Středoškolská technika 2025

Setkání a prezentace prací středoškolských studentů na ČVUT

Návrh a tvorba hry o budování továrny

Zdeněk Koukal

Vyšší odborná škola, Střední škola,

Centrum odborné přípravy,

Sezimovo Ústí, Budějovická 421

Budějovická 421, Sezimovo Ústí

Poděkování

Chtěl bych poděkovat především paní učitelce Mgr. Ludmile Jůzové za cenné rady a paní učitelce Mgr. Daně Kačenové za užitečné připomínky při zpracovávání této práce.

Anotace

Tato práce se zaměřuje na vývoj videohry, popsání benefitů a vlastností, které člověk může získat jejím hraním. Cílem hry tedy je podporovat logické myšlení hráče zábavnou a interaktivní formou. Rovněž se zabývá tím, jak se vlastnosti získané používáním herních mechanik využívají v reálném světě. Práce obsahuje celý proces tvorby počínaje přípravou pracovního prostředí, zahrnující instalaci a konfiguraci herního enginu Unity a programovacího prostředí Visual Studio. Dále se věnuje samotnému programování hry v jazyce C#, přičemž vysvětluje strukturu a funkčnost kódu, jeho výhody i omezení a důvody, které vedly k volbě konkrétních řešení.

Klíčová slova

C#, Unity

Annotation

This thesis focuses on the development of a video game by describing the benefits and features that a person can gain from playing it. The goal of the game is to support the player's logical thinking in a fun and interactive way. It also explores how the skills acquired through the use of game mechanics can be applied in the real world. The thesis covers the entire development process, starting with the preparation of the working environment, including the installation and configuration of the Unity game engine and the programming environment of Visual Studio. Furthermore, it delves into the actual programming of the game in C#, explaining the structure and functionality of the code, its advantages and limitations, and the reasons behind specific choices.

Keywords

C#, Unity

Obsah

1 Úvod

Na začátku školního roku jsem si stanovil cíl naprogramovat videohru. Programování byla věc, která mě na škole vždy bavila, a měl jsem chuť zhotovit nějaký větší projekt. Projekt, který by posunul moje znalosti o několik kroků vpřed, který by mi umožnil propojit kreativitu s logickým myšlením a zároveň byl dostatečně komplexní, aby mě naučil nové techniky a postupy. Vždy jsem měl v oblibě hry se strategickou tematikou, nutí hráče přemýšlet, plánovat a dělat věci efektivně. V poslední době jsem si navíc oblíbil hry zaměřené na tvoření, hry dávající hráčům volnost a možnost budovat s co nejmenším omezením. Tento projekt má ukázat krásu těchto her, jak hry tohoto typu rozvíjí mysl člověka a popularizovat samotný žánr. Myslím si, že her tohoto žánru je na internetu velice málo a je důležité jej ukázat světu. Chci hráčům umožnit zažít radost z objevování, budování a plánování, a zároveň ukázat, že videohry mohou mít i hlubší význam.

Vytvořený projekt se zaměřuje na vývoj strategické videohry, která hráči nejen poskytuje zábavu, ale také rozvíjí jeho logické myšlení. Cílem bylo vytvořit základní demo hry, které ukáže všechny důležité mechaniky a bude možno jej jednoduše rozšiřovat pro více obsahu dle obliby. Hra má kombinovat strategické rozhodování s interaktivními herními mechanikami, a přitom zůstat intuitivní a jednoduchá.

Vývoj hry probíhal v herním enginu Unity s využitím programovacího jazyka C#. Unity bylo zvoleno nejen pro svou flexibilitu a rozsáhlé možnosti, ale především pro mé již nabyté zkušenosti a dovednosti tvorby v tomto enginu. Projekt mi umožnil se dále zdokonalit v programování a prozkoumat složitější koncepty, jako je práce s herními objekty, jejich chování, interakce uživatele s hrou nebo například ukládání a načítání dat. V rámci vývoje jsem věnoval zvláštní pozornost tvorbě skriptů, které ovlivňují hratelnost, a uživatelskému rozhraní, které mělo zůstat jednoduché a intuitivní.

Důležitým prvkem hry je také její menu, které hráči umožňuje snadnou navigaci. Rozhodl jsem se pro minimalistický design, který odráží jednoduchost základního konceptu hry. Protože se hra řadí do žánru sandbox, nemá konkrétní konec. Místo toho hráči poskytuje nástroje a svobodu k tomu, aby si vytvářel vlastní cíle. Hráč si tak sám určuje, jakým způsobem chce hru hrát, co chce budovat a jaké strategie použije. Tento princip mě zaujal nejen v rámci vývoje hry, ale i z pohledu reálného života. Schopnost stanovit si vlastní cíle a aktivně na nich pracovat je podle mě klíčová nejen ve hrách, ale i v běžném životě.

Během vývoje jsem narazil na řadu problémů, které jsem musel postupně překonat. Každá část hry přinášela nové výzvy, ať už šlo o implementaci herních mechanik, optimalizaci výkonu nebo hledání efektivních řešení pro složitější problémy. Často jsem se musel v práci vracet, neboť koncept, pro který jsem se rozhodl, nebyl dále použitelný, a musel jsem ho nahradit jiným a znovu budovat a hledat způsoby správné funkce. Tento proces mi pomohl rozšířit mé znalosti o programování, naučil mě trpělivosti a ukázal mi, jak důležité je přemýšlet nad strukturou kódu a jeho efektivitou.

Tento projekt mi dal nejen cenné zkušenosti, ale také mě utvrdil v tom, že tvorba her je oblast, které bych se chtěl věnovat i do budoucna. Každá část vývoje byla pro mě novou lekcí a věřím, že výsledná hra dokáže hráčům nabídnout nejen zábavu, ale i možnost rozvíjet vlastní kreativitu a strategické myšlení.

2 POPIS VÝSLEDNÉHO PROJEKTU

2.1 Obsah hry

Hra, která byla vytvořena v rámci tohoto projektu, je demo verze hry s názvem Factory Builder. Obsahuje celkem pět různých továren:

- Miner (těžební stroj) – který je nutno položit na ložisko rudy.
- Smelter (huť) – předělá rudu na ingot.
- Constructor (konstruktér) – předělá jeden materiál na jiný.
- Assembly (montážní linka) – spojí dva materiály do jednoho.
- Shipping factory (zasílací továrna) – odesílá všechny kostky k jejich přepočtu za sekundu.

Obrázek 1: Továrny – zleva: Miner, Smelter, Constructor, Assembly, Shipping factory

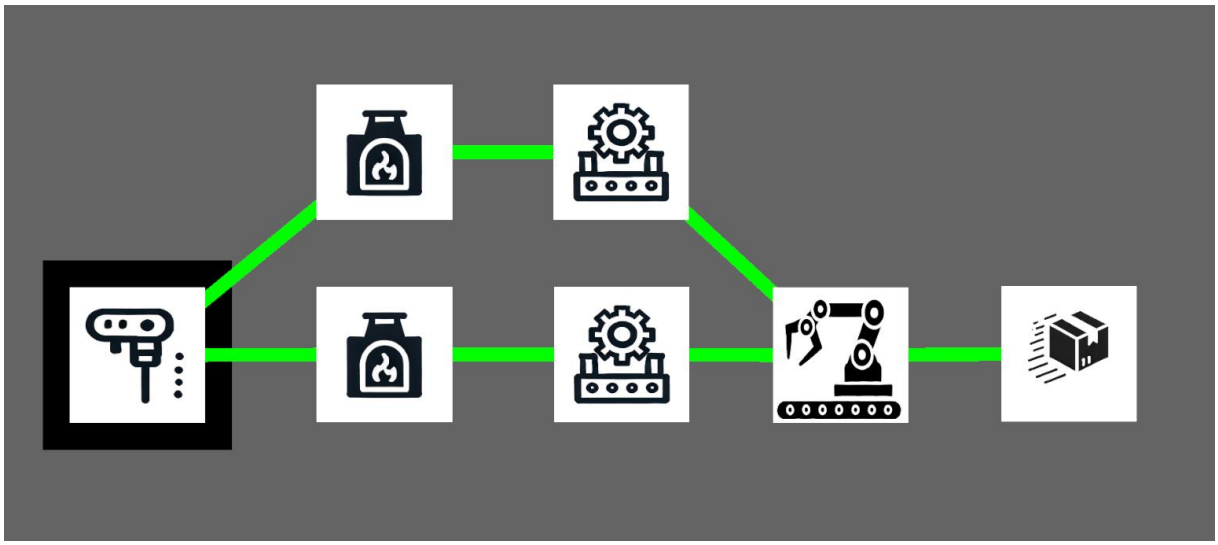
2.2 Základní koncept

Základní koncept hry je těžit rudu a postupně ji zpracovávat na lepší výrobek, například předělat železnou rudu na železný plát. Cílem je vytvářet co nejvíce nejpokročilejších materiálů za sekundu, v tomto případě železných kostek. Pro dosažení tohoto cíle musí hráč podstoupit rozšiřování svojí výrobní linky, přidávání nových továren a jiné. Obojí patří mezi hlavní mechaniky hry.

2.2.1 Režim propojování

Položené továrny je třeba propojit dle jejich potřeby, tak aby si mohly mezi sebou posílat zhotovené materiály (ukázka na obrázku 2). Stačí kliknout na tlačítko se zelenou ikonku pro zapnutí režimu propojení, kliknout na továrnu, která poskytuje vstupní materiál, a propojit ji. Každá továrna může mít libovolný počet vstupů i výstupů, záleží na hráči, kolik zrovna potřebuje. Pokud bylo jakékoliv z propojení uděláno nechtěně, či je třeba jej přepojit, není třeba mazat celé stroje. Stačí vybrat jak vstupní, tak výstupní továrnu, tím se výstup odpojí a může

se změnit. Samozřejmě je zde možnost ho zrušit úplně, v tom případě pomůže kliknutí pravého tlačítka myši, které ruší celý proces.



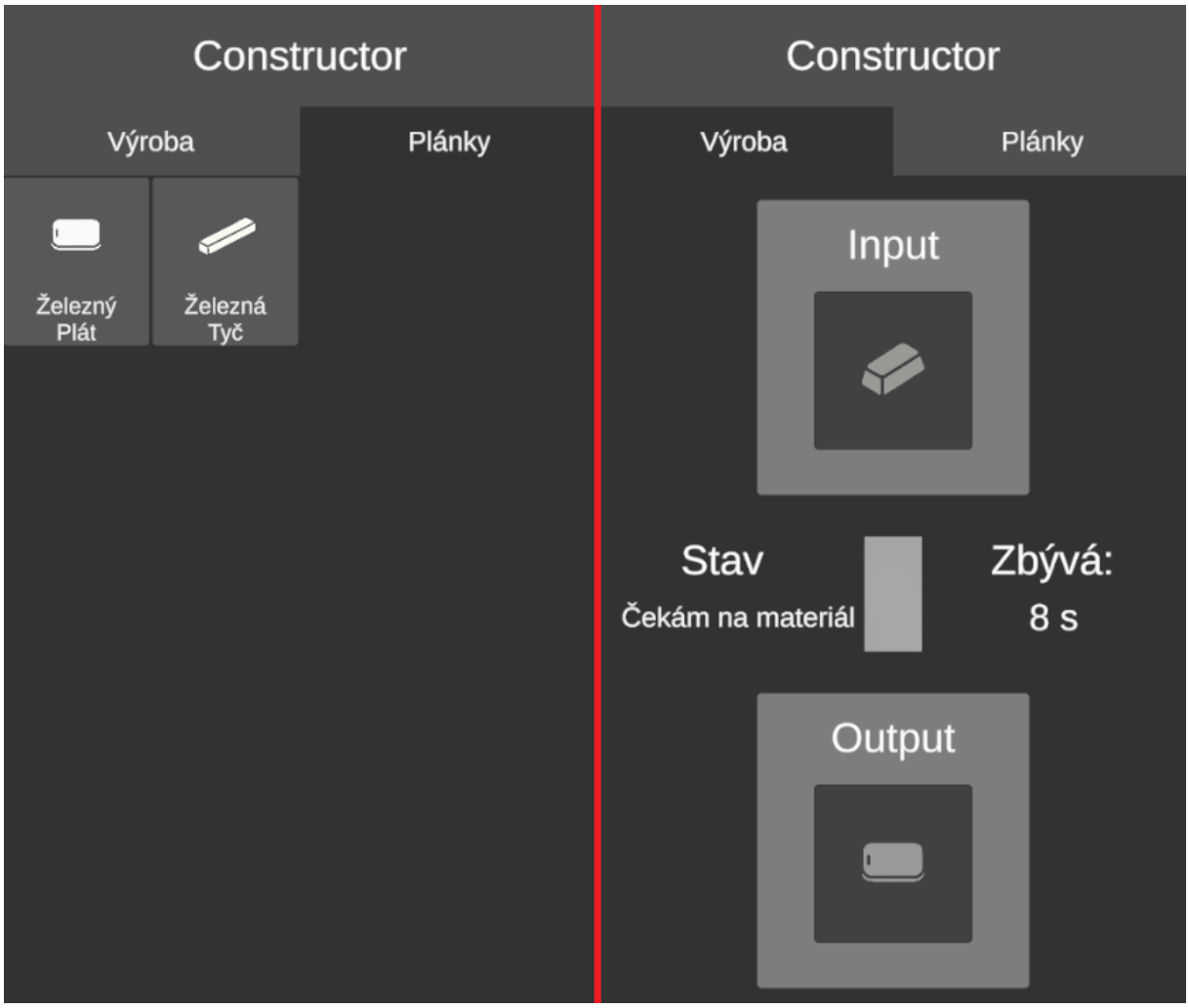
Obrázek 2: Propojené továrny

2.2.2 Rozhraní továren

Poslední věc, kterou je třeba zařídit, je přiřazení správného plánku pro výrobu. Bez něj žádná továrna nemůže fungovat. Toho se dosáhne nakliknutím továrny, a to bez jakéhokoliv aktivního režimu. Velké tlačítko X toto zařídí. Na pravé straně obrazovky se objeví menu se všemi dostupnými plánky. Po vybrání jakéhokoliv z nich se aktivuje panel výroby, který zodpovídá za grafické zobrazení veškerých informací o továrně. Ukazuje potřebné materiály pro výrobu, jak dlouho bude výroba trvat a stav továrny.

Zároveň je třeba zajistit, aby byly stroje správně propojené. Jestliže výstupní stroj, který byl továrně přiřazen, nemá stejný vstupní materiál jako výstupní materiál první továrny, k žádné výměně nedojde.

Ukázka plánku Constructoru je na obrázku 3.



Obrázek 3: Rozhraní továrny Constructor

Tímto jsou shrnuty veškeré implementované mechaniky. Výsledná hra je velice jednoduchá, a přitom s ní přichází spousty benefitů a zábavy.

3 CÍL HRY

3.1 Základní myšlenka

Základní myšlenka je stejně jednoduchá jako samotná hra. Způsob, jakým se transportují a zpracovávají materiály, je velmi úzce spojen s tokem informací v moderních softwarových aplikacích. Tento vztah mezi automatizací v této hře (a obecně v jakékoliv automatizační hře) a v reálném světě softwarového inženýrství je ještě výraznější díky tomu, že oba systémy jsou postaveny na stejných principech efektivity, optimalizace a automatizace. Jinými slovy, hra zaměřená na automatizovanou industrializaci a průmysl, který nám přinesl automatizaci v podobě sociálních sítí a dalších digitálních systémů, mají mnohem více společného, než by se na první pohled mohlo zdát. Zdrojem inspirace této myšlenky bylo video od tvůrce Tony Zhu [2].

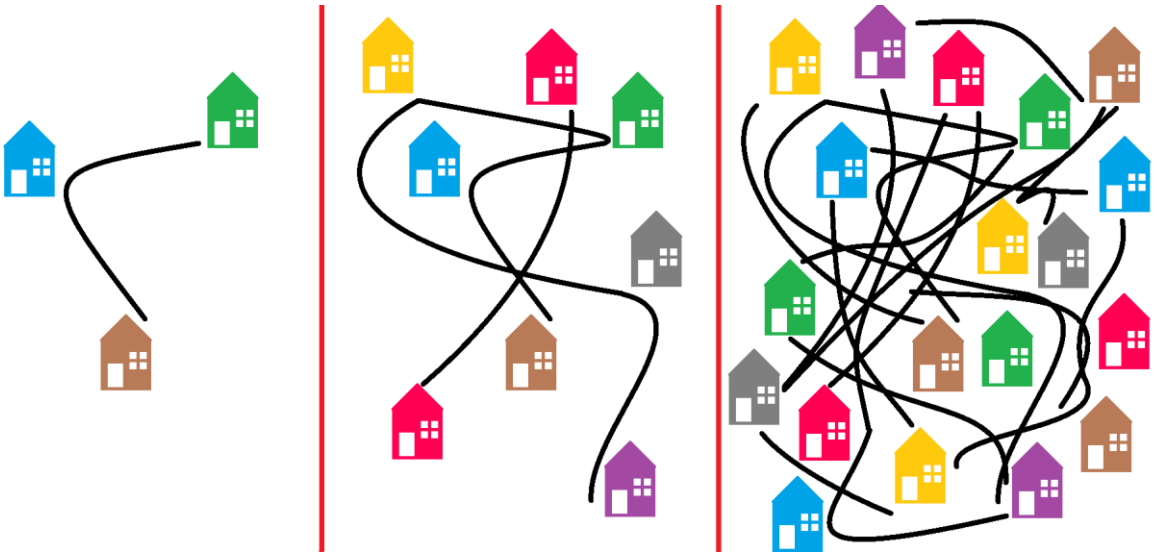
3.2 Špagety

Důvod, proč začínáme právě slovem špagety, je ten, že se jedná o jeden z nejrychlejších, nejjednodušších a nejlepších způsobů, jak jasně ukázat podobnosti Ať už jste hráčem her, jako je tato, nebo softwarovým inženýrem, pojem „špagety“ je vám dobře znám, protože označuje chaotickou a nepřehlednou strukturu, která vzniká při špatně organizovaném návrhu. Pokud ne, na obrázku 4 je názorný příklad.

Nalevo jsou domky tří kamarádů. Kamarád v hnědém domku chce zavolat kamarádovi v zeleném domku, a tak byla zřízena telefonní linka. Jelikož se jejich kamarád v modrém domku nechce cítit vyřazen, místo rovné čáry od zeleného k hnědému domku telefonní linku postaví tak, aby se mohl připojit i modrý domek.

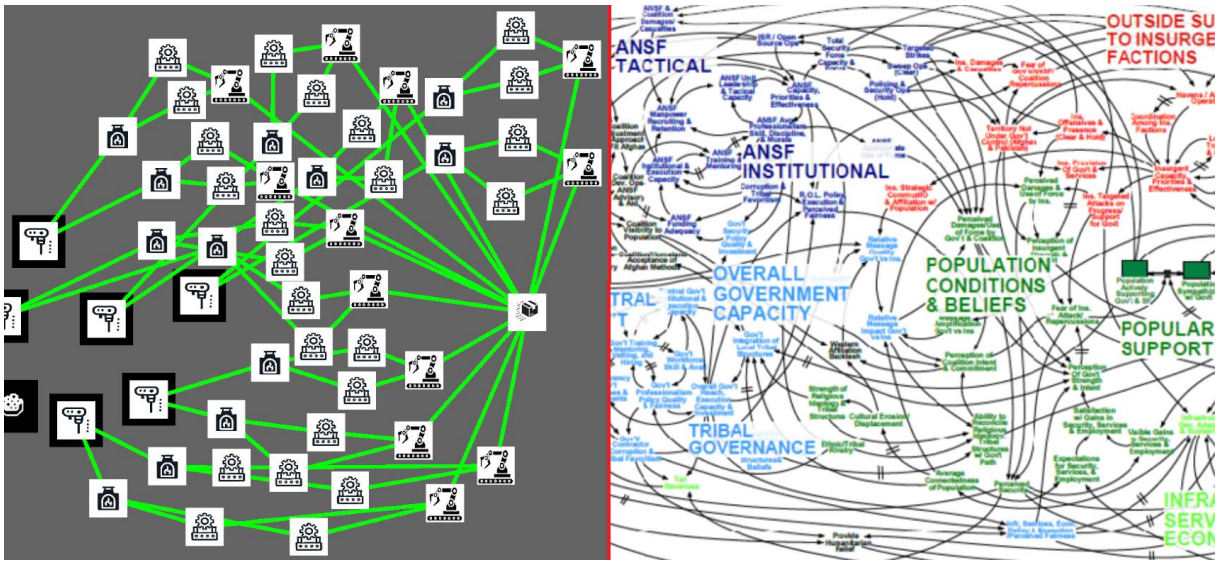
V prostředním obrázku můžeme vidět, že nově se do sítě chtějí připojit i ostatní domky, až na červené. Ty chtějí mít vlastní telefonní linku, kde si budou moct volat pouze mezi sebou. Dosavadní telefonní síť je tedy rozšířena a červeným domkům je zřízena nová.

Tímto tempem to pokračuje až do doby, kdy je propojené celé městečko. Jenže nyní nejde pořádně poznat, kdo je jak s kým propojen. A než se nadějete, byly vytvořeny „špagety“.



Obrázek 4: Domky a problém telefonní linky

Nejedná se pouze o věc při grafickém znázornění, existuje také termín „špagetový kód“. Je používán programátory pro zdrojový kód jakéhokoliv programu, který není dostatečně strukturovaný do jednotlivých logicky souvisejících částí. Na obrázku 5 lze vidět, jak vypadají „špagety“ ve výsledné hře oproti „špagetám“ v diagramu architektury mikroservisů.



Obrázek 5: Porovnání architektury výroby železných kostek a architektury mikroservisů COIN Dynamics [3]

Výsledek je velice nepřehledný a špatně se v něm orientuje, což navazuje na další kapitolu.

3.3 Code tracing

3.3.1 Co je code tracing?

Při hraní může nastat problém: nevyrábí se dostatek železných tyčí pro naši montážní továrnu. Ta je v tomto případě nucena čekat a není aktivní. Je třeba se tedy podívat o krok zpět, na

konstruktér, který je zodpovědný za výrobu železných tyčí. Zde je zjištěno, že i on nemá dostatek vstupního materiálu, železných ingotů. Je třeba jít znovu o krok zpět, kde je možno vidět, že jedna huť je připojena do spousty dalších továren. Problém je náhle jasný, je potřeba více továren pro výrobu železných ingotů. Jenže pro něj netěžíme dostatek železné rudy, je třeba tedy vyřešit i tento problém. Tento příklad je krásná ukázka code tracingu, v češtině obvykle přeloženého jako „sledování kódu“ či „procházení kódu“. Tímto se označuje proces, při kterém se sleduje průběh vykonávání kódu, většinou za účelem ladění nebo analýzy toho, jak program vykonává jednotlivé kroky.

3.3.2 Čitelnost

Kdyby bylo nutné provést code tracing na obrázku 5, tento úkon by byl téměř nadlidský. Musí se dbát ohled na čitelnost, neboli jak snadno může jiný člověk nebo samotný autor porozumět vytvořenému dílu. Ať už to je právě struktura továren či napsaný kód, a to i bez podrobného zkoumání nebo logiky.

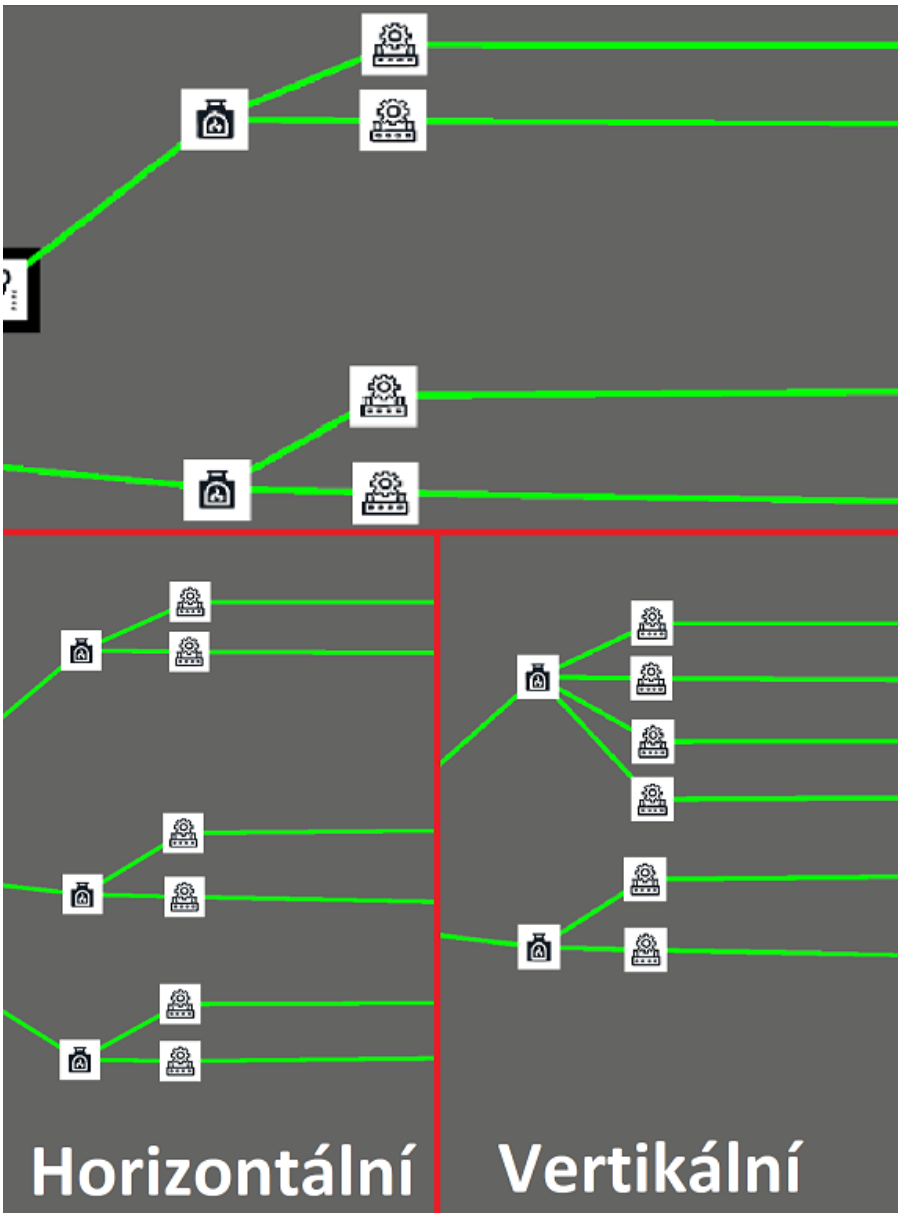
3.4 Škálování

Problém popsaný v kapitole 3.3.1 lze vyřešit například pomocí škálování. V reálném světě je možné si toto představit jako problém se serverem. Každá větev výrobní linky představuje jeden server a způsob, kterým se škálují tyto servery, je naprosto stejný jako ve Factory Builderu. Každá továrna dané větve je jako procesor daného serveru a jakmile je potřeba zvýšit výkon, můžeme tak učinit pomocí vertikálního či horizontálního škálování.

Vertikální škálování je proces, při kterém dojde ke zvýšení výkonu jednoho serveru přidáním hardwaru. V rámci Factory Builderu to znamená přidání více továren na stejnou výrobní větev.

Oproti tomu horizontální škálování přidává další servery do systému, čímž je výkon rozložen mezi více strojů. Ve Factory Builderu je tohoto dosaženo pomocí zvýšení počtu výrobních větví.

Ukázka původního stavu a různých druhů škálování je na obrázku 6.



Obrázek 6: Škálování

Poslední možností by bylo vylepšení hardwaru jako takového. Podobné hry tohoto žánru tuto možnost replikují buď více typy továren, či přidáním mechaniky pro vylepšení dané továrny. Například implementace vylepšovacích modulů, které zrychlí továrnu, na kterou jsou aplikovány.

3.5 Vyvažování zátěže

Výrobní odvětví neboli větve se tedy dají nahradit servery a každá továrna či stroj představují jejich procesory. V tomto případě se materiály, které jimi procházejí, dají představit jako uživatelé, kteří se právě chtějí připojit na web.

Mít nadbytek materiálů čekajících na další továrnu je věc, která hráči v podstatě nemusí vadit. Pokud místo materiálů čekají lidé, věci se mají jinak. Představa čekání několika desítek sekund

pro připojení na web, jako je YouTube, je prakticky nemožná. Pro správnou efektivitu je tedy potřeba správně škálovat. Jelikož počet těžebních ložisek je omezen, pokud je žádáno vyrobit co nejvíce kostek za sekundu, žádný materiál nesmí čekat.

3.6 Upstream / Downstream závislosti

Tato kapitola je úzce spojená s minulými kapitolami. Vysvětluje, proč padají webové stránky, co udělat, aby k tomu nedocházelo, a jak obecně funguje jejich údržba.

Zůstaneme u analogie, že materiály tvořené ve Factory Builderu jsou stejná věc jako informace zpracováváné v systémech softwarového inženýrství. Je zde vidět obecný tok. Směr, kde se materiály/informace nižší hodnoty postupně přepracovávají do materiálů/informací vyšší hodnoty.

Jestliže byla zvýšena těžba železné rudy, aby bylo možné vytvářet více železných plátů, je třeba se zamyslet, kam železné pláty dále putují a jak ovlivní další výrobu. Zde se aplikuje podobnost mezi Factory Builderem a reálným světem, o kterém byla řeč. Nyní má více lidí přístup na naši webovou stránku, ale to postupně zahlcuje náš systém. Upstream závislost, neboli služba, která poskytuje data pro další služby, je v tomto případě moc veliká na to, abychom ji mohli zpracovat. Což v tomto případě negativně ovlivňuje všechny downstream komponenty.

Z tohoto příkladu vyplývá, že pokud je vyráběno osm železných ingotů za sekundu, je třeba downstream služba, která je schopna zpracovat osm železných ingotů. Nebo dvě služby, přičemž obě budou schopné zpracovat čtyři železné ingoty.

Jestliže cokoli a kdykoli přestane fungovat, všichni si řeknou, jak se něco takového může stát. Chtěli se jenom připojit na webovou stránku či si zahrát svoji oblíbenou videohru. Jsou to firmy s obratem miliard korun, s prostory plné nejvýkonnější serverů. Já zmáčknu tlačítko pro vstoupení na stránku a stránka nefunguje. Co je špatně? Jak je možné, že takové firmy nejsou schopné ani udržet vlastní program v chodu, natož aby jej vyvíjeli dál.

Nyní je jasné, co všechno musí tyto firmy kontrolovat. Co musí podstoupit, aby jejich servery neustále běžely hladce. Nejsou to žádné tajné speciální techniky, ale je třeba spousta lidí, která tuto práci dělá. Práci, která vyžaduje logické myšlení, být schopen se zamyslet, kde je problém, co je třeba k jeho vyřešení a jak toto provést. Všechny tyto vlastnosti jsou poskytovány a dále rozvíjeny hraním tak jednoduché a zábavné videohry.

4 UNITY EDITOR

4.1 Úvod do Unity Editoru

Unity Editor je hlavním nástrojem pro vývoj her v Unity Engine. Umožňuje editaci herního prostředí po grafické stránce, správu objektů, nastavení fyziky, skriptování programů a mnoho dalšího. Díky přehlednému rozhraní a širokým možnostem konfigurace je oblíbeným nástrojem pro začátečníky i profesionální vývojáře.

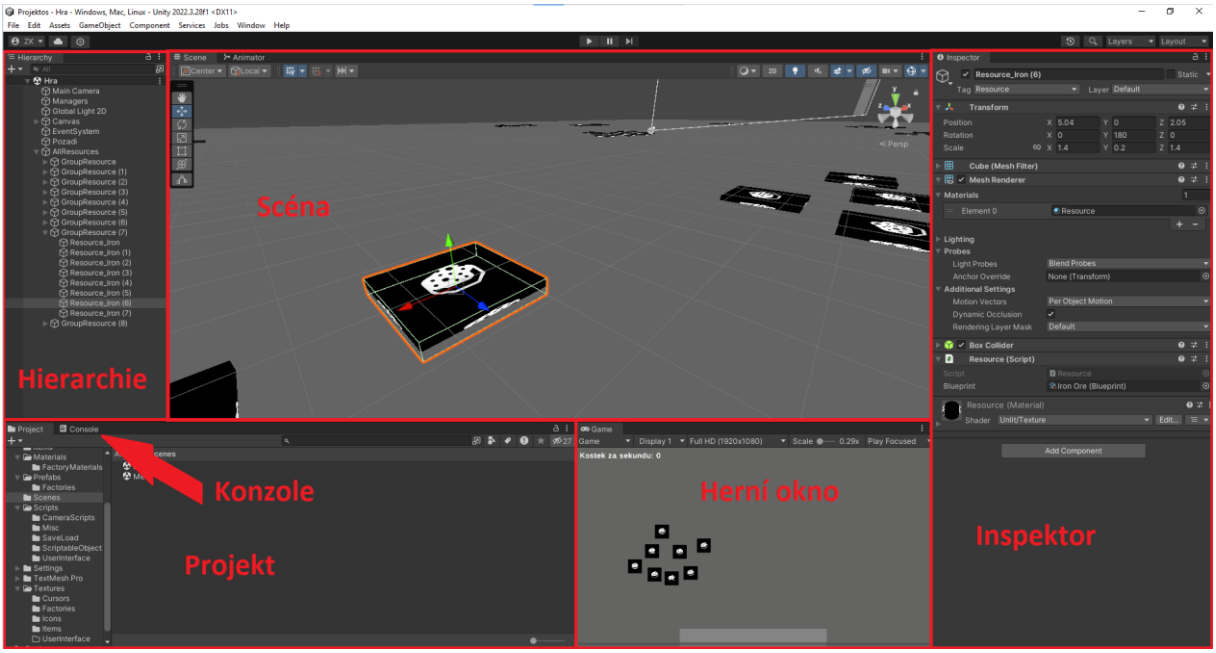
Hlavní funkcí Unity Editoru je poskytovat uživateli přehledný pracovní prostor, ve kterém lze organizovat herní scény, nastavovat chování objektů a testovat výsledky v reálném čase. Editor je modulární a nabízí různé panely, které lze přizpůsobit podle potřeb uživatele. K doplnění informací byla využita oficiální dokumentace od Unity Technologies [4].

4.2 Struktura Unity Editoru

Unity Editor je rozdělen do několika hlavních částí, které společně tvoří pracovní prostředí pro vývojáře. Každá část má vlastní okno, které funguje stejně jako okna ve Windows. Lze tedy říct, že Unity Editor je něco jako plocha pro organizaci různých pracovních komponentů vývoje (viz obrázek 7). Mezi ty nejdůležitější patří:

1. Hierarchy (Hierarchie) – seznam všech objektů, které se nacházejí v aktuální scéně.
2. Scene (Scéna) – vizuální reprezentace herního světa, kde lze objekty umisťovat, měnit jejich velikost a upravovat jejich vlastnosti.
3. Game (Herní okno) – náhled na to, jak hra vypadá z pohledu hráče. Toto okno se používá pro testování herního prostředí.
4. Inspector (Inspektor) – zobrazuje detailní informace o vybraném objektu. Zde lze upravovat komponenty, měnit hodnoty parametrů a přidávat nové vlastnosti.
5. Project (Projekt) – správa všech souborů, skriptů, textur, modelů a dalších assetů použitých ve hře.
6. Console (Konzole) – zobrazuje chybové zprávy, varování a další výstupy během testování hry.

Každý z těchto komponentů má své specifické využití a je klíčovou, až nenahraditelnou pro efektivní práci na tomto projektu.



Obrázek 7: Struktura Unity Editoru

4.2.1 Hierarchie

Okno Hierarchie obsahuje seznam všech herních objektů, které se nacházejí ve scéně. Každý objekt v Unity, ať už se jedná o kameru, postavu nebo světelný zdroj, je reprezentován jako GameObject. Objekty se zde organizují do hierarchické struktury, což umožňuje vytvářet složitější vztahy mezi jednotlivými prvky scény.

Například pokud vytvoříme objekt „Auto“ a přidáme k němu kola jako podřízené objekty, pohyb auta automaticky ovlivní i pohyb kol. Tato hierarchie usnadňuje správu složitějších objektů a umožňuje efektivní práci s transformacemi, přesněji s pozicí, rotací či měřítkem.

4.2.2 Scéna

Okno Scéna slouží k vizuální editaci herního světa. Vývojář zde může pracovat s tranformacemi objektů, upravovat jejich vlastnosti a testovat jejich interakce. Unity Editor nabízí několik nástrojů pro manipulaci s objekty:

- Move Tool – umožňuje posouvat objekty v prostoru.
- Rotate Tool – slouží k otáčení objektů kolem os.
- Scale Tool – umožňuje změnu velikosti objektů.
- Rect Tool – užitečný pro práci s UI prvky a 2D objekty.

Kromě těchto základních nástrojů lze využít i Grid and Snapping, což jsou funkce pro přesnější umisťování objektů do takzvaného World Gridu, neboli do světové mřížky. V Unity hraje roli referenční mřížka scény, ke které se pomocí Grid and Snapu přicvaknout objekty v prostoru.

4.2.3 Herní okno

Okno hry slouží k náhledu na hru z pohledu hráče. Zobrazuje scénu tak, jak ji vidí aktivní kamera, umožňuje testovat herní mechaniky, uživatelské rozhraní či vizuální prvky bez nutnosti sestavení hry. Také obsahuje panel Statistik, který poskytuje údaje o výkonu hry, jako jsou její snímky za sekundu, využití procesoru, počet vykreslených objektů a jiné.

Na rozdíl od okna Scéna, kde je možné volně manipulovat s objekty a kamerou, je v herním okně interakce omezena pouze na chování naprogramované ve skriptech. Vývojář zde vidí výsledek své práce v prostředí, které simuluje finální verzi hry.

4.2.4 Inspektor

V Inspektoru jsou zobrazovány veškeré informace o aktuálně vybraném objektu. Každý GameObject v Unity může obsahovat různé komponenty, které definují jeho chování. Mezi základní patří:

- Transform – určuje pozici, rotaci a měřítko objektu.
- Mesh Renderer – stará se o vykreslování objektu.
- Collider – definuje kolizní oblast objektu.
- Rigidbody – umožňuje objektu reagovat na fyzikální síly.

Je možné přidávat nové komponenty nebo upravovat jejich vlastnosti. Dříve zmíněné komponenty především definují vlastnosti objektu samotného, pomocí skriptů je však možné ovlivňovat i ostatní objekty. Díky této vlastnosti se v každém Unity projektu najdou prázdné objekty, které obsahují pouze skripty jako jejich komponenty pro správu celé scény. Díky těmto prvkům je možné široké přizpůsobení chování objektů.

4.2.5 Projekt

V okně Projekt se nacházejí všechny soubory použité v projektu. Toto okno funguje jako průzkumník souborů Unity, kde lze organizovat skripty, textury, modely a další assety do složek. Nejdůležitější součástí práce s projektem je správná organizace souborů, rozdělení assetů do složek, například:

- Scripts – pro C# skripty
- Textures – pro obrázky a textury
- Prefabs – pro předpřipravené objekty
- Scenes – pro uložené herní scény

Díky této organizaci se lze lépe orientovat v projektu a efektivněji spravovat jeho obsah.

4.2.6 Konzole

Okno Konzole slouží k zobrazování chybových hlášení, varování a dalších informací během běhu hry. Vývojář zde může sledovat výstupy ze skriptů, což pomáhá při ladění a odstraňování problémů.

V Unity se toto řeší pomocí příkazů jako Debug.Log() zobrazení vlastní zprávy v konzoli, což je užitečné při testování různých funkcí a chování objektů.

4.3 Instalace Unity balíčků

Než mohl být zahájen samotný vývoj, bylo nutné do projektu importovat několik klíčových balíčků. Unity nabízí možnost správy balíčků prostřednictvím Package Manageru, ve kterém lze vidět již nainstalované balíčky či vyhledávat nové. Aby byl umožněn přístup k širšímu výběru balíčků, bylo nutné změnit zdroj na Unity Registry. Tento krok zpřístupnil všechny základní balíčky, které Unity nabízí, což je výhodné především při zakládání nového projektu, který nebude mít složité a specifické požadavky.

Pro tento projekt byly vybrány dva zásadní balíčky. Prvním byl nový Input System, který slouží k nastavení ovládání hry. Umožňuje detekovat vstupy z klávesnice, myši nebo jiných ovládacích zařízení a reagovat na ně v rámci herní logiky. Tento systém je flexibilnější než výchozí způsob detekce vstupů a umožňuje snadnější konfiguraci klávesových zkratek i podporu více ovládacích prvků. Druhým důležitým balíčkem byl TMP_Text (TextMesh Pro), který je určen pro práci s textem v uživatelském rozhraní, také známý jako UI (User Interface). Tento balíček poskytuje lepší možnosti formátování textu než standardní textové komponenty v Unity a umožňuje například úpravu písma, zarovnání či různé vizuální efekty.

Uživatelské rozhraní, o kterém byla řeč, hraje v každé hře důležitou roli a je úzce propojeno s kamerou, která je jednou z klíčových součástí Unity. Kamera funguje jako virtuální objektiv a určuje, co hráč ve hře uvidí na obrazovce. Uživatelské rozhraní se pak do tohoto pohledu vkládá jako vrstva, která může překrývat herní prostředí a zobrazovat různé interaktivní prvky, jako jsou tlačítka, ukazatele nebo textová pole. Díky tomu lze snadno vytvořit jednoduché a přehledné ovládání, které informuje hráče o důležitých aspektech hry.

Po dokončení těchto příprav bylo pracovní prostředí plně připraveno k zahájení vývoje. Další balíčky je možné přidat i v průběhu vývoje, takže se není třeba strachovat. Tyto dva byly pro začátek dostačující. S těmito základními nástroji bylo možné pustit se do samotného programování a začít vytvářet první herní prvky.

5 TVORBA SKRIPTŮ

Programování hraje klíčovou roli při vývoji jakéhokoliv projektu v Unity, protože právě skripty určují chování objektů, herních mechanik a interakcí s hráčem. Unity využívá programovací jazyk C#, který umožňuje vytvářet dynamické herní systémy a je velice flexibilní.

Tato kapitola se zaměřuje na nejzábavnější a největší část z celého projektu – tvorbu skriptů v Unity, tedy na proces, který přináší projekt k životu.

5.1 Základy programování v Unity pomocí jazyku C#

Nejprve je třeba si říct, jak je programování v Unity odlišné od klasického programování, například v konzoli. Tato práce předpokládá, že čtenář je již obeznámen s minimálními až základními znalostmi programování. Deklarace a inicializace hodnot, datové typy či třídy by měly být termíny, se kterými je již obeznámen.

V Unity nově přibydou základní metody Start() a Update(), které jsou tvořeny automaticky v rámci programu Visual Studio, dále můžeme použít metodou Awake(), která je s těmito metodami úzce spojená, zde je jejich příklad:

```
public class Skript : MonoBehaviour
{
  private void Awake()
  {

  }

  private void Start()
  {

  }

  private void Update()
  {

  }
}
```

5.1.1 Awake()

Metoda Awake() se volá při inicializaci objektu, ještě před tím, než je objekt aktivován či spuštěn. Její primární použití je pro inicializaci proměnných.

5.1.2 Start()

Metoda Start() je volána těsně po metodě Awake(), přímo po spuštění objektu, kterého je skript součástí. Hlavním důvodem, proč jsou tyto dvě metody rozděleny, i když se obě volají hned na samotném počátku, je dříve zmíněná inicializace. Občas je třeba pracovat s proměnnými jiných

skriptů již na začátku, to však není možné, pokud nejsou inicializované, takto se dá předejít mnoha kompilačním chybám.

5.1.3 Update()

Metoda Update() je volána na začátku každého snímku v průběhu hry, je tedy neustále aktualizována. V případě použití výchozího Input Systému je zde detekce vstupů, kdy se několikrát za sekundu kontroluje, zda byla zmáčknuta určitá klávesa. Tento způsob je velice neefektivní a zatěžuje výpočetní techniku, proto je doporučeno využití nového Input Systému.

5.2 Menu

První viditelná část hry při spuštění je její menu, které slouží jako základní navigační prvek pro hráče. Zde se uživatel může procházet různými panely pomocí tlačítek, které jsou s nimi propojeny.

5.2.1 Panel

V Unity panel představuje grafický kontejner, který lze využít jako pozadí pro ostatní prvky uživatelského rozhraní, například tlačítka, texty nebo obrázky. Funguje jako objekt, jehož hlavním účelem je vizuální oddělení ovládacích prvků. Tyto prvky se však také dají seskupit pod jedním panelem jako rodič, v tomto případě je možné jej využít pro lepší seskupení hierarchie.

5.2.2 Tlačítko

Tlačítko v uživatelském rozhraní Unity je interaktivní objekt, který reaguje na kliknutí myší nebo v případě mobilu dotyk a spouští přiřazenou akci. Lze ho kombinovat s balíčkem TextMesh Pro, pro lepší kvalitu a čitelnost textu.

5.2.3 Obsah menu

Mezi hlavní tlačítka, které jsou obsažena v menu, patří „Načíst hru“, které se zobrazí pouze za podmínky, že se na počítači nachází soubor *data.game*, vytvořený při předchozím ukládání hry. Pokud tento soubor neexistuje, hráč je nucen zmáčknout tlačítko „Nová hra“. Dále jsou přístupná tlačítka „Autor“, které popisuje původ hry, a „Konec“, které hru ukončí. Pro poslední tlačítko „Nastavení“ byl vytvořen program, ale nastavení jako takové se zdálo přebytečné, nebylo jej třeba pro dokázání cíle této práce, a tak bylo zanecháno pouze ve vypnuté formě.

5.2.4 Skript Menu

Skript Menu se stará nejen o menu samotné, ale o všechny prvky s ním propojené. Ať už jsou to podřadné panely či pauza.

Na začátku jsou deklarovány tři proměnné, přičemž každá z nich se liší svým účelem a způsobem přístupu.

První proměnná je veřejná instance třídy Menu, která je statického původu. To znamená, že je sdílená napříč všemi instancemi třídy a lze k ní přistupovat bez vytvoření objektu. Přestože je veřejná, její přístupnost k zápisu je omezena použitím { get; private set; }. Taková hodnota může být změněna pouze v rámci třídy, nikoli zvenčí.

Druhá proměnná má datový typ GameObject. Jako GameObject se považuje jakýkoliv objekt, který je ve scéně Unity, neboli cokoliv, co se dá zařadit do hierarchie. Zároveň je zde použit atribut [SerializeField], který je stejně jako všechny atributy psán vždy v hranatých závorkách. Tento atribut umožňuje viditelnost proměnné v inspektoru, přestože má přístupnost nastavenou na private, neboli soukromou, a tak se proměnná může deklarovat zde. Tento postup ulehčuje mnoho práce, není třeba hledat objekty pomocí kódu, stačí jej přesunout do příslušné kolonky v inspektoru.

Poslední proměnná, neboli třetí v pořadí, je datového typu boolean, představuje pravdu nebo nepravdu (true nebo false). Používá se především v podmínkách či smyčkách.

```
public static Menu instance { get; private set; }
[SerializeField] private GameObject panelPauza;
public bool pauza;
```

Dále jsou tyto proměnné inicializovány, proměnné instance se zakládají v metodě Awake(), protože jich bude dále třeba v metodě Start(), kde musíme mít jistotu, že již budou inicializovány.

```
private void Awake()
{
   instance = this;
}
```

Při spuštění programu je deklarována další proměnná typu bool pauzaPovolena, která zajišťuje, že hráč může v dané situaci hru přerušit. Při inicializaci proměnných se toto hned uplatňuje, kde podmínka zajišťuje zákaz puštění panelu pauzy, pokud buildIndex scény se rovná nule. V našem případě má buildIndex nulu, pokud scéna Menu je aktivní.

```
bool pauzaPovolena;
void Start()
{
   pauza = false;
   if (SceneManager.GetActiveScene().buildIndex == 0)
   {
      pauzaPovolena = false;
   }
   else
   {
      pauzaPovolena = true;
   }
```

```
}
```

Následují veřejné metody pro přepínání scény, ukončení hry, společně s metodami pro aktivaci a deaktivaci prvků. Všechny tyto metody jsou tvořeny s ohledem na zajištění správné funkce spouštění a vypínání pauzy, a to pomocí příkazu Time.timeScale, který určuje rychlost plynutí času. Při nastavení takové hodnoty na nula je čas plně zastaven, jednička symbolizuje základní rychlost a větší čísla ji násobí.

Poslední částí skriptu Menu je metoda OnPauza, která se volá pomocí Input Systému. Řadí se tedy mezi dynamické metody, které při zavolání vyžadují dané informace, které můžeme vidět v kulatých závorkách za jménem metody. V tomto případě se jedná o kontext akce, který je předem nastavený v Input Controls, řadící se mezi položky Input System balíčku.

```
public void OnPauza(InputAction.CallbackContext context)
{
  if (context.phase == InputActionPhase.Started && pauzaPovolena)
  {
    if (panelPauza.activeSelf)
    {
      Time.timeScale = 1f;
    }
    else
    {
      Time.timeScale = 0f;
    }
    panelPauza.SetActive(!panelPauza.activeSelf);
    pauza = !pauza;
  }
}
```

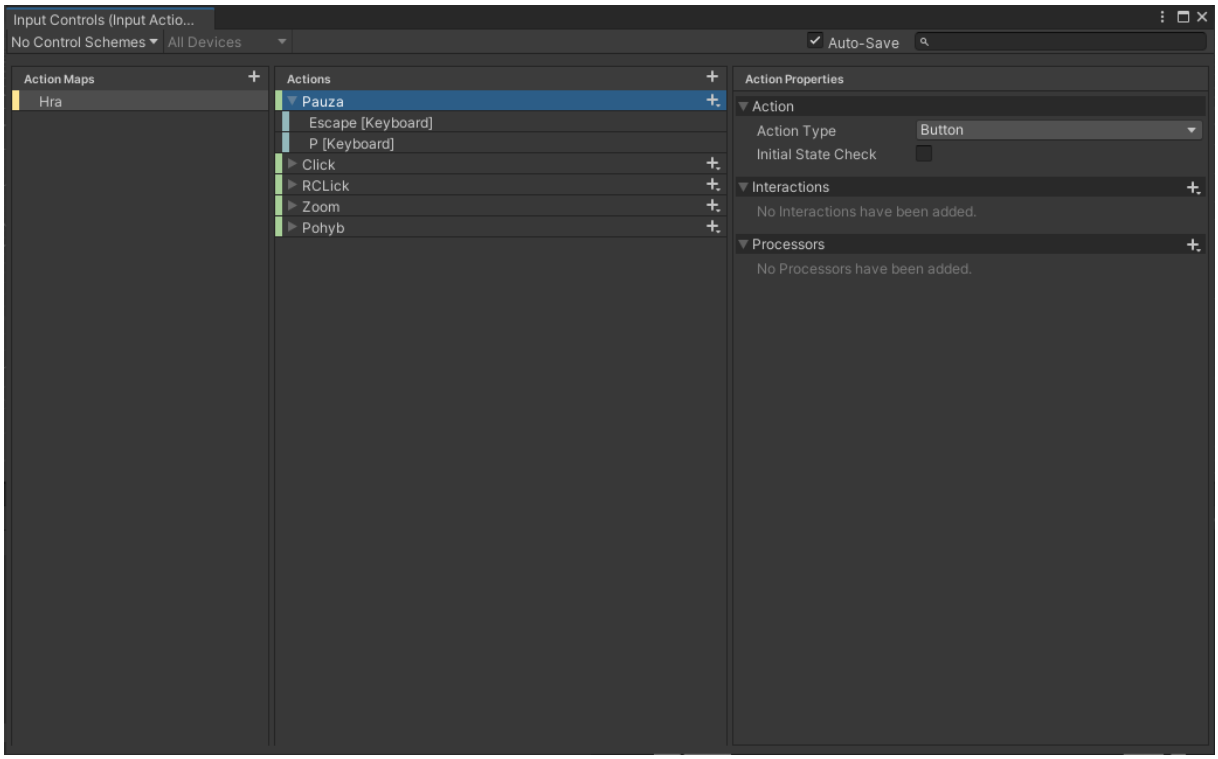
5.3 Input System

Jak již bylo dříve zmíněno, Input System od Unity hraje v tomto projektu velikou roli.

Pro fungování tohoto balíčku je třeba importovat položku Input Controls (na obrázku 8). Po jeho rozkliknutí si můžeme povšimnout, že se skládá z Action Map. Různé action mapy se používají v různých scénách, například pokud chceme, aby v jedné scéně byla pauza pomocí stlačení klávesy P a v druhé pomocí stlačení klávesy Escape, tak místo ověřování buildIndexu v kódu můžeme použít tyto mapy. Input Controls je komponent, který je schopen monitorovat vstupní klávesy dané mapy.

Každá mapa má vlastní akce, některé jsou brány jako tlačítka, jiné jako hodnoty. Pokud je akce brána jako tlačítko, lze pomocí něj zvolit, jaká metoda má proběhnout v momentu, kdy je zmáčknuto. Toto nejen usnadňuje práci s kódem, kdy stačí pouze vytvořit metodu a čekat, než je zavolána, zároveň je šetřena výpočetní technika.

Poslední věc, kterou je třeba určit, jsou vlastnosti akce, především nastavení dané klávesy, pomocí které vše započne.



Obrázek 8: Okno Input Controls

5.4 Save / Load system

Jedna z nejdůležitější a zároveň nejtěžších mechanik na vytvoření je ukládání a načítání pokroku ve hře. Zde je základní koncept skriptů, jak tohoto dosáhnout:

- DataManager – určuje, zda se má hra uložit, či načíst.
- DataPersistance – určuje, jaká data se mají uložit/načíst.
- FileDataHandler – provádí ukládání a načítání souboru.
- GameData – data.

Základní koncept byl převzat od tvůrce Shaped by Rain Studios [1]. Na první pohled složitý, ale při hlubším poznání je velice jednoduchý.

5.4.1 DataManager

Vše začíná ve skriptu DataManager, zde jsou napsány metody jako je SaveGame() či LoadGame(). Má vlastní statickou instanci, pro zjednodušení volání metod a proměnných všech skriptů dříve zmíněných v konceptu.

Základní metoda je NewGame() která při zavolání vytvoří novou instanci proměnné gameData.

```
public void NewGame()
```

```
{
   gameData = new GameData();
}
```

Při požadavku načtení dat je potřeba říct skriptu FileDataHandler, aby načetl všechna data, a ta inicializujeme jako gameData. Jestliže se nepodařilo žádná data načíst, neboli neexistuje soubor, ze kterého by FileDataHandler mohl číst data, zavolá se metoda NewGame().

```
public void LoadGame()
{
   this.gameData = dataHandler.Load();
   if (gameData == null)
   {
      NewGame();
   }
}
```

Při zavření hry či při vrácení se do menu je potřeba data naopak uložit, to provede vytvoření metody SaveGame(), kde příkaz FileDataHandleru uloží data do souboru.

```
public void SaveGame()
{
   dataHandler.Save(gameData);
}
```

Tímto se spravují základní funkce ukládání a načítání, ty ovšem budou muset být v pozdější fázi rozvinuty.

5.4.2 DataPersistance

Skript DataPersistance je originální v jeho jednoduchosti. Jeho základem totiž není základní třída, ale interface. Definuje se sadou metod, vlastností nebo událostí, jejich konkrétní chování se ale nastavuje v každé třídě zvlášť. Interface nelze instancovat, implementuje se jako součást ostatních tříd. Zde je třída Factory, která jej implementuje:

```
public class Factory : MonoBehaviour, DataPersistence
```

Každá třída, která implementuje interface, musí mít všechny jeho metody, a to bezpodmínečně. Pro tento projekt byly zvoleny metody LoadData() a SaveData(). To znamená, že každá třída s daty, která je třeba uložit, obsahuje interface DataPersistance, kde se daná data ukládají a načítají. Načítání je velice jednoduché, stačí jen zavolat metodu LoadData(), ale ukládání má jednu malou zápletku. Při volání metody SaveData() se vykoná nahrazení dat v dané třídě, kde je volána. Je ale potřeba, aby se data uložila v oblasti třídy GameData. Toho se dosáhne pomocí klíčového slova ref. Ref umožňuje metodě změnit obsah proměnné, která do ní byla přidána, místo pouhé práce s daty. Metody jsou tedy určeny takto:

```
public interface DataPersistence
{
   void LoadData(GameData data);
   void SaveData(ref GameData data);
}
```

Nyní je připraven skript, pomocí kterého se určuje, jaká data se mají ukládat. Takže ve skriptu DataManager se musí upravit metody tak, aby se našly všechny skripty, které implementují interface DataPersistance, a dané metody vykonaly. Zavede se tedy metoda, která vrací list DataPersistance skriptů. Ty se najdou tak, že se prohledají všechny skripty, které implementují základní vlastnost. To jest MonoBehavior, nezbytná třída, díky které lze používat metody Start(), Update(), přistupovat ke komponentům objektu a další. K procházení všech skriptů, které mají jako jejich součást MonoBehavior, se použije interface IEnumerable, který umožňuje procházet kolekci pomocí cyklu foreach. V MonoBehavior musí funkce IEnumerable najít interface DataPersistance.

```
private List<DataPersistence> FindAllDataPersistenceObjects()
{
   IEnumerable<DataPersistence> dataPersistenceObjects =
FindObjectsOfType<MonoBehaviour>().OfType<DataPersistence>();
   return new List<DataPersistence>(dataPersistenceObjects);
}
```

Se znalostí objektů obsahujících inteface DataPersistance se upraví SaveGame() a LoadGame() tak, aby se při ukládání našly všechny objekty a aktivovaly jejich metody SaveData(). Dále při načítání se musí aktivovat příslušné metody LoadGame().

```
public void SaveGame()
{
   this.dataPersistenceObjects = FindAllDataPersistenceObjects();
   foreach (DataPersistence dataPersistenceObj in dataPersistenceObjects)
   {
      dataPersistenceObj.SaveData(ref gameData);
   }
   dataHandler.Save(gameData);
}

public void LoadGame()
{
   this.gameData = dataHandler.Load();
   if (gameData == null)
   {
      NewGame();
   }
   dataPersistenceObjects = FindAllDataPersistenceObjects();
   foreach (DataPersistence dataPersistenceObj in dataPersistenceObjects)
   {
      dataPersistenceObj.LoadData(gameData);
   }
}
```

5.4.3 FileDataHandler

Doteď bylo řečeno, že data byla nějak ukládána do souboru, nyní je čas říct, jak přesně tento úkon provést. FileDataHandler je jedna z mála tříd, která má vlastní konstruktor. Konstruktor je speciální metoda třídy, která se zavolá při vytvoření objektu. Používá se k inicializaci nezbytných proměnných nebo k nastavení objektu do výchozího stavu. Funguje tedy na stejném principu jako dříve zmíněná inicializace proměnné gameData v metodě NewGame().

Pro FileDataHandler je podstatná cesta k souboru, kam má data uložit. Dále jak se má jmenovat daný soubor a zda má používat šifrování dat. Aby se data nemohla dát jednoduše modifikovat, je použito šifrování, zajišťuje tedy vyšší bezpečnost.

Takto vypadá konstruktor pro třídu FileDataHandler:

```
public FileDataHandler(string dataDirPath, string dataFileName, bool useEncryption)
{
   this.dataDirPath = dataDirPath;
   this.dataFileName = dataFileName;
   this.useEncryption = useEncryption;
}
```

Metody, které tento skript obsahuje, jsou:

- GameData Load()
- void Save(GameData data)
- string EncryptDecrypt(string data)

Metoda Load() vrací hodnoty ve formě třídy GameData, data jsou uložena ve formátu JSON (JavaScript Object Notation). JSON je jednoduchý formát pro ukládání a výměnu dat, který je čitelný pro lidi i počítače. Používá textový zápis objektů podobný zápisu v JavaScriptu, viz název JSON. Jelikož zápis je textový, lze jej pomocí StreamReaderu a jeho metody ReadToEnd() načíst jako string. Pro použití StreamReaderu se dle jeho notace musí nejprve aktivovat FileStream, neboli kanál, který umožňuje proud dat. Ten nemůže být konstantně aktivní, jinak by zatěžoval výpočetní techniku, proto je třeba jej deaktivovat. Zde je ale použit příkaz using(), který umožňuje mít aktivní jak FileStream tak StreamReader pouze po dobu jejich použití.

Zároveň je třeba dbát na zvýšenou pozornost proti chybám. Na rozdíl od chyb, které se můžou vyskytnout v programu za doby jeho použití, jsou chyby propojené se zpracováním souborů na mnohem větší škále. Je zde proto použita nejen podmínka kontrolující, zda cesta k souboru je validní, ale i konstrukce try-catch, která slouží k ošetření chyb a výjimek.

Přečtená data jsou uložena ve formátu string, nejprve je zkontrolováno, zda je šifrování zapnuto. Pokud ano, musí se zavolat metoda EncryptDecrypt(), která data dešifruje.

Nakonec se pomocí JsonUtility převedou data z formátu string do formátu GameData, která jsou v DataManageru dále používána.

```
public GameData Load()
{
   string fullPath = Path.Combine(dataDirPath, dataFileName);
   GameData loadedData = null;
   if (File.Exists(fullPath))
   {
      try
      {
         string dataToLoad = "";
         using (FileStream stream = new FileStream(fullPath, FileMode.Open))
         {
            using (StreamReader reader = new StreamReader(stream))
            {
               dataToLoad = reader.ReadToEnd();
            }
         }
         if (useEncryption)
         {
            dataToLoad = EncryptDecrypt(dataToLoad);
         }
         loadedData = JsonUtility.FromJson<GameData>(dataToLoad);
      }
      catch (Exception e)
      {
         Debug.LogError("Nastal error při načítání dat ze složky: " + fullPath + "\n" + e);
      }
   }
   return loadedData;
}
```

Pro ukládání dat je použit skoro kompletně stejný kód, jen je zkonstruován naopak.

Poslední věc, o kterou se stará skript FileDataHandler, je šifrování a dešifrování dat pomocí metody EncryptDecrypt(), kde jsou data načítána jako string. Pro dosažení chtěných výsledků je použita operace XOR (eXclusive OR). Patří mezi jednu z nejjednodušších metod pro šifrování a dešifrování.

Princip této operace spočívá v tom, že XOR je bitová operace. Mezi dvěma hodnotami vrací výslednou hodnotu, která má 1 na bitové pozici, kde jsou vstupy různé, a 0 na pozici, kde jsou stejné. Pokud je použit stejný klíč pro šifrování i dešifrování, výsledek bude identický. Proces šifrování i dešifrování je tedy použit stejný, což zjednodušuje celý kód.

V praktické části to znamená, že data[i] je aktuální znak v řetězci data. EncryptionCodeWord je dané klíčové slovo a závorka za ním zaručuje, že v případě, kdy je řetězec dat delší než klíčové slovo, tak se začne číst od znova. Tento proces je opakován, dokud nejsou přečtena veškerá data.

```
private string EncryptDecrypt(string data)
{
   string modifiedData = "";
   for (int i = 0; i < data.Length; i++)
```

```
    {
        modifiedData += (char)(data[i] ^ encryptionCodeWord[i % encryptionCodeWord.Length]);
    }
    return modifiedData;
}
```

5.4.4 GameData

GameData reprezentují veškerá data k uložení v jazyce C#. Pro možnost uložení dat je nejdříve třeba serializace třídy. To znamená, že objekty této třídy mohou být převedeny do formátu, který lze uložit do souboru, poslat přes síť nebo jinak zpracovat. Toho dosáhneme pomocí atributu [System.Serializable], ten ovšem musí být přidán před definicí třídy, jinak nebude rozpoznaná jako serializovatelná.

Tato třída je oproti ostatním velice jednoduchá, obsahuje pouze data, která budou uložena. Vše, co je třeba uložit, se nachází v této třídě. Aby bylo možné vytvářet nová data, musí být v konstruktoru zadány inicializace pro každou proměnnou. Tímto se při zavolání nového konstruktoru všechny proměnné inicializují jako nové, a tak může započít nová hra.

```
[System.Serializable]
public class GameData
{
    public List<FactoryData> factories;

    public GameData()
    {
        factories = new List<FactoryData>();
    }
}
```

Pro lepší modifikaci skriptu bylo usouzeno, že továrny budou mít vlastní třídu, která bude obsahovat pouze důležitá data pro tyto objekty. Takto je vše volně modifikovatelné a může existovat několik továren pouze s jednou třídou GameData. Zároveň je tímto vyřešen i problém vytváření nových továren. Pokud by existovala pouze jedna třída, bylo by třeba manuálně přidávat objekty do listů proměnných, navíc by nebyla žádná záruka, že existuje stejný počet všech proměnných vázajících se na počet továren. S inicializací třídy je nyní třeba zadat i veškerá data, která jsou potřebná pro její správnou funkčnost.

Každá továrna vlastní svůj skript s potřebnými daty, které bude třeba uložit, ale je důležité nezapomenout na data objektu, jako je například pozice či id. Tato data jsou nesmírně důležitá pro načítání továren ze souboru. Vlastní identita továren zaručuje správné přiřazení pozice pro danou továrnu a zároveň hraje důležitou roli při načítání výstupních továren, do kterých budou vyrobené materiály v dané továrně poslány dál.

```
[System.Serializable]
public class FactoryData
{
    public string id;
    public string factoryType;
    public Vector3 factoryPosition;
    public List<string> outputFactoryIds;
```

```
    public Blueprint blueprint;
    public Item inputItem;
    public Item inputItem1;
    public Item outputItem;

    public FactoryData(string id, string factoryType, Vector3 factoryPosition, List<string> outputFactoryIds, Blueprint
blueprint, Item inputItem, Item inputItem1, Item outputItem)
    {
        this.id = id;
        this.factoryType = factoryType;
        this.factoryPosition = factoryPosition;
        this.outputFactoryIds = outputFactoryIds;
        this.blueprint = blueprint;
        this.inputItem = inputItem;
        this.inputItem1 = inputItem1;
        this.outputItem = outputItem;
    }
}
```

5.4.5 Finalizace Save / Load systému a Prefab Manager

Se zhotoveným základním konceptem přichází další problém. Nyní je program schopen načítat a ukládat vlastní data, která jsou mu přiřazena. Továrny je ale třeba instancovat do herního prostředí, neboť jsou to objekty, se kterými se dá nadále zacházet. Pro dosažení tohoto úkonu je třeba nový skript.

Každá továrna tedy musí mít vlastní předlohu objektu, která bude při zapnutí hry načtena. Tato předloha se nazývá prefab, je to taková šablona pro herní objekt, která se dá opakovaně použít ve stejné scéně, pomocí jejího instancování. Každý prefab má na sobě skript Factory, který zaručuje jeho funkčnost továrny. Tento skript má také již od začátku staticky dáno, jaký typ továrny daný prefab je. Díky této funkci je možno pro každou načtenou třídu FactoryData přiřadit vlastní prefab.

Tyto prefaby jsou uloženy v listu ve skriptu, který je zodpovědný za výstavbu továren. Takovéto skripty jsou potřeba jenom jednou na celou scénu, proto mají ve jméně Manager, každý skript s takovýmto jménem je tedy instancován jako instance sama sebe. Pomocí této instance je zaručena snazší přístupnost.

Dále bylo třeba zhotovení metody GetFactoryPrefab(), která vrací GameObject prefab k instancování. K dosažení tohoto cíle načte metoda GetFactoryPrefab() list prefabů ze skriptu FactoryBuildManager zodpovědného za výstavbu továren. Taktéž daný list zkontroluje pro shodu typu továrny k načtení s typem továrny prefabu.

```
public GameObject GetFactoryPrefab(string factoryType)
{
    factoryPrefabs = FactoryBuildManager.instance.factoryPrefabs;
    foreach (GameObject prefab in factoryPrefabs)
    {
        if (prefab.GetComponent<Factory>().GetFactoryType() == factoryType)
        {
            return prefab;
        }
```

```
    }
    return null;
}
```

Nově získaný prefab je nyní třeba instancovat, společně se skriptem Factory. Každý vytvořený skript typu Factory je zaznamenán do předem určené statické třídy StaticClassos, která slouží k zachování informací, které je třeba mít uložené po dobu hraní. Později se tento list bude velice hodit. Navíc, aby mohl skript továrny získat svá potřebná data, je potřeba mu v této metodě taktéž předat jeho unikátní identifikátor.

```
public GameObject SpawnFactory(FactoryData factoryData)
{
    GameObject prefab = GetFactoryPrefab(factoryData.factoryType);
    if (prefab == null)
    {
        Debug.LogError("Factory prefab not found: " + factoryData.factoryType);
        return null;
    }
    GameObject newFactory = Instantiate(prefab, factoryData.factoryPosition, prefab.transform.rotation);
    Factory scriptFactory = newFactory.GetComponent<Factory>();
    StaticClassos.allFactoryScripts.Add(scriptFactory);
    scriptFactory.id = factoryData.id;
    return newFactory;
}
```

Nakonec je upravena metoda LoadGame() ve skriptu DataManager na finální verzi. Pro každou nalezenou instanci třídy FactoryData, která se nachází v datech načtených ze souboru, je vytvořen příslušný objekt.

```
public void LoadGame()
{
    this.gameData = dataHandler.Load();
    if (gameData == null)
    {
        NewGame();
    }
    foreach (FactoryData factoryData in gameData.factories)
    {
        PrefabManager.instance.SpawnFactory(factoryData);
    }
    dataPersistenceObjects = FindAllDataPersistenceObjects();
    foreach (DataPersistence dataPersistenceObj in dataPersistenceObjects)
    {
        dataPersistenceObj.LoadData(gameData);
    }
}
```

5.5 Výstavba továren

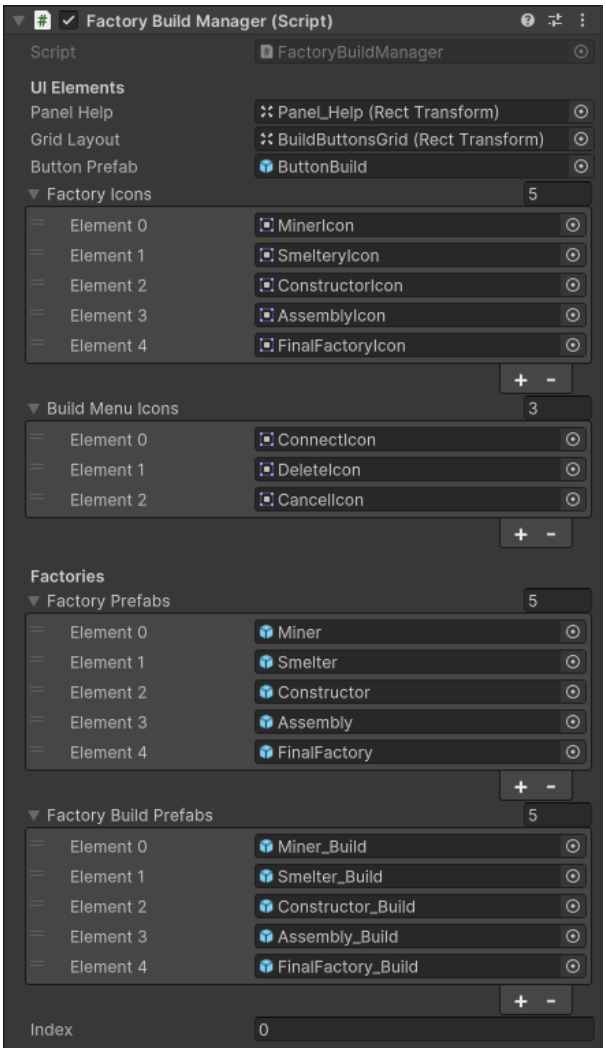
Se znalostmi tvoření továren na začátku hry je nyní třeba tyto informace aplikovat i do průběhu hry s reakčními schopnostmi na hráče. Hráč musí být schopen si vybrat typ továrny, který chce postavit, doplněný krátkým panelem, který obeznámí hráče s vlastnostmi dané továrny. Dále je

potřebný skript zodpovědný za samotnou výstavbu a zajištění možnosti položení. Hráč nesmí být schopen pokládat továrny do sebe, nebo na specificky vyznačená místa, kam nepatří. Je tedy třeba vytvořit:

- Menu pro vybírání továren – build menu.
- Panel nápovědy – panel Help.
- Skript pro pokládání – Click.

5.5.1 Factory Build Manager

Skript Factory Build Manager odpovídá za načtení vybraného prefabu továrny do skriptu Click a zobrazení panelu nápovědy (viz obrázek 9). Pro každý prefab je zhotoven jeho dvojník, který se používá při režimu pokládání. Jestliže je továrnu možno položit, tento prefab se zobrazí jako modrý, pokud ne, zbarví se do červena. Pro rozpoznání prefabů v menu je třeba vytvořit ikony, které znázorňují konkrétní typ továrny, a tím i odpovídající prefab (na obrázku 10). Všechny tyto prvky jsou zadány do listů, kde každý index všech listů znázorňuje konkrétní továrnu.



Obrázek 9: Ukázka skriptu Factory Build Manager v Unity inspektoru

Společně s pokládáním továren jsou zde zařízeny režimy propojení a mazání, které jsou znázorněny pomocí příslušných ikon. Pro vypnutí jakéhokoliv režimu bylo přidáno i tlačítko zrušení, které je znázorněno černým X (na obrázku 10).

Obrázek 10: Znázornění build menu

Tlačítka jsou dynamicky generována podle počtu prefabů v listu FactoryPrefabs společně s přidanými tlačítky pro různé režimy. Tlačítka samotná jsou generována dle předem vytvořeného prefabu tlačítko, který na sobě nese objekt s komponentou obrázku nesoucí jméno „Icon“. Tlačítko vytvořené přes kód takto nic neudělá, je třeba mu přidat Listener(), neboli

naslouchač. Naslouchač určuje, jaká metoda se má provést při zmáčknutí tlačítka. V případě tlačítek zodpovědných za vybrání továren je to metoda SelectIndex(), vyžadující číslo indexu továrny, která se má nastavit jako vybraná pro položení. Při tlačítkách zodpovědných za režimy musely být metody přidány manuálně.

```
private void GenerateButtons()
{
    for (int i = 0; i < factoryPrefabs.Count; i++)
    {
        GameObject obj = Instantiate(buttonPrefab, gridLayout);
        var icon = obj.transform.Find("Icon").GetComponent<Image>();
        var button = obj.GetComponent<Button>();
        icon.sprite = factoryIcons[i];
        int indexCopy = i;
        button.onClick.AddListener(() => SelectIndex(indexCopy));
    }
    for (int i = 0; i < buildMenuIcons.Count; i++)
    {
        GameObject obj = Instantiate(buttonPrefab, gridLayout);
        var icon = obj.transform.Find("Icon").GetComponent<Image>();
        var button = obj.GetComponent<Button>();
        icon.sprite = buildMenuIcons[i];
        if (i == 0)
        {
            button.onClick.AddListener(() => Connect());
        }
        else if (i == 1)
        {
            button.onClick.AddListener(() => Delete());
        }
        else if (i == 2)
        {
            button.onClick.AddListener(() => Cancel());
        }
    }
}
```

Při zavolání metody SelectIndex() musí být zároveň spuštěn panel Help, který na základě indexu vybírá příslušný text do nápovědy. Toto zaručuje, že hráč bude mít možnost se lépe orientovat ve hře.

```
private void SelectIndex(int i)
{
    index = i;
    scriptClick.prefab = factoryPrefabs[index];
    scriptClick.prefabBuild = factoryBuildPrefabs[index];
    PanelHelpActive(true);
}
```

Ostatní metody nastavují prefaby na null, aby nebylo možné pokládat mimo režim stavění a taktéž všechny proměnné, které by mohly vytvářet chyby ve hře.

5.5.2 Skript Click

Skript zodpovědný za veškerou interakci s myší se nazývá Click. Využívá technologii Reycast pro detekci kolize s objektem. Nejdříve je třeba definovat počátek a rotaci paprsku, ta se získá pomocí transformace kurzoru myši. Dále se zjistí, zda byl zasáhnut nějaký objekt v určité vzdálenosti, pokud tak není učiněno, další operace se neprovedou.

```
Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
if (Physics.Raycast(ray, out hit, 20))
{

}
```

Poté se k zasaženému objektu odkazuje pomocí příkazu hit.gameObject. Dále je třeba zjistit, v jakém režimu se hráč zrovna nachází. Pokud má zrovna vybraný prefab k položení, musí být ověřeno, zda se továrna může položit. Toho je možné dosáhnout pomocí zkontrolování všech colliderů.

Collider je komponenta v Unity, která definuje kolizní oblast. Stejně jako paprsky z příkazu raycast reagují pouze na objekty s colliderem. Je třeba zajistit, aby neexistoval žádný collider objektu s tagem Factory.

Tagy slouží k jednoduchému rozpoznání objektů v Unity, pomocí příkazu CompareTag() se dají snadno odlišit v podmínkách.

Nejdříve se tedy zjistí souřadnice polohy, na kterou je zrovna ukazováno pomocí hit.point. Objekt je ale třeba nejdříve posunout o polovinu jeho velikosti nahoru. Při instancování na danou polohu je střed objektu položen na souřadnice zadané do metody Instance(). V tomto případě je žádoucí, aby továrny nebyly zasazeny půlkou v zemi, ale byly na ni postaveny. Proto k příkazu hit.point je přidán trojrozměrný vektor, který zvyšuje proměnou souradnicePolohy o polovinu výšky prefabu. Přičemž polovina výšky prefabu je dosáhnutá pomocí získání MeshRendereru objektu a použitím vlastnosti extents.

```
Renderer rendererPrefab = prefab.GetComponent<Renderer>();
souradnicePolohy = hit.point += new Vector3(0, rendererPrefab.bounds.extents.y, 0);
```

Se znalostí souřadnice polohy se nyní pomocí příkazu Physics.OverLapbox() zjistí všechny collidery nacházející se ve vzdálenosti a každý porovná s podmínkami, které nejsou přijatelné pro položení objektu. Pokud je nějaká podmínka platná, nastaví hodnotu colliderIsFactory na true, čímž zamezí položení továrny.

```
Collider[] colliders = Physics.OverlapBox(souradnicePolohy, new Vector3(rendererPrefab.bounds.extents.x,
rendererPrefab.bounds. extents.y, rendererPrefab.bounds. extents.z));
bool colliderIsFactory = false;
foreach (Collider collider in colliders)
{
   if (collider.gameObject == predloha)
```

```
    {
        continue;
    }
    if (collider.CompareTag("Factory") || collider.CompareTag("Miner") || predloha.transform.position.y > 1)
    {
        colliderIsFactory = true;
        break;
    }
}
if (colliderIsFactory)
{
    placeble = false;
}
```

Jestliže továrna má přiřazený tag Miner, jedná se o těžební stroj, který je zodpovědný za vytváření materiálů místo jejich procesu. Přichází navíc podmínka, zda je položen na příslušnou surovinu, od ní dostane plánek (blueprint), který mu určuje, jakou surovinu je třeba generovat a v jakém intervalu.

Pokud není vybrán žádný prefab továrny, program zjišťuje, zda má hráč kurzor na továrně, pokud ano, nastaví proměnou mouseOverFactory na true.

```
if (hit.collider.CompareTag("Factory") || hit.collider.CompareTag("Miner"))
{
    mouseOverFactory = true;
}
```

Obě tyto části kódu jsou uloženy v metodě Update(). Jejich výstupy jsou boolean hodnoty, které se porovnávají v metodě OnClick(), která se volá, když hráč zmáčkne levé tlačítko myši. Jestliže je prefab možno položit, spustí se podobná metoda jako v Prefab Manageru, metoda SpawnFactory(). Naopak pokud není vybrán prefab, zkontroluje se, zda hráč nechce továrny propojit, neboli jestli není režim propojení zapnutý. Každý režim má vlastní metodu, v případě režimu propojení je to metoda ConnectionHandler(). Ta si ukládá první nakliknutou továrnu a přidá do jejího listu outputFactories, který označuje všechny továrny, do kterých se má posílat materiál, a druhou nakliknutou továrnu. Jestliže je ale druhá továrna již součástí tohoto listu, program musí celý list prohledat odzadu a příslušnou továrnu z něj vymazat. Prohledání odzadu je důležitě, protože v momentu, kdy je odstraněný objekt z listu, celý list se posune o jedno pole dozadu, aby zaplnil díru, která tak nastane. Při přičítání by se na index odstraněného objektu přesunula další továrna a ta by byla přeskočena.

```
private void ConnectionHandler()
{
    if (factoryConnect == null)
    {
        factoryConnect = hit.collider.gameObject.GetComponent<Factory>();
    }
    else if (factoryConnect != hit.collider.gameObject.GetComponent<Factory>() &&
!hit.collider.CompareTag("Miner"))
    {
        for (int i = factoryConnect.outputFactories.Count - 1; i >= 0 ; i--)
        {
            if (factoryConnect.outputFactories[i] == hit.collider.gameObject.GetComponent<Factory>())
```

```
        {
            factoryConnect.outputFactories.RemoveAt(i);
            factoryConnect = null;
            return;
        }
    }
    factoryConnect.outputFactories.Add(hit.collider.gameObject.GetComponent<Factory>());
    factoryConnect = null;
  }
}
```

V případě, že režim propojení není vybrán, otestuje se režim odstranění. Ten, pokud je objekt jedna z továren, jej zničí.

```
private void DeleteHandler()
{
  Destroy(hit.collider.gameObject);
}
```

5.6 Princip funkce továren

Každý objekt v Unity, který je označen tagem Factory, na sobě nese skript se stejným názvem. Skript Factory zodpovídá za veškeré dění uvnitř továrny. Hlavními metodami jsou:

- ProcessHandler() – kontroluje vstupy, pokud jsou plné, spustí metodu ProcessItem().
- ProcessItem() – zpracuje materiál.
- ItemTransferHandler() – kontroluje výstup a odesílá jej do další továrny.
- UpdateLineRenderers() – graficky znázorňuje propojení s výstupními továrnami.

5.6.1 Základní proměnné

Navíc je důležité zmínit, že zde se provádí metody SaveData() a LoadData() z dříve zmíněného Save / Load systému. Ty pracují se základními daty potřebných pro provoz továren. Patří mezi ně například globálně unikátní identita (GUID) proměnné id či samotné materiály továrny nazvané Item.

```
public string id;
public string factoryType;
public Blueprint blueprint;
public Item inputItem;
public Item inputItem1;
public Item outputItem;
public List<Factory> outputFactories;
public List<string> outputFactoryIds;
```

5.6.2 ProcessHandler()

Metoda ProcessHandler, uložená v metodě Update(), s sebou nese velice jednoduchý kód, který určuje stav továrny. Pokud je přítomen vstupní materiál a je místo jej zpracovat, spustí metodu ProcessItem().

```
if (inputItem == null)
{
   stav = "Čekám na materiál";
}
else if (outputItem == null)
{
   processing = true;
}
else
{
   stav = "Úložiště je plné";
}
if (processing)
{
   ProcessItem();
}
```

5.6.3 ProcessItem()

Metoda ProcessItem je zodpovědná za zpracování materiálu nižší hodnoty na materiál vyšší hodnoty. Tento proces trvá nějaký čas, pro každý plánek je kompletně jiný. Je třeba využít Time.deltaTime pro zápis uplynulého času do proměnné časovač typu float. Jestliže tato proměnná bude větší než čas procesu uložený v plánku, tak dojde k vymazání vstupního materiálu, který je nahrazen materiálem výstupním ze stejného plánku. Nakonec je vše uvedeno do počátečního stavu.

```
stav = "Pracuji";
casovac += Time.deltaTime;
if (casovac < blueprint.casProcesu)
{
   return;
}
inputItem = null;
outputItem = blueprint.outputItem;
processing = false;
casovac = 0;
```

5.6.4 ItemTransferHandler()

S možností výskytu více výstupních továren bylo nejdříve třeba zhotovit kód ItemTransferHandler určující, do jaké továrny má výstupní materiál putovat. Tohoto bylo dosaženo pomocí ukládání posledního indexu listu outputFactories. Jestliže je továrna na místě indexu plná či nemá vybrán plánek, index se posune pomocí cyklu for. Pokud jsou obě podmínky splněny, je třeba zajistit, zda je daný materiál chtěný následující továrnou. V případě že je tomu tak, je materiál předán a metoda ukončena.

```
if (lastFactoryIndex == outputFactories.Count - 1)
{
   lastFactoryIndex = 0;
}
for (int i = lastFactoryIndex; i < outputFactories.Count; i++)
{
   lastFactoryIndex = i;
   if (outputItem == null || outputFactories[i]?.blueprint == null)
   {
      continue;
   }
   if (outputFactories[i].blueprint.inputItem == blueprint.outputItem && outputFactories[i].GetInputItem() == null)
   {
      outputFactories[i].GiveOuputItemToInput(outputItem);
      outputItem = null;
      return;
   }
}
```

5.6.5 UpdateLineRenderers()

Poslední službou ve skriptu Factory je metoda UpdateLineRenderers(), zodpovědná za vykreslování čar s downstream závislostí. Metoda je volána za podmínky, že je změněn počet továren v listu outputFactories. V ten moment vymaže veškeré stávající vykreslovače čar, neboli lineRenderery a vytvoří nové.

```
foreach (LineRenderer lineRenderer in lineRenderers)
{
   Destroy(lineRenderer.gameObject);
}
lineRenderers.Clear();
foreach (Factory factory in outputFactories)
{
   GameObject lineRendererObject = new GameObject("LineRenderer");
   LineRenderer lineRenderer = lineRendererObject.AddComponent<LineRenderer>();
   lineRendererObject.transform.SetParent(transform);
   lineRenderer.SetPosition(0, transform.position);
   lineRenderer.SetPosition(1, factory.gameObject.transform.position);
   lineRenderers.Add(lineRenderer);
}
```

5.6.6 Návaznost na uživatelské rozhraní továren

Jelikož uživatelské rozhraní továren pouze zobrazuje informace o nakliknuté továrně, je třeba mu o ní poskytnout informace. Jedná se například o přítomnost materiálů, které jsou zasílány v metodě Update(). Pro optimalizaci výpočetní techniky se tato část kódu se vykoná pouze tehdy, pokud je daná továrna nakliknutá.

```
if (selected)
{
   FactoryUI.instance.UpdateInputItem(inputItem);
   FactoryUI.instance.UpdateOutputItem(outputItem);
}
```

5.7 Uživatelské rozhraní továren

Uživatelské rozhraní továren slouží jako sběrnice všech panelů, obrázků či tlačítek, které jsou součástí rozhraní továrny. Při kliknutí na továrnu se aktivuje metoda LoadUI(GameObject factory) potřebující objekt scény.

5.7.1 LoadUI()

Metoda LoadUI na vybraném objektu najde skript Factory, zobrazí jeho počáteční stav a nastaví proměnnou selected na true. Tímto se začnou zobrazovat veškeré údaje o dané továrně.

```
public void LoadUI(GameObject f)
{
   factory = f;
   scriptFactory = factory.GetComponent<Factory>();
   scriptFactory.Selected(true);
   factoryType = scriptFactory.GetFactoryType();
   nazev.text = factoryType;
   panelInput.SetActive(true);
   inputImage = panelInput.transform.Find("Input").GetComponent<Image>();
}
```

5.7.2 Výběr plánku

Poslední částí je výběr plánku, ten funguje na stejné bázi jako výstavba továren. Předem jsou instancovány prefaby tlačítek, každé nesoucí vlastní plánek. Při kliku se provede metoda BlueprintChange(Blueprint b), která plánek předá skriptu Factory a nastaví příslušné obrázky v rozhraní továren.

```
public void BlueprintChange(Blueprint b)
{
   blueprint = b;
   scriptFactory.GiveBlueprint(b);
   inputImage.sprite = blueprint.inputItem.ikona;
   outputImage.sprite = blueprint.outputItem.ikona;
}
```

6 ZÁVĚR

Hlavním cílem této práce bylo naprogramování videohry se základními mechaniky tzv. žánru Factory Builder. Tohoto cíle bylo úspěšně dosaženo. Přes Profiler v Unity bylo ověřeno, že hra běží na stabilních 200 snímcích za sekundu a zabírá okolo 1 GB RAM.

Původní návrh práce obsahoval výběr úrovní, které hráče budou učit o daných továrnách. Ze začátku se toto jevilo jako skvělý nápad, při delším zamyšlení je ale velice nepraktický. Nutnost přehrání všech takovýchto úrovní před začátkem samotného hraní je na první pohled odrážející. Dalším faktorem je potřeba přepínání scén. V takto malém demu doba načítání skoro nepatrná, ovšem tento projekt je dělán s co největší podporou budoucí expanze. Představa neustálého se vracení zpět do menu a hledání správné úrovně, která vysvětlí daný problém, je při nejlepším nudná. Proto byl tento bod nahrazen malým panelem v pravé horní části obrazovky při výstavbě továren, kde je velice stručně popsáno, co daná továrna dělá.

Největší problém při tvorbě této práce byl rozhodně systém propojování továren, jejich ukládání a načítání. Nejprve byl zhotoven prototyp, který byl schopen mít pouze jednu výstupní továrnu. Této továrně byla při serializaci dat automaticky přidělena výchozí Unity identita. Díky této identitě bylo možné zpátky načíst objekt ze souboru do proměnné bez problémů, to se však změnilo nahrazením dané proměnné za list. Výchozí identity přidělené od Unity nyní byly nedostačující, do listu se načítali jako null. Pro vyřešení tohoto problému bylo nutno každé továrně vytvořit místo výchozí identity vlastní globálně unikátní identitu. Bylo tedy zapotřebí naprogramovat metodu, která by byla schopna přiřadit každé identitě uložené v listu výstupních továren její vlastní objekt a naopak. Zároveň musel být při instanci továren implementován nový kód, zodpovědný za tvorbu listu všech aktuálně nacházejících se továren ve scéně. Z tohoto listu jsou poté porovnávány továrny a jejich identity. Vyřešením tohoto problému vznikla jedna z částí práce, na kterou jsem velice pyšný.

Teoretická část práce popisuje vlastnosti, které hráč získává hraním hry. I když nejsou ničím nějak tajné či speciální, stále se jedná o vysoce hodnotné dovednosti. Existuje spousta vysoce placených pozic v oboru softwarového inženýrství, kde se dají uplatnit. Vše díky hraní prosté videohry.

Nakonec bych tento projekt považoval za úspěšný. Jsou oblasti, kde by se dalo dále vyvíjet, výhradně kontent. Jelikož je grafické znázornění vykreslováno pomocí komponenty Line Renderer, který je náročný na výpočetní techniku, a spousty kódu se odehrávají v metodě Update(), je zde prostor pro zlepšení v oblasti optimalizace. Avšak toto nepopírá fakt, že jsem s výsledným produktem velice spokojen a mohu jej nazvat svým dosavadním největším úspěchem.

7 POUŽITÁ LITERATURA

[1] YOUTUBE. How to make a Save & Load System in Unity. Online. 2022. Dostupné z: https://www.youtube.com/watch?v=aUi9aijvpgs&t=1211s. [cit. 2025-01-12].

[2] YOUTUBE. Factorio teaches you software engineering, seriously. Online. 2023. Dostupné z: https://www.youtube.com/watch?v=vPdUjLqC15Q&list=WL&index=2. [cit. 2025-02-21].

[3] TARACHENKO, Andrei. Death By a Thousand Microservices. Online. Renegate Otter. 2023. Dostupné také z: https://renegadeotter.com/2023/09/10/death-by-a-thousand-microservices. [cit. 2025-02-24].

[4] UNITY TECHNOLOGIES. Unity. Online. Unity Documentation. 2025. Dostupné z: https://docs.unity3d.com/Manual/index.html. [cit. 2025-03-3].

8 SEZNAM OBRÁZKŮ