FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Prohlížeč scény v aplikaci I3T
Student:	Dan Rakušan
Vedoucí:	Ing. Petr Felkel, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Počítačová grafika
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

I3T je interaktivní nástroj pro podporu výuky geometrických transformací, který vyvíjejí studenti na katedře počítačové grafiky a interakce FEL ČVUT. Původní aplikace byla vytvořena v rámci studentské diplomové práce a nyní je vyvíjena nová vylepšená verze. Ta vyžaduje nový návrh a rozšíření 3D prohlížeče scény (3D viewportu) pro zobrazování a manipulaci scén.
1) Seznamte se s funkcemi a se stávající implementací nástroje I3T, s předchozí prací studentů [1,2] a výsledky uživatelského průzkumu ovládání 3D nástrojů [3].
2) Analyzujte stávající řešení 3D viewportu aplikace I3T a potřeby nové implementace: propojení s 2D částí a jádrem I3T, způsoby interakce, manipulátory a intuitivní výběr manipulované matice.
3) Navrhněte nové řešení 3D viewportu do stávající aplikace I3T.
4) Implementujte nový 3D viewport do aktuální verze aplikace I3T. K implementaci použijte C++ a knihovny Dear ImGui a OpenGL.
5) Navrhněte vhodná kritéria testování a implementovanou část aplikace otestujte.

[1] Michal Folta: Systém na výuku transformací, Diplomová práce, https://dspace.cvut.cz/handle/10467/64836
[2] Daniel Gruncl: Manipulátory pro editaci transformačních matic a skriptování v I3T, Bakalářská práce, https://dspace.cvut.cz/handle/10467/94657
[3] Vít Zadina: Testování užitečnosti nástroje pro výuku transformací, Bakalářská práce, https://dspace.cvut.cz/handle/10467/83400

Elektronicky schválil/a Ing. Radek Richtr, Ph.D. dne 22. února 2023 v Praze.

Bakalářská práce

PROHLÍŽEČ SCÉNY V APLIKACI I3T

Dan Rakušan

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Petr Felkel, Ph.D.
12. května 2023

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Rád bych poděkoval vedoucímu své bakalářské práce, Ing. Petru Felkelovi, Ph.D. za cenné připomínky a mnoho rad, vřelý přístup a především za dostatek trpělivosti při vytváření této práce. Dále bych chtěl poděkovat ostatním členům týmu I3T za veškerou spolupráci v průběhu práce. V neposlední řadě bych také rád vyjádřil vděk všem testerům, kteří se zúčastnili testování.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 12. května 2023 .....

Abstrakt

Práce rozšiřuje aplikaci na výuku grafických transformací I3T o nový 3D prohlížeč scény. Ten zobrazuje efekty transformačních matic, jež uživatel může flexibilním editorem matic interaktivně upravovat. Výsledný prohlížeč scény byl úspěšně integrován do stávající aplikace, byl rozšířen o známé funkce z ostatních 3D programů a obohacen možnostmi interakce s transformacemi přímo ve 3D scéně. Nové řešení bylo podrobeno uživatelskému testování použitelnosti a byla ověřena jeho schopnost nahradit zastaralejší verzi aplikace pro výuku transformací v předmětu Programování grafiky na ČVUT.

Klíčová slova počítačová grafika, 3D transformace, I3T, OpenGL, C++, 3D prohlížeč scény, Dear ImGui, 3D manipulátory

Abstract

I3T is an interactive tool for teaching 3D transformations. This thesis focuses on creating a new 3D scene view for this application which displays the effects of transformation matrices that the user can interactively modify using a flexible matrix editor. The resulting 3D scene view was successfully integrated into the existing application and expanded with well known features of other 3D programs. Furthermore, a new method of interaction with transformations directly in the 3D scene was added. The new solution was subjected to user usability testing, confirming its ability to replace an older version of the application for teaching transformations in the Graphics programming class at CTU.

Keywords computer graphics, 3D transformations, I3T, OpenGL, C++, 3D viewport, Dear ImGui, 3D manipulators

Úvod

Před každým studentem počítačové grafiky bude eventuálně stát téma takzvaných „grafických transformací“, sady transformací ze světa lineární algebry, kterými lze pohybovat objekty v grafické scéně či které umožňují takové scény vůbec na našich obrazovkách zobrazit. Tyto transformace mají typicky podobu 4×4 matic, které je možné pomocí maticového násobení skládat a následně jimi násobit libovolné souřadnice nebo vektory, a tím je v nějakém prostoru přesunout či jinak měnit. Například lze vynásobit všechny vrcholy nějakého 3D modelu, a tím ho ve scéně posunout na danou pozici či pootočit.

Manipulace 3D objektů pomocí matice s 16 čísly je typicky pro nové studenty poměrně neintuitivní a často mají problémy tématu porozumět. V počítačové grafice může mnohdy s pochopením tématu pomoct vizualizace. Tu dokáže ještě povznést možnost přímé interakce, která studentovi pomůže budovat potřebné porozumění a intuici. Takovou míru interakce tradiční média jako učebnice či články těžko dosáhnou a užitečných výukových materiálů s možností interakce není mnoho [1, s. 3].

Proto v roce 2016 vznikla na Fakultě elektrotechnické ČVUT v rámci studentské diplomové práce Michala Folty aplikace jménem I3T [1]. Ta studentovi na jedné části obrazovky představí editor zmíněných transformačních matic a na té druhé zobrazí 3D scénu, která dynamicky reaguje na jakékoli úpravy matic a zobrazí jejich účinek na 3D modely. Aplikace byla po jejím vzniku aktivně zařazena do výuky předmětu Programování grafiky na Fakultě elektrotechnické i na Fakultě informačních technologií ČVUT.

Nicméně se ukázalo, že aplikace I3T měla své vady. Hlavní kritikou studentů bylo její zastarale působící uživatelské rozhraní, které se v jistých případech špatně ovládalo. Nevýhodou byla také možnost spuštění aplikace pouze na platformě Windows [2, s. 1]. Padlo tedy rozhodnutí aplikaci přepracovat.

Od roku 2020 je na katedře počítačové grafiky a interakce FEL ČVUT vyvíjena nová vylepšená verze, která má za cíl nejen vyřešit nedostatky původní aplikace, ale zároveň i rozšířit její funkcionalitu. Na této verzi I3T se během posledních let vystřídalo už několik studentů v podobě bakalářských i diplomových prací, případně jiných projektů. Přesto se však ještě nedostala do stavu, kdy by mohla být využívána k výuce a některé klíčové funkce původního I3T Michala Folty stále chybí.

Nedodělaný je také 3D prohlížeč scény aplikace, jímž se zabývá tato práce. Ten byl původně přímo přejatý z první verze aplikace I3T a následně přepsán v bakalářské práci Daniela Gruncla [3]. V práci byl vyvíjen poměrně odděleně od samotné aplikace I3T, jelikož ta byla stále v prvotní fázi vývoje, a nebyl jejím hlavním cílem. Práce se spíše soustředila na novou funkci takzvaných manipulátorů a prohlížeč scény byl vyvinut především pro

demonstraci jejích výsledků. Z tohoto důvodů byl výsledný prohlížeč scény velice strohý, nerobustně integrovaný do zbytku aplikace a některé potřebné funkce v něm zcela chyběly či nebyly dostatečně dodělané.

Hlavním cílem Danielovy práce byla tvorba manipulátorů, které slouží jako vizuální nástroje pro úpravu transformačních matic přímo v 3D prostoru scény. V první verzi aplikace I3T lze 3D scénu pouze ovlivňovat změnami v oddělené komponentě editoru matic. Manipulátory představují možnost ovlivňovat matice editoru přímo v prohlížeči scény, a tím vytvořit dvousměrné propojení matic transformací s 3D prostorem. To umožňuje zobrazit nejen efekt změny matice na objekty v prostoru, ale i naopak, jaký efekt mají určité operace v prostoru na čísla matice.

Tato práce má za úkol dostat prohlížeč scény do prakticky použitelného a kompletního stavu, aby mohl být nástroj zařazen do výuky a v nedaleké době i veřejně vydán. V práci je navrženo a realizováno nové řešení prohlížeče scény. Na práci Daniela Gruncla částečně navazuje implementací již předem prozkoumaných manipulátorů do nového řešení a zároveň jsou prozkoumány další nové způsoby propojení 3D scény s komponentou editoru matic.

V práci pro zachování již existující terminologie používané v projektu bude pro prohlížeč scény používán pojem *viewport*, případně *3D viewport*. A komponenta editoru matic bude nazývána *workspace*, případně *2D workspace* ve smyslu pracovního prostoru editoru matic. Ten je nazván 2D k jasnému odlišení od 3D světa zobrazeného viewportem. Tyto anglické pojmy, jako některé ostatní, budou v jistých případech co nejvíce smysluplně skloňovány. Tento přístup byl zvolen především, aby se v textu neobjevovaly přeložené pojmy, které nejsou v češtině obecně dobře ustálené.

Práce je rozdělena na dvě části: teoretická a praktická. V části teoretické je provedena analýza stávajícího řešení, analýza požadavků na nové řešení a rešerše potřebných témat. Část praktická se bude zabývat detailní implementaci navrženého řešení s použitím jazyka C++, grafického API OpenGL a GUI knihovny Dear ImGui.

Cíle práce

Hlavním cílem této práce je vytvořit pro novou verzi aplikace I3T nové řešení 3D prohlížeče scény (*3D viewportu*), které bude obsahovat všechny důležité funkce původní první verze I3T Michala Folty [1], aby mohla nová verze I3T tuto aplikaci nahradit pro účely výuky. Nové řešení je třeba robustně integrovat do uživatelského rozhraní i existujících systémů současné aplikace I3T, která je stále ve vývoji.

Vedlejším cílem je do nového řešení prohlížeče scény dle možností přidat i nové funkce či zlepšení, které budou mít za cíl vylepšit jeho použitelnost. Při hledání nových funkcionalit 3D pohledu se práce inspiruje existujícími programy, které mají podobné zaměření práce s 3D scénou, jako jsou například viewporty editorů herních enginů (Unreal Engine, Unity) nebo 3D nástrojů jako je Blender.

Dalším cílem je v novém řešení viewportu navázat na práci Daniela Gruncla a do aplikace integrovat manipulátory popsané v jeho práci. Práce také prozkoumá ostatní možnosti propojení 3D scény s editorem matic, které případně dále podpoří jejich dvousměrnou interakci.

Nakonec bude nové řešení podrobeno uživatelskému testováním použitelnosti. Čímž bude ověřena jeho schopnost nahradit původní I3T jako výukový nástroj a vyhodnocen přínos všech nových funkcí.

Kapitola 1
Teoretická část

1.1 První verze nástroje I3T

I3T je desktopová aplikace, jejíž první verzi vyvinul Michal Folta ve své diplomové práci [1]. Aplikaci tvoří dvě klíčové části: 2D editor matic (*workspace*) a 3D prohlížeč scény (*viewport*). Editor matic je realizován jako *node editor*[1]. Jedná se o 2D plochu, ve které lze vytvářet takzvané krabičky[2]. Krabičky mají vstupy a výstupy a lze je navzájem spojovat či vkládat do sebe. Uživatel může skládat transformace vložením krabiček transformací do krabičky „sekvence“ v požadovaném pořadí. Pořadí je možno jednoduše v sekvenci měnit pouhým přetažením krabiček transformací myší. Sekvenci je možné přiřadit 3D model, a tím zobrazit efekt její výsledné matice na body modelu. Celá situace je zachycena 3D prohlížečem scény, který z pohledu uživatelem ovládané kamery modely dle zadaných transformací zobrazuje.

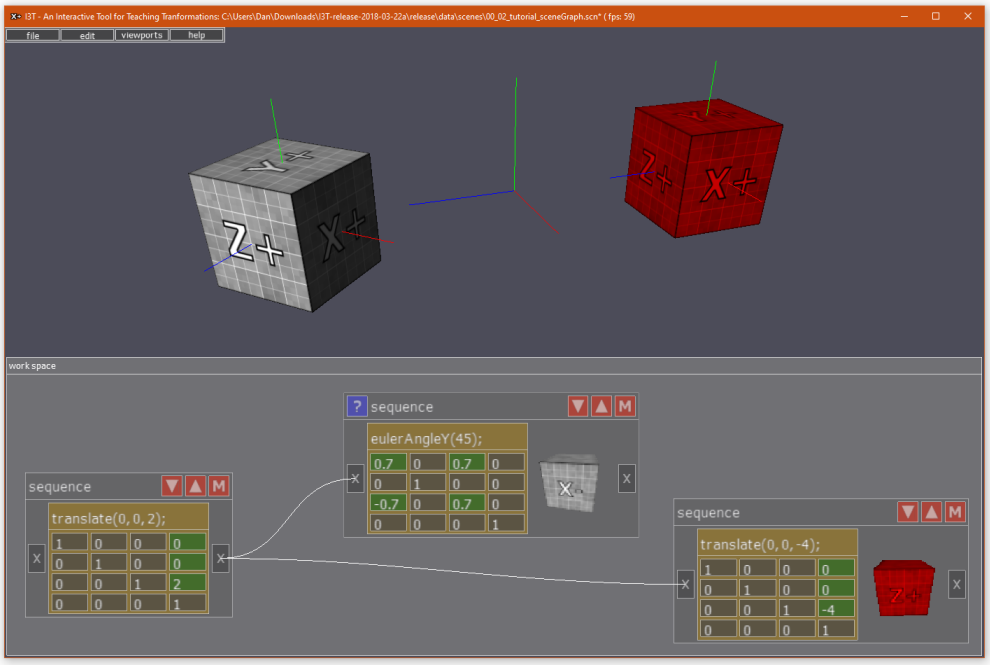
Sekvence transformací se mohou větvit a spojovat s dalšími sekvencemi a modely. Celkově sekvence tvoří „strom“ reprezentující graf scény (lépe řečeno orientovaný acyklický graf). Modelu sekvence se předá finální matice vytvořená vynásobením všech matic příslušné cesty mezi kořenem grafu scény a danou sekvencí. Jinými slovy, každá matice vypočítá svůj výstup zjištěním výstupní matice svého rodiče a zprava jí vynásobí svými vlastními „lokálními“ transformacemi.

Na obrázku 1.1 je příklad grafu scény v I3T. Zde každá sekvence obsahuje jednu transformaci a dvě z nich sdílí stejnou sekvenci jako rodiče. Sekvence šedé a červené kostky pracují se vztažnou soustavou sekvence vlevo, která svět posouvá o 2 jednotky po ose z. Šedá kostka je v této soustavě pootočena kolem osy y a červená kostka posunuta zpět mínus 4 jednotky po ose z. Červená kostka (respektive střed její vztažné soustavy) se tedy nakonec nachází na světové souřadnici $(0, 0, -2)$. Kdyby byla sekvence červené kostky odpojena od svého rodiče, červená kostka by se nacházela na souřadnicích $(0, 0, -4)$.

Na začátku řetězce transformací, před první sekvencí, se také může nacházet krabička kamery. Do té může uživatel vložit projekční a pohledovou matici a sledovat, kde se ve scéně kamera nachází a jak vypadá její pohledový objem (*frustum*). Na krabičku kamery lze napojit krabičku „Obrazovka“ (*Screen*), která zobrazí 3D scénu z pohledu dané kamery. Na obrázku 1.2 je příklad scény demonstrující vlastnosti ortografické projekce.

[1] Node editor je velice rozšířená GUI komponenta, bohužel nemá ustálený český název.

[2] Jak je nazývá Michal Folta ve své práci[1, s. 26]



Obrázek 1.1 Původní aplikace I3T Michala Folty. Ve workspace jsou vidět 2 sekvence sdílející stejného rodiče.

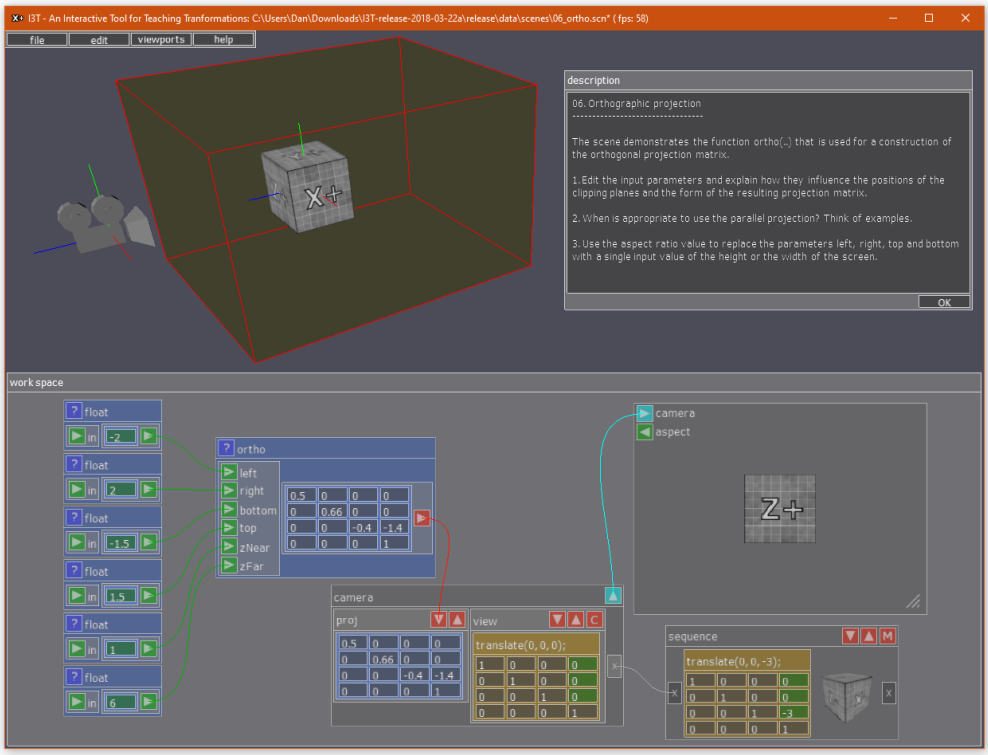
Spoje mezi krabičkami mohou také přenášet data. Sekvence má výstup typu matice, který obsahuje vynásobené pouze do ní vložené matice, bez transformace jejího rodiče. Tato datová spojení využívá hlavně typ krabiček nazývaný „operátory“. Operátory dokáží provádět složitější numerické operace jako například maticovou inverzi. Také zajišťují způsob, jak provádět matematické operace s čísly, vektory, maticemi a kvaterniony. Mezi jednotlivými datovými typy také umožňují provádět konverzi. Operátory jsou například využity ve scéně na obrázku 1.2 k sestavení matice ortografické projekce z jednotlivých parametrů pozic rovin pohledového objemu.

Na rozdíl od ostatních existujících výukových programů vytvořených pro demonstraci základních konceptů počítačové grafiky se I3T nesoustředí na specifické situace se statickým zadáním [1, s. 3]. Naopak I3T pomocí svého flexibilního systému krabiček dokáže vyobrazit širokou škálu příkladů. Vytvořené scény v I3T jde uložit a následně načíst ze souboru, což umožňuje scény lehce sdílet. Do prostředí workspace je možné vložit poznámky a v nové rozpracované verzi I3T přibyla i podpora tutoriálů, které uživatele provedou příklady přímo v rozhraní programu.

1.2 Analýza stavu nové verze aplikace na začátku práce

1.2.1 Předchozí práce studentů

Nová verze aplikace měla za cíl vyřešit hlavní nedostatky první verze I3T, zejména její uživatelské rozhraní a neudržitelný kód. Vývoj nezačal zcela od začátku, ale skoro všechny



Obrázek 1.2 Scéna první verze I3T demonstrující kameru s ortografickou projekci. Pro vytvoření projekční matice jsou využity krabičky operátorů (modré krabičky).

části aplikace byly přepsány. V minulosti na ní již pracovalo 10 studentů. Pro úplnost a zasazení do kontextu je zde stručně uveden seznam autorů všech předchozích prací a jejich příspěvků:

- **Michal Folta, diplomová práce, 2015–2016**
  Původní tvůrce aplikace.

- **Lukáš Pilka, školní projekt, 2017**
  Prvotní návrhy a testování UI a vizuální identita I3T.

- **Vít Zadina, bakalářská práce, 2019**
  Testování užitečnosti nástroje pro výuku transformací. Zadina prozkoumával např. oblíbenost klávesových zkratek a navrhnul podobu nového UI.

- **Marek Nechanský, bakalářská práce, 2019**
  Automatické rozmísťování krabiček.

- **Filip Uhlík, bakalářská práce, 2019**
  Logovací framework. Systém na zaznamenávání uživatelské interakce.

- **Mirek Müller, semestrální projekt, 2020–2021**
  Prototyp nového uživatelského prostředí, zejména dokovatelných oken. Vytvořil také hlavní okno tutoriálů.

- **Martin Herich, bakalářská práce, 2021**
  Restrukturalizace interaktivního nástroje pro výuku transformací I3T a reimplementace grafického rozhraní pomocí knihovny Dear ImGui.
- **Daniel Gruncl, bakalářská práce, 2021**
  Přepracování viewportu a přidání manipulátorů, též se věnoval podpoře skriptování.
- **Sofie Šašorina, bakalářská práce, 2021**
  Částečná implementace UI některých krabiček.
- **Jaroslav Holeček, diplomová práce, 2023**
  Reimplementace GUI krabiček (vytvořil knihovnu DIWNE[3]) a teoretický základ pro automatické nabízení úloh dle znalostní úrovně studenta.

Pro tuto práci jsou především relevantní práce Marina Hericha [2], Jaroslava Holečka [4], Víta Zadiny [5] a Daniela Gruncla [3].

Marin Herich vytvořil nové jádro aplikace a velkou část uživatelského rozhraní. Součástí práce Jaroslava Holečka byl vývoj knihovny DIWNE, která realizuje uživatelské rozhraní editoru matic. Vít Zadina provedl ještě před začátkem implementace nové verze návrh uživatelského rozhraní aplikace a testoval užitečnost UI prototypu aplikace. Jeho práce se viewportu věnovala velice okrajově, provedl však uživatelský průzkum klávesových zkratek, který je v této práci využít. Daniel Gruncl, jak již bylo zmíněno, pracoval na implementaci předchozího viewportu a manipulátorů.

Na začátku práce se na vývoji I3T aktivně podíleli 3 studenti:

- Martin Herich spravuje git repozitář projektu a příležitostně na něm pracuje v rámci své budoucí diplomové práce.
- Jaroslav Holeček na své diplomové práci stále pracuje, ale relevantní knihovna DIWNE, která je její součástí je již dokončena.
- Adam Loucký nově pracuje na rozšíření tutoriálů souběžně s touto bakalářskou prací.

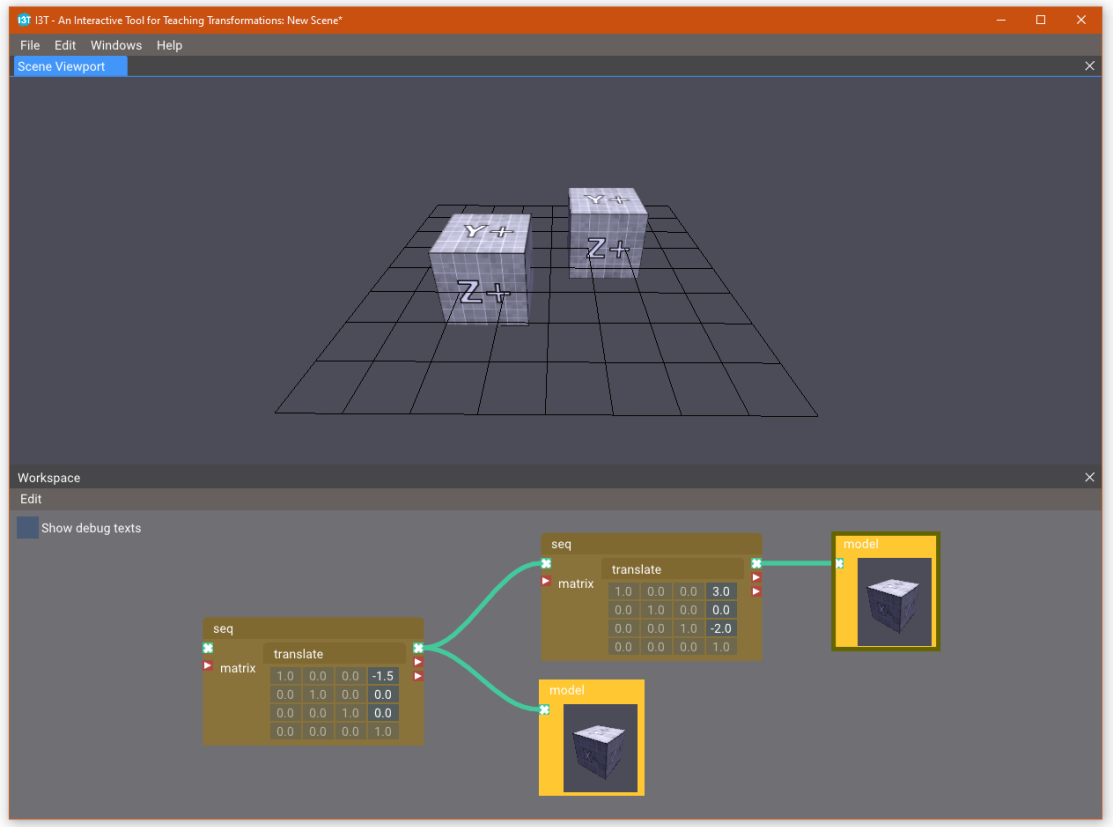
Během této práce každý týden probíhaly konzultace s vedoucím a zbytkem týmu I3T, na kterých byly společně probrány jakékoli poznatky, pokrok a následující postup. Na těchto konzultacích bylo možné ostatním prezentovat jakékoli nové funkce a získat odezvu.

1.2.2 Stav aplikace a prohlížeče scény

Na začátku této práce je nová verze I3T stále ve stádiu vývoje a není ještě pro výuku použitelná. Většina hlavních částí aplikace, jako logický systém krabiček, uživatelské rozhraní krabiček a okenní systém, je dokončeno, ale obsahují mnoho menších problémů a chyb, které jsou postupně týmem I3T opravovány. Podoba aplikace na začátku této práce je vidět na obrázku 1.3.

Ve 2D workspace se například vyskytují chyby při mazání transformací v sekvencích. Ostatní napojené krabičky na mazání nereagují, jelikož efekt transformace stále v sekvenci po smazání přetrvává. Některé neošetřené případy spojování krabiček, jako zapojení sekvence sama do sebe, způsobují spadnutí programu. Po smazání sekvence přetrvávají její spojení na již neexistující krabičku a podobně.

[3] *Dear IMGUI Wrapper Node Editor* – implementace node editoru pomocí knihovny Dear ImGui



Obrázek 1.3 Podoba nové rozpracované verze I3T v době začátku této práce.

Co je však zásadně nedokončené, je existující řešení prohlížeče scény. To bylo vyvinuto v práci Daniela Gruncla [3] pro účely implementace nové funkcionality manipulátorů. Tato implementace vznikla v době, kdy byl zbytek aplikace stále v brzkém stádiu vývoje, a proto její vývoj probíhal samostatně a odděleně od zbytku aplikace. Cílem jeho práce bylo také navrhnout a implementovat manipulátory, nikoli nutně vytvořit nové řešení prohlížeče scény pro I3T. Jeho implementace je tedy pro potřeby I3T poněkud nedostačující a v době začátku této práce prakticky nepoužitelná.

V prohlížeči scény chybí mnoho důležitých funkcí. Chybí možnosti krabičky modelu, jako změna 3D modelu nebo viditelnosti, a při jejím smazání ze 3D scény nezmizí její model. Modely jsou v kódu natvrdo definované a nenačítají se ze souborů. Krabičce kamery zcela chybí reprezentace ve scéně a vytvoření krabičky obrazovky způsobí pád programu. Některé manipulátory, jako škálování a obecná rotace, nefungují a manipulátory nelze v krabičce sekvence skládat. Kamera scény se podivným způsobem točí kolem středu scény, i když se na něj nedívá a nepodporuje předdefinované pohledy.

Dalších problémů je mnoho, a nepochybně mnoho z nich plyne z polopatické integrace existujícího prohlížeče scény do zbytku aplikace, která proběhla až po ukončení práce Daniela Gruncla.

Již před zahájením této práce bylo rozhodnuto, že prohlížeč scény vyžaduje nové řešení. S potřebou nového řešení přímo zadání této práce počítá a analýzou předchozího řešení se v textu nebude dále zabývat.

1.2.3 Architektura nové verze aplikace I3T

Hlavní změnou z hlediska uživatele oproti první verzi I3T je využití knihovny Dear ImGui[4] pro uživatelské rozhraní aplikace. Návrh krabiček se také lehce změnil, aby byly více přehledné [5, s. 25] a více se podobaly ostatním konvenčním node editorům, na které mohou být uživatelé zvyklí z ostatních programů.

Došlo k zásadní změně struktury projektu. Původní kód první verze I3T byl vysoce provázaný a bylo těžké provádět změny v kódu bez nečekaných vedlejších efektů. Například uživatelské rozhraní krabiček bylo přímo navázané na logiku aplikace vyhodnocující matematické výrazy [2, s. 6].

V novém projektu nejsou jasně definovány typické architektonické vrstvy. Zdrojový kód aplikace je rozdělen do složek, které shlukují související zdrojové soubory a tvoří co nejméně provázané a co nejvíce soudružné moduly. Struktura modulů je znázorněna diagramem balíčku 1.4. Mezi balíčky jsou šipkami znázorněny jejich závislosti a v diagramu jsou také vidět některé ostatní moduly, které nejsou pro tuto práci zcela důležité.

Důležitou částí aplikace je její jádro (balíček *Core*). To obsahuje v balíčku *Core/Nodes* implementaci grafu scény, která se používá k vyhodnocení akcí uživatele ve workspace. Tento balíček je zcela oddělen od uživatelského rozhraní a je převážně samostatný.

V jádře se dále nachází třída *Application* reprezentující celou aplikaci, která má na starosti její hlavní smyčku (*game loop*), její okno a centralizovaný přístup k jejím ostatním modulům.

V balíčku *Core/Input* se nachází manažer pro uživatelský vstup. Ten přechovává a sbírá pomocí knihoven GLFW[5] a Dear ImGui.

Balíček *Core/Resources* má na starost správu zdrojů jako 3D modely, shadery a textury. Tento balíček využívá pro reprezentaci a načítání zdrojů existující interní knihovnu *pgr-framework*, která je určena jako základ pro úlohy studentů na cvičeních předmětu PGR (Programování grafiky). Tato knihovna je pro potřeby této aplikace zcela nevhodná, jelikož je jejím cílem pouze usnadnit práci studentům. Její reprezentace materiálů, textur i geometrie je pevně daná a nelze jí pro potřeby I3T nijak upravovat. Celý tento balíček bude třeba od základu předělat.

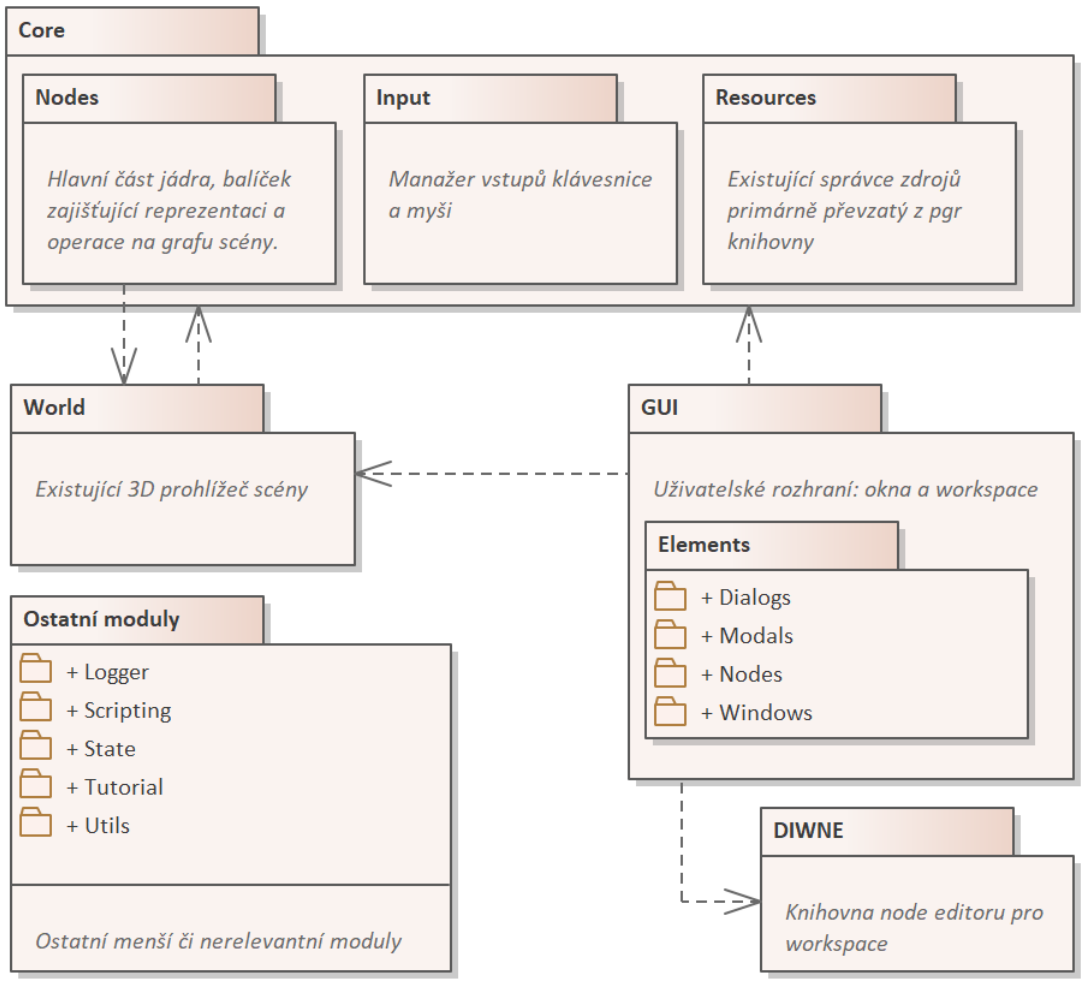
Jádro není nijak zásadně abstrahované od použitých knihoven jako GLFW, Dear ImGui či OpenGL. V této práci budou tyto knihovny považované za neměnné. Objevuje se v něm také široké využití globálních statických tříd a singletonů, které nejsou zcela vhodné a mohou způsobovat problémy [6, s. 335]. Těm se ani nové řešení nevyhne, jelikož na nich jsou již závislé velké části aplikace, a revize celkové architektury není předmětem práce.

Balíček *GUI* obsahuje implementaci uživatelského rozhraní pomocí knihovny Dear ImGui. Uživatelské rozhraní aplikace využívá systém dokovacích oken, který se podobá okenním systémům vývojových prostředí jako je Visual Studio nebo IntelliJ IDEA. Hlavní okno aplikace obsahuje prázdnou plochu, do které lze další okna vložit. Okna mohou být umístěna kdekoli, i mimo hlavní okno aplikace. Do jedné poloviny libovolného okna může být vloženo okno jiné nebo se okna mohou vkládat do sebe jako záložky. Své okno má workspace i viewport. Toto je velice vítaná změna oproti první verzi I3T, kde byl viewport vždy v pozadí hlavního okna a workspace byl pevně umístěn v dolní části obrazovky.

V balíčku *GUI* je také implementována UI vrstva *node editoru* workspace. V ba-

[4] Open source GUI knihovna https://github.com/ocornut/imgui

[5] Graphics Library Framework – https://en.wikipedia.org/wiki/GLFW



Obrázek 1.4 Diagram balíčků znázorňující relevantní strukturu kódu I3T.

líčku *GUI/Elements/Nodes* se nachází jednotlivé třídy krabiček, které jsou postaveny na knihovně *DIWNE*[6], kterou vyvinul pro účely I3T ve své práci Jaroslav Holeček [4, s. 92] a nachází se v odděleném balíčku *DIWNE*.

Posledním důležitým balíčkem je balíček *World* existujícího prohlížeče scény, který má tato práce za cíl nahradit.

1.3 Analýza nového řešení

V této sekci budou nejprve na základě předchozí analýzy a rozhovorů s vedoucím práce specifikovány funkční a nefunkční požadavky na nové řešení.

[6] Akronym pro „Dear IMGUI Wrapper Node Editor“

1.3.1 Cílové skupiny uživatelů

Nepochybně je dobré nejprve shrnout, pro jaké uživatele je aplikace I3T vlastně určena. Tomu se již velice hluboce věnoval Vít Zadina ve své bakalářské práci a skupiny, které definoval zde budou krátce shrnuty.

Zadina uživatele řadil do čtyř cílových skupin [5, s. 33] seřazených od nejméně četných po nejčetnější:

- **Učitelé**
  Učitelé používají aplikaci I3T k výuce. Mohou ji použít například na přednáškách k vysvětlení dané problematicky týkající se transformací. Jednotlivé příklady sami vymýšlí a uvítají v I3T snadné ovládání při tvorbě výukových scén a tutoriálů. Vytvořené scény mohou studentům dát jako práci na cvičení nebo je zadat k samostudiu doma.

- **Studenti**
  Studenti jsou rozděleni do dvou skupin:

  - **Samostudium**
    Studenti si předpřipravené scény nebo tutoriály mohou spustit doma k rozšíření svých znalostí nebo jako zadané domácí úlohy. Případně mohou v programu vytvářet jednoduché scény a vyzkoušet si na nich některé situace, se kterými se setkali ve své semestrální práci.

  - **Na cvičení**
    Na cvičení studenti typicky dostanou předpřipravenou scénu a řídí se pokyny cvičícího. Pokud jim něco není jasné, mohou se zeptat.

- **Programátoři**
  Programátoři jsou zkušení uživatelé, kteří mohou I3T používat k vyobrazení některých složitějších situací, se kterými se setkali například při vývoji nějaké grafické aplikace. Případně mohou vytvářet scény za účelem vysvětlení dané problematiky někomu jinému podobně jako učitelé.

- **Kdokoli se zájmem o grafiku**
  Tato skupina je nejobecnější. Jedná se o uživatele, které zajímá problematika grafických transformací. Nemusí se s I3T setkat ve školním kontextu, ale například na aplikaci I3T narazili na internetu, když si četli článek o nějaké problematice, ve kterém byla nasdílena I3T scéna demonstrující daný problém.

Detailnější popis, který obsahuje i persony každé ze skupin lze nalézt v práci Víta Zadiny [5] na straně 33.

1.3.2 Požadavky na nové řešení

Tato podsekce položí a detailně popíše všechny jednotlivé funkční a nefunkční požadavky. Funkční požadavky jsou zkráceně pojmenovány ve tvaru „F*XX*“, kde písmena X jsou nahrazena číslem požadavku. Podobně jsou pojmenovány i nefunkční požadavky ve tvaru „NXX“.

Funkční požadavky jsou obecně kategorizovány do tří skupin:

Původní: Funkcionality, které obsahovalo původní I3T Michala Folty, a nijak se zásadně nezměnily. Jejich implementace se může mírně lišit z povahy nového řešení, ale z pohledu uživatele se jedná o stejné funkce.

Rozšíření: Funkcionality, které jistým způsobem původní I3T obsahovalo, ale budou zásadně rozšířeny nebo zcela přepracovány.

Nové: Zcela nové funkcionality, které původní I3T neobsahovalo.

V tabulce 1.1 jsou shrnuty všechny funkční požadavky a v tabulce 1.2 jsou shrnuty všechny požadavky nefunkční.

Tabulka 1.1 Funkční požadavky

Zkratka	Název	Skupina	Priorita
F01	Reprezentace krabičky „Model“ ve 3D scéně	Původní	Vysoká
F02	Reprezentace krabičky „Kamera“ ve 3D scéně	Původní	Vysoká
F03	Vykreslení pohledu kamery pro krabičku „Obrazovka“	Původní	Vysoká
F04	Zobrazení bazických vektorů transformace	Původní	Vysoká
F05	Zobrazení 3D modelu	Původní	Vysoká
F06	Zobrazení primitiv	Původní	Vysoká
F07	Základní osvětlovací model	Původní	Vysoká
F08	Načtení 3D modelu ze souboru	Rozšíření	Vysoká
F09	Průhlednost	Rozšíření	Střední
F10	Vykreslení 3D scény nezávislou kamerou	Původní	Vysoká
F11	Uživatelské rozhraní viewportu	Rozšíření	Vysoká
F12	Správa 3D modelů	Nové	Střední
F13	Manipulátory	Nové	Vysoká
F14	Selekce objektu	Nové	Střední
F15	Zvýraznění obrysu 3D modelu	Nové	Střední
F16	Zobrazení vlivu krabiček	Nové	Střední
F17	Světová mřížka s osami	Nové	Nízká
F18	Indikátor světových os	Nové	Nízká

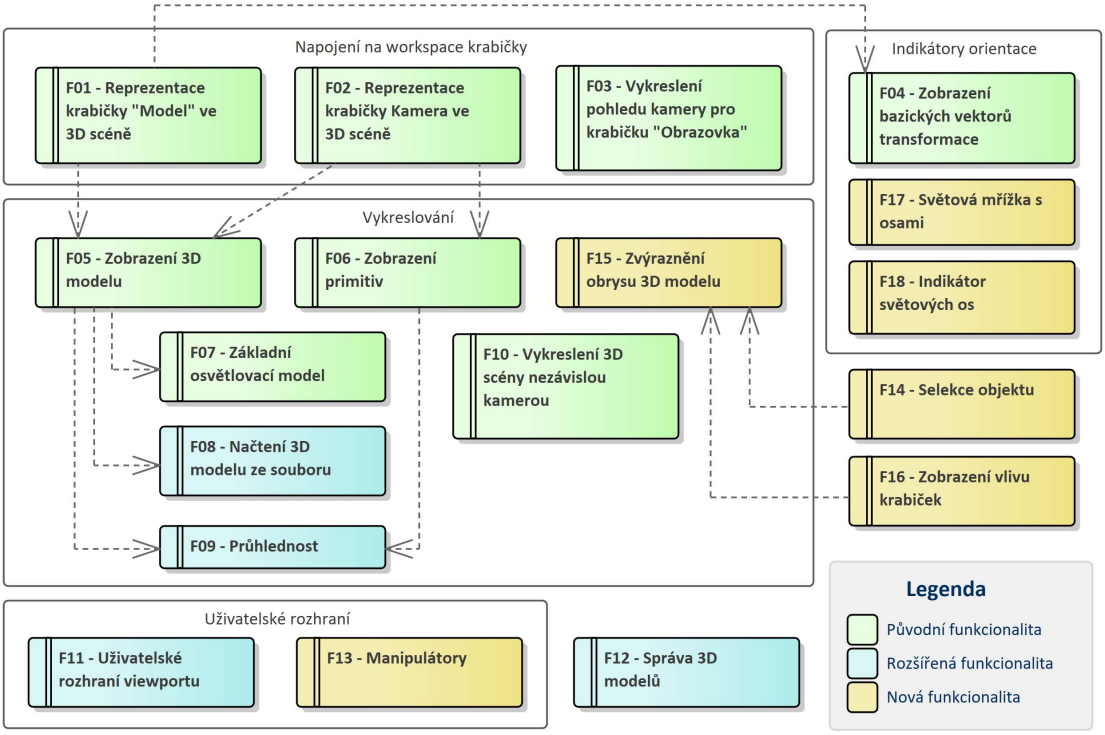
Funkční požadavky jsou také shrnuty v diagramu na obrázku 1.5, ve kterém jsou požadavky seskupené do logických celků a jsou mezi nimi naznačeny některé významné závislosti.

V následujícím textu budou jednotlivé požadavky detailně popsány. Mimo popis samotných funkčních požadavků je u mnohých uvedeno jejich odůvodnění, a popřípadě i porovnání s původní aplikací I3T. Požadavky také mohou odkazovat na jeden či více dalších požadavků, které blíže specifikují relevantní funkci.

***F01* - Reprezentace krabičky „Model“ ve 3D scéně** Krabička „Model“ z workspace musí mít odpovídající reprezentaci modelem ve 3D scéně. Původní I3T obsahovalo modely

Tabulka 1.2 Nefunkční požadavky

Zkratka	Název
N01	Nízké hardwarové požadavky
N02	Vysoká kompatibilita
N03	Možnost budoucí serializace
N04	Jednoduché uživatelské rozhraní
N05	Kvalita obrazu



Obrázek 1.5 Diagram funkčních požadavků

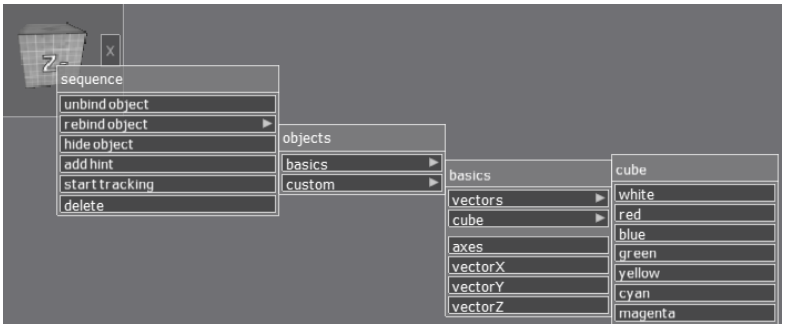
přímo zabudované do krabičky „Sekvence“. V novém návrhu Víta Zadiny byly modely od sekvence odděleny do samostatné krabičky „Model“, kterou lze se sekvencí propojit násobením matic [5, s. 49]. Na obrázku 1.6 je vidět rozbalené kontextové menu krabičky sekvence původní aplikace I3T. Jelikož byl model její součástí, v tomto menu se mísily možnosti sekvence s možnostmi modelu. Reprezentace krabičky ve scéně je vždy vykreslována s použitím příslušné grafické transformace, kterou má krabička „Model“ na vstupu.

V novém řešení má samostatná krabička „Model“ své vlastní kontextové menu. To bude obsahovat možnost skrýt model z 3D scény. Společně s 3D modelem se budou na příslušném místě vykreslovat i bazické vektory transformace modelu (***F04***).

Krabička vždy zobrazuje jeden 3D model. Tento model bude možné změnit z nabídky již přednačtených výchozích 3D modelů, které jsou součástí každé scény. Uživatel také bude mít možnost do aktuální I3T scény načíst jeho vlastní model ze souboru (***F08***).

Jak je vidět na obrázku 1.6, v původním I3T měly základní modely jako např. vektory

a kostky několik variant s různým obarvením textur (pouze částečné přimíchání dané barvy do barev textury). Toto bude v novém řešení zobecněno na všechny modely jako samostatná možnost volitelná pro všechny (i vlastní) modely. Díky tomu půjde jednoduše změnit barvu specifického modelu a odlišit ho tím od instancí stejného modelu ve scéně.



Obrázek 1.6 Kontextové menu krabičky sekvence původní aplikace I3T s rozbaleným menu změny modelu

Krabička dále bude mít možnost svůj 3D model ve scéně zobrazit průhledně. V komplikovanějších scénách je pro jasnost užitečné některé modely částečně zprůhlednit, aby potenciálně nepřekrývaly důležité části scény. Tato funkce je blíže popsána v požadavku ***F09***.

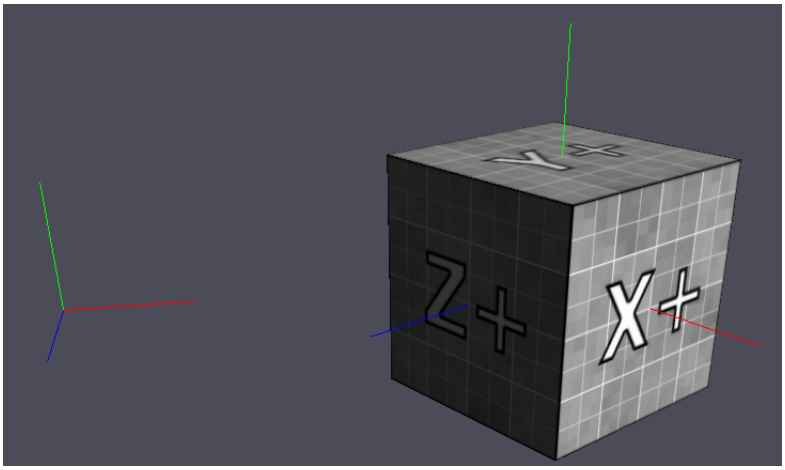
Nakonec by přímo uvnitř krabičky v rozhraní 2D workspace měl být zobrazen malý náhled zachycující podobu objektu ve viewportu. Tento náhled je třeba vykreslovat v nízkém rozlišení a snížené kvalitě. Není třeba se přílíš zabývat jeho optimalizací, viz ***N01***.

F02 **- Reprezentace krabičky „Kamera“ ve 3D scéně** Dle matic krabičky kamery je nutné na správné pozici zobrazit 3D model kamery. Objektiv kamery bude orientován opačným směrem než směrový vektor její pohledové matice (tedy bude namířen na cíl pozorování). Odděleně od kamery se také vykreslí poloprůhledná reprezentace pohledového objemu kamery (***F09***). Modelu i frustu kamery půjde z kontextového menu krabičky zakázat či povolit vykreslování. V tomto menu bude také možnost změnit barvu vykreslovaného frusta k případnému odlišení od fruster ostatních kamer. Stejně jako u modelu se na pozici kamery budou vykreslovat bazické vektory kamery (***F04***). Ty jsou v tomto případě definovány inverzí pohledové matice kamery.

F03 **- Vykreslení pohledu kamery pro krabičku „Obrazovka“** Do krabičky kamery lze připojit krabičku „Obrazovka“, která ve svém obsahu musí zobrazit aktuální pohled na 3D scénu dané kamery. Ta je definována pohledovou a projekční maticí.

F04 **- Zobrazení bazických vektorů transformace** Všechny modely a kamery ve scéně by měly mít volitelně vyobrazené 3 bazické vektory jejich aktuální transformace. Ty stačí vykreslovat jako 3 různobarevné úsečky o jednotkové délce.

F05 **- Zobrazení 3D modelu** Viewport musí být schopen zobrazit 3D model ve scéně dle zadané grafické transformace. Modely by měly být osvětlené a otexturované.



Obrázek 1.7 Vyobrazení bazických vektorů celého světa (vlevo) a modelu (vpravo) v původní aplikaci I3T.

***F06* - Zobrazení primitiv** Ve scéně je třeba vykreslovat i parametricky zadané úsečky, trojúhelníky a plošky pro případy kdy není vhodné pro dané účely vytvářet individuální 3D modely pro načtení ze souboru. Tuto funkci lze využít pro vykreslování úseček bazických vektorů transformací nebo vykreslování pohledových objemů kamer.

***F07* - Základní osvětlovací model** 3D modely ve scéně musí mít alespoň základní per-pixel Phong osvětlení.

***F08* - Načtení 3D modelu ze souboru** K zobrazení libovolného modelu je třeba modely načíst ze souboru a převést do datové podoby, kterou viewport bude schopen vykreslit. Podporované by měly být 3D formáty specifikace glTF 2.0 (přípony *.gltf* a *.glb*).

Modely, které aplikace bude načítat, budou typicky obsahovat textury. Formát glTF obsahuje potřebné informace o texturách modelu. Textura může být uložena přímo v glTF souboru v binární či Base64 kódované podobě. Také však může být textura uložena jen jako relativní cesta odkazující na samostatný soubor [7]. Je třeba, aby například v případě přesunu souboru modelu aplikací I3T došlo i k přesunu všech takto relativně navázaných zdrojů.

F09* - Průhlednost** V požadavcích ***F01 a ***F02*** je vyžadováno vykreslování poloprůhledných objektů, které v OpenGL není zcela triviální. Z povahy kamerových fruster je nutné vykreslovat poloprůhledné objekty, které se navzájem protínají (v případech, kdy se protínají objekty reprezentující pohledové objemy kamer).

Standardní vykreslovací technika seřazení poloprůhledných objektů, která je například vyučována v předmětu PGR, si s některými poloprůhlednými objekty nedokáže poradit bez vizuálních artefaktů. Nové řešení prohlížeče scény by mělo obsahovat techniku průhlednosti, která nevyžaduje seřazení primitivů geometrie a vyhodnocuje průhlednost zvlášť pro každý fragment. Skupina takových technik se nazývá *Order-independent transparency* (OIT). Tomuto tématu se dále věnuje sekce 1.4.2.

***F10* - Vykreslení 3D scény nezávislou kamerou** V hlavním okně viewportu je 3D scéna zobrazena nezávislou kamerou, kterou může uživatel ovládat. Tato kamera není součástí grafu scény a nemá nic společného s krabičkou kamera (proto je „nezávislá“). Tato kamera bude podporovat základní funkce, které mělo původní I3T. Půjde přepínat mezi módy *orbit* a *trackball*. V obou módech se kamera točí kolem zadaného bodu, který lze tažením myši posunout. Tažením jiného tlačítka myši se bude kamera kolem tohoto bodu točit. V módu *trackball* se kamera bude točit jako by byla napojená na virtuální trackball umístěný uprostřed obrazovky. V módu *orbit* se kamera kolem bodu točí, ale nikdy nedojde k rotaci po ose jejího pohledu. Pozice kamery v tomto módu lze jednoduše definovat úhlem azimutu a elevace.

Kamera bude také mít možnost přepnout na jeden z přednastavených pohledů, které kameru natočí tak, aby její směr pozorování byl souběžný s vybranou světovou osou. Kameru rovněž půjde ve scéně vycentrovat tak, aby její cíl pozorování byl zdánlivý střed scény nebo světa. To je užitečné, když se uživatel ve scéně „ztratí“ a neví jak se dostat zpět.

***F11* - Uživatelské rozhraní viewportu** Viewport je třeba začlenit do uživatelského rozhraní I3T. Knihovna Dear ImGui dokáže vykreslit na zadaném místě existující OpenGL texturu. Nové řešení viewportu musí tuto texturu poskytnout. Mimo vykreslení samotného viewportu je také nutné vytvořit pomocí standardních funkcí knihovny Dear ImGui uživatelské rozhraní, které uživateli umožní upravovat nastavení jednotlivých funkcí viewportu. Toto by mělo být provedeno minimálně v podobě kontextových menu v horní liště okna viewportu, podobně jak to je v okně workspace.

F12* - Správa 3D modelů** V aplikaci by měl být systém, který poskytuje přístup k potřebným prostředkům viewportu (modely, textury, shadery) a je zodpovědný za jejich životní cyklus. Prostředky by měly být udržovány v mezipaměti, aby se zamezilo zbytečnému opětovnému načítání. Tento systém by měl umožnit rozlišovat mezi výchozími a vlastními prostředky (viz ***F01). Původní I3T takový systém mělo, ale využíval pro geometrii modelů, textury i materiály vlastní řešení založené na konfiguračních souborech. Nová verze I3T by měla pro všechny tyto účely využít již existující robustní formát, jak je popsáno v požadavku ***F08***.

***F13* - Manipulátory** Nový viewport musí obsahovat funkci manipulátorů, která by měla vycházet z již provedeného návrhu manipulátorů z práce Daniela Gruncla [3]. Manipulátory se budou ve 3D scéně zobrazovat při výběru transformace, kterou podporují. Mezi podporované transformace by měly patřit transformace translace, škálování, rotace, projekce a „lookAt“. Manipulátory vždy musí pracovat v prostoru, který je určen jim předcházejícími transformacemi v grafu scény.

***F14* - Selekce objektu** Ve viewportu bude možné kliknout na model, a tím ho vybrat. Při výběru 3D modelu zároveň dojde k výběru ekvivalentní krabičky ve 2D workspace. A stejně jako ve workspace se model ve 3D scéně vhodným způsobem zvýrazní, aby byl odlišen od ostatních nevybraných objektů.

Tato možnost vychází z ostatních 3D programů, ve kterých je možnost selekce objektů ve 3D scénách běžná. Při výběru a zvýraznění modelu ve 3D scéně je snadné identifikovat, o který model se jedná ve 2D workspace. Stejný princip funguje i naopak, kdy při výběru

krabičky modelu ve 2D workspace uživatel může snadno najít odpovídající model ve 3D scéně.

F15* - Zvýraznění obrysu 3D modelu** Pro implementaci požadavku ***F14 je nutné, aby mohly být 3D modely ve vykreslované scéně vhodným způsobem zvýrazněny. Ve 2D workspace se krabičky zvýrazňují barevným obrysem kolem krabičky. Pro jednoduchou korelaci mezi vybraným prvkem ve 2D workspace a vybraným prvkem ve 3D scéně by měly být objekty ve 3D scéně zvýrazněny podobným způsobem, tedy vykreslením barevného obrysu kolem 3D modelu.

F16* - Zobrazení vlivu krabiček** Zvýraznění objektů ve 3D scéně z požadavku ***F15 při selekci z požadavku ***F14***

Funkce zvýrazňování objektů z požadavku ***F15*** lze využít také k zvýrazňování jiných modelů než jen těch vybraných. Pokud je krabička transformace napojena přímo či nepřímo na krabičku modelu, může transformaci tohoto modelu ovlivnit. Pro danou krabičku lze z grafu scény zjistit, kterých modelů se změna jejích hodnot může týkat. Množinu těchto modelů je užitečné vizualizovat, a právě k tomu může být navíc využita funkce zvýrazňování modelů ve 3D scéně.

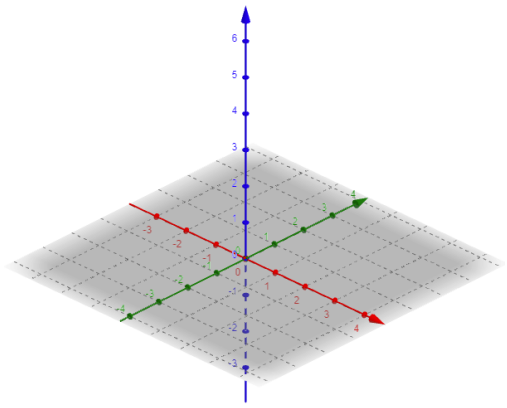
Při selekci jakékoli krabičky by tedy měly být v novém viewportu obrysem zvýrazněny všechny modely, na které je vybraná krabička napojena.

***F17* - Světová mřížka s osami** Prostředí I3T je úzce spjato s čísly. Pokud chce například uživatel translační maticí posunout objekt na nějaké místo, je dobré, aby předem zhruba věděl, o kolik jednotek je třeba objekt posunout, aby tak mohl takové číslo rovnou zadat do matice transformace bez zbytečného hádání. Podobně je dobré mít odhad, na jakých souřadnicích se objekty ve scéně nachází nebo kde přesně je střed světa $(0, 0, 0)$. Ve středu světa vykreslovala první verze aplikace I3T vektory standardní báze. Tím bylo uživateli napovězeno, kterým směrem jsou orientované jednotlivé osy světa. Tyto vektory však ve vzdálenosti jedna od středu světa končily. Tudíž orientaci vzdálených objektů vůči světovým bázím bylo těžké určit.

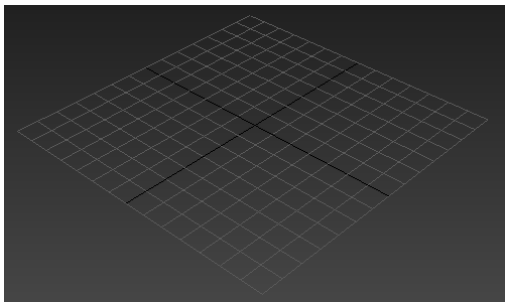
V těchto situacích se hodí mít ve scéně mřížku s osami. Mřížky jsou v prostředích grafických kalkulaček téměř vždy v nějaké formě dostupné, právě kvůli jejich blízké souvislosti s numerickými operacemi. Příkladem je program GeoGebra, který zobrazuje i u všech os číselné popisky (Obrázek 1.8). První verze I3T mřížku vůbec neobsahovala a do nového I3T Daniel Gruncl přidal mřížku ve formě předpřipraveného 3D objektu, vykreslovaného pomocí linek. Podobně mřížku řeší například program *Autodesk 3ds Max* (Obrázek 1.8).

Problém nastává, když chce uživatel pracovat s objekty, které jsou od mřížky vzdáleny. Mimo mřížku se dostáváme zpět do situace, kdy je světové vzdálenosti opět těžké odhadnout. Navíc může mřížka uživateli vizuálně připomínat zemní rovinu, podle které se může orientovat. U takové roviny se dá očekávat, že bude pokračovat do nekonečna a tvořit v dáli horizont.

Právě takové mřížky se v mnoha moderních 3D programech běžně objevují a jsou vhodné pro využití v I3T. Používá je například open-source program Blender, se kterým bývají studenti ČVUT dobře obeznámeni nebo také herní engine Unity. Na obrázku 1.9 jsou vidět mřížky v těchto programech. Společnou vlastností bývá postupné mizení mřížky se zvyšující se vzdáleností od kamery a vícero úrovní velikosti čtverečků mřížky. Podobně realizovanou mřížku by mělo obsahovat i nové řešení I3T.

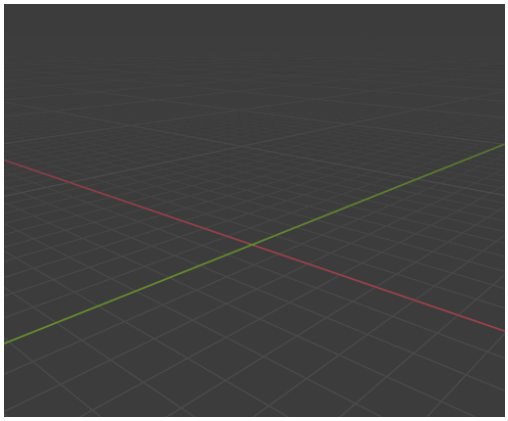


(a) GeoGebra

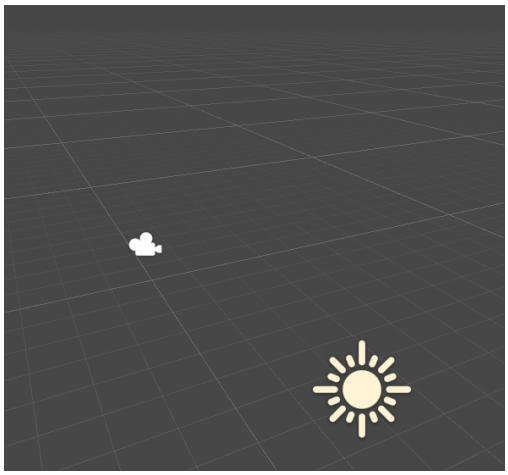


(b) 3ds Max (2014)

Obrázek 1.8 Mřížky v programech GeoGebra a 3ds Max



(a) Blender



(b) Unity

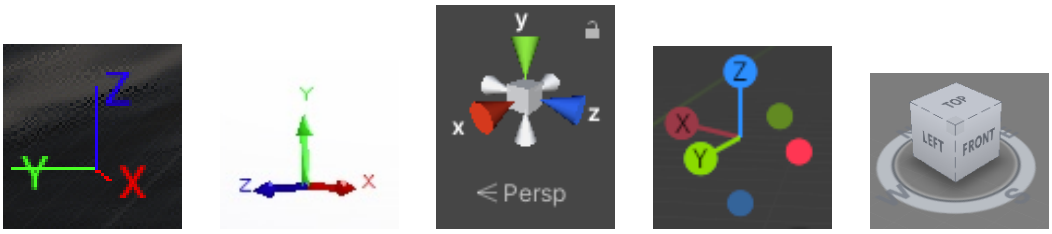
Obrázek 1.9 Nekonečné mřížky v programech Blender a Unity

***F18* - Indikátor světových os** Zkoumáním ostatních grafických programů obsahujících 3D viewporty si lze povšimnout, že mnoho z nich ve svém uživatelském rozhraní obsahuje v nějaké podobě indikátor světových os. Prvek, který různými způsoby ukazuje uživateli, jak je svět, na který se dívá, vůči němu orientován. Typicky je vyobrazen jako malý obrázek v některém z rohů 3D viewportu, ukazující v nějaké podobě hlavní osy světa orientované stejně jako aktivní kamera. Tím umožňuje uživateli ve všech případech pouhým pohlédnutím rychle zjistit, jaká je orientace kamery v případech, kdy to není ze samotné scény zcela zjevné. Indikátor často doplňují i textové popisky jednotlivých os.

Jak bylo zmíněno v požadavku ***F17***, v I3T se často pracuje s čísly, jako jsou souřadnice objektů. Tudíž je pro uživatele důležité vždy snadno poznat, jakým směrem jsou orientovány hlavní osy světa. Indikátor os nekonečnou mřížku z požadavku ***F17*** navíc dobře doplňuje, v případech kdy uživatel přímo nevidí hlavní osy na mřížce, když je od os vzdálen. Mřížka také neindikuje, kterým směrem je daná osa kladná, a kterým záporná.

Zatímco z indikátoru os je možné směr os lehce rozpoznat.

Na obrázcích 1.10 jsou vidět různé indikátory os z ostatních 3D programů. Jak je vidět může se jednat o jednoduché znázornění normalizovaných vektorů tří hlavních os s textovými popisky, jako v případě Unreal Enginu nebo CAD programu SolidWorks. Nebo komplexnější 3D reprezentace jako v programech Unity a Autodesk Maya, které používají své poměrně specifické designy. Tyto složitější indikátory také reagují na kliknutí a například nastaví pohled kamery tak, aby byl rovnoběžný s vybranou osou. V případě Autodesk Maya je tento koncept rozšířen na takzvanou *ViewCube*[7], kterou lze nastavit pohled kamery rovnoběžně nejen s hlavními osami, ale také například v 45° krocích od nich. Kostka indikátoru je označena nápisy udávajícími směr, ale neuvádí přímo popisky os a spíš je určena pro ovládání kamery, než indikaci orientace. Nejspíš kvůli tomu tento indikátor v programu Maya doplňuje druhý jednodušší indikátor os podobný těm z Unreal Engine a SolidWorks.



(a) Unreal Engine **(b)** SolidWorks **(c)** Unity **(d)** Blender **(e)** Maya

Obrázek 1.10 Příklady indikátorů os z různých 3D programů.

Zlatou střední cestu zdánlivě razí Blender, jehož indikátor umožňuje základní ovládání kamery, a přitom se stále velice podobá trojici základních os. Dobrou vlastností indikátorů z Unity a Blenderu je schopnost přehledně vyobrazovat i opačný směr dané osy. V případech ostatních indikátorů si uživatel opačný směr musí odvodit ze směrů kladných. Celkově je pro I3T vhodný indikátor podobný tomu z Blenderu, mimo jiné také díky jeho jednoduchosti, a měla by se v novém řešení prohlížeče scény objevit alespoň jeho neinteraktivní verze.

***N01* - Nízké hardwarové požadavky** Při vývoji je nutné brát v potaz dopady na výkon jednotlivých funkcí. Aplikace by měla plynule běžet na většině systémů (cíl 60+ fps). Optimalizace výkonu však není nejvyšší prioritou, hlavní prioritou je docílení potřebných funkcí. Použité techniky ale musí být v této fázi z hlediska výkonu realistické nebo s možností budoucí optimalizace.

***N02* - Vysoká kompatibilita** Aplikaci je nutné spustit i na zastaralejších systémech, které například nedisponují nejnovější verzí OpenGL. Bylo rozhodnuto, že aplikace bude využívat nejvýše specifikaci OpenGL 3.3. Nelze také používat OpenGL rozšíření podporované pouze jedním výrobcem. Aplikace bude spustitelná na operačních systémech Windows, Linux a MacOS.

[7] Kterou si nechali patentovat https://patents.google.com/patent/US7782319.

***N03* - Možnost budoucí serializace** Odděleně od tohoto projektu je vyvíjena funkce ukládání a načítání I3T scén. Řešení viewportu by mělo s tímto počítat a umožnit lehce měnit jeho stav.

***N04* - Jednoduché uživatelské rozhraní** Aplikace se celkově snaží docílit co nejjednoduššího uživatelského rozhraní, které by mělo být maximálně intuitivní a uživatele by nikdy nemělo zahltit příliš velkým množstvím informací.

***N05* - Kvalita obrazu** Pohled hlavního okna prohlížeče scény by měl být vždy vykreslován v rozlišení, ve kterém byl skutečně vykreslen. Nemělo by v důsledku změny velikosti okna docházet k rozmazání (jak tomu bylo v původní implementaci).

Měl by také podporovat nějakou formu antialiasingu, přičemž postačí zabudovaná implementace MSAA v OpenGL.

1.4 Rešerše metod

V této sekci je blíže prozkoumána teorie či způsoby implementace některých funkcí, které jsou třeba ke splnění funkčních požadavků popsaných v sekci minulé.

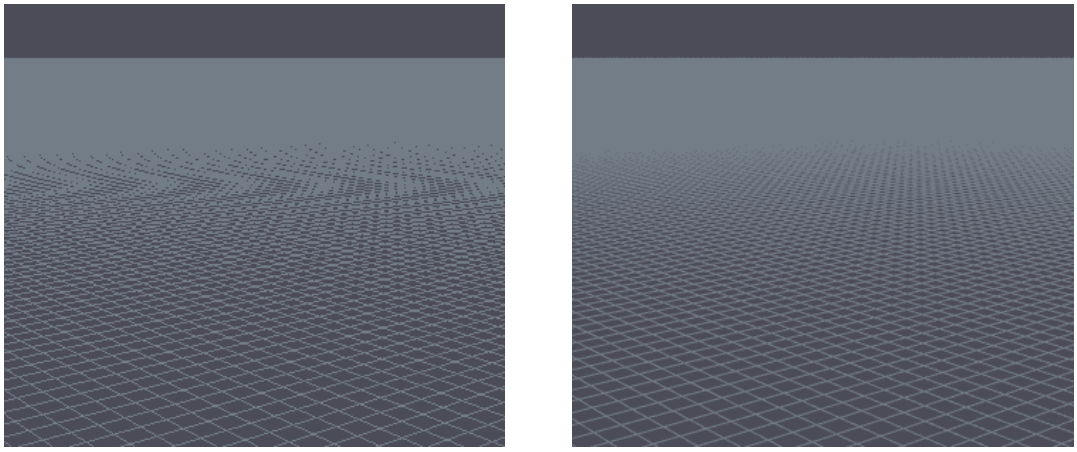
1.4.1 Nekonečná mřížka

Existuje mnoho způsobů jak vykreslovat mřížky. V případě, že je implementován systém načítání 3D modelů ze souborů, lze jakoukoli mřížku vymodelovat a ve scéně vykreslit stejně jako jakýkoli jiný 3D model. Trochu elegantnější řešení je vygenerovat geometrii mřížky procedurálně a poté vykreslovat stejně jako u 3D modelu trojúhelníky nebo ještě lépe vykreslovat pouze úsečky, jelikož mřížka nemusí mít nutně žádný objem. Pokud má ale mřížka pokračovat zdánlivě do nekonečna začnou se tyto přístupy setkávat s problémy. Velice velká mřížka vykreslená pomocí primitiv začne vyžadovat obrovské množství geometrie, která musí být přechovávána v paměti a eventuálně musí někde končit. K docílení opravdu zdánlivě nekonečné mřížky je potřeba ošetřit případy, kdy se kamera od geometrie mřížky značně vzdálí a mřížku je třeba posunout blíž k pozici kamery, aby nikdy nedosáhla jejího konce. Geometrie mřížky je také víceméně neměnná, jelikož je velice neefektivní jí často regenerovat. V případě vykreslování pomocí primitiv úseček (jako OpenGL *GL_LINES*) jsou možnosti jejich šířky závislé na implementaci a linky širší než 1 pixel jsou oficiálně *deprecated* [8, s. 343].

Další možností je k vykreslení mřížky využívat pouze otexturovanou plošku, a nahradit jednotlivá geometrická primitiva opakující se průhlednou texturou, která obsahuje rastrový obrázek malé části mřížky. Tato metoda může využít existující funkce OpenGL jako mipmapping a anizotropní filtrování ke snížení šumu či artefaktů, které se mohou na mřížce objevit v dálce u horizontu, kde jediný pixel pokrývá velkou část mřížky s mnoha linkami (a tedy velkou část textury). Při vykreslování jednotlivých linek mřížky pomocí primitiv vzniká podobný problém v případech, kdy jsou linky nedostatečně antialiasované a vzniká rušivý „Moaré“ efekt[8]. Tento efekt je vcelku všudypřítomný, ale antialiasing či zmíněné způsoby vzorkování textur ho dokáží značně potlačit. Tato metoda má podobné

[8] Rušivý optický efekt, který vzniká překrýváním pravidelných vzorů (https://cs.wikipedia.org/wiki/Moaré).

omezení jako předchozí metoda využívající skutečnou geometrii. Mřížku je stále nutné posouvat a úpravy jejího vzhledu vyžadují upravení její textury.



(a) Žádné AA **(b)** 4x MSAA

Obrázek 1.11 Příklad mřížky vykreslené pomocí primitiv GL_LINES s a bez antialiasingu

Mřížku je také možné vykreslit zcela odděleně od jakékoli geometrie ve světovém prostoru, a to čistě pomocí shaderu. Taková metoda je popsána na internetovém blogu *A Slice Of Rendering* [9]. Přes celou obrazovku lze vykreslit jedinou plošku speciálním shaderem, ve kterém je každému pixelu obrazovky určeno, zdali je součástí mřížky. V opačném případě se mu nastaví průhledná alfa hodnota. Jestli daný pixel je nebo není součástí mřížky, lze určit čistě matematickým výpočtem uvnitř shaderu. Z pohledové a projekční matice se vypočítá paprsek vycházející ze středu promítání do daného pixelu (respektive do polohy pixelu/fragmentu na přední rovině pohledového objemu). Tento paprsek dále protíná zadní rovinu pohledového objemu a uvnitř něho tvoří úsečku, která ve světových souřadnicích reprezentuje daný pixel. Pro libovolný pixel je tedy možné výpočtem průniku této úsečky a roviny mřížky získat světový bod na mřížce, který daný pixel reprezentuje. Dle souřadnic tohoto bodu se stačí rozhodnout, zdali leží či neleží na jedné z linek mřížky, a nastavit mu náležitou barvu a hloubku.

Jelikož je celá mřížka vykreslena v shaderu a plyne z matematické definice, je velice jednoduché ji upravovat. Také nikdy není třeba se zabývat její pozicí, protože vyhodnocuje každý pixel. Linky mřížky jsou v shaderu vykresleny lehce rozmazané, což zamezí aliasingu, a míra rozmazání je postupně zvyšována pomocí hodnot shaderových derivací světové pozice bodů mřížky, čímž se potlačují artefakty na horizontu. Tato metoda byla vybrána jako vhodná a je v další kapitole implementována.

1.4.2 Průhlednost

Poloprůhledné objekty není zcela jednoduché v OpenGL vykreslovat. Mají totiž vlastnost zobrazovat i to, co je za nimi. Při vykreslování pomocí hloubkového bufferu (*depth* bufferu) je tato jejich vlastnost poněkud nepříjemná, jelikož při vykreslování již nelze předpokládat, že fragment s nižší hloubkou bude plně zakrývat všechny ostatní fragmenty na stejné pozici s hloubkou vyšší. Ve výsledku tedy dle pořadí vykreslování plošek může dojít k tomu, že se některé plošky, které by měly být skrz jinou poloprůhlednou plošku vidět, nezobrazí,

protože byly vykresleny až po zmíněné poloprůhledné plošce a tedy neprošly hloubkovým testem.

I kdyby byl tento problém s hloubkovým bufferem nějakým způsobem vyřešen, stále je nutné dodržet striktní pořadí vykreslování poloprůhledných plošek, aby bylo dosaženo správného alfa míchání. Jak je uvedeno v knize *Moderní počítačová grafika*: „Při zpracování poloprůhledných ploch je třeba dodržet zásadu jejich vykreslování (a s ním spojeného míchání barev) odzadu dopředu. Jiný způsob není možný, protože alfa míchání není komutativní operací“[10, s. 363]. Ve stejném zdroji je posléze popsán postup vykreslování s hloubkovým bufferem, který vykreslování poloprůhledných plošek nejprve odloží, a poté všechny poloprůhledné plošky vykreslí v pořadí od nejvzdálenějších po nejbližší. Tento postup však vyžaduje každý snímek seřazení individuálních plošek všech objektů ve scéně a pro složitější modely je velice neefektivní [10, s. 363].

Vhodným řešením je použití jedné z technik ze skupiny technik nazývané *Order-independent transparency* (OIT), které nevyžadují seřazení primitivů geometrie a vyhodnocují průhlednost zvlášť pro každý fragment. Tyto techniky se typicky dělí na dvě skupiny: „exaktní“ a „aproximační“ [11]. Pro účely I3T byla vybrána aproximační technika jménem *Weighted blended order-independent transparency* (WBOIT) [12], která je jednoduchá na implementaci a dosahuje vysokého výkonu.

Kapitola 2
Praktická část

2.1 Technologie

Klíčové technologie, které I3T používá jsou C++ a grafické API OpenGL. Pro správu nativních oken a uživatelského vstupu využívá knihovnu GLFW[1]. K realizaci uživatelského rozhraní je použita knihovna Dear ImGui [13].

K verzování je použit systém *git* a repozitář projektu je hostován na neveřejném školním serveru GitLab. K sestavování projektu je používán nástroj CMake. Vývoj v rámci této práce proběhl v IDE CLion s toolchainem Visual Studia 2022 Community Edition.

Po každém nahraní změn na server GitLab je aplikace sestavena službou GitLab CI v prostředí Linuxu. Po sestavení jsou spuštěny existující unit testy a je zkontrolován formát kódu nástrojem *clang-format*.

Příležitostně byl k ladění grafické stránky programu využit program NVIDIA Nsight Graphics[2].

Ještě je vhodné zmínit, že se v kódu pro názvy členských proměnných používá varianta maďarské notace[3], ve které mají tyto proměnné prefix `m_`.

2.2 Návrh nového řešení

2.2.1 Princip Dear ImGui

Uživatelské rozhraní I3T je realizováno knihovnou *Dear ImGui*, z čehož plyne využití jejího *immediate mode* paradigmatu[4] i v návrhu viewportu. Je dobré principu Dear ImGui nejdříve porozumět, a proto je v následujících odstavcích krátce popsán.

Uživatelské rozhraní je každý snímek zcela definované posloupností příkazů, které celé UI vždy budují znovu od základu. V jednom snímku může být zobrazeno rozhraní zcela odlišené od snímku předchozího. K samotnému vykreslení snímku dojde jednou na jeho

[1] GraphicsLibraryFrameworkhttps://www.glfw.org/
[2] https://developer.nvidia.com/nsight-graphics
[3] https://en.wikipedia.org/wiki/Hungarian_notation
[4] Detailní popis paradigmatu Dear ImGui lze nalézt zde: https://github.com/ocornut/imgui/wiki/About-the-IMGUI-paradigm

konci, vykreslením všech takzvaných `DrawList`ů, obsahující seznamy vykreslovacích příkazů, které byly jednotlivými voláními funkcí Dear ImGui v daném snímku postupně vytvořeny.

Na akce uživatele se reaguje dynamicky každý snímek přímo při budování UI. Na rozdíl od „konvenčních“ *retained mode* GUI knihoven se k zpracování uživatelského vstupu nepoužívají události a callbacky, nýbrž se kód dotazuje na stav uživatelského vstupu každý snímek přímo při sestavení prvků UI.

Výpis kódu 2.1 vystihuje princip Dear ImGui příkladem s tlačítkem. Voláním `ImGui::Button()` se na aktuální pozici v okně objeví tlačítko. Uvnitř této metody je přímo vyhodnocen vstup uživatele a je z ní navrácena logická hodnota `bool`, která udává, zdali uživatel na tlačítko právě klikl (respektive zdali bylo na oblast tlačítka právě kliknuto, tlačítko nemuselo být minulý snímek vykresleno). Na stejném místě v kódu, kde bylo tlačítko vytvořeno, lze tedy rovnou reagovat na jeho stisknutí. Jelikož je tato reakce přímo v kódu budujícím UI prvky, může být reakcí například vytvoření dalších UI prvků, a případně další reakce na ně.

```
1 if (ImGui::Button("Click me"))
2 {
3     std::cout << "Button was clicked" << std::endl;
4 }
```

Kód 2.1 Příklad ImGui tlačítka

2.2.2 Architektura nového prohlížeče scény

Celou tuto sekci doplňuje návrhový model tříd v diagramu na obrázku 2.1, který znázorňuje nejdůležitější třídy pro veřejné rozhraní prohlížeče scény. Ostatní neméně důležité třídy zajišťující samotnou implementaci jsou popsány později ve svých oddělených sekcích.

Nový prohlížeč scény zcela nahrazuje původní složku (balíček) *World* a poskytuje jednoduché a jednotné rozhraní, které existující části kódu uživatelského rozhraní mohou volat. Třídy z balíčku *GUI* realizující uživatelské rozhraní I3T vykreslují vlastní OpenGL obsah předáním textury knihovně Dear ImGui. Specificky zařadí vykreslovací příkaz aktuálnímu ImGui `DrawList`u pomocí metody `DrawList::AddImage()`[5], které se předá identifikátor předpřipravené OpenGL textury. Nový viewport je navržen tak, aby tyto textury na požádání pro daný snímek poskytoval (dle požadavku ***F11***).

Veřejné rozhraní Viewport

Hlavní třídou nového prohlížeče scény je třída `Viewport` definovaná v souboru `Viewport.h`. Instance této třídy poskytuje hlavní rozhraní[6] viewportu, které může ostatní kód aplikace využívat. Nejprve je nutné viewport inicializovat metodou `init()`, které lze volitelně předat instancí třídy `ViewportSettings`, což je struktura dat upřesňující obecná nastavení viewportu (dle požadavku ***N03***). Při inicializaci jsou načteny základní potřebné zdroje, což může chvíli trvat, proto je inicializace metodou `init()` vyčleněna z konstruktoru objektu, aby mohla být zavolána odděleně od konstrukce objektu.

[5] Případně metodou `ImGui::Image()`, která interně volá `DrawList::AddImage()`.

[6] Rozhraní v širším smyslu. Nejedná se o C++ rozhraní (*interface*), nýbrž o klasickou (neabstraktní) třídu.

K objektu viewportu mají přístup třídy uživatelského rozhraní a po inicializaci mohou volat jeho tři hlavní vykreslovací funkce, které jsou specificky navrženy pro každý ze tří požadovaných pohledů:

- drawViewport(renderTarget, width, height, options)
  Vykreslení hlavního pohledu viewportu nezávislou kamerou (***F10***). Použité v *GUI* třídě `ViewportWindow`.

- drawScreen(renderTarget, width, height, view, projection, options)
  Vykreslení pohledu ze specifické kamery pro krabičku „Screen“ (***F03***). K použití ve třídě krabičky `WorkspaceScreen`. Této metodě se předává pohledová (*view*) a projekční (*projection*) matice dané kamery.

- drawPreview(renderTarget, width, height, object, options)
  Vykreslení náhledu objektu pro krabičku „Model“ (***F01***). K použití ve třídě krabičky `WorkspaceModel`. Parametr *object* je ukazatel na objekt reprezentující krabičku „Model“ ve 3D scéně a bude popsán později.

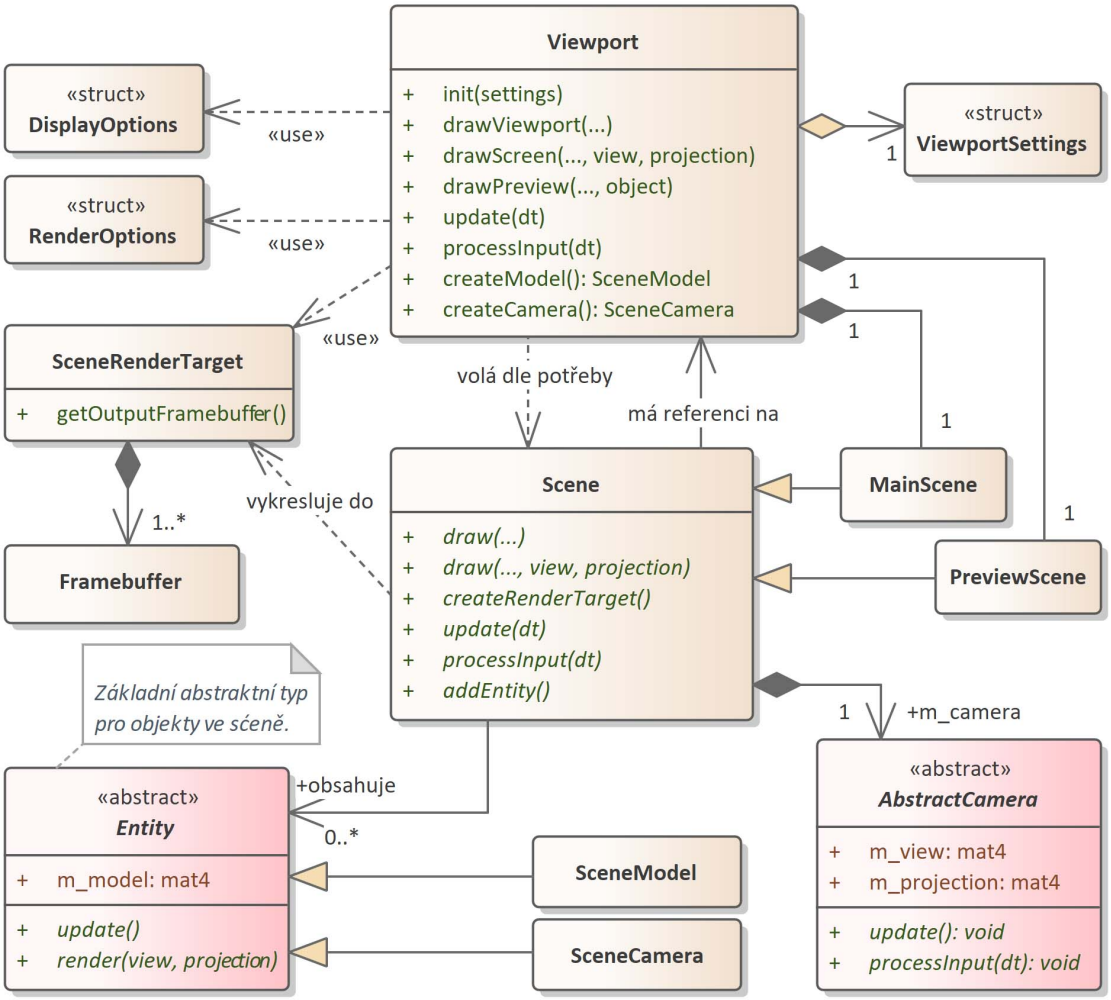
K vykreslení viewportu v UI je třeba navrátit z vykreslovacích metod texturu. To zajišťuje třída `SceneRenderTarget`, která se rozhraní viewportu předá jako parametr *renderTarget*. Tento parametr je předáván pomocí ukazatele, který může být při prvním volání prázdný. V tom případě je před vykreslením viewportem inicializován dle ostatních předaných parametrů. Třídy volající tyto metody tedy nemusí nijak jeho inicializaci řešit a pouze musí někde tento ukazatel mezi snímky přechovávat. Po vykreslení z ukazatele *renderTargetu* stačí získat identifikátor výsledné OpenGL textury.

Při vykreslování scény viewportu typicky nestačí pouze jediná textura. Někdy je potřeba mnoho datových úložišť pro dočasné uchování dat nebo pro jiné pomocné účely. Třída `SceneRenderTarget` samotná žádná obrazová data nedrží, ale udržuje seznam jedné či více instancí třídy `Framebuffer`. Tyto instance jsou wrappery OpenGL *Framebuffer* objektů a budou podrobněji popsány v sekci 2.4.4. Výsledný identifikátor OpenGL textury lze po vykreslení získat z „výstupního“ `Framebuffer` objektu, který je obdržen z objektu `SceneRenderTarget` zavoláním metody `getOutputFramebuffer()`.

Objekt `SceneRenderTarget` je viewportem inicializován parametry *width* a *height*, které specifikují dimenze výsledné textury v pixelech. Na jeho inicializaci se také podílí parametry objektů `RenderOptions` a `DisplayOptions` (výše znázorněny jen jako parametr *options*), což jsou datové struktury, které blíže specifikují, jakým způsobem se má textura vykreslit. Mezi možnosti `RenderOptions` může například patřit nastavení antialiasingu nebo zda textura obsahuje alfa kanál. Kupříkladu hlavní viewport by měl mít co nejkvalitnější obraz (vysoký antialiasing) a nemělo by za ním být vidět pozadí okna (nemusí mít alfa kanál). Naopak náhled modelu antialiasing mít nemusí a je žádoucí ho vykreslit s alfa kanálem, aby zbytečně nepřekrýval „tělo“ (pozadí) krabičky. `DisplayOptions` udávají, které typy objektů se mají ve scéně vykreslit. V případě pohledu pro krabičku „Kamera“ by se ve scéně měly objevovat jen modely krabiček „Model“ a nikoli modely ostatních kamer či pomocné objekty jako bazické vektory modelů či pohledové objemy kamer.

Scéna a její entity

Třída `Viewport` sama o sobě je určena pouze pro účely vytvoření jednoduchého rozhraní pro třídy ostatní. Lze jí chápat jako fasádu pro složitější rozhraní ostatních tříd, ze kterých



Obrázek 2.1 Návrhový model tříd nového řešení prohlížeče scény. V diagramu jsou zahrnuty pouze některé důležité třídy.

se nové řešení viewportu skládá (takzvaný *facade pattern* [6, s. 104]). Její implementace spravuje instance třídy `Scene` reprezentující oddělené 3D scény. Třída `Scene` má již obecné rozhraní, které není nijak zásadně specifické I3T a zajišťuje většinu důležitých funkcí viewportu. Viewport obsahuje dvě scény: scénu hlavní a scénu pro náhledy. Obě tyto scény mají svoji instanci třídy `Scene` přechovávanou jako proměnnou v objektu `Viewport`. Také obě vyžadují trochu odlišené chování, a proto jsou jejich instance typu podtříd `MainScene` a `PreviewScene`, které ze třídy `Scene` dědí. Metody viewportu `drawViewport()` a `drawScreen()` vykreslují hlavní scénu, metoda `drawPreview()` scénu pro náhledy.

Scéna představuje samostatný 3D svět a udržuje seznam instancí třídy `Entity`, které reprezentují objekty ve světě. Je zodpovědná za samotné vykreslování 3D světa. To probíhá v její metodě `Scene::draw()`, která má dvě přetížené verze:

1. draw(renderTarget, width, height, displayOptions)

2. draw(renderTarget, width, height, view, projection, displayOptions)

Parametry pro tyto přetížené metody jsou sesbírány `draw` metodami `Viewport`u a jejich význam byl již výše popsán.

Metody se liší pouze parametry pohledové (*view*) a projekční (*projection*) matice, které definují kameru, ze které je scéna vykreslena. První verze metody volá verzi druhou a tyto parametry doplňuje z členské proměnné scény typu `AbstractCamera`. Každá scéna má právě jednu vlastní kameru, kterou lze použít pro výchozí pohled na scénu, a případně jí může uživatel i ovládat. Tato kamera je blíže popsána v požadavku ***F10*** a sekci 2.4.3.

Druhá verze metody *draw* dle zadaných parametrů vykresluje scénu z pohledu kamery definované pouze parametry *view* a *projection*. Krabička „Screen“ skrz volání `Viewport::drawScreen()` této metodě přímo předává své parametry kamery k vykreslení (***F03***).

Scéna je vykreslena do předaného objektu `SceneRenderTarget`, který musí být vždy složen ze správných framebufferů ve správném pořadí, které implementace scény očekává. To je zajištěno tím, že právě ve třídě scény je tento objekt vytvořen metodou `Scene::createRenderTarget()`, při prvním volání vykreslovacích metod rozhraní `Viewport`, kdy jim je předán prázdný ukazatel *renderTarget* k inicializaci. Vykreslovacím metodám třídy `Scene` by tedy měly vykreslovací metody rozhraní `Viewport` vždy předat již inicializovaný objekt `SceneRenderTarget`. Po vykreslení je tento objekt předán vnějšímu kódu (třídám v *GUI*), aby mohl být následující snímek opět předán k vykreslování.

Podtřídy scény mohou `draw` metody ve své implementaci přepsat společně s metodou `createRenderTarget()`, čímž si mohou dle potřeby pozměnit, jaké framebuffery budou k vykreslovaní použity. Scéna vždy před vykreslením připraví framebuffery a následně prochází svůj seznam entit a volá jejich vykreslovací metody. Bližší popis implementace scény je v sekci 2.4.

Entita je abstraktní třída, která reprezentuje jakýkoli objekt nacházející se ve scéně, a drží vlastnosti, které jsou všem objektům společné. Předchozí implementace prohlížeče scény Daniela Gruncla [3] pro entity používala takzvaný *entity component system*, který ctí princip *composition over inheritance* a který je hojně využíván při vývoji her a herních enginů [14]. V tomto systému jsou vlastnosti entity vždy definovány jí přiřazenými komponentami, čímž lze vlastnosti entit lehce měnit a libovolně skládat bez jakékoli duplikace kódu. Pro složité hry a herní enginy je tento systém ideální díky jeho vysoké flexibilitě a modulárnosti.

Minulá implementace tohoto systému však trpěla přílišnou komplexitou a nepřehledností. V novém řešení prohlížeče scény bylo rozhodnuto tento systém ve jménu jednoduchosti opustit. Požadavky I3T jsou momentálně velice specificky určeny a neplánuje se žádné zásadní rozšíření. Pro entity je použita standardní dědičná hierarchie a většina potřebných vlastností je přímo definována v základní abstraktní třídě `Entity`. Tento přístup flexibilní není, ale zato je velice přímočarý a pro potřeby I3T dostačující.

Dalším zjednodušením oproti minulé implementaci je absence grafu scény specifického pro viewport. Dříve bylo možné entitě přiřadit ostatní entity jako potomky, které následně sdílely s rodičem jejich grafickou transformaci a tedy pracovaly s rodičovskou vztažnou soustavou. V novém řešení má každá entita jednu jedinou transformaci, pomocí které se vykresluje a která vždy pracuje ve světovém prostoru. Specifické případy, kdy jedna entita potřebuje pracovat se vztažnou soustavou jiné, lze ošetřit přímo v implementaci dané podtřídy. Každé entitě lze tedy zadat transformaci dle stavu jádra, která bude použita k vykreslovaní, a není třeba jakkoli synchronizovat vedlejší graf scény. Hierarchie entit je implementována v sekci 2.4.2.

Mezi podtřídy `Entity` patří třídy `SceneModel` a `SceneCamera`, jejichž instance vytvoří a přidá do hlavní scény rozhraní `Viewport` při volání metod `Viewport::createModel()` a `Viewport::createCamera()`. Ty volají krabičky „Model“ a „Kamera“, aby získaly referenci na entitu, která je reprezentuje ve 3D scéně. Tuto referenci v sobě uchovají a mohou jí později využít k úpravě všech relevantních vlastností těchto entit, jak je popsáno v požadavcích ***F01*** a ***F02***, zejména mohou svým entitám aktualizovat data transformací, což je blíže popsáno v sekci 2.7.

Krabička „Model“ může svou instanci třídy `SceneModel` předat metodě `Viewport::drawPreview()` jako parametr *object*, která vykreslí jeho náhled (součást požadavku ***F01***). K tomuto účelu je právě určena scéna pro náhledy `PreviewScene`, do které lze jakýkoli objekt dočasně vložit a vykreslit její kamerou.

Uživatelský vstup viewport vyhodnocuje v metodě `Viewport::processInput()`, která by měla být volána jednou před vykreslením hlavního viewportu (pomocí `Viewport::drawViewport()`). Ten jediný momentálně vyžaduje uživatelský vstup. Volání metody `processInput` je přímo delegováno instanci hlavní scény `MainScene`, která zavolá metody vyhodnocující vstup všech objektů, které o něj stojí. Zejména tedy své kamery, která poté reaguje na pohyby myši uživatele. Obecně se ve třídách viewportu využívá k přístupu k uživatelskému vstupu již existující globální statická třída `InputManager` definovaná v jádře v balíčku *Core/Input*, která byla navržena v jedné z předchozích prací [2, s. 26]. Tento přístup má nevýhodu, že z rozhraní tříd není zjevné, odkuď se uživatelský vstup vlastně bere. Nicméně je to přístup využívaný v celé aplikaci a v principu se hodně podobá přístupu Dear ImGui. `InputManager` má za cíl abstrahovat uživatelský vstup od specifické GUI knihovny a je v kódu viewportu lepší volat jej než přímo metody Dear ImGui, protože takto má aplikace nad formátem vstupů kontrolu.

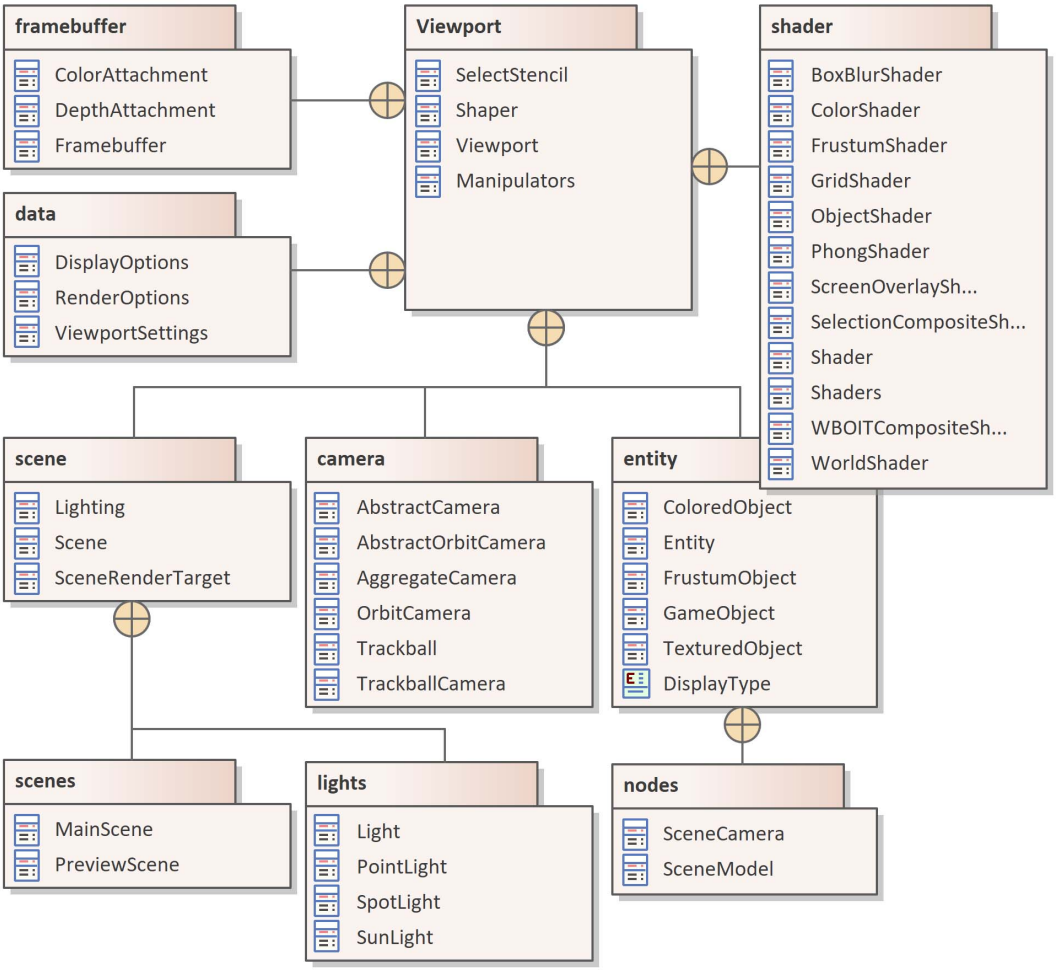
Poslední důležitou metodou rozhraní `Viewport` je metoda `update()`. Ta umožňuje scénám a jejich entitám provádět jakékoli logické operace, které nejsou vázány přímo na vykreslování. Je po vykreslování volána hlavní smyčkou aplikace a stejně jako metoda `processInput()` očekává parametr *dt* (*delta time*), který specifikuje dobu od posledního volání. Ta je využita k odstranění závislosti některých operací na frekvenci volání těchto metod (a tudíž docílení nezávislosti od snímkové frekvence aplikace).

2.3 Struktura modulu Viewport

Na diagramu 2.2 je vidět struktura nového balíčku *Viewport*, který nahrazuje původní balíček *World*, který je znázorněn na diagramu 1.4 v sekci analýzy původní implementace 1.2.2. Tento balíček je hlavní část nové implementace. Některé z jeho tříd již byly nastíněny v minulé sekci 2.2.2 a o většině z jeho tříd pojednává následující sekce 2.4.

Mimo balíček *Viewport* byly provedeny větší či menší změny v balíčcích jádra a uživatelského rozhraní. V jádře byl zcela přepsán balíček pro správu zdrojů *Core/Resources*, který popisuje sekce 2.5. V balíčku *Core/Input* došlo ke změnám ve zpracování uživatelského vstupu v třídě `InputManager`, ze které byla také vyčleněna třída `WindowManager` pro správu dokovatelných oken. K připojení nového prohlížeče scény s grafem scény I3T byly také provedeny menší úpravy v balíčku *Core/Nodes*. Ty jsou popsány v sekci 2.7.

V neposlední řadě došlo k napojení viewportu na uživatelské rozhraní krabiček a oken v balíčku *GUI*. Tomu se věnuje sekce 2.6.



Obrázek 2.2 Diagram balíčku *Viewport*, který nahrazuje původní balíček *World*. V balíčcích je znázorněna i většina tříd.

2.4 Implementace scény

V této sekci jsou popsány třídy, které třída `Scene` používá při vykreslování 3D scény. Nejdřív jsou popsány shadery využívané k vykreslování entit, jejichž dědičná hierarchie je popsána jako další. Poté je vysvětlena třída `Framebuffer`, která spravuje OpenGL framebuffer objekty (FBO), do kterých se výsledky vykreslování ukládají. Následovně jsou popsány kamery, které definují a ovládají pohled na scénu a na konci sekce je obecně popsáno, jakým způsobem vykreslování pomocí těchto tříd v metodě `Scene::draw` probíhá.

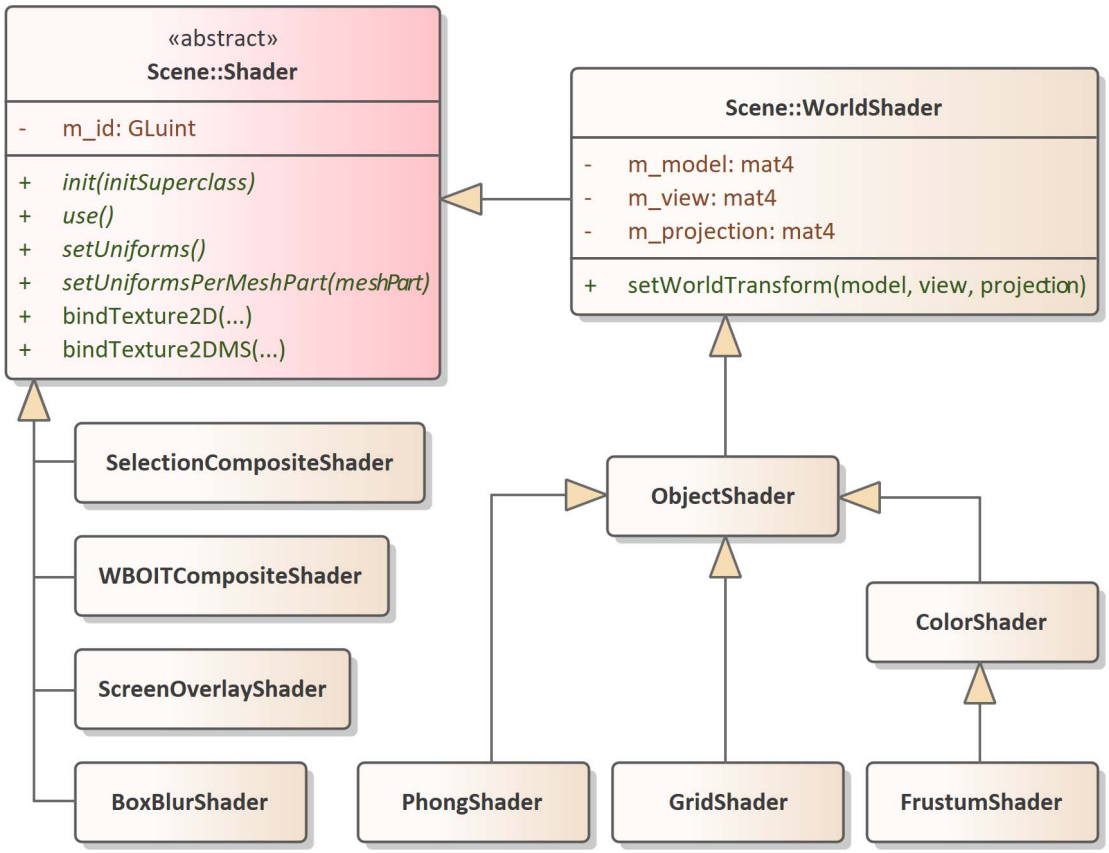
2.4.1 Shadery

Zdrojové soubory GLSL shaderů se nachází ve složce pro data *Data/Shaders* oddělené od zdrojových souborů C++ kódu I3T. Ze souborů jsou načítány novým správcem zdrojů (viz 2.5), který vrací identifikátor vytvořeného OpenGL shader objektu. Každý shader v kódu reprezentuje jedna podtřída abstraktní třídy `Shader`, jejíž instance jsou vytvářeny

a udržovány *singleton* třídou `Shaders`, která na každý shader poskytuje veřejnou referenci a je zodpovědná za jejich načítání i případné znovu načtení při běhu aplikace.

Jednotlivé třídy shaderů jsou vždy specifické danému GLSL shaderu a obsahují členské proměnné reprezentující jeho *uniformy*. Tyto členské proměnné je možné z kódu aplikace nastavovat a poté jsou jejich hodnoty předány GLSL shader objektu metodou `setUniforms`, kterou podtřídy implementují. Při vykreslování entity ve scéně instanci daného typu shaderu sdílí a vždy před vykreslením shaderu nastaví jeho proměnné dle svých vlastních vlastnosti a zavolají metodu `setUniforms`. Instance třídy `Shader` tedy nejsou obecně určeny pro udržování specifických dat nýbrž pouze pro použití jako rozhraní s GLSL shaderem. Některé proměnné shaderu však stačí nastavit jen jednou před vykreslováním, pokud je nic nepřepisuje jako například nastavení osvětlení.

Některé GLSL shadery obsahují uniformy, které se objevují i v shaderech ostatních. Podtřídy `Shader` proto tvoří dědičnou hierarchii, která je zachycena v diagramu na obrázku 2.3. Většina shaderů pro vykreslování objektů ve scéně například dědí z třídy `WorldShader`, která nastavuje uniformy modelové, pohledové a projekční matice, používané ve *vertex* shaderech pro transformaci vrcholů geometrie objektů ve scéně. Následuje krátký popis všech shaderů:



Obrázek 2.3 Diagram tříd balíčku *Viewport/shader*.

`PhongShader`

Používá se pro vykreslování osvětlených a otexturovaných 3D modelů. K osvětlení je

použit osvětlovací model Phong-Blinn [15]. V shaderu je možné kombinovat více texturovacích technik najednou (*multitexturing*). Specificky podporuje barevnou, spekulární, normálovou, AO a emisivní texturu. Normálové mapování používá implementaci popsanou na stránkách *LearnOpenGL* [16].

Osvětlení je spravováno třídou `Lighting` z balíčku *Viewport/scene*, jejíž instanci obsahuje každá scéna a následujícím útržkem kódu `PhongShader`u nastavuje pole uniformů specifikující pozice a parametry světel ve scéně:

```
1     Shaders::instance().m_phongShader->use();
2     m_lighting->setUniforms(*Shaders::instance().m_phongShader);
```

V kódu 2.2 je příklad přidání slunce do hlavní scény.

```
1 SunLight* sun = new SunLight();
2 sun->intensity = 0.8f;
3 sun->color = glm::vec3(0.93, 0.98, 1.0);
4 sun->direction = glm::vec3(-0.73, -0.64, -0.21);
5 sun->pos = glm::vec3(0, 4, 0);
6 m_lighting->addLight(sun);
7
```

Kód 2.2 Příklad přidání slunce do hlavní scény.

`ColorShader`
Jednoduchý shader pro vykreslování objektů jednou specifickou barvou a nebo dle barev uložených v jejich vrcholech.

`FrustumShader`
Podtřída `ColorShader`u využívaná k vykreslování pohledového objemu (frusta) dané kamery. Pomocí inverze násobku projekční a pohledové matice uvnitř vertex shaderu transformuje body NDC kostky[7], aby byla vytvořena reprezentace pohledového objemu ve světovém prostoru.

`GridShader`
Používá se k vykreslení nekonečné mřížky.

`ObjectShader`
Shader pro společné uniformy „objektů“ ve scéně, v tuto chvíli drží pouze uniform nastavující míru průhlednosti.

`WorldShader`
Shader pro všechny objekty, které pracují ve světovém prostoru. Nastavuje projekční, pohledovou a modelovou matici. Shaderům nastavuje mimo individuální matice také PVM[8] matici, která vzniká vynásobením těchto matic dohromady a „normálovou“ matici, pro transformaci normál bez deformace způsobené neuniformním škálováním.

[7] Kostka s rohy v bodech $(-1,-1,-1)$ a $(+1,+1,+1)$.
[8] $Projection * View * Model$

`WBOITCompositeShader`
Tento shader je využíván v implementaci WBOIT pro vytvoření finálního obrazu vyhodnocením bufferu odkrytí a akumulace.

`SelectionCompositeShader`
Využívaný pro skládání framebufferů při vykreslování zvýraznění obrysů objektů.

`ScreenOverlayShader`
Opět využívaný pro skládání obrazů při vykreslování zvýraznění obrysů objektů.

`BoxBlurShader`
Shader, který je používán k rychlému rozmazání obsahu framebufferu. Rozmazání provádí v lineárním čase relativně k velikosti kernelu rozmazání a vyžaduje dva průchody (vertikální a horizontální).

2.4.2 Hierarchie entit

Základní třídou hierarchie entit je abstraktní třída `Entity`, která reprezentuje objekt ve scéně. Všechny entity sdílí některé základní proměnné. Nejdůležitější proměnnou je `m_modelMatrix`, který entitám určuje jejich modelovou transformaci.

Entity si také drží instanci shaderu, který ve výchozím stavu používají k vykreslování, které probíhá v metodě `render(shader, view, projection)`. Entita má přetížené verze této metody, které nevyžadují předání shaderu. V tom případě entita používá shader v její členské proměnné `m_shader`. Co ale vyžaduje vždy je pohledová a projekční matice, která je posléze společně s maticí `m_modelMatrix` použita k vykreslení entity.

Další důležitou proměnnou entit je `m_visible`, která určuje zdali má být entita ve scéně vykreslena. S touto proměnnou souvisí enum `m_displayType`, kterým lze entitu zařadit do jedné ze skupin viditelnosti. Toto je ve viewportu využíváno pro skrytí specifických typů entit.

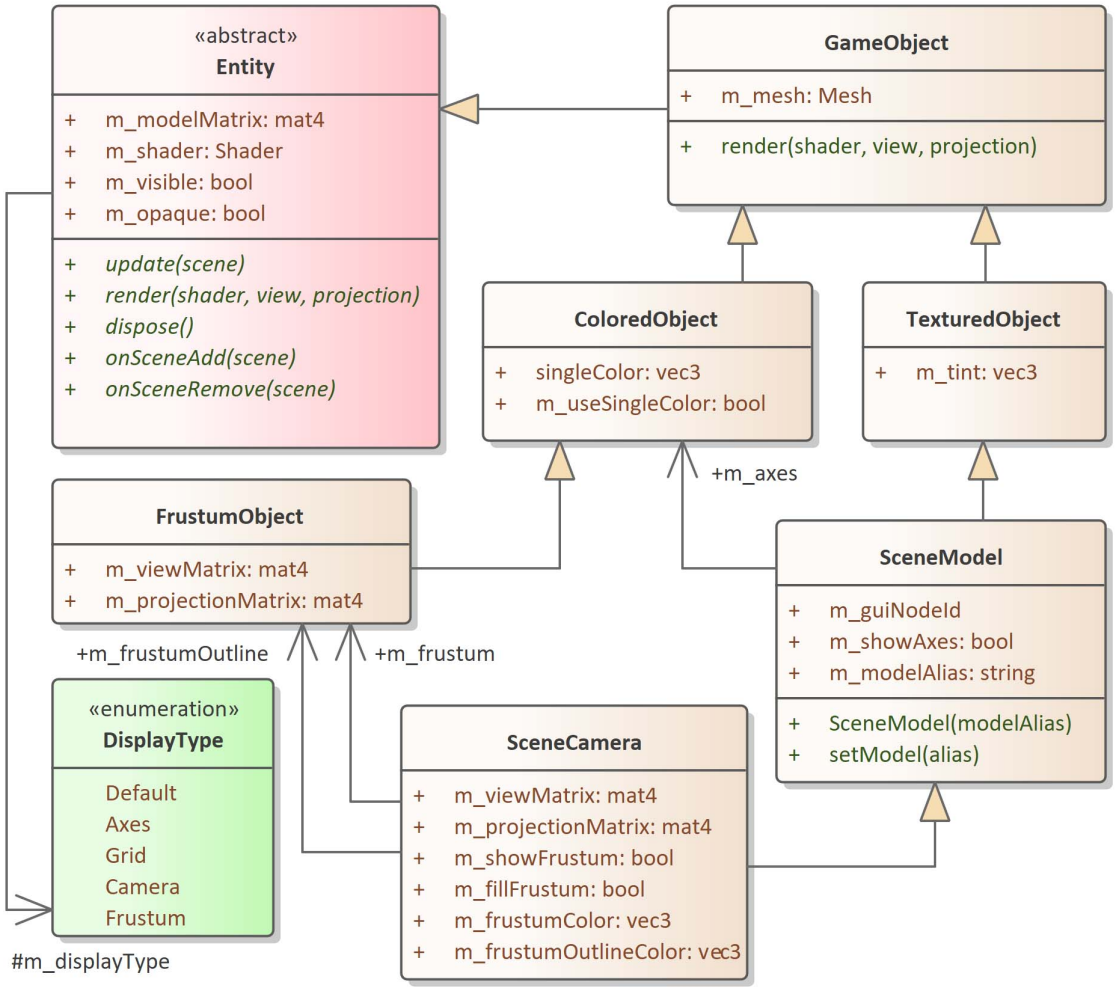
Z abstraktní třídy `Entity` dědí třída `GameObject`, která reprezentuje objekt ve scéně tvořený trojúhelníky nebo primitivy čar, a poskytuje základní implementaci metody vykreslování `GameObject::render()`. Uchování geometrie a její vykreslování zajišťuje třída `Mesh`, jejíž instanci lze vytvořit z geometrie definované v kódu nebo načtením plnohodnotného 3D modelu ze souboru.

Ze třídy `GameObject` dědí třídy `ColoredObject` a `TexturedObject`. Ty se liší v tom, které shadery používají k vykreslování. `ColoredObject` využívá `ColorShader` a `TexturedObject` vykresluje otexturované 3D modely pomocí `PhongShaderu`.

Pro krabičku „Model“ a „Kamera“ jsou speciálně určené entity `SceneModel` a `SceneCamera`, které jsou detailně popsány v sekci 2.6. Pro tyto entity jsou důležité metody `Entity::onSceneAdd()` a `Entity::onSceneRemove()`, které jim umožňují v momentě přidání do scény vytvořit další vlastní entity, které spravují. Tímto se entity vyhýbají potřebě mít systém potomků, který by zkomplikoval nakládání s modelovými transformacemi.

Když je entita `SceneModel` přidána do scény, vytvoří další entitu pro reprezentaci bazických vektorů transformace (***F04***), kterou zároveň přidá do scény a drží si k ní referenci jako členskou proměnnou. V metodě `Entity::update()` pak této vnitřní entitě vždy aktualizuje modelovou transformaci, aby odpovídala té její. Pokud je entita modelu odebrána ze scény, tak v metodě `onSceneRemove()` také ze scény odebere entity, které spravovala.

Podobně nakládá entita `SceneCamera` s entitou `FrustumObject` vykreslující její pohledový objem.



Obrázek 2.4 Diagram tříd balíčku *Viewport/entity*.

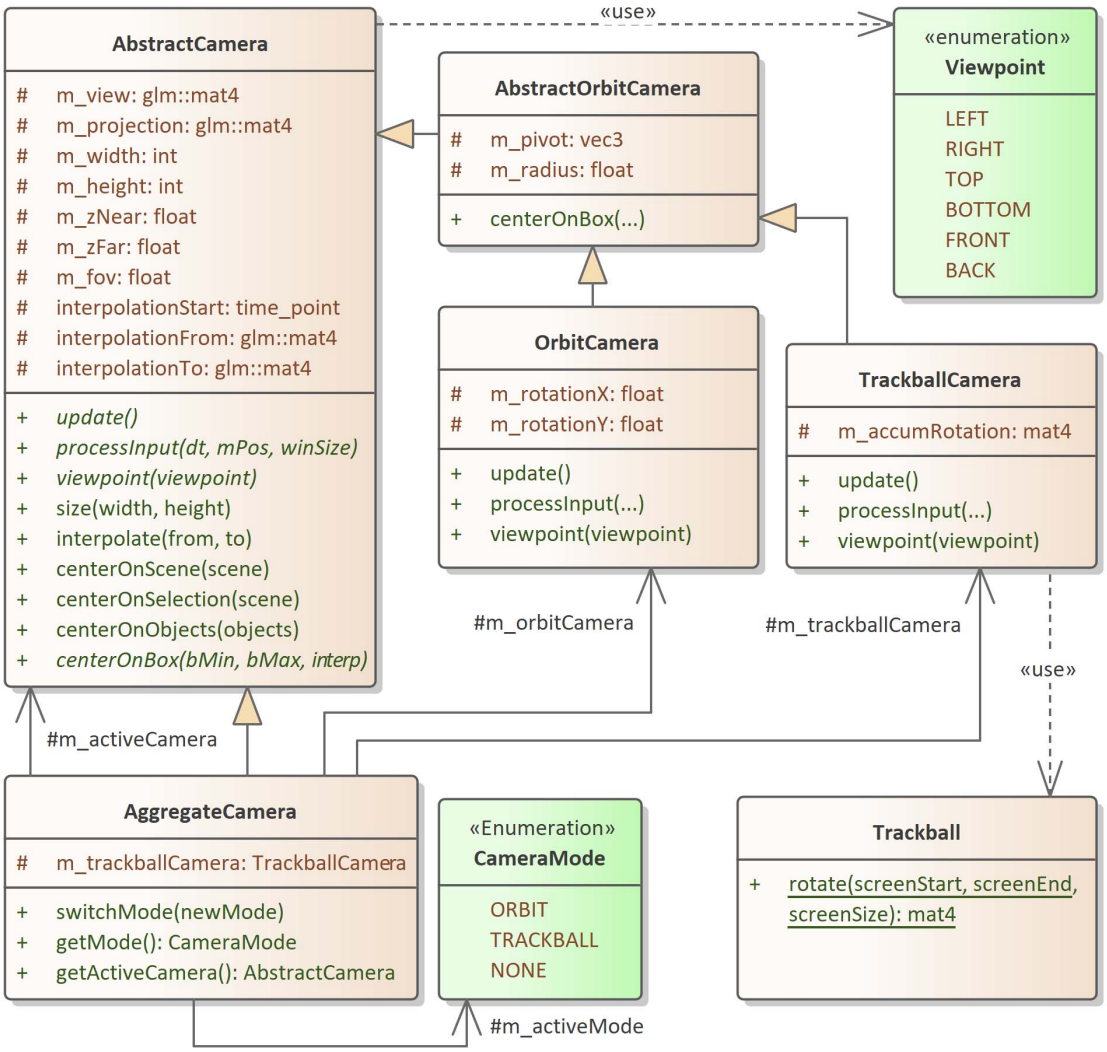
2.4.3 Kamery

Pro zobrazování scény z nezávislého pohledu se používá třída `AbstractCamera`. To je abstraktní třída, ze které primárně dědí třídy `OrbitCamera` a `TrackballCamera`, které zajišťují kamerové módy *orbit* a *trackball*, popsané v požadavku ***F10***. Diagram tříd balíčku *Viewport/camera* je vidět na obrázku 2.5.

Třída `AbstractCamera` definuje rozhraní kamer a drží základní proměnné společné pro všechny kamery, zejména projekční a pohledovou matici. Dle svých parametrů tyto matice generuje v metodě `update()`. Mezi společné parametry patří proměnné jako: šířka a výška obrazu (v pixelech), vzdálenost přední a zadní roviny pohledového objemu (*zNear* a *zFar*), vertikální úhel zorného pole a parametry interpolace.

Při volání metody `update()` uloží kamera do svých členských proměnných novou projekční a pohledovou matici, které lze následně získat voláním getterů `getView()` a `getProjection()`. Před touto operací by měly být kameře nastaveny požadované rozměry k výpočtu poměru stran pomocí metody `size(width, height)`.

Uživatelský vstup kamera zpracovává v metodě `processInput()`, která přijímá mimo



Obrázek 2.5 Diagram tříd balíčku *Viewport/camera*.

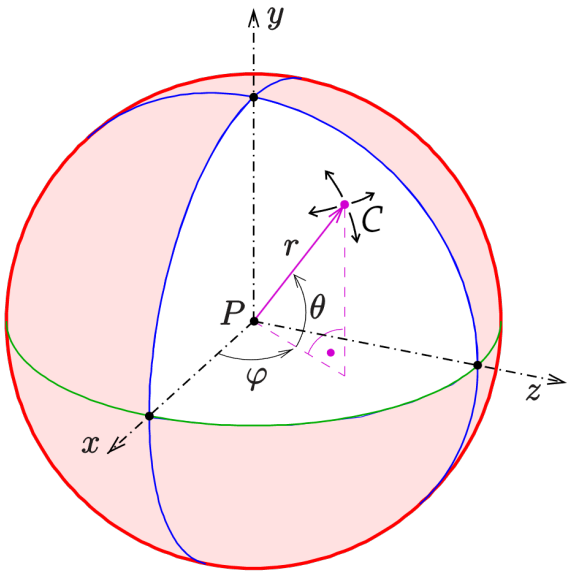
deltaTime také parametry aktuální pozice myši relativní k aktivnímu oknu a také jeho dimenze.

`AbstractOrbitCamera` je abstraktní podtřída `AbstractCamera` a reprezentuje kameru, která se dívá na daný bod (*pivot*), od kterého je vzdálena specifickou vzdálenost (*radius*). Z této třídy dědí kamery obou módů `OrbitCamera` a `TrackballCamera`.

Orbit kamera

`OrbitCamera` k výpočtu pohledové matice využívá úhel *rotationX* a *rotationY*. V astronomické obzorníkové soustavě by se jednalo o úhly azimutu a výšky, kdyby na pozici kamery byla hvězda nebo jiné vesmírné těleso [17]. Na obrázku 2.6 je azimut (*rotationX*) reprezentován úhlem θ a může nabývat hodnot 0° až 360°. Výška je označena úhlem φ a nabývá hodnot $-90°$ až 90°. Pozice kamery je znázorněna na povrchu koule bodem C a cíl pozorování kamery (*pivot*) jejím středem P. Proměnná vzdálenosti kamery *radius*

je délka úsečky r. Tyto proměnné tvoří sférické souřadnice, které určují kde na kouli se středem v bodě P a poloměrem r se kamera nachází.



Obrázek 2.6 Diagram principu *orbit* kamery [18].

V metodě `update()` *orbit* kamery je ze zmíněných proměnných vypočítána světová pozice kamery a pomocí metody `glm::lookAt` z knihovny *OpenGL Mathematics* je vytvořena pohledová matice. Této metodě je společně s pozicí kamery předán její *pivot* a vektor určující směr „nahoru“ (*up* vektor), který je získán rotací vektoru $(0, 1, 0)$ úhly azimutu a výšky. Tento vektor upřesňuje otočení kamery kolem směru svého pohledu a zajišťuje, že je horizont světa v tomto módu vždy na obrazovce vodorovný.

Pohybem myši nahoru a dolů na obrazovce je přímo upravován úhel výšky a pohybem doleva a doprava je upravován úhel azimutu. Vzdálenost kamery od pozorovaného bodu lze upravit kolečkem myši. Z vypočtené pohledové matice lze získat *right* a *up* vektory kamery, které jsou navzájem kolmé a „rovnoběžné s obrazovkou“ (rovnoběžné s průmětnou projekce [10, s. 305]), a dle nich pohybem myši posouvat bod pozorování *pivot*, čímž dojde k posunu kamery. Tuto operaci lze provést potažením myši se stisknutým kolečkem myši. Rotaci kamery lze provést tažením myši se stisklým pravým tlačítkem. Tato tlačítka byla zvolena dle doporučení uživatelského průzkumu Víta Zadiny [5, s. 42]. Pravé tlačítko myši bylo zvoleno, protože v rámci této práce do I3T přibyla možnost selekce, s čímž Zadina ve svém průzkumu počítal.

Analogie s obzorníkovou soustavou není zcela přesná. Úhel výšky *rotationY* totiž může nabývat hodnot i mimo rozsah $-90°$ až $90°$, aby bylo možné vertikálním pohybem myši kamerou otáčet bez omezení. Je tedy možné, aby úhel *rotationY* tento rozsah přesáhl, a jeho skutečný rozsah je $(-\infty, +\infty)$. Pokud je hodnota *rotationY* mod 360 v rozsahu $(90, 270)$ tak je kamera „vzhůru nohama“ a její *up* vektor směruje „dolů“ (úhel mezi ním a vektorem $(0, -1, 0)$ je menší než $90°$).

V případě, kdy je kamera vzhůru nohama je ovládání úhlu azimutu obráceno, aby se otáčení scény pohybem myši doleva a doprava chovalo stejně i s kamerou vzhůru nohama. Aby nedošlo k náhlému obrácení ovládání azimutu, kamera zaznamenává, zdali

byla vzhůru nohama v momentě, kdy započal tah myši, a ovládání prohodí až když začne další tah myši. Tedy během jednoho tahu myši nikdy nedojde k náhlé změně ovládání.

Trackball kamera

`TrackballCamera` je *orbit* kameře velice podobná a ovládá se stejným způsobem. Liší se však ve způsobu reprezentace rotace kolem bodu a také ve způsobu interpretace pohybu myši uživatele. Pozice myši je při rotaci promítnuta na 3D plochu „virtuálního trackballu“ (*virtual trackball*), pomocí které lze docílit dojmu, že uživatel rotuje kameru uchycením pomyslné koule a potažením touto koulí otáčí jako by otáčel fyzickou koulí trackballové myši (jedna taková je vidět na obrázku 2.7).

Obrázek 2.7 Trackball myš Logitech TrackMan [19].

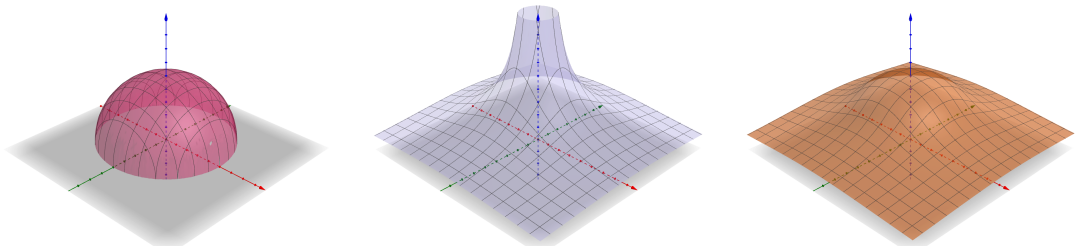
Rotace po virtuálním trackballem je definovaná úhlem a osou rotace, proto `TrackballCamera` používá k její reprezentaci obecnou rotační matici namísto dvou úhlů jako *orbit* kamera. Tato matice postupně akumuluje dílčí rotace snímků, kdy uživatel aktivně otáčí kamerou. Každý takový snímek jsou pozice myši předchozího a aktuálního snímku promítnuty na 3D plochu virtuálního trackballu, která je znázorněna na obrázku 2.8 **(c)**. Na obrázku osy x (červená) a y (zelená) představují 2D pozici myši na obrazovce, ze kterých je dopočítána souřadnice z (modrá), aby se výsledný bod nacházel přímo na oranžové ploše. Z dvou takto promítnutých bodů (předchozí a aktuální pozice myši) a středu prostoru virtuálního trackballu (na obrázku bod, ve kterém se protínají všechny osy) se vytvoří rovina, jejíž normálový vektor definuje osu rotace. Úhel rotace je poté získán z úhlu mezi dvěma vektory vycházejících ze středu prostoru do promítnutých bodů pozice myši.

Plocha virtuálního trackballu je složena z polokoule (2.8 **(a)**), kterou uživatel „uchopuje“, a hyperbolické plochy (2.8 **(b)**), která se polokoule na jisté její kružnici dotýká a vytváří jemný sklon mimo kouli, který pokračuje do nekonečna (v osách x a y). Kdyby byla plocha tvořena pouze polokoulí, tak by při tažení myši mimo kouli došlo k náhlé změně rotační osy, ve chvíli kdy se projekce bodů začnou nacházet zcela mimo polokouli. Tuto náhlou změnu zakřivení hyperbolické plochy eliminuje.

Implementace tohoto výpočtu se nachází ve statické třídě `Trackball` a byla přímo převzata z interní knihovny *pgr-framework*, využívanou k výuce předmětu PGR na ČVUT FEL/FIT. Detailní popis implementace lze také najít v přednáškách tohoto předmětu. Pro účely I3T byla upravena, aby brala v potaz poměr stran okna. Souřadnice myši je třeba nejdříve přepočítat do rozsahu $\langle 0, 1\rangle$ a poloměr pomyslné koule je upraven dle poměru stran, aby pomyslná koule nikdy nebyla v okně příliš malá nebo naopak velká. Pomyslnou

kouli virtuálního trackballu lze v programu vizualizovat zapnutím debugovacího překrytí v hlavní liště programu Help » Debug trackball camera.

Z osy a úhlu rotace se vytvoří rotační matice, která se násobením složí s členskou proměnnou matice dosavadní rotace, která je následně použita v `update` metodě kamery k výpočtu pohledové matice.



(a) Koule **(b)** Hyperbolická plocha **(c)** Plocha virtuálního trackballu

Obrázek 2.8 Složení plochy virtuálního trackballu pomocí funkcí dvou proměnných koule a hyperbolické plochy.

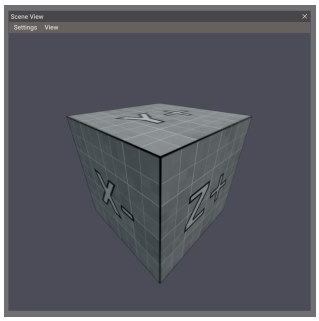
Projekce

K projekci je u všech kamer používána perspektivní transformace. Její matice je také generována za pomoci knihovny GLM. Standardně se k tomuto účelu používá funkce `glm::perspective()`, které stačí předat úhel zorného pole, poměr stran (poměr šířky a výšky obrazu) a vzdálenosti přední a zadní roviny pohledového objemu. Problém s touto metodou je ten, že očekává vertikální úhel zorného pole, tedy úhel mezi horní a dolní rovinou pohledového objemu. Pokud je tento úhel metodě předáván vždy stejný, tak v momentě, kdy je poměr stran větší než jedna a výška okna začne klesat, začne poměr stran prudce růst[9] a s ním se začne roztahovat pohledový objem do stran, aby byl zachován konstantní vertikální úhel zorného pole. To má za následek, že se obsah scény začne rychle vizuálně zmenšovat, protože se zvyšuje šířka pohledového objemu, ale šířka obrazu zůstává stejná. Tedy v situacích, kdy má okno vysoký poměr stran, mohou malé změny v jeho výšce mít velice velký vliv na velikost scény na obrazovce, což je nežádoucí, protože je pak nutné scénu opětovně přibližovat. Vysokého poměru stran je v I3T lehké dosáhnout pouhým dokováním okna na horní polovinu obrazovky. Na obrázcích 2.9 je tento efekt vidět. Pokud by se na obrázku **(b)** dolní okraj okna posunul jen o trochu výš, kostka ve scéně by skoro nebyla vidět.

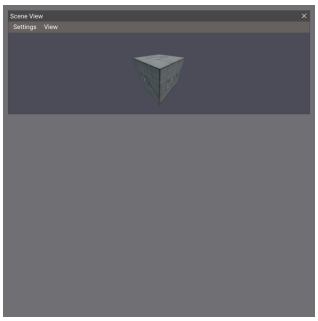
Tento problém lze jednoduše vyřešit tím, že se zadaný FOV úhel použije jako vertikální úhel zorného pole, když je poměr stran menší než jedna. A jako horizontální úhel zorného pole (úhel mezi levou a pravou rovinou pohledového objemu), když je poměr stran větší než jedna. V případě poměru stran rovnému jedné na tom nezáleží.

Knihovna GLM neumožňuje funkci `glm::perspective` předat horizontální úhel zorného pole, ale stačí této funkci předat převrácenou hodnotu poměru stran a následně prohodit první dva prvky na diagonále výsledné matice, což po bližším průzkumu im-

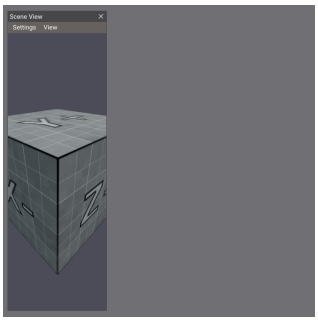
[9] Asymptoticky do nekonečna, stejně jako funkce $\frac{1}{1-x}$ v intervalu $\langle 0, 1)$.



(a) Poměr stran ≈ 1



(b) Poměr stran ≈ 5



(c) Poměr stran ≈ 0.4

Obrázek 2.9 Změna velikosti okna s konstantním vertikálním úhlem zorného pole. Je vidět že na obrázcích **(a)** a **(c)** je velikost kostky stejná, zatímco na obrázku **(b)** je kostka mnohem menší.

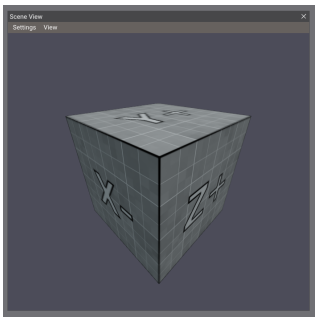
plementace této funkce provede ekvivalentní výpočet jako definice perspektivní matice pomocí horizontálního úhlu zorného pole.

Trochu jasnější postup je sestavit projekční matici pomocí funkce `glm::frustum` a pozice levé, pravé, dolní a horní roviny pohledového objemu nastavit manuálně. Ze zadaného FOV úhlu a vzdálenosti k blízké rovině pohledového objemu lze vypočítat, na jakých pozicích by měly ostatní jeho roviny být v případě, že je poměr stran roven jedné. Skutečný poměr stran je poté případně převrácen, aby byl vždy větší než jedna, a příslušná dvojice rovin je dle něho posunuta dál od „středu“ pohledového objemu vynásobením příslušných pozic hodnotou poměru stran. Tento postup je zachycen v kódu 2.3.

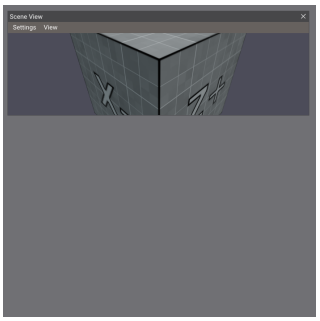
```
1  float scale = m_zNear * tan(glm::radians(m_fov / 2));
2  float l, r, b, t;
3  r = t = scale;
4  l = b = -scale;
5  float aspect = std::min(m_width, m_height) / (float)std::max(m_width, m_height);
6  if (m_width > m_height) {
7      t *= aspect;
8      b *= aspect;
9  } else {
10     l *= aspect;
11     r *= aspect;
12 }
13 return glm::frustum(l, r, b, t, m_zNear, m_zFar);
```

Kód 2.3 Tvorba perspektivní transformace s přepínáním mezi vertikálním a horizontálním úhlem zorného pole dle poměru stran.

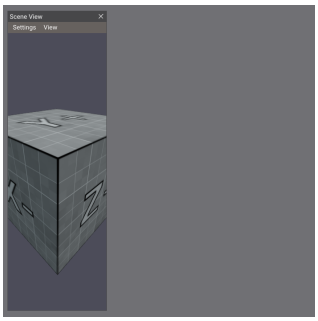
Na obrázcích 2.10 jsou zachyceny stejné situace jako na obrázcích 2.9, ale s použitím této metody. Je vidět, že i při vysokém poměru stran na obrázku má kostka na obrázku **(b)** stejnou velikost jako kostky v ostatních případech.



(a) Poměr stran ≈ 1



(b) Poměr stran ≈ 5



(c) Poměr stran ≈ 0.4

Obrázek 2.10 Změna velikosti okna s přepínáním mezi vertikálním a horizontálním úhlem zorného pole.

Přepínání módů, pohledy a interpolace

Oba módy kamer sjednocuje třída `AggregateCamera`, která mezi nimi přepíná a přitom se prezentuje pouze jako jediná kamera. Při přepínání módu přepočítává interní parametry kamer, aby při změně módu nedošlo k náhlému pohybu kamery. Všechny metody abstraktní třídy `AbstractCamera` deleguje právě aktivní kameře, kterou si drží jako členskou proměnnou.

Kamery podporují přepínání mezi přednastavenými módy *FRONT*, *BACK*, *LEFT*, *RIGHT*, *TOP* a *BOTTOM*. Při přepnutí módu jsou interní parametry kamery nastaveny tak, aby odpovídaly požadovanému pohledu.

Mimo přednastavené módy může být jakákoli kamera vycentrována na aktuální selekci ve 3D scéně pomocí klávesové zkratky 0. Nebo může být vycentrována na celou scénu a zabrat tak ve svém pohledu všechny objekty ve scéně. Tato možnost má klávesovou zkratku Home.

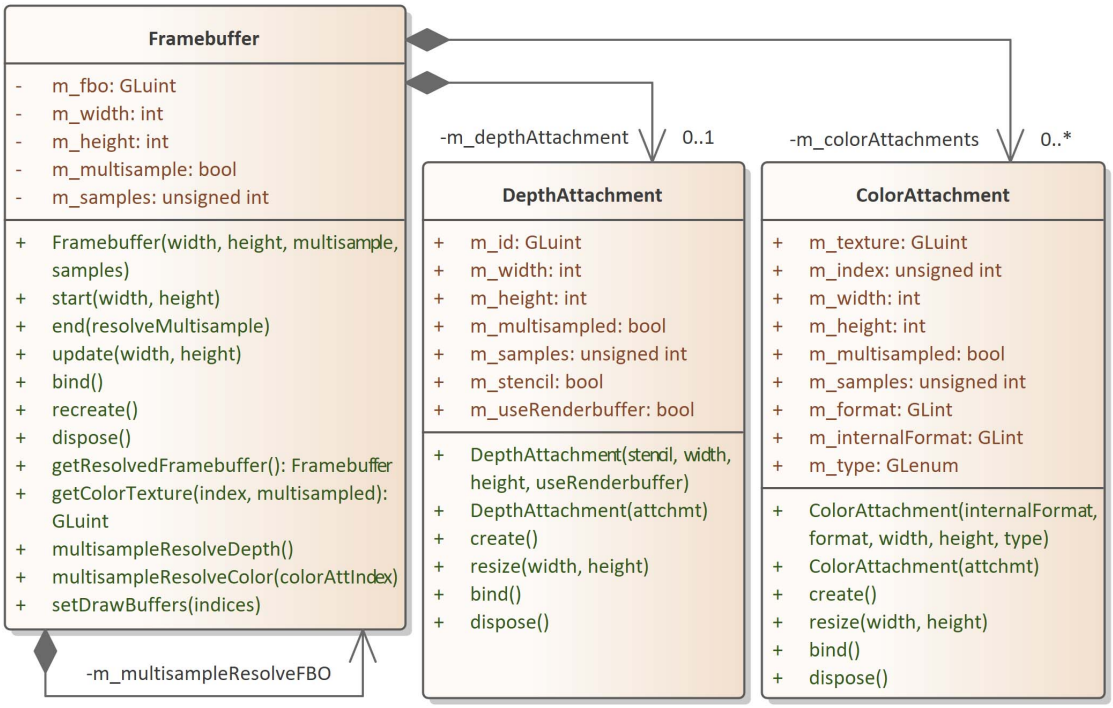
Aby bylo zabráněno náhlým skokům kamery při přepínání módů nebo centrování, kamera obsahuje jednoduchou implementaci interpolace mezi dvěmi pohledovými maticemi. Pomocí metody `interpolate(from, to)` je tato interpolace spuštěna. V momentě volání je nastaven časový bod a během následujících `interpolationPeriod` sekund je mezi těmito maticemi interpolováno pomocí *GLM* metody `glm::interpolate`. Výsledek interpolace dočasně přepíše pohledovou matici, kterou vrací metoda `AbstractCamera::getView()`.

2.4.4 Framebuffer

Pro snadnou práci s OpenGL framebuffer objekty (FBO) byla vytvořena třída `Framebuffer`, která zásadně napomáhá naplnění požadavku ***N05***. Její instance spravuje životní cyklus jednoho objektu framebuffer a abstrahuje většinu jeho funkcí. Pomocí jednoduchého rozhraní je možné FBO vytvořit s požadovanými parametry a specifikovat, ze kterých příloh (*attachments*) se skládá. Rozhraní také umožňuje většinu vlastností framebufferu dynamicky měnit a případně celý framebuffer opětovně inicializovat. Dále je také téměř automaticky zajištěna funkce *multisampling* využívající implementaci multisample antialiasingu (MSAA) v OpenGL.

Framebuffer se skládá z několika příloh (*attachments*). Může obsahovat jednu hloubkovou přílohu (*depth attachment*, případně obsahující *stencil* buffer) a mnoho příloh barev-

ných (*color attachments*). Přílohy reprezentují jedno úložiště (*buffer*) s obrazovými daty. V OpenGL jako úložiště může sloužit například 2D textura nebo *renderbuffer* objekt. *Renderbuffer* je optimalizovaný typ úložiště, které nelze v shaderech číst a třída `Framebuffer` ho volitelně používá pro hloubkovou přílohu [20].



Obrázek 2.11 Diagram tříd balíčku *Viewport/framebuffer*.

Jak jejich název napovídá, hloubkové přílohy umožňují při vykreslování využít paměť hloubky (*depth* buffer) k ukládání hloubek fragmentů a realizaci hloubkového testu (*depth* testu). Barevné přílohy ukládají různé formáty obrazu až s čtyřmi barevnými komponentami (*RGBA*) [21].

`Framebuffer` objekt lze vytvořit pomocí jeho konstruktoru a metod `addColorAttachment()` a `setDepthAttachment()`, které framebufferu přidávají dané přílohy. K vytvoření základního framebufferu s jednou hloubkovou a jednou barevnou přílohou lze využít statickou metodou `Framebuffer::createDefault()`. Jinak je ale po konstrukci nutné specifikovat, z jakých příloh se framebuffer bude skládat. Framebufferu může být nastavena maximálně jedna hloubková příloha a libovolný počet barevných příloh, jak je znázorněno v diagramu tříd na obrázku 2.11.

Objekt `Framebuffer` používá pro tvorbu OpenGL objektů princip odložené inicializace (*lazy initialisation*). Tedy při konstrukci `Framebuffer` objektu a přidání příloh dochází pouze k nastavení příslušných proměnných. K inicializaci všech potřebných OpenGL objektů (FBO, textur, renderbufferů) dojde až při prvním volání funkce `start()`, `update()` nebo `recreate()`. V tu chvíli je vytvořeno FBO a jeho identifikátor je uložen do proměnné `Framebuffer::m_fbo`. Také je všem přílohám zavolána funkce `create()`, která vytvoří příslušný OpenGL objekt a uloží jeho identifikátor do členské proměnné přílohy (`DepthAttachment::m_id` a `ColorAttachment::m_texture`). Funkcí `bind()` jsou přílohy přiřazeny aktivnímu FBO a funkce `resize()` příloze inicializuje OpenGL objekty dle svých

aktuálních hodnot členských proměnných.

V případě objektu `DepthAttachment` je možné proměnnou `m_stencil` nastavit, zdali bude hloubková příloha obsahovat mimo *depth buffer* i *stencil buffer*. Pokud ano, implementace využije 32 bitový formát s 24 bity pro *depth buffer* a 8 bity pro *stencil buffer*, jinak je všech 32 bitů použito pro *depth buffer*. Proměnnou `m_useRenderbuffer` lze nastavit, zdali se jako úložiště obrazových dat bude používat 2D textura nebo objekt *renderbuffer*.

Barevná příloha `ColorAttachment` používá vždy pro úložiště 2D texturu. Její proměnné `m_format`, `m_internalFormat` a `m_type` jsou přímo předány OpenGL funkci `glTexImage2D`, která specifikuje OpenGL objekt textury.

Ještě před tím, než jsou přílohy inicializovány, jim jsou nastaveny stejné rozměry jako objektu `Framebuffer`. V jednom FBO je možné mít přílohy jiných velikostí, ale to současná implementace nepodporuje. Ve většině situací postačí vytvořit zcela oddělený `Framebuffer` objekt obsahující přílohy stejné velikosti.

Celý proces inicializace lze zopakovat metodou `recreate()`, která všechny OpenGL objekty smaže a proces inicializace spustí od začátku.

Příklad použití třídy `Framebuffer` je vidět v kódu 2.4. Při prvním cyklu `for` smyčky proběhne na řádku 6 v metodě `start()` odložená inicializace a framebuffer je připraven metodami `glBindFramebuffer()` a `glViewport()` k vykreslování. Voláním metody `end()` na řádku 8 je jako aktivní nastaven výchozí framebuffer s identifikátorem 0. Výslednou texturu barevné přílohy lze získat metodou `getColorTexture()`, která vrátí identifikátor její textury. Pokud této metodě není specifikován index přílohy, tak vždy vrátí přílohu na první pozici (index 0).

```
1  Framebuffer* fbo = new Framebuffer(200, 100, false, 0);
2  fbo->addColorAttachment(ColorAttachment(GL_RGB, GL_RGB, fbo->m_width, fbo
       ->m_height, GL_UNSIGNED_BYTE));
3  fbo->setDepthAttachment(DepthAttachment(true, fbo->m_width, fbo->m_height,
        true));
4
5  for (...) {
6      fbo->start();
7      // Render stuff
8      fbo->end();
9      GLuint renderedTexture = fbo->getColorTexture();
10 }
```

Kód 2.4 Příklad vytvoření a používání objektu `Framebuffer`

Metodě `start()` lze předat parametry rozměrů *width* a *height*. V případě, že se tyto parametry od aktuálních rozměrů liší, dojde k opětovnému vytvoření OpenGL objektů příloh s novými rozměry (identifikátor FBO však zůstane stejný). Pokud je třeba rozměry framebufferu změnit a není žádoucí framebuffer hned poté nastavit jako aktivní, lze použít metodu `update(width, height)`, kterou metoda `start()` interně volá.

Multisample Antialiasing

Důležitou funkcí třídy `Framebuffer` je zjednodušení práce s *multisampled* verzemi textur a renderbufferů OpenGL. Třídy `Framebuffer`, `DepthAttachment` a `ColorAttachment`

všechny mají proměnné `m_multisampled` a `m_samples`. První z těchto proměnných určuje, zdali je *multisampling* aktivní. Druhá pak, kolik vzorků má *multisampling* používat[10].

Tyto proměnné lze nastavit při konstrukci objektu `Framebuffer` nebo zavoláním metody `setMultisampled()`, která může být volána i po odložené inicializaci objektu a způsobí, že se uvnitř objektu celé FBO i s přílohami smaže a znovu vytvoří metodou `recreate()`.

Stejně jako rozměry jsou všem přílohám při inicializaci nebo v metodách přidávající přílohy nastaveny hodnoty `m_multisampled` a `m_samples` dle hodnot `Framebuffer` objektu, kterému patří, protože není možné, aby v rámci jednoho FBO objektu měly přílohy při vykreslování odlišné nastavení MSAA [8, s. 244].

Při MSAA barevné přílohy používají *multisampled* textury typu `GL_TEXTURE_2D_MULTISAMPLE`. Tyto textury se kvůli vzorkování chovají poněkud odlišně od „standardních“ textur `GL_TEXTURE_2D`. Zejména je nelze přímo vykreslit na obrazovku a je nutné na nich nejdříve provést takzvaný *multisample resolve*. Tento proces vyhodnotí vzorkování a převede *multisampled* texturu opět na „jedno-vzorkovou“ (*single-sampled*) texturu typu `GL_TEXTURE_2D`.

K vyhodnocení vzorkování je třeba pro každou *multisampled* texturu mít k dispozici druhou *single-sampled* texturu stejných rozměrů, do které je *multisampled* textura zkopírována operací `glBlitFramebuffer`, čímž dojde k vyhodnocení. Z toho důvodu instance třídy `Framebuffer` v sobě obsahuje ukazatel na druhou instanci stejné třídy jménem `m_multisampleResolveFBO`. V diagramu na obrázku 2.11 je tato reference sama sebe znázorněna.

Při odložené inicializaci *multisampled* objektu `Framebuffer` je vytvořena tato druhá instance třídy `Framebuffer`, kterou „rodičovská“ *multisampled* instance framebufferu bude využívat k vyhodnocování vzorků. Tento vyhodnocovací framebuffer je inicializován hodnotami proměnných původní „rodičovské“ instance, ale její proměnné `m_multisampled` a `m_samples` jsou výslovně nastaveny na hodnoty `false` a `0`. Figuruje tedy jako *single-sampled* verze *multisampled* objektu `Framebuffer`.

Ukazatel `m_multisampleResolveFBO` vyhodnocovacího framebufferu již má výchozí hodnotu `nullptr` a inicializace framebufferů se tedy nemůže dále cyklit. Jelikož se jedná o kopii, jsou zduplikovány i všechny přílohy původního *multisampled* framebufferu a vyhodnocovací framebuffer obsahuje jejich *single-sampled* kopie ve stejném pořadí.

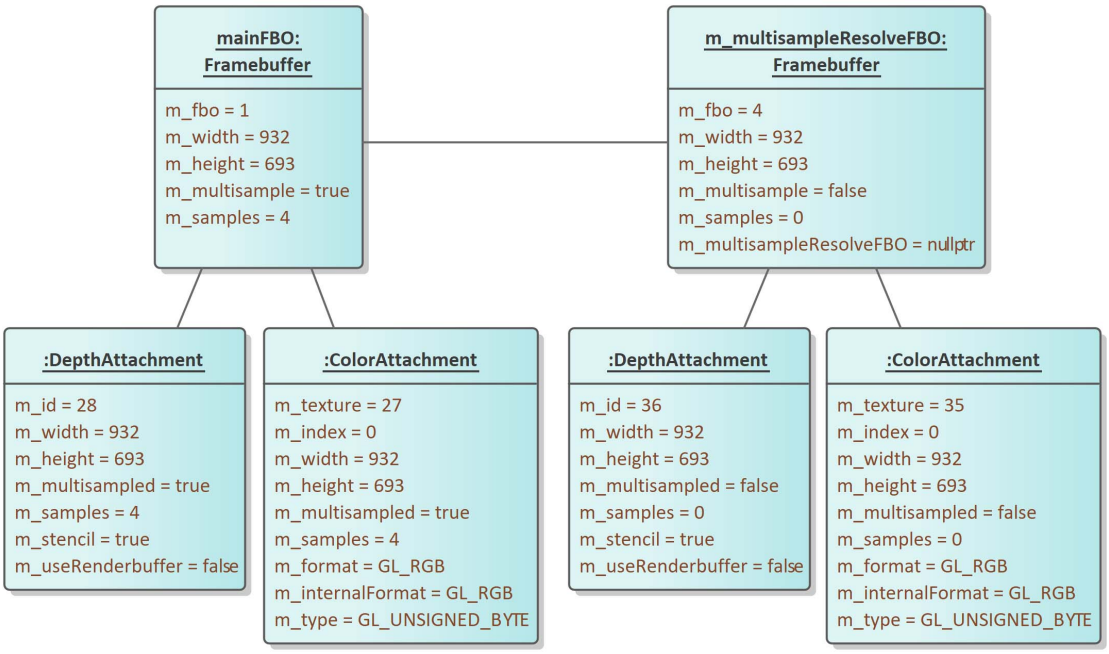
Jakým způsobem může *multisampled* instance třídy `Framebuffer` vypadat, je znázorněno v objektovém diagramu na obrázku 2.12. V něm je příklad základního *multisampled* framebufferu s jednou hloubkovou a jednou barevnou přílohou. Je vidět, že obsahuje ukazatel na druhou instanci `Framebuffer`, která původní framebuffer zrcadlí, s výjimkou nastavení *multisamplingu*.

Po ukončení vykreslování metodou `Framebuffer::end()` je nutné vyhodnocení vzorkování vyvolat. Pokud je metodě `end()` předán parametr `true`, provede se vyhodnocení první barevné přílohy[11], což je užitečné, pokud framebuffer obsahuje pouze jednu barevnou přílohu. Jinak je možné pomocí metod `multisampleResolveColor(index)` a `multisampleResolveDepth()` vyhodnotit specifickou barevnou nebo hloubkovou přílohu.

Výsledná *single-sampled* textura se poté nachází v dané příloze vyhodnocovacího framebufferu. Metodou `getColorTexture()` *multisampled* framebufferu lze přímo získat vyhodnocenou texturu, pokud je její parametr *multisample* nastaven na logickou hodnotu

[10] Musí být jedna z hodnot OpenGL pole `GL_NUM_SAMPLE_COUNTS`. Výchozí hodnota je 4.

[11] Očekává se, že v první příloze se typicky nachází výsledný obraz.



Obrázek 2.12 Objektový diagram jedné instance třídy `Framebuffer` s 4xMSAA, která obsahuje druhou instanci `Framebuffer` k vyhodnocení vzorkování. Přílohy vpravo se id těch vlevo liší hodnotami proměnných `m_multisampled` a `m_samples`.

`false`. Případně je možné získat ukazatel na interní vyhodnocovací framebuffer metodou `getResolvedFramebuffer()` a s tím nakládat, jak je libo. K této metodě je dobré dodat, že v případě vypnutného *multisamplingu* vrátí ukazatel na ten stejný objekt `Framebuffer`, ve kterém je volána.

2.4.5 Vykreslování scény

Vykreslování scény probíhá v její metodě `draw()`, ve které jsou z předaného objektu `SceneRenderTarget` získány jednotlivé objekty `Framebuffer`. Do těch scéna postupně vykresluje a v jednom z nich sestavuje finální obraz. Výsledný `Framebuffer` je na konci vykreslování nastaven `SceneRenderTarget`u jako výstupní pomocí metody `setOutputFramebuffer()`.

Jak je popsáno v podsekci „Scéna a její entity“ v sekci návrhu 2.2.2, pokud je volána první verze metody `draw()` bez parametrů matic kamery `view` a `projection`, jsou tyto matice získány z vlastní kamery scény `m_camera`.

Objekt `SceneRenderTarget`, který je scéně předán, je právě ten objekt, který dříve vytvořila v metodě `createRenderTarget()`. Pokud jí je předán jiný `SceneRenderTarget`, může dojít k chybě, pokud má jinou strukturu. V kódu 2.5 je příklad inicializace objektu `SceneRenderTarget` dle zadaných `RenderOptions`.

Vytvořený `SceneRenderTarget` poté scéna dostane jako parametr v metodě `draw`, kde může jednotlivé framebuffery opět získat voláním metody `SceneRenderTarget::getFramebuffer()` s daným indexem pořadí příslušného framebufferu. V kódu 2.6 je útržek skutečné implementace metody `Scene::draw()`, ve kterém z předaného `SceneRenderTarget`u získává jednotlivé framebuffery, které využívá k vykreslování. Mimo hlavní framebuffer používá pomocné framebuffery pro vykreslování průhlednosti a zvýraznění (selekci)

```
Ptr<SceneRenderTarget> Scene::createRenderTarget(const RenderOptions& options)
{
  Ptr<SceneRenderTarget> renderTarget = std::make_shared<SceneRenderTarget>();
  renderTarget->setRenderOptions(options);

  Ptr<Framebuffer> framebuffer1 = ...
  framebuffer1->addColorAttachment(...);
  framebuffer1->setDepthAttachment(...);
  renderTarget->addFramebuffer(framebuffer1);

  ...
  renderTarget->addFramebuffer(framebuffer2);

  return renderTarget;
}
```

Kód 2.5 Příklad vytvoření objektu `SceneRenderTarget` dle vykreslovacích možností `RenderOptions` ve třídě `Scene`

objektů.

```
Ptr<Framebuffer> mainFBO = renderTarget.getFramebuffer(0).lock();
Ptr<Framebuffer> transparentFBO = renderTarget.getFramebuffer(1).lock();
Ptr<Framebuffer> selectionFBO = nullptr;
Ptr<Framebuffer> selectionBlurFBO = nullptr;
Ptr<Framebuffer> selectionBlurSecondPassFBO = nullptr;
if (drawSelection)
{
    selectionFBO = renderTarget.getFramebuffer(2).lock();
    selectionBlurFBO = renderTarget.getFramebuffer(3).lock();
    selectionBlurSecondPassFBO = renderTarget.getFramebuffer(4).lock();
}
```

Kód 2.6 Získání framebufferů zpět z objektu `SceneRenderTarget`.

Obecně proces vykreslování může být shrnut do několika kroků:

1. Nastavení shader uniformů „globálních“ pro celé vykreslování. Například uniformy osvětlení scény.
2. Aktivace vybraného framebufferu metodou `Framebuffer::start(width, height)` (tento krok framebuffer dle potřeby inicializuje a nastaví jeho dimenze).
3. Vymazání obsahu bufferů metodou `glClear`.
4. Iterace přes všechny instance `Entity` v seznamu entit scény nebo jeho podmnožině:
   a. Případně přeskočit danou entitu v závislosti na jejím nastavení a `DisplayType`.
   b. Zavolat entitě metodu `Entity::render()` s parametry předané pohledové a projekční matice.
5. Ukončení vykreslování do framebufferu metodou `Framebuffer::end()`. V tomto kroku lze případě framebufferu přikázat, ať provede *multisample resolve* voláním pro to určených metod nebo předáním hodnoty `true` přímo metodě `end()`.

6. Nastavení výsledného framebufferu metodou `SceneRenderTarget::setOutputFramebuffer()`. Pokud je výsledný framebuffer *mutlisampled*, musí na něm být před navrácením nejdříve proveden *multisample resolve*.

2.5 Správa zdrojů

Nový správce zdrojů `ResourceManager` se nachází v balíčku *Core/Resources* a zajišťuje načítání zdrojů a jednoduchý přístup k nim dle požadavku ***F12***. Jedná se o *singleton*, jehož instanci lze v jakékoli části kódu získat voláním `Core::ResourceManager::instance()` (zkráceně `RMI`).

Podporované zdroje jsou momentálně shadery, textury a modely. Při načtení shaderu je navrácen identifikátor vytvořeného OpenGL objektu. Stejně tomu tak je i pro textury.

Modely jsou načteny do objektu třídy `Mesh`, která drží buď manuálně generovanou geometrii (např. vygenerovanou pomocí třídy `Shaper`) nebo geometrii načtenou ze souboru knihovnou *assimp*.

Nejzákladnější tři metody pro získání těchto zdrojů z rozhraní `ResourceManager`u jsou:

- texture(path)
  Této metodě je předána cesta k obrázkovému souboru textury a je navrácen její OpenGL identifikátor.

- shader(vertpath, fragpath)
  Této metodě je předána cesta k glsl vertex shaderu a také k glsl fragment shaderu. Navrácen je identifikátor OpenGL shaderu.

- mesh(path)
  Této metodě je předána cesta k souboru 3D modelu a je navrácen ukazatel na objekt `Mesh`. Tento ukazatel je takzvaně *non-owning* a není třeba se životním cyklem navráceného objektu zabývat.

Smyslem těchto metod je, že pokud se v jakékoli části kódu vyskytne potřeba získat referenci například na daný model, stačí zavolat `RMI.model(cesta)` a tento objekt získáme.

Hlavní předností správce zdrojů je zajištění, že se žádné zdroje nenačítají víckrát. V tomto případě při prvním volání metody `model()` `ResourceManager` vytvoří z cesty unikátní identifikátor *ID*, který vznikne zahašováním řetězce cesty. Načtený zdroj a identifikátor jsou zabaleny do objektu `Resource`, který je následně přidán do hašovací tabulky `m_resourceMap` s klíčem daného identifikátoru.

Pří následujícím pokusu o načtení modelu stejnou cestou již správce zdrojů neprovede načítání, nýbrž cestu opět zahašuje a zkusí ji najít v hašovací tabulce. Pokud jí najde tak navrátí již existující model, přechovávaný v objektu `Resource` pomocí `void` ukazatele.

Načítání jednotlivých zdrojů je implementováno v soukromých metodách `loadShader()`, `loadTexture()` a `loadModel()`. Shadery a textury se načítají s pomocí knihovny *pgr-framework* a načítání modelů zajišťuje třída `Mesh`.

Tento systém sám o sobě pracuje velice dobře. Pokud je obava, že bude načítání spuštěno v nevhodnou chvíli, je možné zdroje přednačíst někdy na začátku a navrácené reference na zdroje zahodit. Později však, když je nutné reference na zdroje získat, budou volání načítacích metod správce zdrojů pouze vracet reference na již načtené modely.

Správce zdrojů tento systém rozvíjí o krok dále a umožňuje modely „označkovat“ takzvanými *aliasy*. Metody zmíněné výše všechny mají další přetížené verze, které navíc přijímají jako první argument řetězec *alias*:

- texture(alias, path)
- shader(alias, vertpath, fragpath)
- mesh(alias, path)

Řetězec *alias* může obsahovat cokoli, měl by však být unikátní danému zdroji. Tyto metody provádí to samé jako jejich verze bez *aliasu*, ale zároveň pro zdroje, dané cestami registruje předaný *alias* v další hašovací tabulce správce zdrojů `m_aliasMap`. Ta umožňuje daným aliasem získat referenci na zdroj, stejně jako to lze identifikátorem z hašovací tabulky `m_resourceMap`.

Poté co je zdroj načten jednou z metod výše již není třeba znát přesnou jeho cestu, ale k získání reference na něj stačí znát jeho *alias*. To umožňují metody:

- textureByAlias(alias)
- shaderByAlias(alias)
- meshByAlias(alias)

Systém *aliasů* umožňuje jednoduše implementovat správu *výchozích* a *vlastních* zdrojů jak je popsáno v požadavku ***F12***. Stačí totiž jednou zdroje z cesty a daného aliasu načíst. A poté lze všechny ze správce zdrojů získat předáním pouhého seznamu *aliasů*. Tento přístup je již v implementaci využit pro správu modelů výchozích a podpora vlastních modelů bude přidána, až bude v aplikaci I3T dokončena funkce serializace.

Většina logiky implementace `ResourceManager`u se nachází v metodě `getData()`, která je zcela agnostická typu zdrojů, se kterými zachází. Stejně tak je agnostický i objekt `Resource`, který data drží univerzálním `void` ukazatelem. Jednotlivé metody načítající zdroje metodu `getData()` všechny používají a pouze reagují na její instrukce (zdali zdroj načíst nebo přetypovat existující referenci a vrátit ji).

Do správce zdrojů by tedy případně mělo být velice jednoduché implementovat správu i jiných typů objektů, než zde byly popsány. Nebo třídu `ResourceManager` zcela abstrahovat a metody specifické typům zdrojů přesunout do implementační podtřídy.

Mesh

Třída `Mesh` z balíčku *Core/Resources* reprezentuje 3D model, který lze přímo vytvořit statickými metodami `Mesh::create()` předáním dat geometrie a nebo je načítat ze souborů pomocí metody `Mesh::load()`, čímž realizuje požadavek ***F08***. K načítání 3D modelů ze souboru se používá knihovna *assimp*, která podporuje širokou škálu 3D formátů a načítá je do své společné datové struktury typu `aiScene`.

Objekt `Mesh` se skládá z částí `MeshPart` a po vytvoření ihned nahrává data geometrie grafické kartě pomocí funkcí OpenGL. Geometrii nahrává k vykreslování metodami `glDrawArrays()` nebo `glDrawElementsBaseVertex()` dle způsobu vytvoření. Těmito metodami může vykreslovat pomocí primitivů `GL_LINES` nebo `GL_TRIANGLES`.

Data vrcholů se skládají z několika atributů. Povinným atributem je pozice vrcholu a nepovinně může geometrie také obsahovat atributy normálového vektoru, souřadnic textur, *tangent* vektoru a barvy vrcholu.

Části `MeshPart` obsahují informaci o materiálu a texturách jednotlivých částí 3D modelu.

Objekt `Mesh` lze vykreslit metodou `render()`, případně lze vykreslit části `MeshPart` individuálně pomocí metody `renderMeshPart()`.

Shaper

Třída `Shaper` umožňuje velice prostým rozhraním vytvářet jednoduchou geometrii skládající se z primitivů čar nebo trojúhelníků. Tato třída realizuje funkční požadavek ***F06*** a využívá se k tvorbě geometrie například pro bazické vektory transformací ***F04***.

Třída obsahuje jako statické proměnné `ResourceManager` *aliasy*, na některé jednoduché objekty. Například jednotkovou krychli složenou z plošek či čar nebo „obrazovkovou“ plošku, využívanou k vykreslení obsahu shadery přes celou obrazovku pro skládání obrazu v shaderech `BoxBlurShader`, `ScreenOverlayShader`, `SelectionCompositeShader`, `WBOIT-CompositeShader` nebo v shaderu pro vykreslování nekonečné mřížky `GridShader`.

2.6 Propojení viewportu s uživatelským rozhraním

Způsob propojení viewportu s uživatelským rozhraním byl z velké části již konceptuálně popsán v sekci 2.2.2. Implementace tohoto propojení se skládá z úpravy již existujících tříd z balíčku *GUI*:

- `ViewportWindow` realizuje dokovatelné okno hlavního pohledu na scénu nezávislou kamerou a souvisí s požadavky ***F10*** a ***F11***. Do existující třídy je třeba přidat zpracování uživatelského vstupu a vykreslování nového viewportu pomocí metody `Viewport::drawViewport()`.
- `WorkspaceModel` vytváří UI krabičky „Model“ ve 2D workspace. Souvisí s požadavkem ***F01***. Existující krabičku je třeba reprezentovat entitou ve 3D scéně a také je nutné do jejího UI přidat vykreslování náhledu jejího 3D modelu pomocí metody `Viewport::drawPreview()`.
- `WorkspaceCamera` je zodpovědná za UI krabičky „Kamera“. Souvisí s požadavkem ***F02***. Krabička musí být ve 3D scéně reprezentována modelem kamery a jejího pohledového objemu.
- `WorkspaceScreen` zajišťující krabičku „Screen“ ve 2D workspace, která umožňuje zobrazit pohled dané kamery ve scéně dle požadavku ***F03***. Do jejího rozhraní je třeba přidat vykreslování tohoto pohledu pomocí metody `Viewport::drawScreen()`.

V každé z těchto tříd je využité veřejné rozhraní `Viewport`, ke kterému mají vždy přístup pomocí instance *singletonu* třídy `Core::Application`, která instanci třídy `Viewport` vytváří a inicializuje při spuštění aplikace I3T. Třídy viewportu se všechny nachází v namespace jménem `Vp`. V kódu mimo balíček *Viewport* se tedy například na hlavní třídu viewportu `Viewport` odkazuje pomocí `Vp::Viewport`. Namespace `Vp` bude v některých případech v textu uváděno k odlišení tříd nového viewportu od tříd ostatního kódu.

2.6.1 Dokovatelné okno viewportu

Třída `ViewportWindow` realizuje dokovatelné okno hlavního pohledu na scénu nezávislou kamerou. Nachází se v balíčku *GUI/Elements/Windows* a je součástí namespace *UI*. K vykreslení scény využívá metodu `Viewport::drawViewport()`. Referenci na instanci viewportu tato třída dostane přímo v konstruktoru, když je její instance vytvořena třídou `Application`.

Mimo referenci na viewport také obsahuje objekty `Vp::RenderOptions` a `Vp::DisplayOptions`, které specifikují nastavení hlavního viewportu. Tato nastavení je možné v okně upravovat pomocí horní menu lišty. V pravém horním rohu je umístěn indikátor os implementovaný v sekci 2.12.6. Obsah celého okna v pozadí vyplňuje neprůhledná textura pohledu na scénu.

V kódu 2.7 je vidět útržek implementace vykreslování tohoto okna. Nejdříve je na řádcích 1–5 vykreslena menu lišta viewportu[12]. Následně je na řádcích 10–14 vykreslen indikátor os a manipulátory. To, jestli uživatel v tomto snímku interagoval s menu lištou nebo manipulátory, je zaznamenáváno do proměnných `menuInteraction` a `manipulatorInteraction`. Pokud s těmito prvky uživatel neinteragoval a okno je právě aktivní, tak je na řádcích 19–21 vyhodnocen uživatelský vstup pro viewport samotný. Zdali je okno aktivní (má *focus*) se lze dotázat statického `InputManager`u metodou `isInputActive()`.

Následuje vykreslení viewportu funkcí `drawViewport()` na řádku 24. Této metodě je předána členská proměnná `m_renderTarget` typu `SceneRenderTarget`, do které je uložena výsledná textura. Při prvním volání této metody (v prvním snímku) je v této proměnné jen prázdný ukazatel a uvnitř metody `drawViewport` je mu přiřazen správně inicializovaný objekt `SceneRenderTarget` (viz 2.2.2). Dále jsou metodě předány rozměry okna `windowWidth` a `windowHeight` a nakonec také objekty nastavení vykreslování `m_renderOptions` a `m_displayOptions`.

Na řádku 25 je po vykreslení získán z objektu `m_renderTarget` výsledný `Vp::Framebuffer` obsahující vykreslenou texturu. Ta je nakonec na řádku 27 předána knihovně Dear ImGui k zobrazení zařazením příkazu vykreslení obrázku aktuálnímu `DrawList`u okna pomocí funkce `DrawList::AddImage()`. Této funkci je předán identifikátor výsledné textury, souřadnice rohů okna na obrazovce (`windowMin`, `windowMax`) a také UV souřadnice textury (`ImVec2(0, 1), ImVec2(1, 0)`), které texturu vertikálně překlopí, jelikož Dear ImGui má počátek souřadnic $(0, 0)$ „vlevo nahoře“ a OpenGL „vlevo dole“.

V kódu se také objevují volání objektu `m_channelSplitter`. Jedná se o Dear ImGui objekt typu `ImDrawListSplitter`, který umožňuje zaměnit pořadí vykreslování. Specificky v tomto případě je pomocí něho odloženo vykreslování manipulátorů (a indikátoru os) až za vykreslování viewportu. Pokud by byly vykresleny před ním, textura viewportu by je zakrývala. Ještě před vykreslením je však nutné z jejich vykreslovacích metod zjistit, zdali došlo k interakci s uživatelem. Jsou tedy vykresleny před viewportem, ale pomocí `ImDrawListSplitter`u jsou v `DrawList`u Dear ImGui posunuty na pozici za příkazem vykreslení viewportu.

Na řádku 7 je „kanál“ `DrawList`u rozdělen na 2 části. První část s indexem 1 obsahuje vykreslené manipulátory a indikátor os, druhá část s indexem 0 (menším než index první části) obsahuje vykreslený viewport. Na řádce 31 jsou tyto části v `DrawList`u opět spojeny v pořadí jejich indexů.

[12] Je dobré zmínit, že „vykreslení“ pomocí Dear ImGui funkci typicky vyhodnocuje i uživatelský vstup.

```
bool menuInteraction = false;
if (ImGui::BeginMenuBar()) {
  menuInteraction |= showViewportMenu();
  ImGui::EndMenuBar();
}

m_channelSplitter.Split(ImGui::GetWindowDrawList(), 2);
m_channelSplitter.SetCurrentChannel(ImGui::GetWindowDrawList(), 1);

m_viewport->m_manipulators->drawViewAxes(m_windowPos, m_windowSize);
bool manipulatorInteraction = false;
if (m_viewport->getSettings().manipulator_enabled) {
  manipulatorInteraction |= m_viewport->m_manipulators->drawManipulators(m_windowPos,
     m_windowSize);
}

m_channelSplitter.SetCurrentChannel(ImGui::GetWindowDrawList(), 0);

if (InputManager::isInputActive(getInputPtr()) && !menuInteraction && !manipulatorInteraction
     && m_renderTarget) {
  glm::vec2 relativeMousePos = WindowManager::getMousePositionForWindow(this);
  m_viewport->processInput(ImGui::GetIO().DeltaTime, relativeMousePos, m_windowSize);
  m_viewport->processSelection(m_renderTarget, relativeMousePos, m_windowSize);
}

m_viewport->drawViewport(m_renderTarget, windowWidth, windowHeight, m_renderOptions,
     m_displayOptions);
Ptr<Vp::Framebuffer> framebuffer = m_renderTarget->getOutputFramebuffer().lock();
if (framebuffer) {
  ImGui::GetWindowDrawList()->AddImage((void*)(intptr_t)framebuffer->getColorTexture(),
     windowMin, windowMax, ImVec2(0, 1), ImVec2(1, 0));
} else {
  ImGui::Text("Failed to draw viewport!");
}
m_channelSplitter.Merge(ImGui::GetWindowDrawList());
```

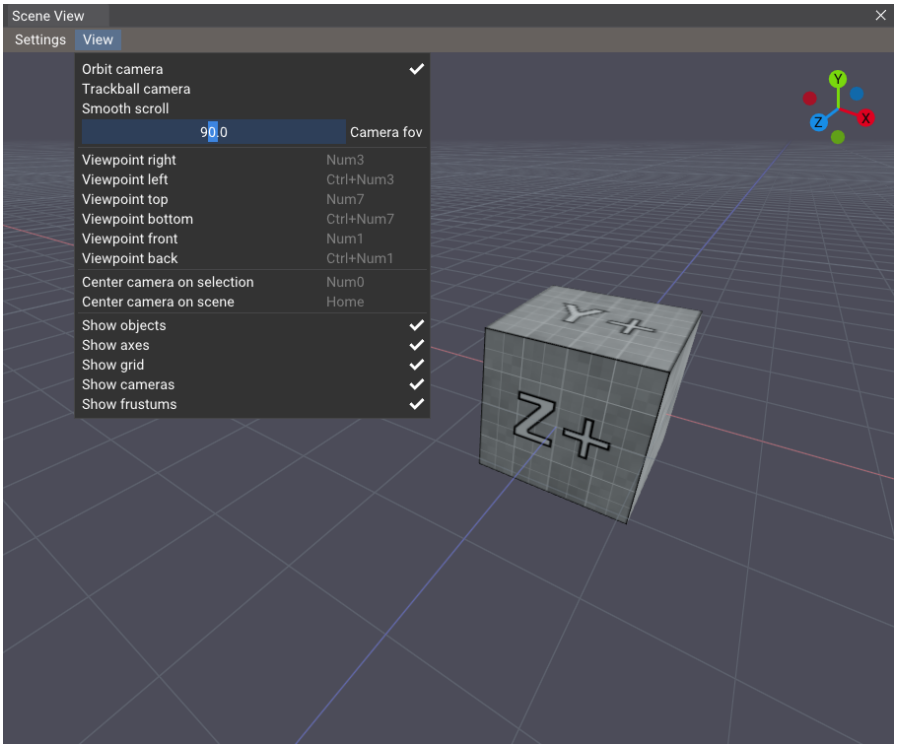
Kód 2.7 Útržek implementace vykreslování hlavního okna viewportu.

Na obrázku 2.13 je vidět podoba okna hlavního pohledu prohlížeče scény. Vlevo nahoře se nachází menu lišta s položkami Settings a View. Menu „Settings“ obsahuje základní možnosti MSAA, průhlednosti, zvýraznění objektů a manipulátorů. Detailnější nastavení některých funkcí se nachází v okně „Preferences“ dostupném z hlavní menu lišty programu Edit 〉 Preferences.

Menu „View“ je na obrázku 2.13 rozbalené a obsahuje možnosti zobrazení různých typů objektů (mřížky, kamer, pohledových objemů apod.), které přímo upravují objekt `Vp::DisplayOptions` předávaný vykreslování. Také obsahuje možnosti kamery jako nastavení jejího módu, FOV a možnosti přepnutí kamery do přednastaveného pohledu a jejího vycentrování na scénu nebo vybraný objekt. Některé tyto možnosti jsou v implementaci `ViewportWindow` pomocí `InputManager`u namapované na různé klávesové zkratky.

2.6.2 Krabička „Model“

Krabička „Model“ získá při konstrukci od viewportu referenci na objekt typu `Vp::SceneModel`, který jí ve 3D scéně reprezentuje. Tuto referenci si uchovává ve své členské proměnné *m_viewportModel*. Získání této reference je jednoduše provedeno následujícím voláním:



Obrázek 2.13 Hlavní okno nového viewportu s rozbaleným menu View.

```
1 m_viewportModel = App::get().viewport()->createModel(getId());
```

Obdobným způsobem je tato reference zničena v destruktoru krabičky pomocí:

```
1 App::get().viewport()->removeEntity(m_viewportModel);
```

Pomocí této reference může krabička upravovat vlastnosti 3D modelu ve scéně a pomocí svého kontextového menu upravovat všechny jeho potřebné vlastnosti. Pomocí proměnných entity modelu `m_visible` a `m_showAxes` lze nastavit viditelnost modelu a jeho bazických vektorů transformace. Proměnné `m_opaque` a `m_opacity` nastavují vlastnosti průhlednosti. Pro poloprůhledné objekty lze případně pomocí `m_backFaceCull` povolit či zakázat vykreslování primitiv, které jsou odvráceny od kamery, což může pomoct snížit počet vizuálních artefaktů v případě, že je pro vykreslování použita standardní technika seřazené průhlednosti. Pro vykreslování náhledu modelu v této krabičce se v zájmu výkonu tato standardní technika využívá, a proto je typicky žádoucí mít `m_backFaceCull` nastaven na logickou hodnotu `true`.

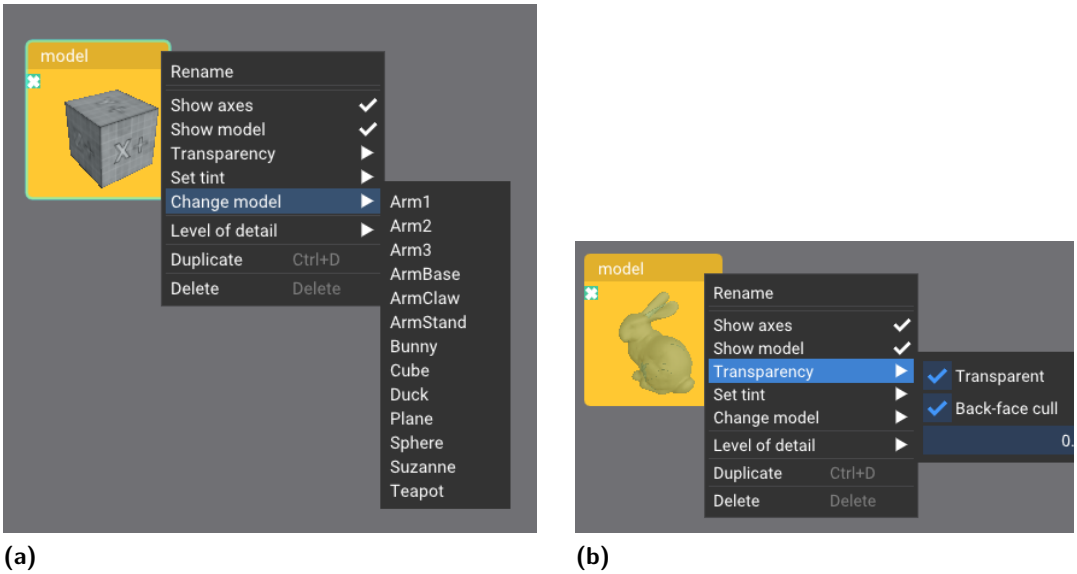
V kontextovém menu krabičky „Model“ se také nachází možnost změnit 3D model, který krabičku ve scéně reprezentuje, na jeden z výchozích modelů, které `ResourceManager` načítá při spuštění aplikace z konfiguračního souboru JSON (viz 2.5). Možnost načtení vlastního modelu uživatelem v aplikaci momentálně není, protože tato funkce úzce souvisí s funkcí serializace scén, která není dokončena a pracuje na ní jeden z ostatních členů týmu I3T. Systém *aliasů* `ResourceManager`u je však na tuto funkci připraven a je možné ji docílit stejným způsobem, jakým je implementována správa modelů výchozích. Stačí tedy pro načítání modelů vytvořit UI dialog a v implementaci serializace ukládat příslušné

cesty a *aliasy* modelů.

Změnit 3D model ve scéně může krabička voláním metody `setModel(alias)` s argumentem *aliasu* požadovaného modelu. Ten může získat ze seznamu výchozích modelů voláním:

```
1 ResourceManager::instance().getDefaultResources(ResourceType::Model);
```

To vrací seznam *aliasů*, ze kterých může být jeden předán metodě `setModel()`. Barvy 3D modelu entity `SceneModel` lze pozměnit změnou barvy proměnné `m_tint`, která je později předávána shaderu `PhongShader`, který do textur modelu lehce přimíchá danou barvu. Podoba nové krabičky model je na obrázku 2.14, na kterém je vidět i její kontextové menu s rozbaleným podmenu změny 3D modelu.



(a) (b)

Obrázek 2.14 Nová podoba krabičky „Model“ s již plně funkčním kontextovým menu. Vlevo je kostka s menu změny modelu, vpravo je poloprůhledný králíček nabarvený na zeleno se zaškrtnutou možností průhlednosti.

V obsahu („těle“) krabičky se nachází náhled 3D modelu krabičky. Ten se vykresluje pomocí metody `Viewport::drawPreview()`, které je předána reference na entitu modelu `m_viewportModel`. Podobným způsobem jako v hlavním okně viewportu probíhá vykreslování do instance třídy `Vp::SceneRenderTarget`, kterou si krabička uchovává jako členskou proměnnou. Stejným způsobem jako v hlavním okně probíhá i předání výsledné textury knihovně Dear ImGui. Jelikož není třeba vyhodnocovat vstup, je implementace vykreslování pouze na několik řádků, jak je vidět v kódu 2.8.

```
1 int width = m_textureSize.x * diwne.getWorkAreaZoom();
2 int height = m_textureSize.y * diwne.getWorkAreaZoom();
3 App::get().viewport()->drawPreview(m_renderTarget, width, height, m_viewportModel,
      m_renderOptions);
4 Ptr<Vp::Framebuffer> framebuffer = m_renderTarget->getOutputFramebuffer().lock();
5 if (framebuffer) {
6   ImGui::Image((void*)(intptr_t)framebuffer->getColorTexture(), ImVec2(width, height), ImVec2
      (0.0f, 1.0f), ImVec2(1.0f, 0.0f));
7 } else {
8   ImGui::Text("Failed to draw preview!");
9 }
```

Kód 2.8 Útržek implementace vykreslování náhledu krabičky „Model“.

Stejně jako `ViewportWindow` přechovává krabička „Model“ možnosti `Vp::RenderOptions`, které nastavují viewport, aby vykresloval do textury s průhledným pozadím a v co nejméně náročné kvalitě (vypnutí MSAA a WBOIT). `Vp::DisplayOptions` nejsou pro vykreslování náhledu modelu potřeba, jelikož je v náhledové scéně vždy jen jeden objekt. Požadované rozměry výsledné textury náhledu se mohou měnit dle úrovně přiblížení 2D workspace.

2.6.3 Krabička „Kamera“

Krabička kamery v sobě nic nevykresluje, ale je ve 3D scéně reprezentována velice podobně jako krabička „Model“. Reference na její reprezentaci je typu `Vp::SceneCamera`, který z typu `Vp::SceneModel` dědí, a tedy jsou kameře k dispozici stejné možnosti jako krabičce „Model“. Nicméně ve svém kontextovém menu umožňuje upravit jen svojí viditelnost a viditelnost svých bazických vektorů. Jako 3D model je jí vždy přednastaven speciální model kamery, který je neměnný[13].

Krabička „Kamera“ má však navíc své vlastní možnosti týkající se reprezentace jejího pohledového objemu (*frusta*), které navíc kromě modelu své kamery vykresluje. Entita `SceneCamera` spravuje dvě další entity typu `Vp::FrustumObject`. Tyto entity využívají `Vp::FrustumShader` k vykreslení reprezentace pohledového objemu kamery. První vykresluje frustum poloprůhlednými ploškami a druhá pomocí neprůhledných barevných linek. Proměnnou objektu `SceneCamera` `m_fillFrustum` je možné vykreslování poloprůhledné výplně zakázat a proměnnou `m_showFrustum` lze vykreslování frusta zakázat úplně. Barva výplně a linek frusta se ovládá proměnnými `m_frustumColor` a `m_frustumOutlineColor`. Všechny tyto možnosti se nachází v kontextovém menu krabičky.

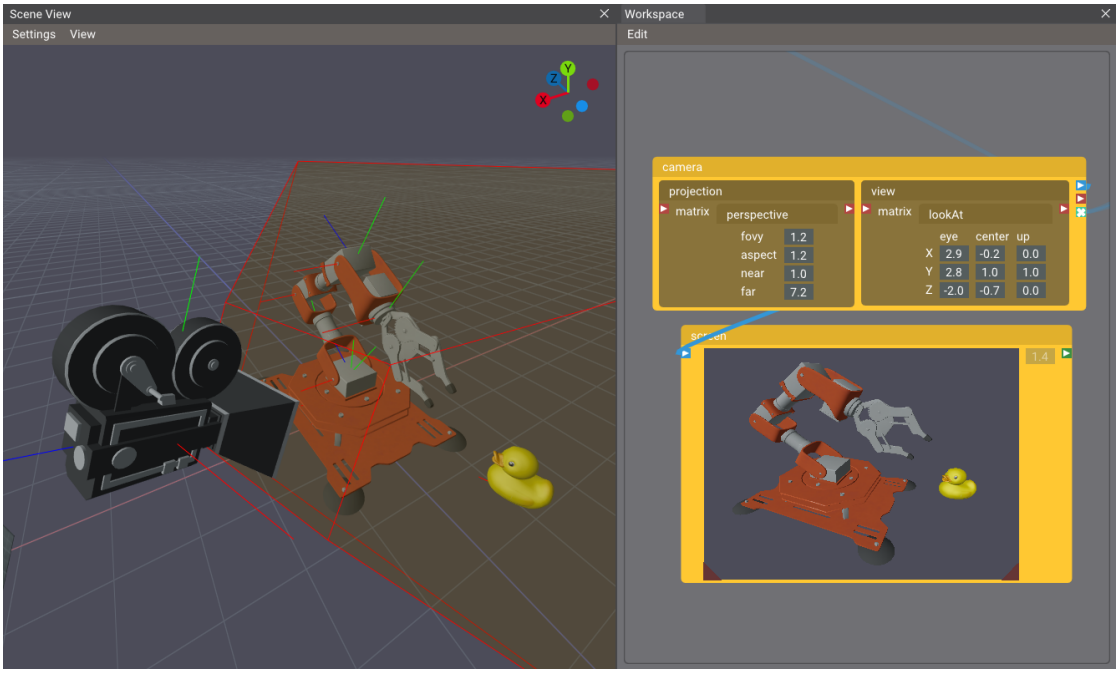
Model kamery i reprezentace jejího pohledového objemu je vidět na obrázku 2.15.

2.6.4 Krabička „Obrazovka“

Krabička „Obrazovka“, na rozdíl od krabiček kamer a modelů, žádnou reprezentaci ve scéně nemá. Jejím účelem je pouze zobrazit scénu pohledem definovaným projekční a pohledovou maticí, kterou dostává na svém vstupu. Ke krabičce obrazovky se zpravidla připojuje jedna krabička „Kamera“, která obrazovce předá své matice, a jejíž pohled je poté v krabičce obrazovky zobrazen.

Vykreslování pohledu obrazovky probíhá velice podobně vykreslení náhledu krabičky „Model“. K vykreslení je použita metoda `Viewport::drawScreen()`, která mimo již jistě

[13] Změnit model samozřejmě lze, ale stačí jeden model pro kamery.



Obrázek 2.15 Kamera a obrazovka pozorující jednu z testovacích scén.

povědomé parametry přijímá parametry pohledové a projekční matice, definující požadovaný pohled. Tyto matice krabička získává ze svého vstupu voláním `getNodebase()->getData().getScreen()` na řádku 5 v kódu 2.9. Volání `getNodebase()` je detailněji popsáno v následující sekci 2.7.

```
1  int width = m_textureSize.x * diwne.getWorkAreaZoom();
2  int height = m_textureSize.y * diwne.getWorkAreaZoom();
3
4  // Get projection and view matrix from screen input
5  std::pair<glm::mat4, glm::mat4> screenValue = getNodebase()->getData().getScreen();
6
7  App::get().viewport()->drawScreen(m_renderTarget, width, height, screenValue.second,
       screenValue.first, m_renderOptions, m_displayOptions);
8  Ptr<Vp::Framebuffer> framebuffer = m_renderTarget->getOutputFramebuffer().lock();
9  if (framebuffer) {
10   ImGui::Image((void*)(intptr_t)framebuffer->getColorTexture(), ImVec2(width, height), ImVec2
       (0.0f, 1.0f), ImVec2(1.0f, 0.0f));
11 } else {
12   ImGui::Text("Failed to draw the screen!");
13 }
```

Kód 2.9 Útržek implementace vykreslování pohledu kamery krabičky „Obrazovka“.

Obrazovka metodě `Viewport::drawScreen()` stejně jako hlavní okno předává možnosti `Vp::RenderOptions` a `Vp::DisplayOptions`. `RenderOptions` jsou nastaveny podobně jako u krabičky modelu, tedy s vypnutým antialiasingem a průhledností WBOIT. Na rozdíl od náhledu modelu má výsledná textura obrazovky neprůhledné pozadí, protože v tomto případě není žádoucí, aby obrazovka splývala s „tělem“ krabičky.

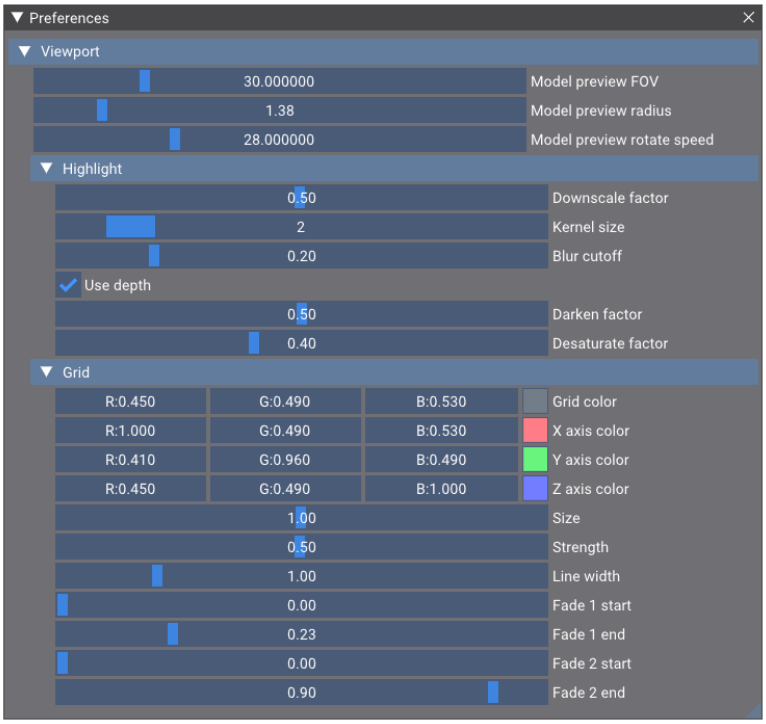
Pomocí `DisplayOptions` je obrazovce zakázáno vykreslování ostatních kamer (a jejich pohledových objemů), mřížky a bazických vektorů. Výsledný obraz tedy obsahuje jen

3D modely samotné a žádné další rušivé elementy. Možnosti `DisplayOptions` lze kdykoli za běhu upravit, a tedy je případně snadné individuálním obrazovkám do kontextového menu tyto možnosti přidat.

Finální podoba krabičky „Obrazovka“ je vidět vpravo dole společně s krabičkou kamery na obrázku 2.15.

2.6.5 Okno nastavení

V odděleném okně v horní liště aplikace Edit › Preferences se nachází jakékoli ostatní nastavení viewportu, které se do lišty okna viewportu samotného nevešla nebo které není třeba, aby měl uživatel při práci s programem po ruce. Uživatelské rozhraní tohoto okna povětšinou přímo upravuje obecná nastavení viewportu `ViewportSettings`, na které lze získat referenci voláním `App::get().viewport()->getSettings()`. Aktuální podoba tohoto okna je vidět na obrázku 2.16.



Obrázek 2.16 Okno nastavení

2.7 Propojení s jádrem I3T

V minulé sekci 2.6 byla v popisu implementace krabiček vynechána jedna důležitá část, a to jakým způsobem entity `SceneModel` a `SceneCamera` reprezentující krabičky získávají hodnoty samotných transformací, které mají reprezentovat. Nejdříve je nutné nastínit, jakým způsobem jsou krabičky 2D workspace implementovány.

Krabičky ve 2D workspace pouze zrcadlí stav jim ekvivalentních krabiček z jádra aplikace. Slouží tedy pouze jako vrstva uživatelského rozhraní pro skutečné krabičky v jádře, které provádí veškerou logiku jako výpočty a propagaci změn. Každá krabička 2D workspace si drží referenci na ekvivalentní krabičku uvnitř jádra I3T. Workspace krabičky dědí ze základního typu `WorkspaceNodeWithCoreData`, který obsahuje členskou proměnnou jménem `m_nodebase`, což je instance třídy `Core::Node`. Ta se nachází v jádře aplikace (v balíčku *Core/Nodes*) a slouží jako základní typ všech krabiček jádra. Z jakékoli krabičky 2D workspace lze její ekvivalentní krabičku jádra získat voláním `this->getNodebase()`.

V krabičce „Model“ pro nastavení správné transformace tedy stačí její entitě `SceneModel` při vykreslování každý snímek nastavit proměnnou modelové matice `m_modelMatrix` na hodnotu modelové matice krabičky modelu v jádře (typu `Core::Model`). To lze provést útržkem kódu 2.10.

```
1 Core::Model* modelNode = dynamic_cast<Core::Model*>(this->getNodebase().get());
2 if (modelNode) {
3     m_viewportModel.lock()->m_modelMatrix = modelNode->m_modelMatrix;
4 }
```

Kód 2.10 Přiřazení modelové matice entitě `SceneModel` z GUI krabičky „Model“.

Velice podobně se aktualizují data i entity `SceneCamera` v krabičce „Kamera“. Entitě kamery se nenastavuje matice modelová, nýbrž matice pohledová a projekční, které určují pozici, orientaci a pohledový objem kamery. Nastavení těchto matic je provedeno ve vykreslovací metodě třídy `WorkspaceCamera` útržkem kódu 2.11. V něm se reference na krabičku v jádře přetypuje na typ `Core::Camera`, ze které jsou entitě `SceneCamera` nastaveny příslušné matice.

```
1 Core::Camera* cameraNode = dynamic_cast<Core::Camera*>(this->getNodebase().get());
2 if (cameraNode) {
3   auto viewportCameraPtr = m_viewportCamera.lock();
4   viewportCameraPtr->m_projectionMatrix = cameraNode->m_projectionMatrix;
5   viewportCameraPtr->m_viewMatrix = cameraNode->m_viewMatrix;
6 }
```

Kód 2.11 Přiřazení pohledové a projekční matice entitě `SceneCamera` z GUI krabičky „Kamera“.

2.7.1 Reakce na změny krabiček v jádře

Nevýhoda přístupu popsaného výše je fakt, že *GUI* krabičky `WorkspaceModel` a `WorkspaceCamera` jsou určeny pouze k vykreslování a nikoli k aktualizaci dat jejich reprezentace. Vykreslovací metody krabičky jsou například pouze volány, pokud je krabička zrovna vidět na obrazovce. Proto je v nové implementaci použit robustnější systém *callbacků*, které umožňují sledovat změny dat krabiček na úrovni jádra. Tento systém je navíc využit v implementaci zvýraznění vlivu transformací, která vyžaduje spouštět kód pouze v okamžiku, kdy nastane změna ve struktuře grafu scény (viz sekce 2.11).

Tento systém je založen na návrhovém vzoru pozorovatele (*observer pattern* [6, s. 104]). Třídě `Core::Node` byly přidány čtyři metody, které zařadí („zaregistrují“) předané zpětné volání do seznamu, který si instance třídy `Core::Node` přechovávají. Když poté uvnitř krabičky dojde k nějaké akci, všechny zpětné volání zaregistrované dané akci jsou zavolány. Zpětná volání lze registrovat pomocí následujících metod krabičky `Core::Node`:

- **`addUpdateCallback(callback)`**
  Tato metoda registruje zpětné volání, které je spuštěno při volání metody `Node::updateValues()`, která je volána jádrem kdykoli dojde ke změně dat krabičky.

- **`addDeleteCallback(callback)`**
  Tato metoda registruje zpětné volání, které je spuštěno v destruktoru krabičky. Tedy informuje o smazání krabičky.

- **`addPlugCallback(callback)`**
  Tato metoda registruje zpětné volání, které je spuštěno, když je jakýkoli výstup dané krabičky zapojen do vstupu jiné krabičky. Zpětná volání jsou spuštěna v metodě `Core::Node::plug()`.

- **`addUnplugCallback(callback)`**
  Tato metoda registruje zpětné volání, které je spuštěno když je jakýkoli vstup dané krabičky odpojen do výstupu jiné. Zpětná volání jsou spuštěna metodami `Core::Node::unplugOutput()` a `Core::Node::unplugInput()`.

Každá z těchto metod přijímá argument typu `std::function`, který lze metodě předat jako C++ lambda výraz. Při spuštění zpětného volání je zavolán kód uvnitř této předané funkce.

Používat tento systém je velice jednoduché, kódy 2.10 a 2.11 jsou v implementaci pouze zabaleny do lambda výrazu a v konstruktoru *GUI* krabiček zaregistrovány jako zpětná volání funkcí `Core::Node::addUpdateCallback()`.

2.8 Nekonečná mřížka

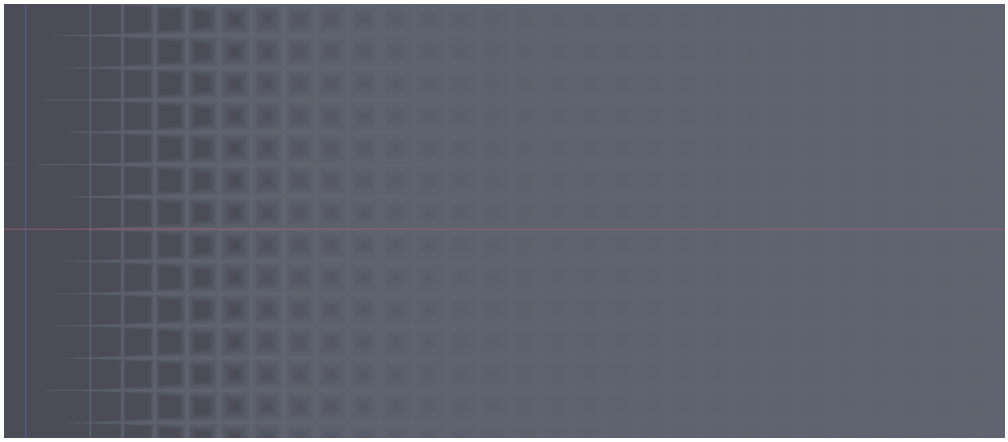
V této sekci je implementována nekonečná mřížka z požadavku ***F17***.

Mřížka je implementována pomocí shaderu `GridShader` a do hlavní scény je přidána jako entita s geometrii jedné plošky, která překrývá celou obrazovku. Mřížka je každý snímek vykreslena přes celou obrazovku a s pomocí průhlednosti se zobrazuje jen tam, kde se má ve světě nacházet. Uvnitř `GridShaderu` pro svůj každý fragment vypočítává správnou hloubku ve světových souřadnicích, a tedy ji mohou ostatní objekty ve scéně bez problému překrývat.

Původní implementace z blogu „A Slice Of Rendering“ [9] byla rozšířena o oddělené vykreslování hlavních os, aby nedocházelo k obarvení příčných čar a aby bylo možné nastavovat tloušťku čar hlavních os odděleně od tloušťky mřížky. Mřížce byla také přidána možnost vykreslovat i vertikální hlavní osu y vykreslením druhé mřížky v rovině xy, která vykresluje pouze hlavní osu y bez mřížky samotné.

Mřížku je možné spatřit v pozadí většiny obrázků v této práci například na obrázku 2.13.

Implementace využívá shaderové derivace k docílení antialiasingu a k potlačení artefaktů u horizontu. Díky nim mají také linky mřížky vždy zdánlivě konstantní velikost. Mřížka je shaderovými derivacemi v daném fragmentu rozmazána dle míry změny světových souřadnic mezi jednotlivými fragmenty na obrazovce. Na obrázku 2.17 je postupné rozmazání mřížky vizualizováno napojením souřadnic světové osy x namísto hodnot shaderových derivací. To znázorňuje, jak se s rostoucími hodnotami derivací mřížka rozmazává.



Obrázek 2.17 Rozmazání mřížky pomocí shaderových derivací vizualizované zapojením souřadnic osy x namísto hodnot derivací.

2.9 Zvýraznění modelů

Různé techniky vykreslování obrysů modelů shrnuje Alexander Ameye na jeho blogu [22]. Jedna z technik, kterou popisuje, spočívá ve vykreslování jednobarevných siluet 3D modelů. Tyto siluety jsou následně dvěma průchody (vertikálním a horizontálním) rychlého *box blur* shaderu rozmazány. Rozmazaná silueta je o něco větší než původní objekt a po rozmazání je možné docílit poměrně ostrého okraje *GLSL* funkcí `smoothstep`. Z rozmazané a zaostřené siluety lze pomocí *stencil* testu vystřihnout oblast, kterou pokrývá původní nerozmazaná silueta. Výsledný obraz je obrys objektu, jehož tloušťku určuje velikost *kernelu* použitého rozmazávacího shaderu. Rozmazávání může probíhat ve framebufferu s nižším rozlišením, než ve kterém je původní obraz vykreslován, čímž lze rozmazávání urychlit.

Na obrázcích 2.18 je celý tento proces znázorněn.

(a) Bez obrysu **(b)** Silueta modelu **(c)** Rozmazaná a zaostřená silueta **(d)** Vyříznutí původní siluety **(e)** Výsledný obrys po kompozici

Obrázek 2.18 Proces postupné tvorby obrazu zvýraznění 3D modelu.

2.10 Selekce objektů

Na straně 2D workspace byly třídám *GUI* krabiček `WorkspaceModel` a `WorkspaceCamera` přidány metody `processSelect()` a `processUnselect()`, které knihovna *DIWNE*, zajišťující UI 2D workspace, volá při změně selekce. V těchto metodách je reprezentacím

krabiček entitám `Vp::SceneModel` a `Vp::SceneCamera` nastavena proměnná `m_highlight` dle selekce na hodnotu `true` nebo `false`. Dle hodnoty této proměnné jsou při vykreslování entity případně zvýrazněny.

Na straně 3D viewportu se při vykreslování objektů do *stencil* bufferu zapisují jejich číselné identifikátory. Při stisknutí levého tlačítka myši je *stencil* buffer na pozici kurzoru myši přečten, a tím se identifikuje, který objekt byl vybrán, o čemž je posléze informováno 2D workspace, které selekci provede.

2.11 Zobrazení vlivu transformací

Vliv transformací je vyhodnocován třídou `ViewportHighlightResolver`. Ta používá zpětné volání krabiček jádra (popsané v sekci 2.7), aby detekovala změny struktury grafu scény. Když ke změně dojde, je spuštěna její metoda `resolve()`, která z každé krabičky „Model“ v grafu scény spustí prohledávání do šířky (algoritmus *BFS*). V momentě, kdy toto prohledávání narazí na jakoukoli právě vybranou krabičku, je z povahy *BFS* jasné, že je tato krabička napojena na daný model, ze kterého vyhledání započalo. V tom případě je prohledávání ukončeno a danému modelu je nastaveno modré zvýraznění, které uživatele informuje, že daný model je možné ovlivnit jednou z právě vybraných krabiček.

2.12 Manipulátory

V této sekci jsou implementovány manipulátory z požadavku ***F13***, které byly navrženy v předchozí práci Daniela Gruncla. Jedná se o manipulátory transformace translace, škálování, rotace, lookAt a projekčních matic. Všechny tyto manipulátory jsou popsány v sekci „2.1.5 Souhrn“ v bakalářské práci Daniela Gruncla [3, s. 11].

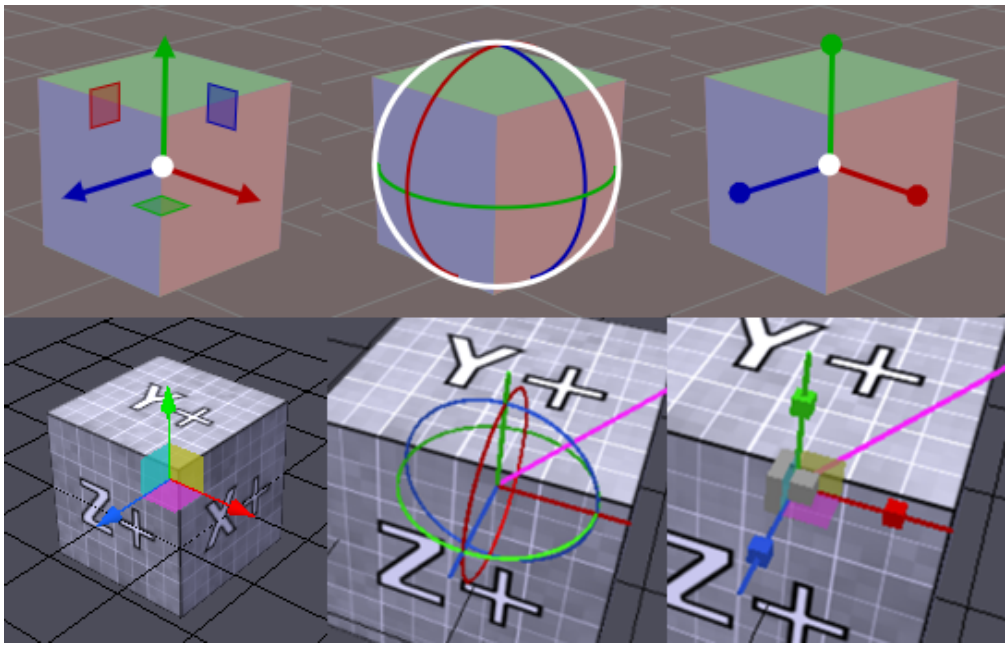
V jeho práci je ještě uveden manipulátor volné transformace, který v této práci není implementován. To bylo rozhodnuto po domluvě s vedoucím práce, protože se tato transformace momentálně nevyskytuje v žádném z tutoriálů a dle výsledků uživatelského testování, které Gruncl provedl, s ním měli uživatelé problémy [3, s. 35].

2.12.1 Knihovna ImGuizmo

Daniel Gruncl ve své práci manipulátory úspěšně implementoval do svého řešení prohlížeče scény. Ten je však v této práci zcela přepracován, a tudíž není možné jeho implementaci manipulátorů přímo využít bez provedení zásadních změn, jelikož je s minulým řešením prohlížeče scény úzce spjata. Manipulátory by tedy bylo nutné převést na nový systém entit, vykreslování a uživatelského vstupu.

V rámci rešerše implementace manipulátorů byla objevena open-source knihovna ImGuizmo [23], která implementuje manipulátory velice podobné těm, co Daniel Gruncl navrhl (porovnání základních manipulátorů je na obrázku 2.19). Tato knihovna je nadstavbou knihovny Dear ImGui a velice univerzálním a modulárním systémem umožňuje do jakékoli stávající aplikace využívající Dear ImGui přidat manipulátory translace, škálování a rotace. Implementace této knihovny je podobná implementaci UI prvků knihovny Dear ImGui. Jednotlivé manipulátory vykresluje pomocí stejné *immediate* metodiky knihovny Dear ImGui, k uživatelskemu vstupu využívá její funkce a k vykreslování používá Dear ImGui `DrawListy`. Knihovna tedy dokáže fungovat zcela odděleně od existujícího prohlížeče

scény a jeho vykreslování jako překrytí (*overlay*). Toto překrytí však do 3D scény prohlížeče scény dokonale zapadá, jelikož jsou knihovně ImGuizmo předány stejné pohledové a projekční matice.



Obrázek 2.19 Porovnání základních manipulátorů translace, škálování a rotace knihovny ImGuizmo [23] s manipulátory z práce Daniela Gruncla [3].

Knihovna ImGuizmo je velice populární a její manipulátory obsahují některé užitečné funkcionality, které původní manipulátory Daniela Gruncla neobsahovaly. Jako indikace záporných os a zobrazení výseče při rotaci. K detekci stisku myši uživatele také využívá matematickou detekci kolize namísto testování individuálních pixelů stencil bufferem, což může být v některých případech flexibilnější, jelikož nezáleží na vzdálenosti kurzoru od vykreslených objektů.

Dalo by se také říct, že jelikož se jedná o pouhé 2D překrytí, vykreslené manipulátory více připomínají prvky uživatelského rozhraní než 3D scény. Toto je však zároveň největší nevýhoda této knihovny, protože dokáže vykreslovat jen základní tvary. Knihovna využívá základní vykreslovací funkce Dear ImGui `DrawList`ů, nikoli standardní OpenGL vykreslovací pipeline. To znamená, že nevyužívá k vykreslování hloubkový buffer, shadery a ani integrovaný OpenGL ořez vrcholů (*clipping*) nebo perspektivní dělení (*perspective divide)*. Ořez vrcholů a perspektivní transformaci provádí velice zjednodušeně přímo ve své implementaci. Není s ní tedy možné vykreslovat složitější modely, ale převážně jen jednobarevné základní tvary a čáry. A samozřejmě zvládne pomocí Dear ImGui vykreslovat text.

Pro manipulátory tento způsob vykreslování stačí. Navíc nic nebrání tomu, aby se společně s manipulátory knihovny ImGuizmo ve 3D scéně zároveň vykresloval další obsah, jako například složitější 3D modely, a dosáhnout tak „hybridního“ přístupu, který by využíval silné stránky obou přístupů.

Nakonec bylo rozhodnuto manipulátory pomocí knihovny ImGuizmo implementovat s tím, že bude její kód rozšířen o podporu projekčních manipulátorů, které budou využívat

již existující kód translačního manipulátoru. V následujících sekcích bylo této implementace úspěšně docíleno a knihovna ImGuizmo byla využita i pro implementaci indikátoru os (***F18***).

2.12.2 Třída Manipulators

Manipulátory jsou implementované v třídě `Manipulators`, která se nachází v balíčku *Viewport*. Následující text doplňuje diagram tříd na obrázku 2.20.

Instanci této třídy si drží instance rozhraní `Viewport` a lze jí získat voláním `Viewport::getManipulators()`. Udržuje seznam právě aktivních manipulátorů `m_activeManipulators`, které reprezentuje interní třída `Manipulator`. Ta drží informace jednoho manipulátoru ve scéně, který manipuluje s jednou krabičkou transformace z jádra typu `Core::Transformation`.

Transformace mají různé typy, které určuje řetězec uvnitř krabičky transformace, který lze získat voláním `node->getOperation()->keyWord`.

Krabičku transformace dostává v konstruktoru a jedná se o krabičku transformace z jádra typu `Core::Transformation`. Transformace mají různé typy, které určuje řetězec uvnitř krabičky transformace, který lze získat voláním `node->getOperation()->keyWord`.

Instanci této třídy si drží instance rozhraní `Viewport` a poskytuje následující rozhraní:

Třída `Manipulators` poskytuje následující veřejné metody:

- addManipulator(node) a clearManipulators()

  Tyto metody jsou volány metodou `WorkspaceDiwne::manipulatorStartCheck3D()`, kterou volá kód 2D workspace na konci svého vykreslování.

  Metodou `manipulatorStartCheck3D()` workspace reaguje na jakékoli změny selekce krabiček. V případě, že změna nastane, je zavolána metoda `clearManipulators()`, která vymaže seznam právě aktivních manipulátorů. Následně je pro každou vybranou transformaci zavolána metoda `addManipulator(node)` s touto transformací jako argument.

  Uvnitř metody `addManipulator()` dojde k vyhodnocení, zdali je možné pro tuto transformaci zobrazit manipulátor. Toto provádí soukromá metoda `determineManipulatorType()`. Typ každé transformace `Core::Transformation` je určen řetězcem `keyWord`, který lze získat voláním `transformation->getOperation()->keyWord`. Tento řetězec metoda `determineManipulatorType()` vyhledá v hašovací tabulce `m_operationMap`, která k němu přiřadí jeden z typů `ManipulatorType`. Hašovací tabulka je předvyplněna řetězci transformací, které třída `Manipulators` podporuje. Pokud řetězec v hašovací tabulce není nebo je, ale není podporován, je manipulátoru nastaven typ `UNKNOWN` nebo `UNIMPLEMENTED`.

- drawManipulators(windowPos, windowSize)

  V této metodě probíhá vykreslení manipulátorů a zároveň případná úprava jejich matic. Je volána při vykreslování hlavního okna viewportu ve třídě `ViewportWindow`. Je nutné jí předat pozici okna na obrazovce a také jeho rozměry. Rozměry okna musí být přesně stejné jako ty předané vykreslovací metodě viewportu, aby překrytí manipulátorů vykreslené pomocí jeho pohledové a projekční matice bylo přesně zarovnané s jeho 3D scénou.

Pro každý aktivní manipulátor tato metoda nejprve získá „lokální“ a „referenční“ matice manipulátoru. Lokální matice je matice transformace, kterou manipulátor upravuje (`m_editedMatrix`). Referenční matice (`m_referenceSpace`) je výsledek násobení všech transformací, které se nacházejí v grafu scény **před** transformací, kterou upravujeme. Tyto matice z transformace a grafu scény nastavuje metoda `updateManipulatorMatrices()`, která k průchodu grafu scény používá již existující iterátor z jádra `Core::SequenceTree`. V případě, že se jedná o projekční transformaci, je ještě do proměnné manipulátoru `m_auxillaryMatrix` uložena pohledová matice, aby mohla být použita ke správnému umístění pohledového objemu ve světě scény.

Z rozhraní `Viewport` je také získána pohledová a projekční matice kamery hlavní scény.

Tyto matice jsou poté předány vykreslovacím metodám manipulátorů, které jsou popsány v následujících sekcích. Při vykreslení stejně jako v Dear ImGui dochází ke zpracování uživatelského vstupu, a tedy po vykreslení je navrácena již upravená matice.

Upravená matice je poté předána krabičce jádra, která ji zpracuje a aktualizuje všechny svoje potřebné hodnoty. V některých případech (projekční manipulátory) není krabičce předána celá matice, ale jsou upraveny některé její „výchozí“ hodnoty (viz 2.12.4).

- drawViewAxes(windowPos, windowSize)

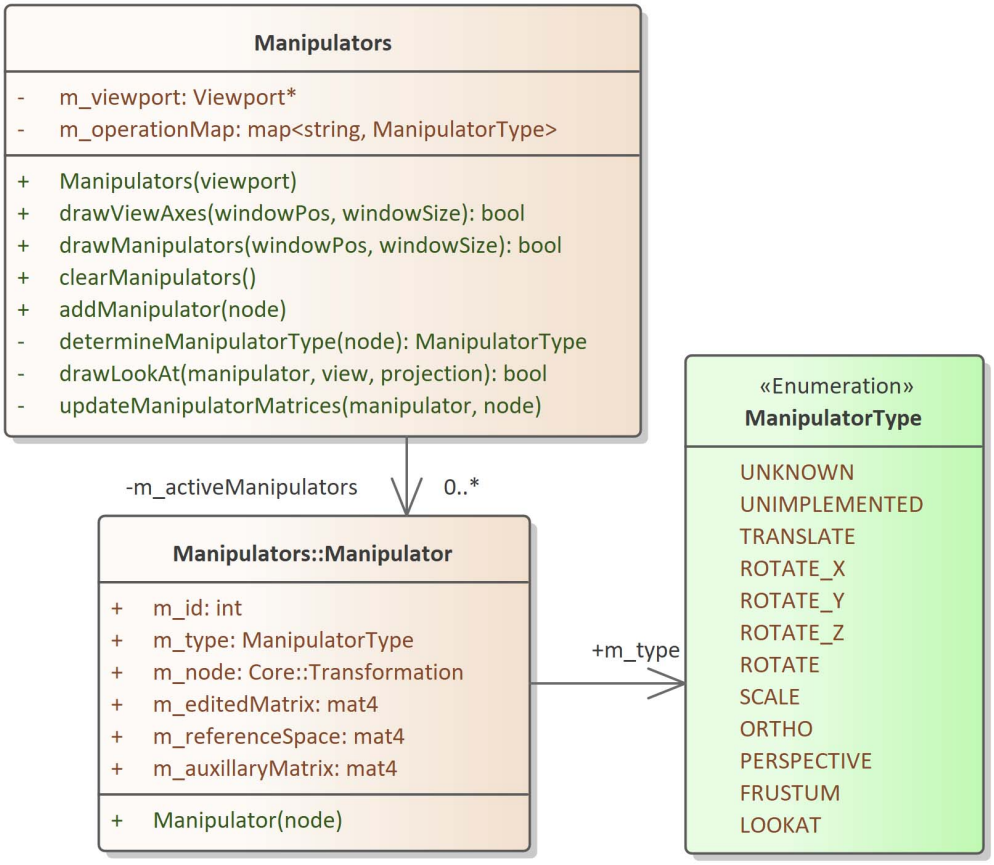
  Tato metoda je volána společně s předchozí metodou `drawManipulators()` v hlavním okně viewportu (`ViewportWindow`) a vykresluje indikátor os v pravém horním rohu okna. Dále je popsaná v sekci 2.12.6.

2.12.3 Manipulátory translace, škálování a rotace

Všechny tyto manipulátory knihovna ImGuizmo obsahuje a vykreslují se v metodě `Manipulators::drawManipulators()` pomocí její metody `ImGuizmo::Manipulate()`. Ta přijímá parametry `view`, `projection`, `operation`, `mode`, `matrix` a `deltaMatrix`. Parametry `view` a `projection` jsou parametry pohledové a projekční matice. Ty jsou získány z kamery hlavní scény. Parametr `operation` určuje o jaký typ manipulátoru se má jednat. V tabulce 2.1 je k jednotlivým transformacím přiřazena hodnota parametru `operation`, který je metodě `ImGuizmo::Manipulate()` předán.

Tabulka 2.1 Přiřazení typu ImGuizmo manipulátoru k transformacím translace, škálování a rotace.

Transformace	ImGuizmo::OPERATION
Translace	TRANSLATE
Škálování	SCALE
Rotace po ose X	ROTATE_X
Rotace po ose Y	ROTATE_Y
Rotace po ose Z	ROTATE_Z
Rotace kolem libovolné osy	ROTATE
Rotace kvaternionem	ROTATE



Obrázek 2.20 Diagram třídy `Manipulators`.

Parametr `mode` je vždy nastaven na hodnotu `ImGuizmo::LOCAL`, která určuje, že má metoda pracovat s lokálním prostorem jí upravované matice.

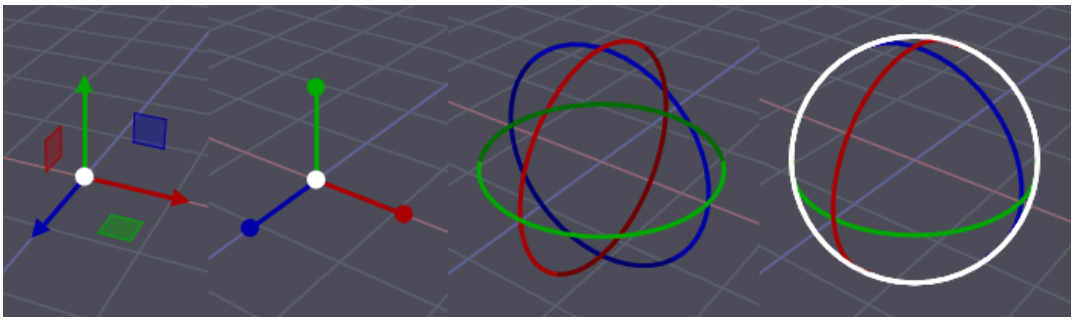
Parametr `matrix` je matice, kterou manipulátor upravuje. Tato matice také určuje pozici, na které se manipulátor zobrazí. Protože je nutné, aby byl manipulátor vykreslen na pozici ve světovém prostoru, je parametru `matrix` předána matice `combinedMatrix` (kombinovaná matice), která vznikne vynásobením referenční matice `m_referenceSpace` a lokální matice `m_editedMatrix`.

Poslednímu parametru `deltaMatrix` je předána prázdná matice, do které ImGuizmo uloží *deltu* (poslední změnu) matice `matrix`.

Z výsledné změny v parametru `deltaMatrix` je třeba po vykreslení získat změnu relativní k matici, která je upravována. ImGuizmo totiž vrací změnu matice předané parametru `matrix`, což je matice kombinace referenční a lokální matice. Manipulátor však potřebuje upravit pouze lokální matici manipulované transformace. Změnu lze ze světového prostoru kombinované matice převést na změnu lokální matice, a to vynásobením matice světové změny zprava původní neupravenou kombinovanou maticí a zleva její inverzí.

Touto lokální změnou je poté zprava vynásobena matice upravované transformace a výsledná matice je předána krabičce transformace pomocí metody `Core::Transformation::setValue()`.

Na obrázcích 2.21 jsou tyto nové základní manipulátory vidět.



Obrázek 2.21 Nové základní manipulátory implementované knihovnou ImGuizmo. Zleva doprava: translace, škálování, rotace po jednotlivých osách a nakonec obecná rotace po libovolné ose nebo kvaternionem.

Perspektivně korektní manipulátor rotace

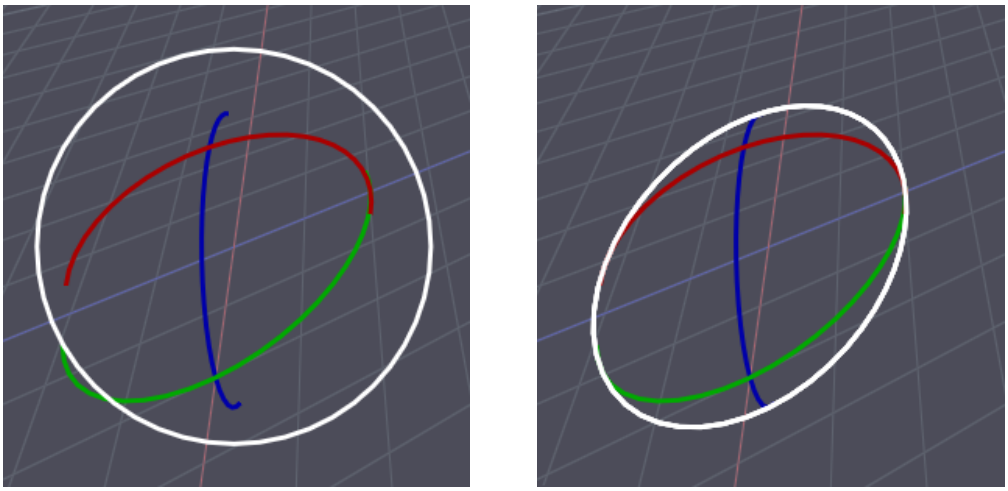
Složený manipulátor pro rotaci v knihovně ImGuizmo může volitelně obsahovat manipulátor rotace kolem normály obrazovky (dále jen „obrazovkový manipulátor“). Ten se vykresluje jako kružnice procházející póly pomyslné koule, na které všechny manipulátory rotace leží (taková kružnice se také nazývá „hlavní kružnice“). Tato kružnice je také natočena přesně směrem ke kameře. Tedy rovina, na které leží, je kolmá na vektor vedoucí k oku kamery. V současné implementaci se vždy vykresluje jako dokonalá kružnice v prostoru obrazovky. To není žádný problém, pokud se k vykreslování scény používá ortografická projekce. V případě projekce perspektivní se však na kružnici neaplikuje žádná perspektivní deformace na rozdíl od ostatních kružnic hlavních rotačních os. Ty jsou definovány ve světovém prostoru, následně transformovány pohledovou a projekční maticí a nakonec vykresleny na obrazovce jako pole úseček (*segmentů*).

To způsobuje, že v jistých situacích přestane být tento manipulátor na obrazovce správně reprezentován, protože se pomyslná koule, na které má ležet, začne perspektivou deformovat na elipsu, přičemž on je stále reprezentován kružnicí. To je obzvlášť zjevné, když se manipulátor nachází například ve velké blízkosti kamery nebo na krajích obrazovky. Tento problém se dále umocní, čím větší je zorné pole kamery.

Problém lze poměrně jednoduše vyřešit tím, že se bude kružnice tohoto manipulátoru definovat ve světovém prostoru a vykreslovat stejně jako ostatní kružnice manipulátorů hlavních os. V tomto případě se bude manipulátor správně deformovat, ale na rozdíl od jeho ortografické verze se ostatní rotační manipulátory nebudou vždy nacházet uvnitř jeho kružnice, což může být nežádoucí, pokud má uživatel tento obrazovkový manipulátor mimo jiné chápat jako reprezentaci pomyslné koule celého rotačního manipulátoru.

To je opět způsobeno perspektivou a nastává v případech, kdy je pomyslná koule manipulátoru blízko kameře a její polokoule přivrácená ke kameře začne obrazovkový manipulátor zakrývat. V tu chvíli se mohou některé části kružnic ostatních rotačních manipulátorů nacházet mimo kružnici manipulátoru obrazovkového, což pro uživatele, který předpokládal, že obrazovkový manipulátor je zároveň okraj pomyslné kružnice, může být velice matoucí.

Ke správnému vykreslení okrajů koule je třeba vykreslit její horizont, na který se pohled kamery dívá. Výpočet kružnice horizontu je znázorněn obrázkem 2.23 a v upravené



(a) (b)

Obrázek 2.22 Porovnání manipulátoru rotace před a po korekci perspektivy.

implementaci ImGuizmo jej provádí funkce `CalculateSphereHorizon()`. Ta z pozice kamery, pozice koule a jejího poloměru vypočítá, o kolik je nutné posunout hlavní kružnici natočenou ke kameře směrem k ní a o kolik je nutné zmenšit její poloměr, aby byla získána kružnice horizontu. Vykreslením horizontu koule, namísto její hlavní kružnice natočené ke kameře, je dosaženo perfektní reprezentace obrysu koule.

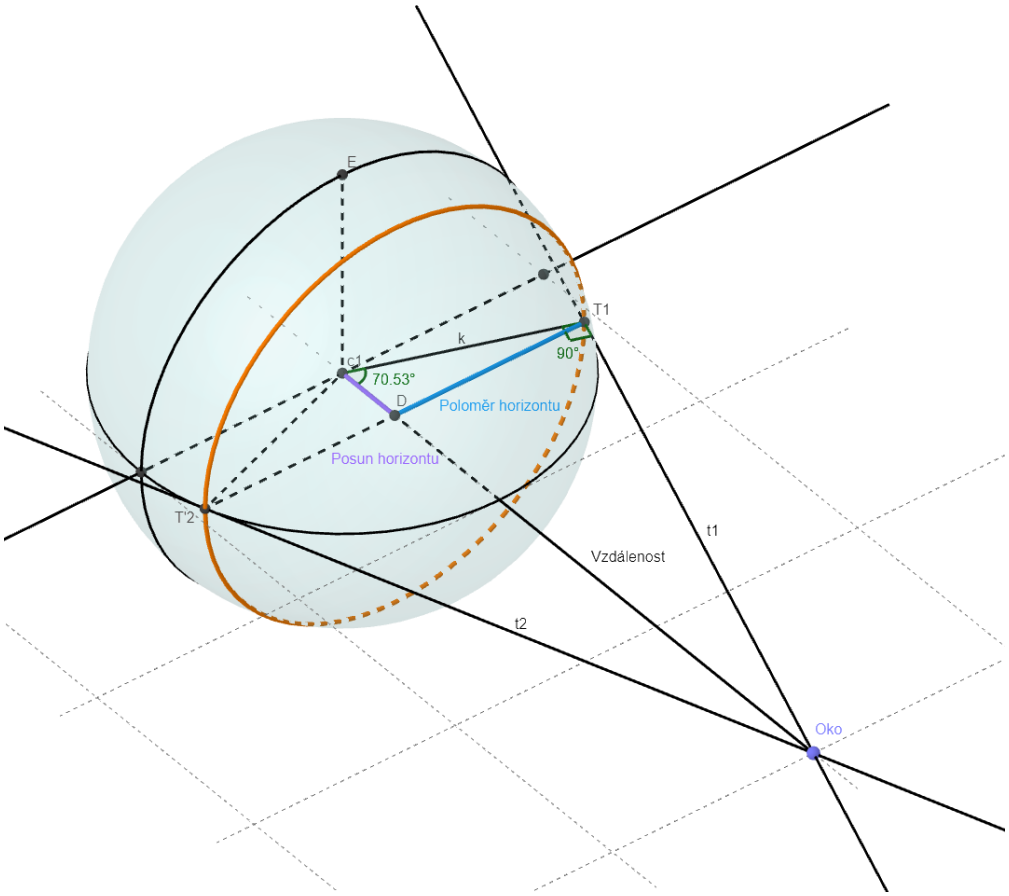
2.12.4 Manipulátory projekce

Manipulátory projekce knihovna ImGuizmo neobsahovala a byla v této práci o ně rozšířena. Vykreslují se metodou `ImGuizmo::ManipulateProjection()`. Tato metoda neupravuje přímo projekční matice, ale pouze parametry jejich fruster *left*, *right*, *bottom*, *top*, *near* a *far*. Manipulátory projekce jsou používány pro projekční transformace „ortho“, „frustum“ a „perspective“.

Metodě jsou opět předávány parametry aktuální pohledové a projekční matice kamery hlavní scény. Dalšími parametry jsou `editedProjection`, `editedView` a `params`. Parametr `editedProjection` je projekční matice, kterou je manipulováno. Tento parametr je pouze ke čtení a matici samotnou nijak neupravuje. Parametru `m_editedView` je předána matice manipulátoru `m_auxillaryMatrix`, kterou metoda `updateManipulatorMatrices()` ještě před vykreslováním nastaví na pohledovou matici kamery, ze které pochází upravovaná projekční transformace.

Posledním parametrem je `params`, což je pole `float`ů o šesti prvcích, každý z prvků odpovídá jedné z hodnot frusta v pořadí *left*, *right*, *bottom*, *top*, *near*, *far*. Toto pole je metodě předáno vynulované a implementace projekčního manipulátoru do jednotlivých hodnot nastaví, o kolik se daný parametr změnil. Jedná se tedy o podobný princip zaznamenání *delty* jako v případě parametru `deltaMatrix` metody `ImGuizmo::Manipulate`.

Metoda `ImGuizmo::ManipulateProjection()` přijímá jakoukoli projekční matici a používá se pro manipulaci projekčních transformací „frustum“, „ortho“ i „perspective“. Manipulátor se skládá z šesti barevných madel, které odpovídají každé z hodnot frusta. Implementace je velice podobná implementaci translačního manipulátoru, jelikož pro úpravu



Obrázek 2.23 Diagram výpočtu horizontu koule

frusta postačuje pouhé posunutí po jedné ose individuálních madel.

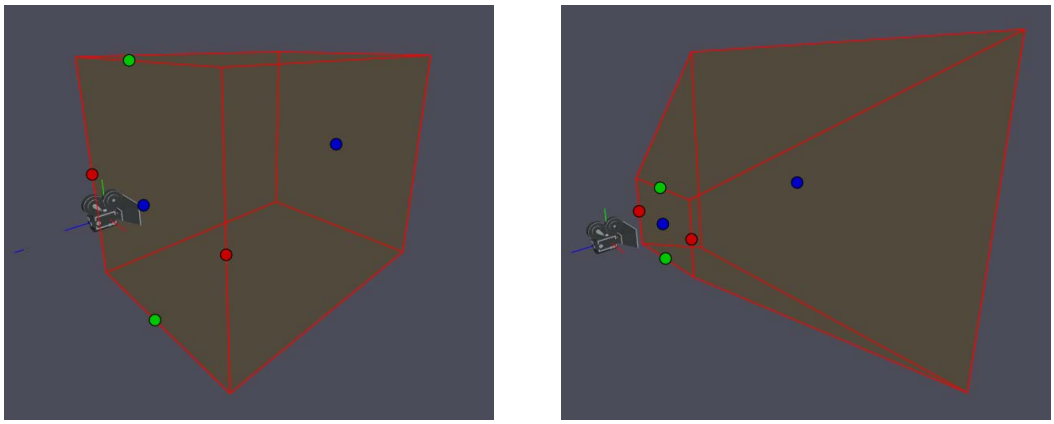
Pozice madel jsou v implementaci definovány v NDC prostoru, který je následně transformován inverzí projekční a pohledové matice, aby byly získány jednotlivé pozice madel ve světovém prostoru, ve kterém jsou následně posouvány. Pohyb madel je však přepočítán do prostoru pohledu, aby mohly být změny hodnot v poli `params` jednoduše použity k definici nové projekční matice.

Madla parametrů *left*, *right*, *bottom*, *top* a *near* jsou v NDC prostoru umístěny na blízké rovině pohledového objemu (souřadnici z rovné -1). Parametry *far* je umístěn na rovině vzdálené. Jinak jsou madla vždy umístěny uprostřed hran NDC kostky.

Podoba manipulátoru pro projekční matice je vidět na obrázku 2.24.

Manipulátor mimo madla také vykresluje drátěný model pohledového objemu, který splývá s pohledovým objemem krabičky „Kamera“, pokud je transformace do ní zapojena. Drátěný model je vykreslován včetně ořezu svých segmentů. Knihovna ImGuizmo však neumožňuje správně v prostoru vykreslovat trojúhelníky kvůli absenci složitějšího ořezávacího algoritmu. Pokud by bylo třeba vykreslovat poloprůhledné frustum pro projekční manipulátory i vně kamery, bylo by nutné toto vykreslování zajistit pomocí vykreslování viewportu. Současná implementace by však měla být zcela dostačující.

Po vykreslování jsou změny pozic jednotlivých parametrů frusta z pole `params` vyhodnoceny a krabičkám transformací jsou nastaveny nové hodnoty. V případě transformací



(a) ortho **(b)** perspective

Obrázek 2.24 Projekční manipulátory

„frustum“ a „ortho“ jsou pomocí hodnot `params` přímo upraveny ekvivalentní parametry krabiček transformací. Pro perspektivní transformaci jsou z hodnot `params` vypočítány hodnoty poměru stran a vertikálního zorného pole. A ty jsou společně s hodnotami *near* a *far* krabičce perspektivní transformace nastaveny.

K nastavení všech parametrů projekčních matic se používají metody transformačních krabiček `Core::Transformation::setDefaultValue()`, které přijímají takzvané „výchozí“ obecné parametry právě jako hodnoty *left*, *right*, *fovy*, *aspect* atd., ze kterých se vypočítávají výsledné matice v implementaci krabiček.

2.12.5 Manipulátor LookAt

V neposlední řadě byl implementován manipulátor kamerové transformace „LookAt“. Tuto transformaci definují dva body: bod pozice kamery a bod cíle pozorování. Dalším parametrem je *up* vektor, který však tento manipulátor neupravuje.

Implementace tohoto manipulátoru se nachází v metodě `Manipulators::drawLookAt()` a je velice jednoduchá. Používá dva manipulátory translace knihovny ImGuizmo, které vizuálně propojí čárou. Manipulátory translace fungují identicky jako ty, které byly popsány v sekci 2.12.3.

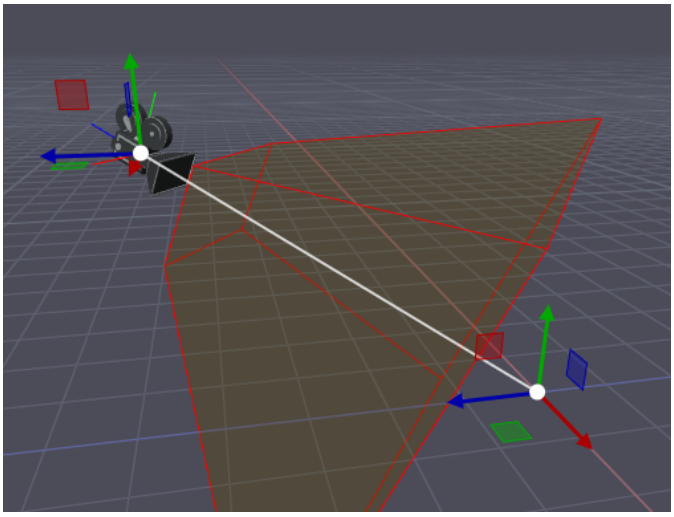
Krabičce transformace jsou upravené body pozice kamery a cíle pozorování nastaveny pomocí výchozích parametrů metodou `Core::Transformation::setDefaultValue()`.

Podoba manipulátoru je vidět na obrázku 2.25.

2.12.6 Indikátor světových os

Do knihovny ImGuizmo byla také přidána nová metoda `ImGuizmo::ViewAxes`, která vykresluje prvek indikátoru světových os (***F18***) a je volána z rozhraní manipulátorů pomocí metody `Manipulators::drawViewAxes()`.

Tato metoda je pouhým upravením již existující metody `ImGuizmo::ViewManipulate()`, která vykresluje prvek podobný indikátoru ViewCube z programu Autodesk Maya (který je vidět na obrázku 1.10 **(e)**).



Obrázek 2.25 Manipulátor „LookAt“

Metoda `ImGuizmo::ViewAxes` vykresluje v rohu okna malý pohled, ve kterém pomocí ortografické projekce vykreslí šest obarvených linek a koleček s popisky os, jejichž vykreslování je seřazené podle jejich hloubky. Jednotlivé body definující pozice konců linek a pozice koleček jsou vždy orientovány dle předané pohledové matice, která je získána z kamery hlavní scény. Tím prvek indikuje, kterým směrem jsou jednotlivé osy orientované v hlavním pohledu prohlížeče scény.

Indikátor světových os je vidět například v levém horním rohu na obrázku 2.13.

Kapitola 3
Uživatelské testování

Nové řešení prohlížeče scény bylo podrobeno uživatelskému testování použitelnosti. Testování se celkem zúčastnilo pět testerů, kteří o testování projevili zájem vyplněním úvodního dotazníků, který byl rozeslán na Discord serverech ČVUT FIT a FEL. Tři z testerů byli studenti FELu a jeden student FITu. Všichni studenti právě studovali předmět PGR a spadají do cílové skupiny „Studentů“ (viz 1.3.1). O testování také projevil zájem jeden z učitelů z FITu, kterého nejlépe reprezentuje cílová skupina „Programátor“.

Úvodní dotazník realizovaný přes službu Google Forms se testerů dotazoval na ročník jejich studia a zdali absolvovali relevantní předměty (zejména lineární algebru a PGR). Měli také ohodnotit jejich znalosti grafických transformací a vypsat grafické programy, se kterými mají zkušenosti. Všechny otázky dotazníku se nachází v příloze A.

Testování probíhalo distančně přes komunikační platformu Discord[1]. Testerům byl poslán *.zip* soubor obsahující spustitelný *.exe* soubor I3T a všechny soubory testovacích scén. Testeři sdíleli svoje obrazovky a s jejich svolením bylo na straně moderátora celé testování, včetně zvuku, nahráváno programem OBS[2].

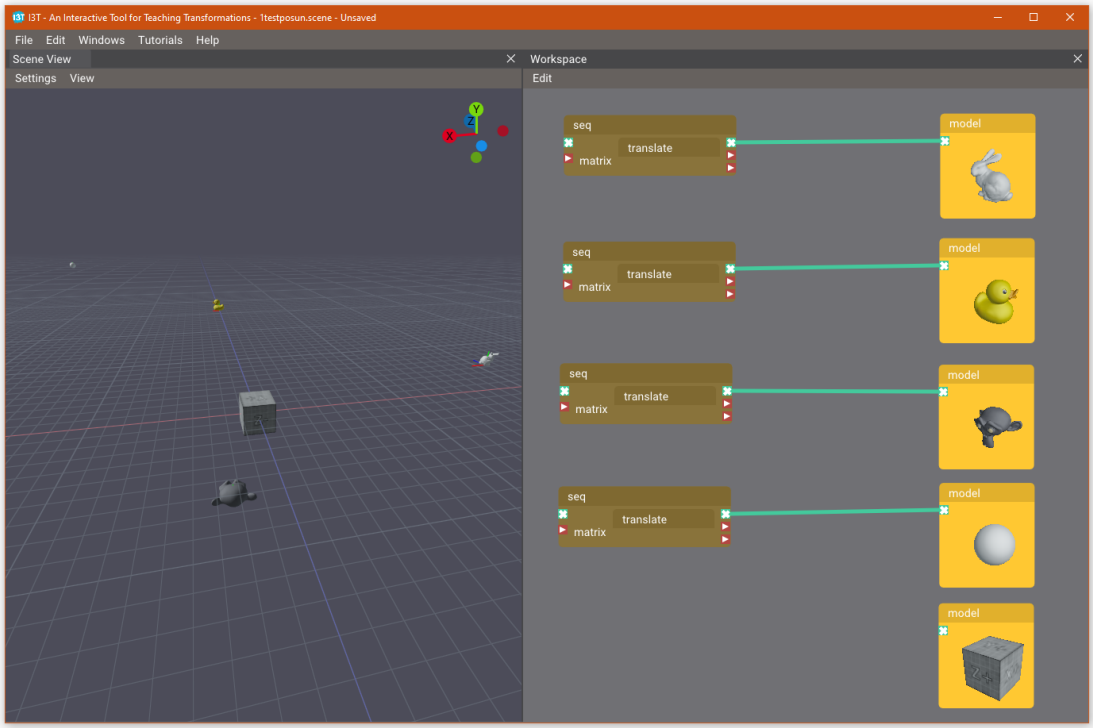
Testovací scénář se skládal z pěti testovacích scén. Každá ze scén se zaměřovala na testování specifických funkcí, ale navzájem se scény doplňovaly. Každá scéna měla nějakou úlohu nebo pár dílčích úloh (podúloh), které měl uživatel splnit. Úlohy byly uživateli zadávány verbálně z předpřipraveného testovacího scénáře. Průchod všemi pěti scénami trval v průměru něco přes jednu hodinu. Na konci testování byli uživatelé vyzváni k ohodnocení obtížnosti všech úloh na stupnici 0 (nejlehčí) až 10 (nejtěžší). Případně byly znovu probrány jakékoli potíže, se kterými se potýkali.

V následujících sekcích jsou jednotlivé scény popsány. V každé sekci je popsáno co bylo hlavní změření scény, jak se předpokládalo, že k ní testeři budou postupovat, a na jejich konci je popsáno, jak k úloze scény testeři ve skutečnosti přistupovali a jaké měli problémy nebo připomínky.

[1] https://discord.com/
[2] https://obsproject.com/

3.1 1. Úloha – Roztroušené modely

Průměrná obtížnost dle testerů: **2.2**
Scéna úlohy je vidět na obrázku 3.1.



Obrázek 3.1 První testovací scéna. Kamera je na obrázku pootočena, aby byly vidět všechny modely. Ve výchozím stavu je vidět pouze kostička a opička.

S touto úlohou testování začínalo. Ve scéně se nachází pět modelů na různých pozicích podél os x a z. Jeden objekt se nacházel lehce nad rovinou xz a jeden pod ní.

1. **První dojem** – Nejprve byl tester dotázán na první dojmy z aplikace, na co myslí, že se dívá, a jaká je souvislost mezi 3D pohledem na scénu a 2D workspace. Následně, pokud na to nepřišli uživatelé sami, jim byl zhruba nastíněn princip krabiček 2D workspace aplikace, jelikož ta není předmětem testování.

2. **Ovládání kamery** – Následně byli testeři pobídnuti, ať si 3D scénu prohlédnou a ať se zkusí zblízka podívat na všechny jednotlivé modely ve scéně. Toto mělo za cíl otestovat ovládání výchozí *orbit* kamery. Pro prohlédnutí objektů lze použít funkci vycentrování selekce umístěné v menu View 〉 Center camera on selection. Jeden z objektů byl umístěn daleko od středu scény, aby ho bylo těžší nalézt.

3. **Orientace ve scéně** – Dalším úkolem bylo prověřit orientaci uživatele ve scéně. Nejprve byl tester vyzván, ať ve scéně identifikuje střed světového prostoru bod $(0, 0, 0)$. Poté měl zhruba odhadnout souřadnice jednotlivých modelů ve scéně. V této scéně jsou záměrně schované čísla matic ve 2D workspace, nelze tedy souřadnice modelů přímo vyčíst z transformačních matic. Cílem této úlohy bylo zjistit, zdali dokáží uživatelé

pomocí nekonečné mřížky a indikátoru os rychle odhadnout pozice modelů. A nebo se spíš pokusí tyto informace hledat v maticích 2D workspace.

4. **Posun modelů** – Následující úlohou bylo jakýmkoli způsobem modely posunout na pozici světového středu. Tato úloha měla za cíl otestovat, zdali uživatelé najdou nově přidané manipulátory a jaká bude na ně jejich reakce. Souběžně s touto prací proběhlo testování tutoriálů jedním ze členů týmu I3T Adamem Louckým. V jeho testování bylo povšimnuto, že měli testeři problém s výběrem krabiček, protože je lze vybrat pouze kliknutím na jejich horní lištu a nestačí kliknout do „těla“ krabičky. V první testovací úloze jsou krabičky transformací přepnuté do módu „Label“, ve kterém se vlastně skládají pouze ze zmíněné horní lišty. Cílem bylo (mimo tedy skrytí souřadnic modelů pro 3. podúlohu) uživatelům napomoct manipulátory najít. Při testování byli testeři po 3. podúloze informováni o možnosti přepnout transformace pomocí kontextového menu do ostatních režimů „Set values“ a „Full“, ve kterých už je obsah transformace vidět.

Poznatky

V 1. podúloze všichni testeři pochopili, jak program zhruba funguje. Testeři vždy rychle pochopili, že vpravo ve 2D workspace vidí nějaké transformace napojené na 3D modely ve workspace, a ty stejné 3D modely jsou vidět ve 3D scéně.

Nalezení všech možných pohybů kamery (rotace, přiblížení, posun) většině testerů nedělalo problém. Jeden tester chvíli nemohl přijít na to, jak se kamera posouvá, ale po chvíli na stisk kolečka myši přišel.

Většina testerů však narazila na problém ve chvíli, kdy měli v rámci 2. podúlohy kameru posunout k jednomu z modelů. Testeři, byť zdánlivě pochopili ovládání kamery, se často na cílový model kamerou podívali a pak začali kameru přibližovat otáčením kolečka myši. To je nejdříve začalo skutečně posouvat směrem k cílovému objektu, ale za okamžik se dostali do stavu, kdy se kamera přiblížila bodu svého pozorování natolik, že se zcela přestala pohybovat směrem k modelu. Do této situace se dostali čtyři z pěti testerů, i když dva z nich uváděli, že mají zkušenosti z Blenderu, ve kterém se kamera chová naprosto stejně. Jediný tester, který s tímto neměl problém, již s Blenderem pracoval a evidentně byl na toto chování zvyklý.

Problém byl, že testeři nechápali, kolem čeho se kamera vlastně točí, a tudíž je překvapilo, když se kamera přestala při přibližování hýbat, protože dosáhla jejího *pivotu* (bodu otáčení).

První tester, který se do této situace dostal, měl zkušenosti s herním enginem Unity a uvítal by možnost volné kamery ovládané klávesami WSAD. Třetí tester by naopak uvítal možnost ovládání jako v Unreal Enginu, se kterým měl zkušenosti. V něm lze kameru posouvat rovnoběžně s vodorovnou rovinou xz pohybem myši s podrženým levým tlačítkem myši.

Během dalších úloh si však všichni testeři s tímto problémem na ovládání kamery zvykli a již si na něj nestěžovali. Nezdá se tedy, že by to byl zásadní problém, a možná se jedná jen o sílu zvyku. Každopádně by tomuto problému šlo předejít implementací ovládacích schémat kamer z různých 3D programů a nechat si uživatele vybrat, které preferuje.

Jedna možnost, která by mohla pomoct aktuálnímu ovládání kamery, je přidání malého

indikátoru pozice bodu otáčení. Na obrázku 3.2 je takový indikátor vidět v aplikaci Nira[3], která zobrazuje fotogrammetrické skeny. Tento indikátor je malá šedá tečka, která se pozvolna objeví, když se kamera začne posouvat nebo rotovat, a při zastavení pohybu opět zmizí. Tento indikátor by uživateli mohl naznačit, jakým způsobem kamera funguje, aby se vyvaroval problému, který měli testeři.

Obrázek 3.2 Indikátor bodu kolem kterého se kamera otáčí v aplikaci Nira.

Všichni testeři se pokusili najít možnost vycentrování kamery na model, ke kterému se chtěli přiblížit. Jeden tester ihned náhodou našel klávesovou zkratku 0, protože je velice podobná té v Blenderu (,). Další dva testeři tuto možnost poměrně rychle našli v menu viewportu View. Poslední dva testeři tuto možnost však najít nedokázali a menu lišty viewportu si vůbec nevšimli. Na objekt zkoušeli vycentrovat dvojitým kliknutím a poté tuto možnost hledali v hlavní menu liště programu. Jeden z nich ji také hledal přímo v kontextovém menu krabičky modelu.

Menu lišta viewportu by mohla být zvýrazněna použitím samostatných tlačítek, které se budou nad viewportem vznášet, podobně jak tomu je v programu OpenSimCreator[4], který také využívá Dear ImGui. Obrázek menu lišty viewportu tohoto programu je vidět na obrázku 3.3.

Obrázek 3.3 Menu lišta viewportu programu OpenSimCreator.

Testeři si rychle osvojili selekci objektů kliknutím levého tlačítka myši a tuto selekci hned využívali při vycentrování kamery na modely. V této i následujících úlohách také selekci mnohdy využívali k nalezení odpovídající krabičky ve 2D workspace z 3D viewportu a naopak.

S 3. podúlohou testeři neměli problém a poměrně jednoduše s pomocí mřížky souřadnice modelů odhadli. Střed světa také snadno našli a to nejen díky tomu, že se v něm nacházela výchozí kostička, ale také protože se tam kříží světové osy na mřížce. Souřadnice vzdálené koule odhadli pomocí velkých čtverců na mřížce a při identifikaci kladné či

[3] https://nira.app/
[4] https://github.com/ComputationalBiomechanicsLab/opensim-creator

záporné osy využili indikátor os v rohu, který všem přišel srozumitelný. Při určení souřadnic opičky, která se nachází pod mřížkou, jeden tester podotkl, že by se hodila možnost ortografické projekce, kterou by mohl využit společně s horním pohledem kamery k přesnějšímu určení souřadnic.

Na indikátor os také zkoušeli někteří testeři klikat, aby změnili pohled kamery. Indikátory os v ostatních programech tuto možnost obvykle obsahují a bylo by vhodné ji v budoucnu do indikátoru os v I3T přidat.

Knihovna ImGuizmo obsahuje metodu `ViewManipulate()`, která vykresluje prvek podobný indikátoru ViewCube z programu Autodesk Maya (který je vidět na obrázku 1.10 **(e)**) a tuto funkci obsahuje. Touto metodou byla současná implementace indikátoru os I3T inspirovaná a nemělo by tedy být obtížné tuto funkci do něj ještě navíc přidat.

Pro posunutí modelů do středu světa testeři nejčastěji použili přepojování sekvencí transformací ve 2D workspace. Modely lze do středu světa posunout jednoduchým odpojením do nich zapojeným translačních transformací. Dalším častým postupem bylo přepnutí translačních matic do módu, ve kterém je možné je přímo upravovat („Set values“ a „Full“) a zadání nulových souřadnic. Případně testeři posunuli čísla transformace myší, aby se model k $(0, 0, 0)$ přiblížil. Většina testerů objevila manipulátory až jako poslední způsob úpravy matice. Ti testeři, kteří manipulátory našli, neměli s posouváním modelů pomocí manipulátoru translace žádný problém. Dva testeři si v rámci první úlohy manipulátorů nevšimli vůbec. Nalezení manipulátorů by nejspíš pomohla změna způsobu selekce krabiček, jak bylo popsáno výše v popisu 4. podúlohy.

3.2 2. Úloha – Robot

Průměrná obtížnost dle testerů: **2.6**
Scéna úlohy je vidět na obrázku 3.4.

Druhá scéna obsahuje robotickou ruku, která je složena z mnoha kloubů. Tyto klouby jsou za sebou poskládány v grafu scény a předpřipravenými rotačními transformacemi lze robota ovládat.

Cílem testera je upravením pouze rotačních matic docílit, aby se „dráp“ (poslední kloub) robotické ruky nejprve dotknul kostičky ve scéně. Po dotknutí kostičky by se robotická ruka měla znovu upravit tak, aby se dotkla kachničky ve scéně.

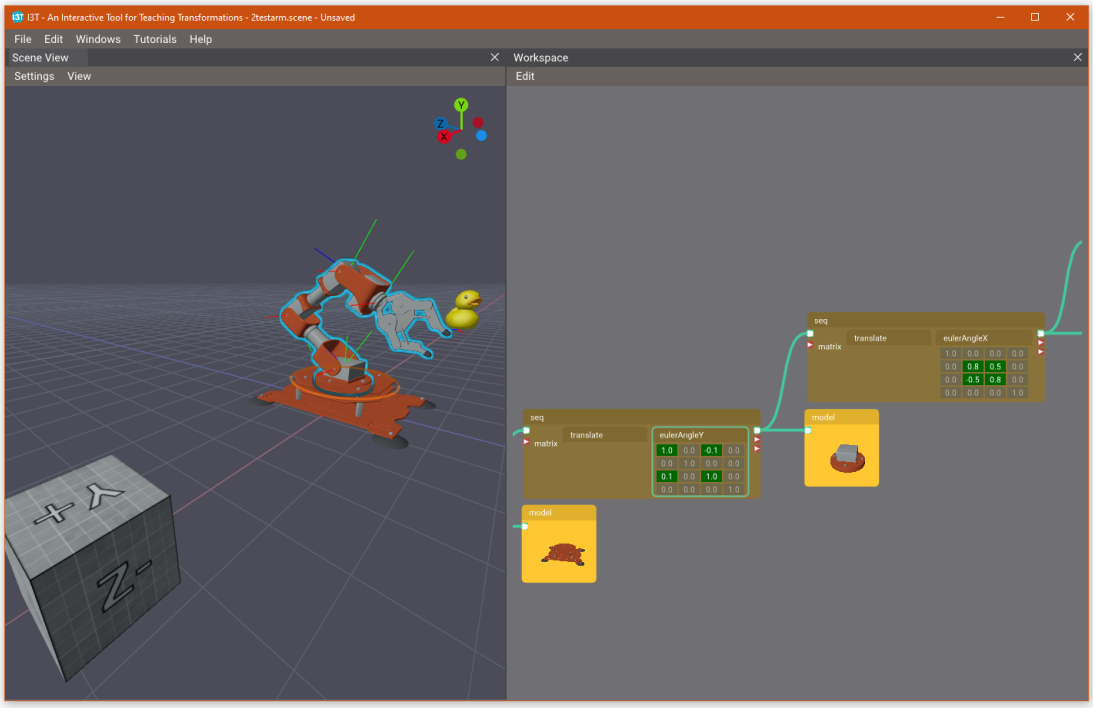
V této úloze se vyskytují manipulátory rotace kolem jedné osy, které má za cíl otestovat.

V této úloze je také hezky demonstrován systém zvýrazňování vlivů matice. Když se v grafu scény vybere rotační matice základny robota, celá konstrukce robota je modře zvýrazněna. Výběrem prvního kloubu jsou zvýrazněny všechny po něm následující, při výběru druhého klouby již není první kloub zvýrazněn, ale jsou zvýrazněny pouze klouby následující a tak dále. Vedlejším cílem je zjistit, zdali si tohoto faktu testeři všimnou.

Poznatky

Dva testeři, kteří neobjevili manipulátory v první úloze, začali robota poněkud krkolomně ovládat zadáváním jednotlivých čísel do rotačních matic. V této úloze si oba testeři manipulátorů nakonec všimli, načež je začali používat ke zbytku úlohy.

Žádný z testerů v této úloze nenarazil na problémy s ovládáním manipulátorů. V jistých případech testeři trochu zaváhali, když se jim nepodařilo manipulátor správně uchopit



Obrázek 3.4 Druhá testovací scéna s robotem.

nebo ovládat. Uchopit manipulátor rotace kolem jedné osy je možné pouze za světlý půlkruh, který je nejblíže kameře. Testeři se někdy pokusili manipulátor uchopit za jeho tmavý půlkruh, který je od kamery vzdálený. Typicky však hned následně chytli půlkruh světlý a žádný z testerů tento problém nezmínil. Podobně v jistých situacích testeři narazili na problém s ovládáním manipulátoru, který je natočen skoro rovnoběžně se směrem pohledu kamery. V tom případě však testeři pouze pootočili kameru a opět žádnému z nich to nestálo za zmínku. Tyto problémy byly před testováním známé, ale z testování se tedy nezdá, že jsou zásadní.

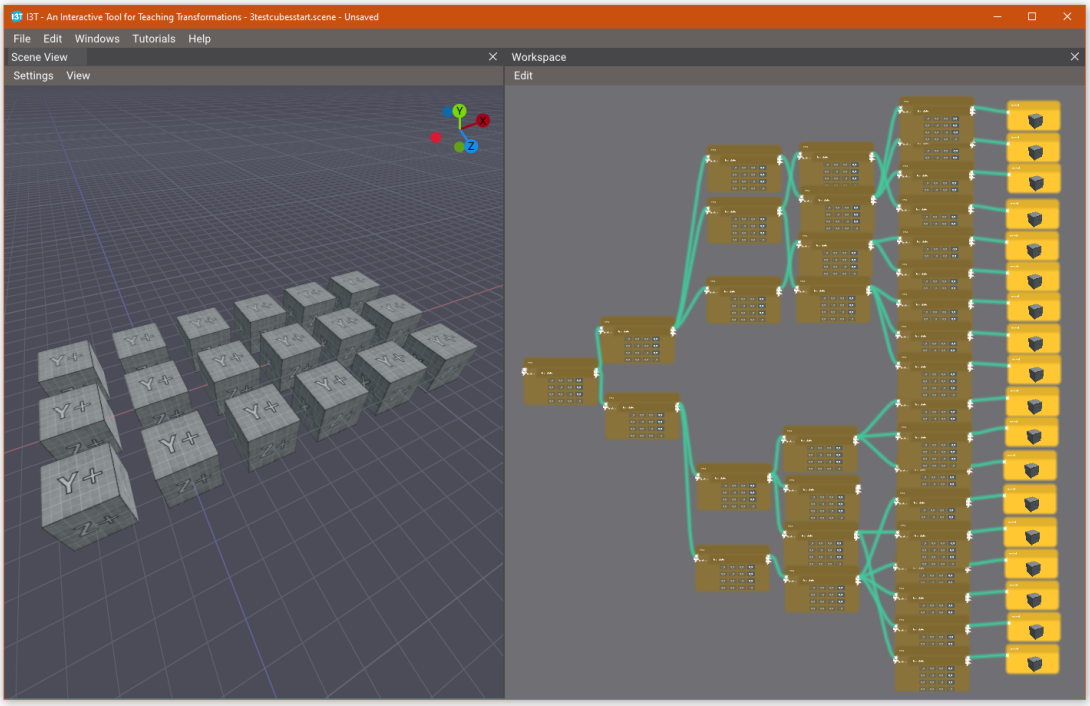
V této úloze nebylo třeba rotační transformace upravovat nijak přesně, proto zcela stačilo využití manipulátorů. Jeden tester, který nejprve rotace zadával ručně čísly, však poznamenal, že pomocí manipulátorů nelze dosáhnout přesného úhlu rotace. Uvítal by tedy možnost „krokování“, při kterém by se manipulátor otáčel v diskrétních krocích například 45°. Náhodou tuto funkci knihovna ImGuizmo podporuje a mělo by tedy být jednoduché ji v budoucnu implementovat.

3.3 3. Úloha – Kostky

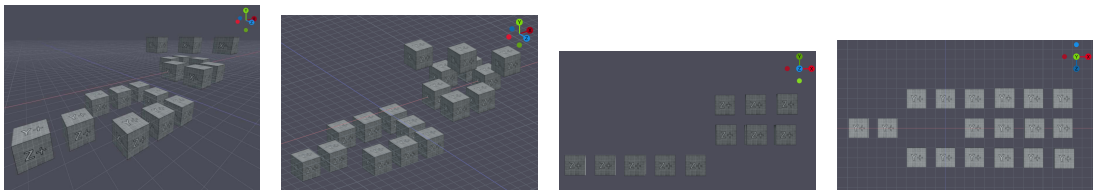
Průměrná obtížnost dle testerů: **4.6**
Scéna úlohy je vidět na obrázku 3.5.

V této úloze měli testeři za cíl posunout kostičky ve scéně tak, aby jejich rozestavení odpovídalo kostičkám na obrázku, který byl testerovi předložen. Výchozí postavení kostiček je vidět na obrázku 3.5 a cílové rozestavení je vidět na obrázcích 3.6.

Kostičky v této scéně jsou součástí spletitého grafu scény, který obsahuje mnoho



Obrázek 3.5 Třetí testovací scéna s kostkami.



(a) **(b)** **(c)** **(d)**

Obrázek 3.6 Cílové rozestavení kostiček třetí testovací úlohy.

translačních transformací, které ovlivňují mnoho kostek najednou.

Cílem této scény je vyzkoušet, zdali bude tester využívat selekci a vyobrazení vlivu transformací k usnadnění orientace ve velice složité scéně.

Některé skupiny kostek v grafu scény sdílí transformaci translace tak, aby stačilo k docílení jejich správně pozice poupravit jen jedninou transformaci. Tři kostky vpravo nahoře (na obrázcích **(a) a (b) 3.5**), které je nutné všechny posunout nahoru však schválně do takové skupiny nepatří, aby bylo nutné je posunout individuálně.

Poznatky

Dva testeři k řešení této úlohy využili přímé zadávání čísel do matic transformací a zvýraznění vlivu transformací nebo manipulátory vůbec nevyužili. Přesto využili selekci kostiček ve 3D scéně k nalezení správné krabičky „Model“ ve 2D workspace. Zbylí tři testeři manipulátory i zvýraznění používali a graf scény samotný díky nim příliš nezkoumali. Po

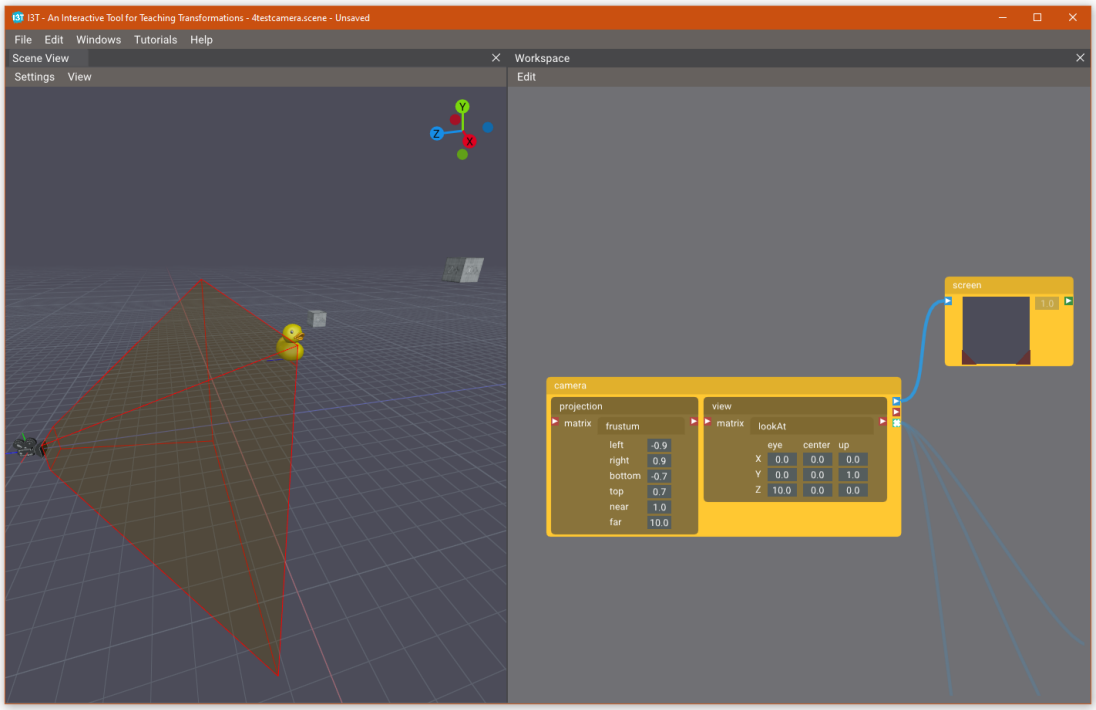
dotázání všichni testeři správně vysvětlili, co znamená modrý obrys zvýrazňování vlivu transformací.

Zajímavým poznatkem v této úloze bylo, že dva testeři v jednom momentě očekávali, že se při selekci více transformací najednou automaticky zobrazí manipulátor, který bude všechny upravovat najednou. V případě některých množin krabiček opravdu existovala transformace, které pohne všechny najednou, ale pokud jsou vybrány transformace individuálních krabiček, každá z nich posouvá pouze kostičku, ke které je připojena, a pro každou z těchto transformací je zobrazen individuální manipulátor. Toto může být důsledek přílišného spolehnutí na manipulátory. Po bližším prohlédnutí grafu scény však byli testeři vždy schopni vysvětlit, proč tomu tak není.

Jeden tester zmínil, že by uvítal možnost selekce více objektů zároveň ve 3D scéně tak, jak tomu je ve 2D workspace.

3.4 4. Úloha – Kamera

Průměrná obtížnost dle testerů: **6.0**
Scéna úlohy je vidět na obrázku 3.7.

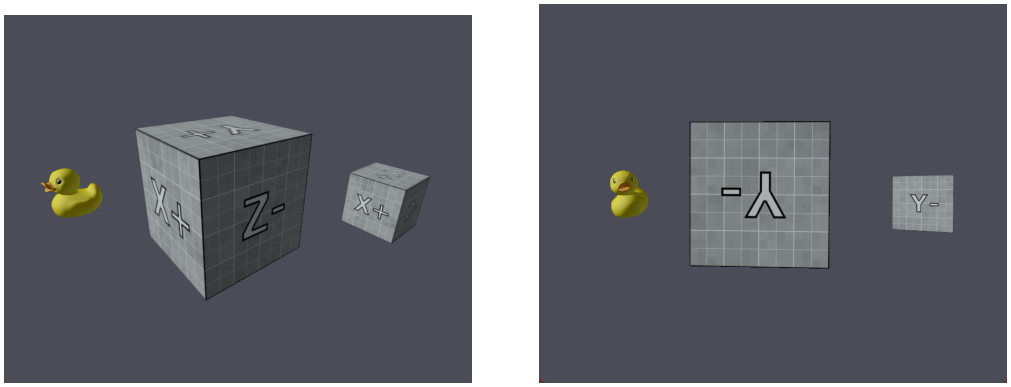


Obrázek 3.7 Čtvrtá testovací scéna s kamerou.

V této úloze bylo cílem upravit matice kamery tak, aby bylo dosaženo pohledu jako na obrázku 3.8 **(a)**. Následně měly být naopak upraveny pouze objekty ve scéně tak, aby bylo docíleno pohledu jako na obrázku 3.8 **(b)**.

Úloha je zaměřena na otestování matice „LookAt“. K dosažení potřebného pohledu je také nutné posunout vzdálenou rovinu projekční matice. K tomu mohl tester použít manipulátor projekce a nebo jen posunout hodnotu *far* přímo v krabičce projekční matice.

Po docílení prvního pohledu měl tester za cíl upravit modely ve scéně tak, aby se pohled podobal druhému pohledu na obrázku, ve kterém jsou modely natočené směrem ke kameře. K tomu měly modely ve scéně předpřipravené transformace rotace kolem libovolné osy, který tato úloha mimo jiné testovala.



(a) (b)

Obrázek 3.8 Cílové obrazy kamery ve 4. testovací úloze.

Poznatky

Testeři poznamenali, že u manipulátoru „LookAt“ je těžké poznat, který bod reprezentuje pozici kamery, a která bod pozorování. Obecně však s jeho používáním neměli problém.

Testeři byli dotázáni, jaký mají význam jednotlivá čísla v krabičce „LookAt“, která byla přepnuta do módu „Set values“ a zobrazovala souřadnice bodu pozice kamery a bodu cíle pozorování.

Někteří z testerů se domnívali, že souřadnice „eye“ reprezentovali vektor směru pohledu kamery. Jeden z testerů si také spletl významy bodů „eye“ a „center“. Po chvilce práce s manipulátorem si však testeři ověřili, že se opravdu jedná o pozice bodů kamery a cíle pozorování.

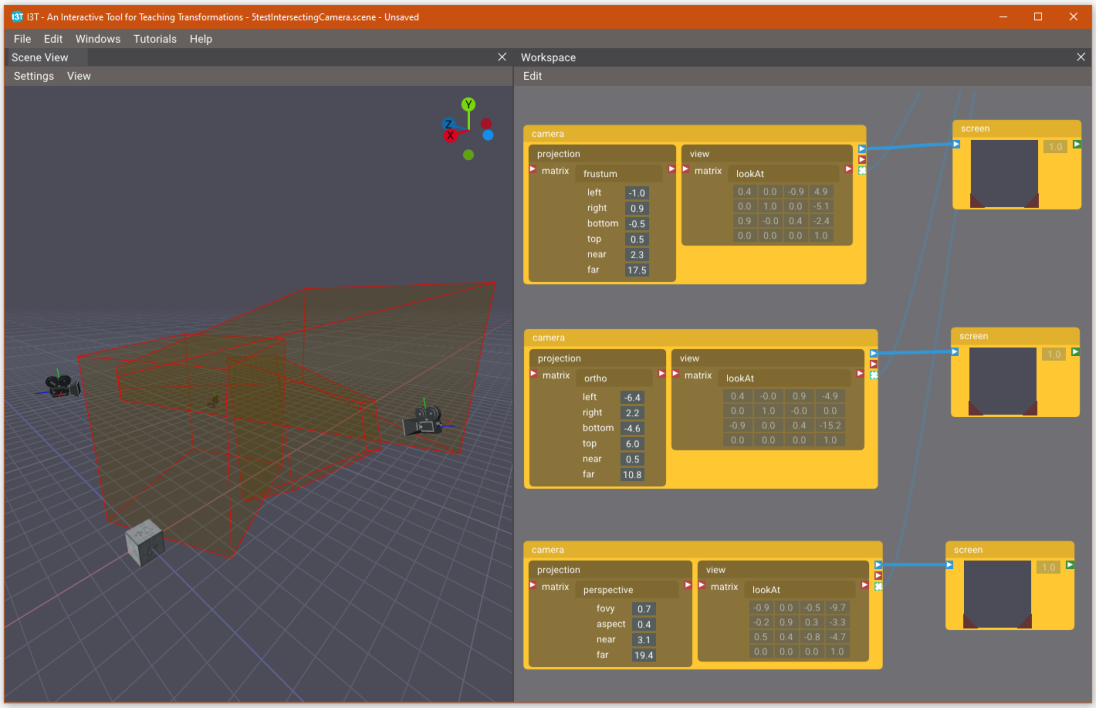
V druhé části úlohy jednoho testera matl manipulátor obrazovkové rotace a preferoval by, pokud by bylo možného ho skrýt. Další tester poznamenal, že by uvítal možnost vidět zvýraznění modelu i na obrazovce kamery.

Celkově však testeři neměli s úlohou větší potíže, byť jí v průměru hodnotili jako poměrně obtížnou.

Mnozí testerů měli velké potíže nalézt madlo pro zvětšení pohledu obrazovky „Screen“. Jeden tester poznamenal, že si trojúhelníků v rozích všiml, ale myslel, že se jedná o součást vykreslené scény. Testeři také zkoušeli tahat za roh krabičky nebo možnost hledali v jejím kontextovém menu. Jistě by bylo vhodné trojúhelníčky pro změnu dimenzí obrazovky nějak ve 2D workspace zvýraznit.

3.5 5. Úloha – Protínající se kamery

Průměrná obtížnost dle testerů: **4.6**
Scéna úlohy je vidět na obrázku 3.9.



Obrázek 3.9 Pátá testovací scéna s protínajícími se kamerami.

V této úloze bylo cílem posunout kostičku ve scéně tak, aby byla vidět na obrazovkách všech tří kamer současně, což mělo za cíl vyzkoušet zdali se tester dokáže orientovat v prostoru plném protínajících se fruster a zdali se mu podaří lehce nalézt průnik pohledových objemů kamer.

Poté byl tester vyzván, ať poupraví frusta kamer tak, aby model zcela vyplňoval jejich obrazy. Tento úkol navazuje na minulou testovací úlohu a má za cíl otestovat manipulátory projekce.

Poznatky

Testeři neměli zásadní problém kostičku do správného místa posunout. Byly vyzváni ať zkusí změnit barvy výplně pohledového objemu každé kamery a většina shledala, že byla scéna trochu přehlednější. Možná by měly nově vytvořené kamery vždy vybírat náhodnou barvu, aby výchozí barvy pohledových objemů nebyly podobné. Po změně barvy také jeden tester uvedl, že by uvítal kdyby byla barva pohledového objemu nějakým způsobem znázorněna i přímo v krabičce „Kamery“, aby mezi nimi bylo snadné rozlišovat.

Jeden tester poznamenal, že by madla manipulátoru perspektivní projekce měla být umístěna na vzdálené rovině pohledového objemu namísto té blízké, protože při nízké

hodnotě jejího parametru *near* se mohou nacházet moc blízko u sebe, jelikož se směrem k pozici kamery pohledový objem perspektivní transformace zužuje.

3.6 Shrnutí výsledků testování

Všichni testeři splnili testovací úlohy bez větších problémů a dojmy všech testerů byly vcelku pozitivní. Za zmínku jistě stojí problém s přibližováním kamery, na který testeři naráželi v první úloze. S ovládáním kamery se však vždy vypořádali, a když objevili možnost vycentrování kamery na selekci, tak jí hojně využívali, čímž se vyhnuli neustálému posouvání kamerou.

Funkci selekce modelů ve 3D scéně testeři hojně využívali k nalezení odpovídající krabičky ve 2D workspace. Stejně tak lehce porozuměli zobrazování vlivu transformací a využívali jej k usnadnění orientace v grafu scény.

Z testování je jasné, že manipulátory jsou obecně preferovaný způsob manipulace objekty ve scéně. Jejich použití však také může uživatele odradit od porozumění samotným maticím transformací, jelikož je tak snadné pomocí manipulátorů docílit zadané úlohy. Testovací úlohy byly vždy zadané s daným cílem a všichni testeři se shodli, že jsou manipulátory ve většině případů nejjednodušší způsob jak zadaných cílů dosáhnout. Pro cílové skupiny „Učitelů“ a „Programátorů“ jsou tedy manipulátory velký úspěch, jelikož usnadňují tvorbu scén.

Pro cílové skupiny „Studentů“ či nezkušených zájemců o grafiku je určit přínos manipulátorů trochu záludnější. Na jednu stranu umožňuje studentům lehce ovládat transformace jinak než čísly, ale na stranu druhou jim umožňují se snadně vyhnout nutnosti pečlivého zadávání čísel do matic transformací, a přesto docílit požadovaného výsledku. Stejným způsobem je možné pohlížet i na existující funkci „tahání“ čísel v matici. Pokud uživatel zatáhne za číslo v například rotační matici, dojde k plynulé rotaci pouhým tažením myši, čímž se student také vyhne přesnému zadávání čísel, a tedy případné nutnosti skutečného porozumění čísel rotační matice.

Nicméně všichni studenti, kteří se testováni zúčastnili, funkci manipulátorů chválili a preferovali by ji mít k dispozici, i kdyby měly používat I3T pouze k prohloubení svých znalostí.

Je ale zjevné, že v některých případech by bylo dobré zvážit, zdali by nemělo být použití manipulátorů (a případně i „tahání“ čísel matic) omezeno. Zejména v případech, kdy je po studenti požadováno docílit nějakého řešení v domácí úloze, nebo v úloze na cvičení. Tedy v případech, kdy možná uživatelé nechtějí přímo pochopit látce, ale spíš jen docílit úspěšného řešení.

Kapitola 4
Diskuze

4.1 Vykreslování přímo na obrazovku

Všechny pohledy viewportu jsou vykreslovány nejprve do samostatných textur (pomocí framebufferů). Tyto textury se poté přidávají metodou `AddImage()` do `DrawList`u aktuálního podokna. Volání `AddImage()` samotné však nic nevykresluje, pouze zařazuje vykreslovací povel do fronty `DrawList`u. Vykreslování probíhá až později na konci celého snímku voláním metod `ImGui::Render()` a příslušné vykreslovací metody použitého backendu (v tomto případě `ImGui_ImplOpenGL3_RenderDrawData()`). To znamená, že veškeré textury hlavního viewportu, pohledů kamer i náhledů modelů musí být všechny připraveny a uloženy do jednotlivých textur. Nelze například vykreslit náhled jednoho modelu do textury, tuto texturu vykreslit na obrazovku a hned jí znovu využít pro vykreslení dalšího modelu ve stejném snímku, čímž by se ušetřilo trochu paměti grafické karty. Takovou optimalizaci by bylo možné využít pouze v případě framebufferů se stejnými parametry, tedy právě u náhledů modelů, kterých může být mnoho.

Větším problémem je ale fakt, že je vůbec nutné pro vykreslovaní vždy používat textury a následně teprv přikázat ImGui, aby je vykreslilo. Toto vykreslení v rámci ImGui na konci snímku je totiž zcela zbytečné. Mnohem lepší by bylo vykreslovat přímo na obrazovku bez využití ImGui jako prostředníka. Toho je možné v ImGui docílit registrací vlastního vykreslovacího callbacku, který je stejně jako každý jiný vykreslovací příkaz zavolán na konci snímku. V něm lze provést své vlastní vykreslování na potřebném místě, a tím předejít oběma zmíněným nedostatkům.

Bohužel je tento přístup velice komplikovaný a nepovedlo se ho robustně implementovat. Uvnitř hlavního okna se podařilo takového vykreslování pomocí callbacku dosáhnout. Začnou se však objevovat problémy, když se podokno workspace nebo viewportu vytáhne mimo hlavní okno. V tu chvíli ImGui pro toto podokno vytvoří nové nativní okno operačního systému, aby ho mohlo dále vykreslovat. Toto nové okno však má svůj vlastní OpenGL kontext odlišný od kontextu hlavního okna. Změna kontextu znamená, že je potřeba některé OpenGL objekty znovu vytvořit, což spolehlivě provést není triviální, a při vývoji nového prohlížeče scény se vždy předpokládalo, že OpenGL kontext bude stejný. I pokud by problém se změnou kontextu byl vyřešen, tato implementace by nejspíš byla závislá na specifickém ImGui backendu, který se může v budoucnu změnit. Situaci nepomáhá velice malé množství dokumentace této funkce a potřeba detailního porozumění

implementaci backendu. Je tedy ke zvážení, zda stojí za to věnovat energii optimalizaci této části vykreslování. Snížení výkonu způsobené využíváním `DrawList`ů je totiž vyváženo zárukou, že bude vždy fungovat, jelikož jde o ustálenou funkci Dear ImGui.

4.2 Framebuffer

Objekt `Framebuffer`, používaný ke správě OpenGL framebufferů, momentálně nepodporuje možnost mezi jednotlivými framebuffery sdílet své barevné nebo hloubkové přílohy. Při vykreslování WBOIT průhlednosti a zvýraznění objektů dochází ke kopírování hloubkové přílohy mezi dvěma framebuffery. Přitom by nejspíš stačilo, aby byla hloubková příloha mezi framebuffery sdílena, což současná implementace neumožňuje.

Framebuffery se také chovají neoptimálně při změně rozměrů okna. Vždy snaží docílit rozlišení, kterému jim je metodou `start()` předáno. V případě, že ale uživatel aktivně mění rozměry okna, dochází každý snímek k destrukci a vytvoření nových textur nebo renderbufferů. Přitom není nutné, aby při změně velikosti okna měly framebuffery vždy přesně stejné rozměry jako okno. Tento problém by mělo být možné vyřešit přidáním možnosti změnu velikostí framebufferu na pár snímků či milisekund odložit. V momentě, kdy uživatel přestane oknu měnit rozměry, by framebuffer mohl být lehce rozmazaný, ale vzápětí by se opět obnovil na stejné rozměry jako okno.

4.3 Napojení manipulátorů na zpětná volání jádra

V implementaci vykreslování manipulátorů dochází každý snímek k výpočtu referenčního prostoru každého z aktivních manipulátoru. Tento výpočet rekurzivně prochází část grafu scény, a tudíž ho není vhodné spouštět každý snímek a to navíc zvlášť pro každý manipulátor, kterých může být aktivních mnoho najednou.

Manipulátory by mohly být napojeny na systém zpětných volání krabiček z jádra, aby k vypočtu referenčního prostoru docházelo pouze v případech, že se některá z matic předcházející manipulované transformaci změnila nebo byla pozměněna struktura grafu. Nejspíš by šlo i průchod grafem pro všechny aktivní manipulátory sjednotit, aby nebyly stejné části grafu procházeny vícekrát v jednom snímku. Příliš složité řešení by však mohlo vytvořit mnoho nepříjemných krajních případů.

Kapitola 5
Závěr

Cílem práce bylo vytvořit pro novou verzi aplikace I3T plně funkční prohlížeč scény, který obsahuje veškeré důležité funkce první verze aplikace I3T a který je do existující rozpracované aplikace robustně integrován. Tento cíl byl splněn a nové řešení prohlížeče scény obsahuje všechny nejdůležitější funkce. Těmi jsou hlavní okno s náhledem na 3D scénu používající nezávislou kameru, náhledy v krabičkách „Model“, vykreslování scény pro krabičky „Screen“ a reprezentaci krabiček „Model“ a „Kamera“ ve scéně.

Jedinou původní funkcí, která v novém řešení chybí, je načítání vlastních modelů, na kterém v rámci serializace scén pracuje jeden z ostatních členů týmu I3T a na které je současný systém správy zdrojů plně připraven.

Řešení bylo navrženo specificky pro potřeby existujících částí aplikace a klade důraz na jednoduchost. Vykreslování prohlížeče scény bylo integrováno do uživatelského rozhraní a okenního systému aplikace pomocí jednoduchého rozhraní vykreslovacích funkcí. Na logické jádro aplikace bylo nové řešení skrz třídy uživatelského rozhraní připojeno systémem zpětných volání. Požadované krabičky byly napojeny na nové řešení prohlížeče scény a byly jim implementovány všechny volby, které byly popsány ve funkčních požadavcích.

Nové řešení původní prohlížeč scény v mnoha směrech rozšířilo. Byla přidána širší podpora načítání formátů 3D modelů a podpora poloprůhledného vykreslování i složitějších 3D modelů. Nové řešení se inspirovalo 3D programy, jako například Blender a Unity a přidalo do 3D scény nekonečnou mřížku a indikátor os pro usnadnění orientace ve scéně.

Bylo navázáno na práci Daniela Gruncla a dle jeho návrhu byly pomocí knihovny ImGuizmo implementovány manipulátory, které umožňují intuitivně upravovat matice transformací přímo ve 3D scéně a vizualizovat, jak se tyto transformace za sebou skládají.

Práce také přidala zcela novou funkcionalitu výběru objektů ve 3D scéně a propojení 3D pohledu s 2D grafem krabiček rozšířilo o možnost výběrem transformace zvýraznit ve 3D scéně objekty, na které má daná transformace vliv.

Řešení prošlo uživatelským testováním použitelnosti a nebyly nalezeny žádné zásadní problémy. Bylo však popsáno mnoho možných budoucích vylepšení a objeveny některé menší nedostatky současné implementace.

Souběžně s touto prací pokračoval vývoj ostatních částí aplikace a během posledních měsíců se aplikace opravdu zásadně posunula kupředu. Je plánováno aplikaci toto léto veřejně vydat pod *open-source* licencí na portálu GitHub. Dá se předpokládat, že již v příštím běhu předmětu Programování grafiky bude aplikace do výuky aktivně zapojena

a nahradí zastaralou první verzi I3T, která byla k výuce používána posledních několik let.

Příloha A

Úvodní dotazník uživatelského testování

1. V jakém ročníku (ne semestru) jsi?
   - Prvním
   - Druhém
   - Třetím
   - Čtvrtém
   - Další
2. Chodíš na FEL nebo FIT? (FIT / FEL)
3. Měl/a jsi Lineární Algebru? (ANO / NE)
4. Jaké máš znalosti grafických transformací (zhruba)?
5. Které z těchto předmětů jsi absolvoval/a?
   - FEL - HRY
   - FIT - BI-BLE - Blender
   - FIT - BI-VHS - Virtuální herní světy
   - FIT - BI-MGA - Multimediální grafické aplikace
6. Měl/a jsi předmět PGR (Programování grafiky)? (ANO / NE)
7. Měl/a jsi předmět PGR povinně? (ANO / NE)
8. Máš předmět PGR tento semestr? (ANO / NE)
9. Máš zkušenosti s některými z těchto 3D programů? Případně s nějakými jinými?
   - Blender
   - Cinema 4D

- Unity
- Unreal Engine
- Godot Engine
- ZBrush
- Houdini
- 3DS Max
- Maya
- Jiné Autodesk produkty či CAD nástroje (případně konkrétní napiš níže)
- Žádné z nich

10. Slyšel/a jsi někdy o programu I3T? Pokud ano pracoval/a si s ním někdy nebo si o něm něco pamatuješ?
11. Jakým způsobem se ti hodí provést testování?
    - Distančně
    - Osobně
12. Kdy bys na něj měl/a čas? Případně napiš jakékoli poznámky.
13. Máš PC s Windows?
14. Zanechej prosím na sebe kontakt.

Seznam zkratek

AO	Ambient Occlusion (ambientní okluze)
API	Application Programming Interface
CAD	Computer Aided Design
CI	Continuous Integration (kontinuální integrace)
ECS	Entity Component System
FBO	Frame Buffer Object
FEL	Fakulta elektrotechnická
FIT	Fakulta informačních technologií
FOV	Field Of View (zorné pole)
FPS	Frames Per Second (snímků za vteřinu)
GLM	OpenGL Mathematics
GLSL	OpenGL Shading Language
GUI	Graphical User Interface (grafické uživatelské rozhraní)
I3T	Iteractive Tool for Teaching Transformations
IDE	Integrated Development Environment (vývojové prostředí)
JSON	JavaScript Object Notation
MSAA	MultiSample AntiAliasing
NDC	Normalized Device Coordinates (normalizované souřadnice zařízení)
OIT	Order-Independent Transparency
PGR	Programování grafiky
UI	User Interace (uživatelské rozhraní)
WBOIT	Weighted Blended Order-Independent Transparency

Glosář

Balíček

Balíček v kontextu programování typicky znamená shluk zdrojových souborů do nějakých celků. V této práci jde o složky obsahující hlavičkové (.h) a implementační (.cpp) soubory C++. 1, 10, 26

Desktopová aplikace

Samostatná aplikace vyvinutá pro operační systém stolního počítače, na rozdíl od mobilní či webové aplikace určené pro snadné využití i na přenosných zařízeních jako telefony či tablety. 1, 5

Graf scény

Datová struktura v podobě orientovaného acyklického grafu typicky používaná k uspořádání objektů v prostoru scény [10, s. 398]. 1

Moaré efekt

Vizuální artefakt způsobený překrýváním pravidelných obrazců. https://cs.wikipedia.org/wiki/Moare 1

Node editor

Komponenta uživatelského rozhraní, který umožňuje na 2D ploše upravovat uzly v grafu. 1

Pohledový objem (frustum)

Oblast v prostoru, kterou zabírá kamera. Někdy se nazývá záběr, nebo anglicky viewing frustum či viewing volume [10, s. 316]. 1, 5, 15

View matice

View matrix či pohledová matice je transformace, která převádí světové souřadnice do souřadnic prostoru kamery, čímž určuje pozici pozorovatele a stanoví směr a cíl pozorování [10, s. 307]. 1, 5, 15

Bibliografie

1. FOLTA, Michal. *Systém na výuku transformací*. 2016. Dostupné také z: http://hdl.handle.net/10467/64836. Diplomová práce. FEL ČVUT. Vedoucí práce Petr FELKEL.
2. HERICH, Martin. *Restrukturalizace interaktivního nástroje na výuku transformací I3T a reimplementace grafického rozhraní pomocí knihovny Dear ImGui*. 2021. Dostupné také z: http://hdl.handle.net/10467/96746. Bakalářská práce. FEL ČVUT.
3. GRUNCL, Daniel. *Manipulátory pro editaci transformačních matic a skriptování v I3T*. 2021. Dostupné také z: http://hdl.handle.net/10467/94657. Bakalářská práce. FEL ČVUT.
4. HOLEČEK, Jaroslav. *Adaptivní učení v softwarovém nástroji I3T pro výuku geometrických transformací*. 2023. Dostupné také z: http://hdl.handle.net/10467/107077. Diplomová práce. FEL ČVUT.
5. ZADINA, Vít. *Testování užitečnosti nástroje pro výuku transformací*. 2019. Dostupné také z: http://hdl.handle.net/10467/83400. Bakalářská práce. FIT ČVUT.
6. MCCONNELL, S. *Code Complete*. Pearson Education, 2004. Developer Best Practices. ISBN 9780735636972. Dostupné také z: https://books.google.cz/books?id=LpVCAwAAQBAJ.
7. KHRONOS GROUP, The Khronos® 3D Formats Working Group. *glTF™ 2.0 Specification Version 2.0.1* [online]. 2021. [cit. 2022-11-13]. Dostupné z: https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html.
8. KHRONOS GROUP. *The OpenGL® Graphics System: A Specification (Version 3.3 (Core Profile))* [online]. 2010. [cit. 2023-04-10]. Dostupné z: https://registry.khronos.org/OpenGL/specs/gl/glspec33.core.pdf.
9. DUBÉ, Marie-Eve. How to make an infinite grid. *A Slice Of Rendering* [online]. 2020 [cit. 2022-11-21]. Dostupné z: https://asliceofrendering.com/scene%5C%20helper/2020/01/05/InfiniteGrid/.
10. ŽÁRA, Jiří; BENEŠ, Bedřich; SOCHOR, Jiří; FELKEL, Petr. *Moderní počítačová grafika (2. vydání)*. Computer press, 2005. ISBN 80-251-0454-0.
11. MOGHADDAM, Mahan Heshmati. Order independent transparency. *LearnOpenGL* [online]. 2020 [cit. 2022-12-27]. Dostupné z: https://learnopengl.com/Guest-Articles/2020/OIT/Introduction.

12. MCGUIRE, Morgan; BAVOIL, Louis. Weighted blended order-independent transparency. *Journal of Computer Graphics Techniques*. 2013, roč. 2, č. 4. Dostupné také z: http://jcgt.org/published/0002/02/09/.

13. CORNUT, Omar. *Dear ImGui Bloat free Graphical User interface for C++* [online]. 2014. [cit. 2022-02-07]. Dostupné z: https://github.com/ocornut/imgui.

14. MARTIN, Adam. *What's an Entity System?* [online]. 2011. [cit. 2022-12-01]. Dostupné z: http://entity-systems.wikidot.com/.

15. BLINN, James F. Models of light reflection for computer synthesized pictures. In: *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*. 1977, s. 192–198.

16. VRIES, Joey de. Normal Mapping. *LearnOpenGL* [online]. 2014 [cit. 2022-09-28]. Dostupné z: https://learnopengl.com/Advanced-Lighting/Normal-Mapping.

17. REICHL, Jaroslav; VŠETIČKA, Martin. *Obzorníková soustava souřadnic* [online]. [B.r.]. [cit. 2023-04-22]. Dostupné z: http://fyzika.jreichl.com/main.article/view/931-obzornikova-soustava-souradnic.

20. OPENGL WIKI. *Renderbuffer Object — OpenGL Wiki* [online]. 2013. [cit. 2023-05-01]. Dostupné z: http://www.khronos.org/opengl/wiki_opengl/index.php?title=Renderbuffer_Object&oldid=9643.

21. OPENGL WIKI. *Image Format – OpenGL Wiki* [online]. 2022. [cit. 2023-05-01]. Dostupné z: http://www.khronos.org/opengl/wiki_opengl/index.php?title=Image_Format&oldid=14900.

22. AMEYE, Alexander. *5 ways to draw an outline* [online]. 2021. [cit. 2022-11-21]. Dostupné z: https://alexanderameye.github.io/notes/rendering-outlines/.

23. GUILLEMET, Cedric. *ImGuizmo* [online]. 2016. [cit. 2022-10-04]. Dostupné z: https://github.com/CedricGuillemet/ImGuizmo.

Zdroje obrázků

18. AG2GAEH. *Sférické souřadnice* [online]. 2015. [cit. 2023-04-20]. Dostupné z: https://commons.wikimedia.org/wiki/File:Kugelkoord-def.svg. CC BY-SA 4.0, obrázek upraven pro potřeby práce.
19. LANGEC. *Trackball (Logitech TrackMan)* [online]. 2005. [cit. 2023-04-22]. Dostupné z: https://commons.wikimedia.org/wiki/File:Logitech-trackball.jpg. CC BY 2.0.

Obsah přiloženého média

Přiložené médium obsahuje zdrojové kódy aplikace, včetně GLSL shaderů a upravené knihovny ImGuízmo. Ve složce `bin` se nachází spustitelná verze programu pro Windows, kterou lze načíst soubory testovacích scén *.scene* z adresáře `test-scenes` z hlavní lišty programu File › Open.

Součástí zdrojových kódu nejsou všechny závislosti kvůli snížení velikosti archivu (jsou však volně dostupné na internetu).

Jedná se o *commit* `b8d154` z neveřejného školního GitLab repozitáře :
https://gitlab.fel.cvut.cz/i3t-diplomky/i3t-bunny

V budoucnu by se měl repozitář veřejně objevit na následující adrese:
https://github.com/i3t-tool/i3t

Upravená knihovna ImGuizmo (*commit* `72ed47`) se také nachází ve veřejném repozitáři:
https://github.com/xDUDSSx/ImGuizmo

- readme.txt ..... popis obsahu média
- bin ..... adresář se spustitelnou verzí aplikace
  - I3T.exe ..... spustitelný soubor aplikace
- test-scenes ..... adresář se scénami testovacích úloh
- src ..... zdrojové kódy implementace
  - Source ..... C++ zdrojové kódy aplikace
  - Data
    - Shaders. ..... adresář s GLSL soubory shaderů
  - Dependencies .3 ImGuizmo ..... upravená verze knihovny ImGuzimo
    - pgr-framework ..... Interní knihovna pgr-framework
- thesis ..... složka práce ve formátu LaTeX
- thesis.pdf ..... text práce ve formátu PDF