VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

PROHLÍŽEČOVÁ HRA S UMĚLOU INTELIGENCÍ

BROWSER GAME WITH ARTIFICIAL INTELLIGENCE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. MICHAL MORAVEC**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. VLADIMÍR BARTÍK, Ph.D.**
SUPERVISOR

BRNO 2018

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Moravec Michal, Bc.**
Program: Informační technologie Obor: Počítačová grafika a multimédia
Název: **Prohlížečová hra s umělou inteligencí**
Browser Game with Artificial Intelligence
Kategorie: Umělá inteligence

Zadání:

1. Seznamte se s tvorbou single-page webových aplikací, problematikou NoSQL databází a umělou inteligencí využívanou v počítačových hrách.
2. Analyzujte požadavky na prohlížečovou hru založenou na těžbě surovin, obchodu s nimi a různých soubojích mezi hráči, kterou bude možné hrát i proti virtuálnímu soupeři. Navrhněte NoSQL databázi pro ukládání informací o hře.
3. Navrhněte hru dle požadavků. Dále navrhněte koncepci využití umělé inteligence založené na genetických algoritmech pro účely hry proti virtuálnímu soupeři. Návrh konzultujte s vedoucím.
4. Navrženou prohlížečovou hru implementujte.
5. Otestujte funkčnost hry a proveďte experimenty vyhodnocující více řešení umělé inteligence.
6. Zhodnoťte dosažené výsledky a diskutujte další možné pokračování tohoto projektu.

Literatura:

- Sivanandam, S.: Introduction to genetic algorithms. Springer Verlag, 2006.
- Tiwari, S.: Professional NoSQL, 1st Edition. Wrox, 2011.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Bartík Vladimír, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 28. října 2018

Abstrakt

Práce se zabývá návrhem a implementací webové hry, kterou může hrát více hráčů přes síť internet. Stěžejním prvkem hry je budování ekonomiky, hráči však spolu mohou i spolupracovat (obchodování) a hrát proti sobě (souboje). Pro perzistentní úložiště postupu je použita NoSQL databáze, kde v práci je popsán její návrh a implementace. Mimo reálných hráčů se ve hře vyskytují agenti/boti, kteří hru hrají automatizovaně pomocí stavových automatů vygenerovaných genetickými algoritmy. V práci je popsán princip návrhu a fungování jak genetických algoritmů, tak samotných stavových automatů.

Abstract

Thesis describes design and implementation of a web browser game, which can be played by multiple players via the internet. The main goal is to manage the economy, although players can cooperate (trading) or play against each other (battles). NoSQL database is used for persistent storage of progress, which is also described in the thesis. Apart from human players there are also agents/bots, which play the game autonomously via state machines generated by genetic algorithms. Paper describes design and functionality of either the genetic algorithms, but also the state machines.

Klíčová slova

Hra, web, webová hra, prohlížečová hra, autentifikace, autorizace, javascript, react, Apollo, NoSQL databáze, MongoDB, stavový automat, genetický algoritmus

Keywords

Game, web, web game, browser game, authentification, authorization, javascript, react, Apollo, NoSQL database, MongoDB, state machine, genetic algorithm

Citace

MORAVEC, Michal. *Prohlížečová hra s umělou inteligencí*. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Vladimír Bartík, Ph.D.

Prohlížečová hra s umělou inteligencí

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Vladimíra Bartíka, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Michal Moravec
13. května 2019

Poděkování

Tímto děkuji svému vedoucímu za to, že mi umožnil pracovat na práci s vlastním zadáním a také za cenné rady, které mi poskytl v průběhu vypracování a implementace práce. Také mu děkuji za nápady ohledně úprav zadání, díky kterým dává celý projekt smysl pro vypracování v rámci diplomové práce.

Obsah

Kapitola 1

Úvod

Když se mluví o hrách a umělé inteligenci, většinou se jedná o agenty, kteří v rámci hry interagují buď s ostatními agenty nebo s lidským hráčem tak, jak byli přesně naprogramováni. Když se však mluví o umělé inteligenci obecně, jsou dnes velmi oblíbené metody strojového učení, které dokážou řešit různé abstraktní problémy. Cílem této diplomové práce je nejprve vyvinout prohlížečovou hru a poté umělou inteligenci, která bude hru hrát. Umělou inteligencí myslím agenty (v konkrétním případě stavové automaty), které hru hrají pomocí přesně určených pravidel. Na tato pravidla však bude muset přijít genetický algoritmus, který bude nasměrován pomocí jednoduché ohodnocovací funkce. Přínosem pro čtenáře bude nejen informace o tom, jak tento systém navrhnout, ale jak ho znovupoužít na jiné druhy problémů. Dále zde budou uvedeny důležité informace pro čtenáře, kteří se zajímají pouze o vývoj prohlížečových her jako single page aplikací.

V první řadě je v tomto textu popsán návrh a implementace webové prohlížečové hry. Nejprve je proveden průzkum trhu a srovnání se stávajícími hrami, které se této hře podobají. Dále je uvedena důležitá teorie, která je třeba k pochopení použitých technologií: NoSQL databáze, single page aplikace, webové sockety a technologie GraphQL. U single page aplikací bude také uvedena historie a srovnání s ostatními druhy aplikací, díky kterému bude popsáno rozhodnutí, proč je tato hra implementována jako single page aplikace. Jelikož pro komunikaci v reálném čase jsou použity webové sockety, budou zmíněny jiné přístupy, jak tento problém řešit a zdůvodněny výhody právě webových socketů. Na závěr studijní části bude popsána technologie GraphQL a její výhody. Mimo jiné bude popsána architektura hry, která bude také zakreslena do grafu. Před návrhem databáze bude nastíněn základní princip hry, který je pro pochopení návrhu důležitý. Stěžejním prvkem hry je budování ekonomiky, kterému je kladen velký důraz. V textu bude popsán princip, jakým způsobem je toto obecně vyřešeno jak v základním principu hry, tak v definici statických herních prvků. Jako poslední bude v návrhu hry popsán návrh jejího uživatelského rozhraní. Po návrhu hry je popsána její implementace, kde budou popsány konkrétní databázové modely, ale také návrh frontendu a backendu z hlediska napojení na technologii GraphQL, pomocí které je implementována téměř veškerá logika hry. Kromě GraphQL bude uvedena implementace stránek a komponent hry, kde budou uvedeny konkrétní návrhové vzory pro zjednodušení implementace a znovupoužitelnost kódu v knihovně React.

V druhé řadě je popsán systém využívající genetické algoritmy pro účely generování stavových automatů, které automatizovaně hrají výše zmíněnou webovou hru tak, aby reální lidé, kteří hru hrají, měli pocit, že hrají vždy proti reálných lidem. Nejprve je zmíněna teorie nutná k pochopení principu fugování genetických algoritmů, poté samotný návrh systému a jeho implementace. Systém bude implementován čistě pro účely této hry, avšak v textu

bude popsáno, jak tento algoritmus využít i pro jiné druhy problémů. Před návrhem bude popsána motivace k využití algoritmu. Poté bude popsán návrh a implementace jednotlivých stavebních prvků.

Předposlední kapitola bude obsahovat vyhodnocení výsledků genetického algoritmu, kde bude nejprve popsána implementace testovacího nástroje spolu s napojením na genetický algoritmus, ale také metodika vyhodnocení. Testování bude rozděleno na dvě části, kde první část bude sloužit pro vysvětlení výstupu nástroje a druhá část pro nalezení nejvýhodnější konfigurace parametrů pro spuštění nástroje.

Práce bude shrnuta v závěru, kde bude mimo jiné uvedeno, které funkce se povedly a nepovedly implementovat, ale také plán do budoucna. Po dokončení této práce totiž plánuji pokračovat ve vývoji hry.

Kapitola 2

Historie a stávající webové hry

Webové prohlížečové hry jsou specifickým typem her, které jsou spustitelné a hratelné pouze pomocí webového prohlížeče [8]. Hlavními výhodami je možnost hraní těchto her z kteréhokoliv zařízení, které podporuje prohlížení internetových stránek pomocí webového prohlížeče, ale také neexistence nutnosti cokoliv stahovat nebo instalovat. Ačkoliv hra běží na webovém serveru, neznamená to, že musí nutně umožňovat hráčům interakci mezi sebou. V této pasáži bude popsána jedna taková hra, ale také příklady jiných her, které interakci mezi hráči umožňují. Tyto hry zde však budou primárně uvedeny proto, že obsahují herní principy, kterými jsem se inspiroval při návrhu webové hry, o které bude řeč v tomto textu.

Pro pochopení souvislostí zde bude také stručně popsána historie webových her a technologií, na kterých jsou/byly založeny. Tato část je důležitá pro následné pochopení rozhodování, které vedlo k použití technologií webové hry, která je popsána v tomto textu.

2.1 Historie

První prohlížečové hry využívaly pouze technologie DHTML pro zobrazení jednoduchého uživatelského rozhraní. Jednou z prvních prohlížečových her byla hra Earth: 2025 naprogramovaná Mehulem Patelem v roce 1996. Jednalo se o strategickou hru zasazenou do postapokalyptické budoucnosti. Přestože byla ukončena roku 2009, komunita hráčů založila hru Earth Empires, kam se všichni hráči přesunuli [1].

Významným milníkem pro prohlížečové hry byl vznik technologie Flash, kterou nejprve vyvíjela firma FutureWave Software. Tato firma byla však později koupena firmou Macromedia, která technologii přejmenovala na Macromedia Flash. Tato technologie společně s programovacím jazykem ActionScript pomohla vzniku mnoha prohlížečových her. Nevýhodou je však nutnost instalace dodatečného softwaru do webového prohlížeče, aby bylo možné aplikace vytvořené ve Flashi spouštět. V dnešní době je vlastníkem Flashe firma Adobe, která plánuje do roku 2020 přerušit jeho vývoj.

Dnešní prohlížečové hry jsou převážně vyvíjeny s pomocí HTML5 a Javascriptu. 2D hry mohou využít HTML5 komponenty Canvas, do které je možno vykreslovat grafické objekty. Pro vývojáře 3D her je zde technologie WebGL (odvozeno od OpenGL).

[1] https://pcdreams.com.sg/history-of-web-browser-games/

2.2 Inkrementální hry

Jedná se o specifický žánr her, kde základním konceptem je snaha o nejvyšší zisk herní měny. Hráč začíná od nuly a cílem je získávat herní měnu. Zpravidla tyto hry nemají žádný konec, takže hráč získává herní měnu donekonečna. V některých hrách je zakomponován restart, který hráči smaže veškerý postup, avšak mu pomůže v dalším průchodu hrou tak, že herní měnu získává rychleji, než v předešlém průchodu. Společným prvkem je základní herní smyčka:

1. Získat herní měnu
2. Za herní měnu nakoupit prostředky, díky kterým se generuje více herní měny
3. Bod 1

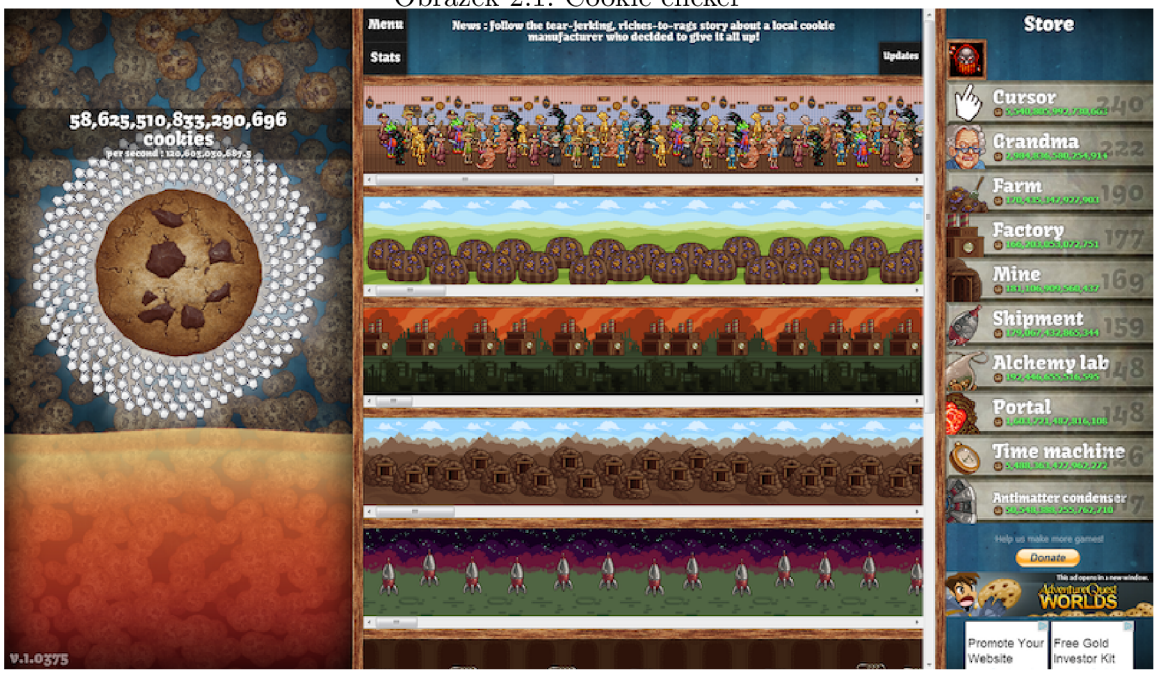
Jelikož hráč nemá jak herní měnu ztratit, může hru nechat zapnutou a ta se tak může hrát sama. Některé inkrementální hry umožňují získávání herní měny, i když je samotná hra vypnutá. Při zapnutí hry se totiž vypočítá množství herní měny, které by hráč získal, kdyby hru nechal spuštěnou. Díky této vlastnosti se těmto hrám říká také idle (líný/nečinný).

Ačkoliv se může zdát, že takto jednoduchý koncept není zábavný, opak je pravdou. Příkladem je hra Cookie clicker, kterou hrají miliony lidí. Tento koncept her vznikl jako parodie na stávající hry, které fungují v principu stejně, avšak tento primitivní základní koncept je hráči zastřen ať už formou např. budování armády, města nebo farmy.

Cookie clicker

Jedná se o nejznámější inkrementální webovou hru. Cílem hry je nasbírat/vyrobit co nejvíce sušenek. Na začátku nemá hráč žádnou a musí je sbírat klikáním. Jakmile nasbírá určitý počet, může si za ně nakoupit automatizovanou výrobu. Automatizovaná výroba hráči vyrábí sušenky, takže už není potřeba je vyrábět ručně klikáním. Ve hře se tedy jen čeká na dosažení dostatečného množství sušenek, za které je možné nakoupit vylepšení automatizované výroby, která potom generuje větší množství sušenek.

Obrázek 2.1: Cookie clicker [2]



Clicker heroes

Další velice známá a hraná inkrementální hra. Cílem je porazit co nejvíce nepřátel. Zpočátku, stejně jako ve hře Cookie clicker, musí hráč nepřátele zabíjet klikáním. Zabitím nepřítele je odměnou určitý počet zlaťáků, za které si může hráč najímat hrdiny, kteří bojují za něj. Čím dále hráč postupuje, tím více zlaťáků dostává a může si tedy najímat silnější hrdiny, kteří mu pomáhají v postupu hrou.

[2] Zdroj: https://medium.com/@kaleb.nekumanesh/cookie-clicker-analysis-bf3787aa96d7

Obrázek 2.2: Clicker heroes [3]

Hra je rozdělena do zón, kde v každé zóně se nachází 10 nepřátel. Každý desátý nepřítel, který se nachází na konci zóny, je tzv. boss, neboli silnější nepřítel, který více vydrží a pro jeho zabití je nutné to stihnout v určitém časovém limitu. Po dosažení určité zóny je také možné hru resetovat (vymazat postup), čímž se získá speciální druh herní měny, za kterou je možné kupovat vylepšení, které přetrvávají mezi jednotlivými resety a zároveň urychlují postup. Zároveň umožňují postup do dalších zón, kam by se hráč bez těchto resetů jinak nedostal.

2.3 Strategické hry

Dalším žánrem jsou hry strategické, které se dají rozlišit podle dvojího dělení:

1. Budovatelské/ekonomické
2. Bojové

V budovatelských strategiích neexistují souboje. Hráč se tedy stará o ekonomiku svého města/základny/vesnice. Jestli u takové hry existuje interakce s ostatními hráči, je provedena pomocí obchodu. Cílem je většinou dosáhnout ekonomické převahy nad soupeři. Některé hry však umožňují stavění města bez žádného soupeře.

Bojové strategie vyžadují rekrutování armády, pomocí které mezi sebou hráči soupeří. Některé strategické hry vyžadují jak stavění města/základny, tak rekrutování armády. U jiných strategií je hráči umožněno velet pouze armádě a ekonomické prvky jsou úplně vypuštěny.

1. Real-time (hra se odehrává v reálném čase)
2. Tahové

[3] Zdroj: https://store.steampowered.com/app/363970/Clicker_Heroes/

Strategické hry v reálném čase vyžadují udělování příkazů v reálném čase. V těchto hrách je nejen potřeba ovládat strategické myšlení, ale zároveň být rychlejší, než soupeř.

Opakem jsou strategie tahové, ve kterých má každý hráč prostor na udělené příkazů v rámci tahu. Jakmile všichni hráči odehrají daný tah, pokračuje se dalším. Výhoda těchto her je v dostatku času, který má každý hráč na rozmyšlenou.

Travian

Jedna z nejznámějších prohlížečových strategií. Naprogramovaná byla v roce 2004 Gerhardem Müllerem. Hra je zasazena do období antiky, kde se v ní nacházejí 3 národy: římané, galové a germáni. Jedná se o bojovou strategii v reálném čase, která zároveň vyžaduje budování ekonomiky města. Hra poskytuje interakci s ostatními hráči buď pomocí obchodu, nebo pomocí soubojů. Co se týče ekonomiky, ve hře jsou 4 základní suroviny: dřevo, hlína, železo a obilí. Tyto suroviny se získávají stavěním a vylepšováním příslušných budov okolo vesnice. Ze získaných surovin je možno stavět budovy ve vesnici, které umožňují stavění armády, zvýšení produkce a nebo snížení času stavění budov.

Obrázek 2.3: Travian [4]

Hra obsahuje mapu, kam jsou hráči umísťování podle času registrace směrem od středu k okrajům. Při registraci si hráč může zvolit sektor (jihovýchod, severozápad, aj.) a poté je umístěn na okraj mapy. Tímto je zařízeno, že hráči, kteří hru hrají dlouho, nepřijdou do styku s novými hráči. Kdyby chtěl takovýto silnější hráč zaútočit na někoho slabšího na okraji mapy, jeho armáda tam bude muset dlouhou dobu cestovat, což se nevyplatí.

[4] Zdroj: https://www.kingdoms.com/com/

Kapitola 3

Studium technologií k webové hře

Webová prohlížečová hra popsaná v tomto textu využívá ve velké části programovacího jazyka Javascript, ve kterém je nejen naprogramována, ale ve kterém jsou také naprogramovány některé knihovny a závislosti hrou využívané. Všechny tyto komponenty, ale také např. NoSQL databáze, budou rozebrány v této části textu.

3.1 NoSQL databáze

NoSQL databáze by se daly přeložit jako databáze, které používají jiný než relační model (oproti databázím relačním). Toto však nemusí být pravda, protože i NoSQL databáze mohou používat relační model, avšak nesmí ho obsahovat pouze exkluzivně. Přesnější definicí by mohl být anglický výraz "Not Only SQL", tedy databáze obsahující nejen SQL. NoSQL databáze tedy relační model používat nemusí, ale mohou [3].

Relační databáze jsou konzistentní (každé čtení vrátí buď výsledek posledního zápisu nebo chybu) a zároveň zajišťují vysokou dostupnost dat (na každý dotaz je vrácena odpověď, avšak bez garance, že obsahuje poslední zapsanou hodnotu). Kvůli CAP teorému (také Brewerův teorém, podle Erica Brewera) však mají problém s tolerancí oddělení (systém funguje dále i přes ztráty paketů v síti), protože teorém tvrdí, že pro distribuovaný datový sklad není možné zaručit více jak 2 z těchto 3 podmínek. Po rozdělení dat mezi více systémů, které jsou propojeny do počítačové sítě, může v relační databázi dojít k mnoha problémům. Pro relační systém řízení báze dat (RSŘBD) je toto velice narušující. Vzhledem k úrovni rozdělení dat totiž lze v mezních případech přijít i o konzistenci dat v rámci modelu ACID. V tomto případě by bylo třeba na každém databázovém serveru udržovat vlastní databázové schéma. S tímto také souvisí problém se škálovatelností.

Na druhé straně jsou zde NoSQL databáze, které umí dobře dostupnost a toleranci oddělení. Dají se tedy lehce škálovat dokonce i pomocí levného hardwaru. Cenou je však podle CAP teorému konzistence. V NoSQL databázích jsou kopie dat uloženy na různých serverech, což znamená, že v momentu, kdy se změní nějaká hodnota, databáze vrátí aplikaci informaci o tom, že se daná hodnota změnila. Zde mohou nastat 2 případy: eventuální a transakční konzistence. U eventuální konzistence může nastat případ, že aplikace může po získání informace o tom, že byla data změněna, požádat databázi o čtení dané hodnoty, kdy tato hodnota ještě nebyla zapsána na všech databázových serverech. V tomto případě tedy databáze vrací ještě starou verzi hodnoty. U transakční konzistence vrací databáze aplikaci informaci o změně hodnoty až po zapsání změn do všech distribuovaných částí. Když tedy aplikace zažádá o data, nemůže se stát, že by dostala stará data [7].

Kromě škálovatelnosti se NoSQL databáze dají dobře použít pro ukládání jak strukturovaných a polostrukturovaných, tak i nestrukturovaných dat. Strukturovaná data mají pevně definovaný formát a velikost jednotlivých položek. Data polostrukturovaná mohou být např. ve formátu JSON, XML nebo podobné, kdy jsou sice strojově čitelná a zpracovatelná, avšak mohou obsahovat data, která nemají pevně danou délku nebo datový typ. Nestrukturovaná data již nejsou ani strojově jednoduše zpracovatelná (mohou však být zpracovatelná pomocí např. strojového učení). Příkladem takových dat jsou např. textové dokumenty nebo multimédia. Většina NoSQL databází pracuje bez databázového schématu. Důvodem je, že se do databáze dají uložit data, která mezi sebou mají určitý vztah, avšak tyto vztahy nejsou vynuceny databázovým schéma.

Relační databáze používají normalizaci dat, která se používá kvůli:

- Zabránění v duplikaci dat.
- Vynucení reálných vztahů, takže data jsou organizována do logických celků.
- Změny datové struktury se dají provést na jednom místě, což usnadňuje a urychluje aktualizace dat.
- Zachování integrity dat.

Normalizace dat je v relačních databázích důležitá, protože místo duplikace dat v jednotlivých tabulkách se používá výrazu spojení (join). Data z jednotlivých tabulek se dají spojit pomocí nějakého vztahu. Jelikož ale NoSQL databáze umožňují levné škálování, je možné si dovolit data duplikovat. Normalizace také vytváří vysoce strukturovaná data, která se však nemusí hodit pro všechny případy. Výhodou je rychlý zápis změn, protože lze měnit pouze určité malé části dat uložené v různých tabulkách. Při požadavku čtení, kde je třeba spojovat data z více tabulek však může nastat výrazné zpomalení. Čím více tabulek je třeba spojit pro získání výsledku, tím je dokonce toto čtení pomalejší. V případě správného návrhu NoSQL databází může být taková operace pouze jednoduché získání dokumentu. V systémech, u kterých se klade důraz na rychlost, je nutné, aby databázoví administrátoři trávili hodně času optimalizací databázových dotazů. Většina stráveného času je právě kvůli normalizaci. Normalizace ale není špatná věc, dokonce u vysoce strukturovaných dat je u relačních databází nezbytná. Každopádně v některých případech není potřeba a denormalizace dat může přispět k zrychlení získávání výsledků.

Proč tedy nepoužít relační databáze a denormalizovat data v nich pro urychlení? Dalo by se určitě zavést dvě verze databáze, kde v jedné verzi budou data normalizovaná pro rychlý zápis změn a v druhé verzi data denormalizovaná pro rychlé čtení. Proč ale v druhém případě spíše nepoužít rovnou NoSQL databázi, když je pro tento účel, narozdíl od relačních DB, specializovaná? Nejlepším kompromisem je tedy použít relační databázi pro strukturovaná data a NoSQL databázi pro data polostrukturovaná a nestrukturovaná.

Existuje několik druhů NoSQL databází. Dále budou popsány konkrétně tyto:

- Úložiště s položkami klíč-hodnota
- Databáze obsahující dokumenty
- Databáze se sloupci, které shlukují data
- Grafové databáze

Úložiště s položkami klíč-hodnota

Tento druh databáze neobsahuje žádné schéma a je optimalizován pro výkon s velkou dostupností. Je zaručena možnost rozdělení, replikace a automatického obnovení dat. Z názvu vyplývá, že se jedná o jednoduchou hashovací tabulku obsahující dvojice klíč-hodnota, kde jedinou možností, jak získat z této struktury data, je dotázat se pomocí klíče. Reprezentace hodnoty může být různá: MemcachedDB používá textový řetězec, Redis a Riak KV podporují textové řetězce, JSON, ale i binární data (BLOB).

Většinou používají také jednoduché rozhraní pro získávání, změnu a mazání hodnot (buď pomocí jednoho klíče pro získání, změnu a smazání jedné hodnoty nebo pole klíčů pro získání více hodnot najednou).

Tabulka 3.1: Ukázka úložiště klíč-hodnota

Klíč	Hodnota
user1	Michal
user2	Petr

Tabulka 3.2 je jednoduchým příkladem dvojic uložených do úložiště. Tyto úložiště jsou schopné zvládat konstantní proud operací, ať už čtení nebo zápis s velmi nízkou latencí. Mohou být také nastaveny, aby ukládaly data na disk. Hodnoty jsou uloženy binárně jako blob, takže odpadá potřeba indexovat data pro snížení doby pro čtení hodnot.

Co se týče škálovatelnosti a výkonu, jsou tyto úložiště asi nejlepší. Problémem je ale samozřejmě neexistence většiny komplexních funkcí, které jsou v ostatních úložištích/databázích dostupné. Pro svůj specifický účel ale svoji funkci splňují výborně.

Databáze obsahující dokumenty

Tyto databáze jsou nejvíce oblíbené v rámci NoSQL databází. Hodnoty jsou v databázi uloženy jako dokumenty, které mohou být v různých formátech jako XML, JSON, atd. Jedním z hlavních odlišovacích znaků je funkce vkládání metadat do uložených dokumentů. Toto umožňuje aplikaci získat dokumenty nejen na základě databázového dotazu, ale také na základě obsahu uložených hodnot.

Tabulka 3.2: Ukázka databáze s dokumenty

Klíč	Hodnota

1	`{`
	`  "user": "Michal",`
	`  "score": 0`
	`}`

2	`{`
	`  "user": "Petr",`
	`  "score": 10`
	`}`

Rozdělení, replikace a automatická obnova dat jsou zde dostupné bez další dodatečné implementace. Kvůli existenci replikace mohou být data uložena jednoduše na různých databázových serverech. Zde pak pro čtení dat dává smysl použít techniku map-reduce, která vrátí data z více různých serverů, spojí je dohromady a pak vrátí aplikaci. Obecně funguje funkce map tak, že vstupem a výstupem je pole prvků, kde pro každý prvek se specifikuje funkce, která tento prvek jakýmkoliv způsobem upraví (nebo dokonce smaže) a vloží do výstupního pole. Funkce reduce má na vstupu pole, ale na výstupu je pouze jeden prvek, který je výstupem aplikace specifikované funkce na všechny prvky pole. Funkce map se v databázi s dokumenty používá pro filtraci a řazení dat, funkce reduce pak pro provádění agregačních funkcí, jako je např. součet všech prvků.

V reálné databázi je nejprve aplikována operace map pro každý pracovní uzel (databázový server) na jejich lokální data a výsledek je uložen do dočasného úložiště. Jeden z uzlů, který je specifikovaný jako vedoucí, neboli master, zajistí, aby se zpracovala pouze jedna kopie redundantních dat. Na závěr jsou vytvořena data na základě výstupních klíčů. Další operací je shuffle, neboli zamíchání. Pracovní uzly přerozdělí data na základě výstupních klíčů vygenerovaných funkcí map a zamíchání je provedeno tak, že všechna data, která patří k určitému klíči, musí být umístěna na stejném pracovním uzlu. Poslední částí je funkce reduce, která zpracuje výsledná data paralelně po výstupním klíči.

Databáze se sloupci, které shlukují data

Stejně jako u relačních databází jsou data u těchto databází uložena v řádcích a sloupcích tabulky. Tyto databáze však obsahují sloupce, ve kterých jsou data logicky shluknuta. Toto logické shluknutí se nazývá column family.

Tabulka 3.3: Ukázka databáze se sloupci, které shlukují data

ID	Uživatel
0	Jméno: Michal Skóre: 0
1	Jméno: Petr Skóre: 10

Ke shluku dat, která jsou uložena v jednom ze sloupců, se dá přistoupit pomocí klíče. Výhodou je škálovatelnost, protože shluky dat mohou být uloženy na různých databázových serverech. Tyto databáze byly inspirovány databází BigTable od Googlu, která byla navržena speciálně tak, aby mohla běžet narozdíl od relačních databází v paralelním prostředí. Výhodou shluků dat je možnost získávat data, která jsou na nich postavená. Zatímco v relačních databázích jsou potřeba operace spojení (join), zde stačí díky chytrému uspořádání dat pouze klíč ke správnému sloupci.

Grafové databáze

NoSQL databáze jsou často kritizovány kvůli neexistenci operace spojení (join). V NoSQL databázích totiž nechybí operace spojení jako taková, protože jsou data ukládána v rámci jedné úrovně (nejsou zanořená), ale reprezentace vztahů. V ostatních NoSQL negrafových databázích je sice možné vztahy definovat, ale u grafových databází je tato definice nejvýhodnější. Místo sloupců a řádků totiž tyto databáze používají grafickou reprezentaci dat.

Všechny uzly grafu odpovídají reálným objektům. Hrany pak definují vztahy mezi těmito uzly/objekty. Každý uzel ví o svých sousedních uzlech. Při zvýšení počtu dat v databázi zůstává cena kroku z jednoho uzlu na jiný stejná.

Narozdíl od úložiště klíč-hodnota, které je optimalizováno pro získávání hodnot, jsou grafové databáze optimalizovány pro procházení dat mezi uzly. Obecně by bylo nalezení libovolných dat v grafové databázi pomalejší než v úložišti klíč-hodnota.

3.2 Single page aplikace

Single page aplikace je webová aplikace, která používá pouze jednu HTML stránku jako obálku pro všechny ostatní stránky aplikace. Uživatelská interakce je pak implementována pomocí klientského Javascriptu, HTML a CSS. Většina implementace je provedena na frontendu, narozdíl od klasického návrhu, kdy většina implementace je vždy na webovém serveru/backendu a při uživatelské interakci dojde k opětovnému načtení stránky. Svým chováním a implementací připomínají nativní aplikace, avšak s tím rozdílem, že běží v procesu webového prohlížeče, narozdíl od nativních aplikací, které běží v samostatném procesu v rámci operačního systému. Proč jsou dnes single page aplikace tak oblíbené při implementaci webových aplikací? K tomu je nejprve třeba znát trochu historie změn na webu v posledních letech [4].

Historie vývoje webu

Když v roce 1990 Tim Berners-Lee implementoval HTML a protokol HTTP, vytvořil tím World Wide Web (WWW), neboli celosvětovou síť. V této době byly webové stránky pouze složeny z HTML souborů, které interpretoval webový prohlížeč. Obsah na stránce byl statický a obsahoval převážně text a obrázky. Jednotlivé stránky byly mezi sebou propojeny odkazy, které po jejich prokliku uživatele přesměrovaly na jinou webovou stránku. Později byly vyvinuty serverové skriptovací jazyky pro tvorbu dynamických stránek, jako PHP nebo Active Server Pages (ASP). Stránka byla tedy dynamicky sestavena na webovém serveru a pak odeslána klientovi do prohlížeče, což umožnilo implementaci mnohem složitějších a vyspělejších webových aplikací, které mohly také reagovat na uživatelské vstupy.

Kolem roku 1996 byl představen Javascript jako skriptovací jazyk pro klientskou stranu webových aplikací. Díky Javascriptu mohli vývojáři implementovat novou logiku aplikace na straně klienta, což umožnilo nové možnosti v implementaci dynamických stránek. Většina vývojářů však v té době vyvinula weby zaměřené na logiku na straně serveru a neviděla v Javascriptu žádný potenciál. Ten byl tehdy využíván hlavně k dynamickým změnám uživatelského rozhraní. Je třeba také zmínit, že v této době byla uživatelská přívětivost webových aplikací nízká kvůli malé rychlosti připojení k internetu a nutnosti po každé interakci se stránkou celého znovunačtení stránky ze serveru.

Další změna přišla okolo roku 2000 kdy byly představeny technologie Flash a Java applety. Na webové stránce bylo možné mít vloženou aplikaci implementovanou ve Flashi nebo Javě, která poskytovala prostředí pro uživatelskou interakci. Tyto vložené objekty se samozřejmě vyskytují na webových stránkách i dnes, každopádně jsou některými prohlížeči blokovány kvůli bezpečnosti. Další nevýhodou je nutná instalace aplikací třetích stran (pro spuštění aplikací ve Flashi je nutná jeho instalace, stejně tak Java a Java applety). Okolo roku 2007 přišel Microsoft s technologií Silverlight, což byl další plugin do prohlížeče. Tato technologie se však neuchytila, protože měla stejné nevýhody jako Flash a Java.

Další možností tvorby aplikace v rámci jedné webové stránky bylo vložení tzv. iframů. Znovunačítání stránky se při uživatelské interakci dělo pouze v rámci tohoto iframu, nikoliv v rámci celé stránky. Toto řešení však představilo další problémy s bezpečností a navíc neřešilo problém znovunačtení stránky, protože se dělo dále, avšak jen v určité části stránky. Vývojáři ještě navíc museli udržovat jak hostitelskou verzi aplikace, tak tu vloženou (v iframu) a navíc řešit komunikaci mezi těmito dvěma částmi.

Javascript narozdíl od již zmíněných pluginů je však součástí prohlížeče. Jeho obrovskou výhodou je fakt, že funguje, aniž by musel uživatel instalovat dodatečný software. Dále umožňuje měnit obsah stránky, aniž by bylo třeba jakékoliv znovunačtení/obnovení stránky. Problémem v této době bylo však to, že Javascript nebyl dostatečně použitelný co se týče množství funkcionality jako Flash, a navíc byly interprety Javascriptu v prohlížečích pomalé. Toto ve výsledku napomohlo rozšíření Flashe.

Ajax

Změna k lepšímu pro Javascript začala okolo roku 2005 s nástupem technologie Ajax, neboli Asynchronous Javascript and XML. Tato technologie nebyla ničím nová, byla pouze kombinací stávajících technologií a objektu XMLHttpRequest. Obrovskou výhodou této technologie byla možnost asynchronních požadavků z klienta na server, což umožnilo překreslení pouze části stránky. Asynchronní operace také znamenaly zlepšení responzivity webových aplikací. Tyto operace totiž neblokují uživatelské rozhraní, které zůstává i při průběhu této operace použitelné k uživatelské interakci. Když je operace dokončena, neboli přijde výsledek od serveru, je pouze překreslena část stránky.

Dále byly implementovány knihovny jQuery a Prototype, které usnadňovaly manipulaci DOMu (Document Object Model). Tyto knihovny implementují abstrakci nad základními funkcemi Javascriptu, ale také umožňují vytvoření volání Ajaxu pomocí méně řádků kódu a ošetřují také různé chování v různých prohlížečích. Celkově pro vývojáře tyto knihovny přinesly velké usnadnění implementace aplikací využívajících klientský Javascript a Ajax. Kombinace knihoven a Ajaxu pomohlo překonat původní odpor k používání Javascriptu. Společnosti jako Microsoft nebo Yahoo se zapojily do vývoje nejznámějších Javascriptových knihoven, což dále pomohlo rozšíření těchto knihoven mezi programátorskou komunitu.

HTML5 a Javascript

Dalším krokem k modernímu webu bylo představení HTML5, u kterého se začlo s implementací kolem roku 2006. První veřejný fungující návrh HTML5 specifikace byl zveřejněn v roce 2008. HTML5 obsahuje mnoho nových prvků moderního webu, které jsou specifické pro Javascript. Umožňuje ovládat komunikaci, grafiku, multimédia a další, kde Javascript zastává důležitou roli. Také přispívá k motivaci zlepšit v prohlížečích interpret/engine Javascriptu. Právě kvůli HTML5 máme dnes rychlejší a hardwarem akcelerované webové prohlížeče. Dále přichází na řadu technologie jako Node.js nebo Meteor, které umožňují běh Javascriptu na serveru, díky čemuž se Javascript stává pro vývojáře ještě zajímavější.

Okolo roku 2006 skupiny W3C a WHATWG (Web Hypertext Application Technology Working Group) spolupracují na novém HTML standardu založeném na HTML4. Tato verze HTML však neměla žádnou revizi od roku 1997 a také neseděla do vývoje moderního webu. Nový standard HTML5 zahrnuje mnoho nových HTML elementů jako video a audio kvůli redukci závislostí na softwaru třetích stran (pluginů). Standard také obsahuje více než 50 nových API (Application Programming Interface - rozhraní pro programování aplikací), které umožňují zpřístupnění velkého množství funkcionality, která je na moderním webu

potřeba. Příkladem může být např. kreslení grafiky pomocí Javascriptu do elementu canvas nebo vložení multimédií do stránky bez nutnosti softwaru třetích stran jako Flash. Dále je možné vytvořit stále obousměrné spojení mezi klientem a serverem pomocí webových socketů (bude popsáno dále).

Další důležitou funkcí, která je zahrnuta v HTML5, je možnost využití API hardwaru zařízení, na kterém běží webová aplikace. Příkladem je využití geolokačního API, které umožňuje zjištění polohy zařízení, které obsahuje GPS modul. Dále je možné přistupovat např. k mikrofonu nebo webové kameře zařízení. Tyto funkce dále pomáhají k adopci Javascriptu jako jazyka pro vytváření mobilních aplikací.

Mobilní web a Javascript

V posledních letech se chytré telefony a tablety staly hlavními zařízeními, které lidé používají. Tyto zařízení mají různé operační systémy (Android, iOS, případně Windows), kde na každý OS je třeba pro implementaci aplikace znát specifické programovací jazyky a technologie. Jednou z cest jak vytvořit multiplatformní aplikaci je tvorba webové aplikace. Tyto aplikace jsou totiž nezávislé na druhu zařízení, protože běží v prohlížeči. Jelikož je webový prohlížeč dostupný na kterémkoliv dnešním operačním systému a zařízení, je tedy možné webové aplikace použít k oslovení co největšího počtu koncových uživatelů.

S pomocí HTML5 a Javascriptových API je mnohem jednodušší vyvinout webovou aplikaci, která je orientovaná na mobilní zařízení. Responzivní design umožňuje měnit vzhled stránky na základě velikosti obrazovky zařízení, takže stránka může být přizpůsobena jak na mobilní zařízení, tak na stolní počítače. Díky Javascriptovým API lze využívat polohu zařízení, orientaci pomocí akcelerometru, nebo události vygenerované při doteku uživatele na dotykový displej. Dokonce se dají využít vývojové platformy jako PhoneGap, které Javascriptovou aplikaci zabalí do kontejneru, který se pak dá nahrát do obchodů s aplikacemi na jednotlivých platformách (Google Play pro Android a App Store pro iOS).

Dokonce jsou v dnešní době už i operační systémy, které jsou naprogramovány v Javascriptu, nebo aspoň podporují implementaci aplikací v Javascriptu, které pak běží nativně v rámci OS. Windows 8 je takovým příkladem, kde se daly implementovat aplikace do Windows Store. Dalšími přiklady je např. Firefox OS nebo Chrome OS.

Na druhou stranu však Javascript stále není dostatečně vyzrálý jako jazyk pro vývoj aplikací, protože postrádá řadu důležitých aspektů, které vývojáři od takového jazyka očekávají. V dnešní době sice již existuje standard ECMAScript 6, avšak vývojáři se stále nemohou spolehnout na jeho plnou podporu ve všech webových prohlížečích. Řešením tohoto problému je použití Javascriptového preprocesoru, který je schopen transpilovat kód napsaný např. pomocí standardu ECMAScript 6 do standardu ECMAScript 5.

Javascriptové preprocesory

Javascript ve standardu ECMAScript 5 neobsahuje funkce jako importovatelné moduly, třídy (v Javascriptu se však dají používat prototypy) nebo rozhraní. Tyto funkce se dají napodobit pomocí stávajících struktur v jazyce, avšak je třeba kód psát tak, aby byl do budoucna udržovatelný. Může se totiž stát, že pak kód obsahuje těžko odhalitelné chyby.

Řešením je použití Javascriptových preprocesorů. Tyto preprocesory jsou vlastně překladače kódu, které transpilují Javascriptový (v novém standardu nebo nadmnožinu Javascriptu) kód do Javascriptu ve standardu ECMAScript 5, který je podporovaný téměř ve všech prohlížečích.

Existuje mnoho preprocesorů, nejznámějšími příklady jsou:

- Babel - transpiluje ECMAScript 6 na ECMAScript 5
- CoffeeScript - nadmnožina Javascriptu, umožňuje psaní kódu v sytaxi podobné jazyku Ruby
- TypeScript - opět nadmnožina Javascriptu, jednou ze stěžejních funkcí je přidání datových typů k proměnným a následná typová kontrola

Na konci této kapitoly provedu srovnání tradiční webové aplikace, nativní a single page aplikace. Nejprve tedy popíšu aplikaci tradiční webovou.

Tradiční webové aplikace

Jak již bylo zmíněno, tak od vzniku skriptovacích jazyků jako PHP nebo ASP měli vývojáři možnost psát aplikace, jejichž logika bežela na serveru. Tyto aplikace běží v prohlížeči a nevyžadují žádnou instalaci dodatečného softwaru na straně klienta. Aktualizace takové aplikace probíhá tak, že se nahraje nová verze na server, tudíž uživatel v jeden moment přistoupí na stránku, která je v nějaké určité verzi a po aktualizaci přistoupí na verzi novou. Pro uživatele je tedy taková aktualizace naprosto bez problému a jakékoliv instalace/interakce. Toto je obrovskou výhodou proti nativním aplikacím, u jejichž návrhu je nutné počítat s tím, že každý uživatel musí aplikaci aktualizovat sám, tudíž mohou existovat takoví uživatelé, kteří mají aplikaci v jiné verzi, než jiní. Navíc u nativních aplikací je třeba řešit, že mohou běžet na různých operačních systémech a různém hardwaru.

Tradiční webová aplikace vykresluje webovou stránku v momentu, když obdrží HTTP požadavek z webového prohlížeče uživatele. Takto sestavená stránka je pak odeslána klientovi zpět jako HTTP odpověď, což způsobí překreslení stránky. Při takovémto překreslení je nahrazena stávající stránka stránkou novou.

Díky tomu, že většina logiky aplikace je na serveru, je vývoj frontendu vysoce zjednodušen. Javascript se v tomto případě používá např. na vykreslení animací. Když uživatel interaguje se stránkou, ve všech případech se tím vytvoří HTTP požadavek na server, který musí takový požadavek vždy zpracovat. Nejen že to způsobí pokaždé znovunačtení stránky, ale navíc při čekání na odpověď webová aplikace nereaguje na žádné další interakce od uživatele. Toto způsobuje celkově horší uživatelskou přívětivost. Při přerušení připojení k internetu aplikace přestává fungovat úplně.

Dalším aspektem tvorby tradiční webové aplikace je držení a správa stavu. Stav je z převážné části udržován na serveru pomocí mechanismů jako sezení (session) a data jsou uložena straně serveru kompletně. Toto způsobuje problém, že při každém požadavku uživatele je třeba znovu data a stav získávat. Toto způsobuje uživateli další dodatečné čekání, což snižuje responzivitu.

Řešením, jak zvýšit responzivitu takové aplikace, je použití Ajaxu na operace, které se v rámci aplikace dějí často. Když uživatel chce přistoupit na nějakou stránku v rámci systému, nebo se jedná o jednorázové operace, lze tuto logiku řešit na serveru. Když však v aplikaci existuje např. formulář, u kterého je třeba v průběhu vyplňování validovat vstupy pomocí serveru nebo načítat ze serveru dodatečná data, je výhodné použít Ajax pro zlepšení responzivity. Taková hybridní aplikace se pak nazývá Rich web application, nebo Rich internet application (RIA). Je zřejmé, že tento přístup způsobí přesun části logiky na klienta, tudíž i složitější implementaci klientského kódu.

Nativní aplikace

Nativní aplikace jsou samostatně spustitelné aplikace, které je však třeba před spuštěním nainstalovat. Když jsou zkompilovány, mohou běžet pouze na jedné platformě, což je zároveň jejich výhoda a nevýhoda. Jsou totiž navrženy tak, aby využívaly lokálního hardwaru a operačního systému. Jelikož ale existuje takováto závislost na hardwaru a OS, nasazení nativních aplikací může být komplikované. V každé verzi aplikace je totiž nutné vzít v potaz verzi operačního systému, ovladačů, nebo procesorovou architekturu (32 nebo 64 bitová). Díky tomuto je vývoj nativních aplikací složitější a fáze nasazení softwaru se stává hlavním problémem. Jelikož existuje mnoho operačních systémů jako Windows, Mac OS, Linux, Android, iOS, atd., tak když tvůrce aplikace chce cílit na co nejširší publikum, musí také svou aplikaci zkompilovat pro všechny tyto platformy. Jakmile nějakou z platforem nezahrne, ochuzuje se o potenciální uživatele. Většinou je k tomu třeba většího množství vývojářů, kde každý z nich se specializuje na vývoj aplikace pro jednu platformu. Zpravidla tedy platí, že pro každou platformu je třeba aspoň jednoho vývojáře, což pochopitelně prodražuje celkový vývoj.

Veškerá logika aplikace je implementována na jednom místě, neexistuje tedy rozdělení na server a klienta, kde by bylo třeba se rozhodnout, kam přesunout více nebo méně logiky. Narozdíl od klasických webových aplikací si nativní aplikace mohou držet lokální stav pomocí lokální databáze nebo souborů. Toto umožňuje tvorbu uživatelsky přívětivé aplikace, jelikož se nemusí čekat na data ze vzdálených serverů. Další výhodou je možnost fungování aplikace i když uživatel není připojen k internetu.

Single page aplikace (SPA)

Jak bylo již popsáno na začátku této kapitoly, SPA jsou webové aplikace, které se skládají pouze z jedné webové stránky, která je používána jako obálka, do které se pomocí Javascriptu vkládají dynamicky HTML elementy a další prvky stránky, když uživatel se stránkou interaguje. Neexistují zde žádné požadavky na server, které by způsobily znovunačítání celé stránky a nejsou zde ani žádné vložené stránky jako v případě řešení pomocí iframu. SPA si berou výhodné věci jak z nativních, tak z klasických webových aplikací. Načtení stránky se děje pouze při první návštěvě uživatele webu, při jakékoliv interakci (i přecházení na jiné stránky) jsou pak implementovány pouze pomocí Javascriptu. Jakmile aplikace potřebuje nějaká data ze serveru, je k tomu použit Ajax. Po obdržení Ajaxových dat je pak pomocí Javascriptu překreslena část stránky, kde se data změnila.

Co se týče správy stavu, SPA si mohou držet lokální stav v prohlížeči uživatele, protože není potřeba po počátečním načtení stránky stránku znovunačítat. Stav se dokonce dá držet, i když uživatel není připojen k internetu pomocí HTML5 lokálního úložiště. Po jeho opětovném připojení k internetu se dá jeho stav synchronizovat se serverem opět bez nutnosti znovunačtení stránky.

SPA spojují výhodné věci jak z nativních, tak z klasických webových aplikací. Mohou běžet na všech platformách, na kterých běží webové prohlížeče, ale zároveň mají lokální správu stavu a responzivitu nativní aplikace. Jelikož se stále jedná o webové aplikace, když je třeba aktualizace, stačí ji provést na jednom místě (serveru) a tím je automaticky rozdistrubuována všech uživatelům. Uživatelé tyto aplikace nemusí instalovat, vývojáři je mohou také mnohem častěji aktualizovat (hlavně kvůli výhodě v předešlé větě). Většina logiky je implementována na straně klienta, server je využíván hlavně k autentifikaci, validaci a trvalému uložení dat do databáze. Jsou vysoce responzivní, takže uživatel má stejné uživatelský komfort jako nativní aplikace. Při čekání na data ze serveru aplikace nezamrzne. Uživateli

se dá zobrazit animace, která ho v reálném čase upozorní, že se v rámci aplikace právě něco načítá. Při tomto načítání však může aplikaci používat nadále. Všechny tyto výhody dělají z SPA skvělý nástroj k vývoji webových aplikací nové generace.

3.3 Webové sockety a komunikace v reálném čase

Jak bylo nastíněno v sekci 3.2, jsou webové sockety obsaženy v rámci standardu HTML5. Před HTML5 byla komunikace mezi okny prohlížeče a rámci (iframe) omezena z bezpečnostních důvodů. Jak se ale web vyvíjel a webové aplikace začaly slučovat obsah a webové aplikace z jiných stránek, bylo potřeba nějak zařídit, aby spolu mohly tyto aplikace komunikovat. Řešením byla technologie Cross-Document Messaging (CDM) implementovaná do většiny nejpoužívanějších prohlížečů, která zavedla bezpečnou komunikaci z různých zdrojů (cross-origin) mezi okny prohlížeče a rámci. Technologie definuje postMessage API jako základní cestu, jak posílat a přijímat zprávy/data. Existuje mnoho případů, proč vývojář aplikace potřebuje využít obsah z různých domén, např. chat nebo sociální sítě, pro komunikaci v rámci okna prohlížeče. CDM poskytuje cestu, jak předávat asynchronní zprávy mezi jednotlivými Javascriptovými kontexty [10].

HTML5 specifikace pro CDM dále ujasňuje a vylepšuje zabezpečení domén zavedením pojmu origin (zdroj), který se skládá ze schéma, hostitele a portu. Dvě URI jsou považovány jako patřící k jednomu zdroji právě tehdy, když mají stejné schéma, hostitele a port. Cesta (část URI za portem) už není považována jako hodnota zdroje. Následující příklady URI nepatří do stejného zdroje:

- Nesouhlasí schéma: http://www.mujweb.cz a https://www.mujweb.cz
- Nesouhlasí hostitel: https://www.mujweb.cz a https://www.jinyweb.cz
- Nesouhlasí port: https://www.mujweb.cz:3000 a https://www.mujweb.cz:3001

Následující příklad ukazuje dvě URI, které patří do jednoho zdroje:

- Souhlasí schéma, hostitel a port, nesouhlasí cesta: https://www.mujweb.cz/stranka1 a https://www.mujweb.cz/stranka2

CDM překonává tuto limitaci povolením zasílání zpráv mezi různými zdroji. Při odesílání zprávy odesílatel specifikuje zdroj příjemce a při obdržení zprávy je pak zdroj odesílatele součástí této zprávy. Zdroj zprávy je zajištěn prohlížečem a nemůže být podstrčen. Na straně příjemce je pak možné se rozhodnout, které zprávy ignorovat. Možností je také použití tzv. whitelistu, neboli seznamu URI, které obsahují pouze důvěryhodné zdroje.

CDM je skvělým příkladem HTML5 specifikace, která umožňuje zjednodušení komunikace mezi webovými aplikacemi. Každopádně je zaměřena pouze na komunikaci mezi okny a rámci prohlížeče. Neřeší ale, narozdíl od webových socketů, komplikace v rámci protokolu.

Jeden z předních autorů HTML5 specifikace, Ian Hickson, přidal webový socket do komunikační sekce této specifikace. Původně byl webový socket nazván TCPConnection, každopádně se pak vyvinul do samostatné specifikace. Ačkoliv dnes existuje webový socket mimo HTML5 specifikaci, je důležitý pro dosažení komunikace v reálném čase u moderních webových aplikací založených na HTML5.

Přehled starších HTTP architektur

K pochopení důležitosti webových socketů je třeba se nejprve podívat na starší architektury, které používají HTTP. Zde je konektivita vyřešena pomocí protokolů HTTP/1.0 a HTTP/1.1. Tyto protokoly jsou typu požadavek-odpověď v modelu klient/server, kde klientem je webový prohlížeč, který zasílá HTTP požadavek na server, který klientovi odpovídá s požadovanou webovou stránkou a případně dodatečnými potřebnými daty. HTTP bylo také navrženo pro získávání dokumentů. Rozdíl mezi verzemi 1.0 a 1.1 je takový, že starší verze otevírá separátní spojení pro každý požadavek na server, což však u novějších webů, kde je pro úspěšné načtení stránky třeba načítat desítky (nebo stovky) různých zdrojů, není moc škálovatelné. Verze 1.1 umožňuje již znovupoužití již otevřeného spojení. Prohlížeč je tedy schopen po načtení např. základního HTML znovupoužít dané spojení pro načtení kaskádových stylů a Javascriptu, což výrazně urychluje načtení celého webu. HTTP je bezstavové, což znamená, že každý požadavek interpretuje jako unikátní a nezávislý. Tento přístup má samozřejmě své výhody a nevýhody. Výhodou je např. že server si nemusí držet informaci o sezení (session), díky čemuž není vyžadováno ukládání žádných dat. Na druhou stranu je však v každém požadavku posílat dodatečné informace. Hlavní nevýhody HTTP vycházejí z následujícího:

- HTTP bylo navrženo pro sdílení dokumentů, ne pokročilých interaktivních webových aplikací
- Množství informací, které musí HTTP odeslat mezi klientem a serverem, může hodně narůst z důvodu časté interakce

HTTP je poloduplexní (half-duplex), neboli umožňuje pouze jednosměrnou komunikaci. Klient nejprve zašle požadavek na server, který pak odpovídá. Toto je pro interaktivní aplikace neefektivní, což mělo za následek snahu vývojářů, kteří přišli s řešením následujících metod: polling, long polling a streamování.

Když uživatel navštíví stránku, je na server odeslán HTTP požadavek. Server po potvrzení požadavku odesílá odpověď prohlížeči zpět. Tato informace však po nějaké době může být již neaktuální (např. stav akcií). Pro získání aktuálních informací musí tedy uživatel stránku znovunačíst, což není optimální.

Polling umožňuje pravidelné automatické dotazování serveru, což je výhodné, když je známo, za jak dlouho se data aktualizují. Když toto známo není, je pak server zbytečně zahlcen zbytečnými požadavky. Long polling získává informace od serveru tak, že odešle požadavek, na který je odpovězeno pouze tehdy, když jsou data aktualizována. Server tedy drží spojení do vypršení, nebo do doby aktualizace dat. Když pak klient obdrží buď data nebo vypršení spojení, odesílá požadavek znovu. Když je však třeba posílat velké množství aktualizací, nemá long polling oproti polling žádné výhody. Další nevýhodou je neexistence žádné standardizované implementace.

U streamování odesílá klient požadavek a server udržuje spojení otevřené, pokud je využíváno (buď donekonečna, nebo je nastaven časový limit). Toto řešení se může zdát jako nejvýhodnější pro aktualizace dat, které chodí v neočekávaných intervalech. Server však v tomto případě nikdy neodešle klientovi HTTP odpověď, která by znamenala ukončení spojení. Spojení je tedy nepřetržitě otevřeno. Proxy servery a firewally mohou HTTP odpovědi ukládat do paměti (buffer), což může způsobit vyšší latenci v rámci komunikace na síti.

Tyto metody umožňují komunikaci téměř v reálném čase, avšak zároveň vyžadují zasílání HTTP hlaviček, které obsahují spoustu nepotřebných dat navíc. Latenci také zvyšuje nutnost čekání klienta na každou odpověď před možností odeslání dalšího požadavku.

Webové sockety

Řešením problémů, které byly popsány v předchozí kapitole, jsou webové sockety, které umožňují celoduplexní (full-duplex), obousměrné spojení. V tomto případě se HTTP požadavek stává jediným požadavkem, který otevírá websocketové spojení. Po připojení je pro komunikaci využíván tento jeden kanál jak pro komunikaci klient-server, tak server-klient. Latence je snížena tak, že server může zasílat zprávy klientovi kdy je třeba. Server nemusí čekat na požadavek klienta a zároveň klient může serveru zasílat zprávy kdykoliv.

Webové sockety snižují množství přenesených dat po síti, šetří využívané hardwarové prostředky (hlavně využití procesoru). Dále zjednodušují implementaci aplikací, kde je třeba komunikace v reálném čase. Díky definovaným standardům umožňují implementaci znovupoužitelných komponent v aplikaci, které mohou být napojeny na různé servery, které podporují webové sockety. Umožňují TCP styl komunikace pro HTML5 aplikace bez ohrožení bezpečnosti v rámci prohlíče. Všechny tyto výhody shrnují motivaci, proč je výhodné webové sockety u prohlížečové hry využít.

3.4 GraphQL

Podle oficiální dokumentace je GraphQL popsáno jako dotazovací jazyk, který byl navržen pro implementaci klientských aplikací poskytující intuitivní a flexibilní syntax a systém pro popis jejich datových požadavků a interakcí. Hlavním cílem GraphQL je sjednotit všechny datové požadavky aplikace do jednoho koncového bodu a snížit tím počet požadavků, kterými se žádá o data [9].

Než budu popisovat GraphQL, je třeba zmínit systém pro získávání dat, který se GraphQL snaží nahradit. Jedná se o REST API, které slouží k získávání, vytváření, úpravě a mazání dat v systému, které poskytuje sadu definovaných koncových bodů pro přístup k jednotlivým částem datových transakcí. Nevýhodou REST API je nedostatek flexibility, protože při změně požadavků a implementace aplikace je třeba pokaždé měnit tyto koncové body. I kdyby vývojář udržoval spolu s požadavky aplikace koncové body, je zde problém se získáváním dat. Většinou je REST API implementováno tak, že jeden koncový bod odpovídá jedné tabulce v databázi. Operace CRUD (create - vytvoření, read - čtení, update - úprava, delete - smazání) jsou pak zpřístupněny pomocí HTTP metod (většinou POST, GET, PUT, DELETE) na jednotlivých koncových bodech. Když si tedy uživatel zažádá o data na některém koncovém bodě, dostane veškerá data, která se vyskytují v databázi v rámci tabulky, která patří k danému koncovému bodu. Toto se dá vyřešit implementací parametrů, kde pak může uživatel data filtrovat (např. pomocí ID, nebo může být implementováno stránkování). I když si ale uživatel vyžádá data s filtrem, vždy dostane minimálně jeden záznam z tabulky, který obsahuje všechny parametry. Příkladem může být tabulka s uživateli, ve které je uloženo uživatelské ID, jméno, e-mail, heslo a další informace. Aplikace může vyžadovat pouze jméno, ale daný koncový bod REST API vrátí všechny položky. Dalším problémem je, když uživatel nebo aplikace potřebuje data z více tabulek. V tomto případě je třeba vytvořit tolik dotazů, z kolika tabulek jsou požadována data. Tyto data je pak třeba spojit v klientské aplikaci, což může být náročný proces. Toto by se teoreticky dalo řešit vytvořením dodatečných koncových bodů, které by daná spojení dat provedla na straně serveru a uživateli vrátila již spojená data. Toto řešení je však velice nepraktické, protože počet databázových spojení může být enormní.

Cílem GraphQL je řešení těchto problémů. Jak bylo zmíněno, GraphQL obsahuje pouze jeden koncový bod, pomocí kterého se uživatel dotazuje na veškerá data v rámci aplikace.

```
{
  resources {
    id
    player
    type
    count
  }
}
```

Obrázek 3.1: Příklad GraphQL dotazu, který žádá o suroviny

Jelikož obsahuje vlastní dotazovací jazyk, umožňuje uživateli vytvářet pro každý dotaz úplně vlastní dotazy, které mohou data z tabulek filtrovat a spojovat, ale také specifikovat jednotlivé parametry, které jsou vyžadovány. Toto nejen usnadňuje implementaci klientské aplikace, ale zároveň snižuje množství přenesených dat po síti.

Základy GraphQL

Dotazovací jazyk GraphQL leží v aplikační vrstvě aplikace a je možné ho použít pro jakýkoliv backend, který splňuje požadavky GraphQL protokolu. Data jsou zde popsána pomocí dotazů a grafu (místo koncových bodů pro každou tabulku jako v REST API). Tento graf umožňuje mnohem větší flexibilitu a škálovatelnost. Úprava definice dat je velice jednoduchá, protože se vlastně jedná o přidávání nebo odebírání uzlů z grafu. Data jsou reprezentována jako list položek, který může obsahovat další, vnořené položky, které mají vztahy s ostatními uzly.

Jednotlivé GraphQL dotazy jsou pak prováděny pomocí jednoho koncového bodu, který umožňuje dotazy na všechna možná data v rámci aplikace. Toto umožňuje mít pouze jednu URL adresu, která poskytuje skrz API všechna data. Klienti si tedy mohou v rámci aplikace navrhnout vlastní dotazy a nezatěžovat tak dodatečnou implementací vývojáře backendu/API, což umožňuje vyšší flexibilitu a udržovatelnost aplikace.

Jednoduchý příklad GraphQL dotazu je obr. 3.1. Při odeslání takovéhoto dotazu na server, který podporuje GraphQL má za výsledek odpověď 3.2.

Jak lze vidět z příkladu, struktura odpovědi je přesně definována dotazem klienta, což mu umožňuje velkou volnost. Když se změní požadavky klienta na data, stačí pouze změnit strukturu a odeslat dotaz na stejný koncový bod.

GraphQL používá pro definici dat datové typy, které je nutné definovat na straně GraphQL serveru. Díky tomu jsou klienti schopni znát strukturu dat, které se budou vyskytovat v odpovědi. Dále jsou zde mechanismy pro filtrování dat. Jednotlivé položky mohou být totiž definovány jako funkce, které se dají volat s parametry. Tyto parametry pak mohou dále měnit chování funkce nebo strukturu dat v odpovědi. Když by byl u příkladu se surovinami specifikován např. parametr id, uživatel by se mohl místo na všechny suroviny dotazovat pouze na jednu surovinu. Tato funkcionalita však samozřejmě musí být implementována na straně GraphQL serveru.

Kromě flexibilního a efektivního čtení hodnot umožňuje GraphQL také operace CRUD. Oproti REST API, kde jsou tyto funkce specifikovány pomocí HTTP metod, GraphQL umožňuje tři typy operací, které jsou nezávislé na HTTP metodách:

```
{
  "resources": [
    {
      "id": 0
      "player": "Michal",
      "type": "wood",
      "count": 0
    },
    další suroviny
    { ... }
  ]
}
```

Obrázek 3.2: Příklad GraphQL odpovědi, která obsahuje seznam surovin

```
mutation harvest($type: String!) {
    harvest(type: $type) {
        id
        count
    }
}
```

Obrázek 3.3: Příklad GraphQL mutace, která mění stav surovin zadaného typu

- Query (dotaz) - umožňuje čtení dat (popsáno)
- Mutation (mutace) - umožňuje změnu dat, zároveň ale obsahuje odpověď pro klienta
- Subscriptions (odebírání) - možnost rozeslání zpráv pomocí webových socketů při jakékoliv akci

Příkladem mutace je 3.3. Tato operace změní data v databázi a zašle klientovi zpět data, o která si zažádal (id a count). Vývojář backendu je zodpovědný za implementaci těchto operací.

Co se týče odebírání, server rozesílá události všem klientům, kteří jsou přihlášení k jejich odebírání. Rozeslání události může být implementováno v kterékoliv GraphQL funkci (dotaz nebo mutace).

Kapitola 4

Studium genetických algoritmů

Genetické algoritmy jsou podmnožinou evolučních algoritmů, které pro výpočet řešení využívají principů Darwinovy teorie přirozeného výběrů druhů (evoluce). Každý druh v rámci populace reprezentuje řešení, které se dá ohodnotit pomocí tzv. fitness funkce. Na vstupu této funkce je jedinec a na výstupu je číslo. Genetické algoritmy řeší tzv. optimalizační problémy, kde v tomto případě se snaží nalézt buď minimální nebo maximální hodnotu fitness funkce. Jedinec je zpravidla reprezentován binárně (jedničkami a nulami), ale dovoleny jsou také jiné formy reprezentace (např. pomocí celých čísel) [6]. Princip algoritmu by se dal popsat pomocí těchto kroků:

1. Inicializace počáteční populace (může být buď náhodně vygenerovaná nebo vygenerovaná pomocí nějaké heuristiky pro lepší počáteční řešení, když je to možné)
2. Zpravidla náhodný výběr jedinců z populace, kteří mají vysoké skóre ohodnocené fitness funkcí (tímto krokem se vyřadí jedinci s malým skóre z populace)
3. Vygenerování nové generace pomocí následujících funkcí:
   (a) Křížení - vstupem jsou 2 rodiče, kteří vyprodukují zpravidla 2 potomky tak, že oba potomci obsahují genetický kód obou svých rodičů
   (b) Mutace - vstupem je jedinec, u kterého je náhodně změněn genetický kód
4. Ohodnocení nové generace pomocí fitness funkce
5. Kontrola, zda-li byla splněna ukončující podmínka:
   (a) Pokud byla splněna, ukončení algoritmu, kde výstupem je jedinec s nejvyšším skóre (případně výsledná populace, nebo i historie všech populací pro výpočet statistik)
   (b) Pokud splněna nebyla, pokračuje se bodem 2

Tento obecný popis algoritmu může být v každém kroku v jistých směrech odlišný, když je to třeba v rámci řešení dané úlohy. Příkladem je tzv. elitismus, kdy se jedinec ze stávající populace zkopíruje do populace následující beze změn. Zpravidla se takto kopíruje jedinec s nejvyšším skóre, kde motivací je zachování nejlepšího řešení z jedné generace na druhou. Výběrem jedinců náhodou se může stát, že se takovýto jedinec ztratí.

Je také potřeba si dát pozor na nastavení pravděpodobností, podle kterých se aplikují genetické operátory mutace a křížení. Operátor křížení se aplikuje zpravidla s mnohem vyšší

pravděpodobností, než operátor mutace, ale nemusí to tak být ve všech případech, opět zde záleží na typu řešené úlohy. Je ještě třeba uvést, že operátory křížení a mutace se aplikují po sobě tak, že nejprve se zjistí podle vygenerované pravděpodobnosti, zdali se aplikuje operátor křížení, a potom se zjistí, zdali se aplikuje operátor mutace (aplikace operací tedy není vzájemně exkluzivní).

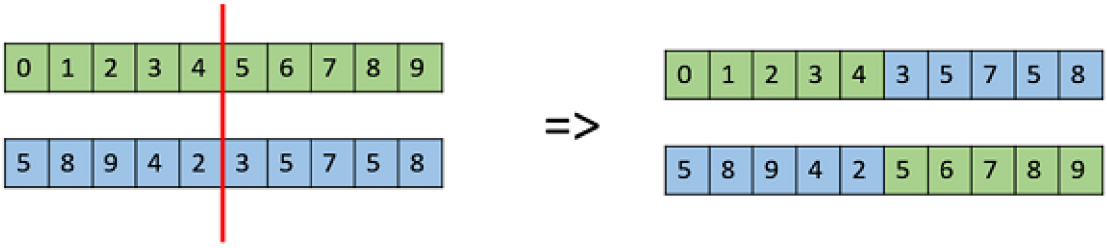
4.1 Křížení

Genetický operátor zajišťující přenos genetického kódu z generace na generaci. Dokáže generovat nová řešení pomocí rekombinace řešení stávajících. Metoda však není schopna prohledat stavový prostor tak, aby nalezla řešení, které se nedají vygenerovat z počáteční generace. K tomuto účelu slouží mutace. Pro ilustraci uvedu 2 příklady křížení:

Jednobodové křížení

Vstupem funkce jsou chromozomy 2 rodičů. Nejprve je třeba zjistit, zdali jsou tyto chromozomy stejně velké. Když nejsou, použije se délka menšího chromozomu. Z délky chromozomu se pak vybere náhodný bod, podle kterého bude provedeno křížení. Výsledný potomek pak obsahuje jednu část chromozomu prvního rodiče a jednu část chromozomu druhého rodiče. U tohoto křížení jako výstup dávají smysl pouze jeden nebo dva potomci, více potomků by pak už mělo shodný chromozom s ostatními sourozenci. Při potřebě generovat více než 2 potomky je tu pak vícebodové křížení.

Obrázek 4.1: Ukázka jednobodového křížení [1]

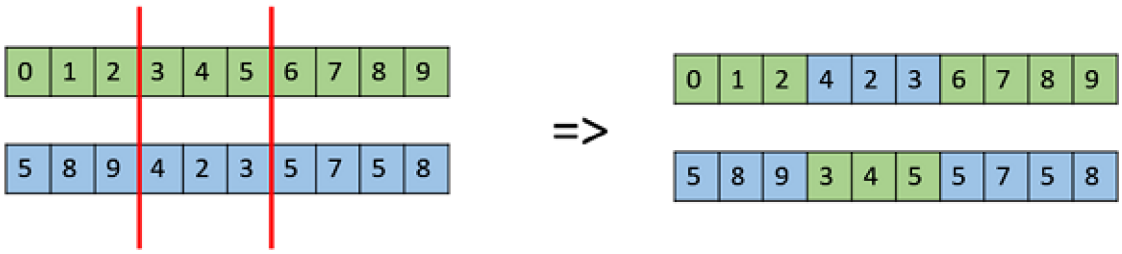


Vícebodové křížení

Funguje obdobně jako jednobodové křížení s tím rozdílem, že náhodně vygenerovaných bodů, podle kterých se provádí křížení, je více. Nejčastější variantou je dvoubodové křížení, u kterého může vzniknout až 9 různých kombinací potomků.

[1] Zdroj: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm

Obrázek 4.2: Ukázka vícebodového křížení [2]



4.2 Mutace

Druhý genetický operátor zajišťuje změnu genetického kódu z generace na generaci pomocí pouze jednoho jedince. Změna kódu se děje vždy pomocí stochastické funkce. Příkladem může být například změna hodnoty genu nebo záměna genů v rámci chromozomu.

4.3 Ukončení algoritmu

Algoritmus je ukončen, když je splněna podmínka pro jeho ukončení. Mezi tyto podmínky patří:

1. Skóre nejlepšího/průměrného/nejhoršího jedince v rámci populace dosáhlo určité hodnoty
2. Bylo dosaženo určitého počtu generací
3. Změna skóre nejlepšího/průměrného/nejhoršího jedince mezi generacemi (může se sledovat i několik generací) dosáhla určité hodnoty

[2] Zdroj: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm

Kapitola 5

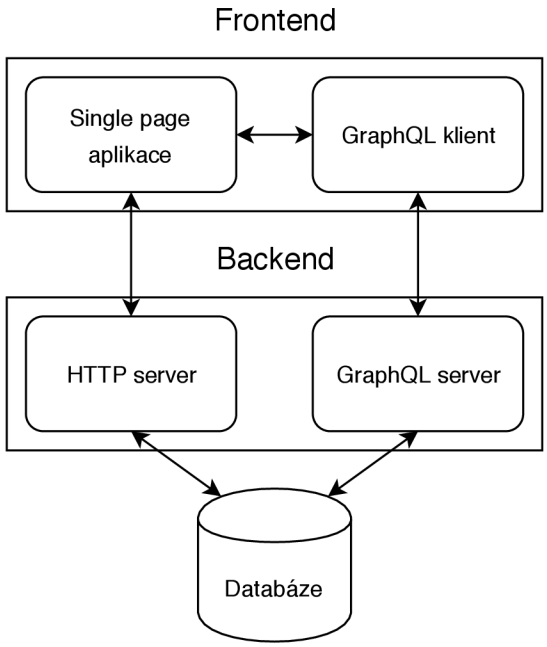
Návrh webové hry

Návrh hry by se dal rozdělit na návrh backendu (NoSQL databáze, HTTP server Express.js, GraphQL server, serverová část webových socketů) a frontendu (React, knihovna Apollo, klientská část webových socketů). V této kapitole bude popsán návrh architektury celé hry, ale také jednotlivých komponent.

5.1 Architektura

Než popíšu návrh konkrétních stavebních prvků aplikace dopodrobna, chtěl bych nejprve popsat architekturu. Pro ukládání perzistentních dat je nutné využít databáze, kterou využívá backend aplikace, konkrétně GraphQL server, který data poskytuje GraphQL klientu. Jelikož je hra standardní webovou aplikací, je pro samotné zobrazení stránky nutné využít HTTP server. Frontend tedy funguje tak, že samotná stránka se zobrazí pomocí HTTP serveru, ale dynamická data získaná z databáze se získají pomocí GraphQL, které zároveň zajišťuje aktualizace těchto dat na základně uživatelských vstupů.

Obrázek 5.1: Diagram architektury prohlížečové hry



5.2 Základní princip hry

Při vymýšlení konceptu hry jsem se inspiroval různými herními koncepty z mnoha jiných her. Hra je vlastně strategií zkombinovanou s inkrementální hrou. Hra obsahuje suroviny, které musí hráč buď sbírat, nebo za ně postavit pokročilejší suroviny nebo budovy, pomocí kterých se dá výroba surovin automatizovat. K samotné automatizaci je nejen nutné mít dostupné obyvatele, ale také správné nástroje, které jsou obyvateli přiřazeny k práci v budově. Obyvatelé pak musí být ubytováni a hráč musí mát dostatečnou produkci jídla, aby měli co jíst. Snažil jsem se držet co se týče návaznosti surovin a významu budov co nejvíce reality, a proto má hráč na začátku k dispozici možnost sbírat surové dřevo, ze kterého může vyrábět dřevo zpracované, pomocí kterého se pak dají vyrábět nástroje a budovy. Ve hře je kladen velký důraz na ekonomickou část hry, tudíž jsem si co nejvíce usnadnil možnost hru rozšířit o další suroviny a budovy. Hráč může také kromě budování ekonomiky interagovat s ostatními hráči buď pomocí obchodu nebo válek.

Je ale hra v něčem originální oproti konkurenci? Opravdovou konkurencí této hry jsou převážně hry prohlížečové, které umožňují hráčům hrát proti ostatním. Když hru srovnám s Travianem, je zde mnoho rozdílných herních principů. V Travianu má hráč k dispozici pouze 4 suroviny, se kterými si vystačí po celou hru. Naopak je kladen důraz na souboje mezi hráči a stavbu nových vesnic, o které je třeba se starat. Hráč je tedy motivován postupovat hrou tím, že se stará o stále větší množství vesnic a stará se o obranu. U mojí hry je hráč naopak motivován výrobou nových surovin, avšak prvek interakce s ostatními pomocí válek je zde stále zachován. Hráč se celou hru stará pouze o jednu vesnici/město, což mi přišlo jako velké vylepšení oproti Travianu, kde se v pozdní fázi hry musel hráč většinu času starat o každou vesnici zvlášť. Toto způsobilo nutnost provádět stejné nebo podobné úkony v každé vesnici

a přispívalo k repetitivnosti. Moje hra v momentálním stavu je plně hratelná, avšak mám v plánu ji ještě hodně vylepšit, aby obsahovala dostatek obsahu pro hráče a byla zábavná.

Další výhodu vidím v implementaci hry jako single page aplikaci. Většina stávajících velmi hraných a oblíbených prohlížečových her jsou implementovány jako standardní klient-server aplikace, kde při jakékoliv akci hráče musí dojít k znovunačtení celé stránky. Single page aplikace je zároveň mnohem rychlejší pro uživatele, tak i méně náročná na serverové prostředky, protože není třeba na serveru po každé akci hráče sestavovat celou stránku znovu (stačí jenom poslat data, která jsou důležitá pro aktualizaci stavu). Co se týče dalších odlišností od konkurence, tak ty budou popsány dále v závěru.

Před popisem databáze je třeba podrobněji vysvětlit jednotlivé části hry, od kterých se odvíjí samotný návrh databáze:

Základní suroviny

Tyto suroviny se dají získat buď manuální těžbou hráče, nebo vyrobením v budově. K jejich získání nejsou potřeba žádné jiné suroviny. Budova, která vyrábí tento druh suroviny, potřebuje pouze obyvatele, kteří jsou vybavení správným nářadím. Příkladem je surové dřevo, kamení a jídlo.

Pokročilé suroviny

K získání tohoto druhu suroviny již jsou třeba buď základní nebo jiné pokročilé suroviny. Surovinu může vyrobit buď hráč manuálně, nebo výrobu automatizovat v budově. Budovy, které tyto suroviny vyrábějí, potřebují kromě vybavených pracovníků také dostatečnou produkci závislých surovin. Příkladem je prkno, k jehož vyrobení je třeba dřevo, které se vyrábí ze surového dřeva. Mezi pokročilé suroviny také spadá nářadí a zbraně, k jejichž výrobě jsou třeba jiné suroviny.

Budovy

Budovy slouží k automatizaci výroby surovin. Stejně jako pokročilé suroviny, také budovy se vyrábějí z jiných surovin. Rozdílem je však schopnost přiřazovat budovám obyvatele, kteří výrobu automatizují. Ve hře existují 2 druhy budov: budovy produkující suroviny a budovy ubytující obyvatele. Budovy produkující suroviny fungují tak, že produkují suroviny po přiřazení obyvatele vybaveného příslušným nástrojem. Aby mohl hráč obyvatele mít, musí ale také stavět budovy, ve kterých obyvatele ubytuje. Tyto budovy spotřebovávají jídlo, které je nutné mít pro jejich stavbu.

Obyvatelé

Obyvatelé mohou být využiti pro automatizaci výroby, jak bylo uvedeno výše, ale také jako obchodníci a vojáci. Pro provedení obchodu s ostatními hráči je třeba nejen postavit tržiště, ale také mu přiřadit obchodníky, kteří samotné obchody provedou. Dalším využitím je armáda. Hráč může postavit kasárny, ve kterých se dají rekrutovat vojáci, kde místo nástrojů jsou obyvatelé vybaveni zbraněmi a je možné je využít na obranu vesnice, nebo útok na jiné hráče.

Obchod

Každý obchod je reprezentován transakcí mezi hráči. Aby mohl obchod vůbec proběhnout, jeden hráč musí obchod vytvořit, kde specifikuje, kolik surovin nabízí a kolik surovin hledá. Nelze vytvořit obchod, který pouze hledá suroviny a žádné nenabízí, nebo suroviny jen nabízí a žádné nehledá. Pro úspěšné dokončení obchodu je nutné schválení od jiného hráče, který obchod akceptuje.

Armáda

Souboje fungují na podobném principu jako obchod, ale s tím rozdílem, že souboj proběhne pouze díky jednomu hráči (nedávalo by smysl, aby obránce musel akceptovat souboj). Souboj je reprezentován obdobnou transakcí jako obchod, avšak kromě surovin, které po vítězném souboji útočník získá, musí být navíc vypočítány ztráty vojáků jak na útočníkově, tak na obráncově straně.

5.3 Databáze

Z důvodu implementace hry v Javascriptu jak na backendu (node.js), tak na frontendu, ale také kvůli horizontální škálovatelnosti pro rozšiřitelnost hry do budoucna a potenciální připravenost pro velké množství uživatelů jsem zvolil NoSQL databázi MongoDB. V této sekci bude popsán návrh všech částí (kolekcí) databáze použité ve hře.

Kolekce uživatele

Každému hráči se ukládá jeho postup do databáze, odkud se po každém dotazu na server získává aktuální stav hry. Z toho vyplývá, že každý hráč musí mít ve hře vytvořen účet, pomocí kterého se jednoznačně identifikuje jeho záznam v databázi. K tomuto slouží autorizace. Nevýhodou je, že když si někdo chce hru pouze vyzkoušet, aby vůbec zjistil, že ho baví, musí projít zdlouhavou registrací.

Kvůli tomuto je ve hře také možnost hrát bez registrace. Zde však nastává několik problémů:

1. Kam ukládat postup hráče?
2. Jak zjistit, že po opětovné návštěvě stránky se jedná o stejného hráče, který hru hrál předtím?
3. Co se stane, když se hráč rozhodne, že ho hra baví a chce si založit účet? Musí začít od začátku nebo se mu převede jeho postup z dočasného účtu na jeho registrovaný účet?
4. Co se stane, když hra přestane hráče bavit? Po jaké době je možné mu odstranit jeho postup?

Jelikož hráči nemají účet ve hře, a tudíž ani uživatelský záznam v databázi, zdálo by se logické ukládat postup přes cookie do počítače uživatele. Zde však nastává problém, že tyto data jsou potenciálně přístupná danému uživateli, tudíž hrozí riziko jejich nežádoucí úpravy za účelem obohacení/zrychlení postupu/podvodu ve hře. Aby si takovýto hráč hru plnohodnotně vyzkoušel, je potřeba ho nechat interagovat s ostatními hráči a tudíž eliminovat riziko podvádění. Toto se dá vyřešit jedině tak, že postup hráče se bude ukládat do

databáze. Jak ale identifikovat daného hráče? Při prvním spuštění hry v tomto testovacím módu se hráči vytvoří v databázi záznam, stejně jako hráči registrovanému. Rozdílem ale je, že neregistrovanému hráči se vygeneruje unikátní hash, který se uloží pomocí cookie do jeho počítače. Tímto hashem se pak neregistrovaný hráč automaticky identifikuje (samotný uživatel neví o existenci žádného hashe). Nevýhodou je, že tímto způsobem se postup ukládá pouze na jediném zařízení, každopádně výsledek by byl stejný v případě ukládání celého postupu do zařízení uživatele.

Zde nastává první bezpečnostní problém: když vstoupím do hry v testovacím módu hry a nemám v cookie uložen identifikační hash, hra mi vytvoří záznam v databázi a vygeneruje hash. Teoreticky je tedy možné nechat si vygenerovat hash, vymazat cookies a tuto proceduru opakovat do nekonečna (a nebo do pádu kterékoliv backendové komponenty). Je tedy nutné vyřešit tyto věci: omezit možnosti použití robota, který by uměle vytvářel účty ve hře, včas mazat neaktivní účty a omezit celkový počet vytvořených neregistrovaných účtů. Před každým startem hry v testovacím režimu musí hráč projít systémem Recaptcha od společnosti Google. Tento systém by měl omezit vytváření účtů robotům. Jednotlivé účty budou smazány po týdnu neaktivity a celkový počet takovýchto testovacích účtu je omezen na 10 000. Když se dosáhne tohoto limitu, žádní noví hráči si hru nemohou vyzkoušet bez registrace a musí počkat, dokud nebude odstraněn nějaký z neaktivních účtů.

Když se hra uživateli zalíbí, bude mít také možnost se zaregistrovat a převést svůj postup z testovacícho účtu na svůj registrovaný účet. Tímto se také automaticky smaže jeho testovací účet.

Kolekce uživatelů obsahuje základní informace o registrovaném uživateli. Uživatel se může registrovat několika způsoby: pomocí e-mailové adresy a hesla nebo autorizací svého stávajícího účtu na Facebooku nebo Googlu přes OAuth 2.0. Kolekce tedy obsahuje e-mail, heslo, facebook-id a google-id. Kromě těchto parametrů obsahuje také uživatelské jméno, které musí být unikátní a podle kterého je hráč v rámci hry dohledatelný (když s ním bude chtít někdo obchodovat nebo na něj zaútočit, uvidí jeho uživatelské jméno). Posledním parametrem je hráčovo skóre, které je automaticky počítáno z jeho ekonomické, obchodní a vojenské výkonnosti (konkrétní výpočet je uveden v kapitole 6.3).

Tabulka 5.1: Kolekce uživatelů

Sloupec	Datový typ
username	String
password	String
email	String
score	Number

Kolekce surovin

Jak bylo popsáno v sekci 5.2, hra obsahuje základní a pokročilé suroviny. V databázi je potřeba ukládat jejich počet, kterému hráči daná surovina patří a co je to za surovinu. Kromě těchto dynamických parametrů, které se mění v průběhu hry, jsou suroviny definovány statickými parametry, které suroviny definují pro všechny hráče: např. suroviny, které jsou potřeba k vyrobení suroviny (u pokročilé), nebo čas, za který se surovina vyrobí (u základní). Statické parametry budou popsány v sekci 5.4. Kromě těchto základních 3 dynamických parametrů je třeba v databázi držet také časové razítko, kdy byl naposledy

změněn stav suroviny. Jelikož získávání základní suroviny trvá určitý čas, je třeba zabránit hráči, aby mohl získat suroviny více, než je možné na úrovni backendu. Při získání suroviny je nejprve provedena kontrola, zda-li má hráč na surovinu nárok (od poslední doby, kdy byl změněn stav suroviny, uběhl předem definovaný čas, po jehož uplynutí má hráč na surovinu nárok).

Kromě tohoto časového razítka je také třeba ukládat čas v případě automatizovaného získávání suroviny. Když je hráč přihlášen ve hře, suroviny mu přibývají v reálném čase žádostmi backendu o aktualizaci stavu. Řekněme, že hráč má produkci jednoho dřeva za sekundu. Každou vteřinu se mu tedy aktualizuje stav a zvýší počet dřeva o jedno. Když se ale hráč odhlásí a přihlásí se až po nějaké době, hra musí být schopná mu přidělit suroviny, které by získal, kdyby byl po celou dobu přihlášen. K tomuto je tedy třeba ukládat toto časové razítko.

Tabulka 5.2: Kolekce surovin

Sloupec	Datový typ
player	String
type	String
count	Number
autoUpdated	Date
playerUpdated	Date

Kolekce budov

U budov narozdíl od surovin přibývá další dynamický parametr, a to počet přiřazených pracovníků. Budovy, které ubytovávají obyvatele, mají tento parametr vždy nulový, jelikož počet ubytovatelných lidí je vždy statický v rámci budovy. Pro výpočet celkového počtu obyvatel tedy stačí sečíst násobky počtu budov, které ubytovávají obyvatele, s jejich kapacitou.

Budovy produkující suroviny však už tento parametr využívají. Pro přiřazení obyvatele do budovy je třeba, aby měl správný nástroj, v budově byla volná kapacita (když je nedostatečná, je třeba postavit více budov daného typu) a v případě, že je výroba suroviny závislá na nějaké jiné, je třeba dostatečná produkce této závislé suroviny.

Díky počtu přiřazených pracovníků v budovách a celkovému počtu obyvatel lze snadno dopočítat dostupný počet obyvatel, kteří se dají přiřadit do budov nebo rekrutovat do armády.

Tabulka 5.3: Kolekce budov

Sloupec	Datový typ
player	String
type	String
count	Number
assigned	Number

Kolekce obchodních nabídek

Každá obchodní nabídka se skládá z nabízené a požadované suroviny. U těchto surovin je třeba znát jejich typ (např. dřevo, kamení, luk, sekera) a počet. Je třeba uvést, že obchodovat lze pouze se surovinami, nikoliv s budovami aj.

Každá obchodní nabídka může existovat ve 2 stavech: vytvořená a akceptovaná. Když hráč nabídku vytvoří, musí specifikovat nabízenou a požadovanou surovinu. Hra do nabídky automaticky doplní také jednoznačnou identifikaci nabízejícího uživatele (ID) a časové razítko, kdy byla nabídka vytvořena. Takto vytvořená nabídka se může zobrazit v seznamu pro ostatní hráče, kteří z nich mohou vybírat. Nejprve je však provedena kontrola, zda-li nabízející hráč disponuje dostatečným počtem nabízené suroviny.

Jiný hráč může takto vytvořenou nabídku akceptovat, kde nejprve proběhne kontrola, zda-li akceptující hráč má dostatečný počet surovin na provedení transakce a potom je do záznamu v kolekci doplněno ID tohoto uživatele a časové razítko provedení obchodu.

Tabulka 5.4: Kolekce obchodních nabídek

Sloupec	Datový typ
offeringResourceType	String
offeringResourceCount	Number
offeringPlayer	String
seekingResourceType	String
seekingResourceCount	Number
seekingPlayer	String
tradeCreated	Date
tradeFinished	Date

Kolekce rekrutovaných vojáků

Poslední kolekcí je kolekce, která funguje obdobně, jako kolekce se surovinami. Místo surovin je zde však uložen počet rekrutovaných vojáků. Jelikož narozdíl od surovin není možné vojáky rekrutovat automaticky, je třeba informaci o jejich typu a počtu držet v separátní kolekci. Určitě by také postačilo je ukládat do kolekce surovin a parametry s časovými razítky nevyužívat, každopádně pro toto řešení jsem se rozhodl také pro odlehčení surovinové kolekce, která už takto bude udržovat většinu informací.

Tabulka 5.5: Kolekce rekrutovaných vojáků

Sloupec	Datový typ
player	String
type	String
recruited	Number

5.4 Definice statických herních prvků

Kromě dynamických parametrů, které jsou uloženy v databázi, jsou všechny herní prvky definovány také dodatečnými statickými parametry. Aby byla hra do budoucna rozšiřitelná, snažil jsem se, aby byly tyto definice co nejvíce obecné. Pro lepší uživatelskou přívětivost jsou objekty ve hře rozděleny kromě základních kategorií do pokročilých kategorií. Tyto pokročilé kategorie jsou postaveny tak, aby vždy každý herní objekt mohl v základu využít dynamický parametr uložený v databázi.

- Suroviny - nadkategorie zahrnující objekty, které využívají kolekci surovin v databázi
  - Sběrné suroviny - surovina se dá získat bez jiných surovin (např. surové dřevo)
  - Vyrobitelné suroviny - k vyrobení suroviny jsou třeba jiné suroviny (např. prkno, sekera, luk)
    - Nástroje - nutné k přiřazení pracovníků do budov (např. sekera, luk)
      - Zbraně - nutné k rekrutování armády (např. luk)
  - Produkce - počet surovin, které se vyrobí za jednotku času (např. dřevo za sekundu)
- Budovy - objekty, které využívají kolekci budov
- Armáda - objekty, které využívají kolekci rekrutovaných vojáků

Každá kategorie obsahuje určité objekty, které jsou pevně zadefinované. Při prvním přihlášení hráče do hry jsou tyto objekty uloženy s nulovými počty do databáze a tím pádem se jedná o počáteční stav hry. Po hráčově interakci se vždy mění jen dynamické parametry v databázi.

Statické parametry se mění pouze v případě aktualizace hry, kdy by bylo potenciálně potřeba do hry přidat nové objekty. V tomto případě je třeba všem hráčům dogenerovat do databáze dané objekty s nulovým počtem. Tímto je zajištěna rozšiřitelnost hry do budoucna.

Jak je vidět z hierarchie, hra obsahuje 3 základní nadkategorie, každá využívající určitou kolekci v databázi. Tyto nadkategorie musí být dostatečně obecné, aby obsahovaly společné parametry, které se vyskytují i v podkategoriích. Každá podkategorie pak může obsahovat nějaké dodatečné parametry, které se v kategorii v hierarchii nad ní nevyskytují. Kromě tohoto dělení se také tyto kategorie dají využít pro rozdělení hry do logických celků pro hráče.

5.5 Uživatelské rozhraní

Celý frontend hry je navržen tak, aby se jednalo o tzv. single-page aplikaci. Při prvním načtení stránky (ať už je uživatel kdekoliv) se stránka sestaví na serveru a odešle klientovi. V tomto případě se jedná o SSR (server-side rendering). Když uživatel interaguje se stránkou (ať už samotným hraním, nebo přecházením na jiné obrazovky/podstránky v rámci aplikace), vše se děje pouze na klientovi. Tento přístup má 2 výhody: oproti čistému SSR se šetří výkon minimálním možném volání serveru, ale zároveň je zde zachován SSR pro první načtení stránky pro umožnění SEO optimalizace pro vyhledávače. Při prvním načtení by se totiž mohla poslat jen základní struktura stránky a i v tomto případě by se vyrenderovala stránka pomocí klientského javascriptu. To by ale mělo nevýhodu, že vyhledávače by viděly pouze tuto základní strukturu.

Samotná aplikace je rozdělena na několik logicky oddělených celků:

Úvodní stránka

Jediná stránka, na kterou se uživatel dostane nepřihlášen. Zároveň je na tuto stránku přesměrován z kterékoliv jiné stránky, pokud není přihlášen. Stránka slouží jako představení hry a umožňuje uživateli vytvoření účtu, případně přihlášení v případě, že účet již má, nebo vytvoření zkušebního účtu, který byl popsán v sekci 5.3. V případě, že je uživatel přihlášen, je z této stránky automaticky přesměrován na stránku **Sběr surovin**.

V počáteční verzi hra umožňuje pouze přihlášení pomocí e-mailové adresy a hesla, do budoucna je však v plánu přihlášení pomocí dalších metod pro urychlení registrace (např. pomocí Google nebo Facebook účtu). Po registraci je také uživateli odeslán potvrzovací e-mail, ve kterém je odkaz, kde po jeho kliknutí je uživateli potvrzen účet a může začít s hraním. V plánu je také možnost resetování hesla v případě, že ho uživatel zapomněl.

Získávání základních surovin

První stránka, na které se uživatel objeví po přihlášení. Při rozehrání hry jediná možnost, jak začít získávat suroviny. Na stránce jsou zobrazeny všechny základní suroviny, které může uživatel sbírat. Suroviny jsou zobrazeny pomocí karet, kde u každé karty je obrázek dané suroviny a tlačítko, které umožňuje surovinu získat. Oproti ostatním herním objektům je získávání základních surovin omezeno časově.

Výroba pokročilých surovin

Vyrábění surovin na této stránce je podmíněno získáním základních surovin na předchozí stránce. Zobrazení je opět provedeno pomocí karet, každopádně u každé pokročilé suroviny je navíc uvedena informace o potřebných vstupních surovinách, které jsou potřeba k jejich výrobě. Narozdíl od předchozí stránky se suroviny vyrábějí bez časového limitu. Pro uživatelskou přívětivost je také možné zvolit množství surovin, které se má vyrobit. Na výběr je buď 1, 10, 100 nebo maximální možné množství. Když uživatel nemá dostatek vstupních surovin, nelze pokročilou surovinu vyrobit.

Stavba budov

Tato obrazovka slouží ke správě a stavbě budov. Funguje úplně stejně jako předchozí obrazovka, oddělení je tedy čistě logické. Obsahuje budovy pro ubytování obyvatel (po postavení budovy jsou hráči přičteni noví obyvatelé) a budovy produkující suroviny. Na této obrazovce se však budovy dají jen postavit, pro přiřazení obyvatel do budov slouží další obrazovka.

Obyvatelé

Na této obrazovce se hráč dozví, kolik má celkem, dostupných a přiřazených obyvatel do budov. Kromě tohoto jsou zde zobrazeny budovy, do kterých lze obyvatele přiřazovat (nejsou tu tedy budovy pro ubytování obyvatel a kasárna. Nachází se tu i obchod, do kterého se dají přiřazovat obyvatelé, ze kterých se stávají obchodníci, což umožňuje hráči obchodovat s ostatními. Před přiřazením obyvatele do budov je třeba mít dostatečný počet budov a surovin, když jsou pro danou produkci potřeba, ale také dostatečný počet nástrojů, kterými jsou obyvatelé vybaveni pro práci v dané budově.

Obyvatelé jdou z budov také odebírat, avšak pouze tehdy, když nejsou přiřazeni v budově, která vyrábí surovinu, na které je závislá produkce jiné suroviny. Nemůže se tedy stát, že by byla produkce jakékoliv suroviny záporná.

Obchod

Po postavení obchodu a přiřazení obchodníků v předchozí obrazovce je hráči umožněno obchodovat s ostatními na této obrazovce. Obrazovka je rozdělena na 2 záložky: vytváření vlastních nabídek a seznam všech nabídek. Na záložce vytváření nabídek je hráči umožněno zadat typ a množství nabízených surovin a typ a množství poptávaných surovin. Po jejím vytvoření se nabídka zobrazí v seznamu nabídek, kde ji vidí jak hráč, který ji vytvořil (v tomto případě ji může zrušit), ale také ostatní hráči, kteří ji mohou přijmout a provést tím obchod.

Kasárna

Poslední obrazovkou jsou kasárny. Zde narozdíl od obchodu jsou obyvatelé přiřazováni přímo, stejně jako v obrazovce Obyvatelé. Bylo by zbytečné, aby hráč musel obyvatele přiřazovat dvakrát: nejprve v obrazovce Obyvatelé a pak zde. Další podobnost s obrazovkou Obyvetelé je ta, že pro úspěšnou rekrutaci armády je třeba mít dostatečné množství zbraní, které jsou danému obyvateli/vojákovi přiřazeny.

Kapitola 6

Návrh umělé inteligence

Umělá inteligence bude definována pomocí stavového automatu, pomocí jehož pravidel bude hru hrát. Genetický algoritmus slouží čistě jen k vygenerování takového automatu, který bude hrát hru co nejlépe, jak je to možné. V této kapitole bude popsán návrh stavového automatu a pak samotného genetického algoritmu. Pro správnou funkci genetického algoritmu je potřeba simulátor hry, který zde bude také popsán spolu s odůvodněním jeho existence.

6.1 Motivace k využití umělé inteligence

K využití agentů, kteří budou hru hrát automaticky, jsem se rozhodl ze dvou hlavních důvodů. Prvním a hlavním důvodem je vyvážení hry. Ekonomika hry musí být navržena tak, aby hráč hrou zpočátku postupoval relativně rychle a aby se ve hře stále něco dělo. Tato fáze hry musí být navržena tak, aby hráče hra začala bavit. Když už bude hrát déle, očekává se, že ho hra baví a je třeba zvolnit tempo, aby se stále odemykaly nové možnosti hry, ale tak pomalu, aby hráč u hry vydržel co nejdéle.

Jednotlivé parametry hry chci tedy nastavit podle agentů, kteří hru hrají optimálním způsobem a sledovat jak dlouho hra trvá. Podle délky hry se pak dá s parametry hýbat a dále sledovat, jak se délka hry změnila. Tento způsob je mnohem rychlejší, než pro každou skupinu parametrů nechávat hru hrát reálné lidi. Místo reálné hry můžu agenty nechávat hrát simulátor a sledovat počet odehraných tahů (sekund).

Druhou motivací je pak nedostatek reálných hráčů při spuštění hry. Očekávám, že zpočátku budou hru hrát jednotky nebo desítky lidí. Aby jim hra nepřišla prázdná, budou s nimi hrát i automatičtí agenti, kteří se budou tvářit jako reální hráči. Když hra dosáhne určitého počtu hráčů, agenti nejspíš už nebudou potřeba. Pak už se budou využívat jen k vyvážení hry.

6.2 Stavový automat

Jediná část hry, kde hráč musí čekat, je obrazovka sběru surovin. V ostatních částech hry se všechny akce dají provést v jakýkoliv čas, kdy je potřeba, bez čekání. Pro zjednodušení umělé inteligence a následného návrhu simulátoru jsem čas navrhl tak, že každá akce v rámci umělé inteligence bude probíhat právě jednou za sekundu. Pro sběr surovin zde nenastává žádný problém, avšak pro všechny ostatní části hry je umělá inteligence znevýhodněna tím, že může vykonávat oproti hráči akce právě jednou za sekundu. Jelikož je ale tento interval

dostatečně krátký, nebude mít na úspěšnost umělé inteligence velký vliv. Hlavní výhodou robota bude hlavně to, že hru může hrát kontinuálně bez přestávek.

Při návrhu architektury automatu jsem hledal optimální průnik mezi jeho jednoduchostí kvůli omezení možného stavového prostoru pro genetický algoritmus, ale zároveň dostatečnou flexibilitou pro dobré vyjadřovací schopnosti. Automat se skládá z jednotlivých stavů, kterých může být teoreticky neomezené množství. Každý stav pak obsahuje dvojici podmínka a akce. Pro každý krok simulátoru se prochází jednotlivé stavy automatu sestupně. Když je podmínka stavu splněna, vykoná se příslušná akce a procházení automatu končí. V jeden krok je tedy možné vykonat vždy maximálně jednu akci. Když podmínka stavu splněna není, pokračuje se na další stav automatu, dokud buď aspoň jedna podmínka není splněna, nebo v automatu už neexistuje více podmínek. Může se tedy stát, že automat v rámci simulace neodehraje nic, jelikož celou hru obsahuje nesplnitelné podmínky.

Při návrhu architektury automatu jsem hledal optimální průnik mezi jeho jednoduchostí kvůli omezení možného stavového prostoru pro genetický algoritmus, ale zároveň dostatečnou flexibilitou pro dobré vyjadřovací schopnosti. Automat se skládá z jednotlivých stavů, kterých může být teoreticky neomezené množství. Každý stav pak pak obsahuje dvojici podmínka a akce. Pro každý krok simulátoru se prochází jednotlivé stavy automatu sestupně. Když je podmínka stavu splněna, vykoná se příslušná akce a procházení automatu končí. V jeden krok je tedy možné vykonat vždy maximálně jednu akci. Když podmínka stavu splněna není, pokračuje se na další stav automatu, dokud buď aspoň jedna podmínka není splněna, nebo v automatu už neexistuje více podmínek. Může se tedy stát, že automat v rámci simulace neodehraje nic, jelikož celou hru obsahuje nesplnitelné podmínky.

6.3 Simulátor hry

Pro testování úspěšnosti stavových automatů v rozumném čase je potřeba simulátor. Pro ohodnocení automatu však nestačí funkce, která by každý automat ohodnotila a vypočítala skóre. Jelikož se jedná o hru více hráčů, očekává se mezi hráči interakce. Pro jednoduchost je simulátor navržen tak, že pro každý tah se nechají odehrát všichni agenti (stavové automaty) postupně. Pořadí je pro každý tah vždy stejné. Simulátor také řeší interakci mezi agenty, kde akce může vyžadovat např. obchodování nebo zaútočení na jiného agenta.

Důležitým požadavkem je, aby simulátor byl co se týče parametrů shodný se hrou. Jelikož parametry jsou definovány v konfiguračním souboru, simulátor by měl umět číst tento soubor a simulovat hru s těmito parametry. Samotný simulátor je inicializován s počátečními parametry nezávislými na hře jako např. počet iterací.

Výstupem simulátoru, jak již bylo uvedeno, je skóre každého jedince po odehrání stanoveného počtu tahů. Jelikož je simulátor využíván genetickým algoritmem, je třeba skóre navrhnout tak, aby agoritmus přicházel postupně na čím dál lepší řešení. Tohoto jsem dosáhl tak, že základní akce v rámci hry, sbírání základních surovin, je odměněno nejnižším skóre a pokročilejší akce skóre vyšším. Na sbírání surovin přirozeně navazuje výroba pokročilých surovin, dále stavba budov a nakonec přiřazování obyvatel do budov.

Celý systém výpočtu stavových automatů je navržen tak, aby byl simulátor dynamickým prvkem. Při potřebě automatů řešící/automatizující jiné problémy stačí jen naprogramovat simulátor řešící daný problém a napojit ho na genetický algoritmus, který ho jen spouští při každé generaci. Systém může být použit nejen pro generování automatů do jiných her, ale i do aplikací, jejichž běh se dá automatizovat.

6.4 Genetický algoritmus

Hlavním prvkem celé umělé inteligence je genetický algoritmus, který je schopen generovat stavové automaty. Jeho funkcí je nalézt takový automat, který dosáhne v simulátoru nejvyššího skóre v rámci populace. Základním rozhodnutím je návrh chromozomu, který by měl ideálně reprezentovat stavový automat. Rozhodl jsem se pro kódování pomocí celých čísel, kde chromozom se skládá z dvojic podmínka a akce. Číslo reprezentuje, o jakou podmínku nebo akci se jedná. Tato reprezentace má výhodu v tom, že chromozom je do budoucna rozšiřitelný při požadavku o přidání dodatečných podmínek nebo akcí.

Algoritmus by měl mít základní nastavitelné parametry:

1. maximální počet generací pro ukončení algoritmu
2. velikost populace, aneb počet jedinců v rámci populace
3. délka chromozomu, neboli počet stavů automatu (délka musí být vždy dělitelná dvěmi, každý stav musí obsahovat podmínku a akci)
4. pravděpodobnosti křížení, mutace a dalších jevů ovlivněných pravděpodobností

Pro výběr jedinců do další generace jsem se rozhodl pro výběr pomocí turnaje. Jelikož chci v každé generaci zachovat počet jedinců, proběhne výber následovně: jedinec z nové populace je vybrán z předchozí populace tak, že se vybere jako nejlepší z N náhodně vybraných.

Pro operátor křížení jsem se rozhodl pro vícebodové (konkrétně dvoubodové) křížení z důvodu větší variability možností vytvořených potomků. Zde ale nastává problém kvůli kódování chromozomu, kde mezi sebou nechci křížit jednotlivé podmínky a akce, ale celé stavy. Stav je v tomto případě tedy nedělitelný blok.

Co se týče mutací, potřebuji využít dvě mutace. První mutací je náhodná změna podmínky nebo akce tak, aby byl automat stále validní. Nesmí se tedy vygenerovat neexistující číslo podmínky nebo akce. Druhou mutací je náhodná záměna stavů. Stejně jako u křížení je při záměně třeba zachovat stavy. Cílem této mutace je vyzkoušení možnosti, zdali záměnou stavů není možné přijít na shodné nebo lepší řešení. Když je rozhodnuto, že se provede mutace, musí se algoritmus s další pravděpodobností rozhodnout, kterou z těchto dvou mutací vybere. Tato pravděpodobnost by měla být také jednou z inicializačních parametrů.

Při inicializaci algoritmu je třeba vytvořit počáteční populaci. Ta musí obsahovat validní jedince (obsahují existující podmínky a akce). Neplánuji využít žádné aproximace, která by mohla pomoci s rychlejším nalezením výsledku.

V každé generaci je také třeba ohodnotit všechny jedince pomocí fitness funkce. K tomuto účelu slouží dříve zmíněný automat. Automat se nainicializuje s jedinci, kteří se vyskytují v aktuální generaci a spustí se (automat je tedy nutné spustit pro každou generaci).

Kapitola 7

Implementace webové hry

V této kapitole budou popsány konkrétní algoritmy a technologie, které byly použity k implementaci webové hry. Dále budou popsána řešení různých problémů, které jsem v rámci implementace řešil.

7.1 Databáze

Pro definici databázových modelů jsem se rozhodl pro knihovnu **mongoose**, která umožňuje nejen jednoduše definovat datové typy jednotlivých položek daného dokumentu databáze, ale také napojení na MongoDB. Ve výsledku se dají vytvářet javascriptové objekty, pomocí kterých se instancuje daný model (třída), nad kterým jsou definovány metody, které se dají volat. Pro každou třídu lze definovat statické metody, které se dají volat, aniž by bylo třeba třídu instancovat, a metody třídní, k jejichž volání je sice třeba model instancovat, ale lze z nich přistupovat k parametrům daného konkrétního modelu. Jediný model, u kterého jsem potřeboval rozšířit stávající metody, je model uživatelů. Zde jsou navíc implementovány metody na získání uživatelského profilu bez hesla a zašifrování hesla pomocí metody bcrypt.

Po registraci hráče se totiž nejprve musí zajistit, aby jeho heslo nebylo uloženo v databázi v čitelné formě. Potom se vytvoří jeho záznam v tabulce uživatelů a zároveň je pro něj inicializována hra. V každé tabulce, která drží stav hry, jsou vytvořeny nové záznamy pro každou surovinu a budovu, která existuje ve hře. Podle konfiguračního souboru jsou nastaveny dodatečné parametry, které se liší od těch původních.

Aby mohl hráč automatizovat výrobu surovin, musí přiřadit obyvatele do budovy. Aby je tam mohl přiřadit, musí je mít k dispozici. Aby je mohl mít k dispozici, musí je mít kde ubytovat. Aby je mohl ubytovat, musí mít obyvatelé co jíst. Jelikož každá budova stojí kromě stavebních materiálů také jídlo za sekundu, které se dá vyrábět pouze v budově, tak by hráč nemohl nikdy nic automatizovat, kdyby nějaké obyvatele od začátku nedostal. Zároveň z těchto počátečně dostupných obyvatel jsou někteří z nich automaticky přiřazeni k budově, které generuje jídlo tak, aby byla výroba jídla taková, aby přesně pokryla spotřebu jídla.

7.2 GraphQL server

K implementaci GraphQL serveru používám knihovnu **apollo-server-express**, která umožňuje jak jednoduchou inicializaci GraphQL datových typů a mutací, ale také odebírání (implementace funkce subscribe, která interně využívá webové sockety) pro aktualizace stavu

ostatním hráčům v případě jeho změny u kterékoholiv jiného hráče (např. vytvoření obchodní nabídky: přidání dané nabídky do seznamu nabídek ostatním hráčům).

Pro inicializaci GraphQL serveru je třeba definovat datové typy (types), mutace (mutations) a odebírání (subscriptions).

Datové typy

Na nejvyšší úrovni mám definovaných pět hlavních typů:

```
type Query {
  players: [User]
  resources: [Resource]
  buildings: [Building]
  offers: [Offer]
  army: [Army]
}
```

Obrázek 7.1: Ukázka pěti hlavních GraphQL typů

- **User** - pro získávání informací o hráčích (např. jméno hráče v tabulce obchodních nabídek)
- **Resource** - pro zobrazení a úpravu základních a pokročilých surovin
- **Building** - pro zobrazení a úpravu budov a obyvatel, kteří jsou do budov přiřazeni
- **Offer** - pro vytváření a zobrazení obchodních nabídek
- **Army** - pro vytváření a zobrazení armády

Všechny datové typy odpovídají databázovým modelům až na typ User, kde nejsou z důvodu bezpečnosti zahrnuty položky jako e-mail a heslo.

Mutace

Pro každý datový typ musí být definovány operace, které umožňují hraní hry. Z hlediska bezpečnosti je třeba kromě validace vstupů na frontendu provádět také validace na backendu. Tyto validace mám implementovány právě v GraphQL mutacích. Než popíšu princip mutací, nejprve uvedu příklad mutace, které zajišťují změnu surovin (základních a pokročilých).

```
type Mutation {
  harvest (
    type: String!
  ): Resource
  construct (
    type: String!,
    quantity: Int!
  ): ConstructPayload
}

type ConstructPayload {
  resource: Resource
  inputs: [Resource]
}
```

Obrázek 7.2: Ukázka mutace

Každá mutace musí mít v definici definován vstup a výstup. Jelikož v tomto případě mutace pro výrobu pokročilých surovin upravuje stavy několika surovin (vstupních a výstupních), je třeba mít pro výstup speciální datový typ **ConstructPayload**.

Základní suroviny mají nejjednodušší mutaci. Jediná interakce, která se od hráče vyžaduje, je jejich sebrání. Jediným parametrem, který je od frontendu potřeba, je typ sbírané suroviny. Jak bude popsáno dále, sběr základních surovin je omezen časově, což je validováno na frontendu. Toto časové omezení chci však ověřovat i na backendu. Do databáze ukládám časové razítko, které je aktualizováno při každé změně dané hodnoty. Když přijde nový požadavek o změnu, nejprve se podívám, zdali je rozdíl aktuálního času a razítka uloženého v databázi větší než doba, kterou má trvat sebrání dané suroviny. Z důvodu nespolehlivého přenosu paketů přes síť internet je třeba počítat s tím, že se pakety mohou zdržet, a dorazit tak, že mezi nimi není dodržen interval aspoň jedné vteřiny. Zde toleruji interval kratší o 0.1 sekundy (sběr surovin trvající 1 sekundu může trvat 0.9 sekundy, sběr trvající 5 sekund může trvat 4.9 sekundy, apod.).

Konstrukce pokročilejších surovin už vyžaduje kromě typu vyráběné suroviny také parametr určující, kolik surovin se má vyrobit. V aktuální verzi hry je na výběr ze čtyř možností: 1, 10, 100 a maximální možný počet. Validace probíhá tak, že se musí zjistit, zdali má hráč dostatečný počet vstupních surovin v rámci daného požadovaného výstupního množství. V případě maximálního možného množství se pouze validuje, zdali může hráč vyrobit aspoň jednu surovinu. Kromě validace se u maximálního možného množství také musí vypočítat aktuální množství surovin, které hráč dostane.

U budov jsou definovány tři mutace: stavba, přiřazení a odebrání obyvatele do/z budovy. Stavba budov funguje stejně jako konstrukce pokročilých surovin s tím rozdílem, že vstupy jsou suroviny, ale výstupy jsou budovy. Parametry jsou také stejné jako v předchozím případě.

Přiřazování a odebírání obyvatel z budov je potřeba validovat pomocí jiných pravidel. Nejprve je třeba vůbec zjistit, zdali má hráč dostatečný počet nepřiřazených obyvatel, pak zdali je pro daného obyvatele vyroben dostatečný počet nástrojů, se kterými bude v budově pracovat. Jelikož jsou budovy rozděleny na ty, do kterých se dají obyvatelé přiřadit a na ty, které obyvatele ubytovávají, je také třeba kontrolovat, zdali se hráč nesnaží přiřadit obyvatele do budovy, do které to nejde. Co se týče odebírání, tak nelze z budovy odebrat

více obyvatel, než tam bylo přiřazeno (např. když přiřadím do budovy 6 obyvatel a pokusím se odebrat 10) a také nelze způsobit, aby byla výsledná produkce záporná (např. budova, která produkuje jídlo a díky které mohu stavět budovy pro ubytování obyvatel, musí mít přiřazených tolik obyvatel, aby byla produkce jídla aspoň nulová). Parametry těchto dvou akcí jsou opět stejné: je třeba znát typ budovy, kam chci obyvatele přiřadit/odebrat a počet obyvatel.

Jedním s hlavních důvodů implementace automatizované produkce je také zajištění výroby surovin, když hráč zrovna není přihlášen do hry. Když se hráč přihlásí do hry po nějaké době, je potřeba mu dát tolik surovin, kolik mu za danou dobu vyprodukovali obyvatelé přiřazení do budov. K tomu slouží další časová známka, která se opět aktualizuje po změně stavu suroviny, ale pouze když změna proběhne pomocí budovy surovinu produkující.

Další mutace se týkají obchodu. Z pohledu hráče vytvářejícího nabídku jsou zde dvě akce: vytvoření a smazání nabídky. Z pohledu hráče, který si nabídky prohlíží a chce, aby proběhl obchod, je zde akce přijmutí nabídky. Pro vytvoření nabídky je potřeba znát 4 parametry: typy a počty surovin, které hráč nabízí a požaduje. V rámci jedné obchodní nabídky lze tedy nabízet a požadovat vždy jen jednu surovinu. Pro přijetí či odebrání stačí znát pouze unikátní ID vytvořené nabídky.

V rámci validace je třeba při vytváření obchodní nabídky ověřit, zdali má nabízející hráč dostatek surovin a při akceptování nabídky zdali má příjemce dostatek surovin požadovaných nabízejícím hráčem.

Poslední skupinou mutací jsou mutace upravující stav armády. Zde jsou ekvivalenty přiřazení/odebrání do/z budovy = verbování a rozpuštení armády. Parametry jsou opět typ vojáka a jejich počet. Dále je zde mutace, která řeší útočení. Nejprve popíšu validaci. Jediným parametrem pro mutaci útočení je unikátní ID hráče, na kterého je třeba zaútočit. Je tedy třeba zajistit, že hráč nemůže útočit sám na sebe.

Logika výpočtu ztrát při souboji funguje tak, že se nejprve zjistí síla armád útočníka a obránce. Síla útočníka se spočítá tak, že se projdou všechny druhy vojáků, a u každého druhu se vezme jeho úročná síla a vynásobí se jejich počtem. Tyto dílčí síly se pak sečtou. U obránce je pak spočítá obraná síla, která se bere místo z parametru útočné síly z obranné síly. Po tomto výpočtu mohou nastat tři možnosti: síla armády útočníka je větší, než síla armády obránce, síly jsou vyrovnané nebo obránce je silnější než útočník.

Když je útočník silnější, obránce automaticky přichází o veškerou armádu. Síla obránce je však použita k výpočtu ztrát útočníka. Postupně je iterováno jednotlivými druhy vojáků, kde se od síly obránce postupně odečítají jednotlivé útoky útočníka. Když je síla obránce vyšší, než útočná síla aktuálního iterovaného vojáka, útočník o daného vojáka přichází.

Při vyrovnaných silách útočníka a obránce přichází o celou armádu útočník a obránci se pak vybere náhodný přeživší. Technicky tedy přijdou o celou armádu oba, ale obránci se přičte náhodný voják na závěr souboje. Je důležité zmínit, že obránce nemůže získat žádného vojáka, kterého by před soubojem neměl. Náhodný přeživší se vybírá pouze z vojáků, kteří mají před soubojem nenulový počet.

Poslední možností je silnější obránce. Zde přichází opět o celou armádu útočník a ztráty obránce se počítají obdobně jako v prvním případě ztráty útočníka, ale od celkové síly útočníka se odečítají jednotlivé síly obranných sil vojáků obránce.

Odebírání

Koncept odebírání je implementován pomocí webových socketů, kde cílem je v reálném čase komunikovat jak od klienta k serveru, tak od serveru ke klientovi. Všichni klienti odebírají

potenciální změny stavu, které jsou vytvářeny jinými klienty. Tímto lze zajistit, že když některý hráč vyvolá akci ve hře, která má ovlivnit někoho jiného, ovlivněný hráč se o akci dozví v reálném čase.

Tento mechanismus používám u obchodu a válčení. Mám zadefinované celkem 4 typy odebírání: obchod byl vytvořen, obchod byl smazán, obchod byl přijat a bylo zaútočeno na hráče. Při vyvolání kterékoliv z těchto akcí se rozešle všem hráčům aktualizace stavu.

Pro funkci tohoto mechanismu v GraphQL používám návrh zasílání zpráv PubSub (Publish - Subscribe), který je implementován v několika knihovnách. Pro stávající vývojové účely používám aktuálně knihovnu pubsub, avšak do produkčního řešení ji plánuji nahradit jinou knihovnou.

V rámci GraphQL resolverů je třeba nastavit jednotlivé funkce pro odebírání:

```
Subscription: {
  offerCreated: {
    subscribe: () => pubsub.asyncIterator(['offerCreated'])
  },
  ...
  playerAttacked: {
    subscribe: () => pubsub.asyncIterator(['playerAttacked'])
  }
}
```

Obrázek 7.3: Ukázka odebírání

GraphQL se pak po vygenerování akce (publish) postará, aby byla daná zpráva rozeslána všem příjemcům, kteří jsou přihlášeni k jejich odběru (subscribe). V rámci samotných mutací je ale třeba nejprve zprávy vygenerovat.

```
pubsub.publish('offerAccepted', {
  offerAccepted: acceptedOffer
})
```

Obrázek 7.4: Ukázka rozeslání aktualizace příjemcům

Jak je vidět na příkladu, kromě názvu může zpráva také obsahovat data, která jsou všem příjemcům rozeslána. Toto je ideální pro aktualizace stavů příjemců. V příkladu je v proměnné acceptedOffer uložena obchodní nabídka, která obsahuje veškeré informace důležité k aktualizaci stavu surovin hráče, který danou nabídku vytvořil.

7.3 GraphQL klient

Aby mohla celá aplikace fungovat, je kromě GraphQL serveru třeba implementace GraphQL klienta, která běží na prohlížečích jednotlivých klientů. Touto implementací je knihovna Apollo, která kromě jednoduchého rozhraní pro komunikaci s GraphQL serverem obsahuje implementaci lokálního stavu, kde se pak drží celý stav hry.

Pro každou herní obrazovku potřebuji specifikovat GraphQL dotazy (query), mutace, a u obrazovek, kde se nachází interakce s ostatními hráči, také odebírání.

Získávání základních surovin

U obrazovky sbírání surovin potřebuji na začátku pouze načíst aktuální stav surovin:

```
{
  resources {
    id
    player
    type
    count
  }
}
```

Obrázek 7.5: Ukázka GraphQL definice datových typů pro obrazovku získávání základních surovin

Unikátní ID suroviny potřebuji proto, že Apollo při každé mutaci automaticky aktualizuje stav dané suroviny v lokální paměti. Typ suroviny potřebuji k zobrazení popisku hráči, aby věděl, o jakou surovinu se jedná. Stejně tak počet surovin, který se díky mutacím aktualizuje. Jelikož se suroviny ukládají do globálního úložiště a jsou potenciálně využívaány i na jiných obrazovkách (např. obchod), zjišťuji si zároveň jméno hráče pro potenciální identifikaci v rámci obchodních nabídek.

Když hráč sebere surovinu, je zavolána funkce, která provede mutaci. Výstupem mutace je aktualizovaný stav sebrané základní suroviny.

```
mutation harvest($type: String!) {
  harvest(type: $type) {
    id
    count
  }
}
```

Obrázek 7.6: Ukázka GraphQL mutace pro sebrání základní suroviny

Jak lze vidět z příkladu, parametrem mutace je typ suroviny a výstupem je aktualizovaná surovina, ze které však vyžaduji pouze její unikátní ID a počet pouze k účelu aktualizace.

Vyrábění pokročilých surovin

Stejně jako u získávání základních surovin, je zde potřeba nejprve načíst pokročiké suroviny. Jelikož jsou základní a pokročiké suroviny uloženy ve stejné tabulce v databázi, při prvním načtení stránky se v obou případech načtou stejné suroviny. GraphQL dotaz je tedy totožný s předchozím případem.

Změnou je však mutace. Díky možnosti vyrobit více surovin najednou je zde navíc parametr quantity, který určuje vyráběné množství. Výstupem mutace jsou aktualizované suroviny, které mám rozdělené na vstupní suroviny a výstupní surovinu.

```
mutation construct($type: String!, $quantity: Int!) {
  construct(type: $type, quantity: $quantity) {
    resource {
      id
      count
    }
    inputs {
      id
      count
    }
  }
}
```

Obrázek 7.7: Ukázka GraphQL mutace pro výrobu pokročilých surovin

Toto rozdělení je čistě kvůli jednodušší implementaci na straně backendu. Vstupní suroviny i výstupní surovina jsou opět uloženy ve stejné tabulce.

Stavba budov

Narozdíl od surovin jsou budovy uloženy v jiné tabulce, tudíž je třeba je pomocí lehce upraveného GraphQL dotazu získat spolu se surovinami. Z budov potřebuji opět znát všechny parametry včetně počtu přiřazených obyvatel pro účely obrazovky obyvatel.

Mutace stavění budov aktualizuje stav vstupních surovin a stav budov, které jsou výstupem. Parametry jsou stejné jako v předchozím případě.

```
mutation build($type: String!, $quantity: Int!) {
  build(type: $type, quantity: $quantity) {
    building {
      id
      count
    }
    resources {
      id
      count
    }
  }
}
```

Obrázek 7.8: Ukázka GraphQL mutace pro stavbu budov

Obyvatelé

Jelikož tato obrazovka zobrazuje celkový počet obyvatel a kolik z nich je dostupných, je třeba kromě budov a surovin také načíst stav armády. Když je obyvatel rekrutován, nelze ho využít na přiřazování do budov.

Dále jsou zde zbývající dvě mutace, které slouží k přiřazování do a odebírání obyvatel z budov.

```
mutation assign($type: String!, $quantity: Int!) {
  assign(type: $type, quantity: $quantity) {
    building {
      id
      assigned
    }
    resources {
      id
      count
    }
  }
}
```

Obrázek 7.9: Ukázka GraphQL mutace pro přiřazení obyvatel do budov

Jako příklad uvádím mutaci přiřazování, každopádně mutace odebírání je totožná až na jiný název metody, kterou volá. Obě mutace mají na vstupu typ budovy, ve které je třeba změnit stav obyvatel a počet přiřazených obyvatel. Výstupem jsou opět změněné vstupní suroviny a výstupní budova, u které se ale narozdíl od stavby mění parametr přiřazených obyvatel.

Obchod

Na této obrazovce se kromě obchodních nabídek zobrazuje počet dostupných obchodníků. Z databáze je tedy nutné získat jak aktuální stav surovin, se kterými pak hráč bude obchodovat, tak budovy, protože je potřeba znát počet přiřazených obyvatel do obchodu, ale i samotné obchodní nabídky, které se pak zobrazí v seznamu.

Jak bylo zmíněno v předešlé části GraphQL server, tato obrazovka musí mít nastavenu funkcionalitu odebírání. Kromě mutací, které se starají o přidávání, odebírání a přijímání obchodních nabídek, jsou zde také metody pro odebírání. Musí zde být ošetřeny případy, že jiný hráč např. přijal moji nabídku a mě by se tedy měl v reálném čase aktualizovat stav surovin.

```
subscription offerCreated {
  offerCreated {
    id
    offeringResourceType
    offeringResourceCount
    offeringPlayer
    seekingResourceType
    seekingResourceCount
    tradeCreated
  }
}
```

Obrázek 7.10: Ukázka GraphQL odebírání pro vytvoření obchodní nabídky

Jak lze vidět z příkladu, odebírání funguje velice obdobně jako mutace, avšak jediné, co je třeba nadefinovat, je výstup.

Jelikož se aktualizace stavu rozesílají všem hráčům, kteří jsou aktuálně na této obrazovce, je potřeba ošetřit, aby se suroviny z obchodu měnily jen hráčům, kteří jsou účastníky obchodu. Když hráč přijme obchodní nabídku, tak vytvářejícímu hráči se musí odečíst nabízené suroviny a přičíst požadované suroviny. Všem hráčům pak nabídka zmizí z tabulky a hráči, který nabídku přijal, se přičtou nabízené suroviny a odečtou požadované. Ošetření probíhá pomocí ověření uživatelského jména, které je unikátní. Reálně se suroviny mění vždy na straně serveru a všem hráčům server pouze oznamuje aktualizace stavů.

Jelikož server odešle všem hráčům informaci o přijaté nabídce, všichni hráči vědí o provedeném obchodu. Jelikož jsou ale obchodní nabídky stejně veřejně dostupné, myslím si, že v tomto případě to není bezpečnostní riziko.

Armáda a souboje

Obrazovka armády funguje stejně jako obrazovka, ve které se přiřazují obyvatelé do budov. Logika je zde lehce pozměněna tak, že vojáci mají separátní databázový model, který musí být přidán do každé mutace na této stránce.

Obrazovka, na které probíhají souboje, obsahuje pouze jednu mutaci a jedno odebírání. Mutace se zavolá, když hráč na někoho zaútočí.

```
mutation attack($targetPlayer: ID!) {
  attack(targetPlayer: $targetPlayer) {
    attacker
    defender
    attackerArmy {
      id
      recruited
    }
    defenderArmy {
      id
      recruited
    }
  }
}
```

Obrázek 7.11: Ukázka GraphQL mutace pro útok

Pro budoucí zobrazení výsledků bitvy obsahuje výsledek mutace stavy obou armád, které se zúčastnily boje. Odebírání je úplně totožné, přičemž zde už je problém s tím, že při rozesílání aktualizace stavu se pošle všem hráčům výsledek bitvy. Tím pádem všichni hráči znají stav armády hráčů, kteří na sebe zaútočili (normální uživatelé se tyto stavy nedozvědí, ale kdyby někdo odposlouchával síťovou komunikaci, tak tyto nové stavy armád může zjistit). V rámci diplomové práce jsem toto již neřešil, protože se do budoucna rozhoduji, zdali tuto informaci zpřístupním všem hráčům oficiálně, nebo vyřeším filtr aktualizací na straně serveru.

7.4 Stránky a komponenty

Celá aplikace je rozdělena na stránky, které mohou volitelně obsahovat znovupoužitelné komponenty. Každá stránka hry je dostupná pomocí vlastní URL adresy. Při prvním načtení se daná stránka sestaví na straně serveru (server side rendering = SSR). Při jakémkoliv přechodu na jinou stránku v rámci hry se však už všechny aktualizace dějí na straně klienta. Výhodou knihovny next.js je právě to, že je třeba danou stránku nadefinovat pomocí React komponent pouze jednou a knihovna se již postará o správnou funkcionalitu serverového či klientského vykreslení.

Komponenty vyššího řádu

Pro co největší znovupoužitelnost kódu používám návrhový vzor high order komponent, neboli komponent vyššího řádu [2]. Než ale začnu popisovat implementaci tohoto vzoru, nejprve bych se chtěl pozastavit nad jiným návrhovým vzorem, který se používá v rámci knihovny Apollo a její napojení na React komponenty. Jedná se o předávání parametrů pomocí render metody. Když mám např. GraphQL dotaz, obecným výsledkem je booleovská proměnná, která obsahuje informaci o tom, zdali se čeká na výsledek dotazu, dále výsledek daného dotazu, který může buď obsahovat data nebo chybu. Abych mohl tyto parametry využít ve svém kódu, musím obalit část kódu, kde chci mít tyto parametry dostupné,

Apollo komponentou Query. Problém nastává při požadavku použití více takovýchto GraphQL dotazů, protože je nutné tyto komponenty vnořovat do sebe a kód se tak stává méně přehledný.

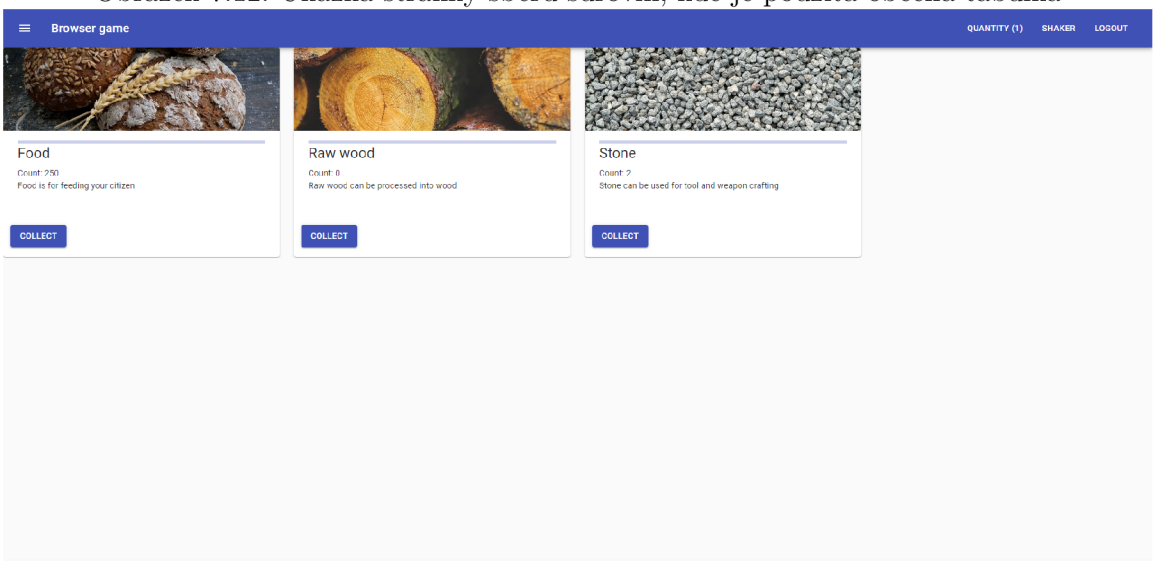
Zde právě může pomoci návrhový vzor komponent vyššího řádu. Pomocí knihovny `recompose`[1] můžu tuto funkcionalitu mít v separátním zdrojovém souboru a tato knihovna se mi postará o obalení celé aktuální komponenty komponenty vyšších rádů. Hlavní výhodou je převod kódu z vnořených komponent na komponenty definované v seznamu. Každé komponentě vyššího řádu se samozřejmě dají specifikovat parametry, díky kterým mohou být dostatečně obecné, aby se daly znovupoužít v kterékoliv obrazovce aplikace.

Příkladem takové komponenty vyššího řádu jsou komponenty withQuery a withMutation. Pro každé volání GraphQL nebo mutaci již není potřeba do sebe vnořovat jednotlivé komponenty, ale pouze použít tyto 2 komponenty vyššího řádu.

Komponenty

Jelikož jsem se snažil o co největší znovupoužitelnost kódu, tak je na každé stránce použita obecná tabulka, která na kartách zobrazuje dané suroviny, budovy a ostatní věci. Obecná tabulka se stará o to, aby byly karty zobrazeny správně na jakémkoliv zařízení, tedy aby bylo zobrazení responzivní. Jednotlivé karty se pak liší na každé obrazovce, avšak základní struktura, která je společná pro všechny karty, se nachází v implementaci obecné karty [1] [5].

Obrázek 7.12: Ukázka stránky sběru surovin, kde je použita obecná tabulka



Jediná odlišnost se nachází na stránce obchodu, kde se nalézá tabulka s obchodními nabídkami. Ta obsahuje stránkování obsahu, každopádně do budoucna plánuji s možností filtrace obchodních nabídek. Dále je zde komponenta, která umožňuje tvorbu nových obchodů. Zde se nachází pouze možnost výběru suroviny pomocí rozklikávacího menu a textová pole umožňující zadání pouze číselných hodnot pro specifikaci kvantity nabízených a poptávaných surovin.

[1] Zdroj: https://github.com/acdlite/recompose

Kapitola 8

Implementace genetického algoritmu

V této sekci bude popsána implementace všech částí nástroje, který využívá genetický algoritmus pro generování stavových automatů pro účely automatického hraní hry. Nástroj je naprogramovaný v programovacím jazyce Python 3 [1].

8.1 Třída Agent

Jak bylo popsáno v sekci 6, nástroj se skládá z 3 částí: genetického algoritmu, který spouští simulátor, který obsahuje jednotlivé agenty (hráče). Třída Agent reprezentuje právě jednoho takového umělého hráče pomocí svého DNA, dále však obsahuje stav všech objektů, které se ve hře vyskytují (suroviny, budovy, obyvatelé, skóre, atd.). Každý agent kromě surovin obsahuje implementaci metod, které jsou nutné k vyhodnocení podmínek a provádění akcí v rámci simulace. Jelikož u některých metod je nutná interakce s ostatními agenty, obsahuje každý agent odkaz na simulátor, který si drží v paměti všechny agenty v rámci dané simulace.

Každý agent musí být nastaven podle parametrů z konfiguračního souboru hry, který je využívaný i v samotné hře. Při jakékoliv změně parametru se tedy změna projeví jak ve hře, tak v simulátoru. Nebylo by optimální, aby si každý agent načítal ze souboru konfiguraci zvlášť, a proto se o toto stará simulátor. Jelikož má každý agent na simulátor odkaz, můžou být inicializační parametry načteny ze simulátoru (paměti) místo z disku.

Kromě inicializačních metod obsahuje každý agent metody, které se starají o vyhodnocení podmínek a metody, které provádějí akce. Každá podmínka a akce je definována pomocí unikátního ID, které koresponduje s jednotlivými genomy v rámci DNA (resp. dvojicemi genomů podmínka - akce). Společnou vlastností metod vyhodnocujících podmínky je booleovská návratová hodnota. V aktuální verzi obsahuje nástroj následující podmínky:

1. Podmínka, která je vždy splněna, tzn. vrací vždy pravdivou hodnotu.
2. Zdali je počet dané suroviny nulový (parametrem je typ suroviny).
3. Zdali se dá vyrobit daná surovina nebo postavit daná budova. Podmínka tedy zjišťuje, zdali má hráč dostatečný počet vstupních surovin.

[1] https://docs.python.org/3/

4. Zdali se dá do budovy přiřadit člověk.

Do budoucna plánuji množinu podmínek rozšířit, protože momentálně všechny podmínky operují s jednotkovým množstvím surovin.

Co se týče akcí, nástroj v aktuální verzi obsahuje následující:

1. Sebrat/vyrobit surovinu.
2. Postavit budovu.
3. Přiřadit obyvatele do budovy.
4. Vytvořit náhodný obchod, kde agent nabízí náhodné suroviny v náhodné kvantitě (obchod se vždy vytvoří pouze s validními počty surovin, podle toho, kolik jich agent má).
5. Přijmout náhodný obchod. Opět je zde provedena validace na korektní počet surovin.

Stejným omezením jako v případě podmínek je zde možnost výroby/stavby pouze v jednotkové kvantitě. Do budoucna opět plánuji rozšíření o konfigurovatelnou kvantitu.

Z testovacích důvodů je dále agent schopen vypsat na výstup všechny možnosti, které jsou v rámci podmínek a akcí dostupné v textové podobě. U každé podmínky a akce je zobrazeno unikátní ID, podle kterého je možné zkontrolovat např. počet podmínek a akcí nebo správnost automatu. Dále je implementována možnost výpisu samotného stavového automatu pomocí DNA.

Jednou z nejdůležitějších metod této třídy je výpočet skóre, podle kterého se ohodnocují jedinci v rámci populace. Návrh této metody má velký vliv na fungování celého genetického algoritmu. Výsledný stavový automat by měl umět vyrábět suroviny, ze kterých by měl vyrobit suroviny pokročilé, které spolu se základními surovinami může použít je stavbě budov. Do postavených budov by pak měl umět přiřadit obyvatele. Každou z těchto akcí, jak jsou vypsány po sobě, jsem ohodnotil tak, že čím více složitá akce, tím větší odměna za ní. Za každou sebranou/vytěženou základní surovinu je odměnou jedno skóre. Po vytěžení např. 10 základních surovin má hráč skóre 10. Za vyrobení pokročilé suroviny se už skóre za jednu surovinu násobí pěti (10 pokročilých surovin tedy dává skóre 50). Za postavenou budovu je násobitel 10 za každou budovu a 20 za přiřazeného obyvatele do budovy. Jelikož se skóre za každou složitější akci rapidně zvyšuje, genetický algoritmus by měl preferovat nová řešení, která obsahují právě tuto složitější akci.

8.2 Třída Simulator

Jelikož jsou všechny metody pro změnu parametrů simulace implementovány ve třídě Agent, je tato třída relativně jednoduchá. Hlavním úkolem této třídy je načtení konfiguračního souboru do paměti, jeho následné zpřístupnění agentům a pak zavolání inicializačních metod agentů. Kromě inicializace se pak samozřejmě stará o běh samotné simulace. Při vytvoření instance simulátoru je třeba specifikovat dva parametry: celkový počet kroků simulace a pole agentů. Agenti v poli musí být instanciováni, avšak o inicializaci jejich parametrů se stará simulátor.

Samotný běh simulace je opět velmi jednoduchý. O běh simulace se stará cyklus, který končí po dosažení maximálního počtu kroků. V každé iteraci cyklu se postupně volají metody agentů, které nejprve aktualizují jejich stav surovin (toto se děje pouze tehdy, pokud

má agent přiřazeného obyvatele v budově, která suroviny generuje) a pak se zavolá funkce iterate, která byla popsána v předchozí sekci.

Po úspěšném ukončení simulace se každému agentovi spočítá skóre, což je výstupem simulátoru. Genetický algoritmus pak může vyhodnotit úspěšnost dané generace.

8.3 Třída GeneticAlgorithmWrapper

Předchozí dvě třídy jsou pouze pomocníky pro tuto třídu, která se stará o výpočet nejlepšího možného stavového automatu, který bude hrát hru co nejlépe.

Při její inicializaci je možné specifikovat několik parametrů:

1. Počet podmínek
2. Počet akcí
3. Maximální počet kroků simulace
4. Velikost populace
5. Délka DNA jedince (musí být dělitelní dvěma)
6. Maximální délka evoluce
7. Pravděpodobnost křížení
8. Pravděpodobnost mutace
9. Pravděpodobnost, s jakou se provede buď mutace pomocí náhodné výměny hodnot v DNA nebo mutace pomocí náhodné změny hodnoty v DNA

Všechny tyto parametry mají výchozí hodnoty, takže lze tuto třídu instancovat bez jakékoliv specifikace parametrů.

K usnadnění implementace genetického algoritmu využívám knihovnu DEAP [2], která poskytuje jednoduché rozhraní a již implementované základní funkce. Algoritmus běží buď předem specifikovaný počet kroků nebo může být ukončen uživatelem pomocí přerušení. V obou případech je pak uložena poslední populace do souboru a uživateli je ukázán nejlepší jedinec v rámci této populace.

Při inicializaci (instancování této třídy) jsou nastaveny parametry specifikované uživatelem, registrovány funkce v rámci knihovny DEAP nebo moje vlastní funkce, inicializována počáteční populace (buď náhodným vygenerováním nebo načtením ze souboru, když takový soubor existuje) a vyhodnoceno skóre této populace. Skóre jsou pak přiřazena jednotlivým jedincům v populaci pro následné použití v dalších krocích algoritmu (evoluci).

Co se týče evoluce, jedná se o klasický cyklus, kde si postup ukládám do proměnné `generation`. Nejprve jsou vybráni kandidáti do nové generace, kteří jsou pak následně zkopírováni (jedná se o hlubokou kopii dat). Podle nastavené pravděpodobnosti je provedeno křížení následující upravenou funkcí 8.1:

[2] https://deap.readthedocs.io/en/master/

```
def twoPointCrossover(self, ind1, ind2):
  size = min(len(ind1), len(ind2))
  cxpoint1 = random.randint(1, size // 2)
  cxpoint2 = random.randint(1, (size // 2) - 1)
  if cxpoint2 >= cxpoint1:
    cxpoint2 += 1
  else:
    cxpoint1, cxpoint2 = cxpoint2, cxpoint1

  cxpoint1 *= 2
  cxpoint2 *= 2

  ind1[cxpoint1:cxpoint2], ind2[cxpoint1:cxpoint2] \
      = ind2[cxpoint1:cxpoint2], ind1[cxpoint1:cxpoint2]

  return ind1, ind2
```

Obrázek 8.1: Ukázka funkce, která se stará o křížení dvou jedinců

Jedná se o upravenou funkci z knihovny DEAP, která kříží dva jedince vygenerováním bodů křížení a následnou záměnou částí DNA. Kvůli důvodu popsaném v návrhu GA jsem si funkci musel upravit tak, aby byly body křížení generovány jako násobky dvou. Nejmenší nedělitelná jednotka jsou totiž dvě čísla v rámci DNA místo jednoho. Jak lze z funkce vidět, noví jedinci se vytvoří místo starých jedinců, nealokuje se tedy žádná dodatečná paměť a záměna je provedena tzv. na místě. Na začátku funkce je pak ještě logika, která zaručuje správné křížení dvou jedinců, kteří nemají stejně dlouhé DNA, avšak v aktuální verzi algoritmu se toto nemůže stát. Je to zde výhradně z důvodů budoucího rozšíření.

Dále se vygeneruje pravděpodobnost mutace, kde je dále rozhodnuto podle další pravděpodobnosti, jaký druh mutace se provede. První možností je nastavení náhodných čísel, druhou je náhodná záměna.

```
def uniformInt(self, individual, indpb):
  size = len(individual)

  for i in range(0, size, 2):
    if random.random() < indpb:
      individual[i] = random.randint(0, self.conditions - 1)
      individual[i + 1] = random.randint(0, self.actions - 1)

  return individual,
```

Obrázek 8.2: Ukázka mutace pomocí nastavení náhodných čísel

Jak lze vidět na funkci 8.2, náhodná čísla se generují pro celou dvojici. V proměnných `self.conditions` a `self.actions` jsou uloženy celkové počty podmínek a akcí stavových automatů.

Mutace se záměnou hodnot je opět upravená funkce z knihovny DEAP, kde jsem musel upravit logiku tak, aby se zaměnily místo jednotlivých genomů dvojice genomů.

Po výpočtu křížení a mutace se musí opět vyhodnotit skóre této nově vytvořené populace, což se děje pomocí inicializace a spuštení simulátoru s uživatelem specifikovanými parametry. Na závěr každé iterace jsou spočítány dodatečné statistiky pro následné vykreslení výsledků algoritmu do grafu. Když není algoritmus ukončen nebo se ještě nedosáhlo maximálního možného počtu kroků, pokračuje se znovu další iterací/generací.

Kapitola 9

Vyhodnocení výsledků genetického algoritmu

V této kapitole bude popsána metodika vyhodnocení výsledků genetického algoritmu a zároveň implementace testovacího nástroje. Dále budou ukázány tabulky a grafy, na kterých budou zobrazeny výsledky genetického algoritmu pro různá nastavení inicializace.

9.1 Implementace testovacího nástroje

Testovací nástroj slouží k jednoduchému spouštění genetického algoritmu s různými parametry. Po skončení algoritmu posbírá výsledky a postará se o jejich vykreslení do grafu. Jelikož komponenty, které tento nástroj využívá, jsou naprogramovány v Pythonu, je v něm implementován i tento nástroj. K vykreslení výsledků používám knihovnu `matplotlib`, která dokáže pomocí jednoduchého rozhraní v Pythonu vykreslit téměř jakýkoliv graf.

Ke správné funkci nejprve naimportuji všechny 3 moduly, které byly popsány v předchozí sekci. Stěžejní funkce, která spouští genetický algoritmus, bere stejné parametry, jako třída `geneticAlgorithmWrapper`. V nástroji si však mohu nastavit vícero konfigurací, mezi kterými se pak můžu přepínat a spouštět genetický algoritmus s různými parametry.

Pro zobrazení výsledků do grafu je třeba provádět jednotlivé kroky genetického algoritmu z nástroje a nikoliv z třídy `geneticAlgorithmWrapper`, protože potřebuji do grafu vykreslit statistické výsledky z jednotlivých kroků evoluce.

9.2 Metodika vyhodnocení

Pro vyhodnocení funkčnosti nástroje budu a nebudu před spuštěním měnit následující parametry z těchto důvodů:

Počet iterací simulátoru: Nebudu měnit, protože vysoký počet iterací zbytečně zpomaluje délku evolučního kroku bez jakékoliv přidané hodnoty. Při vyšším počtu iterací je sice výsledné skóre vyšší, avšak rozdíl mezi jedinci je pouze násobek simulace s nižším počtem iterací.

Velikost populace: Nebudu měnit. Velikost populace ovlivňuje celkový počet kroků evoluce, které jsou nutné k dosažení výsledku. Zpravidla platí, že čím větší velikost populace, tím méně kroků evoluce je třeba k dosažení výsledku.

Velikost jedince (počet stavů automatu): Budu měnit, abych ukázal flexibilitu genetického algoritmu hledat různé automaty s různými optimálními řešeními pro daný

počet stavů. Nejprve ukážu nejlepší možný automat, který poté srovnám s automatem vygenerovaným pomocí GA a případně daný výsledek vysvětlím. Možnosti: 2, 3, 4.

Pravděpodobnosti: Budu měnit. Jak bylo popsáno dříve, GA obsahuje interně celkem 3 různé konfigurovatelné pravděpodobnosti, které mají vliv na rychlost nalezení řešení. Ve vyhodnocení by tedy mělo být vidět, jak velký vliv mají. Každá pravděpodobnost bude mít 3 možnosti:

- Křížení: 0.5, 0.7, 0.9
- Mutace: 0.2, 0.5, 0.8
- Generování náhodného genu v rámci mutace nebo náhodné prohození dvou genů: 0.25, 0.5, 0.75

Jelikož existuje velké množství možností, bude testování probíhat ve dvou krocích. Nejprve budu měnit velikost jedince. Hlavním cílem tohoto kroku bude vysvětlení vygenerovaných automatů. V druhém kole se pak budou měnit pravděpodobnosti, kde bude cílem optimalizovat ideální konfiguraci pro nalezení řešení v co nejnižším počtu generací.

9.3 První kolo testování

Pro každou konfiguraci bude spuštěn genetický algoritmus do té doby, dokud nenalezne nejlepší možné řešení. Výsledkem tohoto testu bude nejlepší možný nalezený stavový automat. Automaty jsou celkem 3, kde u každého bude popsáno, proč byl vygenerován právě on. V tabulace bude uvedeno pouze DNA nejlepšího nalezeného automatu, které bude poté interpretováno.

Pro úplnost uvedu ostatní parametry, které se v rámci tohoto testování nebudou měnit:

- Velikost populace: 1000
- Počet iterací simulátoru: 20
- Pravděpodobnost křížení: 0.5
- Pravděpodobnost mutace: 0.2
- Pravděpodobnost, zdali bude mutace záměna genů nebo změna genu: 0.5

Tabulka 9.1: Vyhodnocení konfigurací pro první kolo testování

Počet stavů automatu	DNA nejlepšího automatu
2	17, 9, 8, 1
3	17, 9, 15, 13, 8, 1
4	27, 19, 19, 13, 2, 1, 5, 9

Automat s dvěma stavy

Nejmenší automat, který se dá pomocí genetického algoritmu vygenerovat, je právě tento. Automat s jedním stavem se vygenerovat nedá, protože by se na jedince s dvěma geny (jednou dvojicí) nedala použít operace křížení. Jedná o automat, který dokáže pouze sbírat základní suroviny.

Automat s dvěma stavy umí kromě sběru základních surovin i vyrábět suroviny pokročilé. V tabulce 9.1 je uvedeno následující DNA: 17, 9, 8, 1, které by se dalo interpretovat takto:

1. Když můžeš vyrobit dřevo (podmínka 17), vyrob dřevo (akce 9)
2. Když je počet krumpáčů roven nule (podmínka 8), seber surové dřevo (akce 1)

Na první pohled by se mohlo zdát, že druhá podmínka nedává smysl. Nejlepší možný automat však vypadá takto:

1. Když můžeš vyrobit dřevo (podmínka 17), vyrob dřevo (akce 9)
2. Vždy (podmínka 0) seber surové dřevo (akce 1)

Nejprve je třeba zkontrolovat, zdali může automat vyrobit dřevo, které se vyrábí ze surového dřeva. Když nemůže, vždy vytěží aspoň surové dřevo. Funkce tohoto automatu je tedy taková, že v jednom kroku vytěží surové dřevo a v dalším z něj vyrobí dřevo zpracované. Podmínka pro sebrání dřeva tedy musí být vždy splněna. Z obou automatů logicky plyne, že u obou je druhá podmínka vždy splněna, protože u prvního automatu počet krumpáčů nemůže nikdy přesáhnout nulu. Podmínky jsou tedy logicky ekvivalentní a automaty mají totožnou funkci.

Automat s třemi stavy

Tento automat již umí i stavět budovy, avšak chybí mu dodatečný stav, pomocí kterého by mohl do této budovy přiřadit obyvatele. Dle tabulky 9.1 má následující DNA: 17, 9, 15, 13, 8, 1. Interpretace je následující:

1. Když můžeš vyrobit dřevo (podmínka 17), vyrob dřevo (akce 9)
2. Když můžeš vyrobit krumpáč (podmínka 15), postav obchod (akce 13)
3. Když je počet krumpáčů roven nule (podmínka 8), seber surové dřevo (akce 1)

Stejně jako u předchozího automatu, jsou zde na první pohled nesmyslné stavy. První podmínka je však na první pohled správně a slouží k výrobě dřeva, stejně jako u předchozího automatu. Dále následuje stavba budovy, která je podmíněna zvláštní podmínkou. V interní logice hry stojí výroba obchodu a krumpáče 1 dřevo. Když je tedy možné vyrobit krumpáč, je zároveň možné vyrobit i obchod. Druhý stav tedy dává také smysl. Poslední stav je opět stejný jako u předchozího automatu a slouží k výrobě dřeva.

Celkově se tedy automat snaží vyrábět dřevo, když na to má suroviny. Když nemá, pokusí se v případě dostatku surovin vyrobit aspoň obchod. Když není splněna ani jedna z těchto podmínek, automat sebere surové dřevo.

Automat se čtyřmi stavy

Poslední a nejlepší automat, který zde bude popsán. Je definován pomocí DNA: 27, 19, 19, 13, 2, 1, 5, 9, kde interpretace je následující:

1. Když můžeš přiřadit obyvatele do obchodu (podmínka 27), přiřaď obyvatele do obchodu (akce 19)
2. Když můžeš postavit doupě sběračů (podmínka 19), postav obchod (akce 13)
3. Když je počet surového dřeva roven nule (podmínka 2), seber surové dřevo (akce 1)
4. Když je počet košíků roven nule (podmínka 5), vyrob dřevo (akce 9)

Na první pohled nesmyslné stavy jsou již pravidlem. První stav jako jediná dává smysl a slouží k přiřazování obyvatel do budovy. Jelikož je obchod jediná budova ve hře, kde obyvateli není třeba přiřadit žádný nástroj, aby v ní mohl pracovat, může tento stav existovat v automatu se čtyřmi stavy. Druhý stav se stará o postavení obchodu, kde doupě sběračů stojí stejně surovin (1 dřevo), takže ve výsledku funguje podmínka stejně, jako kdyby se hlídal stav surovin pro postavení obchodu. Třetí stav je pro sběr surového dřeva, které je sbíráno pouze tehdy, když je jeho počet nulový. Poslední podmínka pak slouží k výrobě dřeva, kde podmínka je v tomto případě vždy splněna, protože počet košíků nemůže přesáhnout nulu.

9.4 Druhé kolo testování

Cílem této části je nalézt optimální konfiguraci parametrů, při které je nalezeno řešení v co nejmenším počtu kroků evoluce. Jak bylo popsáno výše, budu zde měnit pouze parametry týkající se pravděpodobností. Ostatní parametry zůstanou nastaveny staticky s těmito hodnotami:

- Počet iterací simulátoru: 20
- Velikost populace: 1000
- Velikost jedince (počet stavů automatu): 3

Pro zobrazení v hlavičkách tabulky zjednoduším názvy pravděpodobností takto:

- **Křížení** - pravděpodobnost, zdali se provede křížení
- **Mutace 1** - pravděpodobnost, zdali se provede mutace
- **Mutace 2** - pravděpodobnost, zdali se provede buď mutace, která zaměňuje stavy automatu mezi sebou, nebo vygeneruje náhodný stav

Pro každou konfiguraci nechám genetický algoritmus spustit desetkrát a v tabulce pak budou zobrazeny následující statistické hodnoty: průměrný počet kroků evoluce, směrodatná odchylka, medián, minimální a maximální hodnota.

Tabulka 9.2: Vyhodnocení konfigurací pro druhé kolo testování

Křížení	Mutace 1	Mutace 2	Průměr	Směrodatná odchylka	Medián	Min	Max
0.5	0.2	0.25	306.5	425.59	93.5	6	1458
0.5	0.2	0.5	128.2	159.5	98.0	5	585
0.5	0.2	0.75	24.6	14.96	23.5	8	60
0.5	0.5	0.25	91.9	104.65	43.5	4	305
0.5	0.5	0.5	32.3	35.03	14.5	5	116
0.5	0.5	0.75	75.4	100.57	32.0	11	361
0.5	0.8	0.25	79.2	90.35	43.0	4	307
0.5	0.8	0.5	37.6	30.82	37.5	6	114
0.5	0.8	0.75	38.5	26.2	41.0	6	73
0.7	0.2	0.25	155.1	92.98	188.0	2	270
0.7	0.2	0.5	71.4	63.5	59.5	3	180
0.7	0.2	0.75	138.9	179.0	55.0	2	617
0.7	0.5	0.25	50.3	63.81	24.5	2	216
0.7	0.5	0.5	35.9	42.31	22.5	7	158
0.7	0.5	0.75	78.8	85.97	42.0	5	263
0.7	0.8	0.25	47.2	56.81	32.0	9	210
0.7	0.8	0.5	41	25.12	33.5	13	95
0.7	0.8	0.75	52.3	31.35	47.0	7	103
0.9	0.2	0.25	378.5	408.74	307.5	6	1523
0.9	0.2	0.5	172.2	130.6	178.0	11	502
0.9	0.2	0.75	90.5	73.66	68.5	6	212
0.9	0.5	0.25	38.3	37.96	17.5	2	129
0.9	0.5	0.5	116.9	132.2	69.5	11	447
0.9	0.5	0.75	92.9	118.08	29.5	6	365
0.9	0.8	0.25	34.9	29.23	20.5	2	87
0.9	0.8	0.5	26.7	11.96	23.0	10	43
0.9	0.8	0.75	46	76.3	11.0	1	264

Z tabulky 9.2 lze vyčíst, které konfigurace jsou výhodné pro inicializaci genetického algoritmu. Aby výpočet trval co nejméně tahů, je třeba nejen zvolit konfiguraci, která má nejmenší průměrnou hodnotu a medián, ale i směrodatnou odchylku kvůli konzistenci. Jelikož jsem provedl pouze deset měření pro každou konfiguraci, nejsou výsledky tolik přesné jako by byly např. se 100 měřeními, každopádně dávají aspoň minimální nadhled. Nejvýhodnější jsou tyto dvě konfigurace:

- Křížení: 0.5, Mutace 1: 0.2, Mutace 2: 0.75 (má nižší průměr, ale vyšší odchylku)
- Křížení: 0.9, Mutace 1: 0.8, Mutace 2: 0.5 (má nižší odchylku, ale vyšší průměr)

S oběma těmito konfiguracemi stačí genetickému algoritmu zhruba 25 generací k nalezení optimálního automatu se třemi stavy.

Kapitola 10

Závěr

Hlavním cílem této práce bylo navrhnout a implementovat prohlížečovou hru spolu s genetickým algoritmem generujícím stavové automaty, které hru automatizovaně hrají. Nejprve byl proveden průzkum trhu co se týče konkurenčních her a pak byla popsána veškerá teorie ke všem částem projektu, která je důležitá k pochopení jejího návrhu a implementace. V části návrhu byly popsány funkce, které jsem chtěl do hry implementovat, kde se povedlo téměř vše až na pár výjimek, jako je např. možnost hraní bez registrace hráče. Tyto funkce, mimo dalších, které ani nebyly plánovány pro dokončení této práce, však plánuji do budoucna dokončit. Oproti těmto funkcím však byly dokončeny stěžejní funkce hry jako je sběr základních surovin, výroba surovin pokročilých, stavba budov, možnost přiřazení obyvatel do budov společně s automatizací surovin, možnost vytváření obchodních nabídek spolu s možností jejich přijímání, což umožňuje obchodování mezi hráči. Hra také umožňuje vybrat ze seznamu registrovaných hráčů soupeře a zaútočit na něj. Hráč se také dokáže zaregistrovat a poté přihlásit do hry. Genetický algoritmus umí vygenerovat automat schopný hraní hry.

V implementační části byly uvedeny části kódu technologie GraphQL, pomocí které byla implementována převážná část logiky hry. Dále byl navržen a popsán z hlediska implementace genetický algoritmus spolu s uvedením důvodů, proč je vlastně využíván.

V kapitole vyhodnocení výsledků pak byla popsána metodika vyhodnocení spolu s návrhem vyhodnocovacího nástroje. Vyhodnocení bylo rozděleno do dvou částí, kde v první části byly popsány automaty, které genetický algoritmus generuje. Tyto automaty byly rozděleny podle počtu stavů a srovnány s optimálními automaty hrajícími hru nejlépe, jak je možné v rámci svých možností. V druhé části vyhodnocení pak byla nalezena optimální konfigurace parametrů genetického algoritmu, pro kterou je při spuštění algoritmu nalezení nejvhodnějšího automatu provedeno v co nejmenším možném počtu tahů.

10.1 Pokračování vývoje

Hra je sice v hratelné fázi, avšak mám v plánu mnoho vylepšení. Prvním z nich je technologický strom. Momentálně má hráč od začátku hry odemčené veškeré suroviny. Toto teď není problém, protože je celkový počet surovin malý. Když si hráč založí nový účet a přihlásí se, měl by vidět jen malý počet surovin, aby nebyl zahlcen zbytečnými informacemi. Jestliže plánuji do budoucna do hry přidat další suroviny, musím také přidat technologický strom, který zajistí jejich postupné odemykání. Další výhodou stromu je možnost hráče zdržet v postupu hrou a tím pádem ho u hry déle udržet.

V aktuální verzi hry na sebe můžou hráči útočit, hra však pouze vypočítá ztráty a upraví stavy armád obou hráčů. Funkčnost soubojů je sice zaručena, avšak pro samotnou hru toto nemá žádný velký význam. Plánuji tedy přidat tyto nové funkce: možnost ukrást suroviny a herní mapu. Když hráč zaútočí na jiného hráče a vyhraje, tak získá suroviny podle počtu vojáků, kteří mu po souboji přežili a zároveň počtu zbývajících surovin, které má obránce. Herní mapu chci implementovat hlavně z důvodu zavedení časového intervalu nutného k provedení útoku. V aktuální verzi hry se totiž souboj provede v ten samý moment, kdy je vyvolán útočníkem.

Abych motivoval hráče k válčení, budou se na mapě vyskytovat speciální místa, které budou po zabrání armádou generovat speciální surovinu, která nepůjde získat ve vesnici. Aby na sebe hráči útočili, logicky bude těchto míst méně než počet hráčů v okolí. K čemu ale hráčům tato speciální surovina bude? Kromě tohoto totiž plánuji hru rozdělit do různých epoch (např. pravěk, starověk, středovek, moderní doba), kde v každé době se bude vyskytovat jiná speciální surovina. Když hráč postoupí do nové epochy, tato speciální surovina bude pak dostupná v jeho vesnici a na mapě se objeví místa s novou speciální surovinou pro aktuální dobu.

Posledním nápadem, který zde zmíním, je možnost pasivního hraní. V jakékoliv online hře spolu hrají hráči s různými dovednostmi a může se stát, že někdo chce hru hrát pouze kvůli budování ekonomiky. Někdo jiný má ale radši válečný aspekt hry a tak bude na ekonomický založeného hráče útočit a nedá mu šanci se dále rozvinout. Jelikož se takovýto hráč nemá jak bránit, plánuji implementaci pasivního hraní. Hráč se bude moct přepnout do speciálního módu, ve kterém přijde o veškerou armádu, ale nikdo na něj nebude moct útočit. Když se pak rozhodne znovu se zapojit do hry, bude muset počkat určitou dobu, než ho hra přepne zpět do aktivního módu. V této době má možnost si postavit armádu, aby pak měl šanci na případně souboje s ostatními. U této funkce bude nutné vyladit dobu, po kterou se bude hráč přepínat do aktivního módu, aby tato funkce nebyla zneužívána.

Literatura

[1] Components and Props – React.
URL https://reactjs.org/docs/components-and-props.html

[2] Higher-Order Components – React.
URL https://reactjs.org/docs/higher-order-components.html

[3] Dhall, C.: *Relational vs. No-Sql*. Berkeley, CA: Apress, 2018, ISBN 978-1-4842-1073-4, s. 109–132, doi:10.1007/978-1-4842-1073-4_5.
URL https://doi.org/10.1007/978-1-4842-1073-4_5

[4] Fink, G.; Flatow, I.: *Introducing Single Page Applications*. Berkeley, CA: Apress, 2014, ISBN 978-1-4302-6674-7, s. 3–13, doi:10.1007/978-1-4302-6674-7_1.
URL https://doi.org/10.1007/978-1-4302-6674-7_1

[5] Freeman, A.: *Components and Props*. Berkeley, CA: Apress, 2019, ISBN 978-1-4842-4451-7, s. 249–286, doi:10.1007/978-1-4842-4451-7_10.
URL https://doi.org/10.1007/978-1-4842-4451-7_10

[6] Sivanandam, S. N.: *Introduction to genetic algorithms*. Berlin New York: Springer, 2007, ISBN 978-3-540-73190-0.

[7] Tiwari, S.: *Professional NoSQL*. Indianapolis, IN: Wiley, 2011, ISBN 047094224X.

[8] Vanhatupa, J.-M.: Browser Games: The New Frontier of Social Gaming. In *Recent Trends in Wireless and Mobile Networks*, editace A. Özcan; N. Chaki; D. Nagamalai, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ISBN 978-3-642-14171-3, s. 349–355.

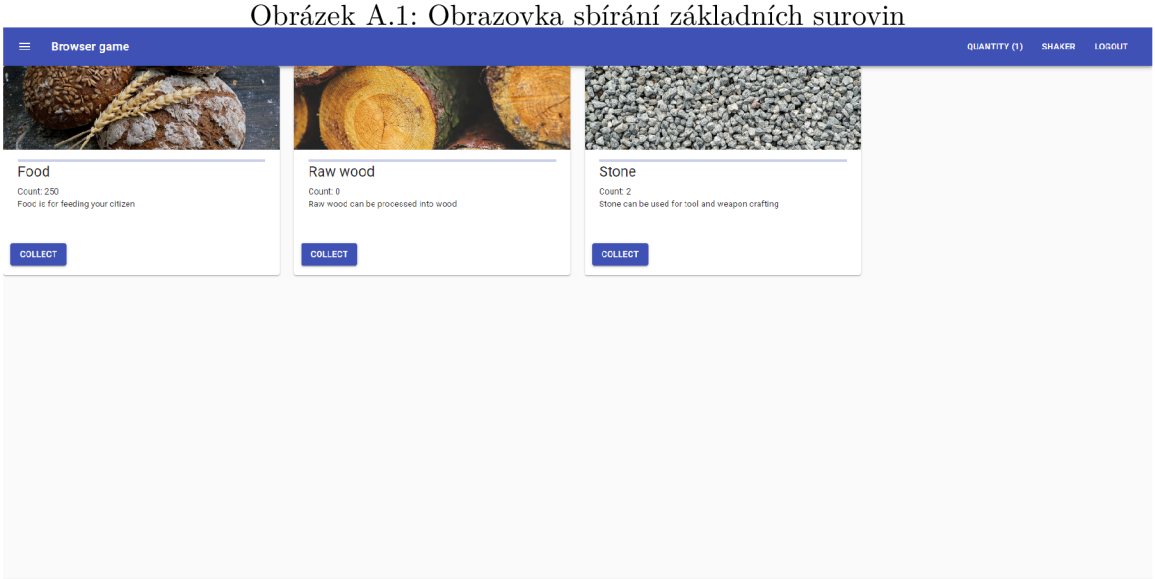
[9] Vázquez-Ingelmo, A.; Cruz-Benito, J.; García-Peñalvo, F. J.: Improving the OEEU's Data-driven Technological Ecosystem's Interoperability with GraphQL. In *Proceedings of the 5th International Conference on Technological Ecosystems for Enhancing Multiculturality*, TEEM 2017, New York, NY, USA: ACM, 2017, ISBN 978-1-4503-5386-1, s. 89:1–89:8, doi:10.1145/3144826.3145437.
URL http://doi.acm.org/10.1145/3144826.3145437

[10] Wang, V.; Salim, F.; Moskovits, P.: *The WebSocket API*. Berkeley, CA: Apress, 2013, ISBN 978-1-4302-4741-8, s. 13–32, doi:10.1007/978-1-4302-4741-8_2.
URL https://doi.org/10.1007/978-1-4302-4741-8_2

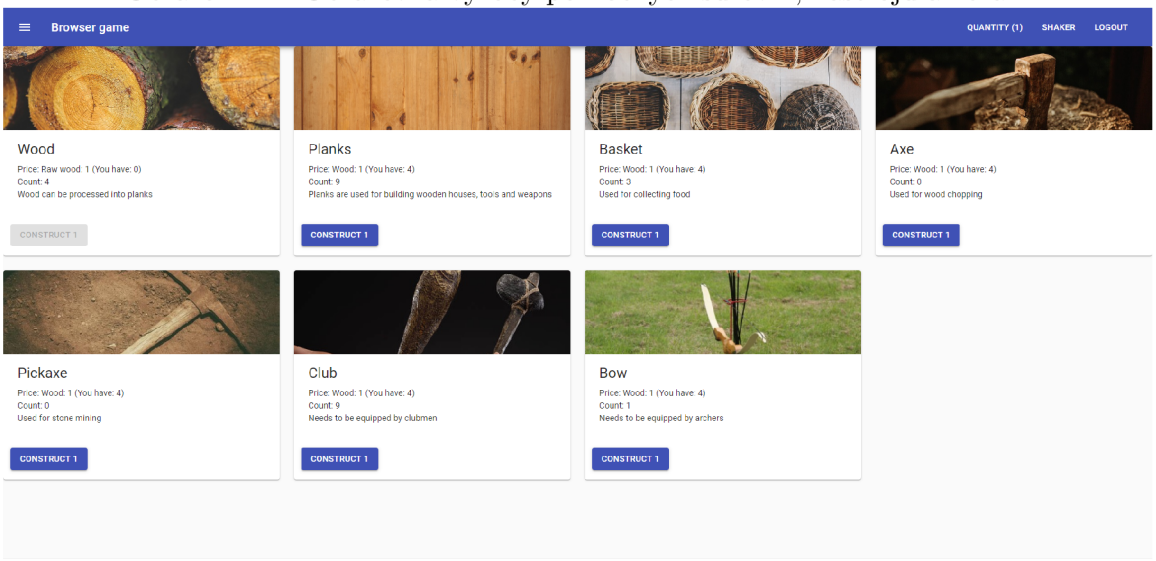
Příloha A

Obrázky ze hry

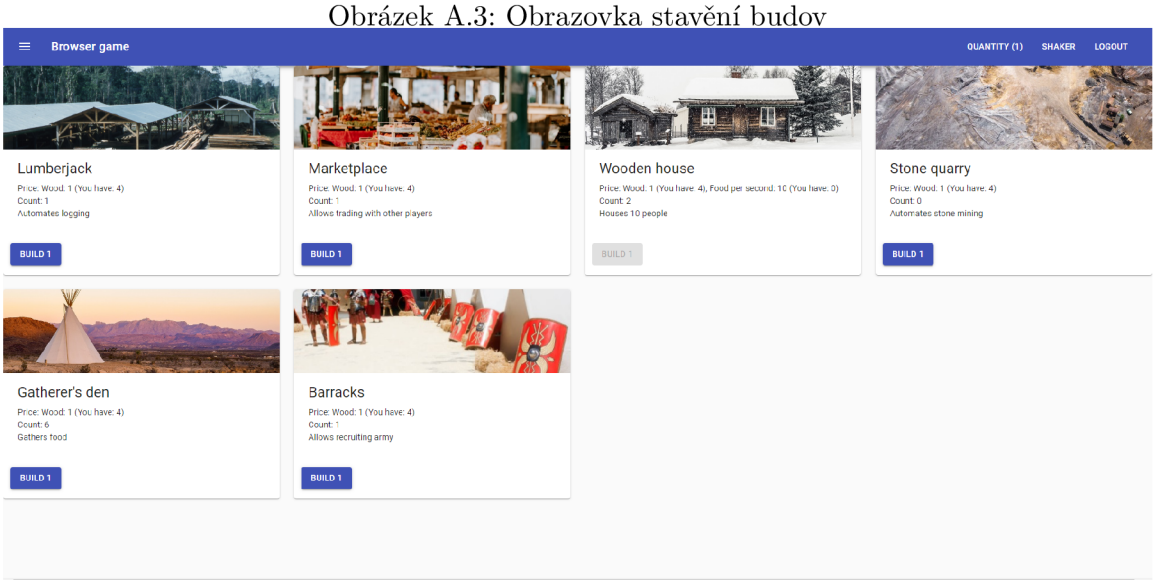
Obrázek A.1: Obrazovka sbírání základních surovin



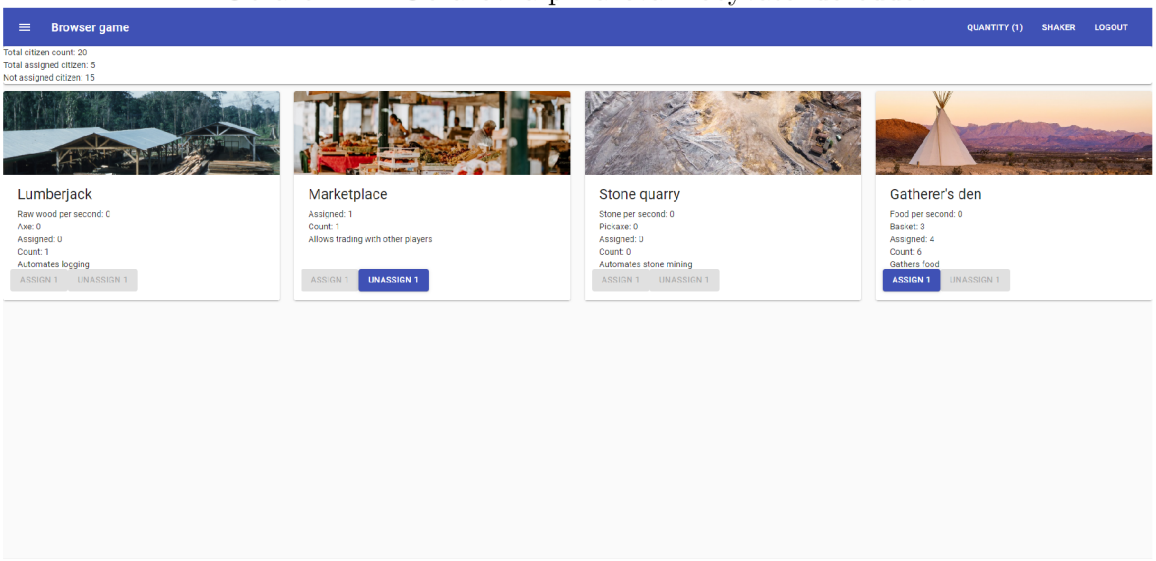
Obrázek A.2: Obrazovka výroby pokročilých surovin, nástrojů a zbraní



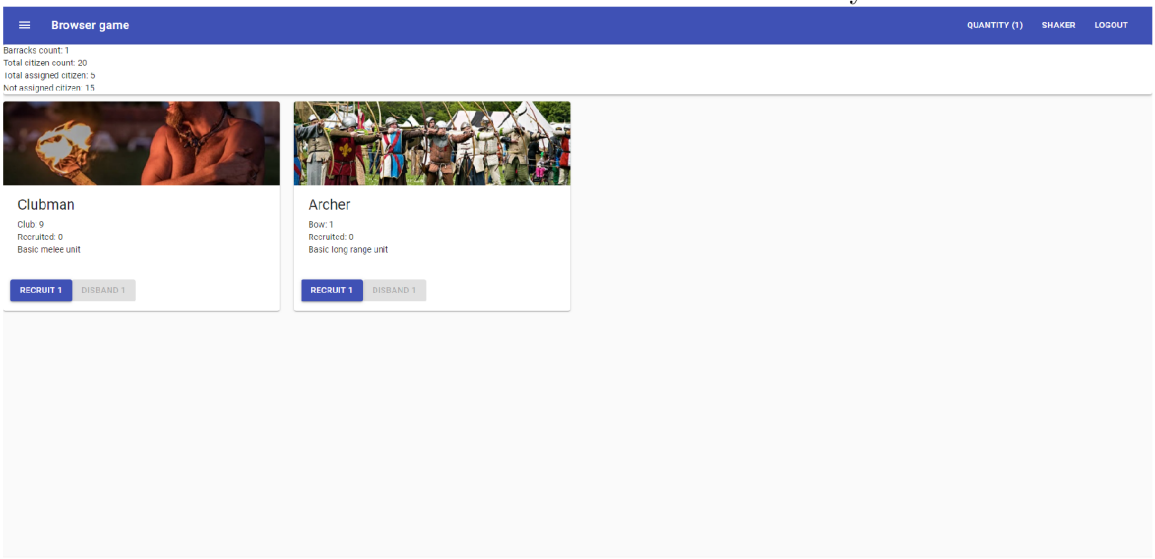
Obrázek A.3: Obrazovka stavění budov



Obrázek A.4: Obrazovka přiřazování obyvatel do budov



Obrázek A.5: Obrazovka rekrutování armády



Příloha B

Seznam podmínek a akcí genetického algoritmu

Tabulka B.1: Seznam akcí genetického algoritmu

ID	Význam
0	Seber jídlo
1	Seber surové dřevo
2	Seber kamení
3	Vyrob sekeru
4	Vyrob košík
5	Vyrob luk
6	Vyrob palici
7	Vyrob krumpáč
8	Vyrob prkno
9	Vyrob dřevo
10	Postav kasárny
11	Postav doupě sběračů
12	Postav dřevorubce
13	Postav obchod
14	Postav kamenolom
15	Postav dřevěný dům
16	Přiřaď obyvatele do kasáren
17	Přiřaď obyvatele do doupě sběračů
18	Přiřaď obyvatele do dřevorubce
19	Přiřaď obyvatele do obchodu
20	Přiřaď obyvatele do kamenolomu
21	Přiřaď obyvatele do dřevěného domu
22	Vytvoř náhodný obchod
23	Potvrď náhodný obchod

Tabulka B.2: Seznam podmínek genetického algoritmu

ID	Význam
0	Vždy splněno
1	Když je počet jídla nulový
2	Když je počet surového dřeva nulový
3	Když je počet kamení nulový
4	Když je počet seker nulový
5	Když je počet košíků nulový
6	Když je počet luků nulový
7	Když je počet palic nulový
8	Když je počet krumpáčů nulový
9	Když je počet prken nulový
10	Když je počet dřeva nulový
11	Když je možné vyrobit sekeru
12	Když je možné vyrobit košík
13	Když je možné vyrobit luk
14	Když je možné vyrobit palici
15	Když je možné vyrobit krumpáč
16	Když je možné vyrobit prkno
17	Když je možné vyrobit dřevo
18	Když je možné postavit kasárny
19	Když je možné postavit doupě sběračů
20	Když je možné postavit dřevorubce
21	Když je možné postavit obchod
22	Když je možné postavit kamenolom
23	Když je možné postavit dřevěný dům
24	Když je možné přiřadit obyvatele do kasáren
25	Když je možné přiřadit obyvatele do doupěte sběračů
26	Když je možné přiřadit obyvatele do dřevorubce
27	Když je možné přiřadit obyvatele do obchodu
28	Když je možné přiřadit obyvatele do kamenolomu
29	Když je možné přiřadit obyvatele do dřevěného domu

Příloha C

Instalace a spuštění hry a genetického algoritmu

Instalace a spuštění hry

Pro správný běh hry je třeba mít nainstalován `node.js` verze minimálně 8 a `npm` 6. Dále je nutné mít databázi `MongoDB` ve verzi 4. Pro instalaci pak stačí spustit:

```
npm install
```

Tento příkaz stáhne a nainstaluje všechny potřebné závislosti pro spuštění hry.

Hra se pak dá spustit buď ve vývojovém nebo produkčním módu. Pro spuštění ve vývojovém módu stačí hru spustit pomocí tohoto příkazu:

```
npm run dev
```

Pro produkční mód je třeba nejprve hru přeložit (výstupem je kód optimalizovaný tak, aby zabíral co nejméně místa a šetřil tak data posílaná po síti) a pak spustit pomocí těchto příkazů:

```
npm run build
npm start
```

Instalace a spuštění genetického algoritmu

Genetický algoritmus je spustitelný pomocí `Pythonu 3`, kde kvůli použití knihovny `DEAP` je nutné mít program `pip` (případně `pip3`, pokud je v počítači také nainstalován `Python 2` s příslušným programem `pip`), pomocí kterého je možné knihovnu nainstalovat:

```
pip install deap
```

nebo

```
pip3 install deap
```

Pro spuštění genetického algoritmu s vlastními parametry pak stačí z hlavní složky hry spustit:

```
python3 ga/plotter.py
```

Tento program spustí genetický algoritmus se zadanými parametry (parametry je nutné změnit přímo ve zdrojovém kódu skriptu) a zobrazí graf skóre v rámci generací. Genetický algoritmus se také dá spustit s přednastavenými parametry pomocí:

```
python3 ga/geneticAlgorithmWrapper.py
```

Pro postupné výstupy skóre jednotlivých generací však doporučuji algoritmus spouštět pomocí skriptu `plotter.py`.