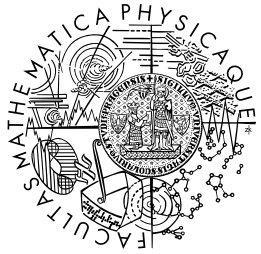


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Petr Kotáb

Možnosti návrhového vzoru Entity-Component-System: případová studie

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D.
Studijní program: Informatika

Praha 2024

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Chtěl bych především poděkovat mému vedoucímu, Mgr. Pavlu Ježkovi, Ph.D., za čas, připomínky a rady, které mi věnoval, bez nichž by tato práce nemohla vzniknout. Také děkuji svému původnímu vedoucímu, Mgr. Jakubovi Gemrotovi, Ph.D., za čas a konzultace. Dále bych chtěl poděkovat svým přátelům a rodině za jejich velkou podporu.

Název práce: Možnosti návrhového vzoru Entity-Component-System: případová studie

Autor: Petr Kotáb

Katedra: Katedra softwaru a výuky informatiky

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

Abstrakt: Práce se zabývá měřením výkonu ECS knihoven pro programovací jazyk C#. Na rozdíl od často prováděných jednoduchých testů je cílem této práce provést komplexnější měření na ukázkové hře. Výsledkem práce je ukázková neinteraktivní hra simulující vesničany těžící suroviny v otevřeném světě. Pro umožnění měření výkonu ECS knihoven byla připravena abstrakční vrstva, která umožňuje jednu implementaci hry spouštět nad různými ECS knihovnami. Před samotným měřením jsme jednotlivé ECS knihovny rozdělili do kategorií a pro tyto kategorie stanovili hypotézu stran očekávané výkonnosti. Na závěr jsme provedli sadu měření, kterými se nám podařilo naši hypotézu potvrdit.

Klíčová slova: návrhový vzor Entity-Component-System počítačové hry simulace případová studie

Title: Exploring Options of Entity-Component-System Design Pattern: A Case Study

Author: Petr Kotáb

Department: Department of software and computer science education

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

Abstract: The thesis focuses on measuring the performance of ECS libraries for the C# programming language. Unlike the often conducted simple tests, the goal of this thesis is to perform more complex measurements on a sample game. The result of this thesis is a sample non-interactive game simulating villagers harvesting resources in an open world. To enable the measurement of ECS libraries performance, an abstraction layer was prepared, allowing a single game implementation to launch with different ECS libraries. Before the actual measurement, we categorized the individual ECS libraries and formulated a hypothesis regarding the expected performance for these categories. In the end, we conducted a series of measurements that allowed us to confirm our hypothesis.

Keywords: design pattern Entity-Component-System computer games simulation case study

Obsah

1 Úvod **4**
- 1.1 Entity-Component-System . . . 4
- 1.2 ECS knihovny . . . 5
- 1.3 Abstrakční vrstva . . . 6
- 1.4 Cíle práce . . . 6

2 Hra **7**
- 2.1 Herní požadavky . . . 7
  - 2.1.1 Charakteristiky hry . . . 7
  - 2.1.2 Simulace . . . 8
- 2.2 Herní specifikace . . . 9
  - 2.2.1 Herní svět . . . 9
  - 2.2.2 Vesnice . . . 10

3 Analýza **12**
- 3.1 Volba frameworku/enginu pro hru . . . 12
- 3.2 Abstrakční vrstva . . . 13
  - 3.2.1 Rozhraní ECS knihoven . . . 13
  - 3.2.2 Inlinování funkce . . . 16
  - 3.2.3 Reprezentace systémů . . . 16
  - 3.2.4 ECSFactory . . . 17
  - 3.2.5 Komponenty jako třídy a komponenty jako struktury . . . 18
- 3.3 Umělá inteligence . . . 19
- 3.4 Generovaný svět . . . 23
  - 3.4.1 Generování terénu . . . 23
  - 3.4.2 Generování terénu na CPU vs. GPU . . . 25
  - 3.4.3 Navigace ve vygenerovaném terénu . . . 26
- 3.5 Měření . . . 30
  - 3.5.1 Jak budeme měření provádět . . . 31
  - 3.5.2 Volba frameworku pro měření . . . 32

4 Popis implementace **33**
4.1 Abstrakční vrstva . . . 33
4.1.1 Systémy . . . 33
4.1.2 ECSFactory . . . 34
4.2 Hra . . . 35
4.2.1 Herní stavy . . . 35
4.2.2 LevelState a herní svět . . . 36
4.2.3 Assety . . . 37
4.2.4 Správa managed typů . . . 38
4.2.5 Uživatelské rozhraní . . . 40
4.2.6 Herní data . . . 41
4.2.7 Inventáře a předměty . . . 42
4.2.8 Suroviny . . . 43
4.2.9 Vesnice a vesničané . . . 45
4.3 Měření . . . 48

5 Měření **49**
5.1 Spouštění měření . . . 49
5.2 Hypotéza . . . 50
5.2.1 Cache . . . 50
5.2.2 Arch type . . . 50
5.2.3 Stanovení hypotézy . . . 51
5.3 Měřené knihovny . . . 52
5.4 Výsledky . . . 53
5.4.1 První měření . . . 54
5.4.2 Druhé měření . . . 55
5.4.3 Třetí měření . . . 56
5.4.4 Velikosti světa . . . 57
5.4.5 Závěr měření . . . 58
5.4.6 Závěrečná pozorování k vybraným ECS knihovnám . . . 58

6 Závěr **59**
6.1 Cíle . . . 59
6.1.1 Splnění cíle 1 . . . 59
6.1.2 Splnění cíle 2 . . . 60
6.1.3 Nedostatky řešení . . . 60
6.2 Možná budoucí vylepšení . . . 61
6.2.1 Možná vylepšení hry . . . 61
6.2.2 Možná vylepšení měření . . . 62

Seznam použité literatury **63**

Přílohy **66**

Kapitola 1

Úvod

Vytváření her je komplexní proces, který zahrnuje mnoho fází, včetně prototypování a iterací. Často se ocitneme v situaci, kdy je nezbytné výrazně upravit již existující kód. Z tohoto důvodu je klíčová vysoká flexibilita architektury kódu.

Jedním z přístupů k řešení tohoto problému je využití návrhového vzoru Component, který poskytuje vysokou flexibilitu. Tento návrhový vzor je běžně používán v herních frameworcích a enginech, jako je například Unity [1] (koncept MonoBehaviour) nebo MonoGame [2] (koncept GameComponent). Návrhový vzor Component je dobře představen v knize *Game Programming Patterns* [3].

Nicméně existují pokročilejší přístupy, které mohou nabídnout ještě lepší flexibilitu a výkon. Jedním takovým je návrhový vzor Entity-Component-System (ECS). I přesto, že se v současné době nepoužívá tak často jako návrhový vzor Component, nabízí značné výhody při tvorbě her. Proto se na něj v této práci zaměříme.

1.1 Entity-Component-System

ECS je návrhový vzor používaný pro vývoj her. Jeho základními stavebními kameny jsou entity, komponenty a systémy.

1. **Entita (Entity):** Reprezentuje objekt v herní scéně, může být cokoliv od hráče po strom nebo zvíře.

2. **Komponenta (Component):** Entita je složena z komponent, které ji popisují. Komponenty obsahují pouze data a žádnou herní logiku. Například hráč může mít komponentu `Control`, která popisuje, jak se ovládá, strom může mít komponentu `Resource`, která popisuje, kolik dřeva hráč získá, pokud jej pokácí, a zvíře může mít komponentu `PathFollow`, která popisuje cestu, po které se pohybuje. Všechny tři entity mohou mít komponenty

`Appearance`, která popisuje, jak daný objekt vypadá, a `Position`, popisující, kde se daný objekt nachází v herní scéně. Dále hráč a zvíře mohou mít komponentu `Movement`, popisující jejich pohyb.

3. **Systém (System):** Systém obsahuje herní logiku. Systémy pracují nad určitou množinou komponent. V každé iteraci vezmou všechny entity obsahující tuto množinu komponent a provedou nad nimi určité operace. Například:

   - **`InputSystem`:** Pracuje nad entitami s komponentami `Control` a `Position`, řídí logiku jejich ovládání.
   - **`PathMovementSystem`:** Pracuje nad entitami s komponentami `PathFollow` a `Movement`, řídí logiku jejich pohybu po cestě.
   - **`RenderSystem`:** Pracuje nad entitami s komponentou `Appearance` a řídí logiku jejich vykreslení.

1.2 ECS knihovny

Aby bylo možné efektivně využívat návrhový vzor ECS, je nezbytné implementovat odpovídající infrastrukturu. Samotná implementace této infrastruktury představuje obtížnou a časově náročnou úlohu, což vede mnoho vývojářů k volbě již existujících knihoven implementujících infrastrukturu pro tento návrhový vzor.

V této práci se zaměříme na knihovny pro programovací jazyk C#, neboť se jedná o programovací jazyk běžně používaný pro hry a aplikace. I když se omezíme pouze na C#, existuje velké množství těchto knihoven.

Jedním ze způsobů, jak si vývojáři mohou vybrat ECS knihovnu, je použít porovnání. *CSharpECSComparison* [4] je projekt na platformě GitHub, který porovnává populární ECS knihovny pro C# na základě funkcionalit, vlastností a závislostí. Vývojář by si tedy mohl říci, které vlastnosti a funkcionality by od takové knihovny požadoval a na základě toho si vybrat. Ovšem možnosti, které nám jednotlivé ECS knihovny nabízejí, se často příliš neliší, proto dává smysl porovnávat jednotlivé knihovny na základě výkonu. Bohužel *CSharpECSComparison* nám neposkytuje žádné informace o výkonu jednotlivých knihoven.

Ecs.CSharp.Benchmark [5] je projekt na platformě GitHub, který provádí jednoduché výkonnostní testy ECS knihoven pro C#. Výsledky těchto testů mohou být užitečné při výběru knihovny na základě výkonu. Problém však může spočívat v tom, že tyto testy jsou příliš jednoduché. Každý test, který *Ecs.CSharp.Benchmark* provádí, pracuje pouze s velmi jednoduchými komponentami, kde každá komponenta obsahuje pouze jedno celé číslo. Testy jsou rozděleny do několika sad.

První sada vytváří entity s těmito komponentami. Druhá sada spouští jednoduché systémy, které pouze sčítají výše zmíněná celá čísla na těchto komponentách.

Jelikož jsou testy příliš jednoduché, v případě rozsáhlejšího projektu, jako je hra, by se výsledky mohly lišit. Jednotlivé entity mohou mít velké množství komponent, z nichž každá může být složitější. Systémy mohou být také mnohem komplexnější a provádět mnohem složitější úkoly než pouhé sčítání celých čísel.

Jedním ze způsobů, jak by bylo možné lépe porovnat výkon jednotlivých ECS knihoven, by bylo vytvoření hry, na které by bylo možné měřit výkon jednotlivých knihoven. Taková hra by musela být nezávislá na konkrétní ECS knihovně. Měla by také obsahovat netriviálně velký svět s větším počtem entit. To by nám umožnilo lépe provádět měření. Jedním z cílů této práce bude vytvoření takové hry.

Poté, co budeme mít tuto hru k dispozici, bude možné vytvořit výkonnostní test, který by dokázal změřit výkon příslušné ECS knihovny na této hře. Pomocí tohoto testu bude možné udělat výkonnostní porovnání jednotlivých ECS knihoven. Vytvoření tohoto porovnání bude dalším cílem této práce.

1.3 Abstrakční vrstva

Vzhledem k tomu, že plánovanou hru budeme chtít využít k porovnávání různých ECS knihoven, je nezbytné navrhnout ji tak, aby fungovala nezávisle na konkrétní ECS knihovně. S ohledem na odlišné API jednotlivých ECS knihoven je nezbytné vytvořit abstrakční vrstvu pro ECS. Pro každou knihovnu, kterou budeme chtít porovnávat, poskytneme implementaci této vrstvy. Samotná hra poté pracuje s touto abstrakční vrstvou.

Vytvoření takovéto vrstvy představuje netriviální úkol, navzdory podobnosti rozhraní jednotlivých ECS knihoven. Významný rozdíl spočívá v rozhraní systémů, které se výrazně liší mezi jednotlivými knihovnami.

1.4 Cíle práce

Pro shrnutí, tato práce bude mít následující cíle:

1) Vytvořit hru pomocí ECS, která bude nezávislá na konkrétní ECS knihovně.

2) Pomocí vytvořené hry porovnat relativní výkon populárních ECS knihoven pro programovací jazyk C#.

Kapitola 2

Hra

V této kapitole si specifikujeme, jak bude vypadat naše hra. Nejprve si rozebereme, jaké vlastnosti od ní vyžadujeme, a následně provedeme samotnou specifikaci.

2.1 Herní požadavky

Nyní si upřesníme požadavky co chceme od naší hry, následně také zvolíme herní žánr.

Než budeme pokračovat, připomeňme si prvně, že naší hru chceme použít pro měření výkonu ECS knihoven. Budeme tedy chtít aby byla nezávislá na konkrétní ECS knihovně. Konkrétní ECS knihovnu bude možné zvolit před spuštěním hry.

2.1.1 Charakteristiky hry

Aby nám naše hra napomohla k splnění práce, budeme chtít, aby splňovala následující charakteristiky:

1. **Nezávislost na konkrétní ECS knihovně:** Nezávislost na konkrétní ECS knihovně pro nás bude důležitá, abychom hru mohli použít pro měření výkonu jednotlivých knihoven. Hra bude namísto konkrétní ECS knihovny pracovat pouze s abstrakční vrstvou. Bude pro nás žádoucí, aby se výkon naší hry běžící s konkrétní ECS knihovnou implementující abstrakční vrstvu, blížil k výkonu hry, která by používala pouze konkrétní ECS knihovnu bez abstrakční vrstvy.

2. **Hra by neměla využívat velmi častého přidávání a odebírání komponent:** Přidání a odebrání komponenty představuje pro některé ECS knihovny náročnější operaci. Ovšem to jestli hra využívá velmi častého přidávání a odebírání komponent, je závislé z velké části na implementaci.

Kdyby naše hra používala velmi častého přidávání a odebírání komponent, došlo by k znevýhodnění některých ECS knihoven, které by jinak na tom mohly být výkonnostně lépe. Ovšem je nutné zmínit, že odebírání komponent v některých případech nelze předejít. Typickým příkladem bývá odebírání entity například v případě její smrti.

Ukažme si na příkladu, proč je velmi časté přidávání a odebírání komponent závislé z velké části na implementaci. Konkrétně bychom chtěli reprezentovat efekt toho, že entita hladoví. Pokud by nám velmi časté přidávání a odebírání komponent nevadilo, tak by jeden z typických přístupů byl vytvořit `Starvation` komponentu a přidat ji na každou entitu, která začne hladovět. V případě, že se příslušná entita nasytí jídlem, tak přestane hladovět a dojde k odebrání této komponenty. Pokud bychom chtěli stejnou mechaniku navrhnout bez častého přidávání a odebírání komponent, tak jeden z možných způsobů by byl dát každé entitě, která může hladovět, `Hunger` komponentu, ve které by byl flag `Starvation`, který by rozhodoval o tom, jestli daná entita hladoví či nikoliv.

3. **Velký počet entit:** Velký počet entit nám usnadní měření výkonu naší hry. Také nám zaručí, že systémy budou vykonávat netriviální množství práce. Zároveň si tím vyzkoušíme velký výkon, kterým ECS disponuje. Velký počet entit nám také implikuje větší herní svět.

4. **Adekvátní složitost hry:** Bude pro nás důležité, aby naše hra nebyla příliš jednoduchá. Budeme se chtít vyhnout tomu, aby naše měření nebylo jenom dalším jednoduchým měřením výkonu. Ovšem taky si musíme dát pozor na to, aby nebyla příliš složitá, jelikož to by odvádělo pozornost od problému, který řešíme.

2.1.2 Simulace

Žánrem naší hry bude simulace, ale řekněme si nejprve, co to vlastně je. Simulace je žánr videoher, který se snaží simulovat agenty v jejich prostředí. U nás agenti budou vesničané a jejich prostředí bude příroda.

Proč jsme zvolili právě simulaci? Pro simulaci je jednodušší (oproti ostatním žánrům) vymyslet herní design, který má velký počet entit a netriviálně velký svět. Další velkou výhodou je malí vliv uživatele na průběh hry, to nám umožní lépe provádět měření. Konkrétně v naší hře bude hráč spíše pozorovatel, který sleduje jak simulace probíhá.

2.2 Herní specifikace

Nyní si představíme specifikaci naší hry. Celá hra se dá rozdělit na dva hlavní prvky. Konkrétně svět a vesnice. Specifikujeme si tedy každý z nich.

Prvně než si začneme specifikovat jednotlivé prvky, je nutné upřesnit, že se jedná o 2D hru. Hráč v této hře zastává roli pozorovatele a celý svět pozoruje skrze kameru pohledem shora. Ovládá pouze pozici a přiblížení této kamery.

2.2.1 Herní svět

Herní svět je náhodně generovaný s pevnou velikostí a skládá se z několika biomů. Každý biom je reprezentován jinou barvou a nacházejí se v něm odlišné suroviny. Zároveň pro každý biom je určeno, zda po něm mohou vesničané chodit nebo stavět. Herní svět obsahuje následující biomy:

1. *Voda* (`Water`): Reprezentuje vodní plochu. Není na ní možné chodit ani stavět.
2. *Pláž* (`Beach`): Jedná se o písečné oblasti v okolí *vody*. Je na ní možné chodit i stavět.
3. *Louka* (`Plain`): Na *louce* je možné chodit i stavět. Je na ni možné najít *jeleny*.
4. *Les* (`Forest`): V *lese* je možné chodit i stavět. Nachází se v něm velké množství *stromů*.
5. *Hora* (`Mountain`): Jedná se o kamenitou *horu*. Je možné na ní chodit i stavět. Nachází se na ní *naleziště kamene*.
6. *Vysoká hora* (`High Mountain`): *Vysoká hora* je vždy obklopena *horou*. Je pokryta sněhovou plochou. Je možné na ní chodit, ale není možné na ní stavět. Nachází se na ní *naleziště železné rudy*.

Jak již bylo zmíněno, v jednotlivých biomech se mohou nacházet suroviny. Suroviny se rozdělují do několika typů. Typ suroviny udává, jak dlouho trvá ji sklidit, jaký předmět lze získat po jejím sklizení, a počet těchto předmětů, které lze získat po jejím sklizení. Hra obsahuje následující suroviny:

1. *Strom* (`Tree`): Po pokácení *stromu* lze získat předmět *dřeva*. *Stromy* se nacházejí ve velkém množství v biomu *lesa*.
2. *Naleziště kamene* (`Rock`): Sklizením *naleziště kamene* je možné získat předmět *kamene*. *Naleziště kamene* se nacházejí v biomu *hory*.

3. *Naleziště železné rudy* (`Deposit`): Po sklizení *naleziště železné rudy* je možné získat předmět *železná ruda*. *Naleziště železné rudy* se objevuje v biomu *vysoká hora*.

4. *Jelen* (`Deer`): *Jeleni*, na rozdíl od jiných typů surovin, umí chodit. Náhodně se pohybují po biomu *louky*, ve které se také objevují. Po sklizení jelena je možné získat předmět *masa*.

Předměty získané skrze sklizení suroviny je dále možné zpracovat v příslušné budově. Každý předmět se může zpracovávat jinak dlouho. Hra obsahuje následující předměty:

1. *Dřevo* (`Wood`): *Dřevo* se dá dále zpracovat na *prkno*.

2. *Kámen* (`Stone`): *Kámen* je možné přepracovat na *cihlu*.

3. *Železná ruda* (`Ore`): *Železnou rudu* lze vypálit a tím získat *železo*.

4. *Maso* (`Meat`): *Maso* je možné uvařit a tím lze získat *jídlo*.

5. *Prkno* (`Plank`): *Prkna* je možné použít pro stavbu budov.

6. *Cihla* (`Brick`): *Cihlu* je také možné použít pro stavbu budov. Jedná se o lehce pokročilejší surovinu oproti *prknu*.

7. *Železo* (`Iron`): *Železo*, stejně jako *Prkno* a *Cihlu*, lze také použít pro stavbu budov. Jedná se o pokročilou surovinu.

8. *Jídlo* (`Food`): *Jídlo* je předmět, který musí vesničané požívat, aby neumřeli hlady.

2.2.2 Vesnice

V herním světě se náhodně nacházejí vesnice. V každé se nachází několik budov a vesničané. Každá vesnice má přehled o předmětech, kterými disponuje. Pro všechny vesnice je také pevně stanovena stavební fronta, kde pro každou budovu v této frontě je daná cena. Jakmile vesnice nasbírá dostatek surovin pro další budovu z této fronty, dojde k odečtení těchto surovin a následně se objeví příslušná budova. Budova se nemůže objevit moc blízko ani moc daleko od budov, které se již ve vesnici nacházejí.

Každý vesničan má svoje *pracovní místo* (`WorkPlace`). Vesničané vykonávají práci na základě své profese. Profese vesničanovi udává, jaký předmět má pro vesnici obstarávat. Jeho práce spočívá v hledání a sklizení příslušné suroviny, kterou následně odnese do svého *pracovního místa*, kde ji zpracuje a výsledný

předmět odnese do *skladiště*. Každý vesničan má právě jednu z následujících profesí:

1. *Dřevorubec* (`Woodcutter`): Obstarává *prkna*.
2. *Horník kamene* (`StoneMiner`): Obstarává *cihly*.
3. *Horník železa* (`IronMiner`): Obstarává *železo*.
4. *Lovec* (`Hunter`): Obstarává *jídlo*.

Každý vesničan má také hlad. Pokud hlad vesničana bude příliš velký, tak vesničan zemře. Pokud hlad vesničana stoupne nad určitou hodnotu a ve skladišti dané vesnice se bude nacházet *jídlo*, tak vesničan dojde ke skladišti a hlad si sníží snědením jídla.

Všechny vesnice začínají s *hlavní budovou*, *skladištěm* a *budovou dřevorubce*. Budovy lze rozdělit do dvou kategorií, a to speciální a budovy pro zpracování předmětů. V budovách pro zpracování předmětů je možné přepracovat předměty na jiné předměty a zároveň slouží jako *pracovní místo* pro vesničany. Vždy, když je postavena nová budova pro zpracování předmětů, objeví se nový vesničan, kterému bude přidělena příslušná profese podle této budovy. Speciální budovy slouží k speciálním účelům a patří mezi ně všechny ostatní budovy. V každé vesnici se mohou nacházet následující typy budov:

1. *Hlavní budova* (`MainBuilding`): Jedná se o speciální budovu, která reprezentuje danou vesnici. Každá vesnice má právě jednu hlavní budovu. Nemá žádný jiný zvláštní účel.
2. *Skladiště* (`Stockpile`): Jedná se o speciální budovu. Každá vesnice začíná s právě jedním skladištěm a nemůže jich mít více. Ve skladišti je možné skladovat jednotlivé předměty, kterými vesnice disponuje.
3. *Budova dřevorubce* (`WoodcutterHut`): Budova pro zpracování předmětů, konkrétně *prken*. Slouží jako pracovní místo pro *dřevorubce*.
4. *Budova horníka* (`MinerHut`): Budova pro zpracování předmětů, konkrétně *cihel*. Slouží jako pracovní místo pro *horníka kamene*.
5. *Kovárna* (`Smithy`): Budova pro zpracování předmětů, konkrétně *železa*. Slouží jako pracovní místo pro *horníka železa*.
6. *Budova lovce* (`HunterHut`): Budova pro zpracování předmětů, konkrétně *jídla*. Slouží jako pracovní místo pro *lovce*.

Kapitola 3

Analýza

V této kapitole prozkoumáme, jaká rozhodnutí byla provedena během implementace a odůvodníme jejich zvolení.

3.1 Volba frameworku/enginu pro hru

Chceme vytvořit hru a s její pomocí porovnat výkon ECS knihoven. Implementace se tedy bude skládat ze dvou částí a to samotné hry a měření, které pomocí naší hry bude měřit výkon jednotlivých ECS knihoven.

Nyní provedeme volbu enginu/frameworku, který použijeme pro naši hru. Pro tuto volbu si nejprve rozeberme požadavky, které od enginu/frameworku chceme:

1. **Podpora pro programovací jazyk C#:** V sekci 1.2 jsme si uvedli, že se zaměříme na knihovny pro programovací jazyk C#. Proto tento programovací jazyk budeme chtít použít pro naši hru.

2. **Podpora pro 2D hry s pohledem shora:** Tento požadavek vyplývá ze specifikace hry, konkrétně ze sekce 2.2.

3. **Možnost spouštět hru vícekrát po sobě a z jiného C# projektu:** Kromě naší hry budeme mít také měření, které pomocí naší hry změří výkon různých ECS knihoven. Jak naše hra, tak i naše měření budou samostatné C# projekty. Od frameworku/enginu budeme vyžadovat, aby nám umožnil spouštět hru vícekrát po sobě s různými implementacemi abstrakční vrstvy z našeho měření, které je v jiném C# projektu než naše hra.

V přehledu [6] technologií používaných hrami v herním obchodě Steam [7] je možné nahlédnout, že mezi nejpopulárnější herní enginy/frameworky pro C# patří Unity [1], Godot [8] a Microsoft XNA Framework.

Unity a Godot jsou velké herní enginy s vlastním editorem. Oba dva podporují 2D hry s pohledem shora. Ovšem tyto enginy odstiňují uživatele od některých nízkoúrovňových částí implementace a přebírají kontrolu nad C# projektem. Bylo by pro nás problematické s nimi splnit třetí požadavek.

Microsoft XNA Framework je pouze knihovna tříd bez vlastního editoru. Tento framework je vhodný pro 2D hry s pohledem shora a oproti herním enginům Unity a Godot tento framework volí nízkoúrovňový přístup, díky kterému uživatel není odstíněn od nízkoúrovňových částí implementace a má plnou kontrolu nad projektem. Tím pádem tento framework splňuje všechny naše požadavky.

Ačkoliv by tento framework byl vhodným kandidátem, bohužel již není delší dobou firmou Microsoft podporován. Ale jelikož se jednalo o velice populární herní framework, tak po skončení jeho podpory vzniklo několik open source reimplementací, které v podpoře a vývoji tohoto herního frameworku pokračují. Mezi nejpopulárnější z nich patří FNA [9] a MonoGame [2]. Oba tyto frameworky mají téměř identické API, my pro implementaci naší hry zvolíme MonoGame z důvodů osobní preference autora.

3.2 Abstrakční vrstva

Abstrakční vrstva pro nás je důležitá, jelikož chceme, aby naše hra byla nezávislá na konkrétní ECS knihovně. Program jako takový bude pracovat namísto ECS knihovny pouze s touto abstrakční vrstvou. Pro každou ECS knihovnu, kterou budeme měřit bude nutné vytvořit implementaci této vrstvy. K volbě konkrétní ECS knihovny dojde před spuštěním programu.

3.2.1 Rozhraní ECS knihoven

Pro návrh abstrakční vrstvy bude nejprve nutné rozebrat, jak jsou jednotlivé prvky ECS reprezentovány při jejich implementaci. Krom základních členů, jako jsou *entity*, *komponenty* a *systémy*, si představíme ještě *world* a *query*.

1. ***Component*:** *Komponenty*, jako takové, bývají často reprezentovány třídou nebo strukturou. Jak již bylo zmíněno v úvodu, *komponenty* obsahují pouze data, nikoliv žádnou logiku. Proto tyto třídy a struktury neobsahují žádné metody.

2. ***Entity*:** *Entita* by, podobně jako v návrhovém vzoru Component, mohla být reprezentována jako kolekce svých *komponent*. Ovšem to se v praxi nedělá, namísto toho se používá řešení, které vede k lepšímu výkonu. Většina ECS implementací reprezentuje *entity* pouze jako jednoduchý identifikátor a jednotlivé *komponenty* jsou na *entity* mapovány skrze *world*.

3. ***World*:** *World* je spravce *entit* a *komponent*. Tento správce slouží jako kontejner pro všechny *entity* a *komponenty* v herním světě. Jeho rozhraní nabízí funkce, pomocí niž je možné vytvářet a mazat jednotlivé *entity*. Často obsahuje také funkce pro přidávání a odebírání *komponent* jednotlivým *entitám*.

   Některé ECS implementace reprezentují *entity* pomocí malé struktury, obsahující identifikátor dané *entity* společně s referencí na *world*, do kterého patří. To umožňuje mít na této struktuře bohaté rozhraní s funkcemi pro přidávání a odebírání *komponent*.

4. ***System*:** Reprezentace *systémů* se v jednotlivých ECS implementacích dost liší. Některé vyžadují, aby *systém* byl třída dědící od abstraktního předka, jiné zase volí opačný extrém a dovolují *systém* reprezentovat téměř jakkoliv. Navzdory velkým odlišnostem se ve velkém počtu ECS implementací často objevuje objekt, který mohou používat jednotlivé systémy pro iterování entit s určitou množinou komponent.

5. ***Query*:** *Query* je objekt, který lze použít pro iterování *entit* s určitou množinou *komponent*. Pro jeho vytvoření je většinou nutné poskytnout informaci, nad jakými typy *komponent* bude *query* pracovat. Je například možné mít *query*, které pracuje nad *komponentami* `Movement` a `Position`. Takové *query* by pak bylo schopné iterovat nad všemi *entitami*, které mají *komponenty* `Movement` a `Position`. Pomocí tohoto *query* by bylo možné implementovat `MovementSystem`, který by řídil logiku pohybu.

Pro reprezentaci *systémů* v naší abstrakční vrstvě bude nejprve nutné si přiblížit, jak jednotlivé ECS knihovny reprezentují *systémy*. Jak již bylo zmíněno, reprezentace *systémů* se v jednotlivých ECS knihovnách dost liší. Významný rozdíl spočívá v způsobech, jakými iterují *entity*. Tyto způsoby se dají rozdělit do tří kategorií, kde každá kategorie obsahuje tři způsoby. Nyní si rozebereme jednotlivé kategorie a způsoby. Vždy nejprve představíme pseudokód pro způsoby z této kategorie a poté danou kategorii popíšeme.

1. kategorie

1. `Query((entity) => { /**/ })`
2. `Query((component1, component2) => { /**/ })`
3. `Query((components1[], components2[]) => { /**/ })`

Způsoby z této kategorie mají většinou definovanou metodu `Query` (neplést s *query* jako objektem pro iterování *entit*). Tato metoda bývá často implementována na *query* nebo na *world*. Této metodě se předává lambda funkce, která je v případě 1 volána na každou iterovanou *entitu*, v případě 2 volána na množinu žádoucích instancí *komponent* každé iterované *entity*, v případě 3 volána na množinu polí, které obsahují žádoucí instance *komponent* a kde jako index do těchto polí lze použít identifikátor příslušné *entity*.

2. kategorie

4. `void Update(entity) { /**/ }`

5. `void Update(component1, component2) { /**/ }`

6. `void Update(components1[], components2[]) { /**/ }`

Způsoby z druhé kategorie vyžadují, aby *systém* byl třída dědící od rozhraní nebo předka. Toto rozhraní nebo předek nabízí abstraktní metodu `Update`, kterou je nutné přetížit. Při iteraci je v případě 4 metoda `Update` volána na každou iterovanou *entitu*, v případě 5 na množinu žádoucích instancí *komponent* každé iterované *entity*, v případě 6 na množinu polí, které obsahují žádoucí instance *komponent* a kde jako index do těchto polí lze použít identifikátor příslušné *entity*.

3. kategorie

7. `foreach (entity in entities) { /**/ }`

8. `foreach ((component1, component2) in entities) { /**/ }`

9. ```
   for (int i = 0; i < entitiesCount; i++)
     { components1[i], components2[i], /**/ }
   ```

Třetí kategorie nabízí kolekce nebo iterátory, které lze iterovat. Tyto kolekce nebo iterátory lze získat voláním příslušné funkce na *world* nebo *query*. V případě 7 tato kolekce nebo iterátor obsahuje všechny *entity*, které chceme iterovat. V případě 8 tato kolekce nebo iterátor obsahuje n-tice žádoucích instancí *komponent* každé iterované *entity*. V případě 9 je namísto kolekce použita množina polí, které obsahují žádoucí instance *komponent* a kde jako index do těchto polí lze použít identifikátor příslušné *entity*.
   ```

3.2.2 Inlinování funkce

Při pohledu na způsoby pro iterování entit z minulé sekce přirozeně vyplývá pro každou entitu zavolat funkci (konkrétně na místě komentáře `/**/`), která danou entitu zpracuje. Ovšem volání funkce nás stojí čas. Tento čas není moc velký a je většinou zanedbatelný v porovnání s časem vykonávání funkce samotné. Ovšem my bychom chtěli volat funkci pro každou žádanou entitu a v případě, že by tato funkce byla jednoduchá (její vykonání by trvalo malé množství času), tak se jednotlivé časy volání této funkce nasčítají a naše řešení by vedlo ke ztrátě na výkonu. Předtím, než navrhneme samotné systémy, bude nutné si tuto problematiku více přiblížit.

Některé ECS knihovny se snaží tomuto problému předejít tím, že namísto toho, aby jejich systémy přijímaly funkci, kterou zavolají na každou entitu, poskytnou uživateli kolekce nebo iterátory, které si uživatel sám zpracuje. Tím se volání funkce vyhnou úplně. Znamená to tedy, že náš způsob ublíží na výkonu pouze některým knihovnám a to by mělo vliv na naše porovnání. Konkrétně se jedná o způsoby 3, 6, 7, 8 a 9 z minulé sekce.

JIT používá optimalizaci, při které se nahradí místo volání funkce samotným obsahem dané funkce. Díky tomu může být volání takové funkce zadarmo. Konkrétně se jedná o takzvané *inlinování funkce* (*inline expansion*). Ovšem JIT tuto optimalizaci ne vždy provede, jelikož v některých případech se to nevyplatí nebo to dokonce udělat nemůže.

Pro více informací o tom, kdy dochází a nedochází k *inlinování funkcí*, je možné nahlédnout do článků od Davida Notaria [10] a Vance Morrisona [11]. Konkrétně pro nás jsou důležité dva body.

1. K *inlinování funkce* nedojde, pokud se jedná o virtuální volání funkce.
2. K *inlinování funkce* nedojde, pokud je kód dané funkce příliš velký.

Je nutné zmínit, že tyto články jsou již starší a nyní jsou pravidla pro inlinování méně přísná. Například JIT je schopný v některých případech provést *inlinování funkce* i přesto, že se jedná o virtuální volání funkce.

3.2.3 Reprezentace systémů

Jak již bylo v minulé sekci naznačeno, každý systém bude mít tedy funkci, kterou bude volat na každou entitu. Zároveň každý systém iteruje jednotlivé entity jiným způsobem. Systémy tedy budeme reprezentovat tím, že si pořídíme dva typy.

1. **`EntityProcessor`:** Tento typ bude mít na sobě již zmiňovanou funkci pro zpracování jedné entity. Nazvěme ji `Process`.

2. **`IECSSystem`:** Jedná se o rozhraní. Pro každou ECS knihovnu, kterou budeme chtít měřit, implementujeme třídu, která bude dědit od tohoto rozhraní. Toto rozhraní bude nabízet metodu `Update`, ve které implementujeme iteraci přes jednotlivé entity. Každá instance takového systému bude mít také instanci `EntityProcessor`, jejíž metodu `Process` při iteraci zavolá na jednotlivé entity.

Tím jsme vyřešili různé reprezentace systému, ale ještě zbývá problém s *inlinováním funkce*. Vyřešíme to tím, že jednotlivé `EntityProcessor` budou struktury, které budou dědit od rozhraní `IEntityProcesor`. Instance systému poté přijme typ konkrétního `EntityProcessor` jako generický argument.

Nyní si vysvětlíme, proč je pro nás důležité, že jednotlivé `EntityProcessor` jsou struktury.

Při první konstrukci generického typu, který má hodnotový datový typ jako parametr, se vytvoří specializace dané třídy, ve které je daný parametr nahrazen daným typem. Je nutné poznamenat, že k tomuto dochází pouze v případě dosazení hodnotového datového typu (struktury jsou hodnotové datové typy). V případě referenčních datových typů se vytváří pouze jedna jediná specializace, ve které je daný argument nahrazen typem `object`. Pro více informací o tomto procesu je možné nahlédnout do článku *Generics in the runtime (C# programming guide)* [12] od firmy Microsoft.

Při použití konkrétního systému, kterému předáme jako generický argument jeden z `EntityProcessor`, dojde tedy k výše zmíněnému procesu. Konkrétně dojde k vytvoření specializace pro již zmiňovanou metodu `Update`. Tato specializace iteruje jednotlivé entity a na každé z nich volá metodu `Process` na konkrétním `EntityProcessor`. Na tomto volání může JIT provést *inlinování*.

Může se stát, že metoda `Process` na konkrétním `EntityProcessor` bude příliš velká a nedojde k jejímu *inlinování*. Tomu můžeme napomoci s použitím atributu `MethodImplOptions.AggressiveInlining` [13], který JITu napoví, že bychom chtěli konkrétní metodu *inlinovat* při jakémkoliv volání. Jedním z efektů při použití tohoto atributu je, že dojde k navýšení velikosti kódu, při které je kód dané funkce považován za příliš velký pro *inlinování*. Může se stát, že velikost kódu dané funkce bude i po použití tohoto atributu příliš vysoká. V takovém případě ale bude cena vykonání dané funkce výrazně vyšší než její volání.

3.2.4 ECSFactory

Každá implementace abstrakční vrstvy bude mít několik typů, které bude muset implementovat. Ovšem pro snazší práci s abstrakční vrstvou by bylo lepší, kdyby bylo možné celou konkrétní implementaci reprezentovat jedním typem. Tento typ

by byl zodpovědný za konstrukci jednotlivých typů z konkrétní implementace abstrakční vrstvy.

Zavedeme si tedy abstraktní třídu `ECSFactory`. Každá konkrétní implementace abstrakční vrstvy poskytne třídu, která bude dědit od `ECSFactory`.

Vzniká nám tu problém a to v tom, že pokud chceme vytvořit instanci systému (třídy, která dědí od `IECSSystem`) je nutné ji předat kromě typu konkrétního `EntityProcessor` také typy jednotlivých komponent se kterými bude systém pracovat (musí je předat *query*). Ovšem tato informace je redundantní, jelikož typy těchto komponent jsou zřejmé z konkrétního `EntityProcessor`, který předáváme. Tímto nám vznikají méně přehledné a zdlouhavé řádky kódu pro vytváření jednotlivých systémů. Například, pokud bychom měli `ArchSystem` jako typ systému ECS knihovny *Arch* [14] a `PathFollowSystem` jako `EntityProcessor`, který pracuje nad komponentami `Location` (obsahuje informaci o pozici entity), `Movement` (obsahuje informace o pohybu entity) a `PathFollow` (obsahuje informace o pohybu entity po cestě), tak konstrukce instance tohoto systému by vypadala takto: `new ArchSystem<PathFollowSystem, Location, Movement, PathFollow>()`.

Kromě výše zmíněného problému je také další problém v tom, že třída zodpovědná za konstrukci těchto systémů musí vědět o tom, jaký `EntityProcessor` pracuje s jakými komponentami. Také, pokud bychom chtěli přidat nebo odebrat komponentu pro nějaký `EntityProcessor`, tak bychom museli tuto změnu provést na více místech (na místě definice daného `EntityProcessor` a na místě jeho konstrukce).

Abychom tyto problémy vyřešili, předáme zodpovědnost za konstrukci těchto systémů třídě `ECSFactory`. Ta skrze konstruktor dostane datový typ systému. Tato třída bude obsahovat metodu `CreateSystem`, která přijme instanci konkrétního `EntityProcessor` a pomocí reflexe zkonstruuje konkrétní finální systém. Výše zmíněný příklad by tedy s tímto řešením vypadal takto: `factory.CreateSystem(new PathFollowSystem())`.

Je také nutné zmínit, že konkrétní implementace abstrakční vrstvy musejí poskytovat vícero tříd dědících od `IECSSystem`, jelikož je nutné definovat systém, které pracují s jednou komponentou, poté systém, který pracuje s dvěma komponentami a tak dále. `ECSFactory` tedy v konstruktoru přijímá více typů systémů a při zavolání `CreateSystem` vybere jeden z nich na základě počtu generických argumentů na typu `EntityProcessor`.

3.2.5 Komponenty jako třídy a komponenty jako struktury

Jednotlivé ECS knihovny mají různé požadavky na typech jednotlivých komponent. Některé chtějí, aby tyto typy byly třídy, jiné zase chtějí, aby tyto typy byly struktury, a jiné povolují kombinaci obou variant.

Možné řešení by bylo mít dva typy pro každou komponentu, kde jeden by byl třída a druhý struktura. Ovšem toto řešení by vedlo k duplicitnímu kódu.

Řešení, které bylo použito na tento problém, je všechny typy komponent definovat jako struktury. V případě, že daná ECS knihovna vyžaduje, aby jednotlivé typy komponent byly třídy, tak si definuje svou třídu `ComponentWrapper`, která jako generický argument přijme typ dané komponenty. Instance této třídy budou mít v sobě uloženou instanci konkrétní komponenty.

Některé ECS knihovny také požadují, aby jednotlivé komponenty implementovaly specifické rozhraní nebo atribut. V případě těchto knihoven nám postačí toto rozhraní nebo atribut implementovat na příslušný `ComponentWrapper` (ten nemusí být nutně třídou).

3.3 Umělá inteligence

Jak již víme ze sekce 2.2.2, naše hra bude obsahovat vesničany. Každý vesničan bude reprezentován entitou a bude vykonávat své akce na základě jednoduché umělé inteligence. Každý vesničan na sobě bude mít `Behavior` komponentu, která bude popisovat jeho chování. Také zde bude `BehaviorSystem`, který bude zpracovávat logiku umělé inteligence pro každou entitu s `Behavior` komponentou.

Způsobů, kterými lze umělou inteligenci aneb chování dané entity popsat nebo definovat je více. My se zaměříme pouze na ty používanější, které jsou dostatečné pro implementaci umělé inteligence pro naše vesničany:

1. **Stavové automaty:** Přímočarý způsob, kterým lze definovat chování entit, je *stavový automat*. Každý stav tohoto automatu reprezentuje činnost, kterou může entita provést. K jednotlivým stavům mohou být také definovány přechody. Pro více informací o použití *stavových automatů* pro umělou inteligenci je možné nahlédnout do materiálu *AI - Finite State Machines* [15] od *Newcastle University*.

   Ve vývoji her bývá pro implementaci *stavových automatů* pro AI často použit návrhový vzor `State` (více o něm je možné se dočíst v již zmiňované knize *Game Programming Patterns* [3]).

   V případě naší hry bychom měli pro každý stav *stavového automatu* třídu nebo strukturu s metodou `Update`. Každý vesničan by měl ve své `Behavior` komponentě uloženou instanci na svůj současný stav. Přechod na jiný stav by byl reprezentován přiřazením instance jiného stavu.

   Mezi stavy našich vesničanů by mohlo patřit:

(a) **`MoveToState`:** Tento stav by nastavil příslušná data v `Movement` komponentě a poté by čekal, než vesničan dojde na specifikovanou pozici. Poté by nastal přechod na specifikovaný stav.

(b) **`FindNearestResourceState`:** Stav zodpovědný za nalezení nejbližší suroviny daného typu. Po jejím nalezení by nastal přechod na `MoveToState`, pomocí kterého by vesničan došel k příslušné surovině.

(c) **`HarvestResourceState`:** Tento stav by nastavil entitu suroviny, která se má sklidit, jako `Target` do příslušné komponenty. Poté by vyčkal na její sklizení. Po jejím sklizení by nastal přechod na `MoveToState`, pomocí kterého by vesničan došel ke svému `Workplace`.

(d) **`ProcessResourceState`:** Tento stav by pouze přesunul suroviny pro zpracování do příslušné budovy a poté by vyčkal, než budou zpracovány. Po jejich zpracování by z ní přesunul zpracované předměty do inventáře vesničana a nastal by přechod na `MoveToState`, pomocí kterého by vesničan došel ke skladišti.

(e) **`StoreInStockpile`:** Tento stav by přesunul příslušné předměty do skladiště a poté by nastal přechod na `FindNearestResourceState`.

2. **Stromy chování:** *Stromy chování* jsou složeny z hierarchie vrcholů, která popisuje chování dané entity. Listy *stromu chování* představují konkrétní příkazy, které řídí entitu. Ostatní vrcholy slouží k tomu, aby řídili jak se strom bude procházet. Více o *stromech chování* je možné se dočíst ve článku *Behavior trees for AI: How they work* [16] od Chrise Simpsona.

   Jednotlivé *stromy chování* bývají často reprezentovány třídou nebo strukturou. Jednotlivé vrcholy bývají reprezentovány metodami. Pro konstrukci *stromů chování* se často používá návrhový vzor *Builder* (více o něm je možné se dočíst v knize *Design Patterns: Elements of Reusable Object-Oriented Software* [17]).

   Každý vesničan by měl svou instanci *stromu chování* uloženou v `Behavior` komponentě. Příklad definice chování vesničana by mohl vypadat takto:

```
Builder.Create()
  .Sequence("villager job sequence")
    .Do("find resource", FindResource(resourceType))
    .Do("move to resource", MoveTo(null))
    .Do("harvest resource", HarvestResource)
    .Do("move to workplace", MoveTo(workplace))
```

```
    .Do("process resource", ProcessResource(workplace))
    .Do("move to stockpile", MoveTo(stockpile))
    .Do("store items", StoreItems(stockpile))
  .End()
```

v tomto příkladu je kořenový vrchol `villager job sequence`, který prochází své potomky ze shora dolů. Vrcholy s prefixem `move to` nastaví pohyb vesničana na příslušnou pozici a poté čekají, než k ní vesničan dojde. Vrchol `find resource` nalezne nejbližší surovinu daného typu. Vrchol `harvest resource` nastaví entitu dané suroviny jako `Target` v dané komponentě, poté vyčká na její sklizení. Vrchol `process resource` přesune dané předměty do budovy na zpracování surovin, poté vyčká na jejich zpracování a poté z této budovy přesune zpracované předměty do inventáře vesničana. Vrchol `store items` přesune dané předměty do skladiště.

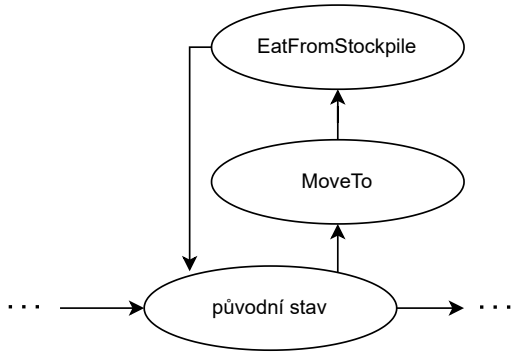
`BehaviorSystem`, který by prošel všechny `Behavior` komponenty, by byl v uvedených způsobech zodpovědný za zpracování AI. V obou přístupech by byla v zmiňované komponentě uložena instance buďto *stavového automatu* nebo *stromu chování*. `BehaviorSystem` by zavolal příslušnou metodu pro zpracovaní na každé takové instanci.

Porovnání *stavových automatů* a *stromů chování* je možné nalézt ve článku *Finite State Machine & Behavior Tree in Robotics* [18] od Minga Kwona. I přestože tento článek porovnává oba způsoby v robotice, tak dané poznatky jsou použitelné i v oblasti umělé inteligence.

Ve výše zmíněném článku je možné se dočíst, že mezi nevýhody *stavových automatů* patří rychle rostoucí komplexita při přidávání nových stavů, pro jejichž přidání je někdy nutné provést velké modifikace.

Zmíněná nevýhoda by nám způsobila potíže například pokud bychom chtěli do výše uvedeného automatu přidat mechaniku hladovění vesničanů popsanou v sekci 2.2.2. Ta by se za pomoci *stavového automatu* dala řešit například tím, že bychom si pořídili další `MoveToState`, díky kterému by vesničan došel do skladiště, a poté by nastal přechod do nového stavu `EatFromStockpileState`, díky kterému by vesničan snědl jídlo ze skladiště.

Ovšem poté, co se vesničan nají, je nutné se vrátit zpět do stavu, který vesničan vykonával předtím, než se šel najíst. Jením z řešení by bylo pořídit si výše zmíněnou dvojici stavů pro každý stav z původního automatu a provést přechod do této dvojice, pokud hlad vesničana klesl pod určitou hodnotu. Poté, co by se vykonal `EatFromStockpileState`, nastal by přechod zpět do původního stavu. Situaci je možné vidět na obrázku 3.1.



Obrázek 3.1 Řešení mechaniky hladu pro jeden vrchol z původního stavového automatu.

Alternativní způsob by byl použít dva stavové automaty. Hlavním automatem by byl automat řešící hlad, který by měl `MainState`, ve kterém by se nejprve zkontrolovalo, zda hladovění vesničana nekleslo pod určitou hodnotou. Pokud ne, tak zavolá původní automat, jinak nastane přechod do již zmiňované dvojice stavů `MoveToState` a `EatFromStockpileState`.

Je lehké nahlédnout, že obě řešení přidávají netriviální složitost navíc do definice chování pro naše vesničany.

Podle výše zmíněného článku mají *stromy chování* dobrou rozšiřitelnost a modularitu. Pokud bychom chtěli rozšířit náš *strom chování* o výše zmíněnou mechaniku hladu, mohli bychom vytvořit nový strom chování, který by použil `Selector` vrchol jako kořen, který by měl dva potomky. Prvním by byl podstrom řešící mechaniku hladu a druhým by byl celí původní strom připojený kořenem jako podstrom nového stromu chování. Celá situace by vypadala takto:

```
Builder.Create()
  .Selector("tree root")
    .Sequence("villager hunger sequence")
      .Condition("is hunger bellow threshold?", IsHungry)
      .Condition("is there food?", IsThereFood(stockpile))
      .Do("go to get food", MoveTo(stockpile))
      .Do("eat food", EatFood)
    .End()

    .Sequence("villager job sequence")
      // original tree as subtree
    .End()
```

```
    .End()
```

Je lehké nahlédnout, že rozšiřitelnost a modularita *stromů chování* jsou opravdu silné vlastnosti. Proto také *stromy chování* použijeme pro definici chování pro naše vesničany.

3.4 Generovaný svět

Jak jsme si specifikovali v sekci 2.2.1, svět v naší hře bude náhodně generovaný. Konkrétně bude obsahovat náhodně generovaný terén, ve kterém budou náhodně umístěny suroviny a vesnice.

Vygenerovaný terén bude 2D plocha, která se bude skládat ze spojitých oblastí. Každá oblast bude odpovídat nějakému biomu a bude mít barvu podle tohoto biomu (například oblast odpovídající biomu vody bude modrá). Některé oblasti mezi sebou budou mít závislosti (například oblast, která reprezentuje biom vysoké hory, bude obklopena oblastí, která reprezentuje biom hory). Je nutné zmínit, že nechceme políčka, ale opravdu spojité oblasti.

Naši hru budeme používat převážně pro měření výkonu ECS knihoven, tím pádem nemáme na generování našeho terénu žádné složité požadavky. Mezi naše požadavky patří:

1. **Generování terénu by nemělo trvat příliš dlouho:** Pokud by generování terénu trvalo příliš dlouho, vedlo by to k delšímu času spouštění naší hry. Vzhledem k tomu, že hru budeme používat pro měření výkonu ECS knihoven a pro každou knihovnu ji pouštět znovu, delší čas spouštění hry by nám zbytečně natahoval dobu vykonávání jednoho měření.

2. **Možnost navigace:** Ze sekce 2.2.2 víme, že hra bude obsahovat vesničany. Pro vesničany bude důležité, aby byli schopni se v náhodně vygenerovaném terénu navigovat. Pokud vesničan bude například chtít jít pokácet strom, bude se chtít po cestě k němu vyhnout vodě.

3. **Žádné vizuální artefakty:** Chceme se vyhnout vizuálním artefaktům, díky kterým by náš terén nevypadal dobře.

3.4.1 Generování terénu

Z požadavků na naši hru, které jsme si představili v sekci 2.1.1, vyplývá, že nechceme naši hru dělat příliš složitou. Terén tedy budeme generovat tím, že si za pomoci šumové funkce nejprve vytvoříme výškovou mapu a tu poté obarvíme. Tento přístup jsme zvoli, jelikož je jednoduchý na implementaci a poskytuje dobře vypadající terén.

Pro vygenerování výškové mapy bude nejprve nutné zvolit šumovou funkci, proto si jednotlivé šumové funkce nejprve rozebereme:

1. **Perlinův šum:** Jedná se o známou šumovou funkci. Tato šumová funkce je rychlá a také jednoduchá na implementaci. Je jí možné použít v libovolné dimenzi, ale nejčastěji se používá ve druhé nebo třetí. Přehledné vysvětlení o tom, co to je Perlinův šum a jak se používá pro generování terénu, je možné nalézt v diplomové práci *Terrain synthesis using noise* [19] od *Tuoma Hyttinena*.

2. **Simplexový šum:** *Simplexový šum* je velmi podobný *Perlinovu šumu*. Přehledné vysvětlení o tom, co to je *Simplexový šum* a jak se používá pro generování terénu je možné nalézt v již zmíněné diplomové práci *Terrain synthesis using noise* [19]. V této práci je možné se dočíst, že oproti *Perlinovu šumu* je *Simplexový šum* rychlejší a produkuje výsledky, které vypadají více přirozeně. Ovšem má jednu velkou nevýhodu, a to patent, který omezuje jeho použití pro komerční účely. Z toho důvodu se v praxi tolik nepoužívá.

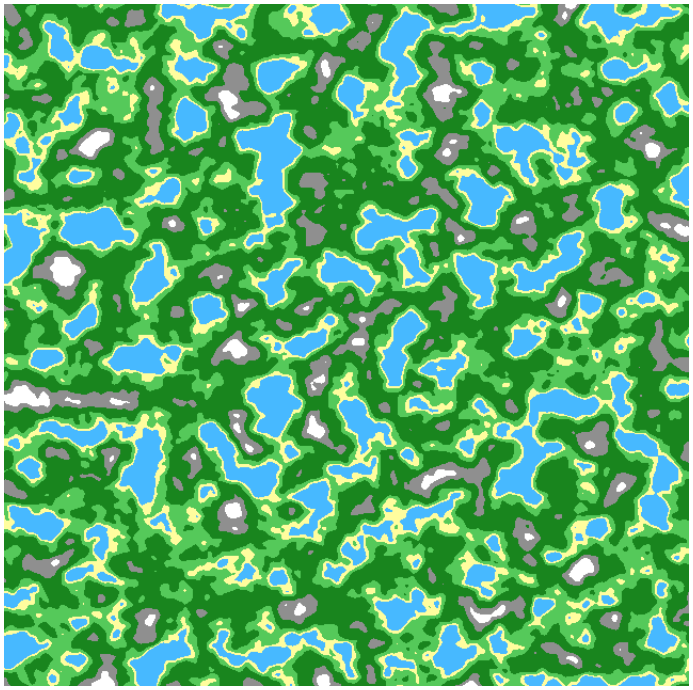
I přesto, že *Simplexový šum* nabízí výhody oproti *Perlinovu šumu*, pro implementaci zvolíme *Perlinův šum* z důvodu již zmíněného patentu. Konkrétně jej budeme používat ve druhé dimenzi.

Nyní za pomoci *Perlinova šumu* vygenerujeme výškovou mapu. Pro generování výškové mapy lze použít Fraktální součet nebo Turbulenci. Jedná se o techniky pro generování více komplexních vzorů. O obou technikách je možné se více dočíst v již zmíněné diplomové práci *Terrain synthesis using noise* [19].

Pro zvolení techniky autor práce provedl experiment a nechal vygenerovat terén pomocí obou technik. Z výsledných terénů autor více preferoval terén vygenerovaný Fraktálním součtem, proto jej zvolíme.

Nyní zbývá pomocí výškové mapy vygenerovat daný terén. Již víme, že náš terén bude 2D plocha. Tuto plochu si rozdělíme na malé atomické části (například pixely) a těmto částem přiřadíme souřadnice. Barva každé části bude odpovídat danému biomu. Každý biom bude mít definovaný výškový interval a barvu. Pro souřadnice každé atomické části si necháme vygenerovat výšku z výškové mapy. Poté na základě této výšky a výškových intervalů jednotlivých biomů získáme příslušný biom. Poté barvu dané části nastavíme na barvu daného biomu. O této technice je možné se více dočíst ve článku *Making maps with noise functions* [20] ze stránky *Red Blob Games*.

Příklad výsledného terénu je možné vidět na obrázku 3.2.



Obrázek 3.2 Příklad jak může vypadat námi vygenerovaný terén.

3.4.2 Generování terénu na CPU vs. GPU

Za použití techniky z minulé sekce jsme získali vygenerovaný terén. Ovšem to, jestli naše technika splňuje naše požadavky, je závislé na tom, zda terén budeme generovat na CPU nebo na GPU. Proto si nyní oba přístupy rozebereme.

Generování terénu na CPU

Při generování terénu na CPU bychom si pořídili jednu nebo více textur a vygenerovali terén v nich za použití způsobu z minulé sekce.

Při generování tohoto terénu je možné zvolit rozlišení, konkrétně specifikací velikosti atomických částí. Volba tohoto rozlišení nám ovlivňuje dobu trvání vygenerování našeho terénu. Pokud bychom zvolili malé atomické části, generování by trvalo příliš dlouho a porušili bychom požadavek na rychlost. Pokud bychom zvolili velké atomické části, generování by trvalo menší dobu, ale zase bychom porušili požadavek na absenci vizuálních artefaktů.

V ideálním případě bychom chtěli velké rozlišení, ale také lepší rychlost. Jeden ze způsobů, jak toho dosáhnout, je paralelizace. Ovšem možnosti paralelizace na CPU jsou oproti GPU omezené, proto si rozeberme, jak by se dal terén generovat na GPU.

Generování terénu na GPU

Terén budeme generovat v reálném čase (vygenerujeme jej v každém snímku znovu) a vždy nám postačí vygenerovat pouze část terénu, která je viditelná hráčem. Tato část bude obdélník se stejnými rozměry jako velikost okna naší hry.

Pořídíme si texturu, do které budeme generovat terén. Tato textura bude mít stejnou velikost jako okno naší hry. Poté si pořídíme fragment shader, kterým do této textury budeme generovat terén. Tento fragment shader v každém snímku na základě pozice a přiblížení kamery pro každý pixel naší textury provede algoritmus popsaný v sekci 3.4.1.

Díky vysoké paralelizaci GPU můžeme vždy zvolit nejmenší velikost atomických oblastí (velikost jednoho pixelu) a stejně budeme mít pořád dostatečnou rychlost.

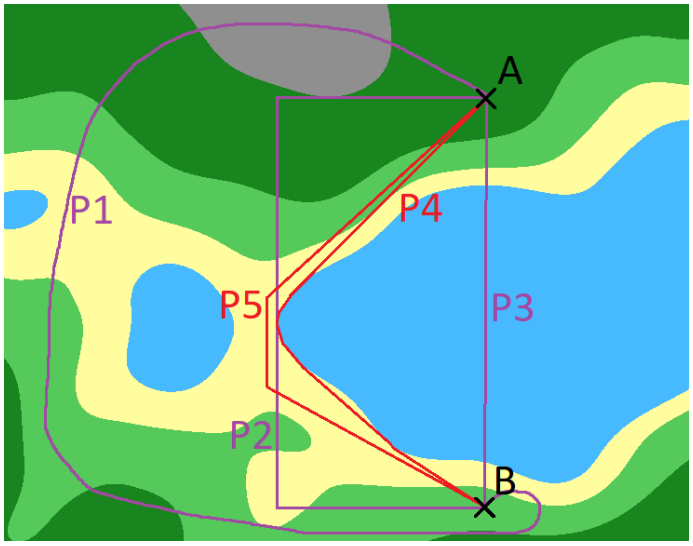
3.4.3 Navigace ve vygenerovaném terénu

Generování terénu na GPU se zdá být ideálním kandidátem, ovšem tím, že terén budeme generovat na GPU, nám vzniká problém, jak ve vygenerovaném terénu budeme vlastně provádět navigaci. V této sekci tento problém vyřešíme.

Vesničané v naší hře budou často potřebovat nalézt cestu v námi vygenerovaném terénu k nějaké entitě. Budou například chtít dojít ke skladišti nebo ke svému pracovnímu místu. Jelikož k hledání cesty bude docházet poměrně často, budeme chtít, aby bylo dostatečně rychlé. Dále budeme chtít, aby cesty působily přirozeně. Příklady vhodných (přirozených) a nevhodných (nepřirozených) cest je možné vidět na obrázku 3.3.

Fialové cesty jsou nevhodné. Konkrétně cesta $P1$ je nevhodná, jelikož vede zbytečnou velkou oklikou. Cesta $P2$ je nevhodná, protože vede pouze podél os. Cesta $P3$ je nevhodná, jelikož vede přes vodu.

Červené cesty jsou vhodné. Cesta $P4$ je nejkratší cestou z bodu A do bodu B. Cesta $P5$ sice není nejkratší cesta, ale je pro nás dostatečně dobrou aproximací a je pro nás dostačující.



Obrázek 3.3 Příklad cest z bodu A do bodu B. Fialové cesty jsou nevhodné. Červené cesty jsou vhodné.

Navigační struktury

Pro navigaci v terénu použijeme datovou strukturu. Tato datová struktura reprezentuje terén způsobem, který umožňuje snadné hledání cest mezi dvěma body. Často k tomu používají algoritmus pro hledání nejkratších cest. Typickou volbou bývá algoritmus A*. Datové struktury, které lze pro navigaci použít, nazvěme navigační struktury.

Velkou část navigačních struktur tvoří navigační meshe (nebo také zkráceně navmesh). Ty jsou často používány populárními frameworky a enginy, jako je například Unity. Navigační meshe vezmou plochu terénu po které lze chodit, a rozdělí ji do oblastí. Každá taková oblast je tvořena konvexním polygonem. Poté si vytvoří graf, kde každý takovýto polygon je reprezentován vrcholem a každé sousední polygony mají mezi sebou hranu. V tomto grafu se poté hledají nejkratší cesty. Je nutné upozornit, že existuje velké množství navigačních meshů a některé se od tohoto popisu mohou lišit. Pro více informací o navigačních meshích lze nahlédnout do studie *A comparative study of navigation meshes* [21].

V sekci 2.1.1 jsme si uvedli, že nechceme, aby naše hra byla příliš složitá, jelikož by to odvádělo pozornost od problému, který řešíme. Z toho důvodu nebudeme používat navigační mesh jako navigační strukturu pro naši hru, jelikož i přesto, že použití navigačního meshe pro navigaci není složitá úloha, tak jeho konstrukce složitá já.

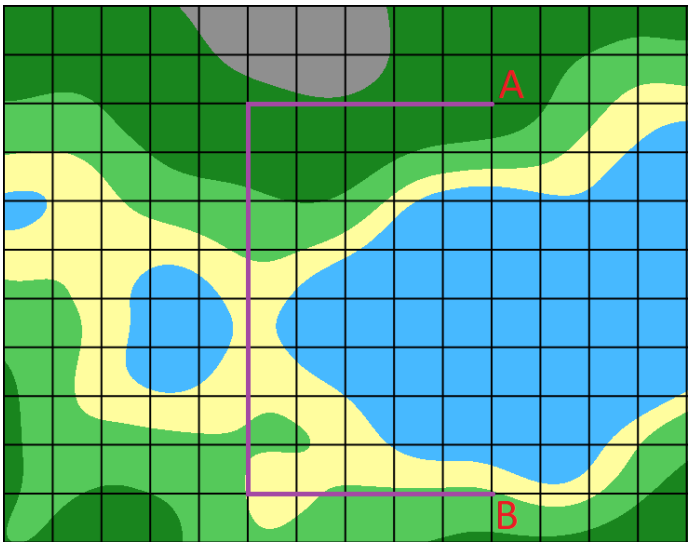
Jako navigační strukturu zvolíme dvourozměrnou mřížku. Tu si budeme reprezentovat jako pole, kde každý prvek tohoto pole ponese informaci o tom,

v jakém biomu se nachází daný vrchol mřížky. Tuto strukturu zvolíme, protože je jednoduchá na konstrukci. Cesty, které díky této mřížce získáme, nebudou nejlepší možnou nejkratší cestou (a nebudou ve skutečnosti ani nejkratší), ale budou pro nás dostatečně dobrou aproximací.

Tvorba cesty

Tvorbu cesty začneme tím, že získáme cestu, která bude z počátku nevhodná. Poté ji vylepšíme a tím získáme cestu, která již bude vhodná.

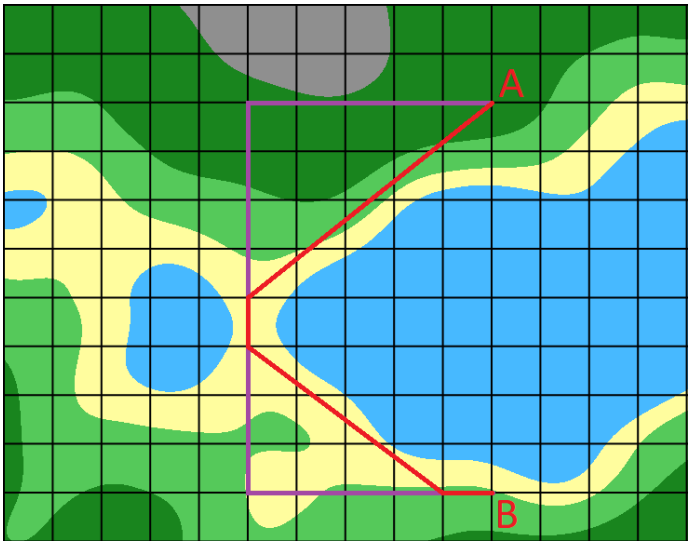
Pro výpočet vzdálenosti mezi dvěma body v naší mřížce zvolíme Manhattanskou metriku. Pro hledání nejkratších cest použijeme algoritmus A*. První tvar naší cesty získáme tím, že pomocí A* algoritmu nalezneme cestu v naší mřížce. Příklad toho, jak by taková cesta mohla vypadat, je možné vidět na obrázku 3.4. Je lehké nahlédnout, že se podobá cestě $P2$ z obrázku 3.3, která je pro nás nevhodná.



Obrázek 3.4 Příklad cesty z bodu A do bodu B nalezené A* algoritmem v naší mřížce.

Následně provedeme úpravu této cesty. Začneme v prvním bodu cesty a provedeme ray cast do druhého bodu. Pokud jsme nenarazili na vodu, pokusime se provést ray cast ke třetímu bodu. To budeme opakovat, dokud náš ray cast nenarazil na vodu. V takovém případě můžeme cestu zjednodušit a odstranit z ní všechny body mezi prvním a posledním úspěšným. Poté se přesuneme ke druhému bodu a postup opakujeme. Celí tento proces provádíme, dokud nedojdeme do posledního bodu naší cesty. Zjednodušení fialové cesty je možné vidět na obrázku 3.5, kde fialová cesta představuje původní cestu a červená cesta její zjednodušení.

Ovšem problém spočívá v tom, že k dispozici máme pouze mřížku, která obsahuje data o našem terénu, tím pádem nemůžeme provést klasický ray cast.



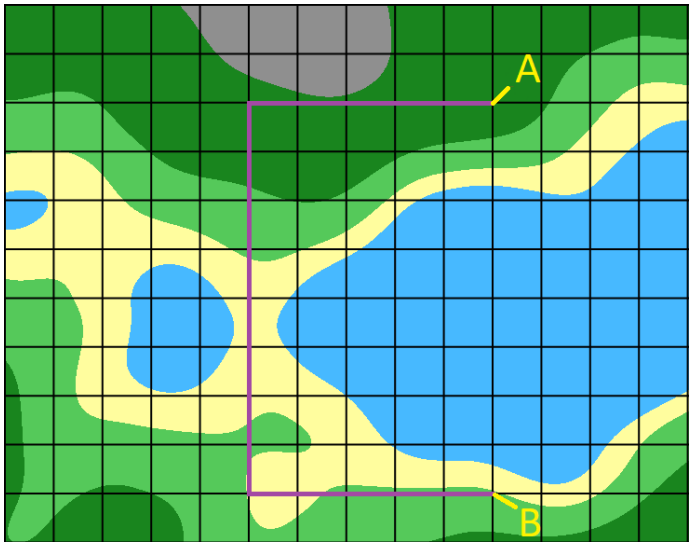
Obrázek 3.5 Příklad cesty z bodu A do bodu B nalezené A* algoritmem v naší mřížce, která byla následně zjednodušena. Fialová cesta je originální nevhodná cesta. Červená cesta je zjednodušená, již vhodná, cesta.

Proto provedeme pouze jeho aproximaci. Začneme v bodu N a chceme provést aproximaci raycastu do bodu M. Spočítáme si směr z bodu N do bodu M a tímto směrem se budeme posouvat vždy o nějakou malou vzdálenost Δ. Pokaždé. když se posuneme. tak si spočítáme nejbližší bod z naší mřížky. Pokud se jedná o bod s biomem vody, tak ray cast selhal. V případě, že jsme takto došli až do bodu M a nenarazili jsme na bod vody, ray cast byl úspěšný.

Mohlo by se zdát, že použití této aproximace by vedlo v příliš nevhodné cesty, ovšem za předpokladu, že se body v naší mřížce budou nacházet dostatečně blízko sebe, tak bude výsledná cesta dostatečně dobrou aproximací červené cesty z obrázku 3.5.

Další problém, který je nutné vyřešit, je, že vesničané a entity, ke kterým se vesničané navigují, se většinou nenacházejí v žádném z bodů naší mřížky. Problém vyřešíme tak, že před spuštěním A* algoritmu si k vesničanovi a dané entitě nalezneme nejbližší body v naší mřížce a nalezneme nejkratší cestu mezi těmito body. Poté, předtím než provedeme zjednodušení cesty, přidáme pozici vesničana jako první bod naší cesty a pozici entity, ke které se vesničan naviguje, jako poslední bod naší cesty.

Situaci je možné vidět na obrázku 3.6. Vesničan se nachází v bodu A, který není bodem naší mřížky, a chce se navigovat k entitě umístěné na bodu B, který také není bodem naší mřížky. Na začátek a konec cesty po provedení algoritmu oba body přidáme a tím získáme žluté úseky. Přidáním těchto bodů by nám mohly vzniknout nepřirozené artefakty, ovšem ty budou spraveny během zjednodušování cesty.



Obrázek 3.6 Příklad cesty z bodu A do bodu B pro entity, které se nenacházejí na některém z bodů naší mřížky.

Zbývá nám vyřešit, jak získáme již zmiňovanou mřížku. Abychom ji získali z našeho terénu, bude potřeba jej navzorkovat. Podobně jako při jeho generování si terén rozdělíme na atomické části a střed každé takové atomické části bude bodem v naší mřížce. Protože celý náš terén generujeme na GPU, bude nutné toto navzorkování také provádět na GPU. Pořídíme si proto compute shader, pomocí kterého terén navzorkujeme na GPU, a tím vytvoříme mřížku, kterou poté přesuneme na CPU.

Ovšem problém nastává v tom, že MonoGame framework, který pro tvorbu naší hry používáme, nemá podporu pro compute shadery. Z toho důvodu namísto nativního MonoGame zvolíme jeho fork [22] od Markuse Hötzingera, který tuto podporu přináší.

3.5 Měření

Náš projekt bude reprezentován C# solution. Toto C# solution se skládá ze dvou C# projektů a to hry (`WorldSimulator.csproj`) a poté měření (`WorldSimula-`

`tor.Benchmarks.csproj`), které používá onu hru pro měření výkonu ECS knihoven. V této sekci se budeme věnovat analýze tohoto měření.

3.5.1 Jak budeme měření provádět

Prvně si přiblížíme, co to je *game loop*, jelikož to budeme potřebovat k popisu měření. Následně si popíšeme samotné měření.

Game loop

Game loop si můžeme představit jako cyklus, který běží, dokud nedojde k ukončení hry. Pseudokód pro *game loop* může vypadat například takto:

```
while (!shouldExit)
{
  Update();
  Draw();
}
```

Pseudokód obsahuje `while` cyklus, který se opakuje, dokud nedojde k nastavení proměnné `shouldExit` na `true`. V těle tohoto cyklu se volají dvě funkce a to `Update`, která je zodpovědná za zpracování herní logiky, a `Draw`, která je zodpovědná za vykreslení hry.

I naše hra obsahuje *game loop*. Logika v naší hře se skládá ze systémů. Ty se dělí na *update systémy*, které řeší zpracování herní logiky, a *render systémy*, které řeší vykreslování. *Update systémy* běží uvnitř funkce `Update` a *render systémy* běží uvnitř funkce `Draw`. Je nutné upozornit, že tento popis je zjednodušen a více dopodrobna se implementaci hry a systémů budeme věnovat v následující kapitole.

Popis měření

Budeme mít stanovené dva pevné počty iterací. První bude přípravný počet iterací a druhý bude měřený počet iterací. Měřený počet iterací bude větší než přípravný počet iterací.

Článek *Tips for Software Performance Testing* [23] od *Nilushana Costy* zmiňuje pět tipů pro měření výkonu. Jedním z těchto tipů je provést před měřením výkonu takzvanou *warmup* fázi, během které program dosáhne stabilního stavu. Jistý *warmup* bude za nás provádět framework, který zvolíme v příští sekci. Ovšem jedná se o *warmup* prostředí jako takového. Naše *warmup* fáze před začátkem měření dostane hru do nějakého již lehce rozehraného stavu.

Warmup fázi provedeme před samotným měřením. Během této fáze odsimulujeme přípravný počet iterací našeho *game loop*. Během této fáze nic neměříme.

Po provedení *warmup* fáze spustíme samotné měření. Během měření vezmeme tu samou instanci hry, která běžela přípravný počet iterací během *warmup* fáze, a necháme její *game loop* běžet měřený počet iterací. Během fáze měření budeme také měřit čas, během kterého hra provádí měřený počet iterací svého *game loop*. Tento čas bude výsledným časem pro jednu konkrétní ECS knihovnu.

Výše popsaný test provedeme pro každou ECS knihovnu, kterou budeme měřit. Pomocí výsledných časů můžeme relativně porovnat výkon naší hry s různými ECS knihovnami.

Při měření výkonu jednotlivých ECS knihoven ignorujeme některé jejich funkcionality. Konkrétně se jedná o funkcionality rozšíření ECS. Jedná se například o funkcionality známe pod názvy *reakční systémy* (*reaction systems*) nebo *relace* (*relations*). V našem měření je budeme ignorovat, protože chceme měřit klasické ECS bez rozšířeních.

3.5.2 Volba frameworku pro měření

Nyní zvolíme framework, který budeme používat pro měření. Prvně si popíšeme vlastnosti, které od tohoto frameworku požadujeme, poté provedeme samotný výběr. Mezi požadované vlastnosti patří:

1. **Korektnost:** Chceme, aby naměřené výsledky byly spolehlivé. Klíčovým faktorem bude, aby výsledek vyprodukovaný pro danou ECS knihovnu byl porovnatelný s výsledkem vyprodukovaným pro jinou ECS knihovnu.

2. **Jednoduchost:** Budeme chtít, aby framework byl jednoduchý na používání. Nechceme dávat přednost jednoduchosti před korektností, ale pokud bude na výběr mezi frameworky, které splňují první požadavek, tak upřednostníme ten, který je jednodušší na používání.

Pro výběr frameworku použijeme platformu GitHub. Na stránce [24] je možné vidět seznam repozitářů na platformě GitHub s topicem *performance* seřazené podle počtu hvězd. Je možné nahlédnout, že nejpopulárnějším frameworkem pro měření výkonu je *BenchmarkDotNet* [25].

BenchmarkDotNet je framework, pomocí kterého lze měřit výkon metod. Tento framework splňuje oba naše požadavky, ale jedná se o framework primárně určený na micro benchmarky, zatímco naše měření je spíše macro benchmarkem. I přesto jej pro naše měření použijeme, primárně protože je velmi jednoduchý na použití.

Kapitola 4

Popis implementace

V této kapitole si představíme klíčové a zajímavé části implementace. Pro více detailů o implementaci je možné nahlédnout do kódu, který obsahuje dokumentační komentáře.

Pro celý projekt používáme jedno C# solution. To obsahuje dva C# projekty. Prvním je hra (`WorldSimulator.csproj`) a druhým je měření (`WorldSimulator.Benchmarks.csproj`), které používá vytvořenou hru pro měření výkonu ECS knihoven. Projekt hry se skládá ze tří částí. První částí jsou implementace ECS knihoven, které implementují abstrakční vrstvu. Nad nimi je samotná abstrakční vrstva, která je druhou částí. Třetí částí je hra samotná, která namísto konkrétní ECS knihovny používá abstrakční vrstvu.

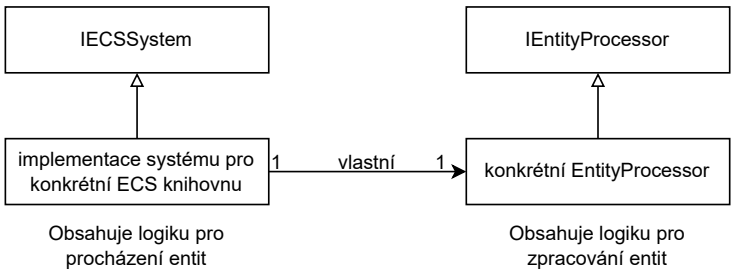
4.1 Abstrakční vrstva

V jmenném prostoru `WorldSimulator.ECS.AbstractECS` se nachází abstrakční vrstva. V jmenném prostoru `WorldSimulator.ECS` kromě abstrakční vrstvy lze také nalézt jednotlivé její implementace.

4.1.1 Systémy

V ECS jsou systémy zodpovědné za iterace a zpracování entit. Jelikož každá ECS knihovna iteruje přes entity jiným způsobem, a zároveň chceme definovat zpracování entit (herní logiku) pouze jednou, bude nutné implementaci rozdělit do několika tříd. Vztah tříd, které souvisejí se systémy, je možné vidět na obrázku 4.1.

`IECSSystem` je rozhraní, ze kterého dědí každý systém. Každá ECS knihovna poté poskytne svou implementaci systému. Tyto implementace obsahují pouze logiku pro iteraci neboli procházení přes jednotlivé entity. Každá implementace systému bude vlastnit konkrétní `EntityProcessor` dědicí od `IEntityProcessor`.



Obrázek 4.1 Vztahy tříd souvisejících se systémy.

Jednotlivé `EntityProcessor` jsou zodpovědné za zpracování entit. Jednotlivé implementace rozhraní `IECSSystem` jsou generické typy, které přijímají typ entity procesoru a také typy komponent, které od entit vyžadují jako generické parametry.

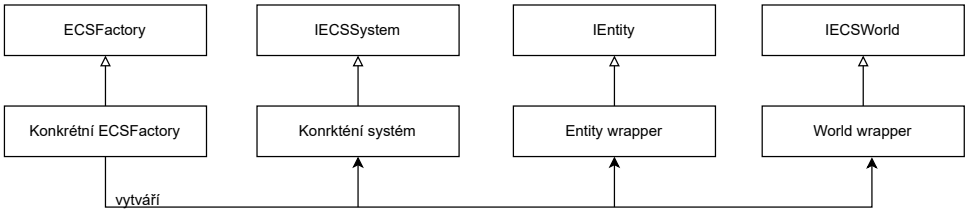
Třídu `ArchSystem<TEntityProcessor, TComponent>` si uvedeme jako příklad. Jedná se o implementaci systému. Tento systém obsahuje logiku pro iteraci entit z ECS knihovny *Arch* [14]. Příkladem konkrétního `EntityProcessor` je například třída `MovementSystem`, která řeší zpracování entit s komponentami `Position` a `Movement` a obsahuje logiku zodpovědnou za jejich pohyb.

Z pohledu hry jednotlivé implementace `EntityProcessor` představují samotné systémy, jelikož hra nekouká na to, jak jsou entity iterovány, a zajímá jí jenom samotná herní logika. Proto se kód, který pracuje s hrou, odkazuje na `EntityProcessor` jako na systém. Ovšem z pohledu abstrakční vrstvy je nutné rozlišovat `System` jako typ, který obsahuje logiku pro iteraci entit, a `EntityProcessor` jako typ, který obsahuje logiku pro zpracování entit. Autor si je vědom, že tyto pohledy mohou vést k matoucí terminologii, ovšem v kódu jsou již používány na příliš mnoho místech.

4.1.2 ECSFactory

`ECSFactory` je abstraktní třída. Typy, které z ní dědí, jsou zodpovědné za tvorbu instancí souvisejících s abstrakční vrstvou. Vztah typů souvisejících s `ECSFactory` je možné vidět na obrázku 4.2.

Typ, který dědí od `ECSFactory`, musí implementovat metody pro vytváření instancí wrapperu entity a wrapperu *worldu*. Skrze konstruktor svému předku také předá typy jednotlivých systémů (tříd dědicích od `IECSSystem`). Wrapper entity je třída dědicí od rozhraní `IEntity` a je zodpovědná za přidávání a odebírání komponent na dané entitě. Wrapper *worldu* je třída dědicí od rozhraní `IECSWorld`. Tento wrapper implementuje metodu `Update`, která je zodpovědná za aktualizaci stavu *worldu*.



Obrázek 4.2 Vztahy tříd souvisejících s `ECSFactory`.

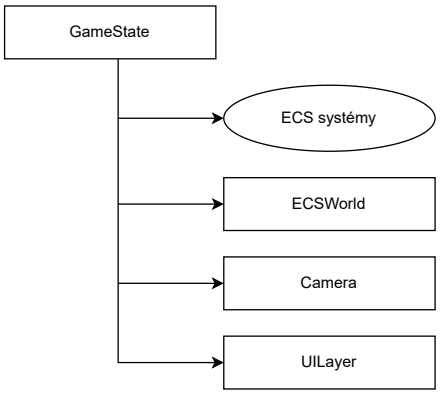
4.2 Hra

Celá hra je reprezentována třídou `Game`, té je skrze konstruktor předána konkrétní instance `ECSFactory`. Jednotlivé instance `ECSFactory` reprezentují jednotlivé ECS knihovny, a volba této instance představuje volbu ECS knihovny, se kterou bude hra pracovat.

První herní stav je hře předán skrze metodu `SwitchState`. Samotná hra je spuštěna metodou `Run`.

4.2.1 Herní stavy

Celá hra se může skládat z několika herních stavů. Každý herní stav je reprezentován třídou dědicí od `GameState` a představuje herní obrazovku. Hra momentálně obsahuje pouze herní stav `LevelState`, který reprezentuje herní obrazovku obsahující gameplay, ale díky abstrakci herních stavů by bylo snadné ji rozšířit například o herní obrazovku herního menu nebo herní obrazovku s nastavením hry. Vztahy tříd souvisejících s třídou `GameState` lze vidět na obrázku 4.3.



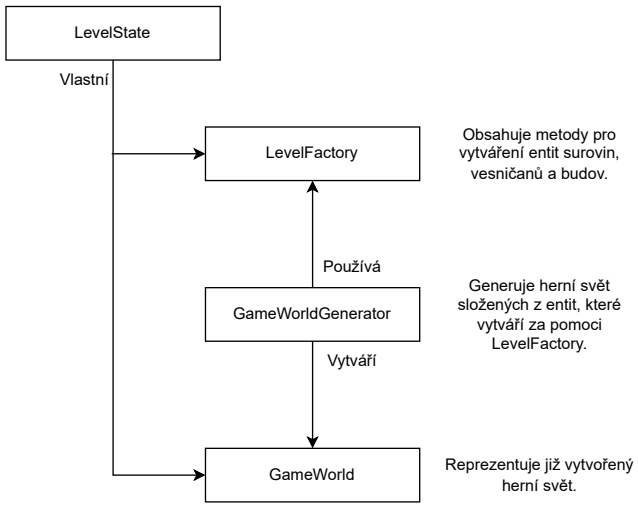
Obrázek 4.3 Vztahy tříd souvisejících s `GameState`.

Každý herní stav obsahuje svoji instanci `ECSWorld`, která reprezentuje *world* herního stavu. Tím pádem každý herní stav má svoji množinu entit. Dále má každý herní stav svoje systémy, kameru a instanci třídy `UILayer`, která spravuje prvky uživatelského rozhraní pro daný herní stav. Konkrétní herní stavy (instance tříd dědicích od `GameState`) jsou zodpovědné za vytváření svých entit, systémů a prvků uživatelského rozhraní.

V jeden moment může být aktivní pouze jeden herní stav. Ten lze zvolit za pomoci metody `SwitchState` na třídě `Game`. Vždy jsou aktivní pouze systémy v aktivním herním stavu.

4.2.2 LevelState a herní svět

`LevelState` je herní stav, který reprezentuje herní obrazovku obsahující samotný gameplay. Obrázek 4.4 popisuje vztahy na této třídě.

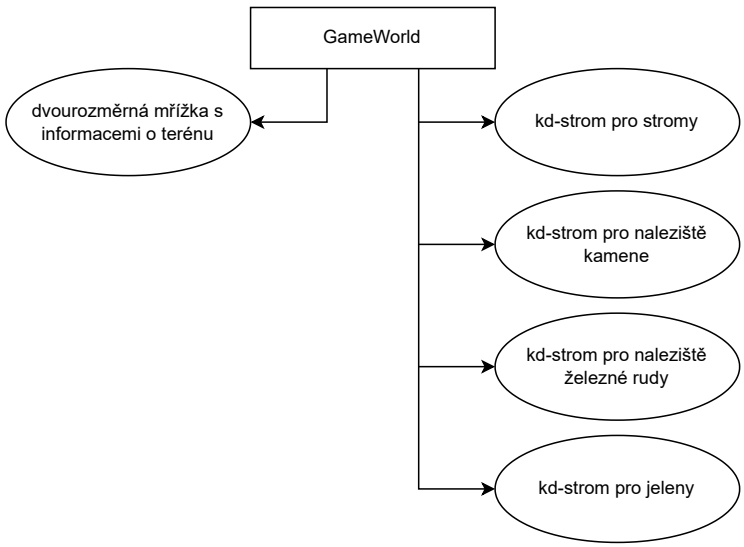


Obrázek 4.4 Vztahy tříd souvisejících s `LevelState`.

Třída `LevelFactory` obsahuje metody pro vytváření entit surovin, budov a vesničanů. Tu využívá třída `GameWorldGenerator` pro tvorbu entit, ze kterých generuje herní svět. Ta se krom vytváření samotného terénu stará také o počáteční rozpoložení surovin a vesnic. Výsledný vygenerovaný herní svět je poté reprezentován třídou `GameWorld`.

Pomocí třídy `GameWorld` lze získávat informace o herním světě. Je možné nalézt nejbližší surovinu určitého typu k dané pozici, nalézt cestu mezi dvěma body nebo získat informace o terénu na dané pozici.

Na obrázku 4.5 lze vidět, že `GameWorld` obsahuje kd-strom pro každý typ suroviny. Pomocí této datové struktury lze efektivně nalézt nejbližší surovinu k dané pozici. Těmto kd-stromům se budeme více věnovat v sekci 4.2.8. `GameWorld` v sobě také obsahuje dvourozměrnou mřížku s informacemi o terénu, která je používána například pro hledání cest.



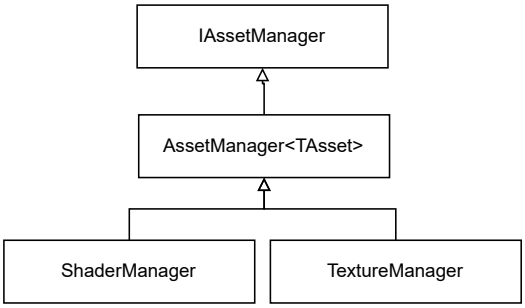
Obrázek 4.5 Datové struktury obsažené v `GameWorld`.

Instance jednotlivých kd-stromů z obrázku 4.5 jsou uchovávány v `Dictionary` uvnitř třídy `GameWorld`. Toto `Dictionary` mapuje typ suroviny (instanci třídy `ResourceType`) na konkrétní kd-strom. Díky tomuto `Dictionary` můžeme snadno získat kd-strom pro konkrétní typ suroviny.

4.2.3 Assety

Na obrázku 4.6 lze vidět hierarchii tříd souvisejících s managementem assetů. `AssetManager<TAsset>` obsahuje společnou logiku pro nalezení a načtení assetů ze složky a pro správu slovníku, ve kterém jsou uloženy načtené assety pro generický typ `TAsset`. Jednotliví potomci této třídy poskytují implementace metody `Load` pro načtení assetu konkrétního typu. I přesto, že hra momentálně obsahuje pouze assety pro textury a shadery, bylo by díky abstrakci asset managerů snadné hru rozšířit i o jiné typy assetů, jako jsou například zvuky.

Všechny assety se nachází ve složce `Content`. Jednotlivé typy assetů mají poté v této složce svou podsložku. Textury se nacházejí v podsložce `Textures` a



Obrázek 4.6 Hierarchie dědičnosti tříd souvisejících s managementem assetů.

shadery v podsložce `Shaders`. Je nutné zmínit, že složka `Shaders` obsahuje již zkompilované shadery.

Veškeré instance rozhraní `IAssetManager` jsou uchovány ve třídě `Game`. Pomocí ní je také možné přistupovat k konkrétním instancím (`ShaderManager` a `TextureManager`). Při inicializaci hry dojde k zavolání metody `LoadAll` na každé takové instanci. To vede k načtení všech assetů daného typu.

4.2.4 Správa managed typů

Vytváříme hru, abychom ji později mohli používat pro měření výkonu ECS knihoven. Aby hru pro měření bylo možné použít, je klíčové, aby byla nezávislá na konkrétní ECS knihovně. Z tohoto důvodu hra namísto konkrétní ECS knihovny používá abstrakční vrstvu. V sekci 3.2.5 jsme si zmínili, že jednotlivé komponenty v abstrakční vrstvě reprezentujeme jako struktury. Ovšem jedna z knihoven, které budeme měřit (konkrétně *Svelto.ECS* [26]), vyžaduje, aby komponenty byly *unmanaged typy*. V této sekci se zaměříme na to, co to jsou *unmanaged typy*, a jak jsme tento problém vyřešili.

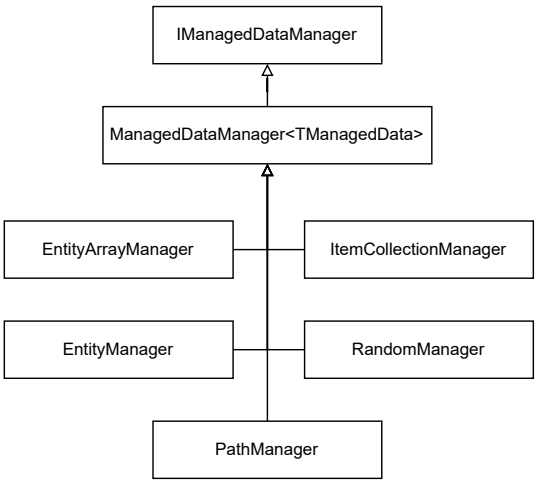
Definici *unmanaged typů* lze nalézt ve článku *Unmanaged types (C# reference)* [27] od firmy *Microsoft*. Ve článku se lze dočíst, že *unmanaged typem* je každý z následujících datových typů:

1. `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `nint`, `nuint`, `char`, `float`, `double`, `decimal`, nebo `bool`.
2. Jakýkoliv výčtový typ.
3. Jakýkoliv typ ukazatele.
4. Jakýkoliv tuple složený z *unmanaged typů*.

5. Jakákoliv uživatelem definovaná struktura, která obsahuje pouze fieldy *unmanaged typů*.

Naše komponenty jsou uživatelem definované struktury, tím pádem je pro nás klíčový poslední bod této definice. Konkrétně naše komponenty nesmí obsahovat jiné fieldy, než fieldy *unmanaged typů*. Ovšem v některých případech bychom takové fieldy potřebovali. Jedná se zejména o instance třídy *IEntity* a jakékoliv pole nebo kolekce. Například entity vesničanů v naší hře obsahují `Villager` komponentu, ve které si uchovávají referenci na entitu vesnice, do které spadají. Dalším příkladem je *PathFollow* komponenta, která přidá entitě schopnost chodit po cestě, která je v této komponentě definována. Cesta je definována za pomocí pole pozic.

Pro řešení tohoto problému byli zavedeny manažery instancí *managed typů*. Na obrázku 4.7 lze vidět hierarchii dědičnosti tříd souvisejících s těmito manažery. Prázdný interface `IManagedDataManager` slouží, abychom mohli instance jednotlivých manažerů spolu ukládat do jedné kolekce. Od tohoto interface dědí abstraktní třída `ManagedDataManager<TManagedData>`, kde generický parametr `TManagedData` představuje *managed typ*, jehož instance chceme spravovat. Od této třídy dědí třídy, které se starají o správu konkrétních *managed typů*.



Obrázek 4.7 Hierarchie dědičnosti tříd souvisejících se správou instancí *managed typů*.

Třída `ManagedDataManager<TManagedData>` obsahuje abstraktní metody pro vytváření instancí spravovaného *managed typu*. V jednotlivých manažerech jsou uchovávány spravované instance odpovídajícího *managed typu*. Do manažera je možné instancí vložit nebo vytvořit novou instanci přímo uvnitř manažera. Obě

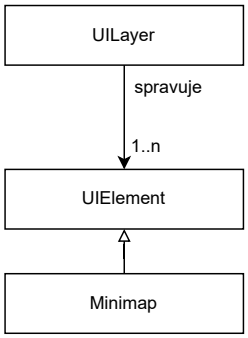
tyto akce nám navrátí ID, za pomoci kterého lze k instancím daného *managed typu* přistupovat. V našich komponentech namísto instancí těchto *managed typů* uchováváme jejich ID.

V případě, že chceme odebrat danou komponentu obsahující ID instance *managed typu*, je nutné odpovídající instanci odebrat z manažera, aby nedošlo k *memory leaku*. V naší hře toto řešíme za pomoci `DeathSystem`, který řeší smrt entit. V případě smrti entity, tento systém před odstranění dané entity zkontroluje, zda na sobě daná entita nemá komponentu obsahující ID instance *managed typu*. Pokud ano, tak příslušnou instanci odebere z příslušného manažera.

Instance jednotlivých manažerů uchováváme ve třídě `Game`. Pomocí této třídy je možné k nim také přistupovat. Třídy pro typy jednotlivých manažerů je možné vidět na obrázku 4.7.

4.2.5 Uživatelské rozhraní

Vztahy tříd souvisejících s uživatelským rozhraním je možné vidět na obrázku 4.8. Třída `UILayer` je zodpovědná za správu prvků uživatelského rozhraní pro jeden konkrétní herní stav. Každý prvek uživatelského rozhraní dědí od třídy `UIElement`. Hra momentálně obsahuje pouze jediný prvek uživatelského rozhraní a tím je minimapa (reprezentovaná třídou `Minimap`), ale je navržena tak, aby bylo snadné ji rozšířit i o další prvky uživatelského rozhraní, jako jsou například labely, tlačítka, nebo action bar.



Obrázek 4.8 Diagram vyobrazující vztahy tříd, které souvisejí s uživatelským rozhraním.

Mezi prvky uživatelského rozhraní existuje stromová hierarchie. Každý prvek může mít několik potomků. Jednotliví potomci poté pracují s pozici relativní vůči svému rodiči.

Minimapa

Minimapa je reprezentována třídou `Minimap`, která dědí od `UIElement`. Pomocí minimapy chceme znázornit, jak vypadá herní svět a jaká jeho část je momentálně viditelná kamerou. Jejím cílem je usnadnit hráči orientaci v herním světě.

Minimapa se skládá z mapy terénu, view framu a rámečku. Pro vykreslení mapy terénu používáme stejný fragment shader jako pro vykreslení terénu naší hry. Pouze použijeme jiné rozlišení a scale. Poté vykreslíme view frame, ten představuje výsek terénu, který hráč momentálně vidí skrze kameru. Pro jeho vykreslení si najdeme na jakou pozici v minimapě odpovídá umístění kamery a poté vykreslíme obdélníkový rámeček, který odpovídá rozměrům kamery naškálovaných podle velikosti minimapy. Aby nám view frame nepřesahoval hranice minimapy, vykreslujeme jej s nastaveným `ScissorRectangle`. Jedná se o MonoGame nastavení, pomocí kterého lze provádět oříznutí. Na závěr kolem minimapy vykreslíme rámeček. Ten je reprezentován texturou.

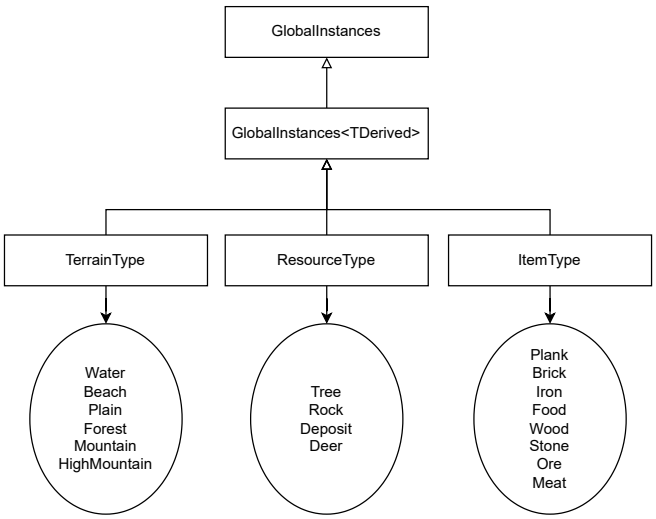
4.2.6 Herní data

Mezi herní data patří informace o biomech, surovinách a předmětech. Herní data jsou immutable a lze k nim přistupovat globálně. Vztahy tříd souvisejících s herními daty je možné vidět na obrázku 4.9.

`GlobalInstances<TDerived>` je abstraktní třída, kde generický parametr `TDerived` může být pouze typ, který je jejím potomkem. Tato třída si za pomoci statiky interně ukládá všechny instance `TDerived` a umožňuje k nim globální přístup za pomoci jejich identifikátoru.

Jednotliví potomci třídy `GlobalInstances<TDerived>` představují jednotlivé typy herních dat, jako je například typ biomu (`TerrainType`), typ suroviny (`ResourceType`), nebo typ předmětu (`ItemType`). Tyto třídy obsahují informace o daném typu herních dat. Například typ biomu (`TerrainType`) obsahuje informaci o tom, zda je možné v daném biomu stavět, a také informaci o tom, zda je možné v daném biomu chodit.

Je nutné, aby jednotliví potomci třídy `GlobalInstances<TDerived>` měli privátní konstruktor. Tím zaručíme, že nebude možné vytvářet nové instance mimo danou třídu. Všechny instance těchto potomků jsou definovány jako veřejné readonly statické fieldy přímo uvnitř těchto tříd. Například herní data reprezentovaná třídou `TerrainType` obsahují instance pro biom vody (`Water`), biom pláže



Obrázek 4.9 Diagram vyobrazující třídy a důležité instance související s herními daty.

(`Beach`), biom lesa (`Forest`) a další biomy. Na obrázku 4.9 lze vidět vypsané jednotlivé instance jednotlivých typů herních dat.

Abstraktní třída `GlobalInstances` obsahuje pouze module initializer, pomocí kterého zavolá `RuntimeHelpers.RunClassConstructor` na každém neabstraktním typu, který od ní dědí. Tím máme zaručeno, že naše data jsou v čas vytvořené, jinak by k jejich vytvoření došlo až při prvním přístupu k jejich třídám a do té doby bychom k nim nemohli přistupovat skrze třídu `GlobalInstances<TDerived>`.

4.2.7 Inventáře a předměty

Na konci sekce 2.2.1 jsme si uvedli jednotlivé typy předmětů, které jsou součástí naší hry. Jednotlivé předměty mohou mít vesničané ve svém inventáři (například pokud je získali sklizením suroviny a nyní je odnášejí, aby je zpracovali). Dále jsou předměty skladovány ve skladišti. Vztahy tříd související s předměty je možné vidět na obrázku 4.10.

Obrázek 4.10 Vztah tříd souvisejících s herními předměty.

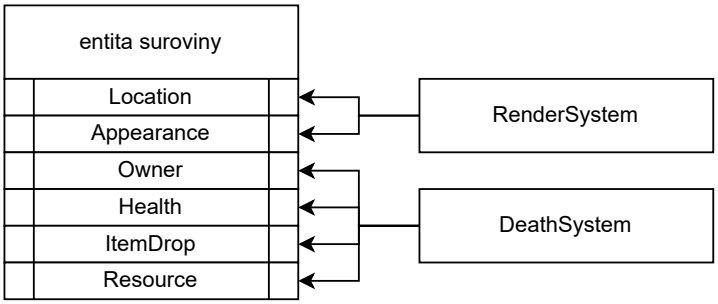
Z minulé sekce víme, že instance třídy `ItemType` reprezentují jednotlivé typy předmětů. Každý typ předmětu má také své ID. Struktura `ItemCollection` reprezentuje kolekci předmětů. Uvnitř sebe má pole, kde jednotlivé prvky tohoto

pole reprezentují počty předmětů uvnitř kolekce. Index každého prvku odpovídá ID daného typu předmětu. Například předmět *prkno* má ID 0 a element z tohoto pole s indexem 0 odpovídá počtu prken uvnitř dané kolekce. Kromě tohoto pole struktura `ItemCollection` také obsahuje pomocné metody pro manipulaci s kolekcemi předmětů.

Třída `Inventory` je komponenta. Entity s touto komponentou mají inventář, do kterého mohou ukládat předměty. Tato třída obsahuje jediný field a tím je ID na instanci `ItemCollection`. Tuto komponentu na sobě kromě entit vesničanů a skladišť mají také entity budov pro zpracování předmětů (například budova dřevorubce). Do tohoto inventáře přesouvají vesničané předměty pro zpracování, a entita dané budovy pro zpracování surovin zde také ukládá produkt získaný zpracováním daného předmětu.

4.2.8 Suroviny

Jednotlivé suroviny jsou reprezentovány jako entity. Komponenty, ze kterých jsou tyto entity složené, a systémy, které tyto komponenty zpracovávají, je možné vidět na obrázku 4.11. `RenderSystem` se stará o vykreslení entity. `Appearance` komponenta nese informaci o tom, jak entita vypadá, a `Location` obsahuje informaci o tom kde se nachází. `DeathSystem` řeší logiku smrti dané entity, konkrétně v případě surovin dojde k jejich smrti v případě sklizení.



Obrázek 4.11 Entita suroviny s jejími komponentami a systémy, které tyto komponenty zpracovávají.

Každá surovina má `Health` komponentu, ve které je uložena informace o aktuálním počtu životů. Entita obsahující `DamageDealer` komponentu může zaútočit na entitu obsahující `Health` komponentu. Při tomto útoku se počet životů v příslušné `Health` komponentě sníží a pokud klesne na nulu, tak `DeathSystem` kromě smazání mrtvé entity nahlédne, zda zemřelá entita obsahuje `ItemDrop` komponentu. Pokud ano, tak předměty definované v této komponentě převede

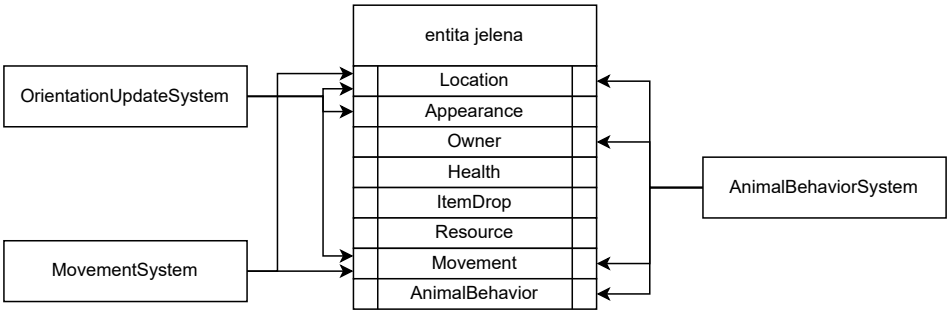
do inventáře entity, která danou entitu zabila. Stejným způsobem funguje i sklízení surovin, kde útočící entitou je vesničan, který útočí na entitu dané suroviny a tím ji sklízí. Po jejím zabití neboli sklizení obdrží do svého inventáře příslušné předměty.

Suroviny na sobě mají také `Owner` komponentu. Ta obsahuje instanci typu dědícího od `IEntity`, která reprezentuje danou entitu. `DeathSystem` vyžaduje tuto komponentu, aby mohl danou entitu smazat.

Jak již bylo zmíněno, existuje kd-strom pro každý typ suroviny. Jedná se o datovou strukturu, pomocí které lze efektivně vyhledávat v prostoru. Entity jednotlivých surovin jsou uloženy v těchto kd-stromech. Tyto kd-stromy poté využívají vesničané, kteří v rámci svého povolání provádějí úlohu nalezení nejbližší suroviny daného typu. Pokud si vesničan rezervuje danou surovinu (začne pohyb směrem k ní a následně ji sklidí), dojde k odebrání této suroviny z příslušného kd-stromu. Tím zamezíme tomu, že více vesničanů nebude sklízet stejnou entitu suroviny. Pro implementaci kd-stromu používáme Nugget balíček *KdTree* [28].

Dále je nutné zmínit, že `DeathSystem` pracuje nad všemi entitami s komponentami `Health` a `Owner`. V případě, že zemřelá entita má na sobě komponentu `Inventory` nebo komponentu `ItemDrop`, tak převede dané předměty. A v případě, že zemřelá entita je vesničan a ten útočil na entitu s komponentou `Resource` (daná entita byla surovina), tak je nutné entitu vrátit do příslušného kd-stromu.

Entita *jelena* se od ostatních entit surovin liší. Její komponenty a systémy, které zpracovávají tyto komponenty, je možné vidět na obrázku 4.12. Je možné nahlédnout, že oproti entitám ostatních surovin má entita *jelena* navíc komponenty `Movement`, která přidává entitám schopnost chodit a `AnimalBehavior`. `AnimalBehaviorSystem` poté pracuje se všemi entitami s komponentami `Location`, `Owner`, `Movement` a `AnimalBehavior` a jeho hlavním úkolem je řídit chování zvířat, konkrétně jejich náhodný pohyb. Krom tohoto úkolu se tento systém stará také o aktulizaci pozice entit zvířat v příslušném kd-tree.



Obrázek 4.12 Entita jelena s jejími komponentami a systemy `AnimalBehaviorSystem`, `MovementSystem` a `OrientationUpdateSystem`.

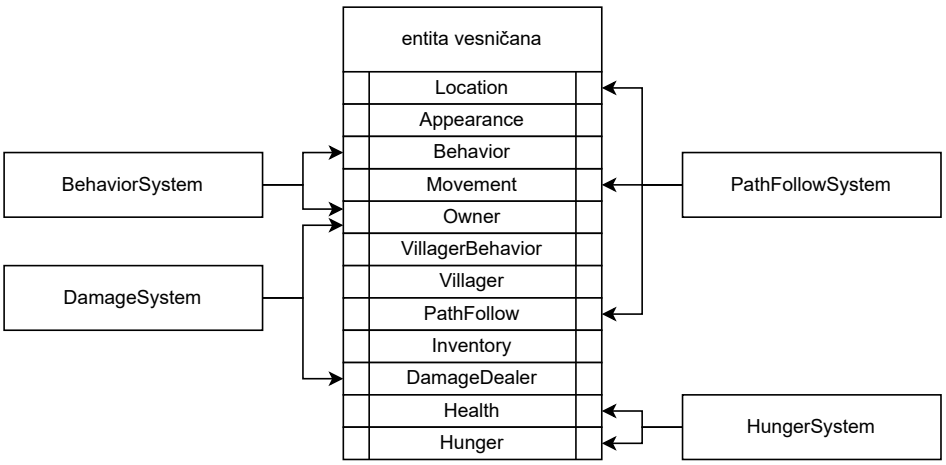
`MovementSystem` se stará o pohyb entit. Jak `MovementSystem`, tak `AnimalBehaviorSystem`, pracují s `Movement` komponentou, ale `AnimalBehaviorSystem` řídí, jak se pohyb bude odehrávat a `MovementSystem` se stará o samotný pohyb. `OrientationUpdateSystem` aktualizuje orientaci entity. V závislosti na tom, zda se entita pohybuje doleva nebo doprava, tak převrátí texturu dané entity podle osy y, aby její orientace odpovídala směru, ve kterém se pohybuje.

4.2.9 Vesnice a vesničané

Ze sekce 2.2.2 víme, že v herním světě se budou nacházet vesničané a vesnice. Jednotlivé vesnice budou obsahovat budovy. Každá budova, každý vesničan a i samotná vesnice jsou reprezentovány jako entity. V této sekci si tyto entity představíme.

Vesničané

Diagram s entitou vesničana je možné vidět na obrázku 4.13. Kromě vyobrazených systémů souvisí s vesničanem také `MovementSystem`, `DeathSystem`, `OrientationUpdateSystem` a `RenderSystem`. Tyto systémy byly popsány v minulé sekci a nyní se jim nebudeme věnovat.



Obrázek 4.13 Entita vesničana s jejími komponentami a systemy `BehaviorSystem`, `PathFollowSystem`, `DamageSystem` a `HungerSystem`.

Již zmiňovaný `DamageSystem` řeší logiku poškození. Z minulé sekce víme, že díky tomuto systému je vesničanům umožněno sklízet suroviny. `HungerSystem` řeší logiku hladovění. V `Hunger` komponentě je uložen aktuální stav hladovění

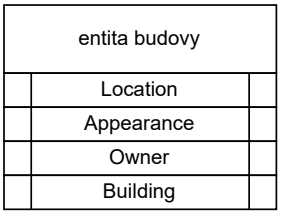
vesničana. Tento stav neustále roste a pokud přesáhne určitou hranici, vesničan začne ztrácet životy. Pokud vesničan sní jídlo, stav hladovění se nastaví na nulu.

`BehaviorSystem` se stará o chování vesničanů. Ze sekce 3.3 víme, že pro chování vesničanů používáme *strom chování*. Každý vesničan má právě jeden *strom chování* a `BehaviorSystem` pouze volá metodu *Tick* na daném *stromě chování*, která jej zpracovává.

Některé části z tohoto *BehaviorTree* každého vesničana se starají o generování cest. Například pokud vesničan potřebuje dojít k entitě dané suroviny. Vygenerovaná cesta je poté uložena do `PathFollow` komponenty. O logiku chození po cestě se stará `PathFollowSystem`. Je nutné zmínit, že samotný pohyb vždy řeší `MovementSystem`, `PathFollowSystem` pouze nastavuje hodnoty v `Movement` komponentě na základě cesty uložené v `PathFollow` komponentě.

Budovy

Diagram entity budovy je možné vidět na obrázku 4.14. Vyobrazené komponenty má na sobě každá budova. V komponentě `Village` je uchováván odkaz na instanci vesnice, do které budova spadá. Budova skladiště má na sobě navíc `Inventory` komponentu, ve které ukládá suroviny, které vesnice vlastní. Hlavní budova, oproti diagramu, neobsahuje žádné dodatečné komponenty.



Obrázek 4.14 Entita budovy s jejími komponentami.

Důležitým typem budov jsou budovy pro zpracování surovin. Diagram vyobrazující jejich komponenty a související systémy je možné vidět na obrázku 4.15. Je možné vidět, že oproti entitě z obrázku 4.14 má budova pro zpracování surovin navíc komponenty `Inventory`, `ResourceProcessor` a `VillagerSpawner`.

`VillagerSpawningSystem` je zodpovědný za vytváření nových vesničanů. Ke každé budově pro zpracování surovin je přidělen právě jeden vesničan, a pokud tento vesničan zemře, tak se spustí odpočet. Po uplynutí tohoto odpočtu dojde k vytvoření nového vesničana. Při jeho vytvoření mu bude zkonstruován a přidělen *strom chování* a příslušná budova mu bude nastavena jako pracovní místo.

Obrázek 4.15 Entita budovy pro zpracování surovin s jejími komponentami a systémy `ResourceProcessingSystem` a `VillagerSpawningSystem`.

`ResourceProcessingSystem` se stará o logiku kolem zpracování surovin. Budovy pro zpracování surovin na sobě mají `Inventory` komponentu. `ResourceProcessingSystem` má na sobě nastavený recept a pravidelně nahlíží, zda se v inventáři dané suroviny nacházejí požadované suroviny. Pokud ano, tak zahájí zpracování, po jehož dokončení přemístí nově vytvořené předměty opět do inventáře. Z něj je může vesničan převzít a odnést do skladiště.

Vesnice

Každá vesnice je entita, která obsahuje komponenty `Location`, `Village` a `Owner`. `VillageBuildingSystem` se stará o stavění budov v dané vesnici. Má v sobě uloženou stavební frontu budov, kde pro každou budovu má také její cenu. Ve `Village` komponentě jsou uchovávány odkazy na jednotlivé entity budov, které se ve vesnici nacházejí. Jakmile `VillageBuildingSystem` zjistí, že vesnice má dostatek surovin na další budovu ze stavební fronty, vytvoří pro ni entitu a přidá na vytvořenou entitu odkaz do `Village` komponenty.

Klíčovou částí při vytváření entity budovy je volba pozice, kde se entita má vytvořit. Nechceme, aby budovy byly umístěny v biomech, kde nelze stavět, a také nechceme, aby se nacházely moc blízko nebo moc daleko od ostatních budov. Pro volbu pozice budov je zvolena konstantní minimální a maximální vzdálenost. Tyto vzdálenosti udávají, v jakém rozmezí se mohou nové budovy od již postavených budov objevit.

Pro volbu pozice byl použit přímočarý algoritmus. Jako první se vybere náhodná již postavená budova a vezme se její pozice. Poté se kolem této pozice vytvoří prstenec s rozměry podle konstantní minimální a maximální vzdálenosti. V tomto prstenci se poté vybere náhodná pozice. Na závěr se zkontroluje, zda je možné na této pozici stavět, a pokud ano, tak je tato pozice vybrána jako pozice pro novou budovu. Pokud na dané pozici není možné stavět, tak se celý algoritmus zopakuje.

4.3 Měření

Měření je druhým C# projektem (`WorldSimulator.Benchmarks.csproj`) našeho C# solution. S použitím naší hry měří výkon jednotlivých ECS knihoven. V sekci 3.5.2 jsme se rozhodli, že pro toto měření použijeme *BenchmarkDotNet*. Celé měření se dělí na dvě fáze, a to přípravy a samotného měření.

Pro měření máme definované následující konstanty:

1. `seed`: Seed pro generování náhodných čísel. Stejným seedem pro každé měření zajistíme konzistentnost.

2. `deltaTime`: Čas v sekundách. Tento čas používáme pro odsimulovaní jednoho kroku naší hry (jedné iterace *game loop*, který byl přiblížen v sekci 3.5.1).

3. `setupIterationCount` Udává počet kroků odsimulování naší hry během fáze přípravy.

4. `benchmarkIterationCount`: Udává počet kroků odsimulování hry během fáze měření.

V C# projektu (`WorldSimulator.Benchmarks.csproj`) máme definovaný test reprezentovaný třídou `ECSBenchmarks`. Tento test obsahuje jediný vstupní parametr, a tím je `ECSFactoryType` typu `Type`. Tímto parametrem jsou předávány typy dědící od `ECSFactory`, které reprezentují jednotlivé ECS knihovny.

Během fáze přípravy nejprve zkonstruujeme konkrétní `ECSFactory`. Poté vytvoříme novou instanci naší hry (třídy `Game`) a odsimulujeme jeden její snímek za pomocí `RunOneFrame`, tím zaručíme, že hra bude inicializována. Poté v cyklu voláme metodu `UpdateOnce`, která vykoná jednu iteraci *game loop* naší hry. Počet opakování tohoto cyklu udává konstanta `setupIterationCount`.

Během fáze měření používáme tu samou instanci hry vytvořenou během fáze přípravy. Na této instanci v cyklu opět zavoláme metodu `UpdateOnce`. Počet opakování tohoto cyklu je dán konstantou `benchmarkIterationCount`.

Kapitola 5

Měření

Náš projekt se skládá ze dvou částí a to hry a poté měření, které využívá vytvořenou hru pro měření výkonu jednotlivých ECS knihoven. Analýzu tohoto měření jsme provedli v sekci 3.5 a jeho implementaci jsme popsali v sekci 4.3. V této kapitole si nejprve ukážeme, jak lze měření spustit, následně si stanovíme hypotézu, poté si přiblížíme ECS knihovny, které budeme měřit, a na závěr si prezentujeme výsledky tohoto měření.

5.1 Spouštění měření

Mezi přílohami projektu lze najít adresář `src` obsahující zdrojové soubory a sestavený projekt měření. Pomocí aplikace `src\WorldSimulator.BenchMarks\bin\Release\net7.0\WorldSimulator.Benchmarks.exe` lze spustit měření. Pro úpravu parametrů měření je potřeba upravit a znovu sestavit zdrojový kód. Fieldy relevantní k měření je možné nalézt v tabulce 5.1. Pro každý field je v této tabulce zobrazen jeho název a poté jeho popis. První čtyři z těchto fieldů se nacházejí ve třídě `WorldSimulator.BenchMarks.ECSBenchmarks`. Poslední z nich se nachází ve třídě `WorldSimulator.Level.World`.

název fieldu	popis fieldu
`seed`	Seed pro generování náhodných čísel.
`deltaTime`	Odsimulovaný čas mezi dvěma iteracemi *game loop*.
`setupIterationCount`	Počet iterací *game loop* během warmup fáze.
`benchmarkIterationCount`	Počet iterací *game loop* během měření.
`Size`	Výška a šířka herního světa.

Tabulka 5.1 Seznam fieldů relevantních k měření.

5.2 Hypotéza

V této sekci si stanovíme hypotézu našeho měření. V našem měření měříme, jak dlouho trvá odsimulovat určitý počet iterací naší hry s jednotlivými ECS knihovnami. Ty se dají rozdělit na kategorie podle určitých vlastností. V této kapitole stanovíme, jak si myslíme, že se jednotlivé kategorie umístí.

5.2.1 Cache

Využití cache je klíčový faktor pro výkon ECS knihovny, proto si nyní připomeneme, co to vlastně je a jak funguje.

Program si svoje data uchovává v paměti RAM. Ovšem přístup do této paměti je z hlediska času drahý. Pro představu u moderních pamětí takovýto přístup může trvat vyšší desítky nanosekund. Z tohoto důvodu se využívají menší paměti, takzvané cache, které disponují mnohem vyšší rychlostí, ale mnohem menší velikostí. Pro představu moderní procesory používají cache, u kterých přístup může trvat menší desetiny nanosekund.

Při přístupu k datům se procesor nejprve podívá, zda nemá data již v této cache. Pokud ano, jedná se o takzvaný *cache hit* a data si z ní načte. Pokud ne, jedná se o takzvaný *cache miss* a je nutné data načíst z hlavní paměti.

Již víme, že *cache miss* jsou drahé, proto je potřeba pro vysoký výkon je minimalizovat. Ale jak přesně procesor rozhoduje o tom, jaká data budou v cache a jaká ne? Využívá se takzvané *časové* a *prostorové lokality*. *Časová lokalita* spočívá v tom, že pokud jsme přistoupili k nějakým datům, je velká šance, že k nim brzy budeme chtít přistoupit znovu. Na druhou stranu *prostorová lokalita* spočívá v tom, že pokud jsme přistoupili k nějakým datům, je velká šance, že budeme chtít přistoupit také k datům, které se nacházejí blízko nich.

Jedna z možností, jak minimalizovat počet *cache miss*, je využít sekvenčního přístupu. Pokud budeme přistupovat k datům, které jsou v paměti hned za sebou, tak díky *prostorové lokalitě* bude počet *cache miss* velmi malí.

Je nutné upozornit, že výše popsaný model cache je úmyslně zjednodušen. Velice detailní popis cache lze nalézt v *Cache Memories* [29] od *Alana Jaye Smitha*. Stručnější popis a také informace o tom, jak lze cache využít, lze nalézt v již zmiňované knize *Game Programming Patterns* [3], konkrétně v kapitole *Data Locality*.

5.2.2 Arch type

Některé ECS knihovny pro lepší výkon používají *arch type*. Jedná se o datovou strukturu pro ukládání komponent. Knihovny, které používají tuto datovou

strukturu, mají většinou velký výkon, proto před stanovením hypotézy si použití *arch type* lehce přiblížíme.

Jak již bylo zmíněno, *arch type* je datová struktura. Tato datová struktura představuje typ entity. Tento typ je jednoznačně určen typy všech komponent dané entity. Je možné si jej představit jako tabulku, kde jednotlivé sloupečky odpovídají komponentám a v každém řádku se nacházejí instance komponent pro danou entitu. Například můžeme mít *arch type* vyobrazený v tabulce 5.2. Tento *arch type* je jednoznačně určen typy komponent `Position` (představující pozici entity), `Health` (představující počet životů entity) a `Damage` (představující poškození, které entita může udělit jiné entitě).

entita	`Position`	`Health`	`Damage`
hráč	(0, 0)	10	3
nepřítel	(5,4)	6	1
npc	(-4,8)	16	9

Tabulka 5.2 Tabulka vyobrazující *arch type*, který je jednoznačně určen komponentami `Position`, `Health` a `Damage`. Tento *arch type* obsahuje tři entity, konkrétně hráče, nepřítele a npc.

Jednotlivé *arch type* jsou uloženy ve *world*. Každý *arch type* má v sobě několik polí, konkrétně jedno pole pro každý typ komponenty. V každém poli jsou poté uloženy instance příslušných komponent. V případě *arch type* z tabulky 5.2 by tento *arch type* obsahoval jedno pole pro `Position` komponenty, jedno pole pro `Health` komponenty a jedno pole pro `Damage` komponenty.

Pokud bychom chtěli iterovat přes všechny entity s danými komponentami, stačilo by nám projít příslušná pole všech *arch type* s těmito komponentami. Tato iterace by byla, díky sekvenčnímu přístupu zmíněnému v minulé sekci, velice rychlá. Ke *cache miss* by docházelo pouze při přechodu na další *arch type*.

Nevýhodou knihoven založených na *arch type* bývá pomalé přidávání a odebírání komponent. Pokud je entitě přidána nebo odebrána komponenta, dojde ke změně jejího *arch type*. Při této změně se vezmou instance všech komponent dané entity a přesunou se do nového *arch type*.

Pro více informací o *arch type* a o tom, jak je ECS knihovny využívají, je možné nahlédnout do série článků *ECS back and forth* [30] od Michela Cainiho.

5.2.3 Stanovení hypotézy

Nyní stanovíme hypotézu. Jednotlivé ECS knihovny lze rozdělit do kategorií na základě jistých vlastností. My tyto kategorie popíšeme a stanovíme jaké bude jejich pořadí při výkonnostním porovnání.

Nejrychleji vyjdou ECS knihovny, které vyžadují, aby byly komponenty reprezentovány jako struktury a zároveň používají *arch type*. Jak již bylo zmíněno, díky sekvenčnímu přístupu, který *arch type* využívá, dojde k minimalizaci počtu *cache miss*. To má za následek vysoký výkon. Z toho důvodu tyto knihovny vyjdou nejlépe. Označme tuto kategorii jako kategorii 1.

Některé ECS knihovny vyžadují, aby jednotlivé komponenty byly reprezentovány jako třídy. Tyto knihovny vyjdou nejhůře. V případě, že komponenta je třída, znamená to, že její proměnné musejí být pointery. Z toho důvodu iterace přes komponenty v těchto knihovnách vede ve velmi velký počet *cache miss*, kvůli kterému se tyto knihovny umístí nejhůře. Označme tuto kategorii jako kategorii 3.

Zbývají knihovny, které používají *arch type* a zároveň vyžadují, aby komponenty byly reprezentovány jako třídy. Tyto knihovny se umístí hůře než knihovny z kategorie 1, ale lépe než knihovny z kategorie 3. Označme tuto kategorii jako kategorii 2.

5.3 Měřené knihovny

V této sekci se budeme zabývat ECS knihovnami, které budeme měřit. První si uvedeme, odkud měřené ECS knihovny bereme, a poté si jednotlivé ECS knihovny krátce představíme.

V sekci 1.2 jsme zmínili, že existuje repositář *EcsCsharpBenchmark* [5] na platformě GitHub obsahující výkonnostní porovnání ECS knihoven pro C#. Tento repositář byl inspirací této práce, která také porovnává jednotlivé ECS knihovny, ale na místo jednoduchých testů je porovnává na hře. V našem měření budeme měřit knihovny, které byli porovnávány ve zmiňovaném repositáři *EcsCsharpBenchmark* [5].

Od zahájení této práce byl repositář *EcsCsharpBenchmark* [5] několikrát upraven. Během úprav byli přidávány a odebírány některé ECS knihovny z porovnání. V této práci se zaměříme na knihovny, které byly v repositáři *EcsCsharpBenchmark* [5] porovnávány během commitu *db67d1d* [31].

Nyní si představíme jednotlivé ECS knihovny, které budeme měřit. U každého si uvedeme přeložený popisek, který je možné nalézt v repositáři dané knihovny. Tyto popisky jsou doslovně převzaté a autor práce neručí za korektnost informací, které jsou jejich obsahem.

1. **Arch [14]:** Vysoko-výkonnostní ECS knihovna založená na Archtype a Chunks určená pro herní vývoj a data-oriented programování.

2. **DefaultEcs [32]:** ECS framework, který si klade za cíl být přístupný s minimálními omezeními, zatímco zachovává co největší výkon pro vývoj her.

3. **Entitas [33]:** Nejpopulárnější open-source ECS framework pro C# a Unity.
4. **HypEcs [34]:** Lightweight a snadno použitelná ECS knihovna s efektivní sadou funkcionalit pro tvorbu her.
5. **LeoECS [35]:** Lightweight ECS framework pro C#. Mezi hlavní cíle tohoto frameworku patří výkon, nulová nebo minimální alokace, minimalizace využití paměti a absence závislostí na jakémkoliv herním enginu. (Přeloženo pomocí ChatGPT.)
6. **LeoEcsLite [36]:** Lightweight ECS framework pro C#. Mezi hlavní cíle tohoto frameworku patří výkon, nulová nebo minimální alokace, minimalizace využití paměti a absence závislostí na jakémkoliv herním enginu. (Přeloženo pomocí ChatGPT.)
7. **MonoGameExtended.Entities [37]:** Moderní vysoce výkonnostní ECS framework založený na Artemis.
8. **RelEcs [38]:** Lightweight a snadno použitelná ECS knihovna s efektivní sadou funkcionalit pro tvorbu her.
9. **Svelto.ECS [26]:** Reálný ECS framework pro C#. Umožňuje psát zapouzdřený, oddělený, udržovatelný, vysoce efektivní, datově orientovaný, cache-přátelský kód bez bolesti.

V minulé sekci jsme si definovali kategorie, do kterých jednotlivé ECS knihovny spadají. Do kategorie 1 patří *Arch*, *HypEcs* a *Svelto.ECS*. Do této kategorie také zařadíme *DefaultEcs*, *LeoECS* a *LeoEcsLite*, které sice *arch type* nepoužívají, ale používají datovou strukturu, která také využívá sekvenčního přístupu pro minimalizaci počtu *cache miss*. V kategorii 2 je pouze *RelEcs*. Do kategorie 3 spadají *MonoGameExtended.Entities* a *Entitas*.

5.4 Výsledky

Analýzu měření jsme provedli v sekci 3.5. Poté v sekci 4.3 jsme si rozebrali implementaci měření. V této sekci se budeme věnovat výsledkům tohoto měření.

Předtím než si prezentujeme výsledky měření, tak si shrňme, jak jej provádíme. Prvně si vytvoříme novou instanci hry (instanci třídy `Game`) s danou ECS knihovnou (s danou instancí třídy dědící od `ECSFactory`). Poté odsimulujeme přípravný počet iterací (provedeme `setupIterationCount` iterací *game loop* naší hry). Následně odsimulujeme měřený počet iterací (`benchmarkIterationCount`) a přitom budeme měřit čas. To několikrát zopakujeme (přesněji, námi zvolený

framework to za nás několikrát zopakuje) a získáme průměrný čas, jak rychle hra zvládne odsimulovat měřený počet iterací (`benchmarkIterationCount`). Test provádíme pro každou měřenou ECS knihovnu (ty jsme si rozebrali v minulé sekci).

Měření budeme spouštět vícekrát s různými velikostmi herního světa. Velikost herního světa je definována v `GameWorld.Size`. Tento statický field obsahuje výšku a šířku herního světa. V sekci 4.3 jsme si popsali jednotlivé parametry našeho měření, následující tabulka udává jejich nastavení:

název parametru	nastavená hodnota
`seed`	0
`deltaTime`	$1/60$
`setupIterationCount`	$30 * 60$
`benchmarkIterationCount`	$60 * 60$

Tabulka 5.3 Nastavení parametrů našeho měření.

Parametr `deltaTime` je nastaven na $1/60$ s, a to odpovídá 60 FPS (snímkům za sekundu). Poté parametr `setupIterationCount` odpovídá zhruba 30 sekundám simulace, a parametr `benchmarkIterationCount` je přibližně 1 minuta simulace. Měření byla prováděna na počítači s procesorem AMD Ryzen 3 1200 Quad-Core, pamětí 16GB RAM a grafickou kartou NVIDIA GeForce GTX 1650 s pamětí 4 GB VRAM.

5.4.1 První měření

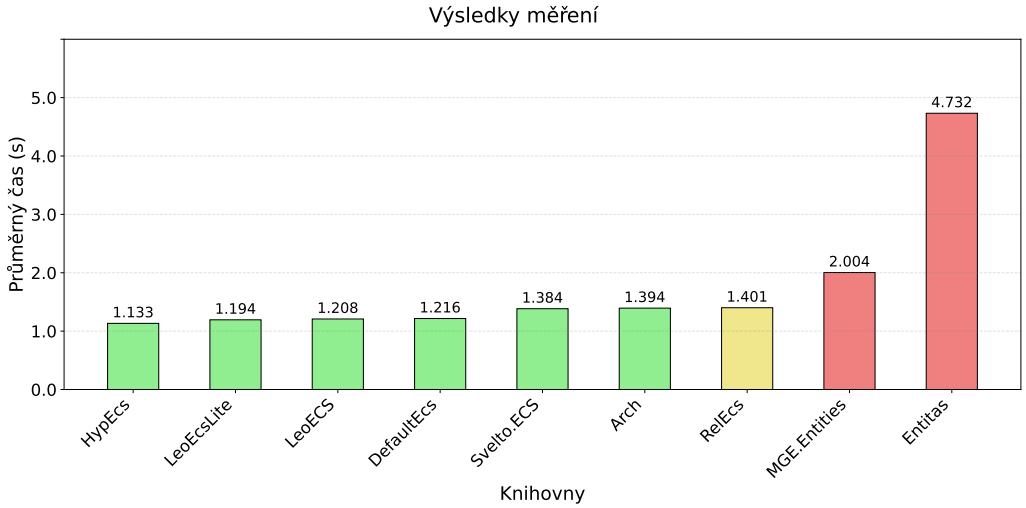
První měření bylo spuštěno s velikostí světa 8192x8192. Výsledky tohoto měření jsou shrnuty v tabulce 5.4, která zachycuje výsledné časy prvního měření. V prvním sloupci jsou názvy jednotlivých ECS knihoven. Ve druhém sloupci jsou naměřené časy. Ve třetím sloupci je možná chyba a ve čtvrtém standardní odchylka. Jednotlivé řádky jsou seřazené podle naměřených časů.

Z tabulky 5.4 jsme sestavili graf, který je možné vidět na obrázku 5.1. Tento graf je obarvený podle kategorií, které jsme si stanovili v sekci 5.2.3. Knihovny, které spadají do kategorie 1, jsou obarveny zeleně, následně knihovny, které patří do kategorie 2, jsou obarveny žlutě a knihovny které, jsou součástí kategorie 3, jsou obarveny červeně.

V sekci 5.2.3 jsme si stanovili hypotézu, podle které mají nejlépe vyjít knihovny z kategorie 1 (viz zelená barva). Poté se mají umístit knihovny z kategorie 2 (viz žlutá barva). A na závěr se mají umístit knihovny z kategorie 3 (viz červená barva). Při nahlédnutí do tabulky 5.4 nebo grafu z obrázku 5.1 je lehké pozorovat, že tomu tak opravdu je. Ovšem je nutné podotknout, že knihovny *Arch* a *RelEcs*

knihovna	Průměrný čas	Chyba	StdDev
HypEcs	1.133 s	0.0218 s	0.0194 s
LeoEcsLite	1.194 s	0.0188 s	0.0176 s
LeoECS	1.208 s	0.0233 s	0.0229 s
DefaultEcs	1.216 s	0.0243 s	0.0451 s
Svelto.ECS	1.384 s	0.0172 s	0.0161 s
Arch	1.394 s	0.0098 s	0.0092 s
RelEcs	1.401 s	0.0166 s	0.0156 s
MonoGameExtended.Entities	2.004 s	0.0340 s	0.0318 s
Entitas	4.732 s	0.0244 s	0.0228 s

Tabulka 5.4 Výsledky prvního měření pro jednotlivé ECS knihovny.



Obrázek 5.1 Grafy vyobrazující výsledky prvního měření zachycené v tabulce 5.4. Graf je obarvený podle kategorií ze sekce 5.2.3.

mají velmi podobný výsledek a pokud nahlédneme na chybu z tabulky 5.4 tak je možné pozorovat, že časy jsou téměř identické. Vyplývá z toho to, že pro tento počet entit požadavek na to, aby jednotlivé komponenty byly struktury, nemusí nutně vést k velkému výkonnostnímu zlepšení.

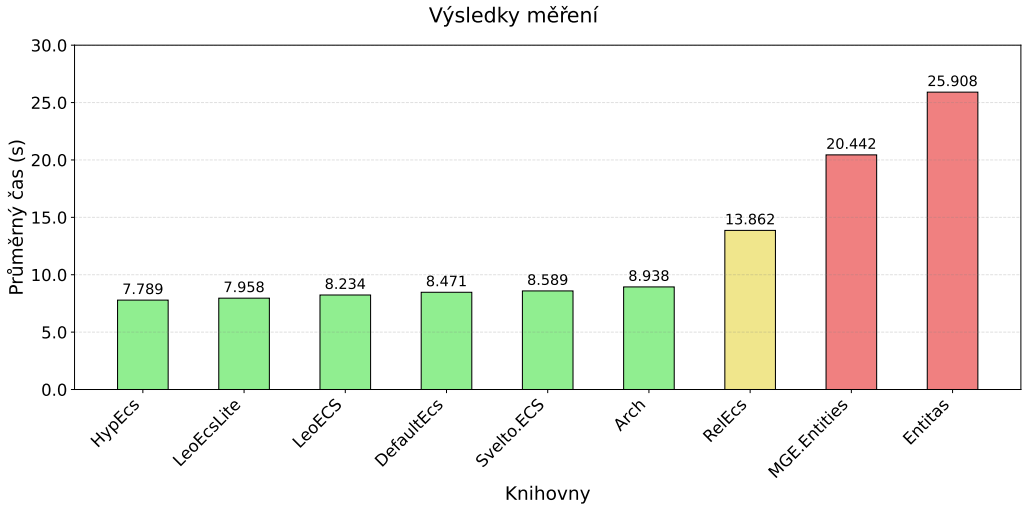
5.4.2 Druhé měření

Druhé měření bylo spuštěno s velikostí světa 16384x16384. Výsledky tohoto měření je možné vidět v tabulce 5.5 a grafu znázorněném na obrázku 5.2.

Z tabulky 5.5 a grafu 5.2 je možné vypozorovat významnější rozdíl mezi knihovnami první kategorie (zelená barva) a knihovnou druhé kategorie (žlutá barva). Předpokládáme, že tento rozdíl je způsobený větším počtem *cache misses*,

knihovna	Průměrný čas	Chyba	StdDev
HypEcs	7.789 s	0.1542 s	0.3044 s
DefaultEcs	7.958 s	0.1585 s	0.3129 s
LeoECS	8.234 s	0.1647 s	0.3011 s
LeoEcsLite	8.471 s	0.1670 s	0.3177 s
Svelto.ECS	8.589 s	0.1711 s	0.3571 s
Arch	8.938 s	0.1775 s	0.4253 s
RelEcs	13.862 s	0.2707 s	0.3424 s
MonoGameExtended.Entities	20.442 s	0.2375 s	0.2222 s
Entitas	25.908 s	0.5035 s	0.6183 s

Tabulka 5.5 Výsledky druhého měření pro jednotlivé ECS knihovny.



Obrázek 5.2 Grafy vyobrazující výsledky druhého měření zachycené v tabulce 5.5. Graf je obarvený podle kategorií ze sekce 5.2.3.

který se pro větší počet entit více projevuje a je způsoben tím, že knihovna *RelEcs* používá třídy namísto struktur pro reprezentaci komponent (viz sekce 5.2.3). Lze pozorovat, že hypotéza zde byla také splněna.

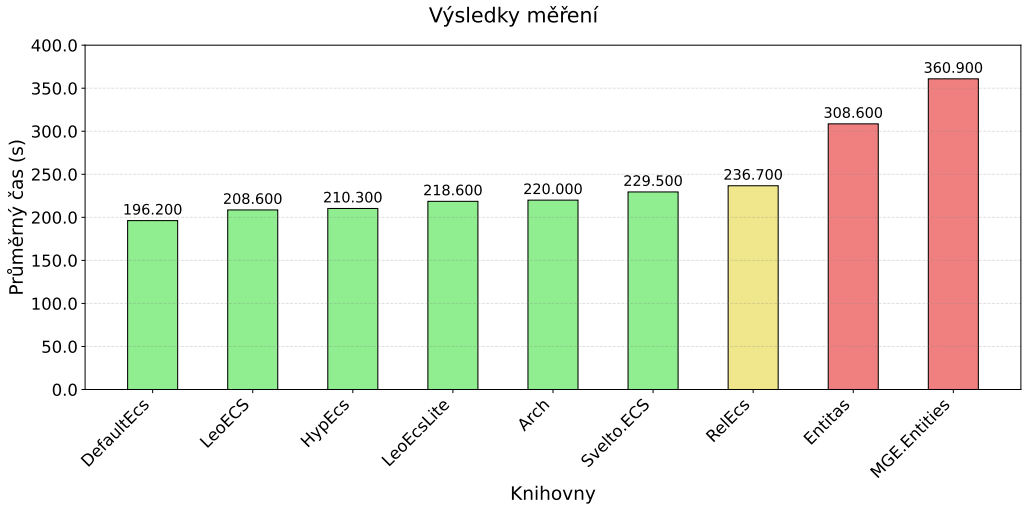
5.4.3 Třetí měření

Třetí měření bylo spuštěno s velikostí světa 32768x32768. Výsledky tohoto měření je možné vidět v tabulce 5.6 a grafu znázorněném na obrázku 5.3.

Z tabulky 5.6 a grafu 5.3 lze pozorovat, že výsledky jednotlivých kategorií jsou opět, podobně jako v případě prvního měření, blíže u sebe. Předpokládáme, že hra pro tuto velikost světa tráví spoustu času ve složitějších datových strukturách jako jsou *stromy chování* a *kd-stromy*, tím pádem knihovny první kategorie (zelená

knihovna	Průměrný čas	Chyba	StdDev
DefaultEcs	196.2 s	3.91 s	6.63 s
LeoECS	208.6 s	3.32 s	3.10 s
HypEcs	210.3 s	2.05 s	1.82 s
LeoEcsLite	218.6 s	3.08 s	2.88 s
Arch	220.0 s	4.27 s	3.99 s
Svelto.ECS	229.5 s	4.52 s	5.38 s
RelEcs	236.7 s	3.60 s	3.37 s
Entitas	308.6 s	6.11 s	6.01 s
MonoGameExtended.Entities	360.9 s	1.86 s	1.74 s

Tabulka 5.6 Výsledky třetího měření pro jednotlivé ECS knihovny.



Obrázek 5.3 Grafy vyobrazující výsledky třetího měření zachycené v tabulce 5.6. Graf je obarvený podle kategorií ze sekce 5.2.3.

barva) nezískávají tolik výkonu na dobrém využití *cache*. Je možné pozorovat, že hypotéza je stále splněna.

5.4.4 Velikosti světa

Nyní si rozebereme, co použité velikosti světů vlastně znamenají. Pro lepší pochopení kontextu autor práce provedl test na realtime strategické hře *Age of Mythology*. Byl spuštěn scénář s prázdnou mapou (bez překážek). Pro frakci byl zvolen bůh Zeus a velikost mapy byla zvolena normální. Vesničan (jednotka villager) z jednoho okraje mapy ke druhému došel za zhruba 60 sekund.

V naší hře se vesničané pohybují rychlostí 120 jednotek za sekundu. Velikost mapy pro první měření je 8192x8192. Vesničan by tedy tento svět zleva doprava

přešel za zhruba 68 sekund. Lze tedy tuto velikost světa do jisté míry srovnat s normální velikostí světa ve hře *Age of Mythology*. Velikost mapy pro druhé měření je 16384x16384 a vesničan by tento svět přešel zleva doprava za zhruba 137 sekund. Lze tedy do jisté míry tuto velikost světa považovat za nadstandardní. Velikost mapy pro třetí měření je 32768x32768 a vesničan by tento svět přešel zleva doprava za zhruba 273 sekund. Lze tedy tuto velikost světa do jisté míry považovat za velmi velkou.

5.4.5 Závěr měření

Ve výsledcích pro všechna tři měření je možné pozorovat, že rozdíl mezi ECS knihovnami kategorie 1 (zelená barva) a ECS knihovnami kategorie 3 (červená barva) není tak velký. Pokud by si tedy uživatel vybíral mezi ECS knihovnami a přišlo by mu, že pro něj je architektura nebo způsob využívání některé z knihoven kategorie 3 (červená barva) přínosnější nebo přehlednější oproti ostatním ECS knihovnám, staly by knihovny kategorie 3 (červená barva) také za zvážení.

5.4.6 Závěrečná pozorování k vybraným ECS knihovnám

V průběhu práce autor práce narazil na několik postřehů ohledně měřených ECS knihoven. V této sekci na tyto postřehy nahlédneme.

Jedním zajímavým pozorováním je, že knihovna *Svelto.ECS* vyžaduje, aby bylo během kompilace známo, z kterých komponent se jednotlivé entity budou skládat. Při nahlédnutí do výsledků měření je možné pozorovat, že toto omezení nevedlo k většímu výkonnostnímu zlepšení oproti ostatním knihovnám z této kategorie.

Knihovny *HypEcs* a *RelEcs* jsou od stejného autora. Autor těchto knihoven udělal nejprve *RelEcs* a poté *HypEcs*. *HypEcs* se mu podařilo udělat rychlejší zejména proto, že použil struktury místo tříd pro reprezentaci komponent.

Knihovny *LeoECS* a *LeoEcsLite* jsou od stejného autora. Obě tyto knihovny mají podobný výkon, ovšem při nahlédnutí do výsledků měření projektu *Ecs.Csharp.benchmark* [5] (konkrétně commitu *db67d1d* [31]), je možné pozorovat, že tyto dvě knihovny se umístily hůře. Hlavním důvodem je to, že autor projektu *Ecs.Csharp.benchmark* použil Nugget balíčky pro tyto knihovny, které byly vytvořeny jiným autorem. Autor projektu *Ecs.Csharp.benchmark* dokonce tvrdí, že tyto balíčky byly sestaveny v debug konfiguraci.

ECS knihovna *Entitas* je na platformě GitHub [39] nejpopulárnější (má nejvíce hvězdiček) ECS knihovnou pro C#. I přesto, že v našem měření vyšla nejhůře, její výkon je stále dost dobrý. K její popularitě také přispívá větší podpora pro herní engine *Unity* [1].

Kapitola 6

Závěr

V této kapitole zhodnotíme, zda jsme splnili cíle práce. Nejprve se zaměříme na splněné cíle a uvedeme si nedostatky našeho řešení. Na závěr nahlédneme na možná budoucí vylepšení.

6.1 Cíle

Nyní si připomeňme, jaké jsou cíle této práce:

1) Vytvořit hru pomocí ECS, která bude nezávislá na konkrétní ECS knihovně.

2) Pomocí vytvořené hry porovnat relativní výkon populárních ECS knihoven pro programovací jazyk C#.

6.1.1 Splnění cíle 1

Pro splnění cíle 1 byla vytvořena hra. Specifikaci této hry jsme stanovili v sekci 2.2. Analýzu hry jsme provedli v kapitole 3 a její implementaci jsme rozebrali v sekci 4.2.

V sekci 2.1.1 jsme si stanovili požadavky, které od naší hry vyžadujeme. Konkrétně jsme chtěli, aby hra splňovala následující požadavky:

p1) Nezávislost na konkrétní ECS knihovně.

p2) Hra by neměla využívat velmi častého přidávání a odebírání komponent.

p3) Velký počet entit.

p4) Adekvátní složitost hry.

Ke splnění požadavku *p1* jsme vytvořili abstrakční vrstvu, kterou hra používá namísto konkrétní ECS knihovny. Analýzu této vrstvy jsme provedli v sekci 3.2 a implementaci jsme si rozebrali v sekci 4.1.

Hlavní důvod požadavku *p2* je zabránit velké ztrátě na výkonu knihovnám založených na *arch typu*. V naší implementaci využíváme odebírání komponent v případě mazání entity, které provádíme, pokud entita zemřela nebo byla sklizena. Pokud nahlédneme do výsledků prezentovaných v sekci 5.4, je možné pozorovat, že mazání entity neprovádíme příliš často, jinak by se ECS knihovny založené na *arch typu* umístili hůře.

Požadavek *p3* jsme splnili. Naše hra obsahuje velký svět s velkým počtem entit reprezentujících suroviny ke sklizení. Krom samotných surovin se v herním světě také nacházejí entity vesnic. Součástí vesnic jsou poté také entity budov a entity vesničanů.

Požadavek *p4* jsme vyžadovali ze dvou důvodů. První bylo zamezit tomu, aby naše hra nebyla příliš jednoduchá. Chtěli jsme se vyhnout tomu, aby naše měření, na které hru používáme, nebylo jenom dalším jednoduchým testem (viz sekce 1.2). Druhým důvodem bylo naopak zamezit příliš složité hře, jelikož bychom odváděli pozornost od problému, který řešíme. Složitost naší hry považujeme za adekvátní.

6.1.2 Splnění cíle 2

Pro porovnání výkonu jsme si vytvořili měření, které měří výkon naší hry s různými ECS knihovnami. Analýzu tohoto měření jsme provedli v sekci 3.5 a jeho implementaci jsme si rozebrali v sekci 4.3.

V kapitole 5 jsme si nejprve stanovili hypotézu. Poté jsme nahlédli na knihovny, které budeme měřit, a na konci jsme si prezentovali výsledky našeho měření.

6.1.3 Nedostatky řešení

I přesto, že jsme splnili stanovené cíle, tak některé věci v našem řešení jsme mohli udělat lépe. Nyní si je rozebereme:

1. **Komplexnější generování terénu:** I přesto, že autor považuje složitost hry za adekvátní, tak použití shaderů pro vygenerování terénu bylo pravděpodobně zbytečně komplexní. Pro účely této hry by postačovala jednodušší technika, která by například pouze předgenerovala náhodný terén.

2. **Občasné chození vesničanů přes vodu:** Generování náhodného terénu bylo nastaveno tak, aby terén pokud možno neobsahoval žádný ostrov. I přesto, že pravděpodobnost na vytvoření ostrova je nízká, tak občas k jeho vytvoření dojde. V případě, že se na tomto ostrově objeví jelen a některý z vesničanů se vydá jej ulovit, tak pro jeho ulovení přejde přes vodu.

Tento nedostatek by se dal vyřešit dvěma způsoby. První způsob je úprava algoritmu pro generování terénu. Ovšem tento způsob je nežádoucí, jelikož by zvedl složitost tohoto algoritmu. Druhý způsob by bylo zavedení detekce ostrovů. Bylo by nutné prozkoumat, jak ostrovy detekovat, aniž bychom výrazně zhoršili načítání nebo výkon hry. Jelikož hru používáme primárně pro měření ECS knihoven a nenastává příliš často, není pro nás nějak zásadní.

3. **Zdlouhavý proces přidávání nové ECS knihovny:** Hra je nezávislá na konkrétní ECS knihovně a lze ji tedy spouštět s různými ECS knihovnami. Proto, aby bylo možné s danou ECS knihovnou spustit, je nutné implementovat odpovídající třídy. Ovšem implementace těchto tříd může trvat delší dobu. Pro vyřešení by bylo nutné prozkoumat jiné návrhy abstrakční vrstvy, které zachovávají stejné klíčové vlastnosti jako ta momentální. Naštěstí ECS knihovny nepřidáváme příliš často, proto tento problém nepovažujeme za zásadní.

6.2 Možná budoucí vylepšení

Nyní si představíme některá vylepšení, která by do našeho řešení bylo možné implementovat. Prvně nahlédneme na možná vylepšení, která by bylo možné přidat do naší hry. Na závěr prozkoumáme možná vylepšení našeho měření.

6.2.1 Možná vylepšení hry

Mezi možná vylepšení hry patří:

1. **Akční menu pro vytváření a mazání entit:** Hra by mohla obsahovat akční menu, skrze které by bylo možné vytvářet a mazat entity v herním světě. Skrze toto menu by hráč mohl například vytvořit novou vesnici a nebo odebrat suroviny kolem již existující vesnice. Přidání tohoto menu by bylo usnadněné, jelikož kód je navržen tak, aby do hry bylo lehké přidávat nové prvky uživatelského rozhraní.

2. **Čas a denní cyklus:** Hra by obsahovala denní cyklus, podle kterého by se střídaly den a noc. Přes noc by byl aktivní efekt tmy. Hráč by také mohl mít schopnost ovládat rychlost času ve hře skrze speciální menu. Tato rychlost by neměla vliv jenom na samotný denní cyklus, ale i na fungování celé hry.

3. **Interakce mezi vesnicemi:** Jednotlivé vesnice spolu momentálně žádným způsobem neinteragují. Do hry by bylo možné přidat interakce mezi vesnicemi, které by více oživily dění v herním světě. Mezi tyto interakce by

mohlo patřit například obchodování a válčení. Hra by také mohla obsahovat akční menu, skrze které by hráč mohl tyto interakce vynucovat nebo rušit.

4. **Různé náročnosti terénu:** Jednotlivé biomy by mohly mít definovanou náročnost, která by měla vliv na rychlost pohybu entit, které se v nich nacházejí. Například entity by v biomu vysoké hory mohly chodit s poloviční rychlostí.

5. **Více variací textur:** Hra by mohla pro některé entity mít více textur. Při vytváření dané entity by se poté vybrala náhodná z nich. Například vesničané by mohli mít texturu pro muže a ženu nebo strom by mohl mít texturu pro listnatý a jehličnatý.

6. **Hlavní menu:** Při zapnutí hry by se uživateli nejprve ukázala obrazovka hlavního menu. Ta by obsahovala uživatelské rozhraní, pomocí kterého by si hráč mohl zvolit některá nastavení, jako například velikost herního světa. Tvorba tohoto menu by byla usnadněna, jelikož kromě snadné rozšiřitelnosti o prvky uživatelského rozhraní je kód také navržen tak, aby bylo snadné přidávat další herní obrazovky.

6.2.2 Možná vylepšení měření

Mezi možná vylepšení související s měřením patří:

1. **Měření během hraní:** Hra by měřila výkon hry se zvolenou ECS knihovnou během hraní. Bylo by možné měřit výkon například posledních x iterací *game loop*. Pokud bychom do hry také přidali možnost měnit ECS knihovnu za běhu, mohli bychom za běhu pozorovat, jak se výkon mění s jednotlivými ECS knihovnami.

2. **Měření jednotlivých systémů:** Kromě měření výkonu celé hry s konkrétní ECS knihovnou bychom mohli také měřit výkon jednotlivých systémů. Mohli bychom poté pozorovat vliv ECS knihovny na rychlost různých systémů.

Seznam použité literatury

Dostupnost online zdrojů jsou zkontrolovány k 17.7.2024.

[1] Domovská stránka herního enginu Unity. URL: https://unity.com/.

[2] Domovská stránka herního frameworku MonoGame. URL: https://monogame.net/.

[3] Robert Nystrom. *Game programming patterns*. Genever Benning, 2014.

[4] GitHub reprozitář projektu CSharpECSComparison. URL: https://github.com/Chillu1/CSharpECSComparison.

[5] GitHub reprozitář projektu Ecs.CSharp.Benchmark. URL: https://github.com/Doraku/Ecs.CSharp.Benchmark.

[6] Přehled technologií používané hrami v herním obchodu Steam. URL: https://steamdb.info/tech/.

[7] Domovská stránka herního obchodu Steam. URL: https://store.steampowered.com/.

[8] Domovská stránka herního enginu Godot. URL: https://godotengine.org/.

[9] Domovská stránka herního frameworku FNA. URL: https://fna-xna.github.io/.

[10] David Notario. "Jit Optimizations: Inlining (II)". In: *Microsoft Learn* (lis. 2004). URL: https://learn.microsoft.com/en-us/archive/blogs/davidnotario/jit-optimizations-inlining-ii.

[11] Vance Morrison. "To Inline or not to Inline: That is the question". In: *Microsoft Learn* (srp. 2008). URL: https://learn.microsoft.com/en-us/archive/blogs/vancem/to-inline-or-not-to-inline-that-is-the-question.

[12] "Generics in the runtime (C# programming guide)". In: *Microsoft Learn* (dub. 2024). URL: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generics-in-the-run-time.

[13] *MethodImplOptions Enum - .NET API reference*. Microsoft. URL: https://learn.microsoft.com/en-us/dotnet/api/system.runtime.compilerservices.methodimploptions?view=net-8.0.

[14] GitHub reprozitář ECS knihovny Arch. URL: https://github.com/genaray/Arch.

[15] Výukové materiály AI - State Machines. Newcastle University. URL: https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/aitutorials/1state-basedai/AI%20-%20State%20Machines.pdf.

[16] Chris Simpson. "Behavior trees for AI: How they work". In: *Game Developer* (čvc. 2014). URL: https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work.

[17] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[18] Mingu Kwon. "Finite State Machine & Behavior Tree in Robotics". In: *MINGU KWON* (2024). URL: https://www.mingukwon.com/posts/finite-state-machine-&-behaviour-tree.

[19] Tuomo Hyttinen. "Terrain synthesis using noise". Dipl. pr. University of Tampere, květ. 2017. URL: https://trepo.tuni.fi/bitstream/handle/10024/101043/GRADU-1494236249.pdf.

[20] "Making maps with noise functions". In: *Red Blob Games* (čvc. 2015). URL: https://www.redblobgames.com/maps/terrain-from-noise/.

[21] Wouter van Toll et al. "A comparative study of navigation meshes". In: *Proceedings of the 9th International Conference on Motion in Games*. MIG '16. Burlingame, California: Association for Computing Machinery, 2016, s. 91–100. ISBN: 9781450345927. DOI: 10.1145/2994258.2994262. URL: https://doi.org/10.1145/2994258.2994262.

[22] GitHub reprozitář forku MonoGame frameworku, který byl použit v této práci. URL: https://github.com/cpt-max/MonoGame.

[23] Nilushan Costa. "Tips for Software Performance Testing". In: *Medium* (čvn. 2020). URL: https://medium.com/swlh/tips-for-software-performance-testing-9f1f12da08c5.

[24] Reprozitáře s topicem performance na platformě GitHub seřazené podle počtu hvězd. URL: https://github.com/topics/performance?l=c%23.

[25] GitHub reprozitář knihovny BenchmarkDotNet. URL: https://github.com/dotnet/BenchmarkDotNet.

[26] GitHub reprozitář ECS knihovny Svelto.ECS. URL: https://github.com/sebas77/Svelto.ECS.

[27] "Unmanaged types (C# reference)". In: *Microsoft Learn* (ún. 2024). URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/unmanaged-types.

[28] GitHub reprozitář knihvony KdTree. URL: https://github.com/codeandcats/KdTree.

[29] Alan Jay Smith. "Cache Memories". In: *ACM Comput. Surv.* 14.3 (zář. 1982), s. 473–530. ISSN: 0360-0300. DOI: 10.1145/356887.356892. URL: https://doi.org/10.1145/356887.356892.

[30] Michele Caini. "ECS back and forth". In: *skypjack on software* (ún. 2019). URL: https://skypjack.github.io/2019-02-14-ecs-baf-part-1/.

[31] Commit GitHub projektu Ecs.CSharp.Benchmark. V práci byli použity ECS knihovny, které byli také použity projektem Ecs.CSharp.Benchmark při tomto commitu. URL: https://github.com/Doraku/Ecs.CSharp.Benchmark/tree/db67d1da27186eda8198627c69f95041ed01e12e.

[32] GitHub reprozitář ECS knihovny DefaultEcs. URL: https://github.com/Doraku/DefaultEcs.

[33] GitHub reprozitář ECS knihovny Entitas. URL: https://github.com/sschmid/Entitas.

[34] GitHub reprozitář ECS knihovny HypEcs. URL: https://github.com/Byteron/HypEcs.

[35] GitHub reprozitář ECS knihovny LeoECS. URL: https://github.com/Leopotam/ecs.

[36] GitHub reprozitář ECS knihovny LeoEcsLite. URL: https://github.com/Leopotam/ecslite.

[37] GitHub reprozitář projektu MonoGame.Extended. URL: https://github.com/craftworkgames/MonoGame.Extended.

[38] GitHub reprozitář ECS knihovny RelEcs. URL: https://github.com/Byteron/RelEcs.

[39] Domovská stránka platformy GitHub. URL: https://github.com/.

Přílohy

Přehled elektronických příloh

1. `src` – Adresář obsahující zdrojové soubory a sestavený projekt měření. Součástí zdrojových souborů jsou dva projekty (WorldSimulator.csproj a WorldSimulator.Benchmarks.csproj).
2. `build` – Adresář obsahující sestavený projekt hry World Simulator.
3. `README.md` – Textový soubor obsahující opis seznamu příloh, dodatečné informace k seznamu příloh a kontakt na autora práce.