OSTRAVSKÁ UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA
KATEDRA INFORMATIKY A POČÍTAČŮ

Dokumentace a implementace funkcí, zlepšujících použitelnost softwaru

DIPLOMOVÁ PRÁCE

Autor práce: Jan Kunetka
Vedoucí práce: Mgr. Robert Jarušek, Ph.D.

2025

UNIVERSITY OF OSTRAVA
FACULTY OF SCIENCE
DEPARTMENT OF COMPUTERS AND INFORMATICS

Documentation and implementation of functions that improve the usability of the software

DIPLOMA THESIS

Author: Jan Kunetka
Supervisor: Mgr. Robert Jarušek, Ph.D.

2025

OSTRAVSKÁ UNIVERZITA
Přírodovědecká fakulta
Akademický rok: 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Jan KUNETKA**
Osobní číslo:	**P22157**
Studijní program:	**N0613A140036 Aplikovaná informatika**
Specializace:	**Informační systémy**
Téma práce:	**Dokumentace a implementace funkcí, zlepšujících použitelnost softwaru**
Zadávající katedra:	**Katedra informatiky a počítačů**

Zásady pro vypracování

Cílem diplomové práce je úspěšné aplikování vybraných funkcí, zlepšující kvalitu použitelnosti softwaru z pohledu uživatele, na konkrétní desktopové aplikaci, vyvinuté v enginu Unity.

1. Úvod do problematiky
2. Návrh použitých funkcí
3. Implementace funkcí
4. Shrnutí a závěr

Rozsah pracovní zprávy:	**40 – 80 stran**
Rozsah grafických prací:	
Forma zpracování diplomové práce:	**tištěná**

Seznam doporučené literatury:

1. Unity Documentation: Unity User Manual 2021.3 (LTS) [online]. San Francisco: Unity Technologies, c2021 [cit. 2022-10-12]. Dostupné z: https://docs.unity3d.com/Manual/index.html
2. Unity Learn: Welcome to Unity Learn [online]. San Francisco: Unity Technologies, c2022 [cit. 2022-10-12]. Dostupné z: https://learn.unity.com/
3. A Comparative Research on Usability and User Experience of User Interface Design Software. International Journal of Advanced Computer Science and Applications. 2022, 13(8), 9. Dostupné z: doi:10.14569/IJACSA.2022.0130804
4. A Systematic Mapping Study of Software Usability Studies. International Journal of Advanced Computer Science and Applications. 2021, 12(9), 14. ISSN 21565570, 2158107X. Dostupné z: doi:10.14569/IJACSA.2021.0120927
5. A systematic review of software usability studies. International Journal of Information Technology. 2017, 24. ISSN 2511-2104, 2511-2112. Dostupné z: doi:10.1007/s41870-017-0048-1

Vedoucí diplomové práce: **Mgr. Robert Jarušek, Ph.D.**
Katedra informatiky a počítačů

Datum zadání diplomové práce: **24. listopadu 2022**
Termín odevzdání diplomové práce: **24. dubna 2024**

Mgr. Robert Jarušek, Ph.D.
vedoucí diplomové práce

doc. RNDr. PaedDr. Hashim Habiballa, PhD., Ph.D.
vedoucí katedry

doc. RNDr. Martin Kotyrba, Ph.D.
garant studijního programu

V Ostravě dne 20. listopadu 2022

ABSTRAKT

Cílem diplomové práce je implementace a zdokumentování postupu vývoje několika vybraných funkcí, které zlepšují uživatelskou použitelnost softwaru. Jedná se o funkce, jejichž úkolem je zpříjemnit uživateli používání zvoleného programu, například zkrácením času nutného k vykonání akcí, nebo odstraněním repetitivních činností.

V rámci práce dojde k bližšímu seznámení s pojmem použitelnost ve vývoji softwaru a proč se jím vůbec zabývat. Potom se práce přesune na hlavní náplň, kterou bude návrh a implementace několika funkcí použitelnosti do vybraného příkladového editačního programu. Vybrány budou funkce, které jsou dnes často zastoupené ve většině editačních programů.

Ohledně příkladového programu, ten bude převzat z mé bakalářské práce [1], jež se zaměřovala na jeho sestavení. Bude se jednat o desktopovou hru vyvinutou v herním enginu Unity. Jedná se o aplikaci, složenou z herní a editační části, kde v editační části je uživatel schopen vytvářet nový obsah, který bude herní část využívat. Tato aplikace dobře poslouží pro demonstraci vytvořených řešení funkcí. Obsahuje totiž několik různorodých editorů pro úpravu rozdílného obsahu. Všechny řešené funkce budou mezi editory sdíleny, na čemž bude možné dobře ukázat univerzálnost jejich implementace.

Hlavním přínosem této práce budou vytvořené knihovny implementovaných funkcí, které si bude moci programátor stáhnout za účelem snadného zavedení funkcí do vlastního projektu. K tomu ke každé knihovně bude přiložen jako zdroj detailních informací tento dokument, kde by sloužil jako informační doplněk pro pochopení fungování implementovaných funkcí.

Klíčová slova: použitelnost, UI, Unity, C#, funkce, undo, redo, zkratky, vstupní vazby

ABSTRACT

The goal of this master thesis is to implement and document several selected features, which improve the usability of software. These features serve to make the use of programs more pleasant for the user, for example by reducing the time required to perform actions or by eliminating repetitive activities.

The thesis will provide a closer look at the concept of usability in software development, and why it matters. The work will then move onto designing and implementing several usability features into a selected editing program as an example. The features that will be selected are often present in most programs of this type.

Regarding the selected program, it will be taken from my bachelor's thesis [1], which focused on its development. It was a desktop game, developed in the Unity game engine. The application is split into a game part and an editing part, where in the editing part the user can create new content for the game. This kind of application will serve perfectly for demonstrating the implemented solutions, because it contains several different editors for editing different content. All implemented features will be shared between the editors, which will make it a good example of showing how universal the implementation is.

The main contribution of this work will be the creation and publication of libraries, each representing an implemented feature that the programmer will be able to download and easily introduce into their own project. In addition, each library will be accompanied by this document as a source of detailed information, with the goal of serving as an information supplement for understanding the how the implemented features work.

Keywords: usability, UI, Unity, C#, features, undo, redo, shortcuts, input bindings

ČESTNÉ PROHLÁŠENÍ

Já, níže podepsaný student, tímto čestně prohlašuji, že text mnou odevzdané závěrečné práce v písemné podobě je totožný s textem závěrečné práce vloženým v databázi DIPL2.

Ostrava dne

……
podpis studenta

PODĚKOVÁNÍ

Rád bych zde poděkoval svému vedoucímu Mgr. Robertu Jaruškovi, Ph.D. za jeho rady a pomoc s dotažením práce do cíle. Dále bych rád poděkoval Mgr. Evě Janíkové za její pomoc s pravopisnou korekturou a nakonec taky celé své rodině za jejich nehynoucí trpělivost.

Prohlašuji, že předložená práce je mým původním autorským dílem, které jsem vypracoval samostatně. Veškerou literaturu a další zdroje, z nichž jsem při zpracování čerpal, v práci řádně cituji a jsou uvedeny v seznamu použité literatury.

V Ostravě dne . . . . . .

. . . . . .

(podpis)

OBSAH

ÚVOD

Unity je populární program s bohatou komunitou, která zásobuje internet velkou spoustou kvalitní dokumentace, avšak i pro tento program se najdou takové systémy pokročilejšího rázu, kterým se velké pozornosti nedostává.

Převážně se jedná o vestavěné editory, kdy uživatel může tvořit nový obsah pro hru ve stejné aplikaci (více v mé bakalářské práci [1]). I když důležitým funkcím tohoto odnoží se nějaké pozornosti dostává (editory map, pracování s mřížkou, apod), funkcionalitě, která zlepšuje použitelnost těchto editorů se komunita věnuje pouze v limitovaném množství.

Mou snahou bude obohatit tuto oblast vývojem a nasazením několik těchto typů funkcí na větší projekt. Zároveň jejich fungování zdokumentovat v této práci a výsledky zveřejnit jako open-source na internetu.

CÍLE PRÁCE

Cílem diplomové práce je zdokumentování implementace několika funkcí, zlepšující uživatelskou použitelnost softwaru. Tato dokumentace a implementace budou následovně volně zveřejněny se záměrem usnadnit dalším programátorům práci při jejich zavádění do dalších projektů.

Dílčí cíle práce

1. **Úvod do problematiky a analýza současného stavu**

V této kapitole bude čtenář nejdříve uveden do problematiky práce. Dále bude nahlédnuto na nynější stav dokumentace vybraných funkcí na webu a také oblast akademických prací, které se tímto nebo podobným tématem již zabíraly.

2. **Návrh řešení**

Tahle kapitola bude věnována funkcím vybraným pro implementaci. Bude vysvětlena jejich podstata a proč byly právě pro tuto práci zvoleny. Potom bude následovat představení příkladového programu, do kterého se funkce nasadí.

3. **Implementace navrženého řešení**

Od této části již začíná vlastní práce. Bude se věnovat způsobu implementace vybraných funkcí, a to od základních vlastností až po složitější problémy. Speciální pozornost bude věnována čitelnosti a znovupoužitelnosti kódu pro snadné využití výsledků této práce v jiných projektech.

4. **Shrnutí výsledků a závěr**

Nakonec dojde ke shrnutí dosažených výsledků. Nahlédne se na výsledné podoby implementovaných funkcí a vytypuje se jejich výsledná funkcionalita a jaké jsou jejich limitace. Následovat bude závěr a možnosti pokračování.

1 ÚVOD DO TÉMATIKY

Na začátku je vhodné představit a alespoň povrchně nahlédnout do oblastí, kterými se práce bude zabývat v praktické části. Tato kapitola se podívá na rozdíly mezi použitelností a užitečností a na význam pojmu *funkce, zlepšující použitelnost*, jenž se objevil v úvodu. Také zde bude stručně představen program Unity, do kterého je práce situována, a bude i vytyčena podstata návrhových vzorů v programování.

1.1 Použitelnost

Použitelnost je atribut kvality, který měří, jak snadné je pro uživatele pracovat s uživatelským rozhraním. V prostředí softwaru se jedná o jeden z nejzásadnějších faktorů udržitelnosti uživatelů [2]. Pokud má uživatel problém se v programu, či webové stránce zorientovat či něco najít, je vyšší šance, že spíše začne hledat alternativní řešení, než aby program dále využíval [3].

Použitelnost se skládá z **pěti** různých kritérií, kterým by se měla věnovat pozornost při navrhování produktů. Pokud se při designu softwaru na tato kritéria dbá, dá se výsledný program označit jako použitelný [4].

Těmito kritérii jsou:

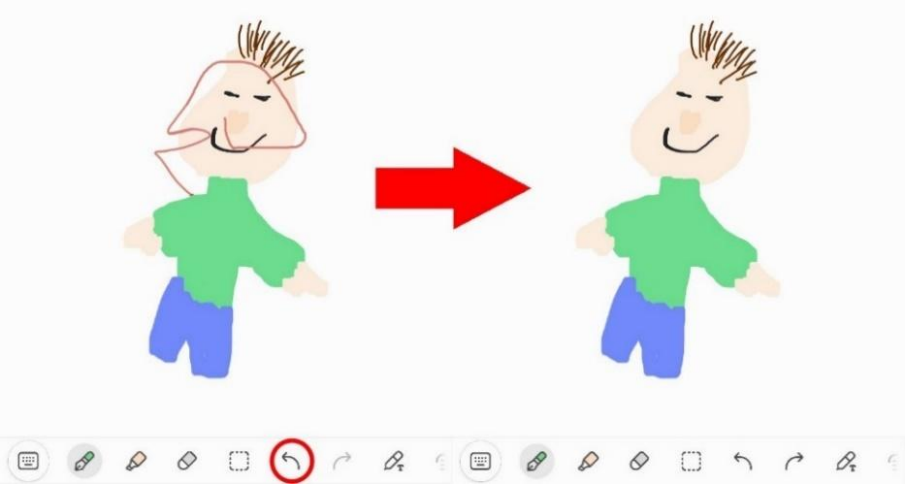
- **Efektivnost** – S jakou přesností dokážou uživatelé splnit své úkoly. Jde o snahu vhodné komunikace programu s uživatelem, aby se snížila šance vzniku chyb [5]. Vhodným příkladem je třeba upozornění designéra, že v herním editoru zapomněl nastavit startovní pozici hráče. Obrázek 1 ukazuje možné řešení takovéto situace.

Obrázek 1 - Po kliknutí na Uložit se objevila připomínka, že designer nenastavil pozici vstupu a výstupu.

- **Účinnost** – Jak rychle dokáže uživatel své úkoly dokončit, neboli kolik kliků potřebuje pro splnění úlohy. Dobrým příkladem funkce, zlepšující účinnost jsou *klávesové zkratky*, jež jsou schopny nahradit několik kliků v menu stisknutím jedné kombinace tlačítek [3].
- **Přitažlivost** – Jak vhodný a přehledný je grafický vzhled a rozložení programu [3]. Dobrý příkladem tohoto kritéria je webová stránka google.com (obrázek 2). Jejím primárním cílem je umožnit uživatelům vyhledávat informace na internetu, což reflektuje minimalistický design stránky, obsahující pouze to nejnutnější.

Obrázek 2 - Vzhled webové stránky google.com

- **Tolerance chyb** – Do jaké míry je software navržený, co se týče práce s chybami. Tomuto kritériu jde především o minimalizaci výskytu uživatelských chyb a rychlého usměrnění uživatele v případě, že se nějaká chyba objeví [5]. Jednou z nejčastěji se vyskytujících funkcí, pomáhající s tolerancí chyb, bývá *Undo/Redo systém* (vrátit zpět/znovu). Ten umožňuje uživatelům rychle vrátit nechtěnou akci. Například, když umělec v grafickém editoru špatně nakreslí čáru. Místo toho, aby ji musel pomalu smazat, může prostě stisknout tlačítko a čára tak zmizí (zobrazeno vizuálně na obrázku 3).



Obrázek 3 - Undo v grafickém editoru

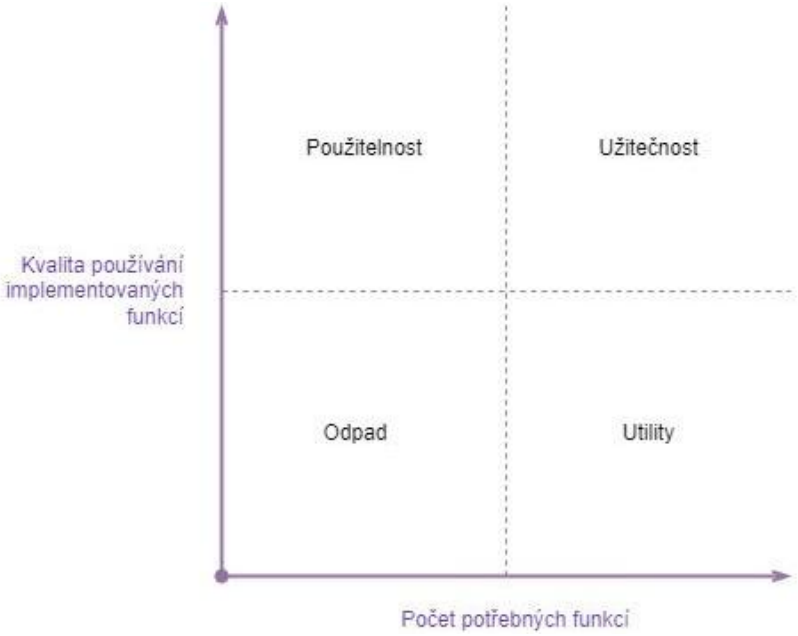
- **Lehká adaptace** – Jak snadno dokážou noví uživatelé splnit své úkoly a o kolik snadněji je dokážou splnit během dalších použití programu [4]. Dobrým způsobem, jak zjednodušit pochopení programu pro uživatele, je vhodná ikonografie. Například tlačítko pro smazání dat bude označeno ikonou odpadkového koše. Ten v materiálním světě označuje místo, kam jdou nepotřebné věci, tudíž dává smysl být použit jako ikona [5].

1.1.1 Porovnání s utility

Dalším atributem kvality je **utility**. Ten zde zmiňuji, protože má s použitelností velmi silný vztah a je třeba ujasnit, jestli se projekt nebude náhodou blížit spíše k němu.

Kdy použitelnost se zaměřuje na to, jak dobře se dá s daným softwarem pracovat, utility měří, jestli software vůbec **obsahuje funkce**, které uživatel potřebuje [3][2]. Například, když IDE chybí vestavěný našeptávač, ale má dobře fungující hlasové ovládání, které však uživatelé nebudou používat. Toto IDE bude tedy hodně ztrácet v měření utility, protože obsahuje pro uživatele nepotřebnou funkci a k tomu mu chybí důležitá funkce jako je našeptávač.

Použitelnost a utility jsou pro software stejně důležité a dohromady jej dělají **užitečným** [2]. Tento vztah je vizualizovaný pomocí grafu na obrázku 4.



Obrázek 4 - Vztah použitelnosti a utility

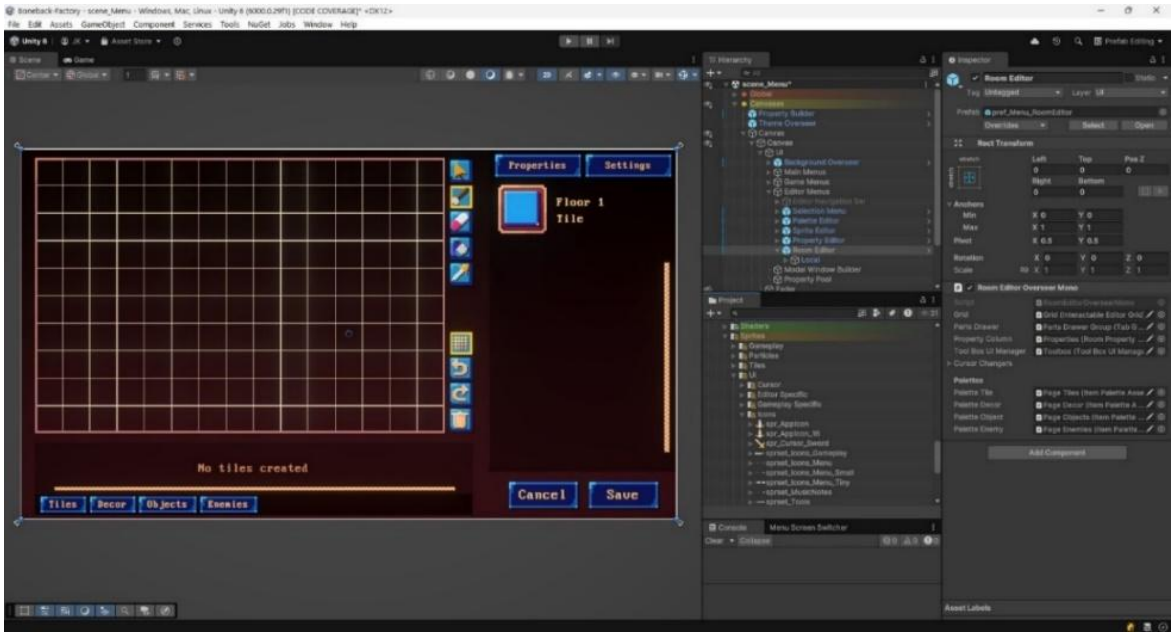
Je možné vidět, že čím více pro uživatele použitelných funkcí software obsahuje, tím má lepší utility. Zároveň čím lépe se uživateli s těmito funkcemi pracuje, tím roste použitelnost. Ideální stav je tehdy, kdy software obsahuje spoustu dobře použitelných funkcí, které uživatel využije. V takovém případě se mluví o **užitečném softwaru**.

1.1.2 Vztah k projektu

Co se vztahu těchto vybraných atributů kvality k projektu týče, snahou bude do něho přidat nové funkce, které ulehčí práci s jinými, již implementovanými funkcemi. Z pohledu atributů kvality se tedy bude tato diplomová práce **zaměřovat na utility** (přidávání funkcí), avšak výsledkem by mělo být navýšení použitelnosti jiných, již existujících funkcí. Oba atributy jsou tedy pro projekt relevantní.

1.2 Unity Game Engine

Unity je nástroj, pro který budou funkce v implementační části vyvíjeny. Jedná se o, v dnešní době, jeden z nejpopulárnějších programů pro vývoj hlavně 2D a 3D her i klasických programů. Kromě počítačových her podporuje také tvorbu pro herní konzole, web, mobilní telefony a virtuální i agmentovanou realitu [6]. Na obrázku 5 je možné vidět dnešní podobu tohoto editoru.



Obrázek 5 – Vzhled Unity editoru

Tato práce již počítá s tím, že čtenář do jisté míry zná a *umí pracovat s tímto programem*. Pro její pochopení je nutné znát základní koncepty, mezi něž patří například scény, objekty, prefaby a komponenty. Tyto principy byly již určitým způsobem popsány v mé bakalářské práci [1] (kapitola 2) a tudíž se tato diplomová práce zde o základech dále rozepisovat nebude. Bakalářská práce také obsahuje odkazy na další vědecké papíry, věnující se základům v Unity, proto pokud čtenář potřebuje rychle doplnit své znalosti, může se poohlédnout tam.

V následujících podkapitolách dojde jen ke stručnému popisu těch částí editoru, které budou relevantní pro implementační část.

1.2.1 UI Systém

Obě funkce, které budou pro práci implementovány, potřebují nějakým způsobem interagovat s uživatelským rozhraním. V tomto roce (2025) má Unity 2 oficiální systémy: uGUI a UI Toolkit. Co se rozdílů týče, tak:

- **uGUI** – Je starší řešení, které využívá objekty a komponenty pro sestavení uživatelského rozhraní [7]. Jeho workflow je dost specifický pro vývoj v Unity, avšak má vysokou podporu a pokrývá více méně veškeré potřebné situace.
- **UI Toolkit** – Je nový systém, který má v budoucí době uGUI nahradit. Využívá moderní způsob navrhování UI, dost podobný webovým technologiím. Dnes však

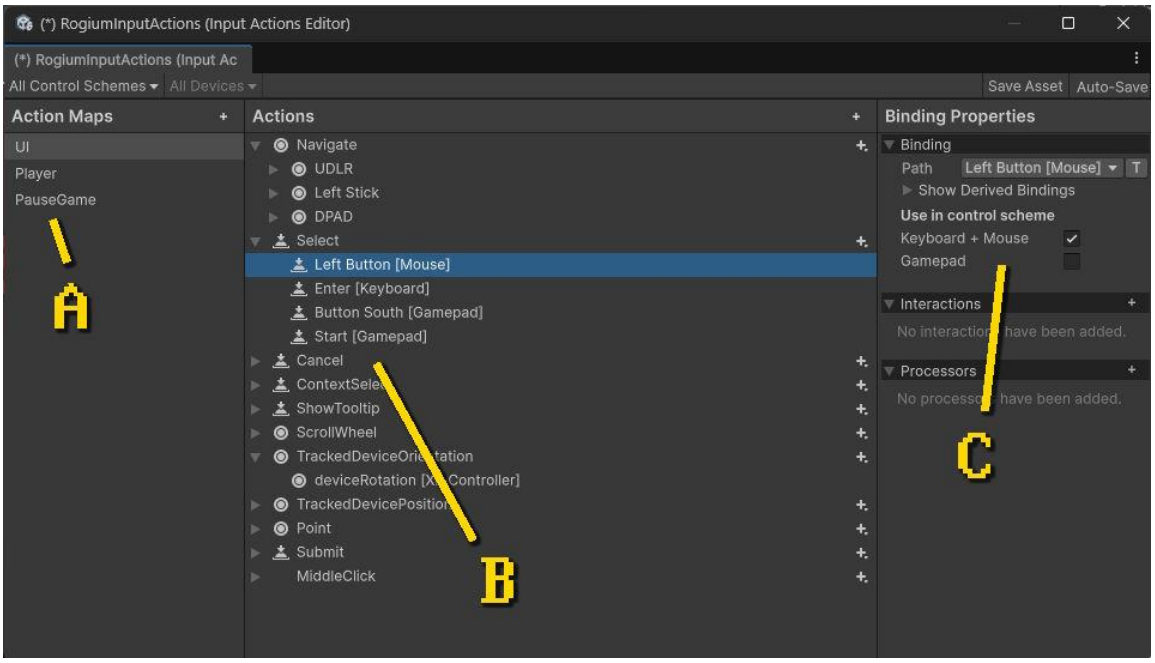
stále postrádá několik funkcí, které by z něho dělaly absolutní volbu (UI v herním světě, práce s shadery) [8].

Díky své rozšířenosti, vysoké podpoře a oficiálním doporučením společností pro využití v herním UI [8] bude pro tento projekt využit systém **uGUI**.

1.2.2 Input Systém

Jedna z implementovaných funkcí budou klávesové zkratky, je proto nutné zmínit, s jakým input systémem se bude v projektu pracovat. V dnešní době má Unity k dispozici dva oficiální: Input Manager a Input System Package. Co se rozdílů týče:

- **Input Manager** – Je starší systém, vestavěný přímo do Unity. Obsahuje absolutní minimum funkcí a i samotná dokumentace doporučuje spíše používat novější řešení [9].
- **Input System Package** – Nový systém navržený jako náhrada Input Managera. Dostupný je přes package manager. Jeho hlavními výhodami jsou, že je mnohem flexibilnější, dokáže rozlišovat vstupní zařízení a může mít několik vazeb pro jednu input akci. Na rozdíl od staršího systému jeho akce není nutné číst v Update metodě MonoBehaviour scriptů, protože využívá návrhový vzor Observer [10]. Má také vlastní konfigurační editor, který je možné vidět na obrázku 6.



Obrázek 6 – Editor input akcí v Unity

Ze zobrazeného editoru jde vyčíst princip datové struktury tohoto systému, kterou bude vhodné znát pro lepší pochopení implementační části. Vše je postaveno na **input akcích** (B). Ty reprezentují jednotlivé aktivity, které se v programu mohou provést. Příkladem může být například *pohni kursor*, *zvol položku, proveď undo operaci*, apod.

Každá input akce má k sobě přiřazeno konečně mnoho **vazeb** (C). Ty představují tlačítkové kombinace, kterými se akce spouštějí. Například šipky pro pohyb kursoru, levý klik myší pro zvolení položky nebo CTRL+Z pro undo operaci. U vazeb je možné se setkat i s pojmem **kompozity**. To jsou v podstatě jen skupiny pro další vazby. Pod 4-dimenzinálním kompozitem je si třeba možné představit kombinaci tlačítek šipka nahoru, dolů, vlevo a vpravo.

Nakonec každá input akce musí být přiřazena k nějaké **akční mapě** (A). Ty slouží k tomu, aby input akce od sebe logicky oddělily. Záměrem je, že akční mapy se aktivují/deaktivují dle potřeby programu, třeba pokud je právě spuštěné prostředí menu, zapni akční mapu zvanou UI, která drží input akce ovládající toto menu. Speciální vlastností akčních map je taky možnost jich mít několik aktivních najednou. Tato vlastnost přináší mnohem více flexibility, která se určitě využije při vývoji zkratek.

Kvůli své bohaté funkcionalitě, flexibilitě a ulehčení mnoha aspektů se v implementační části bude pracovat s Input System Package.

1.3 Návrhové vzory

Protože v implementační části bude snaha o vytvoření pochopitelného a znovupoužitelného kódu, stávají se návrhové vzory nevyhnutelným tématem. Pro vývoj budou určitě použity, proto by bylo vhodné je trochu představit. V této podkapitole se tedy stručně vysvětlí základy návrhových vzorů a představí se pár základních typů, které jsou již v příkladovém projektu využity a nebudou středem pozornosti ve vývoji nových funkcí.

1.3.1 Co jsou to návrhové vzory

Návrhové vzory v softwarovém designu jsou znovupoužitelná řešení pro obecné problémy. Jedná se v podstatě o předpřipravené plány, které si programátor poupraví pro své řešení [11][12]. Návrhové vzory jsou spíše pokročilejší programovací techniky, jejichž znalost však dokáže programátorovi podstatně ulehčit práci.

Tyto vzory je možné detailněji popsat *čtyřmi* základními charakteristikami [12]:

- **Znovupoužitelnost** – Jsou obecné natolik, že je možné je použít na podobné problémy v různých projektech a na rozdílných technologiích.
- **Standardizace** – Podobně jako UML mají vzory sdílený slovník, takže je snadnější je pochopit při kolaboraci. (Například vzor *továrna* pracuje se dvěma pojmy: *továrna* a *produkt*, které budou za každé situace znamenat to samé.)
- **Účinnost** – Díky tomu, že vzory již existují jakožto řešení určitých obecných problémů, nemusí programátoři vymýšlet zcela nová řešení. Takové ušetření tedy vede ke zrychlení vývoje.
- **Flexibilita** – Jelikož jsou vzory abstraktním řešením, může je vývojář jednotlivě přizpůsobovat svým potřebám.

Dle jejich přístupu ke kódu se návrhové vzory dělí do *tří* hlavních skupin [12][13]:

- **Tovární** vzory jsou zaměřeny na tvoření nových objektů. Pomáhají logiku vytváření oddělovat od zbytku systému.
- **Strukturální** vzory řeší problémy s uspořádáním objektů v komplikovanějších strukturách. Běžně pracují s dědičností.
- **Behaviorální** vzory jsou vhodné pro případy, kdy objekty musí mezi sebou komunikovat.

Toto rozdělení pomáhá se ve vzorech lépe zorientovat a umožňuje snadněji najít řešení pro aktuální problémy.

1.3.2 Singleton

Singleton, který spadá mezi tovární vzory [12], se v příkladovém projektu objevuje velmi často. V implementační části se na něho určitě narazí, proto dává smysl jej zde stručně představit.

Tento vzor pomáhá řešit situace, kdy potřebujeme, aby vytvořený objekt byl jediným v celé aplikaci. K tomu musí mít globální přístup z každé další třídy.

Aby se z třídy stal singleton, musí:

- Mít privátní konstruktor. Další třídy nesmí vytvořit objekty této třídy.
- Dát globální přístup ostatním třídám skrz veřejnou statickou proměnnou (většinou označenou jako *Instance*).

Zdrojový kód 1 ukazuje abstraktní třídu, kterou mohou ostatní třídy zdědit, aby se z nich stal singleton. Proměnná *Lazy* vytvoří novou instanci, až když bude potřeba [14]. K této instanci se pak přistupuje přes hodnotu *Instance*, jak je možné vidět ve zdrojovém kódu 2. Nesmí se však zapomenout do dědící třídy přidat privátní konstruktor, aby nebylo možné tvořit jeho instance v jiných třídách.

```
1. public abstract class Singleton<T> where T : Singleton<T>
2. {
3.     private static readonly Lazy<T> Lazy = new(() => (Activator.CreateInstance(typeof(T), true) as T)!);
4.     public static T Instance => Lazy.Value;
5. }
```

Zdrojový kód 1 - Abstraktní třída Singleton.

```
1. GameClock.Instance.Pause();
```

Zdrojový kód 2 - Volání metody singletonu. GameClock spravuje běh času v herním světě.

Tato varianta vzoru singleton ovšem nebude v Unity fungovat, pokud chceme udělat singleton z třídy, dědící MonoBehaviour. Jelikož se v takovém případě jedná o komponent Unity objektů, je nutné spíše zajistit, aby ve scéně vždy existoval pouze jeden komponent, a to pouze na jediném objektu. K tomu MonoBehaviour scripty nesmí používat konstruktor [15]. I když tyto vlastnosti vedou k trochu odlišné podobě kódu vzoru singleton, přístup z ostatních tříd se nemění.

1.3.3 Observer

Asi nejvíce používaným návrhovým vzorem v příkladovém projektu je observer. Jedná se o behaviorální vzor, založený na principu odběrů. Jeden objekt vždy notifikuje skupinu jiných objektů o eventech, které v něm nastanou [16]. Jazyk C# má tento vzor v sobě nativně zabudovaný ve formě eventů. Detailněji se tedy budeme zabírat právě touto podobou.

Obecně, aby třída A (sledovatel) a třída B (sledována) mohly mezi sebou navzájem komunikovat pomocí observer vzoru, je potřeba, aby:

- **Třída B vyvolávala event**. Jedná se o proměnnou, představující metodu (tedy nějaká forma *delegate*) a označenou klíčovým slovem *event*. Třída B pak tuto proměnnou vyvolává v některé své interní metodě.
- **Třída A se přihlásila k odběru eventu třídy B**. Tím se myslí, že třída A musí zaregistrovat nějakou svou metodu, pro kterou potřebuje, aby se spustila v momentě vyvolání eventu ve třídě B.

Pro konkrétní příklad možného použití vzoru observer uvedu situaci vyvolání konce hry v momentě, kdy hráčova postava zemře. Ve zdrojovém kódu 3 je možné vidět, že když hráč ukončí svůj život, vyvolá event *OnDeath*. Zdrojový kód 4 pak ukazuje herního manažéra, který se při své aktivaci přihlašuje k odběru *OnDeath* eventu metodu zvanou *EndGame()*, pomocí které ukončí hru.

```
1. public class PlayerController : EntityController
2. {
3.     public event Action OnDeath;
4.
5.     private void Die()
6.     {
7.         OnDeath?.Invoke();
8.     }
9. }
```

Zdrojový kód 3 - Příklad vyvolání eventu vzoru Observer

```
1.  public sealed class GameplayOverseerMono : MonoSingleton<GameplayOverseerMono>
2.  {
3.      [SerializeField] private PlayerController player;
4.
5.      private void OnEnable()
6.      {
7.          player.OnDeath += EndGame;
8.      }
9.
10.     private void OnDisable()
11.     {
12.         player.OnDeath -= EndGame;
13.     }
14. }
```

Zdrojový kód 4 - Příklad odběru eventu ve vzoru Observer

2 ANALÝZA SOUČASNÉHO STAVU

V této kapitole se nahlédne na dnešní stav akademických prací, co se použitelnosti v softwaru týče. Specificky se zaměří na řešení použitelnosti v oblasti vestavěných herních editorů, a to ideálně implementovaných v herním enginu Unity. Pokud budou existovat, nahlédne se i na práce řešící funkce, vybrány pro tuto práci: Undo/Redo a klávesové zkratky.

2.1 Akademické práce

V provedeném průzkumu bylo zjištěno, že vědecké práce v oblasti použitelnosti softwaru se spíše zaměřují na uživatelském testování. Samotnou implementaci většinou řeší jen okrajově, pokud vůbec. Co se specifičtější oblasti použitelnosti v herních editorech týče, tam je situace obdobná. I přes často se odlišující témata dokážou již existující práce dobře tuto diplomovou práci doplnit, ať už detailním rozborem pojmu použitelnost nebo seznámením s podobami jiných herních editorů.

Bendik Lund Flogard svou práci, *Usability in video game editors* [17], zaměřil na testování použitelnosti ve čtyřech již existujících herních editorech. V rámci své práce detailně představil pojmy jako *herní editor*, *uživatelský obsah* (user-generated content) a *testování použitelnosti ve hrách*. Následovně provedl analýzu vybraných her a jejich editorů, na nichž dále na skupině začátečníků zkoumal jejich individuální přístupy a obtíže s používáním editorů. Výsledky z analýzy prezentoval ve své poslední kapitole, kde se mu podařilo odhalit několik možných slabin a nedostatků v použitelnosti. Tuto práci dobře doplní především svým detailním rozborem herních editorů (kapitola 2), jelikož se spíše zaměřuje především na implementaci a již počítá s určitou znalostí v oblasti herních editorů.

Petra Stehlíková ve své práci, *Metody a metriky pro měření použitelnosti software* [18], nejdříve shrnuje význam pojmů použitelnost a kvalita softwaru. Pokračuje pak průzkumem oblasti existujících řešení sledování použitelnosti v softwaru, kde zjistila nedostatky měření použitelnosti pro desktopové aplikace. Následně pak implementuje vlastní řešení, které se snaží nedostatky potlačit. Svým detailním popisem pojmu použitelnost v softwaru (kapitola 3) velmi dobře doplňuje teoretickou část této práce.

Jan Rádl je se svou prací, *Editor pro stavebnici Hubelino* [19], asi nejblíže k této, co se tématu týče. Popisuje v ní postup vývoje editoru kuličkových drah pro stavebnici Hubelino. Stejně jako tato práce, Rádl si pro svůj vývoj zvolil herní engine Unity, jehož vlastnosti na začátku velmi rychle shrnuje, než se přesune k detailnímu popisu nejdůležitějších prvků vývoje. V rámci práce řeší skládání a propojování kostek, funkce vrátit zpět/znovu (Undo/Redo), uživatelské rozhraní a práce se soubory. Následně zmiňuje problémy, se kterými se během vývoje setkal, a způsob jejich řešení. Tuto práci převážně doplňuje svým náhledem na postup vývoje editoru v Unity. Rovněž také přispívá svým zaměřením na stejné prvky použitelnosti (Undo/Redo, klávesové zkratky) jako tato práce. Funkce však popisuje ve velmi jednoduché podobě, nebo je pouze zmiňuje.

Martin Khol a jeho práce, *Využití návrhových vzoru ve vývoji herních aplikací* [20], řeší využívání návrhových vzorů ve vývoji herních aplikací. Nejdříve představuje několik vybraných vzorů, u kterých popisuje jejich účel, výhody a nevýhody. Následně dané vzory aplikuje na ukázkový příklad v herním enginu Unity. K této práci dobře přispívá popisem vzorů jako jsou Singleton nebo Observer, jejichž tématiku rozvádí více do hloubky než tato diplomová práce.

Nakonec bych ještě zmínil svou bakalářskou práci, *Desktopová hra v Unity s možností sdílení uživatelského obsahu* [1]. Ta řeší a popisuje vývoj hry, která se zaměřuje na tvorbu uživatelského obsahu pomocí několika vestavěných editorů. Tématem je sice odlišná od této diplomové práce, avšak k ní přináší detailnější pohled na podobu a funkčnost editorů (kapitola 4.3), které budou vybranými funkcemi ovlivněny. Bakalářská práce taktéž popisuje základy používání enginu Unity (kapitola 2), což může usnadnit pochopení implementační části.

2.2 Jiné dokumentace

Jak bylo zmíněno výše, vybrané funkce pro tento dokument byly Undo/Redo a klávesové zkratky + jejich přemapování. V provedené analýze by bylo vhodné prozkoumat, do jaké míry je jejich implementace v Unity již na internetu zdokumentována.

2.2.1 Undo/Redo

Pro naprogramování základní funkcionality Undo/Redo systému v Unity se na internetu manuálů najde více než dost. Už jenom na webové stránce Youtube po vyhledání pojmu vyskočí velká spousta video tutoriálů. Čistě pro zajímavost tady jsou názvy dvou nalezených:

- „*Command Pattern in Unity, Part 3: Undo/Redo Functionality*“ z kanálu Board To Bits Games.
- „*InGame Tilemap Editing – Part 7: The Build History – Undo & Redo Steps – Unity Tutorial*“ z kanálu Velvary.

I samotná společnost Unity poskytla zdroj, ze kterého je možné základy Undo/Redo pochytit a naimplementovat: oficiální e-book, věnující se návrhovým vzorům (str. 73-79) [21].

Všechny vyjmenované princip dobře vysvětlují, i když se až tak moc nesoustředí na budoucí znovupoužitelnost kódu (vyjma Unity e-booku). Co však ale vysloveně chybí je nějaké hlubší nahlédnutí do tématu. Zmíněné tutoriály vysvětlují, jak pracovat z jednoduchými akcemi jako třeba Přesuň se na místo nebo Vytvoř objekt. Co když ale potřebujeme jedním tahem nastavit hodnotu posuvníku nebo vykreslit čáru na kreslicí mřížce? Kanál Velvary se něčemu takovém věnoval ve svém dalším videu, avšak ukázkový kód byl napsán pro velmi úzkou situaci.

2.2.2 Přemapování klávesových zkratek

Co se klávesových zkratek, specificky základů jejich přemapování týče, tak video tutoriálů, návodů a dokumentace je na webu velké množství. Opět na stránce Youtube se dá najít několik vysoce kvalitních tutoriálů. Jako příklady uvedu:

- „*How to Rebind Your Controls in Unity (With Icons!) | Input System*“ z kanálu Sasquatch B Studios.
- „*Complete and Persistent Control Rebinding with the New Input Systém – Unity Tutorial*“ z kanálu samyam.
- „*Unity Input System in Unity 6 (5/7): Rebinding Input System controls*“ z oficiálního kanálu Unity.

Kromě video tutoriálů, Unity Technologies nabízejí také oficiální dokumentaci ke svému systému vstupů, kde je přemapování dobře popsáno [22]. K tomu také přibalují příkladový projekt k balíčku svého nového input systému.

Zásadní problém však se všemi těmito manuály je, že jsou mířené pouze pro jednoduché přemapování ovládání, což tradičně bývá změna jednoho tlačítka na jiné. V systému klávesových zkratek se spíše počítá s možností přemapovávat celé kombinace tlačítek (třeba CTRL+U na CTRL+SHIFT+Q) a nic takového už existujícími materiály pojato není.

2.3 Zhodnocení

Co se týče zhodnocení existujících manuálů na téma vybraných funkcí, o to se stará tabulka 1.

Tabulka 1 - Stav existující dokumentace k vybraným funkcím

	Oblast	Manuály
Undo/Redo	Základy	Video tutoriály, oficiální e-book [21]
	Prvky, ovládány taháním myší	Nejsou
Přemapování zkratek	Základy	Video tutoriály, oficiální dokumentace + příkladový projekt
	Variabilní kombinace	Nejsou
	Hledání duplicitních kombinací	Video tutoriály, diskusní fóra (ale pouze pro základní)

Jak jde z tabulky vyčíst, základy obou systémů jsou na internetu dobře podchyceny. Co se však týče specifičtějších problémů, které se musí pro oba systémy vyřešit, pro ty existující dokumentace podstatně pokulhává, pokud tedy vůbec existuje. Vytvořený dokument a k němu implementované knihovny funkcí by tak mohly doplnit prázdná místa v internetové dokumentaci.

Co se akademických prací týče, práce Bendika Lund Flogarda [17] řeší testování použitelnosti několika herních editorů. Petra Stehlíková [18] se ve své práci zabývá popisem použitelnosti a vývojem softwaru pro její měření. Jan Rádl [19] svou práci zaměřuje na popis implementace editoru stavebnice. Práce od Martina Khola [20] je věnována průzkumu programovacích návrhových vzorů v herních aplikacích. Má práce [1] potom pokládá základy aplikace, se kterou bude tento dokument dále pracovat v implementační části.

Práce jsou porovnány v tabulce 1.

Tabulka 2 – Porovnání vybraných akademických prací

	Přístup k použitelnosti	Využívá Unity
Usability in video game editors [17]	Testování	Ne
Metody a metriky pro měření použitelnosti software [18]	Měření	Ne
Editor pro stavebnici Hubelino [19]	Implementace	Ano
Využití návrhových vzorů ve vývoji herních aplikací [20]	Znovupoužití kódu	Ano
Desktopová hra v Unity s možností sdílení uživatelského obsahu [1]	Neřeší	Ano

Můžeme vidět, že pouze práce [19] se nějakým způsobem věnuje implementaci funkcí pro zlepšení použitelnosti, avšak jak již bylo zmíněno, pouze ve velmi omezené podobě. Ostatní práce, i když tuto práci skvěle doplňují svým vlastním pohledem na použitelnost, se implementaci funkcí pro zlepšení použitelnosti nijak nezabývají.

Oproti výše uvedeným pracím, tato nebude řešit testování ani měření použitelnosti. Bude se čistě věnovat popisem vytvořené implementace funkcí pro zlepšení použitelnosti, a to v mnohem větším detailu, než je tomu u práce [19]. V popisu bude kladen důraz na znovupoužitelnost kódu (čemuž se věnuje práce [20]) s cílem usnadnit přenesení algoritmů do vlastních projektů.

3 NÁVRH ŘEŠENÍ

Tahle kapitola se zaměří na představení funkcí, které byly vybrány pro dokumentaci v implementační části práce. Vysvětlí se, co každá funkce obnáší, aby bylo možné si udělat lepší představu o způsobu její implementace. K tomu se ještě přidá vysvětlení důvodu, proč byla pro práci zvolena. Nakonec se ještě představí program, do kterého budou funkce zavedeny. Ten zde bude plnit roli příkladu, aby šlo vidět, jak takové funkce aplikovat na projekt větší komplexity.

3.1 Vybrané funkce

Kvůli času byly pro tuto práci vybrány právě dvě funkce: **Undo/Redo systém** a **klávesové zkratky + jejich mapování**. Tyto funkce se běžně objevují v editačních programech, jako například všechny programy patřící pod Microsoft Office, všechny IDE od společnosti Jetbrains, Paint.NET nebo třeba taky samotný engine Unity. V praxi jsou obojí funkce velmi užitečné a většinu času je jejich přítomnost v editačních softwarech považována za samozřejmost.

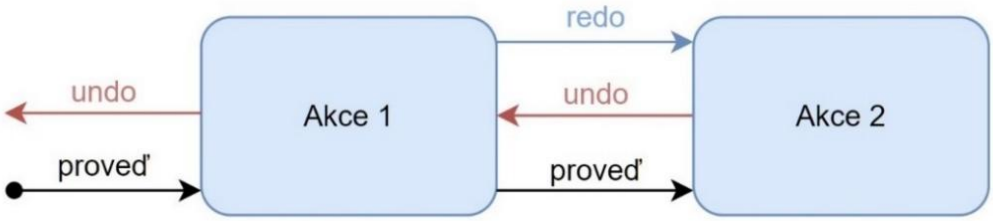
I když se však jedná o často užívané funkce, způsobům, jak je v Unity naimplementovat, není na internetu věnována dostatečná pozornost. Je sice pravda, že se nejedná o funkce spojené s herním prostředím, avšak, jak již bylo zmíněno v úvodu, i v programu Unity se vyvíjejí editační programy nebo integrované editory, které už tyto funkce dokážou plně využít.

3.1.1 Undo/Redo systém

První z vybraných funkcí pro implementaci je systém Undo/Redo. Jedná se asi o nejrozšířenější a uživateli nejvyužívanější funkci, která zlepšuje použitelnost softwaru.

Princip systému

Princip fungování Undo/Redo pomáhá vysvětlit obrázek 7.



Obrázek 7 - Princip fungování Undo/Redo systému

V plnohodnotném Undo/Redo systému je veškerá uživatelova aktivita vnímaná jako posloupnost akcí (**modré bloky**). Pokud použijeme kreslicí editor jako příklad, tak tyto akce mohou představovat akce jako třeba *nakresli čáru*, *vyplň barvou* nebo *aplikuj efekt*.

Jednotlivé akce vždy uživatel provede jednu podruhé (**černá čára**). Když ovšem provede nějakou nevhodně (třeba špatně nakreslí čáru), může požádat systém o její „vrácení" neboli operaci *undo*. Vrácení (**červená čára**) provede původní akci přesně opačně (například *nakresli čáru → smaž čáru*, *vyplň barvou → vyplň předešlou barvou*) a tím objekt vrátí do stavu v předešlém kroku. Takto může uživatel snadno a rychle opravovat své chyby.

Pokud se však stane, že uživatel aktivoval operaci undo nechtěně nebo si její výsledek rozmyslel, může využít k ní opačnou akci: redo. Redo (**modrá čára**) znovu automaticky provede původní provedení (*smaž čáru → nakresli čáru*). Nutno dodat, že pokud uživatel provede operaci undo a poté nějakou zcela jinou akci (třeba *vyplň barvou*), všechny existující redo akce budou smazány. K tomuto jevu dochází, protože systém už nemůže vědět, k jaké změně došlo.

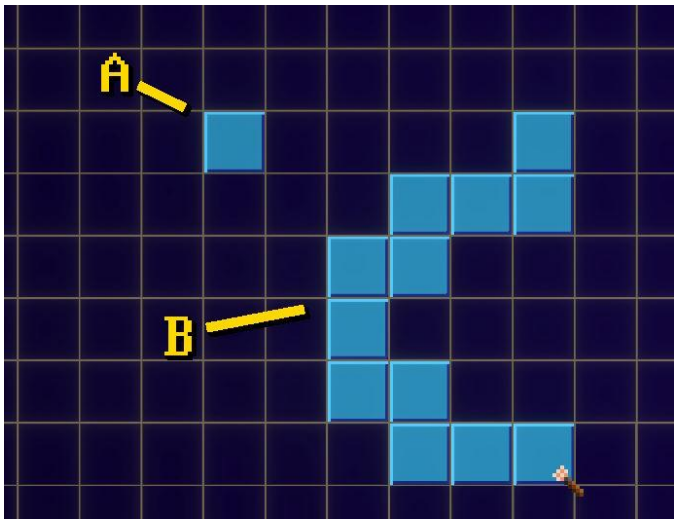
Důvod implementace

Pro program Unity je možné na internetu najít několik tutoriálů a návodů, jak tuto funkci zavést do projektů. Ve většině případů se v nich však řeší **pouze naprosté základy**, kdy zavádějí do podpory systému jen velmi jednoduché akce (viz kapitola 2.2.1). V této práci se v rámci implementace Undo/Redo taky budou řešit jednoduché základy, ale to spíše z důvodu snadnějšího pochopení pozdějších, komplexnějších problémů.

Bude popsáno, jak naprogramovat kostru tohoto systému a ukáže se i několik příkladů zavedení jeho podpory do určitého počtu UI prvků. Pak však dojde k přesunu na složitější případy, na které jednoduchý systém nebude stačit.

Hlavním problémem bude zajištění podpory pro **kreslicí mřížku**. Jedná se o mřížku, ve které uživatel myší dosazuje hodnoty do jednotlivých buněk. Většinou se s ní setkáme v kreslících editorech (malování, gimp), kde buňky představují jednotlivé pixely obrázku. Můžeme se s ní však také setkat v editorech úrovní, kde uživatel může vkládat do buněk dlaždice, po kterých bude hráč chodit, nebo objekty, se kterými bude hráčova postava interagovat.

Hlavním problémem zde bude nutnost odlišit uživatelovy tahy myší od jednotlivých kliknutí. Lépe tento problém pomáhá pochopit obrázek 8.



Obrázek 8 – Problém podpory Undo/Redo pro kreslicí mřížku

Je na něm zobrazena kreslící mřížka a několik dlaždic, které uživatel vykreslil. Případ **A** by šel snadno vyřešit jednoduchou akcí *Nakresli dlaždici*, avšak co s případem **B**? Jak poznat, zda uživatel vykreslil linii dlaždic jedním tahem, dvěma tahy s menší pauzou mezi nimi, nebo že dokonce každou dlaždici přidal pečlivě samostatným kliknutím?

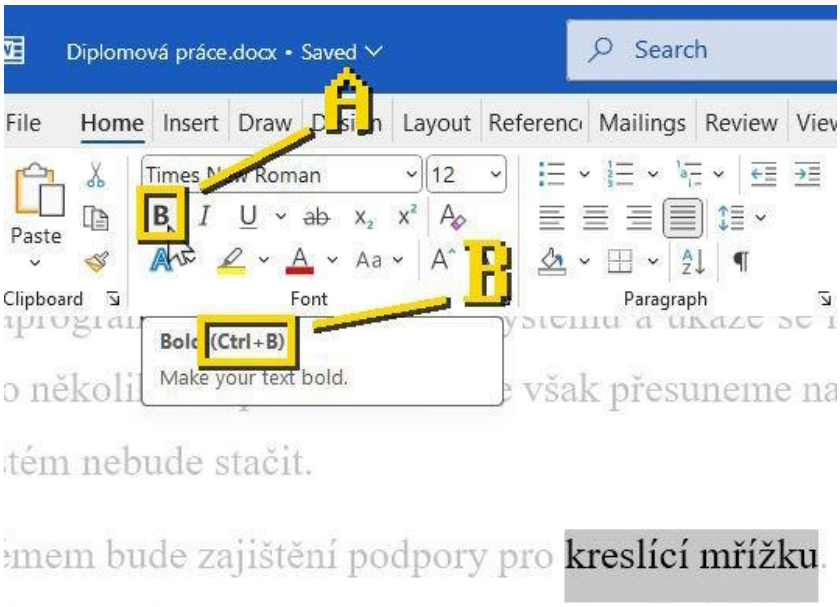
Právě tomuto problému bude věnována podstatná část implementace Undo/Redo systému. Kromě řešení pro kreslicí mřížku se pokusí problém zabstraktnit, aby jej bylo možné využít i pro další podobné případy.

3.1.2 Systém klávesových zkratek

Druhá funkce, které se bude tato práce věnovat, jsou klávesové zkratky a jejich mapování. Tato funkce se také často objevuje v editačních programech, kde je zkušenějším uživatelům schopna podstatně urychlit práci.

Princip systému

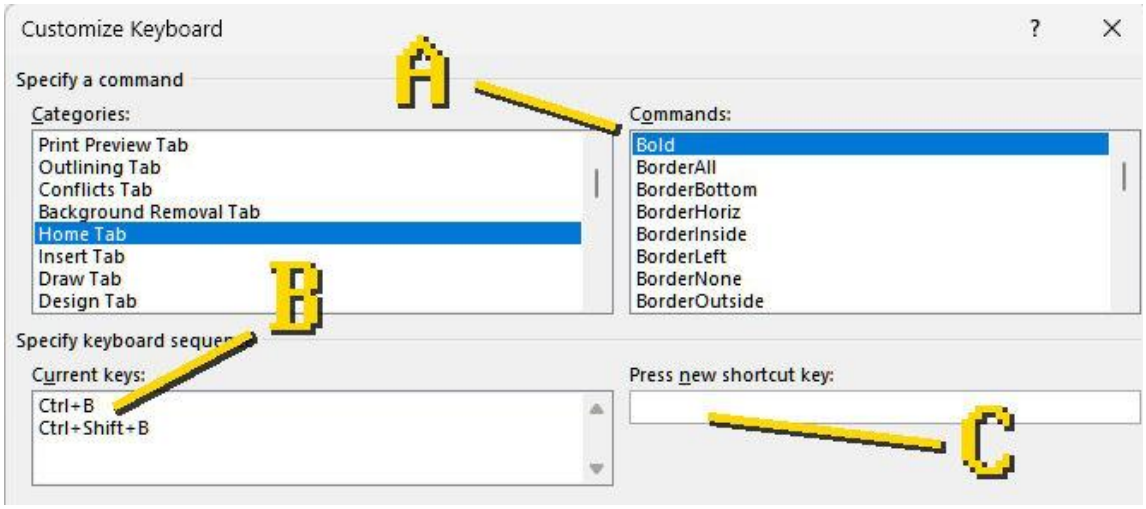
Principem funkce je, že vybraným akcím/funkcím v aplikaci přiřazuje určitou kombinaci kláves. Takováto kombinace kláves se označuje jako **klávesová zkratka** nebo prostě zkratka a jejím stisknutím se provede k ní přiřazená akce [23]. Obrázek 9 ukazuje příklad využití systému klávesových zkratek v programu Word.



Obrázek 9 - Příklad akce, která může být aktivována klávesovou zkratkou

Kliknutím na tlačítko (A) je možné převést vybraný text na jeho tučnou variantu. V popisu tlačítka je však možné vidět, že tato akce má přiřazenou klávesovou zkratku CTRL+B (B). Stisknutím této kombinace kláves se provede ztučnění stejně, jako kdyby uživatel stiskl tlačítko.

Ve většině programů má uživatel i možnost si tyto klávesové zkratky sám přemapovat. V takových případech program disponuje rozhraním, které je pro tyto operace uzpůsobené. Jeho příklad je možné vidět na obrázku 10.



Obrázek 10 - Mapování klávesových zkratek ve Wordu

Ten představuje okno, ve kterém je možné přiřazovat zkratky ve Wordu. To obsahuje několik UI prvků, které jsou pro rozhraní mapování klávesových zkratek typické:

A) List všech akcí, kterým je možné přiřadit klávesové zkratky.

B) List všech klávesových zkratek, patřící pod danou akci. Word podporuje variabilní počet zkratek, avšak jiné programy mohou dovolit pouze jednu nebo dvě takové zkratky.

C) Vstupní pole pro novou zkratku. Zde může uživatel na pole kliknout a stisknout libovolnou kombinaci kláves pro uložení nové zkratky. Word nemá limit pro počet kláves ani pro druhy kláves. Jiné programy mohou ovšem obsahovat limit pro počet kláves i pro možné klávesy, které se mohou v kombinaci objevit. Například Jetbrains Rider dovoluje nastavit zkratky s libovolným počtem modifikátorů (třeba CTRL+F, CTRL+SHIFT+F, apod), ale nepovoluje kombinace ostatních kláves (není povolena kombinace třeba F+G, CTRL+F+A, apod).

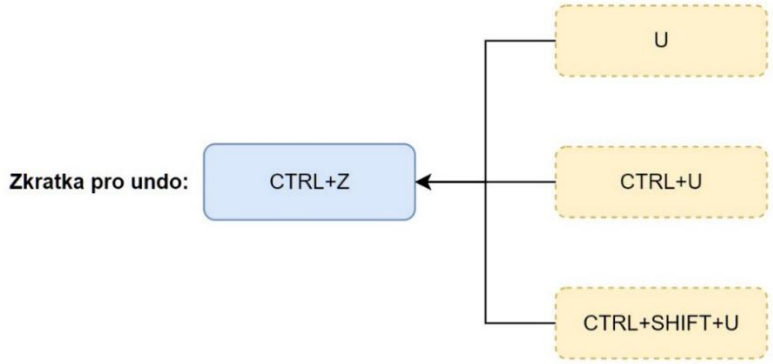
Důvod implementace

Samotné přiřazování akcí ke klávesovým zkratkám je velmi elementární, tudíž bude v této práci zmíněno pouze letmo. Mapování vlastních zkratek si však díky své možné komplexitě zaslouží důkladnější zaměření.

V herním prostředí se systém mapování klávesových zkratek většinou využívá pro úpravu ovládání a je zde spíš označován jako tzv **Input Rebinding**. Jedná se o variantu, která většinou umožňuje pro každou akci namapovat pouze jedno, maximálně dvě klávesy/tlačítka. Kombinace kláves většinou nepodporuje.

Právě pro tuhle variantu systému existuje na internetu velká spousta tutoriálů a i samotné Unity přikládá ke svému Input System Package implementovaný příklad (viz kapitola 2.2.1). U většiny těchto existujících materiálů však panuje jeden zásadní problém: *Neřeší, jak pro klávesovou zkratku přijmout jako vstup variabilní počet kláves*.

Co se touto větou myslí pomáhá vyjasnit obrázek 11.



Obrázek 11 - Možné kombinace klávesových zkratek pro 1 akci

Pokud například chce uživatel změnit klávesovou zkratku pro operaci undo z CTRL+Z na CTRL+U, nemá s tím Unity problém. Toto dokáže systém přemapování vazeb v základní formě zajistit. Pokud ovšem chce změnit kombinaci CTRL+Z na kombinaci s jiným počtem kláves, třeba CTRL+SHIFT+U nebo pouze U, narazí na limitaci. Limitaci, která už není existujícími materiály pokryta.

Převážná část implementace mapování klávesových zkratek se tedy bude věnovat hlavně tomuto problému. Pokud bude výsledek úspěšný, mohla by práce přinést dobré objasnění na to, jak v programu Unity naimplementovat systém klávesových zkratek s variabilními kombinacemi kláves.

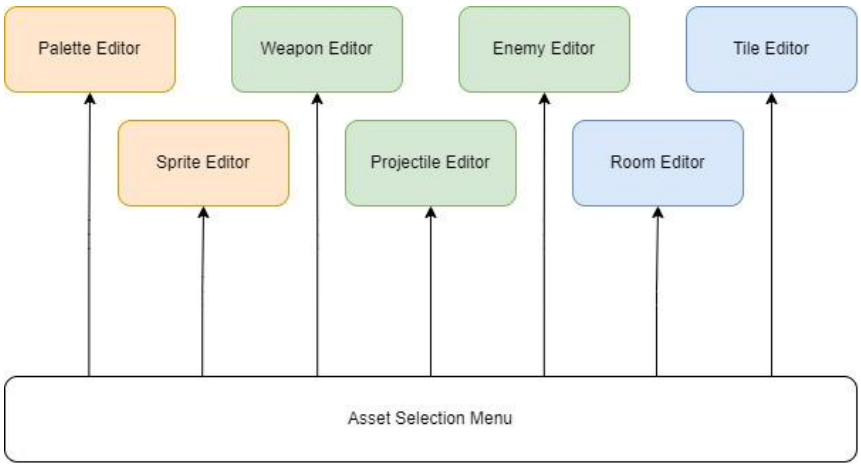
3.2 Představení příkladového programu

Nyní by bylo vhodné ještě představit projekt, se kterým se bude v implementační části pracovat. Jeho použití v této práci umožní prakticky ukázat, jak zavést implementované funkce do komplexnějšího projektu.

Využit bude program, který byl hlavním podnětem mé bakalářské práce [1]. Jednalo se o hru, ve které hráč procházel místnostmi, bojoval s nepřáteli a sbíral zbraně. Hlavním zaměřením práce však byla možnost tyto místnosti, nepřátele, zbraně a další vytvářet a upravovat pomocí vestavěných editorů.

V této práci se budou využívat tyto zmíněné vestavěné editory. Poslouží jako perfektní příklad pro ukázání modularity implementovaných funkcí, tedy jejich způsob být využity v několika rozdílných situací.

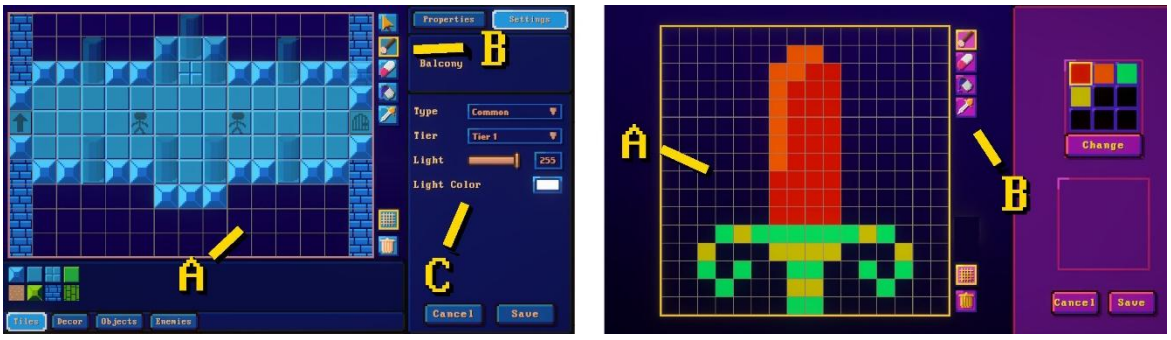
Nastínit situaci editační části aplikace pomáhá obrázek 12.



Obrázek 12 - Struktura editační části příkladové aplikace

Program obsahuje mnoho editorů. Každý slouží k upravení zcela odlišného druhu assetu (grafika, zbraň, místnost), princip editace je však vždy stejný, kdy každý editor by dokázal využít Undo/Redo systém pro vrácení nevhodných akcí a klávesové zkratky pro urychlení editací.

Nemá však smysl zde detailně rozebírat každý editor. Nové funkce se budou do nich napojovat stejným způsobem a tento popis byl již proveden v mé bakalářské práci. Pro potřeby této práce postačí jen trochu nahlédnout na podobu dvou podobných editorů: editor *místností* a *grafiky*. Jejich podoba je vyobrazena na obrázku 13.



Obrázek 13 - Podoba editorů místností a grafiky

Obsahují pro Undo/Redo systém zajímavou *kreslicí mřížku* (A), představenou v kapitole 3.1.1, a pro systém klávesových zkratek vhodný příklad: kreslicí nástroje (B). Kromě toho, editor místností (vlevo) ještě navíc v sobě obsahuje několik UI prvků (C)

(roztahovací seznamy, posuvníky, textová pole, apod), které se objevují i v jiných částech aplikace a které rovněž budou vyžadovat podporu Undo/Redo systému.

Pro hlubší pochopení funkcí editorů, interakce assetů a vztahu k herní části je doporučeno se podívat do mé bakalářské práce [1]. Kromě potřebných informací pro nové funkce se těmto tématům práce věnovat nebude.

4 IMPLEMENTACE UNDO/REDO

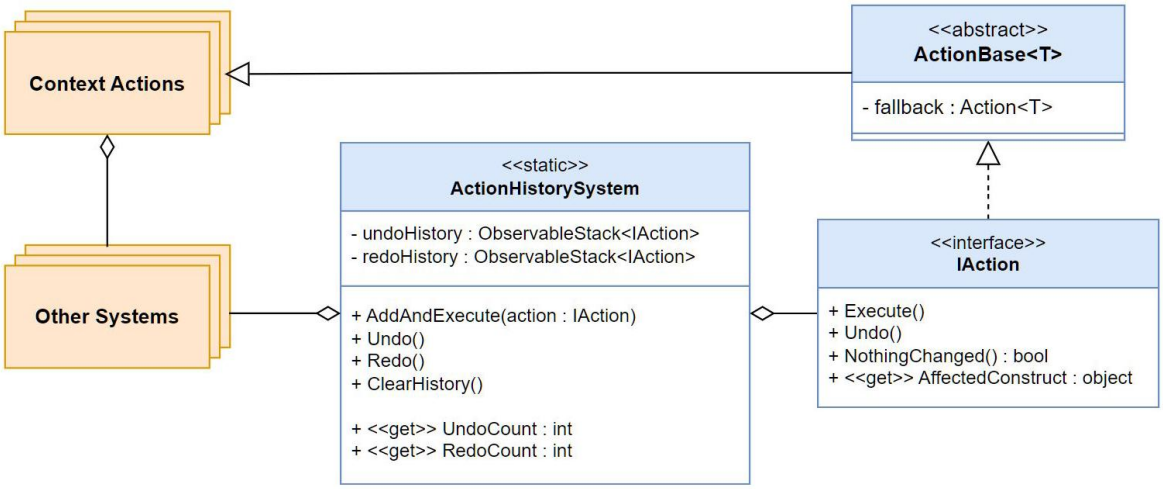
V této kapitole se popíše, jakým způsobem byl implementován systém Undo/Redo. Pro jeho úspěšnou implementaci je nutné, aby podporoval:

- Nastavení všech interaktivních UI prvků (roztahovací seznamy, posuvníky, přepínače, apod).
- Nástroje, pracující s kreslicí mřížku (štětec, guma, apod), (viz kapitola 3.1.1).

Nejdříve se vysvětlí, jak systém funguje pro jednoduché akce, díky čemuž bude snazší pochopit způsob interakce systému se zbytkem aplikace. Následně se tato dokumentace začne zabývat způsobem, jakým systém shlukuje akce. Díky této vlastnosti je možné zavést podporu i pro komplexní konstrukty, jako je třeba kreslicí mřížka.

4.1 Kostra systému

Na obrázku 14 je možné vidět diagram tříd základní funkcionality Undo/Redo systému. (pozn.: graf neobsahuje funkční prvky pro shlukování akcí.)



Obrázek 14 – Diagram tříd základní kostry Undo/Redo systému

Celý systém je postavený na myšlence návrhového vzoru **Command**, kdy každá akce, která by měla být podporována systémem Undo/Redo (třeba *Napiš text* nebo *Vyplň barvou*), je reprezentována konkrétní třídou (v obrázku 14 soubor tříd **Context Actions**). Každá taková třída obsahuje informace nutné pro provedení akce a jejího vrácení i metody, provádějící samotnou akci.

4.1.1 Vyvolávač a správce akcí

Středem systému Undo/Redo je statická třída **ActionHistorySystem** (dále AHS). Ta jednak zastává funkci spravování undo a redo historie (ve třídě reprezentovány 2 zásobníky). Zároveň ale funguje jako komunikátor s ostatními systémy (na obrázku 14 **Other Systems**). Pokud tyto systémy potřebují provést nějakou akci, registrovatelnou v historii, nebo zavolat undo/redo poslední akce, musí volat metody třídy AHS. Tato struktura opět vychází z návrhového vzoru Command, kde AHS zde zastává funkci vyvolávače (Invoker).

Třída je statická, protože jí tato vlastnost umožňuje být snadně přístupnou ostatním systémům. Jelikož se jedná o jednodušší variantu Undo/Redo systému (pouze 1 zásobník pro undo i redo historii), je toto řešení dostačující.

Ve zdrojovém kódu 5 je možné vidět, jak může vnější systém AHS volat. Na vybraném příkladě volání provádí třída, reprezentující roztahovací seznam (IPDropdown). Seznamu byla v tento moment nastavena nová hodnota, a tudíž ji chce zapsat do undo historie.

```
1. private void WhenValueChanged(int value)
2. {
3.   ActionHistorySystem.AddAndExecute(new UpdateDropdownAction(this, value, lastValue, whenValueChange)
4.   lastValue = value;
5. }
```

Zdrojový kód 5 – Volání ActionHistorySystem pro registraci akce

Na řádku 3 vybraná metoda volá statickou metodu **AddAndExecute**, která patří třídě AHS. Do ní předává svou specifickou akci v podobě třídy, obsahující všechny relevantní informace (volající konstrukt, novou a starou hodnotu, fallback funkci). Podoba AddAndExecute() je vyobrazena ve zdrojovém kódu 6.

```
1. public static void AddAndExecute(IAction action, bool blockGrouping = false)
2. {
3.     if (action == null) return;
4.     if (action.NothingChanged()) return;
5.
6.     action.Execute();
7.     redoHistory.Clear();
8.     DecideGroupingResponseFor(action, blockGrouping);
9.     lastAction = action;
10.}
```

Zdrojový kód 6 – Provedení a přidání akce do historie

Po kontrolách hodnoty akce (ř. 3 a 4) je vidět, že nejdříve se spustí, co akce požaduje (ř. 6). Následuje vyčištění redo historie (ř. 7), protože pokud uživatel učinil před akcí undo, zcela tak změnil posloupnost akcí. Obsah redo historie by už tudíž nedával smysl. Nakonec metoda

DecideGroupingResponseFor() na ř. 8 rozhodne, jestli bude akce přidána do undo historie, nebo do právě otevřené skupiny akcí. Tomuto chování se však věnuje kapitola 4.2.2.

Kromě AddAndExecute je vhodné nahlédnout i na funkcionalitu AHS metod **Undo()** a **Redo()**, jelikož se jedná o klíčové metody systému. Zdrojový kód 7 nahlíží na Undo a zdrojový kód 8 na Redo.

```
1. public static void Undo()
2. {
3.     if (currentGroup != null) AddCurrentGroupToUndo();
4.     if (undoHistory.Count == 0) return;
5.
6.     IAction newestAction = undoHistory.Pop();
7.     redoHistory.Push(newestAction);
8.     newestAction.Undo();
9. }
```

Zdrojový kód 7 - Metoda Undo()

Pokud undo historie obsahuje nějaké akce, vytáhne se z ní ta nejnovější (ř. 6) a přesune se do redo historie (ř. 7). Nakonec se vykoná její specifická undo operace (ř. 8).

```
1. public static void Redo()
2. {
3.     if (redoHistory.Count == 0) return;
4.
5.     IAction newestAction = redoHistory.Pop();
6.     undoHistory.Push(newestAction);
7.     newestAction.Execute();
8. }
```

Zdrojový kód 8 - Metoda Redo()

Redo() funguje obdobně. Jestliže se v redo historii nachází nějaká akce, je z ní vytáhnuta (ř. 5) a vloží se do undo historie (ř. 6). Následně se provede její Execute() metoda, tedy původní průběh akce (ř. 7).

4.1.2 Třídy akcí

Aby bylo možné jednotlivé akce, jako *Změň text* nebo *Vykresli hodnotu*, držet v kolekcích, musí se převést do podoby datových hodnot. V systému je tedy každá unikátní akce reprezentována svou vlastní třídou (*UpdateSliderAction, UseToolAction*, atd), která vždy obsahuje všechny relevantní informace, potřebné k vykonání akce.

Aby ovšem AHS mohl s těmito třídami pracovat nezávisle na jejich obsahu, je potřeba určitá forma abstraktizace, které se dosáhlo využitím rozhraní **IAction**. To implementuje každá třída reprezentující akci, uložitelnou v undo i redo historii. Využití tohoto rozhraní bylo možné vidět ve výše uvedených zdrojových kódech, a i v grafu na obrázku 14, který ukazuje jeho metody.

O úroveň níže je v projektu využita ještě abstraktní a generická třída **ActionBase<T>**. Tu dědí každá třída konkrétní akce, kromě několika speciálních (viz kapitola 4.2). Tato třída také implementuje rozhraní IAction, ke kterému navíc přidává akcím extra funkcionalitu. Tu by však bylo vhodnější vysvětlit až po probrání podoby tříd konkrétních akcí.

Třídy konkrétních akcí všechny používají stejnou šablonu. Zdrojový kód 9 ukazuje podobu jedné z těchto tříd: *UpdateDropdownAction*. Ta představuje akci nastavení hodnoty roztahovacího seznamu.

```
1.  public class UpdateDropdownAction : ActionBase<int>
2.  {
3.      private readonly IPDropdown dropdown;
4.      private readonly int value;
5.      private readonly int lastValue;
6.
7.      public UpdateDropdownAction(IPDropdown dropdown, int value, int lastValue, Action<int> fallback)
8.      : base(fallback)
9.      {
10.         this.dropdown = dropdown;
11.         this.value = value;
12.         this.lastValue = lastValue;
13.     }
14.
15.     protected override void ExecuteSelf() => dropdown.UpdateValueWithoutNotify(value);
16.
17.     protected override void UndoSelf() => dropdown.UpdateValueWithoutNotify(lastValue);
18.
19.     public override bool NothingChanged() => value == lastValue;
20.
21.     public override object AffectedConstruct
22.     {
23.         get
24.         {
25.             try { return dropdown?.gameObject; }
26.             catch (MissingReferenceException) { return null; }
27.         }
28.     }
29. }
```

Zdrojový kód 9 - Třída UpdateDropdownAction

V ní je možné vidět, že obsahuje informaci o nově nastavené a předešlé hodnotě (ř. 4 a 5). K tomu drží informaci o tom, který roztahovací seznam bude touto akcí ovlivněn (ř. 3).

Když se přeskočí konstruktor, třída dále obsahuje tři metody a jednu vlastnost. Všechny pochází původně z IAction rozhraní (ActionBase<T> jen mírně upravuje Execute a Undo).

- **ExecuteSelf()** (ř. 15) provede požadovanou akci třídy. V tomto případě tedy nastaví roztahovací seznam na jeho novou hodnotu.
- **UndoSelf()** (ř. 17) tuto akci vrátí, tedy nastaví roztahovací seznam na původní hodnotu.
- **NothingChanged()** (ř. 19) je kontrolující metoda, která zjišťuje, jestli akce vůbec něco změnila (Došlo ke změně hodnoty?).

- **AffectedConstruct()** (ř. 21) je veřejná vlastnost, umožňující přístup ke konstruktu, který je touto akcí ovlivněn. V tomto případě jde o jednu specifickou instanci třídy *IPDropdown*, která v příkladovém projektu ovládá roztahovací seznam.

4.2 Problémy a jejich řešení

Takto implementovaný systém bude dobře fungovat pro základní akce za nejlepších podmínek. Ovšem jako každý další existující systém, tak i pro Undo/Redo se během testování narazilo na několik komplexních problémů, které bylo potřeba vyřešit.

4.2.1 Problém chybějícího konstruktu

Tím méně složitým byl problém chybějícího konstruktu. Jak již bylo vysvětleno v předešlé kapitole, každá akce v Undo/Redo systému je závislá na nějakém konstruktu, který je jí ovlivněn. Může jím být třeba roztahovací seznam, textové pole nebo kreslicí mřížka. Tyto konstrukty vždy začaly s nějakou výchozí hodnotou, která jim pak byla uživateli změněna. Po nějaké době však mohli uživatelé tuto hodnotu vrátit pomocí operace undo. *Co by se však stalo, kdyby uživatel chtěl provést takovou undo operaci, ale konstrukt již neexistoval (byl zničen nebo deaktivován)?*

Obrázek 15 tento problém pomáhá vysvětlit na praktickém příkladu. V editoru místnosti uživatel upravuje vlastnosti *výstupní brány*. Jedná se o objekt kterého když se hráčova postava při hraní dotkne, opustí nynější místnost a přesune se do nové. Této výstupní bráně je možné nastavit, jakým směrem se hráč vydá po interakcí s ní a do jakého typu místnosti se dále přesune.

Obrázek 15 - Problém chybějícího konstruktu

Obrázek je rozdělen do několik po sobě jdoucích kroků, které je třeba vykonat, aby problém nastal. Ty jsou:

1. Uživatel vložil do mřížky výstupní bránu a nastavuje druh místnosti, do které se hráčova postava po interakci přesune (více o typech místností v mé bakalářské práci, kapitola 4.2.3). Pro nastavení typu se využívá roztahovací seznam (A), jemuž uživatel změní hodnotu z *Common* na *Rare*.
2. Uživatel na kreslicí mřížce zvolí vedlejší dlaždici (B). Jelikož brána již není zvolena, nedává smysl vykreslovat její upravitelné vlastnosti. Roztahovací seznam se tedy smaže.
3. Uživatel si rozmyslel svou úpravu brány a chce hodnotu *Rare* vrátit zpět na původní *Common*. Největší smysl tedy dává vyvolat undo operaci (C).
4. Nastane problém. Jelikož roztahovací seznam neexistoval v době provedení undo operace, nebyl tedy přenastaven na původní hodnotu. Proto, po opětném vybrání výstupní brány, se vlastnost typu další místnosti vygenerovala s nejnovější hodnotou *Rare* (D).

Řešení

Tento problém byl vyřešen umožněním akcím upravit pouze data, pokud přiřazený konstrukt nebyl detekován. Tedy pokud byl pro akci detekován chybějící konstrukt, nenastav požadovanou hodnotu na konstruktu, ale rovnou atributům v datové třídě upravovaného assetu.

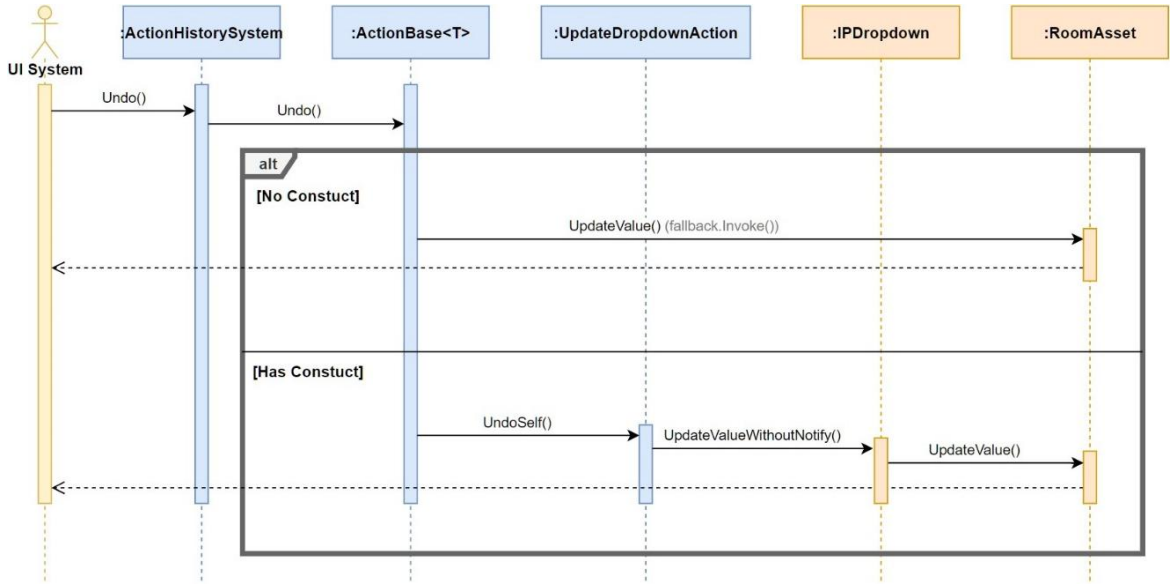
Tohoto řešení bylo dosáhnuto přidáním abstraktní, generické třídy **ActionBase<T>**, přestavenou zdrojovým kódem 10.

```
1. public abstract class ActionBase<T> : IAction
2. {
3.     private readonly Action<T> fallback;
4.
5.     protected ActionBase(Action<T> fallback) => this.fallback = fallback;
6.
7.     public void Execute()
8.     {
9.         if (AffectedConstruct == null)
10.        {
11.            fallback?.Invoke(Value);
12.            return;
13.        }
14.        ExecuteSelf();
15.    }
16.
17.    public void Undo()
18.    {
19.        if (AffectedConstruct == null)
20.        {
21.            fallback?.Invoke(LastValue);
22.            return;
23.        }
24.        UndoSelf();
25.    }
26.
27.    public abstract bool NothingChanged();
28.
29.    protected abstract void ExecuteSelf();
30.
31.    protected abstract void UndoSelf();
32.
33.    public abstract object AffectedConstruct { get; }
34.}
```

Zdrojový kód 10 - Třída ActionBase<T>

Děděna každou konkrétní akcí, tato třída přidává k metodám Execute() a Undo() schopnost kontrolovat, zdali jejich konstrukt stále existuje (ř. 9 a 19). Pokud ano, volá se unikátní varianta těchto metod, kterou si nastavuje každé akce (ř. 14 a 24). Jinak se volá tzv. **fallback** metoda (ř. 11 a 21), jež se přiřazuje v konstruktoru každé akce. Jejím úkolem je zajistit nastavení správné hodnoty čistě na datové úrovni.

Toto chování pomáhá lépe ukázat sekvenční diagram na obrázku 16. Ten vyobrazuje, jak nyní systém reaguje na provedení undo operace stejným způsobem, jako v dříve zmíněném příkladě na obrázku 15.



Obrázek 16 - Řešení problému chybějícího konstruktu

Je vidět, že pokud konstrukt již neexistuje, zcela se přeskočí jeho nastavování a rovnou se nastaví hodnota ve třídě RoomAsset, která představuje místnost v datové úrovni a obsahuje informace o uspořádání jednotlivých dlaždic a objektů. Jakmile by uživatel znova zvolil výstupní bránu v editoru místnosti, roztahovací seznam pro nastavení druhu další místnosti již bude mít správnou hodnotu.

4.2.2 Problém podpory tahání myší

Tento problém byl již popsán v kapitole návrh na str. 25 na příkladu kreslicí mřížky. Během implementace se ovšem zjistilo, že daný problém se netýká pouze jí. Stejným problémem trpěl také UI prvek: posuvník. Bylo by tedy vhodné zde problém ještě zobecnit, než se práce přesune k vysvětlení řešení.

Obecně se bude problém dále označovat jako problém podpory tahání myší. Proč se takhle nazývá, pomáhá vysvětlit obrázek 17, který ukazuje praktický případ tohoto problému. Uživatel si přeje nastavit, jak dlouho bude trvat animace vybraného assetu zbraně. K nastavení takové hodnoty se v programu využívá UI prvek: *posuvník*. Po čase si však své nastavení uživatel rozmyslí a přeje si jej vrátit.

Obrázek 17 - Problém podpory tahání myší

Nastavení hodnoty je rozděleno do několika kroků, jdoucích po sobě:

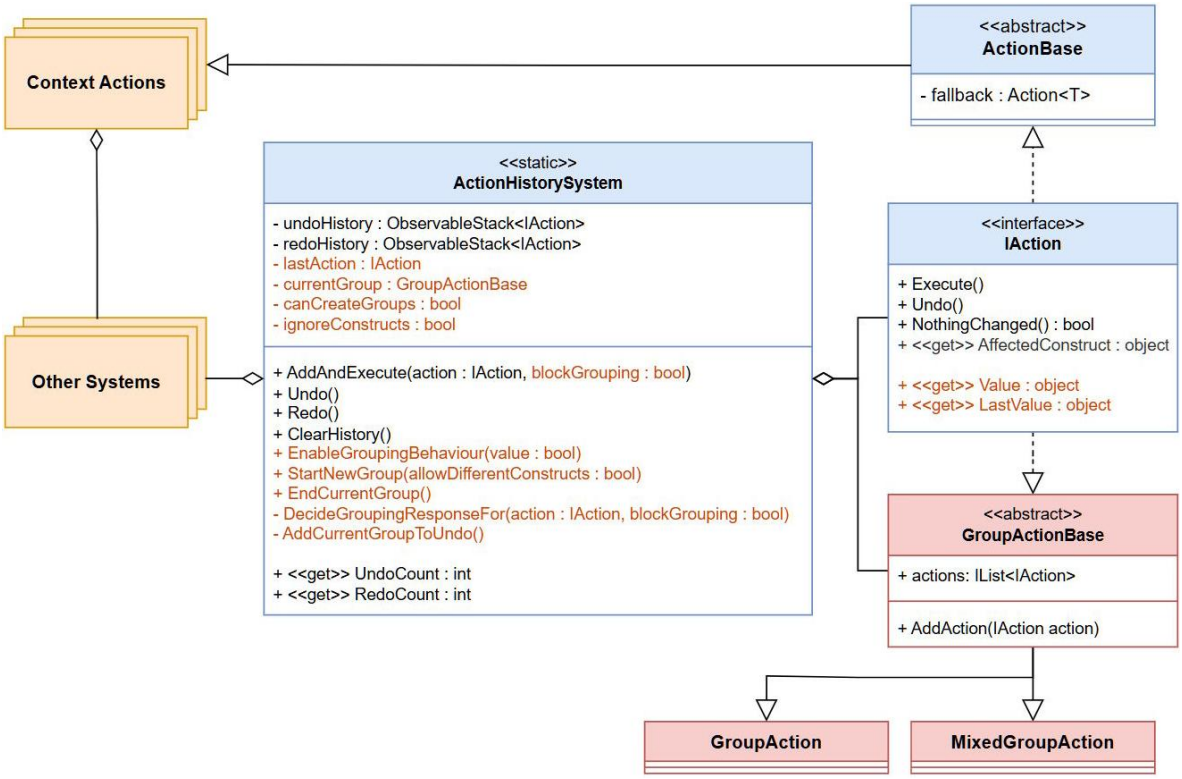
1. V úvodním stavu má posuvník hodnotu nastavenou na 40.
2. Uživatel myší potáhnul posuvník na novou hodnotu.
3. Tuto novou hodnotu si však rozmyslel a radši by použil onu původní. Tlačítkem tedy vyvolal undo operaci a očekával, že se hodnota posuvníku vrátí na původní hodnotu (40). Místo toho se však hodnota zmenšila o náhodné číslo. Zkusil tedy akci zopakovat, avšak pokaždé došlo jen k drobnému, náhodnému přiblížení k původní hodnotě.

K tomuto chování došlo, protože během tahání posuvníku se každý herní snímek zaznamenala do undo historie nová akce. Té se ovšem pod starou a novou hodnotou vždy zaznamenávala čísla, mezi kterými se posuvník přesunul zrovna **ten daný snímek**. Akce si již nepamatovala, jaká byla počáteční hodnota posuvníku o několik snímků před.

Stejným způsobem by se chovalo i undo pro nakreslenou čáru dlaždic na kreslicí mřížce (viz obrázek 8). Po vyvolání undo operace by se odmazávaly pouze ty dlaždice, které se stihly vykreslit v aktivním herním snímku.

Řešení

Tento problém byl vyřešen dodáním možnosti **shlukovat akce** do AHS. Systém má tak schopnost považovat několik akcí, spojené určité podmínkou, za jednu jedinou. Toto řešení přidává novou funkcionalitu do AHS a nové typy akcí. Obohacený systém jde dobře vidět na obrázku 18, který představuje původní diagram tříd Undo/Redo systému (obrázek 14), avšak již rozšířen o veškerou funkcionalitu shlukování akcí (zobrazených **oranžově**).



Obrázek 18 – Diagram tříd Undo/Redo systému, obohacený o shlukování akcí

Je v něm vidět, že veškeré shlukování zařizuje třída AHS. Ta bude pro jeho fungování využívat nové druhy akcí, pro které platí, že jsou všechny sloučeny pod abstraktní třídou *GroupActionBase*. Než se však k těmto třídám práce přesune, bylo by vhodné vysvětlit, jak shlukování v AHS vlastně funguje.

Jakým způsobem se vlastně vytváří skupiny akcí? Které akce se budou shlukovat? K dosažení odpovědí na tyto otázky by bylo vhodné začít v metodě *AddAndExecute()*. Ta byla již ukázána ve zdrojovém kódu 6, ve kterém bylo možné zahlédnout na ř. 8 metodu **DecideGroupingResponseFor()**. Jejím úkolem je rozhodovat o tom, co se dále stane s požadovanou akcí. Zdrojový kód 11 metodu představuje a pomůže popsat princip jejího fungování.

```
1. private static void DecideGroupingResponseFor(IAction action, bool blockGrouping = false)
2. {
3.     if (canCreateGroups && !blockGrouping)
4.     {
5.         if (currentGroup == null)
6.         {
7.             currentGroup = (ignoreConstructs) ? new MixedGroupAction() : new GroupAction();
8.             currentGroup.AddAction(action);
9.             return;
10.        }
11.
12.        if (ignoreConstructs || action.AffectedConstruct == lastAction?.AffectedConstruct)
13.        {
14.            currentGroup.AddAction(action);
15.            return;
16.        }
17.
18.        AddCurrentGroupToUndo();
19.    }
20.
21.    undoHistory.Push(action);
22.}
```

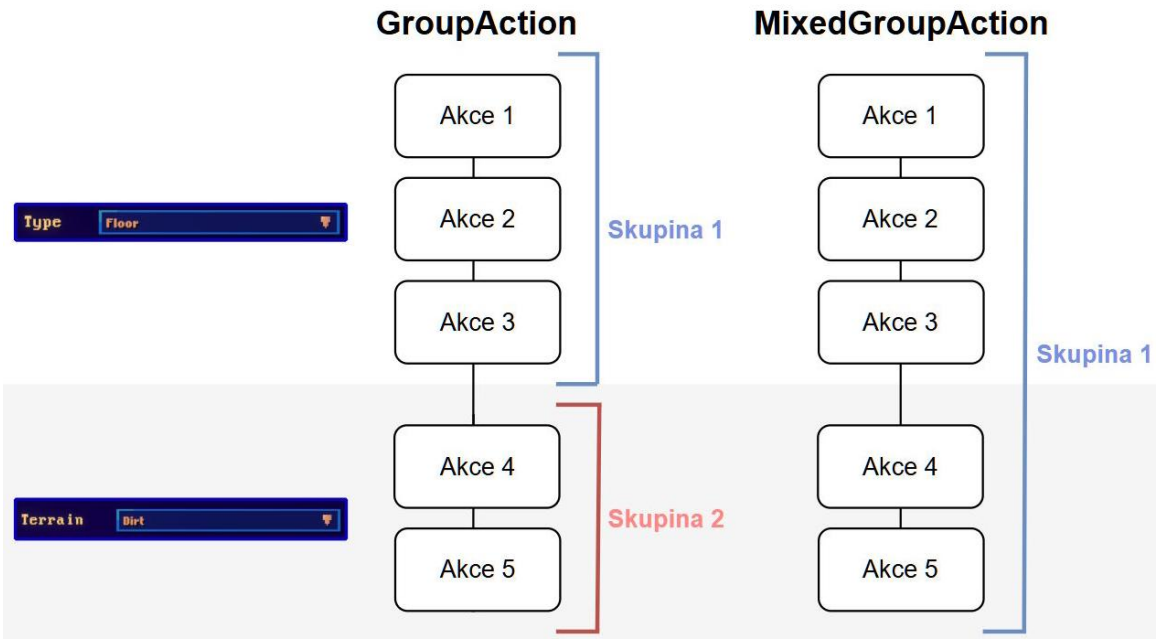
Zdrojový kód 11 - Metoda DecideGroupingResponseFor()

Na ř. 3 je vidět, že pokud je shlukování vypnuté nebo si programátor při volání *AddAndExecute()* metody nepřál akci přidat do skupiny, je akce prostě přidána do undo historie (ř. 22).

Když je akcím shlukování povoleno, přesune se algoritmus dál. Nejdříve se na ř. 5 zkontroluje, jestli je nějaká skupina právě rozpracovaná. Pokud není, vytvoří se nová instance shlukové akce (ř. 7), do níž se vybraná akce přidá (ř. 8), po čemž se metoda ukončí. Do undo historie není zatím nic přidáno.

Na ř. 7 je rovněž možné zaznamenat, že se může vytvořit jeden ze dvou možných druhů shlukovacích akcí: **GroupAction** a **MixedGroupAction**. Ty se chovají téměř identicky. V obou je možné shromažďovat další akce a zároveň se k nim dá přistupovat jako k jakékoli jiné akci díky jejich využívání rozhraní *IAction* (viz obrázek 18).

Velkým rozdílem odlišujícím od sebe obě třídy je, že: *GroupAction může obsahovat pouze akce, ovlivňující stejný konstrukt*. Pro lepší pochopení této věty je vhodné se podívat na obrázek 19, který rozdíl vizualizuje.



Obrázek 19 - GroupAction vs MixedGroupAction

Obrázek obsahuje dva rozdílné konstrukty (vlevo). V tomto případě se jedná o dva odlišné roztahovací seznamy, na nichž uživatel provedl pět akcí (5x nastavil hodnotu). První tři akce provedl na prvním konstruktu a další dvě na druhém. Tyto akce se postupně slučují do skupiny. Rozdíl tohoto slučování však je v tom, že když GroupAction narazí na akci nad jiným konstruktem, tak **ihned ukončí právě otevřenou skupinu**, ihned ji vloží do undo historie a započne novou. MixedGroupAction žádnou takovou kontrolu nedělá a shlukování se pro ni musí ukončit manuálně.

Teď už je možné se přesunout zpět ke zdrojovému kódu 11 na ř. 12, kde se akce přidá do otevřené skupiny, ale pouze pokud má stejný konstrukt jako předešlá akce nebo metoda pracuje s MixedGroupAction. Jestliže byla úspěšně přidána do skupiny, metoda se ukončí.

Nakonec na ř. 18 je metoda **AddCurrentGroupToUndo()**, která proběhne, když je nějaká skupina otevřená a nová akce do ní přidána stále nebyla. Tato metoda neudělá nic jiného, než že vezme právě otevřenou skupinu a přidá ji do undo historie. Co se týče právě zpracované akce, tak ta je přidána do undo historie na ř. 21.

Vnější interakce

Systém shlukování akcí třída AHS spravuje automaticky, avšak programátor má možnost jeho chování ovládat pomocí několika veřejných metod. Těmi důležitými jsou:

- **StartNewGroup(bool allowDifferentConstructs)** – Ukončí otevřenou skupinu a započne novou. Booleovská hodnota určuje, jestli nová skupina bude moci obsahovat mix konstruktů (tedy jestli využije MixedGroupAction).
- **EndCurrentGroup()** – Ukončí otevřenou skupinu a přesune ji do undo historie.
- **EnableGroupingBehaviour()** – Zapne/vypne veškeré shlukování akcí.

Právě díky těchto metod bylo možné vyřešit problém podpory tahání myši. Pro připomenutí, jak pro posuvník, tak pro kreslicí mřížku vždy bylo nutné rozpoznat, jestli uživatel myší nastavil hodnotu jedním tahem nebo několika kliknutími. Už toto tvrzení napovídá, jak daný problém vyřešit: *začít novou skupinu, když uživatel stiskne levé tlačítko, a ukončit skupinu, když tlačítko pustí*.

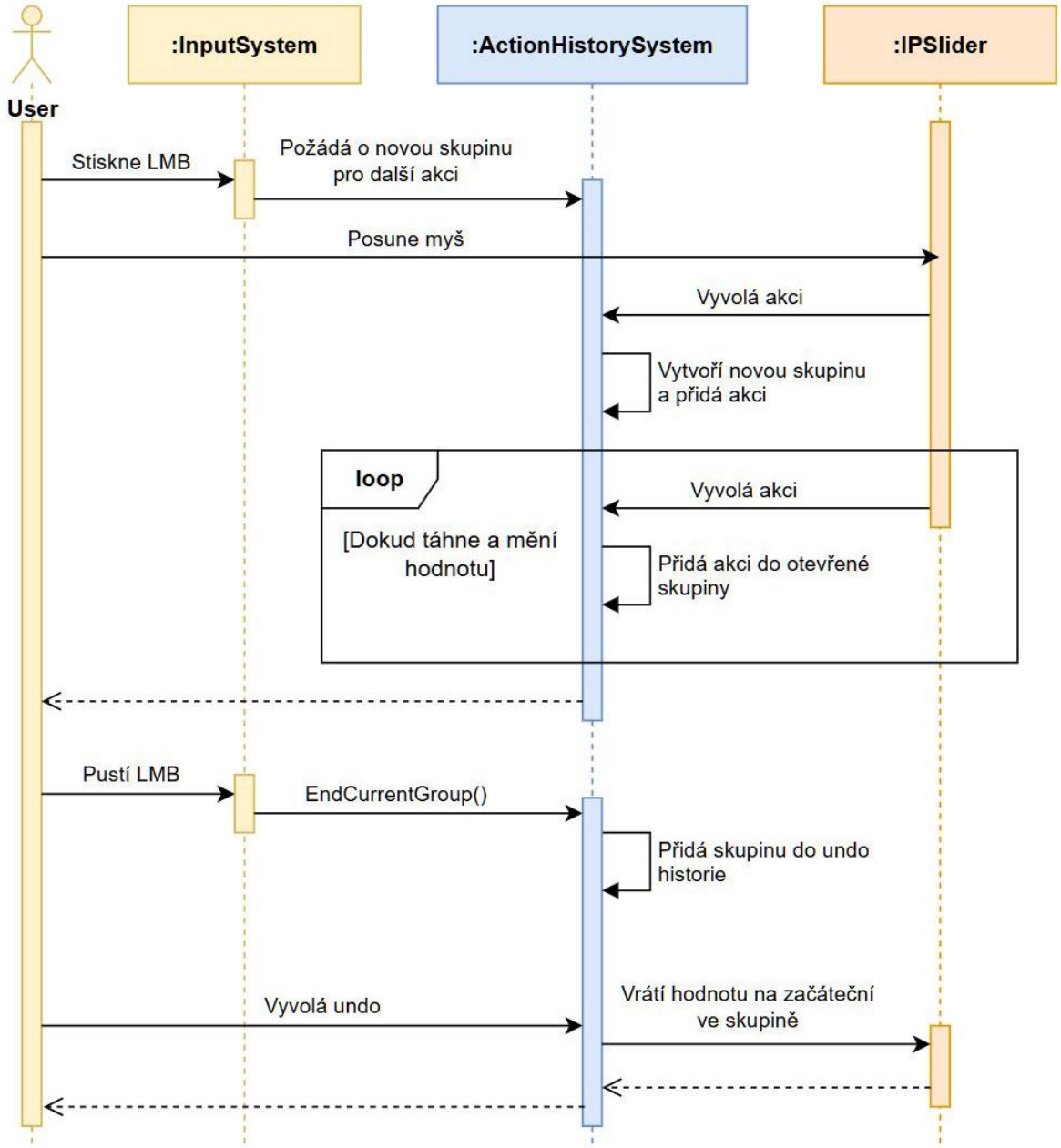
Tohoto cíle dosáhneme propojením potřebných metod (StartNewGroup() a EndCurrentGroup()) s potřebnými ovládacími prvky. Kód jejich napojení je možné vidět ve zdrojovém kódu 12.

```
1. UI.Select.OnPress -= ActionHistorySystem.StartNewGroup;
2. UI.Select.OnRelease -= ActionHistorySystem.EndCurrentGroup;
3. UI.ContextSelect.OnPress -= ActionHistorySystem.StartNewGroup;
4. UI.ContextSelect.OnRelease -= ActionHistorySystem.EndCurrentGroup;
5.
6. UI.Select.OnPress += ActionHistorySystem.StartNewGroup;
7. UI.Select.OnRelease += ActionHistorySystem.EndCurrentGroup;
8. UI.ContextSelect.OnPress += ActionHistorySystem.StartNewGroup;
9. UI.ContextSelect.OnRelease += ActionHistorySystem.EndCurrentGroup;
```

Zdrojový kód 12 - Napojení shlukování akcí na myš

K propojení se používá systém C# eventů (viz kapitola Observer 1.3.3). Nalevo jsou eventy, které se vyvolají po stisknutí/puštění input akcí Select (která může být aktivována levým tlačítkem myši) a ContextSelect (může být aktivována pravým tlačítkem myši). Napravo jsou pak požadované metody. Tento kód je volán v metodě, která v projektu nastavuje potřebné hodnoty input systému a jelikož tato metoda může být volána v průběhu programu opakovaně, je nutné vždy nejdříve odběry odhlásit (ř. 1 až 4), aby nedocházelo k jejich multiplikaci a metody se tak nevolaly opakovaně.

Tímto způsobem byl problém podpory tahání myší vyřešen. Obrázek 20 už jenom shrnuje princip fungování systému shlukování akcí. Jedná se o sekvenční diagram, vyobrazující situaci, kdy uživatel nastavuje hodnotu posuvníku a následovně vyvolává undo operaci. Stejným způsobem by se shlukování chovalo i při nastavování dlaždic na *kreslicí mřížce*.



Obrázek 20 - Shlukování akcí pro nastavení posuvníku

4.3 Připomínky a limitace

Tato část již obsahuje jenom pár připomínek, které vyšly ze zkušeností, získaných při implementaci systému.

4.3.1 Kdy systém zavést do projektu?

Pokud zvažujete Undo/Redo systém zavést do svého projektu, je ideální to provést co nejdřív. Každý konstrukt, který by měl být Undo/Redo systémem podporován, musí obsahovat způsob, jak vybrané proměnné nastavovat hodnotu pomocí nějaké veřejné metody. Taky musí mít někde uloženou svou předešlou hodnotu této proměnné, aby bylo možné její stav potom pomocí operace undo vrátit.

Pokud se Undo/Redo do nějakého projektu přidává dodatečně, je nutné počítat s extensivním refaktorováním a přidáváním funkcionality.

4.3.2 Co když je v systému otevřená skupina a vyvolá se undo?

Po vyvolání undo operace se otevřená skupina automaticky uzavře, vloží do undo historie a provede se na ní undo operace. O všechny tyto aktivity se postará metoda *AddCurrentGroupToUndo*(), umístěna v metodě Undo() (viz zdrojový kód 7).

4.3.3 Limitace

Systém je schopen pracovat pouze s jedním zásobníkem pro undo a redo historii. Pokud by bylo potřeba držet několik takových historií, bylo by nutné systém rozšířit. Možné by bylo vhodné rovnou přidat nějakého správce undo/redo historií. Ten by držel jednotlivé instance AHS (třída by již nebyla statická), mezi kterými by bylo možné přepínat dle vnějších podmínek.

Příkladový projekt obsahuje několik editorů (viz obrázek 12). Dávalo by například smysl mít jednu historii pro každý editor v případě, že by bylo možné tyto editory mít otevřené najednou. Toto ovšem projekt nepodporuje, tudíž řešení, kdy se po opuštění editoru obě historie smažou, je zcela dostačující.

5 IMPLEMENTACE KLÁVESOVÝCH ZKRATEK

Tato kapitola se zaměří na to, jakým způsobem byla naimplementovaná funkcionalita klávesových zkratek. Nahlédne se zde na způsob, jak propojit input akce systému od Unity s generickými akcemi specifickými pro projekt. Většina dokumentace však bude využita pro popis přemapování kombinací vazeb. To bude zahrnovat:

- Popis UI prvku, jenž umožní přemapování jedné kombinace pro jednu akci.
- Jak byla vyřešena možnost přemapování s variabilním počtem tlačítek (viz obr. 11).
- Jak se řešila situace nalezení již použité kombinace.
- Zmínka o způsobu uložení aktuálního mapování input akcí do souboru.

Nakonec se proberou limitace aktuální podoby systému.

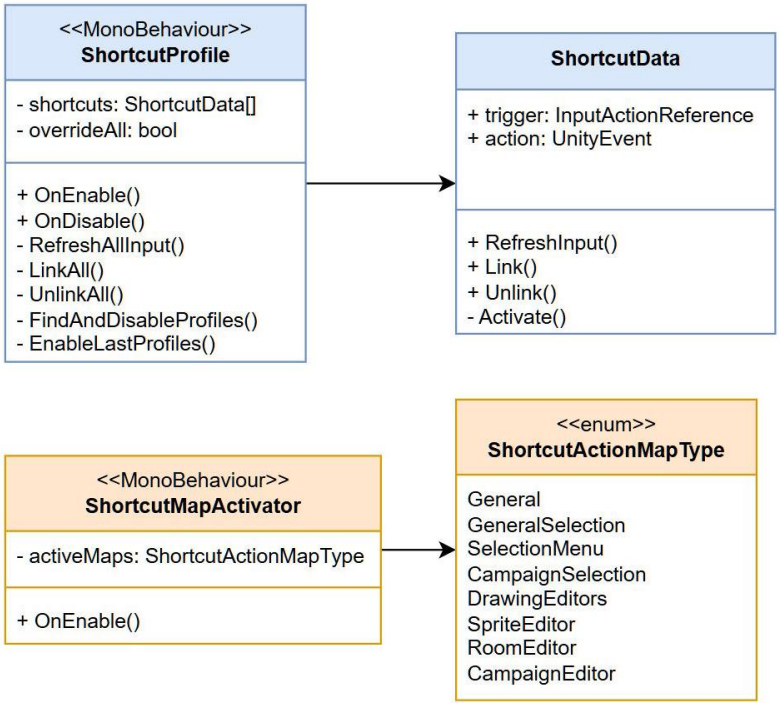
5.1 Propojení zkratek a akcí

Tento systém zajišťuje, že je v projektu možné stisknout například kombinaci kláves CTRL+Z, čímž se aktivuje například operace undo. Ve shrnutí tento systém umí:

- K jedné input akci napojit několik generických akcí (metod). Ty už budou nějakým způsobem ovlivňovat aktivní prvky (třeba proveď undo, ulož provedené změny, apod).
- Přepínat mezi několika akčními mapami (tedy například pro editor místností chceme používat jiné zkratky než třeba pro editor zbraní).

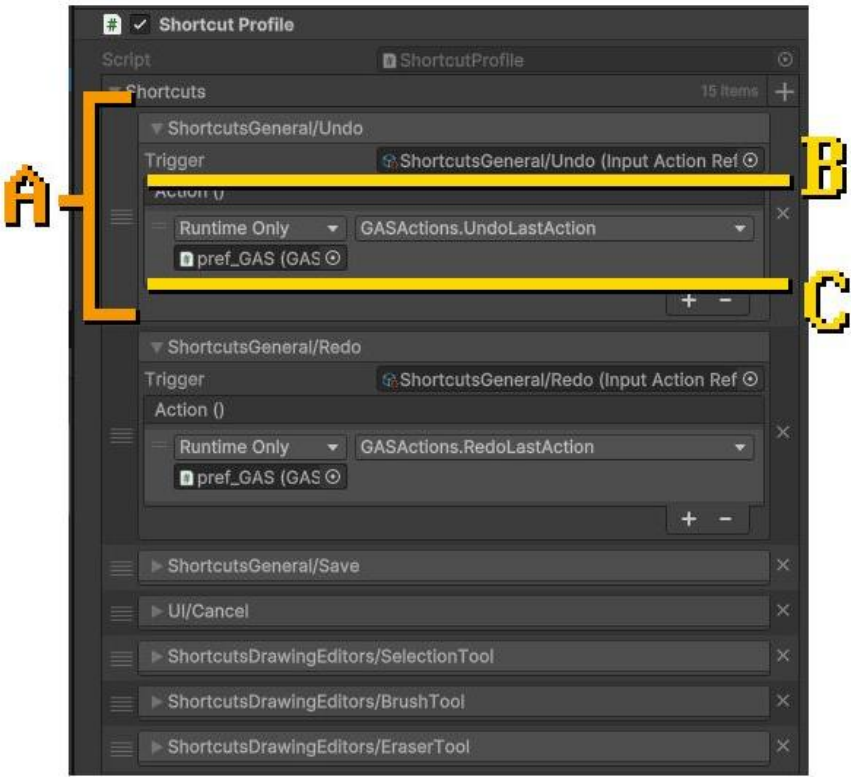
5.1.1 Kostra systému

Obrázek 21 znázorňuje diagram tříd systému propojení zkratek a akcí. Jeho základ je postavený na dvou hlavních komponentech.



Obrázek 21 – Diagram tříd propojení zkratek s akcemi

Hlavním aktérem systému je třída **ShortcutProfile**. Skrz tu je designér schopen v inspektoru propojovat input akce a generické akce. Jak takové nastavení v inspektoru vypadá je ukázáno na obrázku 22.

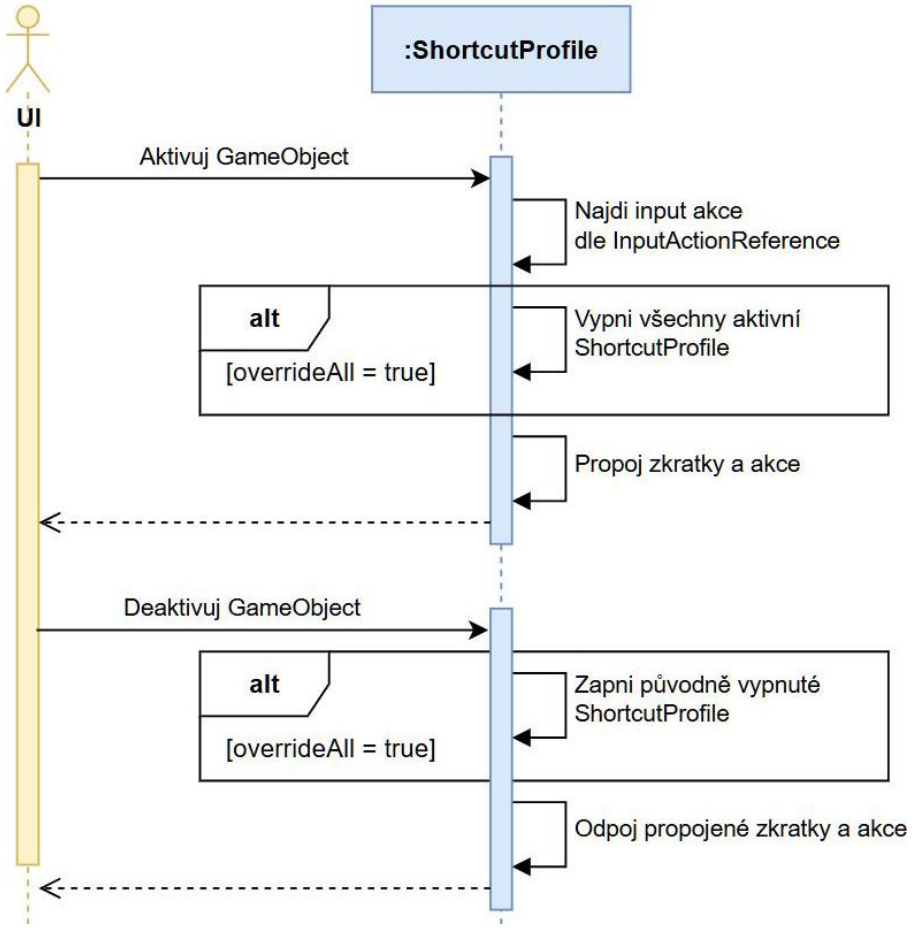


Obrázek 22 - Komponent ShortcutProfile

ShortcutProfile obsahuje pole *shortcuts*, jenž v sobě drží položky typu ShortcutData. Každá jedna položka reprezentuje link mezi akcemi (A) a umožňuje designérovi nastavit dvě hodnoty, kde každá reprezentuje jednu stranu propojení. Tyto hodnoty jsou:

- **Trigger** (B) – vyžaduje nějakou hodnotu typu *InputActionReference*, tedy třídu, která odkazuje na specifickou input akci, patřící pod určitou akční mapu [24]. Dává smysl zde dosadit třeba input akci undo, redo, uložení, vytvoř nový, atd.
- **Action** (C) – je představena třídou UnityEvent, které může designér přiřadit jednu nebo více metod. Po jeho vyvolání se všechny metody spustí. Jedná se o obdobu C# eventů, avšak na rozdíl od nich je možné s ní operovat v inspektoru. Tady se nastavuje, co se provede, když uživatel aktivuje přiřazenou input akci. Třeba když stiskne kombinaci pro undo, tak se vyvolá metoda UndoLastOperation() na přiřazeném MonoBehaviour komponentu GASActions.

Princip fungování systému za běhu je velmi jednoduchý. Jeho postup je představen sekvenčním diagramem na obrázku 23.



Obrázek 23 - Sekvenční diagram fungování propojovacího systému

Je důležité zmínit, že ShortcutProfile funguje, pouze pokud je jeho GameObject aktivní. Jinak by mohlo docházet k problémům jako je například aktivování několika rozdílných akcí stejnou zkratkou.

5.1.2 Prioritizace profilů

V editačním programu může nastat situace, že uživatel má otevřený editor, který má přiřazený svůj vlastní ShortcutProfile. Během práce s editorem však na uživatele může vyskočit modální okno, které má také svůj vlastní ShortcutProfile (třeba se chce, aby zkratkou ESC se okno zavřelo). *Jak zajistit, že když uživatel zkratku stiskne, tak se neaktivuje akce, přiřazena ShortcutProfile, patřící otevřenému editoru, ale aktivuje se pouze ta, která patří modálnímu oknu?*

Tento problém pomáhá vyřešit booleovská hodnota **overrideAll**, patřící pod ShortcutProfile. Ta je určena přesně pro takové situace, kdy se má objevit ShortcutProfile, který má vyšší prioritu než všechny právě aktivní. V sekvenčním diagramu na obrázku 23 jde vidět, že se provede extra aktivita, pokud je tato hodnota nastavená na *true*.

Když se ShortcutProfile s overrideAll = true aktivuje, vyhledá všechny existující a aktivní profily, kterým deaktivuje činnost. Tím zabrání vyvolávání nechtěných akcí. Odkazy na tyto profily si uloží do paměti a jakmile dojde k jeho deaktivaci, původní profily opět aktivuje.

Toto řešení funguje *řetězovitě*, takže pokud se po aktivaci modálního okna č. 1 objevilo například další okno č. 2, které taky mělo overrideAll = true, tak tento nejnovější profil deaktivuje profil okna č. 1. Po zavření okna č. 2 se opět aktivuje profil okna č. 1, a potom, co se i to zavře, se aktivují původně aktivní profily editoru.

5.1.3 Přepínání akčních map

Ať už z důvodu lepší organizace nebo protože editor input akcí již není schopen utáhnout množství zkratek, designér bude chtít input akce rozdělit do několika akčních map (třeba co editor, to mapa). Pokud tak udělá, asi určitě nebude chtít, aby všechny akční mapy byly aktivní zároveň. To by mohlo způsobovat volání nechtěných vstupů.

Tento problém v systému řeší komponent **MapActivator**. Ten plní jen jednu, velmi prostou funkci: *se svou aktivací zapne designérem vybrané akční mapy*.

O tom, které mapy aktivuje, rozhoduje nastavení parametru *activeMaps*. Jedná se enum typu **ShortcutActionMapType**, jehož hodnoty jsou specifické pro projekt a programátor je tudíž musí zadat na začátku manuálně (pro příkladový projekt existuje mapa zkratek pro obecné akce, téměř každý editor a výběrová menu, viz obrázek 21). Tento enum využívá atribut *Flags*, který mu umožňuje nastavit několik hodnot najednou tím, že jej převede na bitové pole [25].

5.2 Přemapování zkratek

Systém, který umožňuje uživatelům měnit kombinace tlačítek, které je třeba stisknout pro vyvolávání akcí. Podporuje:

- Měnit kombinace pro klávesnici, myš i herní ovladače.
- Dva vstupy pro každou input akci a pro každé podporované zařízení. Kdyby uživatel chtěl například provést undo zkratkou CTRL+Z a zároveň taky levým bočním tlačítkem myši.
- Kombinace složené z 1-3 tlačítek, kdy je možné kombinace o jednom tlačítku převést na kombinace s více tlačítky.
- Kompozity, tedy vazby držící v sobě další vazby. Ty se (hlavně ve hrách) využívají pro pohybové klávesy (WASD) nebo pro přibližování pomocí kolečka myší.
- Ukládání do JSON souboru, umožňující uchování informace a externí editaci.

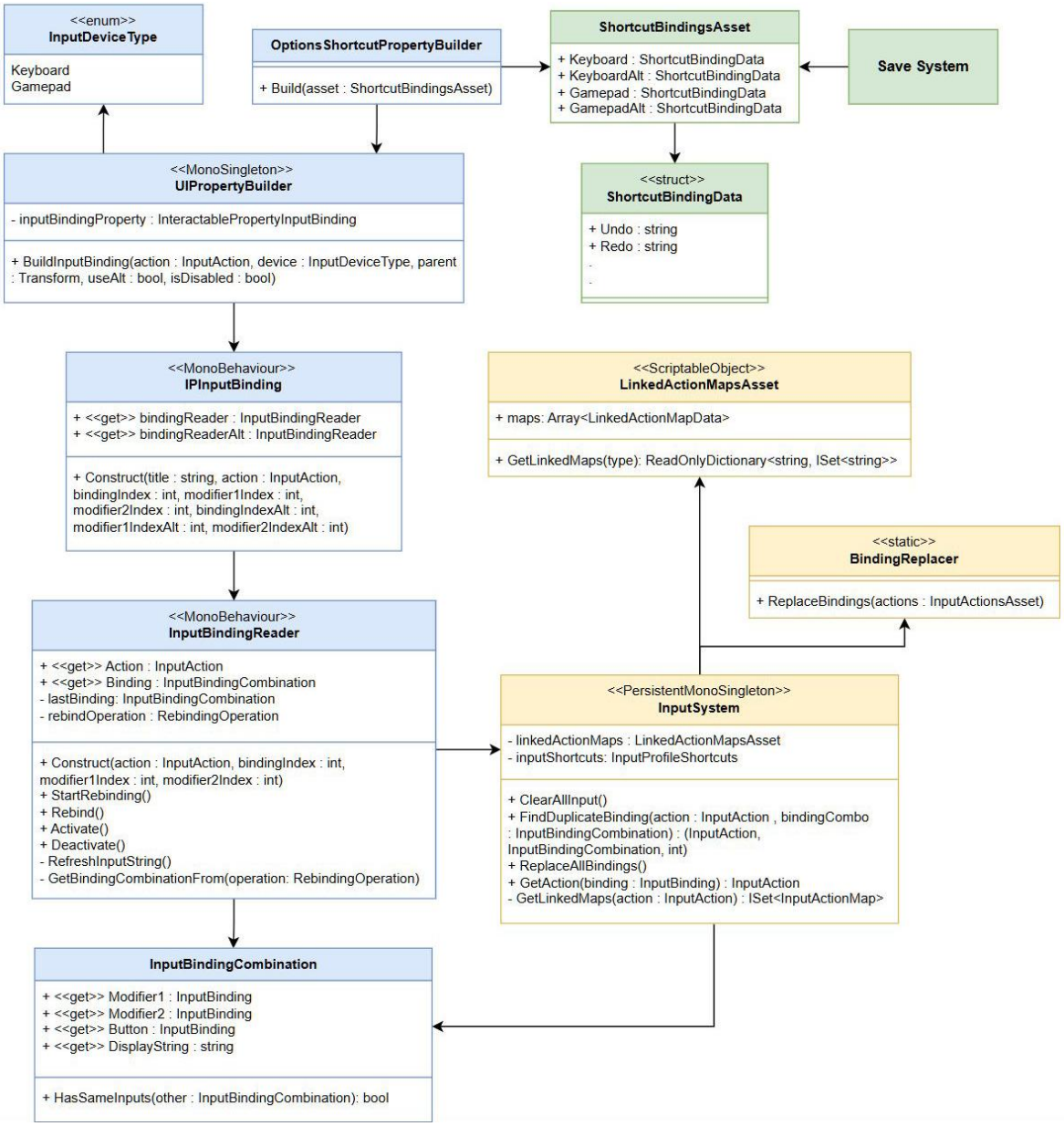
Nejdříve se nahlédne na základní postup změny kombinace pro vybranou zkratku (tzv. přemapování nebo rebinding), následovat budou pak popisy řešení těch největších problémů. Konec kapitoly se zaměří na existující limitace systému.

5.2.1 Kostra systému

Na obrázku 24 je vyobrazen diagram tříd těch nejdůležitějších prvků systému přemapování klávesových zkratek. Jelikož se jedná o celkem komplexní systém, byl diagram pro lepší orientaci rozdělen barevně na tři části:

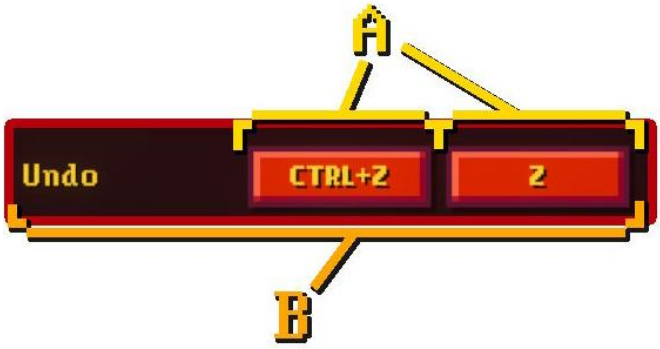
- **Modrá část** – Představuje UI a datovou část systému, točící se převážně okolo třídy InputBindingReader, která aktivuje samotné přemapování.

- **Žlutá část** – Input Systém, tedy systém, který v projektu spravuje input akce a vstupy.
- **Zelená část** – External Storage, tedy systém, který načítá a ukládá změněné zkratky na úložiště.



Obrázek 24 – Diagram tříd systému přemapování zkratek

Srdcem celého systému je třída **InputBindingReader**. Ta představuje UI prvek, skrz který je uživatel schopen měnit kombinaci tlačítek pro specifickou input akci. Podoba tohoto prvku v aplikaci je ukázána obrázkem 25.



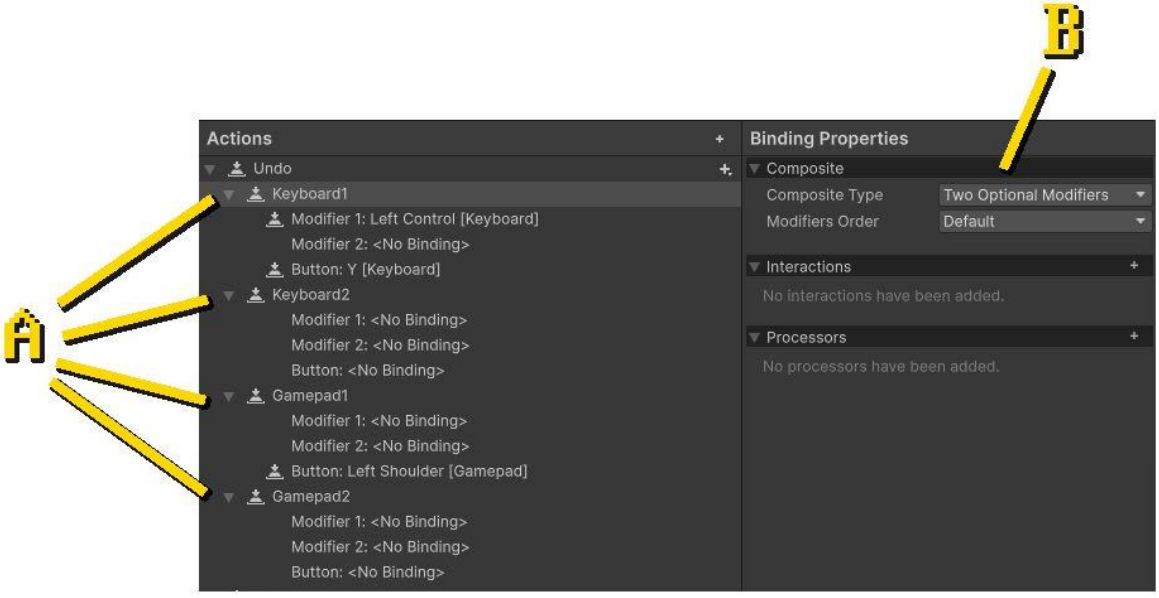
Obrázek 25 - UI prvek přemapování zkratky. Celý je reprezentován třídou IPInputBinding

Uživatel s ním pracuje tak, že stiskne přemapovací tlačítko (A) s textem aktuální kombinace. To změní svou podobu a uživatele vyzve, aby zadal novou kombinaci. Po jejím zadání se tlačítko vrátí do původní stavu a nová kombinace se uloží.

Nutno dodat, že samotný InputBindingReader je reprezentován pouze přemapovacím tlačítkem. Plnohodnotný UI prvek obsahující titulek, tlačítko přemapování hlavní vazby a tlačítko přemapování alternativní vazby (B), je reprezentován třídou **IPInputBinding**.

5.2.2 Jak připravit input akci

Všechny input akce, které budou moci být přemapovány, musí mít určitou podobu svých vazeb, aby v systému nedocházelo k chybám. Na obrázku 26 je možné vidět případ pro systém ideálně navržené input akce a jejích vazeb.



Obrázek 26 - Jak má vypadat input akce

Je nutné, aby každá input akce měla *alespoň jednu vazbu pro každé vstupní zařízení*. Pokud hodlá programátor ve svém projektu využívat i sekundární (alternativní) vazby, je nutné mít vazby dvě. Také musí zajistit, aby existovala vazba pro obě podporovaná zařízení (A).

Pokud bude chtít programátor využít možnost přemapování s variabilními kombinacemi, musí zajistit, aby vazby využívaly kompozitní typ **TwoOptionalModifiersComposite** (B). Jedná se o nový typ kompozitu, vytvořený speciálně pro systém, který slouží hlavně jako označovač vazeb, které takové chování budou dovolovat.

Kromě toho, že je tento typ detekován přemapovacím systémem jako speciální kompozit, je jeho speciální vlastností se aktivovat právě tehdy, když je *alespoň jedno z jeho tlačítek stisknuto*. Tím se odlišuje od například vestavěného TwoModifiersComposite, který se aktivuje jen tehdy, když jsou všechna jeho tlačítka aktivována.

5.2.3 Jak vytvořit InputBindingReader

Než se v práce zaměří na to, jak vlastně samotné přemapování funguje, bylo by vhodné vysvětlit, jakým způsobem se vlastně tento UI prvek musí vytvořit. Jeho vytvoření a samotnou inicializaci je nutné provést **skrz kód**. Prosté přetáhnutí jeho prefabu do požadovaného menu nestačí. Jedná se o jednu z limitací systému (více v kapitole 5.3).

Každou novou čtečku vstupů (InputBindingReader) je nutné vytvořit přes metodu **BuildInputBinding()** ve třídě **UIPropertyBuilder**. Jedná se o továrnu na UI elementy, která specificky pro čtečku vstupů nastavuje hodnoty a určuje, *kolik* čteček přesně se má vlastně vytvořit, pokud vybraná vazba je typu kompozit. Kód této tovární metody je představen ve zdrojovém kódu 13.

```
1. public void BuildInputBinding(InputAction action, InputDeviceType device, Transform parent, bool useAlt,
2.                               bool isDisabled = false)
3. {
4.     int bindingIndex = InputSystemUtils.GetBindingIndexByDevice(action, device);
5.     int bindingIndexAlt = useAlt ? InputSystemUtils.GetBindingIndexByDevice(action, device, true) : -1;
6.
7.     if (action.bindings[bindingIndex].isPartOfComposite)
8.     {
9.         if (action.bindings[bindingIndex - 1].IsTwoOptionalModifiersComposite())
10.        {
11.            ConstructInputBinding(action.name, true);
12.            return;
13.        }
14.
15.        while (bindingIndex < action.bindings.Count && action.bindings[bindingIndex].isPartOfComposite)
16.        {
17.            string title = $"{action.name}{action.bindings[bindingIndex].name.Capitalize()}";
18.            ConstructInputBinding(title);
19.            bindingIndex++;
20.            bindingIndexAlt++;
21.        }
22.        return;
23.    }
24.    ConstructInputBinding(action.name);
25.}
```

Zdrojový kód 13 - Metoda BuildInputBinding() ve třídě UIPropertyBuilder

Jako vstup každá čtečka vstupů vyžaduje (ř. 1):

- Input akci, se kterou bude sloučena (tou může být třeba Proveď undo nebo Vyber tužku).
- Typ zařízení, pro které bude snímat vstupy (podporované jsou klávesnice + myš a herní ovladač).
- Svého budoucího rodiče v hierarchii (tedy kde bude v menu umístěna).
- Jestli má obsahovat i alternativní vstup. Pokud ano, bude mít svou podobu totožnou s obrázkem 25. Jinak bude obsahovat pouze jedno přemapovací tlačítko.
- Jestli má po konstrukci povolena přijímat vstupy uživatele.

Na ř. 4 je třeba zjistit index vazby, kterou bude čtečka přemapovávat. O tenhle úkol se postará pomocná metoda *GetBindingIndexByDevice()*, která vyžaduje jako vstupy požadovanou akci, zařízení, pro které má index najít, a jestli má vyhledat hlavní nebo alternativní vazbu. Na ř. 5 se tento index hledá pro alternativní vazbu.

Následně na ř. 7 se rozhodne, jestli je vazba pod indexem součástí nějakého kompozitu. Jestli ne, tak na ř. 24 prostě vytvoř čtečku pro řešenou vazbu. Jinak přejdi do závorek.

Nyní se bude rozhodovat o tom, s jakým druhem kompozitu algoritmus pracuje. Pokud je hlavní kompozit typu *TwoOptionalModifiersComposite*, který se využívá pro detekci klávesových zkratek (viz více 5.2.5), tak vytvoř pouze jednu čtečku, ale připrav její

parametry pro přítomnost možných modifikátorů (ř. 9-13, upravenou konstrukci značí hodnota *true* na ř. 11).

Jinak, jestli je kompozit jiného druhu než TwoOptionalModifiersComposite, tak prostě vytvoř čtečku pro každou jeho podvazbu (ř. 15-21). Tímto způsobem se třeba převede 2D vektorový kompozit, držící pohybové klávesy hráče WASD, na čtyři různé čtečky.

Metodou **ConstructInputBinding()**, kterou proces vždy skončí, se potom už jen vytvoří instance prefabu, držící komponent *IPInputBinding*, kterému se nastaví vhodné hodnoty pomocí jeho vlastní metody *Construct()*. Mezi ně patří i nastavení jeho dvou InputBindingReader prvků.

5.2.4 Proces přemapování

Samotný proces přemapování započne, jakmile uživatel klikne na přemapovací tlačítko. Když tak udělá, vyvolá se v **InputBindingReader** metoda **StartRebinding()**. Její podoba je ukázána ve zdrojovém kódu 14.

```
1. public void StartRebinding()
2. {
3.     action.Disable();
4.     ui.ShowBindingDisplay();
5.     OnRebindStartAny?.Invoke(action, binding, buttonIndex);
6.     OnRebindStart?.Invoke();
7.     rebindOperation = action.PerformInteractiveRebinding(buttonIndex)
8.                             .OnCancel(_ => StopRebinding())
9.                             .OnComplete(FinishRebinding)
10.                            .OnMatchWaitForAnother((modifier1Index != -1 || modifier2Index != -1) ?
11.                                                   EditorDefaults.Instance.InputWaitForAnother : 0)
12.                            .WithCancelingThrough("")
13.                            .WithTimeout(EditorDefaults.Instance.InputTimeout)
14.                            .Start();
15.}
```

Zdrojový kód 14 - Metoda StartRebinding() ve třídě InputBindingReader

Nejdříve se na ř. 3 input akce deaktivuje, aby mohla být přepsána. Pomocí ř. 4 se zobrazí UI, říkající uživateli, že má stisknout svou kombinaci. Přemapování pak už obstará vestavěná metoda *PerformInteractiveRebinding()*. Zjednodušeně: Tato metoda aktivuje přemapování pro vazbu, uloženou pod daným indexem, a začne naslouchat novým vstupům. Všechny parametry a výsledky přemapování ukládá do datového typu *RebindingOperation*, který následně posílá dál do ukončovacích metod.

Tyto ukončovací a další metody, nastavující parametry přemapovací operace, je možné vidět na ř. 8-14. Co se týče jejich funkcionality, tak:

- *OnCancel()* – Co se stane, když je proces přemapování ukončen bez vhodného výsledku.
- *OnComplete()* – Co se stane, když došlo k úspěšnému ukončení přemapování.
- *OnMatchWaitForAnother()* – Jak dlouho má proces počkat potom, co uživatel namapoval poslední tlačítko. Tato metoda je kritická pro možnost snímání variabilních kombinací. Je možné vidět, že čas je jí nastaven pouze pokud se počítá s možností více tlačítek. Jinak je čas nastavený na 0.
- *WithCancelingThrough()* – Kterým tlačítkem se proces přemapování zruší (většinou bývá ESC). Jelikož projekt vyžaduje možnost mapovat zkratky na všechny klávesy, tedy i ESC, je tato vlastnost vypnuta.
- *WithTimeout()* – Jak dlouho nechat proces přemapování běžet.
- *Start*() – Začne proces přemapování.

Jestliže uživatel zadal svou kombinaci správně, proces se ukončí a přesune do metody **FinishRebinding()**, ukázané ve zdrojovém kódu 15.

```
1. void FinishRebinding(InputActionRebindingExtensions.RebindingOperation operation)
2. {
3.     InputBindingCombination c = GetBindingCombinationFrom(operation);
4.     Rebind(c);
5.
6.     (InputAction duplicateAction, InputBindingCombination duplicateCombination, int duplicateIndex) =
7.      InputSystem.Instance.FindDuplicateBinding(action, binding);
8.     if (duplicateAction != null)
9.     {
10.        ModalWindowBuilder.Instance.OpenWindow(new ModalWindowData.Builder()
11.            .WithMessage($"…")
12.            .WithAcceptButton("Override", () => OverrideDuplicateBinding(duplicateAction,
13.                                                                         duplicateCombination,
14.                                                                         duplicateIndex))
15.            .WithDenyButton("Revert", RevertBinding)
16.            .Build());
17.    }
18.    else ActionHistorySystem.AddAndExecute(new UpdateInputBindingAction(this, binding, lastBinding,
19.                                                                        Rebind));
20.    StopRebinding();
21.}
```

Zdrojový kód 15 - Metoda FinishRebinding()

Ta převážně řeší problémy, ke kterým může během přemapování dojít. Na ř. 3 se metodou *GetBindingCombinationFrom()* získá informace o kombinaci tlačítek, kterou uživatel vlastně stisknul (více o této metodě v 5.2.5). **InputBindingCombination** je datová třída, která představuje stisknutou kombinaci *Modifikátor 1* + *Modifikátor 2* + *Tlačítko*. Veřejnou metodou *Rebind()* se potom přepíše vazba v input akci.

Na ř. 6-17 se řeší případ nalezené duplicity, ale této části se věnuje kapitola 5.2.6.

Nakonec se na ř. 18 přemapování zaregistruje do Undo/Redo systému a na ř. 20 se operace definitivně ukončí metodou *StopRebinding()*. Ta opět aktivuje input akci, resetuje rebindOperation a vypne UI, které říkalo uživateli, že má zadat novou kombinaci.

5.2.5 Problém variabilního počtu tlačítek

V dalších kapitolách se nahlédne na pár problémů, které implementovaný způsob systému zkratek řeší. Tím asi největším byl *problém neschopnosti přemapovat kombinaci zkratky složenou z určitého počtu tlačítek na kombinaci s jiným počtem tlačítek*. Tento problém byl již detailně popsán v kapitole návrhu (3.1.2, str. 30), takže jen pro připomenutí: Takováto funkcionalita není v současné verzi Input Systému (1.14) podporována.

Nejbližší, co Unity k této funkcionalitě ve svém systému má, jsou kompozity *OneModifierComposite* a *TwoModifiersComposite*. Obojí představují vazby, schopné přijmout určitý počet tlačítek dohromady (první 2, druhá 3), avšak musí obsahovat *přesně* tento počet. Nemají schopnost obsahovat v sobě prázdné vazby.

Tento problém implementovaný systém řeší pomocí dříve zmíněného speciálního kompozitu TwoOptionalModifiersComposite (kapitola 5.2.2) a následujícího postupu.

Řešení

V dříve zmíněné metodě StartRebinding() se nastavuje pro přemapování vlastnost OnMatchWaitForAnother(), která byla již vysvětlena v minulé kapitole. Dávala procesu prostor nasnímat všechna tlačítka, která uživatel pro kombinaci zadal a přesně tuhle schopnost systém využívá k získání celé nasnímané kombinace.

Ve FinishRebinding() metodě bylo možné vidět, že před samotným přemapování systém nejdříve potřebuje převést získanou informaci do vhodné datové podoby, (zdrojový kód 15, ř. 3). To provádí metodou **GetBindingCombinationFrom()**, která je představena ve zdrojovém kódu 16.

```
1. private InputBindingCombination GetBindingCombinationFrom(RebindingOperation operation)
2. {
3.     InputBindingCombination newBinding;
4.     List<InputControl> controls = operation.candidates.Where(c => c is ButtonControl
5.                                                                 and not AnyKeyControl
6.                                                                 and not DiscreteButtonControl).ToList();
7.     if (modifier1Index == -1 || modifier2Index == -1)
8.     {
9.         newBinding = new InputBindingCombination.Builder().From(binding)
10.                                                 .WithButton(controls[0].path.FormatForBindingPath())
11.                                                 .Build();
12.         return newBinding;
13.     }
14.
15.     newBinding = controls.Count switch {
16.         1 => new InputBindingCombination.Builder().From(binding)
17.                                                 .ClearPaths()
18.                                                 .WithButton(controls[0].path.FormatForBindingPath())
19.                                                 .Build(),
20.         2 => new InputBindingCombination.Builder().From(binding)
21.                                                 .ClearPaths()
22.                                                 .WithModifier1(controls[0].path.FormatForBindingPath())
23.                                                 .WithButton(controls[1].path.FormatForBindingPath())
24.                                                 .Build(),
25.         _ => new InputBindingCombination.Builder().From(binding)
26.                                                 .ClearPaths()
27.                                                 .WithModifier1(controls[0].path.FormatForBindingPath())
28.                                                 .WithModifier2(controls[1].path.FormatForBindingPath())
29.                                                 .WithButton(controls[2].path.FormatForBindingPath())
30.                                                 .Build()
31.     };
32.     return newBinding;
33. }
```

Zdrojový kód 16 - Metoda GetBindingCombinationFrom

Jejím cílem je vytvořit datovou třídu, držící informaci o zadané kombinaci. Toho dosahuje tak, že nejdříve na ř. 4 kolekci získaných kandidátů vyfiltruje způsobem, aby obsahovala pouze samotná tlačítka (ne speciální konstrukty jako AnyKey, tedy stisknuto jakékoliv tlačítko).

Dále se na ř. 7-13 ptá, jestli je tato čtečka schopna přijímat modifikátory. Pokud ne, ihned vytvoří a vrátí kombinaci, obsahující pouze nejnovější tlačítko.

Jestli ovšem čtečka modifikátory přijímá, je třeba zjistit počet stisknutých tlačítek a na základě toho kombinaci vytvořit. O to se starají ř. 15-30. Je také vidět, že třída InputBindingCombination má v sobě zabudovanou vlastní třídu *Builder*, která umožňuje programátorovi snadnější konstrukci nové kombinace. Tato třída byla navržena podle návrhového vzoru builder, který zpřehledňuje inicializaci tříd a je často užíván i v samotném input systému. Pro doplnění, proměnná *binding* označuje aktuální kombinaci, uloženou ve čtečce¨, která právě zprostředkovává přemapování. Po vytvoření kombinace se data z ní využijí v metodě *Rebind()* (zdrojový kód 15, ř. 4), kterou se přepíšou vazby v řešené input akci.

Takhle funguje přepisování vazeb. Pro plně funkční řešení je ale potřeba ještě dořešit jeden problém: **Unity neumí pracovat s TwoOptionalModifiersComposite**. Co se tím myslí,

dokáže dobře ukázat následující tvrzení: *Máme dvě vazby, kdy obě jsou typu TwoOptionalModifiersComposite. První je nastavena na hodnotu CTRL+U, druhá má hodnotu U. Jak rozpoznat, která vazba byla vyvolána po stisknutí klávesy U?*

Jelikož TwoOptionalModifiersComposite se aktivuje hned, jakmile je alespoň jedno jeho tlačítko stisknuto, spustily by se obě vazby. Takový výsledek je ovšem nevhodný. Jak ho tedy vyřešit?

Možných řešení bylo několik. Nakonec se ale vybralo prosté **nahrazení TwoOptionalModifiersComposite** vestavěnými kompozity v momentě, kdy už uživatel nepotřebuje dále vstupy přemapovávat. Myslí se tím, že uživatel vždy upravuje zkratky a vstupy v nějakém menu nastavení. Po svých změnách toto menu opustí a zkratky dále pak využívá v jiných částech aplikace. Jelikož Unity má již vestavěné kompozity pro 1-3 tlačítka (*Binding, OneModifierComposite, TwoModifiersComposite*) a umí s nimi pracovat, dává smysl v momentu opuštění menu prostě vlastní kompozitu nahradit těmito vestavěnými.

O nahrazení se stará statická třída **BindingReplacer**, která, pomocí své veřejné metody *ReplaceAllBindings(),* nahradí všechny TwoOptionalModifiersComposite za vestavěné varianty. Příkladem, kombinaci CTRL+U nahradí OneModifierComposite a kombinaci CTRL+SHIFT+U kompozitem TwoModifiersComposite.

TwoOptionalModifiersComposite však bude nutné zase vrátit, pokud uživatel bude potřebovat znovu poupravit vstupy. Možným řešením by bylo napsat algoritmus, jenž by převáděl vestavěné kompozity na TwoOptionalModifiersComposite. Jelikož je ale systém schopen změny ukládat na externí úložiště, tak se snadnějším řešením ukázalo prostě asset vstupů vygenerovat znova. Tento úkol spadá na metodu *ClearAllInput()* ve třídě InputSystem. Ta je volána v momentě připravení menu nastavení.

5.2.6 Problém nalezené duplicity

Obecně, v programech uživatel většinou nechce, aby stejná kombinace spouštěla dvě nebo dokonce více akcí najednou (například aktivovala undo a redo zároveň). Proto je potřeba pro tento problém zajistit nějaký limit.

Řešení – nalezení duplicity

Systém tento problém začíná řešit, jakmile uživatel zadal novou kombinaci v přemapování a algoritmus se přesunul do metody FinishRebinding(). Ve zdrojovém kódu 15, na ř. 6 a 7,

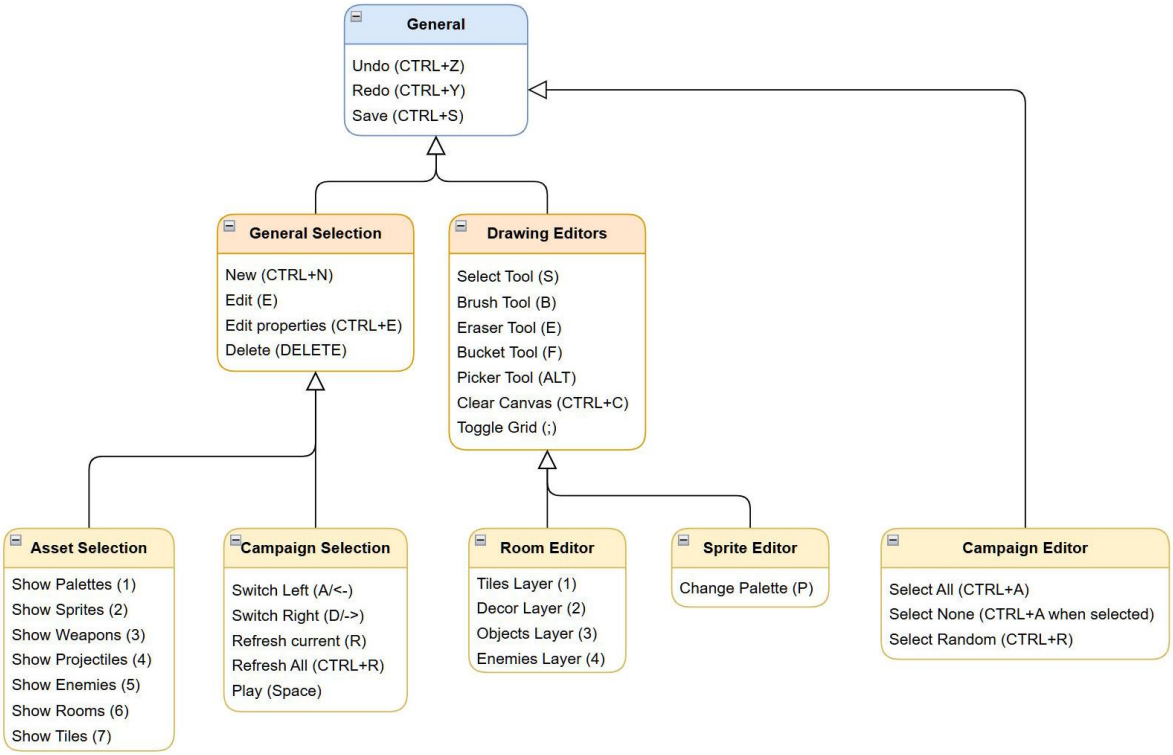
šlo vidět, že byla volána metoda **FindDuplicateBinding()** nad třídou *InputSystem*. Jejím úkolem je pokusit se najít možnou duplicitní kombinaci, přiřazenou stejné nebo i jiné input akci. Pokud nějakou takovou kombinaci najde, vrátí ji spolu s input akcí, ke které je vázána.

Tuto duplicitu ovšem nedává smysl vyhledávat ve všech akčních mapách. Měla by existovat logika, že stejná kombinace může provádět rozdílné akce v různých editorech/menu (například CTRL+R může ve výběrovém menu náhodně zvolit položky a přitom může v editoru zbraní resetovat všechny upravené položky).

Dávalo by tedy určitě smysl prohledávat stejnou akční mapu, ve které je zasazena upravovaná input akce. Co ale jiné mapy?

Jak bylo zmíněno v kapitole 1.2.2, jednou z vlastností akčních map je možnost jich mít aktivních několik najednou. Tuto vlastnost příkladový projekt využívá. Jelikož obsahuje velký počet editorů, obsahuje taky velký počet zkratek, kdy některé jsou čistě specifické pro vybraný editor a některé by měly být mezi editory sdíleny.

Hierarchii zkratek příkladového programu je možné vidět na obrázku 27. Tabulky představují akční mapy a šipky označují vztah „zahrnuje.“ (Například mapa *Room Editor* má být aktivní zároveň s mapou *Drawing Editors* a *General*.)



Obrázek 27 - Hierarchie zkratek příkladového programu

Z existence hierarchie jde vyčíst, že by určitě nebylo vhodné, aby třeba input akce *Room Editor – Tiles Layer* měla stejnou kombinaci jako *General – Undo*, protože mají být aktivní zároveň, a tudíž by se překrývaly.

Tento problém byl vyřešen vytvořením ScriptableObject assetu, který drží informace o propojených mapách. V systému je to třída **LinkedActionMapsAsset**, která obsahuje kolekci propojení akčních map, ze které je možné vyčíst, se kterými dalšími mapami spolupracuje specifická akční mapa. K assetu vždy přistupuje *InputSystem* na začátku hledání duplicitní kombinace, kdy díky jeho pomoci je schopen vždy připravit správné akční mapy pro vyhledávání.

Řešení – přepsání duplicity

Pokud se po přemapování detekuje duplicitní kombinace, na uživatele vyskočí modální okno, které se zeptá, jak s duplikátem dále naložit. Na výběr mu dá možnosti:

- **Přepsat** – smazat kombinaci duplikátu a uchovat nově zadanou.
- **Vrátit** – smazat nově zadanou kombinaci a nijak neměnit duplikát.

Pokud uživatel zvolí možnost *Přepsat*, spustí se metoda **OverrideDuplicateBinding()**, jíž je možné vidět ve zdrojovém kódu 17.

```
1. void OverrideDuplicateBinding(InputAction duplicateAction, InputBindingCombination combo,
2.                               int duplicateIndex)
3. {
4.     ActionHistorySystem.StartNewGroup(true);
5.     InputBindingReader duplicateReader = null;
6.     foreach (InputBindingReader r in FindObjectsByType<InputBindingReader>(FindObjectsSortMode.None))
7.     {
8.         if (r.action.id != duplicateAction.id) continue;
9.         if (!r.Binding.HasSameInputs(combo)) continue;
10.        if (r.buttonIndex != duplicateIndex) continue;
11.        duplicateReader = r;
12.        break;
13.    }
14.
15.    InputBindingCombination empty = new InputBindingCombination.Builder().From(combo).AsEmpty();
16.
17.    ActionHistorySystem.AddAndExecute(new UpdateInputBindingAction(duplicateReader, empty, combo,
18.                                          c => RebindAction(duplicateAction, c, duplicateIndex-2,
19.                                                            duplicateIndex-1, duplicateIndex)));
20.    ActionHistorySystem.AddAndExecute(new UpdateInputBindingAction(this, binding, lastBinding,
21.                                          c => RebindAction(action, c, modifier1Index,
22.                                                            modifier2Index, buttonIndex)));
23.    ActionHistorySystem.EndCurrentGroup();
24.}
```

Zdrojový kód 17 - Metoda OverrideDuplicateBinding

Jejím cílem je nejen přepsat nalezený duplikát na datové úrovni, ale také zajistit, že jeho InputBindingReader (pokud existuje) se správně aktualizuje. Aby mohla metoda fungovat, přijímá jako vstupy input akci, u které byl nalezen duplikát, index duplicitní vazby a také uživatelem nově zadanou kombinaci (ř. 1 a 2).

Na ř. 4 a 23 se zajišťuje podpora systémem Undo/Redo. Jelikož se v jednom momentě počítá se změnou jednoho až dvou existujících InputBindingReaderů, tak se manuálně vytváří nový shluk akcí, umožňující přijmout více konstruktů (viz kapitola 4.2.2, str 43).

Na ř. 5-13 se vyhledá InputBindingReader, kterému je duplicitní kombinace přiřazena. Kdyby se nenašel, změna se aktualizuje jenom na datové vrstvě. Potom se už jen na ř. 15-22 nejdříve přepíše duplikát a nově zadaná kombinace tak, aby bylo vše zaznamenáno v AHS. *UpdateInputBindingAction* při volání své metody Execute() spouští metodu Rebind() na své přiřazené čtečce.

Kdyby uživatel v dříve zmíněném vyskakovacím okně zvolil možnost *Vrátit*, zavolala by se metoda **RevertBinding()**, která by jen pomocí metody Rebind() vrátila kombinaci aktuální čtečky na hodnotu před přemapováním.

5.3 Limitace

Tato část se zaměří na ty nejvýraznější limitace systémů *Propojení zkratek s akcemi* a *Přemapování zkratek*. Taky se pokusí navrhnout možná řešení pro jejich odstranění, či zmírnění.

5.3.1 Kombinace smí mít max 3 tlačítka

Jednotlivé kombinace mohou být složeny z maximálně tří tlačítek, tedy *modifi1 + modif2 + tlačítko*. V dnešní době pro tuto limitaci neexistuje řešení. Práce s modifikátory v Unity je zatím příliš striktní, kdy input systém obsahuje vestavěnou variantu maximálně pro tři tlačítka (TwoModifiersComposite).

Systém musí na konci přemapování převést speciální kompozity na jejich vestavěné protějšky (viz kapitola 5.2.5). Pro vyšší počet tlačítek už ale vestavěný kompozit neexistuje.

5.3.2 Max 2 vazby pro každé zařízení

Systém byl navržen tak, aby každá input akce měla pro každé podporované zařízení, dvě vazby, které lze přemapovat. Tento počet je považován za dostatečný pro většinu případů. Kdyby ovšem programátor vyžadoval přemapovatelných vazeb více, bylo by nutné systém poupravit.

V metodách *BuildInputBinding()* nebo *GetBindingByDevice()* se s druhou vazbou pracuje pomocí booleovské hodnoty (*useAlt*). Tu by bylo možné třeba nahradit celým číslem, označujícím, kolik vazeb umožní UI prvek čtečky přemapovávat.

5.3.3 Chybí podpora stejných kombinací na jedné mapě

Do systému je zakomponována detekce a nahrazení duplikátních kombinací (kapitola 5.2.6), avšak možnost zachování této nalezené duplikace spolu s nově nastavenou už systémem podporována není, i když některé další programy tuto vlastnost mají (Jetbrains IDE).

Možným způsobem, jak tuto vlastnost přidat, by bylo dodat možnost *Zachovat obojí* do modálního okna, které se objeví při nalezení duplicity. Pro tuto volbu by se potom použila prostě metoda *Rebind()*, která by přepsala starou kombinaci aktuální čtečky na novou.

Do systému detekce by potom musela být ještě zavedena schopnost detekovat všechny nalezené duplikáty a ne jenom první nalezený.

5.3.4 Nutnost inicializace čtečky přes kód

Velkou limitací systému je nutnost UI prvky čteček vstupů inicializovat přes kód. Nejlepší variantou by bylo prostě přetáhnout prefab tohoto prvku do scény a nastavit mu jednotlivé atributy skrz inspektor.

Možným řešením této limitace by bylo převést inicializaci těchto atributů, nyní prováděnou v metodě *BuildInputBinding()* ve třídě UIPropertyBuilder, do metody Awake() nějakého komponentu na prefabu čtečky.

5.3.5 Potřeba manuálního přepínání akčních map

Při nastavování zkratek v komponentech *ShortcutProfile* je osvědčeným postupem k nim vždy připojit *ShortcutMapActivator*. Ten potom nastavit tak, aby aktivoval akční mapy, týkající se vybraných zkratek. Jedná se ovšem o celkem zbytečný krok, který designéra pouze zdržuje. Nejlepší by bylo tuto funkcionalitu automatizovat, což už se ovšem nestihlo.

Stačilo by vzít funkcionalitu ShortcutMapActivator a přesunout ji do ShortcutProfile, kdy všechny požadované akční mapy by se vysbíraly v metodě Awake() z právě nastavených zkratek.

6 SHRNUTÍ

V této poslední kapitole se už jenom shrnou dosažené výsledky. Provede se přehled toho, co bylo naimplementováno a čeho jsou ve výsledku vytvořené funkce schopné.

Pro práci byly naimplementovány celkem dvě funkce: Undo/Redo a klávesové zkratky + jejich přemapování. U obou bylo dosaženo jejich základní funkcionality. Detailněji však funkce **Undo/Redo** je schopna:

- Vracet provedené akce (undo) a znovu provádět vrácené akce (redo).
- Shlukovat akce nad stejným konstruktem.
- Shlukovat akce nad různými konstrukty (po manuálním zažádání).

Co se **klávesových zkratek** a jejich přemapování týče, tyto funkce umožňují:

- Propojovat input akce s generickými akcemi v projektu.
- Tato propojení držet v profilech aktivujících se dle potřeby.
- Přemapovat prostou vazbu (1 tlačítko) na jinou, stejné délky.
- Přemapovat kombinaci, složenou z až tří tlačítek na jinou, stejné nebo kratší délky.
- Přemapovávat pro klávesnici + myš a herní ovladač, a to až dvě možné vazby.
- Detekovat duplicitní kombinaci ve zvoleném shluku akčních map.
- Ukládat aktuální stav přemapování do JSON souboru.

Každý systém vybrané funkce je složený z několika tříd (viz obrázky 18, 21, 24), kdy každá zastává určitou a důležitou funkcionalitu. Následující tabulky tyto třídy a jejich funkcionality popisují se záměrem udržení této informace na jednom místě.

Klíčové třídy, implementovány pro systém Undo/Redo, jsou popsány v tabulce 2.

Tabulka 3 - Funkcionality tříd Undo/Redo systému

ActionHistorySystem	Hlavní bod interakce mezi Undo/Redo a ostatními systémy. Spravuje zásobníky historií a zajišťuje shlukování akcí.
IAction	Nové akce by měly spíš implementovat ActionBase. Toto rozhraní reprezentuje jakýkoli druh akce, který může být zpracován AHS a je určen spíš speciální druhy.

ActionBase<T>	Reprezentuje klasické akce pro Undo/Redo systém. Na rozdíl od IAction přidává možnost měnit datovou vrstvu, pokud konstrukt přiřazený k akci nebyl nalezen.
GroupActionBase	Drží základní funkcionalitu, potřebnou všemi akcemi, které jsou schopny do sebe shlukovat jiné akce.
GroupAction	Představuje akci, schopnou shlukovat další akce, které ale musí ovlivňovat stejný konstrukt.
MixedGroupAction	Představuje akci schopnou shlukovat další akce, které mohou ovlivňovat rozdílné konstrukty.

Co se týče klávesových zkratek, tabulka 3 se zaměřuje na hlavní komponenty systému, propojující zkratky s generickými akcemi v projektu.

Tabulka 4 - Funkcionalita tříd systému propojení zkratek a akcí

ShortcutProfile	Drží v sobě propojené páry input akce – eventy. Když se aktivuje, umožní eventy spouštět přiřazenými input akcemi. Po deaktivování toto propojení přeruší.
ShortcutData	Představuje jeden pár input akce – eventy. Ten je možný propojit a odpojit.
ShortcutMapActivator	Po svém probuzení aktivuje akční mapy, které mu designér nastavil. Zbytek deaktivuje.
ShortcutActionMapType	Enum, který představuje akční mapy se zkratkami. Je využit v ShortcutMapActivator pro nastavení, které mapy se mají aktivovat.

Tabulka 4 vypisuje vysvětlivky tříd systému přemapování klávesových zkratek.

Tabulka 5 – Funckionalita tříd systému přemapování zkratek

TwoOptionalModifiers Composite	Speciální druh kompozitu, vycházejícího z vestavěného TwoModifiersComposite. Na rozdíl od něj je aktivován, jakmile alespoň jedno jeho tlačítko bylo stisknuto. V systému funguje hlavně jako označení, že k dané vazbě se má přistupovat jako k vazbě, která může obsahovat dva volitelné modifikátory.
InputBindingReader	Reprezentuje čtečku, schopnou přemapovat jednu vazbu jedné input akce. Spravuje celý proces přemapování.
InputBindingCombination	Datová podoba tlačítkové kombinace.
IPInputBinding	Reprezentuje UI prvek čtečky (viz obrázek 25). Řeší tvorbu čteček, název a status aktivity.
UIPropertyBuilder	Továrna na UI prvky. Specificky pro prvek čtečky se ve vytvářející metodě BuildInputBinding() rozhoduje, kolik čteček se vytvoří na základě typu kompozitu vazby.
InputDeviceType	Enum, obsahující systémem podporovaná zařízení.
InputSystem	Hlavní správce všech vstupů v projektu. Dává přístup k odkazům všech existujících input akcí, jejich vazbám a akčním mapám. K tomu umožňuje v nich najít duplikáty.
InputSystemUtils	Obsahuje v systému často užívané pomocné metody jako detekce TwoOptionalModifiersComposite, získání vazby dle zařízení nebo upravování dat pro uložení do JSON.
LinkedActionMapsAsset	ScriptableObject, držící shluky akčních map, které mají být prohledávány dohromady při hledání duplikátů.
BindingReplacer	Nahrazuje TwoOptionalModifiersComposite vestavěnými variantami.
ShortcutBindingsAsset	Datový asset, který drží všechny input akce označeny jako zkratky ve formě textových řetězců.
ShortcutBindingData	Drží textové řetězce, představující kombinace jednotlivých zkratek za účelem uložení na externí úložiště.

ZÁVĚR

Cílem diplomové práce bylo zdokumentování a implementace několika vybraných funkcí, které měly za úkol zlepšovat použitelnost softwaru. Tyto funkce měly být vyvinuty pro projekty vytvořené v herním enginu Unity a následně měly být se svou dokumentací volně zveřejněny na internetu za účelem usnadnění práce dalším programátorům.

V první řadě se vybraly řešené funkce. Těmi se staly *Undo/Redo* a *klávesové zkratky + jejich přemapování*. Následně byl proveden průzkum již existujících dokumentací, návodů a akademických prací, věnujících se způsobu implementace těchto funkcí. Zjistilo se, že existující manuály dobře pokrývají jejich způsob implementace v enginu Unity, avšak pro Undo/Redo neřeší problémy nastávající při zavedení podpory tohoto systému na často se vyskytované UI prvky posuvníků a kreslicí mřížky. Ohledně zkratek – pro ně nebyl nalezený žádný způsob, jak zajistit možnost přemapování s variabilními kombinacemi. Co se akademických prací týkalo, ty tuto oblast téměř vůbec neřešily a pokud ano, tak jen velmi povrchně.

V další části byly detailněji přiblíženy vybrané funkce. Došlo k popisu jejich základních principů, následovalo představení těch nejvážnějších problémů, které bylo potřeba vyřešit. Potom se ještě představil příkladový projekt, který sloužil jako prostředí pro vysvětlení implementace funkcí.

V implementační části se nejdříve popsal způsob fungování vytvořené funkce Undo/Redo. Popsala se základní kostra systému, jež byla silně založena na návrhovém vzoru Command. Následoval popis řešení dvou hlavních problémů, na které se při implementaci narazilo: *problém chybějícího konstruktu*, tedy když bylo zavoláno undo a UI prvek zrovna chyběl a *problém tahání myší*, tedy chování systému při operaci s UI prvky, které šlo ovládat taháním myši. Po Undo/Redo následoval popis systému napojení klávesových zkratek na generické akce v projektu. U něj se vysvětlilo rozdělení do profilů a jejich systém zjednodušené prioritizace. Pokračovalo se s vysvětlením těch nejdůležitějších aspektů systému přemapování zkratek. Popsal se základ systému, jakým způsobem bylo nutné připravit input akce a vytvářet čtečky, ale hlavně se vysvětlil způsob řešení *problému variabilních kombinací*, tedy jakým způsobem se povedl odstranit stávající limitaci enginu Unity. Potom už se jenom dovysvětlil způsob, jakým systém řeší detekci duplicitní kombinace. Pro obě funkce se také vytyčily jejich dosavadní limitace.

Výsledné implementace funkcí se ukázaly jako cíli převážně dostačující. U obou se povedlo navrhnout funkcionalitu, která nejen plní základní potřeby, ale také řeší všechny problémy, vytýčené v návrhové i implementační části. Obě funkce byly také napsány převážně programátorsky přístupným kódem, který umožňoval snadnou operaci se systémy. Slabším místem se ovšem ukázal způsob ukládání přemapovaných zkratek do JSON souboru, jenž pro správné převedení vyžadoval úpravy na několika místech v systému.

Každá funkce byla volně zveřejněna na samostatném githubu. Ve výsledku se tedy bavíme o dvou knihovnách pro engine Unity, ke kterým je přiřazen i tento dokument jako pomůcka pro lepší pochopení principu fungování funkcí.

Do budoucna by určitě bylo nejlepší, kdyby se podařilo odstranit limitace systémů, vytyčené v kapitolách 4.3.3 a 5.3. Také je možné, že společnost Unity konečně zavede oficiální podporu přemapování variabilních kombinací, jak zmiňují jejich TODO komentáře v knihovně Input System. Pokud by se tak stalo, bylo nutné systém přemapování celý přepracovat.

RESUMÉ

Tato diplomová práce řeší zdokumentování a implementace několika vybraných funkcí, které mají za úkol zlepšovat použitelnost softwaru. Funkce byly vyvinuty pro projekty vytvořené v herním enginu Unity a následně byly se svou dokumentací volně zveřejněny na internetu za účelem usnadnění práce dalším programátorům.

Nejdříve byly vybrány dvě funkce: *Undo/Redo* a *klávesové zkratky + jejich přemapování*. Následně se práce zabývá analýzou současného stavu, kde je prozkoumána již existující dokumentace, návody a akademické práce, věnující se způsobu implementace vybraných funkcí. Zjistilo se, že existující manuály dobře pokrývají jejich způsob implementace funkcí v enginu Unity, avšak pro Undo/Redo neřeší problémy nastávající při zavedení podpory tohoto systému na často se vyskytované UI prvky posuvníků a kreslicí mřížky. Ohledně zkratek, pro ně se analýzou nenašel způsob, jak zajistit možnost přemapování s variabilními kombinacemi. Současně byla analyzována oblast akademických prací, pro kterou se zjistilo, že se této oblasti téměř vůbec nevěnují a pokud ano, tak jen velmi povrchně.

V další části diplomové práce se detailně přibližují vybrané funkce. Popisují se jejich základní principy a představují se ty nejvážnější problémy. Pak se ještě představuje příkladový projekt, sloužící jako prostředí pro vysvětlení implementace funkcí.

V implementační části práce se nejdříve popisuje způsob fungování vytvořené funkce Undo/Redo. Vysvětluje se zde základní kostra systému, postavena na návrhovém vzoru Command. Po ní následuje popis řešení dvou hlavních problémů, na které se při implementaci narazilo: *problém chybějícího konstruktu* a *problém tahání myší*. Po Undo/Redo následuje popis systému napojení klávesových zkratek na generické akce v projektu a samotný systém jejich přemapování. Kromě základu systému se zde popisuje způsob připravení input akce a vytvoření čtečky. Podstatná část je věnována vysvětlení způsobu řešení *problému variabilních kombinací*, jehož řešení pomohlo překonat limitaci enginu Unity. Nakonec práce vysvětluje způsob, jakým systém řeší detekci duplicitní kombinace. Pro obě funkce práce zmiňuje i jejich dosavadní limitace.

Výsledkem práce jsou dvě volně dostupné knihovny, které programátoři mohou využít pro zavedení funkcí Undo/Redo a klávesových zkratek do svých projektů. K oběma funkcím je i dostupná dokumentace, která pomáhá detailněji pochopit jejich fungování.

SUMMARY

This diploma thesis deals with the documentation and implementation of several selected features which are intended to improve the usability of software. These functions were developed for projects created in the Unity Game Engine and then were freely published together with their documentation, which helps in easing the work of other programmers.

The two features selected were: Undo/Redo and keyboard shortcuts + their remapping. Then the thesis deals with the analysis of the current state, which is focused on existing documentation, manuals and academic works dedicated to the implementation of these selected features. It was discovered that the existing documentation covers the method of implementing these features in the Unity engine well, but with several caveats. For Undo/Redo they don't really solve the problems that arise when adding support for frequently occurring UI elements like sliders and drawing grids. For shortcuts, the current docs don't really handle any kind of possibility of remapping using variable combinations. When it comes to academic works, it was discovered that this area barely addresses this topic and if so, then only very superficially.

The next part of the thesis describes these selected features in detail. It goes through their basic principles and presents the most critical of problems. It then introduces the example project, which serves as an environment for explaining the actual implementation.

In the implementation chapter, first the Undo/Redo feature is described. The system's basic skeleton, which is built using the Command design pattern, is explained here. It is then followed by the solution to the 2 main problems encountered during implementation: the missing construct problem and the dragging mouse problem. After Undo/Redo, the system of connecting keyboard shortcuts to generic actions is explained, followed by the remapping system itself. Part of that explanation is the basis of the system, the method of preparing input actions and creation of the Input Binding reader. The focus is devoted to explaining the method of solving the problem of remapping with variable combinations. The solution to this problem allowed overcoming a limitation of the Unity Game Engine. In the end the thesis also mentions some limitations of the selected features.

The results of this thesis are two freely available libraries that programmers can use to implement the Undo/Redo and keyboard shortcut systems into their projects. These libraries also come with documentation, which helps understanding them in more detail.

SEZNAM POUŽITÉ LITERATURY

[1] KUNETKA, Jan. *Desktopová hra v Unity s možností sdílení uživatelského obsahu*. Online, Bakalářská práce, vedoucí Petr Raunigr. Ostrava: Ostravská univerzita, 2022. Dostupné z: https://portal.osu.cz/StagPortletsJSR168/CleanUrl?urlid=prohlizeni-prace-detail&praceIdno=59399. [cit. 2024-10-31].

[2] NIELSEN, Jakob. *Usability 101: Introduction to Usability*. Online. Nielsen Norman Group. C1998-2024. Dostupné z: https://www.nngroup.com/articles/usability-101-introduction-to-usability/. [cit. 2024-12-02].

[3] KOMNINOS, Amneas. *An Introduction to Usability*. Online. Interaction Design Foundation - IxDF. 2012. Dostupné z: https://www.interaction-design.org/literature/article/an-introduction-to-usability. [cit. 2024-12-02].

[4] INTERACTION DESIGN FOUNDATION - IXDF. *What is Usability?* Online. Interaction Design Foundation - IxDF. 2012. Dostupné z: https://www.interaction-design.org/literature/topics/usability. [cit. 2024-12-02].

[5] VELA, Maricarmen Terán. *Five usability factors that make products usable*. Online. LINKEDIN CORPORATION. LinkedIn. C2024. Dostupné z: https://www.linkedin.com/pulse/five-usability-factors-make-products-usable-maricarmen-ter%C3%A1n-vela/. [cit. 2024-12-03].

[6] *Unity use cases*. Online. Unity. C2024. Dostupné z: https://unity.com/use-cases. [cit. 2025-02-15].

[7] *Unity UI (uGUI)*. Online. UNITY TECHNOLOGIES. Unity UI (uGUI). C2025. Dostupné z: https://docs.unity3d.com/Packages/com.unity.ugui@2.0/manual/index.html. [cit. 2025-02-16].

[8] *Comparison of UI systems in Unity*. Online. UNITY TECHNOLOGIES. Unity Manual. C2005-2025. Dostupné z: https://docs.unity3d.com/Manual/UI-system-compare.html. [cit. 2025-02-16].

[9] UNITY TECHNOLOGIES. *Input Manager*. Online. UNITY TECHNOLOGIES. Unity Manual. C2005-2025. Dostupné z: https://docs.unity3d.com/Manual/class-InputManager.html. [cit. 2025-02-16].

[10] *Basic Concepts*. Online. UNITY TECHNOLOGIES. Input System. C2025. Dostupné z: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.13/manual/Concepts.html. [cit. 2025-02-16].

[11] *What's a design pattern?* Online. REFACTORING.GURU. Refactoring Guru. C2014-2025. Dostupné z: https://refactoring.guru/design-patterns/what-is-pattern. [cit. 2025-02-17].

[12] *Software Design Patterns Tutorial*. Online. GEEKSFORGEEKS. GeeksforGeeks. [2009]. Dostupné z: https://www.geeksforgeeks.org/software-design-patterns/. [cit. 2025-02-17].

[13] *Classification of patterns*. Online. REFACTORING.GURU. Refactoring Guru. C2014-2025. Dostupné z: https://refactoring.guru/design-patterns/classification. [cit. 2025-02-18].

[14] Lazy<T> Class. Online. MICROSOFT. Microsoft Learn. C2025. Dostupné z: https://learn.microsoft.com/en-us/dotnet/api/system.lazy-1?view=net-8.0. [cit. 2025-02-20].

[15] *MonoBehaviour*. Online. UNITY TECHNOLOGIES. Unity Manual. C2005-2025. Dostupné z: https://docs.unity3d.com/6000.0/Documentation/Manual/class-MonoBehaviour.html. [cit. 2025-02-19].

[16] *Observer*. Online. REFACTORING.GURU. Refactoring Guru. C2014-2025. Dostupné z: https://refactoring.guru/design-patterns/observer. [cit. 2025-02-20].

[17] FLOGARD, Bendik Lund. *Usability in video game editors*. Online, Diplomová práce. Trondheim: Norwegian University of Science and Technology, Department of Design, 2017. Dostupné z: http://hdl.handle.net/11250/2448935. [cit. 2024-10-02].

[18] STEHLÍKOVÁ, Petra. *Metody a metriky pro měření použitelnosti software*. Online, Diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta informačních technologií, 2016. Dostupné z: https://theses.cz/id/yliizb/. [cit. 2024-10-02].

[19] RÁDL, Jan. *Editor pro stavebnici Hubelino*. Online, Bakalářská práce. Olomouc: Univerzita Palackého v Olomouci, Přírodovědecká fakulta, 2024. Dostupné z: https://theses.cz/id/xhufox/kidiplom.pdf. [cit. 2024-10-09].

[20] KHOL, Martin. *Využití návrhových vzoru ve vývoji herních aplikací*. Online, Bakalářská práce. Praha: Česká zemědělská univerzita v Praze, Provozně ekonomická fakulta, 2022. Dostupné z: https://theses.cz/id/jx4ys5/zaverecna_prace.pdf_. [cit. 2024-10-09].

[21] *Level up your code with design patterns and SOLID*. Online. 2. doplněné vydání. Unity, c2024. Dostupné z: https://unity.com/resources/design-patterns-solid-ebook. [cit. 2025-05-07].

[22] *Input Bindings*. Online. UNITY TECHNOLOGIES. Input System. C2025. Dostupné z: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.14/manual/ActionBindings.html. [cit. 2025-05-07].

[23] *Klávesové zkratky ve Windows*. Online. MICROSOFT. Microsoft Support. C2025. Dostupné z: https://support.microsoft.com/cs-cz/windows/kl%C3%A1vesov%C3%A9-zkratky-ve-windows-dcc61a57-8ff0-cffe-9796-cb9706c75eec. [cit. 2025-03-10].

[24] *Class InputActionReference*. Online. UNITY TECHNOLOGIES. Unity Manual. C2005-2025. Dostupné z: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.14/api/UnityEngine.InputSystem.InputActionReference.html. [cit. 2025-04-22].

[25] *System.FlagsAttribute class*. Online. MICROSOFT. Microsoft Learn. C2025. Dostupné z: https://learn.microsoft.com/en-us/dotnet/fundamentals/runtime-libraries/system-flagsattribute. [cit. 2025-04-22].

SEZNAM POUŽITÝCH SYMBOLŮ

Unity	Program, sloužící pro vývoj her a aplikací.
C#	Programovací jazyk, používán v Unity.
Software	Obecné označení počítačového programu.
IDE	Vývojové prostředí softwaru.
Vestavěný editor	Umožňuje upravovat obsah aplikace přímo v ní.
UI	Uživatelské rozhraní.
Shader	Program, určující, jakým způsobem se něco vykreslí.
Package Manager	Okno v Unity, přes které je možné do editoru přidávat/odebírat balíčky.
Asset	Soubor, obsahující kontextově spojená data.
AHS	Zkráceně, třída ActionHistorySystem.
LMB	Levé tlačítko myši (LeftMouseButton).
Konstrukt	Jakýkoli element programu, který může být ovlivněn Undo/Redo systémem (posuvník, kreslicí mřížka, roztahovací seznam, atd).
Refaktorování	Měnění kódu, aniž by docházelo ke změně jeho chování.
Modální okno	UI prvek, někdy označovaný jako vyskakovací okno. Většinou se objeví uprostřed obrazovky a vyžaduje po uživateli potvrzení akce.
Rebinding	Změna tlačítek, která aktivují klávesovou zkratku.
Čtečka	UI prvek, umožňující přemapovávat kombinace vstupů. V systému reprezentována třídou InputBindingReader.
Modifikátor	Všechny tlačítka, kromě posledního, v klávesové kombinaci. Například **CTRL**+**ALT**+U.

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM ZDROJOVÝCH KÓDŮ

SEZNAM PŘÍLOH

- Dokumentace diplomové práce (PDF)
- Knihovna systému Undo/Redo
- Knihovna systému klávesových zkratek