Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Aplikace pro generování umělých datových sad

Bc. Lukáš Milar

Diplomová práce

2023

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Lukáš Milar**
Osobní číslo:	**I21282**
Studijní program:	**N0613A140007 Informační technologie**
Téma práce:	**Aplikace pro generování umělých datových sad**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování

Cílem diplomové práce je tvorba aplikace pro automatické generování umělých datových sad pro učení neuronových sítí. Práce bude zaměřena na tvorbu vstupně-výstupního páru snímků, kde vstupní snímek bude odpovídat svrchnímu pohledu na definovaný prostor s náhodně rozmístěnými objekty jednoho typu a výstupní snímek bude obsahovat vhodně vyznačené úchopové body dosažitelných objektů.

Vstupem do aplikace bude 3D model objektu ve standardně používaném formátu (STEP, STL, OBJ atp.) spolu s velikostí zájmové oblasti a definicí úchopových bodů objektu. Výstupem bude sada párů obrázků (dle definovaných parametrů) o dostatečném počtu pro trénování neuronové sítě.

Teoretická část: Stručná rešerše existujících nástrojů pro generování umělých datových sad (datasetů) a nástrojů pro kompozici 3D scény a tvorby jejich průmětů.

Praktická část: Tvorba samostatné aplikace s grafickým rozhraním pro nahrání modelu objektu a definici potřebných parametrů. Testování funkce aplikace a zhodnocení jejího výkonu (časová náročnost pro generování sady). Dokumentace softwaru včetně scénáře demonstrujícího použití.

Rozsah pracovní zprávy: **cca 50 stran**
Rozsah grafických prací:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

GORTLER, Steven J. *Foundations of 3D computer graphics*. Cambridge, Massachusetts: The MIT Press, 2012. ISBN 978-0-262-01735-0.

WATT, Alan. *3D computer graphics*. 3rd ed. New York: Pearson Education, 2000. ISBN 0-201-39855-9.

Vedoucí diplomové práce: **Ing. Dominik Štursa**
Katedra řízení procesů

Datum zadání diplomové práce: **8. listopadu 2022**
Termín odevzdání diplomové práce: **19. května 2023**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

prof. Ing. Antonín Kavička, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 30. listopadu 2022

Prohlašuji:

Práci s názvem Aplikace pro generování umělých datových sad jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 25. 8. 2023

Bc. Lukáš Milar

Poděkování

Chci poděkovat vedoucímu mé diplomové práce, Ing. Dominikovi Štursovi, za odborné vedení a důležité rady při tvorbě této práce. Také chci poděkovat své rodině a přítelkyni za morální podporu po celou dobu tvorby této práce a samotného studia. Své přítelkyni chci také poděkovat za veškerou trpělivost, kterou vůči mně při psaní této práce prokázala.

ANOTACE

Tato diplomová práce je zaměřena na tvorbu aplikace s grafickým uživatelským rozhraním umožňující generaci datových sad pro trénování umělých neuronových sítí. V teoretické části je nejprve přiblížena problematika umělých datových sad. Poté jsou prozkoumány existující nástroje pro generaci datových sad a existující možnosti pro generaci 3D scény a vytváření jejích průmětů. Následně je blíže představen engine Unity. V praktické části jsou nejprve definovány požadavky a vytvořen návrh grafické podoby a základní struktury výsledného projektu. Dále je vytvořen backend, grafická část aplikace a dokumentace výsledné aplikace. Nakonec je provedeno testování funkcí aplikace společně s vyhodnocením jejího výkonu.

KLÍČOVÁ SLOVA

Umělá inteligence, 3D, vykreslování, umělý dataset, náhodný rozptyl, Unity, Python, C#, Spring Boot, RabbitMQ

TITLE

Application for Artificial Datasets Generation

ANNOTATION

This master thesis is focused on the creation of an application with Graphical User Interface, which allows to generate artificial datasets to be used for Neural Network training. First, the problematics of artificial datasets are brought closer in the theoretical part. Afterwards, existing tools for generation of artificial datasets and tools for 3D scene management and creation of the scenes' projections are explored. Then engine Unity is introduced in more detail. In practical part, the requirements, design of the graphic form and basic structure of the final project are created first. Further in this part, backend, graphics part of the application and the documentation of the final application are created. Finally, tests are run on the application's features along with performance evaluation.

KEYWORDS

Artificial intelligence, 3D, Rendering, Artificial Dataset, Random Distribution, Unity, Python, C#, Spring Boot, RabbitMQ

OBSAH

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

SEZNAM ZKRATEK

2D	Two dimensional
3D	Three dimensional
API	Application Programming Interface
ASSIMP	Asset Importer Lib
CAD	Computer Aided Design
CSRF	Cross-site request forgery
DTO	Data Transfer Object
GAN	Generative adversarial network
GNU	GNU's Not Unix!
GPL	GNU General Public License
GUI	Graphical User Interface
IDE	Integrated Development Environment
JDK	Java Development Kit
JSON	JavaScript Object Notation
LM	Language Model
MDL	Material Definition Language
NPM	Node Package Manager
PDF	Portable Document Format
REST	Representational State Transfer
RPC	Remote Procedure Call
RTX	Ray Tracing Texel eXtreme
SDK	Software development kit
UML	Unified Modeling Language
USD	Universal Scene Description
UUID	Universally Unique Identifier
WYSIWYG	What You See Is What You Get
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

ÚVOD

Umělá inteligence v současné době zažívá rapidní vývoj napříč širokým spektrem řešených problémů. Pro udržení rychlosti a kvality tohoto vývoje je však potřeba velké množství vhodných dat pro její trénování, což může v případě skutečných dat představovat problém. Řešení tohoto problému tak může spočívat v tvorbě umělých datových sad.

Cílem této práce je představit existující řešení pro generování umělých datových sad a nástrojů pro správu 3D scén a tvorby jejich průmětů. Dalším cílem je vytvoření aplikace pro generování umělých datových sad, ze kterých bude poté možné trénovat umělé neuronové sítě. Aplikace by měla umožnit uživateli načíst trojrozměrný objekt z některého ze standardních 3D formátů, definovat na něm úchopové body a následně specifikovat parametry pro generaci scén. Na základě těchto parametrů má poté aplikace vygenerovat scény s náhodně rozmístěnými objekty. Výstupem aplikace mají být páry snímků výsledných scén s úchopovými body a bez nich. Konečným cílem je poté pro tuto aplikaci vytvořit dokumentaci a ukázkový příklad jejího použití.

Teoretická část je zaměřena na problematiku existujících řešení v oblasti generování umělých datových sad a nástrojů pro správu 3D scén a tvorbu jejich průmětů. Před samotným představením jednotlivých řešení bude také přiblížena samotná problematika generování umělých datových sad a některé současné trendy v této činnosti. V závěru této části bude poté blíže představen herní engine Unity.

Zaměření praktické části spočívá v samotné implementaci výsledné aplikace. Nejprve budou v této části definovány základní požadavky a případy užití, na jejichž základě poté bude připraveno základní rozdělení aplikace a návrh grafického uživatelského prostředí. Na základě připravených podkladů bude poté provedena implementace jednotlivých částí aplikace a následně vyhotovena její dokumentace. Nakonec bude vytvořená aplikace podrobena testování, na jehož základě bude vyhodnocena její funkčnost a výkon.

1 NÁSTROJE PRO GENEROVÁNÍ UMĚLÝCH DATOVÝCH SAD

Účelem této kapitoly je umožnit jisté základní nahlédnutí do problematiky nástrojů pro generování umělých datových sad. Provedená rešerše si kladla za cíl nalézt některá existující řešení a mimo jiné tak poskytnout jistou inspiraci pro implementaci výsledné aplikace.

Před samotným představením nástrojů, které umožňují generovat umělé datové sady, je však vhodné více přiblížit problematiku umělých datových sad jako takových. Především je v této části diskutován jejich význam vůči datovým sadám získaných sběrem dat a jaké výhody tento přístup přináší.

Skutečná data mohou obsahovat citlivé informace skutečných subjektů a platí na ně tak různé regulace o způsobu jejich použití a informování subjektů o jejich sběru. Jelikož je potřeba tato data sbírat od skutečných subjektů během používání aplikace, nebo pomocí dotazníků, je jejich množství omezené. [1]

V případě uměle vytvořených dat však tato omezení neplatí, jelikož pouze připomínají původní dataset a je možné je generovat v libovolném množství. Při trénování AI modelů je lze použít společně se skutečnými daty, nebo je jimi zcela nahradit. Jejich použití lze vnímat jako alternativu s vyšší ochranou soukromí oproti produkčním datům, protože neobsahují citlivé osobní informace skutečných subjektů. [1]

Problém nedostatku vhodných dat je dnes jednou z hlavních překážek inovace a lze jej řešit umělými daty, které lze generovat pomocí simulací či algoritmů. Aby měly moderní umělá data stejnou distribuci a velikost jako trénovací data, jsou při jejich tvorbě využívány modely hlubokého učení jako generativní soupeřící sítě (GAN) či jazykové modely (LM). V některých případech mohou dokonce oproti původním datům dosahovat umělá data vyšší kvality. [2]

Umělá data lze kategoriemi rozdělit například na video, zvuky či textová data, mezi která patří mimo jiné tabulková data či přirozený jazyk. V posledních letech došlo k velkému posunu v generativních soupeřících sítích díky snaze o generaci realistických obrázků. [2]

Generativní soupeřící sítě (GAN) převádí generativní modelování z úlohy bez učitele na úlohu s učitelem. K tomu jsou využívány dva podmodely, které se trénují současně a na-

vzájem spolu soupeří. Model generátoru generuje nová data, která model diskriminátoru označuje za skutečná, nebo uměle vygenerovaná. Jakmile dokáže model generátoru oklamat model diskriminátoru v polovině případů, trénování končí a model generátoru je vnímán jako schopný generovat uvěřitelná data. Tyto sítě umožňují napříč širokým spektrem problémových domén generovat realistická data a v této oblasti dochází k rychlým změnám. [3]

1.1 StyleGAN3

Nástroj StyleGAN je založen na generativních soupeřících sítích a umožňuje generovat uvěřitelné obrázky zvířat, aut, krajiny či lidských tváří. První verzi vydali výzkumníci ze společnosti NVIDIA v roce 2018 a aktuální verze, známá jako StyleGAN3, pak byla vydána v říjnu 2021. S tímto nástrojem lze také například rotovat objekty či provádět jiné úpravy generováním interpolací mezi obrázky. Díky jeho skvělým výsledkům při generaci se stal velmi populární. [4]

1.2 GAN Lab

GAN Lab běží přímo v prohlížeči a pro své spuštění tak nic jiného nevyžaduje. Veškeré jeho funkce jsou implementovány v jazyce JavaScript a pro trénování modelů v prohlížeči používá knihovnu akcelerovanou GPU jménem Tensorflow.js. S tímto nástrojem lze přímo v prohlížeči generovat a experimentovat s modely generativních soupeřících sítí pro distribuci 2D dat a jejich vnitřní fungování vizualizovat. [5]

1.3 Synthesized.io

Synthesized založili výzkumníci z předních univerzit Spojeného království. Tento nástroj řeší náhradu citlivých dat za taková, která jsou podobná datům produkčním a zároveň mají správnou velikost, čímž automatizuje proces přístupu k datům o vysoké kvalitě. V generaci dat podobných produkci napříč všemi strukturovanými formáty a jejich následnému využití při vývoji aplikací a strojovém učení byl tento nástroj průkopníkem. [6]

1.4 Gretel AI

Gretel poskytuje jednoduché API, se kterými lze generovat libovolné množství dat, anonymizovat je či z nich odstraňovat zaujatosti. Pomocí jednoduchého webového rozhraní lze ovládat jeho služby a lze jej dle potřeby nasadit, či plně spravovat v cloudu. Mimo jiné poskytuje také referenční příklady, které jsou open source. [7]

1.5 Benerator

Benerator je open source software navržený tak, aby jej mohli používat také nevývojáři a slouží ke generaci, migraci a zamlžování dat pro jejich další použití při trénování a testování. Generace probíhá s pomocí nástroje v příkazové řádce a popisu dat ve formátu XML. [1]

1.6 Shrnutí

Vzhledem k různým omezením z hlediska anonymity dat se jeví generace umělých datových sad jako efektivní řešení pro potřeby trénování neuronových sítí. Generovat tato data lze různými způsoby. Jeden z těchto způsobů zahrnuje využití nástrojů hlubokého učení, jako jsou například GAN sítě pro generaci na základě existujících dat. Další možností je využít počítačové algoritmy či simulace.

2 NÁSTROJE PRO KOMPOZICI 3D SCÉNY A TVORBY JEJICH PRŮMĚTŮ

Existuje mnoho různých nástrojů pro správu 3D scén a tvorby jejich průmětů. Oproti jejich celkovému množství zde bude představen pouze malý zlomek. Účelem této kapitoly je tak poskytnout jistý základní náhled do této problematiky a využít provedenou rešerši k zhodnocení možných přístupů při následné implementaci výsledné aplikace.

2.1 Blender

S pomocí Blenderu lze pracovat napříč různými 3D operacemi od modelování a vykreslování až po například tvorbu videa. Jedná se o open source projekt pod licencí GNU General Public License (GPL). Aplikace je multiplatformní a pro konzistentní zážitek používá OpenGL pro své rozhraní. V jazyce Python lze s pomocí jeho API vytvářet specializované nástroje, či jinak aplikaci upravovat. [8]

Zajímavým konceptem v souvislosti s touto aplikací jsou Blender Apps. Jak je uvedeno na blogu Blenderu, Blender Apps mají stejné funkce jako Blender samotný a tyto funkce lze ještě dále rozšiřovat. Jejich princip spočívá v zaměření na konkrétní funkce a možnosti distribuce bez nutnosti instalace. V závislosti na jejich návrhu nemusí být předešlá znalost Blenderu nutná [9]. V době psaní této práce se však stále nejednalo o řešení připravené pro produkci.

2.2 Autodesk Maya

Autodesk Maya slouží k tvorbě 3D grafiky a jedná se o velmi populární profesionální aplikaci využívanou například při tvorbě animovaných filmů či počítačových her. Nachází se zde vestavěné jazyky Maya Embedded Language (MEL) a Python, se kterými lze díky její otevřené architektuře veškeré operace v ní programovat či skriptovat. Pro vykreslování lze využít vykreslovací engine Arnold či některé vykreslovače třetích stran jako například Renderman. [10]

2.3 Houdini

V Houdini jsou pomocí sítí uzlů reprezentovány jednotlivé kroky tvořící výsledný záběr. Tvorba velkých prostředí je v něm snadná díky jeho procedurálnímu přístupu. Do prostředí Houdini Solaris, se kterým lze vytvářet shadery, spravovat světla a obecně definovat vzhled snímku, je integrován vykreslovač Karma. Lze také využívat vykreslovače třetích stran jako Autodesk Arnold či RenderMan od Pixaru. [11]

2.4 Unity

S enginem Unity lze na platformách Windows, Mac a Linux vytvářet realtimové interaktivní zážitky [12]. S pomocí platformy Unity není potřeba při tvorbě aplikace vytvářet vlastní realtimový 3D framework a díky partnerství se všemi velkými platformami lze, bez starostí s vydanými verzemi, na všech podporovaných zařízeních plně využít jejich výkon [13]. Tento engine se tak zdá jako vhodný základ pro vytvoření aplikace pro praktickou část této práce.

2.5 Godot

V enginu Godot je k dispozici mnoho nástrojů umožňujících se plně soustředit na vývoj her. S jeho pomocí lze v jednotném prostředí vytvářet 2D a 3D hry pro různé platformy, mezi kterými jsou mimo jiné také Linux, Windows a macOS. Jelikož je open source, zdarma a šířen pod licencí MIT, není potřeba při jeho použití nic platit a jeho vývoj je řízen komunitou. [14]

2.6 NVIDIA Omniverse USD Composer

S aplikací NVIDIA Omniverse USD Composer lze sestavovat velké scény, spravovat v nich světla a následně je vykreslovat a simulovat. Scéna je popsána a v paměti uchovávána na základě formátu USD od Pixaru, jehož pokročilé funkce jako vrstvy či instancování využívá. Vykreslovač je založen na RTX a s grafickou kartou NVIDIA RTX tak lze v kombinaci s MDL Material Description generovat fyzikálně přesné a zajímavé světy. [15]

Pro úplnost je vhodné více přiblížit formát USD. Jak je uvedeno na stránkách dokumentace, USD je veřejně dostupný a umožňuje v rámci jednoho scénografu nedestruktivně scény editovat, přesouvat je mezi různými aplikacemi a organizovat v nich libovolné množství assetů za pomoci jediné konzistentní API. Pro kompozici a kódování dat v jiných doménách jej lze udržitelně rozšířit, jelikož jeho kompoziční engine a hlavní scénograf si žádnou konkrétní doménu neuvědomují. [16]

2.7 Shrnutí

Kromě nástrojů zmíněných výše existuje mnoho dalších nástrojů pro správu 3D scény a tvorby jejích průmětů. Jak je zřejmé již z uvedeného výčtu, patří mezi ně také herní enginy, s jejichž pomocí lze vytvářet další samostatné aplikace.

Před samotným zahájením tvorby praktické části této práce bylo potřeba učinit rozhodnutí, zda se těmito nástroji pouze inspirovat při tvorbě vlastního nástroje, nebo některý z nich použít jako základ, na kterém bude výsledná aplikace vystavěna.

Jelikož tvorba veškerých komponent aplikace pro správu 3D scény včetně implementace vlastního vykreslovače není předmětem této práce a představovala by značné zvýšení komplexity a časové náročnosti, byla zvolena varianta využití existujícího řešení. Vzhledem k předešlým zkušenostem autora byl pro tento účel konkrétně zvolen herní engine Unity.

3 PRÁCE S ENGINEM UNITY

Jak již bylo řečeno v předešlé kapitole, pro implementaci výsledné aplikace byl zvolen herní engine Unity. Konkrétně byla zvolena verze Unity 2022.3.0f1. Základní koncepty tohoto enginu jsou popsány v následujících podkapitolách.

3.1 Jazyk a API v Unity

Před bližším představením ostatních konceptů tohoto herního enginu je důležité krátce představit konkrétní jazyk a API, které se při práci s ním používají. Jak uvádí dokumentace Unity, ve verzi 2022.3 jsou podporovány profily .NET Standard 2.1 a .NET Framework 4.8, přičemž ve výchozím nastavení je úroveň kompatibility API nastavena na první z nich [17, s. .NET profile support]. Nativně je podporován jazyk C# s možností využití dalších jazyků .NET. [17, s. Creating and Using Scripts]

V následujících částech této práce se v rámci tohoto enginu projednává pouze jazyk C# v příslušné verzi. Jak uvádí dokumentace Unity, pro kompilaci je využit kompilátor Roslyn pro verzi C#9.0, přičemž nejsou zcela podporovány všechny vlastnosti této verze. [17, s. C# compiler]

3.2 Základní koncepty

Před popisem práce s tímto enginem je vhodné také představit některé jeho základní koncepty. V rámci této podkapitoly jsou zmíněny pouze ty, které jsou nějakým způsobem relevantní vzhledem k vývoji výsledné aplikace.

Prvním zde pojednávaným konceptem je herní objekt (angl. GameObject). Jak je uvedeno v dokumentaci, herním objektem jsou reprezentovány veškeré objekty ve hře, ačkoliv sám o sobě žádné funkce nevykonává. Jednotlivé funkce implementují až komponenty, pro které herní objekty slouží jako kontejnery. V Unity Editoru se tak jedná o nejdůležitější koncept. [17, s. GameObjects]

Komponenty představují pro herní objekt jeho funkční části. Editací vlastností komponent lze upravit chování herního objektu, ke kterému jsou připojeny. Jejich seznam v okně inspektoru pak lze zobrazit výběrem daného herního objektu ve scéně nebo v okně hie-

rarchie [17, s. Introduction to components]. Přidání či úpravu funkcí herních objektů lze provést použitím různých komponent a k úpravě jejich vlastností lze využít buďto skripty, nebo okno inspektoru. [17, s. Use components]

Komponentou Transform disponuje každý herní objekt a uchovává informace o jeho stavu v rodičovské hierarchii a o jeho rotaci, pozici a velikosti. Herní objekt bez ní nelze vytvořit, ani ji z něho nelze odstranit. Herní objekt může mít pouze jednoho rodiče, ale mnoho potomků. Pokud má herní objekt potomky, přebírají vlastnosti z jeho komponenty Transform a jejich rotace, velikost a pozice se tak mění podle jejich rodiče. [17, s. Transforms]

Navzdory veliké všestrannosti vestavěných komponent v Unity je pro implementaci vlastních funkcí potřeba vytvářet vlastní komponenty, což lze provést pomocí skriptů. Skripty dědí ze třídy MonoBehaviour a jsou vytvářeny zpravidla přímo v Unity. Jejich použitím lze například reagovat na uživatelský vstup, upravovat vlastnosti ostatních komponent, či vyvolávat herní události. [17, s. Creating and Using Scripts]

Dalším zde popsaným konceptem jsou tzv. prefaby. Jak je uvedeno v dokumentaci, herní objekt a všechny jeho potomky a komponenty včetně hodnot jejich vlastností lze spravovat a uchovávat jako znovupoužitelný asset pomocí systému prefabů. Ve scéně lze poté z těchto prefabů vytvářet instance a jednotlivé prefaby je možné také vnořovat a vytvářet tak komplexní hierarchie. Instance prefabů lze od ostatních instancí stejného prefabu odlišit překrytím jejich nastavení a tato překrytí lze seskupit vytvořením varianty prefabu. [17, s. Prefabs]

Aby bylo možné do aplikace přidat podporu například notifikačních zvuků, je nutné znát komponentu, která je schopná tyto zvuky přehrát. Jak je uvedeno v dokumentaci Unity, k přehrávání audio klipů ve scéně slouží komponenta Audio Source. Touto komponentou lze upravovat vlastnosti zvuku, nebo spustit a zastavit přehrávání daného zvukového klipu. [17, s. Audio Source]

Je zřejmé, že 3D geometrická síť modelu musí být nějakým způsobem reprezentována. V tomto enginu je tak provedeno třídou Mesh. Jak je uvedeno v dokumentaci Unity, třída Mesh se skládá z vícero polí trojúhelníků a lze s ní modifikovat a vytvářet sítě. Data plošek se skládají ze tří indexů vrcholů daného trojúhelníku, přičemž pro každý vrchol

lze uchovávat jeho normály, tečny, pozici a až 8 souřadnic textur [18, s. Mesh]. Lze tedy usoudit, že využitím této třídy je možné zobrazit dynamicky načtené geometrické sítě, které byly předtím převedeny do polí.

Bez znalostí konceptů popsaných výše by jistě nebylo možné efektivně spravovat projekt v tomto herním enginu. Za jeden z nejzásadnějších konceptů pro tuto práci pak lze vyjma herního objektu označit třídu Mesh a její význam při reprezentaci geometrických sítí.

3.3 Tvorba skriptů

V předešlé podkapitole byly představeny obecné základní koncepty potřebné pro obecnou práci v Unity. Pro vytváření samotné logiky pomocí skriptů je však důležité znát další koncepty, které jsou popsány níže.

Všechny skripty v Unity po vytvoření automaticky dědí ze třídy MonoBehaviour. Pomocí této třídy se lze napojit na události jako Start a Update a lze díky ní vytvořené skripty připojit k hernímu objektu. [17, s. Important Classes - MonoBehaviour]

Atribut RequireComponent je užitečný v případech, kdy je požadováno, aby na daném herním objektu byla přítomna konkrétní komponenta. Jelikož je daná komponenta při použití tohoto atributu automaticky přidána k hernímu objektu, snižuje se pravděpodobnost chyb při jeho nastavování. [18, s. RequireComponent]

Ve většině případů je metoda po jejím zavolání dokončena během jediného snímku. Pokud je potřeba provést některou operaci v průběhu vícero snímků, lze k tomu použít korutiny. Korutina je metoda, jejíž synchronní operace běží na hlavním vlákně a nejedná se tak o vlákno. Jelikož umí pozastavit své vykonávání a pokračovat ze stejného bodu na dalším snímku, lze s ní vykonávat úlohy napříč vícero snímky. [17, s. Coroutines]

Ačkoliv skriptovací objekt (angl. ScriptableObject) dědí ze stejné základní třídy jako MonoBehaviour, nelze jej připojit k hernímu objektu ve scéně a musí být místo toho v projektu uložen jako asset. Jelikož je to kontejner dat, spočívá jeho hlavní využití v uchovávání velkého množství dat bez vytváření jejich zbytečných kopií jako tomu je v případě vytváření instancí. Jelikož využívají jmenný prostor a skriptování editoru, lze do nich v editoru hodnoty zapisovat i za běhu. V sestaveném prostředí jsou však určena pouze ke čtení dat vložených během vývoje. [17, s. ScriptableObject]

Se znalostí výše popsaných konceptů při tvorbě skriptů byly otevřeny různé možnosti při pozdější implementaci. Mezi tyto možnosti patří například vykonávání kódu napříč vícero snímky či uchovávání dat ve scéně bez využití instance třídy MonoBehaviour.

3.4 Nástroje pro tvorbu GUI

Výsledná aplikace potřebuje v její grafické části kromě samotné interakce s objekty ve scéně také jisté uživatelské rozhraní. Z tohoto důvodu jsou níže představeny možnosti, kterými lze tento požadavek uvnitř Unity naplnit.

První možností pro řešení této problematiky je UI Toolkit. Jak uvádí dokumentace Unity, UI Toolkit lze využít například k tvorbě uživatelského rozhraní pro editor či aplikací a her. Případné předešlé zkušenosti s tvorbou webových stránek jsou vzhledem k inspiraci webovými technologiemi přenositelné. [17, s. UI Toolkit]

Další možností je Unity UI. Jak je uvedeno v dokumentaci tohoto balíčku, Unity UI pro tvorbu uživatelského rozhraní využívá komponenty a herní objekty. Z tohoto důvodu s ním nelze vytvářet uživatelská prostředí pro editor, ale pouze pro aplikace a hry. [19, s. Unity UI: Unity User Interface]

Z výše zmíněných možností se může Unity UI Toolkit jevit jako vhodnější volba pro tvorbu uživatelského prostředí v rámci této práce. Mimo jiné vzhledem k předešlým zkušenostem však byl nakonec zvolen druhý popsaný přístup. Jak uvádí dokumentace, ačkoliv se má UI Toolkit v budoucnu stát doporučeným systémem pro tvorbu uživatelského prostředí, současná verze nepodporuje některé funkce Unity UI. [17, s. Comparison of UI systems in Unity]

Jak je uvedeno výše, lze v současné době využít vícero přístupů pro implementaci uživatelského rozhraní v Unity. Vzhledem k předešlým zkušenostem autora a delší době existence byl pro použití v konečné aplikaci zvolen přístup využívající Unity UI.

3.5 Rozšiřující balíčky

Pro některé funkcionality může být výhodnější použít existující knihovny namísto vlastní tvorby jejich implementace. V této části jsou tak popsány různé způsoby, kterými lze případné externí knihovny v Unity nainstalovat.

Zřejmě nejdůležitějším konceptem jsou v tomto ohledu balíčky. Jak je uvedeno v dokumentaci, balíčky mohou mít různý obsah od knihoven a nástrojů pro editor či runtime až po kolekce různých assetů. Pomocí okna Package Manager pak lze jednotlivé balíčky spravovat. [17, s. Unity's Package Manager]

Okno Package Manager však není jediný způsob, kterým lze spravovat balíčky. Jak uvádí Harlow, balíčky lze spravovat přes Unity Package Manager, nebo Unity Asset Packages. Unity Asset Packages lze snadno importovat a distribuovat, jelikož to jsou assety zabalené do souboru .unitypackage [20]. Vytvořit vlastní balíček assetů lze pomocí nabídky Assets > Export Package. Importovat tyto balíčky pak lze pomocí nabídky Assets > Import Package. [17, s. Asset packages]

Oproti balíčkům assetů je Unity Package Manager novější a je založený na NPM. Podobně jako v NPM jsou i zde balíčky definovány v package.json a narozdíl od balíčků assetů jsou tyto balíčky nahrávány do vlastní struktury složek namísto do složky assets. Balíčky lze touto formou získávat například z výchozího registru Unity, lokálního disku či Git repozitáře. [20]

Je zřejmé, že v herním enginu Unity lze využívat různé způsoby instalace externích balíčků, které pro něho byly vytvořeny. Tato informace se v případě některých potřebných funkcionalit ukázala při pozdější implementaci jako velmi důležitá a velice usnadnila jejich naplnění.

4 DEFINICE A SPECIFIKACE POŽADAVKŮ NA APLIKACI

Arlow ve své knize uvádí, že „Specifikace softwarových požadavků (SSP) je úplným začátkem procesu tvorby softwaru.“ [21, s. 76] Z tohoto důvodu byly před započetím implementace aplikace definovány a specifikovány požadavky na základě textu zadání této práce.

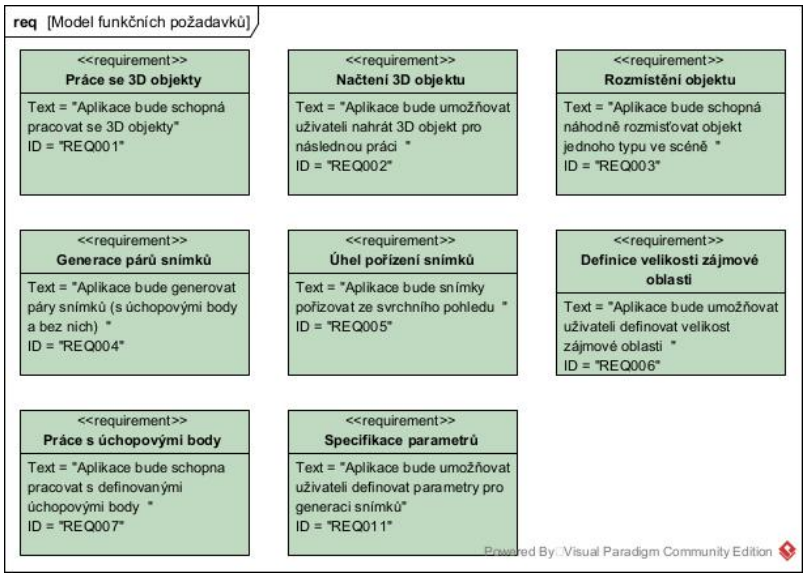
Požadavky lze rozdělit na množinu funkčních a nefunkčních požadavků. Funkční požadavky popisují, které služby se od systému požadují, zatímco nefunkční požadavky vymezují omezení systému či procesu vývoje jako jsou výkonnost, standardy, apod. Tyto požadavky lze zachytit pomocí modelu požadavků a modelu případu užití. [21, s. 75]

Výše popsané modely byly vytvořeny v jazyce UML pomocí nástroje Visual Paradigm Community Edition 17.1. Jak uvádí Arlow, jedná se o jazyk určený k vizuálnímu modelování systémů, který není vázaný na konkrétní metodiku. [21, s. 28]

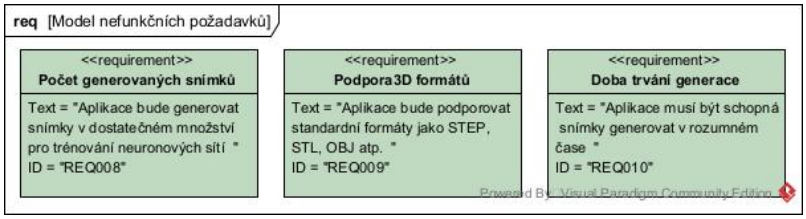
4.1 Vytvoření modelu požadavků

Ačkoliv jazyk UML řeší problematiku požadavků pomocí případů užití, je dle názoru mnoha analytiků a návrhářů potřeba také specifikovat systémové požadavky. K tomu lze využít jednoduchý formát, ve kterém každý požadavek obsahuje jedinečný identifikátor, název systému, klíčové slovo „bude“ a vykonávanou funkci. Výsledný model může být poté vytvářen ve speciálních inženýrských nástrojích či pomocí textu [21, s. 79–80]

Ve výsledném modelu bylo na základě textu zadání identifikováno 8 funkčních požadavků a 3 nefunkční požadavky (obr. 1, 2). Tyto požadavky zahrnují pouze zcela nezbytné funkce, tudíž v nich nejsou zahrnuty funkce vylepšující uživatelský zážitek.



Obrázek 1: Model funkčních požadavků



Obrázek 2: Model nefunkčních požadavků

Z obrázků výše je patrné, že byl zvolen velice jednoduchý model těchto požadavků. Jak uvádí Arlow, dodatečné informace o požadavku lze zachytit pomocí atributů v něm obsažených. Nejčastěji se vyskytuje atribut priorita vyjadřující jeho prioritu vůči ostatním požadavkům. Vždy je však důležité vytvářet co nejjednodušší atributy přínosné pro projekt. [21, s. 81–82] V tomto případě by dodatečné atributy nepřinášely žádný další užitek, a to ani v případě atributu priority, jelikož veškeré uvedené požadavky jsou z hlediska fungování konečné aplikace nezbytné.

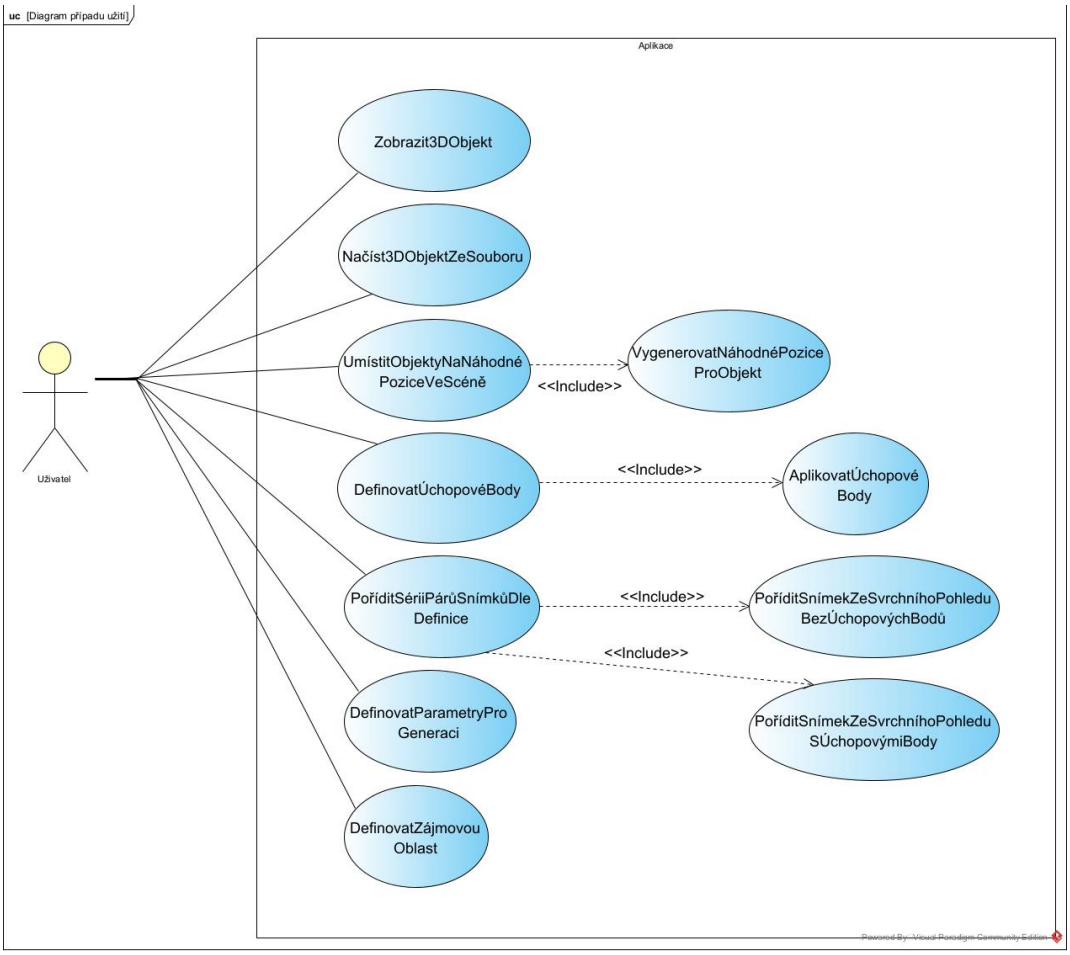
4.2 Vytvoření modelu případu užití

Po vytvoření modelu požadavků byl vytvořen diagram případu užití. Před jeho uvedením je však vhodné přiblížit z čeho se takový diagram obecně skládá a v čem jeho tvorba spočívá.

Nejprve je nutné stanovit hranice systému, nebo-li subjekt, a co je či není jeho součástí. Subjekt je definován aktéry a případy užití, které vyjadřují přínos této skupině. Aktér není součástí systému a vyjadřuje roli externí entity, jako je uživatel či další systém, při použití systému. [21, s. 92–93] Výsledná aplikace nevyžaduje nutně komunikaci s externími systémy, ani správu uživatelských rolí. Z tohoto důvodu se dále uvažuje pouze jediný aktér, kterým je samotný uživatel.

Po vymezení těchto základních prvků bylo dále potřeba definovat jednotlivé případy užití. Arlow uvádí, že případy užití se zpravidla nachází uvnitř hranic systému a vyjadřují jakým způsobem jej bude daný aktér využívat. Jejich název musí být složen ze slovesné vazby. [21, s. 95]

Vyjma základních vazeb mezi aktéry a případy užití byly v diagramu použity také relace «include» pro bližší modelování jednotlivých případů užití. Dle Arlowa lze s relací «inlcude» v rámci scénáře klientského případu užití zahrnout také chování z dodavatelského, které poté tvoří jeho důležitou součást. [21, s. 121–122] Tímto způsobem byly rozšířeny případy užití týkající se náhodného rozmístění objektů, definice úchopových bodů a pořízení párů snímků dle definice. Výsledný diagram je znázorněn na obrázku 3.



Obrázek 3: Model případu užití

5 PŘÍPRAVA PŘED IMPLEMENTACÍ APLIKACE

Před samotnou implementací aplikace bylo potřeba přesněji určit z jakých částí bude složena a jak bude vypadat její grafická podoba. Návrhem se v následujících částech rozumí pouze velice základní pojednání o základních odpovědnostech aplikace. Tímto pojmem tak není myšlen proces vývoje softwaru a není tu tak popsán proces tvorby příslušných diagramů v jazyce UML.

5.1 Návrh rozdělení aplikace

Jak již bylo nastíněno v kapitole 3, pro vytvoření grafického rozhraní aplikace byl zvolen herní engine Unity 2022.3.0f1. Ten je však potřeba spíše pro samotné vykreslení objektů a scén, zatímco samotné načítání modelů a generace náhodných pozic může být realizováno zcela bez pomoci tohoto enginu. Z tohoto důvodu byly tyto části rozděleny do backendu a frontendu.

Na straně backendu je v tomto rozdělení řešeno načítání 3D objektů ze standardních 3D formátů a náhodná generace pozic dle zadaných parametrů. Frontend se naopak stará o přímou interakci s uživatelem skrze grafické rozhraní a samotné vykreslování objektů a scén. Toto rozložení tudíž umožňuje snazší aktualizaci jedné z těchto komponent či její plné nahrazení jinou, a tak výsledný produkt činí časově odolnější.

Dále bylo důležité určit alespoň obecný způsob, kterým se budou předávat potřebná data o objektu mezi těmito částmi. Jak uvádí Gortler, geometrii tvoří kolekce trojúhelníků, které tvoří 3 vrcholy. Pro každý vrchol musí být minimálně specifikována jeho pozice pomocí dvou či tří čísel v závislosti na typu geometrie. Dále mohou být specifikovány další atributy skrze jiná číselná data [22, s. 4]. Je tudíž zřejmé, že potřebná data o objektu lze přenášet na úrovni vrcholů v číselné formě nezávisle na použitých technologiích.

Zvoleným rozdělením bylo možné od sebe lépe oddělit jednotlivé oblasti funkcí výsledné aplikace ještě před započetím samotné implementace. Další výhodou zvoleného přístupu je snazší možnost případné výměny dané komponenty, což může pomoci zvýšit časovou odolnost.

5.2 Návrh GUI

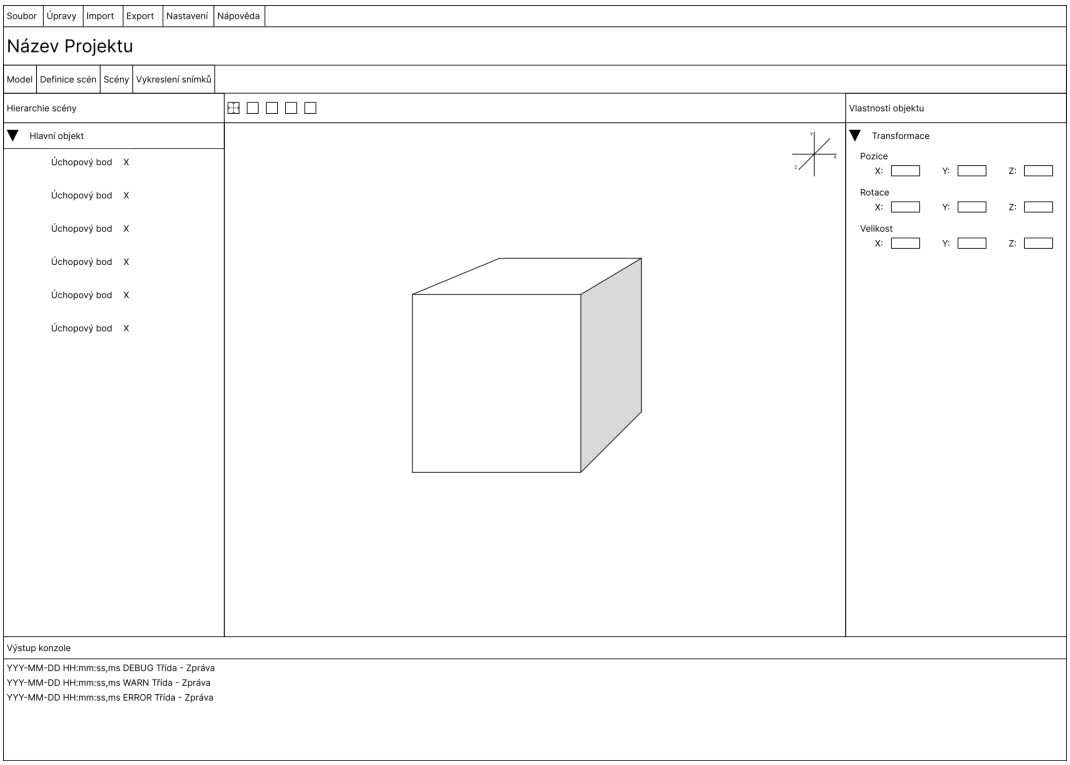
Smyslem následujících návrhů uživatelského rozhraní aplikace bylo vytvořit základ, na který se bude možné během vývoje odkazovat a předem získat jisté povědomí o vhodnosti zamýšleného rozhraní. Nebylo však zamýšleno jako šablona, kterou je nutno plně dodržet.

Pro vytvoření těchto návrhů byl použit nástroj Figma. Jak uvádí Bracey, „Figma je aplikace pro návrh rozhraní, která běží v prohlížeči... Figma umožňuje vybudovat knihovny znovupoužitelných komponent, ke kterým má přístup celý tým.“ [23] (překlad autora) Využití tohoto nástroje tak tento proces značně usnadnilo.

Nejprve byl navržen společný základ rozhraní, tedy prvky vyskytující se ve všech situacích (obr. 4). Těmi jsou nástrojová lišta, panel zobrazující název projektu a umožňující přepínání aktivní obrazovky v horní části okna. Ve spodní části se pak nachází konzole pro informování uživatele o prováděných operacích.

Obrázek 4: Návrh základu rozhraní

Poté byla navržena podoba obrazovky pro zobrazení načteného modelu. Jak je na obrázku 5 vidět, počítá se zde například s možností výběru jednotlivých úchopových bodů a jejich dodatečných úprav v podobě změny pozice a rotace.

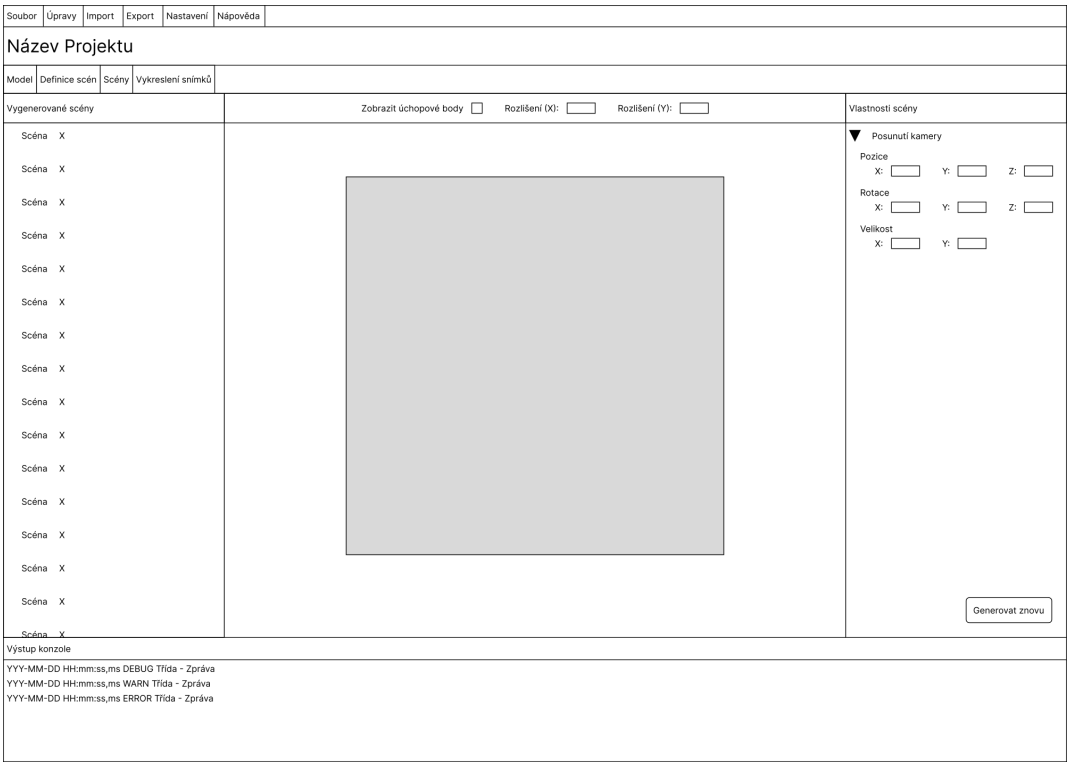


Obrázek 5: Návrh obrazovky modelu

Před samotným vykreslením scén je bude nutné vygenerovat dle parametrů zadaných uživatelem. Z tohoto důvodu byl také vytvořen návrh obrazovky umožňující provést zmíněnou generaci a nastavit jí potřebné parametry (obr. 6).

Nakonec byla navržena obrazovka umožňující procházet jednotlivé vygenerované scény a vykreslit je do série obrázků (obr. 7). Dále se zde počítá s možností skrýt úchopové body při zobrazení náhledu výsledné scény.

Obrázek 6: Návrh obrazovky definice scén



Obrázek 7: Návrh obrazovky scén

Vytvoření výše popsaných návrhů grafického rozhraní pomohlo přiblížit možný způsob implementace požadavků aplikace tak, aby bylo její používání uživatelsky přívětivé. Dále tento krok vytvořil jakýsi obecný cíl koncového vzhledu aplikace, podle kterého se poté bylo možné řídit při samotné implementaci namísto náhodného pokládání prvků.

6 IMPLEMENTACE APLIKACE

Jak již bylo nastíněno v kapitole 5.1, pro implementaci výsledné aplikace byla zvolena strategie oddělené logiky a GUI. Záměrem tohoto rozhodnutí je dosáhnout větší nezávislosti mezi komponentami starajícími se o parsování modelů z 3D formátů či výpočty při generaci pozic pro objekty a jejich úchopových bodů a komponentami zajišťujícími samotné vykreslení scény. Zvolená strategie též eliminovala nutnost použití stejného jazyka pro všechny části aplikace.

6.1 Implementace backendu aplikace

Vzhledem k rozdělení aplikace popsanému v předešlé kapitole nebylo nutné backend aplikace stavět na stejných technologiích jako její grafickou část. Zároveň však bylo potřeba vybrat takovou formu komunikace s okolním světem, aby bylo dostatečně snadné tyto dvě části později propojit.

Z tohoto důvodu byly prozkoumány různé technologie, které umožňují implementovat potřebné funkce a zároveň poskytují formy implementace API, které bude možné provolat z grafické části aplikace. Nakonec byl zvolen přístup využívající vícero programovacích jazyků. Konkrétní technologie použité pro jeho realizaci jsou popsány v následujících podkapitolách.

6.1.1 Implementace API backendu

K implementaci části backendu aplikace spravující API byl po prozkoumání dostupných možností zvolen jazyk Java s frameworkem Spring Boot 3.1.0 a pro správu vytvořeného projektu bylo použito IDE IntelliJ IDEA 2022.3 (Ultimate Edition). Tato volba byla provedena na základě předešlých zkušeností autora s těmito technologiemi.

Projekt byl vytvořen pro SKD Oracle OpenJDK 20.0.1 s úrovní jazyka Java 17. Vytvořený projekt byl poté rozdělen do balíčků na základě funkcí (viz tabulka 1). Jak uvádí Hickey, hlavní zaměření by nemělo být na to, jak systém vykonává svou činnost, ale co umí. V projektu organizovaném dle funkcí je snadné najít kód dané funkce a jejich složitost je od zbytku kódu izolována. [24]

Název složky	Oblast
api	Funkce související s API
common	Funkce společné pro všechny části projektu
models	Funkce související s modely
queueing	Funkce související s frontami zpráv
scenes	Funkce související se scénami
sockets	Funkce související se Socket.IO

Tabulka 1: Základní složky projektu ve Spring Boot a jejich význam

S tímto rozdělením byly nejprve implementovány mechanismy pro navázání obousměrné komunikace s klientem pomocí knihovny implementace Socket.IO pro jazyk Java. Jak uvádí dokumentace této knihovny, „Socket.IO je knihovna, která umožňuje nízkolatenční, obousměrnou a událostmi řízenou komunikaci mezi klientem a serverem. Je postavena na protokolu WebSocket a poskytuje další záruky jako záložní návrat k HTTP long-polling či automatické obnovení spojení.“ [25] (překlad autora)

Pro použití této formy komunikace ve Spring Boot byla použita knihovna Netty Socket.IO. Jak je uvedeno v jejím repozitáři, „Tento projekt je otevřenou Java implementací Socket.IO serveru. Je založen na serverovém frameworku Netty.“ [26] (překlad autora)

Aby bylo možné možnosti této knihovny využívat, bylo potřeba vytvořit příslušné třídy. Pro nastavení serveru byla vytvořena třída SocketIOConfig (viz zdrojový kód 1). Poté byla pro spuštění tohoto serveru vytvořena třída SocketIOServerCommandLineRunner (viz zdrojový kód 2).

```
@Configuration
public class SocketIOConfig {
    @Value("${socket-server.host}")
    private String host;

    @Value("${socket-server.port}")
    private Integer port;
    @Bean
    public SocketIOServer socketIOServer() {
        com.corundumstudio.socketio.Configuration config = new
            com.corundumstudio.socketio.Configuration();
        config.setHostname(host);
        config.setPort(port);
        return new SocketIOServer(config);
    }
}
```

Zdrojový kód 1: Třída SocketIOConfig. Kód převzat z: [27]

```
@Component
public class SocketIOServerCommandLineRunner implements CommandLineRunner {
    private final SocketIOServer server;

    public SocketIOServerCommandLineRunner(SocketIOServer server) {
        this.server = server;
    }

    @Override
    public void run(String... args) throws Exception {
        server.start();
    }
}
```

Zdrojový kód 2: Třída SocketIOServerCommandLineRunner. Kód vytvořen podle: [27]

Jakmile byla implementována základní forma tohoto způsobu komunikace, byl přidán jednoduchý mechanismus pro správu připojených klientů a jejich operací. Pro tyto účely byla kvůli jednoduchosti použita třída ConcurrentHashMap pro uchování těchto hodnot na základě klíče namísto jejich ukládání do databáze či do datové struktury HashMap.

Jak je uvedeno na stránce Baeldung, HashMap není, narozdíl od ConcurrentHashMap, implementací se zabezpečenými vlákny. Při použití ConcurrentHashMap je při použití více vláken zajištěna paměťová konzistence u operací s klíči a hodnotami [28]. Využitím ConcurrentHashMap tak bylo možné snadno zpracovávat jednotlivé klienty a operace na základě jejich id, která do ní byla vložena bez porušení konzistence v případě práce s vlákny.

Výše popsaný mechanismus byl poté použit jak pro upozornění konkrétního klienta o dokončení konkrétní operace, tak pro posílání informativních zpráv. Samotné předávání dat však touto formou řešeno nebylo kvůli limitacím velikosti zasílaných zpráv. Namísto toho byly vytvořeny endpointy REST API.

Celkem byly vytvořeny 4 endpointy. Dva slouží pro vyvolání akce a dva pro získání jejího výsledku. Pro provolání těchto endpointů je vyžadováno, aby byl klient připojený k Socket.IO serveru skrze mechanismus popsaný výše a získané ID předával v hlavičce daného požadavku.

Princip spočívá ve vyslání požadavku na endpoint metodou POST s příslušnými daty, které jsou zde poté zpracovány a na jejich základě je vytvořen záznam operace pro daného klienta. Jakmile je operace hotová, uloží se její výsledek do dočasného souboru, jehož cesta je uložena do pole objektu s informacemi o operaci. Poté je klient pomocí socketu informován o dokončení operace společně s url, ze které si může pomocí metody GET výsledek operace stáhnout.

Vzhledem ke skutečnosti, že veškeré vyvolávané požadavky vyžadují, aby byl klient napřed připojen přes Socket.IO, byl přidán jednoduchý bezpečnostní mechanismus. Tento mechanismus způsobí, že požadavek bude při absenci, či neplatnosti příslušného ID v jeho hlavičce odmítnut.

Zmíněný mechanismus využívá SecurityFilterChain z frameworku Spring Security 6.1.2 (viz zdrojový kód 3). Pro tuto třídu byl vytvořen vlastní filtr, který z hlavičky vyextrahuje

ID klienta a zhodnotí zda je tato hodnota validní (viz zdrojový kód 4). Při následném obsluhování požadavku se hodnota ID získává z hlavičky stejným způsobem.

```
@Configuration
@EnableWebSecurity
public class ApiSecurityConfiguration {

    private final ActiveClientsService activeClientsService;

    public ApiSecurityConfiguration(ActiveClientsService activeClientsService) {
        this.activeClientsService = activeClientsService;
    }

    @Bean
    public FilterRegistrationBean<UserFilter> userFilterFilterRegistration(UserFilter
        filter) {
        FilterRegistrationBean<UserFilter> registration = new
            FilterRegistrationBean<>(filter);
        registration.setEnabled(false);
        return registration;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
        {
        http.csrf(AbstractHttpConfigurer::disable)
                .authorizeHttpRequests(
                        authorize ->
                            authorize.requestMatchers("/api/v1/hello**").permitAll()
                                .requestMatchers("/api/v1/**").permitAll()
                                .anyRequest().permitAll()
        ).addFilterBefore(new UserFilter(activeClientsService),
            AuthorizationFilter.class);

        return http.build();
    }
}
```

Zdrojový kód 3: Třída ApiSecurityConfiguration. Kód vytvořen podle: [29], [30]

```
@Component
public class UserFilter extends OncePerRequestFilter {
    private static final Logger logger = LoggerFactory.getLogger(UserFilter.class);

    private final ActiveClientsService activeClientsService;

    public UserFilter(ActiveClientsService activeClientsService) {
        this.activeClientsService = activeClientsService;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
        response, FilterChain filterChain)
            throws ServletException, IOException {
        try {
            String userID = request.getHeader("client-id");
            if (userID == null) {
                throw new AccessDeniedException("No client id present");
            }
            UUID uuid = UUID.fromString(userID);

            if (!activeClientsService.isClientRegistered(uuid)) {
                throw new AccessDeniedException("No such client");
            }

            logger.info(userID);
            filterChain.doFilter(request, response);
        } catch (AccessDeniedException | RuntimeException ex) {
            response.sendError(HttpServletResponse.SC_UNAUTHORIZED, ex.getMessage());
        }
    }
}
```

Zdrojový kód 4: Třída UserFilter. Kód vytvořen podle: [29], [31]

Jak je na předešlých zdrojových kódech zjevné, v nastavení bezpečnosti byla vypnuta ochrana proti CSRF. Tento krok byl proveden pro zjednodušení lokálního vývoje a před případným nasazením na server je potřeba toto nastavení náležitě upravit.

Jelikož bylo v případě operace parsování modelů potřeba pracovat se soubory, byl tento požadavek nastaven jako Multipart. Jak uvádí Khan, při použití Multipart požadavků dochází k posílání dat různých typů v rámci jednoho HTTP volání a lze v nich posílat také multipart soubory, kterými mohou být například obrázky či zvukové záznamy. [32]

V současné implementaci nebyla z časových důvodů přidána kontrola pro datový typ posílaných souborů a pro daný soubor je pouze zjištěna jeho koncovka pomocí knihovny Apache Commons IO. Případný nepodporovaný formát je tudíž odhalen až při samotném zpracování požadované operace. Při implementaci budoucích rozšíření je tak doporučeno tuto kontrolu implementovat například stanovením jeho typu na základě obsahu.

Jelikož jsou výsledky operací ukládány do dočasných souborů, je jejich obsah při vyžádání nahrán do proudu (angl. stream). Poté jsou klientovi zpřístupňovány pomocí třídy InputStreamResource. Jak je uvedeno v dokumentaci Springu, třída InputStreamResource pro daný InputStream implementuje rozhraní Resource [33]. Díky této skutečnosti tak bylo možné jednoduše takto otevřený proud předat instanci této třídy a umožnit klientovi obsah souboru stáhnout.

Zvolený přístup umožňuje obejít limitace spojené s velikostí výsledků, zároveň je však těmito kroky celý proces zpomalen. V případě budoucích rozšíření by tak mohl být tento mechanismus optimalizován kupříkladu přímým využitím spojení přes Socket.IO v případě dat o dostatečně malé velikosti.

Pro jednotnou správu případných výjimek, které nastanou při volání některého z endpointů, byl vytvořen základní ControllerAdvice, který dané výjimky zachytí a vrátí klientovi odpověď s příslušným HTTP kódem. Jak uvádí Paraschiv, s anotací @ControllerAdvice lze ovládat návratový kód a tělo odpovědi a zároveň je možné na jednu metodu namapovat vícero výjimek. Tímto způsobem tak lze spravovat výjimky v jedné globální komponentě namísto v mnoha @ExceptionHandlers. [34]

Provedením těchto kroků byla vytvořena vrstva aplikace schopná komunikovat s grafickou částí aplikace a navázat s ní trvalé spojení pomocí Socket.IO či reagovat na dané požadavky skrze definované endpointy. Tato vrstva je zároveň schopna delegovat tyto požadavky na ostatní služby a získané výsledky následně poskytnout klientovi. Možná budoucí rozšíření této vrstvy zahrnují kontrolu typu souborů v požadavcích, optimalizaci

prováděných operací dle velikosti dat či vylepšení z hlediska nastavení bezpečnosti, aby bylo vhodné backend aplikace nasadit na server.

6.1.2 Implementace parsování modelů a generace scén

Jak již bylo řečeno výše, operace parsování modelů ze souborů a generace scén dle vstupních parametrů byly implementovány v odlišné technologii od samotného API, které se stará o komunikaci s grafickou částí aplikace. Pro implementace této části byl zvolen jazyk Python 3.11 a vývojové prostředí PyCharm 2022.3.1 (Community Edition), též na základě jistých předešlých zkušeností autora s tímto jazykem. Hlavním důvodem však byla existence knihoven podporujících potřebné operace.

Při tvorbě projektu v Pythonu byl stále dodržován princip rozdělení dle funkcí (viz kap. 6.1.1). Bližší forma organizace projektu však byla přizpůsobena odlišným konvencím, které jsou v rámci tohoto jazyka aplikovány, což mělo za následek méně vytvořených souborů.

Hlavní zdrojové kódy tak byly umístěny do balíčku src, ve kterém byly dále rozděleny do balíčků dle jednotlivých funkcí (viz tabulka 2). Pro snadné oddělení testů od implementace pak byl na stejné úrovni vytvořen balíček test, který kopíruje strukturu prvního zmíněného balíčku. Tyto testy byly vytvořeny s pomocí standardního balíčku unittest a nebylo tak pro ně potřeba instalovat další závislosti.

Název složky	Oblast
models	Funkce související s modely
queueing	Funkce související s frontami zpráv
scenes	Funkce související se scénami

Tabulka 2: Základní složky projektu v Pythonu a jejich význam

Jakmile byla stanovena struktura tohoto projektu, byla nejprve řešena otázka parsování 3D modelů z různorodých formátů. Nakonec byla k tomuto účelu použita knihovna Assimp, respektive knihovna assimp-py 1.0.7, která ji umožňuje použít v jazyce Python. Jak uvádí dokumentace knihovny Assimp, tato knihovna umožňuje načíst geometrické scény z různých 3D formátů. Mimo jiné jsou podporovány například formáty .obj, .dxf, .stl, .ply

a mnoho dalších. Je zde tak podpora také některých CAD formátů [35]. Jejím použitím tak byl splněn požadavek na podporu načítání souborů ze standardních 3D formátů.

Pro snazší přenos do dalších částí aplikace byly vytvořeny datové třídy, do kterých byly získané informace o modelu a jeho jednotlivých vrcholech a ploškách vkládány. Takto získané třídy poté byly převedeny do slovníků a následně serializovány do řetězce JSON.

Jakmile bylo možné parsovat 3D modely ze široké škály standardně používaných formátů, byla řešena problematika generace scén a tím pádem také algoritmů pro generaci náhodných hodnot. V rámci této práce byly z časových důvodů implementovány pouze dva přístupy. Těmito přístupy jsou náhodná generace s kontrolou kolizí a generace vzorkováním z Poissonova disku.

Pro generaci náhodných pozic prvním popsaným způsobem byla použita knihovna NumPy. Jak je uvedeno v její dokumentaci, jedná se o knihovnu poskytující multidimenzionální pole, nástroje pro operace nad nimi či základní statistické operace a v Pythonu tak tvoří základní balíček pro vědecké výpočty. [36] Generace hodnot tímto způsobem využívá rovnoměrné rozdělení pravděpodobnosti. V případě budoucích rozšíření by však mohly být přidány další rozdělení pravděpodobnosti využívající tuto knihovnu a s tím také příslušné parametry umožňující s nimi pracovat.

Jelikož je tato forma generace zcela náhodná, je velice pravděpodobné, že se jednotlivé modely budou na vygenerovaných pozicích překrývat. Aby se tomuto jevu co nejvíce zabránilo, byla přidána kontrola kolizí, která má za cíl takové situaci předcházet.

Vzhledem ke skutečnosti, že vytvoření vlastní kontroly kolizí není předmětem této práce, byla k tomuto účelu použita knihovna trimesh. Jak je uvedeno v dokumentaci, knihovna trimesh se snaží pro snadnou manipulaci a analýzu poskytnout objekt Trimesh. Knihovna umí načítat a pracovat s trojúhelníkovými sítěmi [37]. S využitím této knihovny bylo snadné kontrolovat jednotlivé kolize při generování každé jednotlivé pozice ve 3D prostoru.

Zároveň však tento zcela náhodný způsob se striktní kontrolou kolizí značně zpomaloval generaci scén a mohl vést až k nekonečné smyčce. Z tohoto důvodu byla pravidla pro kolize značně uvolněna přidáním hodnoty maximálního počtu pokusů před opuštěním generace. Tato hodnota byla nastavena pevně na hodnotu 100. Parametr počtu objektů tak s touto

úpravou neoznačuje přesný počet objektů, které se budou ve scéně nacházet, ale jejich maximální počet.

Jak již bylo uvedeno výše, druhý způsob generace využívá vzorkování z Poissonova disku. Jak uvádí Davies, vzorkováním z Poissonova disku vzniká přirozenější vzor, jelikož vytváří úzce vtěsnané body, které však u sebe nejsou blíže než v minimální specifikované vzdálenosti [38]. Pro tento typ generace byla využita knihovna poisson-disc 0.2.1, se kterou ji bylo možné aplikovat na libovolné rozměry zájmové oblasti.

Generace rotací probíhá vždy zcela náhodně, ale zároveň respektuje původní nastavenou rotaci z parametrů vyjádřenou v kvaternionech. Pro získání této hodnoty ze vstupních dat byla opět využita knihovna NumPy. V obou případech byly jednotlivé pozice, rotace a velikost aplikovány pomocí afinních transformací. Jak uvádí dokumentace BrainVoyager, jedinou afinní transformační maticí o rozměrech 4x4 lze vyjádřit libovolnou kombinaci posunu, rotace, změny velikosti či zkosení [39]. Pro usnadnění práce s těmito maticemi byla použita knihovna Pyrr.

Po generaci náhodných transformací pomocí výše popsaných způsobů byl také implementován způsob aplikace těchto transformací na definované úchopové body. Princip spočíval stejně jako v případě aplikace transformací na samotné objekty v násobení matic o rozměrech 4x4. Vždy tak byly aplikovány jak transformace původního úchopového bodu, tak transformace původního modelu, ke kterému daný bod patří.

Výsledná implementace však bohužel nedosahovala správných výsledků z hlediska rotací úchopových bodů při jejich načtení v grafické části aplikace, a tak zde byly tyto výsledky ignorovány. Transformace úchopových bodů jsou tak na její straně přepočítány znovu na základě stejného principu, avšak s využitím struktur v enginu Unity.

K výše popsaným způsobům generace scén byla přidána také podpora simulace. Použití simulace lze zvolit vstupními parametry a aplikuje se až po vygenerování základních transformací některou z metod popsaných výše. Jako fyzikální engine byla použita knihovna PyBullet (viz zdrojový kód 5). Jak uvádí Chand, „Pybullet je pythonovský modul pro fyzikální simulaci a robotiku založený na Bullet Physics SDK.“ [40] (překlad autora)

```
def simulate_physics(verts: list, model_coords: list[SceneModelsCoords], steps: int,
    time_step: float):
    p.connect(p.DIRECT)
```

```
p.setAdditionalSearchPath(pybullet_data.getDataPath())
p.setGravity(0, 0, -10)
p.setTimeStep(time_step)
p.setRealTimeSimulation(0)
p.loadURDF("plane.urdf")
box_ids = []

for model_coord in model_coords:
    startPos = [model_coord.position.x, model_coord.position.z,
        model_coord.position.y]
    startOrientation = [model_coord.rotation.x, model_coord.rotation.z,
        model_coord.rotation.y, model_coord.rotation.w]

    col_id = p.createCollisionShape(p.GEOM_MESH, vertices=verts,
        meshScale=[model_coord.scale.x, model_coord.scale.z, model_coord.scale.y])

    box_ids.append(p.createMultiBody(baseMass=1, baseCollisionShapeIndex=col_id,
        basePosition=startPos, baseOrientation=startOrientation))

for i in range(steps):
    p.stepSimulation()
...
p.disconnect()
```

Zdrojový kód 5: Simulace pomocí PyBullet. Kód vytvořen podle: [40]

Nakonec byl přidán soubor .env umožňující parametrizovat některé aspekty při spuštění. Těmito aspekty jsou například údaje pro připojení k brokeru zpráv, počet procesů pro daný typ operace, či názvy daných front. Pro práci s tímto souborem byl použit balíček python-dotenv.

Provedením výše popsaných kroků byla do jádra aplikace přidána podpora pro parsování modelů ze standardních 3D formátů a jejich konverze do jednotného formátu pro pozdější použití v grafické části aplikace. Zároveň byla přidána podpora pro generaci scén dle zadaných parametrů s možností výběru mezi dvěma způsoby generace náhodných pozic. Možná budoucí rozšíření této funkcionality zahrnují například obohacení nabídky těchto algoritmů a jejich příslušných parametrů.

6.1.3 Implementace komunikace mezi službami

V přípravné fázi tvorby této práce bylo v plánu implementovat celý backend v jazyce Python. Jak je patrné z předešlých podkapitol, tato myšlenka byla nakonec kvůli problémům při zpracování dlouho běžících operací opuštěna ve prospěch dalšího rozdělení této části do samostatných částí napsaných v jiných jazycích. Backend byl tudíž postaven na architektuře mikroslužeb.

Jak uvádí Vinka, v architektuře mikroslužeb jsou funkce odděleny do samostatných komponent a tyto funkce tak lze upravovat bez ovlivnění zbytku architektury. Výsledná aplikace se pak skládá ze samostatně vyvíjených mikroslužeb [41]. Během vývoje bylo skutečně možné jednotlivé části vyvíjet samostatně. Nejprve však bylo potřeba zvolit způsob komunikace mezi jednotlivými mikroslužbami.

Dle Vinky jsou mikroslužby schopné spolu komunikovat i navzdory jejich oddělení a je typické, že služba pro své operace potřebuje pomoc ostatních. K propojení mikroslužeb pak lze využít například broker, REST API či vzdálené volání procedur (RPC) [41]. V rámci této práce byl zvolen první jmenovaný způsob.

Před představením samotné implementace je vhodné více přiblížit princip tohoto způsobu komunikace. Vinka uvádí, že díky frontě zpráv mohou části aplikace poslat asynchronně zprávy do fronty. Mezi jednotlivými mikroslužbami poté message broker působí jako prostředník, který obdrží zprávy od producentů a předá je konzumentům. [41]

Pro implementaci komunikace prostřednictvím message brokerů bylo potřeba jej vhodně zvolit. Nakonec byl k tomuto účelu zvolen RabbitMQ 3.8. Jak Vinka uvádí, díky asynchronnímu zpracování je s RabbitMQ možné do fronty umístit zprávu bez okamžitého zpracování, a tudíž je ideální pro zachování rychlé odezvy při zpracování blokujících či dlouho běžících úloh. [41]

RabbitMQ podporuje mimo jiné protokol pro posílání zpráv AMQP 0-9-1. V rámci tohoto protokolu je ditstribuce zpráv do front prováděna přes exchange na základě pravidel. Broker dopředu definuje výchozí direct exchange, která nemá jméno a směrovací klíče všech jejích front jsou stejné jako jejich jména. Ačkoliv tedy zprávy nejsou nikdy posílány přímo do front, při použití direct exchange to vypadá jako kdyby ano [42]. V rámci této práce tak byla pro jednoduchost využita direct exchange.

Před samotnou implementací bylo potřeba určit, jaké role budou jednotlivé služby zaujímat. Jak bylo popsáno již v kapitole 6.1.1, služba napsaná ve frameworku Spring Boot slouží pro přímou komunikaci s klientem. Jednotlivé úlohy tak potřebuje delegovat na službu napsanou v Pythonu a zároveň od něho potřebuje získat výsledek požadované úlohy. Obě služby mají produkovat a zároveň konzumovat zprávy z fronty zpráv. Počátek komunikace mezi těmito službami však vždy začíná na straně první výše zmíněné služby.

Nakonec byl zvolen způsob komunikace s pomocí vícero front. Konkrétně byly specifikovány dva páry front dle daného typu operace. V rámci každého páru tak byla specifikována fronta pro vstupní požadavky a výsledky daných operací.

Pro implementaci komunikace s RabbitMQ na straně Springu byla použita knihovna Spring AMQP a jednotlivé třídy a balíčky související s touto komunikací byly umístěny do balíčku queueing. Nejprve zde byla vytvořena třída pro konfiguraci (viz zdrojový kód 6). Poté byly vytvořeny třídy, které řeší přijímání a odesílání zpráv do specifikovaných front.

```
@Configuration
public class RabbitMQConfig {
    @Value("${queues.exchange}")
    private String topicExchangeName;
    @Value("${queues.model-parsing.request}")
    private String modelParsingRequestQueueName;
    ...
    @Bean
    Queue modelParsingRequestQueue() {
        return new Queue(modelParsingRequestQueueName, false);
    }
    ...
    @Bean
    TopicExchange exchange() {
        return new TopicExchange(topicExchangeName);
    }

    @Bean
    Binding binding(@Qualifier("modelParsingRequestQueue") Queue queue, TopicExchange
        exchange) {
        return BindingBuilder.bind(queue).to(exchange).with(queue.getName());
    }
```

```
    ...
}
```

Zdrojový kód 6: Třída RabbitMQConfig. Kód vytvořen podle: [43]

Pro komunikaci na straně Pythonu byla použita knihovna Pika. Jak je uvedeno v dokumentaci této knihovny, „Pika je čistá Pythonovská implementace protokolu AMQP 0-9-1, která se snaží zůstat vcelku nezávislá na podkladové knihovně pro síťovou podporu.“ [44] (překlad autora)

Účelem této služby bylo sloužit jako tzv. worker a vykonávat jednotlivé úlohy dle parametrů ve zprávách přicházejících z příslušné pracovní fronty (angl. Work Queue). Jak je uvedeno v tutoriálu RabbitMQ, hlavní myšlenkou pracovních front je odklad vykonávání výpočetně náročné operace jejím zapouzdřením do zprávy, která se pošle do fronty. Worker procesy poté tyto úlohy z fronty odebírají a vykonávají dle nich danou činnost. Jelikož jsou takto úlohy mezi jednotlivými workery sdíleny, je snadné práci paralelizovat a škálovat přidáním dalších workerů. [45]

Tato část však řešila pouze způsob, jakým v této části vyvolat provedení dané operace dle parametrů. Jelikož bylo potřeba z prováděných operací poslat výsledná data zpět na stranu služby ve Spring Boot, výsledky operací byly opět zapouzdřeny do zpráv a poslány do příslušných front dle daného typu úlohy.

Nakonec zde byly vytvořeny dva druhy workerů vykonávající požadované úlohy dle typů operací. Zároveň byl vytvořen hlavní spouštěcí soubor artificial_generation.py, který spustí počet workerů daného typu dle hodnoty v konfiguračním souboru a všechny spustí jako samostatný proces. Implementace však byla nastavena tak, aby počet vytvořených procesů nepřekročil počet jader procesoru a zároveň aby bylo možné nastavit minimální počet workerů pro parsování modelů.

Vytvořená forma komunikace nabízí široké množství možností pro budoucí rozšíření. Jednou z těchto možností je kupříkladu připojení další mikroslužby spravující existující neuronovou síť, kterou by tak bylo možné trénovat výstupy získanými z grafické části aplikace a která by poté mohla sama provádět návrhy rozmístění úchopových bodů. Další možné rozšíření pak může spočívat v připojení služby spravující renderovací farmu, čímž by bylo

možné snížit výpočetní vytížení na straně klienta při vykreslování výsledných párů snímků z vygenerovaných scén.

Kroky popsanými výše byla implementována vnitřní komunikace mezi službami backendu. Díky použití message brokera RabbitMQ bylo tuto komunikaci možné implementovat nezávisle na konkrétním jazyce použitém pro jednotlivé služby a zároveň snadno zpracovávat časově náročné úlohy. Výsledné řešení lze v budoucnu rozšířit připojením dalších mikroslužeb, které by mohly spravovat například konkrétní neuronovou síť pro trénování či renderovací farmu umožňující vykreslovat vygenerované scény.

6.1.4 Dockerizace backendu

Jelikož byla pro implementaci interní komunikace použita fronta zpráv, bylo by potřeba ji před spuštěním backendu nainstalovat do operačního systému a poupravit konfiguraci ostatních komponent. Tato skutečnost nemusí nutně představovat problém v případě nasazení na server, ale pro lokální použití již ano. Z tohoto důvodu byly podniknuty kroky pro dockerizaci backendu aplikace.

Před popisem provedené dockerizace je vhodné přiblížit, co je Docker a z jakých hlavních částí se skládá. Jak je uvedeno na stránkách IBM, „Docker je open source platforma, která umožňuje vývojářům sestavit, nasadit, spustit, aktualizovat a spravovat kontejnery – standardizované, spustitelné komponenty, které kombinují zdrojové kódy aplikací s knihovnami operačního systému (OS) a závislostmi potřebnými pro spuštění tohoto kódu v libovolném prostředí.“ [46] (překlad autora)

V obrazu Dockeru (angl. Docker images) jsou umístěny všechny potřebné knihovny, nástroje a zdrojové kódy. Dockerový kontejner poté představuje interaktivní běžící instanci obrazu Dockeru. Vytvoření obrazu Dockeru lze automatizovat souborem Dockerfile. Pro aplikace skládající se z vícero kontejnerů lze využít Docker Compose, který jednotlivé služby aplikace specifikuje v souboru typu YAML. Tento soubor lze využít nezávisle na jazyce daných programů a veškeré kontejnery aplikace tak lze s pomocí Docker Compose spustit jediným příkazem. [46]

Pro potřeby backendu tak byl vytvořen soubor YAML pro využití funkcí Docker Compose. V případě RabbitMQ byl použit existující obraz Dockeru, který byl přímo v tomto souboru definován. Podoba této části souboru je vyobrazena ve zdrojovém kódu 7

```
version: '3.8'
    services:
        ...
        rabbitmq3:
        container_name: "artificial-generation-rabbitmq"
        image: rabbitmq:3.8-management-alpine
        environment:
        - RABBITMQ_DEFAULT_USER=myuser
        - RABBITMQ_DEFAULT_PASS=mypassword
        ports:
        # AMQP protocol port
        - '5672:5672'
        # HTTP management UI
        - '15672:15672'
```

Zdrojový kód 7: Část YAML souboru definující RabbitMQ. Kód vytvořen podle: [47]

V případě vlastních vytvořených služeb bylo již potřeba vytvořit vlastní Dockerový obraz. Pro obě části backendu aplikace tak byly vytvořeny soubory Dockerfile. Vytvořený Dockerfile pro část aplikace vytvořenou ve Springu je vyobrazen ve zdrojovém kódu 8.

Z důvodu vývoje v operačním systému Windows bylo potřeba do tohoto obrazu nainstalovat také balíček dos2unix pro konverzi konce řádků v souboru mvnw. Jak uvádí Anderson, různý způsob řešení konce řádků na systému Windows a systémech jako Linux či macOS může představovat problém. Pokud se v Linuxovém systému někdo pokusí spustit soubor vytvořený v systému Windows, nebude znak \r chápán jako konec řádku. Toto lze vyřešit instalací nástroje dos2unix, který konce řádků převede [48]. Bohužel i s instalací tohoto programu docházelo k problémům s konci řádků a soubor mvnw je tak stále nutné ručně převést do formátu konce řádků používaném na linuxových systémech.

```
# syntax=docker/dockerfile:1

FROM eclipse-temurin:17-jdk-jammy as base
```

```
WORKDIR /app

RUN apt-get update && apt-get install -y dos2unix

COPY .mvn/ .mvn
COPY mvnw ./mvnw
COPY pom.xml ./pom.xml

RUN dos2unix ./mvnw && apt-get --purge remove -y dos2unix && rm -rf
    /var/lib/apt/lists/*

RUN ./mvnw dependency:resolve

COPY src ./src

FROM base as development
CMD ["./mvnw", "spring-boot:run", "-Dspring-boot.run.profiles=default",
"-Dspring-boot.run.jvmArguments=
'-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*:8000'"]

FROM base as build
RUN ./mvnw package

FROM eclipse-temurin:17-jdk-jammy as production
EXPOSE 8080
COPY --from=build /app/target/artificial-generation-backend-*.jar
    /artificial-generation-backend.jar
CMD ["java", "-Djava.security.edg=file:/dev/ ./urandom", "-jar",
    "/artificial-generation-backend.jar"]
```

Zdrojový kód 8: Dockerfile pro projekt ve Spring Boot. Kód vytvořen podle: [49], [48]

Soubor Dockerfile byl pro projekt napsaný v Pythonu vytvořen podobným způsobem. Jako základní obraz Dockeru byl použit python:3.11.3-bullseye a do výsledného obrazu byl zkopírován pouze obsah složky src a soubor requirements.txt, na základě kterého se při sestavování nainstalují potřebné balíčky.

Jak již bylo řečeno výše, pro snadné spuštění této části aplikace, byl vytvořen soubor docker-compose, který využívá výše popsané soubory Dockerfile. V rámci tohoto souboru byly také specifikovány hodnoty některých nastavení pro jednotlivé části, které tak nahrazují tyto hodnoty v souborech application.yml pro projekt ve Spring Boot a .env pro projekt v Pythonu. Po vytvoření tohoto souboru bylo možné celý backend spustit jediným příkazem bez nutnosti instalace programového vybavení potřebného pro spuštění jednotlivých součástí backendu.

Provedením výše popsaných kroků bylo značně zjednodušeno spouštění a následné použití backendu na lokálním počítači. Díky vytvořeným souborům Dockerfile a docker-compose tak pro spuštění celé této části aplikace stačí jediný příkaz bez nutnosti manuální instalace jednotlivých programových vybavení s výjimkou Dockeru samotného.

6.2 Implementace grafické části aplikace

Jak již bylo zmíněno v kapitole 3, pro implementaci grafické části byl zvolen herní engine Unity 2022.3.0f1 s jazykem C# 9.0 (viz kap. 3.1) na základě předešlých zkušeností autora s tímto programovým vybavením. Tato část byla vytvářena s pomocí vývojového prostředí Visual Studio 2022.

6.2.1 Architektura grafické části aplikace

Před představením jednotlivých částí grafické části aplikace je vhodné blíže představit její obecnou strukturu. V této podkapitole jsou tudíž popsána obecná řešení společná pro celou aplikaci.

Struktura projektu v Unity byla založena stejně jako v případě backendu (viz kap. 6.1.1) na rozdělení dle funkcí a nikoliv dle infrastruktury. I v tomto případě tak byla usnadněna správa jednotlivých částí.

Jednotlivé zdrojové kódy, materiály, modely a prefaby byly tudíž rozděleny do složek dle funkcí, které byly umístěny do složky „App“. Tyto základní složky a jejich bližší oblasti zájmu jsou vyobrazeny v tabulce 3.

Výše popsané obecné oblasti byly poté, s výjimkou složky Profiling, dále rozděleny dle konkrétních podoblastí týkajících se konkrétních funkcí. Toto rozdělení je přiblíženo v ta-

Název složky	Oblast
Common	Funkce používané napříč aplikací
Core	Funkce související s hlavními funkcemi aplikace
Profiling	Funkce související s testováním výkonu aplikace

Tabulka 3: Základní složky projektu v Unity a jejich význam

bulce 4. Jednotlivé oblasti tak byly jasně odděleny do svých vlastních kategorií, se kterými bylo poté možné snadno izolovaně pracovat a v případě obecných funkcí je také rozšiřovat pro konkrétní funkce bez ovlivnění ostatních částí aplikace.

Název hlavní složky	Název složky	Oblast
Common	Commands	Obsluha požadavků ve formě objektů
	Convertors	Definice rozhraní pro konverzi mezi doménou a DTO
	DI	Nastavení Dependency Injection kontejneru
	Enums	Definice výčtu s chováním
	EventSystem	Systém událostí aplikace
	FileSelection	Výčet hodnot pro výběr souborů
	GUI	Obecné grafické komponenty
	Logging	Logování v aplikaci
	ScreensManaging	Správa obrazovek
	Serialization	Definice rozhraní pro serializaci
Core	BackendConnection	Komunikace s backendem
	Models	Správa modelů a úchopových bodů
	ProjectContext	Správa kontextu projektu
	SceneDefinition	Správa definice scén
	Scenes	Správa scén

Tabulka 4: Bližší pohled na složky projektu v Unity a jejich význam

Třídy byly dále rozděleny do jmenných prostorů dle výše popsaných složek, aby bylo toto rozdělení explicitně vyjádřeno také v samotném kódu. Jmenné konvence zdrojových kódů tohoto projektu se snaží respektovat jmenné konvence uvedené na webu Microsoft Learn (viz [50]).

Jednotlivé rozdělené oblasti poté bylo potřeba naučit spolu komunikovat. K tomuto účelu byl nejprve vyzkoušen přístup přes třídy typu Manager. Jak uvádí Kesler, instance třídy

Manager se ve hře vyskytuje pouze jednou, nelze vytvořit její kopii a k její implementaci lze využít návrhový vzor Jedináček. [51]

Tento přístup umožnil přistupovat k potřebným funkcím napříč jednotlivými částmi aplikace bez nutnosti explicitního přiřazování referencí pro dané oblasti. Zároveň však snížil čitelnost kódu z hlediska informace o závislostech jednotlivých tříd a logika byla vždy vázána na konkrétní implementaci. Z tohoto důvodu byl tento přístup opuštěn ve prospěch vkládání závislostí (angl. Dependency Injection).

Jak uvádí Fowler, základní myšlenkou Dependency Injection je mít samostatný objekt, který doplní vhodnou implementaci dle rozhraní do příslušné třídy. Mezi tři hlavní styly patří Constructor Injection, Setter Injection a Interface Injection [52]. Tento přístup tudíž umožnil explicitně definovat jaké závislosti daná třída má a definovat je proti rozhraní. Tím došlo k dalšímu snížení míry provázání mezi jednotlivými oblastmi aplikace při zachování jasného přehledu o závislostech daných tříd.

Jelikož implementace vlastního systému pro Dependency Injection není předmětem této práce, byl k tomuto účelu využit framework Zenject (resp. fork jménem Extenject). Jak je uvedeno na stránce této knihovny, Zenject byl vytvořen pro Unity jako framework pro Dependency Injection, se kterým lze součásti aplikace se silně oddělenými odpovědnostmi volně provázat. [53]

Tento framework byl nainstalován prostřednictvím instalace z balíčku unity (viz kap. 3.5) získaného z repozitáře na GitHubu ([53]). Nastavení potřebné pro zprovoznění tohoto frameworku v rámci projektu bylo s přiloženou dokumentací snadné a sestávalo ve vytvoření třídy typu Installer a objektu Scene Context ve scéně. Jak je uvedeno v dokumentaci Zenject, všechny závislosti jsou sestaveny v MonoBehaviour SceneContext a pro jejich vyjádření a vyjádření vztahů mezi nimi je potřeba vytvořit tzv. Installer. [53, kap. Hello World Example]

Příklad vytvořené třídy typu Installer je uveden ve zdrojovém kódu 9 a příklad vyžádání dané třídy pak ve zdrojovém kódu 10. Vzhledem k délce těchto zdrojových kódů jsou v ukázkách části nepodstatné pro tento příklad vynechány.

```
public class ApplicationInstaller : MonoInstaller
{
    [SerializeField]
    private DefaultProjectStateScriptableObject _defaultStateScriptableObject;
    public override void InstallBindings()
    {
        Container.Bind<IBackendConnectionService>()
            .To<BackendConnectionService>()
            .AsSingle()
            .NonLazy();

        Container.Bind<IProjectContextService>()
            .To<ProjectContextService>()
            .AsSingle()
            .NonLazy();

        Container.Bind<ISerializer<Model>>()
            .To<ModelSerializer>()
            .AsSingle();
    ...
}
```

Zdrojový kód 9: Část kódu třídy typu Installer. Kód vytvořen podle: [53]

```
public class ScenesDefinitionView : MonoBehaviour
{
    ...
    private IProjectContextService _projectContextService;
    private ISceneService _sceneService;
    private ISceneDefinitionSettingsService _sceneDefinitionSettingsService;

    [Inject]
    public void Construct(IProjectContextService projectContextService,
        ISceneService sceneService,
        ISceneDefinitionSettingsService sceneDefinitionSettingsService)
    {
        _projectContextService = projectContextService;
        _sceneService = sceneService;
        _sceneDefinitionSettingsService = sceneDefinitionSettingsService;
    }
    ...
}
```

Zdrojový kód 10: Ukázka kódu požadujícího dodání závislostí. Kód vytvořen podle: [53]

Pro komunikaci mezi jednotlivými objekty, které musely být umístěny přímo ve scéně, však byl použit způsob komunikace založený na událostech. Správa těchto událostí ve scéně byla postavena na základě tzv. skriptovacích objektů (angl. Scriptable Objects) (viz 3.2). Tyto skriptovací objekty tak v sobě drží seznam posluchačů a umožňují vyvolání příslušné události.

Jelikož bylo potřeba skrze tyto události posílat data různých typů, byla při jejich vytváření využita genericita. Jak uvádí Davis, Unity nepodporuje vytváření instancí generických tříd. Toto omezení lze však obejít tím, že se z dané třídy s definovaným parametrem zdědí. [54]

Kromě samotných tříd reprezentujících jednotlivé události byly také vytvořeny třídy sloužící k jejich naslouchání. Třídy reprezentující události poté byly jednotlivým objektům ve scéně přiřazeny skrze editor po jejich definici jako atribut v příslušném skriptu. Třídy naslouchající těmto událostem poté byly přiřazeny pomocí atributu „RequireComponent“ (viz 3.2).

Zdrojové kódy základních tříd reprezentujících události jsou uvedeny v ukázkách 11 a 12. Základní třídy naslouchající daným událostem jsou poté uvedeny v ukázkách zdrojových kódů 13 a 14.

```
[CreateAssetMenu(menuName = "Events/Application event without parameter")]
public class ApplicationNonParameterEvent : ScriptableObject
{
    private List<IApplicationNonParameterEventListener> _listeners = new
        List<IApplicationNonParameterEventListener>();

    public void Raise()
    {
        for (int i = 0; i < _listeners.Count; i++)
        {
            _listeners[i].OnEventRaised();
        }
    }

    public void RegisterListener(IApplicationNonParameterEventListener listener)
    {
        if (!_listeners.Contains(listener))
        {
            _listeners.Add(listener);
        }
    }

    public void UnRegisterListener(IApplicationNonParameterEventListener listener)
    {
        if (_listeners.Contains(listener))
        {
            _listeners.Remove(listener);
        }
    }
}
```

Zdrojový kód 11: Bezparametrická třída události. Kód vytvořen podle: [55, čas 33:55]

```
public class ApplicationSingleParameterEvent<T> : ScriptableObject
{
    private List<IApplicationSingleParamEventLisener<T>> _listeners = new
        List<IApplicationSingleParamEventLisener<T>>();

    public void Raise(T parameter)
    {
        for (int i = 0; i < _listeners.Count; i++)
        {
            _listeners[i].OnEventRaised(parameter);
        }
    }

    public void RegisterListener(IApplicationSingleParamEventLisener<T> listener)
    {
        if (!_listeners.Contains(listener))
        {
            _listeners.Add(listener);
        }
    }

    public void UnRegisterListener(IApplicationSingleParamEventLisener<T> listener)
    {
        if (_listeners.Contains(listener))
        {
            _listeners.Remove(listener);
        }
    }
}
```

Zdrojový kód 12: Parametrická třída události. Kód vytvořen podle: [55, čas 33:55], [54]

```
public class ApplicationNonParameterEventListenerBase : MonoBehaviour,
    IApplicationNonParameterEventListener
{
    private ApplicationNonParameterEvent _applicationEvent;
    private Action _action;

    public void Init(Action action, ApplicationNonParameterEvent applicationEvent)
    {
        _action = action;
        _applicationEvent = applicationEvent;
        _applicationEvent.RegisterListener(this);
    }

    public void OnEventRaised()
    {
        _action?.Invoke();
    }

    private void OnEnable()
    {
        if (_applicationEvent != null)
        {
            _applicationEvent.RegisterListener(this);
        }
    }

    private void OnDisable()
    {
        if (_applicationEvent != null)
        {
            _applicationEvent.UnRegisterListener(this);
        }
    }
}
```

Zdrojový kód 13: Třída naslouchající bezparametrické události. Kód vytvořen podle: [55, čas 35:00]

```
public abstract class ApplicationSingleParamEventListenerBase<T> : MonoBehaviour,
IApplicationSingleParamEventLisener<T>
{
    private ApplicationSingleParameterEvent<T> _applicationEvent;
    private Action<T> _action;

    public void Init(Action<T> action, ApplicationSingleParameterEvent<T>
        applicationEvent)
    {
        _action = action;
        _applicationEvent = applicationEvent;
        _applicationEvent.RegisterListener(this);
    }

    public void OnEventRaised(T parameter)
    {
        _action?.Invoke(parameter);
    }

    private void OnEnable()
    {
        if (_applicationEvent != null)
        {
            _applicationEvent.RegisterListener(this);
        }
    }

    private void OnDisable()
    {
        if (_applicationEvent != null)
        {
            _applicationEvent.UnRegisterListener(this);
        }
    }
}
```

Zdrojový kód 14: Třída naslouchající parametrické události. Kód vytvořen podle: [55, čas 35:00], [54]

Veškeré třídy událostí a posluchačů využívané v rámci tohoto systému událostí byly pro konkrétní datové typy vytvořeny zděděním ze tříd uvedených výše stejným způsobem, jako je vyobrazeno na ukázkách zdrojových kódů 15 a 16. Z daných tříd poté dědily další třídy naslouchající konkrétním událostem, jako je kupříkladu vytváření či mazání.

```
[CreateAssetMenu(menuName = "Events/Application boolean event")]
public class ApplicationBooleanEvent : ApplicationSingleParameterEvent<bool>
{
}
```

Zdrojový kód 15: Třída události pro datový typ boolean. Kód vytvořen podle: [54]

```
public class ApplicationBooleanEventListener :
    ApplicationSingleParamEventListenerBase<bool>
{
}
```

Zdrojový kód 16: Třída naslouchající události s typem boolean. Kód vytvořen podle: [54]

Tento přístup umožnil explicitně definovat, které objekty ve scéně vyvolávají konkrétní události a kterým událostem konkrétně daný objekt naslouchá. Ačkoliv tento systém funguje a skutečně vedl ke snížení míry svázání objektů ve scéně, lze mu z programátorského hlediska vytknout jisté problémy.

Prvním problémem je nutnost ručního přiřazování konkrétních objektů událostí prostřednictvím editoru. Tato skutečnost zvyšovala riziko chyb z důvodu opomenutí danou referenci přiřadit a zároveň zvyšovala množství práce s nastavením daného objektu.

Další problém spočívá v množství tříd, které pro něho bylo nutné vytvořit. Vzhledem k výše popsaným problémů s genericitou u tříd dědících z typů definovaných Unity vzniklo velké množství tříd s prázdným tělem, jejichž jediným účelem je tuto genericitu před Unity skrýt.

Při budoucím rozšiřování této aplikace by tudíž mohlo být vhodné vzhledem k problémům popsaným výše stávající systém pozměnit. Jednou z možných úprav je namísto skriptova-

cích objektů využívat standardní třídy, čímž by byl eliminován mezikrok s přiřazováním v editoru a zároveň by bylo možné plně využívat genericity.

Další možností aplikovatelnou s tou zmíněnou výše je plně využít návrhového vzoru pozorovatel (angl. Observer). Jak je uvedeno na stránce Refactoring Guru, v rámci tohoto návrhového vzoru může naslouchání notifikacím započít za běhu aplikace a udržování seznamu posluchačů lze delegovat do samostatného pomocného objektu, který lze také povýšit do vysílače centralizovaných událostí [56]. S tímto přístupem by tak bylo možné spravovat události centrálně bez úzkého provázání jednotlivých tříd.

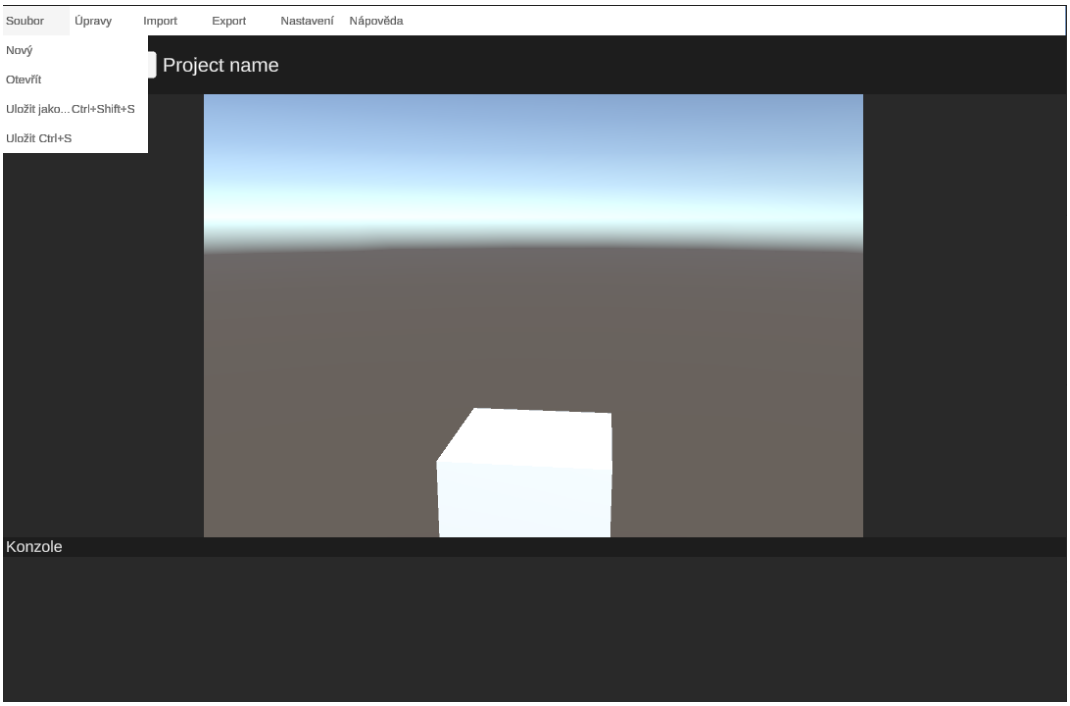
Některé operace bylo potřeba implementovat asynchronně. Z tohoto důvodu byla přidána knihovna UniTask. Jak je uvedeno v repozitáři na GitHubu, Unity používá jedno vlákno, čemuž Task neodpovídá. Proto je potřeba využívat Unitask, který vlákna nepoužívá a je spouštěn vrstvou enginu Unity. [57, kap. Basics of UniTask and AsyncOperation]

Pro potřeby serializace a deserializace dat byl v projektu použit formát JSON a balíček Newtonsoft Json Unity nainstalovaný z repozitáře. Tyto operace konkrétně využívaly třídu JsonSerializer. Jak je uvedeno v dokumentaci, JsonSerializer je nejrychlejší způsob převodu .NET objektu do JSON a tento převod spočívá v mapování názvů vlastností v .NET objektech na názvy vlastností v JSON. [58]

Pro snížení paměťové náročnosti v případě práce s velkými JSON řetězci byly při tomto procesu vždy používány paměťové proudy. Dle dokumentace Json.NET podporuje práci s proudy, přičemž jejich využití je obzvláště důležité při práci s dokumenty většími než 85 kb, kdy pak řetězec není do paměti nahrán celý a místo toho je nahráván po částech, čímž neskončí na haldě pro velké objekty. [59]

Aby byl tento proces jednodušší, byly samotné doménové objekty před serializací převáděny do DTO objektů a opačně v případě deserializace. Těmto DTO objektům se také nastavily atributy serializace pro pevné nastavení jmen vlastností po serializaci. Jak uvádí dokumentace, „atributy mohou být použity pro ovládání toho, jak Json.NET serializuje a deserializuje .NET objekty.“ [60] (překlad autora)

Zároveň s tvorbou výše popsaných systémů také probíhala implementace grafického rozhraní aplikace, která započala vytvořením základního rozpoložení panelů podle návrhu v kapitole 5.2. Tato původní podoba je vyobrazena na obrázku 8.



Obrázek 8: Základní podoba grafického rozhraní

Hlavní myšlenka spočívala v rozdělení okna aplikace na čtyři základní sekce, kterými jsou lišta s menu, stavový panel, hlavní obrazovka a konzole. Všechny tyto sekce byly navrženy tak, aby jejich obsah byl pro uživatele vždy viditelný. V případě hlavní obrazovky se však tento obsah mění v závislosti na hodnotách ve stavovém panelu.

Z uvedených sekcí je v této podkapitole popsán pouze stavový panel. Ostatní sekce jsou přiblíženy v následujících podkapitolách. Tento panel v aplikaci slouží k přepínání aktivní obrazovky, informování o stavu připojení k backendu a k poskytnutí možnosti se k backendu připojit. Implementace přepínání aktivní obrazovky také aktivuje či deaktivuje příslušné objekty ve scéně dle potřeby.

Možnosti v rozbalovací nabídce pro výběr aktivní obrazovky byly implementovány pomocí tříd typu Enumeration (viz zdrojový kód 17) namísto výčtů, aby je bylo možné snadno zobrazit v uživatelském prostředí a zároveň bylo možné implementovat příslušnou logiku bez nutnosti skládání podmínek.

Jak je uvedeno na webu Microsoft Learn, použití výčtů pro řízení toku může vést ke křehkému kódu a pro využití funkcí objektově orientovaného jazyka lze namísto nich využít třídy typu Enumeration. [61]

```
public abstract class Enumeration : IComparable
```

```
{
    public string Name { get; private set; }
    public int Id { get; private set; }

    protected Enumeration(int id, string name)
    {
        Id = id;
        Name = name;
    }

    public static IEnumerable<T> GetValues<T>() where T : Enumeration
    {
        return typeof(T).GetFields(BindingFlags.Public |
                           BindingFlags.Static |
                           BindingFlags.DeclaredOnly)
            .Select(f => f.GetValue(null))
            .Cast<T>();
    }

    public static IEnumerable<string> GetNames<T>() where T : Enumeration
    {
        return GetValues<T>().Select(f => f.Name);
    }

    public static T FromValue<T>(int value) where T : Enumeration
    {
        return GetValues<T>().Single(r => r.Id == value);
    }

    public int CompareTo(object other)
    {
        return Id.CompareTo(((Enumeration)other).Id);
    }

    public override bool Equals(object obj)
    {
        if (obj is not Enumeration otherValue)
        {
            return false;
```

```
            }

            bool typeMatches = GetType().Equals(obj.GetType());
            bool valueMatches = Id.Equals(otherValue.Id);

            return typeMatches && valueMatches;
        }

        public override int GetHashCode()
        {
            return Id.GetHashCode();
        }

        public override string ToString()
        {
            return Name;
        }
    }
```

Zdrojový kód 17: Třída Enumeration. Kód vytvořen podle: [61], [62]

Akce prováděné po výběru aktivní obrazovky byly implementovány s pomocí návrhového vzoru strategie (angl. Strategy). Jak uvádí stránka Refactoring Guru, tento návrhový vzor lze použít pro extrakci algoritmů do samostatných tříd a tím se vyhnout použití velkých podmíněných výrazů [63]. Touto implementací je tak značně usnadněno případné budoucí rozšíření o další obrazovky.

Pro ulehčení práce s formuláři byly také vytvořeny komponenty obsahující popisek a pole pro zadávání hodnot. Tyto komponenty zároveň umožňují získat a nastavovat jejich hodnoty v příslušném datovém typu, což značně ulehčilo kontrolu validity zadávaných dat a jejich automatickou aktualizaci.

V rámci obecných prvků uživatelského rozhraní bylo v neposlední řadě také vytvořeno základní rozhraní pro později využívané seznamy. Toto rozhraní umožňuje definovat například akce při výběru dané položky či jejím smazání. Využití nalezlo zejména při implementaci správy úchopových bodů a scén.

Pro některé uživatelské prvky byly také vytvořeny vlastní ikonky pro znázornění jejich účelu či možných akcí. Tyto ikonky byly vytvořeny v programu Affinity Designer 2.0.3 a jsou vyobrazeny na obrázku 9.

Obrázek 9: Vytvořené ikonky

Kroky popsanými v rámci této podkapitoly byl vytvořen architektonický základ celé aplikace a některé základní prvky uživatelského prostředí. V následujících podkapitolách je popsán způsob implementace konkrétních vyžadovaných funkcionalit, na které bylo možné se díky tomuto vybudovanému základu soustředit.

6.2.2 Implementace logovací konzole

Vzhledem k různým operacím, které může aplikace vykonávat, je vhodné uživatele určitým způsobem průběžně informovat co se v daný okamžik v rámci aplikace děje. K tomuto účelu byla vytvořena logovací konzole zobrazující jednotlivé zprávy logu.

Vytvořená komponenta logovací konzole byla umístěna do spodní části obrazovky, kde je vždy viditelná. Pro zprávy v této konzoli byl vytvořen prefab (viz kap. 3.2), který je při každém zavolání logu instancován do posouvacího panelu konzole.

Pro reprezentaci jednotlivých zpráv v datové vrstvě byla vytvořena speciální třída, která mimo samotné zprávy drží například informace o jejím typu, času pořízení či zdrojové třídě. Jednotlivé typy zpráv tak bylo možné snadno barevně odlišit pomocí definice barvy pro daný typ hexadecimálním řetězcem a následného parsování samotné barvy z této hodnoty při vytváření instance skriptu pro reprezentaci v konzoli. Jak je uvedeno v dokumentaci Unity, k parsování barvy z hexadecimálního řetězce lze využít ColorUtility.TryParseHtmlString. [18, s. ColorUtility.TryParseHtmlString]

Vkládání zpráv do konzole bylo implementováno pomocí vlastního jednoduchého loggeru. Vzhledem ke skutečnosti, pro každou třídu byl zamýšlen vlastní logger, byla zvolena implementace s veřejnými statickými událostmi pro jednotlivé typy zpráv a instančními metodami, které tyto události vyvolávají. Každá instance si drží informace o třídě, která vyvolala její vytvoření, vytváří konkrétní zprávy a vyvolává události, které poté obsluhuje samotná konzole.

Logika logování tímto způsobem tudíž byla oddělena od zbytku aplikace bez nutnosti předávání konkrétních instancí loggeru přes parametr. Jak se později ukázalo, zvolená implementace velice usnadnila zapisování událostí tak, aby je uživatel viděl.

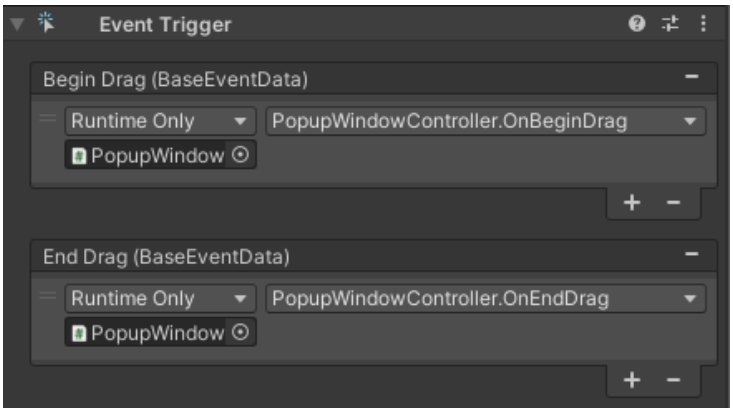
6.2.3 Implementace vyskakovacích oken

Pro přívětivější uživatelský zážitek a efektivní způsob informování uživatele o důležitých informacích byl kromě logovací konzole (viz kap. 6.2.2) implementován také jednoduchý systém vyskakovacích oken. Tato okna je možné vzhledem k limitaci použité technologie obsluhovat pouze v rámci okna aplikace bez možnosti je přesunout například na vedlejší obrazovku.

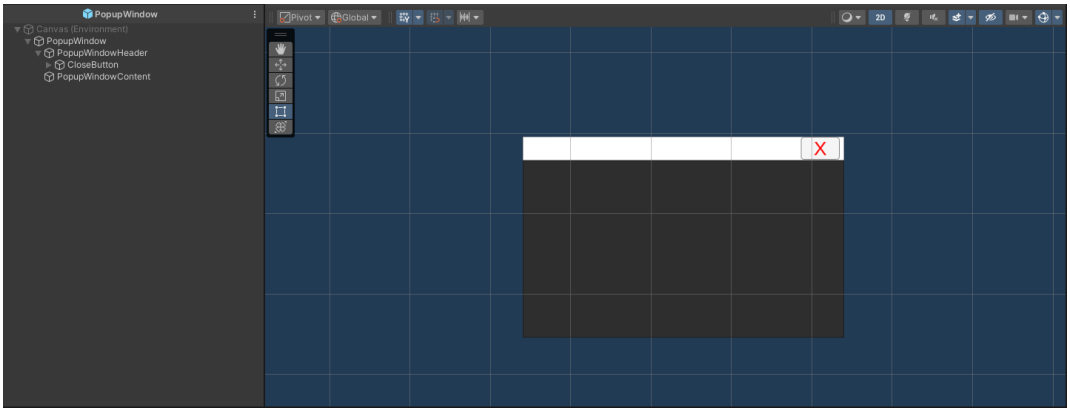
Nejprve bylo vytvořeno základní vyskakovací okno, které neblokuje interakci s ostatními prvky uživatelského rozhraní. K tomu byl vytvořen jednoduchý panel s hlavičkou obsahující tlačítko pro zavření a tělem pro budoucí obsah.

Dále byl k tomuto oknu připojen skript umožňující s ním pohybovat tažením myši. Jak je uvedeno v dokumentaci Unity UI, počátek a konec operace tažení lze zpracovat metodami OnBeginDrag a OnEndDrag definovaných v rozhraních IBeginDragHandler a IEndDragHandler [19, s. Supported Events]. Ačkoliv bylo možné na tyto metody reagovat implementací příslušných rozhraní, byl zde zvolen přístup nastavení příslušných metod pro obsluhu těchto událostí skrze GUI editoru, který je vyobrazen na obrázku 10.

Nakonec byl z tohoto uskupení vytvořen prefab (viz 3.2) ulehčující jeho správu a vytváření dalších typů vyskakovacích oken na jeho základě. Hierarchie a podoba tohoto prefabu je vyobrazena na obrázku 11.



Obrázek 10: Nastavení obsluhy událostí tažení v editoru Unity

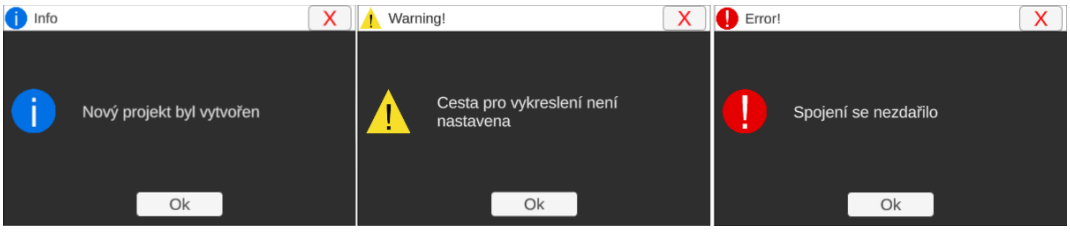


Obrázek 11: Prefab neblokujícího vyskakovacího okna

Poté byl na základě výše popsaného dialogu vytvořen prefab blokujícího dialogu, jehož blokování spočívá v přidání průhledného obalovacího panelu, který však zachytává události kliknutí a umožňuje na tyto události spustit předem definovaný zvuk pomocí komponenty „Audio Source“ (viz 3.2). Tato funkce však nakonec nebyla použita z důvodu nedostatečné knihovny zvuků vhodných k tomuto účelu.

Jak je uvedeno v dokumentaci Unity UI, událost kliknutí lze zpracovat metodou OnPointerClick v rozhraní IpointerClickHandler [19, s. Supported Events]. I v tomto případě byl však zvolen způsob nastavení skrze GUI editoru.

Z tohoto okna byly poté vytvořeny příslušné dialogy s informativními, varovnými a chybovými zprávami, které jsou zobrazeny na obrázku 12. Dále byl na základě tohoto okna vytvořen dialog informující uživatele o skutečnosti, že probíhá akce a dialog s upravitelnou velikostí. Druhý jmenovaný byl později využit při tvorbě interaktivních dialogů.



Obrázek 12: Typy vyskakovacích dialogů

Aby bylo možné tato vyskakovací okna jednoduše používat v různých částech uživatelského prostředí, byl vytvořen správce, který tyto dialogy drží a umožňuje jejich zobrazení a vytváření. Pro jeho implementaci byl zvolen přístup pomocí tzv. jedináčka (angl. Singleton). V tomto případě je však tato instance skryta a veškeré operace se volají přes statické metody. Část zdrojového kódu tohoto správce je uvedena ve zdrojovém kódu 18.

V případě práce se soubory nedošlo na vytváření vlastních komponent a místo toho byla použita již existující komponenta z balíčku „Unity Simple File Browser“ (viz [65]). Tento dialog je tak jediný, který není dále v aplikaci volán přes výše popsaného správce.

```
public class PopupWindowsManager : MonoBehaviour
{
    private static PopupWindowsManager _instance;

    [SerializeField]
    private AcceptPopupWindowController _errorWindowPrefab;
    ...
    private void Awake()
    {
        if (_instance != null && _instance != this)
        {
            Destroy(this);
        }
        else
        {
            _instance = this;
        }
    }
    ...
    public static void ShowErrorDialog(string message)
    {
        AcceptPopupWindowController acceptPopupWindowController =
            Instantiate(_instance._errorWindowPrefab, _instance.transform);
        acceptPopupWindowController.Init(message, "Error!");
    }
    ...
}
```

Zdrojový kód 18: Část kódu správce vyskakovacích oken. Kód vytvořen podle: [64], [65]

Vytvořením výše popsaných vyskakovacích oken a dialogů bylo značně usnadněno další vytváření komponent uživatelského prostředí. Dále se tímto krokem dosáhlo chování, které je bližší standardním aplikacím.

6.2.4 Implementace lišty s menu

Ve snaze vytvořit standardní uživatelské rozhraní byla implementována lišta s menu. Do této lišty se umístily operace, které tématicky nezapadaly do jiných částí grafického rozhraní či bylo potřeba, aby byly dostupné nezávisle na aktivní obrazovce.

Nejprve byla implementována kostra lišty s menu v horní nástrojové liště. V tomto bodě stále nebyly implementovány funkce pro jednotlivé položky menu, ale samotné zobrazení nabídky po kliknutí a její skrytí po posunutí myši mimo ni již funkční bylo. Logika zobrazení podmenu po kliknutí byla založena na [66].

Poté byly implementovány řadiče pro jednotlivé položky menu, které byly přiřazeny k hlavním pěti položkám. Každá položka menu tak držela definici akcí pro všechny své podpoložky a nebylo tak potřeba vytvářet pro každou položku samostaný skript.

Mimo jiné zde byla také přidána možnost zobrazit si dokumentaci aplikace prostřednictvím jedné z položek menu, která způsobí její otevření v prohlížeči zavoláním metody „Application.OpenUrl“. Jak je uvedeno v API manuálu Unity, spuštění tohoto příkazu probíhá pod stejnými právy, které má samotná aplikace a způsobí otevření dané cesty. Na některých platformách jej lze využít k otevírání souborů.[67]

Některé z ostatních položek přímo spouštějí určité operace. V jejich případě byl využit návrhový vzor Command. Jak je uvedeno na webu Refactoring guru, pomocí návrhového vzoru Command se požadavek změní do objektu se všemi potřebnými informacemi a vykonání tohoto požadavku je pak možné například pozdržet [68]. Díky využití tohoto návrhového vzoru tak bylo možné zabránit opětovnému spouštění daných operací dokud stále běží její předešlá instance.

Tato skutečnost byla velmi užitečná zejména v případě operací pro správu projektu, kdy operace pracující se soubory na disku jsou spustitelné pomocí klávesových zkratek. Jelikož danou operaci nelze vyvolat dokud předešlá nebyla dokončena, nebude znovu provedena ani když bude opět vyvolána klávesovou zkratkou.

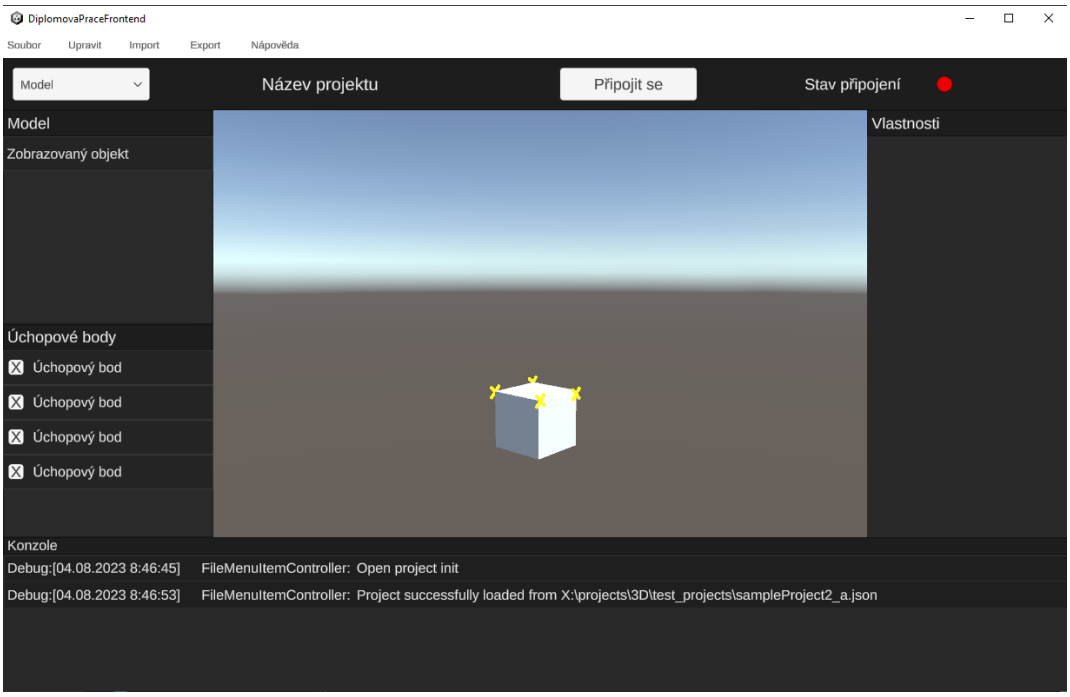
Lišta s menu byla poté průběžně obohacována o konkrétní implementace funkcí dle příslušných oblastí. Při dalším vývoji tak poskytla mimo jiné snadný a rychlý způsob testování některých nově přidávaných funkcí.

6.2.5 Implementace správy projektu

Aby bylo možné snadno pracovat se stavem aplikace jako celku, bylo nutné vytvořit datovou entitu, která bude tento stav uchovávat. Z tohoto důvodu vznikla třída ProjectContextData, která v sobě uchovává veškerá data nutná pro uložení a rekonstrukci projektu.

Zmíněná třída obsahuje například informace o zobrazovaném modelu, veškeré úchopové body či vygenerované scény. Její správa je realizována pomocí servisní třídy, která je potřebným třídám distribuována pomocí dependency injection (viz kap. 6.2.1).

Systém serializace a deserializace byl pak implementován s pomocí tzv. DTO objektů, do kterých jsou data před serializací převedena a formátu JSON (viz kap. 6.2.1). Změna vnitřní implementace tudíž během vývoje nezpůsobovala problémy se změnou definice dat a jejich následným pokusem o načtení dat projektu. Příklad načteného projektu je vyobrazen na obrázku 13.



Obrázek 13: Příklad projektu po načtení

Jako možnou nevýhodu zvoleného formátu JSON lze uvést velikost výsledných souborů po uložení projektu. Tento problém by mohl být v případě budoucích rozšíření vyřešen implementací vlastního formátu a následného využití komprese. Původní formát by poté mohl stále sloužit pro snadný přenos do jiné aplikace.

Přímá manipulace s kontextem aplikace byla uživateli umožněna prostřednictvím lišty s menu a příslušných klávesových zkratek. Tyto operace zahrnují možnost vytvoření nového projektu, načtení projektu ze souboru a jeho uložení. Pro bližší informace o těchto operacích vizte dokumentaci aplikace (Příloha A).

Implementací správy projektu bylo v aplikaci umožněno snadné ukládání a načítání projektu. Velikost výsledných souborů lze v případných budoucích rozšířeních optimalizovat implementací vlastního formátu a komprese.

6.2.6 Implementace komunikace s backendem

Jelikož byly jisté pokročilejší funkce delegovány na backend, bylo nutné implementovat způsob, kterým by s ním mohla aplikace komunikovat. V rámci této aplikace byla k tomuto účelu vytvořena samostatná služební třída, která spravuje připojení a provolává potřebné API a pomocná datová entita, která v sobě drží příslušné informace o připojení.

Jak již bylo řečeno v kapitole 6.1.1, pro rychlou komunikaci a možnost získávání aktualizací z backendu pomocí událostí byla použita implementace knihovny Socket.IO. Pro jejich použití v Unity byla konkrétně použita knihovna SocketIOUnity (viz [69]). Příklad připojení pomocí této knihovny je uveden ve zdrojovém kódu 19.

Přes tuto knihovnu však není řešeno samotné provolávání API spouštějící akce pro parsování modelů ze souboru a generování scén dle definice. K těmto účelům je využíván UnityWebRequest, který provolává příslušné endpointy na API backendu. Pro posílání souborů na tyto endpointy byla využita třída MultipartFormFileSection (viz [17, s. Sending a form to an HTTP server (POST)]).

Zvolený systém tudíž funguje tak, že se klient připojí pomocí Socket.IO k backendu, který mu vrátí unikátní ID, se kterým se poté identifikuje pro všechny budoucí požadavky. Poté se například při požadavku o naparsování dat o modelu ze souboru v požadavku uvede výše zmíněné ID v cestě jako parametr a samotná data souboru se vloží jako tělo typu

```
var connectionUri = new Uri(uri);


SocketIOUnity socket = new SocketIOUnity(uri, new SocketIOOptions
{
    Reconnection = true,
    ReconnectionDelayMax = 1000,
    ReconnectionAttempts = 1,
    ConnectionTimeout = TimeSpan.FromSeconds(1),
    Transport = SocketIOClient.Transport.TransportProtocol.WebSocket
});
```

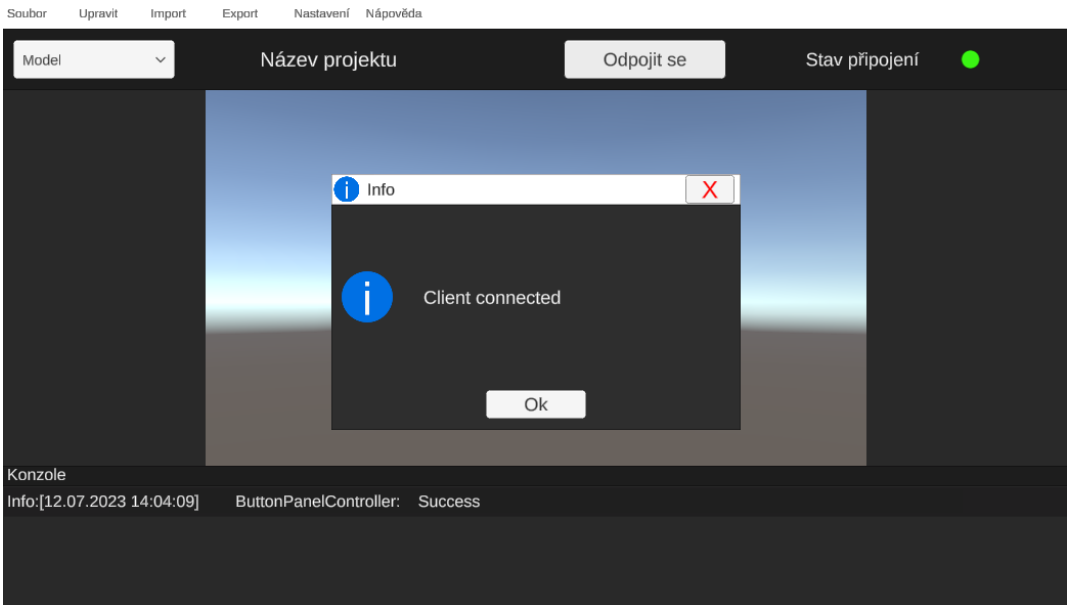
Zdrojový kód 19: Příklad připojení pomocí SocketIOUnity. Kód vytvořen podle: [69]

Multipart file. Backend vrátí odpověď s unikátním ID příslušné operace a klient začne čekat na událost dokončení této operace. Jakmile klient obdrží událost o dokončení dané operace, provede zavolání dalšího požadavku na obdržený endpoint a příslušná data si tím stáhne. Poté z nich získá potřebné informace, se kterými dále pracuje stejně jako v případě řešení implementovaného čistě v rámci Unity.

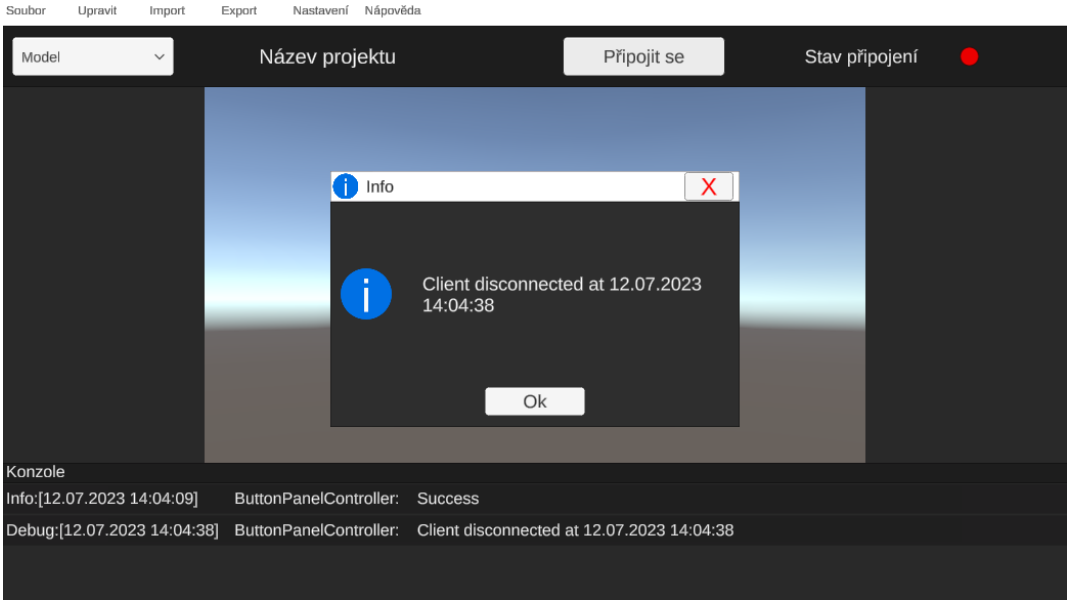
První verze umožňovala připojení k backendu v uživatelském prostředí prostým stiskem tlačítka „Připojit se“. V této fázi nebylo možné stanovit konkrétní cestu hostitele, ani číslo portu a obě tyto hodnoty tak byly pevně nastaveny na „http://localhost“ a „8085“ dle konfigurace lokální instance backendu.

Pro snadné sledování stavu připojení byla také do stavového panelu přidána stavová kontrolka, která se při úspěšném připojení k backendu zbarví na zeleno (obr. 14). Aby byla změna stavu připojení názornější, přidaly se po připojení a odpojení informační dialogy (obr. 15 a 16).

Obrázek 14: Panel se stavem připojení

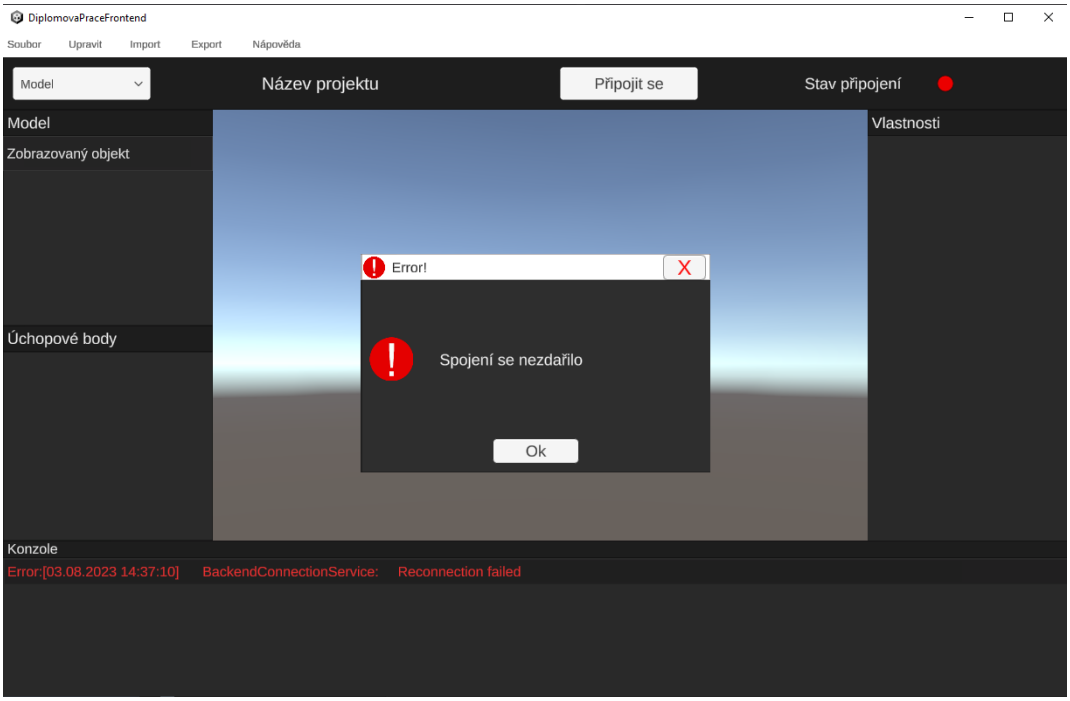


Obrázek 15: Hláška po připojení k backendu



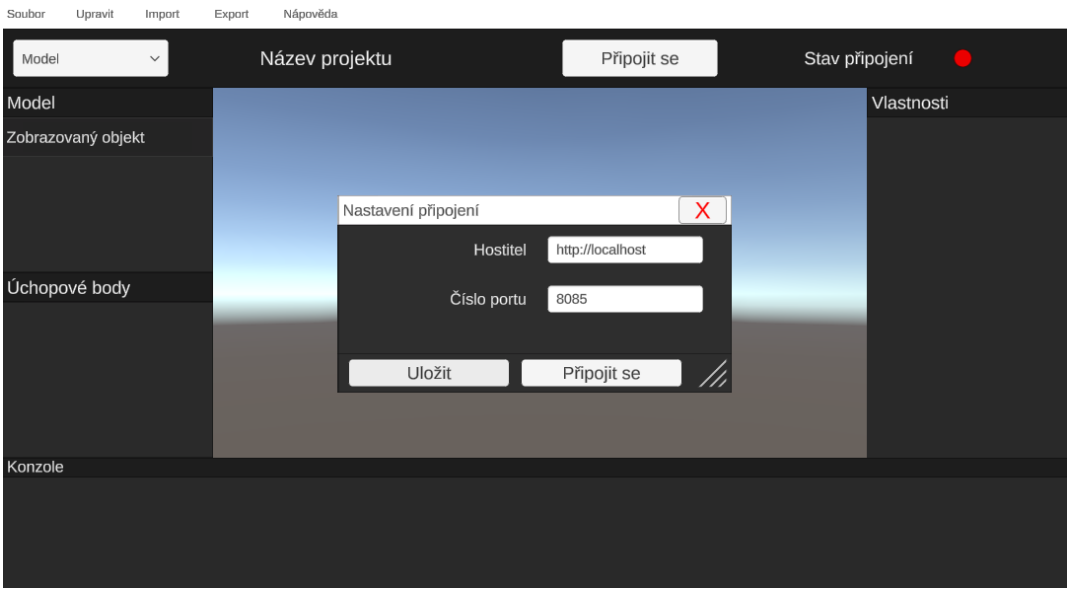
Obrázek 16: Hláška po odpojení od backendu

Aby byla správa komunikace robustnější, byla přidána také obsluha událostí nastávajících při přerušení spojení či neúspěšném pokusu o připojení. V případě problémů s tímto připojením je tak uživatel vždy vizuálně informován prostřednictvím vyskakovacích oken s chybou (obr. 17).



Obrázek 17: Hláška při neúspěšném připojení

Nakonec bylo namísto přímého připojení po stisku tlačítka „Připojit se“ přidáno dialogové okno umožňující zadat parametry pro připojení k backendu. Jako výchozí hodnoty polí v tomto dialogu se nastavily výchozí hodnoty adresy a portu pro připojení k lokálně běžící instanci, které jsou vyobrazeny na obrázku 18.



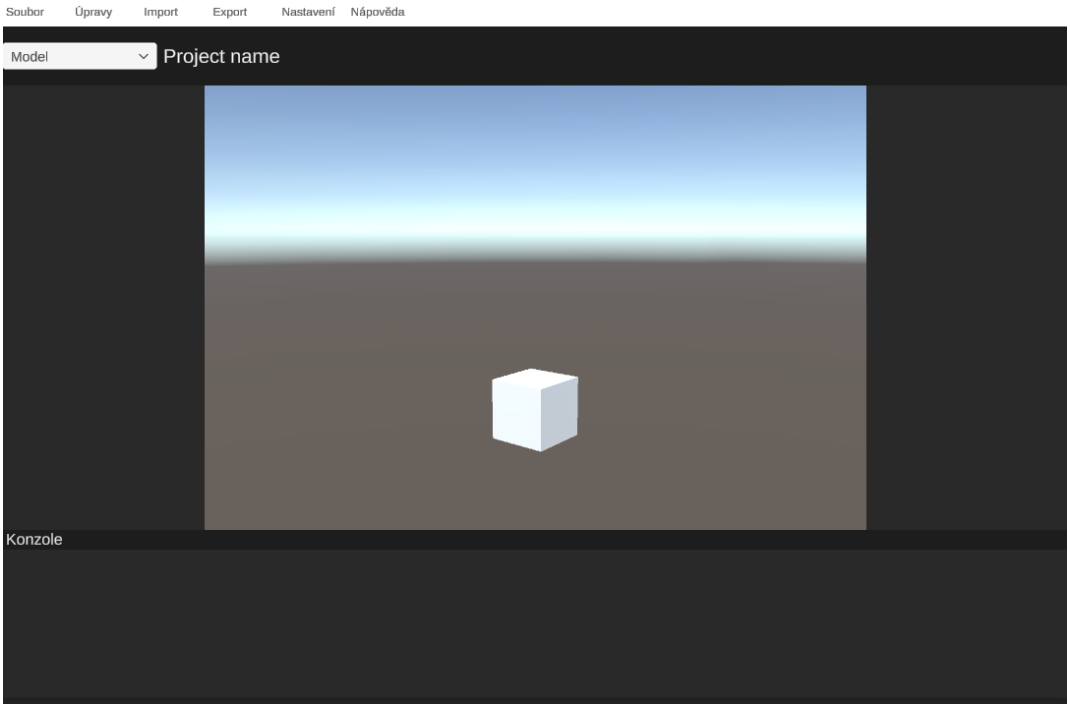
Obrázek 18: Dialog pro připojení k backendu

Díky implementaci výše popsané služební třídy byla implementace pokročilých funkcí využívajících operací implementovaných na backendu vcelku snadná. Jelikož byly tímto krokem potřebné operace zcela odstíněny od zbytku aplikace, neměly změny prováděné v rámci této oblasti vliv na fungování zbytku aplikace.

6.2.7 Implementace správy objektu

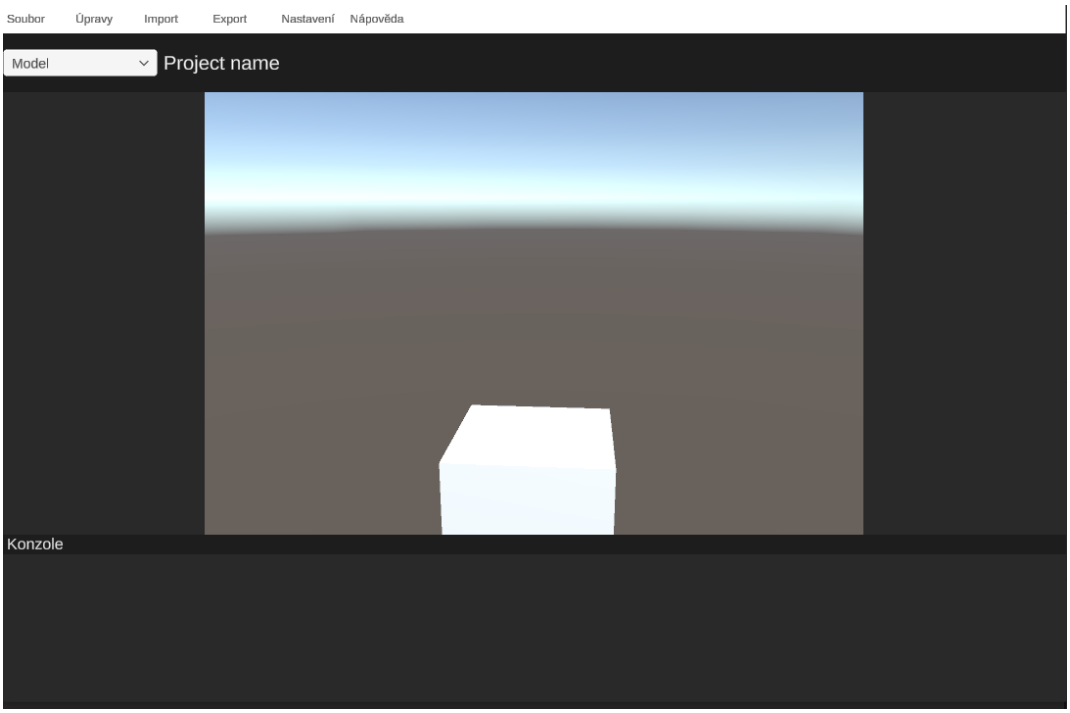
Aby bylo možné později generovat jednotlivé scény, bylo potřeba implementovat systém pro správu objektu a především jeho modelu. Jelikož je hlavním zaměřením správy objektu správa jeho modelu, byly tyto pojmy v rámci aplikace chápány jako synonymum a v následujících částech textu tak bude o tomto objektu též referováno jako o modelu.

V rámci zobrazení modelu byl nejdříve implementován pohyb s kamerou. Otáčení je realizováno tažením myši za držení kolečka myši. Ovládání kamery je implementováno pomocí tvz. gimbalu. Jak uvádí Bradfield, gimbal slouží k udržení rovnováhy objektu při pohybu a lze jej vytvořit pomocí dvou uzlů, kde ten vnější rotuje pouze na ose Y, zatímco vnitřní pouze na ose X [70]. V tomto případě se však vzhledem k odlišným enginům místo uzlů pro implementaci využily herní objekty.



Obrázek 19: Prvotní podoba obrazovky Model

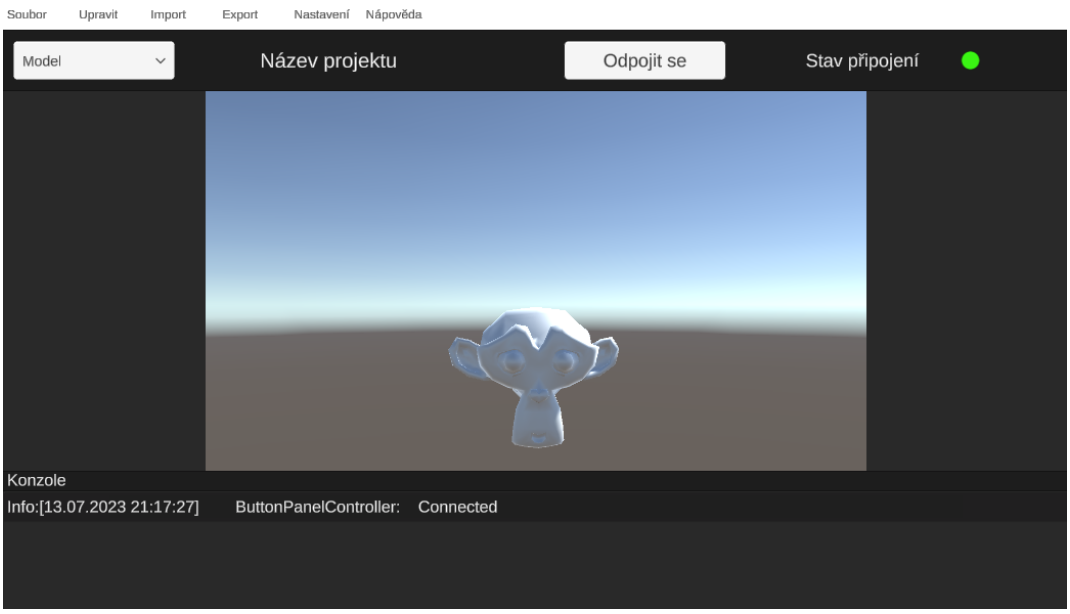
Přiblížení a oddálení pohledu je realizováno otáčením kolečka myši na základě [18, s. Input.mouseScrollDelta]. Obě akce byly implementovány tak, aby šly aktivovat pouze pokud se kurzor nachází nad scénou s objektem a nedocházelo tak k nechtěným interakcím se scénou modelu.



Obrázek 20: Přiblížený objekt v prvotní obrazovce Model

Aby bylo možné zobrazit požadovaný model, byly implementovány dva způsoby pro jeho načtení. První způsob byl implementován přímým načtením z formátu JSON s atributy (viz kap. 6.2.1), které aplikace používá pro reprezentaci příslušného modelu a pro jeho použití tak není nutné připojení k backendu.

Druhý způsob byl implementován jakmile byla vyřešena základní komunikace s backendem (viz kap. 6.2.6) a spočívá v načítání souborů standardních 3D formátů. Získání modelu z daného souboru je delegováno na backend, kterému je tento soubor poslán jako soubor. Ten po zpracování informuje klienta, že je model připravený a klient si jej poté stáhne jako soubor ve formátu JSON a sestaví z něho svou vnitřní reprezentaci modelu podobně jako když jej z tohoto formátu importuje přímo. Příklad takto načteného 3D objektu je zachycen na obrázku 21.

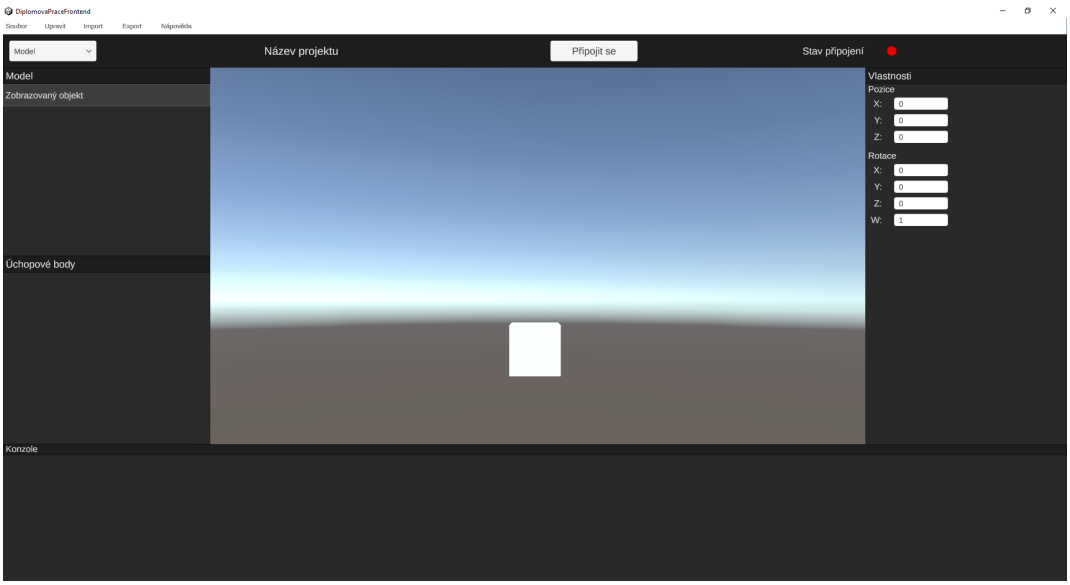


Obrázek 21: Model načtený s pomocí backendu

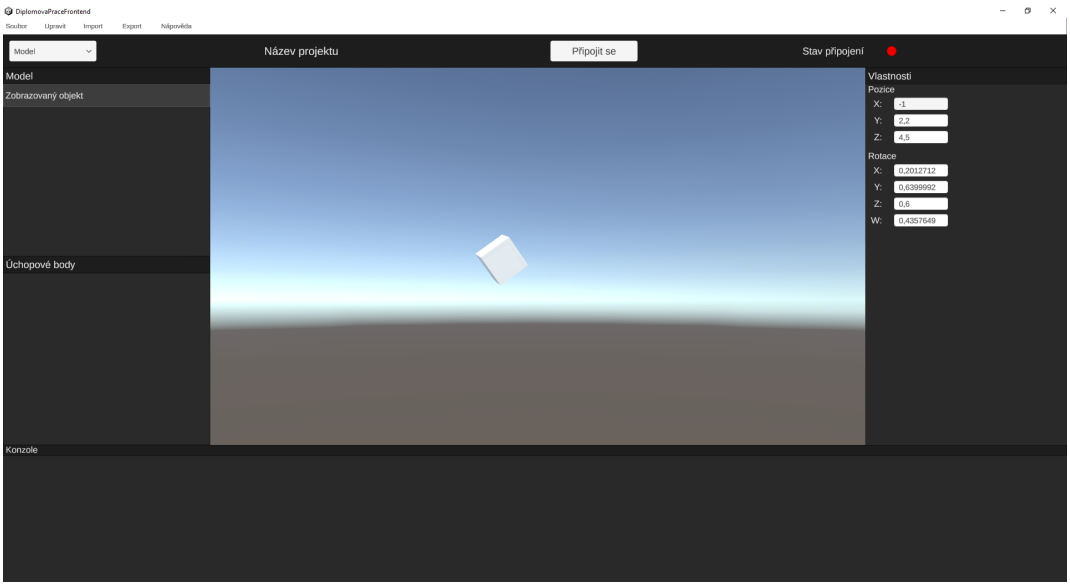
Celý princip pak spočívá v rekonstrukci geometrické sítě na základě vektorů získaných deserializací a indexů vzniknuvších vrcholů pro výsledné plošky. Tato síť je poté také převedena do instance třídy Mesh (viz 3.2), kterou je poté ve scéně nahrazena instance nacházející se zde před načtením nového modelu.

Načtený model je vždy možné vyexportovat do formátu JSON pro pozdější použití nezávisle na typu souboru, ze kterého byl načten. Tento mechanizmus též využívá standardní přístup pro serializaci objektů do tohoto formátu (viz 6.2.1).

Vzhledem ke skutečnosti, že v některých případech může být potřeba změnit zobrazovanému modelu pozici či natočení, byly nakonec přidány také ovládací prvky pro úpravu těchto vlastností (viz obr. 22 a 23). Tyto prvky se nachází pouze v prostoru panelů uživatelského rozhraní a jejich možné budoucí rozšíření může spočívat například v jejich zakomponování do scény samotné či přidání dalších režimů úprav. Pro bližší informace o jejich použití vizte dokumentaci aplikace (viz Příloha A).



Obrázek 22: Prvky pro úpravu transformací modelu



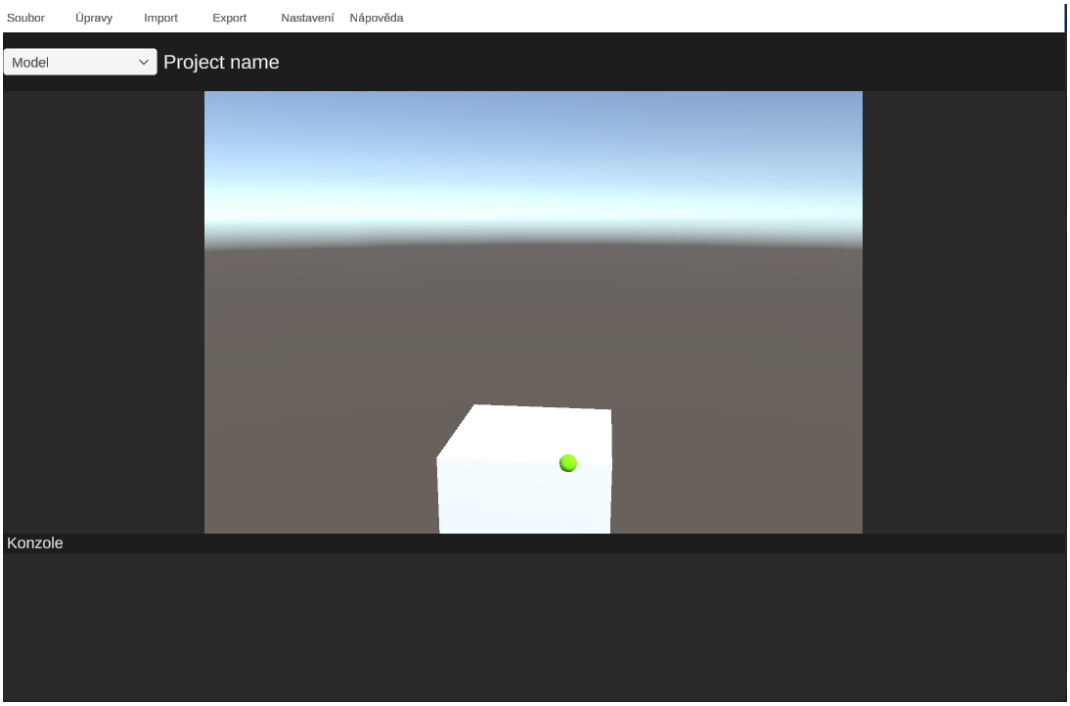
Obrázek 23: Model s upravenými transformacemi

Implementací systému pro správu modelu bylo možné jej zobrazit ve scéně a měnit úhel natočení na tento model. Dále byl tímto splněn jeden z hlavních požadavků na výslednou aplikaci spočívající v možnosti požadovaný model načíst z některého ze standardních formátů používaných pro 3D modely. Pro usnadnění práce s modelem bylo také umožněno upravovat jeho transformace, přičemž se pro tuto funkcionalitu nabízí možná rozšíření.

6.2.8 Implementace správy úchopových bodů

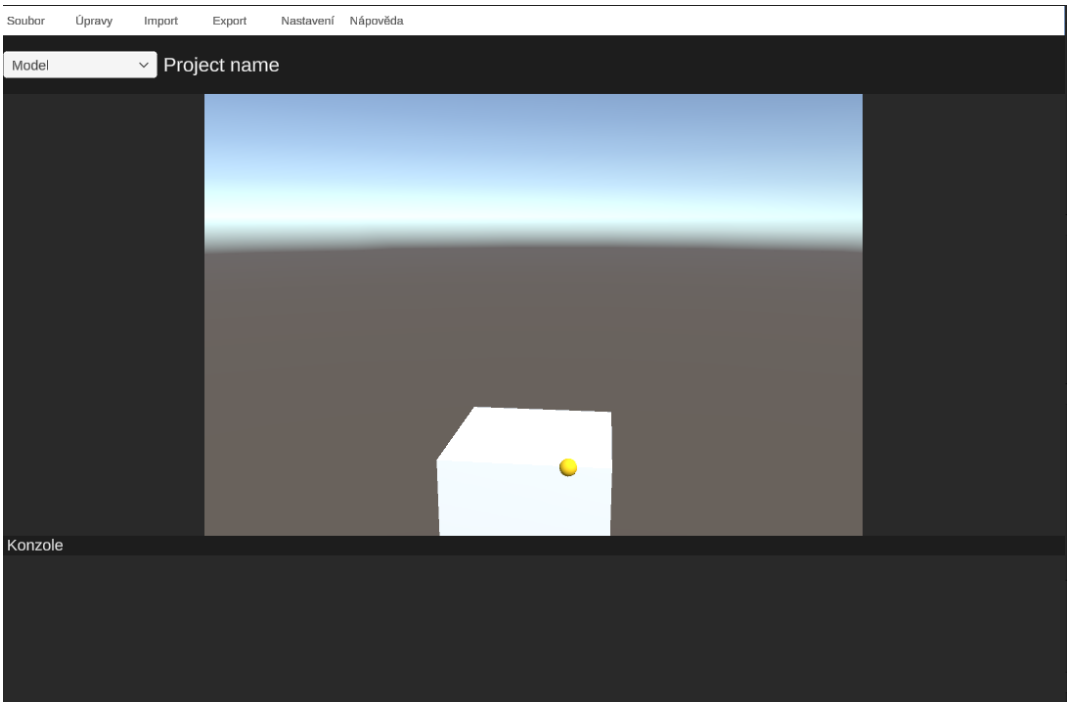
Pro splnění dalšího požadavku na fungování výsledné aplikace bylo potřeba implementovat správu úchopových bodů. V rámci této aplikace byl zvolen přístup, ve kterém uživatel sám zadá, kde se mají jednotlivé úchopové body nacházet.

Nejprve byla implementována základní vizualizace umístění úchopového bodu při najetí na zobrazený model (viz obr. 24). V tomto bodě vývoje byla pouze zkoumána kolize s modelem a nebylo řešeno natočení vytvářených bodů vůči jednotlivým ploškám.



Obrázek 24: Prvotní podoba umisťování úchopového bodu

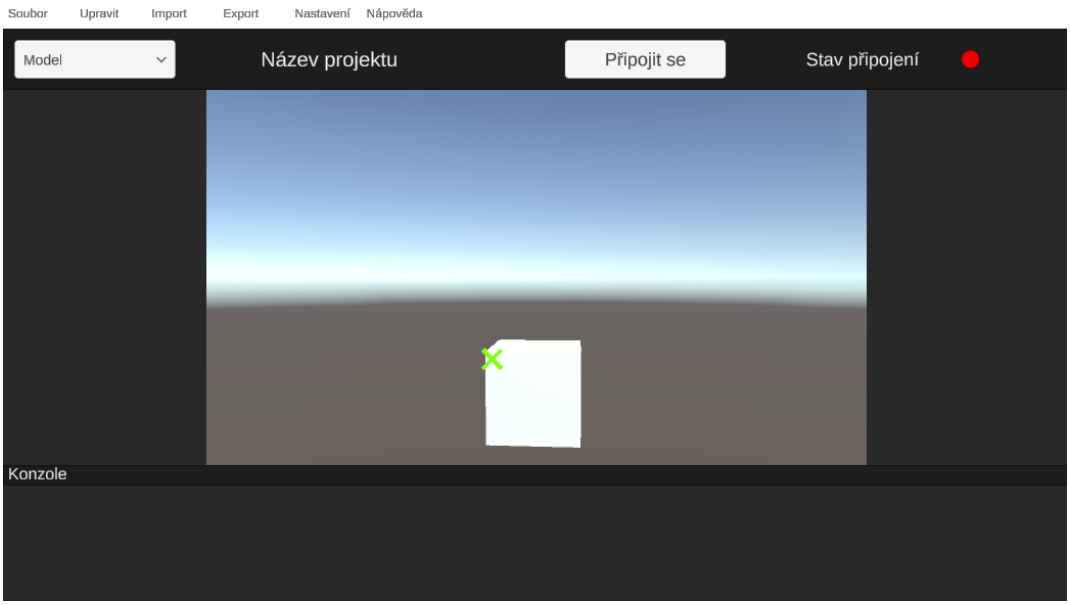
Vytvoření úchopového bodu bylo realizováno kliknutím levého tlačítka myši a jejich základní mazání zde bylo možné pomocí pravého tlačítka myši. Jeho prvotní podoba je zachycena na obrázku 25. V této fázi bylo možné pouze smazat všechny úchopové body a nebyla zde možnost je jednotlivě vybrat, ani s nimi jinak manipulovat.



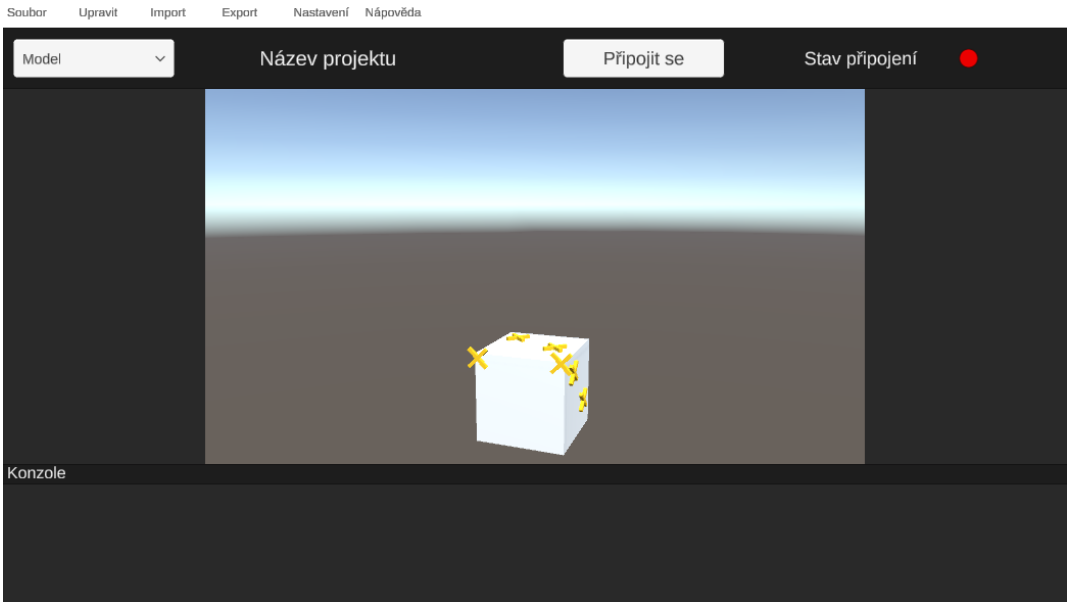
Obrázek 25: Prvotní podoba vytvořeného úchopového bodu

Po implementaci potřebného chování v této základní verzi byla přidána automatická změna natočení úchopového bodu dle příslušného trojúhelníku modelu. Pro získání konkrétního trojúhelníku byl použit index trojúhelníku, který byl zasažen při vyslání paprsku z kamery. Podoba této implementace je vyobrazena na ukázce zdrojového kódu 20.

Aby byl úhel natočení úchopového bodu lépe viditelný, byl pro jeho reprezentaci vytvořen nový model ve tvaru křížku (viz obr. 26 a 27). Tento model byl vytvořen v programu Blender 3.4.0.



Obrázek 26: Vylepšená podoba náhledu úchopového bodu

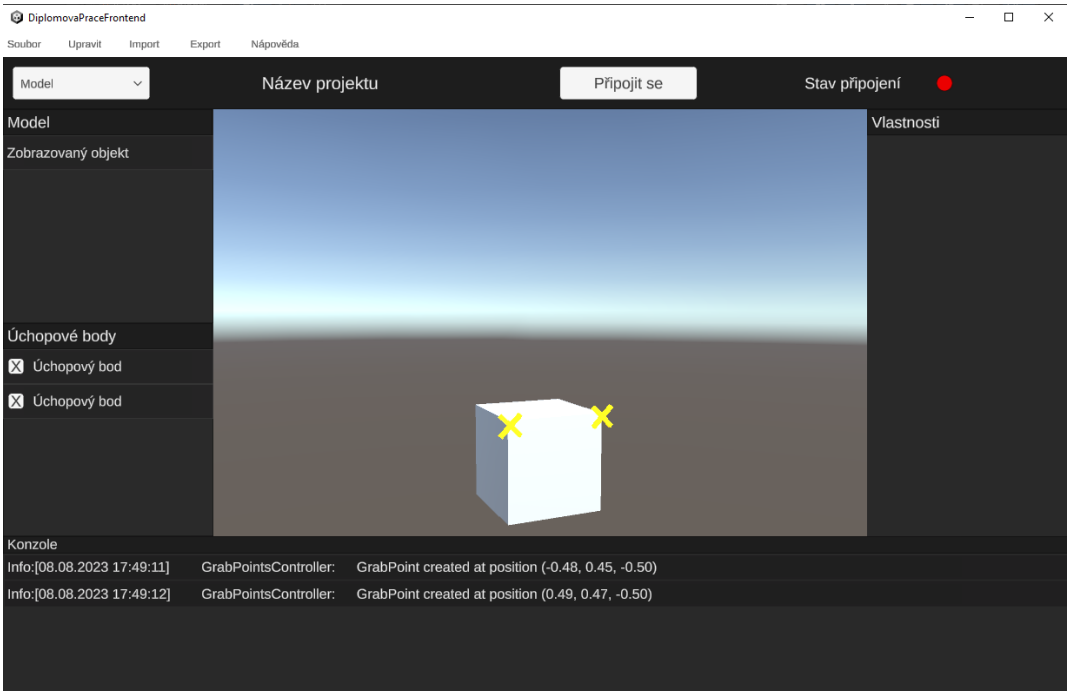


Obrázek 27: Vylepšená podoba umístěného úchopového bodu

```
public GrabPoint CreateGrabPoint()
{
    GrabPoint applicationGrabPoint = null;

    if (Physics.Raycast(_displayCamera.ScreenPointToRay(Input.mousePosition),
        out RaycastHit hitInfo))
    {
        if (hitInfo.collider.gameObject.Equals(_parentObject.gameObject))
        {
            DisplayedGrabPointController grabPoint =
                Instantiate(_grabPointPrefab,
                _grabPointPreview.transform.position,
                    _grabPointPreview.transform.rotation,
                hitInfo.collider.gameObject.transform);

            applicationGrabPoint = new GrabPoint
            {
                GrabPointObject = grabPoint,
                LocalPosition = grabPoint.transform.localPosition,
                LocalRotation = grabPoint.transform.localRotation,
                LocalScale = grabPoint.transform.localScale
            };

            _grabPoints.Add(applicationGrabPoint);
            s_logger.LogInfo($"GrabPoint created at position
                {grabPoint.transform.position}");
        }
    }

    return applicationGrabPoint;
}
```

Zdrojový kód 20: Logika vytvoření úchopového bodu ve scéně. Kód vytvořen podle: [18, s. RaycastHit.triangleIndex]

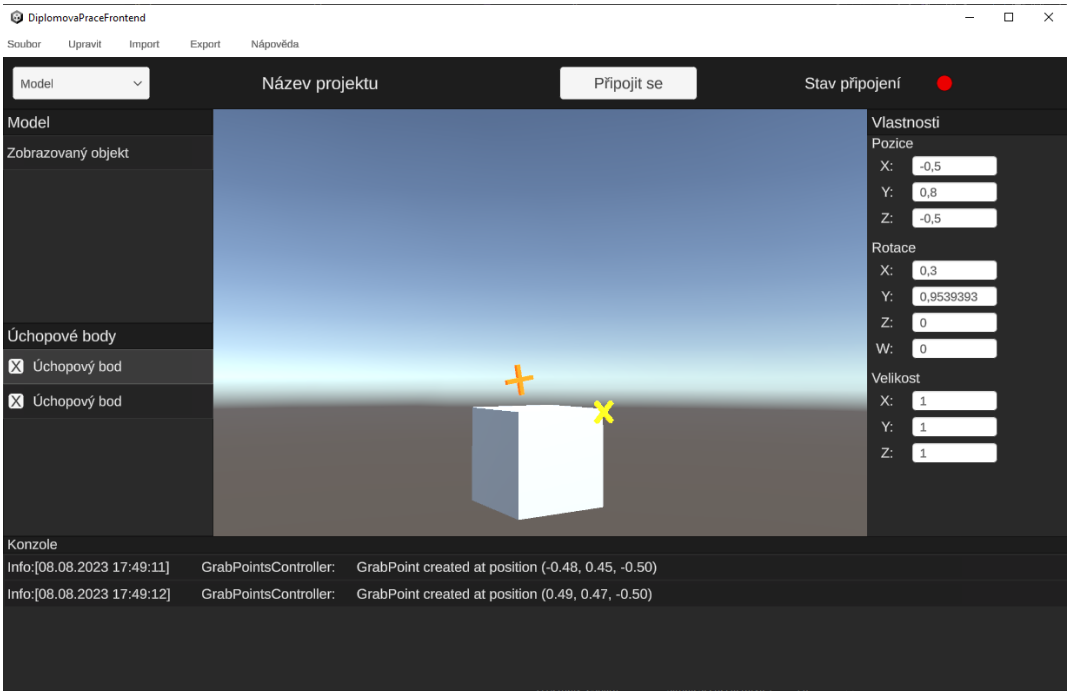
Pro řádné splnění požadavku na možnost správy úchopových bodů byl poté v uživatelském rozhraní implementován jejich seznam, který se automaticky synchronizuje se stavem ve scéně s modelem (viz obr. 28). S pomocí tohoto seznamu již bylo možné jednotlivé úchopové body vybírat a mazat je stisknutím tlačítka křížku v levé části dané položky.



Obrázek 28: Úchopové body v seznamu

Jakmile byla implementována možnost výběru jednotlivých úchopových bodů, přidala se také možnost je dále upravovat (viz obr. 29). Implementované možnosti spočívají v úpravě pozice, rotace a velikosti v lokálních jednotkách a lze tak poopravit případné chyby při jejich vytváření.

Vytvořený systém je možné dále rozšířit například o možnosti změny modelu jednotlivých úchopových bodů či výběru mezi lokálními a globálními souřadnicemi při úpravách transformací. Další možností je umožnit jejich pojmenování pro zvýšení přehlednosti při jejich zobrazování v seznamu.



Obrázek 29: Vybraný a modifikovaný úchopový bod

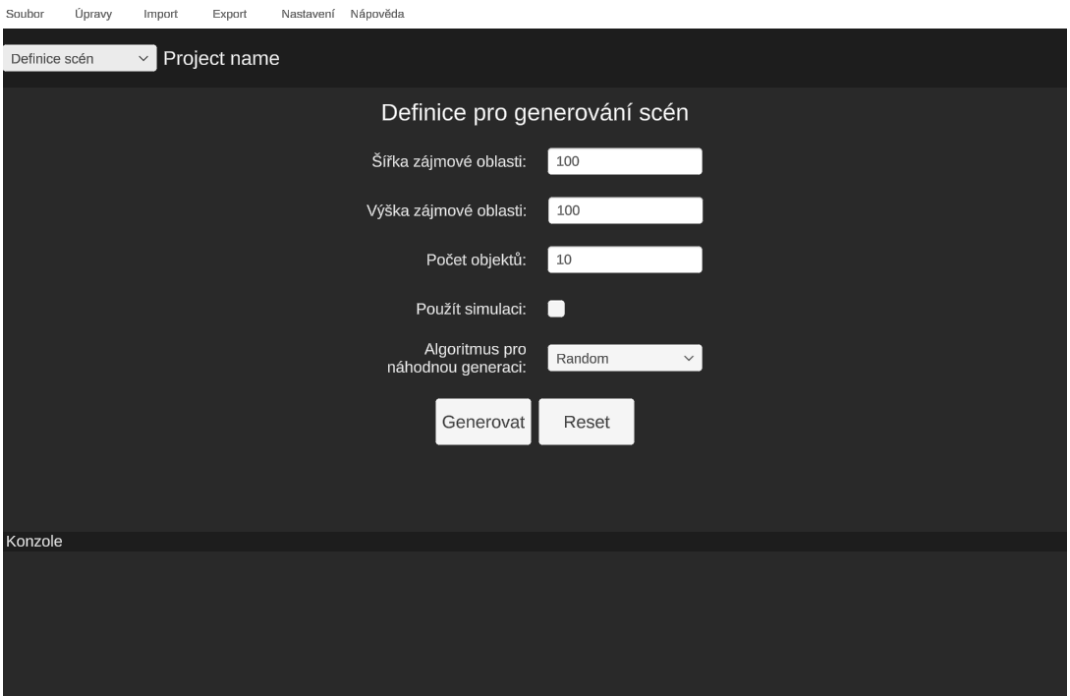
Provedením výše popsané implementace byl splněn požadavek aplikace na správu úchopových bodů. Současný systém umožňuje jejich výběr, úpravy a mazání, přičemž případná budoucí rozšíření mohou spočívat především v přidání dalších možností úprav.

6.2.9 Implementace správy definic scén

Dalším z požadavků na výslednou aplikaci byla možnost zadání parametrů pro generaci scén. V rámci této práce byla tato funkcionalita pojata odděleně od samotné správy scén včetně jejich generace, které jsou popsány v kapitole 6.2.10.

Nejprve byl vytvořen základní formulář pro definici scén, který byl umístěn do samostatného režimu hlavní obrazovky (viz obr. 30). Zadávání číselných hodnot bylo realizováno pomocí vstupních polí nastavených na přijímání celých čísel (viz kap. 6.2.1). Formulář již v tomto bodě také umožňoval vyresetovat změny do stavu před započetím úprav.

Podporované parametry v této fázi zahrnovaly rozměry zájmové oblasti, počet objektů ve scéně, algoritmus použitý pro generaci náhodných pozic a informace, zda se má při generaci použít fyzikální simulace. Dále bylo možné vybrat z algoritmu se zcela náhodným rozmístěním a algoritmu Poissonův disk. Počet vygenerovaných scén byl v této fázi z testovacích důvodů pevně nastaven na 10.



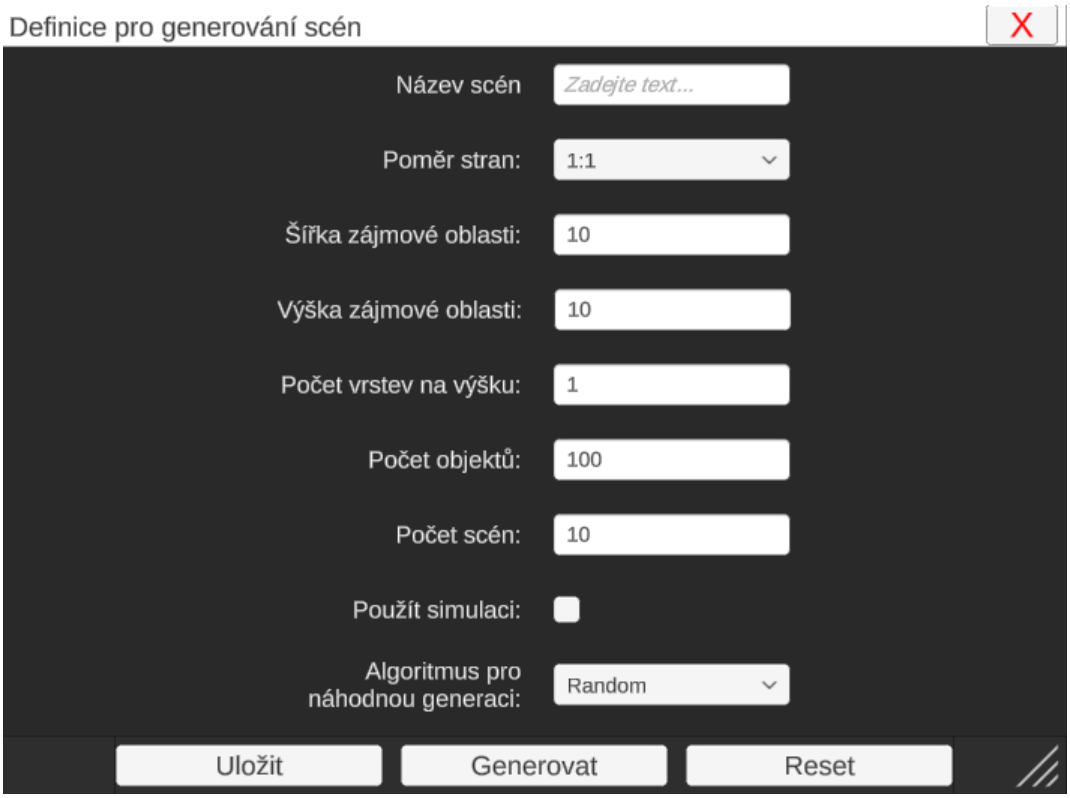
Obrázek 30: Prvotní podoba formuláře pro definici scén

Ačkoliv se krok definice parametrů pro generování scén nachází v celém procesu mezi definicí úchopových bodů a samotným vygenerováním scén, výše popsané umístění se nezdálo jako plně vhodné. Z tohoto důvodu byl vytvořený formulář nakonec přesunut do samostatného vyskakovacího okna, které lze vyvolat v liště s menu pod nabídkou Upravit > Definovat scény.

Zvolený přístup tak umožnil snadno generovat nové scény bez nutnosti přepínání aktivního režimu obrazovky. V průběhu následujícího vývoje se tato skutečnost ukázala jako značně výhodná, neboť při aktivním náhledu na dříve vygenerované scény bylo snazší lépe definovat nové parametry pro příští generaci.

Formulář byl poté rozšířen o možnost zadání společné části názvu vygenerovaných scén, určení poměru stran zájmové oblasti, počet vertikálních vrstev a počet vygenerovaných scén (viz obr. 31). Výběr poměru stran byl nastaven tak, aby umožňoval výběr mezi poměry 1:1, 16:9 a 4:3, přičemž rozměry zájmové oblasti se tomuto poměru vždy přizpůsobí.

Maximální a minimální možné hodnoty polí pro velikost zájmové oblasti a počet objektů se též začaly vzájemně respektovat. Minimální velikost zájmové oblasti se tak začala odvozovat jak od příslušné hodnoty v poměru stran, tak od ohraničujícího kvádru daného modelu (viz [18, s. Bounds]). Maximální hodnota počtu objektů je pak závislá na počtu



Obrázek 31: Vylepšená podoba formuláře pro definici scén

ohraničujících kvádrů modelu, které se vejdou do kvádru definovaného rozměry zájmové oblasti a počtem vertikálních vrstev.

V neposlední řadě byla také přidána možnost uložení nastavených hodnot bez nutnosti generace scén. Zároveň byly skrze skriptovací objekt (viz kap. 6.2.1) nastaveny jeho výchozí hodnoty. Formulář pro definici parametrů potřebných pro následnou generaci scény byl tudíž hotov a vzhledem k oddělení od samotné generace scén bylo usnadněno jeho rozšiřování již během samotného vývoje.

Mezi možná budoucí rozšíření tohoto formuláře patří například možnost zadání dynamického poměru stran zájmové oblasti. Další možností je rozšíření podporovaných algoritmů pro náhodnou generaci pozic ve scéně.

Provedením výše popsaných kroků byl splněn požadavek aplikace pro možnost definice parametrů pro generování scén. Vyjma definice velikosti zájmové oblasti a jména scén bylo také umožněno definovat výšku vertikálních vrstev dané scény, poměr stran zájmové oblasti, konkrétní algoritmus, který má být pro generaci využit, počet objektů ve scéně a také počet scén, které mají být vygenerovány. Případná budoucí rozšíření mohou spočí-

vat například v rozšíření o další algoritmy či přidání podpory dynamického poměru stran zájmové oblasti.

6.2.10 Implementace správy a vykreslování scén

Aby bylo možné vytvořit konečné datové sady, bylo potřeba implementovat systém pro generaci a následnou správu scén včetně možnosti jejich vykreslení do souboru. V této podkapitole je popsán proces implementace příslušných funkcionalit.

Nejprve bylo potřeba implementovat logiku pro generování scén dle parametrů zadaných uživatelem. Pro potřeby základní generace prováděné přímo v aplikaci byla vytvořena třída SimpleSceneFactory, která dle parametrů náhodně generuje pozice a rotace pro model. Tyto hodnoty poté aplikuje na úchopové body.

Náhodná generace zde byla implementována pomocí metody Range a vlastnosti rotationUniform ve třídě Random. Jak je uvedeno v dokumentaci Unity, třída Random je statická a lze s ní snadno generovat pseudonáhodná čísla. Její využití s vícero vlákny je však problematické, jelikož použití mimo hlavní vlákno skončí chybou. [18, s. Random]

Tento způsob generace tak byl při dalším vývoji pojat jako pouze velice základní, jelikož skrze něho nebylo možné využít při generaci vícero vláken. Další nevýhodou této základní implementace zůstalo nerespektování kolizí modelu při generaci, takže výsledné scény obsahují mnoho vzájemného prolínání modelů.

Řešení těchto problémů bylo delegováno na logiku implementovanou na straně backendu (viz kap. 6.1.2), což umožnilo dle potřeby využívat při generaci fyzikální simulace a snížit vytížení grafické části aplikace. Možné rozšíření této funkcionality může spočívat v rozšíření nabídky algoritmů pro generaci náhodných transformací.

Pro oba výše zmíněné způsoby je však princip stejný a spočívá v náhodné generaci pozic a rotací dle zvoleného algoritmu a dalších parametrů, po které se dopočítají transformační matice definovaných úchopových bodů. Tyto transformační matice jsou zde vyjádřeny pomocí struktury Matrix4x4. Jak je uvedeno v dokumentaci Unity, jedná se o transformační matici schopnou pomocí homogenních souřadnic provést lineární 3D transformace. [18, s. Matrix4x4]

Vzhledem k různým možným poměrům stran zájmové oblasti je poté vypočítána také pozice kamery ve 3D prostoru tak, aby celou scénu zachytila. Při určování této pozice byla mimo jiné snaha o to, aby se kamera nacházela ve středu zájmové oblasti a byla umístěna v dostatečné výšce. Každá scéna si tak informace o pozici kamery drží pro vlastní potřebu.

Generování scén dle zadaných parametrů bylo provedením těchto kroků podporováno pro stanovený počet scén, přičemž po vygenerování scén jsou vždy jakékoliv dříve vygenerované zahozeny. Možným budoucím rozšířením tak může být přidání možnosti generace jedné scény bez ztráty ostatních.

Jakmile bylo možné scény vygenerovat, bylo potřeba je také nějakým způsobem zobrazit. Naivní způsob by mohl spočívat ve vytváření objektů ve scéně v Unity pro každou jednotlivou transformaci modelu a úchopových bodů pro každou jednotlivou vygenerovanou scénu. Takový způsob by však byl značně neefektivní a výpočetně náročný.

Namísto toho byl zvolen přístup, kdy jsou pro reprezentaci vygenerovaných scén použity dva objekty ve scéně v Unity, kde jeden slouží pro reprezentaci modelu a druhý pro reprezentaci úchopových bodů. Oba objekty poté takto slouží zároveň pro všechny vygenerované scény.

Tento způsob vykreslování byl realizován bez komponenty MeshRenderer (viz kap.3.2) a namísto toho využíval hodnoty ve vygenerovaných strukturách Matrix4x4 a metodu DrawMeshInstanced ve třídě Graphics (viz zdrojový kód 21). Jak uvádí Monese, DrawMesh vykreslí pro jeden snímek síť včetně světla a stínů bez dodatečných nároků na správu herních objektů a hodí se tak k vykreslení velkého množství sítí. Při použití DrawMeshInstanced je pak využito GPU instancování a na vstupu je potřeba dodat informace o transformacích v poli Matrix4x4. [71]

Pro vykreslení příslušné scény tak s tímto způsobem stačilo pouze aktualizovat příslušná pole Matrix4x4, která již byla vygenerována při samotné generaci scén. Jak se dále ukázalo, není zároveň nutné celou scénu neustále snímat při jejím zobrazení, a tak je v konečné implementaci tato metoda pro vykreslování ve většině případů volána na dobu jednoho snímku, pokud není požadováno jinak.

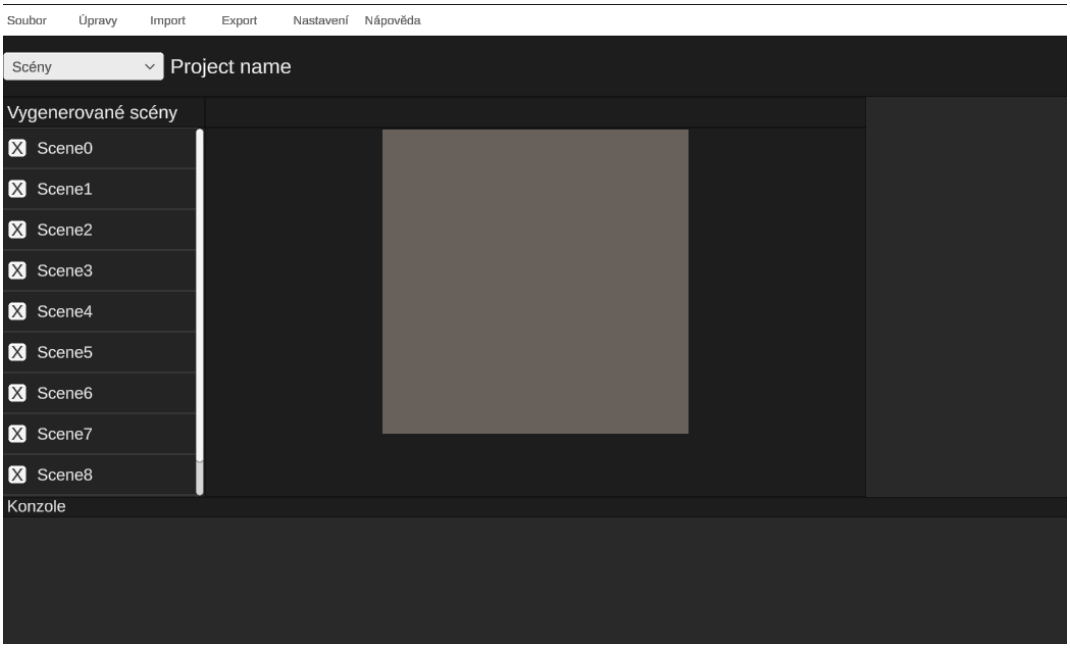
```
public class BulkRenderObject : MonoBehaviour
{
    ...
    public void DisplayScene()
    {
        if (_matrices != null)
        {
            Graphics.DrawMeshInstanced(_mesh, 0, _material, _matrices);
        }
    }

    void Update()
    {
        if (DisplayPeriodically && _matrices != null)
        {
            Graphics.DrawMeshInstanced(_mesh, 0, _material, _matrices);
        }
    }
    ...
}
```

Zdrojový kód 21: Třída pro hromadné vykreslení. Kód vytvořen podle: [71]

Po implementaci mechanizmů pro vykreslení vygenerovaných scén bylo vytvořeno základní rozložení obrazovky pro správu scén (viz obr. 32). Nejprve bylo pro toto rozložení nutné vytvořit komponentu umožňující vygenerované scény zobrazit v seznamu.

Jednotlivé položky zde byly implementovány tak, aby zde scény byly zobrazeny s jejich jménem a tlačítkem pro jejich smazání. Samotné akce mazání a výběru poté využívají systému událostí (viz kap. 6.2.1), čímž byla snížena míra provázání grafického rozhraní a samotné logiky správy scén.

Poté byla implementována logika pro samotné zobrazení náhledu dané scény v tomto rozhraní do textury umístěné v jeho středu, které se provede po výběru příslušné položky v seznamu. Tento systém byl implementován tak, že se do grafického rozhraní umístila komponenta Raw Image, které je přiřazena reference na příslušnou texturu. Jak uvádí



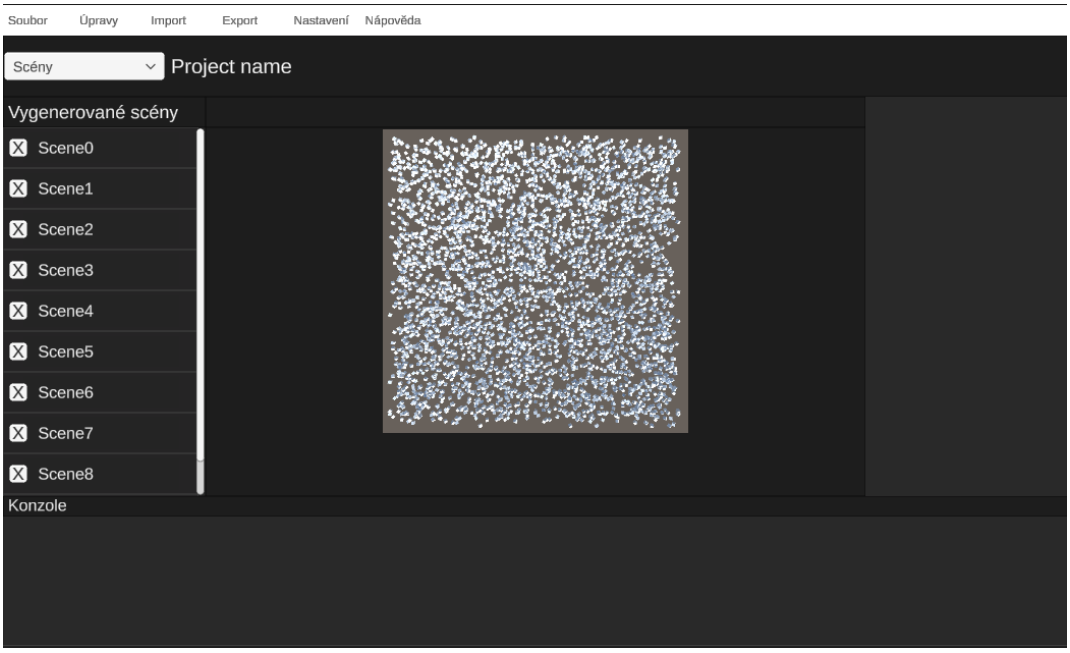
Obrázek 32: Prvotní podoba obrazovky pro zobrazení scén

dokumentace Unity UI, obrázek zobrazovaný v Raw Image může být libovolná textura, kterou lze ve skriptu vyměnit. [19, s. Raw Image]

Výše zmíněná textura je typu Render Texture, aby ji bylo snadné aktualizovat. Jak uvádí dokumentace Unity, jedná se o texturu aktualizovanou za běhu aplikace. Pro její použití v materiálech ji stačí vytvořit a přiřadit do Target Texture v komponentě kamery [17, s. Render Texture]. Její aktualizace dle požadované scény se tak ukázala jako triviální.

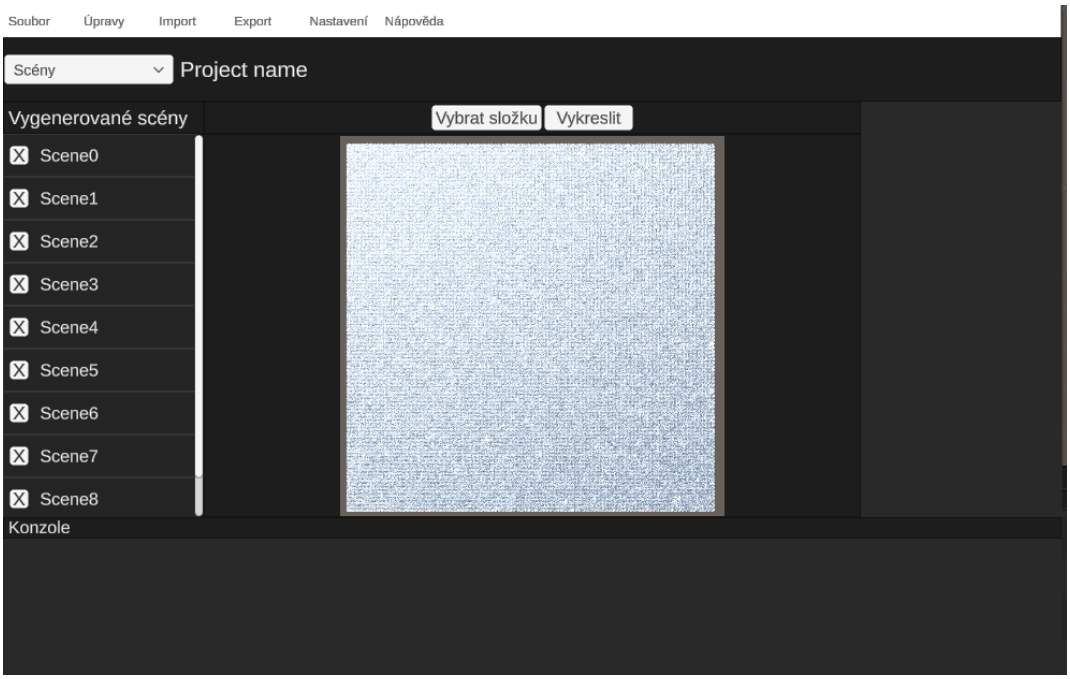
Pro samotné vykreslení je použita odlišná kamera od té používané pro náhled modelu a této kameře se tudíž předá do hodnoty Target Texture reference na texturu popsanou výše. Poté se vyvolá požadavek na vykreslení výše popsaných objektů a kamery, která mimo specifické situace zůstává neaktivní.

V této fázi se tak pouze aktualizovala zobrazovaná scéna pomocí mechanismů popsaných výše. O vybrané scéně nebylo možné zobrazit další informace, ani ji nebylo možné jakkoliv modifikovat s výjimkou jejího smazání (viz obr. 33).



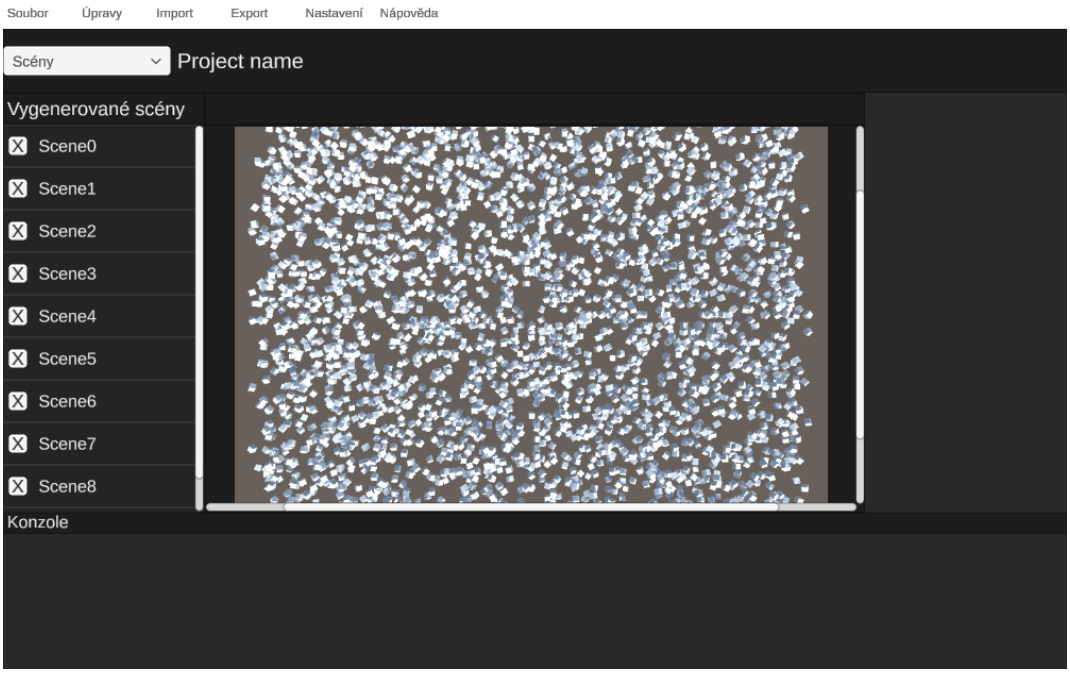
Obrázek 33: Prvotní podoba vybrané scény

Jak se později potvrdilo, zvolené řešení umožnilo preciznější správu systémových zdrojů jak při zobrazování náhledů scén, tak při samotném vykreslení do souboru. Jelikož je v daném okamžiku vždy vykreslen pouze ten počet objektů definovaný v dané scéně, a to pouze na jediný snímek při požadavku na zobrazení scény, jsou značně šetřeny systémové zdroje. Na obrázku 34 je ukázka náhledu scény s milionem náhodně rozmístěných modelů kostek, jejíž vykreslení naivním způsobem by bylo značně náročné.



Obrázek 34: Náhled scény s milionem kostek

V neposlední řadě bylo v této fázi implementováno také přiblížení na zobrazovaný náhled scény. Ten byl realizován přes otáčení kolečkem myši a umožnil tak snadný způsob jak blíže prozkoumat detaily scény. Nebylo však možné toto přiblížení jednoduše resetovat.



Obrázek 35: Přiblížení vybrané scény

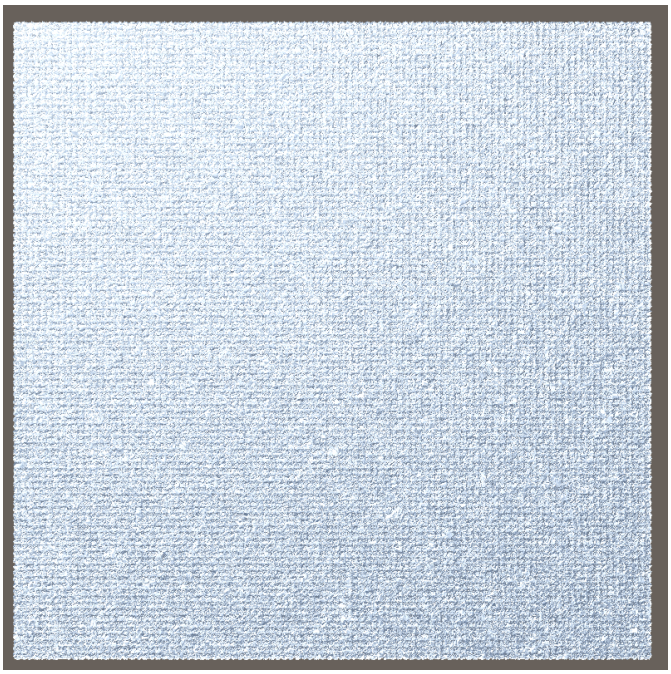
Jakmile byly implementovány základní funktionality vykreslování scén, implementovalo se jejich základní ukládání do souboru. Toto ukládání bylo realizováno pomocí metody EncodeToPNG ve třídě ImageConversion (viz zdrojový kód 22). Jak je uvedeno v dokumentaci Unity, tato metoda zakóduje předanou texturu a vrátí pole bytů, které je poté možné uložit do souboru PNG. [18, s. ImageConversion.EncodeToPNG]

```
private void RenderSceneToFile(int counter, string namePrefix)
{
    Texture2D render = new(_renderTexture.width, _renderTexture.height);
    RenderTexture.active = _renderTexture;
    render.ReadPixels(new Rect(0, 0, _renderTexture.width,
        _renderTexture.height), 0, 0);
    render.Apply();
    RenderTexture.active = null;

    byte[] bytes = ImageConversion.EncodeToPNG(render);
    Destroy(render);

    s_logger.LogDebug($"Saving image to
        {_projectContextService.ProjectContext.RenderPath}" +
        $"/{namePrefix}_{counter}.png");
    File.WriteAllBytes($"{_projectContextService.ProjectContext.RenderPath}" +
        $"/{namePrefix}{counter}.png", bytes);
}
```

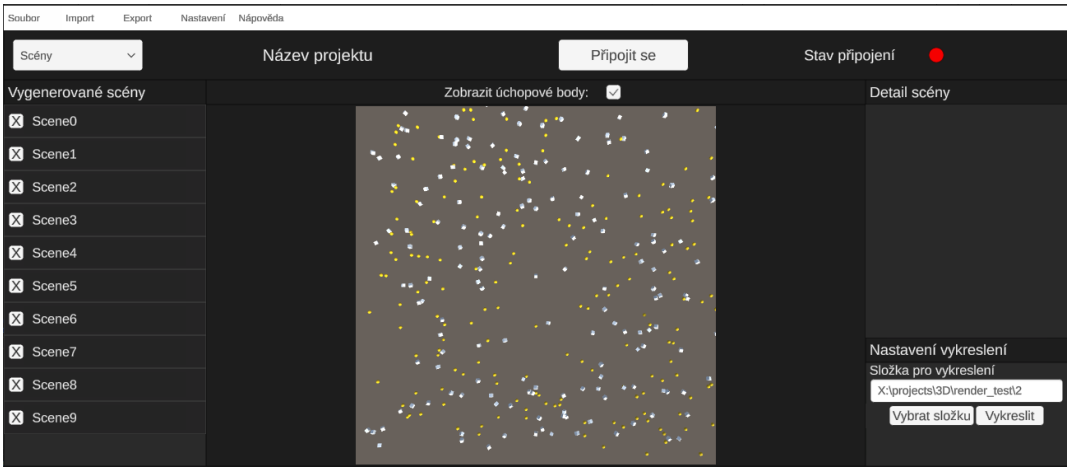
Zdrojový kód 22: Metoda pro vykreslení scény do souboru. Kód vytvořen podle: [18, s. ImageConversion.EncodeToPNG]

Příklad scény s milionem kostek vykreslených do souboru je uveden na obrázku 36. V tomto bodě se však stále jednalo pouze o možnost s pevně daným jménem a bez úchopových bodů, jejichž implementace v rámci generace ještě nebyla dokončena.

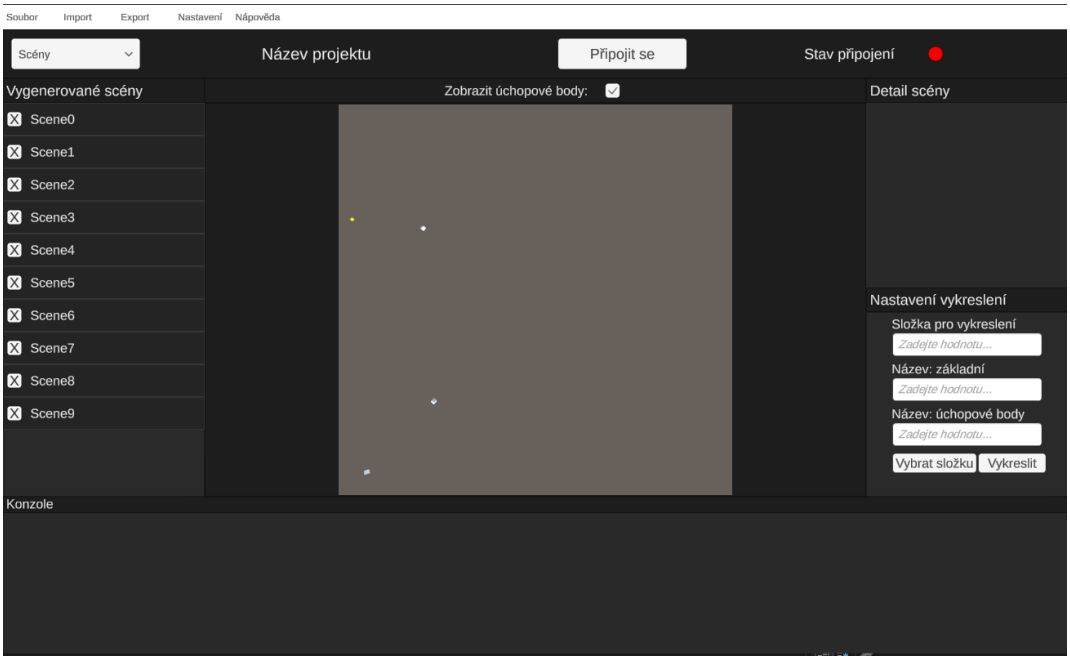
V pozdější fázi vývoje byla tato obrazovka vylepšena o možnost zobrazení či skrytí úchopových bodů pomocí zaškrtávacího pole (viz obr. 37). Dále byla nabídka pro výběr složky a vykreslení přesunuta do pravé spodní části obrazovky a následně rozšířena o textové pole umožňující zadat cestu k výstupní složce. Toto nastavení bylo poté dále rozšířeno o možnost zadání názvu pro snímky s a bez úchopových bodů (viz obr. 38).



Obrázek 36: Vykreslená scéna s milionem kostek



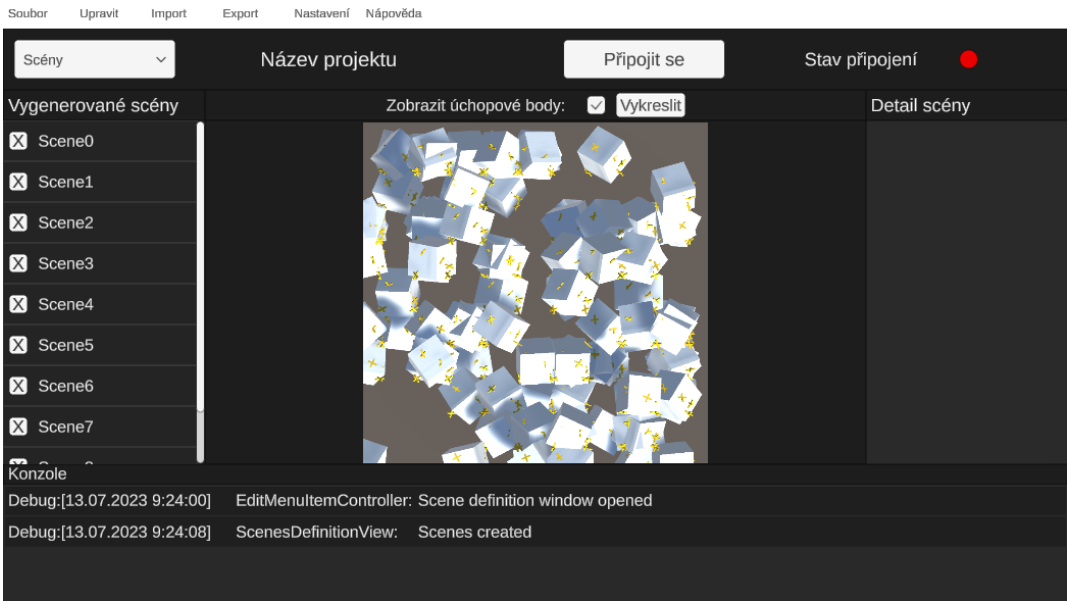
Obrázek 37: Náhled scény s úchopovými body



Obrázek 38: Rozšířená nabídka vykreslení snímků

Jak dokládá obrázek 39, také samotné vykreslování náhledu scén doznalo v tomto bodě vývoje vylepšení. Naklikané úchopové body již byly respektovány a zobrazovány ve scéně. Jejich rotace vůči rodičovskému objektu však byla dořešena později.

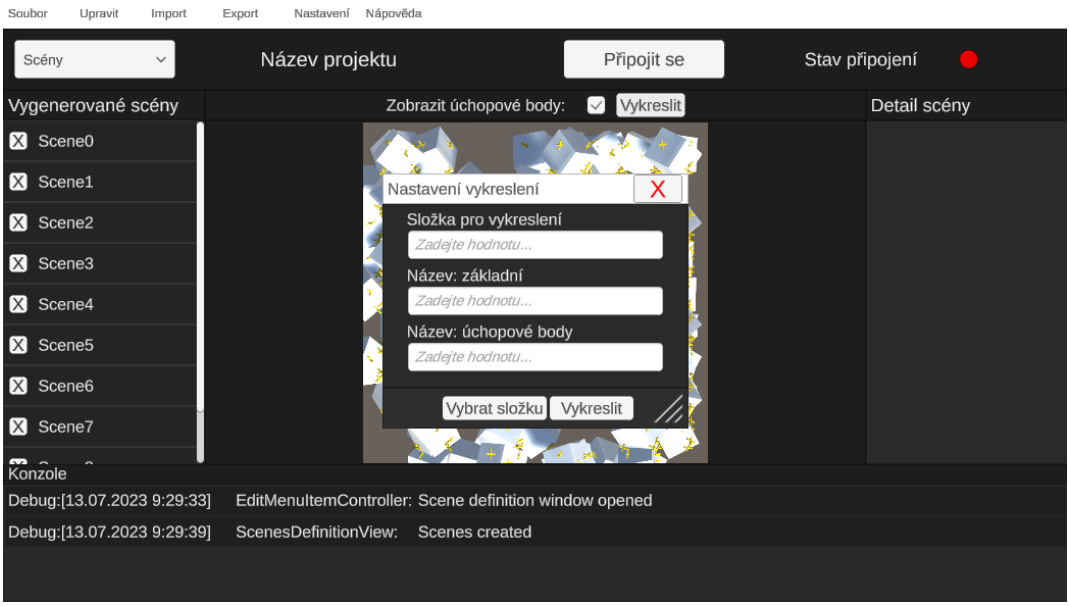
Dále bylo pozměněno nastavení kamery, aby respektovalo nastavené rozměry zájmové oblasti. V této fázi ještě nebyl implementován výše popsaný způsob určení pozice kamery, a tak bylo pozicování implementováno pouze pro izometrický režim kamery. Tento přístup však byl nakonec opuštěn a v konečné implementaci je použit pouze perspektivní režim.



Obrázek 39: Vylepšené vykreslení scén s úchopvými body

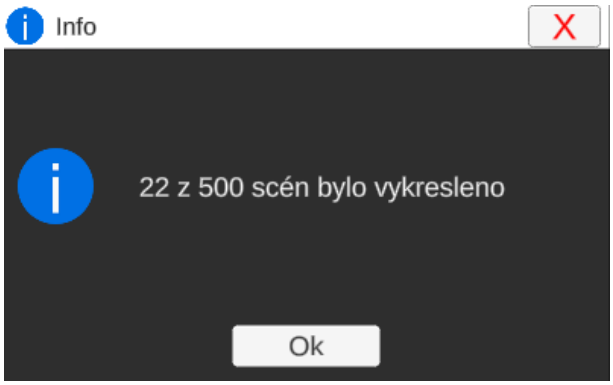
Jak je z obrázku 38 patrné, v případě parametrů pro vykreslení do souboru se z hlediska případného budoucího rozšíření nejednalo o zcela optimální umístění. Z tohoto důvodu bylo toto nastavení později přesunuto do samostatného vyskakovacího okna (viz kap. 6.2.3), které je vyobrazené na obrázku 40. Díky zvolenému způsobu oddělení grafických komponent a způsobu komunikace mezi nimi (viz kap. 6.2.1) byl tento přesun proveden bez úpravy logiky na dříve vytvořeném formuláři pro nastavení vykreslení.

Zároveň bylo přidáno zobrazování vyskakovacího okna načítání, aby byl uživatel informován o probíhající operaci. Vzhledem ke skutečnosti, že proces vykreslování scén do souboru může být zdlouhavý, byla přidána možnost tuto probíhající operaci přerušit klávesovou zkratkou Ctrl+C. Po ukončení této operace se pak vždy zobrazí dialog informující o počtu vykreslených scén (viz obr. 41). Výsledné snímky jsou vždy vykreslovány v rozlišení 1920



Obrázek 40: Nabídka vykreslení snímků v samostatném okně

pixelů na šířku. V rámci budoucích rozšíření by však mohlo být umožněno uživateli zvolit jiné rozlišení.



Obrázek 41: Dialog po konci akce vykreslení scén

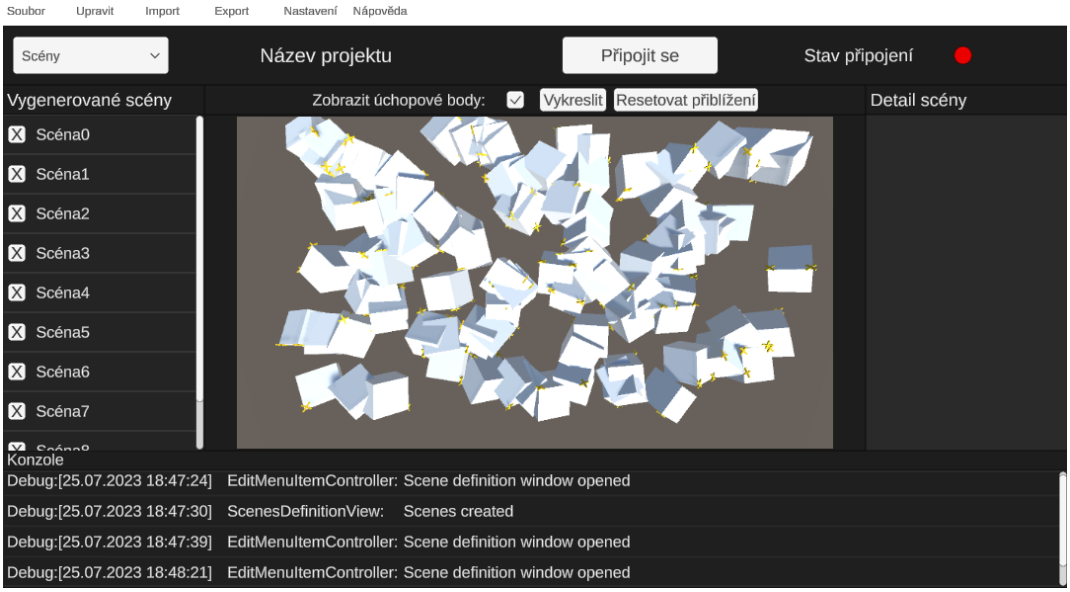
Jak již bylo nastíněno v kapitole 6.2.9, generace scén byla implementována tak, aby podporovala tři typy poměru stran. V předešlých fázích vývoje se však pracovalo pouze s poměrem stran 1:1 a tuto funkcionalitu tak bylo potřeba implementovat. K tomuto účelu bylo v projektu vytvořeno 6 textur s příslušnými poměry stran. V rámci každé dvojice se tak nacházela textura s rozlišením 800 pixelů na šířku, která byla určena pro vykreslení náhledu v aplikaci a textura s rozlišením 1920 pixelů na šířku pro vykreslení do souboru.

Jelikož byly jednotlivé možnosti pro poměr stran implementovány pomocí vytvořené třídy Enumeration (viz kap. 6.2.1), bylo možné do nich vložit mimo jiné i samotné textury.

Nešlo je zde však vložit přímo, a tak byly k tomuto účelu využity skriptovací objekty (viz kap. 3.2). Do těchto objektů byly vloženy reference na páry textur dle poměru stran a samotné objekty poté byly vloženy do složky Resources.

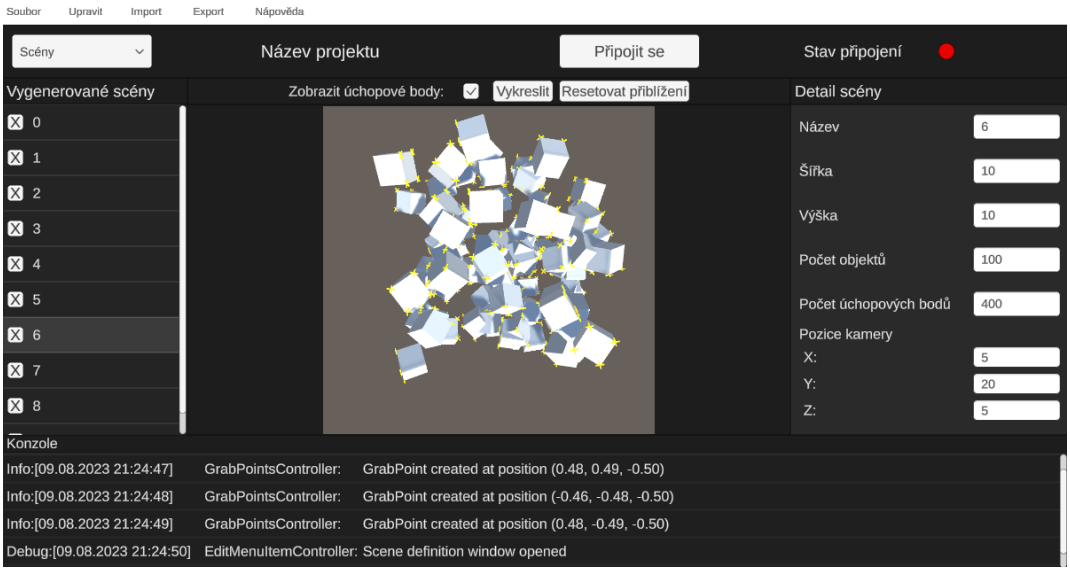
Jak je uvedeno v dokumentaci Unity, pomocí Resources.Load lze přistoupit k veškerým assetům ve složce Resources [18, s. Resources]. Tyto objekty tudíž bylo možné ve třídě AspectRatio, dědící ze třídy Enumeration, načíst pomocí této metody.

Jelikož se mezi data uchovávaná o projektu umístila také informace o zvoleném poměru stran, bylo snadné implementovat následnou výměnu příslušných textur (viz obr. 42). V případě náhledu scény tento proces spočíval v prosté aktualizaci hodnoty Target Texture vykreslující kamery a obrázku v komponentě RawImage, kterému se poté také upravil poměr stran. Při vykreslování do souborů pak pouze stačilo převzít příslušnou texturu a vložit ji do Target Texture v komponentě kamery, která vytvořené scény vykresluje.

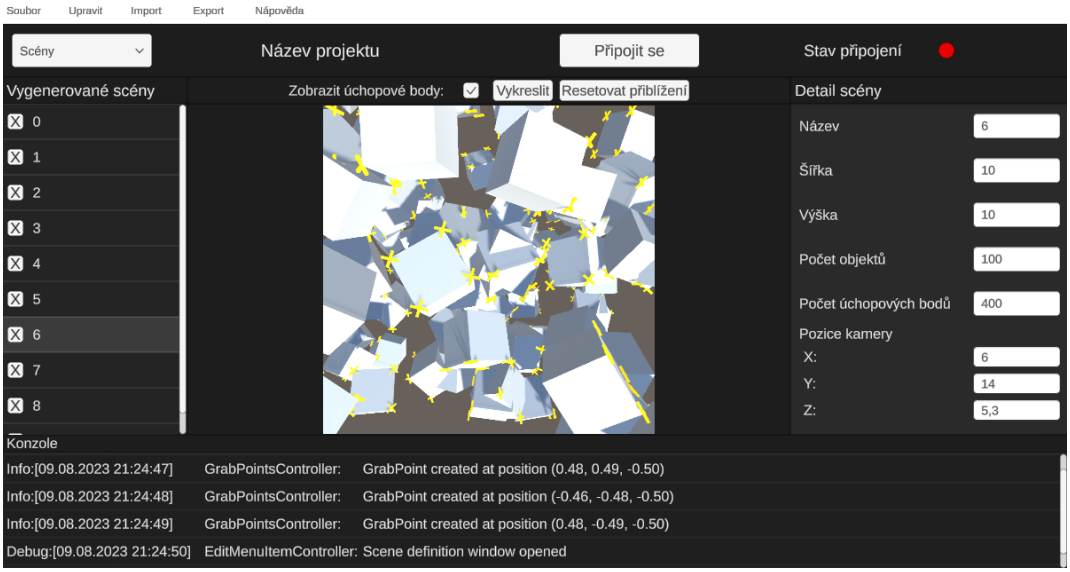


Obrázek 42: Náhled scény s poměrem stran 16:9

Po implementaci všech důležitých funkcí pro správu a vykreslení scén byl přidán panel umožňující zobrazit bližší informace o vybrané scéně (viz obr. 43). Do tohoto panelu byla také přidána možnost modifikovat pozici kamery ve 3D prostoru a tím opravit její případnou nevhodnou pozici (viz obr. 44). Možné budoucí rozšíření funkcí tohoto panelu může spočívat například v přidání více možností editace příslušné scény.



Obrázek 43: Panel detailu scény



Obrázek 44: Náhled scény s upravenou pozicí kamery

Provedením kroků popsaných výše byl splněn požadavek aplikace, aby bylo možné spravovat scény a vytvářet na jejich základě páry snímků s úchopovými body a bez nich. Kromě samotné generace bylo umožněno také jednotlivé scény mazat, zobrazovat je a modifikovat pozici kamery, která ji snímá. Před samotným vykreslením pak bylo umožněno vybrat si složku pro výstup a zadat základní pojmenování pro daný typ snímku ve dvojici. Vytvořené funkcionality lze v budoucnu rozšířit například o možnost výběru rozlišení pro výstupní snímky, přegenerování jedné konkrétní scény dle parametrů namísto všech naráz či rozšíření možností editace vytvořených scén.

6.3 Vytvoření dokumentace výsledné aplikace

Dokumentace vznikala s myšlenkou jejího zahrnutí do výsledné aplikace, kde bude přístupná prostřednictvím nástrojové lišty. Z tohoto důvodu nebyl k jejímu napsání využit typický formát WYSIWYG editoru jako například Microsoft Word, ale formát zvaný AsciiDoc.

Jak je uvedeno v dokumentaci, „AsciiDoc je lehký a sémantický značkovací jazyk primárně navržený pro psaní technické dokumentace... Schopnost vyprodukovat vícero výstupních formátů je jedna z hlavních výhod AsciiDocu“ [72, s. AsciiDoc Language Documentation] (překlad autora) Z tohoto důvodu se tento jazyk jevil jako vhodná volba.

6.3.1 Organizace souborů dokumentace

Stejně jako v případě jiných projektů, i zde bylo potřeba vhodně zvolit způsob jeho organizace a software pro jeho správu. V tomto případě byl k tomuto účelu zvolen editor Visual Studio Code 1.78.2 s příslušným doplňkem (viz [73]). Vzhledem k jednoduché syntaxi tohoto jazyka a pomocným funkcím použitého editoru byl proces psaní svižný a efektivní.

Projekt dokumentace byl vytvořen se strukturou oddělující od sebe uživatelskou dokumentaci od technické. Dále od sebe byla v rámci těchto skupin oddělena dokumentace grafické části aplikace a jejího backendu. Složka images, sloužící pro obrázky, však byla všemi částmi sdílena kvůli jejich rychlejšímu vkládání tímto způsobem.

V rámci skupin popsaných výše byly poté vytvořeny soubory pro jednotlivé kapitoly a podkapitoly, čímž se velmi ulehčila jejich editace. V tabulce 5 je tato struktura vyobrazena na úrovni složek, jelikož vyobrazení na úrovni souborů by vzhledem k množství kapitol bylo příliš rozsáhlé.

Název hlavní složky	Název složky	Oblast
program_docs	backend	Soubory technické dokumentace backendu aplikace
	gui	Soubory technické dokumentace grafické části aplikace
user_docs	backend	Soubory uživatelské dokumentace backendu aplikace
	gui	Soubory uživatelské dokumentace grafické části aplikace

Tabulka 5: Složky v projektu dokumentace a jejich význam

Příklad zdrojového kódu tohoto jazyka je uveden ve zdrojovém kódu 23. Jedná se konkrétně o úryvek z podkapitoly dokumentující konzoli aplikace. Pro bližší informace o struktuře a syntaxi tohoto jazyka vizte dokumentaci ([72, s. AsciiDoc Language Documentation])

```
ifndef::imagesdir[:imagesdir: ../../images]

[#sec:console]
== Konzole

Konzole se vždy nachází v~poslední sekci obrazovky (viz <<sec:basicAppLayout>>)
    a~slouží k~výpisu o~prováděných operacích.
Jsou zde zobrazovány informativní, varovné a~chybové zprávy
    (<<fig:console.consoleMessagePreview>>).

.Ukázka zpráv v~konzoli
image::2023-07-29T15-18-49-846Z.png[Ukázka zpráv
    v~konzoli,id="fig:console.consoleMessagePreview"]

Každá zpráva vypsaná do konzole se vkládá dolu a~má následující formát:

[listing]
Debug|Warn|Error:[dd.MM.yyyy hh:mm:ss] ZdrojováTřída: Zpráva
...
```

Zdrojový kód 23: Ukázka textu napsaného v jazyce AsciiDoc

Aby bylo možné celou dokumentaci později zkompilovat, bylo potřeba všechny tyto části vložit do hlavního souboru (viz zdrojový kód. 24). Zároveň byly v tomto souboru nastaveny další informace o použitých stylech, popiscích, generaci obsahu a dalších vlastnostech. Pro více informací vizte dokumentaci ([72]).

```
:docinfo: shared
:doctype: book
:lang: cs
:toc: left
:toc-title: Obsah
:toclevels: 3
:sectnums:
:sectlinks:
:imagesdir: ./images
:icons: font
:encoding: UTF-8
:figure-caption: Obrázek
:section-refsig: Kap.
:table-caption: Tabulka
:xrefstyle: short
:!chapter-signifier:
:experimental:
:hyphens: cs
:pdf-theme: my-theme.yml

:leveloffset: 0

= Dokumentace aplikace na generování umělých datových sad
:author: Bc. Lukáš Milar

include::./user_docs/user_docs.adoc[leveloffset=+1]

<<<
include::./program_docs/program_docs.adoc[leveloffset=+1]
```

Zdrojový kód 24: Nastavení hlavního dokumentu dokumentace

Aby měl výsledný dokument PDF formátování blíže podobné této práci, byl vytvořen soubor yaml specifikující jednotlivé vlastnosti stylu (viz zdrojový kód. 25). Jak je uvedeno v dokumentaci, pomocí souboru YAML lze pro AsciiDoctor PDF definovat styl. Lze vytvořit nový styl, nebo rozšířit existující. Rozšíření výchozího stylu však představuje nejjednoduší způsob vytvoření stylu. [72, s. Create a Theme]

```
extends: default
page:
  layout: portrait
  margin: [25mm, 15mm, 25mm, 35mm]
  margin-inner: 35mm
  margin-outer: 15mm
  size: A4
base:
  font-size: 12
  line-height: 1.5
  text-align: justify
heading:
  font-size: 17
  font-style: bold
image:
  align: center
  caption:
    align: center
    font-size: 11
table:
  caption:
    align: center
    end: bottom
footer:
  font-size: 12
  recto:
    center:
      content: '{page-number}'
    right:
      content:
  verso:
    center:
      content: $footer-recto-center-content
```

```
  left:
    content:
```

Zdrojový kód 25: Definice stylu v souboru YAML. Kód vytvořen podle: [72, s. Create a Theme]

Provedením kroků popsaných výše byla vytvořena přehledná struktura umožňující snadnou editaci textu dokumentace. Samotná editace byla také usnadněna použitím rozšíření pro editor Visual Studio Code 1.78.2. Zároveň bylo díky přidaným stylům ve formátu yaml možné blíže specifikovat konečnou podobu při následné kompilaci do formátu PDF.

6.3.2 Kompilace dokumentace do konečných formátů

Před distribucí výsledné dokumentace bylo potřeba ji převést do požadovaných formátů. Tento převod byl proveden pomocí preprocesoru Asciidoctor. Jak uvádí dokumentace, AsciiDoc není na rozdíl od formátů jako PDF či HTML 5 publikovatelný, a tak jej Asciidoctor do těchto formátů převádí. [72, s. Asciidoctor Documentation]

Pro ulehčení práce s tímto preprocesorem a eliminaci nutnosti jej instalovat do operačního systému byl využit Dockerový obraz (viz [74]). Likhanov uvádí, že tento obraz již pro generaci formátů jako PDF či HTML obsahuje vše potřebné a lze tak začít generovat dokumentaci bez nutnosti nastavování procesoru Asciidoctor. [75]

Po stažení příslušného Dockerového obrazu jej bylo potřeba nejprve spustit (viz zdrojový kód 26). Poté již bylo možné generovat potřebné formáty pomocí příkazů preprocesoru Asciidoctor (viz zdrojový kód 27).

```
~> docker run -it -v /home/user/directory-with-asciidoc-files:/documents/
    asciidoctor/docker-asciidoctor
```

Zdrojový kód 26: Spuštení Dockerového obrazu obsahujícího Asciidoctor. Kód převzat z: [75]

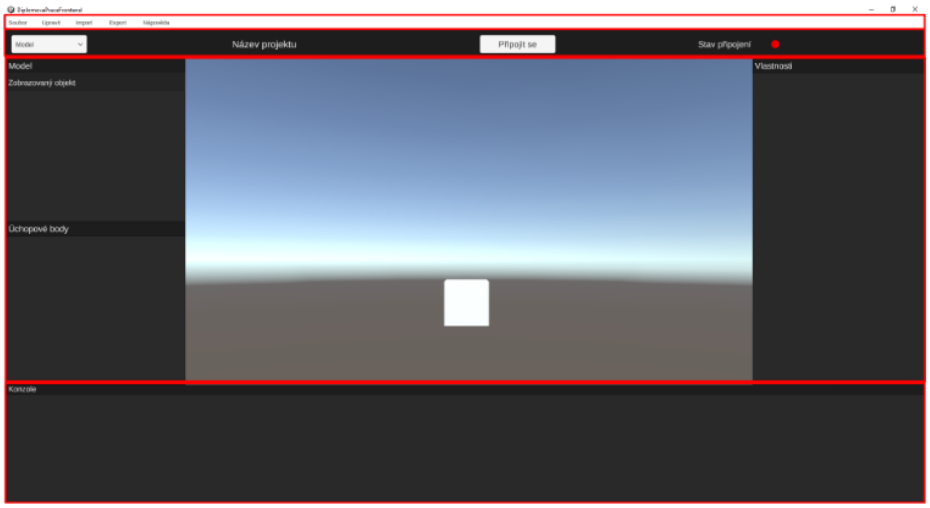
```
~> asciidoctor my-asciidoc-file.adoc
~> asciidoctor-pdf my-asciidoc-file.adoc
```

Zdrojový kód 27: Kompilace pomocí Asciidoctor do HTML a PDF. Kód převzat z: [75]

Pro potřeby této práce byla výsledná dokumentace zkompilována do formátu PDF (viz Příloha A) a formátu HTML. Dokumentace je ve druhém zmíněném formátu dostupná z aplikace prostřednictvím položky Nápověda > Návod v liště s menu, čímž může uživatelům usnadnit její používání. Ukázky podoby obou výsledných formátů jsou vyobrazeny na obrázcích 45 a 46.

1.2. Základní rozhraní aplikace

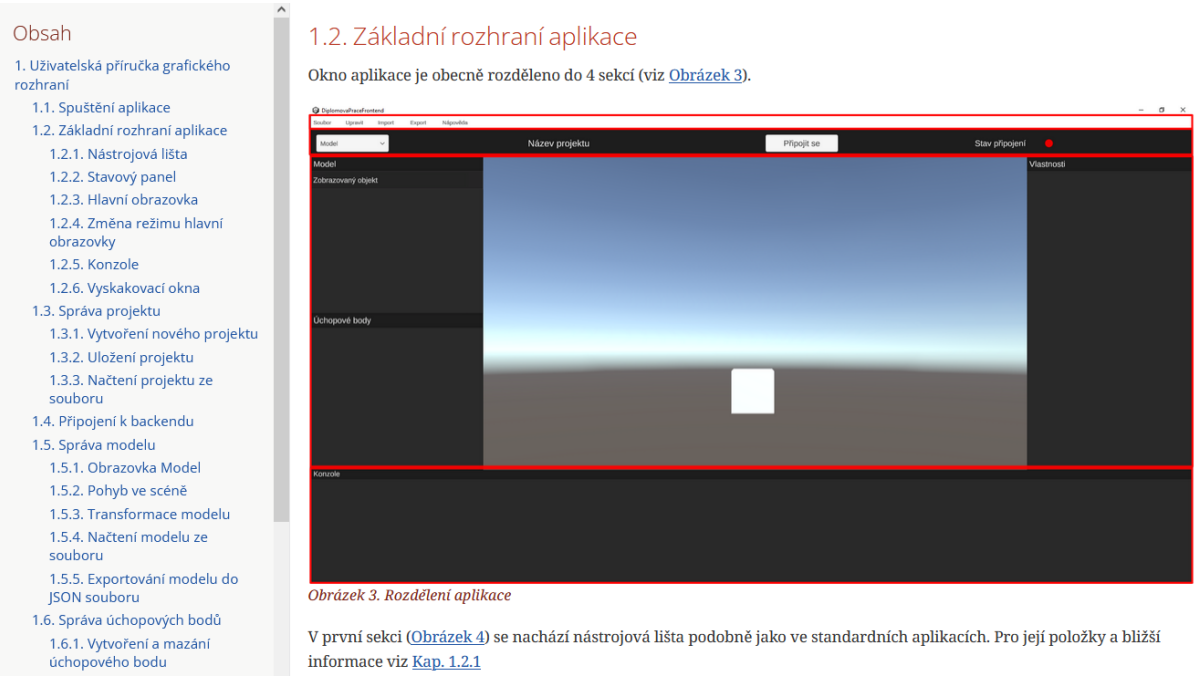
Okno aplikace je obecně rozděleno do 4 sekcí (viz Obrázek 3).



Obrázek 3. Rozdělení aplikace

V první sekci (Obrázek 4) se nachází nástrojová lišta podobně jako ve standardních aplikacích. Pro její položky a bližší informace viz Kap. 1.2.1

Obrázek 45: Ukázka zkompilované dokumentace do formátu PDF



Obrázek 46: Ukázka zkompilované dokumentace do formátu HTML

Provedením kroků popsaných výše bylo možné vytvořenou dokumentaci převést do požadovaných formátů preprocesorem Asciidoctor. Zároveň díky využití Dockerového obrazu nebylo nutné tento preprocesor instalovat do operačního systému, což celý proces ulehčilo. Výsledné formáty poté mohly být přiloženy k této práci a do samotné aplikace, což uživatelům usnadní používání vytvořené aplikace.

7 TESTOVÁNÍ FUNKCE A VÝKONU APLIKACE

V této kapitole jsou popsány výsledky z testování funkcí a výkonu výsledné aplikace. Veškeré testy popsané v této kapitole byly prováděny na operačním systému Windows 10 22H2. Specifikace použité sestavy jsou uvedeny v tabulce 6.

Typ	Hodnota
Model	Lenovo Legion 5 Pro 16ACH6H
Procesor	AMD Ryzen 5 5600H, 3.30 GHz
RAM	16 GB DDR4
GPU	NVIDIA GeForce RTX 3060 6GB

Tabulka 6: Specifikace sestavy použité pro testování aplikace

Jednotlivé funkce aplikace byly testovány, aby bylo potvrzeno splnění požadavků této práce. Zároveň s testováním funkcí aplikace probíhalo také měření jejího výkonu při provádění jednotlivých operací. Měření na straně grafické části probíhalo s pomocí Unity profiling tools a pro zachycení dat byl v balíčku Profiling vytvořen skript (viz zdrojový kód 28). Jak je uvedeno na stránkách Unity, „Unity Profiler je součástí Unity Editoru a přichází s nízkoúrovňovým Profiler API pluginem, abyste mohli kustomizovat vaši analýzu a exportovat profilovací data do ostatních nástrojů.“ [76] (překlad autora)

```
public class CollectStatsScript : MonoBehaviour
{
    ...
    private ProfilerRecorder _systemMemoryRecorder;
    private ProfilerRecorder _mainThreadRecorder;
    ...
    void OnEnable()
    {
        _systemMemoryRecorder = ProfilerRecorder.StartNew(ProfilerCategory.Memory,
            "System Used Memory");
        _mainThreadRecorder = ProfilerRecorder.StartNew(ProfilerCategory.Internal,
            "Main Thread");
    }

    void OnDisable()
```

```
    {
        _systemMemoryRecorder.Dispose();
        _mainThreadRecorder.Dispose();
    }
    ...
    IEnumerator CollectData()
    {
        Debug.Log("Started measuring");
        _currentData = new StringBuilder();
        _currentData.Append("time;memoryInKB;mainThreadTimeMs;\n");
        while (true)
        {
            _currentData.Append($"{DateTime.Now:HH:mm:ss};" +
                $"{_systemMemoryRecorder.LastValue / 1024}" +
                $";{_mainThreadRecorder.LastValue * _nanosecondsToMillisMultiplier}\n");
            yield return new WaitForSeconds(1f);
        }
    }
}
```

Zdrojový kód 28: Skript pro zachycení statistik v Unity. Kód vytvořen podle: [17, s. Rendering Profiler module], [18, s. ProfilerRecorder]

Pro získání dat o využití zdrojů na straně Dockerových kontejnerů byl použit příkaz docker stats. Výstup tohoto příkazu byl pomocí skriptu rozdělen dle kontejnerů a převeden do snadněji zpracovatelného formátu a uložen do souborů pro pozdější zpracování (viz zdrojový kód 29).

```
#!/bin/sh
CONT_SPRING="artificial_generation_spring"
CONT_PYTHON="artificial_generation_python"
CONT_RABBITMQ="artificial-generation-rabbitmq"
DOCKER_STATS=""

function prepareFile() {
    touch $1.txt;
    truncate -s 0 $1.txt;
    echo 'time cpuUsagePercent memoryUsageKB' >> $1.txt
}
```

```
function collectData() {
    grep $1 <<< $DOCKER_STATS |
    awk -v date="$(date +%T)" '{
        if(index($4, "GiB")) {
            gsub("GiB","",$4);
            print date, $2, $3, $4 * 1000 * 1000
        }
        else {
            gsub("MiB","",$4);
            print date, $3, $4 * 1000}
        }' >> $2.txt;
}

prepareFile $CONT_SPRING"_"$1
prepareFile $CONT_PYTHON"_"$1
prepareFile $CONT_RABBITMQ"_"$1

while true;
do
    DOCKER_STATS=$(docker stats --no-stream)
    collectData $CONT_SPRING $CONT_SPRING"_"$1
    collectData $CONT_PYTHON $CONT_PYTHON"_"$1
    collectData $CONT_RABBITMQ $CONT_RABBITMQ"_"$1
done
```

Zdrojový kód 29: Uložení statistik z Dockeru do souboru. Kód vytvořen podle: [77]

Nejprve byl vyzkoušen import modelu kostky s 8 vrcholy ze souboru typu .obj. Mimo jiné vzhledem k jednoduchosti tohoto modelu nebyl problém jej načíst a operace tak celkově zabrala pouze 3 sekundy. Výsledky z vytížení systémových prostředků jsou vyobrazeny v tabulkách 7 a 8.

Počet vrcholů	Doba trvání (s)	Maximální využití RAM (MB)
8	3	722,28

Tabulka 7: Využití zdrojů v Unity při načítání modelů

Kontejner	Maximální využití RAM (MB)	Využití CPU (%)
artificial_generation_python	402,64	0,02
artificial_generation_spring	908,20	12,31
artificial-generation-rabbitmq	129,29	0,51

Tabulka 8: Využití zdrojů v Dockeru při načítání modelů

Následoval pokus o importování komplexnějšího modelu o 7958 vrcholech opět z formátu .obj. Ani v tomto případě nedošlo k žádným problémům a celkově operace trvala 15 sekund. Vytížení jednotlivých částí aplikace při této operaci jsou vyobrazeny v tabulkách 9 a 10. Jak je z těchto dat patrné, nejvíce se změnilo vytížení procesoru. V obou případech však lze konstatovat funkčnost vytvořeného řešení.

Počet vrcholů	Doba trvání (s)	Maximální využití RAM (MB)
7958	15	744,90

Tabulka 9: Využití zdrojů v Unity při načítání komplexnějšího modelu

Kontejner	Maximální využití RAM (MB)	Využití CPU (%)
artificial_generation_python	433,10	95,63
artificial_generation_spring	907,43	18,33
artificial-generation-rabbitmq	141,30	9,98

Tabulka 10: Využití zdrojů v Dockeru při načítání komplexnějšího modelu

Jakmile bylo vyzkoušeno načítání modelů ze souboru, proběhlo testování generace scén. Generace opět probíhala s pomocí backendu a ve scéně byl ve všech případech umístěn model s 507 vrcholy a dva úchopové body. Generování náhodných pozic bylo vždy řízeno vzorkováním Poissonova disku bez použití simulací a celkově byly pro generaci scén v Pythonu vytvořeny 4 procesy.

Nejprve bylo generováno 10 scén s maximálním počtem 37 objektů na ploše 10x10. V rámci této úlohy tak aplikace měla za úkol vytvořit scény, ve kterých může být potřeba vykreslit až 18759 vrcholů bez započítání úchopových bodů. Tuto úlohu zvládla aplikace splnit bez problémů během pěti sekund. Bližší výsledky výkonu této operace jsou vyobrazeny v tabulkách 11 a 12.

Počet scén	Maximální počet modelů ve scéně	Doba trvání (s)	Maximální využití RAM (MB)
10	37	5	755,77

Tabulka 11: Využití zdrojů v Unity při jednodušší generaci scén

Kontejner	Maximální využití RAM (MB)	Využití CPU (%)
artificial_generation_python	470,51	0,05
artificial_generation_spring	883,59	11,31
artificial-generation-rabbitmq	950,20	0,48

Tabulka 12: Využití zdrojů v Dockeru při jednodušší generaci scén

Ve druhém případě bylo generováno 300 scén s maximálním počtem objektů 1463 objektů na ploše 20x20 a 10 vertikálními vrstvami. Dokončit tuto operaci v tomto případě trvalo aplikaci přibližně 7 minut a počet scén odpovídal stanovenému množství. Uživatel by tak v tomto případě již pocítil vyšší náročnost této operace.

V tabulkách 13 a 14 je zachyceno vytížení jednotlivých částí aplikace při generaci scén s tímto nastavením. Z těchto dat lze tedy usoudit, že aplikace je schopná generovat komplexnější scény ve větším rozsahu, avšak bez optimalizace tohoto procesu nelze příliš doporučit generaci na komplexnější úrovni, než zde byla testována.

Počet scén	Maximální počet modelů ve scéně	Doba trvání (s)	Maximální využití RAM (MB)
300	1463	432	2104,61

Tabulka 13: Využití zdrojů v Unity při náročnější generaci scén

Kontejner	Maximální využití RAM (MB)	Využití CPU (%)
artificial_generation_python	201,46	1188,56
artificial_generation_spring	500,78	155,27
artificial-generation-rabbitmq	305,57	17,78

Tabulka 14: Využití zdrojů v Dockeru při náročnější generaci scén

Nakonec byly scény vygenerované během testů popsaných výše vykreslovány do souborů. V tomto případě byla již sledována pouze grafická část aplikace, jelikož pro operaci vykreslování scén není backend využíván. Pro tuto funkcionalitu tak bylo měřeno pouze množství využité paměti a doba trvání při provádění operace.

Veškeré scény byly v obou případech úspěšně vykresleny do správného počtu páru obrázků s úchopovými body a bez nich, čímž lze usoudit, že tato funkcionalita pracuje správně. Jak je vidět v tabulce 15, rozdíl v náročnosti vykreslení 10 nenáročných a 300 komplexních scén uživatel pocítí, ale zároveň ne v takové míře, aby nutně označil aplikaci za nepoužitelnou.

Počet scén	Maximální počet modelů ve scéně	Doba trvání (s)	Maximální využití RAM (MB)
10	37	4	825,09
300	1463	386	2354,75

Tabulka 15: Využití zdrojů v Unity při vykreslování scén

Vytvořená aplikace dle provedených testů splňuje stanovené požadavky na její fungování. Veškeré operace fungují v základních scénářích bez problémů. V případě komplexnějších úloh je zde však stále v případě budoucích rozšíření prostor pro optimalizaci zejména v samotné generaci scén, která je při vysokých hodnotách množství scén ke generaci a maximálního počtu modelů ve scéně již vcelku časově náročná. Při střídmějších hodnotách však dle provedených testů nečiní aplikaci problém požadované operace splnit.

ZÁVĚR

Cílem této práce bylo popsat existující řešení v oblasti nástrojů pro generaci umělých datových sad a nástrojů pro správu 3D scén a tvorby jejich průmětů. Dalším cílem bylo vytvořit aplikaci umožňující načíst objekt ze standardních 3D formátů, definovat úchopové body a definovat parametry pro následnou generaci scén. Aplikace měla být poté schopná náhodně rozmístit objekty ve scéně dle zadaných parametrů a jako výstup poskytnout páry snímků těchto scén, přičemž v každém páru se má nacházet scéna s vyznačenými úchopovými body a bez nich.

V teoretické části byla nejprve přiblížena problematika umělých datových sad a současné trendy v této oblasti, ze kterých bylo blíže představeno využití generativních soupeřících sítí. Poté byly představeny některé konkrétní nástroje, z nichž některé tyto sítě využívají.

Po představení této problematiky byly představeny některé existující nástroje pro správu 3D scén a tvorby jejich průmětů. V této části byl také krátce představen formát USD a jeho výhody. Z představených nástrojů byl poté blíže popsán herní engine Unity. Tento engine byl poté zvolen jakožto základ pro implementaci grafické části aplikace. Provedená rešerše tak poskytla důležitý náhled do zkoumané problematiky, který ovlivnil pozdější implementaci výsledné aplikace jako celku.

Praktická část započala definicí funkčních a nefunkčních požadavků a případů užití na základě zadání této práce. Poté byla provedena příprava před samotnou implementací, v rámci které bylo rozhodnuto rozdělit výslednou aplikaci na dvě samostatné části. Pro grafickou část zde byl také vytvořen návrh uživatelského rozhraní, ve kterém jsou definovány jeho jednotlivé části tak, aby splňovaly definované požadavky.

Na základě těchto podkladů byla poté provedena samotná implementace výsledné aplikace. Nejprve je popsána implementace backendu aplikace, který byl postaven na architektuře mikroslužeb. Architektura se skládá se služby poskytující API pro grafickou část a služby vykonávající operace načítání dat ze standardních 3D formátů a generace scén dle parametrů. První zmíněná služba byla vytvořena ve frameworku Spring Boot 3.1.0, zatímco druhá v jazyce Python 3.11. Pro komunikaci mezi těmito službami byl použit broker zpráv RabbitMQ 3.8 a pro jednoduché spuštění v lokálním prostředí byl nakonec celý backend dockerizován.

Poté byla popsána implementace grafické části aplikace v herním enginu Unity. V této části byla nejprve představena struktura vytvořeného projektu a vytvořeny některé základní prvky uživatelského rozhraní, mezi kterými byly také vyskakovací okna. Následně byla implementována základní správa projektu umožňující jeho ukládání a načítání. Po její implementaci byly také přidány funkce pro komunikaci s vytvořeným backendem.

Následně byly implementovány hlavní funkce aplikace starající se o správu zobrazovaných modelů včetně jejich načítání a po jejich implementaci byly také implementovány funkce pro správu úchopových bodů včetně možnosti jejich výběru, transformace či mazání. S tímto základem byly poté implementovány funkce pro správu definice scén pro jejich generaci a funkce pro jejich následnou správu včetně mazání, modifikace pozice kamery, zobrazování náhledu a vykreslování do souborů.

Pro vytvořenou aplikaci byla poté v jazyce AsciiDoc vytvořena dokumentace obsahující mimo jiné ukázkový příklad použití výsledné aplikace. Dokumentace byla poté pomocí preprocesoru Asciidoctor zkompilována do formátů PDF a HTML, přičemž druhý formát byl zpřístupněn přímo skrze aplikaci.

Nakonec bylo provedeno testování aplikace s cílem zhodnotit její funkčnost a výkon. Provedené testy byly zaměřeny na oblast načítání modelů ze souboru, generace scén a jejich vykreslování do párů snímků. Pro každou oblast byly provedeny dva testy, kdy první představoval jednoduchý případ, zatímco druhý fungoval jako případ náročnější. Aplikace byla poté na základě provedených testů vyhodnocena jako funkční a dostatečně výkonná, avšak s prostorem pro optimalizaci zejména generace scén.

Výslednou aplikaci je možné dále rozšiřovat například přidáním dalších způsobů generace náhodných pozic, možnosti opětovného vygenerování jedné konkrétní scény namísto všech či nabídky exportu vygenerovaných scén do některého ze standardních 3D formátů. Díky architektuře mikroslužeb lze také aplikaci rozšířit například o možnost přímého napojení na konkrétní neuronovou síť, která by poté mohla též pomoci uživateli při definici úchopových bodů a celkově zefektivnit proces vytváření výsledných párů snímků.

POUŽITÁ LITERATURA

[1] KAFESU, Anesu. 13 Tools for Synthetic Data Generation to Train Machine Learning Models. *GeekFlare* [online]. 23. 03. 2023 [cit. 20. 08. 2023]. Dostupné z: `https://geekflare.com/synthetic-data-generation-tools/`

[2] WATSON, Alex. What is Synthetic Data? *Gretel* [online]. 29. 04. 2022 [cit. 20. 08. 2023]. Dostupné z: `https://gretel.ai/blog/what-is-synthetic-data`

[3] BROWNLEE, Jason. A Gentle Introduction to Generative Adversarial Networks (GANs). *Machine Learning Mastery* [online]. 19. 07. 2019 [cit. 20. 08. 2023]. Dostupné z: `https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/`

[4] KLAUDEL, Gaja. StyleGAN 3 – Computer Vision and Image Manipulation. *Appsilon* [online]. 17. 03. 2022 [cit. 20. 08. 2023]. Dostupné z: `https://appsilon.com/stylegan-3-image-manipulation/`

[5] KAHNG, Minsuk, CHAU, Duen Horng a kolektiv. GAN Lab: An Interactive, Visual Experimentation Tool for Generative Adversarial Networks. *GitHub* [online]. 05. 09. 2018 [cit. 20. 08. 2023]. Dostupné z: `https://github.com/poloclub/ganlab`

[6] Synthesized Ltd. The story of synthetic data and our DataOps platform - Synthesized. *Synthesized* [online]. 16. 08. 2023 [cit. 20. 08. 2023]. Dostupné z: `https://www.synthesized.io/company`

[7] Gretel Labs. Welcome to Gretel! *Gretel* [online]. 07. 08. 2023 [cit. 20. 08. 2023]. Dostupné z: `https://docs.gretel.ai/`

[8] Blender Foundation. About. *Blender* [online]. Nedatováno [cit. 20. 08. 2023]. Dostupné z: `https://www.blender.org/about/`

[9] VAZQUEZ, Pablo. Blender Apps. *Blender* [online]. 02. 11. 2022 [cit. 20. 08. 2023]. Dostupné z: `https://code.blender.org/2022/11/blender-apps/`

[10] Arkance Systems. Autodesk Maya - profesionální tvorba 3D grafiky v oblasti digitálních médií. *Arkance Sytems* [online]. 15. 09. 2022 [cit. 20. 08. 2023].

Dostupné z: https://www.arkance-systems.cz/produkty/media-a-design/autodesk-maya-a-maya-lt

[11] SideFX. Film & TV. *SideFX* [online]. 05. 08. 2023 [cit. 20. 08. 2023]. Dostupné z: https://www.sidefx.com/industries/filmtv/

[12] Unity Technologies. Real-Time 3D Development Platform & Editor. *Unity* [online]. 17. 12. 2019 [cit. 20. 08. 2023]. Dostupné z: https://unity.com/products/unity-engine

[13] Unity Technologies. Programming and scripting with Unity. *Unity* [online]. 28. 10. 2021 [cit. 20. 08. 2023]. Dostupné z: https://unity.com/solutions/programming

[14] LINIETSKY, Juan, MANZUR Ariel a kolektiv. Introduction. *Godot Docs* [online]. © 2014 – 2023 [cit. 18. 08. 2023]. Dostupné z: https://docs.godotengine.org/en/4.1/about/introduction.html

[15] NVIDIA. USD Composer Overview — composer latest documentation. *NVIDIA Omniverse Documentation* [online]. 18. 08. 2023 [cit. 18. 08. 2023]. Dostupné z: https://docs.omniverse.nvidia.com/composer/latest/index.html

[16] Pixar Animation Studios. Introduction to USD — Universal Scene Description 23.08 documentation. *Universal Scene Description* [online]. 26. 07. 2016 [cit. 18. 08. 2023]. Dostupné z: https://openusd.org/release/intro.html

[17] Unity Technologies. Unity - Manual: Unity User Manual 2022.3 (LTS). *Unity3D* [online]. 17. 08. 2023 [cit. 21. 08. 2023]. Dostupné z: https://docs.unity3d.com/2022.3/Documentation/Manual/

[18] Unity Technologies. Unity - Scripting API. *Unity3D* [online]. 17. 08. 2023 [cit. 21. 08. 2023]. Dostupné z: https://docs.unity3d.com/2022.3/Documentation/ScriptReference/

[19] Unity Technologies. Unity UI: Unity User Interface | Unity UI | 2.0.0. *Unity3D* [online]. 08. 06. 2023 [cit. 03. 08. 2023]. Dostupné z: https://docs.unity3d.com/Packages/com.unity.ugui@2.0/manual/

[20] HARWOOD, Paul. Simple Guide to Unity Package Management. *Medium* [online]. 24. 12. 2020 [cit. 19. 08. 2023]. Dostupné z: https://medium.com/runic-software/simple-guide-to-unity-package-management-4aea43d1baf7

[21] ARLOW, Jim a Ila NEUSTADT. *UML2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky*. 2., aktualiz. a dopl. vyd. Brno: Computer Press, 2007. ISBN 978-80-251-1503-9.

[22] GORTLER, Steven J. *Foundations of 3D Computer Graphics*. Cambridge, MA: MIT Press, 2012. ISBN 978-0262017350.

[23] BRACEY, Kezz. What is Figma? *Envato Tuts+* [online]. 16. 09. 2022 [cit. 02. 08. 2023]
Dostupné z: https://webdesign.tutsplus.com/what-is-figma--cms-32272a

[24] HICKEY, James. The Life-changing (And Time-saving!) Magic Of Feature Focused Code Organization! *DEV Community* [online]. 21. 11. 2018 [cit. 04. 08. 2023]. Dostupné z: https://dev.to/jamesmh/the-life-changing-and-time-saving-magic-of-feature-focused-code-organization-1708

[25] Socket.IO. Introduction *Socket.IO* [online]. 11. 07. 2023 [cit. 04. 08. 2023]. Dostupné z: https://socket.io/docs/v4/

[26] KOKSHAROV, Nikita. Netty-socketio Overview. *GitHub* [online]. 01. 07. 2023 [cit. 16. 08. 2023]. Dostupné z: https://github.com/mrniko/netty-socketio

[27] UÇAR, Gürkan.Spring Boot Netty Socket.IO Example. *Medium* [online]. 05. 09. 2022 [cit. 14. 08. 2023]. Dostupné z: https://medium.com/folksdev/spring-boot-netty-socket-io-example-3f21fcc1147d

[28] baeldung. A Guide to CuncurrentMap. *Baeldung* [online]. 06. 05. 2023 [cit. 16. 08. 2023]. Dostupné z: https://www.baeldung.com/java-concurrent-map

[29] VMware, Inc. Architecture :: Spring Security. *Spring* [online]. © 2023 [cit. 16. 08. 2023]. Dostupné z: https://docs.spring.io/spring-security/reference/servlet/architecture.html

[30] VMware, Inc. Cross Site Request Forgery (CSRF) for WebFlux Environments :: Spring Security. *Spring* [online]. © 2023 [cit. 16. 08. 2023]. Dostupné z: https://docs.spring.io/spring-security/reference/reactive/exploits/csrf.html#webflux-csrf-using

[31] TRANDAFIR, Emanuel. Extracting a Custom Header From the Request. *Baeldung* [online]. 20. 01. 2023 [cit. 16. 08. 2023]. Dostupné z: https://www.baeldung.com/spring-extract-custom-header-request

[32] KHAN, Uzma. Multipart Request Handling in Spring. *Baeldung* [online]. 07. 05. 2022 [cit. 16. 08. 2023]. Dostupné z: https://www.baeldung.com/sprint-boot-multipart-requests

[33] VMware, Inc Class InputStreamResource. *Spring* [online]. 03. 03. 2015 [cit. 16. 08. 2023]. Dostupné z: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/io/InputStreamResource.html

[34] PARASCHIV, Eugen. Error Handling for REST with Spring. *Baeldung* [online]. 03. 09. 2022 [cit. 16. 08. 2023]. Dostupné z: https://www.baeldung.com/exception-handling-for-rest-with-spring

[35] KULLING, Kim. Introduction — Asset-Importer-Lib March 2022 v5.2.3 documentation. *Asset-Importer-Lib March 2022 v5.2.3 documentation* [online]. © 2020 – 2022 [cit. 17. 08. 2023]. Dostupné z: https://assimp-docs.readthedocs.io/en/latest/about/introduction.html

[36] NumPy Developers NumPy documentation — NumPy v1.25 Manual. *NumPy* [online]. © 2008 – 2022 [cit. 17. 08. 2023]. Dostupné z: https://numpy.org/doc/stable/index.html

[37] DAWSON-HAGGERTY, Michael a kolektiv. Basic Installation — trimesh 3.23.3 documentation. *trimesh 3.23.3 documentation* [online]. © 2022 [cit. 17. 08. 2023]. Dostupné z: https://trimsh.org/index.html

[38] DAVIES, Jason. Poisson-Disc Sampling. *Jason Davies* [online]. 14. 05. 2014 [cit. 17. 08. 2023]. Dostupné z: https://www.jasondavies.com/poisson-disc/

[39] GOEBEL, Rainer. Spatial Transformation Matrices. *BrainVoyager* [online]. 22. 07. 2017 [cit. 17. 08. 2023]. Dostupné z: https://www.brainvoyager.com/bv/doc/UsersGuide/CoordsAndTransforms/SpatialTransformationMatrices.html

[40] CHAND, Shelvin. PyBullet: Getting Started. *Medium* [online]. 23. 08. 2020 [cit. 18. 08. 2023]. Dostupné z: https://medium.com/@chand.shelvin/pybullet-getting-started-a068a0e3d492

[41] VINKA, Elin. Microservices - why use RabbitMQ? *CloudAMQP* [online]. 03. 04. 2020 [cit. 14. 08. 2023]. Dostupné z: https://www.cloudamqp.com/blog/why-use-rabbitmq-in-a-microservice-architecture.html

[42] VMware. AMQP 0-9-1 Model Explained. *RabbitMQ* [online]. 01. 09. 2011 [cit. 16. 08. 2023]. Dostupné z: https://www.rabbitmq.com/tutorials/amqp-concepts.html

[43] VMware, Inc. Getting Started | Messaging with RabbitMQ. *Spring* [online]. 15. 09. 2013 [cit. 16. 08. 2023]. Dostupné z: https://spring.io/guides/gs/messaging-rabbitmq/

[44] GARNOCK-JONES, Tony, ROY, Gavin M., Pivotal Software, Inc a kolektiv. Introduction to Pika — pika 1.3.2 documentation. *pika 1.3.2 documentation* [online]. © 2009 – 2017 [cit. 15. 08. 2023]. Dostupné z: https://pika.readthedocs.io/en/1.3.2/

[45] VMware. RabbitMQ tutorial - Work Queues — RabbitMQ. *RabbitMQ* [online]. 10. 08. 2011 [cit. 16. 08. 2023]. Dostupné z: https://www.rabbitmq.com/tutorials/tutorial-two-python.html

[46] IBM. What is Docker?. *IBM* [online]. 06. 01. 2020 [cit. 15. 08. 2023]. Dostupné z: https://www.ibm.com/topics/docker

[47] MUSYOKA, Faith. Dockerizing a RabbitMQ Instance using Docker Containers. *Section* [online]. 20. 08. 2021 [cit. 17. 08. 2023]. Dostupné z: https://www.section.io/engineering-education/dockerize-a-rabbitmq-instance/

[48] ANDERSON, Will. Docker for Windows: Dealing With Windows Line Endings. *Will Anderson* [online]. 11. 08. 2016 [cit. 17. 08. 2023]. Dostupné

z: https://willi.am/blog/2016/08/11/docker-for-windows-dealing-with-windows-line-endings/

[49] Docker Inc. Use containers for development. *Docker Docs* [online]. 26. 04. 2021 [cit. 15. 08. 2023]. Dostupné z: https://docs.docker.com/language/java/develop/

[50] WAGNER, Bill, DYKSTRA, Tom, VICTOR, Youssef. C# identifier naming rules and conventions. *Microsoft Learn* [online]. 08. 01. 2023 [cit. 06. 08. 2023]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/identifier-names

[51] KESLER, Thomas. Game Manager. One Manager to Rule them All. *Medium* [online]. 09. 04. 2021 [cit. 04. 08. 2023]. Dostupné z: https://foxxthom.medium.com/game-manager-one-manager-to-rule-them-all-1c06afa72b23

[52] FOWLER, Martin. Inversion of Control Containers and the Dependency Injection pattern. *martinfowler.com* [online]. 23. 01. 2004 [cit. 04. 08. 2023]. Dostupné z: https://martinfowler.com/articles/injection.html#FormsOfDependencyInjection

[53] BAKKER, Mathijs. Extenject: extensions, bug fixes and updates for Zenject. *GitHub* [online]. 15. 11. 2022 [cit. 04. 08. 2023]. Dostupné z: https://github.com/Mathijs-Bakker/Extenject

[54] DAVIS, Micha. Using C# Generic Types With Scriptable Objects in Unity. *Medium* [online]. 25. 08. 2021 [cit. 06. 08. 2023]. Dostupné z: https://medium.com/nerd-for-tech/using-c-generic-types-with-scriptable-objects-in-unity-340cfc32811f

[55] HIPPLE, Ryan. Unite Austin 2017 - Game Architecture with Scriptable Objects. *YouTube* [online video]. 20. 11. 2017. 1 hodina 4 minuty 28 sekund [cit. 06. 08. 2023]. Dostupné z: https://www.youtube.com/watch?v=raQ3iHhE_Kk

[56] Refactoring.Guru. Observer. *Refactoring Guru* [online]. 05. 12. 2018 [cit. 03. 08. 2023]. Dostupné z: https://refactoring.guru/design-patterns/observer

[57] Cysharp, Inc. UniTask. *GitHub* [online]. 10. 11. 2022 [cit. 08. 08. 2023]. Dostupné z: https://github.com/Cysharp/UniTask

[58] Newtonsoft. Serializing and Deserializing JSON. *Newtonsoft* [online]. Nedatováno [cit. 08. 08. 2023]. Dostupné z: https://www.newtonsoft.com/json/help/html/SerializingJSON.htm

[59] Newtonsoft. Performance Tips. *Newtonsoft* [online]. 21. 10. 2014 [cit. 08. 08. 2023]. Dostupné z: https://www.newtonsoft.com/json/help/html/Performance.htm

[60] Newtonsoft. Serialization Attributes. *Newtonsoft* [online]. Nedatováno [cit. 08. 08. 2023]. Dostupné z: https://www.newtonsoft.com/json/help/html/SerializationAttributes.htm

[61] MONTEMAGNO, James, JAIN, Tarun, PINE, David a kolektiv. Use enumeration classes instead of enum types. *Microsoft Learn* [online]. 13. 04. 2022 [cit. 07. 08. 2023]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/enumeration-classes-over-enum-types

[62] SMITH, Steve. Enum Alternatives in C#. *Ardalis* [online]. 27. 07. 2016 [cit. 07. 08. 2023]. Dostupné z: https://ardalis.com/enum-alternatives-in-c/

[63] Refactoring.Guru. Strategy. *Refactoring Guru* [online]. 04. 12. 2018 [cit. 07. 08. 2023]. Dostupné z: https://refactoring.guru/design-patterns/strategy

[64] FRENCH, John. Singletons in Unity (done right) *Game Dev Beginner* [online]. 23. 12. 2021 [cit. 03. 08. 2023]. Dostupné z: https://gamedevbeginner.com/singletons-in-unity-the-right-way/

[65] KULA, Süleyman Yasir. Unity Simple File Browser *GitHub* [online]. 31. 05. 2023 [cit. 03. 08. 2023]. Dostupné z: https://github.com/yasirkula/UnitySimpleFileBrowser

[66] GUBB, Richard. Unity UI drop down menu. *YouTube* [online video]. 30. 10. 2014. 5 minut 43 sekund [cit. 03. 08. 2023]. Dostupné z: https://www.youtube.com/watch?v=-FcrLi49RTg&t=4s

[67] Unity Technologies. Unity - Scripting API: Application.OpenURL. *Unity3D* [online]. 17. 08. 2023 [cit. 21. 08. 2023]. Dostupné z: https://docs.unity3d.com/2022.3/Documentation/ScriptReference/Application.OpenURL.html

[68] Refactoring.Guru. Command. *Refactoring Guru* [online]. 04. 12. 2018 [cit. 03. 08. 2023]. Dostupné z: https://refactoring.guru/design-patterns/command

[69] NAJIM. SocketIOUnity. *GitHub* [online]. 11. 04. 2023 [cit. 03. 08. 2023]. Dostupné z: https://github.com/itisnajim/SocketIOUnity

[70] BRADFIELD, Chris. Camera Gimbal :: Godot 3 Recipes *KidsCanCode* [online]. 19. 08. 2019 [cit. 08. 08. 2023]. Dostupné z: https://kidscancode.org/godot_recipes/3.x/3d/camera_gimbal/

[71] MONESE, Enrico. Rendering meshes in Unity without MeshRenderers. *Edraflame* [online]. 28. 01. 2022 [cit. 09. 08. 2023]. Dostupné z: https://www.edraflame.com/blog/unity-graphics-drawmesh-drawmeshinstanced/

[72] ALLEN, Dan, WHITE, Sarah a kolektiv. Asciidoctor Documentation Site. *Asciidoctor Docs* [online]. 06. 09. 2013 [cit. 31. 07. 2023]. Dostupné z: https://docs.asciidoctor.org/

[73] Asciidoctor. AsciiDoc - Visual Studio Marketplace. *Visual Studio Marketplace* [online]. 21. 07. 2023 [cit. 31. 07. 2023]. Dostupné z: https://marketplace.visualstudio.com/items?itemName=asciidoctor.asciidoctor-vscode&ssr=false

[74] Asciidoctor. asciidoctor/docker-asciidoctor - Docker Image. *Docker Hub* [online]. 14. 08. 2023 [cit. 23. 08. 2023]. Dostupné z: https://hub.docker.com/r/asciidoctor/docker-asciidoctor/

[75] LIKHANOV, Vladimir. Generating documents with the Asciidoctor Docker image. *DocsMatter* [online]. 10. 08. 2021 [cit. 02. 08. 2023]. Dostupné z: https://www.docsmatter.org/post/asciidoc-generating-docs-with-docker/

[76] Unity Technologies. Optimize performance and quality | Unity's profiling tools. *Unity* [online]. 23. 06. 2022 [cit. 21. 08. 2023]. Dostupné z: https://unity.com/features/profiling

[77] ZAKARIA, Amine. Extracting memory usage from docker stats using bash tricks. *Zakaria Amine* [online]. 04. 12. 2019 [cit. 21. 08. 2023]. Dostupné z: http://www.zakariaamine.com/2019-12-04/monitoring-docker/

SEZNAM PŘÍLOH

PŘÍLOHA A

Přiložený archiv ve formátu zip.

Archiv má následující strukturu:

- backend
  - ArtificialGenerationBackend.zip – Zdrojové kódy backendu napsané ve Spring Boot
  - ArtificialGenerationWorkers.zip – Zdrojové kódy backendu napsané v Pythonu
  - docker-compose.dev.yml – Soubor Docker Compose pro spuštění backendu
- diagrams – Vytvořené diagramy požadavků a případů užití
- docs
  - compiled - Zkompilovaná dokumentace ve formátu PDF
  - source – Zdrojové kódy dokumentace
- frontend
  - compiled
    - diplomovaPraceFrontend_linux – Spustitelná aplikace pro OS Linux
    - diplomovaPraceFrontend_windows – Spustitelná aplikace pro OS Windows
  - package – Projekt zabalený do formátu unitypackage
  - source – Zdrojové kódy grafické části práce