$$\delta_y = \begin{cases} 0 & \text{pro sudé } x, \\ \frac{\sqrt{3}}{2} \cdot s & \text{pro liché } x. \end{cases}$$

Tento systém odpovídá tzv. *odd-q* offsetové reprezentaci, při které jsou liché sloupce vertikálně posunuty pro dosažení pravidelného šachovnicového vzoru. Záporné znaménko v ose Z vychází z orientace světového souřadného systému ve vývojovém prostředí Unity, kde osa Z směřuje směrem „dopředu“ na obrazovce. Použití záporného směru zajišťuje, že rostoucí hodnota souřadnice y odpovídá vizuálnímu pohybu dolů po mapě.



Obrázek 4.1: Geometrie hexagonální mřížky ve *flat-top* orientaci.

Převod z offsetových souřadnic (x, y) do kubických souřadnic (q, r, s), které splňují invariant $q + r + s = 0$, se provádí následovně:

$$q = x, \quad r = y - \left\lfloor \frac{x - (x \bmod 2)}{2} \right\rfloor, \quad s = -q - r$$

Tento převod odpovídá *odd-q* orientaci, kde jsou liché sloupce (osa x) vertikálně posunuty. Pomocí kubických souřadnic je následně možné snadno určovat sousední šestiúhelníky, počítat vzdálenosti nebo nalézat symetricky umístěné šestiúhelníky vůči danému středu nebo ose.

Formování mapy

Na základě indexačních souřadnic (x, y) lze nyní určit světové souřadnice šestiúhelníku (X, Y, Z), kde složka Y pouze rozlišuje výškové vrstvy: pozemní (Y_p) a vzdušné (Y_v). Herní mapa sleduje přibližně eliptický tvar, přičemž základní struktura je dána elipsou s poloosami a a b. Tato elipsa definuje hlavní oblast, do které jsou šestiúhelníky generovány.

Pro dosažení větší variability se část šestiúhelníků může nacházet i mírně za hranicí elipsy. O tom, zda budou tyto šestiúhelníky vloženy do herního světa, rozhoduje šumová funkce. Samotné generování probíhá v rozsahu:

$$-a \leq x \leq 0, \quad -b \leq y \leq b,$$

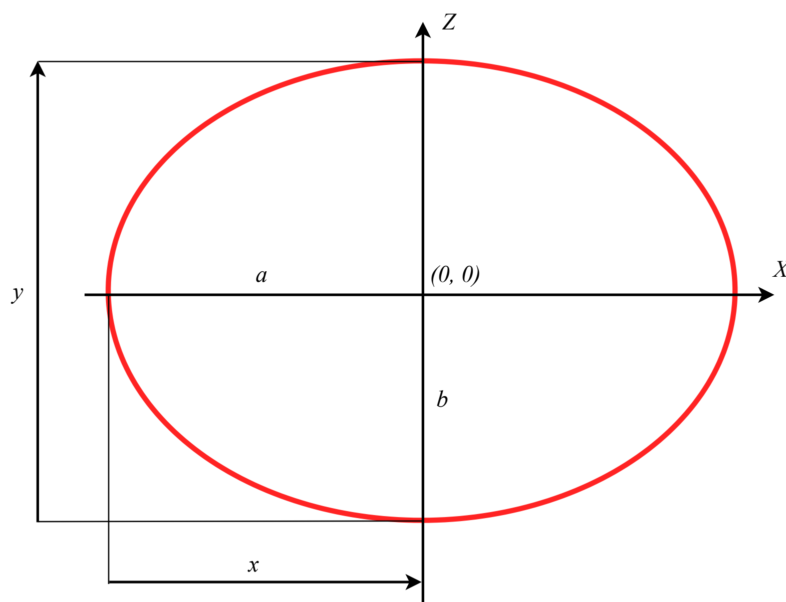
což umožňuje vygenerovat pouze polovinu mapy (viz obr. 4.2). Následným zrcadlením vůči osám $X = 0$ a $Z = 0$ je dosaženo symetrie herní mapy.

Pokud je konkrétní světová pozice šestiúhelníku vyhodnocena jako vhodná, vloží se do herní mapy:

- pozemní šestiúhelník na souřadnice (X, Y_p, Z),
- vzdušný šestiúhelník na souřadnice (X, Y_v, Z).

Navíc, pokud $x < 0$, jsou na opačné straně mapy vygenerovány zrcadlené šestiúhelníky na souřadnicích:

- pozemní šestiúhelník na souřadnice $(-X, Y_p, -Z)$,
- vzdušný šestiúhelník na souřadnice $(-X, Y_v, -Z)$.



Obrázek 4.2: Rozsah indexů při generování šestiúhelníkové mřížky.

Pro určení zda světová pozice šestiúhelníku je vyhodnocena jako vhodná, je použita normalizovaná vzdálenost vzhledem k hlavním poloosám elipsy a a b, spočítaná podle rovnice:

$$d = \sqrt{\left(\frac{X}{a}\right)^2 + \left(\frac{Z}{b}\right)^2}$$

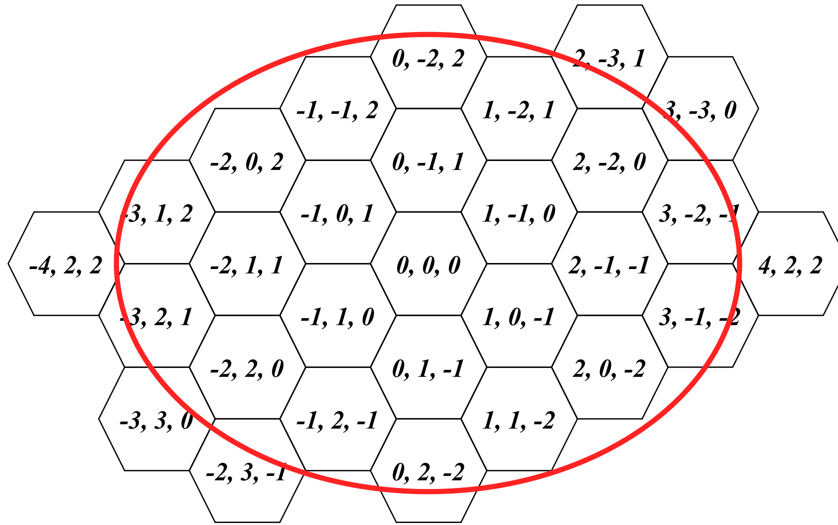
Hodnota d udává relativní vzdálenost od středu.

- Pokud $d \leq 1$, bod leží uvnitř hlavní elipsy a patřičné šestiúhelníky jsou bezpodmínečně umístěny na mapě.
- Pro body ležící za hranicí elipsy ($d > 1$) je aplikován šum, který na základě prostorové pozice (X, Z), náhodného seedu a vhodného škálování viz rovnice 4.1 upravuje rozhodovací práh t viz rovnice 4.2. Tento práh rozšiřuje mapu o maximálně 15 % a umožňuje generovat šestiúhelníky i vně ideální elipsy. Pokud je d větší než šumem upravený práh t, šestiúhelník se nevygeneruje.

$$n = \texttt{noise}(seed, X \cdot scale, Z \cdot scale) \tag{4.1}$$

$$t = 1 + (1{,}15 - 1) \cdot n \tag{4.2}$$

Tímto přístupem se kombinuje striktní geometrická struktura (elipsa) s procedurální variabilitou (šum), což umožňuje dosáhnout rovnováhy mezi kontrolovaným tvarem mapy a přirozeným nepravidelným vzhledem. Konceptuální příklad výsledku vygenerované šestiúhelníkové mřížky je na obrázku 4.3.



Obrázek 4.3: Konceptuální příklad vygenerované šestiúhelníkové mřížky s kubickou indexací (q, r, s) šestiúhelníků.

4.1.2 Generování biomů

V této fázi je na vygenerovanou mapu aplikováno rozmístění biomů na základě šumové funkce. Speciální logika pak zajišťuje vznik vodních ploch podél okrajů mapy a jejich variabilní napojení na pevninu. Výsledkem je přiřazení biomu každému šestiúhelníku v pozemní vrstvě, přičemž šestiúhelníky ve vzdušné vrstvě jsou jednotně označeny biomem vzduch.

Proces mapování biomů

Každý šestiúhelník je indexován kubickou souřadnicí (q, r, s), kde osa $q = 0$ odpovídá světové ose $X = 0$. Pro zajištění symetrie herní mapy se biom generuje pouze pro šestiúhelníky splňující $q \leq 0$. Následně se určí i jeho zrcadlový protějšek s indexačními souřadnicemi $(-q, -r, -s)$ a tomuto šestiúhelníku se přiřadí stejný biom.

Určení biomu

Pro určení typu biomu je využita šumová funkce, která na základě světové pozice šestiúhelníku (X, Z) a náhodného seedu vrací hodnotu v intervalu $\langle 0, 1 \rangle$ (podobně jako v rovnici 4.1). Tato hodnota je následně zařazena do jednoho z předem definovaných pásem, z nichž každé odpovídá konkrétnímu biomu:

- $\langle d_p, h_p)$: pláně,
- $\langle d_l, h_l)$: les,
- $\langle d_k, h_k \rangle$: kopce.

Navíc je zavedena samostatná logika pro generování vodních ploch, která zohledňuje pozici vůči elipsovitému tvaru mapy. V oblasti s $Z < -1$ se na základě normalizované vzdálenosti od středu mapy spočítá hodnota p viz rovnice 4.3.

$$p = \texttt{clamp}\left(\sqrt{\left(\frac{X}{a}\right)^2 + \left(\frac{Z}{b}\right)^2}, 0, 1\right) \tag{4.3}$$

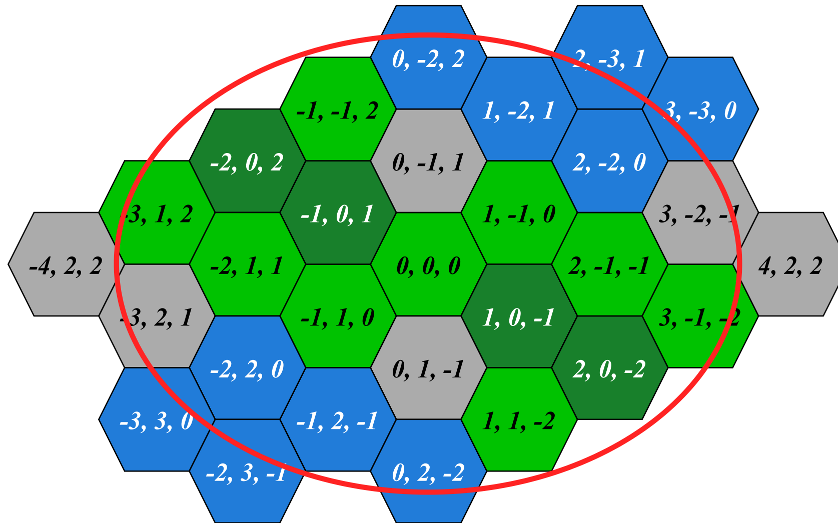
Tato hodnota reprezentuje pravděpodobnostní míru, jak blízko se bod nachází k okraji elipsy. Na jejím základě jsou vodní biomy určeny následovně:

- Pokud $p \geq 0{,}7$, je daný šestiúhelník bezpodmínečně označen jako vodní. Tím je zajištěna konzistentní přítomnost vodní plochy v okrajových částech mapy.
- Pro oblast $0{,}5 \leq p < 0{,}7$ je rozhodnutí ovlivněno šumem. Tato přechodová zóna má navázat na souš s určitou variabilitou, a proto je výpočet pravděpodobnosti upraven podle rovnice 4.4.

$$v = w_p \cdot p + w_n \cdot \texttt{noise}(seed, X \cdot scale, Z \cdot scale) \tag{4.4}$$

Kde w_p zastupuje váhu deterministické složky a w_n zastupuje váhu šumové složky. Přičemž platí, že $w_p + w_n = 1$ a zároveň $w_p, w_n \in \langle 0, 1 \rangle$, tedy obě váhy jsou kladné reálné hodnoty. Pokud výsledná hodnota v překročí prahovou hodnotu t, tj. $v > t$, je daný šestiúhelník označen jako vodní. Parametr t slouží k určení míry „přísnosti“ generování vody v přechodové zóně.

Konceptuální příklad výsledku vygenerovaných biomů na šestiúhelníkové mřížce je na obrázku 4.4.



Obrázek 4.4: Konceptuální příklad vygenerované herní mapy s kubickou indexací (q, r, s) šestiúhelníků. Světle zelená barva označuje pláně, tmavě zelená představuje les, šedá značí kopce a modrá označuje vodní biomy.

4.2 Přehled jednotek

Hráč má ve hře k dispozici několik typů bojových jednotek, z nichž každá plní specifickou roli v boji a strategii. Každý typ jednotky má své silné a slabé stránky, jak z hlediska boje,

tak pohybu. Jednotky jsou navrženy tak, aby fungovaly v rámci systému herní rovnováhy podobného principu *kámen-nůžky-papír*, kdy se určité typy efektivněji uplatňují proti jiným, nebo na specifických typech terénu. Cílem návrhu bylo podpořit hráčovo strategické rozhodování.

4.2.1 Typy jednotek

Ve hře je dostupných sedm typů jednotek:

- **Základna** – okolo základny je možné nasazovat vlastní jednotky. Její zničení znamená okamžitý konec hry.
- **Voják** – základní pozemní jednotka s nízkou odolností. Pokud se nachází v biomu *les*, stává se nezaměřitelnou pro vzdušné jednotky.
- **Džíp** – rychlá pozemní jednotka. Pokud se nachází v biomu *pláně*, získává schopnost způsobit lehké plošné poškození. V biomu *kopce* je však její pohyb zpomalen.
- **Tank** – pozemní jednotka s vysokou odolností a silným poškozením. Může útočit i na vzdušné jednotky. V biomu *kopce* je její pohyb zpomalen.
- **Loď** – jednotka schopná pohybu výhradně po vodních plochách. Disponuje dlouhým dosahem útoku a představuje důležitý prvek kontroly nad vzdušnou oblastí.
- **Helikoptéra** – rychlá vzdušná jednotka schopná útočit jak na pozemní, tak i na vzdušné cíle.
- **Stíhačka** – speciální vzdušná jednotka na jedno použití, která po nasazení provede silný plošný útok na pozemní cíle a následně odletí z bojiště.

4.2.2 Základní atributy jednotek

Každá jednotka je definována následujícími atributy:

- **Zdraví** – počet bodů, které jednotka musí ztratit, než je eliminována.
- **Poškození** – množství způsobeného poškození při útoku.
- **Dosah útoku** – maximální vzdálenost, na kterou jednotka může zaútočit.
- **Obnovovací čas útoku** – časový interval, který musí uplynout, než může jednotka znovu vystřelit.
- **Dosah pohybu** – maximální vzdálenost, na kterou se může jednotka přesunout.
- **Obnovovací čas pohybu** – časový interval, který musí uplynout, než může jednotka znovu provést pohyb.
- **Cena nákupu** – množství zdrojů potřebné pro nasazení jednotky.
- **Zisk za eliminaci** – odměna za zničení jednotky.
- **Vrstva pohybu** – definuje, zda se jednotka pohybuje po zemi nebo ve vzduchu.

- **Vrstva útoku** – specifikuje, jaké cíle může jednotka napadat (např. pouze pozemní, pouze vzdušné nebo oboje).
- **Přípustné biomy pro pohyb** – typy terénu, po kterých se jednotka může pohybovat.

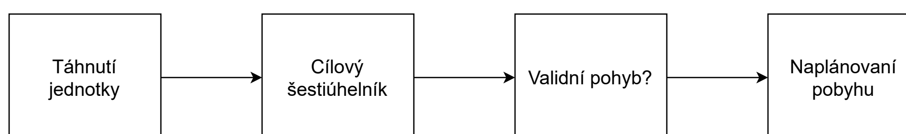
4.3 Akce jednotek

Pro většinu jednotek platí základní chování akcí jako **pohyb** viz podsekce 4.3.1, **útok** viz podsekce 4.3.2 a **nákup** viz podsekce 4.3.3. Specifické chování těchto akcí u daných jednotek je upřesněno v patřičných podsekcích.

4.3.1 Pohyb

Pohyb jednotky lze vykonat přetažením dané jednotky na cílový šestiúhelník pomocí kurzoru. Pokud je daný pohyb platný (např. nepřekračuje hodnotu atributu **dosah pohybu**, směřuje do přípustného terénu, není obsazený a není zarezervovaný), je cílový šestiúhelník zarezervován a jednotka zahájí přesun (viz názorné schéma na obrázku 4.5). Vizuálně je tento záměr indikován šipkou vedoucí k cíli a zároveň je spuštěn časovač podle atributu **obnovovací čas pohybu**. Progres časovače je reprezentován postupným naplňováním barvy ve směru šipky.

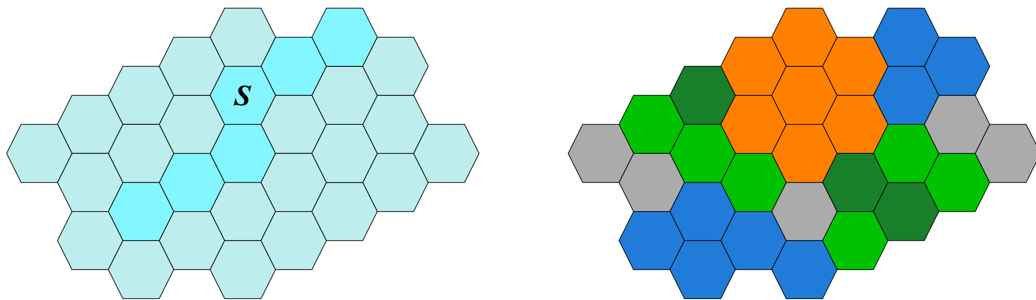
Rezervací cílového šestiúhelníku je zajištěno, že po dokončení pohybu bude jednotka skutečně umístěna do zamýšlené pozice (pokud mezitím nebude eliminována). Po naplánování pohybu jej již nelze stornovat ani upravit. Toto chování je záměrné a podporuje strategické rozhodování. Viditelná šipka totiž upozorňuje i protivníka na chystaný pohyb, čímž je podpořena možnost taktické reakce. Zároveň je tím zamezeno matoucím praktikám, jako je časté rušení pohybů nebo jejich využívání k dezinformaci protivníka. Po vypršení časovače je jednotka přesunuta na cílový šestiúhelník.



Obrázek 4.5: Abstraktní schéma zachycující situaci naplánování pohybu jednotky.

Pohyb pro stíhačku

Přetažením stíhačky na libovolný vzdušný šestiúhelník přejde stíhačka do režimu automatického postupu, ve kterém ji nelze dále ovládat. Podle zvoleného cílového šestiúhelníku se vytvoří trasa v podobě linky směřující až k okraji mapy. Stíhačka se bude po této trase pohybovat postupně podle časovače pohybu. Při zastavení na jednotlivých pozicích cílí svůj útok na jednotky nacházející se na pozemních sedmi šestiúhelnících viz obrázek 4.6.



(a) S indikuje stíhačku, která se postupně pohybuje po zvolené trase ke kraji mapy.

(b) Z aktuální pozice stíhačka cílí na jednotky nacházející se na oranžově vyznačených šestiúhelnících.

Obrázek 4.6: Situace zachycující nálet stíhačky, která se pohybuje ve vzdušné vrstvě (viz obr. 4.6a) a může útočit na jednotky na pozemní vrstvě (viz obr. 4.6b).

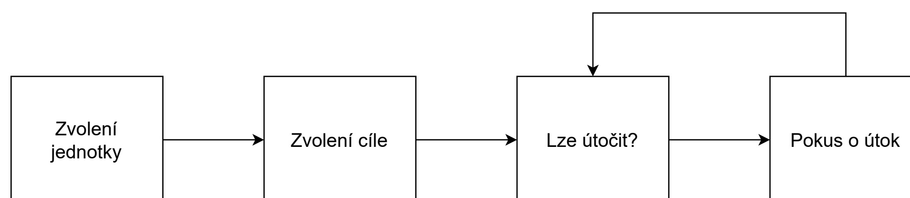
4.3.2 Útok

Mechanismus útoku je realizován výběrem jednotky hráčem, čímž je označena jako *vybraná*. Následné kliknutí na nepřátelskou jednotku ji označí jako *cílovou* a spustí akci útoku za předpokladu, že jsou splněny veškeré podmínky pro její provedení (viz názorné schéma na obrázku 4.7). Po aktivaci se spustí časovač dle **obnovovacího času útoku** dané jednotky. Pokud během časovače nedojde k přerušení z důvodu zneplatnění cíle, tak po jeho uplynutí je vystřelen projektil, který letí směrem k cíli a po zásahu mu odečte odpovídající hodnotu zdraví dle atributu **poškození** dané jednotky.

Útok probíhá opakovaně, dokud hráč nezmění cíl nebo se cíl nestane neplatným. Pokud je během probíhajícího útoku cíl zneplatněn (například opustí dosah dle atributu **dosah útoku**), je útok přerušen a časovač je okamžitě resetován. Změní-li hráč během útoku cílovou jednotku, systém se nejprve pokusí dokončit aktuální útok, a teprve poté přejde na nový cíl. Toto chování je záměrné a slouží jako herní mechanika, která od hráče vyžaduje určitou míru závazku – jakmile je útok zahájen, jednotka se jej pokusí dokončit, což podporuje strategické plánování a odpovědnost za učiněná rozhodnutí. Probíhající útok je přitom vizuálně indikován spojovacím paprskem mezi útočníkem a cílem.

Paprsek zároveň slouží jako vizuální indikátor průběhu útočného časovače, jeho intenzita totiž odráží zbývající čas do vystřelení. Pokud druhá jednotka útok opětuje, je paprsek rozdělen mezi obě strany, čímž je vizuálně znázorněn probíhající souboj. Každá polovina paprsku reprezentuje vlastní časovač dané jednotky.

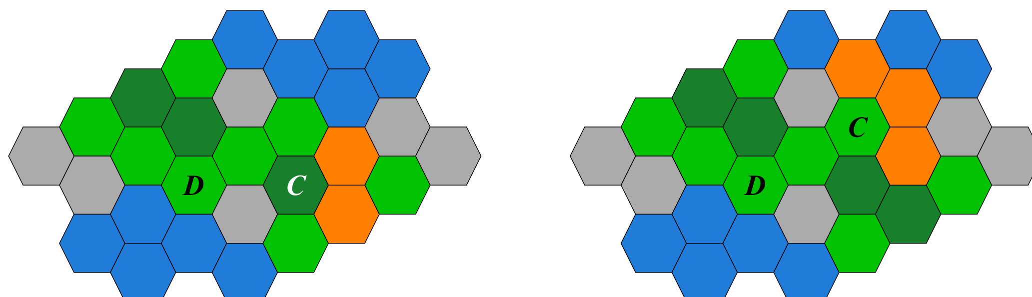
Hráč má také možnost označit jako cíl jednotku, která se aktuálně nenachází v dosahu útoku. V takovém případě se útok aktivuje automaticky, jakmile se cílová jednotka dostane do dosahu nebo se jinak stane validní. Tento mechanismus umožňuje plánovat útoky dopředu a zvyšuje plynulost ovládání. Cíl lze však kdykoli rovněž zrušit, například v situaci, kdy hráč předpokládá, že se v blízké době objeví prioritnější jednotka, a chce tak mít časovač připravený k okamžitému použití.



Obrázek 4.7: Abstraktní schéma zachycující situaci zvolení cíle k útoku.

Útok pro džíp

Mechanismus útoku džípu se liší pouze v tom, že pokud se tato jednotka nachází v biomu pláně, může při zásahu nepřátelské jednotky projektil roztříštit na několik menších střel. Tyto střely následně míří na další nepřátelské jednotky nacházející se za původním cílem ve směru střelby (viz obrázek 4.8).



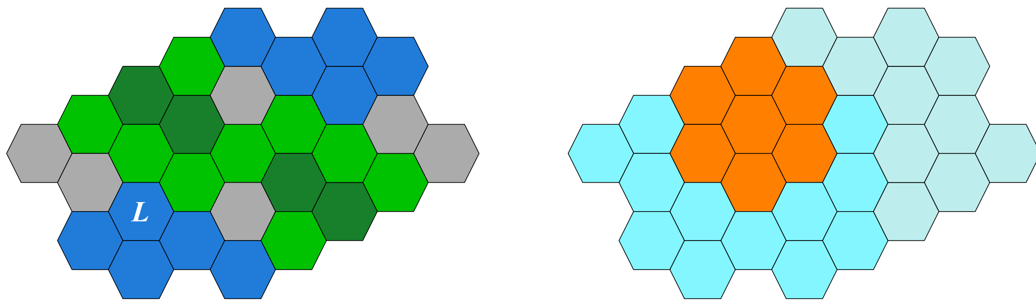
(a) Situace se dvěma dalšími zasaženými cíli. (b) Situace se třemi dalšími zasaženými cíli.

Obrázek 4.8: Schopnost džípu (vyznačen pomocí D) umožňuje plošný útok. Na základě úhlu střelby může plošná oblast zahrnovat buď dva šestiúhelníky (viz obr. 4.8a), nebo tři šestiúhelníky (viz obr. 4.8b). Kromě hlavního cíle (vyznačen pomocí C) tak může džíp zasáhnout až tři další jednotky, pokud se nacházejí na oranžově vyznačených šestiúhelnících.

Útok pro loď

U lodě se jako cíl nevybírá konkrétní jednotka, ale šestiúhelník na vzdušné vrstvě, který společně se svým okolím loď střeží viz obrázek 4.9. Jakmile se v tomto prostoru objeví vzdušná nepřátelská jednotka, loď automaticky zahájí útok třemi dávkovými výstřely.

S tímto je přidán speciální atribut lodě **obnovovací čas střežení**, který určuje, kdy je možné střeženou oblast změnit.



(a) *L* indikuje loď, která je umístěna na vodní šestiúhelník v pozemní vrstvě.

(b) Dle pozice lodi jsou tyrkysově vyznačeny šestiúhelníky vhodné pro výběr střežení a oranžově aktuálně střežené šestiúhelníky.

Obrázek 4.9: Loď se pohybuje výhradně po vodních biomech (viz obr. 4.9a) a útočí na vybranou oblast ve vzdušné vrstvě (viz obr. 4.9b). Dle dosahu útoku lze na vzdušné vrstvě vybrat skupinu sedmi šestiúhelníků, které loď střeží.

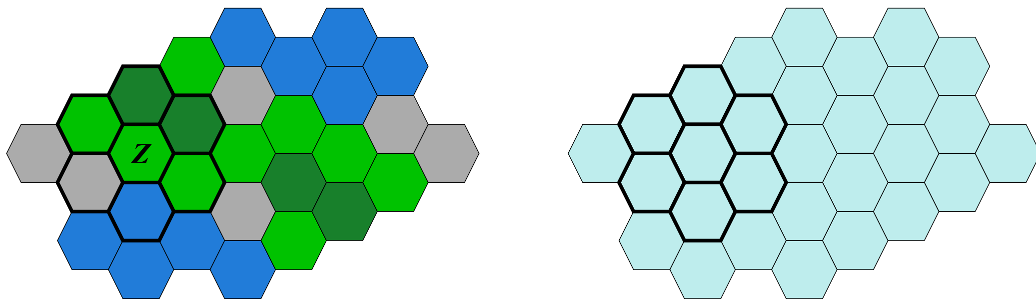
4.3.3 Nákup

Jednotky lze nakupovat prostřednictvím panelu umístěného ve spodní části obrazovky uprostřed. Panel zobrazuje aktuální částku peněz a dostupné ovladatelné jednotky (s výjimkou základny) s jejími ikonami spolu s jejich cenou viz nákres na obrázku 4.10.

Samotný nákup jednotky se provádí přetažením požadované jednotky z nákupního panelu na herní mapu, konkrétně na platný šestiúhelník v okolí základny viz obrázek 4.11. Platnost cílového šestiúhelníku závisí na tom, zda je biom vhodný pro pohyb dané jednotky (pozemní nebo vzdušné), zda není šestiúhelník jinou jednotkou obsazen nebo rezervován.

peníze	Voják cena	Džíp cena	Tank cena	Loď cena	Helikoptéra cena	Stíhačka cena

Obrázek 4.10: Nákupní panel zobrazující aktuální množství peněz a dostupné ovladatelné jednotky spolu s jejich cenami.



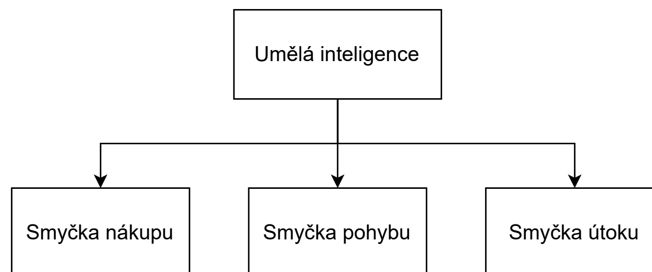
(a) *Z* indukuje základnu v jejímž okolí lze nasazovat jednotky pohybující se na pozemní vrstvě.

(b) Pro jednotky pohybující se na vzdušné vrstvě lze jednotku nasadit v okolí nad základnou.

Obrázek 4.11: Zvýrazněné šestiúhelníky tlustší čárou indikují možné šestiúhelníky pro nasazení jednotek na pozemní vrstvu (viz obr. 4.11a) a vzdušnou vrstvu (viz obr. 4.11b).

4.4 Umělá inteligence

Úkolem umělé inteligence je zajišťovat nákup jednotek, jejich pohyb a výběr cílů pro útok. Centrální řídicí prvek umělé inteligence na základě obnovovacích časovačů pohybu, útoku nebo specifického režimu střežení (v případě lodě) obsluhuje své jednotky prostřednictvím volby optimálních strategických akcí. V případě nákupu se jedná o periodické rozhodování, zda jednotku zakoupit, a pokud ano, jaký typ zvolit na základě aktuálního stavu hry. Tento proces lze chápat jako tři paralelní smyčky: nákup, pohyb a útok (viz obr. 4.12).



Obrázek 4.12: Umělá inteligence periodicky zvažuje nákup jednotek ve smyčce nákupu a v rámci smyček pohybu a útoku obsluhuje každou jednotku podle jejího časovače pro pohyb nebo útok.

4.4.1 Mapa vlivu

Pro podporu rozhodování umělé inteligence je na herní mapě vybudován systém *mapy vlivu*, jehož principem je ukládání hodnot relevantních pro rozhodování, které může umělá inteligence využívat při volbě akcí [8].

Na mapě jsou evidovány dva typy hodnot v rámci systému mapy vlivu (viz obr. 4.13):

- **Přítomnost hráče**: Každá jednotka přičítá hodnotu 1 šestiúhelníku, na kterém se nachází, a zmenšené hodnoty sousedním šestiúhelníkům v jejím bezprostředním

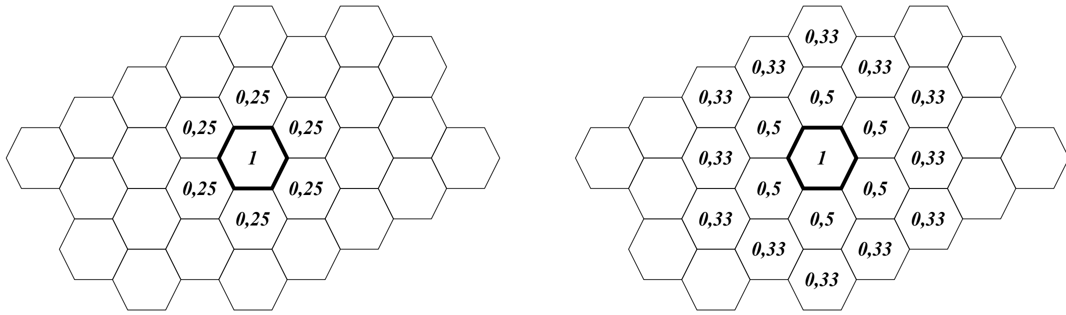
okolí (v rámci prvního prstence) na nákladě povolené vrstvě pohybu jednotky (viz obr. 4.13a).

- **Kontrola hráče**: Určuje míru strategického vlivu na základě dosahu útoku jednotky a povolené vrstvy útoku. Jednotka přičítá hodnotu 1 šestiúhelníku, na kterém stojí, a nižší hodnoty vzdálenějším šestiúhelníkům v jejím dosahu. Intenzita vlivu klesá podle vzdálenosti dle vztahu:

$$\text{přidaná hodnota} = \frac{1}{r+1},$$

  kde r je číslo prstence (viz obr. 4.13b).

Hodnoty jsou při přesunu patřičně aktualizovány a s klesající intenzitou napomáhají jednotkám volit bezpečnější směr postupu nebo identifikovat oblasti s vyšší koncentrací jednotek.



(a) Příklad rozložení hodnot přítomnosti hráče, kde jednotka stojí na šestiúhelníku s tučně zvýrazněným obrysem.

(b) Příklad rozložení hodnot kontroly hráče, kde jednotka stojí na šestiúhelníku s tučně zvýrazněným obrysem a má dosah útoku dva.

Obrázek 4.13: Ukázka rozložení hodnot pro přítomnost hráče (viz obr. 4.13a) a kontroly hráče (viz obr. 4.13b) v rámci mapy vlivu.

4.4.2 Analýza a rozhodovací mechanismus

Při nákupu je třeba zohlednit aktuální množství dostupných prostředků, složení a počet vlastněných jednotek, stejně jako složení a počet jednotek protivníka. Cílem je, aby nákupní smyčka reagovala na aktuální stav hry a pořizovala jednotky, které nejlépe odpovídají strategickým potřebám dané chvíle. Po rozhodnutí, že je v dané chvíli vhodné jednotku zakoupit, je na základě stávající situace ve hře a vzájemných vztahů mezi typy jednotek (např. loď je efektivní proti helikoptérám, helikoptéra proti džípům apod.) určena relativní vhodnost dostupných typů, přičemž je vybrána ta s nejvyšší prioritou, což zde lze považovat jako faktor **vhodnosti pro nákup**.

U akcí pohybu a útoku jednotky jde o získání dostupných pozic (šestiúhelníků) pro přesun nebo cílů (jednotek, nebo šestiúhelníků v případě lodi) pro útok. Tyto možnosti jsou následně ohodnoceny na základě několika předem definovaných faktorů specifických pro danou akci. Jelikož jsou jednotlivé faktory udržovány v normalizované podobě, lze je poté vzájemně kombinovat váženým součtem podle rovnice:

$$\text{skóre} = \sum_{i=1}^{n} F_i \cdot w_i, \tag{4.5}$$

kde F_i je daný faktor a w_i váha příslušná k tomuto faktoru. Po ohodnocení dostupných možností je vybrána ta s nejvyšším výsledným skóre.

V některých případech jednotek a jejich akcí je optimální rozhodnutí zřejmé i bez složitého hodnocení všech dostupných možností. Příkladem toho je jednotka stíhačka, která má svým chováním danou roli pro příležitostné použití. Její strategií je zakoupení v situaci, kdy se na mapě vyskytuje místo s vysokou koncentrací jednotek na pozemní vrstvě. Z toho plyne, kdy tuto jednotku zakoupit a jako cíl zvolit pozici s nejvyšším zahuštěním, což poskytuje systém mapy vlivu.

Jednotka loď má na základě svého chování roli efektivní protivzdušné obrany. Jelikož dokáže střežit oblast sedmi šestiúhelníků, měla by preferovat koncentrovaná místa v dosahu. Vzhledem k tomu, že se může pohybovat pouze po vodních biomech a nedokáže poškozovat pozemní jednotky včetně základny, je jejím cílem při pohybu odplout od své základny, aby uvolnila místo pro nasazení dalších lodí, a zároveň zůstat v její blízkosti kvůli obraně. Optimální vzdálenost od základny závisí na počtu aktivních lodí.

Jednotka džíp má na biomu pláně speciální schopnost, která jí umožňuje způsobovat plošné poškození, a proto by měla při výběru cíle zohledňovat zahuštění nepřátel. Jednotka voják naopak těží z výskytu v lese, kde je chráněna před útoky vzdušných jednotek. Biom typu kopce zpomaluje některé pozemní jednotky, a proto je v rámci pohybu jednotek třeba zohlednit preferované i nežádoucí typy terénu.

Z těchto a dalších důvodů plyne zavedení následujících faktorů ovlivňující rozhodování v pohybu a útoku jednotek:

Faktory pro volbu pohybu

- **příležitost možných cílů** – poměr počtu dostupných cílů z dané pozice vůči maximálnímu možnému počtu cílů,
- **preference biomu** – vhodnost terénu vzhledem k typu jednotky,
- **vzdálenost od nepřátelské základny** – pro posun směrem k cíli,
- **podpora spojenci** – blízkost vlastních jednotek,
- **nebezpečí** – intenzita kontroly hráče v oblasti,
- **rozprostření na vodní ploše** – určení optimální pozice pro loď.

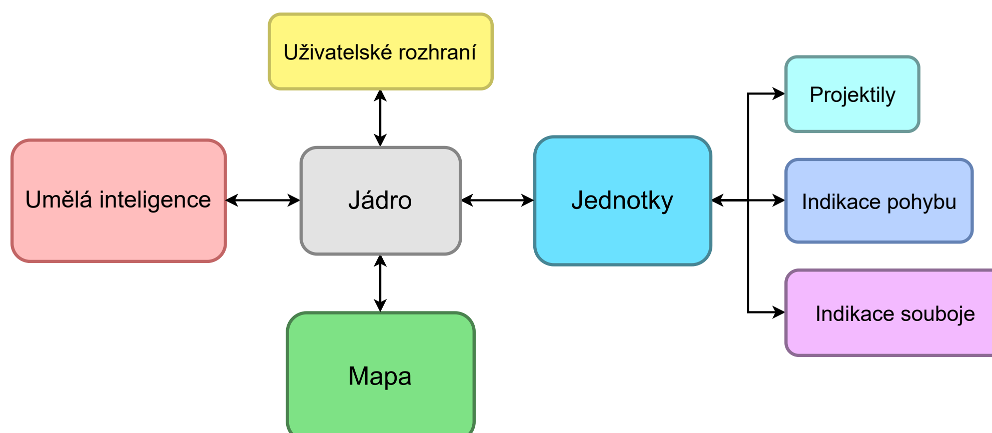
Faktory pro volbu útoku

- **vzdálenost od cíle** – blízkost jednotky k cíli,
- **zdraví cíle** – výhodnost útoku na oslabené cíle,
- **preference typu cíle** – vhodnost cíle vzhledem k útočníkovi,
- **zahuštění** – pro možnost zasažení více cílů.

Kapitola 5

Implementace

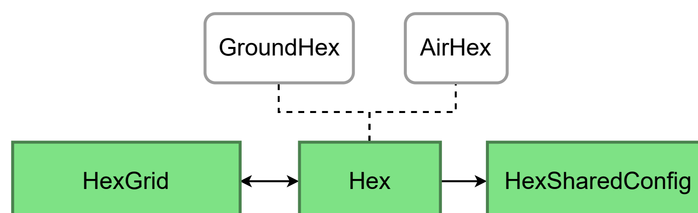
Tato kapitola se věnuje popisu implementace hry v prostředí Unity ve verzi 2022.3.44f1. Popis jednotlivých částí je provázán s přehledovým schématem na obrázku 5.1, jehož barevné rozlišení usnadňuje orientaci v následujícím popisu komponent.



Obrázek 5.1: Abstraktní schéma zachycující strukturu implementace, jehož barevné schéma doprovází popis dílčích komponent v rámci této kapitoly.

5.1 Herní mapa

Tato sekce se zaměřuje na komponenty, které zajišťují realizaci herní mapy (viz obr. 5.2), včetně jejího generování a správy *mapy vlivu*.



Obrázek 5.2: Komponenty zajišťující realizaci herní mapy.

5.1.1 Správce mapy

Třída `HexGrid` slouží jako centrální správce herní mapy. Zajišťuje generování šestiúhelníkové mřížky (viz sekce 5.1.3) i rozvržení biomů (viz sekce 5.1.4). Kromě toho poskytuje řadu metod pro přístup k datům a manipulaci s jednotlivými poli. Mapa je interně reprezentována dvěma oddělenými slovníky typu `Dictionary<Vector3Int, Hex>`, které mapují kubické souřadnice na objekty typu `Hex`. Jeden slovník slouží pro pozemní vrstvu a druhý pro vrstvu vzdušnou.

Vzdušná vrstva není ve výchozím stavu zobrazena. Automaticky se zobrazí při manipulaci se vzdušnou jednotkou nebo v jiných situacích, kdy je to potřeba. Po skončení interakce je opět automaticky skryta. Alternativně lze její zobrazení ručně přepínat pomocí pravého tlačítka myši.

Třída `HexGrid` poskytuje mimo jiné následující funkcionalitu:

- získání šestiúhelníku na základě souřadnic a vrstvy (pozemní nebo vzdušná),
- určení sousedních šestiúhelníků v daném rozsahu,
- výpočet vzdálenosti mezi dvěma šestiúhelníky v kubickém systému,
- získání prstence šestiúhelníků kolem zvoleného šestiúhelníku,
- výpočet spojnice mezi dvěma šestiúhelníky (linku šestiúhelníků).

V rámci podpory umělé inteligence třída spravuje *mapu vlivu*, která slouží k vyhodnocování přítomnosti a kontroly hráče nad jednotlivými oblastmi. Pro tyto účely uchovává mimo jiné:

- maximální vzdálenost mezi dvěma šestiúhelníky v kubickém systému na mapě (pro účely škálování a normalizace),
- maximální vzdálenost mezi dvěma vodními šestiúhelníky v kubickém systému v rámci jedné vodní oblasti (relevantní pro lodní jednotky),
- maximální hodnoty přítomnosti hráče (`maxPlayerPresence`) pro pozemní a vzdušnou vrstvu zvlášť,
- maximální hodnotu kontroly hráče nad mapou (`maxPlayerControl`),
- odkazy na konkrétní šestiúhelníky s těmito maximy (např. nejvíce zahuštěné místo).

Hodnoty maximální vzdálenosti mezi šestiúhelníky jsou vypočítány jednorázově při spuštění hry. Důvodem je, že mapa je procedurálně generovaná a její rozměry se mohou mezi jednotlivými hrami mírně lišit. Ze stejného důvodu se určuje i maximální vzdálenost mezi šestiúhelníky ve vodní oblasti, jejichž rozložení se může měnit navíc v závislosti na výsledku generování biomů.

Hodnoty související s přítomností a kontrolou se dynamicky přepočítávají při změnách stavu mapy. Tyto změny jsou zachyceny metodou jednotky `SetCurrentHex()`, která je volána při nasazení jednotky, jejím přesunu nebo odstranění z mapy.

5.1.2 Šestiúhelník

Třída `Hex` reprezentuje jednotlivá pole na herní mapě, na která lze umisťovat jednotky. Každý šestiúhelník uchovává informace o své pozici v kubickém souřadnicovém systému, biomu, typu vrstvy (pozemní nebo vzdušná viz obr. 5.3), stavu obsazení (volný, rezervován nebo obsazen) a případné jednotce, která se na něm aktuálně nachází.

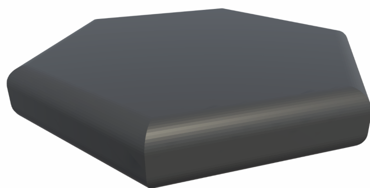
Zvýraznění a obrys šestiúhelníků je řešeno pomocí samostatných **Shader Graph** shaderů:

- `GoundHexSG` a `AirHexSG` – zajišťují vizuální zvýraznění šestiúhelníků dle jejich stavu. Aktivací příznaku `Highlight` se přidává emisní efekt, který indikuje možné pole pro přesuny nebo nasazení jednotek. U specifické jednotky loď také využito pro indikaci možných cílových polí pro střežení.

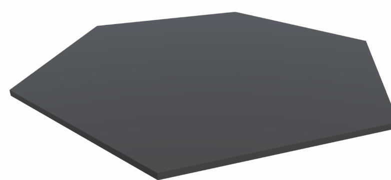
- `HexOutlineSG` – je aplikován na dodatečnou mesh, která je v patřičném tvaru pro zprostředkování obrysu a je položena na šestiúhelníku. Barva obrysu se mění v závislosti na aktuálním stavu: obrys skryt, přejetí myší. U specifických jednotek jako loď nebo stíhačka také: zacílení jednotkou nebo indikace že je cílové pole v dosahu. Použité barvy jsou definovány v konfigurační třídě `HexSharedConfig`.

Každý šestiúhelník také zpracovává herní logiku týkající se přítomnosti hráčských jednotek (`playerPresence`) a kontroly území (`playerControl`), které jsou využívány pro rozhodování umělé inteligence. Tyto hodnoty se šíří do okolních šestiúhelníků s klesající intenzitou v závislosti na vzdálenosti.

Přepočet těchto hodnot probíhá v metodě jednotky `SetCurrentHex()`, která je volána při nasazení, přesunu nebo odstranění jednotky z mapy, čímž je zajištěn konzistentní stav *mapy vlivu*. V jejím rámci je nejprve odečten vliv z předchozího šestiúhelníku a jeho okolí a následně přičten vliv k novému cílovému šestiúhelníku a jeho okolí. Rozsah ovlivněného okolí závisí na konkrétní veličině: v případě `playerPresence` se jedná pouze o sousední šestiúhelníky, zatímco u `playerControl` je ovlivněný prostor určován dosahem útoku dané jednotky.



(a) Model šestiúhelníku použitý pro pozemní vrstvu. Může mít přiřazen biom typu pláně, les, kopce nebo voda.



(b) Model šestiúhelníku použitý pro vzdušnou vrstvu. Implicitně má biom typu vzduch, což ve hře odpovídá bílému průhlednému vzhledu.

Obrázek 5.3: Ukázka modelů šestiúhelníků vytvořených v nástroji Blender[27].

[27] Blender: https://www.blender.org/

5.1.3 Generování šestiúhelníkové mřížky

Nutno podotknout, že v implementaci je použita šumová funkce `OpenSimplex2.Noise2()`[28]. V pseudokódech nacházejících se v této sekci 5.1.3 a v sekci 5.1.4 je však pro přehlednost abstrahována skutečnost, že v jednotlivých případech se funkce volá s mírně odlišným měřítkem a seedem.

Postup algoritmu 2 vychází ze sekce 4.1.1 a slouží k vytvoření šestiúhelníkové mřížky v záměrně nedokonalém tvaru elipsy. Prochází polovinu roviny a pro každou pozici vypočítává odpovídající světové souřadnice i souřadnice v kubickém systému. Následně ověřuje, zda daná pozice spadá do hlavní elipsy. Pokud se nachází za okrajem elipsy, šance na její zachování závisí na hodnotě šumu, která umožňuje mírné nepravidelnosti na kraji mapy.

Při akceptování šestiúhelníku jsou do scény umístěny jeho pozemní i vzdušné varianty. Aby se docílilo symetrie mapy, jsou šestiúhelníky navíc zrcadleny.

Algoritmus 2 Generování šestiúhelníkové mřížky s eliptickým a šumovým ořezem

```
 1: function GENERATEHEXGRID
 2:     halfHeight ← b_ellipse
 3:     halfWidth ← a_ellipse
 4:     for y ← −halfHeight to halfHeight do
 5:         for x ← −halfWidth to 0 do
 6:             xy ← (x, y)
 7:             worldPos ← CALCULATEHEXWORLDPOS(xy)
 8:             qrs ← OFFSETTOCUBE(xy)
 9:             distance ← ELLIPSENORMALIZEDDISTANCE(worldPos)
10:             if distance > 1 then              ▷ Okraj mapy s variabilním výskytem
11:                 gridNoise ← EVALUATEGRIDNOISE(worldPos)
12:                 threshold ← lerp(1, 1.15, gridNoise)
13:                 if distance > threshold then
14:                     continue
15:                 end if
16:             end if
17:             PLACEHEXES(qrs, worldPos)
18:             if x < 0 then
19:                 qrs_mirrored ← (−qrs.q, −qrs.r, −qrs.s)
20:                 worldPos_mirrored ← (−worldPos.x, worldPos.y, −worldPos.z)
21:                 PLACEHEXES(qrs_mirrored, worldPos_mirrored)
22:             end if
23:         end for
24:     end for
25: end function
```

5.1.4 Generování biomů

Algoritmy 3 a 4 vycházejí ze sekce 4.1.2 a zajišťují přiřazení biomu jednotlivým šestiúhelníkům herní mapy.

Algoritmus 3 určuje konkrétní typ biomu na základě šumové funkce a pozice šestiúhelníku. Biomy pláně, les nebo kopce jsou stanoveny podle hodnoty terénního šumu. V oblasti

[28] OpenSimplex: https://github.com/KdotJPG/OpenSimplex2/blob/master/csharp/OpenSimplex2.cs

přípustné pro výskyt vody se dále zvažuje možnost přítomnosti vodního biomu. Tato oblast je rozdělena na přechodové pásmo, kde rozhoduje kombinace vzdálenosti od středu a dodatečného šumu a na vzdálenější pásmo, kde se voda vyskytuje s jistotou.

Algoritmus 4 pak aplikuje výběr biomu na levou polovinu mapy a následně jej zrcadlí na druhou polovinu mapy, čímž je zajištěno symetrické rozvržení biomů.

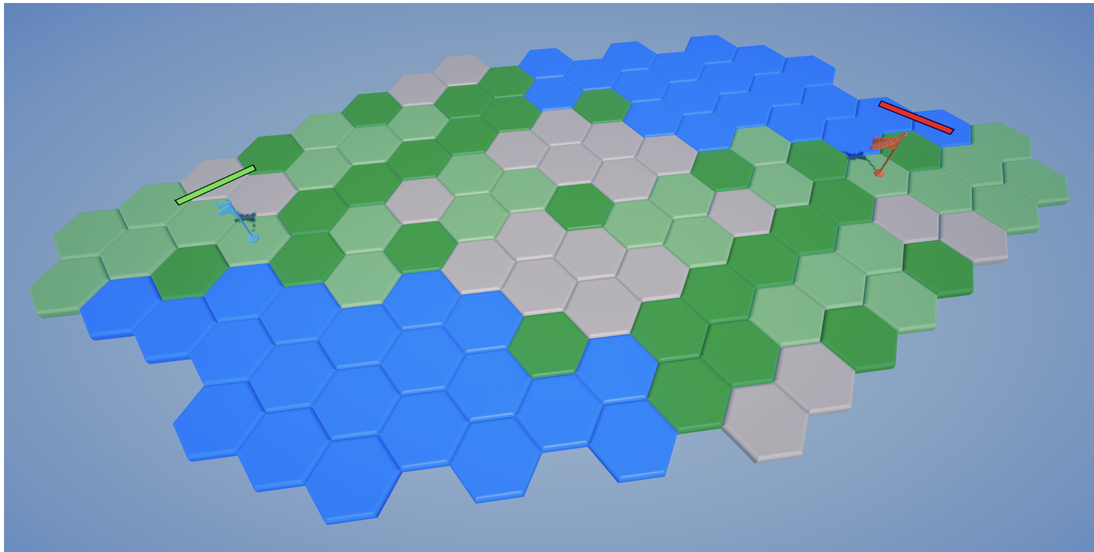
Na obrázku 5.4 je možné vidět vygenerovanou herní mapu pro jednu konkrétní instanci hry.

Algoritmus 3 Generování typu biomu podle šumu a pozice

```
 1: function GenerateBiome(worldPos)
 2:     terrainNoise ← EvaluateTerrainNoise(worldPos)
 3:     if terrainNoise < plainsThreshold then
 4:         biomeType ← Plains
 5:     else if terrainNoise < forestThreshold then
 6:         biomeType ← Forest
 7:     else
 8:         biomeType ← Hills
 9:     end if
10:     if worldPos.z < −1 then                ▷ Oblast mapy přípustná pro přítomnost vody
11:         prob ← EllipseNormalizedDistance(worldPos)
12:         if 0.5 < prob < 0.7 then           ▷ Pásmo přechodu mezi souší a vodou
13:             waterNoise ← EvaluateWaterNoise(worldPos)
14:             if prob · 0.65 + waterNoise · 0.35 > 0.6 then
15:                 biomeType ← Water
16:             end if
17:         else if prob ≥ 0.7 then            ▷ Vzdálené pásmo
18:             biomeType ← Water
19:         end if
20:     end if
21:     return biomeType
22: end function
```

Algoritmus 4 Generování biomů zrcadleně přes středovou osu

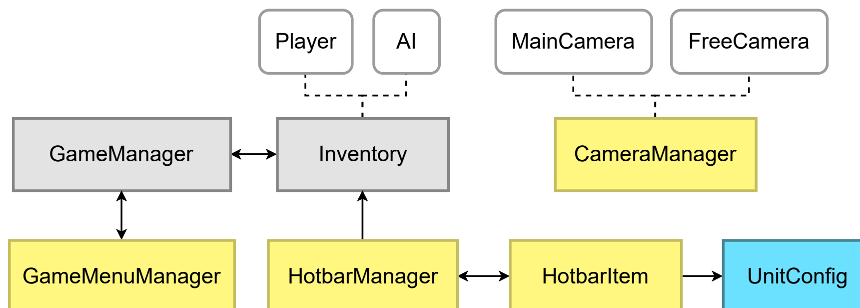
```
 1: function GenerateBiomes
 2:     for all hex in hexMap do
 3:         if hex.qrs.q > 0 then
 4:             continue                       ▷ Zpracování pouze levé části mapy
 5:         end if
 6:         leftHex ← hex
 7:         worldPos ← CalculateHexWorldPos(leftHex)
 8:         biomeType ← GenerateBiome(worldPos)
 9:         qrs_mirrored ← (−qrs.s, −qrs.r, −qrs.s)
10:         rightHex ← GetHex(qrs_mirrored, Ground)
11:         SetBiome(leftHex, biomeType)
12:         SetBiome(rightHex, biomeType)
13:     end for
14: end function
```



Obrázek 5.4: Příklad vygenerované herní mapy na začátku hry se základnami.

5.2 Správa herního jádra a uživatelského rozhraní

Tato sekce popisuje komponenty zajišťující správu herního průběhu, stav jednotlivých stran a významných prvků pro interakci uživatele s hrou viz obr. 5.5.



Obrázek 5.5: Komponenty zajišťující správu herního jádra a významné prvky interakce s hráčem.

5.2.1 Správa stran a stavu hry

Ve scéně se nacházejí objekty zastupující obě strany konfliktu nazvané: `Player` (hráč) a `AI` (protivník). Oba objekty obsahují inventář `Inventory` se stejnou výchozí částkou a s inkrementálním nárůstem peněz za definovanou časovou periodu.

Skript `GameManager` uchovává reference na tyto inventáře a na aktuálně dostupné ovladatelné jednotky obou stran. Zároveň průběžně eviduje statistiky o aktuálně vlastněných jednotkách, které slouží jako podklad pro rozhodování umělé inteligence.

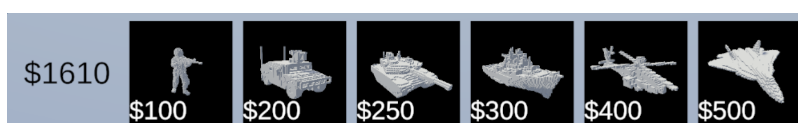
`GameManager` také zajišťuje odměnění příslušné strany při eliminaci jednotky nebo při ukončení hry, pokud byla eliminována základna.

5.2.2 Nákupní panel

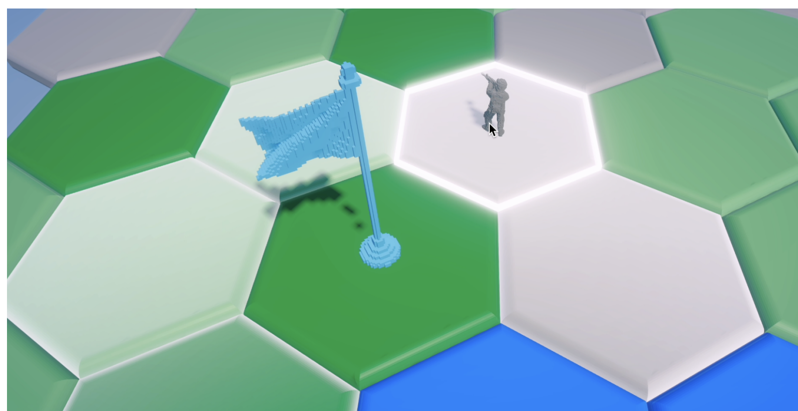
Součástí uživatelského rozhraní je nákupní panel (viz obr. 5.6), který umožňuje hráči nasazení nových jednotek na mapu technikou *drag-and-drop* (viz obr. 5.7). Každá položka v panelu představuje jednu konfiguraci jednotky a zobrazuje její ikonu a cenu.

Skript `HotbarManager` slouží jako centrální správce nákupního panelu. Uchovává informaci o aktuálně vybrané položce a zajišťuje synchronizaci s inventářem jako odečtení prostředků a aktualizace zobrazené částky.

Každá položka v panelu je reprezentována třídou `HotbarItem`, která zajišťuje vizuální zpětnou vazbu během interakce s hráčem. Při stisknutí položky dojde k zobrazení tzv. stínové jednotky, která se pohybuje s kurzorem myši a ukazuje, kam bude jednotka případně umístěna. Platná místa nasazení jsou vizualizována pomocí zvýrazněných šestiúhelníků. Pokud hráč uvolní kurzor nad validním polem, jednotka je zakoupena a nasazena.



Obrázek 5.6: Nákupní panel ve hře.



Obrázek 5.7: Požadovanou jednotku lze z nákupního panelu přetáhnout kurzorem na zvolený šestiúhelník. Dostupné pozice jsou zvýrazněny a aktuálně označený šestiúhelník je doplněn o obrys.

5.2.3 Kamerový systém

Kamerový systém je realizován pomocí třídy `CameraManager`, která slouží jako centrální správce dvou režimů kamery: hlavní (statické) a volné (volně ovladatelné).

Ve výchozím stavu je aktivní hlavní kamera, která zachycuje celou herní mapu z pevně dané perspektivy umístěné v neutrálním bodě vzhledem ke stran konfliktu. Hráč má možnost kdykoliv přepnout do režimu volné kamery a zpět stisknutím klávesy `C`. V tomto režimu lze kamerou volně pohybovat nad herní plochou pomocí kláves `W`, `A`, `S`, `D`. Otáčení kamery je možné podržením prostředního tlačítka myši a tažením kurzoru.

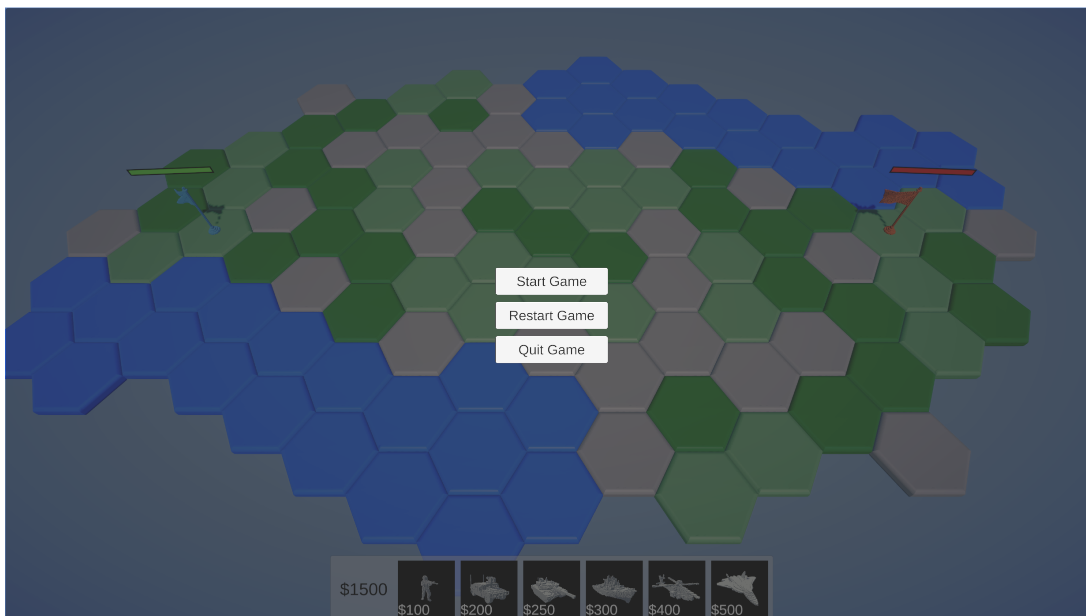
Pomocí metody `GetCurrentCamera()` mohou ostatní komponenty získat aktuálně aktivní kameru, což je využíváno při výpočtu světové pozice kurzoru a interakci s objekty ve hře.

5.2.4 Herní menu

Herní menu je řízeno třídou `GameMenuManager`, která spravuje tři obrazovky: úvodní menu (viz obr. 5.8), pauzovací menu a závěrečné menu po skončení hry.

Při spuštění hry je aktivováno úvodní menu a herní čas je pozastaven. Po stisknutí tlačítka „Start Game“ je hra spuštěna a čas pokračuje. Během hry může hráč kdykoliv stisknutím klávesy `Escape` přepínat mezi aktivním hraním a pauzovacím režimem. Po skončení hry je vyvoláno závěrečné menu.

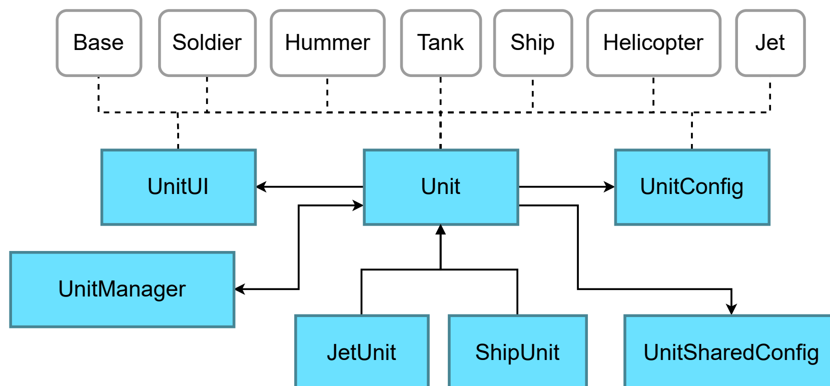
Tlačítko pro restartování hry je dostupné ve všech obrazovkách včetně úvodní. Během zobrazení menu zůstává herní scéna viditelná na pozadí se ztmavením, takže hráč vidí aktuálně vygenerovanou mapu. Pomocí opakovaného restartování lze jednoduše procházet různě vygenerované mapy bez nutnosti opouštět hru.



Obrázek 5.8: Obrazovka na začátku hry.

5.3 Architektura a životní cyklus jednotek

Tato sekce shrnuje architekturu tříd, které se podílejí na zajištění chování, vizualizace a správu herních jednotek. Jednotky mají vlastní konfigurační soubory, specifické typy a mají k dispozici centrálního správce. Schéma na obrázku 5.9 znázorňuje přehled hlavních komponent a jejich vztahů.

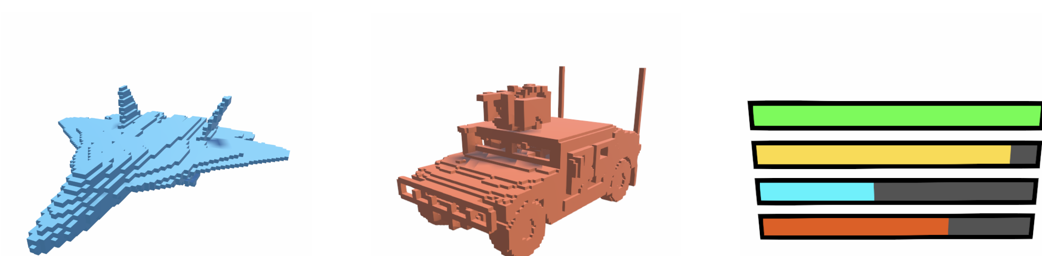


Obrázek 5.9: Struktura komponent souvisejících s jednotkami, jejich konfigurací, uživatelským rozhraním a správou.

5.3.1 Struktura jednotky

Všechny jednotky vycházejí ze základní prefaby jednotky, ke které je v rámci inicializace dle typu jednotky přidán patřičný skript a model, tedy jsou tvořeny dynamicky. Tato prefab má ve výchozím stavu připojen pouze skript `UnitUI` a obsahuje dvě další vnořené prefaby (viz obr. 5.10):

- `StatBars` – obsahuje čtyři separátní panely s objekty `Image`, které dohromady tvoří ukazatel aktuální statistiky (zdravý, časovač útoku, časovač pohybu a časovač střežení). Pomocí nastavení `Image Type` na `Filled` lze poté dynamicky upravovat zobrazovaný stav (viz obr. 5.10c).
- `UnitModels` – po inicializaci jednotky obsahuje dva modely: voxelový model dané jednotky, které byly vytvořeny převedením existujících modelů do voxelové podoby pomocí nástroje MagicaVoxel[29] (viz obr. 5.10a a 5.10b) a hladký stejný model sloužící pro obrys jednotky, na který je aplikován shader `UnitOutline` viz sekce 5.3.8.



(a) Voxelový model spojenecké jednotky typu stíhačka.

(b) Voxelový model nepřátelské jednotky typu džíp.

(c) Ukazatele aktuálních statistik jednotky.

Obrázek 5.10: Ukázka voxelových modelů jednotek a ukazatelů statistik.

[29]MagicaVoxel: https://ephtracy.github.io/

Skript `UnitUI` obsahuje referenci na `UnitModels`, která je později využita pro otáčení jednotky během vykonávání akcí, a dále na objekt `StatBars` obsahující ukazatele statistik. Při inicializaci skript nastaví ukazatele podle typu jednotky: skryje nepotřebné ukazatele, protože časovač pohybu je vhodnější prezentovat přímo v rámci ukazatele pohybu (viz sekce 5.4.1). Časovač útoku je efektivněji řešen pomocí spojového paprsku během souboje (viz sekce 5.5.2). Dále skript skryje ukazatel střežení u jednotek, které nejsou typu loď, neboť tuto vlastnost nevyužívají. U nepřátelských jednotek je barva ukazatele životů nastavena na červenou. Skript zároveň zajišťuje plynulé natáčení ukazatelů statistik směrem ke kameře.

5.3.2 Konfigurace jednotek

Pro konfiguraci jednotlivých jednotek byla vytvořena třída `UnitConfig` odvozená od třídy `ScriptableObject`, která obsahuje atributy zmíněné v sekci 4.2.2 (jako např. zdraví, obnovovací časy atd.) a navíc:

- typ jednotky,
- model jednotky,
- hladký model jednotky sloužící pro obrys,
- prefab projektilu, který daná jednotka používá.

Tímto způsobem poté lze v editoru Unity snadno vytvářet konfigurace pro jednotlivé typy jednotek.

Pro správu všech dostupných konfigurací slouží třída `UnitConfigDatabase`. Tato databáze obsahuje seznam všech instancí `UnitConfig` a poskytuje metodu `GetConfig()`, která umožňuje vyhledání konkrétní konfigurace na základě typu jednotky.

`UnitSharedConfig` slouží k definici sdílených vizuálních vlastností jednotek. Uchovává použitý materiál pro modely jednotek a barvy pro jednotlivé týmy, které jsou následně aplikovány na modely jednotek ve hře. Dále obsahuje všechny barevné varianty využívané k zobrazení obrysů jednotek v daných stavech.

Součástí konfigurace je také speciální materiál pro tzv. stínový model jednotky, který slouží jako vizuální náhled při nákupu nebo plánování pohybu jednotky. Tento náhled využívá hlavní model jednotky definovaný v její specifické konfiguraci (`UnitConfig`), ale je zobrazen s neutrálním zbarvením.

5.3.3 Skripty definující chování jednotek

Při nasazení jednotky v rámci metody `SpawnUnit()` je na instanci jednotky připojen jeden ze skriptů `Unit`, `ShipUnit` nebo `JetUnit`, přičemž `ShipUnit` a `JetUnit` dědí ze základní třídy `Unit`. Tato struktura umožňuje lodím a stíhačkám přepsat vybrané části základního chování definovaného ve třídě `Unit`.

Specifické chování ostatních jednotek, například džípu, je řešeno prostřednictvím specializovaných prefabů projektilů viz sekce 5.5.1, které jsou definovány v konfiguračním skriptu jednotky `UnitConfig`. Namísto obecného skriptu `Projectile`, připojeného k výchozí prefabě střely, používá džíp skripty `HummerProjectile` a `HummerSplashProjectile`, jež implementují specifické vlastnosti jeho střelby.

Loď rovněž využívá speciální skript projektilu `ShipProjectile`. Ten však pouze upravuje styl trajektorie střely tak, že projektil nejprve vystoupá do určité výšky a teprve poté zamíří k cíli.

Speciální vlastnost vojáka je realizována přímo v metodě jednotky `IsTargetInRange()`, která zároveň zajišťuje, že voják nemůže být v biomu les zaměřen vzdušnými jednotkami.

5.3.4 Správce jednotek

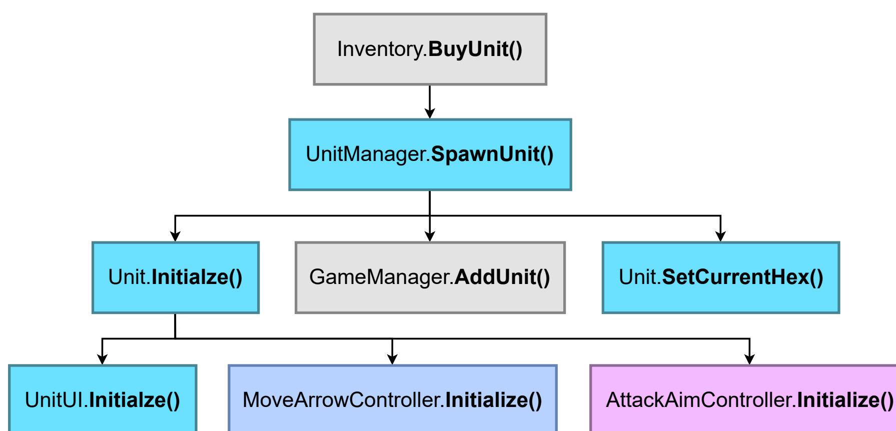
Skript `UnitManager` slouží jako centrální komponenta pro správu jednotek ve hře. Uchovává referenci na databázi konfigurací jednotek (`UnitConfigDatabase`) a základní prefab, ze kterého jsou instance jednotek vytvářeny. Dále zajišťuje samotné nasazení a odstranění jednotek na mapě a udržuje informaci o aktuálně vybrané jednotce hráčem. Jednotky jsou vytvořeny na základě svého typu a při inicializaci obdrží příslušnou konfiguraci a počáteční pozici na mapě.

5.3.5 Inicializace jednotky

Po nasazení je jednotka ihned inicializována pomocí metody `Initialize()`, která získá příslušný konfigurační skript `UnitConfig`. Na základě této konfigurace jsou instancovány příslušné modely: hlavní model jednotky a jeho hladká varianta, které jsou následně umístěny jako podobjekty do objektu `UnitModels`. Současně je na hlavní model aplikována barva odpovídající straně jednotky, přičemž barvy a materiál jsou dostupné ve sdíleném konfiguračním skriptu `UnitSharedConfig`. Modely jsou zobrazeny pomocí komponenty `SortingGroup`, přičemž hlavní model je vykreslován nad hladkým modelem.

Dále jsou na základě konfigurace nastaveny základní atributy jednotky, jako je maximální zdraví, doba obnovy útoku a doba obnovy pohybu.

Zároveň dochází k inicializaci komponenty `UnitUI` a k vytvoření instancí řídicích tříd `MoveArrowController` a `AttackAimController`, které zajišťují indikaci pohybu a probíhajících soubojů.



Obrázek 5.11: Významné akce zprostředkovávající vnik jednotky.

5.3.6 Vznik jednotek

Základny jsou nasazeny přímo v rámci generování mapy. Ostatní ovladatelné jednotky mohou být do hry nasazeny prostřednictvím nákupu a volbou platného šestiúhelníku pro nasazení.

Po výběru místa je jednotka inicializována (viz sekce 5.3.5), přidána do seznamu aktivních jednotek a následně je jí přiřazen šestiúhelník, na kterém se nově nachází. Tento krok

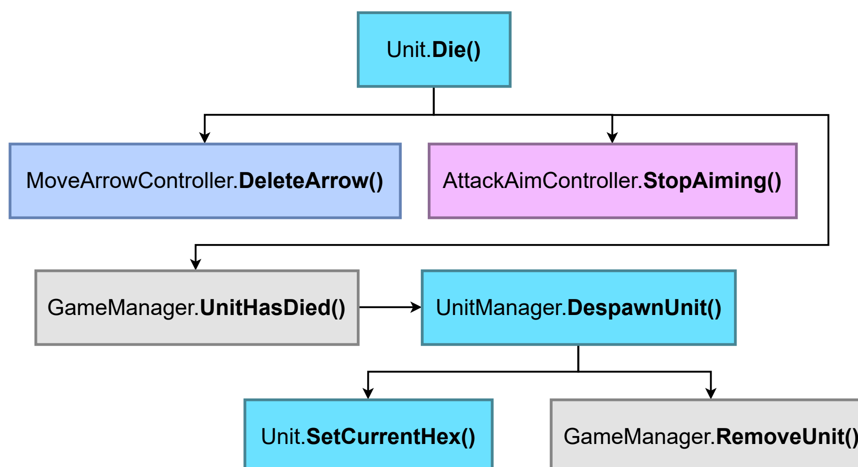
je důležitý mimo jiné kvůli správné evidenci vlivu na mapě, jak je popsáno v sekci 5.1.2. Průběh této akce je schematicky znázorněn na obrázku 5.11.

Před samotným nasazením probíhá výpočet platných šestiúhelníků, na které lze jednotku umístit. Tyto možnosti jsou v uživatelském rozhraní vizuálně zvýrazněny (viz obr. 5.7), přičemž je brán ohled na to, že během výběru hráče se šestiúhelník může stát rezervovaným nebo obsazeným. V kontextu umělé inteligence tyto možnosti slouží jako podklad pro rozhodování.

5.3.7 Zánik jednotek

Při eliminaci jednotky dochází nejprve k odstranění jejích vizuálních indikátorů a následně je o události informován správce hry `GameManager`. Ten provede přidělení odměny příslušné straně, případně ukončí hru, pokud byla zničena základna.

Jednotka je odstraněna ze seznamu aktivních jednotek a v metodě `SetCurrentHex()` je její pozice nastavena na `null`, čímž je její zánik zohledněn v mapě vlivu. Tato situace je schematicky znázorněna na obrázku 5.12.



Obrázek 5.12: Významné akce zprostředkovávající zánik jednotky.

5.3.8 Obrys jednotky

Obrys jednotky je využíván v uživatelském rozhraní k indikaci jejího aktuálního stavu, zejména při výběru jednotky nebo při zaměřování cílů.

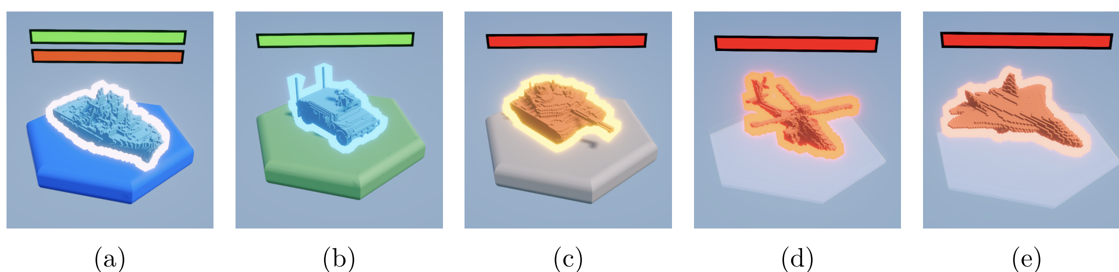
Realizace obrysu je založena na vlastním **Shader Graph** `UnitOutlineSG`, který je aplikován na speciálně připravený hladký model jednotky. Tento hladký model je vytvořen úpravou normál s použitím vyhlazovacího úhlu 180°, což zajišťuje plynulé propojení geometrie napříč celým modelem. Tato úprava je nezbytná, protože hlavní modely jednotek jsou voxelové (tedy s ostrými hranami), na kterých by efekt kontinuálního obrysu nebyl vizuálně funkční.

Samotný `UnitOutlineSG` vytváří efekt obrysu tak, že posouvá vertexy modelu směrem ven podle jejich normalizovaných normál o zadanou tloušťku `Thickness`. Výsledný obrys je vykreslen pomocí definované barvy `BaseColor`.

Pro správné vrstvení při vykreslování je využita komponenta `SortingGroup`, která zajišťuje, že hladký model obrysu je vždy vykreslen pod hlavním modelem jednotky.

Definovány jsou následující stavy obrysu, které mají definovaný vlastní odstín barvy (viz obr. 5.13) ve sdíleném konfiguračním skriptu `UnitSharedConfig`:

- `Hidden` – Obrys jednotky je skryt a není vizuálně zobrazen.
- `Selected` – Jednotka je aktuálně vybrána hráčem (viz obr. 5.13b).
- `Attacked` – Jednotka je nastavena jako cíl útoku vybrané jednotky (viz obr. 5.13d).
- `Hovered` – Jednotka je právě pod kurzorem hráče (viz obr. 5.13a).
- `HoveredSelected` – Jednotka je vybrána a zároveň pod kurzorem, přičemž je zobrazen odlišný obrys než při běžném najetí kurzorem.
- `HoveredAttacked` – Jednotka je cílem útoku a zároveň pod kurzorem, s odlišným zvýrazněním oproti běžnému najetí kurzorem.
- `AttackedOutOfRange` – Jednotka je nastavena jako cíl útoku, avšak momentálně se nachází mimo dosah (viz obr. 5.13e).
- `HoveredInRange` – Jednotka je v dosahu a vhodná pro útok, přičemž je pod kurzorem hráče (viz obr. 5.13c).



(a) (b) (c) (d) (e)

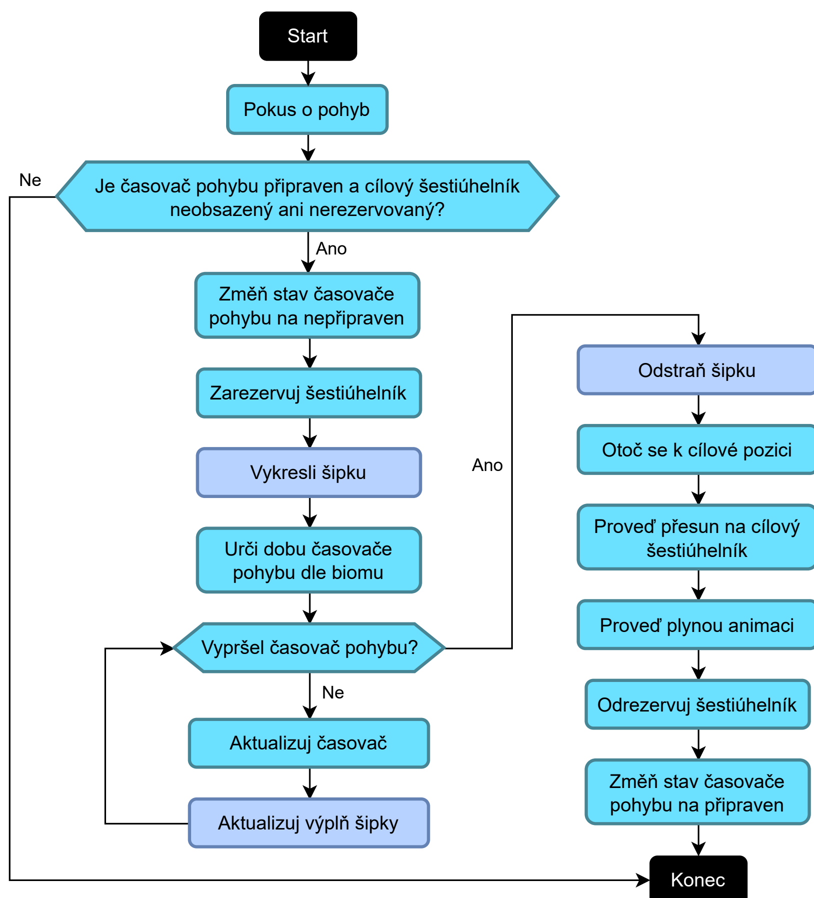
Obrázek 5.13: Ukázka vybraných variant obrysů jednotek.

5.4 Pohyb jednotky

Vstupním bodem pro akci pohybu jednotky je metoda `TryMove()`, která při předání cílového šestiúhelníku (instance třídy `Hex`) spustí korutinu `MoveCoroutine()`, pokud jsou splněny podmínky pro provedení pohybu. V rámci této korutiny je jednotce nastaven stav časovače pohybu na nepřipravený, čímž je blokována možnost zahájit další pohyb. Následně je cílový šestiúhelník rezervován a vykreslena šipka indikující směr pohybu. Během odpočtu časovače pohybu je postupně aktualizována výplň šipky pomocí barvy definované ve sdíleném konfiguračním skriptu `UnitSharedConfig`.

Po vypršení časovače je indikační šipka odstraněna a je naplánována rotace modelu jednotky směrem k cílové pozici prostřednictvím metody `Rotate()`. Po dokončení rotace je provedena samotná změna pozice pomocí metody `Move()`, která nastaví aktuální šestiúhelník jednotky pomocí `SetCurrentHex()`. Zároveň je spuštěna korutina `MoveSmoothly()`, která zajišťuje plynulý vizuální přesun modelu na cílovou pozici. Cílový šestiúhelník je odrezervován a stav časovače pohybu jednotky je opět nastaven na připravený. Průběh této akce je znázorněn ve vývojovém diagramu na obrázku 5.14.

Před přesunem typicky probíhá výpočet platných šestiúhelníků, na které lze jednotku umístit obdobně jako u nákupu jednotky. Tyto možnosti jsou v uživatelském rozhraní opět vizuálně zvýrazněny (viz obr. 5.15) a v kontextu umělé inteligence slouží jako podklad pro rozhodování.

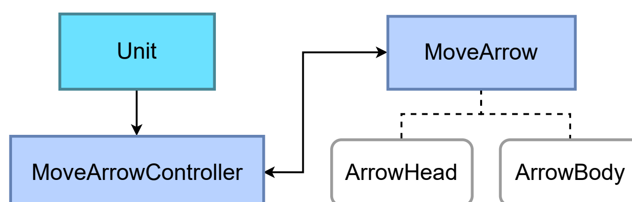


Obrázek 5.14: Vývojový diagram zachycující sekvenci akcí při pokusu jednotky o pohyb na cílový šestiúhelník.

Obrázek 5.15: Příklad výběru cílového šestiúhelníku pro pohyb jednotky typu helikoptéra ve vzdušné vrstvě pomocí techniky *drag-and-drop*.

5.4.1 Indikace pohybu jednotky

Tato sekce popisuje realizaci indikační šipky při pohybu jednotek. Komponenty, které se na této funkcionalitě podílejí, je možné vidět na obrázku 5.16.



Obrázek 5.16: Komponenty zajišťující vykreslení indikační šipky pro pohyb jednotky.

Každá jednotka disponuje vlastním ovládacím prvkem pro indikaci pohybu, reprezentovaným instancí třídy `MoveArrowController`. Tento prvek je při inicializaci nastaven s odpovídající barvou dle příslušnosti ke straně a poskytuje abstrakci pro práci s indikační šipkou prostřednictvím metod k vykreslení šipky, aktualizace míry výplně šipky a odstranění aktuálně vykreslené šipky.

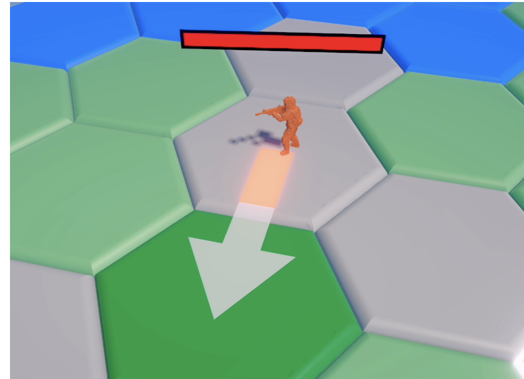
Samotná šipka indikující pohyb jednotky je realizována pomocí komponenty `MoveArrow`, která je složena ze dvou samostatných částí: těla šipky a hlavy šipky.

V metodě `Initialize()` je na základě zdrojového a cílového šestiúhelníku (instance třídy `Hex`) instancováno tělo šipky v odpovídajícím směru. Tělo šipky je automaticky naškálováno tak, aby sahalo až k cílovému bodu s mírným odsazením. Pivot prefaby hlavy šipky je umístěn v základně trojúhelníku reprezentujícího hlavu, což umožňuje přesné umístění hlavy šipky na konec těla. Tento postup zajišťuje plně modulární použití šipky i na libovolné vzdálenosti mezi šestiúhelníky.

Shader `MoveArrowSG`, vytvořený pomocí **Shader Graphu**, zajišťuje vizuální efekt postupné výplně šipky. Na základě světových souřadnic počátku `Start` a konce `End` šipky se určí směr, podél kterého je následně pomocí skalárního součinu vypočtena projekční vzdálenost aktuálního fragmentu podél šipky. Vydělením této hodnoty délkou šipky je získána relativní vzdálenost na šipce, která je porovnána s parametrem `Fill`, který určuje aktuální

míru vyplnění. Poté lze určit část šipky, která má být vyplněna barvou (dle parametru `Color`). Indikační šipky pohybu je možné vidět na obrázku 5.17.

(a) Indikační šipka pohybu pro spojeneckou jednotku.

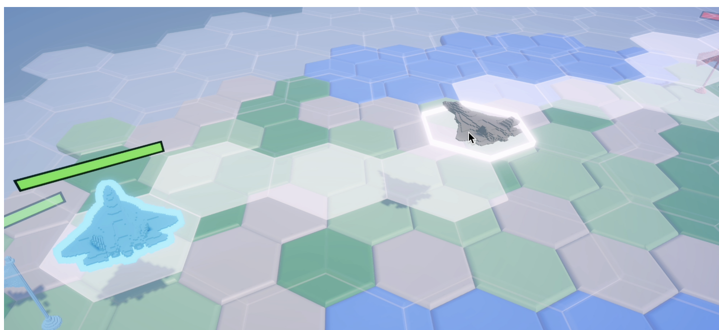


(b) Indikační šipka pohybu pro nepřátelskou jednotku.

Obrázek 5.17: Indikační šipka pohybu znázorňující směr k cílovému šestiúhelníku, na který se jednotka přesune po vypršení časovače pohybu.

5.4.2 Pohyb jednotky typu stíhačka

Jednotka s připojeným skriptem `JetUnit` přepisuje metodu `TryMove()`, čímž realizuje specifické chování stíhačky. Pohyb této jednotky probíhá jako sekvenční přesun po šestiúhelnících tvořících přímku směrem k okraji mapy. Každý krok je realizován pomocí stejné korutiny jako u běžných jednotek `MoveCoroutine()`, která je řízena pohybovým časovačem. Během čekací fáze mezi jednotlivými kroky jednotka střeží okolí na pozemní vrstvě a v případě výskytu nepřátelské jednotky provede útok na některý z dosažitelných šestiúhelníků pod sebou na pozemní úrovni (viz obr. 5.18b). Útok je omezen tak, aby během jedné fáze pohybu vystřelila na každý šestiúhelník maximálně jednou. Celý proces je řízen vnitřní korutinou `MoveAttackCoroutine()`, která postupně volá pohybové a útočné korutiny, čímž umožňuje stíhačce jednat autonomně až do konce své trasy. Po dosažení posledního šestiúhelníku na své trase jednotka automaticky opustí mapu ve směru letu. Výběr trasy hráčem probíhá technikou *drag-and-drop* na šestiúhelník v požadovaném směru, což je znázorněno na obrázku 5.18.



(a) Výběr trasy pohybu stíhačky.

(b) Indikace cílových pozic během čekání na pohyb.

Obrázek 5.18: Po nasazení jednotky typu stíhačka lze zvolit její trasu viz obr. 5.18a. Při výběru stíhačky jsou navíc zvýrazněny šestiúhelníky, na kterých by mohlo dojít k útoku, pokud by se na nich vyskytla nepřátelská jednotka viz obr. 5.18b.

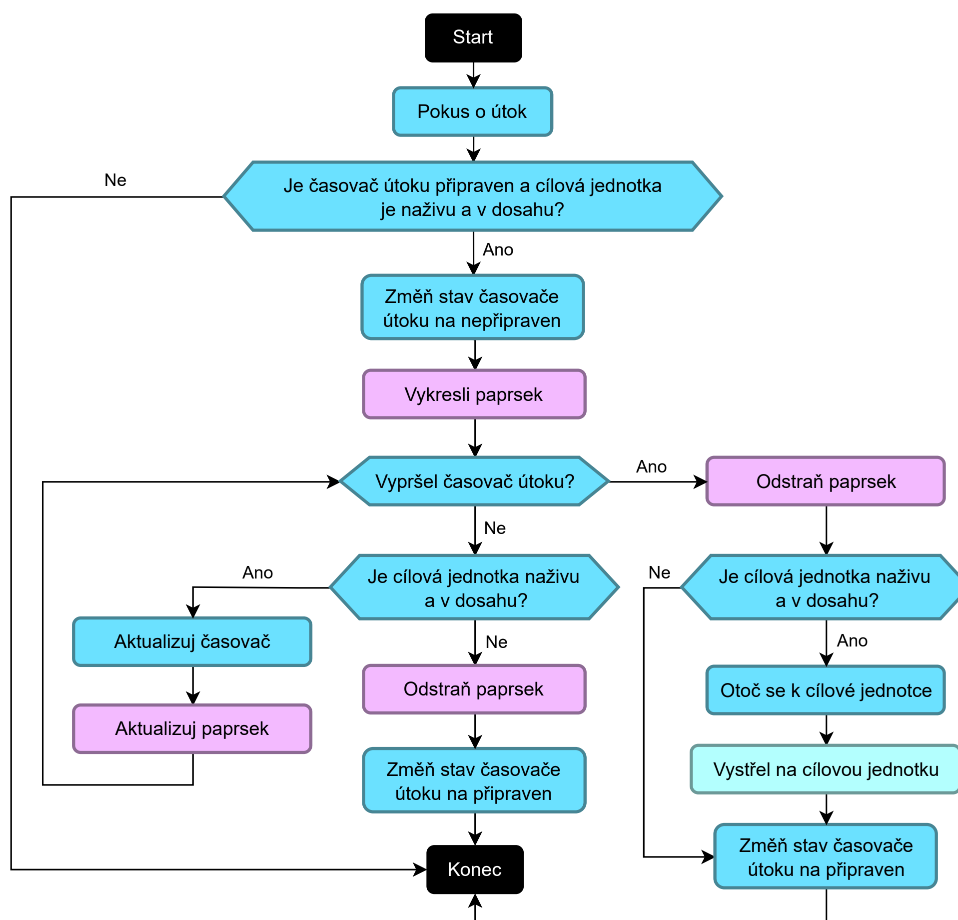
5.5 Útok jednotky

Vstupním bodem pro provedení útoku je metoda `TryAttack()`, která přijímá cílovou jednotku jako parametr. Pokud je cílová jednotka v dosahu a stav časovače útoku zdrojové jednotky je připraven, je spuštěna korutina `AttackCoroutine()`, která realizuje průběh útoku. Během této korutiny je nejprve aktivována vizualizace zaměřování, což propojí útočící a cílovou jednotku paprskem a zároveň zobrazuje průběh dobíjení v podobě intenzity paprsku.

Po dobu čekání na vypršení útočného časovače je průběžně kontrolována platnost cílové jednotky, zda mezitím nedošlo k jejímu eliminaci, vzdálení mimo dosah nebo skrytí v případě jednotky typu voják, a to pomocí metody `IsTargetInRange()`. Pokud dojde k některé z těchto situací, je korutina předčasně ukončena. Jakmile časovač vyprší, jednotka se nejprve otočí směrem k cíli pomocí metody `Rotate()`, přičemž po dokončení rotace je zavolána metoda `Attack()`, která vystřelí projektil vytvořený z prefabu nastaveného v konfiguraci dané jednotky. Průběh této akce je znázorněn ve vývojovém diagramu na obrázku 5.19.

Cílová jednotka zpracuje zásah pomocí metody `TakeDamage()`, kde dojde k odečtení zdraví a aktualizaci indikátoru zdraví. Pokud zdraví poklesne na nulu nebo méně, je vyvolána metoda `Die()`, která ukončí aktivitu jednotky, odstraní ji ze seznamu aktivních jednotek a informuje správce hry o jejím zániku.

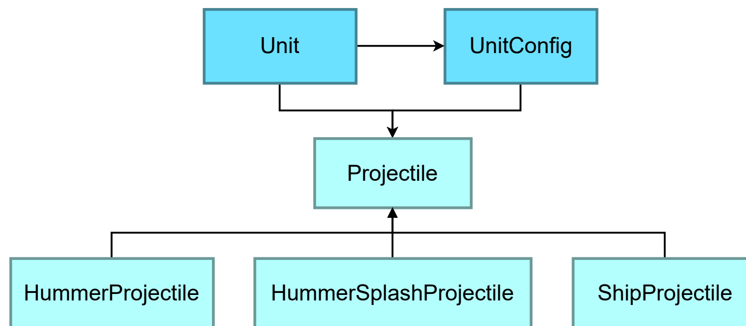
Z pohledu ovládání jednotek hráčem je metoda `TryAttack()` volána opakovaně, což se řídí na základě zvolené jednotky a volby jejího cíle s doprovodem obrysů jednotek viz sekce 5.3.8.



Obrázek 5.19: Vývojový diagram zachycující sekvenci akcí při pokusu jednotky o útok na cílovou jednotku.

5.5.1 Projektily

Tato sekce popisuje realizaci projektilů využívaných při útocích jednotek, včetně specifického chování jednotky typu džíp. Na obrázku 5.20 jsou znázorněny komponenty zprostředkovávající tuto funkcionalitu.

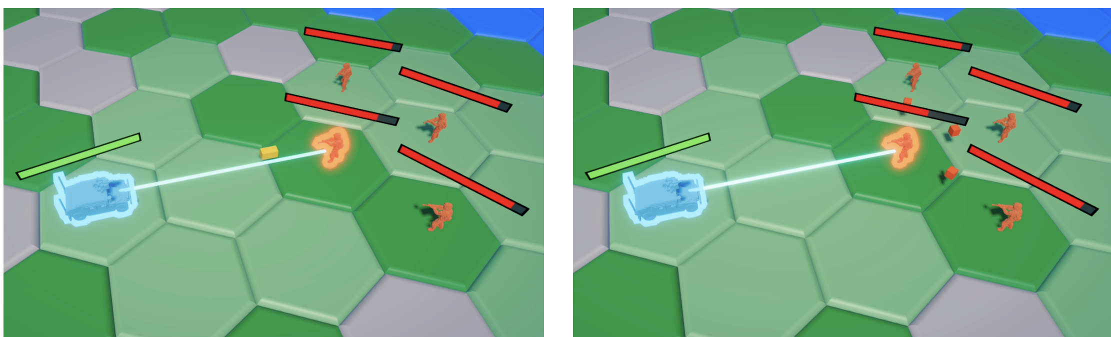


Obrázek 5.20: Komponenty zprostředkovávající funkcionalitu projektilů.

Jednotka v rámci metody `Attack()` vystřelí projektil pomocí metody `Shoot()`, která vytvoří instanci projektilu definovanou v konfiguračním skriptu `UnitConfig`. Projektilu jsou předány informace o zdrojové a cílové jednotce, na základě čehož se orientuje a pohybuje směrem k cíli. Po dosažení cílové jednotky dojde k jejímu zasažení a spuštění metody `TakeDamage()`. Pokud je však cílová jednotka během letu projektilu eliminována, je projektil automaticky odstraněn.

Specifické chování jednotky typu džíp je realizováno pomocí projektilu, jehož prefab má přiřazen skript `HummerProjectile`, definovaný v konfiguraci `UnitConfig` džípu. Tento skript přepisuje metodu `HitTarget()` ze základní třídy `Projectile`. Při zásahu cílové jednotky navíc ověřuje, zda se zdrojová jednotka (džíp) nachází na šestiúhelníku s biomem typu pláně. Pokud ano, tak se nejprve se určí šestiúhelníky nacházející se v prstenci za cílovou jednotkou (vzhledem k původní pozici útoku), následně se z nich vyfiltrují pouze ty, které zároveň sousedí s cílovým šestiúhelníkem a obsahují nepřátelskou jednotku. Pro každou takovou jednotku je poté vytvořen nový projektil s připojeným skriptem `HummerSplashProjectile`, který jí udělí mírně snížené poškození. Tato schopnost džípu je demonstrována na obrázku 5.21.

Projektil specifický pro loď, jak již bylo zmíněno v sekci 5.3.3, slouží výhradně pro vizuální efekt a odlišuje se pouze trajektorií letu.



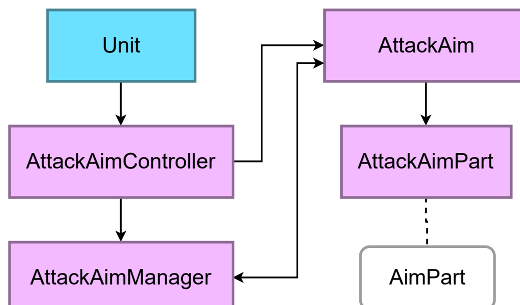
(a) Nejprve je vystřelen hlavní projektil na cíl.

(b) Jelikož se džíp nachází na šestiúhelníku s biomem pláně je projektil po zásahu rozdělen do menších projektilů zasahující dodatečné jednotky.

Obrázek 5.21: Situace demonstrující speciální schopnost jednotky typu džíp.

5.5.2 Indikace souboje jednotek

Tato sekce popisuje realizaci indikačního paprsku při soubojích mezi jednotkami. Komponenty, které se na této funkcionalitě podílejí je možné vidět na obrázku 5.22.



Obrázek 5.22: Komponenty zajišťující vykreslení indikačního paprsku pro útok jednotky.

Podobně jako u indikační šipky pro pohyb má každá jednotka k dispozici i ovládací prvek pro vizualizaci útoku, a to prostřednictvím instance třídy `AttackAimController`. Tento prvek poskytuje tři hlavní metody: `StartAiming()` pro zahájení zaměřování, `SetEmission()` pro úpravu intenzity paprsku a `StopAiming()` pro jeho ukončení.

Vykreslení paprsku zajišťuje třída `AttackAim`, která se skládá ze dvou částí paprsku reprezentovaných instancemi třídy `AttackAimPart`. Každá část uchovává informaci o jednotce, ke které náleží a jednotce, která ji aktuálně vlastní.

Komponenta `AttackAimManager` spravuje všechny aktivní paprsky v seznamu typu `AttackAim` a na základě požadavků z `AttackAimController` zajišťuje jejich vytváření, aktualizaci i mazání. Při volání metody `StartAiming()` (která indikuje, že jednotka zahajuje útok nebo se připojuje k probíhajícímu souboji) vyhledá existující paprsek mezi danou dvojicí jednotek. Pokud existuje, aktualizuje vlastnictví jeho částí. V opačném případě vytvoří nový paprsek. Metoda `StopAiming()` převádí vlastnictví části paprsku na druhou jednotku. Pokud se ani jedna z jednotek nadále souboje neúčastní, je paprsek odstraněn. Systém vlastnictví tak umožňuje metodě `SetEmission()` měnit intenzitu pouze u částí paprsku, které daná jednotka aktuálně vlastní.

Vizualizace paprsku je průběžně aktualizována ve třídě `AttackAim`, která zajišťuje přepočet pozice, orientace a délky obou jeho částí podle aktuální pozice jednotek. Barva části paprsku odpovídá týmu jejího aktuálního vlastníka a vychází z konfigurace definované v `UnitSharedConfig`. Ukázka vizualizace probíhajících soubojů je možné vidět na obrázku 5.23.

(a) Spojenecká jednotka typu stíhačka je aktuálně pod útokem nepřátelského tanku a lodi.

(b) Zde probíhá vzájemný souboj mezi jednotkami typu tank a zároveň zde jednotka typu helikoptéra útočí na nepřátelský tank.

Obrázek 5.23: Vizualizace probíhajících soubojů prostřednictvím paprsků, které zároveň svou intenzitou indikují průběh časovače útoku, tedy zbývající dobu přebíjení.

5.5.3 Útok jednotky typu loď

Jednotka s připojeným skriptem `ShipUnit` realizuje útok odlišně od ostatních typů jednotek. Namísto přímého výběru cílové jednotky střeží oblast zvolených šestiúhelníků prostřednictvím metody `TryAttackHex()`, která uloží zvolený šestiúhelník a automaticky určí sousedící šestiúhelníky v jeho okolí (viz obr. 5.24). Ty jsou následně periodicky kontrolovány v rámci korutiny `AttackHexesCoroutine()` s ohledem na dosah útoku.

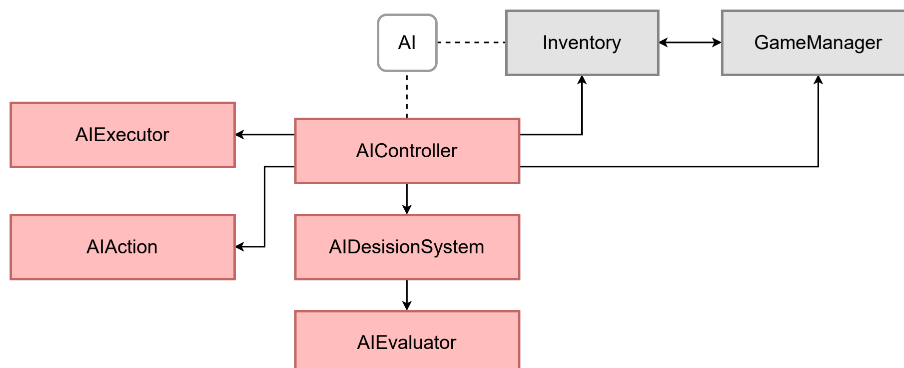
Pokud je v některém z cílových šestiúhelníků detekována nepřátelská jednotka, je zahájen útok voláním výchozí metody pro útok `TryAttack()` definované v základní třídě `Unit`. Ve skriptu lodi je přepsána metoda `Shoot()`, která místo jednoho projektilu vystřelí opakovaně tři projektily s krátkým časovým odstupem.

Tato jednotka jak bylo zmíněno v sekci 4.3.2 disponuje speciálním časovačem střežení, který po danou dobu zabraňuje znovu definování střežených šestiúhelníků.

Obrázek 5.24: Příklad výběru šestiúhelníků ve vzdušné vrstvě ke střežení lodí, která se nachází na pozemní vrstvě.

5.6 Umělá inteligence

Tato sekce popisuje architekturu umělé inteligence, která v rámci hry zastupuje protivníka. Na obrázku 5.25 jsou znázorněny hlavní komponenty odpovědné za pohyb jednotek, útoky a nákup nových jednotek.



Obrázek 5.25: Komponenty podílející se na realizaci protivníka formou umělé inteligence.

5.6.1 Centrální řídicí komponenta

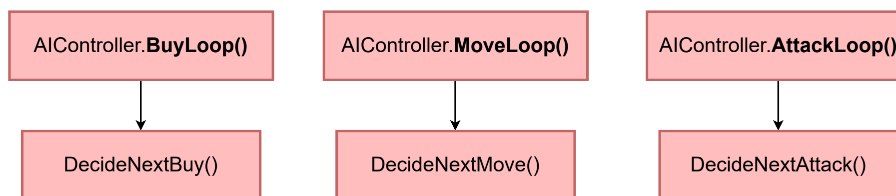
Centrálním prvkem celé umělé inteligence je komponenta `AIController`, která je připojena k objektu ve scéně reprezentující protivníka. Tato komponenta obsahuje referenci na vlastní inventář čímž má přístup k rozpočtu, na vlastní seznam aktivních jednotek a na začátku hry spustí tři nezávislé smyčky v podobě korutin:

- `BuyLoop()` pro rozhodování o nákupu jednotek,
- `MoveLoop()` pro rozhodování o pohybu jednotek,
- `AttackLoop()` pro rozhodování o útoku jednotek.

Smyčky řízení chování

Nákupní smyčka `BuyLoop()` probíhá s delším náhodným zpožděním (v intervalu 3 až 5 sekund) a na základě stavu hry požádá `AIDecisionSystem` o výběr typu jednotky k zakoupení. Pokud není vrácen typ `None`, komponenta `AIController` zkontroluje, zda má dostatek prostředků a zda existuje dostupný šestiúhelník pro nasazení a při úspěchu ji zakoupí.

Smyčky `MoveLoop()` a `AttackLoop()` pravidelně procházejí všechny vlastněné jednotky a pro každou z nich vyhodnocují vhodnost provedení dané akce. Každá jednotka má vlastní hodnotu zpoždění pro útok `nextAttackDelay` a pohyb `nextMoveDelay`, které se nastavují v případě, že jednotka při rozhodování zvolí akci typu `Wait`. Tyto hodnoty určují krátkou pauzu pro daný typ akce (pohyb nebo útok) a jsou vzájemně nezávislé, takže čekání na útok neovlivňuje možnost pohybu a naopak. Pokud časový zámek vyprší a daná jednotka má patřičný časovač připraven (časovač pohybu v `MoveLoop()`, nebo časovač útoku v `AttackLoop()`), je požádán systém `AIDecisionSystem` o rozhodnutí další akce k provedení jednotkou (viz obr. 5.26).



Obrázek 5.26: Hlavní volané metody v rámci patřičných smyčkách pro rozhodnutí následující akce.

Rozhodovací výstup

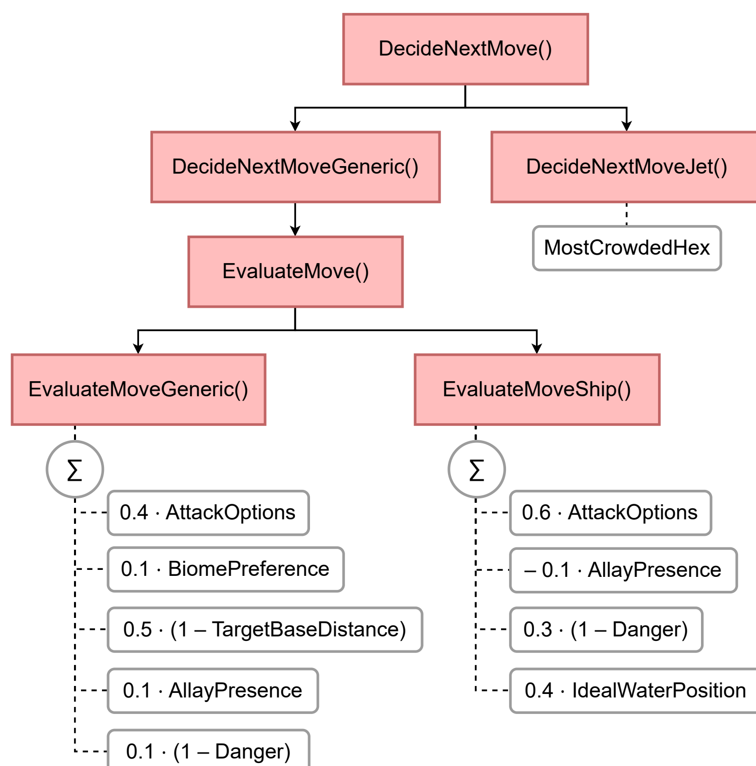
Rozhodnutí umělé inteligence je vráceno ve formě instance třídy `AIAction`, která reprezentuje typ akce (`Move`, `Attack` nebo `Wait`) a podle potřeby nese odkaz na cílovou jednotku nebo šestiúhelník. Například akce typu `Attack` obvykle obsahuje referenci na cílovou jednotku, zatímco akce typu `Move` určuje cílový šestiúhelník. V případě specifických jednotek, jako je loď, která provádí útok na danou oblast (konkrétně daný šestiúhelník) místo konkrétní jednotky, je akce typu `Attack` doplněna právě o cílový šestiúhelník.

Vykonání akce

Provedení akce vyhodnocené rozhodovacím systémem zajišťuje `AIActionExecutor`. Tato třída poskytuje metodu `ExecuteAction()`, která na základě typu akce provede příslušnou metodu dané jednotky. Vstupním bodem pro realizaci pohybu je metoda `TryMove()`, zatímco útok je zajištěn metodou `TryAttack()`, případně `TryAttackHex()` v případě lodí. Akce typu `Wait` je zachycena ve smyčkách `MoveLoop()` a `AttackLoop()`, které v takovém případě nastaví jednotce odpovídající hodnotu časového zpoždění pro daný typ akce.

5.6.2 Pohyb

V rámci smyčky `MoveLoop()` je pro určení další akce pohybu nejprve zavolána metoda `DecideNextMove()` systému `AIDecisionSystem`. Tato metoda slouží jako rozcestník mezi jednotlivými typy jednotek. U jednotky typu stíhačka je rozhodnutí o pohybu určeno přímo, zatímco u ostatních typů jednotek je rozhodování delegováno na systém `AIEvaluator`, který vyhodnocuje možné cílové pozice na základě definovaných faktorů.



Obrázek 5.27: Komponenty zajišťující určení optimální cílové pozice pro pohyb. Význam jednotlivých faktorů je popsán v sekci níže.

Rozhodovací vrstva

Po zavolání metody `DecideNextMove()` se rozlišuje způsob **rozhodování** o pohybu mezi jednotkou typu stíhačka a ostatními jednotkami.

V případě jednotky typu stíhačka je rozhodnutí o jejím pohybu realizováno metodou `DecideNextMoveJet()`. Ta pomocí systému mapy vlivu vyhledá pozemní šestiúhelník s aktuálně nejvyšší hodnotou `playerPresence`, což reprezentuje oblast s největší koncentrací nepřátelských jednotek. Stíhačka je následně vyslána na odpovídající šestiúhelník ve vzdušné vrstvě, který se nachází nad touto pozicí.

Pro ostatní typy jednotek je pohyb určován metodou `DecideNextMoveGeneric()`, která využívá vyhodnocovací vrstvu k ohodnocení dostupných pozic. Tento proces probíhá následovně:

1. Nejprve je vyhledán nejlépe hodnocený dosažitelný šestiúhelník z aktuální pozice jednotky. Každý šestiúhelník je vyhodnocen metodou `EvaluateMove()`, která šestiúhelníku přidělí skóre.

2. Pokud žádná pozice k přesunu neexistuje, jednotka zůstane stát a provede akci typu `Wait`.

3. Jinak je následně zjištěno, zda skóre aktuálního pole není vyšší nebo rovno skóre nejlepší nalezené možnosti. Pokud ano, provede se dodatečné ověření, zda i z původně

nejlepšího šestiúhelníku neexistuje další výhodnější pozice. Pokud ani toto nevede k nalezení výhodnější pozice, jednotka opět setrvá na místě akcí typu `Wait`.

4. V opačném případě je vydána akce typu `Move` k přesunu jednotky na vybraný šestiúhelník.

Tento přístup zajišťuje, že jednotka se pohybuje pouze v případech, kdy existuje reálný strategický přínos oproti setrvání na místě.

Vyhodnocovací vrstva

Po zavolání metody `EvaluateMove()` se rozlišuje způsob **vyhodnocování** o pohybu mezi jednotkou typu loď a ostatními jednotkami.

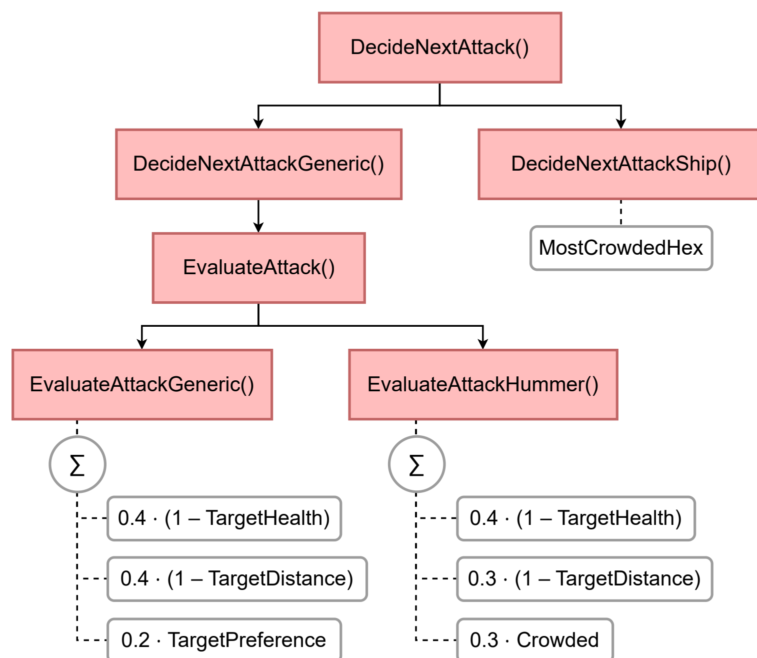
Vyhodnocení jednotlivých možností pohybu je realizováno pomocí patřičné metody pro danou jednotku, která vrací skóre pro daný šestiúhelník. Skóre je určeno jako vážený součet několika dílčích faktorů viz rovnice 4.5, přičemž konkrétní faktory a jejich váhy se mohou lišit v závislosti na typu jednotky, což specificky znázorněno na obrázku 5.27. Pro jednotky se používají následující faktory relevantní při určování pohybu:

- **možnosti útoku z dané pozice** (`AttackOptions`) – počet dostupných cílů z pozice děleno maximální počet možných cílů z pozice,
- **preferovaný biom** (`BiomePreference`) – lehce zvýhodní pozici s preferovanými biomem, nebo znevýhodní na základě typu jednotky,
- **vzdálenost od nepřátelské základny** (`TargetBaseDistance`) – vzdálenost pozice od nepřátelské základny děleno maximální vzdálenost na mapě,
- **přítomnost spojeneckých jednotek** (`AllayPresence`) – počet spojenců děleno maximální počet spojenců v rámci bezprostředně sousedních pozic,
- **nebezpečnost pozice** (`Danger`) – hodnota kontroly hráče děleno maximální hodnota kontroly hráče získané z mapy vlivu.
- **optimální pozice na vodní ploše** (`IdealWaterPosition`) – na základě aktuálního počtu vlastněných lodí je určeno ideální pásmo vzdálenosti od základny. Šestiúhelník je poté ohodnocen podle své odchylky od tohoto ideálu.

Každý faktor vrací hodnotu v intervalu $\langle 0, 1\rangle$, čímž je zajištěna kompatibilita při váženém sčítání.

5.6.3 Útok

Architektura útoku je obdobná jako u pohybu, liší se však tím, že místo vyhodnocování možných pozic pro přesun se soustředí na výběr cílové jednotky (nebo šestiúhelníku) v rámci útoku. Ve smyčce `AttackLoop()` je pro určení další akce útoku nejprve zavolána metoda `DecideNextAttack()` systému `AIDecisionSystem`. Tato metoda slouží jako rozcestník mezi jednotlivými typy jednotek. U jednotky typu loď je rozhodnutí o útoku určeno přímo, zatímco u ostatních typů jednotek je rozhodování delegováno na systém `AIEvaluator`, který vyhodnocuje možné cíle na základě definovaných faktorů.



Obrázek 5.28: Komponenty zajišťující určení optimálního cíle pro útok. Význam jednotlivých faktorů je popsán v sekci níže.

Rozhodovací vrstva

Po zavolání metody `DecideNextAttack()` se rozlišuje způsob **rozhodování** o útoku mezi jednotkou typu loď a ostatními jednotkami.

V případě jednotky typu loď, která preferuje útoky na oblasti s vysokou koncentrací nepřátelských jednotek ve vzdušné vrstvě, je rozhodnutí o útoku realizováno metodou `DecideNextAttackShip()`. Tato metoda získá šestiúhelníky v dosahu útoku a pomocí hodnot přítomnosti hráče, poskytnutých systémem mapy vlivu, zvolí ten s nejvyšší hodnotou. Pokud již daný šestiúhelník jednotka střeží, vyvolá akci `Wait`, v opačném případě zvolí akci `Attack` s tímto cílovým šestiúhelníkem.

Pro ostatní typy jednotek je útok určován metodou `DecideNextAttackGeneric()`, která využívá vyhodnocovací vrstvu k ohodnocení dostupných cílů. Tento proces probíhá následovně:

1. Nejprve jsou získány všechny jednotky, na které může daná jednotka aktuálně zaútočit.

2. Pokud žádný cíl není dostupný, je vydána akce typu `Wait`, čímž jednotka vyčká na další příležitost.

3. U každého dostupného cíle se vypočítá skóre pomocí metody `EvaluateAttack()`, které určuje jeho vhodnost.

4. Následně je vybrán cíl s nejvyšším skóre a vytvořena akce typu `Attack`, která je navrácena rozhodovacímu systému.

Vyhodnocovací vrstva

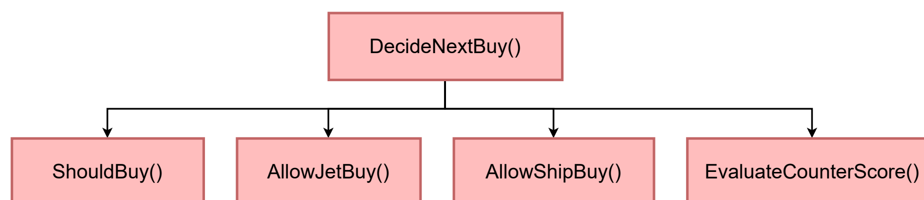
Po zavolání metody `EvaluateAttack()` se rozlišuje způsob **vyhodnocování** o útoku mezi jednotkou typu džíp a ostatními jednotkami.

Vyhodnocení jednotlivých možností útoku je realizováno pomocí patřičné metody pro danou jednotku, která vrací skóre pro daný cíl. Skóre je určeno jako vážený součet několika dílčích faktorů viz rovnice 4.5, přičemž konkrétní faktory a jejich váhy se mohou lišit v závislosti na typu jednotky, což specificky znázorněno na obrázku 5.28. Pro jednotky se používají následující faktory relevantní při určování útoku:

- **zdraví cíle** (`TargetHealth`) – zdraví cíle děleno maximální zdraví cíle,
- **vzdálenost od cíle** (`TargetDistance`) – vzdálenost cíle děleno maximální dosah útoku jednotky,
- **preference typu cíle** (`TargetTypePreference`) – lehce zvýhodní cíle preferované danou jednotkou,
- **koncentrace na pozici cíle** (`Crowded`) – hodnota přítomnosti hráče na pozici cíle děleno maximální hodnota přítomnosti hráče získané z mapy vlivu.

5.6.4 Nákup

V rámci smyčky `BuyLoop()` je pro určení, zda v daný okamžik nakoupit a případně jakou jednotku zakoupit, je volána metoda `DecideNextBuy()` systému `AIDecisionSystem`.



Obrázek 5.29: Komponenty zajišťující optimálního nákup jednotky.

V rámci této metody je voláno několik významných metod (viz obrázek 5.29). Metoda `ShouldBuyUnit()` rozhoduje, zda nákup aktuálně vůbec provést. Její logika zohledňuje následující kritéria:

- pokud má umělá inteligence méně než pět jednotek, nákup je povolen,
- pokud má méně než 150 měny, nákup je v tomto okamžiku zamítnut z důvodu spoření,
- pokud má umělá inteligence více jednotek než hráč, pravděpodobnost nákupu lineárně klesá s rozdílem v počtech jednotek, přičemž odchylka o čtyři jednotky nebo více způsobí, že pravděpodobnost nákupu klesne na nulu,
- ve všech ostatních případech je nákup povolen.

Dále jsou použity metody `AllowJetBuy()` a `AllowShipBuy()`, které ověřují, zda jsou splněny specifické podmínky pro povolení nákupu jednotek typu stíhačka, respektive loď. Tyto jednotky jsou chápány jako spíše příležitostné.

- `AllowJetBuy()` umožní nákpu stíhačky, pokud je na mapě detekována výrazná přítomnost pozemních jednotek hráče a jejich počet přesahuje určitou mez. Toto reflektuje specifickou roli stíhačky jako reakce na hromadění pozemních jednotek.
- `AllowShipBuy()` umožňuje nákup lodí v případě, že hráč má více vrtulníků, než umělá inteligence aktuálně vlastní lodí. To odpovídá protivzdušné roli lodí.

Po těchto ověřeních se pro každou přípustnou jednotku vypočte skóre, které zohledňuje jednak strategickou vhodnost daného typu jednotky vůči aktuálnímu složení jednotek protivníka, a jednak zastoupení tohoto typu v rámci již vlastněných jednotek. Vhodnost se určuje metodou `EvaluateCounterScore()`, která každému typu jednotky přiřazuje ohodnocení na základě počtu nepřátelských jednotek různých typů. Váhy pro jednotlivé kombinace typů jednotek (např. jak vhodné je zakoupit helikoptéru vůči aktuálnímu počtu tanků a ostatních jednotek) jsou předem definovány a zachyceny v tabulce 5.1. Čím vyšší hodnota v tabulce, tím výhodnější je daný typ jednotky pro nákup vůči konkrétní situaci. Takto získané skóre je následně upraveno podle aktuálního zastoupení daného typu jednotky, aby se zabránilo nadměrnému hromadění jednoho typu. Jednotka s nejvyšším výsledným skóre je pak vybrána k nákupu, jak ukazuje algoritmus 5.

Algoritmus 5 Rozhodnutí o nákupu jednotky

```
 1: function decideNextBuy
 2:     if not ShouldBuyUnit then
 3:         return None
 4:     end if
 5:     potentialUnitTypes ← GetControllableUnitTypes
 6:     if not AllowJetBuy then
 7:         Odstraň Jet z potentialUnitTypes
 8:     end if
 9:     if not AllowShipBuy then
10:         Odstraň Ship z potentialUnitTypes
11:     end if
12:     totalCount ← GetTotalUnitCount(Team)
13:     unitScores ← {}
14:     for all unitType ∈ potentialUnitTypes do
15:         baseScore ← EvaluateCounterScore(unitType)
16:         unitTypeCount ← GetUnitCount(unitType, Team)
17:         proportion ← unitTypeCount/totalCount
18:         adjustedScore ← baseScore · (1 − proportion)
19:         unitScores[unitType] ← adjustedScore
20:     end for
21:     return jednotka s nejvyšším skóre v unitScores
22: end function
```

Jednotka	voják	džíp	tank	loď	helikoptéra	stíhačka
voják	0	2	1	3	2	3
džíp	4	0	2	4	1	2
tank	3	3	0	5	3	3
loď	3	2	1	0	5	4
helikoptéra	3	5	2	1	0	4
stíhačka	4	5	3	1	2	0

Tabulka 5.1: Váhy určující vhodnost nákupu jednotky vůči ostatním jednotkám. Každá hodnota vyjadřuje účinnost jednotky v řádku proti jednotce ve sloupci. Vyšší číslo znamená větší výhodu dané jednotky vůči konkrétnímu typu nepřátelské jednotky.

5.7 Testování

V polovině vývoje byla obdržena stručná zpětná vazba od osoby se znalostmi v oblasti herního vývoje. Na základě tehdejšího prototypu bylo identifikováno, že hráč nemá přehled o tom, která jednotka právě na jakou jednotku útočí, případně zda vůbec nějaký útok právě probíhá. Tento problém byl již v té době zohledněn v návrhu a plánovalo se jeho řešení.

Výsledkem se stal systém paprsků, které vizuálně indikují právě probíhající útoky mezi jednotkami. Paprsek zároveň reprezentuje časovač útoku prostřednictvím své intenzity. Díky tomu bylo možné odstranit nadbytečné ukazatele zobrazované nad všemi jednotkami, protože i časovač pohybu je nyní reprezentován prostřednictvím pohybové šipky, jejíž výplň postupně narůstá.

Na konci vývoje hry se testování zúčastnily čtyři osoby ve věku 22 až 23 let s pokročilými herními dovednostmi a četnými zkušenostmi s různými typy her. Testování probíhalo nejprve formou pozorování, zda je hráč schopen intuitivně pochopit ovládání. Následně byli účastníci vedeni sérií dotazů, jejichž cílem bylo zjistit, jak rychle a přesně dokážou pochopit jednotlivé herní mechaniky při postupném seznamování se s hrou.

Při prvním pokusu bez dodaných informací hra na testery působila jako rychle plynoucí a zpočátku měli obtíže s orientací. Postupně však při opakovaném hraní začali chápat základní principy a jak obecně postupovat ve hře. K porozumění však potřebovali vysvětlení některých prvků, zejména významu barevných obrysů jednotek, které vizuálně sdělují užitečné informace o jejich stavu z hlediska uživatelského rozhraní. Rovněž bylo nutné objasnit jednotlivé vlastnosti jednotek a případné specifické chování, které se s nimi pojí.

Přibližně od pátého spuštění hry, kdy již byli hráči obeznámeni s mechanikami nákupu, pohybu, útoku a ovládáním specifických jednotek, se mohli soustředit na vlastní snahu o vítězství.

Testující hráči ocenili základní myšlenku hry a uvedli, že se jim nápad líbil. Pozitivně vnímali také rozhodnutí využít šestiúhelníkové rozvržení herního pole namísto čtvercového. Také vyzdvihli, že hra staví na jednoduchých principech, které lze rychle pochopit, zejména v podobě odlišných vlastností a chování jednotek, jež lze ve hře strategicky kombinovat a vzájemně proti sobě využívat.

Na druhou stranu zazněly i požadavky na přidání tutoriálu či vysvětlivek, například ve formě tabulky atributů jednotek a stručného popisu jejich speciálních vlastností. Dále bylo navrženo zlepšení ovládání volné kamery, konkrétně možnost přibližování a oddalování pomocí kolečka myši. Někteří testující by také preferovali jednodušší způsob pohybu a nákupu jednotek, namísto přetahování (*drag-and-drop*) by uvítali systém založený na prostém

označení a kliknutí (což by mohlo být například součástí nastavení). Obecně by ocenili větší podporu ze strany uživatelského rozhraní, například pomocí výraznějšího zvýraznění vybrané jednotky změnou její barvy, nikoliv pouze obrysem, a podobných zlepšení zvyšujících přehlednost.

5.7.1 Rozšíření

V rámci diskuze zazněla řada návrhů na možná rozšíření hry. Hlavním z nich byla podpora hry pro více hráčů (*multiplayer*). Vzhledem k tomu, že samotný koncept klade důraz na jednotky, které se strategicky uplatňují vůči sobě, nabízí se také systém postupného odemykání nových jednotek. Každá z těchto jednotek by měla unikátní chování a vlastnosti, přičemž by hráči měli k dispozici omezený počet jednotek k výběru ze své odemčené sady. Volba jednotek by navíc mohla být ovlivněna konkrétně vygenerovanou mapou pro daný zápas.

Další návrhy směřovaly na rozšíření herní mapy přidáním nových biomů, zavedení střídání dne a noci, ročních období a proměnlivého počasí, což by dále ovlivňovalo některé jednotky a zároveň hráčovu volbu omezené sady jednotek na začátku hry.

Kapitola 6

Závěr

Cílem práce bylo vhodně navrhnout a realizovat hru žánru real-time strategie s přístupnou učící křivkou, poskytnout automaticky generovanou herní mapu a zprostředkovat protihráče ve formě umělé inteligence. Cíl byl úspěšně splněn.

Nastudované techniky herního vývoje a vhodných vývojových prostředí jsou popsány v kapitole 2. Návrh hry je popsán v kapitole 4 a její implementace v prostředí Unity je popsána v kapitole 5. Návrh jednotlivých vlastností a chování jednotek je obsažen v sekcích 4.2 a 4.3 a popis implementace jejich hlavních akcí jsou v sekcích 5.4 a 5.5. Techniky procedurálního generování byly nastudovány v sekci 3.1, které byly využity pro zvolení vhodné varianty pro generování herní mapy. Návrh generování herní mapy s použitím šumové funkce je vysvětlen v sekci 4.1 a její implementace je popsána v sekci 5.1. Nastudované techniky o umělé inteligenci jsou uvedeny v sekci 3.2, přičemž aplikované techniky jsou vysvětleny v sekci 4.4 a její konkrétní implementace je popsána v sekci 5.6. Uživatelské testování je zaznamenáno v sekci 5.7 a demonstrační video je na přiloženém paměťovém médiu.

Hra ve svém výsledné stavu disponuje automaticky generovanou mapou, přičemž poskytuje variabilitu v opakovaném hraní. Herní mapa se skládá z pozemní a vzdušné vrstvy s celkem pěti možnými biomy (pláně, les, kopce, voda a vzduch), které mohou ovlivňovat vlastnosti některých jednotek. Hra obsahuje celkem sedm jednotek (základna, voják, džíp, tank, loď, helikoptéra a stíhačka), které mohou být nasazovány a ovládány k vykonávání pohybu nebo útoku. Akce pohybu je hráči zobrazena indikační šipkou a akce útoku prostřednictvím paprsků indikující souboje mezi jednotkami. Každá jednotka má své atributy a možné speciální chování s tím, že jsou navrženy tak, aby se navzájem uplatňovaly proti sobě. Hráč má k dispozici nákupní panel, herní menu a kamerový systém. Uživatelské rozhraní je dále podpořeno obrysy šestiúhelníků a jednotek s určitým významem. Protihráč je realizován prostřednictvím umělé inteligence schopen strategického rozhodování.

Tato práce pro mě byla velkým přínosem. Jedná se o moji první plnohodnotnou hru, kterou jsem kdy navrhoval a implementoval. Uvědomuji si, jak komplexní je snaha o zajištění pro mě nejdůležitějšího aspektu hry, zábavnosti hry. Na rozdíl od běžných uživatelských aplikací, kde je cílem co nejrychleji a nejefektivněji vykonat potřebnou činnost, je u hry opakem, a tedy udržet si uživatele co nejdéle vtažené.

Na základě cenné zpětné vazby se nabízí mnoho nápadů, jak hru vylepšit i rozšířit. V rámci dalšího vývoje by bylo vhodné opakování uživatelského testování k zdokonalení prvků zajišťující plynulost ovládání hráčem. Poté by hru nejvíce obohatil režim pro více hráčů a také dodání více obsahu v podobě přidání dalších typů jednotek a biomů.

Literatura

[1] Adams, E. *Fundamentals of Game Design.* 3. vyd. New Riders, 2013. ISBN 978-0-321-92967-9.

[2] Adams, E. a Dormans, J. *Game Mechanics: Advanced Game Design.* 1. vyd. New Riders, 2012. ISBN 978-0-321-82027-3.

[3] Georgios N. Yannakakis, J. T. *Artificial Intelligence and Games.* 1. vyd. Springer Cham, 2018. ISBN 978-3-319-63519-4.

[4] Gregory, J. *Game Engine Architecture.* 3. vyd. CRC Press, 2019. ISBN 978-1-1380-3545-4.

[5] Gustavson, S. Simplex noise demystified. *Linköping University, Linköping, Sweden, Research Report* online, 2005, sv. 1, č. 2, s. 6. Dostupné z: https://muugumuugu.github.io/b00kshelF/generative%20art/simplexnoise.pdf. [cit. 2025-04-15].

[6] Lagae, A.; Lefebvre, S.; Cook, R.; DeRose, T.; Drettakis, G. et al. A Survey of Procedural Noise Functions. *Computer Graphics Forum*, 2010, sv. 29, č. 8, s. 2579–2600. ISSN 1467-8659. Dostupné z: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2010.01827.x.

[7] Mattei, V. Artificial Intelligence in Video Games 101: An Easy Introduction. In: Clayton, M.; Passacantando, M. a Sanguineti, M., ed. *Intelligent Technologies for Interactive Entertainment.* Cham: Springer Nature Switzerland, 2024, s. 40–51. ISBN 978-3-031-55722-4.

[8] Millington, I. *AI for Games, Third Edition.* 3. vyd. CRC Press, 2019. ISBN 9781351053303.

[9] Noor Shaker, M. J. N. *Procedural Content Generation in Games.* 1. vyd. Springer Cham, 2016. ISBN 978-3-319-42716-4.

[10] Patel, A. *Hexagonal Grids* online. Dostupné z: https://www.redblobgames.com/grids/hexagons/. [cit. 2025-04-21].

[11] Perlin, K. An image synthesizer. *SIGGRAPH Comput. Graph.* New York, NY, USA: Association for Computing Machinery, červenec 1985, sv. 19, č. 3, s. 287–296. ISSN 0097-8930. Dostupné z: https://doi.org/10.1145/325165.325247.

[12] Perlin, K. Improving noise. In: *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques.* New York, NY, USA: Association for

Computing Machinery, 2002, s. 681–682. SIGGRAPH ’02. ISBN 1581135211. Dostupné z: https://doi.org/10.1145/566570.566636.

[13] Rabin, S. *Game AI Pro 1: Collected Wisdom of Game AI Professionals*. 1. vyd. CRC Press, 2014. ISBN 978-1-4665-6597-5.

[14] Scratchapixel. Perlin Noise. *Perlin Noise* online. Dostupné z: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/perlin-noise-part-2/perlin-noise.html. [cit. 2025-03-23].

[15] Scratchapixel. Creating a Simple 1D Noise. *Value Noise and Procedural Patterns* online. Dostupné z: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1/creating-simple-1D-noise.html. [cit. 2025-03-23].

[16] Scratchapixel. Creating a Simple 2D Noise. *Value Noise and Procedural Patterns* online. Dostupné z: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1/creating-simple-2D-noise.html. [cit. 2025-03-23].

[17] Tanya Short, T. A. *Procedural Generation in Game Design*. CRC Press, 2017. ISBN 978-1-4987-9919-5.

[18] Togelius, J.; Yannakakis, G. N.; Stanley, K. O. a Browne, C. Search-Based Procedural Content Generation: A Taxonomy and Survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 2011, sv. 3, č. 3, s. 172–186. ISSN 1943-0698.

[19] Unity. *Unity User Manual 2022.3 (LTS)* online. Dostupné z: https://docs.unity3d.com/2022.3/Documentation/Manual/index.html. [cit. 2025-04-15].

Příloha A

Obsah přiloženého paměťového média

Přiložené paměťové médium obsahuje následující položky:

- `src/` – zdrojové soubory hry,
- `build/` – zkompilované zdrojové soubory hry pro platformu Windows,
- `latex-src/` – zdrojové soubory LaTeX,
- `thesis.pdf` – vysázený text závěrečné práce,
- `demo-video.mp4` – demonstrační video,
- `README.md` – manuál.