Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Editor pro stavebnici Hubelino

2024

Jan Rádl

Vedoucí práce:
doc. RNDr. Jan Konečný, Ph.D.

Studijní program: Informatika,
Specializace: Programování a vývoj software

Bibliografické údaje

Autor:	Jan Rádl
Název práce:	Editor pro stavebnici Hubelino
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2024
Studijní program:	Informatika, Specializace: Programování a vývoj software
Vedoucí práce:	doc. RNDr. Jan Konečný, Ph.D.
Počet stran:	33
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Jan Rádl
Title:	Editor for a Hubelino building kit
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2024
Study program:	Computer Science, Specialization: Programming and Software Development
Supervisor:	doc. RNDr. Jan Konečný, Ph.D.
Page count:	33
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

Práce se zabývá vývojem editoru kuličkových drah stavebnice Hubelino v herním enginu Unity. Je podrobně popsán postup tvorby klíčových mechanismů editoru, včetně uživatelského rozhraní. Dále je popsán způsob generování instrukcí ke stavbám vytvořených v tomto editoru. V závěru je poskytnuta uživatelská příručka.

Synopsis

Thesis deals with the development of an editor for Hubelino marble run building kits. This thesis describes in detail the process of creating the key mechanisms of the editor, including user interface. It also describes the way of generating instruction for buildings created in this editor. A user manual is provided at the end.

Klíčová slova: Editor; Unity; Uživatelské prostředí; Exportování instrukcí

Keywords: Editor; Unity; User Interface; Exporting instructions

Děkuji doc. RNDr. Janovi Konečnému, Ph.D. za vedení této bakalářské práce a občasnou výpomoc, Jendovi Konečnému za testování aplikace a své rodině za podporu.

Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

Seznam obrázků

1 Úvod

V této práci se budeme zabývat návrhem a implementací editoru, který umožňuje uživateli intuitivně vytvářet a upravovat virtuální kuličkové dráhy pomocí kostek ze stavebnice Hubelino [1]. Hubelino je stavebnice pro děti, umožňující kreativní konstrukci zmíněných kuličkových drah. Je kompatibilní se stavebnicemi LEGO DUPLO, což umožňuje kombinovat obě sady stavebnic.

Jelikož je stavebnice primárně určena pro děti, je potřeba dbát na vytvoření přívětivého uživatelského prostředí, které usnadní a zpříjemní práci s editorem. Vytváření staveb v editoru tedy musí odpovídat skutečnému sestavování stavebnice, což umožní stavby snadno reprodukovat s reálnými dílky. Editor je proto schopen generovat instrukce k vytvořené dráze, které detailně popisují postup sestavení dráhy krok po kroku.

Editor je vhodný nejen pro děti, ale pro všechny příznivce kuličkových drah a stavebnic stylu LEGO. Poskytuje možnost i těm, kteří si nemohou tyto stavebnice dovolit. To umožňuje širšímu spektru uživatelů experimentovat s tvorbou kuličkových drah nezávisle na věku, finančních možnostech nebo nedostupnosti stavebnic.

Existuje pár nástrojů, které se zabývají podobnou problematikou, což stojí za zmínění.

- **Lego Digital Designer** [2]. Šlo o aplikaci, která byla vyvinuta společností LEGO a umožňovala uživatelům navrhnout a vytvářet vlastní nebo již existující digitální verze klasických LEGO stavebnic. Lego Digital Designer od roku 2023 již není podporován a byl nahrazen nástrojem BrickLink Studio.

- **BrickLink Studio** [3]. Byl vyvinut firmou BrickLink, která navíc provozuje platformu pro prodej LEGO součástek a stavebnic. Je navržen pro stejné účely jako Lego Digital Designer.

Editor sice není tak komplexní a rozsáhlý, ale zmíněná řešení nám umožní pochopit základní funkcionalitu, která je pro uživatele žádoucí.

V rámci této práce jsem se rozhodl pracovat s částmi stavebnice jako s 3D objekty. Proto jsem si pro vývoj zvolil herní engine Unity [4], který poskytuje mnoho nástrojů, umožňující vytvářet atraktivní uživatelské rozhraní a prostředí. Kromě toho Unity poskytuje robustní nástroje pro import a manipulaci s 3D modely. Dále umožňuje implementovat vlastní nástroje a specifické funkce, které umožňují splnění požadavků editoru.

2 Vývoj software

V této kapitole si popíšeme, jak byl editor naprogramován. Předtím si ale vysvětlíme základní pojmy z Unity, které budou v průběhu práce využívány.

2.1 Prostředí Unity

Po vytvoření projektu se zobrazí hlavní okno Unity editoru, ve kterém se nachází mnoho důležitých prvků. Jedním z nich je scéna, což je prostředí, ve kterém můžeme manipulovat s objekty. Mezi objekty patří například modely bloků, kamery nebo světla. Projekt může obsahovat více scén, mezi kterými lze přepínat, ale my si vystačíme s jednou scénou. Objektům můžeme přiřadit funkcionalitu pomocí komponent. Komponenty obsahují různé vlastnosti, které můžeme upravit, a tím změnit nebo rozšířit jejich chování. Mezi základní komponenty patří následující.

- **Transformace**. Je to nejzákladnější komponenta, kterou každý objekt obsahuje. Určuje pozici v rámci scény.
- **Materiál**. Určuje vzhled objektů ve scéně. Například jeho barvu, texturu, lesk nebo průhlednost.
- **Collider**. Definuje oblast objektu určenou pro fyzikální systém Unity. Příkladem využití je detekce kolizí mezi objekty.
- **Script**. Jde o kód napsaný v objektově orientovaném jazyce C#. Díky němu můžeme objektům přidat vlastní funkcionalitu. Můžeme například zařídit reagování na uživatelské vstupy, reakci na různé události, nebo jiné řízení chování objektu.

Zmíněné skripty dědí ze třídy `MonoBehaviour`. To umožňuje používat některé základní metody. Mezi hlavní patří následující.

- **Start.** Tato metoda je volána při inicializaci objektu. Nastavují se zde proměnné, potřebné pro funkčnost objektů. Zde můžeme i zařídit provázání s ostatními skripty.
- **Update.** Tato metoda se zavolá při každém snímku programu. Zde se odchytávají vstupy od uživatele a provádí se zde potřebné operace.

Při tvorbě softwaru v Unity je vhodné rozdělit funkcionalitu do samostatných skriptů s jasně definovanou odpovědností. Můžeme tedy mít samostatný skript pro manipulaci s bloky, ovládání kamery nebo skript pro uživatelské rozhraní. Tohle oddělení umožní snadnější údržbu a rozšiřitelnost kódu. Někdy je potřeba, aby skripty mohly mezi sebou komunikovat. Proto si můžeme v jednotlivých skriptech udržovat odkaz na jiné objekty, což nám umožní přímé volání metod jiných skriptů.

Všechny modely kostek Hubelino a desku jsem převzal ze stránky Thingverse [5], platformy pro sdílení 3D modelů určených pro tisk. Tyto modely jsou pod licencí Creative Commons - Attribution [6], která umožňuje jejich volné použití a modifikaci s uvedením zdroje.

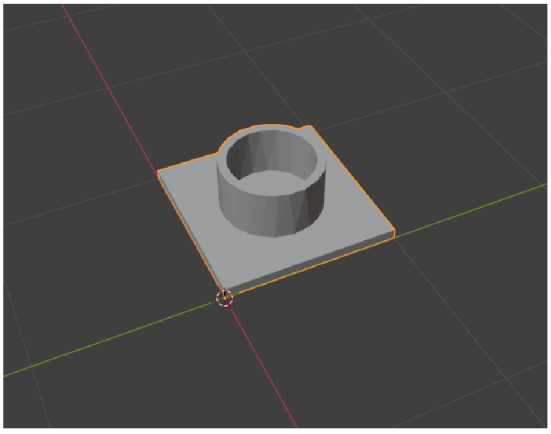
Před importem do Unity jsem modely upravil a dále konvertoval do formátu podporovaného Unity pomocí 3D editoru Blender. Ten slouží pro práci s 3D modely, včetně animace a renderování. Pro usnadnění práce s modely jsem upravil rozměry bloků a upravil jejich velikost tak, aby se zjednodušili některé operace, které budeme s bloky provádět. Následně byly modely exportovány ve formátu OBJ, který je kompatibilní s Blenderem i Unity.

2.2 Tvorba stavební desky

Nejdůležitější funkcí editoru je, aby uživatel mohl pokládat dílky stavebnice, a tím tvořit různé stavby. Aby tvorba odpovídala tomu, jak se stavby vytváří v realitě, musí se dílky připojit na desku.

Bylo by možné vložit do scény model desky a zařídit zarovnávání dílků na konkrétní místa. Jenže stavebnice Hubelino poskytují různé velikosti stavebních desek. Jelikož se chceme přiblížit těmto reálným stavebnicím, je vhodné poskytnout uživateli možnost výběru velikosti desky. Proto při spuštění programu bude muset uživatel nejprve založit nový projekt. Při zakládání projektu zadá délku a šířku desky a z těchto dvou parametrů následně program vytvoří samotnou desku. K tomu jsem vytvořil nový skript `ProjectManager`, ve kterém bude tato funkcionalita implementována.

K tvorbě desky je potřeba část modelu desky. Jde o konkrétní část, která slouží k propojení s dílky stavebnice. Tento model jsem vytvořil z již existujícího modelu desky ze stránky Thingverse. V programu Blender jsem oříznul tento model tak, abych získal pouze požadovanou část.



Obrázek 1: Propojovací část modelu stavební desky.

Nyní lze implementovat tvorbu libovolné desky. Pomocí jednoduchého cyklu se opakovaně umisťují části desky do scény tak, aby ležely vedle sebe. Tím se vytvoří požadovaná deska. Problém je, že desku tvoří řada objektů. V případě větší desky může jít o stovky objektů, což vyžaduje větší množství systémových prostředků, jako je paměť nebo výpočetní výkon. Tím může dojít ke snížení výkonu aplikace, a proto je potřeba jednotlivé části desky spojit do jednoho samostatného objektu.

V Unity můžeme spojit objekty pomocí komponenty `MeshFilter` a metody `CombineMeshes`.

- **MeshFilter** je komponenta, která udržuje odkaz na mesh objektu, definující jeho geometrii. Pracuje s pomocí `MeshRenderer`, který objekt vykresluje.
- **CombineMeshes** je metoda, která slouží k spojení více meshů do jednoho, což je vhodné pro optimalizace výkonu.

Pro každou část desky se uloží mesh a jejich sloučením je následně vytvořen samostatný mesh. Tím získáme jediný objekt reprezentující desku a využité modely části desky mohou být smazány.

2.3 Pokládání bloků

Dále je potřeba implementovat způsob, jakým se budou dílky stavebnice pokládat. Nejdříve je potřeba vyřešit, odkud budeme získávat modely bloků. Proto byla vytvořena třída, obsahující všechny informace o blocích. Jde o třídu `BlockDatabase`, určenou pro správu dat o blocích. Při inicializaci této třídy se do ní nahrají všechny modely bloků, jejich jméno a velikost. Velikost se nám bude hodit při mnoha operacích včetně pokládání. Ostatní skripty mohou komunikovat s touto třídou pro získání informací o blocích.

Se třídou `BlockDatabase` komunikuje skript `BlockPlacer`. V tomto skriptu se provádí veškerá funkcionalita spojená s pokládáním bloků. Blok vybírá podle aktuálně nastaveného identifikátoru `blockIndex`. Samotný proces pokládání bloku probíhá následovně.

1. Zjistí se pozice, kam uživatel umístil kurzor myši na stavební desce.
2. Pozice se následně zaokrouhlí na nejbližší možné místo na desce, kam se může dílek připojit.
3. Na tomto místě se vytvoří náhled bloku, který ukáže uživateli, jak bude blok vypadat, než se ho rozhodne skutečně položit.
4. Pokud se bude náhled bloku prolínat s jiným blokem, náhled bude mít červenou barvu a blok nepůjde položit.
5. Pokud se blok nebude prolínat, uživatel může kliknout myší a blok se skutečně položí.

Pro zjištění místa, kam dílek položit, je potřeba převést pozici kurzoru myši na pozici desky. Nejprve se zjistí pozice kurzoru myši v rámci obrazovky a následně se zavolá metoda `Camera.ScreenPointToRay`, která převádí 2D souřadnice na paprsek ve 3D prostoru. Tento paprsek vychází z pozice kamery a směřuje do scény, což umožňuje zjistit bod v prostoru, kde se paprsek protnul s libovolným colliderem. V tomto případě to bude buď deska, nebo již položený blok. Tím získáme přesné souřadnice na desce, kam blok umísit.

Každý pokládaný blok i deska musí mít vlastní collider. To zajistí, že paprsek bude zastaven daným objektem a vrátí se souřadnice místa protnutí. Bez collideru by paprsek prošel skrz blok a vrátil by místo na desce. Kdyby ani deska neměla collider, paprsek by nevrátil žádnou souřadnici.

Aby bylo možné bloky pokládat pouze na vyznačená místa na desce, je nutné implementovat jednoduchou funkci, která se o to bude starat. Pro tento účel jsem vytvořil další skript `Grid`, který se stará o umístění bloku v rámci desky a bude komunikovat s hlavním skriptem pro pokládání bloků.

Využívá se zde techniky „snapping“. Tato technika umožňuje objektům přichytit se k určitým bodům ve scéně. Dále je potřeba určit vzdálenost mezi jednotlivými body, na které se může blok zarovnat. Deska i modely bloků byly upraveny tak, aby se vzdálenost rovnala číslu jedna. Pro zarovnávání v rámci výšky je to dáno výškou bloku. Metodě se předá pozice získaná protnutím paprsku s colliderem. Tato pozice je dána jako třírozměrný vektor obsahující reálná čísla, určující souřadnice v rámci naší scény. Pak zaokrouhlíme souřadnice objektu na nejbližší násobek hodnoty dělení. Hodnota dělení je určena zmíněnou minimální vzdáleností mezi bloky.

```
1 int x = (int)Math.Round(position.x / length, 0, MidpointRounding.
    AwayFromZero);
2 int y = (int)Math.Floor(position.y / height);
3 int z = (int)Math.Round(position.z / length, 0, MidpointRounding.
    AwayFromZero);
4 Vector3 newPosition = new Vector3(x * length, y * height, z * length
    );
```

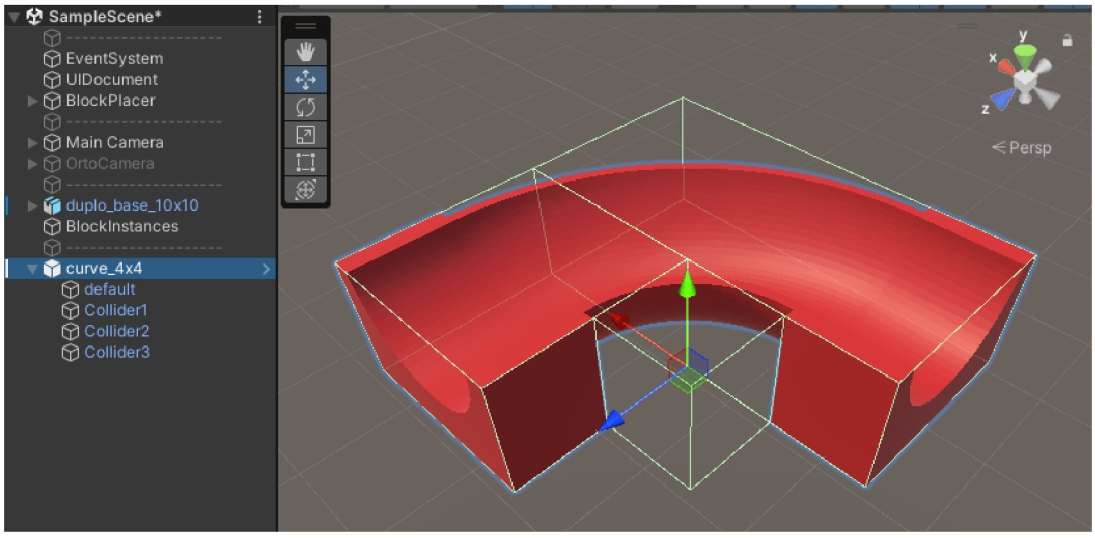
Zdrojový kód 1: Část funkce pro zarovnání polohy bloku

Dále musíme zařídit, aby se blok nenacházel mimo desku. Bude-li mimo desku, pozice se upraví tak, aby se blok nacházel na kraji desky a zároveň nebyl mimo ni. Implementujeme metodu `OutOfBounds`, která pracuje s minimálním a maximálním bodem desky a pokládaných bloků. Minimální bod objektu je jeho nejmenší souřadnice v 3D prostoru, obvykle odpovídající nejlevějšímu a nejníže umístěnému bodu objektu. Maximální bod je naopak určen nejpravějším a nejvyšším bodem objektu. Tyto body určují rozsah objektu v prostoru a díky nim můžeme určit, jestli je část bloku mimo oblast desky. Pak stačí určit vzdálenost jednoho z vyčnívajících bodů objektu od desky a posunout blok správným směrem o tuto vzdálenost.

Jelikož chceme, aby stavby vytvořené v editoru bylo možné postavit se skutečnými dílky, je potřeba zabránit položení bloku, pokud se prolíná s jiným blokem na desce. Tento problém lze řešit více způsoby. Jedním z nich je mít strukturu, ve které by se uchovávaly informace o jednotlivých bodech desky a jejich dostupnosti. Pak by se jen ověřovalo, zda je místo, kam se má blok položit, volné. Nicméně tato metoda by vyžadovala rozsáhlé množství kódu a vyžadovala by více paměti. Také by bylo potřeba u každého bloku uchovávat logiku popisující, jakou oblast na desce zabírá.

Lepší způsob je detekce kolizí pomocí metody `CheckBox`, která je součástí fyzického systému Unity. Metoda na zvolených souřadnicích vytvoří box o zvolené velikosti. V oblasti vyznačené tímto boxem se provede detekce kolizí a vrátí se informace o tom, jestli se v této oblasti nachází nějaký collider. Rozměry i umístění tohoto boxu mohou být stanoveny velikostí collideru pokládaného bloku. Tímto způsobem je možné zjistit, zda je místo, kam chceme blok položit, volné nebo zda je obsazeno jiným objektem. Pokud není volné, uživatel nebude moct daný blok položit.

Některé bloky mají složitější tvar, který nelze vymezit jednoduchým boxem. Proto je potřeba některým blokům přidělit více colliderů, které přesně vymezí požadovaný prostor, který chceme detekovat fyzickým systémem. Objekt běžně může obsahovat pouze jeden collider. Ovšem to lze vyřešit tak, že objektu přidělíme více podobjektů s vlastním colliderem, každý definující jinou oblast zabíranou tímto blokem.



Obrázek 2: Přiřazení více colliderů objektu.

Metoda `CheckBox` bude tedy procházet všechny collidery podobjektů, a pro každý z nich vytvoří box o stejné velikosti. Tím zaručíme správnou detekci kolizí i pro objekty složitějšího tvaru. Posledním krokem je položení skutečného bloku. Po kliknutí myši dojde k nahrazení průhledného bloku skutečným.

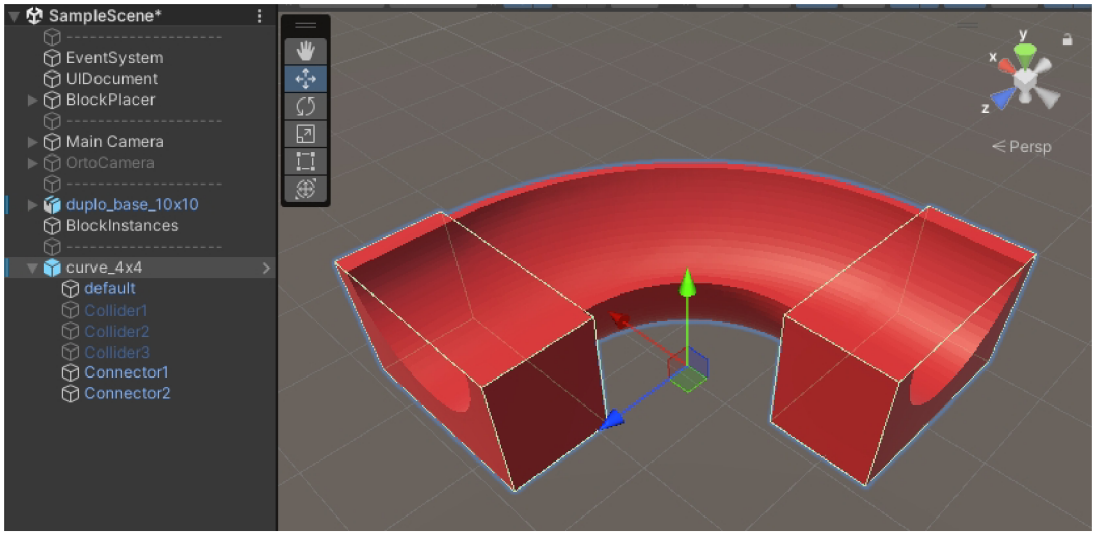
Někdy bude užitečné uchovávat různé informace a funkcionalitu uvnitř bloku. Objektu můžeme přiřadit funkcionalitu tím, že mu přiřadíme skript pomocí metody `AddComponent`. Takovým skriptem je `BlockHandler`, ve kterém budeme uchovávat například ID bloku nebo aktuální barvu bloku. Bude toho obsahovat více, ale to si podrobněji popíšeme v kapitole 2.4.

2.4 Propojování bloků

Jedním z požadavků na editor je, aby sestavené dráhy šlo sestavit pomocí skutečných dílků. Je potřeba tedy zabránit situacím, kdy stavba nebude splňovat tento požadavek. Například tehdy, kdy blok nebude řádně připojen k žádnému jinému bloku nebo desce. Proto je potřeba uživateli zakázat položení izolovaného bloku. K dosažení tohoto cíle budeme potřebovat systém pro propojování bloků.

Nabízí se využití metody `CheckBox`, kterou jsme využili pro detekci kolizí. Mohli bychom jednoduše detekovat bloky, které se nachází pod nebo nad pokládaným blokem a tyto bloky označit jako propojené. Tak to ale udělat nemůžeme, jelikož bloky mají pouze určitá místa, kde se mohou propojit. Například blok představující dlouhou zatáčku pro kuličku má dvě propojovací části, které se nachází na krajích tohoto bloku a propojení je možné jen zdola.

Tohle se může vyřešit obdobným způsobem, jako jsme to dělali při zabrání oblasti bloků pomocí colliderů. Vytvoříme collidery na místech, kde se mohou bloky propojit. Tímto způsobem budeme moci detekovat tyto oblasti pomocí příslušných metod.



Obrázek 3: Vyznačená místa, na kterých se může blok propojit.

Navíc je ale potřeba odlišit tyto oblasti od klasických colliderů. Pro identifikaci těchto oblastí použijeme štítky. Tyto štítky určují, zda se na daném místě mohou bloky propojit, a také směr ze kterého se mohou připojit. Štítek může

být libovolný řetězec, který lze nastavit v inspektoru Unity editoru. Zavedeme tedy tři štítky.

- **ConnectorTop**. Blok se může připojit pouze ze shora.
- **ConnectorBottom**. Blok se může připojit pouze ze zdola.
- **Connector**. Blok se může připojit z obou stran.

Detekování bloků, ke kterým se může pokládaný blok připojit, se bude provádět už při zobrazení náhledu bloku. Tím zamezíme položení bloku v případě, že se nebude moct k ničemu připojit. Jinak by se mohlo stát, že blok bude levitovat.

Dále je potřeba získat seznam všech propojovacích colliderů. Metoda `CheckBox` pouze zjistí, jestli se na daném místě nachází collider a vrátí o tom informaci. My ale potřebujeme získat všechny collidery, které se na tomto místě nachází, abychom mohli zjistit jejich štítek a mohli propojit více než dva bloky. Můžeme využít metody `OverlapBox`, která funguje stejně jako `CheckBox`, ale vrací seznam všech colliderů, se kterými se protnula. Detekce kolizí se bude provádět z míst označených štítky buď nad tímto místem, pod tímto místem, nebo na obou místech. Následně stačí porovnat štítky a uložíme si všechny bloky, se kterými se může blok propojit.

```
1  if (!onGround && (child.CompareTag("Connector") || child.CompareTag(
       "ConnectorBottom"))) {
2     ConnectSurroundingBlocks(bounds,0);
3  }
4  if (child.CompareTag("Connector") || child.CompareTag("ConnectorTop"
       )) {
5     ConnectSurroundingBlocks(bounds,1);
6  }
```

Zdrojový kód 2: Využití štítků k rozhodnutí odkud provést detekci kolizí.

```
1  foreach (var collision in Physics.OverlapBox(pos,ext * 0.9f)) {
2     GameObject obj = collision.gameObject;
3     if (obj.CompareTag("Untagged")) continue;
4     switch (direction) {
5        case 0 when !obj.CompareTag("ConnectorBottom"):
6        case 1 when !obj.CompareTag("ConnectorTop"):
7           connectionBuffer.Add(collision.transform.parent.gameObject)
               ;
8           break;
9     }
10 }
```

Zdrojový kód 3: Nalezení propojení mezi bloky.

Nyní lze zařídit zamezení položení bloku, pokud se nebude moct k ničemu připojit. Uděláme to podobně jako v předchozí části. Pokud metoda `OverlapBox` vydá prázdný seznam, zbarvíme náhled červeně a zamezíme uživateli položení bloku. Nicméně, kdykoliv je blok odstraněn, přeruší se propojení, a tím některé bloky mohou začít levitovat. To znamená, že nebude existovat cesta z levitujícího bloku k desce. Proto je potřeba při odstranění bloku kontrolovat, jestli jsme nepřerušili cestu k desce ze všech bloků, ke kterým byl odstraněný blok připojen.

Každý blok obsahuje skript `BlockHandler`, ve kterém nyní můžeme udržovat seznam připojených bloků. Po odstranění bloku se zavolá funkce pro odpojení odstraněného bloku od ostatních a zkontroluje se, jestli existuje cesta k desce z každého z těchto bloků. Jelikož jsou dílky stavebnice navzájem propojeny, lze si stavbu v editoru představit jako graf, kde bloky jsou uzly, a propojení mezi nimi jsou hrany. To nám umožní si uvědomit, že lze využít jednoduchého algoritmu pro hledání cesty v grafu mezi uzly. Algoritmus funguje následovně.

1. Funkce přidá aktuální blok mezi navštívené bloky.
2. Pokud je výška aktuálního bloku rovna nule, tedy že se nachází přímo na desce, vrátí true.
3. Jinak pro každý nenavštívený připojený blok opakuje postup.
4. Pokud cesta nebyla nalezena, vrátí false.

Pokud nebyla nalezena žádná cesta, označíme všechny navštívené bloky jako levitují. Stačí pouze změnit jejich materiál a barvu, aby bylo uživateli jasné, že levitují. Uživatel následně může buď levitující bloky odstranit, nebo může položit nové bloky tak, aby znovu vznikla cesta k desce. V tomto případě, když připojíme blok k levitujícímu dílku, zkontrolujeme, jestli se připojil i k nelevitujícímu. Tím máme zaručeno, že bude existovat cesta k desce skrz pokládaný blok a můžeme všechny levitující bloky vrátit do původního stavu.

2.5 Implementace Undo a Redo

Operace Undo a Redo, také známé jako operace Zpět a Znovu, jsou klíčovými funkcemi mnoha editorů. Tyto operace umožňují uživatelům vrátit se k předchozím stavům práce nebo znovu provést zrušené akce. Jsou důležité pro zlepšení uživatelského prožitku a usnadnění práce s editorem. V editoru umožňují spravovat historii provedených akcí za jeho běhu. Uživatel má tak možnost se kdykoliv vrátit do předchozího stavu práce, což zajišťuje větší kontrolu nad editačním procesem.

Před implementací této funkcionality je potřeba definovat operace, které mohou být v editoru provedeny a které by měly být reverzibilní. Konkrétně jde o operace položení, smazání a přebarvení bloku. Pokud uživatel provede jednu z těchto operací, má možnost ji navrátit zpět.

Vytvořil jsem nový skript `Reversiblity`, který uchovává informace o provedených operacích. Tyto informace se ukládají do zásobníku. Zásobník je datová

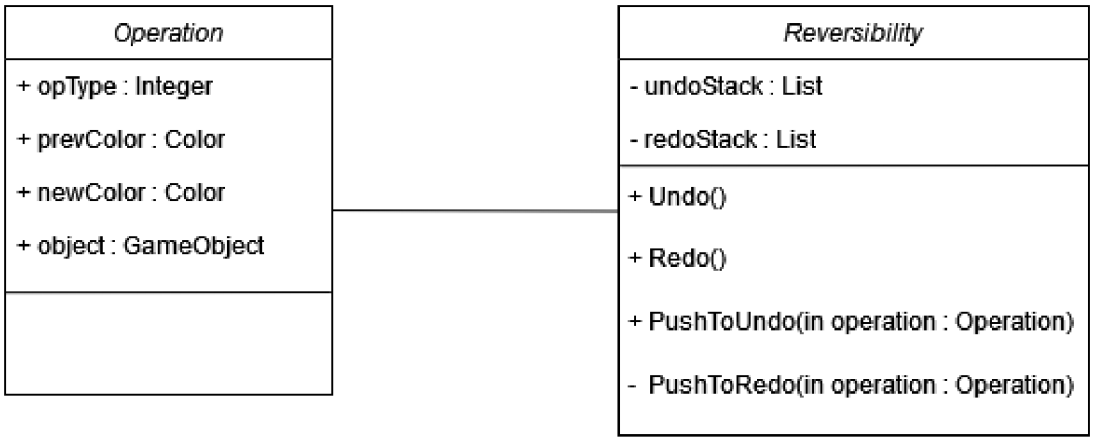
struktura typu Last-In-First-Out, což znamená, že pokud se budeme chtít vrátit o krok zpět, naposledy provedená operace bude první, která se odvolá. Pro manipulaci se zásobníkem jsou klíčové operace `Pop()` a `Push()`, které se musí implementovat s ohledem na podmínky jejich použití. Chceme například omezit počet operací, které mohou být v zásobníku uloženy, aby se zachovala jednoduchost editoru a nezatěžovala se paměť programu.

Dále je potřeba definovat, s jakými daty bude zásobník pracovat. S ohledem na operace, které chceme provádět, se stačí zaměřit pouze na informace o blocích. Konkrétně budou užitečné údaje o poloze, rotaci a barvě. Proto si můžeme uložit celý objekt bloku, ze kterého budeme informace čerpat. Jelikož ale máme možnost provádět operaci Redo, je potřeba uchovávat informaci nejen o původní barvě, ale i o nové, abychom je mohli vhodně prohazovat při provádění operací.

Pokud se uživatel rozhodne vrátit o krok zpět, zavolá se metoda `Undo()`, která vyhodnotí naposledy provedenou operaci a podle toho provede následující.

- Při položení bloku se tento blok smaže.
- Při odstranění bloku se blok znovu položí na původní místo.
- Při přebarvení se blok obarví jeho původní barvou.

Uživatel také musí mít možnost provést znovu operaci, kterou zrušil pomocí Undo. Implementujeme tedy druhý zásobník pro funkci Redo, který bude fungovat stejným způsobem, ale operace položení a odstranění bloku budou v `Redo()` prohozeny. Třída `Reversiblity` by mohla vypadat takto.



Obrázek 4: Zjednodušený diagram tříd pro skript Reversiblity

Každá provedená akce, která byla přidána do zásobníku Undo, musí vést k vyprázdnění zásobníku Redo. Kdykoliv je provedena operace kroku zpět, vracíme se zpět v historii akcí. Pokud bychom nemazali obsah zásobníku Redo, uživatel by se mohl dostat do nekonzistentního stavu a způsobit neočekávané chování programu, kdyby se rozhodnul provést operaci Redo.

Pokud uživatel odstraní blok a rozhodne se vrátit o krok zpět, je nutné blok znovu položit. Jenže znovu inicializovat blok a provádět celý proces jeho vytvoření je zbytečné a neefektivní. Je vhodné minimalizovat inicializaci bloku, proto je potřeba změnit způsob jejich odstraňování. Bloky se neodstraní okamžitě po kliknutí v režimu odstraňování, ale jsou dočasně skryty, zatímco zůstávají v paměti. Ostatní bloky se od něj dočasně odpojí, aby nedošlo k narušení systému kontroly levitujících bloků. Skrytý blok se skutečně odstraní až tehdy, když se nebude nacházet na žádném zásobníku. Při uvolnění operace ze zásobníku se vždy zkontroluje, zda žádná jiná operace na zásobnících nepracuje se stejným objektem. Pokud ne, objekt může být bezpečně odstraněn.

2.6 Uživatelské rozhraní

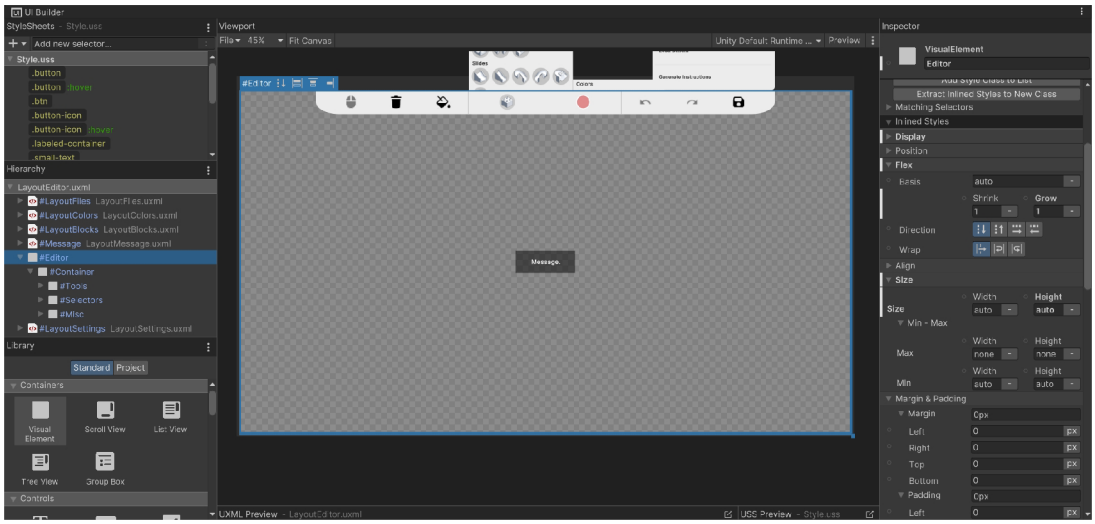
Uživatelské rozhraní je důležitým prvkem každého editoru, poskytující uživatelům nástroje pro usnadnění tvorby a úpravy práce. V editoru musí být rozhraní dobře navržené, aby urychlilo celý proces vytváření a úpravy staveb, usnadnilo orientaci a zvýšilo celkový uživatelský prožitek. V Unity existuje více metod, jak rozhraní vytvářet, přičemž každá má své výhody a účely [7]. Mezi dvě nejrozsáhlejší patří následující.

- Unity UI
- UI Toolkit

Unity UI je systém pro tvorbu uživatelského rozhraní, který je dostupný od verze 4.6. Prvky uživatelského rozhraní jsou zde reprezentovány pomocí komponent, které lze upravovat přímo v inspektoru Unity editoru. Tlačítka, textová pole nebo obrázky jsou reprezentovány jako objekty v hierarchii scény, umístěné do speciálního plátna zvaného „canvas“. To je typ objektu sloužící pro vykreslování těchto prvků v 3D prostoru, což umožňuje s nimi provádět stejné operace jako s jinými objekty.

UI Toolkit je moderní nástroj pro tvorbu uživatelského rozhraní, který byl představen v roce 2021. Nyní je doporučován k vytváření prvků UI a je inspirován standartními metodami tvorby webových stránek. Programátoři s těmito znalostmi mohou snadno využit UI Toolkit pro tvorbu rozhraní. Jelikož je to nová technologie, můžeme být omezeni chybějící funkcionalitou. Narozdíl od Unity UI, který pracuje s prvky jako s objekty v 3D prostoru, UI Toolkit vytváří samostatnou vrstvu oddělenou od 3D prostředí. S vrstvou pracuje UI Builder, který umožňuje vytvářet a editovat prvky uživatelského rozhraní. Tohle oddělení 3D prostoru a uživatelského rozhraní přispívá k efektivitě a přehlednosti práce s UI v Unity.

Rozhodnul jsem se využít UI Toolkit, kvůli zkušenostem s tvorbou webových stránek, a také protože mi přijde více intuitivní. Začal jsem tedy vytvářet layout pro uživatelské rozhraní pomocí „kontejnerů“, které určují pozici prvků a velikost v rámci obrazovky. Kontejnery jsem dále rozdělil na více částí podle potřeby, například pro uspořádání tlačítek.



Obrázek 5: Prostředí pro tvorbu layoutu za pomocí UI Builder.

Prvkům uživatelského rozhraní se nastavují různé parametry, jako jsou rozměry, odsazení, nebo chování vůči ostatním prvkům. Za normálních podmínek se pracuje s pevnými hodnotami v pixelech. Aby ale byla zajištěna responzivita vůči rozlišení obrazovky, je potřeba pracovat s rozměry určených v procentech. To umožňuje, aby se prvky přizpůsobovaly velikosti okna editoru a udržovaly konzistentní vzhled při různých rozlišeních obrazovky.

Při vytváření prvků se často opakuje nastavení parametrů. Proto lze v Unity využít styly. Styly jsou třídy, které ukládají nastavení parametrů, které lze aplikovat na jiné prvky. To usnadňuje práci a šetří kód.

Veškeré funkce editoru by měly být ovladatelné pomocí uživatelského rozhraní. Proto by rozhraní mělo obsahovat tlačítka pro volání těchto funkcí. Pro lepší pochopení funkce každého tlačítka jsem využil obrázky, které popisují, co se po kliknutí stane. Dále jsem využil přechodových animací, například když uživatel najede kurzorem na tlačítko, nebo když si zobrazí menu výběru bloků. Přechodové animace dodávají vizuální zpětnou vazbu o interakcích s prvky, což umožňuje lepší orientaci v editoru a příjemný uživatelský zážitek. Dbal jsem také na logické seřazení tlačítek, pro lepší přehled. Například jsem seskupil přepínače režimu editoru, nebo tlačítka pro výběr bloků a barev.

Po vytvoření layoutu rozhraní je potřeba dodat funkcionalitu. Pro tyto účely jsem vytvořil skript `UIEditorManager`, který se stará o přidělení funkcionality všem prvkům rozhraní. Jelikož je tento skript rozsáhlý, rozdělil jsem odpovědnost do dalších třech tříd `UIBlockMenu`, `UIColorMenu` a `UISettingsMenu`. Každá z nich se stará o samostatné menu, které si uživatel může otevřít. Metoda `OnEnable` je základní funkcí, která se spustí při aktivaci skriptu. V ní si lze odchytnout „root“ prvek, pod kterým se nachází všechny ostatní prvky rozhraní. Tyto prvky získáme pomocí dotazu „Query“ podle jména nebo třídy. Následně prvku můžeme, pomocí události `clicked`, přidělit metodu, která se zavolá po kliknutí myší. Takto jednoduše lze nastavit, co se má stát po kliknutí na tlačítko.

Většina tlačítek volá již existující metodu, jako například tlačítko pro krok zpět, které spustí `Undo()`. Někdy je potřeba, po zavolání metody, upravit chování prvků rozhraní. V editoru se nachází bloky, které ve stavebnici Hubelino mají pouze bílou barvu. Kdykoliv vybereme takový blok, je potřeba zablokovat výběr barev. Když se klikne na tlačítko pro změnu bloku, spustí se metoda, která pozmění parametry jiného prvku. V tomto případě vypne možnost kliknout na tlačítko výběru barev a změní barvu tohoto tlačítka tak, aby uživatel věděl, že není aktivní.

Komunikace mezi programem a UI může probíhat oběma směry. Například, kdykoliv se vyprázdní zásobník `undoStack`, je potřeba o tom skript informovat, aby mohl zrušit možnost kliknutí na tlačítko kroku zpět. Proto si některé skripty uchovávají odkaz na skript `UIEditorManager`, aby mohly předávat potřebné informace pro změnu stavu tlačítek.

Dále je potřeba zajistit, aby editor nebylo možné ovládat, když se aktuálně pracuje s UI prvky. K tomu jsem využil zpětná volání, která umožňují prvkům předat metodu, která se zavolá při určitých událostech během běhu programu. Každému menu jsem zaregistroval zpětné volání, které se zavolá, pokud myš najede nebo vyjede z oblasti uživatelského rozhraní.

```
1 private void RegisterMouseCallbacks() {
2     foreach (var menu in menus) {
3         menu.RegisterCallback<MouseEnterEvent>(MouseEnterCallback);
4         menu.RegisterCallback<MouseLeaveEvent>(MouseLeaveCallback);
5     }
6 }
```

Zdrojový kód 4: Přiřazení zpětných volání pro všechna menu editoru.

V metodách `MouseEnterCallback` a `MouseLeaveCallback` dáme vědět skriptu `BlockPlacer`, jestli může provádět činnost. K tomu stačí zavést proměnnou, kterou bude skript kontrolovat předtím, než začne provádět práci.

Na závěr jsem se rozhodl, že je vhodné uživateli v určitých situacích zobrazit zprávu. Například když se uživatel pokusí udělat neplatnou operaci, jako je přebarvení stavebního bloku, který musí zůstat bílý. Také to lze využít pro informování uživatele o dokončení operace, jako je uložení nebo načtení souboru. Zprávy jsem implementoval jako malá okna uprostřed obrazovky, která jsou schovaná, dokud je nějaká operace nezobrazí. Následně se nastaví zpráva, která má být zobrazena, a po chvíli se zase skryje.

Obrázek 6: Příklad zobrazení zprávy uživateli.

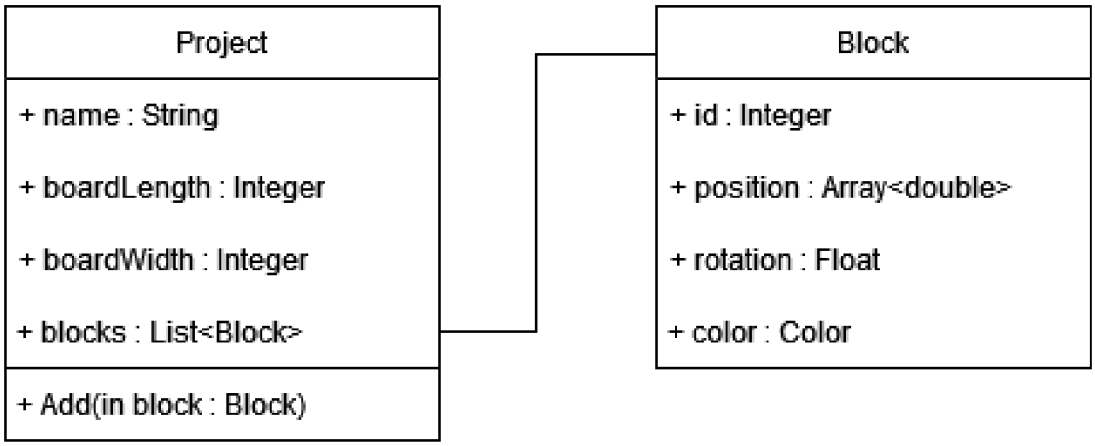
2.7 Práce se soubory

Uživatel se může rozhodnout, že chce projekt dokončit příště, nebo ho chce přenést na jiný počítač. Proto jsem implementoval systém pro ukládání a zpětné načítání staveb v editoru. K tomu jsem vytvořil další skript `Files`, který se stará o práci se soubory.

Unity má vestavenou funkci pro zobrazení prohlížeče souborů, která vrací cestu k souboru. Bohužel, tato funkce je pouze podporovaná v editoru Unity a není k dispozici ve finálním sestavení. Žádná jiná vestavená funkce pro tyto účely v Unity neexistuje. Proto jsem se rozhodl použít balíček „StandaloneFileBrowser“ [8], který umožňuje používat prohlížeč souborů i ve finálním produktu.

Nyní je potřeba vyřešit způsob ukládání souborů. Unity umožňuje serializaci dat do formátu JSON. Tento formát je snadno modifikovatelný, dobře čitelný a jednoduchý na implementaci.

Pro implementaci je potřeba vytvořit třídu, která bude uchovávat všechny informace o projektu a blocích ve stavbě. Jelikož při načtení souboru je potřeba vytvořit odpovídající desku, musí být ve třídě informace o délce a šířce této desky. Dále se musí uložit informace o blocích, jako jsou souřadnice, rotace, barva a typ, aby bylo možné sestavit původní stavbu. Všechny bloky položené v editoru se ukládají do proměnné `blockCache` pro rychlý přístup. Proto můžeme všechny potřebné informace snadno získat z této proměnné. Pokud se ve stavbě nachází levitující bloky, nelze stavbu uložit, dokud uživatel tuto situaci neopraví. Nakonec se zavolá funkce `JsonUtility.ToJson`, která serializuje informace do formátu JSON a získaný řetězec se uloží do souboru.



Obrázek 7: Diagram tříd pro informace určených pro serializaci.

Pro načtení stavby se vybere soubor pomocí prohlížeče souborů a obsah souboru se deserializuje do třídy s informacemi o blocích. List s těmito informacemi se předá třídě `ProjectManager`, která načte projekt, vytvoří desku a předá informace o blocích třídě `BlockPlacer`.

2.8 Generování instrukcí

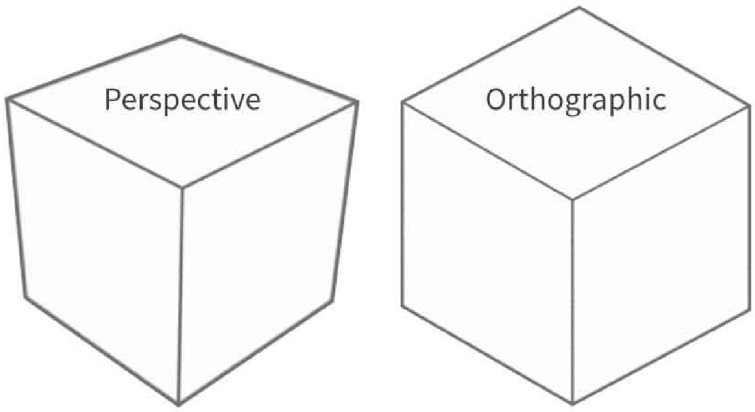
Jedním z požadavků práce je generování instrukcí pro sestavené stavby v editoru. Samotné Hubelino obsahuje instrukce ke všem prodávaným stavebnicím. Tyto instrukce je možné najít na oficiálních stránkách Hubelino [9]. Při vývoji jsem se inspiroval těmito existujícími instrukcemi a snažil jsem se napodobit jejich vzhled. Proto jsem se také rozhodl instrukce generovat do PDF souboru.

Práce s PDF soubory není jednoduchá, proto jsem využil externí knihovny „iTextSharp“ [10]. Knihovna iTextSharp je populární knihovna pro práci s PDF dokumenty v jazyce C#, která poskytuje nástroje pro vytváření a úpravu PDF souborů. Dokáže pracovat s textem, vkládat obrázky, vytvářet grafické prvky a mnoho dalšího. Knihovna je k dispozici pod licencí „open-source AGPL“, což umožňuje bezplatné používání, pokud aplikace zůstane také open-source. Jelikož k této práci je potřeba zveřejnit zdrojové kódy, můžeme knihovnu iTextSharp volně použít.

Abychom uživateli ukázali, jak stavbu sestavit, potřebujeme jednotlivé kroky vizualizovat. Lze například zobrazit bloky vrstvu po vrstvě, začínaje těmi připojenými přímo k desce a postupovat k těm, které jsou nejvýše. Implementovat bychom to mohli tak, že schováme všechny bloky, a pak podle aktuální vrstvy bloky zobrazíme. Je potřeba ale vyřešit, jak to zobrazit v PDF dokumentu.

V Unity lze oznámit kameře, co má v daný moment vykreslit, pomocí volání metody `camera.Render`. Kamera tak nevykreslí scénu na obrazovku, ale vykreslí ji do speciální textury `RenderTexture`, která představuje virtuální obrazovku pro renderování. Výsledný „render“ je možné zakódovat do PNG formátu a uložit do PDF dokumentu.

Pro optimální zobrazení instrukcí je potřeba zvolit vhodné umístění kamery. Navíc lze využít ortografické projekce, což označuje způsob, jakým je scéna zobrazena bez ohledu na perspektivu. Na rozdíl od perspektivní projekce, která simuluje způsob, jakým vidíme svět kolem sebe, ortografická projekce zajišťuje konzistentní velikost objektů bez ohledu na vzdálenosti od kamery. Tento pohled je vhodný pro přesné a konzistentní vykreslování objektů, což je ideální pro zobrazení instrukcí.



Obrázek 8: Porovnání perspektivní a ortografické projekce.

Před samotným generováním kroků je vhodné uživateli vypsat seznam bloků potřebných k sestavení celé stavby. K tomu lze využít ikony bloků, které byly použity i pro tlačítka výběru bloků. Při generování stránky, funkce najde bloky využité ve stavbě, najde příslušnou ikonu a zjistí jejich počet. Bloky se netřídí podle barvy, stejně jako v oficiálních instrukcích. Je to proto, že na desce může být v nejhorším případě kombinace všech typů bloků ve všech barvách, což by vedlo k vložení až 60 obrázků do PDF dokumentu. To by snížilo čitelnost i efektivitu. Na stránku se všemi použitými bloky budeme vkládat ikony vedle sebe. Začne se umístěním první ikony a pozice následující se bude posouvat o šířku předešlé. Pokud se ikony nevejdou na řádek, budou pokračovat na dalším.

Pokračuje se generováním jednotlivých kroků. Generování spočívá v zachycování snímků desky, kde bloky jsou zobrazeny po vrstvách. Bloky na aktuální vrstvě budou mít výraznější barvu než bloky na nižších vrstvách, což uživateli usnadní rozlišení položených a nepoložených bloků. V každém kroku je potřeba vypsat bloky z dané vrstvy. Budeme postupovat obdobně jako v předchozím odstavci, ale tentokrát využijeme pouze jeden řádek. Proto musíme zjistit celkovou velikost všech ikon, které chceme vložit na stránku, a porovnat ji se zbývajícím prostorem stránky. Pokud se ikony na stránku nevejdou, budeme muset určit, jak moc se musí ikony zmenšit. Dále chceme ikony zarovnat na střed stránky. Musíme tedy vypočítat pozici první ikony, opět pomocí velikosti obrázků a zbylého prostoru stránky.

```
1  int totalImageWidth = GetImagesWidth(images.Keys);
2  float remainingSpace = pageWidth - offset * (images.Count - 1);
3
4  if (totalImageWidth > remainingSpace - margin) {
5     // velikost obrázku se určuje v procentech
6     scale /= totalImageWidth / (remainingSpace - margin);
7     foreach (var image in images.Keys) {
8        image.ScalePercent((int)scale);
9     }
10    totalImageWidth = GetImagesWidth(images.Keys);
11 }
12 x = (int)(remainingSpace - totalImageWidth) / 2;
```

Zdrojový kód 5: Vypočet velikosti ikon a souřadnice první ikony.

3 Komplikace při vývoji

Při tvorbě softwaru se vždy setkáváme s různými komplikacemi, což platí i pro tuhle práci. V této kapitole si popíšeme problémy, na které jsem během vývoje narazil.

3.1 Mesh a Box collidery

V Unity jsou různé typy colliderů, které lze použít. Mezi nejčastěji používané patří Mesh collider a Box collider, které se liší v definici oblasti kolizí. Oba typy mají své výhody a využití.

- Box collider je definován oblastí ve tvaru kvádru. Je ideální pro jednoduché tvary a je tvořen jednoduchou geometrií, což umožňuje jednoduchou detekci kolizí.
- Mesh collider je složitější typ collideru, který používá samotnou geometrii objektu pro definici oblasti kolize. Je vhodný pro složitější tvary, ale detekce kolizí je výpočetně náročná.

Při rozhodování o reprezentaci colliderů dílků stavebnice jsem narazil na problém. Ideální by bylo používat Mesh collidery pro detekci kolizí a zabránění položení bloku. Díky tomu by nebylo potřeba objekt rozdělit na podobjekty s vlastním Box colliderem, protože bychom si vystačili s jedním. Kvůli většímu počtu objektů by to ale bylo výpočetně náročné.

Navíc jsem v editoru implementoval metodu pracující s normály, získaných pomocí paprsku vyvolaným metodou `ScreenPointToRay`. Tyto normály určují směr, kterým se objekt odráží od povrchu, se kterým koliduje. Takhle lze získat stranu bloku, na kterou uživatel najel myší. To umožňuje určit správný směr posunutí nového bloku vzhledem k již položenému bloku tak, aby nedošlo k prolnutí. Tento přístup by nefungoval při použití Mesh collideru. Jelikož povrch

některých bloků není rovný, získali bychom špatné normály a pokládaný blok by se mohl posunout špatným směrem. Proto jsem se rozhodl pracovat pouze s Box collidery.

3.2 Získávání komponent

Objektům lze přidat funkcionalitu pomocí komponent. Toho jsem využil při přidělování skriptu `BlockHandler`, který uchovává informace a funkcionalitu položených bloků. Kdykoliv bylo potřeba provést operaci s blokem, stačilo získat příslušnou komponentu a provést potřebnou operaci. V Unity ale práce s komponenty není jednoduchá. Pro práci s komponenty se používají dvě metody.

- **AddComponent**. Metoda přiřazuje objektu zvolenou komponentu a vrací na ni odkaz.
- **GetComponent**. Metoda vrací požadovanou komponentu.

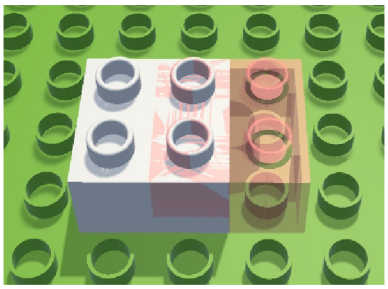
Komponentu bychom mohli získat pomocí metody `GetComponent`. Avšak tato metoda je neefektivní a snižuje výkon aplikace, pokud je volána často, například když se nachází v Update metodě. Proto bylo potřeba najít jiný přístup pro práci s komponentami.

Jelikož metoda `AddComponent` vrací odkaz na přidělenou komponentu, je možné tento odkaz uložit. Proto jsem zavedl proměnnou `blockCache`, ve které se tyto odkazy uchovávají. Ukládání bylo implementováno pomocí slovníku, jehož klíčem je objekt, kterému komponenta náleží. Když je potřeba pracovat s blokem, vyhledáme si ho ve slovníku a získáme odpovídající komponentu. Tímto způsobem jsme se vyhnuli opakovanému použití metody `GetComponent` a udrželi jsme si efektivitu programu.

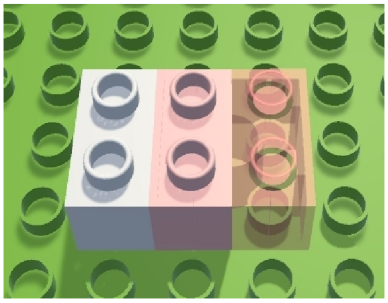
3.3 Z-Fighting

Při práci v 3D prostředí můžeme narazit na jev zvaný „z-fighting“. Ten se projeví tehdy, když plochy objektů ve scéně mají stejnou hloubku ve vztahu k pohledu z kamery. To vede k nepřesnostem ve vykreslování, kdy se polygony objektu překrývají a nejde určit, který z nich má být vykreslen první. To se projeví „blikáním“ ploch, což snižuje kvalitu jejich prezentace.

Existuje několik způsobů, jak tento jev vyřešit. Nejjednodušším řešením je posunout nebo zvětšit jeden z bloků tak, aby byl nad druhým. To bychom ale museli napsat více kódu a nemuselo by to vypadat přívětivě. Lepším přístupem je „offsetting“, což je technika, zajišťující posun vykreslovaných polygonů tak, aby se minimalizoval z-fighting. Tento posun se v Unity provádí pomocí shaderu, kde se nastaví proměnná `Offset`. Shader v Unity je speciální program, který ovlivňuje vzhled a chování materiálů a objektů ve scéně. Tím, že upravíme tento shader a aplikujeme ho na materiál průhledného bloku, zajistíme, že se průhledný blok vždy vykreslí nad klasickými bloky, což zamezí z-fightingu.



Obrázek 9: Ukázka jevu Z-Fighting v editoru.

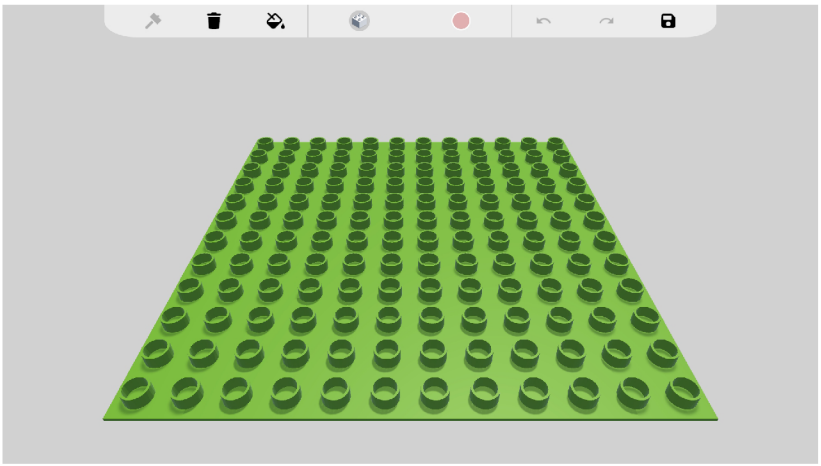


Obrázek 10: Korektní vykreslování bloků se stejnou hloubkou ve scéně.

4 Uživatelská příručka

V následující kapitole se seznámíme s ovládáním naprogramovaného editoru. Editor byl navržen pro platformu Windows a 64bitovou architekturu. Byl testován na verzích 10 a 11 tohoto operačního systému.

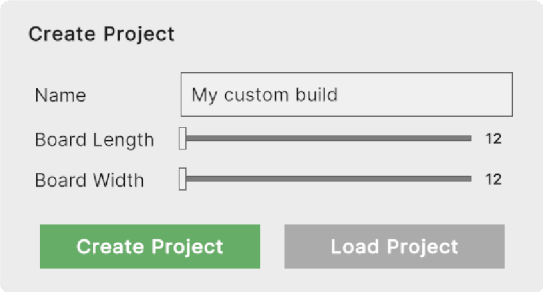
Pro spuštění aplikace není potřeba provádět žádnou instalaci. Stačí otevřít soubor **Hubelino Editor.exe** v adresáři **bin/**. Po spuštění aplikace se zobrazí okno pro vytvoření projektu, kde uživatel zadá požadované parametry. Po vytvoření projektu se zobrazí samotné prostředí editoru.



Obrázek 11: Prostředí editoru po vytvoření projektu.

4.1 Popis nástrojů

Po spuštění aplikace se zobrazí okno tvorby nového projektu. Zde si uživatel může zvolit název projektu a velikost desky. Také má možnost načíst již existující projekt ze souboru.



Obrázek 12: Okno vytvoření nového projektu.

Po vytvoření projektu se přesuneme do editoru. Na horní liště najdeme nástroje určené pro práci s editorem. Postupně si popíšeme funkce jednotlivých nástrojů, které jsou uspořádány zleva doprava.

- **Builder (Stavitel)**. Umožní pokládat vybrané bloky na desku.
- **Remover (Odstraňovač)**. Odstraňuje vybrané bloky.
- **Bucket (Kbelík)**. Obarví vybraný blok na zvolenou barvu.
- **Menu bloků**. Zde si uživatel zvolí, jaký blok chce pokládat.
- **Menu barev**. Zde si uživatel zvolí, jakou barvou chce obarvit bloky.
- **Undo (Zpět)**. Odvolá poslední provedenou operaci.
- **Redo (Znovu)**. Znovu provede poslední odvolanou operaci.
- **Soubory**. Zde si uživatel zvolí, zda chce vytvořit, uložit nebo načíst projekt. Také se může vygenerovat instrukce k postavené stavbě.

Obrázek 13: Nástroje v editoru.

4.2 Ovládání

Pro usnadnění práce s editorem byl program navržen tak, aby šel ovládat pouze myší. Pro zvýšení uživatelského komfortu byly ale implementovány klávesové zkratky, které ovládají různé prvky editoru.

Kamera
Pro snadnější orientaci má uživatel možnost ovládat kameru. Při stisknutí kolečka myši a následným táhnutím lze kameru otáčet a pravým tlačítkem myši ji lze posouvat. Přiblížení nebo oddálení kamery lze provést podržením klávesy CTRL a následnou rotací kolečka myši. Kameru lze posouvat vertikálně podržením klávesy CTRL a stisknutím pravého tlačítka myši.

Pokládání bloků
Pro pokládání bloků přejdeme do režimu stavby kliknutím na ikonu kladívka v nabídce nástrojů nebo klávesovou zkratkou „1“. Pak stačí najet kurzorem myši na desku, na které se zobrazí náhled bloku k položení. Barva náhledu indikuje možnost položení bloku. Pokud je barva bílá, je možné blok umístit kliknutím levého tlačítka myši. Pokud je červená, blok nelze položit, jelikož se buď prolíná s jiným blokem, levituje nebo nelze propojit s jiným blokem.

Rotace
Rotaci pokládaného bloku můžeme změnit dvěma způsoby. Buď rotací kolečka myši nebo stisknutím klávesy „x“.

Odstranění bloků
Pro odstranění bloku přejdeme do režimu odstraňování kliknutím na ikonu koše nebo stisknutím klávesové zkratky „2“. Pak stačí najet myší na blok, který chceme odstranit a kliknout levým tlačítkem myši. Pokud odstranění bloku přeruší propojení jiných bloků k desce, tyto bloky se označí jako levitující, což poznáme červeným zvýrazněním. Tyto bloky se buď musí odstranit nebo je potřeba vložit nové bloky tak, aby z levitujících bloků opět vedla cesta k desce.

Přebarvení bloků
Pro přebarvení položeného bloku přejdeme do režimu barvení kliknutím na ikonu kbelíku nebo stisknutím klávesové zkratky „3“. Pak stačí zvolit barvu, najet myší na blok, který chceme přebarvit, a stisknout levé tlačítko myši. Některé bloky nelze přebarvit na jinou barvu, například stavební bloky, které tvoří základ stavby. Tyto bloky mohou být pouze bílé, proto při jejich zvolení není možné měnit aktuální barvu.

Výběr bloku
Pro zvolení jiného aktuálního bloku klikneme na ikonu aktuálně používaného bloku. Zobrazí se menu, ve kterém kliknutím zvolíme, jaký blok chceme pokládat. Pomocí klávesových zkratek „a“ a „d“ lze mezi bloky

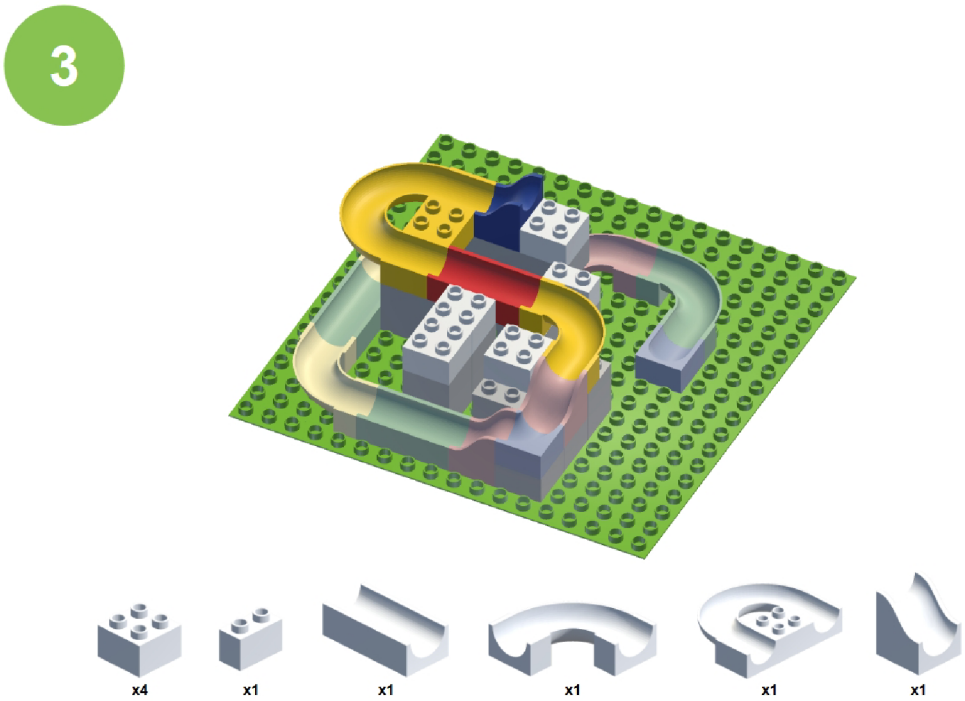
cyklit. Pro změnu barvy pokládaného bloku otevřeme menu barev pomocí tlačítka s aktuální barvou. Mezi barvy lze cyklit pomocí klávesové zkratky „c“. Pokud může aktuální blok být pouze bílý, menu barev nelze otevřít.

Krok zpět a dopředu

Tlačítka Undo a Redo slouží pro odvolání nebo opětovné provedení poslední operace, jako je položení bloku, odstranění nebo přebarvení. Maximum operací, které lze odvolat, je v editoru 10.

Soubory

Následují tlačítka pro vytvoření, uložení a načtení projektu, včetně tlačítka pro vygenerování instrukcí. Po stisknutí tlačítka vytvoření projektu se zobrazí stejné okno, jako při prvním spuštění editoru. Po stisknutí ostatních tlačítek se zobrazí okno, ve kterém vybereme umístění pro uložení nebo načtení souboru. Tlačítko uložení ukládá informace o stavbě do souboru, který lze později načíst pomocí tlačítka načtení. Tlačítko generování instrukcí vytvoří PDF soubor. Na první straně je zobrazena celá stavba a na druhé jsou vypsány všechny bloky, které jsou potřeba pro sestavení stavebnice. Na následujících stranách jsou vizuálně pospány jednotlivé kroky, které ukazují, jak stavbu sestavit vrstvu po vrstvě, a také, které bloky byly použity na konkrétní vrstvě. Tento PDF soubor se po vygenerování automaticky zobrazí.



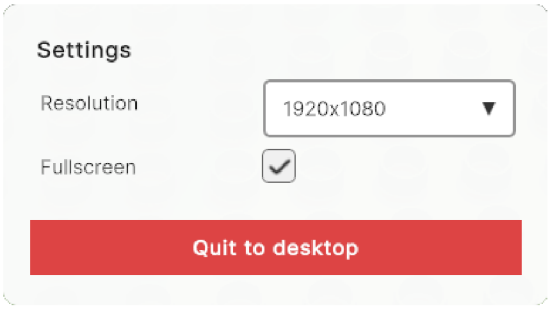
Obrázek 14: Příklad instrukce vygenerovaného PDF souboru.

4.3 Limit výšky stavby

V editoru je možné vytvářet stavby, které nejsou výškově omezeny. Aby ale bylo možné vygenerovat instrukce v přijatelném čase, je v editoru nastaven výškový limit. Tento limit nebrání uživateli v konstrukci stavby libovolné výšky, ale omezuje počet pater ve vygenerovaných instrukcích. Limit je nastaven na 14 pater. Překročení limitu je signalizováno pomocí vykřičníku v pravém dolním rohu obrazovky.

4.4 Nastavení

V editoru je k dispozici menu pro změnu nastavení, které se zobrazí stisknutím klávesy „Esc“. Zde můžeme upravit rozlišení obrazovky, zvolit režim celé obrazovky nebo aplikaci vypnout. Nejvyšší možné rozlišení je „1920x1080“ a nejmenší „800x600“. Opětovným stisknutím klávesy „Esc“ se okno skryje.



Obrázek 15: Menu nastavení.

Závěr

Ve výsledku byl vytvořen kompletní editor kuličkových stavebnic Hubelino. Obsahuje mnoho nástrojů a přívětivé uživatelské rozhraní, které značně uživateli usnadní práci. K postavené stavbě v editoru lze vygenerovat instrukce, které popisují, jak stavbu sestavit krok po kroku.

V editoru je určitě prostor pro rozšíření. Například by bylo možné přidat více druhů bloků, nebo dokonce i implementovat možnost spuštění kuličky po dráze. Navíc editor neobsahuje žádné zvuky, takže by mohly být v budoucnu přidány.

Při tvorbě editoru jsem získal několik zkušeností s herním enginem Unity, které do budoucna určitě využiji. Také jsem si připomenul, jak pracovat s programem Blender, když jsem upravoval rozměry 3D modelů.

Conclusions

As a result, a complete editor for Hubelino marble run building kit was created. It contains many tools and a friendly user interface that will make work greatly easier for the user. Instructions can be generated for the constructed build in the editor, which will describe how to assemble the build step by step.

There is room for improvement in the editor. For example, it would be possible to add more types of blocks, or even an option to launch a ball along the track. Also, editor doesn't include any sounds, so they may be added in the future.

While creating the editor, I gained some experience with the Unity game engine, which I will definitely use in the future. I also remembered how to work with Blender, when I was making changes to the 3D models.

A Obsah elektronických dat

`bin/`
: V tomto adresáři se nachází spustitelná aplikace.

`builds/`
: V tomto adresáři se nachází testovací soubory pro načtení stavby v editoru.

`doc/`
: Zde se nachází samotný text bakalářské práce ve formátu PDF a všechny zdrojové kódy, které jsou potřebné k jeho vytvoření.

`src/`
: V tomto adresáři se nachází projekt, který je možné nahrát do editoru Unity.

`README.txt`
: Textový soubor obsahující potřebné informace ke spuštění aplikace nebo projektu.

Literatura

[1] *Marble run, mind games, building blocks*. 2023. Dostupný také z: ⟨https://www.hubelino.com/⟩.

[2] *O LEGO® Digital Designer softwaru*. Dostupný také z: ⟨https://www.lego.com/cs-cz/service/help/apps_video_games_device_guides/about-lego-digital-designer-kA009000001dcdmCAA⟩.

[3] *BrickLink Studio 2.0*. Dostupný také z: ⟨https://www.bricklink.com/v3/studio/download.page⟩.

[4] *Unity Real-Time Development Platform*. Dostupný také z: ⟨https://unity.com/⟩.

[5] *Thingverse*. Dostupný také z: ⟨https://www.thingiverse.com/⟩.

[6] *CC BY 4.0 Deed | Attribution 4.0 International | Creative Commons*. Dostupný také z: ⟨https://creativecommons.org/licenses/by/4.0/⟩.

[7] *Comparison of UI systems in Unity*. Dostupný také z: ⟨https://docs.unity3d.com/Manual/UI-system-compare.html⟩.

[8] *UnityStandaloneFileBrowser: A native file browser for unity standalone platforms*. Dostupný také z: ⟨https://github.com/gkngkc/UnityStandaloneFileBrowser⟩.

[9] *Hubelino Instructions*. 2023. Dostupný také z: ⟨https://www.hubelino.com/instructions/⟩.

[10] *iTextSharp | iText PDF*. 2021. Dostupný také z: ⟨https://itextpdf.com/products/itextsharp⟩.