



MATEMATICKO-FYZIKÁLNÍ  
FAKULTA  
Univerzita Karlova

## BAKALÁŘSKÁ PRÁCE

Samuel Juraj Koprda

**RPG hra pro 2 hráče v Unity**

Katedra distribuovaných a spolehlivých systémů

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

Studijní program: Informatika se specializací Umělá inteligence

Praha 2025

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

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

V ..... dne.....  
podpis

Chtěl bych především poděkovat mému vedoucímu, Mgr. Pavlu Ježkovi, PhD., za vedení této práce a za jeho cenné rady během konzultací. Dále bych chtěl poděkovat svému bratrovi, který mi pomohl s nápady a svým uměním, a také svým rodičům, kteří mě podporovali a motivovali k dokončení této práce.

Název práce: RPG hra pro 2 hráče v Unity

Autor: Samuel Juraj Koprda

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů

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

**Abstrakt:** V RPG hrách jsou NPC často vyobrazena s různými předměty (např. helma, meč), ale když NPC umře, tak často zmizí z herního světa i její předměty. Nás by zajímalo, zda je možné vytvořit RPG hru, kde jsou předměty persistentní a nezmizí po smrti NPC. Cílem této práce je naprogramovat RPG hru s persistentními předměty, které ve hře zůstanou i po smrti NPC. Tyto předměty by měly být použitelné ostatními NPC a hráči. Vytvořená hra je pro 2 a více hráčů s lokálním multiplayerem. Hráči jsou rozděleni do dvou týmů. Cílem hry je ukradení vítězného předmětu druhého týmu. Hráči mají kontrolu nad postavou ve 2D světě, sbírají suroviny, získávají lepší vybavení a interagují se spojeneckými a neutrálními NPC. NPC sbírají suroviny, vyrábí předměty a bojují jak mezi sebou, tak proti hráčům. Jako součást práce jsme sepsali dokument designu hry, který detailně popisuje fungování hry a její mechaniky. Poté analyzujeme, jak jsme tyto mechaniky implementovaly.

Klíčová slova: RPG hra Unity multiplayer NPC

Title: Název práce v angličtině Two-player RPG game in Unity

Author: Samuel Juraj Koprda

Department: Department of Distributed and Dependable Systems

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

Abstract: In RPG games NPC's are usually shown with a variety of items (e.g. helmet, sword), but when they die its items often disappear from the game world alongside the NPC. We wonder, if it is possible to make an RPG game, where items are persistent and don't disappear after the death of an NPC. The goal of this thesis is to program an RPG game with persistent items, which stay in the game after the death of an NPC. These items should be usable by other NPCs and the players. Our game will be for 2 or more players with local multiplayer. The players are divided into two teams. The goal of the game is to steal the other teams victory item. The players control their character in a 2D world, they gather resources, get better equipment, craft items and interact with allied and neutral NPC's. NPC's gather resources, craft items and fight other NPCs or players. As a part of this thesis, we wrote a game design document, that in detail describes the design and mechanics of the game. Lastly, we analysed how we implemented those game mechanics.

Keywords: RPG game Unity multiplayer NPC

## Obsah

1. Úvod .....	2
2. Analýza konceptů .....	3
2.1. Simulace ekonomiky .....	3
2.1.1. City builder.....	3
2.1.2. Colony simulator .....	6
2.2. NPC a jejich vztah k předmětům.....	7
2.3. Umělá inteligence v ekonomice .....	7
2.4. Umělá inteligence v boji.....	8
2.5. Shrnutí návrhu ekonomiky naší hry .....	8
2.6. Zapojení hráče .....	8
2.6.1. Role hráče .....	8
2.6.2. Druhy interakce mezi hráčem a NPC .....	8
2.6.3. Volba pod-žánru .....	8
2.6.4. Singleplayer vs multiplayer .....	9
2.6.5. Detaily implementace multiplayeru .....	10
2.7. Cíle práce.....	12
2.8. Přispěvatelé.....	12
3. Dokument Herního Designu.....	13
3.1. Koncept hry .....	13
3.2. Postava.....	13
3.2.1. Vzhled.....	13
3.2.2. Pohyb .....	13
3.2.3. Rasa .....	13
3.2.4. Životy .....	13
3.2.5. Zlato.....	13
3.2.6. Inventář .....	14
3.2.7. Vybavení.....	14
3.2.8. Smrt .....	14
3.3. Předmět.....	14
3.3.1. <i>Využívání</i> předmětů .....	15
3.3.2. Pick-upy.....	15
3.3.3. Vítězné předměty.....	15
3.3.4. Seznam předmětů .....	15
3.4. Zbraně, brnění a zranění .....	16
3.4.1. Zbraně a jejich vlastnosti.....	17
3.5. Kouzla.....	21
3.6. Ložiska surovin .....	21
3.7. Objekt .....	22
3.7.1. Truhla .....	22
3.7.2. Obchod .....	22
3.7.3. Dveře .....	22
3.7.4. Postel .....	22
3.7.5. Podstavec .....	22
3.7.6. Kovadlina, pracovní stůl a výheň .....	22
3.8. Vyrábění předmětů .....	22
3.8.1. Pracovní stůl .....	23
3.8.2. Kovadlina .....	23

3.8.3. Výheň .....	23
3.9. Vstup hráče .....	24
3.10. GUI .....	24
3.10.1. Hráčův inventář .....	24
3.10.2. Truhla a obchod .....	25
3.11. Dialogový systém .....	26
3.12. Komunikační bubliny .....	27
3.13. Obrazovka pro více hráčů .....	28
3.14. Vyvolávací kruh .....	28
3.15. Budovy .....	28
3.15.1. Týmové základny .....	28
3.15.2. Kostel .....	29
3.15.3. Kasárny .....	29
3.15.4. Tržiště .....	30
3.15.5. Dřevorubecká chata .....	30
1.1.1. Hornická chata .....	30
3.16. Kočko-lidská vesnice .....	31
3.17. Kmen goblinů .....	31
3.18. Zřícenina .....	31
3.19. Spící Golem .....	31
3.20. Mapa světa .....	31
3.21. Rasy .....	32
3.22. Chování NPC .....	33
3.23. Spuštění hry .....	35
4. Detailní analýza .....	36
4.1. Herní Engine .....	36
4.2. Postavy .....	36
4.2.1. Fyzická interakce .....	36
4.2.2. Pohyb .....	36
4.2.3. Inventář .....	37
4.2.4. Vybavení .....	37
4.2.5. Ovládání inventáře a vybavení .....	39
4.2.6. Vstup hráče .....	40
4.2.7. Rasa .....	41
4.2.8. Schopnosti .....	41
4.2.9. Životy a poškození .....	41
4.2.10. Zlato .....	42
4.3. Předměty .....	43
4.4. Objekty .....	44
4.4.1. Jednorázové .....	44
4.4.2. Dlouhodobé .....	45
4.4.3. Objekty s UI .....	45
4.5. NPC Navigace .....	48
4.5.1. Vyhýbaní se ostatním postavám .....	49
4.5.2. Navigace skrz dveře .....	50
4.6. NPC AI .....	51
4.6.1. Analýza architektur .....	52
4.6.2. Analýza GOAP .....	53
4.6.3. Splnění posledních požadavků .....	54
4.7. Split Screen .....	54

4.8. Herní svět.....	55
4.8.1. Suroviny v herním světe.....	55
4.8.2. Textura herní plochy.....	55
4.8.3. Budovy .....	57
4.8.4. Ohraničení herního světa .....	57
4.9. Systém Frakcí .....	58
5. Uživatelská Dokumentace .....	59
5.1. Přehled.....	59
5.2. Spuštění hry .....	59
5.2.1. Hlavní menu .....	59
5.2.2. Výběr postav.....	59
5.3. Hraní hry.....	61
5.3.1. Počátek hry .....	61
5.3.2. Výroba předmětů .....	67
5.3.3. Dialog Rozkazy .....	72
5.3.4. Neutrální NPC .....	72
5.3.5. Útok na nepřátelskou základnu .....	75
1.1     Vstupy hráčů .....	80
5.3.6. Hráč 1 - Klávesnice .....	80
5.3.7. Hráč 2 - Klávesnice .....	80
5.3.8. Další hráči – Herní ovladač .....	80
6. Vývojová dokumentace .....	82
6.1. Architektura scény Unity.....	82
6.1.1. ScriptableObject .....	83
6.1.2. Prefabs .....	83
6.2. Zpracování Vstupu hráčů .....	83
6.2.1. Vstup pro navigaci UI .....	84
6.3. Interakce s objekty.....	85
6.3.1. NPC .....	85
6.3.2. Hráč .....	85
6.4. Objekty s UI .....	86
6.5. Dialog .....	86
6.6. Předměty.....	87
6.7. Vybavení.....	87
6.8. NPC AI .....	88
6.8.1. Navigace .....	88
6.8.2. Stav světa.....	88
6.8.3. Plány a Node.....	88
6.8.4. Akce.....	89
6.8.5. Cíle .....	90
6.8.6. Vytvoření plánu .....	90
6.8.7. Vykonání plánu .....	91
7. Návody pro vývojáře .....	92
7.1. Jak vytvořit nový předmět .....	92
7.2. Jak vytvořit vybavení .....	94
7.2.1. Jak vytvořit Prefab vybavení .....	95
7.2.2. Jak vytvořit zbraň na dálku a projektil .....	97
7.3. Jak vytvořit efekt .....	99
7.4. Jak vytvořit interaktivní objekty a jejich UI.....	100
7.4.1. UI prefab.....	100

7.4.2. UI komponenta .....	101
7.4.3. Komponenta objektu .....	102
7.5. Jak vytvořit dialog .....	103
7.6. NPC a AI .....	104
7.6.1. Jak vytvořit nový cíl .....	104
7.6.2. Jak vytvořit novou akci .....	106
7.6.3. Jak vytvořit nový Job.....	108
7.6.4. Jak vytvořit nové NPC.....	109
8. Závěr.....	111
8.1. Zhodnocení splnění cílů .....	111
8.2. Potenciální budoucí vylepšení .....	111



# 1. Úvod

Počítačové hry mají často systémy a mechaniky, které jsou abstrakcemi reálného světa. Tyto abstrakce jsou užitečné, protože pro většinu her nedává smysl simulovat každý atom herního světa, nebo světovou ekonomiku. Právě herní ekonomika je často cílem velikých abstrakcí. Mezi časté abstrakce patří:

- 1) obchody, které mají nekonečnou zásobu předmětů, nebo se jejich zásoba za nějaký čas automaticky doplní,
- 2) ceny předmětů, které jsou buď všude stejné, nebo natvrdo zakódované a nezávisí na poptávce a nabídce,
- 3) výroba a přeprava předmětů je často úplně opomenuta.

Další častá herní abstrakce, která se nejčastěji objevuje v RPG [3] hrách, se týká vybavení nepřátelských postav. Humanoidní nepřátelské postavy mají často bojové vybavení v podobě brnění a zbraně, toto vybavení je ale znázorněno pouze v grafické podobě a po porážce takovýchto nepřátel nemá hráč možnost z jejich mrtvol získat žádné vybavení, pokud není jinak specifikováno vývojáři hry. Takováto abstrakce dává často smysl, protože nemusíme vytvářet nový předmět vždy kdy chceme, aby sprite nepřátele používat novou zbraň. Pokud nejsou nepřátelské zbraně lepší, než hráčovy tak je hráč stejně nebude používat a pravděpodobně je pouze prodá a mezitím mu bude toto vybavení pouze zabírat místo v inventáři.

Uvedené abstrakce NPC [1] předmětů a předmětů v ekonomice sice svůj smysl mají, ale

my se domníváme, že by realističtější implementace těchto aspektů mohla vyústit v zajímavé herní situace pro hráče. V rámci této bakalářské práce navrhнемe a naprogramujeme prototyp RPG hry která bude mít realističtější implementaci těchto mechanik. I naše hra však nebude plně realistická a bude také využívat nějaké abstrakce, a proto se pokusíme najít o smysluplný kompromis mezi realismem a hratelností.

V následujících kapitolách si hlouběji rozebereme koncepty a design hry, stanovíme přesnější cíle práce a provedeme analýzu naší implementace.

## 2. Analýza konceptů

V této kapitole se podíváme na detailní implementace herních konceptů a rozhodneme se v jaké podobě bychom je mohli využít a adaptovat pro hru v RPG žánru.

### 2.1. Simulace ekonomiky

Zatímco RPG hry většinou ekonomiku detailně nesimulují, hry žánru *City builder* [4] nebo *Colony simulator* [5] mají ekonomiku často rozvinutější.

Nyní si rozebereme reprezentativní hru z každého z těchto žánrů a prozkoumáme její ekonomické mechaniky a podíváme se, zda se hodí do *RPG* hry.

#### 2.1.1. City builder

Hry žánru city builder většinou simulují město, jeho populaci a ekonomickou aktivitu. Hráč většinou hraje za starostu města a má za úkol vytvořit co nejlepší město. Typickým zástupcem žánru *City builder* je série her *Settlers* (mezi podobné hry patří i *Knights and Merchants* nebo sérii her *Anno*), z níž si vybereme hru *Settlers 2*, u které prozkoumáme její implementaci herní ekonomiky, abychom se mohli rozhodnout, jestli bychom chtěli některé její složky využít v naší hře.

#### Základní popis *Settlers 2*

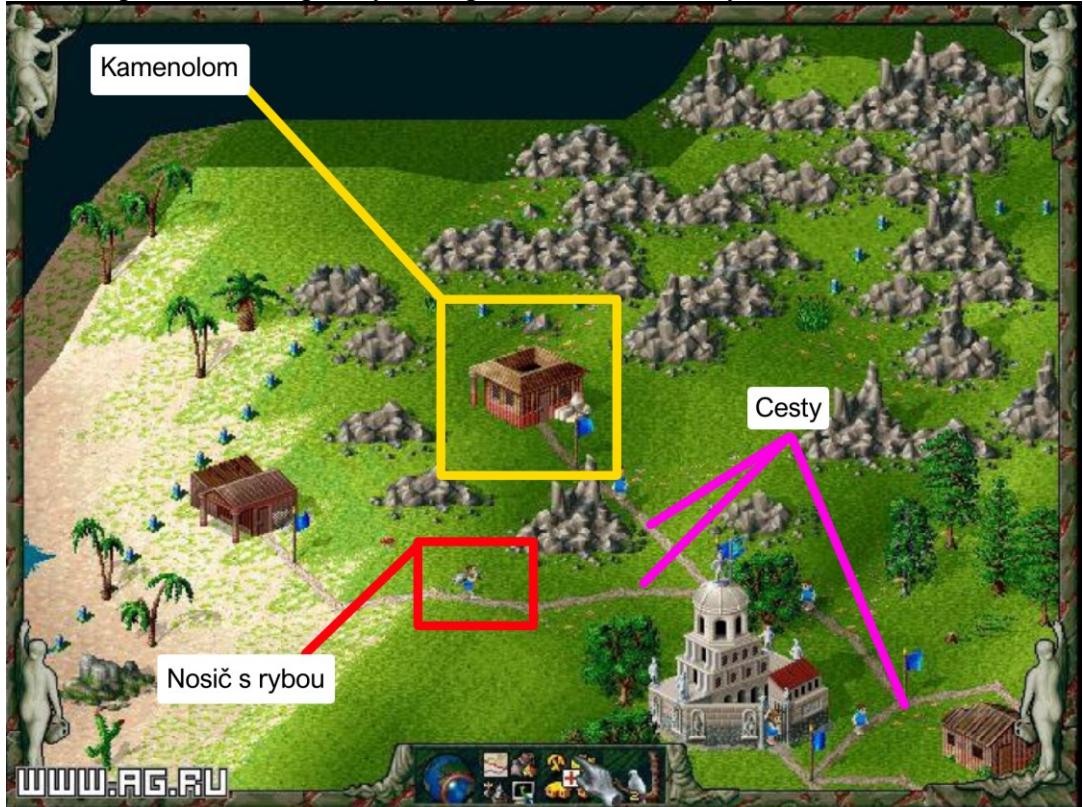
*Settlers 2* spočívá v tom, že hráč buduje budovy, které vyrábějí předměty, buď samy od sebe, z ložisek surovin nebo konverzí z jiných předmětů. Například, kamenolom vyrábí kámen (viz Obrázek 1), farma vyrábí obilí a mlýn vyrábí z obilí chléb. Za některé předměty, například zbraně, můžeme najmout vojáky, které používáme k zabránění území. Stavění budov také stojí předměty, hlavně dřevo a kámen, takže se hráč musí rozhodovat, jestli stavět budovy, které vyrábějí předměty na stavbu budov, aby mohl lépe rozvíjet ekonomiku, nebo předměty na postavení armády, aby mohl zabírat území a ubránit se nepřátelským vojákům. Takovéto rozdělení výroby předmětů nutí hráče plánovat strategicky dopředu.

#### Hlubší analýza ekonomiky *Settlers 2*

Velkou částí ekonomiky hry *Settlers 2* je zahrnutí výrobních řetězců, kde postupně měníme jeden předmět na druhý, dokud nedostaneme vyžadovaný produkt. Například farma vytvoří obilí, ze kterého se ve mlýně vyrobí mouka, ze které se v pekárně vyrobí chléb, který se využije v dole k vytěžení železné rudy, ze které se vyrobí zbraň. A protože některé budovy vyrábějí předměty rychleji, nebo jich potřebují více, aby mohly pracovat tak musejí být postaveny ve správných poměrech, aby byla výroba optimální.

Stavění správných budov je ale jen jednou částí ekonomiky, druhou důležitou součástí je transport předmětů. Budovy musejí být postaveny vedle cest, a po cestách cestují přenašeči předmětů (viz Obrázek 1), což jsou NPC postavy, které přenášejí předměty mezi budovami, podle toho, kde jsou potřeba. Tato NPC standardně přenášejí pouze 1 předmět najednou a neplánují nijak do budoucna. Například, pokud máme 1 železo a dvě kovářské budovy, kde jedna budova potřebuje 1 železo na

vytvoření helmy, ale druhá potřebuje 2 železa na vytvoření meče tak se NPC rozhodne náhodně kam železo donést, i přesto že se z jednoho železa něco užitečné vytvoří pouze v první budově, zatímco ve druhé se bude donekonečna čekat na druhé železo. Tento problém může často nastat, když máme nedostatek důležitého předmětu, např. dřeva, což pak vyústí v pomalou stavbu nových budov.



Obrázek 1  
Screenshot ze hry Settlers 2 zobrazující město hráče – zdroj [2]

### **Ekonomika naší hry**

Velikou výhodou *City builder* her je, že má hráč často přehled o celé ekonomice, a i moc ji ovládat a manipulovat, protože má většinou pohled na hru shora (viz Obrázek 1) a může tudíž vydávat informované příkazy a ekonomiku do detailu řídit. V naší RPG hře, a i v mnoha ostatních, je hráčův pohled omezen na okolí postavy, kterou ovládá, a i jeho schopnost ovlivňovat svět je podobně omezena. Nebudeme tedy chtít, aby měl hráč absolutní kontrolu nad ekonomikou, protože by to pro něho bylo příliš velikou zátěží a neměl by čas soustředit se na standardní RPG mechaniky. Místo toho ekonomiku zautomatizujeme.

### **Automatizace ekonomiky**

Ekonomiku můžeme automatizovat několika způsoby. Jeden z těchto způsobů je použít globální AI [2], které bude mít přehled o celé ekonomice a bude dělat globální ekonomická rozhodnutí, NPC pak budou pouze vykonávat příkazy. Tato implementace je relativně efektivní jak z hlediska využívání herních zdrojů, tak i počítačových zdrojů. Jedna negativní stránka je, že NPC budou reagovat na ekonomické změny, které nastanou kdekoliv v herním světě. Toto není příliš realistické a mi bychom raději chtěli, aby naše NPC reagovala pouze na jejich lokální okolí, protože si myslíme, že to tak bude zábavnější pro hráče.

Toto nás tedy vede k jinému způsobu automatizace, a to implementací NPC, která umí konat samostatná ekonomická rozhodnutí v závislosti na jejich okolí a znalostech. Tato varianta bude teoreticky produkovat realističtější chování NPC, ale přináší sebou i problémy. Plánování akcí ve složité ekonomice pro mnoho NPC může být náročné na počítačové zdroje. A protože se NPC mohou rozhodovat nezávisle na sobě tak se mohou rozhodnou začít vyrábět stejné předměty, což může být problém, když jsou takové předměty drahé nebo je zapotřebí pouze jeden. Tento problém můžeme vyřešit tím, že každému NPC přednastavíme nějakou roli v herním světě, kterou bude vykonávat. Např. můžeme mít pouze 1 kováře, který je jediné NPC, které ková, takže nikdy nevytvorí příliš předmětů. Nebo můžeme mít 2 kováře a každý z nich se může specializovat na kování jiných předmětů ve stejně lokaci (např. zbraně a nástroje). Nebo mohou oba kovat jakékoli předměty, ale budou daleko od sebe a potom nám nemusí duplikace předmětů vadit, protože nám to ušetří čas cestování. Pokud budou mít NPC předem danou roli, tak to nám také mnohonásobně ušetří počítačové zdroje, protože NPC budou mít užší cíle při ekonomickém plánování.

Trochu počítačových zdrojů můžeme ušetřit i tím, že zkrátíme výrobní řetězce. Např. místo toho, abychom z obilí nejprve mleli mouku a až z mouky dělaly chléb tak můžeme chléb pect rovnou z obilí. Toto buď odstraní potřebu pro NPC mlynáře, nebo usnadní plánování pekařů, protože nebudou muset plánovat mletí obilí. Nevýhoda využití této optimalizace je, že bude ekonomika jednodušší a může mít méně zajímavých spojů mezi produkčními řetězci. Pokud budeme mít řetězce délky 2, tak ty nemají žádné meziprodukty, které by mohli sdílet s jinými řetězci. Proto bychom chtěli, aby většina našich řetězců měla délku alespoň 3. Mít délku řetězce větší než 3 v tomto ohledu už nepomáhá, takže řetězce takovýchto délek raději do naší hry nebudeme vkládat.

### Budovy v naší hře

Další komplikací je stavění budov. V *Settlers 2* se rozhodne hráč kde a kdy budovu postavit, NPC pak pouze přinesou požadované suroviny a budova se za nějaký čas postavy a začne v ní pracovat NPC. Stavění budov v RPG hře se sebou přináší komplikace, které ve světě *City builders* neexistují. Implementace stavebního systému má tyto potíže.

- 1) Co když stojí hráč na místě stavby? Vytlačíme hráče pryč, nebo počkáme až odejde? Co když tam není hráč ale NPC, které zase čeká na nás, dokud se my neposuneme?
- 2) Kde a kdy se rozhodnou NPC postavit budovu? Stavění budov je většinou drahá záležitost (záleží na designu hry) a vyžaduje vícero NPC které dodávají potřebné zdroje a stavějí budovu (zase záleží na implementaci a vizi hry, mohli bychom klidně mít jedno NPC a zdroje odečítat z globálního skladiště, pokud bychom chtěli). Nechceme, aby se vícero NPC rozhodlo stavět stejnou budovu, když jedna postačí.
- 3) Kde získáme NPC, které v těchto budovách budou pracovat? Kdybychom spawnuly nové NPC pro každou budovu tak začneme za chvíli mít potíže s procesováním tolika komplikovaných NPC.

Tyto problémy jsou značné a vyžadovaly by vynaložení velkého množství práce, ale my se domníváme, že se herní zážitek dostatečně nezlepší, aby se řešení těchto problémů vyplatilo. Protože stavění budov není nutnou součástí naší vize pro detailnější předmětovou herní ekonomiku, rozhodli jsme se, že herní svět naší hry bude mít nezničitelné budovy, které budou součástí herní mapy.

### 2.1.2. Colony simulator

Hry žánru *Colony simulator* se soustředí na budování malého osídlení, např. kolonie, s malým počtem obyvatelům. Cílem hráče je vést kolonii k prosperitě a zajistit štěstí jejího obyvatelstva. Typickým zástupcem tohoto žánru je hra *Dwarf Fortress*, mezi další patří například *Rimworld* nebo *Oxygen not included*. Hru *Dwarf Fortress* blíže prozkoumáme v další části.

#### Popis ekonomiky *Dwarf Fortress*

Ve hře *Dwarf Fortress* vládne hráč pevnosti trpaslíků. Trpaslíci se řídí relativně komplexním emočním systémem a často nemusejí hráče poslouchat. Hra spočívá v budování pevnosti, většinou jejím vytesáním ze skály, a budováním ekonomických struktur za cílem výroby předmětů, aby byli hráčovi trpaslíci spokojeni s jejich kvalitou života. Na rozdíl od *Settlers 2* buduje hráč pracovní stoly (např. kovadlina, tesařský stůl atd.) místo budov, ve kterých pracují jeho trpaslíci a vyrábějí různé předměty. Tyto pracovní stoly mohou vyrábět mnoho různých předmětů a je na hráči, aby zadal správné výrobní příkazy. Např. v *Carpenter's workshop* se můžou vyrábět postelety, kbelíky, hrnky atd.

#### Analýza efektu realistických předmětů na ekonomiku *Dwarf Fortress*

Celkově jsou základní ekonomické principy ekvivalentní *Settlers 2*, až na jeden důležitý rozdíl: v *Settlers 2* pouze najmeme své vojáky jednorázovým zaplacením předmětů, pročež jsou tyto předměty navždy vymazány a z vojáka je už nikdy nedostaneme zpět. *Dwarf Fortress* ale simuluje své NPC trpaslíky do mnohem většího detailu, každý trpaslík má možnost nosit oblečení a může držet předměty v rukou, pokud mu tedy přikážeme stát se vojákem tak si půjde obléct brnění a vzít zbraň. Poté mu kdykoliv můžeme přikázat, aby se předmětu zbavil, abychom je mohli přidělit jinému trpaslíkovi, nebo je zase přetavit v kov a vyrobít z nich jiné předměty. Pokud voják zemře tak jeho vybavení zůstane na místě jeho smrti, a bude možno ho získat zpět. Nepřátelé jsou simulováni stejným způsobem, a pokud je zabijeme tak získáme všechno vybavení, kterými na vás útočili. Takovéto prolnutí ekonomiky výroby předmětů a vojenských jednotek je zajímavou částí hry a značně ovlivňuje strategie a chování hráčů. Jedním zajímavým příkladem strategie je vyhlašování válek, aby proti vám byly poslány armády, když dokážete tyto armády porazit tak slouží jako značné zdroje kovů.

Toto je v kontrastu s obvyklou implementací smrti a předmětů v RPG hrách, kde se nepřátelé po porážce buď vypaří a nic po sobě nezanechají, nebo po sobě ponechají mrtvolu, ze které můžete dostat nějaké náhodné předměty, které nemusejí vůbec souviset s tím, jak váš nepřítel graficky vypadá. Kdyby ale grafika NPC, a to co z nich skutečně můžete získat spolu souviselo, tak to by mohlo značně zvýšit míru zaujetí hráčů do hry. Pokud hráč vidí, jak NPC medik používá kouzelné lektvary k oživování NPC, tak se může rozhodnou toto NPC zabít, aby tyto lektvary získal. A čím lepší takový předmět, tak tím zajímavější bude situace. Pokud hráč vidí rytíře s jedinečným kouzelným mečem, tak bude muset tohoto rytíře zabít, aby meč získal, což může být obtížné, když má nepřítel tak silnou zbraň. Ano, takováto situace může nastat i v normálních RPG, ale v nich musejí tyto situace vývojáři manuálně vytvořit, tím že důležitým NPC dávají zajímavé předměty do jejich náhodných dropů. Pokud ale implementujeme předmětový systém podobný *Dwarf Fortress*, tak mohou tyto

situace nastat přirozeně a organicky během průběhu hry, což má potenciál být mnohem zajímavější než skriptované situace.

## 2.2. NPC a jejich vztah k předmětům

Existují některé hry, které implementují předmětový systém podobný *Dwarf Fortress*, například hra *Blade and Sorcery*, která je akční RPG, kde bojujete v aréně proti vlnám nepřátel. Nepřátelé mají různé zbraně a když je zabijete tak z nich jejich zbraně vypadnou a vy je můžete ze země sebrat a začít používat. Tato hra je ale pouze akční RPG, a nepřátele spawnuje přímo se zbraněmi a teda ignoruje ekonomickou stránku věci, o kterou se my ale zajímáme, protože NPC už mají veškeré potřebné vybavení.

Aby mohla ekonomka mít dopad na vojenskou stránku hry, tak by NPC měla využívat předměty, které ekonomika vytvoří. Abychom zaručili, že se toto stane tak rozhodneme, že se naše NPC budou spawnovat bez předmětů. Aby mohli pak bojovat, tak si nejprve budou muset vyzvednout vybavení, které ekonomika vyrobila. Toto znamená, že množství vybavení ve hře bude omezeno kapacitou herní ekonomiky. Abychom zabránily přílišnému nedostatku vybavení, což by mohlo způsobovat nezábavné chování NPC, např postávání v řadě nebo útočení bez zbraní, tak rozhodneme, že počet bojových NPC bude ve hře konstantní. Místo toho abychom periodicky spawnovali nové bojové NPC ve vlnách.

## 2.3. Umělá inteligence v ekonomice

Jak *Settlers 2* tak i *Dwarf Fortress* mají jednoduché AI z ekonomického hlediska. Jejich NPC vykonávají právě jednu akci, kterou si buď vyberou ze seznamu akcí (které vygeneroval pracovní systém), nebo kterou jim hráč explicitně přidělí, ale nedokáží sami řetězit více akcí za sebou, a tudíž nedokážou plánovat dopředu. Toto může být problematické, protože se kvůli špatnému plánování ekonomika může zaseknout, například, protože výrobce zbraní sebral poslední železo, aby vyrobil oštěp, přestože nemá k dispozici dřevo, aby ho dokončil, a nenechá nic pro výrobce nástrojů, který by mohl vyrobit krumpáč, který by nedostatek železa vyřešil. Protože naše RPG nebude mít hráče, který by ekonomiku mohl snadno manuálně opravit, tak bychom chtěli, aby naše AI bylo robustnější a mohlo plánovat svoje akce. AI by poté mohlo předejít ekonomickému kolapsu, například tak, že nezačne brát a syslit předměty, dokud jich nebude dost pro zaplacení celé ceny výroby konečného produktu.

Některá naše NPC budou mít roli vojáka, a tak budou potřebovat zbraň a brnění. Jelikož se s takovými předměty neobjeví tak je budou muset získat z ekonomiky. Může se ale stát, že vojáci budou umírat tak rychle, že nebude dostatek zbraní, což může vyústit v to, že některá NPC nebudou moct bojovat, a tak budou jenom nečinně postávat. To může vypadat divně, a také to může pro hráče být ostravné, když nevidí, že by NPC něco dělali. Proto bychom chtěli, aby NPC věděla, jaké předměty potřebují a mohli si sami naplánovat postup akcí, které jim zajistí potřebné předměty (např. vytěží železo, ze kterého vyrobí zbraň).

## 2.4. Umělá inteligence v boji

Naše práce se soustředí na ekonomickou stránku realistických předmětů a na to, jak to ovlivní hráčův zážitek. Samotná mechanika boje nás ale moc nezajímá, nás spíše zajímá, co se stane potom, co je NPC poraženo a co se stane s jeho vybavením. Naše požadavky na bojové AI budou tedy minimální, a bude stačit, pokud bude schopno hráče základně ohrozit. Naše požadavky budou:

- 1) pronásledování nepřátele v blízkosti,
- 2) schopnost obcházet překážky, aby se NPC mohlo dostat k nepříteli,
- 3) smysluplné používání předmětů v boji (např. útočení pomocí zbraně, vypití oživovacího lektvaru).

## 2.5. Shrnutí návrhu ekonomiky naší hry

Naše ekonomika bude založená na produkčních budovách, ve kterých budou NPC vyrábět předměty. Produkční budovy budou součástí herního světa a budou v něm vystavěné během level designu. NPC budou mít vlastní AI a v závislosti na jejich předem dané roli se budou sami rozhodovat ve které budově pracovat a jaké předměty vyrábět. NPC a hráči budou moct využívat vyrobené předměty, aby dosáhly herních cílů.

## 2.6. Zapojení hráče

Aby se naše simulace světa mohla stát hrou tak potřebuje do svého dění zapojit hráče. Už jsme se rozhodli že naše hra bude RPG, teď stačí jenom specifikovat roli hráče a průběh hry.

### 2.6.1. Role hráče

Důležitou částí naší práce a naší hry je, že jakékoliv vybavení, které má NPC je persistentní a může je používat i hráč. Abychom dále propojili hráče s NPC postavami tak bychom chtěli, aby hráč interagoval s herním světem stejným způsobem jako NPC. Hráč bude tedy nejenom moct používat stejné předměty jako NPC, ale bude se také moct přímo zúčastnit ekonomiky, např: těžením surovin nebo vyráběním předmětů.

### 2.6.2. Druhy interakce mezi hráčem a NPC

Hráčův vliv na průběh hry je omezen na okolí jeho herní postavy. Toto nám nutně nevadí, ale mohli bychom chtít, aby měl hráč trochu rozšířenější vliv na chování NPC, aby mohl lépe koordinovat různé ekonomické a bojové operace. Proto umožníme hráči dávat příkazy NPC postavám, pomocí dialogového systému. Např. po interakci s vojákem mu bude moct hráč dát příkaz, aby zaútočil na nepřátele v jiné části světa, nebo aby hráče následoval a pomáhal mu v boji.

### 2.6.3. Volba pod-žánru

Abychom lépe specifikovali podobu naší hry, tak si vybereme pod-žánr do kterého bude patřit. Tyto pod-žánry nejsou oficiální, ale jsou námi vybrány, protože se každý soustředí na jiný aspekt RPG her:

- 1) *příběhové RPG,*
- 2) *open world,*

### 3) bojová aréna.

Příběhové RPG se, jak název napovídá, soustředí na prozkoumávání příběhu. Příběh může být předepsaný nebo by se mohl organicky vyvinout interakcemi mezi NPC a hráčem. Naše implementace NPC ekonomiky se sice dá využít v obou těchto variantách, ale k žádné se nijak zvlášť nehodí. V předepsaném příběhu by hráči raději sledovali příběh, než aby věnovali pozornost ekonomice. Příběh generovaný interakcemi mezi NPC zní slibně, ale naše NPC AI je specializována na ekonomickou stránku hry, ale k příběhové generaci by se víc hodilo nějaké AI, které by bylo specializované k takovému účelu. V *příběhových* RPG se také často mění leveley, a naše ekonomika by pravděpodobně neměla dost času se rozvinou nebo mít značný dopad na hru. Samozřejmě ale záleží na specifické implementaci ekonomiky, a rychlá ekonomika by mohla fungovat i v *příběhových* RPG.

*Bojové Arény* se soustředí hlavně na boje v omezených prostorech, a ekonomika v nich často nefiguruje, takže se nám tento druh RPG moc nehodí.

*Open World* RPG se odehrávají v jednom persistentním a většinou rozsáhlém světě. Hráč má v těchto hrách často mnoho možností co dělat a velikou svobodu se rozhodnout jakou část mapy prozkoumat. Rozsáhlý a persistentní svět umožní naší ekonomice se vyvinout a bude tím pádem mít veliký dopad na průběh hry. Hra má také dost prostoru díky velikosti světa pro mnoho produkčních budov a budeme moci mít obydlí v herním světě, která jsou specializovaná na výrobu specifických předmětů, což může hráč brát v úvahu, když se rozhoduje, kam v herním světě cestovat, což může být zajímavé a smysluplné herní rozhodnutí. Problém, který může s velikou mapou nastat je veliká náročnost počítání rozhodnutí všech NPC v celém herním světě. Moderní hry většinou aplikují metodu *Level of Detail* [3] (dále LoD) pro ulehčení tohoto problému, a to tak, že čím dále od hráče, nebo od jeho zorného pole, tím abstraktněji a hruběji se počítají rozhodnutí NPC. Například produkce budovy může být zjednodušena na průměrný příjem, když je hráč daleko, a když je hráč blízko tak se produkce může realisticky simulovat. My bychom také mohli implementoval LoD optimalizace, ale nebudou pro nás moc efektivní, a to, protože naše NPC jsou relativně komplikované a jejich akce nejsou snadno předvídatelné, což může vyústit ve značné nepřesnosti při nízkém LoD. Přestože nám tato technologie moc nepomůže tak si i tak rozhodneme implementovat hru s otevřeným světem, budeme si pouze muset dát pozor aby bychom ho neudělali příliš veliký, jinak nám dojdou počítacové zdroje.

#### 2.6.4. Singleplayer vs multiplayer

Musíme se rozhodnou, jestli naše hra bude pouze pro jednoho hráče nebo pro více hráčů. RPG hry se hodí jak pro singleplayer, tak i multiplayer, toto platí i pro naši hru. My si myslíme, že naše detailnější implementace ekonomiky, bude lépe vynikat v přítomnosti vícera hráčů, a že interakce mezi hráči a ekonomikou budou mít pozitivní dopad pro naši hru, a proto vytvoříme hru pro více hráčů.

#### Druh multiplayeru

Pokud budeme mít více hráčů, tak budou hráči spolupracovat, budou bojovat každý sám za sebe, nebo budou hráči rozděleni do týmů? Každý z těchto přístupů může fungovat. Kooperativní RPG hry jsou od oka běžnější než ostatní variace, tak raději vybereme rozdělení hráčů do dvou soupeřících týmů, aby bychom vytvořili něco unikátnějšího.

## Druh soupeření

RPG je různorodý žánr her, a i kompetitivní RPG mají tedy mnoho různých způsobů soupeření. Hráči mohou být rozděleni do týmů, ale stále mít stejný cíl, například poražení padoucha, a ten tým který ho první porazí vyhraje. Soupeření nemusí být ani bojového charakteru, hráči mohou soupeřit v počtu splněních *questů*[4] nebo hodnotě vyrobených předmětů. Naše hra je sice ekonomicky založena, ale je hlavně zaměřena na vyrábění předmětů pro vojenské účely, takže by pro naši hru bylo lepší, kdyby týmy soupeřily z bojového hlediska a nejlépe proti sobě. Rozdělíme tedy hráče a NPC do dvou týmů, a každý tým bude mít svoji vlastní nezávislou ekonomiku. Jelikož jsme se už rozhodli, že naše RPG bude *open world* tak do herního světa vložíme kromě těchto dvou týmů také neutrální lokace s nezávislými NPC, které budou hlídat nebo vyrábět zdroje, které budou týmy vyhledávat.

## Podmínka vítězství

Nyní musíme vybrat podmínu vízeztví, která určí, jak může tým vyhrát. Na výběr veliké množství podmínek, ale my si vybereme z těchto 3 standardních: „*Král kopce*“ (*King of the Hill*), „*Souboj na smrt*“ (*Deathmatch*) a „*Zmocni se vlajky*“ (*Capture the Flag*).

Podmínka *Král kopce* vyžaduje ovládáním předem daného teritoria, ovládání je dáné přítomností hráčů (nebo jeho jednotek), hráč musí udržet teritorium po nějaký čas, aby byl prohlášen vítězem.

Podmínka *Souboj na smrt* vyžaduje buď zabítí každého nepřátelského hráče, nebo zabítí daného počtu nepřátele.

Podmínka *Zmocni se vlajky* vyžaduje ukradnutí vlajky (nebo jiného předmětu) nepřátelského týmu a její donesení do své základny.

Pro naši hru si můžeme vybrat jakoukoliv podmínu, náš herní koncept se dá aplikovat tak, aby využíval kterékoli podmínce, proto si vybereme *Zmocni se vlajky*, čistě protože ji Autor preferuje.

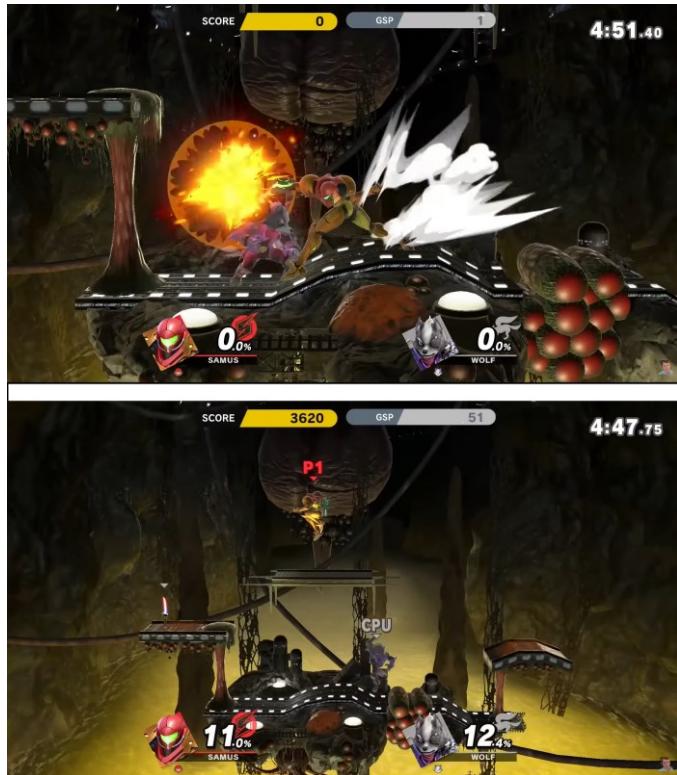
### 2.6.5. Detaily implementace multiplayeru

Nyní se musíme rozhodnout nad implementací multiplayeru po technické stránce. Na výběr máme primárně multiplayer přes internet a multiplayer na stejném počítači. Multiplayer přes internet vyžaduje, aby každý hráč měl svůj vlastní počítač a také vyžaduje po hráčích připojení k internetu. Multiplayer na stejném počítači bude potřebovat herní ovladače pro hru pro více než dva hráče, a hráči budou také moci vidět stejné informace jako ostatní hráči, kvůli sdílené obrazovce. Sdílení informací by mohlo naší hře vadit, protože může snadno překazit různé tajné operace, ale může i být výhodou pro naši hru, protože hráči vidí pouze okolí kolem své postavy, takže často nevědí, co se děje na jiných částech mapy. Toto sdílení informací umožnuje hráčům reagovat na sebe i přes velikou vzdálenost, což může využít v dynamičtější průběhu hry. Obecně si myslíme, že se nám lépe hodí multiplayer na stejném počítači.

#### Místo zobrazení informací

Nyní musíme vybrat, jestli použijeme *rozdelenou obrazovku* (*split screen*) nebo *sdílenou obrazovku* (*shared screen*) (existují i jiné varianty než tyto dvě). Sdílená obrazovka zobrazuje všechny informace ve sdílené sekci a zobrazuje relevantní podmnožinu herního světa pro všechny hráče. Tato metoda funguje dobře, když je relevantní podmnožina světa malá a když operují hráči blízko sebe, nebo když je svět tak malý, že se dá zobrazit celý, často se používá v bojových hrách jako například

*Mortal Combat* nebo *Super Smash Bros Ultimate* (viz Obrázek 2). Metoda rozdělené obrazovky rozděluje obrazovku na díly, kde každý díl zobrazuje relevantní informace pouze pro jednoho hráče, což umožnuje více hráčům se soustředit na různé oblasti herního světa najednou (viz Obrázek 3). Protože plánujeme mít relativně velikou mapu a naši hráči se budou soustředit často na různé části herního světa, tak si vybereme metodu rozdělené obrazovky.



Obrázek 2

Obrázek ukazující změnu kamery při jiných pozicích herních postav, v horní části jsou postavy bliže sobě, a tak je kamera přiblížená, v dolní části jsou postavy daleko od sebe, a tak je kamera oddálená. Zdroj [14]



Obrázek 3

Obrázek ukazující split screen se dvěma hráči. Zdroj [15]

## 2.7. Cíle práce

Cíle naší práce jsou:

- 1) naprogramovat RPG hru se zaměřením na předměty a vybavení postav,
- 2) chceme, aby předměty byly persistentní v herním světě, tj. aby když postava umře, tak z její vybavení a předměty v jejím inventáři budou použitelné ostatními postavami, a nezmizí z herního světa, jak se většinou stane v tradičních RPG hrách.
- 3) implementovat systém pro těžení surovin a vyrábění předmětů,
- 4) implementovat NPC, které těží, vyrábějí, a prodávají předměty,
- 5) implementovat NPC, které používají předměty a vybavení a jsou schopny boje.

## 2.8. Přispěvatelé

Lukáš Juraj Koprda vytvořil většinu spritů, které byly použity v naší hře.

# 3. Dokument Herního Designu

V této kapitole si do detailu znázorníme design hry pro budoucí referenci. K designu využijeme již už zanalyzované herní koncepty z předešlé kapitoly a doplníme všechny potřebné detailey abychom dostali hrátelnou hru.

## 3.1. Koncept hry

Hra je pro více hráčů na stejném zařízení s rozdělenou obrazovkou. Grafika je 2D a pohled hráčů bude seshora dolů. Hra bude boj mezi dvěma týmy hráčů v herním módu *Zmocni se vlajky*. Týmy hráčů budou mít přidělené NPC, které budou těžit zdroje, vyrábět vybavení a bojovat proti nepřátelskému týmu. V herním světě budou také existovat neutrální frakce se svými vlastními NPC, se kterými budou hráči moct uzavřít spojenectví, díky kterému budou NPC spojenecké frakce bojovat za hráčův tým.

## 3.2. Postava

Ve hře bude existovat koncept postavy. Postavy jsou buď ovládány hráčem, nebo to jsou NPC ovládaný vlastní AI. Každá postava má životy, peníze, inventář a vybavení. S některými NPC postavami můžou hráči komunikovat skrz dialogové okno.

### 3.2.1. Vzhled

Postavy jsou znázorněni portrétem tváře postavy ze předu.

### 3.2.2. Pohyb

Pohyb bude okamžitý, bez akcelerace nebo decelerace, a budeme chtít, aby byl portrét a vybavení postavy natočené směrem, kterým se pohybujeme. Změna natočení postavy bude stejně tak plynulá. Pro větší realismus se postavy budou moci tlačit a ovlivňovat jako fyzické objekty.

### 3.2.3. Rasa

Každá postava má přiřazenou rasu. Rasa postavy určuje rychlosť pohybu počet životů, fyzickou velikost a odolnost vůči druhům poškození.

### 3.2.4. Životy

Každá postava má rasou daný počet životů. Pokud postava utrpí zranění tak se počet životů sníží a pokud dosáhne 0 tak postava umře.

### 3.2.5. Zlato

Pouze hráčské postavy mají zlato. Zlato je reprezentované celočíselnou hodnotou a nemá žádnou maximální hodnotu. Zlato je měna, kterou hráči používají pro nákup vybavení.

### 3.2.6. Inventář

Každá postava má inventář, ve kterém může skladovat předměty. Každá postava bude mít inventář s kapacitou 9 předmětů. Postavy mohou předměty ve svém inventáři *použít*, pokud jsou použitelné, nebo je *využít*, pokud se jedná o vybavení.

Menu inventáře je mřížka velikosti 3x3, předměty jsou reprezentovány vlastní ikonou a názvem, pro hlubší analýza menu viz Hráčův inventář 3.10.1.

### 3.2.7. Vybavení

Existují 3 druhy vybavení, *ruka* (zbraně atd.), *tělo* (brnění) a *jiné* (talismány, kouzelné prsteny atd.)

Pokud je předmět *využitelný*, tak má přidělený právě jeden druh vybavení.

Když postava *využije* předmět tak to znamená, že tento předmět přesunula do pole pro vybavení. Předmět může být přesunut pouze do pole pro vybavení, které má stejný druh vybavení jako předmět.

Každé pole má přidělen právě jeden druh vybavení. Každá postava má 5 míst pro vybavení: 2 pro *ruce*, 1 pro *tělo* a 2 pro *jiné*.

Vybavení může být *využito* pouze v jednom místě vybavení, což aplikuje jeho efekty

### 3.2.8. Smrt

Když se počet životů dostane na hodnotu  $\leq 0$  tak postava umře. Po smrti postavy budou všechny předměty uloženy v jejím inventáři a předměty jí *využívány* instanciovány jako pick-upy na místě smrti.

Pick-upy jsou reprezentace předmětů v herním světě, a postavy je mohou uložit do svého inventáře, když jsou v dostatečné blízkosti tím, že s pick-upem provedou interakci.

## 3.3. Předmět

Předměty mohou postavy nosit ve svých inventářích a mohou je skladovat do truhel. Hráči je také mohou prodávat nebo kupovat z obchodů, zatímco NPC postavy považují obchody za normální truhly. Předměty mohou být zdroje, nástroje, zbraně, brnění, kouzla, lektvary atd. Zbraně, nástroje a brnění jsou vybavení, které může postava *využít*, brnění zlepšuje obranu postavy a nástroje a zbraně mohou být *aplikovány*, aby byli *aktivovány* její efekty (např. útok mečem). Některé předměty, jako například kouzla a lektvary mohou být *použity* přímo z inventáře, což aplikuje jejich efekty.

Předmět bude mít tato data:

- sprite,
- jméno,
- popisek,
- cenu,
- seznam efektů po použití,
- příznak určující, jestli je použitelný pouze jednou.

Vybavení má ještě tyto data navíc:

- efekty při využívání,
- druh vybavení,

### 3.3.1. Využívání předmětů

Když postava přesune předmět z inventáře do místa vybavení, tak ho začne *využívat*. Začátek *využívání* aplikuje efekty *využívání* předmětu (např. brnění zvýší obranu) a zobrazí předmět v herním světě (např. meč se zobrazí v ruce postavy)

Využívané předměty druhu *ruka* se dají *aktivovat* tlačítkem pro levou nebo pravou ruku prospektivně. *Aktivace* ručního předmětu aktivuje efekt předmětu, v případě zbraně se jedná o zahrání animace a udělení poškození zasáhnutím cílům.

### 3.3.2. Pick-upy

Pickupy jsou reprezentace předmětů v herním světě, s postavami ani jinými objekty nekolidují, ale aby měli větší fyzickou přítomnost (a nebyly jenom obrázky na zemi) tak budou fyzikálními objekty a budou ovlivněny AOE (velko plošnými) efekty, například výbuchem ohnivé koule, které je po zemi posunou. Postavy je mohou přesunout do svých inventářů, pokud s nimi zinteragují.

### 3.3.3. Vítězné předměty

Aby tým mohl vyhrát tak musí ukrást vítězný předmět druhému týmu. Tým 1 má královskou korunu a tým 2 má atomovou tyčinku. Tyto předměty začnou hrát v základnách svého týmu v podobě pick-upu.

### 3.3.4. Seznam předmětů

Log:

Popis: It's a wooden log.

Cena: 5.

Plank:

Popis: It's a wooden plank.

Cena: 10.

Iron Ore:

Popis: It's iron ore.

Cena: 5.

Iron Bar:

Popis: It's a bar of iron.

Cena: 10.

Crystal Shard:

Popis: A raw magical crystal shard.

Cena: 10.

Refined Crystal:

Popis: Refined magical crystal.

Cena: 20.

Fireball:

Popis: Spell token that will shoot out a ball of fire.

Cena: 60.

Efekty:

Vystřelí projektil směrem, kam se postava dívá. Když projektil do něčeho narazí tak exploduje a udělí 50 *ohnivého* poškození a *knockback* (odhození zpět) všem objektům v kruhu o poloměru 3. Poškození a *knockback* se zmenšuje lineárně vůči poloměru.

Jednorázové:

Ano

Iceshard:

Popis: Spell token that will shoot out an icicle.

Cena: 50.

Efekty:

Vystřelí projektil směrem, kam se postava dívá. Projektil udělí 25 *ledového* a 25 *bodacího* poškození prvnímu objektu, do kterého narazí.

Jednorázové:

Ano.

Health Potion:

Popis: Heals your wounds if you drink it.

Cena: 30.

Efekty:

Přidá postavě 30 životů.

Jednorázové:

Ano.

Shem:

Popis: An ancient artefact used to control golems.

Cena: 1000.

Crown:

Popis: Crown of rule. It can be used to heal.

Cena: 999.

Efekty:

Přidá postavě 20 životů.

Jednorázové:

Ne.

Atomic Snack:

Popis: An object of great power.

Cena: 999.

Efekty:

Stejně jako u předmětu Fireball.

Jednorázové:

Ne.

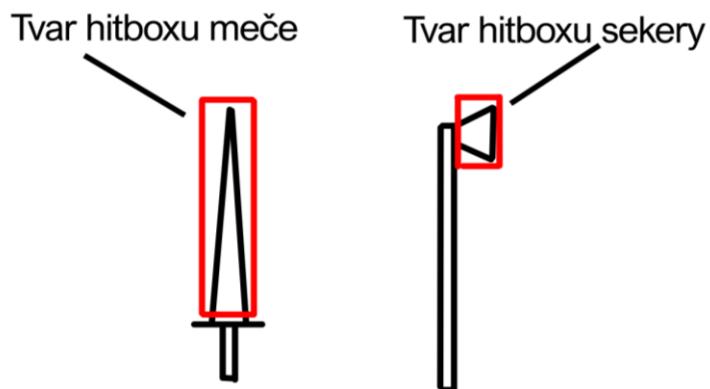
### 3.4. Zbraně, brnění a zranění

Zbraně mohou udělovat různé druhy poškození. Druhy poškození jsou *tupé*, *bodací*, *sekací*, *ohnivé*, *ledové* a *magické*.

Cíle poškození mohou mít slabost nebo odolnost vůči specifickým druhům poškození, tato vlastnost je reprezentována jednou hodnotou pro každý druh poškození zvlášť, tato hodnota působí jako násobitel poškození svého druhu.

### 3.4.1. Zbraně a jejich vlastnosti

Zbraně mají kromě vlastností předmětu ještě navíc hit-box (viz Obrázek 4) a hit efekty. Hit-box je část herního prostoru, která detekuje kolize s herními objekty. Každá zbraň má vlastní tvar hit-boxu, který odpovídá nebezpečné části zbraně, např. meč má hit-box po celé čepeli, a sekera pouze kolem hlavy (viz Obrázek 4). Hit-box je standardně vypnut a zapíná se pouze během animace útoku. Hit efekty jsou efekty, které nastanou, když hit-box detekuje kolizi s postavou nebo objektem, efekty mohou být aplikovány jak na kolidující postavu/objekt tak i na útočícího hráče. Nejběžnější efekty jsou efekty poškození, které udávají kolik a jakého druhu uděluje zbraň poškození a knockback efekt, který zasaženou postavu posune o malou vzdálenost a na velmi malý čas ji také omráčí (její akce jsou ignorovány a nemůže být ovládána). Hodnoty pro délku omráčení a vzdálenost posunutí nebudeme udávat, protože pro nalezení přesných hodnot budeme muset play testovat. Předem ale víme, že omráčení by mělo být dost krátké, aby se postava mohla vyhnout druhému útoku a posunutí přiměřeně veliké vůči použité zbrani (meč vás posune méně než kladivo). Maximální míra posunutí by měla být 3krát délky postavy.

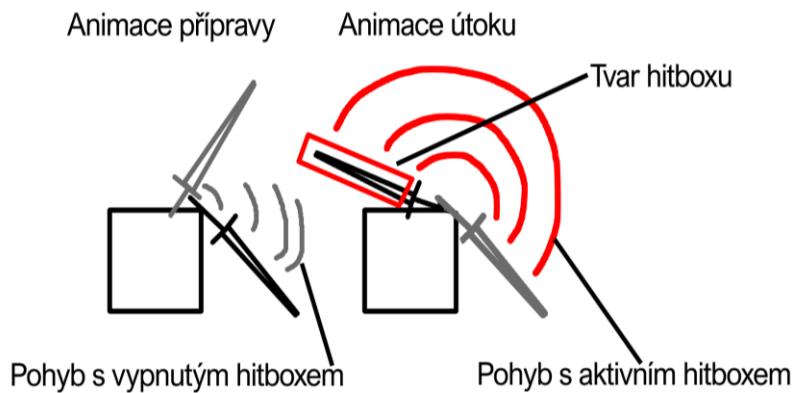


Obrázek 4

Každá zbraň má přidělenou animaci, která je zahrána po použití zbraně. Animace odpovídají způsobům používaní zbraně, a protože se mnoho zbraní používá stejným způsobem, tak jsme jím také dali stejnou animaci. Následuje seznam všech zbraní organizovány dle jejich animace.

### Sečná animace

Zbraně v této sekci mají sečnou animaci. Animace (viz Obrázek 5) se skládá z přípravy (náprahu), seku a návratu (do počáteční pozice).



**Obrázek 5**  
Znázornění sečné animace

#### Axe:

Popis: Used to chop down trees.

Cena: 50

Efekty:

poškození = 20 sekací

#### Pickaxe:

Popis: Used to mine ore.

Cena: 50

Efekty:

poškození = 20 bodací

#### Sword:

Popis: You can use this to hurt people.

Cena: 50

Efekty:

poškození = 15 sekací, 15 bodací

#### Hammer:

Popis: Used for breakings things... and people.

Cena: 60

Efekty:

poškození = 25 tupé

#### Rainbow Sword:

Popis: A legendary sword of great power.

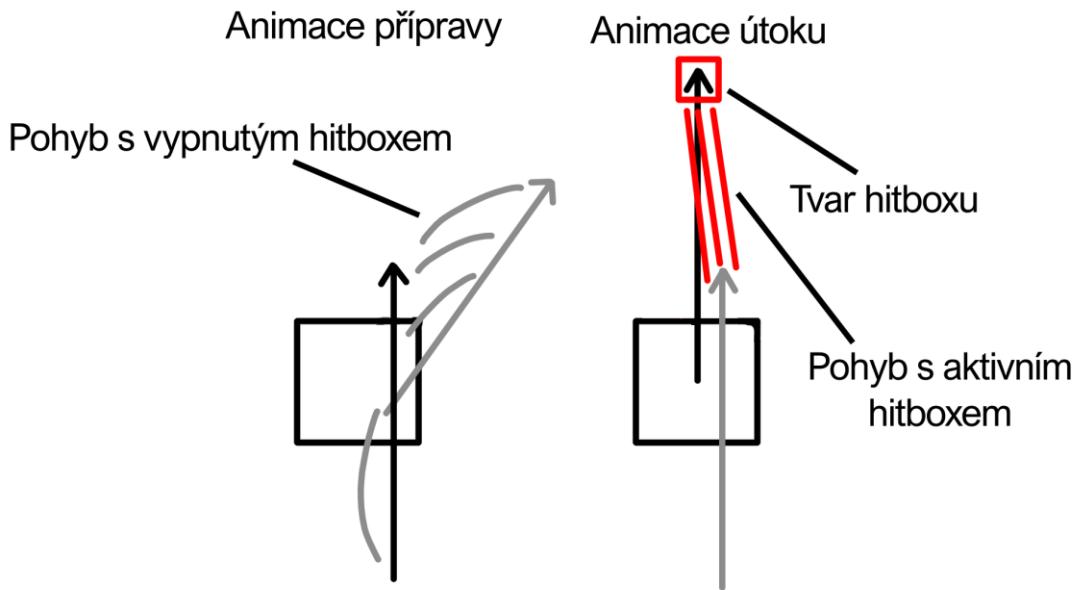
Cena: 666

Efekty:

poškození = 15 sekací, 15 bodací, 15 ohnivé, 15 ledové

#### Bodací animace

Zbraně v této sekci mají bodací animaci. Animace (viz Obrázek 6) se skládá z přípravy (náprahu), bodnutí a návratu (do počáteční pozice).



**Obrázek 6**  
Znázornění bodací animace

Spear:

Popis: Used to hit enemies from just out of their reach.

Cena: 40

Efekty:

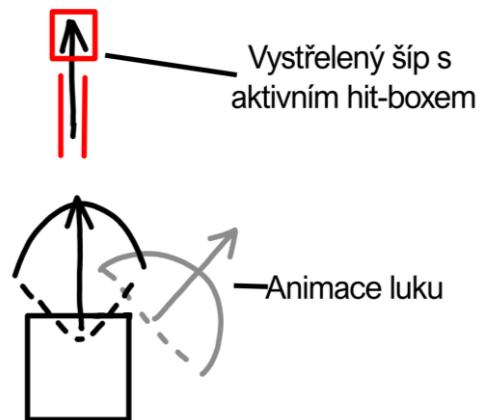
poškození = 30 *bodaci*

### Animace lukostřelby

Na rozdíl od předešlých animací, kde pouze měníme rotaci a pozici zbraně, tak zde navíc používáme klasickou animaci výměny obrázků za vytvořením iluze plynulé změny. Specificky měníme sprite luku tak, aby se s šípem natahoval během translačního pohybu (viz Obrázek 7).

S lukem musíme také představit nový koncept projektileů. Luky sami o sobě žádné poškození nedávají, a mají jediný efekt, vytvoření projektilelu. Projektil cestuje vystřeleným směrem a má svůj vlastní hit-box a vlastní efekty, které při kolizi aplikuje. Efekty mohou být stejné jako u normálních zbraní, ale máme i nové efekty, například odstranění samotného projektilu.

## Animace útoku



**Obrázek 7**  
Znázornění animace lukostřelby

Bow:

Popis: Shoots arrows.

Cena: 70.

Efekty:

Vystřelí šíp rychlostí 15 m/s

Arrow:

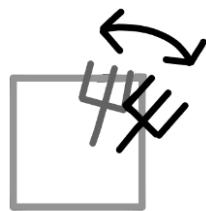
Sebe zničí se po 30s.

Efekty:

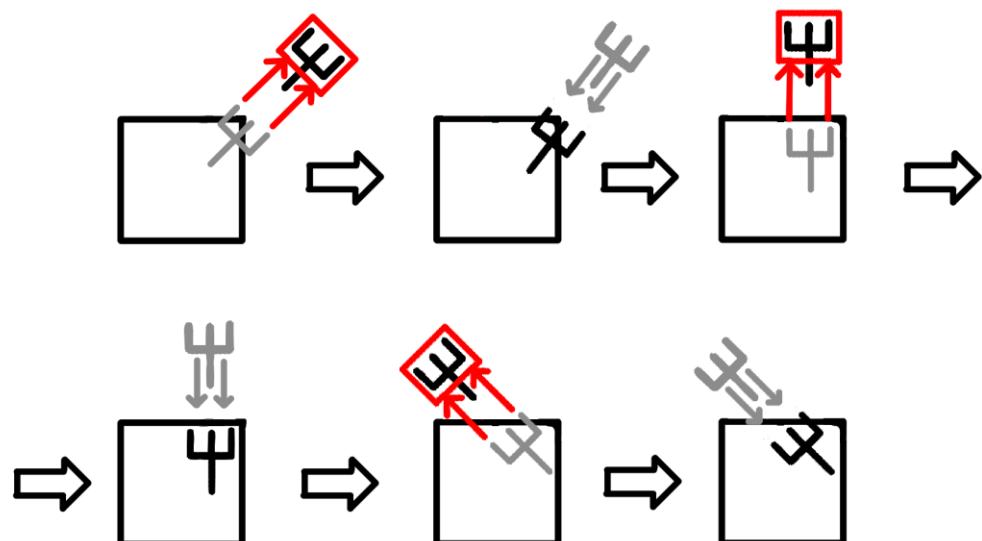
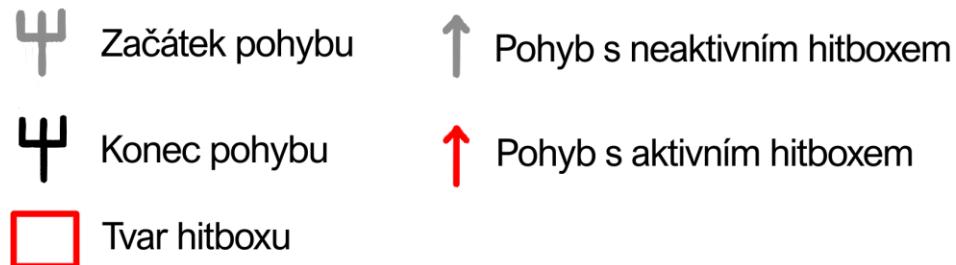
Sebezničení, poškození = 20 bodací

## Animace Drápů

Zbraně v této sekci mají útočnou animaci (viz Obrázek 9), ve které bodnou 3krát, šikmo doprava, dopředu a šikmo doleva. Tyto zbraně mají také idle animaci (animace která je přehrávána, když se s ní neútočí), ve které se zbraň pomalu kýve ze strany na stranu (viz Obrázek 8).



**Obrázek 8**  
Znázornění idle animace drápů



Claw:

Popis: Claws of a monster. Very strong!

Cena: 500

Efekty:

poškození = 50 bodací, 50 sekaci

### 3.5. Kouzla

Kouzla jsou jednorázové předměty. Existují dvě kouzla:

- 1) ohnivá koule, která vyštřelí projektil, který když do něčeho narazí, poškodí ohnivým druhem poškozením všechny cíle v okolí srážky,
- 2) ledový bodec, který vyšle projektil, který poškodí první postavu, do které narazí, ledovým a magickým druhem poškození.

### 3.6. Ložiska surovin

Stromy, železná ruda a magické krystaly jsou 3 druhy ložisek. Existují jako fyzické objekty v herním světě, které mohou být poškozeny a zničeny tím, že na něj postava zaútočí. Není rozdíl v tom, zda je útok proveden zbraní nebo nástrojem. Aby těžení surovin trvalo přiměřeně dlouho, tak budou mít ložiska 500 životů. Pokud je

ložisko zničeno, tak se vytvoří pick-upy surovin, stromy vytvoří 3krát Log, železná ruda 5krát Iron Ore a magické krystaly 5krát Crystal Shard.

## 3.7. Objekt

V herním světe existují objekty, se kterými mohou postavy interagovat (hráči vždy interagují stisknutím tlačítka pro interakci). V této sekci popíšeme, jaké mají efekty.

### 3.7.1. Truhla

Truhla je objekt, který má inventář. Postavy mohou s truhlou interagovat, aby mohly přesouvat předměty ze svého inventáře do inventáře truhly a naopak. Inventář truhly má kapacitu 16. Pro interakci s truhlou bude hráč používat stejný systém GUI [6] jako u vlastního inventáře, s tím, že místo toho, kde se normálně zobrazuje vybavení se zobrazí inventář truhly viz Truhla 3.10.2.

### 3.7.2. Obchod

Obchod má stejnou funkci jako truhla, ale za vložení předmětu dostane hráč množství zlata rovné jeho ceně a pro odebrání předmětu musí hráč zaplatit jeho cenu, pokud cenu zaplatit nemůže, tak předmět nemůže odebrat. GUI pro hráče se zobrazí podobně jako GUI pro truhlu, s tím rozdílem, že pod každým předmětem bude zobrazena jeho cena (viz Truhla a obchod 3.10.2).

### 3.7.3. Dveře

Dveře mohou postavy otevřít a zavřít. Toto nemá žádné omezení, dveře nemohou být zamknuty a hráč týmu 1 může vždy otevřít dveře v základně týmu 2.

### 3.7.4. Postel

V posteli může spát maximálně jedna postava. Spaní v posteli postavě pomalu regeneruje ztracené životy.

### 3.7.5. Podstavec

Podstavci odevzdáte výtežný předmět nepřátelského týmu, po odevzdání předmětu vyhraje tým, který předmět odevzdal.

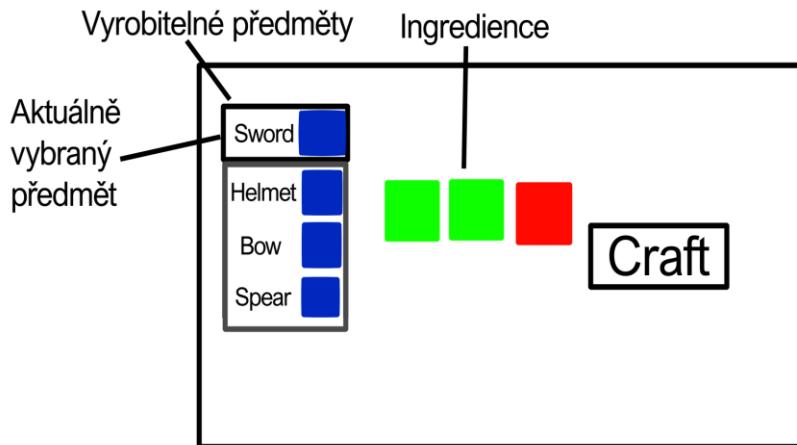
### 3.7.6. Kovadlina, pracovní stůl a výheň

Tyto objekty se používají k vyrábění předmětů. Informace o vyrábění v další sekci.

## 3.8. Vyrábění předmětů

Postavy používají vyráběcí objekty (kovadlina, pracovní stůl a výheň), aby vyrobily předměty. Aby mohla postava vyrobit předmět, tak musí mít správné ingredience (jiné předměty) a použít správný vyráběcí předmět. GUI pro hráče (viz Obrázek 10) bude mít na levé straně zobrazen seznam všech předmětů, které mohou být vyrobeny v tomto vyráběcím objektu. Z tohoto seznamu si hráč může vybrat, který předmět chce vyrobit. Po vybrání předmětu se uprostřed obrazovky zobrazí sprity ingrediencí, které se obarví zeleně, pokud je má v inventáři a červeně, pokud je nemá. Pokud má hráč všechny ingredience, tak může zmáčknout tlačítko na vyrobení

předmětu na pravé straně, což odstraní ingredience z jeho inventáře a nahradí je vyrobením předmětem.



**Obrázek 10**  
Diagram GUI pro vyrábění předmětu

V následující sekci je seznam receptů, které říkají, jaký vyráběcí objekt a jaké předměty potřebujeme pro výrobu daného předmětu. Recepty jsou rozděleny do kapitol podle toho, jaký vyráběcí objekt vyžadují.

### 3.8.1. Pracovní stůl

Výsledný předmět	Ingredience
Refined Crystal	Crystal Shard
Plank	Log
Fireball	3xRefined Crystal
Iceshard	3xRefined Crystal
Bow	3xPlank
Hoodie	Plank
Health Potion	2xRefined Crystal

### 3.8.2. Kovadlina

Výsledný předmět	Ingredience
Axe	2xIron Bar, Log
Hammer	3xIron Bar
Pickaxe	Iron Bar, Log
Spear	Iron Bar, 2xPlank
Sword	2xIron Bar
Helmet	2xIron Bar

### 3.8.3. Výheň

Výsledný předmět	Ingredience
Iron Bar	Iron Ore, Log

## 3.9. Vstup hráče

Hráč bude mít tlačítka, pomocí kterých bude interagovat s hrou, tlačítka budeme potřebovat pro:

- 1) pohyb a navigace, pro pohyb v herním světě a navigaci v herních menu, budeme používat 4 tlačítka určující ortogonální směr,
- 2) přepnutí menu inventáře, 1 tlačítko budé otevře nebo zavře inventář,
- 3) interakci s herními objekty, 1 tlačítko pro otevření menu truhly, otevření dveří, sebrání pick-upu atd.,
- 4) používání vybavení a ovládání menu, vybavení můžeme používat, pokud ho *využíváme* v jedné z našich rukou, protože máme vždy 2 ruce, tak potřebujeme 2 tlačítka, jedno pro každou ruku. Tato tlačítka budeme také používat pro ovládání menu, obecně, tlačítka pro levou ruku bude mít funkci potvrzení (např. výběr předmětu k prodeji nebo koupě v obchodě, výběr předmětu v inventáři), a tlačítka pro pravou ruku bude mít funkci zrušení (např. zrušení výběru předmětu v inventáři),
- 5) navigaci hotbaru, 2 tlačítka, jedno změní vybrané pole za pole nalevo od něj, a druhé změní vybrané pole za pole napravo od něj,
- 6) použití předmětu a vyhození předmětu. 1 tlačítko použijeme k tomu, abychom použili předmět, který máme právě vybraný v hotbaru, alternativně, pokud máme otevřené menu inventáře, tak stisknutí tohoto tlačítka odstraní vybraný předmět z inventáře a v podobě pick-upu ho vyhodí do herního světa.

## 3.10. GUI

Všechny UI [6] elementy hráče jsou zobrazeny exkluzivně v jeho části obrazovky.

Každý hráč vidí vždy své životy a množství zlata v pravém horním rohu svojí části obrazovky. Hotbar ukazující všechny předměty hráčova inventáře je zobrazen v prostředku spodní části jeho obrazovky. Hráč má vždy označeno jedno pole hotbaru, toto pole bude zvýrazněno a stlačením tlačítka pro *použití* předmětu se tento předmět (pokud je *použitelný*) *použije*.

### 3.10.1. Hráčův inventář

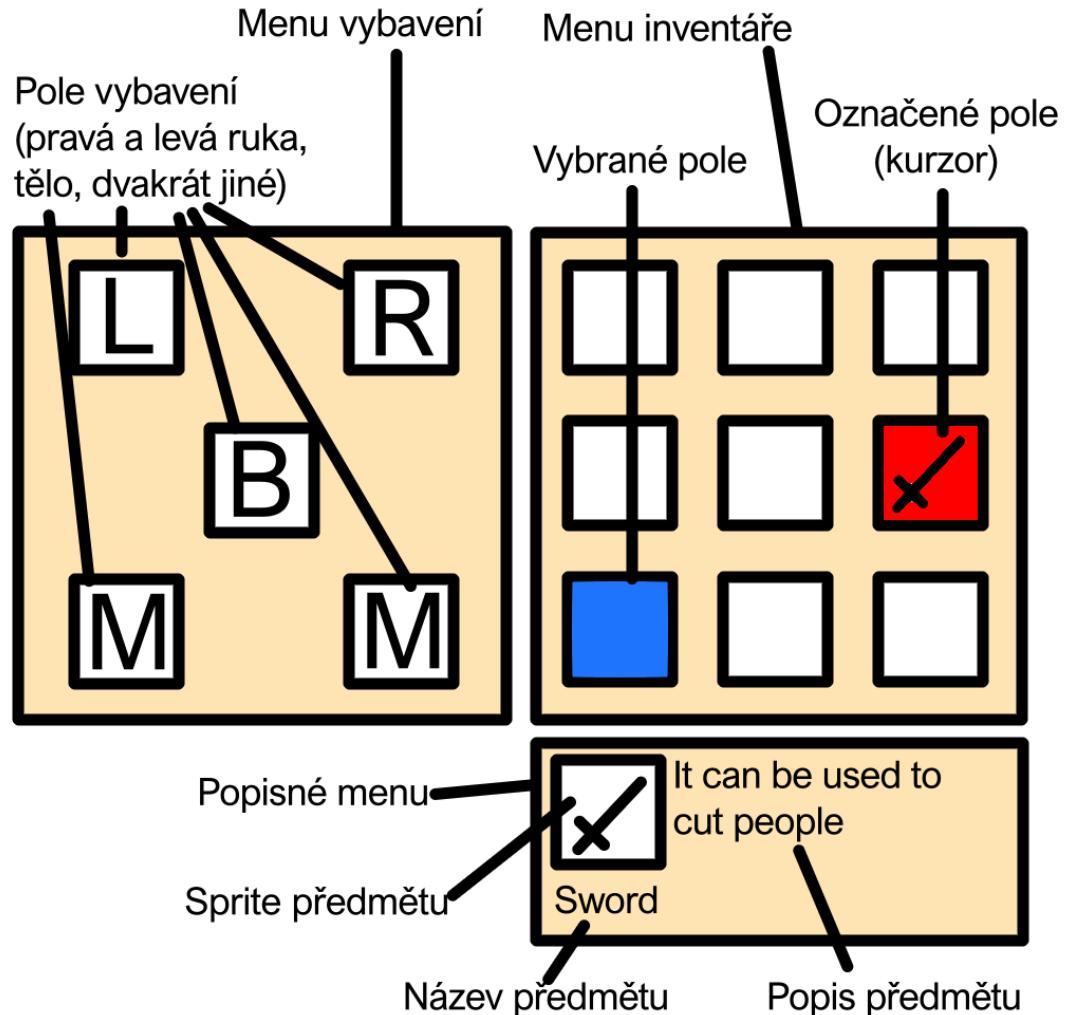
Menu hráčova inventáře (viz Obrázek 11) se zobrazí po stlačení tlačítka pro otevření inventáře (viz Vstup hráče 3.9) a skládá se ze tří částí:

- 1) vybavení, kde je zobrazeno hráčovo *využívané* vybavení,
- 2) inventáře, kde jsou zobrazeny předměty uloženy v hráčově inventáři,
- 3) popisového menu, které zobrazuje popisek předmětu ležící na poli, které máme právě označeno kurzorem.

Menu naviguje hráč pomocí tlačítek k pohybu, kterými mění pozici kurzoru. Pohybem se kurzor vždy pohne na další pole v daném směru, v menu vybavení se kurzor přesune na pole B, když se pohne směrem dolu z L a R nebo nahoru z M, také se na toto pole můžeme dostat pohybem doleva z menu inventáře.

Hráč může předměty přesouvat mezi poli tímto způsobem, nejprve vybere pole stisknutím tlačítka pro potvrzení (aktivace levé ruky, viz Vstup hráče 3.9), poté přesune kurzor na jiné pole a zase stlačí tlačítka pro potvrzení, toto vymění obsahy těchto polí. Předměty mohou být libovolně přesouvány mezi poli v menu vybavení a menu inventáře, s tím, že předměty, které chceme přesunout do pole v menu

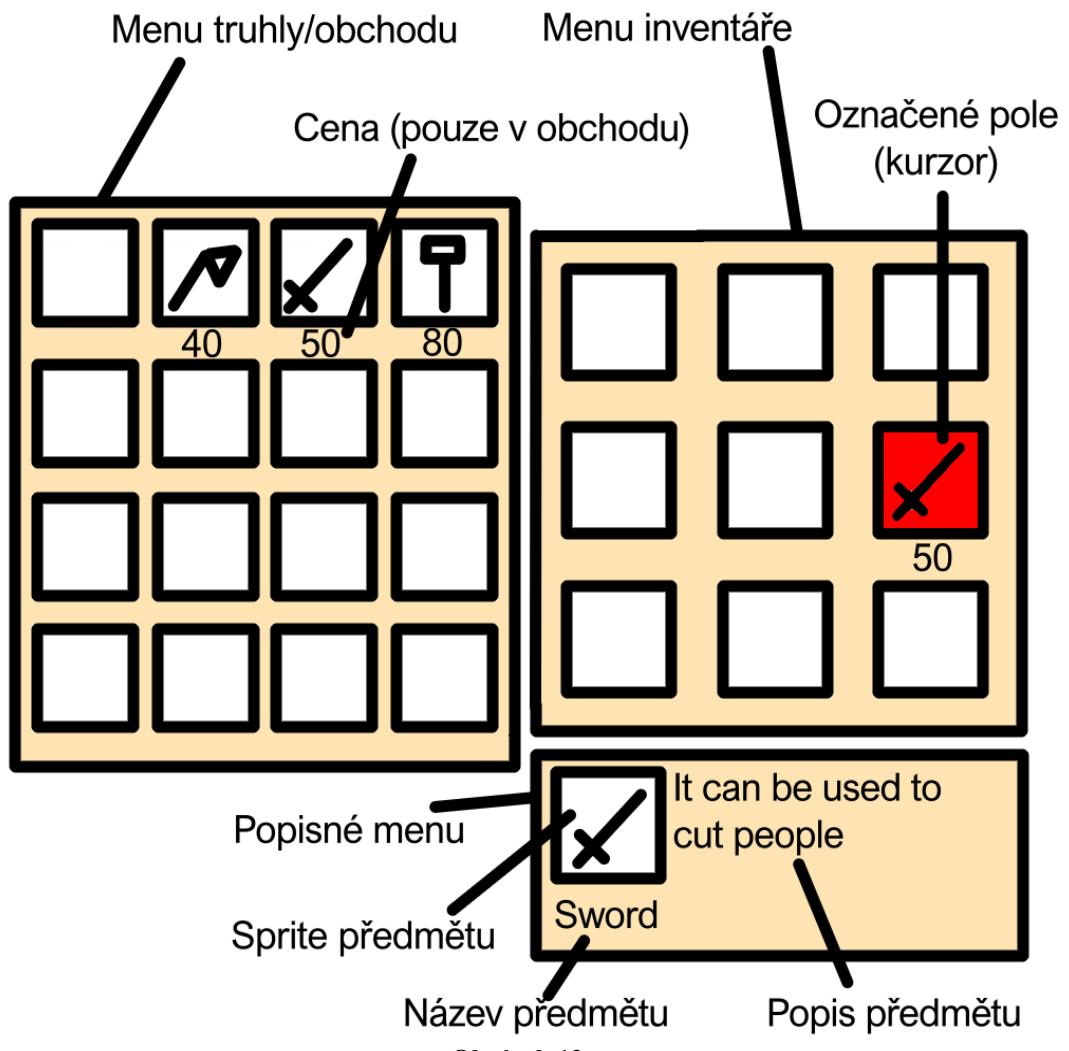
vybavení, musejí být vybavení odpovídajícího druhu (např. předmět, který chceme přesunout do pole B, které je druhu *tělo*, musí také být druhu *tělo*, tedy například helma). Když hráč přesune předmět do pole vybavení tak tento předmět začne být *využíván*. Jakýkoliv předmět můžeme z vybavení nebo inventáře odstranit a vytvořit z něj pickup, když ho označíme kurzorem a stiskneme tlačítko na zrušení (aktivace pravé ruky, viz Vstup hráče 3.9)



Obrázek 11

### 3.10.2. Truhla a obchod

UI truhly a obchodu jsou identické, až na to, že obchod také zobrazuje ceny předmětů. Ovládání a přesouvání předmětů funguje stejně jako v inventáři. UI se také skládá ze stejných částí jako inventář, pouze menu vybavení je nahrazeno menu truhly/obchodu, viz Obrázek 12.



Obrázek 12

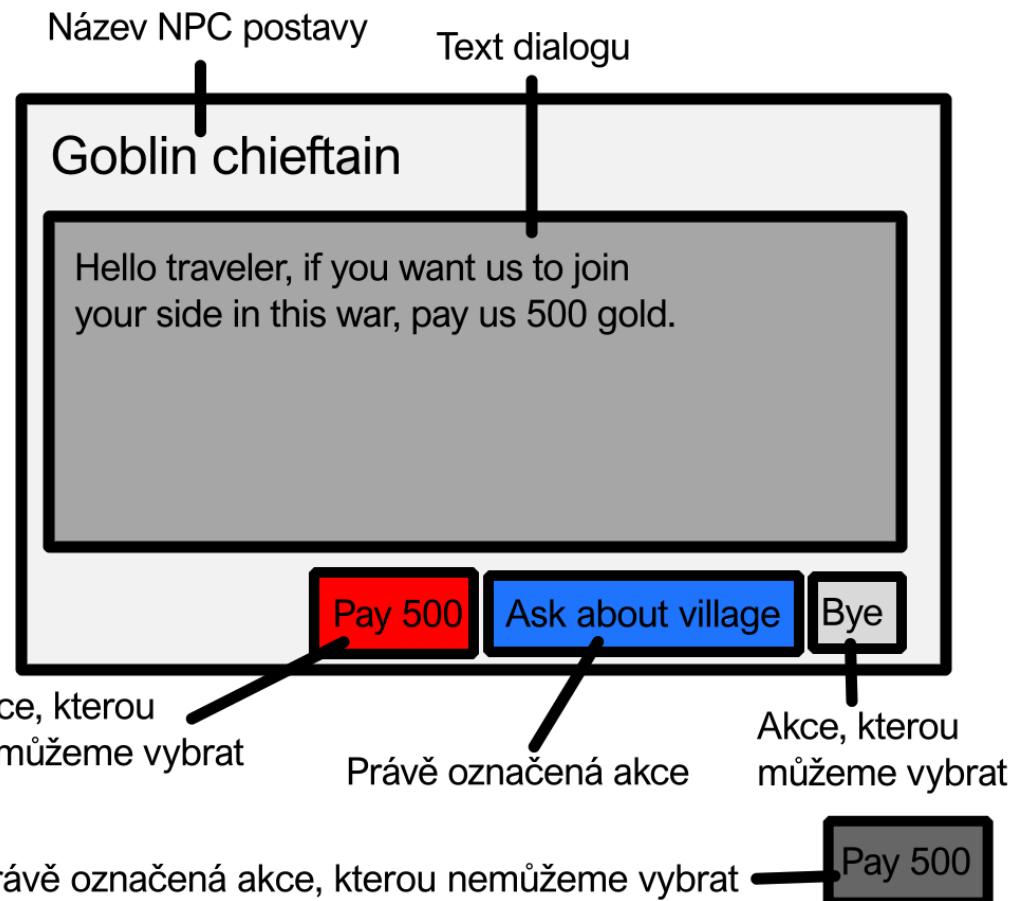
### 3.11. Dialogový systém

Hráč bude moci započít dialog s NPC postavami stlačením tlačítka k interakci v blízkosti NPC postavy a některých objektů (např. spící golem). Během dialogu může hráč získat informace tázáním otázek, změnit chování NPC postav (např. vydáním rozkazu) nebo vykonat speciální akci (vložení šému do spícího golema). Hráč během dialogu vybírá akce, které rozhodují, jak dialog bude pokračovat. Akce jsou námi, designéry, předem vytvořené a nejsou nijak dynamicky generované. Některé akce mohou mít požadavky, které musejí být splněny, aby je mohl hráč vybrat. Například, aby hráč mohl vložit do spícího golema šém, tak potřebuje mít šém v inventáři. Hráč může z dialogu vždy odejít stlačením tlačítka pro interakci, ale dialog může i skončit automaticky, když hráč vybere akci, na kterou už jiný dialog nenavazuje.

Po započetí dialogu se NPC postava a hráčská postava přestanou hýbat a hráči se zobrazí UI pro konverzaci (viz Obrázek 13). UI zobrazuje:

- 1) Název postavy, jméno pro unikátní postavy, jinak pracovní titul a rasa (např. goblin soldier).
- 2) Text dialogu, zde je většinou napsáno, co NPC postava řekla, ale mohou tu být i popisky nebo vnitřní dialog hráče (např, když se hráč rozhoduje, zda vložit šém do spícího golema).

3) Akce, které hráč může vybrat, aby pokračoval v konverzaci. Hráč může změnit označenou akci stlačením tlačítek pro pohyb vlevo a vpravo a výběr akce může potvrdit stlačením tlačítka pro potvrzení (pro použití levé ruky, viz Vstup hráče). Pokud akci můžeme vybrat, tak bude obarvena světle šedě, pokud bude také označena, bude obarvena modře, pokud ji nemůžeme vybrat, tak bude obarvena červeně a pokud ji nemůžeme vybrat a je označena tak tmavě šedě. Chceme ukazovat akce, které hráč nemůže vybrat, aby věděl, že existují a mohl, případně splnit jejich požadavky.



Obrázek 13

### 3.12. Komunikační bubliny

Aby hráči věděli, jaké akce NPC postavy vykonávají, tak jim přidáme komunikační bubliny, pomocí kterých mohou komunikovat, co dělají. Komunikační bublina se zobrazí po dobu 5 sekund nad postavou potom co se rozhodne vykonat nějakou akci. Každá akce má přiřazený vyhovující piktogram, která se v bublině zobrazí, viz Obrázek 14 zobrazující člověka, který se rozhodl těžit suroviny (stromy nebo ložiska kamene nebo ložiska kouzelných krystalů)



Obrázek 14

### 3.13. Obrazovka pro více hráčů

Obrazovka bude vždy rozdělena do čtyřúhelníků, nejprve rozdelením obrazovky vertikálně, poté horizontálně a poté zase vertikálně, dokud nebude alespoň jedna sekce obrazovky na jednoho hráče. Pokud zbydou nějaké sekce tak je spojíme s jednou sousední sekcí, která nějakému hráči už patří.

### 3.14. Vyvolávací kruh

Grafická reprezentace respawn pointu NPC postav nebo hráčů. Každý kruh má přidělenou postavu, kterou po její smrti za daný časový interval respawne

### 3.15. Budovy

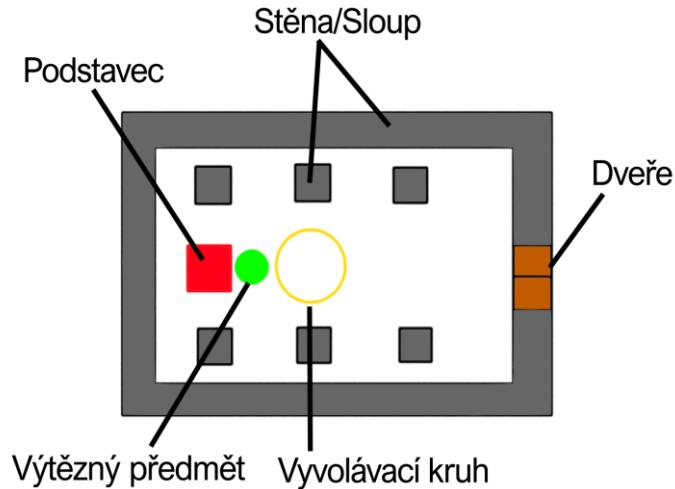
Budovy jsou seskupení postav a objektů. Budovy mají většinou vyvolávající kruhy a slouží jako místa kde NPC vykonávají činnosti, jako například ukládání vyrábění a prodávání předmětů. Měřítko diagramů je takové, že jeden čtvereček zabírá  $1 \text{ m}^2$ . Diagramy ale nejsou přesně rozděleny do čtvercové mřížky a stěny ani nejsou rozděleny do čtverců, protože na přesných velikostech nezáleží a ty mohou být vygenerovány a upřesněny během výroby.

#### 3.15.1. Týmové základny

Základna každého týmu bude seskupení těchto budov: kostel, kasárna, dřevorubecká chata, hornická chata, tržiště.

### 3.15.2. Kostel

Kostel má vyvolávací kruh, který vyvolá hráče jeho týmu po jeho smrti. Také v něm je pick-up vítězného předmětu svého týmu a podstavec očekávající nepřátelský vítězný předmět (viz Obrázek 15).



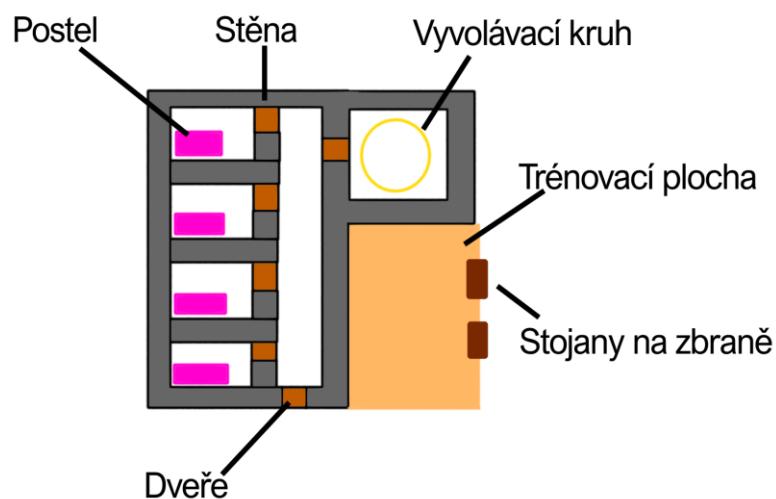
**Obrázek 15**  
*Diagram kostela*

### 3.15.3. Kasárny

Kasárny mají 4 postavy s přidělenými vyvolávacími kruhy. Postavy jsou vojáci, kteří budou strážit základnu svého týmu a budou útočit na nepřátele v jejich blízkosti. Hráči jim budou moci individuálně dávat tyto rozkazy pomocí dialogového okna:

- 1) „Guard“: postava bude stráži místo, kde právě stojí,
- 2) „Follow me“: postava bude následovat hráče,
- 3) „Attack“: postava vybere náhodnou nepřátelskou postavu v herním světě a půjde ji zabít.

Kasárna bude mít 5 místností, 4 ložnice a 1 místnost s vyvolávacím kruhem (viz Obrázek 16). Vedle kasáren bude trénovací plocha s dvěma stojany na zbraně. Trénovací plocha nemá žádný herní efekt je tu pouze kvůli estetice. Stojany na zbraně jsou mechanicky stejné jako truhly, jenom mají jiný sprite.



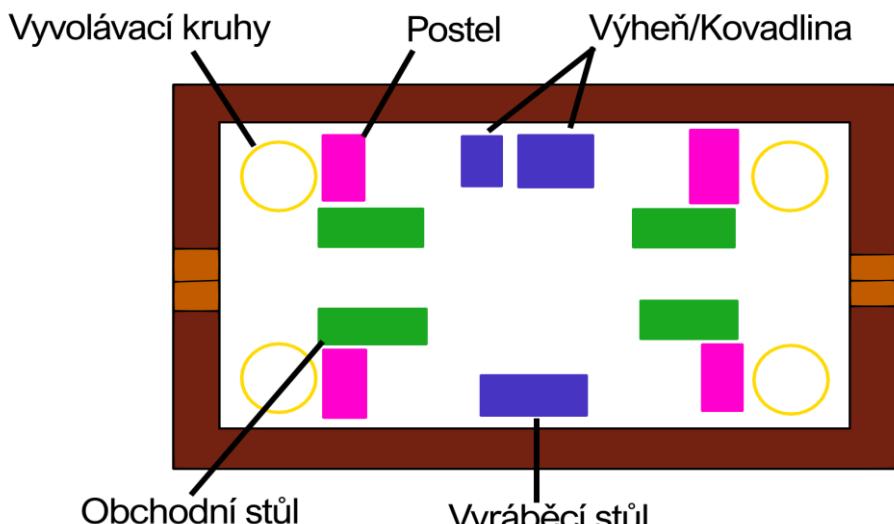
**Obrázek 16**

## *Diagram kasáren*

#### 3.15.4. Tržiště

Zde budou 4 postavy, každá s vlastním vyvolávacím kruhem a obchodem (viz Obrázek 17), také zde bude kovadlina, pracovní stůl a kovárna. Postavy budou doplňovat obchody předměty k němu přidělené. Obchody budou prodávat tyto předměty:

- 1) léčivé lektvary,
  - 2) zbraně na blízko,
  - 3) zbraně na dálku,
  - 4) brnění.



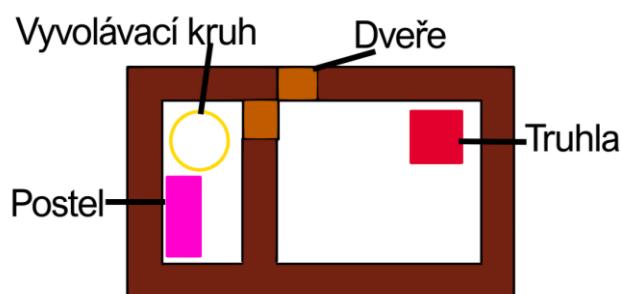
### Obrázek 17

### 3.15.5. Dřevorubecká chata

Jedna postava s vyvolávacím kruhem, která sbírá dřevo (viz Obrázek 18).

### 1.1.1. Hornická chata

Jedna postava s vyvolávacím kruhem, která sbírá železnou rudu a magické krystaly (viz Obrázek 18).



---

### Obrázek 18

#### Diagram Dřevorubecké/Hornické chaty

## 3.16. Kočko-lidská vesnice

Kočko-lidksá vesnice bude mít hornickou chatu pro těžení magických krystalů a budovu s prodavačem prodávající oba druhy kouzel a léčivé lektvary.

## 3.17. Kmen goblinů

Gobliní kmen bude mí 2 lukostřelce, dřevorubce a prodavače luků. Také bude mít náčelníka, kterému můžete zaplatit 500 zlata, aby se kmen stal vaším spojencem, což znamená, že budou gobliním vojákům hráči spojeneckého týmu moct vydávat rozkazy, stejně jako svým vojákům, a že budou goblini nepřáteli druhého týmu a jejich postavy vůči nim budou agresivní. Dřevorubec, prodavač luků a náčelník budou sídlit v jedné velké kruhové budově. A každý lukostřelec bude v malé kruhové chatě před hlavní budovou.

## 3.18. Zřícenina

Zřícenina starého hradu. Je v ní truhla se šémem. Zřícenina je hlídána NPC rasy příšera, která bude agresivní vůči všem ostatním postavám. Na rozdíl od ostatních NPC nemá NPC příšera oživovací kruh. Příšera je vybavena unikátní zbraní „drápy“, které dívají veliké poškození. Drápy, přestože představují přirozenou zbraň příšery, tak se chovají jako jakýkoliv jiný předmět, takže se i jako ostatní zbraně po smrti stanou pick-upem.

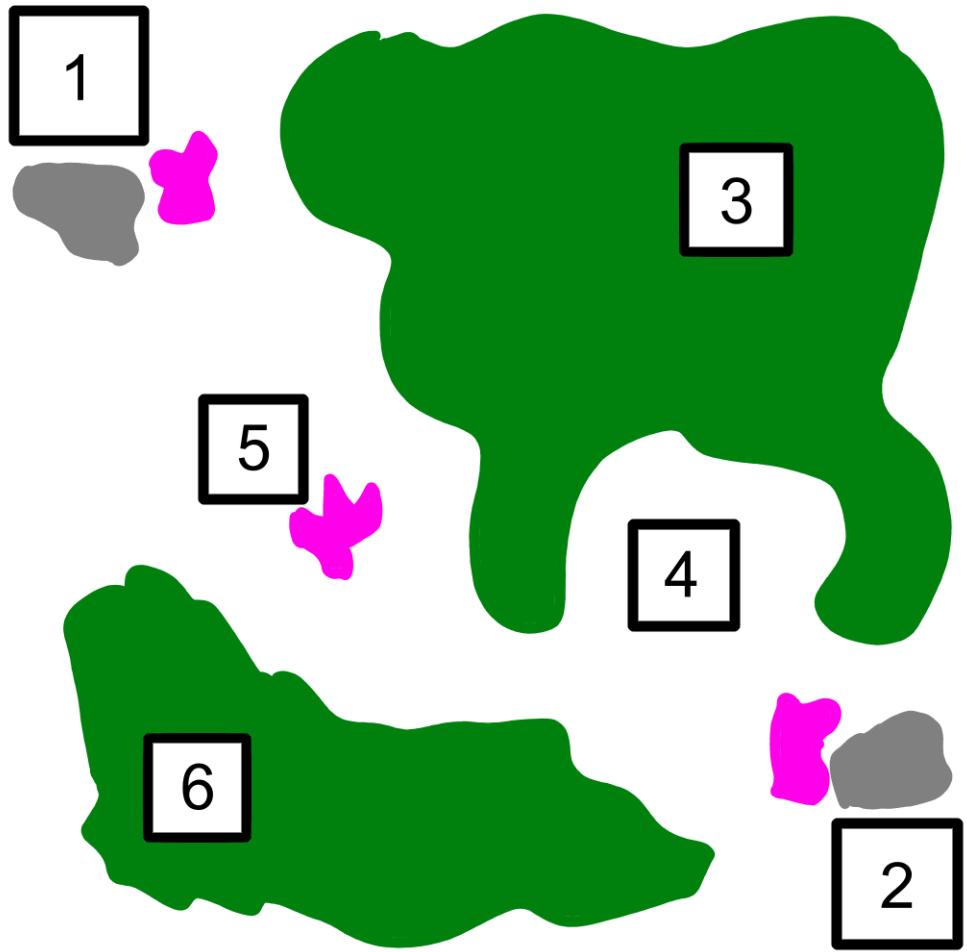
## 3.19. Spící Golem

Osamocený golem, kterého můžete aktivovat s použitím šému. Po aktivaci se stane vaším spojencem a můžete mu dávat stejně rozkazy jako standardním NPC vojákům. Golem nemá oživovací kruh. Místo toho, aby používal zbraně, tak útočí speciálním způsobem pomocí plamene, který mete z jeho očí, jedná se o efekt s plochou rozsahu, který dává ohnivé zranění, dále vysvětleno v další kapitole u rasy Golem. Spící golem je usazen mezi čtyřmi kamennými pilíři, které jsou obrostlé hustým, skoro nepropustným lesem.

## 3.20. Mapa světa

Na tomto obrázku (viz Obrázek 19) jsme znázornili schéma herního světa, které ukazuje rozpoložení lokací, znázorněny očíslovanými čtverci a rozpoložení surovin, znázorněny barvami. Herní svět bude ohraničen neprůchodným oceánem, aby se hráč nemohl ztratit v nekonečnu. Dále, kromě objektů, které mají nějaký herní účel, budeme chtít nakreslit nějaké pozadí, aby se nehrálo v prázdnotě a abychom navodili atmosféru. Specificky bychom chtěli, aby naše pozadí splňovalo tyto požadavky:

- 1) navozovalo atmosféru (např. mrtvá tráva vedle prokletého hrobu),
- 2) informovalo hráče o okolí (např. močálová textura může informovat hráče o přítomnosti jiných příšer),
- 3) pomáhalo s navigací (např. cesty vedoucí mezi důležitými místy),
- 4) být nevýrazné, nechceme příliš upoutávat pozornost od důležitého děje, nebo aby bylo těžké rozlišit prvky popředí (NPC, budov, objektů atd.) od pozadí.



- |                         |                              |
|-------------------------|------------------------------|
| 1) Základna týmu 1      | ● Ložisko kamene             |
| 2) Základna týmu 2      | ● Ložisko magických krystalů |
| 3) Zřícenina            | ● Les                        |
| 4) Kmen goblinů         |                              |
| 5) Kočko-lidská vesnice |                              |
| 6) Spící golem          |                              |

Obrázek 19

### 3.21. Rasy

Každá postava má přidělenou rasu. Rasa udává, jak postava vypadá, kolik má životů, rychlosť pohybu a otáčení postavy a fyzickou velikost postavy v herním světě a případně speciální vlastnosti.

Sprity postav jsou ve tvaru čtverce, takže budeme velikost postavy udávat jako délku hrany. Standardní velikost postavy je 0.8. Jelikož jsou budovy vytvořeny ze čtverců o délce hrany 1, tak takto velikost postav zaručí že budou moct projít dveřmi. Každý sprite postavy zobrazuje pouze hlavu postavy z čelního pohledu.

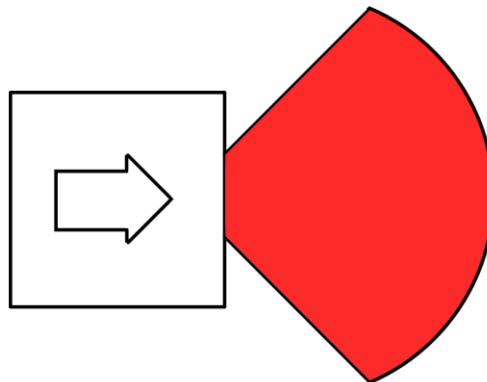
Formát:

1 metr odpovídá 1 jednotce měření v Unity.

Název	Sprity	Počet	Rychlosť	Rychlosť	Velikost	Speciální
-------	--------	-------	----------	----------	----------	-----------

Rasy		životů	(m/s)	otáčení (Stupeň/s)	postavy	vlastnosti
Člověk	1) hlava s blond'atými vlasy 2) hlava s hnědou čepicí a hnědými vlasy	100	10	20	0.8	
Goblin	Zelený goblin s dlouhýma ušima	80	12	24	0.8	
Kočkočlověk	Tmavo-modré vlasy a kočičí uši	100	10	20	0.8	1)
Golem	Golem z rudé hlíny	200	6	12	0.95	4), 5)
Příšera	Rozpůlená masitá hlava z ostrými zuby	200			1.6	2), 3)

- 1) Tato rasa má -50% odolnost (neboli 50% slabost, poškození je zvětšeno o 50%) vůči *ohnivému* poškození, a 50% odolnost vůči *ledovému* poškození
- 2) Tato rasa má schopnost regenerace a sama si časem obnovuje životy, rychlosťí 3 životy/s
- 3) Tato rasa nemůže být vybrána hráči v menu výběru postavy před začátkem hry.
- 4) Tato rasa má 100% odolnost vůči *ohnivému* poškození
- 5) Golem NPC, které můžete aktivovat šémem v herním světě má speciální útok, ale hráčské postavy ho nemají k dispozici. Speciálním útokem před sebou golem v čtvrt kruhu vymete oheň, který dává 35 *ohnivého* poškození. Velikost útoku je zobrazena na Obrázek 20.



**Obrázek 20**  
Diagram útoku golema

### 3.22. Chování NPC

Zde popíšeme obecné chování NPC, jejich cíle a akce, které mohou vykonávat.

## **Obecné cíle**

Tyto cíle jsou společné pro většinu NPC.

Každé NPC bude mít cíl obnovování životů, tento cíl bude mít větší prioritu, čím méně životů, relativně k maximálnímu počtu životů, NPC má. NPC se tento cíl pokusí splnit použitím předmětu, který mu doplní životy.

Každé NPC, které má přidělenou postel (příšera a golem postel nemají) bude mít cíl se v ní vydat. Tento cíl bude mít malou prioritu, která bude časem pomalu růst, tak, aby při vykonáváním normálních pracovních cílů šlo NPC spát přibližně jednou za 5 minut. NPC bude v spát v posteli asi 20 vteřin, po čemž se priorita cíle resetuje a NPC může pokračovat v jiných cílech. Priorita tohoto cíle by neměla překročit prioritu bojových cílů, aby NPC neopustilo bojiště uprostřed bitvy.

## **Pracovní cíle**

Tyto cíle jsou individuálně přiděleny NPC designéry hry, a udávají chování a roly NPC ve hře.

Pro ekonomické chování NPC nám stačí jeden cíl, a to je bude požadovat předměty v truhle. NPC zadáme jaké předměty mají být v jaké truhle (obchod je poddruh truhly) a NPC pak vykoná potřebné akce, aby předměty v truhle byly. Pokud například zadáme, aby v truhle byly předměty Log, tak se NPC nejprve podívá, jestli nějaký Log už v blízkosti neleží, pokud ano tak ho vloží do truhly, a pokud ne tak si najde sekuru a setne strom, aby z něj získalo Log. Pokud NPC řekneme, aby v truhle byly meče, tak se stejným způsobem pokusí najít meče, a pokud žádné nenajde tak je začne vyrábět z dostupných surovin. Samozřejmě nechceme, aby se dvě NPC přetahovali o stejný předmět, a přenášeli ho z jedné truhly do druhé, takže přidáme omezení, že požadovaný předmět nemůžeme získat z jiné truhly. Toto omezení ale nebude platit pro suroviny, ze kterých daný předmět můžeme vytvořit, protože nám v tomto případě žádné přetahovací problémy nevzniknou, jelikož dané suroviny, po vytvoření předmětu, přestanou existovat, takže nezbyde nic, o co by se dalo přetahovat.

## **Vojenské chování**

Pro vojenské chování NPC budeme potřebovat vícero cílů:

- 1) hlídkování, NPC hlídkaře kolem jednoho bodu, tak že se pohybuje mezi náhodně vybranými body kolem daného bodu. Tyto náhodné body se generují v okolí 10 metrů,
- 2) nájezd, NPC si vybere náhodné nepřátelské NPC a snaží se k němu dostat co nejbližše, což aktivuje další cíl,
- 3) boj, pokud se do okolí (~ 10 metrů) NPC dostane nepřátelská postava tak se tento cíl aktivuje s vysokou prioritou. NPC si vybere nejbližší nepřátelskou postavu v tomto okolí a začne na něj útočit. Pokud se nepřítel dostane mimo okolí tak NPC přestane útočit a cíl se vypne. NPC bude útočit jiným způsobem, pokud má zbraň na dálku (např. luk), místo toho, aby se dostalo co nejbližše a útočilo zbraní, tak bude nepřítele pronásledovat z mírné dálky (~6 metrů) a bude střílet, pokud bude blíže než tato vzdálenost,
- 4) vybavení, NPC dostanou seznam, jaké vybavení mají používat. NPC bude mít vysokou prioritu toto vybavení najít a využít, aby bylo připraveno na boj. Jaké vybavení NPC potřebuje bude zadáno obecně. Např, zbraň na blízko, na dálku, zbroj. NPC si pak vybere vybavení, které splňuje daný požadavek. Zbraň na blízko může být meč, kladivo atd.

### 3.23. Spuštění hry

Po stisknutí tlačítka start na hlavním menu se objeví výběr postavy pro 2 hráče, oba hrající na klávesnici, další hráči se nyní budou moct připojit pomocí herního ovladače. Při výběru postavy si hráči budou moct vybrat rasu postavy a za jaký tým chtejí hrát. Když všichni hráči dokončí výběr tak se se jejich postavy objeví v jejich základnách a hra začne.

# 4. Detailní analýza

V této kapitole si rozmyslíme, jak implementovat mechaniky, které jsme navrhli v designovém dokumentu.

## 4.1. Herní Engine

Nejprve se musíme rozhodnout, jestli použijeme nějaký veřejný game engine, nebo jestli si naprogramujeme vlastní.

Naše hra je relativně standardní 2D hra, používá spritovou grafiku, 2D fyziku s detekcí kolizí, čtení vstupu hráčů atd. Nepotřebuje žádné nestandardní nástroje, které by nebyly dostupné ve většině 2D herních enginů, takže si jednoduše vybereme již existující engine, ve kterém hru naprogramujeme. Nejpopulárnější 2D herní enginy jsou Godot a Unity [5], z nichž si mi vybereme Unity, protože s ním má autor už nějaké zkušenosti.

## 4.2. Postavy

První věc, kterou do naší hry naprogramuje budou postavy. Hráči budou interagovat se zbytkem hry skrz svoji postavu a jsou tedy základním pilířem celé hry, ostatní systémy se budou od postav odvozovat podle toho, jak s nimi budou souviseť.

Postava bude implementována jako GameObject, ke kterému připojíme různé komponenty (scripty). Tyto komponenty budou implementovat systémy dle požadavků, které jsme popsali v game design dokumentu (viz Postava 3.2). Dále budeme analyzovat jednotlivé komponenty, které se rozhodneme použít v postavě.

### 4.2.1. Fyzická interakce

Jelikož Unity už má implementován fyzikální systém, tak můžeme jednoduše použít jejich předpřipravené komponenty k tomu, aby postavy interagovaly s ostatními fyzickými objekty. Specificky použijeme komponentu **Rigidbody**, která dovolí, aby naše postava mohla být pohybována fyzikálními silami (např. být tlačena jinou postavou), ostatní komponenty (**Staticbody**, **Kinematicbody**) dovolují pohyb pouze manuálně přes kód, což my nechceme.

### 4.2.2. Pohyb

Pohyb má relativně jednoduché požadavky (viz Pohyb 3.2.2), konstantní rychlosť a rotaci s fyzikální implementací (neprocházet skrz stěny, možnost tlačit jiné postavy). Dále musíme vzít v ohled, že pohyb musí být ovládatelný hráčem i AI.

Fyzikální interakce jsme už vyřešili pomocí **Rigidbody**. K tomu, abychom mohli ovládat pohyb naší postavy, potřebujeme vytvořit novou komponentu, kterou nazveme **Movement**. Tato komponenta bude držet informace potřebné pro pohyb, což je rychlosť pohybu a rotace. GameObjectu s **Rigidbody** komponentou má svoji pozici ovládanou **Rigidbody**, takže budeme změnu pozice řešit komunikací s touto komponentou. Tohoto můžeme dosáhnout dvěma způsoby, buď budeme měnit pozici **Rigidbody** přímo, nebo budeme na **Rigidbody** aplikovat sílu a což pak vyústí ve změnu rychlosti a novou pozici. Dle požadavků chceme, aby náš pohyb neměl žádné zrychlování nebo zpomalování neboli, abychom měli konstantní rychlosť. K tomuto

účelu se používání sil nehodí, takže možnost se silami nevyužijeme a místo toho budeme manuálně počítat další pozici, kterou pak předáme **Rigidbody**. Rotaci implementuje jednoduchým vypočítáním nové rotace a přímou změnou rotace **GameObject** postavy.

Aby mohli hráč i AI pohyb ovládat tak budeme mít public metodu, která bude přijímat vektor zadávající směr pohybu. Hráčův vstup budeme zpracovávat v jeho vlastní komponentě (viz Vstup hráče 4.2.6), která tuto metodu bude volat a AI bude tuto metodu volat ze své vlastní navigační komponenty (viz NPC Navigace 4.5)

#### 4.2.3. Inventář

Každá postava má inventář, ve kterém může skladovat předměty. Skladování předmětů je obecná vlastnost, kterou bychom mohli využít v jiných částech hry např. truhla si musí pamatovat, které předměty v ní jsou a obchod si musí pamatovat, jaké předměty má ve skladu, tyto potřeby se nijak neliší od potřeb postavy pamatovat, si předměty v inventáři, a proto bychom chtěli vytvořit takovou komponentu inventáře, kterou budeme moci používat ve všech těchto případech.

Inventář vyžaduje schopnost držení dat (viz Inventář 3.2.6) a schopnost modifikace (např. přesunutí předmětu mezi poli inventáře, odstranění předmětu z inventáře atd.) držených dat hráčem či AI. Vytvoříme komponentu **Inventory**, která toto implementuje.

Naše AI může tento inventář používat triviálně voláním metod, ale hráč bude potřebovat grafické rozhraní (GUI), které bude převádět hráčovi vstupy na volání metod. Vytvoříme tedy další komponentu **InventoryUI**, která zobrazí grafické rozhraní inventáře po stisknutí správného tlačítka (viz Vstup hráče 3.9). **InventoryUI** pak bude volat odpovídající metody na **Inventory**. Design a ovládání GUI inventáře je popsáno v kapitole GUI 3.10 a analýza GUI obecně je popsána v kapitole Objekty s UI 4.4.3.

#### 4.2.4. Vybavení

Vybavení jsou předměty, které mohou postavy *využít*. Potřebujeme tedy komponentu, které umožní postavě si vybrat, které vybavení *využívá* s ohledem na to, aby bylo vybavení *využíváno* pouze ve správném poli pro vybavení (zbraně v poli pro *rukou*, viz Vybavení 3.2.7). Dále chceme, aby vylo *využívané* vybavení zobrazeno v herním světě, aby hráči mohli vidět jaké vybavení mají jejich spojenci a nepřátelé. Vybavení druhu *ruka* má ještě na rozdíl od ostatního vybavení možnost být *aktivováno*, což vyúsťí v přehrání útočné animace a přidělení efektů všem zasaženým objektům.

Postava má omezený počet 5 míst pro vybavení (viz Vybavení 3.2.7), a pro každé si musíme pamatovat, zda v něm využíváme předmět, jakého druhu pole je (*ruka, tělo, jiné*) a pozici, kde se fyzicky nachází na naší postavě (*tělo* uprostřed, *ruce* po stranách).

Pouze postavy budou mít schopnost využívat vybavení, takže naši novou komponentu vybavení, kterou si nyní vytvoříme, nemusíme nijak generalizovat.

Pozici vybavení můžeme buď zadat pomocí relativních souřadnic nebo můžeme vytvořit prázdný **GameObject** jako dítě **GameObject** jejíž pozice budeme interpretovat jako pozici vybavení. Zadávání souřadnic je méně intuitivní než přesunutí grafický objekt na požadovanou pozici, kvůli čemuž si vybereme možnost s prázdným **GameObject**em, tato možnost má také tu výhodu, že pod daný

`GameObject` můžeme hned generovat instance vybavení, a už budou automaticky zobrazeny na správné pozici.

### Vybavení v herním světě

Vybavení bude v herním světě reprezentováno `GameObject`m, který bude vždy mít komponentu `SpriteRenderer`, což je už před vytvořená unity komponenta, která nám umožní zobrazit sprite a další volitelné komponenty, které budou implementovat efekty vybavení.

### Animace

Vždy když *aktivujeme* vybavení, tak chceme, aby byla přehrána správná animace (viz Zbraně a jejich vlastnosti 3.4.1). Unity má komponentu `Animator`, která implementuje animační systém, který implementuje vše, co bychom potřebovali, proto ho budeme využívat, místo toho, abychom programovali vlastní systém. Abychom mohli animaci přehrát tak musíme nejprve animaci vytvořit. Pro vytvoření animace použijeme komponentu `Animator`, v animaci můžeme měnit všechny vlastnosti `GameObject`u, takže nejenom jeho pozici a rotaci, ale například i stav hitboxu (jeho zapnutí nebo vypnutí), což se nám hodí v sekci Detekce kolizí. Pro přehrání animace stačí zase použít `Animator`.

### Detekce kolizí

Detekce kolizí je důležitá, protože efekty *aktivace* vybavení jsou aplikovány pouze na `GameObject`y, které byly v kolizi s hitboxem vybavení. K detekci kolizí využijeme jednu z mnoha unity `Collider2D` komponent. Každá z těchto komponent má specifický tvar, (např `CircleCollider2D` má kruhový tvar), jelikož hitboxy našich zbraní jsou různorodého tvaru, tak si vybereme `PolygonCollider2D`, který si sami můžeme vytvarovat do požadovaného tvaru.

Když *využijeme* vybavení, tak chceme, aby byl hitbox vypnuty, a zapneme ho pouze při *aktivaci* vybavení. V znázornění animace a hitboxů, viz Zbraně a jejich vlastnosti 3.4.1, ale vidíme, že hitboxy zbraní nejsou zapnuty po celý průběh animace. Toto můžeme snadno vyřešit tím, že hitbox budeme zapínat nebo vypínat přímo přes animaci, místo toho abychom zapínání a vypínání řešili kódem pomocí časovačů.

Dále musíme vyřešit tento problém, kdybychom přímo reagovali na kolize detekované `PolygonCollider2D`, tak by mohl nastat problém, že během útoku detekujeme kolizi se stejným objektem vícekrát (např. když zbraň koliduje s postavou a zbraň nemá efekt knockback, který by postavu odstrčil, tak bude v dalším framu s postavou stále kolidovat). Takovému chování chceme zabránit, protože budeme aplikovat efekty zbraně na stejnou postavu vícekrát za jeden útok, což nechceme. To, že to nechceme, není v game design dokumentu explicitně napsáno, ale vycházíme ze zdravého rozumu, z hodnot poškození a životů postav vidíme, že by většina postav byla zabita jedním seknutím, kdyby poškození bylo aplikováno každý frame, což nedává moc smysl.

Abychom měli větší kontrolu nad kolizemi, tak bychom chtěli vlastní komponentu, která si bude pamatovat objekty, se kterými jsme již kolidovali a dalším komponentám předávat data o kolizích pouze, pokud jsme s nimi ještě nekolidovali (např. pouze při první kolizi). Paměť kolizí vždy po *aktivaci* vybavení vymažeme. Komponentu, který se o kolize bude starat nazveme, `Hand_Item_Controller`.

## Implementace efektů vybavení

Efekty můžeme implementovat několika způsoby, ale určitě chceme, aby náš systém byl modulární, protože se efekty často sdílejí mezi různými vybaveními. Dává teda smysl, abychom pro každý druh efektu vytvořili novou komponentu. Problém nastane s tím, jak tyto komponenty dostat do GameObjectů vybavení. Vybavení, a předměty obecně, budou uložené v podobě ScriptableObject (instance třídy v projektu, která má designérem zapsaná data) do kterých se komponenty ukládat nedají. Toto můžeme vyřešit několika způsoby:

- 1) pro každé vybavení vytvoříme prefab, ke kterému komponenty přidáme.  
Odkaz na tento prefab pak bude uložen v ScriptableObject daného vybavení.
- 2) Pro každý efekt vytvoříme pomocnou třídu, kterou do ScriptableObject vybavení budeme moct uložit, a která připojí svoji komponentu k instanci vybavení. Tyto třídy budou také mít uložena modifikovatelná data svých komponent (např. velikost a druh poškození), které budou použiti k inicializaci komponenty efektu.

Varianta 1 vytvoření prefabu pro každé vybavení a zároveň rozděluje data předmětů mezi prefaby a ScriptableObject, což snižuje přehlednost.

Varianta 2 potřebuje méně prefabů. Vybavení druhu *tělo* a *jiné* budou sdílet jeden prefab, který bude mít komponentu *SpriteRenderer*. Vybavení druhu *ruka* bude pak mezi sebou bude mít rozděleno několik prefabů, s komponenty *SpriteRenderer*, *PolygonColider2D* a *Animator*. Důvod, proč potřebujeme více komponent pro vybavení druhu *ruka* je ten, že většina tohoto vybavení má jiný tvar hitboxu a tento hitbox se dá měnit v editoru pouze v komponentě *PolygonColider2D*.

Varianta 2 nám také dovolí měnit efekty předmětů dynamicky, během běhu hry. Například, pokud bychom chtěli vylepšit náš meč přidáním *ohnivého* poškození, tak bychom vytvořili kopii ScriptableObject meče, a přidali do ní třídu, která drží informace o *ohnivém* poškození.

V našem demu sice nemáme příliš veliký počet vybavení a ani nemáme možnost vylepšování, ale i přesto se nám kvůli přehlednosti lépe hodí možnost varianta 2, a pokud tuto hru budeme rozšiřovat, tak bude možnost varianta 2 určitě nejlepší.

### 4.2.5. Ovládání inventáře a vybavení

Komponenty inventáře i vybavení budou mít interface pomocí kterého s nimi budou hráč a AI integrovat. V případě ovládání pohybu bylo samozřejmé, jaké metody interface volat v případě stisknutí tlačítka pohybu hráčem (např. tlačítko W odpovídá pohybu dopředu). U inventáře to je ale složitější, počet akcí problém není, máme 2 akce, přesunutí předmětu z jedné pozice inventáře na jiné a vyhození předmětu z inventáře, na což nám stačí 2 tlačítka. Problém je že tyto akce vyžadují parametry, jaký předmět a kam ho přesunout, nebo který předmět vyhodit. Přidat jedno tlačítko ke každé kombinaci parametrů je nepraktické, takže budeme chtít vytvořit jednodušší systém, který umožní hráči specifikovat parametry těchto akcí. Tohoto můžeme dosáhnout vytvořením GUI. Pro inventář a vybavení by hráč například mohl mít zobrazenu mřížku polí, kde každé pole je buď prázdné, nebo má sprite odpovídající předmětu který v poli je. Pak by hráč mohl kurzorem vybrat předmět a pole kam předmět přesunout. Hlubší analýzu GUI najdete zde Objekty s UI 4.4.3.

#### 4.2.6. Vstup hráče

Hráč má k dispozici nějaké zařízení, klávesnici nebo herní ovladač, a my potřebujeme mapovat vstupy z tlačítek na akce které hráč může dělat ve hře.

Vstupy (stisknutí, držení a upuštění tlačítka) hráčů můžeme číst mnoha způsoby, Unity ale už pro nás má předpřipravený systém, který vstupy zpracovává, nazývá se Input Systém. Input systém nám umožnuje volat metody v jiných komponentách v reakci na detekci vstupu, takže bychom mohli informace o vstupech hned předávat komponentám které tuto informaci vyžadují (např. předání informace o stisknutí tlačítka W, dopředu, komponentě Movement) Význam vstupů, se ale může změnit v závislosti na kontextu hry (např. pomocí tlačítka WASD hýbeme postavou v herním světě, ale když máme otevřené GUI tak tak chceme navigovat GUI kurzor) proto bychom chtěli mít nějakou třídu která si pamatuje v jakém kontextu jsme a podle toho se rozhodne kterým komponentám informaci o vstupu pošle. Tuto třídu jsme mohli implementovat dvěma způsoby:

- 1) centralizovaná varianta. Budeme mít jednu komponentu, která bude číst procesovat všechny vstupy z Input systému. Komponenta si bude pamatovat kontext, ve kterém se nacházíme (máme otevřené GUI nebo ne) a bude rozhodovat kam vstupy předávat. Tento přístup má výhodu v tom, že pokud nějaká komponenta změní kontext (otevře nebo zavře GUI), tak stačí oznámit změnu kontextu na jednom místě a naše komponenta se už o vše postará. Nevýhoda je, že pro každý kontext musíme určit do jaké komponenty se každý vstup přepošle.
- 2) decentralizovaná varianta. Zde budeme pro každý vstup mít vlastní zpracovávací třídu a místo toho, abychom si pamatovali kontext tak budeme mít seznam delegátů (reference na metodu), kterým budeme předávat vstupy. Výhoda této varianty je, že jiné komponenty mohou selektivně měnit kam se informace vstupu přenáší. Například, pokud by byl hráč zamotán do pavučiny, tak bychom hráči mohli přidat komponentu Pavučina, která odstraní delegáty komponenty Movement z komponent zpracovávající vstupy pro pohyb (např. tlačítka WASD) a vloží místo něj delegáty na svoje metody, kde můžeme například počítat kolikrát jsme se hráč pokusil pohnout, po nějakém počtu můžeme hráče z efektu pavučiny osvobodit, tím že delegáty pohybu vrátíme do zpracovávacích komponent a odstraníme delegáty komponenty pavučiny. Tímto dosáhneme, že když je hráč v pavučině tak se nemůže hýbat, a místo toho musí rychle mačkat tlačítka pohybu, aby se osvobodil. Nevýhodou je, že se musíme starat o to, že když nějaký delegát, odstraníme, tak ho musíme i vrátit zpět, když pomine náš speciální kontext (původní důvod pro odstranění pomine). Problémy mohou také nastat, když se vícero kontextů překrývá (např. jsme zároveň v pavučině i zmrazeni, a oba kontexty vyžadují stejný vstup).

V naší hře potřebujeme 3 kontexty, *normal*, *inMenu*, a *stun*. V *normal* kontextu ovládáme postavu, v *inMenu* kontextu navigujeme menu a v *stun* kontextu nedělájí naše vstupy nic. Obecně ve hře nemáme žádné kontextu, které by měnily význam podmnožiny vstupů, s jednou výjimkou, tlačítko na interakci, to je ale speciální případ, který později dovysvětlíme. Je vidět, že výhodu decentralizované varianty nám nijak nepomáhají, a tak se pro nás lépe hodí centralizovaná varianta.

#### Interakce

Interakce má několik požadavků, vždy interagujeme s nejbližším interagovatelným objektem, který také vždy zvýrazňujeme. Pokud je interakce *průběžná* (např. *průběžná*: Otevření menu obchodu vs *jednorázová*: otevření/zavření

dveří), tak si musíme zapamatovat se kterým objektem jsme v interakci, abychom ji mohli druhým stisknutím tlačítka pro interakci zrušit. Interagovat se dá s NPC, Objekty, a pick-upy.

Takováto komplexita vyžaduje vytvoření nové komponenty, `PlayerInteractManager`, která se o toto všechno bude starat. Dále, z pohledu hráče je tlačítko vždy používáno stejným způsobem a kontext, který se mění závisí vždy na druhém objektu. Na takovou situaci se výborně hodí interface, který implementujeme v každém objektu, se kterým budeme chtít interagovat. Z pohledu `PlayerInteractManager` pak budou všechny interaktivní objekty vypadat stejně, ale vlastní implementaci mohou mít jakoukoliv.

#### 4.2.7. Rasa

Každá postava má nějakou danou rasu. Rasa určuje sprite postavy, a také nějaké její hodnoty, např. počet životů, rychlosť pohybu a míru odolnosti vůči poškození. Jelikož je rasa v podstatě pouze uložitě těchto dat, tak bude nejlepší využít `ScriptableObject`, kterou nazveme `Race`. Vytvoříme také novou komponentu, která bude držet referenci na svoji rasu. Z této komponenty pak budou ostatní komponenty moci načítat požadovaná data.

#### 4.2.8. Schopnosti

Některé postavy, nebo i celé rasy mohou mít speciální schopnosti, např. příšera má schopnost regenerace životů a golem má schopnost vyšlehnout plameny. Regenerace životů je pasivní schopnost (operuje vždy, aniž by ji postava ovládala), a takové můžeme triviálně implementovat přidáním nové komponenty. Aktivní schopnosti (postava je musí aktivovat), jsou ale složitější, protože je musí naše AI být schopné používat (hráči nikdy neovládají postavu s takovou schopností). Jelikož má naše demo pouze jednu aktivní schopnost tak vytvoříme systém, který nám pro ni nejvíce vyhovuje, ale budeme brát ohled na to, abychom mohli v budoucnu snadno přidat další schopnosti.

Mohli bychom přidat akce (viz NPC AI 4.6), které má AI k dispozici, s logikou schopnosti naprogramovanou přímo v nich. Takový přístup by nám stačil pro naši schopnost golema, ale není to modulární přístup a kombinuje AI systém se systémem schopností, což by mohlo v budoucnu dělat problémy, například pokud bychom chtěli, aby i hráči mohli používat schopnosti.

V kapitole Předměty 4.3 jsme si vymysleli zajímavý modulární systém, kde můžeme uložit efekty do jednoho `ScriptableObject` a poté efekty vyvolat, když chceme, tím že předmět použijeme. Což je to, co chceme pro naše schopnosti, a použijeme tedy velmi podobný systém. Vytvoříme si nový `ScriptableObject Ability`, který bude mít uloženy efekty, novou komponentu, která bude mít seznam `Ability` objektů, které může postava použít. AI pak bude moci používat tyto schopnosti analogicky k používání předmětů.

#### 4.2.9. Životy a poškození

Dle game design dokumentu, životy mají postavy a ložiska surovin (viz Ložiska surovin 3.6). Postavy i ložiska také dostávají poškození, když na ně někdo zaútočí. Tyto dva koncepty se liší hlavně v tom, jak na smrt reagují. Postavy vytvoří pickupy ze všech předmětů v inventáři a vybavení, zatímco ložiska vždy vytvoří předem dané pick-upy, podle toho, jakého je ložisko druhu (např. strom vytvoří 3 pickupy

předmětu log). Tento rozdíl ale nesouvisí přímo s životy, ale s tím, jak na změnu životů reagujeme, můžeme tedy jednoduše vytvořit společnou komponentu jak pro postavy, tak pro ložiska a pouze vytvořit nové komponenty, když vyžadujeme specifické reakce.

Naše požadavky pro komponentu životů jsou takovéto:

- 1) chceme si pamatovat maximální životy,
- 2) chceme si pamatovat aktuální životy,
- 3) chceme, aby poškození, které postava dostala, bylo změněno dle odolnosti postavy vůči danému druhu poškození,
- 4) chceme, aby ostatní komponenty mohly reagovat na změnu životů (např. můžeme mít vybavení, které nás náhodně teleportuje, když spadneme pod 50 % životů),
- 5) když máme 0 životů, tak chceme, aby postava umřela a aby na smrt mohly ostatní komponenty reagovat (např. chceme, aby inventář vytvořil pick-upy všech předmětů, které v něm jsou uloženy).

Mohli bychom teoreticky rozdělit změnu poškození v závislosti na odolnosti do své vlastní komponenty, ale to by komplikovalo architekturu, aniž by to pro naši hru přinášelo jakékoli výhody. Pro jiné hry by to ale mohl být lepší přístup.

Například, kdybychom měli odolnost vůči ohni ze dvou zdrojů, 10 % z kouzelného amuletu a 10 % z brnění. Každé z těchto vybavení by vygenerovalo svoji vlastní komponentu odolnosti. Poté, kdybychom vkročili do místnosti, kde magie nefunguje, tak bychom chtěli, aby komponenta kouzelného amuletu přestala fungovat a abychom ztratili její 10% odolnost vůči ohni. Toto může detektovat sama komponenta odolnosti a sama se může také vypnout. Pokud bychom ale měli veškerou odolnost sečtenu v jedné hodnotě, aniž bychom věděli, odkud jsme ji dostaly, tak by pro nás bylo složitější detektovat, kdy tyto odolnosti měnit v takovýchto situacích.

V našem projektu tuto vlastnost ale nevyužíváme, takže můžeme zpracování odolnosti vložit do stejné komponenty jako zbytek požadovaných vlastností.

Je důležité podotknout, že máme dva signály, které posíláme, signál o změně životů a signál o smrti. Mohli bychom si říct, proč posílat signál o smrti, když signál o změně životů posílá novou hodnotu životů, a my víme, že když je počet životů menší než 0, tak jsme vždy mrtví? Problém, který může nastat, je, že můžeme aplikovat efekty v reakci na poškození, které nám počet životů může zvětšit. Například, pokud bychom měli prsten, který nás oživí na 10 životů, když nás zasáhne smrtelná rána, tak bychom neumřeli, když se naše životy dostanou pod 0, protože se ve stejném okamžiku zase dostanou nad 0. Proto musíme mít speciální druhý signál, který ohláší smrt až v okamžiku, kdy si jsme 100 % jistí, že postava umřela, tedy až po aktivaci všech efektů v reakci na změnu životů.

#### 4.2.10. Zlato

Zlato (viz Zlato 3.2.5), herní měna, je relevantní pouze pro hráčské postavy, protože pouze ony ho potřebují, aby z obchodů kupovaly předměty. Požadavky pro systém, který se o množství zlata bude starat, jsou:

- 1) uložení a zpřístupnění množství zlata,
- 2) oznamení změny množství zlata.

Tento systém můžeme implementovat v samostatné komponentě, což nám zajistí přehlednost a umožní nám možnost tento systém použít i u ostatních objektů, kdybychom změnily design hry a chtěli bychom, aby zlato používaly i AI postavy, nebo obchody (obchody nyní akceptují nebo generují zlato z ničeho a nemají žádnou

vnitřní peněženku). Alternativně, jelikož zlato používají pouze hráči, tak bychom mohli implementovat zlato jako součást nějaké komponenty, kterou využívají pouze hráčské postavy.

Vybereme si první variantu, implementace v samostatné komponentě, hlavně proto, že pro druhou variantu nemají hráčské postavy nemají žádné unikátní komponenty, do kterých by dávalo smysl přidat tento systém.

## 4.3. Předměty

Předměty v naší hře existují ve vícero formách. Existují v podobě pickupů, jako předměty v inventáři, truhlách a obchodech nebo jako vybavení využívané postavou.

Jak funguje vybavení, jsme již vysvětlili (viz Vybavení 4.2.4) a pickupy vysvětlíme později (viz Pick-upy).

Předměty v inventářích nemají žádnou speciální funkci a pouze drží informace o daném předmětu a jeho efektech. Data předmětů budeme ukládat v podobě ScriptableObject. Tento objekt bude držet všechny informace o předmětu, jeho typ, ceny, zda je použitelný, jak dlouho trvá použít, je vybavení atd. Musíme si ale rozmyslet, jak implementovat efekty *použití* předmětů a *využití* vybavení a také jak ukládat recepty pro vyrábění předmětů, což si rozmyslíme v následujících podkapitolách.

### Efekty použití a využití

V sekci Vybavení 4.2.4 jsme již přišli na způsob, jak dynamicky vytvářet komponenty efektů *využití* a toto řešení můžeme použít i pro efekty *použití*.

Vytvoříme si komponentu pro každý efekt *použití*, které budou aplikovat svůj daný efekt, a pomocné třídy efektů *využití*, které tyto komponenty vygenerují s předem danými parametry. Pokud je efekt jednorázový, tak můžeme, pro efekty *využití*, komponentu opominout a aplikovat efekt přímo z pomocné třídy. Toto můžeme dělat jenom u efektů *použití*, protože ty se vždy načtou z pomocných tříd, vždy, když použijeme předmět. Efekty *využití* jsou ale vždy vygenerovány jako komponenty pouze jednou, když využijeme vybavení, a pak se opakován aplikují, např. když někoho zasáhneme zbraní.

Například, máme **elixír rychlosti**. Když ho *použijeme*, tak se zavolá funkce *použití* na všech pomocných třídách v seznamu efektů **elixíru rychlosti**. Elixír má pouze jednu takovou třídu, a to **pomocnou třídu zvýšená rychlosť** s uloženou hodnotou 50 %, udávající, o kolik se zvýší rychlosť postavy a hodnotu 2 udávající, na kolik minut zrychlení platí. Funkce pomocné třídy vygeneruje komponentu **zvýšená rychlosť**, kterou inicializuje s hodnotami 50 % pro míru zvýšení rychlosť a 2 pro trvání zrychlení, poté tuto komponentu připojí k postavě. Komponenta **zvýšená rychlosť** zvýší rychlosť postavy o 50 % a za dvě minuty rychlosť sníží na původní úroveň a sama se odstraní od postavy.

Pokud bychom měli elixír oživení, který nám zvýší životy o 40, tak k tomuto nemusíme používat komponentu, a můžeme životy zvýšit přímo z pomocné třídy, protože tento efekt nemá žádné trvání.

Další věc, kterou musíme vyřešit je, jak vytvořit instance pomocných tříd a uložit je do seznamu efektů ve ScriptableObject předmětu. Chceme, aby tyto třídy mohly být vytvářeny a parametrizovány designérem přímo v Unity editoru, toto chceme nejenom proto, že to je správný postup, ale i proto, že to usnadní práci nám, až budeme vytvářet předměty a hledat pro jejich efekty správně hodnoty. Tohoto snadno dosáhneme vytvořením Unity editor scriptu, který všechny pomocné třídy

načte a ke každému vygeneruje tlačítko v editoru, které umožní vytvořit instanci dané třídy.

## Výroba předmětů

Výroba předmětů funguje v naší hře relativně jednoduše, stačí najít předměty, které se dají použít jako ingredience pro výrobu jiného předmětu, přijít k výrobnímu objektu a jednoduše předmět vyrobit (viz Vyrábění předmětů 3.8). Samotné předměty se o výrobu nestarají, o to se starají vyráběcí objekty, ty, ale potřebují znát recepty pro každý vyrobitevný předmět. Otázkou je, jak by tyto recepty měly být strukturované a kde by měly být uložené? Pro místo uložení máme dvě volby:

- 1) budeme mít všechny recepty uloženy v jednom seznamu, v tom případě si každý recept musí pamatovat ingredience i výsledný produkt,
- 2) každý předmět v sobě bude mít uložený seznam ingrediencí, které jsou zapotřebí, aby byl daný předmět vyroben, každý předmět také bude mít označení, zda vůbec vyrobitevný je.

Během implementace naší hry jsme vyzkoušeli obě varianty, nejprve variantu 1 a poté variantu 2. Varianta 1 má výhodu v tom, že jsou všechny informace centralizované, takže je můžeme snadno měnit i načítat, zároveň bychom mohli mít více receptů, které vytváří stejný předmět, což ve variantě 2 nelze udělat bez změn. Problém, na který jsme narazily během implementace, byl ale s nepřehledností. Když máme všechny recepty v jednom seznamu, tak se v něm dá těžko vyznat, musíme dlouho hledat, když chceme změnit specifický recept, a není snadné zjistit, zda už pro nějaký předmět recept máme. Z tohoto důvodu jsme se nakonec rozhodli použít variantu 2. V této variantě máme totiž recepty předmětů vždy uloženy ve ScriptableObject daného předmětu, takže víme, kde ho hledat a snadno zjistíme, jestli pro předmět recept už máme nebo ne.

## 4.4. Objekty

Objekty jsou skupina GameObjects, se kterými mohou postavy interagovat (viz Objekt 3.7). Počítají se mezi ně dveře, obchody truhly, vytvářecí objekty, posteže apod. NPC postavy se za objekty nepovažují, ale i s nimi se dá interagovat stejným způsobem, jako s ostatními objekty, skrz náš dialogový systém. V podkapitole Vstup hráče 4.2.6 jsme si už rozmysleli, že chceme, aby každý objekt měl vlastní komponentu, ve které implementuje svoji funkcionalitu, ale aby všechny komponenty dědily od stejného rodiče, abychom se všemi objekty mohli interagovat stejným způsobem přes stejný interface (přes veřejné metody). Důvod, proč nepoužíváme Interface (C# Interface), ale dědíme od otce, je že chceme, aby si každý objekt zdědil i pole, ve kterých ukládáme důležité informace, např, kteří hráči s objektem právě interagují. Objekty existují fyzicky v herním světě, takže budeme chtít, aby měli hitbox a, aby přes ně nemohli procházet postavy, nechceme ale aby mohli být sami hýbány. Toto implementuje přidáním RigidBody2D a Collider2D komponentu správného tvaru. Některé objekty mají speciální požadavky ohledně jejich fyzické přítomnosti, a u nich provedeme hlubší analýzu.

Máme obecně 3 druhy objektů, jednorázově, dlouhodobé a objekty s UI, které si nyní zvlášť probereme.

### 4.4.1. Jednorázové

Jednorázové objekty jsou ty, se kterými interakce proběhne okamžitě po započetí. Například dveře jsou jednorázový objekt, ty se po interakci bud' otevřou

nebo zavřou. Je důležité podotknout že interakce je okamžitá pouze z pohledu postavy, ta pouze zainteraguje s dveřmi a je jí okamžitě umožněno vykonávat jiné akce, ale dveře na interakci reagují animací, která nějaký čas trvá.

Dále následuje analýza jednorázových objektů.

### Pick-upy

Pick-upy jsou objekty, které představují nějaký předmět v herním světě. Po interakci přidají do inventáře postavy předmět, který představují, a okamžitě se odstraní z herního světa. Aby hráči věděli, jaký předmět dostanou, tak bude každý pickup mít sprite předmětu, který reprezentuje. Z game design dokumentu (viz Pick-upy 3.3.2) víme, že chceme, aby se mohli hýbat v reakci na fyzické síly (např. exploze), proto jim přidáme **RigidBody2D**. Když nastane nějaká exploze, tak její efekt knockback (viz Zbraně a jejich vlastnosti 3.4.1) automaticky předá požadované síly komponentě **RigidBody2D**, která pak už pohyb počítá sama. Aby kolize vůbec mohly být detekovány tak musíme přidat komponentu zadávající tvar kolize. Jelikož nechceme kolidovat s postavami ani jinými objekty a chceme jenom detekovat, zda jsme v dosahu nějakého knockback (viz Zbraně a jejich vlastnosti 3.4.1) efektu, tak nám na kolizním tvaru moc nezáleží, proto si vybereme **BoxCollider2D**, protože je počítání kolizí pro tento tvar levné.

### Dveře

Dveře mají intuitivní funkci, naše požadavky pro ně jsou:

- 1) fyzická přítomnost, která brání vstupu do budov nebo místností,
- 2) možnost dveře otevřít (a poté zavřít), což umožní vstup do budovy,
- 3) schopnost AI navigovat přes dveře.

Zabránění vstupu zajistíme využitím Unity fyzikálního systému, jednoduše tím, že objektu připojíme **RigidBody2D** a **BoxCollider2D**.

Otevírání a zavírání můžeme řešit dvěma způsoby. Bud' vytvoříme Unity animaci, kterou dveře otevřeme (tím že otočíme dveře kolem osy pantů), nebo vytvoříme animaci přes kód, pomocí interpolace rotace dveří (také s rotační osou v místě pantů). V jednoduchém případě jako je otevírání dveří nemá ani jeden přístup nějakou speciální výhodu, tak je relativně jedno, který přístup si vybereme. My jsme se rozhodli použít Unity animace.

Navigace AI přes dveře má mnoho komplikací, souvisí spíše ale s implementací AI než s implementací samotných dveří. Dveře tedy necháme tak jak jsou, a jak s nimi bude interagovat AI si rozebereme až v kapitole Navigace skrz dveře 4.5.2.

### 4.4.2. Dlouhodobé

Dlouhodobé objekty jsou ty, se kterými trvá interakce delší dobu. Poprvé, když s takovým objektem postava interaguje tak si zapamatuje, že s dlouhodobým objektem započala interakci, a dalším pokusem o interakci (např. dalším stlačením tlačítka pro interakci) se interakce vypne. Příkladem dlouhodobého objektu je postel. Když postava s postelí zainteraguje tak ztratí schopnost pohybu, je přemístěna na postel a po dobu interakce se postavě obnovují životy (toto reprezentuje spánek v posteli), když pak postava znova zainteraguje, tak se interakce vypne, postava je přemístěna na místo kde interakci započala a je jí vrácena schopnost pohybu. Postel je zatím jediný dlouhodobý objekt.

### 4.4.3. Objekty s UI

Pokud s objekty s UI interaguje AI postava, probíhá interakce tak, že se AI rozhodne, jaké akce s objektem chce vykonat a poté zavolá veřejné metody objektu

které tyto akce uskuteční. Například, AI chce uložit do truhly 1 log, musí se tedy nejprve přesunout do blízkosti truhly, aby s ní mohla interagovat, a až bude dost blízko zavolá metodu `PutItemInChest(log)`, která přesune předmět log z inventáře postavy do inventáře truhly.

Pro hrácké postavy se tyto objekty vždy chovají jako dlouhodobé objekty, s tím, že se po interakci vždy vygeneruje UI a zpracování vstupu se přepne do UI kontextu, aby hráč mohl UI navigovat a neovládat při tom postavu (viz Vstup hráče 4.2.6).

Jelikož tyto objekty používají UI, tak se musíme nejprve rozhodnou, jak UI implementovat a poté bude následovat analýza objektů s UI.

## UI

UI je grafické rozhraní, a vyžadujeme po něm tyto vlastnosti:

- 1) zobrazovat sprity,
- 2) vypisovat text,
- 3) tlačítka, a jiné interaktivní elementy, jako např. výběr ze seznamu položek,
- 4) navigace mezi digitálními tlačítky pomocí klávesnice, specificky pomocí 4 klávesnicových tlačítek, které udají směr pohybu, např tlačítka šipek, kde, když zmáčkneme šipku nahoru, tak se přesuneme na digitální tlačítko, které je nad právě označeným tlačítkem,
- 5) aby navigace fungovala nezávisle mezi různými hráči.

Naštěstí pro nás, má Unity už implementuje body 1, 2 a 3 UI systém, a body 4 a 5 jsou splněny pomocí Unity Input Systém, který má komponentu `MultiplayerEventSystem`, která se stará o UI navigaci přiřazeného hráče (každý hráč má jednu tuto komponentu). Využijeme tedy tyto předpřipravené systémy. Musíme si ale dát pozor na bod 5, Unity umožňuje automatickou navigaci pomocí klávesnice tak, že se po vytvoření instance tlačítka se ke každému kardinálnímu směru přiřadí jiné tlačítko, na které se přesuneme, když se chceme pohnou v tomto směru. Toto funguje dobře pro jednoho hráče, ale přestane to fungovat, když je hráčů více, může totiž nastat tento problém:

Hráč A otevře svůj inventář, tím se vytvoří instance menu inventáře a vygenerují se spojení mezi tlačítky, poté přijde hráč B vedle hráče A a také otevře svůj inventář, vytvoří se druhá instance menu inventáře a vygenerují se spojení mezi tlačítky, protože jsou hráči, ale blízko sebe tak budou i obě menu blízko sebe, a může se tedy stát, že se vytvoří spojení mezi tlačítky menu hráče B a tlačítky menu hráče A.

Tento problém můžeme vyřešit tak, že automatickou generaci spojení vypneme a místo toho bud' vytvoříme spojení manuálně v Unity editoru, nebo vytvoříme vlastní generátor spojení. Manuální přístup má očividné problémy, zabere hodně času, a pokud se rozhodneme udělat nějakou změnu, tak budeme muset všechna spojení zase předělat. Automatický systém takové problémy nemá, ale jeho implementace není tak triviální, jak se zdá.

Když jsme na tento problém přišli my, tak jsme si napsali jednoduchý generátor spojení, který spojil nejbližší tlačítka v  $90^\circ$  oblouku v daném směru (vzdálenost jsme ještě vázili podle výchylky od požadovaného směru). Tato implementace nám bohužel nevyhovovala, největší problém nám dělala v menu vybavení (viz Hráčův inventář 3.10.1), kde jsme se bud' nemohli dostat na prostřední pole, nebo na jedno z levých polí. Tento problém nám nešel vyřešit, tak jsme si řekli, že když už předem víme, jak budou všechna menu vypadat z game design dokumentu, a když jsou pouze 4, tak můžeme spojení vytvořit manuálně a nebát se, že je budeme muset předělat (např. kvůli změně velikosti inventáře postavy nebo truhly). Spojení jsme tedy nakonec vytvořily manuálně.

Dále jsme si všimli, že se v našich menu objevují stejné podmenu, například podmenu ukazující obsah inventáře se objevuje v menu inventáře, menu truhly a s malou změnou ukazování ceny i v menu obchodu (viz GUI 3.10). Abychom tyto elementy nevytvářeli opakovaně, tak si z nich můžeme vytvořit Unity prefab, který pak můžeme používat v různých menu.

### Truhla a obchod

Jak truhla, tak i obchod slouží jako uložiště předmětů, jediný rozdíl je, že hráči dostanou zlato, když předmět uloží do obchodu a musí zlatem zaplatit, když předmět vyjmou, což je triviální rozdíl a můžeme je tedy analyzovat stejně. Naše požadavky jsou:

- 1) skladování předmětů,
- 2) možnost předměty vyjmout a vložit,
- 3) UI pro hráče.

Skladování předmětu jsme už vyřešili pomocí komponenty inventáře, viz Inventář 4.2.3. Metody pro vyjmání a vkládání předmětů můžeme vložit do nové komponenty **Chest**. UI vytvoříme tak, aby vypadalo jako v game design dokumentu (viz Truhla a obchod 3.10.2), každé pole bude tlačítko, které bude mít vlastní index, a když bude stlačeno, tak předá komponentě **Chest** informaci o tom, že jsme ho stlačili. **Chest** pak implementuje chování přesouvání předmětů v reakci na tyto signály, tak, jak bylo popsáno v game design dokumentu (viz Hráčův inventář 3.10.1).

### Kovadlina, pracovní stůl a výheň

Tyto objekty se liší pouze v tom, jaké mohou zpracovávat recepty, jinak jsou identické. Jediný požadavek, který máme, je, že chceme, aby byli schopny odstranit předměty ingrediencí z inventářů postav a mohli přidat předmět výsledný předmět. Tyto funkce můžeme jednoduše implementovat v nové komponentě **Crafter**. **Crafter** si bude pamatovat, jaký druh výrobního objektu je bude načítat recepty do UI a také kontrolovat, zda hráč má potřebné ingredience, které pak v UI správně obarvíme (viz Vyrábění předmětů 3.8)

### Dialogový systém

Dialogový systém musí umožňovat výběr dialogových možností vedoucích do různých pod-dialogů a musí moci ovlivnit stav světa, například aktivací spícího golema.

UI vytvoříme dle návrhu v game design dokumentu (viz Dialogový systém 3.11). Pro vytvoření větvícího se dialogu je očividně nejlepší použít stromovou datovou strukturu, chceme ale, aby strukturu, obsah a efekty dialogu mohl měnit designér přímo v editoru. Vytvoříme si tedy nový ScriptableObject reprezentující jednu episodu dialogu, kterou nazveme **Dialogue**. **Dialogue** bude obsahovat, požadavky (např. mít specifický předmět v inventáři), seznam efektů, které se aktivují, když se daná epizoda načte, text dialogu, seznam textů akcí, které hráč může vybrat a odkazy na další dialog, na který se přesune, když si vybere danou akci. Aby si hráč mohl vybrat akci tak musí být splněny požadavky dialogu, na který se odkazuje. Implementace textu a odkazů je triviální, ale požadavky a efekty jsou složitější. Jak implementovat efekty jsme si ale už rozmysleli pro používání předmětů a využívání vybavení (viz Vybavení 4.2.4 a Předměty 4.3) Požadavky bychom mohli implementovat stejným způsobem jako efekty, ale protože jsme si předem nerozmysleli, jaké požadavky budeme potřebovat, jsme jednotlivé požadavky vymýšleli postupně během vývoje a implementovali je přímo do třídy **Dialogue**

abychom jeho fungování mohli rychle a snadno otestovat. Až jsme měli všechny požadavky implementované, tak už nebyl důvod implementovat modulární systém, protože jsme věděli, že další požadavky už implementovat nebudeme.

### Podstavec

Podstavec je objekt, kterému musíme odevzdat vítězný předmět, čímž vyhraje nás tým hru. Vlastnost kontroly, zda nějaký předmět máme, a spuštění nějakého efektu pouze po splnění požadavku už existují v dialogovém systému, takže můžeme jednoduše využít dialogový systém k implementaci podstavce.

### Spící golem

Spící golem reprezentuje neaktivního golema. Když do golema vloží hráč šém, tak chceme, aby se z neaktivního golema stal aktivní golem v podobě postavy, který půde na stejném týmu jako hráč a bude poslouchat hráčovi rozkazy jako normální voják. Spícího golem by mohl být reprezentován jako postava, z toho by ale mohli nastat problémy, AI postavy by se mohli teoreticky rozhodnout na něj zaútočit, což nedává moc smysl, když je neaktivní, také, jako postava by mohl být fyzicky odtačen, například do základny jednoho hráče, druhý tým by pak těžko mohl aktivovat golema, i kdyby měli šém, stejně tak nechceme, aby tým, který šém nemá, nešel zničit golema předtím, než se aktivuje. Nechceme tedy aby byl spící golem zničitelný, přesunutelný ani viděn jako postava NPC postavami, což jsou všechno vlastnosti objektů, a proto bude spící golem objekt. Pro aktivaci golema potřebujeme specifický předmět, šém, chceme pak dát hráči na výběr, zda golema aktivovat nebo ne, k čemuž se přesně hodí dialogový systém, který tím pádem využijeme.

## 4.5. NPC Navigace

NPC postavy potřebují mít schopnost navigace v herním světe. Tento problém má několik složek, které musíme vyřešit:

- 1) navigace v těsných prostorech, jako místnosti koridory budov,
- 2) navigace v rozsáhlých vnějších prostorech, kde je méně překážek, například travnaté pole nebo řídký les,
- 3) vyhýbaní se ostatním postavám,
- 4) schopnost navigovat cesty skrz dveře.

Body 3 a 4 jsou speciální případy, které rozebereme později, nejprve se tedy rozhodneme, jaký navigační systém použít, který by splňoval požadavky 1 a 2. Máme několik možností:

- 1) Unity už má navigační systém Unity **Navmesh**, tento systém sice funguje pouze pro 3D hry, ale můžeme udělat několik triků, aby fungoval i pro naši 2D hru. Trik, jak tuto variantu využít spočívá v tom, že svoji hru budeme vytvářet v 3D projektu a pro každý neprůchodný objekt vytvoříme 3D GameObject, který Navmesh, na rozdíl od 2D GameObjectů, už dokáže zpracovat.
- 2) Mohli bychom si stáhnout NavMeshPlus rozšíření pro unity (viz Zdroj 6), které implementuje Navmesh pro 2D hry, čímž nám usnadní její implementaci, ale není to oficiální rozšíření, takže může mít nějaké problémy.
- 3) Můžeme stáhnout A\*Pathfinding Projekt (viz Zdroj 1), který má dvě verze, placenou a zadarmo. Placenou verzi si nemůžeme dovolit, takže nás bude zajímat pouze verze zadarmo. Tato verze implementuje grid based navigaci,

ta funguje tak, že rozdělí herní svět do mřížky s námi daným rozměrem a rozlišením, a každou buňku bud' označí jako průchodnou nebo ne, podle toho, zda koliduje s neprůchodným GameObjectem (druh objektů, které mi definujeme, např. stěna nebo strom). Jedno omezení, které tato verze má, je že velikost mřížky může být pouze  $1000 \times 1000$  buněk, a protože chceme mít dost veliké rozlišení, aby mohli NPC navigovat v těsných prostorech, tak se může stát, že bude velikost naší mřížky moc malá, aby pokryla celou herní mapu.

- 4) Můžeme implementovat gridbased nebo navmesh navigaci manuálně.

Manuální implementace nedává moc smysl, když máme na výběr z tolka dobrých implementací. Varianta 1 se pro naši 2D hru nehodí, a je dominována variantou 2, která, přestože je neoficiální, je lidmi používaná (toto usuzujeme z githubové stránky NavMeshPlus projektu, kde má 2000 hvězdiček a 223 forků, viz Zdroj 6), takže bude pravděpodobně mít dostatečně dobrou kvalitu. Rozhodujeme se tedy mezi verzí 2 a 3. Rozhodujeme se tedy mezi výhodami gridbased a navmesh navigace.

Gridbased navigace může rychleji počítat, zda její buňka je průchodná nebo ne, protože se vždy jedná o jednoduché čtverce (nebo kruhy) rozestavěny v mřížce. Navmesh na druhou stranu rozdělí plochu, kterou musí navigovat, dynamicky na konvexní tvary, které rozdělují plochu na průchodné a neprůchodné regiony, tyto regiony pak musí pospojovat dle toho, zda spolu sousedí. Dlouhé počítání průchodnosti moc nevadí, když se děje pouze na načítání levelu, ale může dělat problémy, když herní svět často mění, například ničením nebo stavěním budov, takovéto změny ale často neděláme, takže pro naši hru to moc veliká výhoda není.

Navmesh má ale rychlejší a detailnější navigaci. Rychlejší, protože v prostorech s málo překážkami generuje veliké regiony, takže jich A\* musí prohledat méně než u gridbased, aby se dostal do cílové polohy. Detailnější, protože v úzkých prostorech s mnoha překážkami může přesně kopírovat kontury překážek a koridorů, na rozdíl od gridbased, který i s malým rozlišením (např. buňka  $1/3$  velikosti postavy) může mít relativně veliký odstup od stěn a může mít veliký problém s úzkými diagonálními koridory, které nejsou zarovnané s mřížkou (koridorem můžeme projít, ale leží mezi dvěma buňkami, které se obě dotýkají stěn koridoru, takže jsou obě neprůchodné a cesta se teda přes koridor nenajde).

Z naší analýzy vyplívá že varianta 2 je očividně lepší pro naši hru než varianta 3, mi jsme si ale vybrali variantu 3 z těchto důvodů:

- 1) variantu 2 jsme našli až v pozdní fázi vývoje,
- 2) na omezení  $1000 \times 1000$  buněk, ve variantě 3, jsme také přišli až v pozdní fázi vývoje, když jsme variantu 3 už vybrali a integrovali do naší hry, naštěstí pro nás, byl tento počet buněk dostatečný pro velikost naší herní mapy.

Kdybychom programovali takovýto projekt znova, určitě bychom vybrali variantu 2, ale pro tento projekt budeme používat variantu 3.

#### 4.5.1. Vyhýbaní se ostatním postavám

Náš navigační systém řeší navigaci v statickém světě, kde se překážky nepohybují, v naší hře ale existuje více pohyblivých postav a my chceme, aby se navzájem vyhýbali a navzájem se nezasekávali, např. v úzkém vchodu nebo koridoru.

Tento problém můžeme vyřešit několika způsoby, můžeme z postav vytvořit neprůchodné objekty, a updatovat kolem nich navigační mřížku, když se pohnou. Toto řeší problém kolizí ve volných prostorech, ale neřeší to problém v koridorech, když je postava v koridoru tak ho zablokuje a pokud máme dvě postavy které chtějí projít na opačnou stranu koridoru, tak se navzájem zablokují a ni jedna nebude moci projít (postavy nenajdou platnou cestu k cíli, takže se nebudou hýbat).

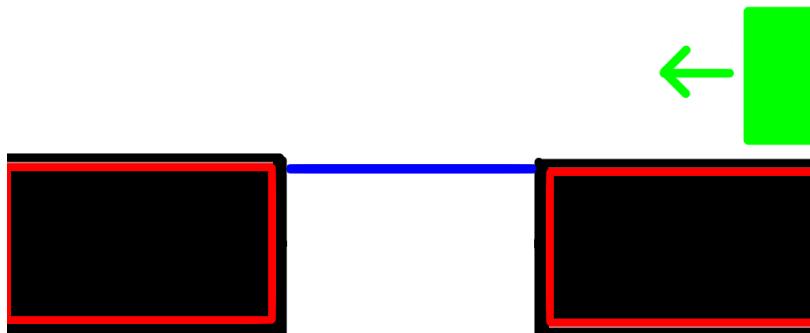
Místo modifikace navigace, můžeme přidat nový systém steering (viz Zdroj 7). Steering je systém, který se snaží dynamicky upravovat směr pohybu, např. za cílem vyhnutí se překážce, bez zapojení dlouhodobého plánovaní. Funguje takto, nejprve dostane hlavní vektor značící směr, kterým se chce pohnout (např. směr k dalšímu vrcholu v pathfinding grafu), poté zjistí, kde se nacházejí překážky (např. pomocí raycastu) a zjistí vektor směrující k těmto překážkám a vydělí je vzdáleností od překážek (chceme, aby navigaci vzdálenější překážky moc neovlivnily, ale blízké, aby měly veliký vliv), potom od hlavního vektoru tyto vektory odečte, výsledný vektor bude náš nový směr pohybu. Toto zase ale funguje pouze ve volných prostorech, ale v úzkých koridorech není kam uhnout a obě postavy se zase zaseknou. Rozhodli jsme se tedy kolize mezi NPC vypnout, aby skrz sebe mohli volně procházet. Toto vyřeší problém kolizí, ale může vyústit v neuhledné překrývaní postav, což bychom mohli vyřešit steeringem nebo vytvořením kruhového kolizního objektu uprostřed každé NPC postavy, tento kolizní objekt by byl menší než sprite postavy, postavy by tedy kolem sebe mohly proklouznout, ale nemohly by se přesně překrývat. Mi jsme nakonec nevyužili ani jednu z těchto metod, protože se při našem testování ukázalo že překrývání NPC postav nebylo vizuálně problematické, jak se nám původně zdálo.

#### 4.5.2. Navigace skrz dveře

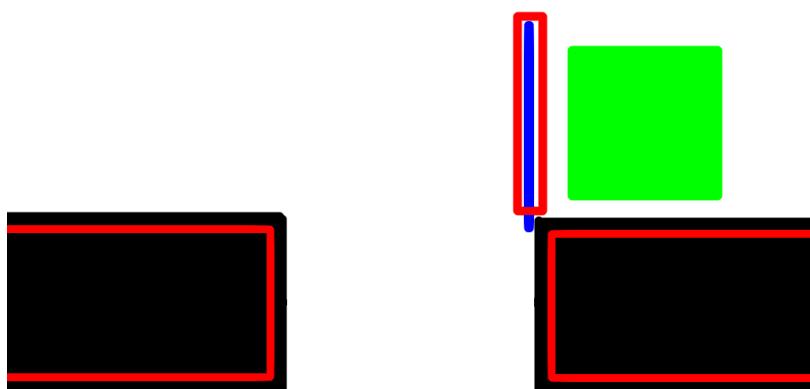
Navigace v herním světě s dveřmi může být velice složitá, podle toho, jak ve hře dveře fungují. Například, pokud mohou dveře být zamčeny nebo odemčeny, tak musíme brát během navigace v úvahu nejenom, zda jsou dveře zamčené, ale také, zda nemáme klíč, který by je mohl odemknout, nebo zda bychom někde mohli klíč získat. Naštěstí pro nás je fungování dveří, v naší hře, jednoduché (viz Dveře 3.7.3), nemáme zámky a jakákoliv postava může otevřít nebo zavřít kterékoliv dveře. Tento fakt nám značně usnadní navigaci v našem světě, protože nemusíme dveře brát v úvahu během plánování, protože můžeme dveře vždy otevřít, když se skrz jedny rozhodneme projít. Otevírání dveří můžeme zajistit jednoduchou komponentou, která detekuje dveře v blízkosti postavy a automaticky je otevře a potom, co se postava od dveří vzdálí, je zavře.

Je důležité nezapomenout, že dveře jsou fyzickým objektem, který brání pohybu skrz něj, což platí, když jsou otevřeny i zavřeny, jediný rozdíl mezi těmito stavami je otočení dveří o  $90^\circ$ . Toto nám může způsobit tento problém, představme si, že se ke dveřím (zobrazených zde Obrázek 21) blížíme z horní pravé strany a chceme projít dveřmi. Když se ke dveřím přiblížíme, tak je otevřeme a zasekneme se mezi dveřmi a stěnou. Protože dveře během navigace ignorujeme tak takto zůstaneme zaseknuti navždy. Toto můžeme napravit tak, že když se dveře otevřou, tak oznamí navigačnímu systému, aby tyto dveře už neignoroval, mi si pak uvědomíme, že tudy nemůžeme projít a dveře obejdeme. Až se vzdálíme tak dveře zavřeme a ty oznamí navigačnímu systému, aby byly zase ignorovány.

● Stěna      ● Neprůchodný prostor  
 ● Dveře      ● Postava



-----



Obrázek 21

## 4.6. NPC AI

Naše NPC mají často více cílů (např. udržet životy nad 50 % a těžit železnou rudu) a také mají jinou prioritu, když těžíme železo, ale někdo nás zraní a padneme pod 50 % životů, tak bychom chtěli hned přestat těžit zlato a zkusit se někde oživit. Jeden z našich požadavků pro AI je tedy, aby bylo schopné pracovat s vícero cíli a bylo schopné dynamicky měnit aktivní cíl v závislosti na herní situaci. Zároveň, protože různé NPC budou mít různé cíle, chceme, aby cíle byly modulární a my je mohli libovolně přidávat a odebírat.

Kvůli složitosti herního světa také není vždy očividně jasné, jak daný cíl splnit. Například, pokud chceme těžit železnou rudu, tak potřebujeme krumpáč, krumpáč můžeme buď najít v nějaké truhle, nebo ho můžeme vyrobit z ingrediencí, které také musíme někde získat. Z tohoto vyplívá další požadavek, chceme, aby naše AI mohlo plánovat do budoucnosti.

Chceme, aby náš AI systém měl modularitu vzhledem k hernímu světu. Během vývoje nechceme vždy měnit kód AI, když přidáme nějaký nový předmět, např. nový meč, tak chceme, aby ho AI vnímala a používala jako jakýkoliv jiný meč. Nebo

např., aby se AI chovalo stejně k obchodům a truhlám, protože pro NPC není v mezi těmito dvěma objekty rozdíl.

Také chceme modularitu vzhledem k akcím, chceme být schopni libovolně odebírat nebo přidávat akce, které AI může použít. Například golem má schopnost vyšlehnout oheň, kterou ostatní postavy nemají, takže mu přidáme akci, která AI tuto schopnost umožní použít. Nebo kdybychom chtěli přidat nový druh postavy, která nemá inventář (např. pes) tak bychom jí chtěli odstranit akce, které s inventářem pracují.

Naše požadavky tedy jsou:

- 1) schopnost prioritizace cílů,
- 2) plánování sekvence akcí,
- 3) modularity cílů,
- 4) modularita herního světa,
- 5) modularity akcí.

#### 4.6.1. Analýza architektur

Na výběr máme z mnoha architektur např. state machines (Zdroj 8), decision trees (Zdroj 9) nebo GOAP (Goal Oriented Action Planning).

State machines se skládá ze stavů, ve kterých se AI, a postava kterou ovládá, může nalézat (stavy mohou reprezentovat cíle, akce, nebo stavy samotné postavy, např. nečinnost, útok, skok nebo pád) a hrany, které popisují, jak mezi stavů můžeme přecházet (např. ze stavu pád se můžeme dostat do stavu nečinnost, ale ne do stavu skok). Problém s takovouto architekturou je, že když přidáme nový stav (např. stav reprezentující akci), tak musíme vždy explicitně přidat nové hrany, které do tohoto stavu vedou. Toto není pouze náročné na udržování, nepřehledné, pokud máme hodně akcí, nemodulární, což je jeden z našich požadavků, ale také neumožnuje dynamické plánování vícera akcí (stavů) za sebou, protože hrany mezi stavů jsou dané designérem. A co je nejdůležitější, tato architektura dokáže smysluplně zvládnout pouze plnění jednoho cíle. Když máme totiž více cílů, tak se v každém stavu (hlavně pro stavy reprezentující akce) musíme rozhodovat jinak pro každý cíl, což v znamená že pro každý cíl musíme vytvořit jeho vlastní state machine, ve kterém jsou stavy propojené tak, aby daný cíl plnily, cože je příliš moc práce a state machines se očividně pro naši hru nehodí.

Decision tree, je strom (druh grafu), který se skládá z vrcholů akcí (vykonají danou akci), nebo z logických vrcholů (rozhodnou se do jakého vrcholu přejít, v závislosti na daných podmínkách). Když chceme, aby AI vybralo akci, tak začneme v kořeni stromu a procházíme jím (tak jak nám logické vrcholy přikazují) dokud se nedostaneme do vrcholu akce, kterou pak vykonáme. Struktura stromu a podmínky logických vrcholů jsou dány designérem, takže nám zase schází modularita, a stejně jako ve state machines nemůžeme dynamicky plánovat, a proto se pro naši hru nehodí.

GOAP (Goal Oriented Action Planning) má k dispozici seznam cílů, které musí splnit, seznam akcí, které má k dispozici a stav světa pro výběr akcí během plánování. Tato architektura se vždy snaží vybrat tu nejlepší akci, která splní ten nejdůležitější cíl a připadá nám nejslibněji, takže provedeme hlubší analýzu.

#### 4.6.2. Analýza GOAP

Cíle a akce architektury GOAP jsou jasně definovány, ale plánování samotné striktně definováno není, z různých zdrojů jsme vypozorovali, že jsou používány primárně 2 přístupy, které si neoficiálně nazveme *triviální* a *komplexní*. Nejprve vysvětlíme, jak funguje tato architektura s triviálním plánováním, a poté vysvětlíme, jak ho rozšiřuje komplexní plánování.

Aby mohl GOAP dynamicky plánovat, tak ho musíme průběžně volat jeho algoritmus, aby mohl případně měnit aktuální cíle nebo akce. Když GOAP algoritmus zavoláme, tak si nejprve spočítá priority svých cílů (cíle si svoji prioritu počítají dynamicky, např. čím déle jsme nespali, tak tím větší prioritu budeme mít pro cíl spánek) a pokusí se splnit cíl s největší prioritou. Pokud tento cíl nepůjde splnit, tak se pokusí splnit následující cíl atd. Cíl se pokusí splnit výběrem nejlepší akce, která daný cíl splňuje. V *triviálním* přístupu, má každá akce přidaný seznam cílů, které splňuje a cenu vykonání akce, když se snažíme splnit cíl tak stačí najít akce, které náš cíl splňují, zkонтrolovat, zda mohou být vykonány a z vykonatelných vybrat tu akci, která má nejnižší cenu. Vybranou akci pak začneme vykonávat. Pokud později najdeme akci, které splňuje důležitější cíl tak aktuální akci ukončíme a začneme plnit tu nově nalezenou. Je vidět že tento *triviální* GOAP už splňuje požadavky 1,3, a 5. A přestože požadavek 2 přímo nesplňuje, existuje trik, který umožní plánování několika akcí za sebou, tento trik spočívá v rozdělení cíle do podcílů (např. rozdělej oheň rozdělíme na: 1) sezeň dřevo, 2) postav ohniště, 3) zapal ohniště), kde každý podcíl splní požadavek některého následujícího podcíle. GOAP tímto způsobem může aplikovat více akcí pro splnění složitějšího cíle.

*Triviální* přístup funguje dobře pokud dokážeme namapovat akce na cíle, které splňují, pokud jsou ale naše akce moc specifické na to nějaký cíl splnit, tak budeme muset na plánovat více akcí za sebou. V *komplexním* plánování budou akce mít přiděleny podmínky (musejí být splněny v daném stavu světa, abychom mohli vykonat danou akci) a efekty (jak stav světa změní po vykonání) a cíle budou mít přiděleny podmínky splnění (když jsou splněny podmínky, tak je splněn cíl). Algoritmus pak funguje stejně jako v triviální verzi, s tím rozdílem, že místo jediné akce hledáme plán akcí, které daný cíl splňují. Plán pak nalezneme buď dopředným plánováním (prohledáváním možných stavů světa, do kterých se můžeme dostat aplikováním akcí) nebo zpětným (podobný jako dopředný, ale začínáme ve stavu cíle a otočíme význam efektů a požadavků akcí).

Zpětné prohledávání funguje obecně lépe, a pro naši hru to platí 2krát tak. Např. pokud chceme naplánovat získání zbraně a máme k dispozici obecnou akci *vyrob předmět* tak v dopředném plánování si musíme naplánovat výrobu všech vyrobitelných předmětů ve hře, zatímco v zpětném plánování už víme že chceme meč, takže naplánujeme pouze jednu akci *vyrob meč*. Ve hrách jako je naše, RPG s otevřeným světem (viz Zdroj 11) a vyráběním předmětů, je plánování akcí náročné, kvůli velikému počtu akcí (velikému branching faktoru), takže chceme tento element co nejvíce minimalizovat, abychom mohli nalézt složitější (a snad i lepší) plány. Zlepšení efektivity plánování můžeme dosáhnout nejenom použitím zpětného plánování ale i jiných technik, například můžeme přidat tagy, které říkají, když se snažíme tento cíl splnit, tak použij/nepoužij tuto akci což sníží branching faktor za cenu trocha manuální práce.

Nakonec, aby algoritmus neplánoval nekonečně dlouhé plány, musíme omezit hloubku prohledávání, původně jsme omezily délku plánů na 4 akce, ale z vypisování nalezených plánů během hry jsme zjistili, že plány v drtivé většině času nepřekračují délku 3 akcí a po experimentálním omezení plánů na délku 3 se NPC

stále chovali dle očekávání, protože toto omezení drasticky snížilo výpočetní náročnost plánování jsme se rozhodli omezit délku plánů na 3.

### Výběr plánovací metody

Nyní se musíme rozhodnout, jakou z těchto plánovacích metod použijeme. Triviální, triviální s pod rozdelením cílů nebo komplexní? Nejprve si všimneme, že triviální varianta (i s pod rozdelením) je pod-množinou komplexní varianty, protože efektem akce může být přímo splnění cíle. Z analýzy víme, že pro plánování výroby předmětů budeme potřebovat komplexní plánování, musíme si tedy vybrat tuto variantu, budeme ale využívat výhody triviální varianty, kde můžeme, abychom zrychlili plánování.

Cíle, které jsou jednoduše splnitelné a můžeme na ně přímo namapovat některé akce (např. akce „spát v posteli“ splňuje cíl „potřeba spát“), implementuje triviální variantou. Některé cíle budeme také pod rozdělovat (např. přidání cílů pro sbírání ingrediencí), cíle nemusíme vždy nutně pod rozdělovat v jedné postavě, víme totiž, že NPC postavy budou pracovat ve skupinách, můžeme tedy dát jedné postavě cíl sbírat základní ingredience a jiné vyrábět složitější předměty, pokud nebudou k dispozici ingredience tak je možné, že postava vyrábějící složitější předměty nenajde plán a bude se věnovat jiným cílům, což nemusí být nutně špatně pro naši hru, pokud bychom chtěli, aby se postavy chovali dle jejich role (např. kovář nepůjde dolovat železo, i když železo potřebuje, raději počká až mu ho vykutá horník). Jaké cíle přesně takto pod rozdělíme se budeme už muset rozhodnout u každého cíle individuálně, a toto rozhodnutí bude také záviset na počtu a roli ostatních NPC, se kterými NPC s daným cílem bude spolupracovat.

Nakonec se musíme rozhodnou, zda použít dopředné nebo zpětné plánování. Zpětné je očividně lepší, ale my jsme se během implementace rozhodli implementovat dopředné prohledávání, protože se nám zdalo jednodušší na implementovat a domnívali jsme se, že naše hra nedosáhne dostatečných rozměrů, aby byla se projevila neefektivita tohoto algoritmu. Naše domněnka byla naštěstí správná, a program dokáže běžet plynule, ale když se podíváme do Unity analyzátoru použitých zdrojů za běhu, tak vidíme, že naše AI využívá veliký počet času a kdybychom ztrojnásobily počet NPC (mapa aktuálně obsahuje přibližně 25 NPC) tak bychom zabírali příliš času, což by vyústilo v zasekávání hry.

#### 4.6.3. Splnění posledních požadavků

Takto se nám teda podařilo splnit požadavky 1,2,3 a 5, musíme teda ještě splnit požadavek 4. Tento požadavek může splnit použitím tagů, které komunikují jejich obecnou funkci. Např meč, sekera a kladivo mohou mít tag „zbraň na blízko“, voják pak může mít akci „zaútoč zbraní na blízko“, která bude operovat se všemi předměty označenými tímto tagem. Tagy budeme moct použít i pro objekty, například truhla a obchod by mohli mít tag „skladová jednotka“, v praxi jsme ale pro objekty tagy nepoužívaly, protože truhly a obchody byli jediné dva objekty, které se chovali identicky, takže jsme nepovažovali za nutné přidávat systém tagů pro objekty jenom kvůli tomuto případu.

## 4.7. Split Screen

Požadavek pro split screen je jasně daný (viz Obrazovka pro více hráčů 3.13):

- 1) Musí být schopný pod rozdělit obrazovku do čtyřúhelníkových částí půlením obrazovky, nebo jednotlivých částí.

Takovýto systém bychom mohli implementovať sami, ale Unity **Input System** už má takovéto pod rozdělování implementované, takže stačí využít tuto implementaci. Abychom tento systém mohli využít, tak stačí přidat do prefabu hráče kamery a zapnout pod rozdělování obrazovky v komponente **Player Input Manager**. Takto vždy, když zaregistrujeme nového hráče, systém automaticky pod rozdělí obrazovku a přidělí do nové sekce nově vytvořenou kamery hráče.

## 4.8. Herní svět

Na herní svět máme tyto požadavky:

- 1) Chceme, aby v herním světě existovaly suroviny (stromy, ložiska kamene atd.),
- 2) chceme v herním světě reprezentovat budovy a seskupení budov,
- 3) chceme herní plochu ohraničit, aby ji hráč nemohl opustit a neztratil se někde v nekonečnu,
- 4) chceme, aby byla herní plocha texturována, v přírodě, aby byla tráva, ve městě cesty nebo pěšinky a uvnitř budov podlahy.

### 4.8.1. Suroviny v herním světe

Suroviny jsou zničitelné **GameObjects**, stačí tedy vytvořit prefab, který do herního světa okopírujeme kolik chceme. Surovin ale může být mnoho a vytvoření nebo odstranění velikého počtu surovin (např. celého lesu) může stát hodně času, můžeme tedy využít Unity grid, která nám umožní přidávat nebo odstraňovat instance prefabů z herní mapy efektivněji, protože nebude muset přetahovat prefab do herního světa, ale postačí si prefab jednou vybrat a pouze naklikat kde suroviny chceme mít.

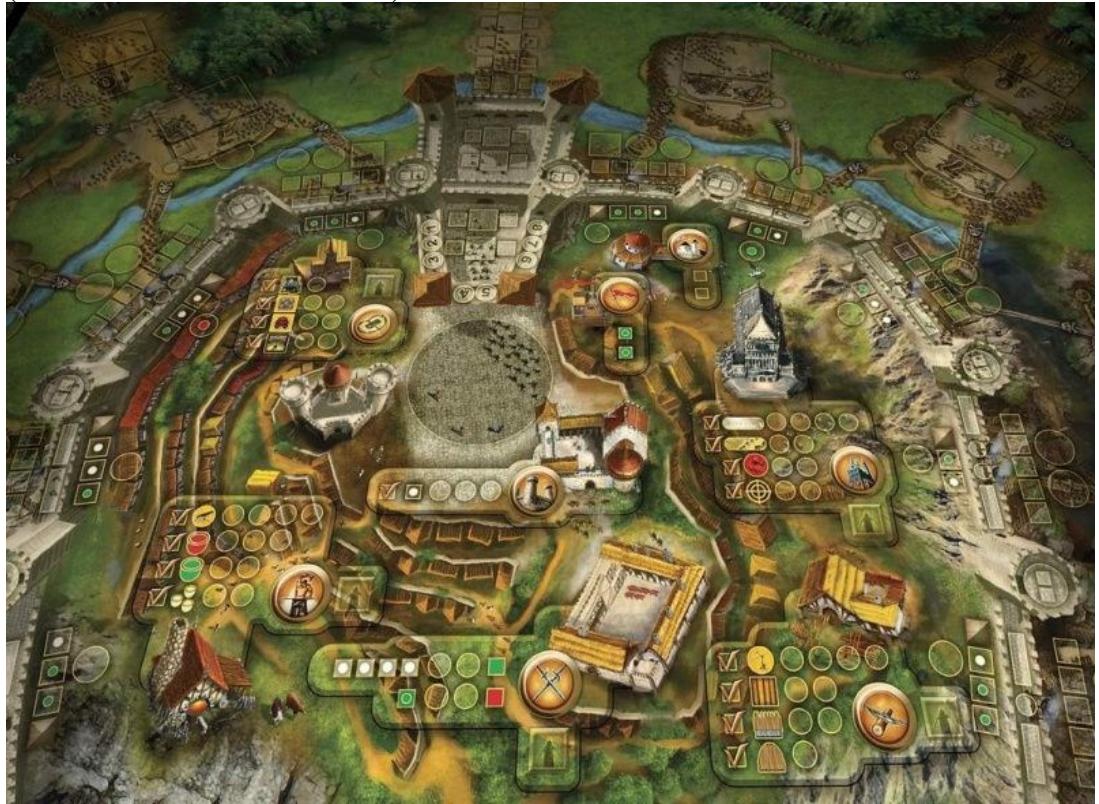
### 4.8.2. Textura herní plochy

Texturu herní plochy můžeme vytvořit několika způsoby, nejčastější je vybudování herní plochy z jednotlivých jednodušších textur. Tento přístup má 2 varianty:

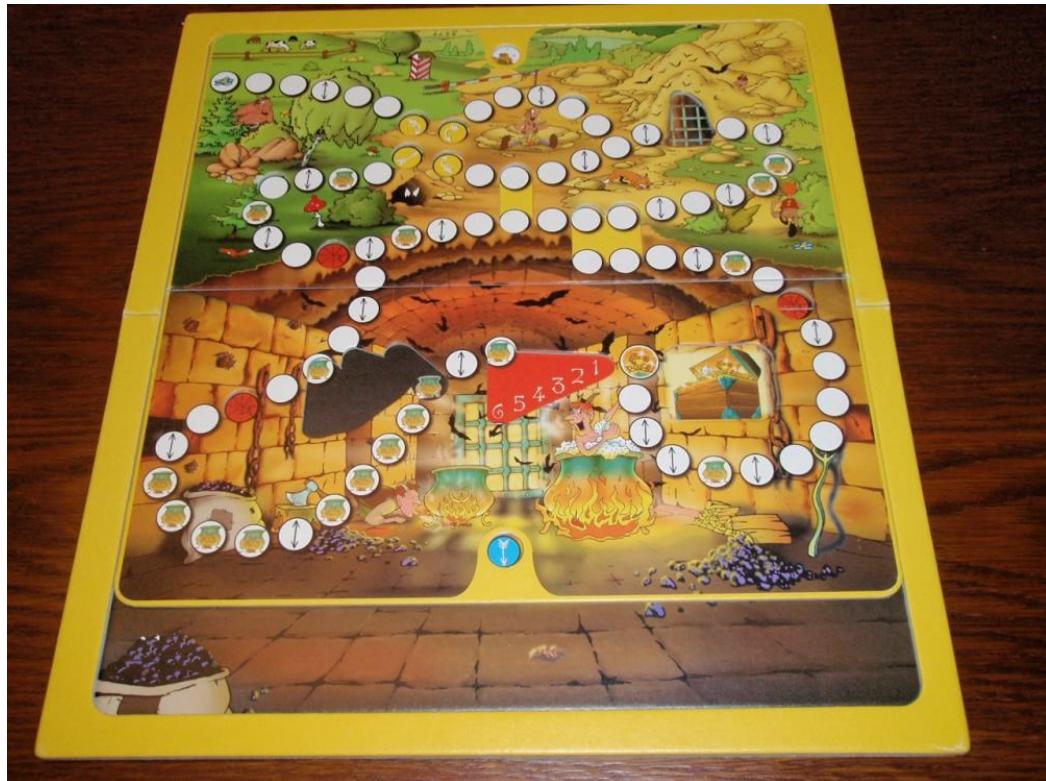
- 1) Herní bude rozdělen do čtvercové mřížky a každá jednodušší textura bude mít takovou velikost, aby zaplnila právě jeden čtverec. Texturu herní plochy pak vytvoříme vkládáním (na toto vkládání existují v Unity nástroje, a můžeme textury do mřížky malovat štětcí) textur do její mřížky. Jednotlivé textury jsou často vytvořené s tím, že se mohou poskládat do větších celků (např. studna může být poskládána ze 4 textur, každá zobrazující jiný roh studny). Tato varianta je často používána v pixel artových hrách, kvůli tomu že takové hry často chtějí, aby byly jejich textury přesně zarovnané.
- 2) Jednoduší textura může mít jakýkoliv tvar a velikost a můžeme je vložit kamkoliv do herní plochy. Textury většinou představují nějaký povrch, nebo celý objekt. Můžeme tedy snadněji, rychleji budovat celé scény (protože se každý celý objekt skládá pouze z jedné textury) a větší svoboda v umístnění textury nám umožnuje být kreativnější a tvorit specifickější scény než ve variantě 1. Tento přístup je častě používán ve hrách, které nepoužívají pixel art, kvůli této výhodám.

Dále existuje málo používaný způsob, a to vytvoření textury herní plochy jako jeden celek. Například nakreslením herní plochy v jiném programu. Tento přístup není vůbec flexibilní, protože, když změníme pozici nějakého prvku v

popředí, tak budeme muset překreslit herní plochu v úplně jiném programu, což zabere veliké množství času. Výhoda je ale, že máme absolutní kontrolu v tom, jak herní plocha vypadá a můžeme do ní přidat nejvíc specifických detailů a nutnost překreslovat můžeme minimalizovat tak, že texturu nakreslíme až na konci vývoje, kde se už pravděpodobně nic měnit nebude. Tento přístup není často používán v počítačových hrách, ale občas se používá v deskových hrách (viz Obrázek 22 a Obrázek 23).



Obrázek 22  
Herní plocha deskové hry Stronghold Zdroj 12



Obrázek 23  
Herní plocha hry Čertův poklad **Zdroj** 13

#### Analýza způsobů texturování

Nyní se rozhodneme, který z těchto tří způsobů se pro naši hru nejlépe hodí. Kreslení celé herní plochy pro nás nehodí, chceme totiž, aby na obrazovce byl přiměřený počet detailů. Pro deskovou hru toto není problém, protože hráč vidí celou desku najednou a ta má velikost normálního obrazu. Naše herní plocha je ale mnohem větší, než co hráč vidí na počítačové obrazovce, velikost obrázku by teda byla porovnatelná malbou velikosti stěny s tím, že hustota detailů by byla stejná jako v obraze normální velikosti, nakreslení takového obrazu zabere jednoduše příliš času, než aby to pro nás bylo praktické. Rozhodneme se tedy mezi variantami 1 a 2 pro skládání textur. Musíme se nejprve zeptat, zda používáme pixel art, nebo něco jiného, protože jsme si umělecký styl hry ještě nevybrali. Rozhodnutí je v našem případě snadné, textury a sprity nám totiž kreslí Lukáš Juraj Koprda (viz Přispěvatelé 2.8), který se specializuje na pixel art, a tak nám nakreslí vše v pixel artu. Protože se varianta 1 více hodí pro pixel art, tak si ji vybereme.

#### 4.8.3. Budovy

Chceme, abychom s budovami mohli operovat jako s celky během designu mapy, abychom nemusely přemísťovat každou zed' zvlášť, když se rozhodneme budovu přesunout. Proto si uděláme GameObject, který bude mít za děti NPC, objekty, stěny a podlahu patřící dané budově.

#### 4.8.4. Ohraničení herního světa

V game design dokumentu (viz Mapa světa 3.20) jsme nad tímto problémem už přemýšlely a bylo rozhodnuto, že svět ohraničíme oceánem a přidáme kolize, které brání zablokovat průchod do této textury.

## 4.9. Systém Frakcí

V game design dokumentu není nijak specifikováno, že existuje nějaký systém frakcí, ale pro fungování hry je ale vidět, že takový systém potřebujeme. Explicitně je zadána existence 2 týmů, do kterých jsou zařazeni všichni hráči a některé NPC, a že tyto dva týmy jsou vůči sobě nepřátelské. Je také popsána existence neutrálních NPC na zbytku mapy, vůči kterým hráčské týmy nejsou nepřátelské a existence NPC příšery, která je nepřátelská vůči všem ostatním postavám (viz Zřícenina 3.18). Také je popsáno, že si hráči budou moci vytvořit z kmene goblinů spojence (viz Kmen goblinů 3.17), čímž se stanou nepřáteli druhého týmu a umožní vydávání rozkazů vojákům.

Také bychom chtěli, aby pick-upy a objekty měli přidělené frakce. Objekty (např. truhly, posteły, kovadliny) jsou většinou v budova, které patří nějaké frakci a bylo by podivné a nežádoucí, aby NPC používali předměty, které jim nepatří. Nežádoucí, protože by NPC neměli cestovat přes celou mapu, aby si vzali předmět z truhly neutrálního NPC nebo přímo z nepřátelské základny, nebo aby používali nepřátelské kovadliny, takové chování nedává moc smysl z příběhového hlediska a z herního hlediska není dobré, protože by pravděpodobně byly zabity v nepřátelském území.

Stejná logika platí i pro pick-upy, pokud 2 nepřátelské dřevorubce na opačných stranách mapy a jeden pokácí strom první, což vytvoří vícero pickupů dřeva, tak nechceme, aby nám druhý dřevorubec šel přes celou mapu toto dřevo sebrat ze země, místo toho, aby pokácel strom blíže sobě. Ano, tento problém se dá vytvořit i přes AI, např. přidáním maximální vzdálenosti do které pick-upy můžeme uvažovat (toto jsme také použili v naší implementaci), ale je lepší mít robustnější systém, který přidává toto omezení navíc což platí i v menších vzdálenostech.

Mohli bychom si říct, že bychom někdy chtěli, aby naše NPC kradli předměty od nepřátele, takové chování by dávalo strategický smysl. Takovéto chování bychom ale raději specificky naprogramovali jako nějakou akci, než aby se AI samo rozhodovalo něco si vzít samo od sebe. A tento systém frakcí nám umožní toto chování snáze naprogramovat, protože budeme vědět které předměty patří našemu nepříteli, takže víme, co bychom mohli ukrást.

Chceme tedy systém, který:

- 1) přidělí frakci každé postavě,
- 2) přidělí frakci předmětům a objektům patřící dané frakci,
- 3) pamatuje si vztahy mezi frakcemi a umožnuje jejich dynamickou změnu.

Takovýto systém je snadno implementovatelný, pro uložení vztahů můžeme vytvořit singleton s maticí všech frakcí, který obsahuje hodnoty enumu vztahů (neutrál, spojenec, nepřítel). Změna vztahu se provede modifikací matice. Abychom mohli zjistit k jaké frakci NPC, pick-up nebo objekt patří, můžeme vytvořit novou komponentu, která bude mít uložený index dané frakce. Tuto komponentu pak inicializujeme správnou hodnotou během inicializace během hry, nebo ji manuálně připíše designér při vytváření levelu.

# 5. Uživatelská Dokumentace

V této kapitole si vysvětlíme, jak se tato hra hraje pomocí popsaného průchodu hrou.

## 5.1. Přehled

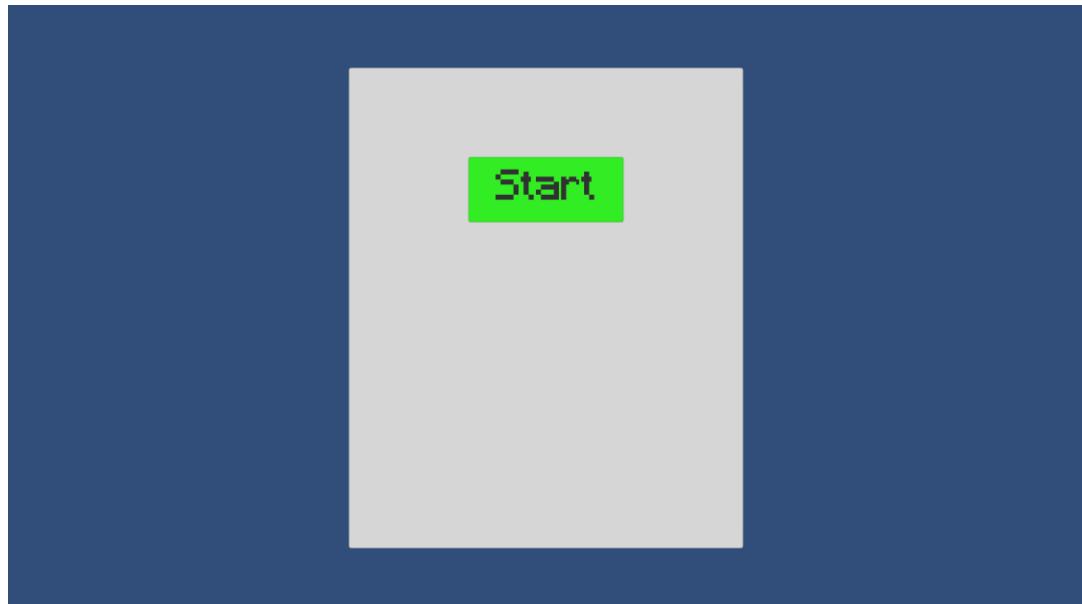
Snacks & Crowns je RPG hra pro více hráčů. Unikátní vlastnost této hry je, že předměty, které hráči a NPC používají jsou jimi vytvořeny z vytěžených surovin. Hráči jsou rozděleny do dvou týmů a jejich cílem je ukrást vítězný předmět nepřátelskému týmu a donést ho k podstavci svého týmu. Oba týmy mají NPC, které sbírají suroviny, vyrábějí předměty a stráží základnu jejich týmu. Na mapě také existují neutrální NPC, se kterými mohou hráči obchodovat.

## 5.2. Spuštění hry

Pro započetí hry, nejprve zapněte program spuštěním programu *Snacks & Crowns.exe*, načež se vám zobrazí hlavní menu (viz Obrázek 24).

### 5.2.1. Hlavní menu

V hlavním menu je pouze jedno tlačítko Start, které nás přesune do výběru postav. Toto tlačítko můžeme stisknout levým kliknutím myši nebo tlačítkem Enter na klávesnici.

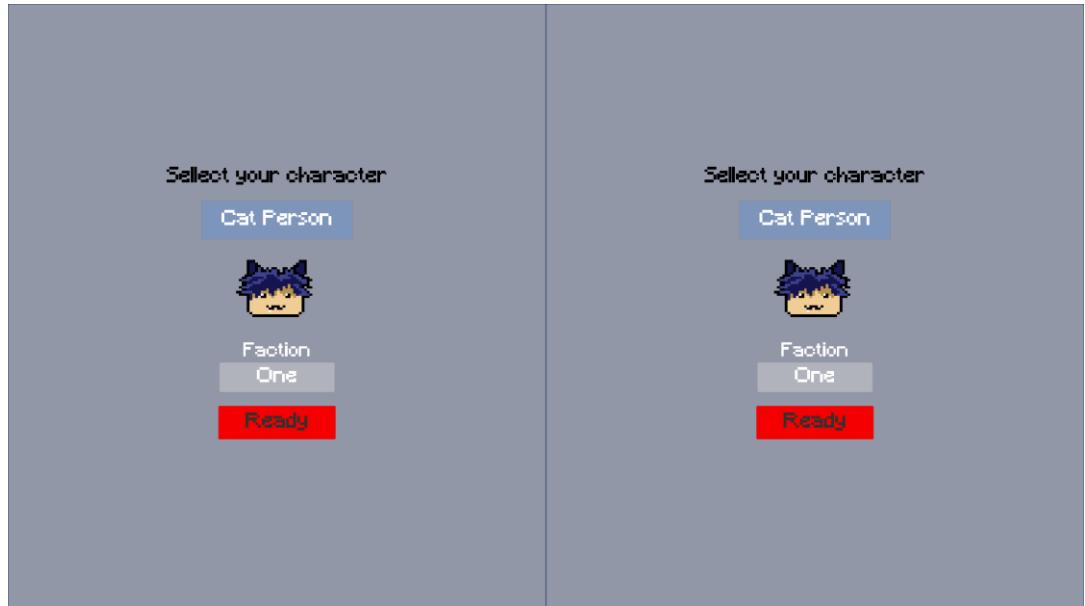


Obrázek 24  
Hlavní menu.

### 5.2.2. Výběr postav

Nyní se nacházíte ve výběru postav (viz Obrázek 25). Obrazovka je rozdělena do dvou částí, a každá část má své vlastní menu výběru postavy. Každá část odpovídá jednomu hráči, a protože by tuto hru měli hrát alespoň 2 hráči tak hned začínáme se dvěma hráči. Oba tito hráči používají klávesnici k ovládání hry, pokud chceme přidat další hráče tak stačí zapojit herní ovladač a stisknout na něm jakékoli tlačítka, což

vytvoří nové pod rozdělení obrazovky pro tohoto hráče. Během tohoto průchodu budeme předpokládat, že hrajete jako hráč 1 a budeme se soustředit pouze na jeho část obrazovky. Ovládání hráče 1 si průběžně vysvětlíme, ale pokud si je chcete v celku přečíst, nebo zjistit ovládání ostatních hráčů, tak je můžete najít v kapitole Vstupy hráčů 1.1.



Obrázek 25

Obrazovka výběru postav.

V menu Výběru postavy si vybereme rasu a tvář naší postavy a k jakému týmu patří. Navigujeme tlačítky **[W]** nahoru a **[S]** dolů, právě vybraná možnost je zbarvena modro-šedě (viz modrý obdélník Obrázek 26). Stlačením tlačítka **[H]** změníme hodnotu pole na další hodnotu.

První pole (červený obdélník) je rasa naší postavy, rasa určuje některé vlastnosti naší postavy, rychlosť, počet životů a odolnosť vůči některým druhům poškození. Člověk nemá žádné speciální vlastnosti ani slabiny, takže si ji vybereme pro tento průchod hrou.

Druhé pole je tvář postavy (zelený obdélník), toto je pouze estetické rozhodnutí a aktuálně mají všechny rasy pouze jednu tvář, kromě člověka, který má dvě.

Třetí pole (modrý obdélník) určí na straně kterého týmu budeme hrát, na výběr je pouze tým One a tým Two. Týmy mají rozdílné geografické pozice a mají jiné výherní předměty, jinak mezi jimi nejsou žádné rozdíly ze stran herních mechanik.

Až budeme spokojeni se našimi výběry, tak stačí přejít na poslední tlačítko Ready (žlutý obdélník) a stisknout **[H]**, čímž potvrďme svůj výběr, až všichni hráči potvrdí svůj výběr tak se načte hra.



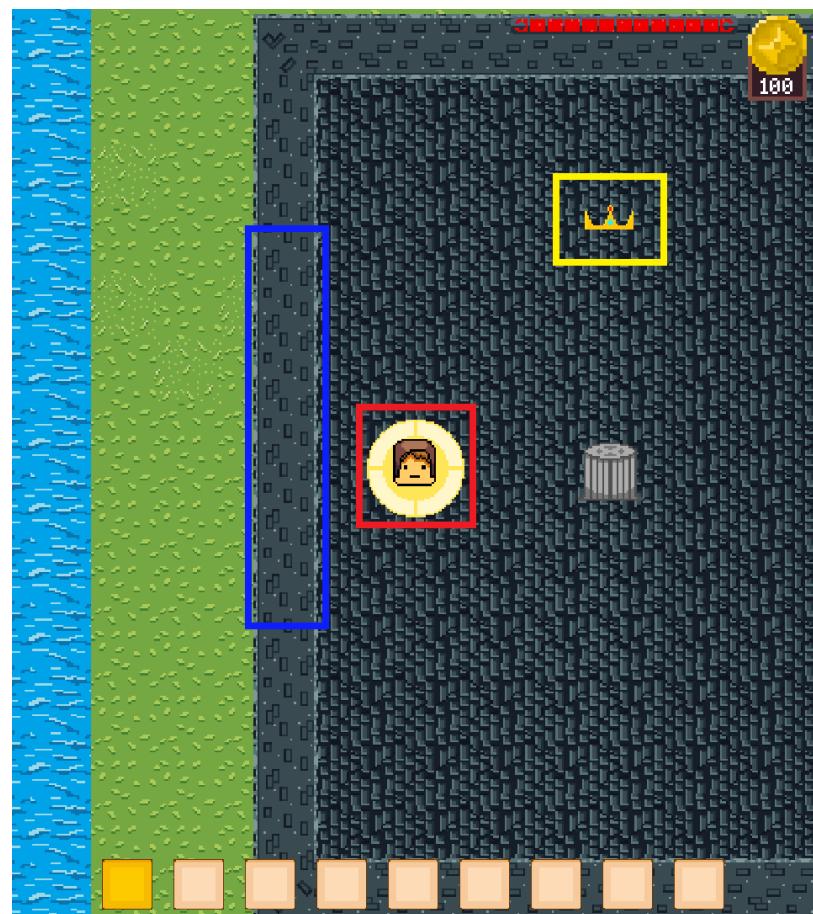
Obrázek 26  
Menu výběru postavy.

## 5.3. Hraní hry

Hlavní hra má obrazovku rozdělenu na tolik částí, kolik je hráčů, takže se pro účely tohoto průvodu hrou budeme soustředit pouze na část obrazovky prvního hráče.

### 5.3.1. Počátek hry

Hned jak hra začne, tak uvidíme naši postavu (viz červený obdélník Obrázek 27), která stojí na vyvolávacím kruhu, kde se vždy znova objeví, pokud umře. Právě se nacházíme v kostelu našeho týmu, zkusme se v něm projít tlačítky [WASD], některé objekty nám mohou bránit v průchodu, například stěny (viz modrý obdélník Obrázek 27). Mohli jsme si všimnout zlaté koruny (viz žlutý obdélník Obrázek 27), toto je vítězný předmět našeho týmu, naším cílem je zabránit nepřátelskému týmu v jeho ukradení, zkusme se ke koruně přiblížit.

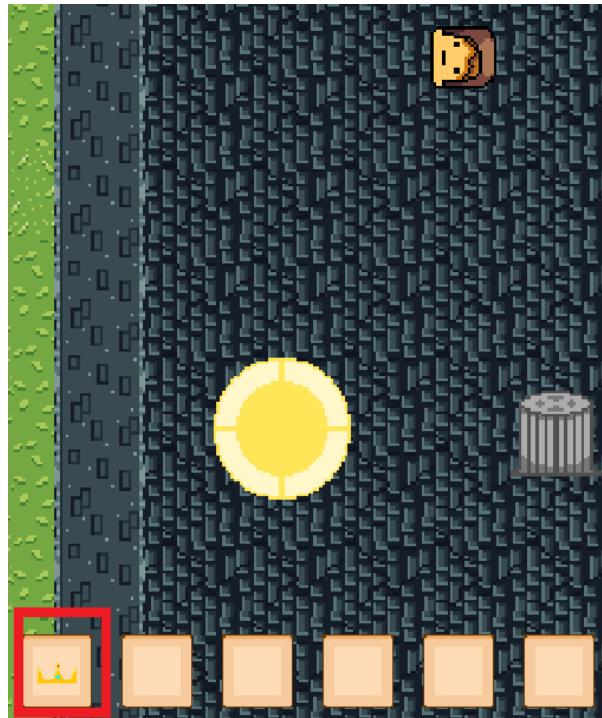


Obrázek 27

Když se ke koruně přiblížíme, tak se zabarví zeleně (viz Obrázek 28), toto znamená se s ní můžeme interagovat. K interakci slouží tlačítka [U], zkuste ho stisknout v blízkosti koruny. Hned jak s korunou za interagujeme, tak zmizí a objeví se v prvním poli na spodku obrazovky (viz Obrázek 29). Tato pole jsou náš hotbar a zobrazují předměty v našem inventáři.

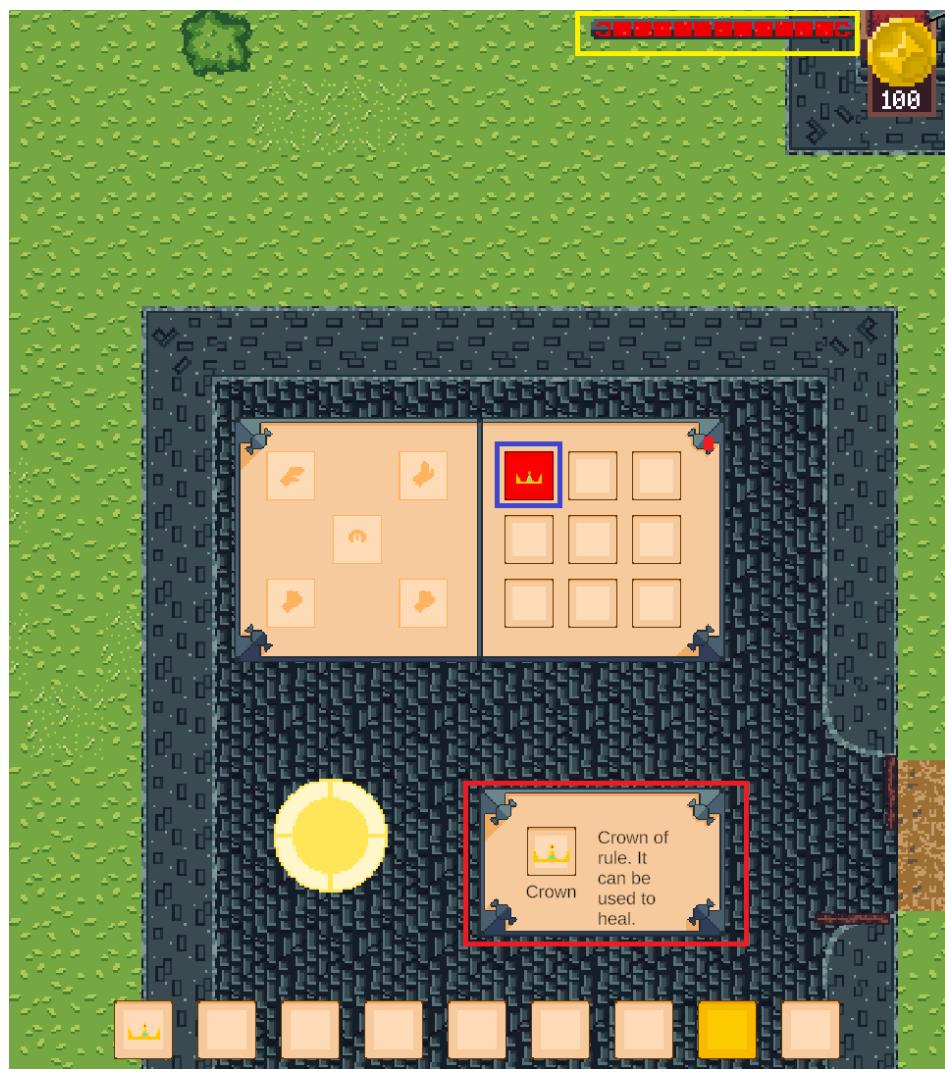


Obrázek 28



Obrázek 29

Abychom se o koruně něco dozvěděli zkusme otevřít náš inventář tlačítkem [I]. Zobrazí se nám menu se 3 částmi, inventářem, vybavením a popiskem (viz Obrázek 30). Aktuálně vybrané pole je obarveno červeně (viz modrý obdélník Obrázek 30) a pokud v něm je nějaký předmět tak se jeho informace zobrazí v popisku (viz červený obdélník Obrázek 30). Vybrané pole můžete změnit tlačítky [WASD], vyberme pole s korunou abychom se o ní mohli něco dozvědět. V popisku vidíme, že může být použita abychom se oživili. Naše životy vidíme nahoře obrazovky, červená náplň oznamuje, kolik životů z maxima právě máme (viz žlutý obdélník Obrázek 30). Aktuálně máme plno životů, takže korunu použít nemusíme.



Obrázek 30

Inventář zavřeme tlačítkem [I]. Abychom mohli vyhrát budeme potřebovat bojovat a k tomu potřebujeme sehnat nějakou zbraň, zbraně mohou být koupeny ve vedlejší budově. Zkusme tedy vyjít z kostela. Abychom z něj mohli vyjít musíme otevřít dveře, což uděláme tím, že s nimi provedeme interakci, stejným způsobem jako s korunou, stačí se k nim přiblížit a stisknout tlačítko U (viz Obrázek 31).



Obrázek 31

Hned vpravo od kostela je tržiště, v této budově můžeme kupovat a prodávat předměty. Tržiště má obchody (viz modré obdélníky Obrázek 32), které slouží

k nákupu a prodeji předmětů, posteče (žluté obdélníky), kde mohou postavy spát, aby si obnovily životy (NPC postavy jdou periodicky spát), vyvolávací kruhy (zelené obdélníky), kde se po smrti NPC za několik minut objeví a vyráběcí objekty, které si vysvětlíme později. Obchod nahore vpravo prodává zbraně na blízko, přiblížme se k němu a interagujme s ním pomocí [U].



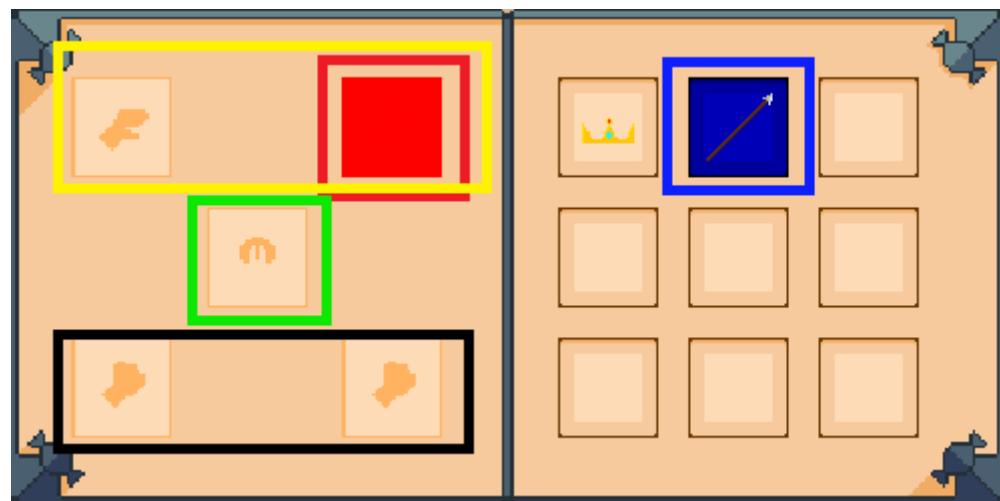
Obrázek 32

Otevře se nám menu, vypadá podobně jako menu inventáře, ale místo sekce pro vybavení se nám zobrazí inventář obchodu (viz zelený obdélník Obrázek 33). Každý předmět má také pod sebou zobrazenou cenu, za kterou ho můžeme kupit (pokud je v inventáři obchodu) nebo prodat (pokud je v našem inventáři). Abychom předmět koupili nebo prodali, tak ho stačí vybrat tlačítka [WASD] a stisknout tlačítko [H]. Abychom mohli kupit předmět, tak potřebujeme mít dost zlata, počet našeho zlata vidíme pod ikonou zlaté mince (viz modrý obdélník Obrázek 33). Pokud dost zlata nemáme tak se nic nestane, jinak se odečte správný počet zlata a mi dostaneme koupený předmět. Pokud předmět prodáme, tak se nám přičte správný počet zlata. V obchodě máme na výběr meč nebo oštěp, poškození ze zásahu je u obou zbraní přibližně stejný, oštěp má delší dosah, ale meč pokryje švihem více prostoru. Dosah se nám možná hodí více, tak si koupíme oštěp.



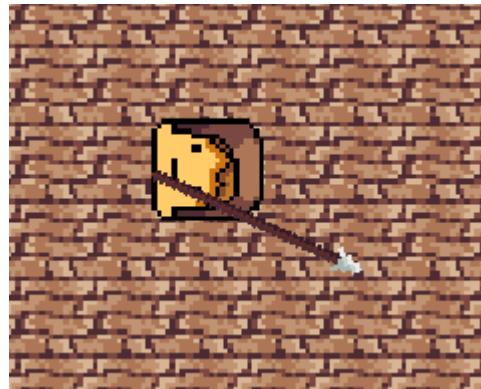
Obrázek 33

Zavřeme menu obchodu stlačením tlačítka [U], a otevřeme inventář [I] abychom oštěp mohli využít jako vybavení. Inventář jsme už viděli, teď se podíváme na levou část, zobrazující naše vybavení (viz Obrázek 34). Vybavení má 5 polí, ve žlutém obdélníku jsou pole pro vybavení do pravé a levé ruky, kam se umístují zbraně, v zeleném obdélníku je pole pro brnění, sem patří helmy a podobně, v černém obdélníku jsou místa pro ostatní předměty, ale v tomto demu žádné předměty, které by sem patřily neexistují. Abychom využili oštěp tak ho musíme přesunout do pole ruky, abychom ho přesunuli, tak se na pole oštěpu přesuneme naším výběrem a zmáčkneme [H], takto se pole označí a zabarví se modře (viz modrý obdélník Obrázek 34), poté přesuneme výběr do pole pravé ruky (viz červený obdélník Obrázek 34). Nyní stačí zmáčknout [H], což přesune předmět z označeného pole do vybraného pole. Nyní využíváte oštěp v pravé ruce.



Obrázek 34

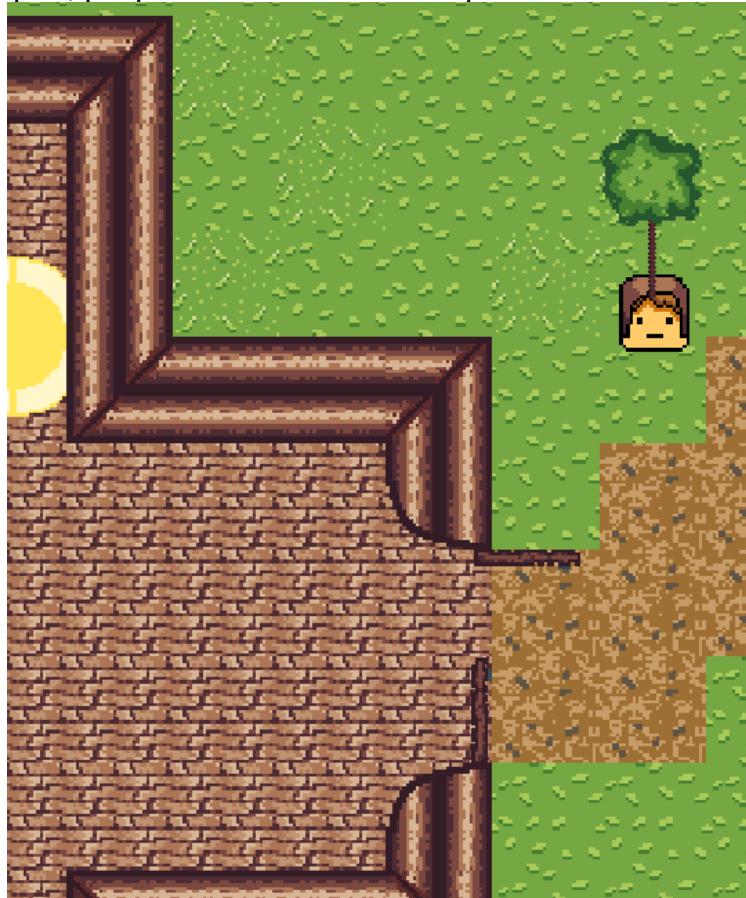
Když zavřeme inventář [I] tak vidíme, že naše postava nyní drží oštěp (viz Obrázek 35). Abychom s oštěpem zaútočili tak stačí zmáčknout tlačítko [J], které aktivuje vybavení v pravé ruce. Když zkusíme zaútočit, tak uvidíme, že se přehraje animace, během této animace může oštěp udělovat poškození zasaženým postavám nebo některým objektům. Pokud bychom měli zbraň v levé ruce, tak bychom zmáčkly tlačítko [H] pro její aktivaci.



Obrázek 35

### 5.3.2. Výroba předmětů

Chtěli bychom také nějaké brnění, ale k tomu nemáme dost peněz po koupi oštěpu, zkusme si tedy vyrobit brnění sami. Nejprve musíme získat suroviny. Budeme potřebovat dřevo a železnou rudu. Abychom získali dřevo musíme pokácet strom, hned vedle pravého východu od tržiště je strom (viz Obrázek 36), zkusme na něj zaútočit oštěpem, pro pokácení stromu budeme potřebovat asi 10 útoků.



Obrázek 36

Jakmile strom pokácíme tak z něj vypadnou 3 dřeva (viz Obrázek 37), tyto můžeme vzít do inventáře stejně jako ostatní předměty [U]. Nyní potřebujeme železnou rudu, tuto můžeme vytěžit pod tržištěm na jeho levé straně. Tam se nachází ložiska železné rudy a kouzelných krystalů (viz Obrázek 38), kouzelné krystaly nepotřebujeme, takže je budeme ignorovat a vytěžíme jedno ložisko železné rudy. Proces probíhá stejným způsobem jako těžení stromu, stačí asi 10krát zaútočit a dostaneme železnou rudu, kterou zase sebereme do inventáře [U].



Obrázek 37



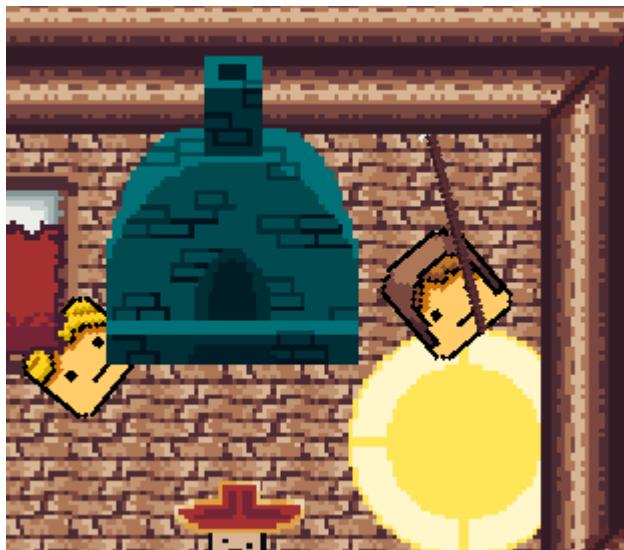
Obrázek 38

Ložisko kouzelných krystalů v modrém obdélníku.

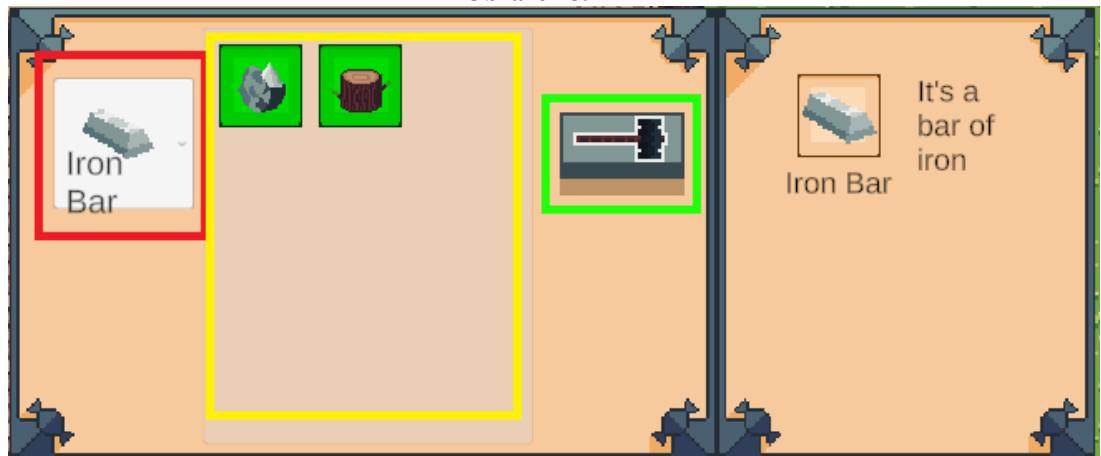
Železná ruda v červeném obdélníku.

Z železné rudy a dřeva můžeme vyrobit železo, vyrábění předmětů se dělá pomocí vyráběcích objektů. Vraťme se do tržiště a přiblížme se k výhni v pravém horním rohu (viz Obrázek 39). Výheň je jeden z vyráběcích objektů, abychom vyrobili předmět musíme s výhni interagovat [U], načež se nám otevře výrobní menu (viz Obrázek 40). Menu má dvě tlačítka, tlačítko na výběr předmětu (červený obdélník), kde můžeme vybrat jaký předmět chceme vyrobit, výheň má pouze jeden

předmět, který může vyrobit, a ten je už automaticky vybrán. Předměty potřebné k výrobě vybraného předmětu se zobrazí ve žlutém obdélníku a pokud dané předměty máme, tak se zabarví zeleně. V zeleném obdélníku je tlačítko na potvrzení výroby předmětu, když totiž toto tlačítko stlačíme tak se odstraní potřebné předměty z našeho inventáře a dostaneme předmět, který jsme vybrali k výrobě. Navigace probíhá stejně jako v jiných menu, tlačítka [WASD] měníme vybraná tlačítka a pomocí [H] ho stiskneme. Už máme vybrané železo k výrobě a máme potřebné ingredience, zmáčkněme tedy tlačítko na výrobu 3krát, abychom vyrobili 3 železa.



Obrázek 39



Obrázek 40

Zbyly nám 2 kusy železné rudy, které už nepotřebujeme, takže je nyní odložíme. Vedle ložiska železné rudy je dělnická chalupa s truhlou (viz Obrázek 41), zkusme interagovat s touto truhlou [U]. Otevře se nám menu podobné inventáři, ale levá polovina je nahrazena inventářem truhly (viz Obrázek 42). Předměty můžeme přesouvat mezi pole inventáře stejným způsobem jako jsme přesunuly oštěp do pole vybavení. Označíme pole [H] s železnou rudou a přesuneme ho do pole truhly [H]. NPC mohou používat předměty v truhlách i předměty na zemi, skladovat předměty v truhlách je tudíž hlavně dobré pro hráče, aby měli přehled, kde jsou uskladněny předměty.

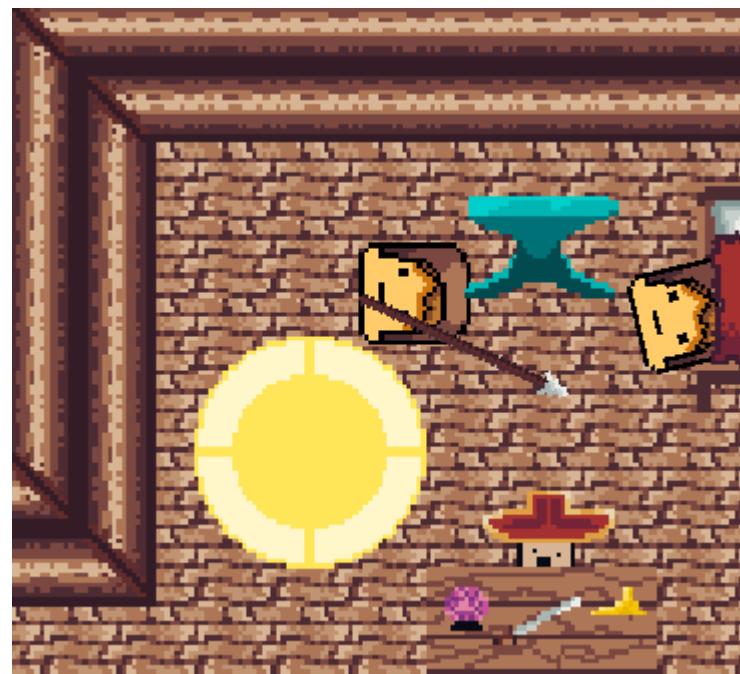


Obrázek 41

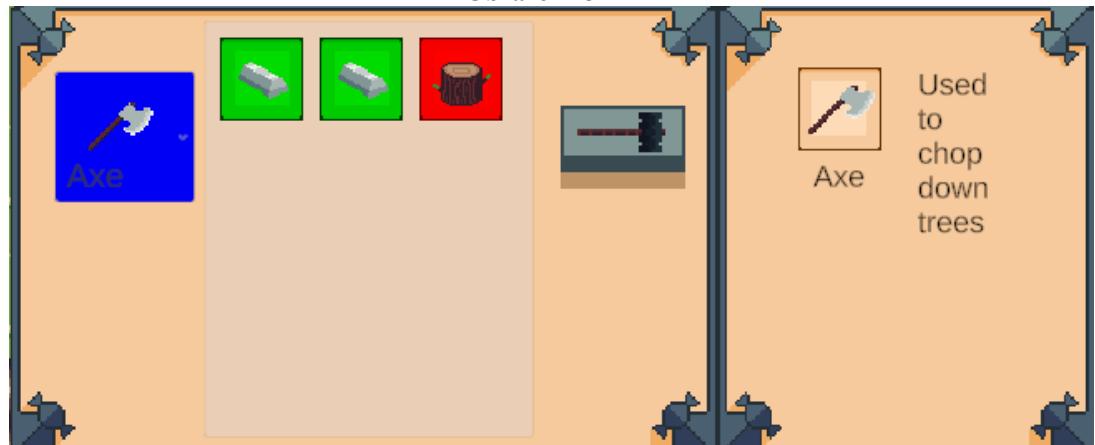


Obrázek 42

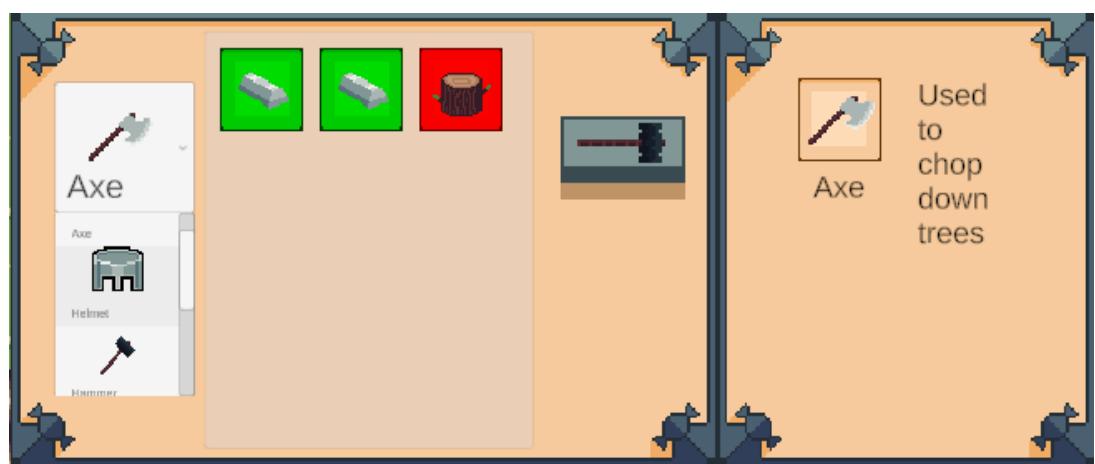
Vrátíme se do tržiště a za interagujeme [U] s kovadlinou, která je další výrobní objekt (viz Obrázek 43). Kovadlina má na výběr více předmětů, takže budeme nejprve muset vybrat správný předmět k výrobě, vybereme tlačítko s právě vybraným předmětem (sekerou, viz Obrázek 44) a stiskneme [H]. Otevře se nám seznam vyrobitelných předmětů (viz Obrázek 45). Tlačítka [W] a [S] vybereme helmu a potvrďme výběr tlačítkem [H] (viz Obrázek 46). Poté stlačíme tlačítko na potvrzení výroby a naše helma bude vyrobena.



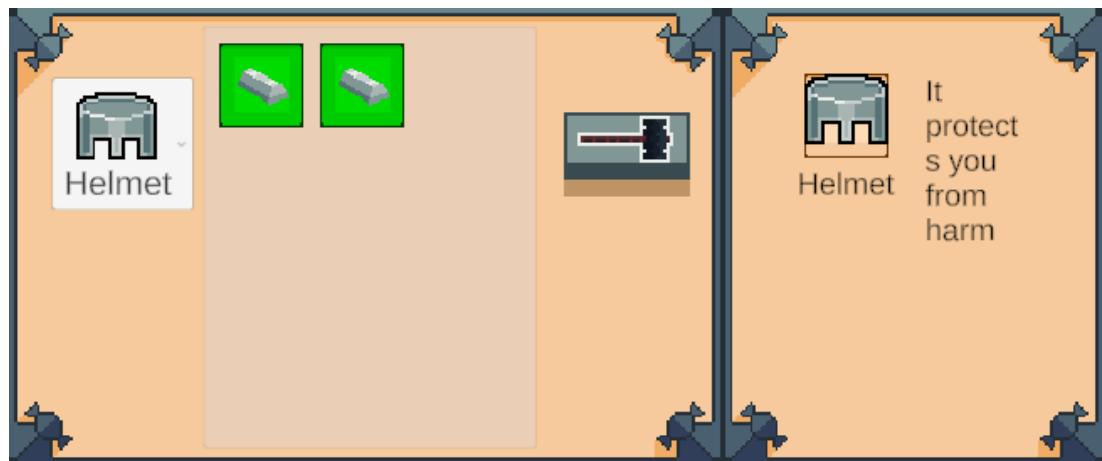
Obrázek 43



Obrázek 44



Obrázek 45



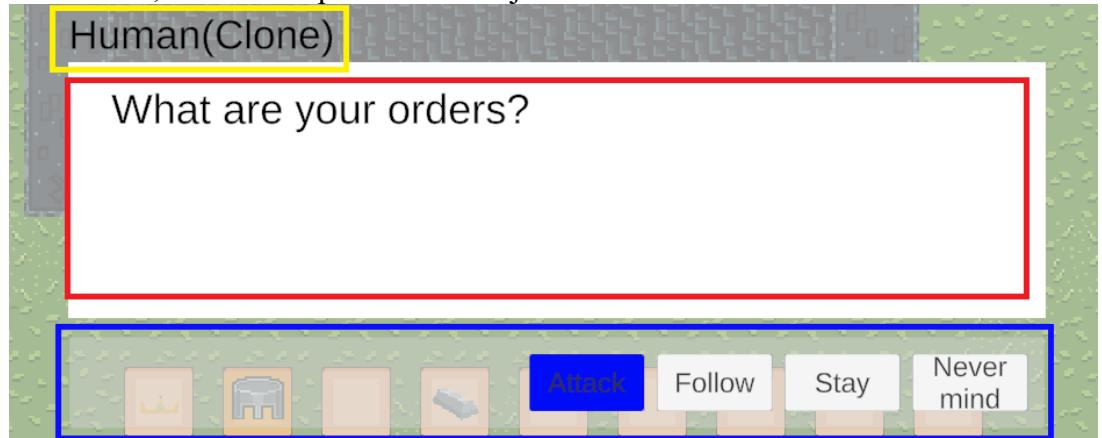
Obrázek 46

Helmu využijeme stejně jako oštěp, otevřeme inventář [I] a přesuneme helmu do pole brnění.

### 5.3.3. Dialog Rozkazy

S některými NPC můžeme započít dialog tlačítkem pro interakci [U], náš tým má NPC vojáky, kteří patrolují kolem naší základny, zkusme s některým z nich započít konverzaci tím, že se k jednomu přiblížíme, dokud se nezvýrazní zeleně a zmákneme [U]. Otevře se nám dialogové menu (viz Obrázek 47). Ve žlutém obdélníku je název postavy, se kterou mluvíme. V červeném obdélníku je text konverzace. V modrém obdélníku máme na výběr z několika možností, jak v konverzaci pokračovat. Možnost můžeme vybrat tlačítka [W] a [S] a potvrdit pomocí [H]. U všech vojáků máme na výběr z těchto 4 možností:

- 1) „Never mind“ konverzaci ukončí,
- 2) Stay přikáže vojákově, aby patroloval kolem tohoto místa,
- 3) Follow přikáže vojákově, aby vás následoval,
- 4) Attack přikáže vojákově, aby zaútočil na nepřátelský tým, útočit bude, dokud neumře, nebo mu nepřikážete něco jiného.



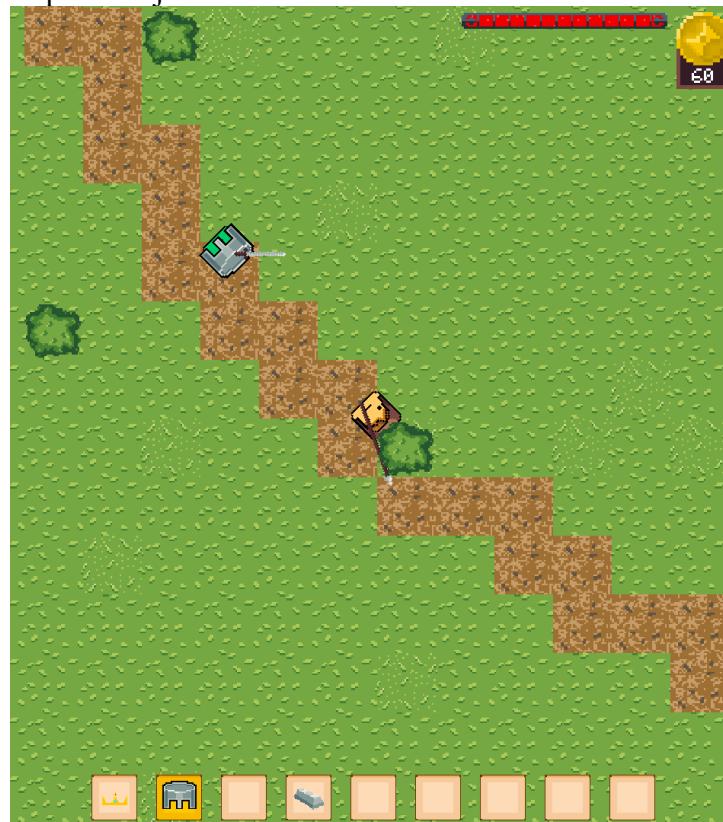
Obrázek 47

Chtěli bychom zaútočit na nepřátelskou základnu, takže mu přikážeme, aby nás následoval (Follow), abychom zaútočili spolu.

### 5.3.4. Neutrální NPC

Abychom de dostali k základně nepřátelského týmu stačí následovat vyšlapanou cestu (viz Obrázek 48). Za chvíliku narazíme na rozdvojení cesty (viz Obrázek 49), rovně se dostaneme k nepřátelské základně, ale odbočka může vést k neutrálním

NPC, které mohou prodávat předměty. Přikážeme našemu vojákoví, aby zde počkal (Stay) a vydáme se po vedlejší cestě.



Obrázek 48

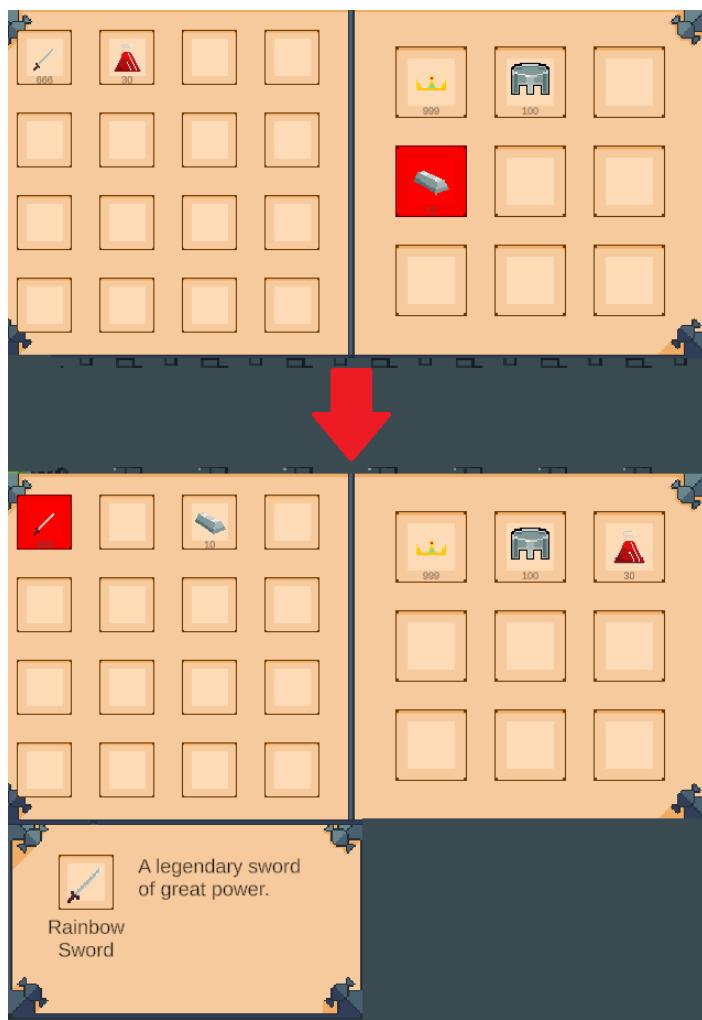


Obrázek 49

Za chvilku nalezneme kočko-lidskou vesnici s obchodem (viz Obrázek 50). Zdá se, že prodávají oživovací lektvary (viz Obrázek 51). Lektvar koupíme a prodáme i naše železo, protože ho stejně už nepotřebujeme. Obchod také prodává unikátní meč, který se nedá vyrobit, bohužel ale nemáme dost peněz, takže ho tu musíme nechat.



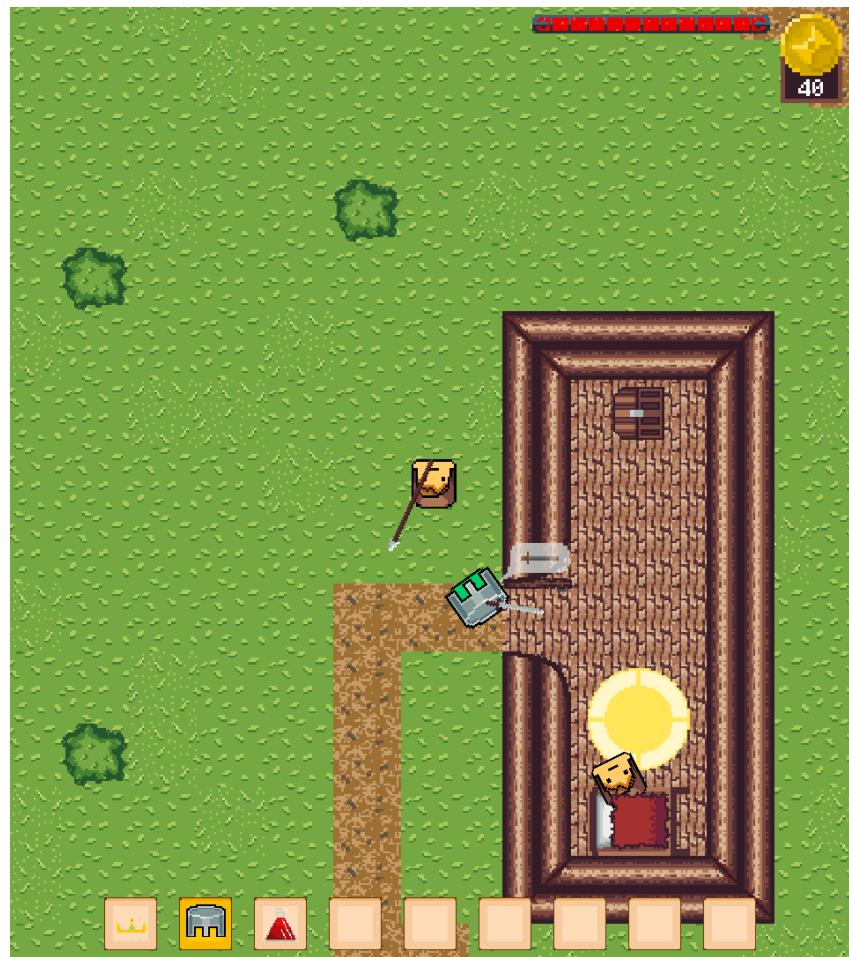
Obrázek 50



Obrázek 51

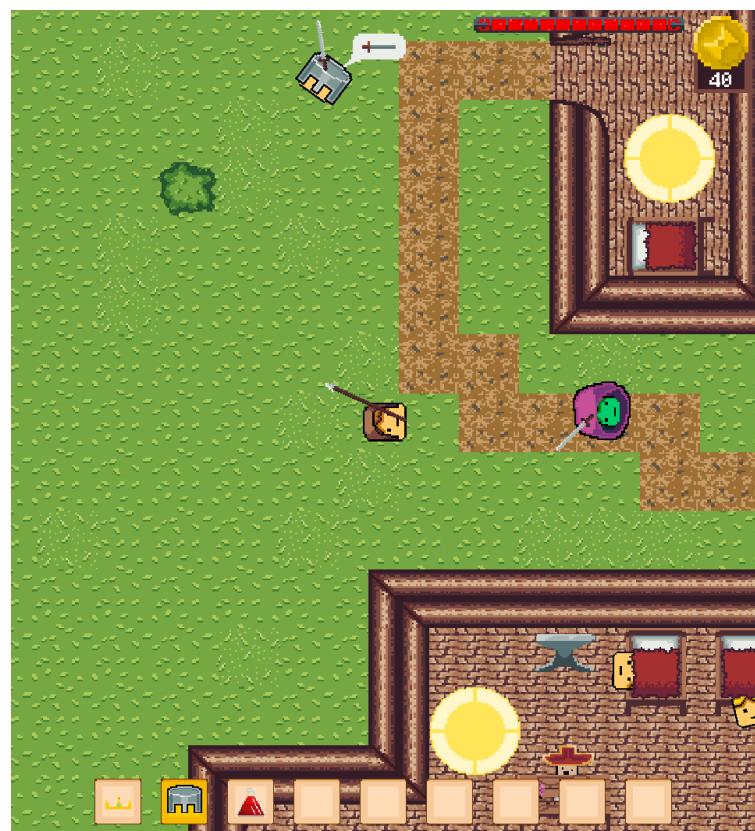
### 5.3.5. Útok na nepřátelskou základnu

Vráťme se po cestě k rozdvojce a přikážeme vojákovi, aby nás zase následoval (Follow). Nalezneme dřevorubeckou chatu nepřátelské základny, náš voják začne sám útočit na nepřátele v blízkosti, a právě útočí na dřevorubce (viz Obrázek 52).

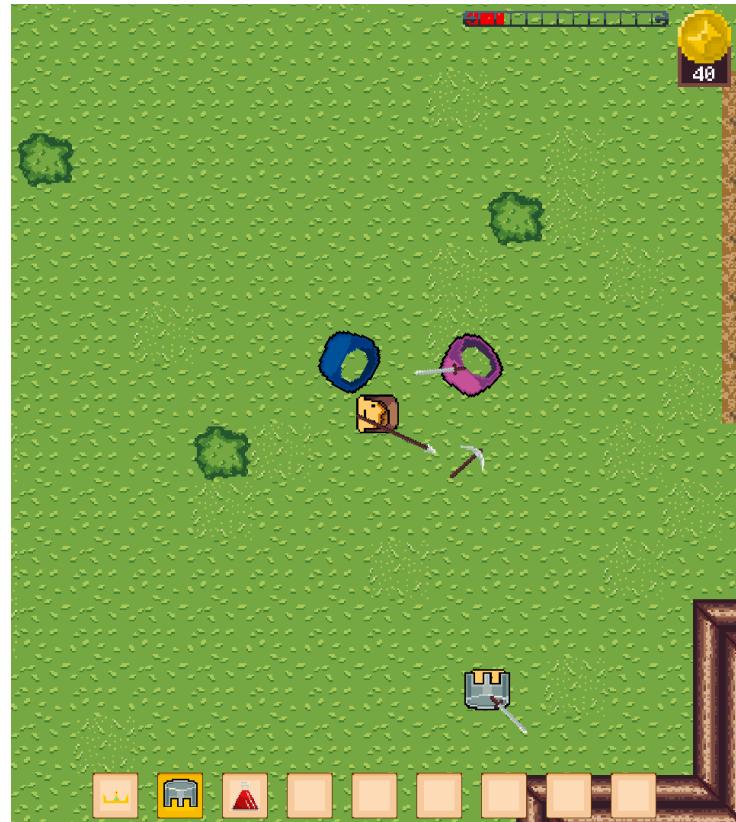


Obrázek 52

Za chvíliku nás objeví patrolující stráže a vypukne boj (viz Obrázek 53), nepřátelé nás budou sledovat a útočit na nás zbraněmi, dokud neutečeme, nebo jeden z nás neumře. Použijeme náš oštěp [J], abychom je zabili. Když postavy umřou tak z nich vypadnou jejich předměty a vybavení, které můžeme vzít a použít, pokud bychom chtěli (viz Obrázek 54).



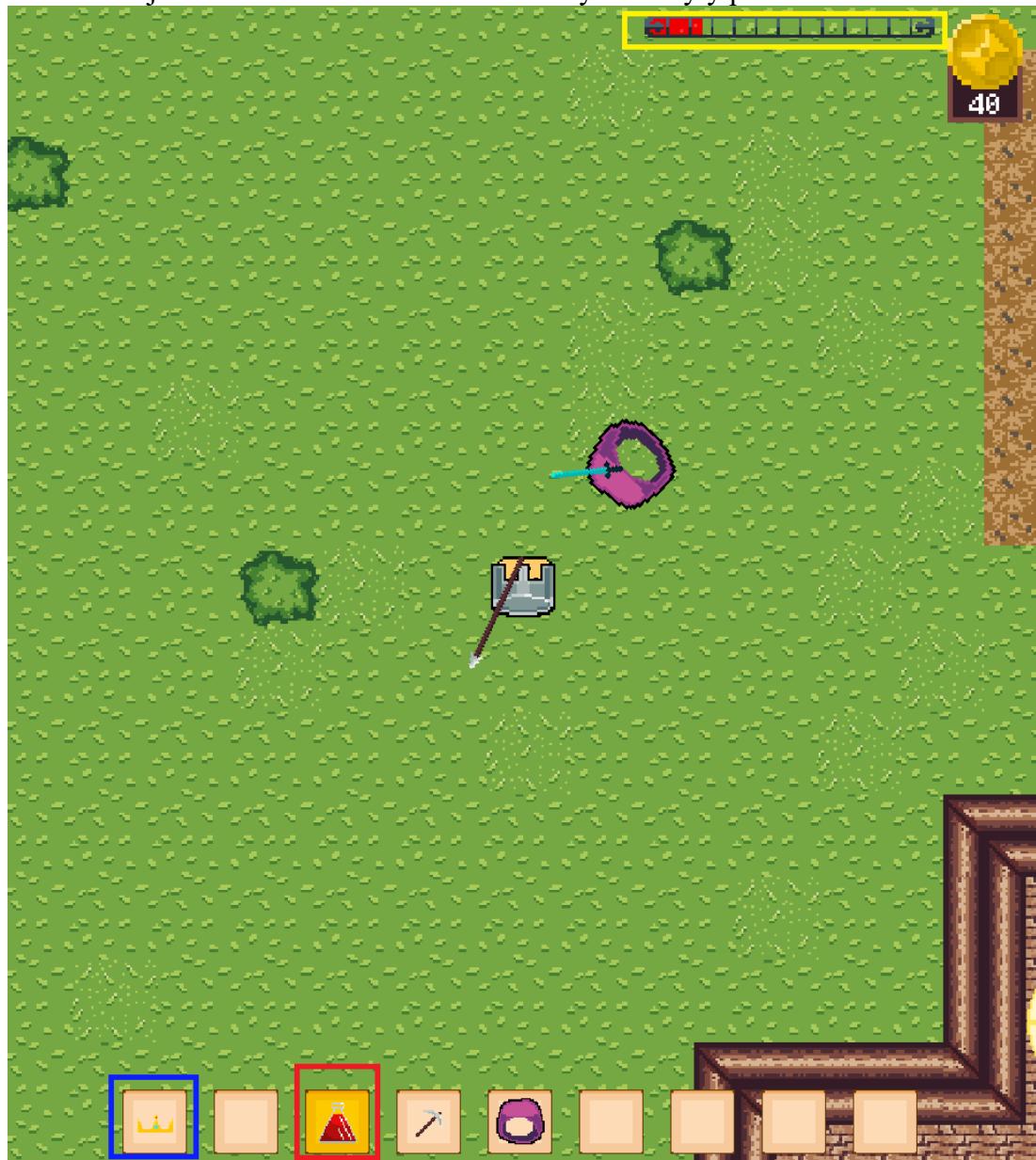
Obrázek 53



Obrázek 54

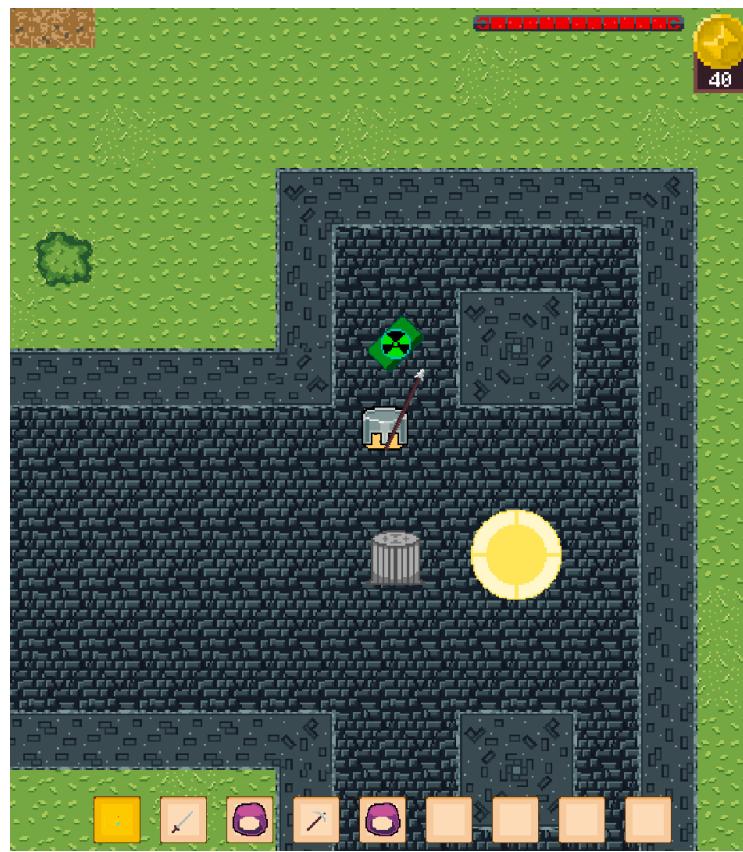
Během boje jsme byli zraněni a jsme nyní skoro mrtvi (viz žlutý obdélník Obrázek 55), k oživení můžeme použít oživovací lektvar, abychom použili předmět tak ho musíme nejprve vybrat (vybrané pole je zbarveno, viz červený obdélník)

v našem hotbaru tlačítky [Q] a [E] a poté stiskneme [K]. Léčivý lektvar je jednorázový předmět, takže zmizí, jakmile ho použijeme, koruna (viz modrý obdélník) nás také dokáže oživit, ale není jednorázové, takže zůstane v našem inventáři. Použijme léčivý lektvar a 2krát korunu abychom byly plně oživeni.

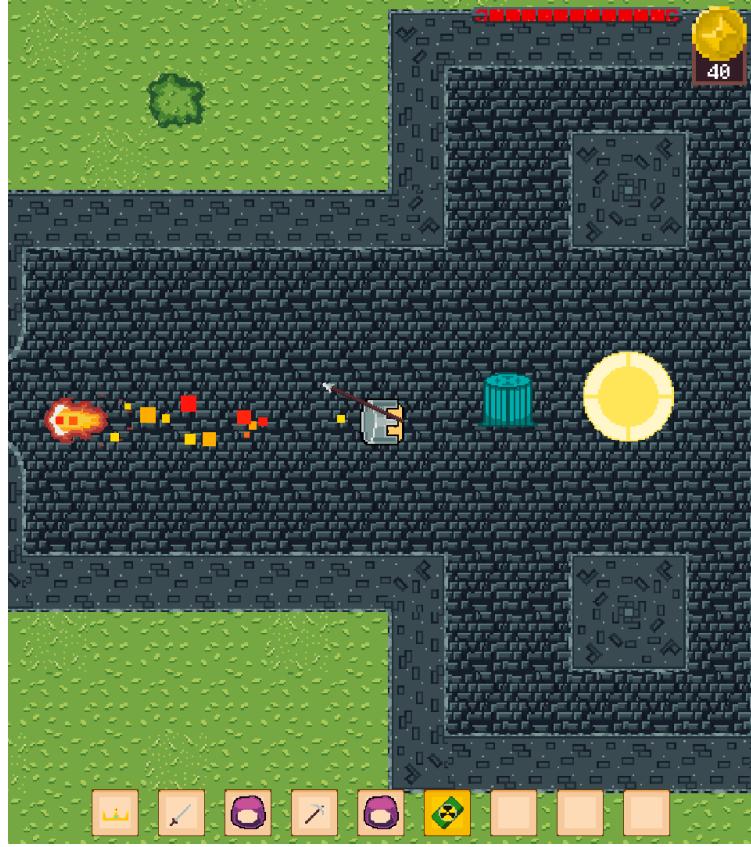


Obrázek 55

Většina stráží je nyní mrtvá, takže se můžeme dostat do nepřátelského kostela a vzít jejich vítězný předmět, atomovou tyčinku (viz Obrázek 56). Stejně jako koruna, má i atomová tyčinka efekt při použití, specificky vytvoří ohnivou kouli, který vybuchne, když narazí do překážky a zraní postavy v okolí exploze (viz Obrázek 57).



Obrázek 56



Obrázek 57

Nepřátelské stráže se ještě nestihli respawnout, takže můžeme snadno odejít z nepřátelské základny a vrátit se do základny našeho týmu, kde můžeme v kostelu najít podstavec kam máme odevzdat ukradený vítězný předmět. Stačí k němu přijít,

za interagovat [U] a potvrdit odevzdání předmětu, čímž vyhrajete hru (viz Obrázek 58).



Obrázek 58

Nyní si zkuste zahrát celou hru sami s hráči na obou týmech.

## 1.1 Vstupy hráčů

Hráči jsou vždy dva na klávesnici, a další se mohou připojit pomocí herních ovladačů, zde je vypsáno, která tlačítka odpovídají, jakým vstupům.

### 5.3.6. Hráč 1 - Klávesnice

- 1) Pohyb – WASD
- 2) Levá ruka/vybrat – H
- 3) Pravá ruka/odebrat/vyhodit předmět – J
- 4) Použít předmět – K
- 5) Otevřít/zavřít inventář – I
- 6) Interakce – U
- 7) Výběr předmětu v hotbaru – QE

### 5.3.7. Hráč 2 - Klávesnice

- 1) Pohyb – šipky
- 2) Levá ruka/vybrat – 1 (numpad)
- 3) Pravá ruka/odebrat/vyhodit předmět – 2 (numpad)
- 4) Použít předmět – 3 (numpad)
- 5) Otevřít/zavřít inventář – 0 (numpad)
- 6) Interakce – 6 (numpad)
- 7) Výběr předmětu v hotbaru – 45 (numpad)

### 5.3.8. Další hráči – Herní ovladač

- 1) Pohyb – levý joystick
- 2) Levá ruka/vybrat – X (xbox), čtverec (playstation)
- 3) Pravá ruka/odebrat/vyhodit předmět – B (xbox), kruh (playstation)
- 4) Použít předmět – A (xbox), křížek (playstation)
- 5) Otevřít/zavřít inventář – L2 (levá spoušť)

- 6) Interakce – Y (xbox), trojúhelník (playstation)
- 7) Výběr předmětu v hotbaru – pravá/levá šipka na d-padu

# 6. Vývojová dokumentace

V této vývojové dokumentaci Vás nejprve provedeme architekturou projektu v Unity a poté jednotlivě přes všechny herní systémy a jejich vztahy.

## 6.1. Architektura scény Unity

V projektu máme 4 scény:

- 1) StartMenu, která se otevře po zapnutí programu, dá se navigovat šípkami a myší.
- 2) CharacterSelector, která se načte po stlačení tlačítka Start, ve scéně StartMenu. V této scéně se vytvářejí instance hráčů a zde si každý hráč vybere z dostupných nastavení jeho postavy.
- 3) Level1, v této scéně se odehrává celá hra. Načte se, jakmile všichni hráči stisknou tlačítko Ready v scéně CharakterSelector.
- 4) TestLevel, tato scéna sloužila k testování systémů v prvních měsících vývoje, později se ale testovalo přímo ve scéně Level1. Do této scény se lze dostat pouze přes Unity editor.

Dále máme SceneTemplate zvaný LevelTemplate, který obsahuje požadované GameObjecty pro vytvoření hratelného světa.

### A\*

Obsahuje Pathfinder komponentu, ve které specifikujete velikost a rozlišení pathfinding systému.

### Grid

Obsahuje Unity komponentu Grid a dítě s komponentou Tilemap. Slouží k vytvoření textury herní plochy a umístování surovin.

### PlayerManager

PlayerManager obsahuje komponentu Player Input Manager (z Unity input system) a komponentu My Join Player From Data. Player Input Manager, automaticky rozděluje obrazovku a stará se o vstupy všech hráčů. Komponenta My Join Player From Data, zaregistrouje hráče do komponenty Player Input Manager a zajistí že se spawnou na spawnpointu jejich týmu, daným prázdnými GameObjecty RespawnPointP1 a RespawnPointP2. Informace o hráčích (jejich tým, rasa, tvář) jsme načetly ze statické třídy StartGameDataHolder, do které jsme data vložili v scéně CharacterSelector.

### PlayerManagerDebugSpawn

PlayerManagerDebugSpawn se chová stejně jako PlayerManager, ale má komponentu My Join Player, která má předem nastavená data pro 2 hráče. Tento GameObject slouží k rychlému debuggování, abychom mohli hru začít s postavami, a nemuseli si vždy vybírat postavy v CharacterSelector scéně. Postavy se vytvoří v souřadnicích (0,0). Tento game Objekt by měl být vypnut, když chceme hrát hru normálně.

### Organizační

GameObjekty Items, Projectiles, NPCs, Trees, Iron Ore Deposits a Crystal Deposits jsou pouze organizační, jednotlivé GameObjekty se pod ně sami zavěší jako děti, aby nám při debuggování nezahľtili hierarchii scény.

### Crafting Recipes

GameObjekt Crafting Recipes má komponentu **Crafting Recipes**, která si načte všechny recepty ze ScriptableObjectů druhu Item ve složce Items. Když pak jakýkoliv jiný GameObject potřebuje seznam receptů tak si ho načte z této komponenty.

#### 6.1.1. ScriptableObjecty

V projektu máme celkem 5 typů konfigurovatelných datových položek v podobě ScriptableObjectů. Každá instance ScriptibleObjektu obsahuje parametry dané jeho typem a liší se od ostatních instancí pouze v hodnotě dat. Všechny definice ScriptableObjectů jsme opatřily Unity příkazem, který nám umožní vytvářet jednotlivé instance v editoru. Pro vytvoření instance stačí zmáčknout pravé tlačítko, vybrat možnost Create a kliknout na, který typ chcete vytvořit. Typy Item a Equipment jsou v spolu v položce Inventory. Zde jsou naše ScriptableObjecty a kde se o jejich obsahu můžete dočíst více:

1. Inventory – Item (viz Předměty 4.3)
2. Inventory – Equipment (viz Předměty 4.3)
3. Race (viz Rasa 4.2.7)
4. Ability (viz Schopnosti 4.2.8)
5. Dialogue (viz Objekty s UI 4.4.3)

#### 6.1.2. Prefabs

Prefaby se nalézají ve složce „Prefabs“, a jsou rozděleny do složek, jejichž názvy a obsahy jsou samo vysvětlující, až na dvě výjimky, které mohou být nejasné a které si zde hlouběji vystvětlíme. Jedná se o složky Prefabs/Items a Prefabs/Items/Equipment.

##### Prefabs/Items

Složka Prefabs/Items obsahuje prefaby pickupů, kde každý prefab reprezentuje pickup obsahující daný předmět.

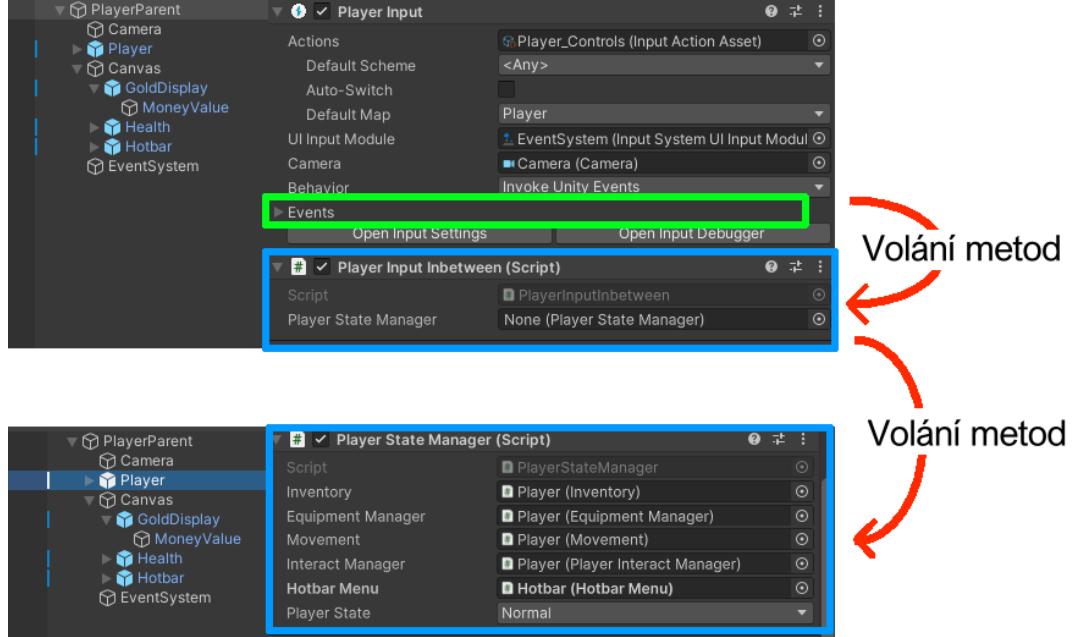
##### Prefabs/Items/Equipment

Složka Prefabs/Items/Equipment obsahuje prefaby *využitého* vybavení. Což jsou, přesněji řečeno, objekty, reprezentující vybavení, které se instancují na postavě, poté co postava dané vybavení *využije*. Tyto prefaby mají připojenou animaci, kterou budou přehrávat při *použití* a poté co postava takovýto prefab instancuje tak k ní také připojí komponenty efektů podle ScriptibleObjectu daného vybavení.

## 6.2. Zpracování Vstupu hráčů

Pro zpracování vstupu hráčů používáme Unity Input System, tento systém přiřadí každému hráči zařízení (ovladač, klávesnice atd.) a Control Scheme, který specifikuje, jaké tlačítka koresponduje, k jaké akci. Komponenta Player Input nám ve vlastnosti *events* (viz Obrázek 59) umožnuje specifikovat, jaké akce mají být zavolány, když hráč aktivuje danou akci. Už víme, že chceme akce interpretovat různými způsoby v závislosti na stavu hráčovy postavy, takže budeme chtít zavolat

metody (každá akce bude mít přiřazenu vlastní funkci) na komponentě PlayerStateManager, která tuto interpretaci provede. Bohužel nemůžeme volat PlayerStateManager přímo z komponenty Player Input, a proto máme vytvořenou komponentu PlayerInputInbetween, která sedí mezi těmito komponentami a umožnuje průchod informacím. Diagram těchto vztahů si můžete prohlédnout zde Obrázek 59.

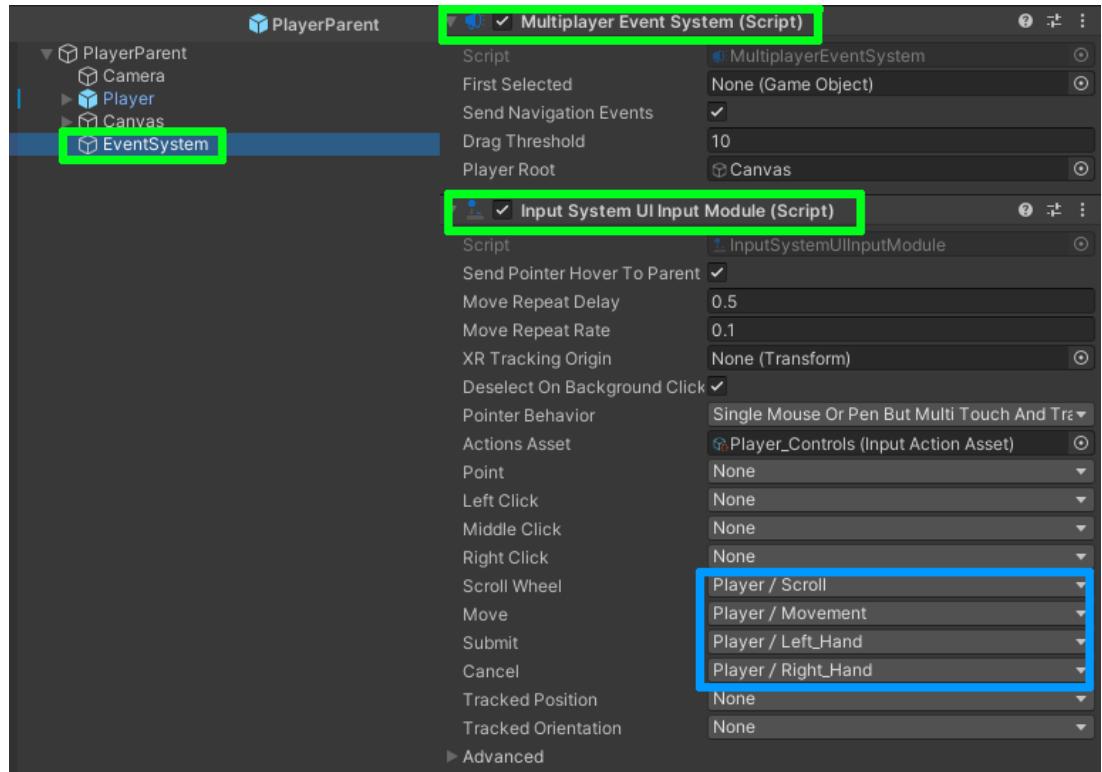


Obrázek 59

Diagram znázorňující hierarchii volání metod pro zpracování vstupů hráče.

### 6.2.1. Vstup pro navigaci UI

Pro UI navigaci využíváme Unity Input System komponenty Multiplayer Event System a Input System UI Input Module, které jsou připojeny ke GameObjektu EventSystem, který je dítětem GameObjektu PlayerParent (viz Obrázek 60), umístění těchto komponent v prefabu hráče je důležité, aby pro něj byly vygenerovány správně. V těchto komponentách máme namapované, které Control Scheme akce odpovídají UI akcím (viz Obrázek 60). Toto zajistí, že se daná UI akce provede, když provedeme danou akci. Proto nevoláme žádnou funkci v PlayerStateManager, když máme akci pohybu a jsme v UI, protože je tato akce už automaticky zpracována touto komponentou.



Obrázek 60

Obrázek ukazující hierarchii prefabu hráče a umístnění komponent Event Systemu.

## 6.3. Interakce s objekty

Interakce s objekty probíhá stejně pro NPC i hráče z pohledu objektu (objekt má pouze metody, které jsou volány), ale rozdíl existuje v rozhodování postav, s jakým objektem, jak interagovat. Zde si rozebereme, jak se hráči od NPC postav liší v tomto rozhodování a managementu interakcí.

### 6.3.1. NPC

NPC postavy konají rozhodování a management interakcí ve svých akcích. Např. akce `PickupItem` se rozhodne interagovat s Pick-upem, čímž přesune daný předmět do svého inventáře, akce `Sleep` se rozhodne započít interakci s postelí a až skončí (úspěšně nebo ne) tak ji ukončí a akce `CraftItem` provede interakci s vyráběcím objektem (např. kovadlina) s parametry určující, který předmět chce vyrobit (např meč.) samotný vyrábějí objekt, se poté už sám postará o odstranění ingrediencí z inventáře postavy a vytvoření nového předmětu.

### 6.3.2. Hráč

Hráč používá pouze jednu komponentu, `PlayerInteractManager`, pro všechny druhy interakcí. Tato komponenta nalezně všechny interakční objekty v okolí a nejbližší zvýrazní, aby hráč věděl, se kterým objektem bude interagovat.

Tato komponenta rozlišuje 2 typy interaktivních objektů, jednorázové a dlouhodobé (viz Objekty 4.4, objekty s UI se považují za jednorázové), pokud hráč za interaguje s jednorázovým objektem, tak `PlayerInteractManager` pouze zavolá metodu `Interact()` na daném objektu. Pokud ale interagujeme s dlouhodobým objektem, tak nejenom zavolá `Interact()` metodu, ale také si objekt zapamatuje.

Pokud hráč pak bude chtít interakci ukončit (např. stlačením tlačítka pro interakci) tak aktivní interakci ukončíme zavoláním metody `UnInteract()` na aktuálním interakčním objektu.

## 6.4. Objekty s UI

Objekty s UI jsou složitější než jednorázové a dlouhodobé objekty, takže si je zde hlouběji vysvětlíme. Každý objekt s UI dědí od třídy `InteractableInMenu` a má uložený prefab svého UI, s tím, že daný prefab má komponentu dědící od třídy `Menu`. Např. objekt `Shop`, má prefab s připojenou komponentou `ShopMenu`.

Když hráč započne interakci s objektem s UI, tak vytvoříme novou instanci UI prefabu, kterou připojíme k `Canvas` objektu daného hráče (k tomuto slouží pomocná metoda `AttachToCanvas()` v třídě `Menu`). Když pak hráč interakci ukončí, tak UI tohoto hráče odstraníme. Takto může více hráčů interagovat se stejným objektem. Každé `Menu` má `Refresh()` metodu, ve které chceme obnovit zobrazovaná data, tuto metodu chceme volat vždy, když se něco na objektu změní (např. když někdo koupí předmět z obchodu, tak zavoláme `Refresh()` na všech UI instancích, aby všichni hráči viděli, že předmět v obchodě už není). `Menu` komponenta zpracovává všechna tlačítka na prefabu.

Každé tlačítko vydává 3 signály, na které si můžeme připravit listener metody. Signály jsou `Select`, `Submit` a `Cancel`. `Select` je aktivován, když označíme tlačítko kurzorem (když se tlačítko stane aktuálně vybraným tlačítkem), když dostaneme tento signál, tak většinou tlačítko zvýrazníme a můžeme např. i změnit popisek aktuálního předmětu, aby odpovídalo novému výběru. Signál `Submit` nastane, když hráč zmáčkne tlačítko na potvrzení, např. v `MenuShop` bychom prodaly nebo koupily vybraný předmět, a v `InventoryMenu` bychom si označili předmět pro přemístění nebo předměty přemístily, pokud jsme už jeden označený měli. Signál `Cancel` většinou něco zruší, nebo má jinou funkci, např. v `InventoryMenu` předmět z inventáře odstraníme a vytvoříme z něj pick-up.

Když si menu dostane signály odpovídající příkazu tak ho předá svému objektu, který příkaz vykoná. Např., když dostane `MenuShop` signál `Select` z 4. tlačítka z obchodů části UI (na rozdíl od části s hráčovým inventářem), tak zavolá metodu `Shop` komponenty `TryToBuyItem()`, s parametry určující 4. předmět a kupujícího hráče. Tato metoda zkонтroluje, zda je koupě možná (zda je dost místa v inventáři hráče a hráč má dostatečný počet zlata) a pokud ano tak provede výměnu zlata za předmět.

### MenuSlot

`MenuSlot` je pomocná komponenta, která nám pomáhá graficky zpracovávat tlačítka. Normální Unity tlačítka pro nás často nejsou graficky dostačující, v mnoha menu totiž chceme, aby tlačítka reprezentovala jednotlivé předměty, chceme tedy, aby měli zobrazeny ikonky předmětů, ale také chceme, aby mohli zvýraznit jejich pozadí, aniž bychom zvýraznili i předmět, tato komponenta změní ikonku a správně zvýrazní pozadí a má několik dalších pomocných metod, takže ji používáme často spolu s normálním tlačítkem, když se jedná o předmětové tlačítko.

## 6.5. Dialog

Dialogový systém funguje podobně jako Objekty s UI, ale s několika rozdíly, které si zde popíšeme, komponenta, která dialog řídí `DialogueManager`, dědí stejně

jako objekty s UI od `InteractableInMenu`. `DialogueManager` má uložený počáteční dialog, ze kterého načte text a možnosti, které vytvoří v podobě tlačítek, kterými může hráč vybrat odpověď. Podle toho, jakou možnost hráč vybere, bude načten další dialog, toto se opakuje, dokud hráč dialog neukončí nebo už neexistuje návazný dialog. Dialogy jsou uloženy v podobě `ScriptableObjektů Dialogue`. Mají uloženy svůj text, odpovědi a odkazy na dialogy, ke kterým odpovědi vedou. Dialogy mohou také mít podmínky (např. má hráč předmět Shem?, má hráč 500 zlata?), které když nejsou splněny, tak se tlačítko, které do tohoto dialogu odkazuje, obarví červeně a hráč ho nemůže vybrat. Dialogy také mohou mít efekty, které se aplikují standardním způsobem, když je daný dialog načten (např. vytvoření spojenectví mezi frakcí hráče a kmenem goblinů).

## 6.6. Předměty

Předměty jsou vždy uloženy jako reference na `ScriptableObjekt` a jak se s nimi operuje je samozřejmé, hlouběji si ale vysvětlíme, co se stane, když se předmět použije, a jak se aktivují jejich efekty. Některé předměty mohou mít délku určující, jak dlouho jejich použití trvá, a proto mají efekty flag určující, zda je chceme aplikovat na začátku použití nebo až na konci. Např. předmět `Icespell` má 2 efekty, `CastEffect`, který vytvoří particle efekt kolem postavy, který chceme, aby započal hned na začátku používání a efekt `SpawnProjectiveOnUseData`, který instanciuje projektil podle daných dat, což chceme, aby se stalo až na konci použití.

## 6.7. Vybavení

Vybavení je trochu složitější, než jsou normální předměty, protože může vytvořit instanci v herním světě, která reprezentuje, že ho postava využívá, zde si rozebereme, jak se tyto instance vytvářejí, kde jsou uloženy a jak mají vypadat prefaby.

Každá postava má komponentu `EquipmentManager`, která si pamatuje, jaké vybavení postava využívá, a volá instanciaci nebo odstranění instancí vybavení v herním světě, když začneme nebo přestaneme využívat vybavení. Instance prefabu a připojení komponent efektů se provádí pomocí metody `InstantiateEquipment()` ve `ScriptableObjektu Equipment`, který také drží všechny vytvořené instance ve své paměti, abychom je později mohli odstranit.

Prefab má několik komponent, `Animator`, který potřebuje mít připojen `Controller` (Unity vygenerovaný animační objekt) odpovídající animacím, které pro něj budeme používat (např. animace `axe_idle` a `axe_use` vyžadují `controller Axe`). Nějakou `Collider2D` komponentu, která udává hitbox našeho vybavení (relevantní hlavně pro zbraně a nástroje, brnění zatím kolize nevyužívá) a naši `Hand_Item_Controller` komponentu, která se ovládá přehrávání animací a chování zbraně (zaručí, abychom nemohli zbraň aktivovat, dokud se nedokončí předchozí aktivace, aktivuje a deaktivuje hitbox atd.). Protože máme `Collider2D` na prefabu, tak musíme pro každou zbraň, která má jiný tvar vytvořit nový prefab s hitboxem, který ji odpovídá, stejně tak, když používáme jiné animace. Důvod proč tato data máme na prefabu a nepředáváme je přes `ScriptableObject`, je, protože to je jednodušší. Tvar hitboxu můžeme snadno natvarovat podle spritu prefabu a pro

animace jsme stejně museli vytvořit nové prefaby, abychom mohli vytvořit jejich animace, takže u nich nám práce navíc ani nevznikla. Prefaby vybavení jsou tedy společné pouze pro vybavení se stejným hitboxem a animací.

Animace jsou také specifické z hlediska eventů, všechny use animace mají na konci event, který oznámi komponentě `Hand_Item_Controller`, že animace skončila. Některé animace mají také speciální eventy, například animace luku má event, který oznamuje, kdy má být vystřelen šíp, `Hand_Item_Controller` tento event dostane a spustí svůj vlastní event, který mohou odposlouchávat ostatní komponenty, například komponenta efektu, který šíp instanciuje.

## 6.8. NPC AI

Zde si popíšeme, jak funguje AI NPC postav a jak spolu souvisejí její různé komponenty.

### 6.8.1. Navigace

Navigace a pohyb postavy jsou ovládány komponentou `NpcAi`. Na této komponentě můžeme zavolat metodu `ChangeTarget()`, které předáme `GameObject`, ke kterému se chceme přesunout a případě minimální vzdálenost, kterou od cíle musíme dosáhnout. Zda jsme úspěšně dosáhli cíle se můžeme dozvědět přečtením pole `reachedEndOfPath`.

### 6.8.2. Stav světa

Stav světa potřebujeme během hledávání plánu, abychom věděli, jak akce modifikují herní svět a abychom věděli, zda tyto modifikace splňují nějaký cíl. K reprezentaci herního světa nám slouží singleton `World` a třída `WorldState`.

Singleton `World` si pamatuje některé z aktuálních aspektů světa, specificky ty, které jsou náročné na počítání a nechceme je vždy počítat znovu, když začínáme vyhledávání nového plánu. Aktuálně si pamatuje všechny truhly a obchody v herním světe a všechny typy předmětů.

Třída `WorldState` je přímo používána v plánování, a drží v sobě všechna data, která bychom mohli používat během plánování. Např. informace o postavě (pozice, životy, obsah inventáře), pickupy v naší blízkosti patřící stejné frakci, truhly a obchody (jako odkazy na `GameObjects`) a jejich obsahy, hashovou tabulkou `completedGoals`, která je používaná pro triviální plánování (viz Analýza GOAP 4.6.2) atd. Důležité je podotknout, jak funguje klonování instance třídy `WorldState`, nejprve vytvoříme novou instanci `WorldState` zavoláním konstruktoru s parametrem `WordState`, který provede mělké okopírování (`shallow copy`), a poté zavoláme některé z metod `Copy-DataToCopy_()`, které provedou hluboké okopírování (`deep copy`), podle toho, jaká data chceme změnit v daném `WordState`. Např. akce `PickupItem` pouze odstraní jeden pickup a přidá jej do inventáře hráče, takže vytvoříme nový `WorldState` a poté zavoláme `CopyItemPickups()` a `CopyInventory()` a poté změníme data nové instance. Toto děláme pro ušetření času a paměti, abychom snížili náročnost plánování.

### 6.8.3. Plány a Node

Plány jsou fronty obsahující třídu `Node`, kde první `Node` reprezentuje první akci, kterou chceme vykonat. Třída `Node` slouží jako vrchol v plánovacím stromu, proto si pamatuje stav světa (`WorldState`) a odkaz na předchozí `Node`. Abychom plán mohli

vykonat, tak si také pamatuje, jakou akci, s jakými parametry, jsme v daném kroku vykonaly (parametry jsou uloženy ve třídě `ActionData` viz Akce 6.8.4).

#### 6.8.4. Akce

Akce jsou komponenty reprezentující akce, které NPC může využít. Během instanciování NPC jsou akce automaticky instanciovány a připojeny k postavě, komponentou `Agent`. Některé akce mohou být specifické pouze pro některé druhy postav, proto komponenta `Agent` při generaci akcí zavolá metodu `IsUsableBy()`, pomocí které může akce oznámit, že daná postava nemá k této akci přístup.

Funkcionalita akce je rozdělena na 2 hlavní části, plánovací a vykonávací, zde si je hlouběji rozebereme.

#### Plánování

Když vytváří třída `Planner` plán (viz Vykonání plánu 6.8.6), tak potřebuje vědět, jak akce změní stav světa a případně jaké parametry byly použity, pokud může akce být použita vícero způsoby. K tomuto slouží metoda `OnActionCompleteWorldStates()`, která vrátí seznam nových `Node`, které mohou vzniknout použitím této akce. Např. pomocí akce `HarvestResource` může NPC těžit železnou rudu, kouzelné krystaly nebo dřevo. Její metoda proto vrátí 3 `Node`, se stavem světa a parametry odpovídající typu suroviny, kterou jsme vytěžily.

Jak jsme viděli na příkladu `HarvestResource`, tak mohou akce mít parametry určeny během plánování, abychom si tyto parametry mohli zapamatovat, tak si pro každou takovou akci vytvoříme pomocnou třídu dědící od třídy `ActionData`, která si bude pamatovat potřebné parametry pro naši třídu.

#### Vykonávání

Vykonávání akce vyžaduje 4 metody, které si zde rozebereme. Když se třída `Agent` rozhodne vykonat akci, tak zavolá metodu akce `Activate()`, tato metoda připraví akci pro její vykonání a případně nače parametry z `ActionData`, které ji byly předány, např. akce `HarvestResource` řekne navigačnímu systému, aby se postava přesunula ke stromu, který jsme dostaly jako parametr, a aby postava *využila* správný nástroj (sekeru), který také vyčteme z parametrů. Potom, co se akce stane aktivní, bude `Agent` každý frame volat metodu `Tick()`, ve které bude probíhat průběžná logika akce a kontrolujeme, zda jsme v akci selhalý nebo uspěli, např. akce `HarvestResource` bude kontrolovat, zda jsme dostatečně blízko cíli (stromu v našem případě), a pokud ano, tak *aktivujeme* náš *využívaný* nástroj, abychom strom pokáceli. Když strom už neexistuje, tak jsme uspěli, a pokud sekeru nemáme, ale strom stále stojí, tak jsme selhali. Při selhání zavoláme metodu `Deactivate()`, která akci deaktivuje a uvede postavu do normálního stavu (např. přestane *využívat* sekeru) a nastaví pole *running* na false. Při úspěchu také deaktivujeme akci a uvedeme postavu do normálního stavu a také nastavíme pole *running* na false a *completed* na true.

#### SubAction

Akce dědící od třídy `SubAction` slouží jako pomocné akce, tyto nejsou nikdy používány během plánování explicitně, ale ostatní akce je mohou použít, aby splnily některé z jejich požadavků. Tyto akce fungují stejně jako normální akce, ale mají často příliš veliký branching faktor, kvůli kterému jsme je nemohli efektivně použít během plánování. např. akce `CraftItem` má branching faktor veliký, jako je počet vyrobiteLNých předmětů ve hře, aktuálně cca 20. Branching faktor je často veliký, protože nevíme, který z mnoha parametrů nám bude užitečný, proto jsme se nechali inspirovat zpětným plánováním a necháme na jiných akcích určit, zda tyto akce

využijeme a s jakými parametry. Pokud chce akce využít nějakou pomocnou akci, tak se ve své metodě `OnActionCompleteWorldStates()` pomocné akce zeptá, zda je splnitelná s potřebnými parametry, a pokud ano tak vytvoří Node s danou pomocnou akcí, kterou použije jako předka pro vlastní Node. Např. akce `HarvestResource` potřebuje sekuru, aby mohla pokáčet strom, sekuru ale postava nemá v inventáři, takže se zeptá pomocné metody `GetTool()` zda můžeme získat sekuru, `GetTool()` se zeptá pomocných akcí `PickUpItem`, `PickItemFromChest` a `CraftItem`, zda mohou získat sekuru, pokud ano, tak dostaneme vrchol s jejich daty, pokud ne, tak sekuru získat nemůžeme a nevrátíme žádný vrchol.

### 6.8.5. Cíle

Cíle jsou třídy dědící od třídy `Goal`. Každá třída reprezentuje jeden cíl, drží aktuální stav cíle, jeho prioritu a podmínky určující, kdy je splněn. Každý cíl může provádět každý frame svoji logiku. Např. cíl `SleepGoal` zvyšuje svoji prioritu podle delta time framu, pamatuje si čas započetí spánku a délku spánku potřebnou ke splnění tohoto cíle. Když si Agent vybere aktivní cíl, tak se ptá pomocí metody `IsCompleted()`, zda jsme ho splnily (např. `SleepGoal` zde kontroluje, zda jsme spali dostatečně dlouho).

NPC postavy mají společný pouze cíl `HealGoal`, který mají už v prefabu, ostatní cíle musíme ale připojit jinak, postavám, které se nerespawnují, můžeme cíle připojit manuálně (např. NPC příšera, viz Zřícenina 3.18), ostatní NPC by ale své cíle zapomněli poté, co se respawnou, protože jejich prefaby jejich specifické cíle nemají.

### Respawn

Nejprve si vysvětlíme, jak se postavy vůbec respawnují. K první instanciaci a pozdějším respawnům slouží komponenta `RespawnPoint`, té předáme prefab postavy, kterou chceme spawnovat a čas, za jak dlouho postavu respawpneme po smrti.

### Jobs

Aby každá postava dostala správné cíle, tak jsme vymysleli novou komponentu `Job`. `Job` reprezentuje jeden nebo více cílů a drží všechny parametry, které bychom chtěli určit. Komponenty typu `Job` připojíme ke stejném GameObjektu, ke kterému je připojena komponenta `RespawnPoint`, když `RespawnPoint` respawnuje postavu, tak zavolá metody `SetGoalsOfAnNPC()` všech `Job` komponent, které připojí správné cíle se správnými parametry k nově spawnuté postavě. Např. `RespawnPoint` dřevorubce by měl připojen `SleepJob`, s referencí na postel v dřevorubecké chatě a `GatherJob` s referencí na truhlu v dřevorubecké chatě s požadavkem naplnění inventáře předmětem `Log`.

### 6.8.6. Vytvoření plánu

Plán vytváří třída `Planner`, dostane cíl, který chceme splnit, počáteční `WorldState` a dostupné akce, poté hledá pomocí BFS do maximální hloubky vrchol, který splňuje daný cíl. Pokud takový vrchol nebyl nalezen, tak vrátí null, pokud jsme ho našli tak vytvoříme plán (frontu tříd `Node`) přechodem z daného vrcholu přes jeho rodiče až do kořene.

Komponenta `Agent` využívá třídu `Planner` pro nalezení plánu pro splnění nějakého cíle. Cíle jsou seřazeny dle jejich priority a `Agent` se vždy snaží splnit cíl s nejvyšší prioritou, pokud pro takový cíl plán nelze nalézt, tak se pokusí splnit cíl

s druhou největší prioritou atd. Když plán existuje, tak si vybereme daný cíl jako aktivní cíl a vezmeme první akci z plánu, která se stane naší aktivní akcí (zavolá se její metoda `Activate()` a bude se opakovaně volat její metoda `Tick()`, viz Akce 6.8.4).

#### 6.8.7. Vykonání plánu

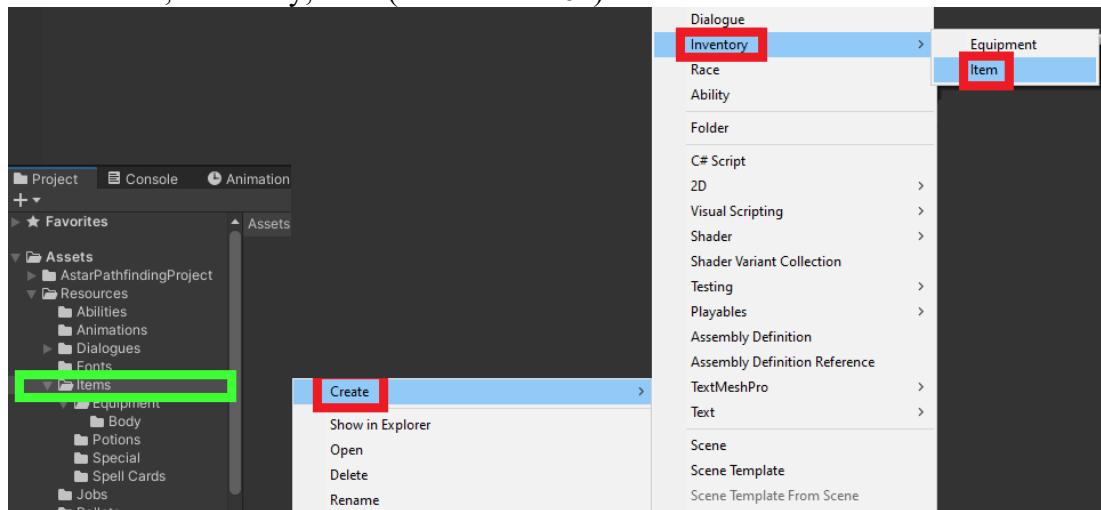
Vykonání plánu je organizováno komponentou `Agent`. Pokud máme nějaký plán a aktivní akci, tak vždy kontrolujeme, zda úspěšně doběhla, čtením jejích vlajek *running* a *completed*. Pokud skončila akce úspěšně (*running* == false a *completed* == true), tak načteme následující akci z plánu, která se stane naší novou aktivní akcí. Pokud v plánu už žádné akce nezbývají, tak oznámíme cíli, že byl splněn a začneme hledat nový plán. Pokud byla akce selhala (*running* == false a *completed* == false) tak plán zahodíme, cíl deaktivujeme a začneme hledat nový plán.

# 7. Návody pro vývojáře

Aby bylo pro vývojáře snazší pochopit, jak vyrobit některé ze složitějších assetů nebo komponent, jsme na ně zde sepsali návody na jejich vytvoření.

## 7.1. Jak vytvořit nový předmět

Abychom vytvořili nový předmět, tak si otevřeme složku Resources/Items (nebo jakémkoliv jeho podsouboru, aby mohli scripty předmětu načíst) a vytvoříme v ní novou instanci ScriptableObject Item stisknutím pravého tlačítka a vybráním možnosti Create, Inventory, Item (viz Obrázek 61).

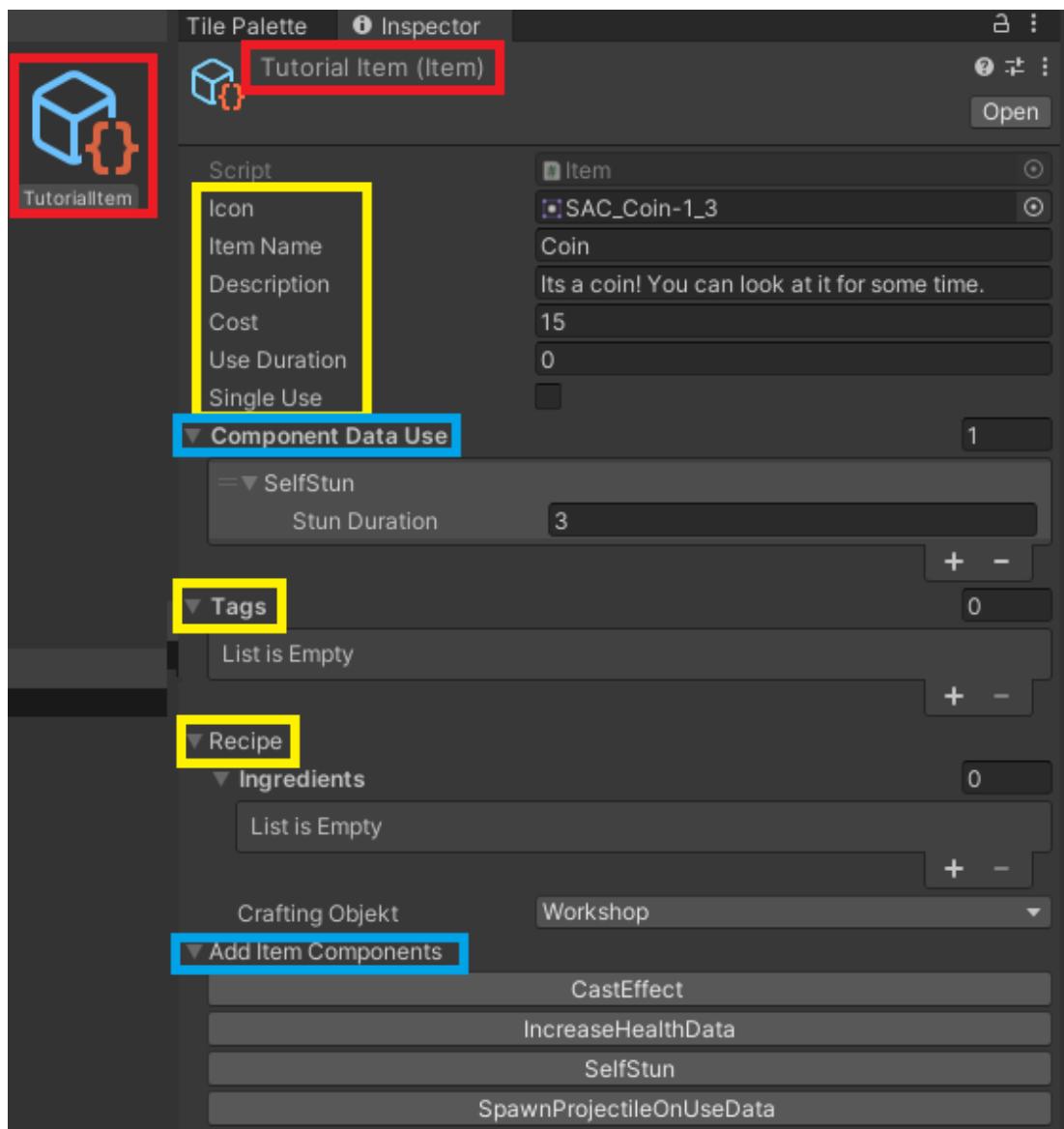


Obrázek 61  
Obrázek ukazující, kde a jak vytvořit instanci Itemu

Nově vytvořený ScriptableObject může být pojmenovaný jakkoli. Když na něj klikneme tak se nám v inspektoru zobrazí jeho datová pole (viz Obrázek 62), které doplníme abychom vytvořili data předmětu. Nyní si individuálně rozebereme všechna datová pole předmětu:

- 1) Icon, ikonka předmětu, je používaný v meny a pick-upech.
- 2) Item Name, jméno předmětu, toto je unikátní identifikátor předmětu, takže by mělo být rozdílné od všech ostatních předmětů. Také je používáno, když se zobrazují informace o předmětu, např. v menu inventáře nebo obchodu.
- 3) Description, popis předmětu, zobrazí se spolu s informacemi o předmětu v některých menu (např. inventář, obchod atd.).
- 4) Cost, cena kupi a prodeje předmětu v obchodech, suroviny mají kolem 10, zbraně kolem 80 a jedinečné předměty několik stovek.
- 5) Use Duration, jak dlouho trvá použití předmětu v sekundách, během tohoto času je postava neovladatelná. Některé efekty se aplikují hned po započetí použití a některé až po uběhnutí této doby, kdy se efekt aplikuje se dá přečíst v jeho scriptu.
- 6) Single use, zda je předmět použitelný pouze jednou, pokud ano, tak je předmět po použití odstraněn z inventáře.
- 7) Component Data use, seznam připojených efektů s jejich daty, které můžeme libovolně změnit. V našem případě máme jediný efekt SelfStun, který nás okamžitě (aplikuje se po uběhnutí Use Duration a my máme Use Duration 0)

- omráčí (postava se stane neovladatelnou) na tři sekundy. Nový efekt můžeme přidat stisknutím tlačítka v sekci Add Item Components (viz Obrázek 62).
- 8) Tags, značky, AI používá, aby vědělo, jak předměty používat. Např. tagy Healing, Attack buff, Armor buff, Attack Spell, Defence Spell říkají, jaký efekt, nebo v jaké situaci by AI mělo předměty použít. Předmět s tagem Healing oživí životě postavy, Attack buff zvýší postavě útok, Attack spell vytvoří útok, který musí být namířený na nás cíl etc. Ostatní tagy slouží jako informace o druhu vybavení, vybavení s tagem Armor budou AI vyhledávat, když budou chtít brnění, vybavení s tagem Melee weapon budou vyhledávat vojáci kteří chtějí zbraň na blízko atd.
  - 9) Recipe, recept k vyrobení tohoto předmětu. Do seznamu ingrediencí můžeme vložit reference na ScriptableObjekty předmětů. Pokud je seznam ingrediencí prázdný tak je předmět nevyrobiteLNÝ. Crafting Objekt reprezentuje druh vyráběcího objektu, který musíme použít abychom tento předmět mohli vyrobit.



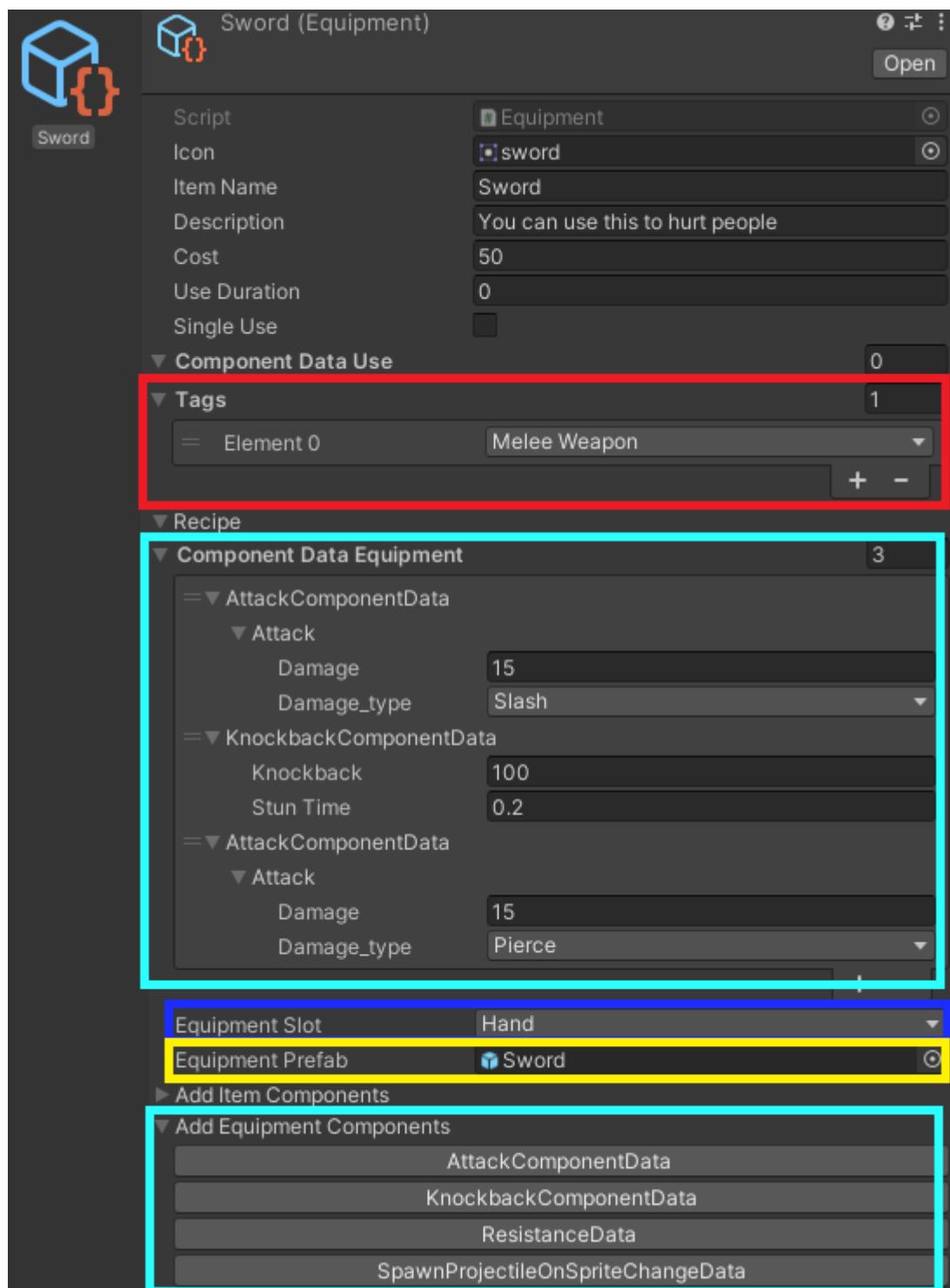
Obrázek 62

Obrázek ukazující data předmětu.

## 7.2. Jak vytvořit vybavení

Vybavení je rozšířený předmět, takže vyžaduje stejná data jako předmět. Nejprve vytvoříme ScriptableObject Equipment ve složce Resources/Items, tak jako předmět (viz Obrázek 61), ale vybereme možnost Equipment místo Item a doplníme do něj data stejně jako do předmětu. Vybavení má ale nějaká datová pole, specifická pro něj, která si zde vysvětlíme na příkladu meče (viz Obrázek 63):

- 1) Tags jsme již měli u předmětu, zde pro meč jsme vybrali tag Melee Weapon, protože je to zbraň na blízko.
- 2) Component Data Equipment, drží data o komponentách vybavení, princip je stejný jako v Component Data Use. Když postava *využije* vybavení, tak se k němu připojí komponenty vytvořeny z těchto dat. Meč takto dostane 3 komponenty, 2 komponenty Attack, které meči přidají efekt udělení poškození a komponentu Knockback, která meči přidá efekt knockback. Nové Datové Komponenty mohou být přiřazeny tlačítkem v podmenu Add Equipment Components. Komponenta SpawnProjectileOnSpriteChangeData je trochu složitější a rozebereme si ji v sekci Jak vytvořit zbraň na dálku a projektil 7.2.2.
- 3) Equipment Slot, ve kterém poli vybavení budeme vybavení moct využívat.
- 4) Equipment Prefab, odkaz na prefab vybavení, v našem případě odkazujeme na prefab meče, který si rozebereme v následující sekci.



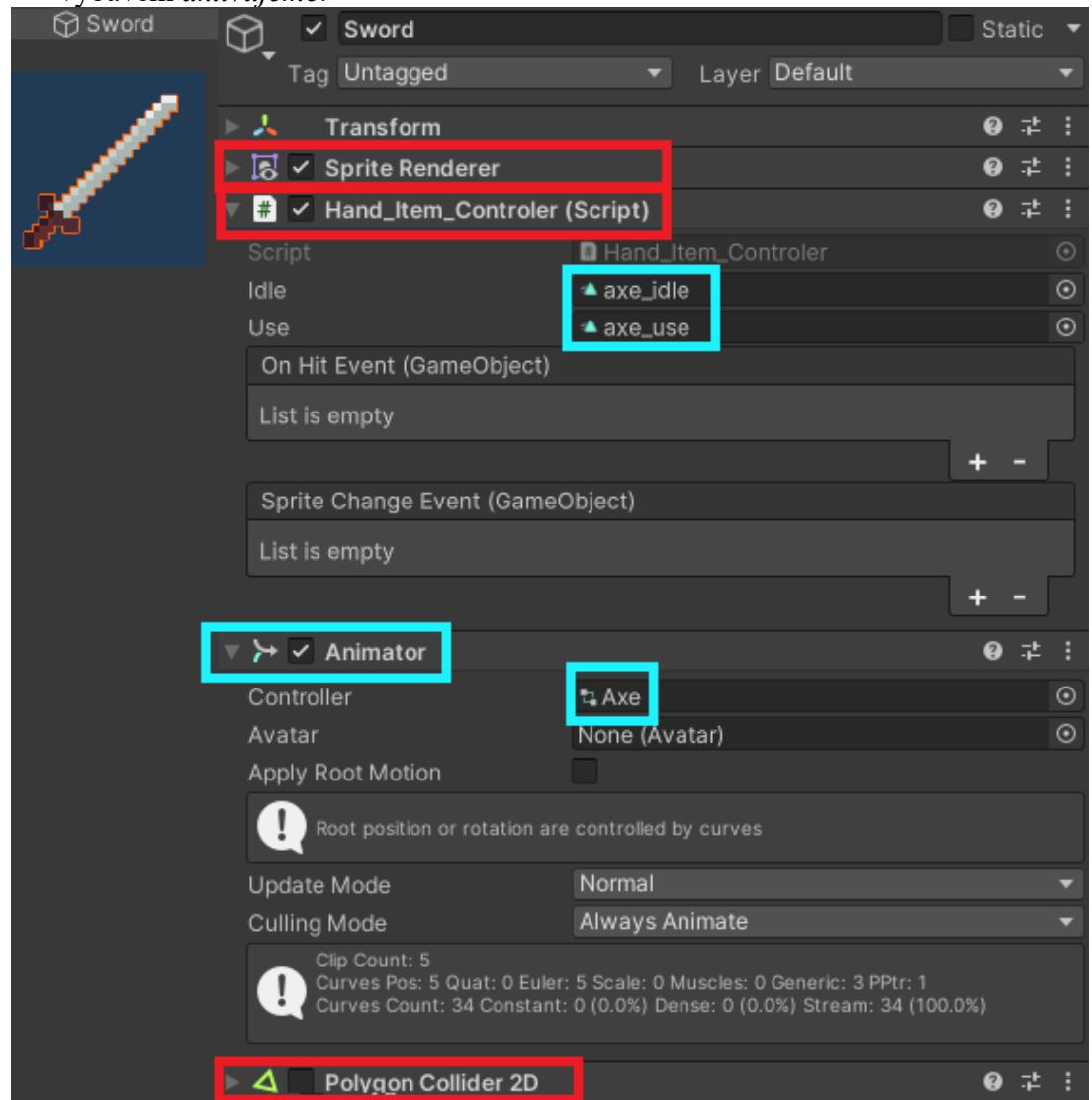
Obrázek 63  
Obrázek ukazující data vybavení

### 7.2.1. Jak vytvořit Prefab vybavení

Zde si rozebereme vytvoření prefabu vybavení, specificky vybavení typu ruka, protože vybavení ostatních typů sdílejí stejný triviální prefab (GameObject pouze s komponentou SpriteRenderer). Jako příklad nám bude sloužit prefab meče (viz Obrázek 64).Prefab je GameObject bez dětí, který má připojen tyto komponenty:

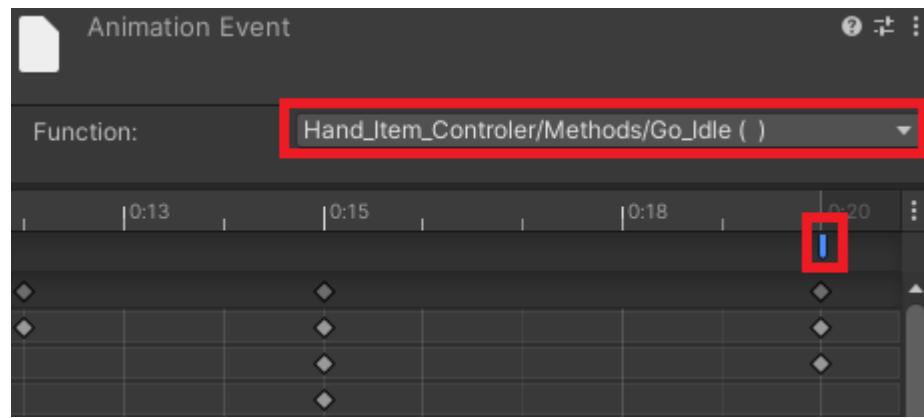
- 1) Sprite Renderer, do které se načte sprite uložen v Icon ScriptableObject vybavení.

- 2) Hand\_Item\_Controller, do které musíme vložit odkazy na Idle a Use animace. Idle animace nemá žádné speciální požadavky, ale Use animace musí mít na konci event, který spustí metodu Go\_Idle() na komponentě Hand\_Item\_Controller (viz Obrázek 65), což ukončí *aktivní* stav vybavení. Můžeme vidět, že jsme pro meč použili stejnou animaci jako pro sekuru, protože jsme usoudili, že budou sekat stejným způsobem.
- 3) Animator, s controllerem Axe, která slouží k přehrávání axe animací.
- 4) Polygon Collider 2D, je vytvarován, aby zahrnul celou čepel. Komponenta zde musí být vypnuta, protože chceme, aby byla zapnuta pouze, když vybavení *aktivujeme*.



Obrázek 64

Obrázek ukazující prefab meče a jeho komponenty.



Obrázek 65  
Obrázek ukazující event animace Use.

### 7.2.2. Jak vytvořit zbraň na dálku a projektil

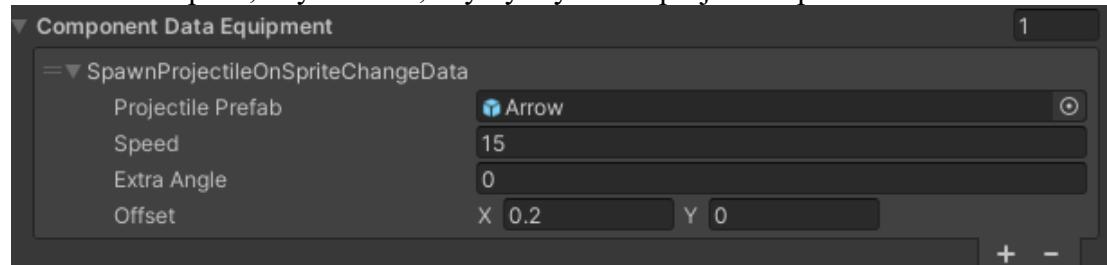
Pro vytvoření střelné zbraně potřebujeme 3 věci:

- 1) Use animaci s eventem, který spustí metodu `SpriteChangeInAnimation()` komponenty `Hand_Item_Controller`. Chceme, aby tento event byl umístěn do momentu, kdy chceme vytvořit náš projektil, např v animaci luku (viz Obrázek 66) umístíme event do momentu, kdy nám šíp zmizí z naší animace.
- 2) Data `SpawnProjectileOnSpriteChangeData` v `ComponentDataEquipment` `ScriptableObject` našeho vybavení, v našem případě našeho luku (viz Obrázek 67). Potřebná data jsou odkaz na prefab projektilu, který bude instanciován po aktivaci `SpriteChange` eventu. Rychlosť projektilu, počáteční úhel spritu projektilu (kdybychom chtěli, aby byl sprite projektilu otočen jinak než v prefabu) a offset pozice instanciování projektilu (defaultní pozice je střed vybavení), abychom mohli přesně specifikovat pozici instanciování projektilu.
- 3) Projektil je prefab se spritem a komponentou `Projectile`, která se stará o pohyb projektilu a inicializaci jeho komponent. Komponenty určující chování projektilu se přidají stejně jako pro předměty a vybavení, přes seznam dat komponent (viz Obrázek 68). V našem případě máme šíp, který má komponenty. Komponentu s efektem udělení poškození, komponentu sebezničení po zásahu a komponentu sebezničení po uplynutí 30 sekund. Nová data komponent se dají přidat pomocí tlačítka v podmenu Add Projectile Components.



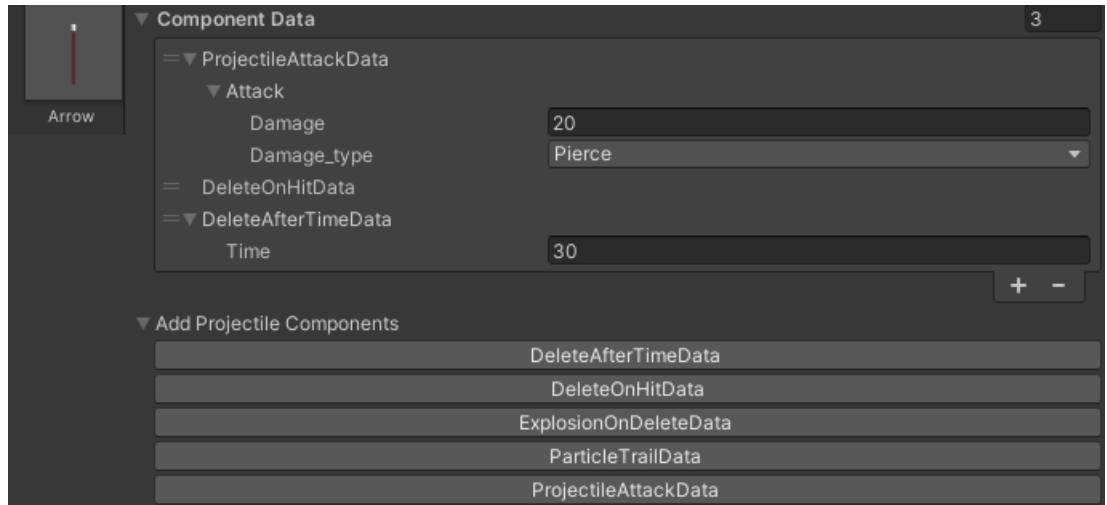
Obrázek 66

Obrázek ukazující SpriteChange event animace a jeho umístění v čase změny spritu, kdy chceme, aby byl vytvořen projektil šípu.



Obrázek 67

Obrázek ukazující data potřebná k vytvoření projektilu.



**Obrázek 68**  
Obrázek ukazující data komponent projektilů.

### 7.3. Jak vytvořit efekt

Abychom vytvořili nový efekt tak potřebujeme vytvořit komponentu, reprezentující samotný efekt a pomocnou třídu, která tuto komponentu vytvoří a připojí na správné místo. Podíváme se na jednoduchou komponentu efektu **WeaponAttack** (viz Obrázek 69). Tato komponenta drží data útoku v proměnné **attack** (**Attack** drží typ útoku a množství poškození) a metodu **Attack()**, která poškození aplikuje na platný **GameObjekt**.

Dále musíme vytvořit pomocnou třídu, která tuto třídu dokáže inicializovat (viz Obrázek 70). Abychom tyto třídy mohli připojit do seznamu přes tlačítko v editoru, tak musejí dědit od správně třídy, v našem případě chceme vytvořit komponentu pro vybavení, takže budeme dědit od **EquipmentComponentData<WeaponAttack>**, s generickým typem třídy, kterou chceme vytvořit. Dále máme uložena data útoku v proměnné **attack**, abychom je mohli v editoru změnit a předat komponentě **WeaponAttack**. Nakonec máme metodu **InitializeComponent()**, která bude zavolána, když potřebujeme vytvořit tuto komponentu (v našem případě, když se vybavení začne *využívat*). Pro komponenty vybavení dostaneme jako parametry **ScriptableObject** daného předmětu a instanci prefabu **Vybavení**. My pouze přidáme komponentu **WeaponAttack** k instanci vybavení, inicializujeme její data a zapíšeme její metodu **Attack()** jako listener k eventu **onHitEvent**.

```

    Unity Script | 6 references
    public class WeaponAttack : MonoBehaviour
    {
        public Attack attack;
        public void Attack(GameObject g)
        {
            Damagable damagable = g.GetComponent<Damagable>();
            if(damagable != null)
            {
                damagable.TakeDamage(attack);
            }
        }
    }

```

Obrázek 69

Obrázek ukazující třídu WeaponAttack

```

    public class AttackComponentData : EquipmentComponentData<WeaponAttack>
    {
        public Attack attack;

        public override void InitializeComponent(GameObject weapon, Item item)
        {
            WeaponAttack weaponAttack = weapon.AddComponent<WeaponAttack>();
            weaponAttack.attack = attack;
            weapon.GetComponent<Hand_Item_Controller>().onHitEvent.AddListener(weaponAttack.Attack);
        }
    }

```

Obrázek 70

Obrázek zobrazující kód pomocné třídy AttackComponentData.

## 7.4. Jak vytvořit interaktivní objekty a jejich UI

Objekty s UI jsou nejsložitější z interaktivních objektů, takže si zde vysvětlíme, jak takový objekt vytvořit.

Objekt s UI se skládá ze 3 částí, komponenty objektu, komponenty UI a UI prefabu. V této sekci si ukážeme, jak každou z částí vytvořit a jak k sobě zapadají na příkladu vyráběcího objektu.

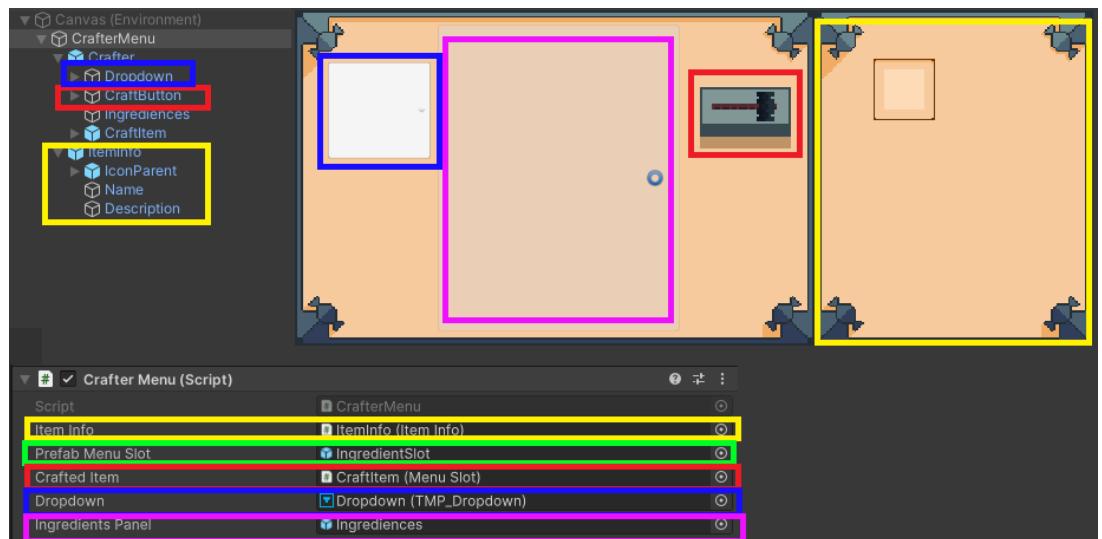
### 7.4.1. UI prefab

UI prefab poskládáme ze základních UI elementů, které nám jsou v Unity k dispozici. Nás prefab pro menu vyráběcího předmětu (viz Obrázek 71) se skládá z panelů, které nám slouží k rozmístění ostatních elementů a jako pozadí. V modrém obdélníku jsme použily tlačítko **Dropdown**, ve kterém budeme vybírat předmět, který chceme vyrobit. V červeném obdélníku je normální tlačítko, které stiskneme, abychom potvrdili výrobu předmětu. Ve žlutém obdélníku je celý nás prefab **ItemInfo**, který zobrazuje informace o vybraném předmětu. Ve fialovém obdélníku je panel, který používá **GridLayoutGroup**, která zajistí, aby se její děti zobrazily v mřížce, sem bude naše UI komponenta generovat obrázky ingrediencí, které potřebujeme pro výrobení vybraného předmětu.

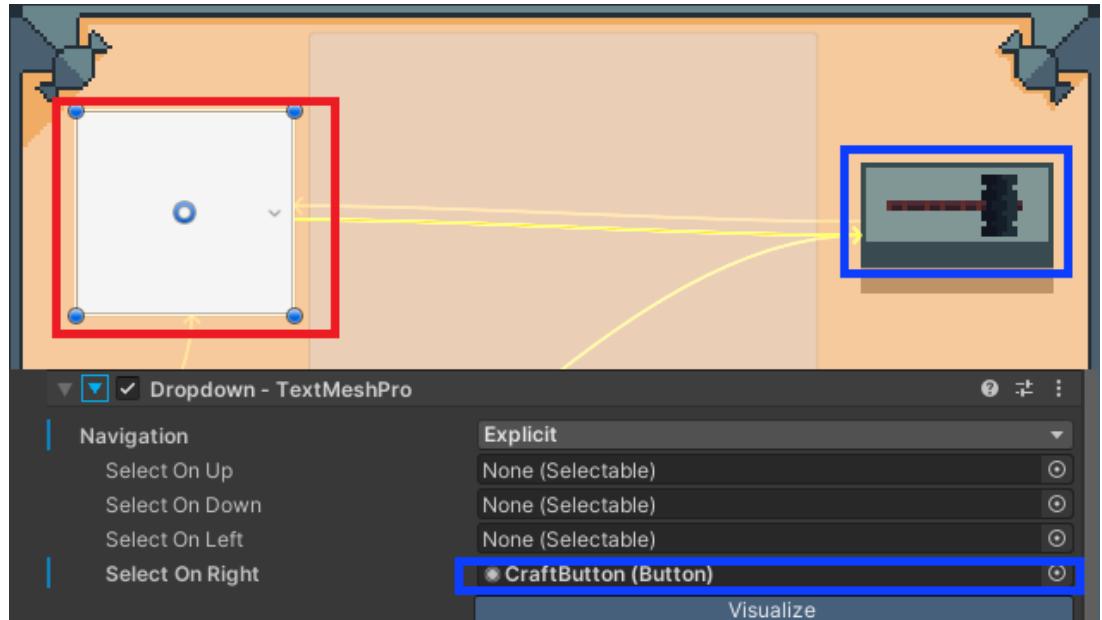
Aby mohl hráč UI navigovat, tak musíme přidat spoje mezi tlačítka (viz Obrázek 72), abychom přidali spoje musíme kliknout na tlačítku a v inspektoru přepnout komponentu **Navigation** do módu **Explicit**, poté přidáme odkazy na jiné tlačítka, kam se hráč pohně po zmáčknutí daného směru.

Nakonec, můžeme do eventů tlačítek přidat odkazy na metody naši komponenty **CrafterMenu**, které chceme, aby byli po zmáknutí tlačítka zavolány. Alternativně se ke tlačítkům můžeme přihlásit z komponenty **CrafterMenu** (např. pokud generujeme tlačítka dynamicky)

UI prefab musí mít připojenou naši UI komponentu **CrafterMenu**, do které musíme vložit odkazy na elementy, aby k nim měla přístup.



Obrázek 71



Obrázek 72

#### 7.4.2. UI komponenta

UI komponenty vždy dědí od komponenty **Menu**, (viz Obrázek 73), samozřejmě vytvoříme odkazy na potřebné třídy (viz červený obdélník), v metodě **Start()** můžeme provést nějakou inicializaci, ale hlavní inicializaci bychom měli provádět v metodě **Initialize()**, ta je totiž zavolána komponentou **Crafter** s inicializačními parametry, které v našem případě obsahují odkaz na hráče a na komponentu **Crafter**. V metodě **Initialize()** (viz žlutý obdélník) musíme vždy určit tlačítko, na kterém hráč započne navigaci v menu, pokud tlačítko neurčíme tak nebude hráč menu moci navigovat. Dále musíme připojit naše UI pod

Canvas objekt hráče, k čemuž slouží metoda `AttachToCanvas()`, tuto metodu zavoláme až potom, co vložíme odkaz na hráče do proměnné `player`. Zbytek kódu inicializuje UI instanci se správnými hodnotami, načež recepty, které nás vyráběcí objekt může vyrábět a vložíme je do `Dropdown` tlačítka a vybereme první předmět jako námi vybraný pro výrobu. Další metody slouží pro změnu receptů nebo výrobu předmětů, a jsou volány po zmáčknutí tlačítka.

```

public class CrafterMenu : Menu
{
    int recipeIndex; // just used to refresh the item description after sale

    List<CraftingRecipe> recipes;
    bool craftable;
    [SerializeField]
    ItemInfo itemInfo;
    [SerializeField]
    GameObject prefabMenuSlot;
    Crafter crafter;
    [SerializeField]
    MenuSlot craftedItem;
    [SerializeField]
    TMP_Dropdown dropdown;
    [SerializeField]
    GameObject ingredientsPanel;
    private void Start()
    {
        Initialize(crafter, player);
    }

    public void Initialize(Crafter crafter, GameObject player)
    {
        this.crafter = crafter;
        this.player = player;

        // select first button
        player.GetComponent<GameManager>().SelectObjectGetComponentInChildren<Button>().gameObject = dropdown.gameObject;

        AttachToCanvas();

        Inventory playerInventory = player.GetComponent<Inventory>();
        dropdown = gameObject.GetComponentInChildren<TMP_Dropdown>();

        LoadRecipes();
        foreach (CraftingRecipe recipe in recipes)
        {
            Item result = recipe.result;
            dropdown.options.Add(new TMP_Dropdown.OptionData(result.name, result.icon));
        }
        dropdown.onValueChanged.AddListener(delegate{ SwitchRecipe(); });
        LoadRecipe(0);
    }

    void LoadRecipes()
    {
        crafter.GetCraftableRecipes();
        foreach (CraftingRecipe recipe in crafter.craftableRecipes)
        {
            Item result = recipe.result;
            dropdown.options.Add(new TMP_Dropdown.OptionData(result.name, result.icon));
        }
    }

    public void SwitchRecipe()
    {
        crafter.Craft(dropdown.value);
    }

    public void EraseRecipe()
    {
        crafter.EraseRecipe(recipeIndex);
    }

    void LoadRecipe(int index)
    {
        crafter.LoadRecipe(index);
    }

    public void Craft()
    {
        crafter.Craft();
    }
}

```

Obrázek 73

#### 7.4.3. Komponenta objektu

Komponenta interaktivního objektu s UI vždy dědí od `InteractableInMenu` (viz Obrázek 74), dále vytvoříme odkaz na prefab našeho menu a enum určující jaký druh vyráběcího předmětu reprezentujeme (viz modrý obdélník). Poté musíme implementovat metody `Interact()` a `UnInteract()`, které započnou a ukončí interakci. K započetí interakce vygenerujeme UI pro daného hráče v metodě `GenerateUi()` (viz červený obdélník), nejprve vytvoříme instanci našeho UI a získáme odkaz na menu komponenty našeho UI, načež komponentu inicializujeme a přidáme ho do seznamu vygenerovaných menu. Při ukončení interakce stačí vymazat UI daného hráče pomocnou metodou `DeleteUi()`.

```

public class Crafter : InteractableInMenu
{
    [SerializeField]
    GameObject prefabCrafterUi;
    [SerializeField]
    CraftingObjekt craftingObjekt;
    4 references
    public CraftingObjekt CraftingObjekt { get => craftingObjekt; }
    //ItemSlot crafterItem;

    GameObject player;

    @ Unity Message | 0 references
    public void Start()
    {
    }

    2 references
    public override void Interact(GameObject new_player)
    {
        player = new_player;
        GenerateUi(player);
    }

    3 references
    public override void UnInteract(GameObject player)
    {
        DeleteUi(player);
    }

    1 reference
    void GenerateUi(GameObject player)
    {
        CrafterMenu crafterMenu = Instantiate(prefabCrafterUi, player.GetComponent<MenuManager>().canvas.transform).GetComponent<CrafterMenu>();
        crafterMenu.Initialize(this, player);
        menus.Add(crafterMenu);
    }
}

```

Obrázek 74

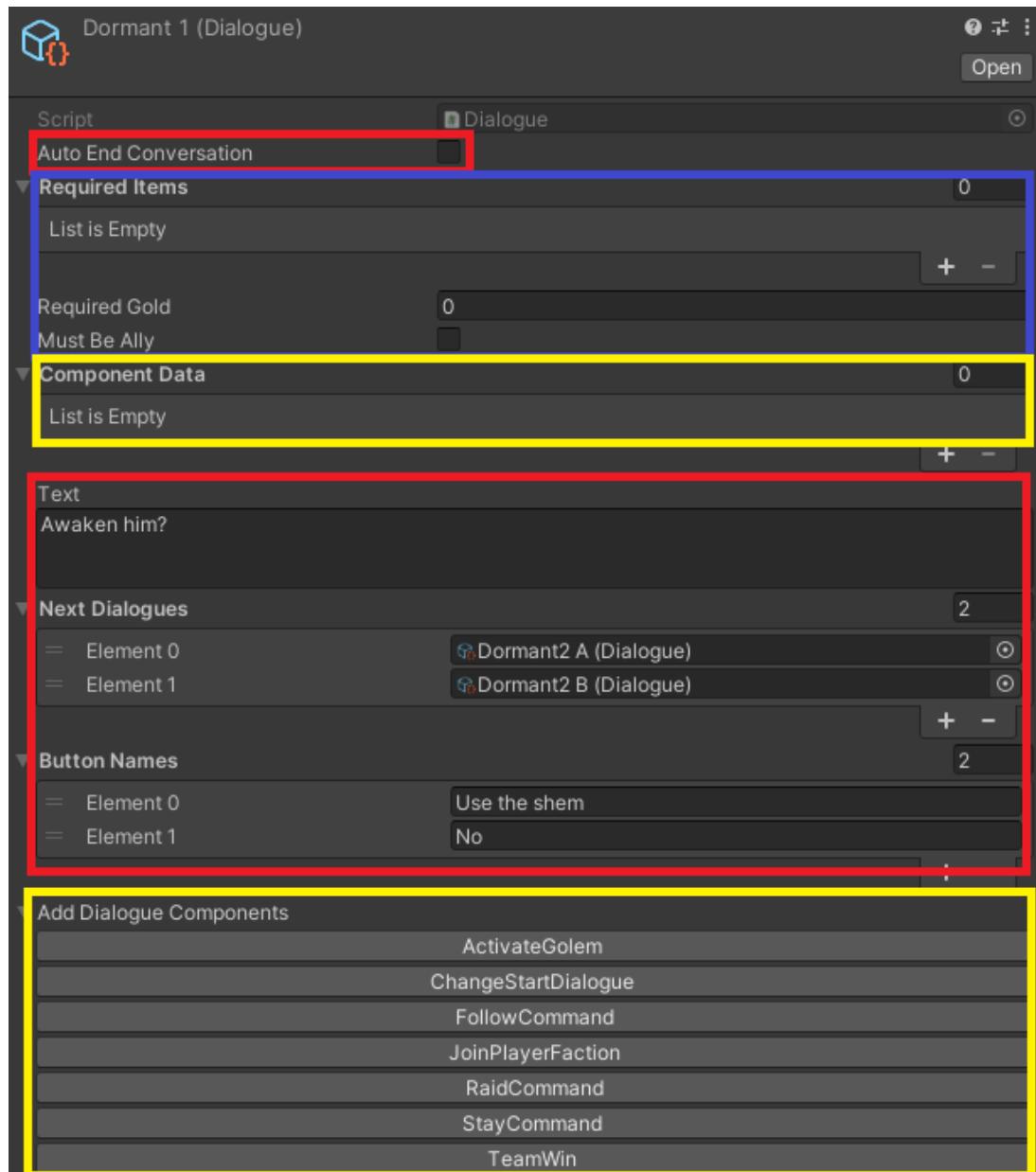
## 7.5. Jak vytvořit dialog

Zde si hlouběji rozebereme, jak vytvořit dialog. ScriptableObject Dialogue se skládá ze 3 hlavních částí, podmínek (modré), efektů (žluté) a návaznosti (červené), viz Obrázek 75.

Abychom mohli dialogu dosáhnout (pokud to není počáteční dialog) tak musejí nejprve být splněny jeho podmínky, máme seznam předmětů Required Items, pokud hráč tyto předměty nemá, tak nemůže vybrat odpověď, která by ho dostala do tohoto dialogu, požadované předměty jsou vždy odstraněny, když se hráč do dialogu dostane. Required Gold značí počet zlata, které je po hráči vyžadované, toto množství zlata je také odstraněno, když hráč dosáhne tohoto dialogu. Nakonec možnost Must Be Ally, která dovolí hráči dosáhnout tohoto dialogu pouze pokud je hráč spojencem postavy se kterou zahájil dialog.

Efekty nastanou vždy, když je dialog dosažen, efekty fungují jako v ostatních případech (viz Jak vytvořit nový předmět 7.1), máme seznam efektů a můžeme je přidávat tlačítka v podmenu Add Dialogue Components.

Po aktivaci všech efektů zkontrolujeme Auto End Conversation, pokud máme tuto možnost zapnutou tak se dialog okamžitě ukončí. Jinak se zobrazí dialogové okno zobrazující text v poli Text. Seznamy Next Dialogues a Button Names reprezentují tlačítka. Button Names drží texty tlačítek a Next Dialogues drží odkazy na dialogy kam se konverzace přesune po stisknutí odpovídajícího tlačítka. Pokud je element z Next Dialogues rovný null (nikam se neodkazuje) tak dané tlačítko konverzaci ukončí.



Obrázek 75

## 7.6. NPC a AI

V této sekci si vysvětlíme, jak vložit NPC do herního světa, jak vytvořit cíle a připojit k NPC cíle a jak vytvořit akce, které bude ke splnění cílů používat.

### 7.6.1. Jak vytvořit nový cíl

Vytvoření cíle si ukážeme na příkladu oživovacího cíle HealGoal (viz Obrázek 76), který bude po NPC požadovat, aby se oživilo, když je zraněno.

Nejprve cíl v konstruktoru inicializujeme (viz červený obdélník Obrázek 76), nás cíl je *triviální* (víme, že k jeho splnění bude zapotřebí pouze jedna akce) takže můžeme dobrovolně přepsat hodnotu MaxPlanDepth na 1, což zastaví hledání plánu pro tento cíl v po hloubce 1, abychom zbytečně neprohledávali delší plány, které víme že nebudou existovat. Dále, protože chceme vědět aktuální životy naši postavy, tak přihlásíme naši UpdatePriority() metodu k healthChange eventu

komponenty `Damagable`. V metodě `UpdatePriority()` zvětšujeme naší prioritu podle míry zranění, čímž zraněnější jsme, tím více se chceme oživit. Priority cílů se pohybují mezi hodnotami 0-10, kde 0-3 jsou cíle, které vykonáváme ve volném čase (spánek), 4-7 jsou cíle, které jsou hlavní náplní naší práce (těžení surovin, hlídkování) a 8-10 jsou cíle, které musíme urgentně vykonat (bojovat proti blízkému nepříteli), tyto hranice nejsou pevně dané, ale slouží pro vývojáře jako návod, jakou prioritu by měli jeho nové cíle mýt.

Nyní musíme implementovat metodu `CompletedByState()` (viz modrý obdélník Obrázek 76), která oznámí plánovači, zda stav splňuje tento cíl. Tento cíl je *triviální* (akce, která cíl plní, explicitně řekne, že cíl plní), takže se stačí podívat, zda obsahuje seznam `completedGoals` nás cíl. Pokud by akce triviální nebyla, tak bychom museli vyčist, zda náš cíl byl splněn z hodnot uložených ve stavu, v našem případě bychom museli do stavu světa přidat novou hodnotu `myHealth`, kterou bychom pak akcemi během plánování modifikovali, zde bychom pak porovnali, zda je míra životů ve stavu větší než aktuální míra životů a pokud ano tak bychom vrátili `true`.

V metodě `CalculatePriority()` (viz žlutý obdélník Obrázek 76) vrátíme prioritu tohoto cíle, kterou jsme si už přepočítali, takže stačí vrátit hodnotu `currentPriority`.

Aby NPC začalo cíl používat tak stačí tuto komponentu k postavě připojit během instanciace, nebo přímo k NPC prefabu před instanciací, cíle postavy jsou totiž načteny pouze jednou, a to okamžitě po instanciaci, komponentou `Agent`.

```

5 namespace GOAP {
6     public class HealGoal : Goal
7     {
8         float currentPriority = 0;
9         public override void Start()
10        {
11            MaxPlanDepth = 1;
12            GetComponent<Damagable>().healthChange.AddListener(UpdatePriority);
13        }
14        void UpdatePriority(float hp, float maxHp)
15        {
16            currentPriority = 10 * (1 - hp / maxHp);
17        }
18        public override bool CompletedByState(WorldState state)
19        {
20            return CloserToGoalCheck(state);
21        }
22        public bool CloserToGoalCheck(WorldState state)
23        {
24            return state.completedGoals.Contains(this);
25        }
26        public override void Complete()
27        {
28            active = false;
29        }
30        public override float CalculatePriority()
31        {
32            return currentPriority;
33        }
34    }

```

Obrázek 76

### 7.6.2. Jak vytvořit novou akci

Vytvoření akce si ukážeme na akci, která bude splňovat předem vytvořený cíl HealGoal (viz Jak vytvořit nový cíl 7.6.1). Akce vždy dědí od třídy Action. Akci musíme vždy inicializovat a říct, jaký druh bubliny se zobrazí, když začne být vykonávána (aby hráč zhruba věděl jakou akci NPC vykonává), v našem případě máme bublinu typu Heal (viz červený obdélník Obrázek 77). Kromě inicializace má dvě hlavní části, které musí implementovat:

- 1) vykonání samotné akce v herním světě,
- 2) efekt akce na stav světa, během plánování.

Nejprve se podíváme na vykonání akce v herním světě, a poté se podíváme, jak daná akce modifikuje stav světa během plánování.

#### Vykonání akce v herním světě

Vykonávání akce začne zavoláním metody `Activate()`, která dostane argumenty `ActionData`, kde jsou případně uloženy naplánované argumenty akce, mi ale žádné argumenty nepotřebujeme, a tak argument nevyužíváme (viz modrý obdélník Obrázek 77). Po aktivaci je každý frame zavolána metoda `Tick()`, naše akce předpokládá, že už v inventáři máme nějaký oživovací předmět, takže zavoláme metodu `UseItem()` se seznamem tagů, které chceme, aby předmět měl (v našem

případě pouze tag healing). Pokud je předmět s tagy v inventáři postavy, tak ho postava *použije* a metoda vrátí true, načež zavoláme metodu **Complete()**, pokud takový předmět nalezen není, tak metoda vrátí false a mi zavoláme metodu **Deactivate()**. Metody **Deactivate()** a **Complete()** v našem případě pouze nastaví správně hodnoty, na které poté reaguje komponenta Agent, ale v jiných akcích by uváděli postavy do defaultního stavu (akce **HarvestResource** při aktivaci *využije* nástroj, a tak ho při ukončení musí přestav *využívat*) (viz žlutý obdélník Obrázek 77).

```

4   <namespace GOAP
5   {
6       <Unity Script | 0 references
7       public class HealAction : Action
8       {
9           HealGoal healGoal;
10          List<ItemTags> healItemTags = new List<ItemTags>() { ItemTags.healing };
11          <Unity Message | 14 references
12          public override void Awake()
13          {
14              speachBubbleType = SpeachBubbleTypes.Heal;
15              base.Awake();
16          }
17
18          public override void Tick()
19          {
20              if (UseItem(healItemTags))
21                  Complete();
22              else
23                  Deactivate();
24          }
25          <17 references
26          public override void Activate(ActionData arg)
27          {
28              base.Activate(arg);
29          }
30
31          public override void Deactivate()
32          {
33              running = false;
34          }
35          <3 references
36          public override void Complete()
37          {
38              running = false;
39              completed = true;
40          }
41
42      <2 references

```

Obrázek 77

### Efekt akce na stav světa během plánování

Nejprve musíme implementovat metodu **IsAchievableGiven()**, kde vrátíme true, pouze pokud dává smysl tuto akci vůbec vykonávat, většinou zde kontrolujeme, zda je akce splnitelná, ale v našem případě se ptáme, zda máme cíl, který má tato akce za úkol splnit (viz červený obdélník Obrázek 78).

Poté, když se **Planner** rozhodne naši akci využít, tak musíme v metodě **OnActionCompleteWorldStates** vytvořit nový vrchol s modifikovaným stavem světa. Vracíme vždy množinu vrcholů, protože některé akce mohou modifikovat stav světa různými způsoby (např. **HarvestResource** může těžit zlato, dřevo nebo krystaly). Abychom mohli naši akci provést, tak potřebujeme mít předmět, který nás dokáže oživit (viz modrý obdélník Obrázek 78), a pokud ho nemáme, tak použijeme

pomocnou metodu `GetRequiredItemWithTags()`, která se pokusí najít pomocnou akci, která do našeho inventáře dostane předmět s danými tagy, tuto akci pak vloží do plánu a vrátí nám vrchol, který tuto akci využívá jako nového předchůdce. Pokud žádná taková akce nebyla nalezena, tak předmět získat nemůžeme a vrátíme prázdní seznam nástupců.

Pokud víme, že budeme mít předmět k dispozici, takže si uděláme shallow copy `WorldState` zavoláním konstruktoru na stav světa předka, hned na něj zavoláme `CopyCompltedGoals()` (viz žlutý obdélník Obrázek 78), která provede deep copy na `completedGoals` a změníme `completedGoals` přidáním komponenty `HealGoal`, která reprezentuje cíl, který touto akcí splníme. Nakonec vytvoříme nový vrchol, s parametry předchůdce, ceny (která není aktuálně v plánování využívaná) plánu, nového stavu světa, této akce a objekt `ActionData` (který je null, protože ho v této akci nevyužíváme). Vrchol přidáme do seznamu, který hned vrátíme.

```

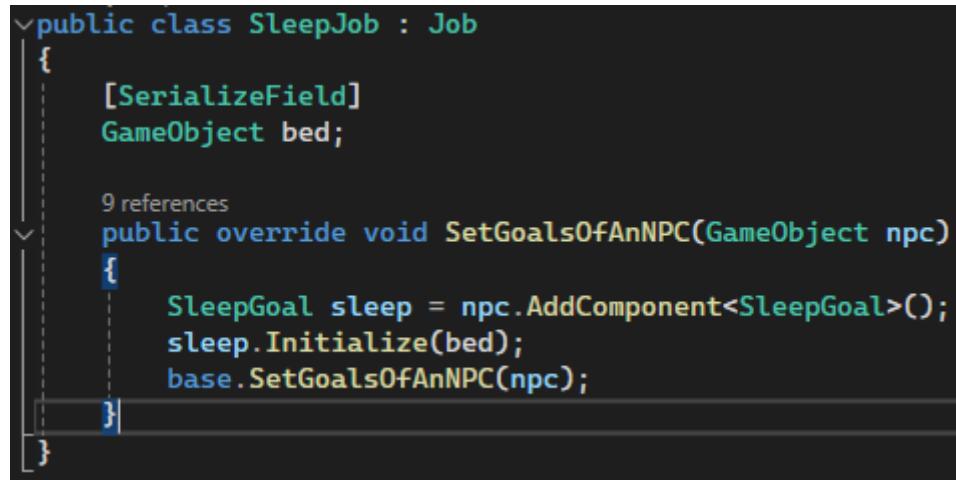
7  public override bool IsAchievableGiven(WorldState worldState)//For the planner
8  {
9      healGoal = GetComponent<HealGoal>();
10     if (healGoal == null) return false;
11     return true;
12 }
13
14 public override List<Node> OnActionCompleteWorldStates(Node parentOriginal)//Tells the plan
15 {
16     List<Node> possibleNodes = new List<Node>();
17     Node parent = parentOriginal;
18     if (!HasItem(parentOriginal.state, healItemTags))
19     {
20         //Debug.Log("Getting item");
21         parent = GetRequiredItemWithTags(parentOriginal, healItemTags);
22         if (parent == null)
23             return possibleNodes;
24     }
25     //else Debug.Log("Already have item");
26     WorldState possibleWorldState = new WorldState(parent.state);
27     possibleWorldState.CopyCompletedGoals();
28     possibleWorldState.completedGoals.Add(healGoal);
29
30
31     possibleNodes.Add(new Node(parent, 1 + parent.cost, possibleWorldState, this, null));
32     return possibleNodes;
33 }

```

Obrázek 78

### 7.6.3. Jak vytvořit nový Job

Pro vytvoření nové Job komponenty stačí dědit od komponenty `Job`, přepsat metodu `SetGoalsOfAnNPC()` a inicializovat komponenty cílů. Např. pro Job `SleepJob` (viz Obrázek 79) si pamatujeme referenci na postel, kde chceme, aby NPC spalo, a poté pouze přidáme danému NPC cíl `SleepGoal`, který inicializujeme danou postelí. Komponentu Job pak stačí připojit k vyvolávacímu kruhu, který Job zavolá na respawnuté NPC.



```
public class SleepJob : Job
{
    [SerializeField]
    GameObject bed;

    9 references
    public override void SetGoalsOfAnNPC(GameObject npc)
    {
        SleepGoal sleep = npc.AddComponent<SleepGoal>();
        sleep.Initialize(bed);
        base.SetGoalsOfAnNPC(npc);
    }
}
```

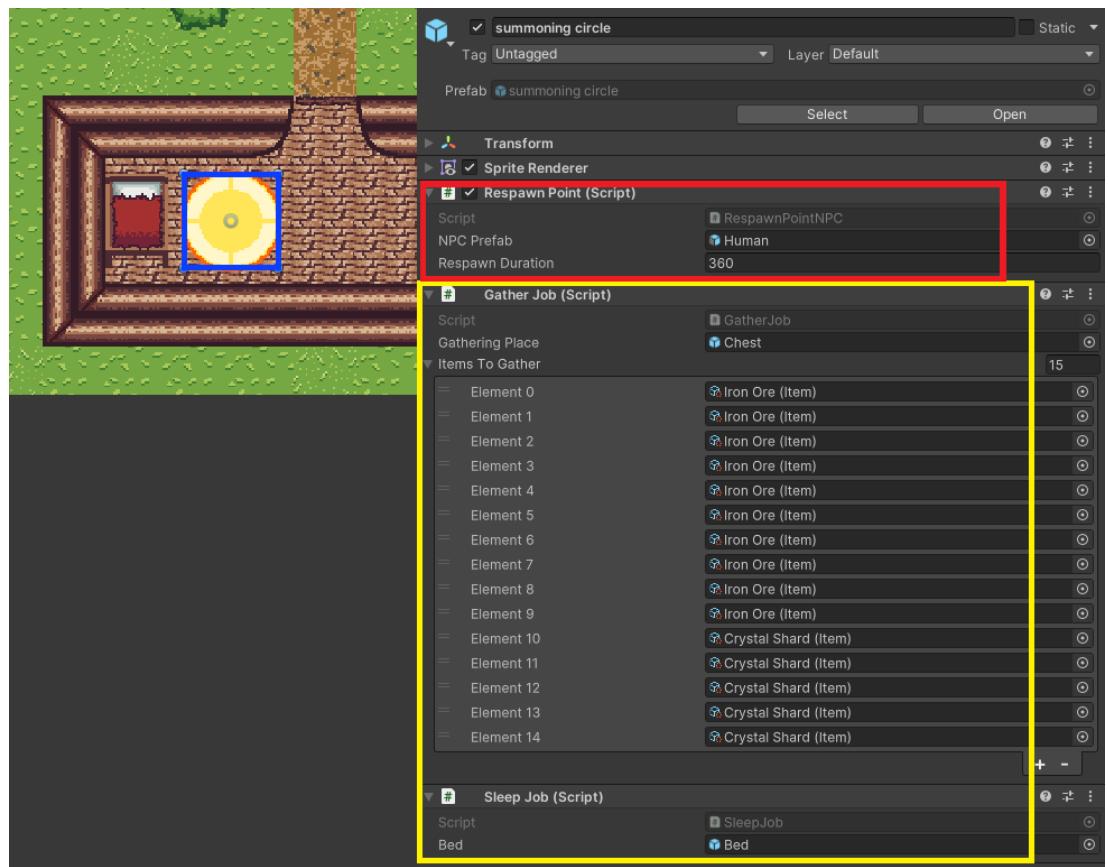
Obrázek 79  
Obrázek se source kódem komponenty SleepJob.

#### 7.6.4. Jak vytvořit nové NPC

Vytvoření nového NPC můžeme udělat dvěma způsoby, podle toho, zda chceme, aby bylo NPC schopno se respawnout. Pokud ne, tak stačí přetáhnout prefab NPC do herního světa, a připojit k instance cílů, které chceme, aby mělo.

Pokud chceme, aby bylo NPC se schopno respawnout, tak musíme použít vyvolávací kruh (viz modrý obdélník Obrázek 80), jehož prefab nalezneme v složce Assets/Resources/Prefabs/Objects. Komponenta RespawnPoint k prefabu připojená (viz červený obdélník Obrázek 80) je zodpovědná za respawnovací logiku a vyžaduje prefab NPC, které bude respawnovat. Aktuální prefaby NPC se liší pouze v druhu rasy, ale pokud bychom chtěli, tak můžeme vytvářet i unikátnější prefaby NPC. Dále vyžaduje čas, za kolik sekund po smrti se NPC respawne (první spawn nastane okamžitě).

Nyní stačí k instanci vyvolávacího kruhu připojit komponenty Jobs, které chceme, aby NPC vykonávalo a vyplnit jejich parametry (viz žlutý obdélník Obrázek 80), v tomto příkladu jsme vytvořili NPC horníka, který chce těžit železo a kouzelné krystaly.



Obrázek 80

# 8. Závěr

Nyní zhodnotíme, zda jsme naše cíle splnili a popíšeme několik nápadů na zlepšení našeho dema.

## 8.1. Zhodnocení splnění cílů

Na počátku této práce jsme si stanovili cíle (viz Cíle práce 2.7), nyní zhodnotíme, zda se nám je podařilo splnit.

- 1) RPG hru jsme úspěšně naprogramovali. Zaměření na předměty jsme zajistili tak, že se postavy mohou vylepšit získáním lepsího vybavení a že jsme se vyhnuli vylepšování postav přes leveley, zkušenosti, skill-pointy a jiné podobné systémy, které se v RPG tradičně používají.
- 2) Předměty jsme učinili persistentní implementací pick-upů, které reprezentují předmět v herním světě, který mohou postavy přesunout do svého inventáře. Když postavy umřou tak se z jejich vybavení a předmětů v jejich inventáři stanou pick-upy.
- 3) Systém těžení surovin jsme implementovali pomocí bojového systému, ložiska surovin mají životy a když „umřou“, tak zmizí a instancují se pick-upy surovin. Vyrábění předmětů jsme implementovali úspěšně použitím vyráběcích objektů, které postavy používají k výrobě předmětů.
- 4) NPC jsou schopna těžení, vyrábění a prodávání předmětů použitím stejných herních mechanik, které používají hráči. Výběr akcí, plánování a prioritizaci cílů jsme implementovali pomocí GOAP architektury.
- 5) Používání předmětů a vybavení a zúčastňování se bojů jsme implementovali jako součást stejné GOAP architektury, kterou jsme splnili cíl 4).

Všechny cíle jsme splnily a považujeme demo a naši práci za úspěšnou. Měli bychom ale nápady pro budoucí vývoj hry, které si rozmyslíme v další sekci.

## 8.2. Potenciální budoucí vylepšení

Naše demo slouží jako demonstrace základních herních mechanik a v této sekci poukážeme, jak bychom demo mohli vylepšit a rozšířit:

- 1) Nejlepší vylepšení naší hry by bylo přidání zvukových efektů a hudby, které zatím ve hře nejsou.
- 2) Menu na změnu herních možností, jako rozlišení a hlasitost zvuku.
- 3) Přidání vícera druhů map/levelů, a menu na jejich výběr před začátkem hry.
- 4) Implementovat více předmětů.
- 5) Přidat více tváří pro každou rasu, pro větší estetickou rozmanitost postav.

# Seznam použité literatury

## Zdroj 1

Aron Granberg, A\*Pathfinding Projekt, naposledy navštíveno: 15.7. 2024, URL:  
<https://arongranberg.com/astar/>

## Zdroj 2

The Settlers 2: Veni, Vidi, Vici, naposledy navštíveno: 17.10. 2024, URL:  
<https://rawg.io/games/the-settlers-2-veni-vidi-vici>

## Zdroj 3

- 1) Markdom3D, What is Level of Detail (LOD), naposledy navštíveno: 30.10. 2024, URL: <https://www.youtube.com/watch?v=pzuJztVAeQ0>,
- 2) level of detail (computer graphics), naposledy navštíveno: 30.10. 2024, URL: [https://en.wikipedia.org/wiki/Level\\_of\\_detail\\_\(computer\\_graphics\)](https://en.wikipedia.org/wiki/Level_of_detail_(computer_graphics))

## Zdroj 4

Quest, naposledy navštíveno: 30.10. 2024, URL:  
<https://cs.wikipedia.org/wiki/Quest>

## Zdroj 5

Dragonfly, naposledy navštíveno: 6.11. 2024, URL:  
<https://www.dragonflydb.io/game-dev/engines/2d>

## Zdroj 6

NavMeshPlus, naposledy navštíveno: 14.4. 2025, URL:  
<https://github.com/h8man/NavMeshPlus>

## Zdroj 7

Understanding Steering Behaviors, naposledy navštíveno: 17.4. 2025, URL:  
<https://code.tutsplus.com/series/understanding-steering-behaviors--gamedev-12732>

## Zdroj 8

Game Engineering 1: State-based AI, AI – Finite State Machines pdf, naposledy navštíveno: 5.5. 2025, URL:  
<https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/aitutorials/1state-basedai/>

## Zdroj 9

Robohub, Introduction to behaviour trees, naposledy navštíveno: 5.5. 2025, URL:  
<https://robohub.org/introduction-to-behavior-trees/>

## Zdroj 10

allaboutai, What is Forward and Backwards Search, naposledy navštíveno: 5.5. 2025, URL:  
<https://www.allaboutai.com/ai-glossary/forward-backward-search/>

## Zdroj 11

Gameopedia, An Introduction to Open World Games, naposledy navštíveno: 5.5. 2025, URL:

<https://gameopedia.com/blogs/an-introduction-to-open-world-games>

**Zdroj 12**

Svět her, Stronghold, druhý obrázek, naposledy navštíveno: 12.5. 2025, URL:  
<https://www.svet-deskovych-her.cz/produkty/818/stronghold>

**Zdroj 13**

Zatrolené hry, Čertův poklad, druhý obrázek, naposledy navštíveno: 12.5. 2025, URL:

<https://www.zatrolene-hry.cz/spolecenska-hra/certuv-poklad-3692/obrazky/>

**Zdroj 14**

Youtube, ZachScottGames, screenshoty z videa **Inkling Unlocked! - Super Smash Bros Ultimate - Gameplay Walkthrough Part 2 (Nintendo Switch)**, naposledy navštíveno: 1.7. 2025, URL:

<https://www.youtube.com/watch?v=HF16nrZFqRI>

**Zdroj 15**

Youtube, MrAndMrsDoGaming, screenshoty z videa **Moorhuhn Kart 4 (2025) Split Screen Steam PC Demo**, naposledy navštíveno: 2.7. 2025, URL:

<https://www.youtube.com/watch?v=CgdW4DBOerI>

## **Seznam použitých zkratek**

### **Z 1**

**NPC** (Non Player Character) je postava vyskytující se ve hře, která není ovládána hráčem. Může být statická nebo ovládána UI a může plnit řadu herních účelů, jako např. bránění hráči pokroku hrou, pomáhat hráči, nebo navádět atmosféru.

### **Z 2**

**AI**, zkratka pro Artificial Intelligence neboli Umělá Inteligence

### **Z 3**

**RPG** (role playing game) definice: [https://en.wikipedia.org/wiki/Role-playing\\_game](https://en.wikipedia.org/wiki/Role-playing_game)

### **Z 4**

**Colony Simulator** je žánr her ve kterých hráč řídí kolonii nebo jiné malé a mladé osídlení. Tyto hry jsou většinou soustředěny na malou populaci jedinců, jejich zdraví a spokojenost. Hráč se snaží rozrůstat svoji kolonii a uspokojit její jedince těžením a využíváním přírodních zdrojů. Zdroj: Glitchwave, naposledy navštíveno: 4.7. 2025, URL: <https://glitchwave.com/games/genre/colony-sim/>

### **Z 5**

**City Builder** je žánr her, ve kterých hráč řídí město. Tento žánr se na rozdíl od colony sim žánru soustředí spíše na prosperitu města a blahobyt její populace obecně než na blahobity specifických jedinců, v některých hrách tohoto žánru se ale hráč stále musí soustředit na strategické využívání přírodních zdrojů. Zdroj: Wikipedia, naposledy navštíveno: 4.7. 2025, URL: [https://en.wikipedia.org/wiki/City-building\\_game](https://en.wikipedia.org/wiki/City-building_game)

### **Z 6**

**UI** je zkratka pro user interface neboli uživatelské rozhraní.

**GUI** je zkratka pro graphical user interface neboli grafické uživatelské rozhraní.

**[Vzor: Přílohy k bakalářské práci, existují-li (různé dodatky jako výpisy programů, diagramy apod.). Každá příloha musí být alespoň jednou odkazována z vlastního textu práce. Přílohy se číslují.]**

[Do tištěné verze se spíše hodí přílohy, které lze číst a prohlížet (dodatečné tabulky a grafy, různé textové doplňky, ukázky výstupů z počítačových programů apod.). Do elektronické verze se hodí přílohy, které budou spíše používány v elektronické podobě než čteny (zdrojové kódy programů, datové soubory, interaktivní grafy apod.). Elektronické přílohy se nahrávají do SISu. Povolené formáty souborů specifikuje opatření rektora č. 72/2017. Výjimky schvaluje fakultní koordinátor pro závěrečné práce.]

## **Přílohy**