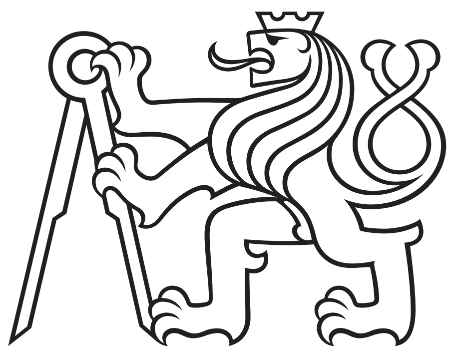
České vysoké učení technické v Praze
Fakulta elektrotechnická

Bakalářská práce

Restrukturalizace interaktivního nástroje
pro výuku transformací I3T a reimplementace
grafického rozhraní pomocí knihovny Dear ImGui

Martin Herich



Softwarové inženýrství a technologie

Katedra počítačů

Vedoucí práce: Ing. Petr Felkel, Ph.D.
2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Herich** Jméno: **Martin** Osobní číslo: **466384**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Restrukturalizace interaktivního nástroje na výuku transformací I3T a reimplementace grafického rozhraní pomocí knihovny Dear ImGui

Název bakalářské práce anglicky:

Pokyny pro vypracování:

Seznamte se s vývojem Interaktivního nástroje na výuku transformací (I3T) [1] a prací M. Folty a M. Nechanského [2, 3]. Navrhněte novou architekturu aplikace tak, aby kód logiky aplikace (propojení modulů a výpočty hodnot) byl oddělen od kódu uživatelského rozhraní. Zdokonalte a zpřehledněte zpracování událostí a konfigurovatelnost menu a klávesových zkratek. Přizpůsobte návrh potřebám knihovny Dear ImGui [4], která bude použita pro tvorbu elementů uživatelského rozhraní nástroje. Navrhněte a implementujte uživatelsky přívětivé nastavení stylů a implementujte návrh GUI dle studie Víta Zadiny [5] a grafického návrhu Lukáše Pilky [6]. Základní funkcionality nástroje (logiku aplikace) otestujte pomocí jednotkových testů.

Seznam doporučené literatury:

[1] Michal Folta. Teaching of Transformations. Diplomová práce, FEL ČVUT, 2016.
[2] Marek Nechanský. Automatické rozmisťování propojených modulů v nástroji I3T. Bakalářská práce, FEL ČVUT, 2019. http://dcgi.fel.cvut.cz/theses/2016/foltamic
[3] Petr Felkel, Alejandra Magana, Michal Folta, Alexa Gabrielle Sears, Bedrich Benes I3T: Using Interactive Computer Graphics to Teach Geometric Transformations. Eurographics Education Papers 2018, http://www.i3t-tool.org/
[4] Lukáš Pilka: Grafické návrhy rozhraní pro nástroj I3T, interní komunikace, 2018.
[5] Vít Zadina. Testování užitečnosti nástroje pro výuku transformací Fakulta informačních technologií, ČVUT, 2019.
[6] Omar Cornut a přispěvatelé. “Ocornut/Imgui.” GitHub, January 17, 2020. https://github.com/ocornut/imgui

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Petr Felkel, Ph.D., Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **12.02.2021** Termín odevzdání bakalářské práce: **13.08.2021**

Platnost zadání bakalářské práce: **30.09.2022**

Ing. Petr Felkel, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studenta

Poděkování

Rád bych poděkoval Ing. Petru Felkelovi, Ph.D. za vedení této práce. Velmi oceňuji jeho odborné rady a plné nasazení v řízení projektu. Dále poděkování patří všem kolegům, kteří se podílejí na zdokonalení nástroje I3T.

Prohlášení autora práce

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.

V Praze dne 11. 8. 2021 *Martin Herich*

Abstrakt

I3T je nástroj pro interaktivní výuku transformací. V současné době však neuspokojuje potřeby svých uživatelů. Má nepřívětivé zastaralé uživatelské rozhraní a není multiplatformní. Text se zabývá analýzou původního řešení a přípravou na přechod k implementaci nového uživatelského rozhraní tvořeného pomocí knihovny Dear ImGui. To obnáší návrh nové architektury aplikace a zpřehlednění starého kódu. Výsledkem práce je nový prototyp odvozený z původního kódu aplikace umožňující nástroj snadněji v budoucnu rozšířit nebo modifikovat, což dopomůže vývoj I3T v brzké době hladce dokončit.

Klíčová slova

I3T, C++, GUI, ImGui, GLFW, 3D transformace, počítačová grafika, OpenGL

Abstract

I3T is an educational tool for interactive teaching of transformations. Unfortunately, the application does not satisfy all users needs. It has obsolete unfriendly user interface and it is not multi-platform. The text describes analysis of former application and preparation for implementation of new user interface created using Dear ImGui library. It involves designing new architecture and make less complex and more readable code. As the result of the work new C++ project based on the former I3T source was created. It allows to easily extend and modify existing code to prepare I3T tool for its completion.

Keywords

I3T, C++, GUI, ImGui, GLFW, 3D transformations, computer graphics, OpenGL

Obsah

1 Úvod **1**

2 Stávající podoba nástroje I3T **3**
- 2.1 Jednotlivé změny . . . 5
- 2.2 Analýza původního kódu . . . 5
  - 2.2.1 Uživatelské rozhraní . . . 5
  - 2.2.2 Krabičky – provázání uživatelského rozhraní a logiky aplikace . . . 5
  - 2.2.3 Přenositelnost . . . 6

3 Návrh řešení **7**
- 3.1 Spolupráce v týmu . . . 7
- 3.2 Požadavky na nové řešení . . . 8
- 3.3 Technologie světa C++ . . . 9
  - 3.3.1 Build systémy . . . 9
  - 3.3.2 Multiplatformní kód . . . 12
- 3.4 Uživatelské rozhraní . . . 14
  - 3.4.1 Správa oken . . . 14
  - 3.4.2 Knihovna Dear ImGui . . . 14
- 3.5 Testování . . . 16
- 3.6 CI/CD . . . 16

4 Návrh architektury a implementace **19**
- 4.1 Struktura adresářů projektu . . . 19
- 4.2 Core . . . 22
  - 4.2.1 Třída Application . . . 22
  - 4.2.2 Příkazy . . . 24
  - 4.2.3 Obsluha vstupu . . . 26
  - 4.2.4 Moduly . . . 29
- 4.3 Datová reprezentace krabiček . . . 30
  - 4.3.1 Třída `Node` . . . 30
  - 4.3.2 Operátor . . . 33
  - 4.3.3 Transformace . . . 34
  - 4.3.4 Sekvence . . . 35
  - 4.3.5 Ostatní krabičky . . . 37
  - 4.3.6 Správa grafu krabiček . . . 37
  - 4.3.7 Přidání nového typu krabičky . . . 39
  - 4.3.8 Jednotkové testy . . . 40

4.4 CI/CD . . . 41
4.5 Uživatelské rozhraní . . . 43
4.5.1 Napojení Dear ImGui na vstup okna . . . 43
4.5.2 UI modul . . . 44
4.5.3 Node editor . . . 44
4.5.4 Vlastní styly . . . 45

5 Diskuze **47**
5.1 Kvalita kódu . . . 47
5.2 Použitelnost . . . 48
5.3 Webová verze . . . 49
5.4 Sjednocení změn . . . 49

6 Závěr **51**

Příloha A Obrázky a diagramy **53**

Příloha B Přiložené soubory **57**

Zkratky **59**

Literatura **61**

Seznam fragmentů kódu

3.1 Typická práce s CMake. 10
3.2 Typická práce s Premake. 11
3.3 Podmíněné kompilování. 13
3.4 Ukázka kódu UI s ImGui. 15

4.1 Příklad nové struktury projektu. 19
4.2 Jednoduchý `CMakeLists.txt` pro I3T. 20
4.3 Knihovna GLFW jako submodul. 21
4.4 Naklonování repozitáře se submoduly. 21
4.5 Inicializace nenaklonovaných submodulů. 21
4.6 Kód třídy `Application`. 22
4.7 Kód funkcí `logicUpdate` a `onDisplay`. 23
4.8 Definice konkrétních příkazů. 24
4.9 Používání příkazů. 25
4.10 Kód třídy `Command`. 26
4.11 Příklad registrace vstupní akce v okně. 28
4.12 Třída `Module`. 29
4.13 Kód operátorů. 33
4.14 Algoritmus pro detekci přítomnosti cyklu v grafu. 37
4.15 Jednotkový test v Google Test. 40
4.16 Jednotkový test v I3T. 40
4.17 CI/CD skript pro I3T. 41
4.18 Ukázka vytvoření uzlu s pomocí knihovny ImGui Node Editor. 44

5.1 Hypotetické slovníky pro jednotlivé jazykové mutace. 48

B.1 Obsah přiloženého CD. 57

Seznam obrázků

2.1 Stávající podoba nástroje. 4
2.2 Původní implementace krabičky. 6
2.3 Hluboká hierarchie tříd kolem krabiček. 6

4.1 Diagram třídy `Application`. 22
4.2 Hierarchie příkazů. 25
4.3 Diagram tříd spojených s obsluhou vstupu. 27
4.4 Detail třídy Node. 30
4.5 Hierarchie typů krabiček. 31
4.6 Popis průběhu aktualizace hodnot v grafu. 32
4.7 Detail implementace vybraných transformací. 35
4.8 Detail třídy `Sequence`. 35
4.9 Vnitřní struktura sekvence. 36
4.10 Zdánlivě cyklické zapojení dvou sekvencí. 36
4.11 UML diagram modulu UI. 43
4.12 Příklady uzlů node editoru. 45
4.13 Výchozí styly poskytované knihovnou Dear ImGui. 45
4.14 Editor stylu I3T. 46

A.1 Uživatelské rozhraní ve stylu *Classic*. 53
A.2 Uživatelské rozhraní ve stylu *Modern*. 54
A.3 Vztahy hlavních tříd aplikace. 55

B.1 Odkaz na repozitář s kódem. 57

Seznam tabulek

2.1 Seznam typů vstupů a výstupů krabiček. 3

3.1 Funkční požadavky. 8
3.2 Nefunkční požadavky. 8

4.1 Všechny typy transformací, které I3T nabízí. 34
4.2 Rozhraní třídy `GraphManager`. 37

Kapitola 1

Úvod

I3T, neboli *Interactive Tool for Teaching Transformations*, je nástroj umožňující interaktivní výuku geometrických transformací používaných nejen v počítačové grafice. Aplikace vznikla jako reakce na nedostatek kvalitních výukových programů poskytujících vyšší stupeň interaktivity a nabízejících kompletní sadu transformací využívaných v počítačové grafice. I3T umožňuje svým uživatelům názorným způsobem pochopit principy základních zobrazení jako jsou translace, změna měřítka, rotace nebo projekce. Uživatelé se seznámí s tím, jakým způsobem mohou být transformace aplikované na vrcholy 3D modelů a jakým způsobem mohou být tyto modely zobrazeny na obrazovku prostřednictvím kamery.

I3T vytvořil v rámci své diplomové práce Michal Folta na Katedře počítačové grafiky a interakce při Fakultě elektrotechnické ČVUT v Praze [15] a v průběhu dalších let pomohli program rozšířit další studenti [44, 30]. Nástroj je již od doby svého vzniku intenzivně používán při výuce látky předmětu Programování grafiky (PGR). Ne vždy se setkal s kladnou odezvou od svých uživatelů, zejména studentů. Byla mu vytýkána zastaralá grafika, neintuitivní ovládání a to, že je funkční pouze na Windows.

Odborní pracovníci a studenti uvedené katedry se rozhodli na tyto problémy reagovat. Lukáš Pilka vytvořil nový grafický návrh [35]. Poté bylo provedeno testování uživatelského rozhraní původního I3T a na základně jeho výsledků vznikl nový prototyp [45]. Pro dokončení modernizace nástroje zbývá vytvořit nové uživatelské rozhraní založené na expertním výzkumu.

Hlavním cílem této práce je navržení nové architektury aplikace tak, aby byl kód uživatelského rozhraní oddělen od kódu logiky aplikace a aby bylo možné při vývoji nového uživatelského rozhraní použít multiplatformní knihovnu Dear ImGui. Při návrhu je třeba přihlédnout k požadavku konfigurovatelnosti ovládání aplikace a uživatelského nastavení stylů.

Kapitola 2

Stávající podoba nástroje I3T

I3T je desktopová aplikace pracující s 3D grafikou prostřednictvím knihovny OpenGL. Byla napsána v jazyce C++ a je cílena na uživatele operačního systému Windows. Její uživatelské rozhraní se skládá ze dvou částí. Celé okno vyplňuje pohled do scény, ve kterém se nachází jednotlivé 3D objekty. Scénu zachycuje volná kamera, kterou lze pomocí myši a kláves ovládat. Část pohledu překrývá poloprůhledné rolovací okno pracovní plochy (nazývané *workspace*) s editorem grafu scény (*node editor*), umožňujícím utvářet hierarchii transformací 3D objektů propojováním připravených funkčních bloků – tzv. krabiček[1] (viz obr. 2.1).

Každá krabička představuje určitou geometrickou transformaci, sekvenci transformací nebo jinou matematickou operaci. Krabička může mít libovolně mnoho vstupů a výstupů a může v sobě držet jednu nebo více hodnot. Hodnota, kterou krabička obsahuje, zpravidla odpovídá typu příslušného výstupu s výjimkou maticového násobení. Tabulka 2.1 obsahuje výčet všech možných typů vstupů a výstupů a jejich barevné kódování.

Krabičky tvoří graf. Propojením výstupu jedné krabičky se vstupem krabičky druhé vzniká v grafu orientovaná hrana. Hrana vede od výstupu do vstupu. Existují tři typy spojů: násobící, přenášející hodnotu a přenášející signál. Krabičky uživatel může mezi sebou libovolně propojovat podle následujících podmínek: a) lze spojit jen vstup s výstupem stejného typu, b) na vstup lze připojit jen jeden výstup, jinak by nebylo možné jednoznačně určit, z jakého výstupu má být hodnota přečtena, c) v grafu nesmí vzniknout cyklus.

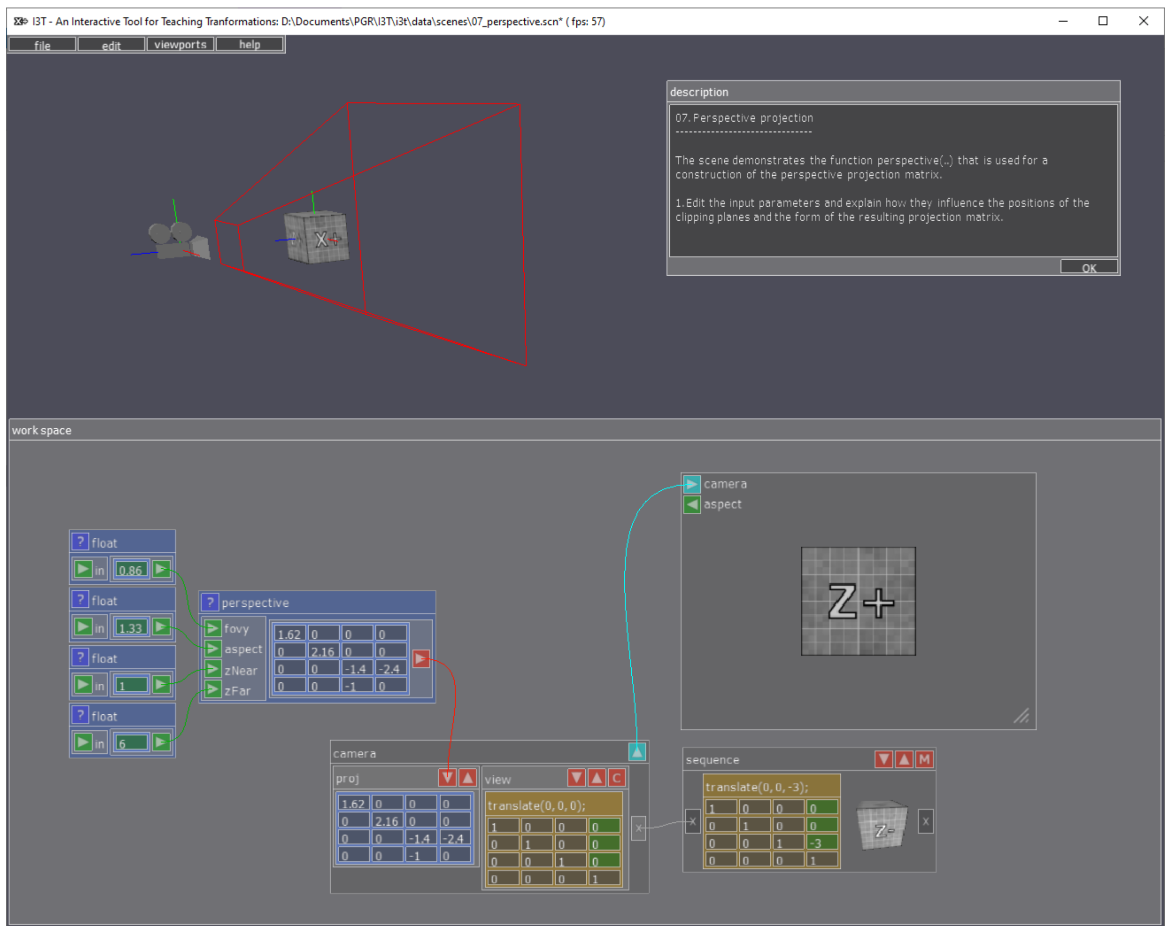
Tabulka 2.1: Seznam typů vstupů a výstupů krabiček.

Typ hodnoty	Barva	
Desetinné číslo		zelená
Třísložkový vektor		modrá
Čtyřsložkový vektor		hnědá
Kvaternion		oranžová
Matice řádu čtyři		červená
Maticové násobení		bílá
Pulz		purpurová
Pohled kamery		tyrkysová

[1] Vycházím z autorova názvosloví [15, s. 26].

V okně *workspace* nastavuje uživatel jednotlivým modelům či kamerám příslušné transformace pomocí krabiček. Transformace je třeba vkládat do *sekvencí*, které umožňují jejich skládání. Sekvence s transformacemi lze dále propojovat do stromu a reprezentovat jimi graf scény, ve kterém má každý uzel transformaci realizovanou jako složení transformací svého rodiče a svých lokálních transformací. Vytvořením krabičky *Camera* je do scény umístěna kamera, jejíž pohled je možné si nechat vykreslit do krabičky *Screen*. Kameře uživatel může nastavit dvě transformace – pohledovou, která ovlivňuje umístění kamery ve scéně a směr jejího pohledu, a transformaci projekční, která určuje, jakým způsobem má být obraz projektován do roviny pohledu kamery.

Pro provádění matematických operací slouží další typ krabiček – operátory. Ty se chovají jako funkce, mohou mít několik vstupů i výstupů. Mnoho operátorů má podobu funkcí z knihovny OpenGL Mathematics (GLM) [17]. Ta implementuje celou řadu matematických operací v úplně stejné podobě, v jaké je poskytuje jazyk OpenGL Shading Language (GLSL) [41].



Obrázek 2.1: Stávající podoba nástroje.

2.1 Jednotlivé změny

V průběhu pěti let vznikly tři práce zabývajících se nástrojem I3T. Dvě z nich se zabývaly podobou nástroje a jeho použitelností [45]. Dvě z nich přispěly ke změně zdrojového kódu. Marek Nechanský se zabýval automatickým rozmisťováním prvků v *node* editoru, změnil a zautomatizoval způsob vytváření jednotlivých krabiček, což vedlo ke zvýšení přehlednosti kódu [30]. Filip Uhlík přidal do aplikace možnost vytváření záznamů uživatelské činnosti prováděné v aplikaci [44]. I3T od té doby umožňuje zapisovat veškerou interakci s uživatelem a další akce, které jsou vhodné pro testování s uživateli.

2.2 Analýza původního kódu

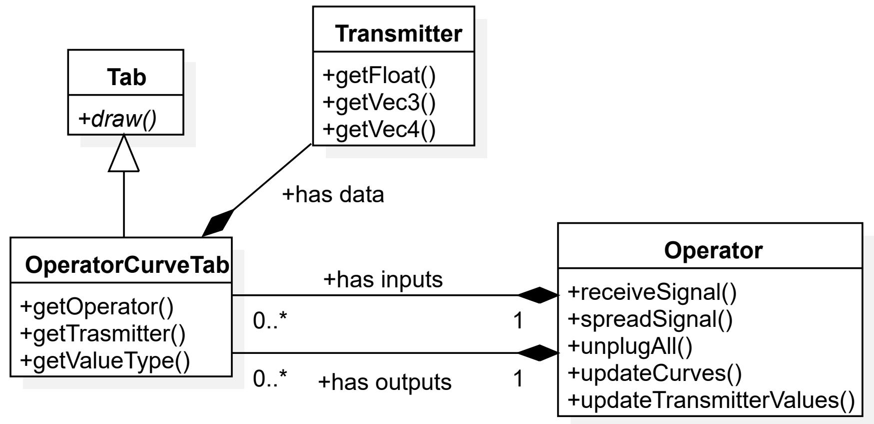
Po krátkém zkoumání kódu se ukázalo, že pravděpodobně nebude příliš snadné provádět jeho úpravy. Je možné, že autor nepočítal s tím, že někdy v budoucnu může být kód modifikován nebo rozšířen. Jednotlivé třídy mají mezi sebou mnoho vztahů i hlubokou hierarchii, někdy i pětinásobnou. Dalším faktem je nedostatečná separace domén aplikace. Zejména datová reprezentace krabiček je příliš úzce spjata s prvky uživatelského rozhraní. Původní kód se potýká s problémy popsanými v následujících podkapitolách.

2.2.1 Uživatelské rozhraní

Prvním objeveným problémem byla nemožnost přizpůsobení uživatelského rozhraní ať už pomocí konfiguračního souboru nebo prostřednictvím zavolání několika funkcí v kódu. I3T nepoužívá žádnou knihovnu, která by se starala o vytváření a obsluhu prvků uživatelského rozhraní. Vše je realizováno jako hierarchické vykreslování texturovaných obdélníků. Kód, který se o tyto záležitosti stará, není příliš modifikovatelný. Vzhledem k tomu, že je nástroji často vytýkáno jeho neintuitivní prostředí, a to, že má zastaralou grafiku [45], je třeba kód uživatelského rozhraní změnit. To se ale jeví v původní verzi implementace jako velmi obtížné.

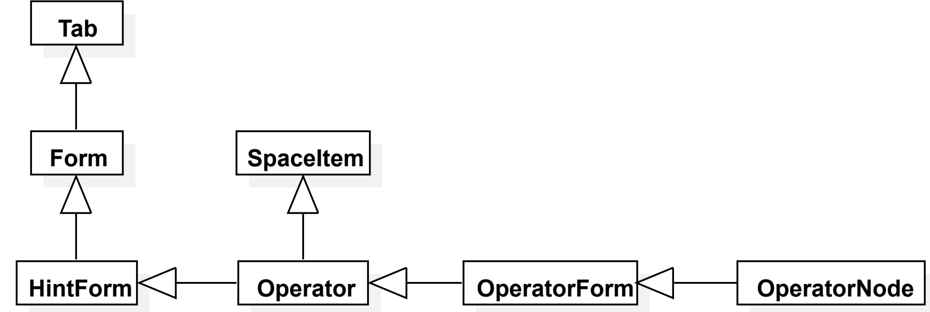
2.2.2 Krabičky – provázání uživatelského rozhraní a logiky aplikace

Dalším problémem, a to se týká zejména kódu krabiček, je nevhodné provázání kódu uživatelského rozhraní a logiky aplikace. Z obrázku ilustrujícího strukturu implementace krabiček (obr. 2.2) je zřejmé, že třída `Operator`, která je základní třídou pro všechny typy krabiček, obsahuje reference na třídu `OperatorCurveTab`. Ta představuje vstup nebo výstup krabičky. Je ale nevhodné, že je tato třída odvozena od třídy `Tab`, která je grafickou reprezentací klikatelného prvku. Ta obsahuje čistě virtuální členskou funkci `Tab::render`, jíž třída `OperatorCurveTab` implementuje. Z tohoto příkladu vidíme, že autor vše odvozoval od grafického uživatelského rozhraní.



Obrázek 2.2: Původní implementace krabičky.

Dalším již zmíněným problémem je příliš hluboká hierarchie jednotlivých tříd, která znesnadňuje jakoukoliv změnu. Jednotlivé třídy jsou provázané skrze dědičnost. Na obrázku 2.3 je uvedena hierarchie rodičů třídy `OperatorNode`, která je koncovou reprezentací krabičky typu operátor. Z tohoto diagramu opět vidíme, že základní reprezentace operátoru je odvozena od uživatelského rozhraní (třída `Tab`).



Obrázek 2.3: Hluboká hierarchie tříd kolem krabiček.

V literatuře zabývající se OOP je často uváděno, že má být preferována kompozice nad dědičností [18]. Dědičnost tvoří totiž příliš silnou vazbu mezi dvěma třídami. Nelze za běhu měnit instanci rodičovské třídy, což činí třídy v našem kódu více provázanými. Obecně je doporučeno se silné provázanosti tříd vyhnout (*Loose coupling* [24]), neboť se kód stává hůře rozšířitelným nebo modifikovatelným. Důsledkem uvedené hierarchie je také například to, že při vytváření instance třídy `OperatorNode` je nutné v jejím konstruktoru předat parametry konstruktorům všech jejích předchůdců.

2.2.3 Přenositelnost

Uživateli často vytýkanou vadou je, že I3T je dostupné pouze pro Windows. Nepříjemností, která znemožňuje jednoduché přenesení kódu na jiné operační systémy, je, že I3T využívá funkce z Win32 API. Tato volání slouží zejména k otevření souborových dialogů nebo systémových dialogů informujících uživatele o chybě, která znemožňuje správně spustit aplikaci.

Kapitola 3

Návrh řešení

Kapitola objasňuje postup modernizace nástroje I3T a technologie, které budou provázet vývojáře aplikace. Je zde zmíněno několik postupů, se kterými se můžeme setkat u jiných C++ projektů. Nejprve probereme problematiku vývojářských nástrojů, zejména těch, které zajišťují sestavování a distribuci knihoven. Poté následuje popis způsobu použití knihovny Dear ImGui pro tvorbu uživatelského rozhraní a na závěr jsou přiblíženy techniky a kroky automatického sestavení a testování kódu (CI/CD).

3.1 Spolupráce v týmu

Nové I3T vzniká ve spolupráci pěti studentů. Každý člen tohoto týmu pracuje na poměrně nezávislé části aplikace. V následujícím výčtu jsou dle abecedy uvedeni jednotliví studenti a charakteristika jejich podílu práce na rozvoji nástroje I3T.

Daniel Gruncl	3D scéna, manipulátory s objekty a skriptování.
Martin Herich	Architektura klíčových částí aplikace, logická reprezentace krabiček a nastavování stylů.
Jaroslav Holeček	Návrh GUI reprezentace krabiček a adaptivní nabídka výukových tutoriálů na základě jejich úspěšnosti.
Miroslav Müller	Podpora snadného vytváření výukových tutoriálů a sada výukových scén.
Sofie Šašorina	Realizace GUI reprezentace krabiček a úprava jejich podoby.

Práce Sofie Šašoriny zabývající se novou grafickou podobou krabiček [46] a Daniela Gruncla [21] zabývající se 3D manipulátory a skriptováním jsou v současné době již publikovány.

3.2 Požadavky na nové řešení

Hlavním požadavkem nového řešení je vytvořit nové přizpůsobitelné a intuitivnější uživatelské rozhraní. To znamená vybavit aplikaci takovými funkcionalitami, které by umožnily měnit její vzhled, případně nastavit vlastní klávesy pro vstupní akce.

Z již uvedených skutečností také vyplývá, že je nutné celý kód aplikace značně restrukturalizovat – pokusit se co nejvíce zredukovat provázanost jednotlivých tříd. Dále je nutné navrhnout nový způsob realizace uživatelského rozhraní, který bude přizpůsoben potřebám knihovny Dear ImGui. Ta byla zvolena pro tento účel díky tomu, že se snadno integruje do existujících projektů, poskytuje dokovatelná okna, jednoduše se používá a je přenositelná. Při realizaci je třeba postupovat tak, aby UI bylo modifikovatelné a nastavitelné a podporovalo změnu stylu, kterou by mělo být možné provést bez znalosti jazyka C/C++. Tento krok samotný nakonec pravděpodobně bude vyžadovat vyřazení velké části starého kódu, ať už proto, že je v něm logika aplikace příliš úzce spjata s kódem uživatelského rozhraní, nebo že kód, který realizuje vykreslování prvků UI, již nebude potřebný, neboť bude nahrazen kódem využívajícím knihovnu Dear ImGui.

Kód by měl být psán tak, aby se v něm mohl kdokoliv dobře zorientovat, a musí ho být možné v budoucnu snadno rozšířit, přidat nové krabičky nebo modifikovat uživatelské rozhraní.

Všechny požadavky, které mělo nové řešení splňovat (navíc od původního), by se daly shrnout do dvou tabulek (3.1, 3.2).

Tabulka 3.1: Funkční požadavky.

Název	**Popis**
F01 – Změna stylu	Styl lze jednoduše modifikovat a uložit/načíst ze souboru.
F02 – Konfigurace vstupních akcí	Aplikace bude umožňovat jednoduše změnit mapování kláves a tlačítek myši.
F03 – Dokovatelná okna	Možnost využití více monitorů.

Tabulka 3.2: Nefunkční požadavky.

Název	**Popis**
N01 – Přenositelnost aplikace	Aplikaci lze použít na platformě Windows, Linux nebo MacOS.
N02 – Oddělený kód krabiček	Kód logické reprezentace krabiček je zcela nezávislý na kódu UI.
N03 – Knihovna Dear ImGui	Uživatelské rozhraní je realizováno pomocí knihovny Dear ImGui.

3.3 Technologie světa C++

Původní I3T je napsáno v jazyce C++ pro platformu Windows, včetně použití funkcí z Win32 API. Je tedy velmi úzce spjato s vývojovými nástroji sady Microsoft Visual Studio. Tato skutečnost je sice pohodlná pro uživatele systému Windows, ale zcela nevyhovující pro uživatele jiných platforem.

Jedním z požadavků na modernizovaný nástroj I3T je, aby byl multiplatformní, tedy dostupný pro uživatele různých operačních systémů. S ohledem na tuto skutečnost je nutné již v samotném začátku projektu zvolit vhodné nástroje a knihovny, které nám plnění tohoto cíle usnadní. Bude výhodou, pokud si zvolíme takové multiplatformní nástroje a knihovny, které jsou oblíbené a široce užívané vývojáři z oblasti počítačové grafiky. Následující podkapitoly představí nástroje zajišťující překlad a sestavení C++ kódu a nástroje na správu knihoven.

3.3.1 Build systémy

Překlad C++ kódu zajišťuje kompilátor. Linker poté zajišťuje propojení dílčích jednotek vedoucí k vytvoření spustitelného binárního souboru nebo knihovny. Kompilátoru lze předat různé parametry, zejména cesty k souborům, které je třeba přeložit. Stejným způsobem lze parametrizovat i spuštění linkeru. Pro snadnější správu projektů je ovšem vhodné tyto úkony automatizovat – pro všechny soubory mít definované globální parametry a například pomocí jednoho příkazu sestavit celý projekt.

Existují nástroje nebo celé sady vývojářských nástrojů, které tyto úkony umí provést. Asi zdaleka nejpoužívanějšími jsou GNU make a MSBuild, který je součástí sady Microsoft Visual Studio [36, 25]. Každý nástroj má ale své vlastní konvence zápisu jednotlivých pravidel. Tyto nástroje jsou také velmi svázané s jednotlivými operačními systémy. Například GNU *toolchain* není jednoduché používat na platformě Windows a MSBuild nelze jednoduše použít na jiných platformách než Windows. To může být jednou z příčin, proč vznikly další nástroje, které umožňují použití na více platformách.

CMake

Nejrozšířenějším nástrojem zajišťujícím sestavení C/C++ projektů je nástroj CMake (Cross Make) [23]. Ten umožňuje zcela jednotným způsobem definovat pravidla sestavení pro celou řadu vývojářských nástrojů[1]. Jedním příkazem pak CMake vygeneruje příslušná pravidla pro danou platformu, vytvoří například Makefile pro *Unix-like* systémy využívající GNU *toolchain* nebo `.vxproj` soubory pro MSBuild (Visual Studio). Vzhledem ke svému širokému využití je přímo podporován různými integrovanými vývojovými prostředími (CLion, Visual Studio).

[1]GNU Make, NMake, Visual Studio, Ninja a mnohé další [23, cmake-generators(7)].

```
1  # In the project root, create cmake output directory "out/"
2  # and move into it.
3  mkdir out && cd out
4
5  # Build MSVC project files in the "out/" directory.
6  cmake -DCMAKE_BUILD_TYPE=Release \
7        -A Win32                   \
8        -G "Visual Studio 16 2019" \
9        ..
10
11 # Or generate GNU Makefiles on Linux.
12 cmake -DCMAKE_BUILD_TYPE=Release \
13       -G "Unix Makefiles"        \
14       ..
15
16 # Build project.
17 cmake --build . -j 4
```

Kód 3.1: Typická práce s CMake.

Výše uvedená sekvence příkazů (ukázka 3.1) provede vygenerování pravidel pro sestavení projektu v *Release* konfiguraci pro MSVC a také pro nástroj Make. Poslední příkaz zahájí s použitím dostupného kompilátoru C/C++ sestavení se čtyřmi vlákny.[2]

CMake ale není jediným nástrojem, který výše popsané úkony umí provést. C++ je jazyk s velmi dlouhou historií, a proto není divu, že existuje více podobných nástrojů, které nabízejí stejné možnosti. Nástroji CMake je často vytýkáno, že jeho vlastní skriptovací jazyk, který vývojář využívá pro psaní souborů CMakeLists.txt, není příliš přehledný. Jedná se o časté volání maker, uživatel jich musí znát poměrně mnoho, aby dosáhl toho, co potřebuje nebo aby mohl CMake naplno využít.

Další omezující skutečností může být to, že pro různé konfigurace sestavení, např. variantu *Release* nebo *Debug*, kdy kompilátor neprovádí žádné optimalizace, musíme znovu generovat soubory pro dané build systémy (neplatí pro Visual Studio). Nelze to tedy provést jen jednou při prvotní přípravě prostředí.

Premake

Premake je nástroj, který má kratší historii než CMake [34]. Podporuje stejné množství *toolchainů* jako CMake. Působí odlehčeně, není zatížen zbytečnými funkcionalitami. Pro někoho může být benefitem, že nevyužívá žádný vlastní skriptovací jazyk. Všechna pravidla jsou psána v jazyce Lua. To může například vyhovovat vývojářům počítačových her, kteří tento jazyk často využívají.

[2]Tato možnost může být ignorována. Každý *build* systém přistupuje ke kompilování za použití více vláken jinak. Například NMake, který je součástí sady nástrojů Visual Studio, vůbec nepodporuje vícevláknové sestavení. Visual Studio (2019) zase neumožňuje specifikovat přesný počet vláken, nabízí pouze parametr `/MP`, který zapne podporu více vláken.

```
# Generate project files for Visual Studio 2019.
premake vs2019

# Or generate GNU makefiles.
premake gmake

# Build project release configuration with GNU make.
make project config=release
```

Kód 3.2: Typická práce s Premake.

Kód 3.2 obsahuje sekvenci příkazů, která provede stejné operace jako ukázka kódu 3.1 s CMake, jen s tím rozdílem, že Premake slouží pouze k vygenerování pravidel pro sestavení. Program není schopen sestavit projekt.

Velkým omezením může být fakt, že tento nástroj není příliš rozšířen. Skoro žádný projekt, jehož knihovny chceme využívat, premake nepoužívá.[3] Kód některých knihoven je sice poměrně jednoduchý, pro takové není složité napsat vlastní premake skript. Pro jiné větší projekty to může být bez bližšího seznámení s kódem nebo s jednotlivými kroky sestavení nesmírně komplikované.

Používání premake je přínosné v tom, že umožňuje jednorázové generování souborů pro konkrétní *build* systémy bez nutnosti častého regenerování projektu. Co na rozdíl od CMake tento nástroj neumí, je identifikace dostupného C/C++ kompilátoru a automatické sestavení. Vygeneruje jen pravidla sestavení pro zadaný build systém.

Další sestavovací systémy

Bazel je nástroj, který vytvořila firma Google. Klade si za cíl snadnou rozšířitelnost funkcionalit a podporu pro více programovacích jazyků, nejen pro C/C++ [19]. Bohužel, stejně jako Premake není příliš hojně rozšířen.

Dalším zajímavým zástupcem je SCons [7]. Jedná se o nástroj, pro který se zapisují pravidla sestavení v jazyce Python. Nástroj se snaží být především náhradou za Unix Make, který může být pro začátečníka složitý. Používá ho například Godot Engine. Jeho slabinou může být, že pracuje pomaleji než jiné *build* systémy [33].

Závěr

CMake není sice jediným nástrojem ve světě jazyka C++, který by nám všechny výše uvedené benefity poskytoval. Je ale jakýmsi standardem. Využívá ho drtivá většina C++ projektů, které cílí k využití na více platformách. Proto jsme se ho rozhodli použít pro sestavení našeho I3T.

[3] Namátkou například GLFW, spdlog nebo assimp.

3.3.2 Multiplatformní kód

CMake zajišťuje pohodlné sestavení aplikace na více platformách. Abychom zajistili to, aby I3T bylo skutečně multiplatformní, je třeba najít způsob, jak spravovat potřebné závislosti – knihovny. I3T jich využívá poměrně mnoho a není elegantní je verzovat a distribuovat spolu se zdrojovým kódem aplikace. Tento přístup by vyžadoval zahrnout do zdrojového kódu verzi každé knihovny sestavené pro všechny tři majoritní operační systémy.[4] Také není příliš vhodné přenechat správu závislostí na uživateli nebo na novém vývojáři, který si chce či potřebuje I3T sestavit ze zdrojového kódu. Pro uživatele operačního systému Windows není instalace knihoven tak snadná jako pro uživatele libovolné linuxové distribuce se zabudovaným balíčkovacím systémem.

Mnoho programovacích jazyků má k dispozici zavedené nástroje, které umožňují jednoduše spravovat závislosti. To ale není případ C++, které na rozdíl od drtivé většiny programovacích jazyků žádnou takovou technologií vlastně nedisponuje [14]. Existuje několik řešení jako například Vcpkg od Microsoftu [27] nebo Conan [22]. Zatím se ale nejedná se o rozšířené nástroje.

Proto jsem se rozhodl využít nativní funkci verzovacího nástroje git – možnosti zahrnutí dalších repozitářů jako submoduly [10]. Repozitář s projektem I3T bude poté možné naklonovat spolu se zdrojovým kódem potřebných knihoven. Jelikož námi použité knihovny využívají CMake, je velice snadné je zahrnout do našeho CMake projektu jako podprojekty. Jediným omezením tohoto přístupu je nemožnost specializovat se na konkrétní verze knihoven. Nástroj git naklonuje kód submodulu odpovídající stavu, který zachycuje poslední *commit* na větvi master.[5] Vyžadujeme-li určitou verzi knihovny, musíme po naklonování zajistit přepnutí verze kódu k verzi nějakého jiného *commitu*, označeného například *tagem*.

Pro zajištění přenositelnosti aplikace je nutné obezřetně zacházet s rozhraními, které operační systémy nabízejí. Operační systémy jsou často psány v jazyce C, případně C++, a poskytují programátorům funkce pro práci s periferiemi, s procesy, s vlákny, se soubory nebo s okny. Pohlédneme-li do historie operačních systémů, zjistíme, že ke konci 80. let došlo k masivnímu rozšíření rozličných distribucí UNIX systémů (FreeBSD, OpenBSD a NetBSD). Aby byla zajištěna přenositelnost aplikací tvořených pro tyto systémy, vyvstala potřeba standardizovat rozhraní operačního systému, které aplikace využívají. Proto IEEE vytvořilo standard POSIX, který většina UNIX/UNIX-like systémů podporuje [40].

Systém Windows bohužel není plně POSIX kompatibilní [26], z toho důvodu nemůžeme použít všechny funkce, které POSIX definuje. Proto bude pro nás dobré striktně se držet pouze funkcí definovaných standardy jazyka C++, které mohou zajistit snadnější přenositelnost. Například Windows neposkytuje možnost využití POSIX vláken. Pokud ale použijeme C++ `std::thread` [4, §iso.30.3.1], přenecháme platformně závislé použití nativních vláken operačního systému na tvůrcích STL knihovny pro danou platformu. Dobrým zdrojem pro sledování aktuálně podporovaných funkcionalit STL knihovny jazyka C++ napříč mnoha kompilátory je web cppreference.com. Ten uvádí v sekci *C++ compiler support*[6] užitečný přehled o verzích kompilátorů, ve kterých je požadovaná funkcionalita

[4] Windows, Linux a MacOS.

[5] V případě submodulu je sice možné cílovou větev explicitně specifikovat, ale i to nemusí být dostačující.

[6] https://en.cppreference.com/w/cpp/compiler_support

již podporována. Pro potřeby vývoje I3T jsou důležité především GCC, Clang a Microsoft Visual Studio.

Pokud by kód I3T nakonec vyžadoval nějaké specifické chování pro různé operační systémy nebo kompilátory, je nutné ho psát do bloků umožňujících podmíněnou kompilaci s použitím maker pro preprocesor (viz kód 3.3).

```
1  #if defined(_MSVC_VER)
2  // Disable Visual Studio's CRT secure warnings
3  // for not using *_s functions (strcpy_s, strlen_s),
4  // which are not portable.
5  #pragma warning(disable:4996)
6  #endif
7
8  #if defined(__linux__) || defined(__unix__)
9  // Open a pipe ...
10 #elif defined(_WIN32)
11 // Do something with WindowsAPI ...
12 #else
13 #error "Unsupported platform"
14 #endif
```

Kód 3.3: Podmíněné kompilování.

3.4 Uživatelské rozhraní

Kapitola stručně shrnuje problematiku, se kterou se setká každý vývojář vytvářející aplikace využívající 3D funkce poskytované grafickou kartou.

3.4.1 Správa oken

I3T je desktopová aplikace, která „běží" přímo v operačním systému. Abychom vůbec mohli začít využívat funkce spojené s 3D grafikou, je nutné vytvořit OpenGL kontext. Ten reprezentuje *framebuffer*, do kterého pomocí OpenGL příkazů vykreslujeme, a příslušné stavy OpenGL. Způsob vytvoření kontextu ale není součástí specifikace OpenGL a je tedy platformně závislý [42]. Jinak budeme vytvářet okno s OpenGL kontextem za použití Win32 API v prostředích Windows s Desktop Window Manager; a jinak za použití OpenGL Extension to the X Window System (GLX) v prostředích *Unix-like* systémů s X Window System.

Aby byl vývoj aplikace usnadněn, často vývojáři sáhnou po knihovně, která výše popsané úkony provede za ně. Příkladem mohou být projekty FreeGLUT, GLFW, SDL2 nebo SFML. Každá knihovna poskytuje trochu jiné funkcionality. FreeGLUT nebo GLFW zajišťuje pouze vytvoření OpenGL kontextu a předání vstupních akcí v okně naší aplikaci. SDL2 umožňuje i načítat textury a SFML dokonce nabízí rozhraní pro jednoduché přehrávání zvuků.

V původním I3T Michala Folty byla použita knihovna FreeGLUT (Free OpenGL Utility Toolkit), ta se nám ale jevila již jako zastaralá. Jedná se o projekt, jehož počátky sahají až do roku 2003. Kód knihovny k dnešním dnům již není udržovaný.[7] Navíc ImGui tuto knihovnu podporuje jen okrajově.[8] Na základě rešerše Miroslava Müllera [28] jsme použili k implementaci nového uživatelského rozhraní knihovnu GLFW (Graphics Library Framework) [2].

3.4.2 Knihovna Dear ImGui

K vytváření prvků uživatelského rozhraní aplikace I3T jsme zvolili knihovnu Dear ImGui [32]. Ta je velice minimalistická a snadno integrovatelná do již existujícího projektu. Poskytuje nám možnost vytvářet celou řadu komponent uživatelského rozhraní, jako jsou například oddělitelná podokna, kontextové nabídky, tlačítka, vstupní boxy a mnohé další. Podporuje také pokročilejší funkcionality, jako je například dokování jednotlivých oken.

Na rozdíl od jiných zavedených frameworků a knihoven pro tvorbu uživatelského rozhraní, jako je například Qt, kde kontrolu nad během aplikace přebírá daný framework [43], ImGui dává veškerou kontrolu programátorovi. Ten má plně ve své režii hlavní smyčku aplikace a obsluhu vstupu. Knihovna ImGui neobsahuje žádné komponenty, které by byly realizovány jako datové struktury, které programátor může vytvářet a například vnoře-

[7] Po několikaleté přestávce ve vývoji, kdy projekt vypadal již jako mrtvý, vyšly dvě nové verze [1].

[8] Autor knihovny uvádí v souboru `backends/imgui_impl_glut.h` v [32] následující doporučení: „GLUT/FreeGLUT IS OBSOLETE PREHISTORIC SOFTWARE. Using GLUT is not recommended unless you really miss the 90's."

vat jednu do druhé. Pracuje jen s daty, které jí předáme. Zobrazování UI je realizováno jako periodické volání funkcí zobrazujících jednotlivé prvky a nastavujících stav ImGui. Tento přístup tvorby uživatelského rozhraní se nazývá Immediate Mode GUI (IMGUI) [8].

Knihovna ImGui podporuje všechna hlavní grafická API,[9] lze ji integrovat do grafické aplikace využívající prakticky jakýkoliv okenní systém. Jediné, co uživatel musí provést, je *napojení* ImGui na daný okenní systém tak, aby ImGui obdrželo vstupní události vznikající v okně.

```
1  #include <GLFW/glfw3.h>
2  #include <imgui.h>
3
4  char buf[1024];
5  float fl = 0.0f;
6
7  /// Called each frame.
8  void Render()
9  {
10     ImGui::NewFrame();
11
12     ImGui::Begin("Test Window");
13
14     ImGui::Text("Hello, world %d", 123);
15     if (ImGui::Button("Save"))
16         MySaveFunction();
17     ImGui::InputText("string", buf, IM_ARRAYSIZE(buf));
18     ImGui::SliderFloat("float", &fl, 0.0f, 1.0f);
19
20     ImGui::End();
21
22     ImGui::Render();
23
24     glfwSwapBuffers(window);
25 }
```

Kód 3.4: Ukázka kódu UI s ImGui (převzato z [32]).

Kód 3.4 ilustruje způsob vytváření UI pomocí ImGui. Funkce `Render` je volána periodicky, například jednou za 1/60 sekundy, a v každém tomto snímku je volána funkce `ImGui::Begin`, která vytvoří okno, do kterého jsou umisťovány prvky UI, jako například `ImGui::Text` nebo `ImGui::Button`. Můžeme si povšimnout, že funkce `ImGui::Button` vrací hodnotu typu `bool`, která indikuje, zda bylo tlačítko stisknuto. Se všemi prvky UI vytvořenými pomocí ImGui se interaguje takto přímo, jak je vidět na řádku č. 17, kdy je číselnému posuvníku předávána přímo adresa proměnné, která má být modifikována.

[9]OpenGL, Vulkan a DirectX [32].

3.5 Testování

Člověk je tvor chybující, proto software, který tvoří, téměř vždy obsahuje nějaké chyby. Abychom na chyby přišli, je nutné kód, který píšeme, otestovat. Asi první způsob testování, který se nabízí, je aplikaci spustit a vyzkoušet si klikáním nebo například zadáváním hodnot do prvků UI, zda funguje tak, jak by měla. Tento přístup je ale velmi naivní. Může být pracný a nemusíme jím otestovat vše, co potřebujeme. Navíc vyžaduje spustitelnou aplikaci, kterou nemusíme během vývoje mít k dispozici.

V našem případě bude největší prioritou automatizovaně otestovat funkčnosti jednotlivých krabiček, tedy jejich propojování a rozpojování, nastavení hodnot a jejich výpočetní vlastnosti. Jedná se o funkcionality, které jsou naprosto izolované od zbytku systému a tedy tvoří samostatnou jednotku. Testy, které testují takovéto části systému, se nazývají *jednotkové*. Vyznačují se svou krátkostí a není pro ně nutné připravovat testovací prostředí.

Interakci jednotlivých částí systému testují *integrační testy*. Nabízí se, abychom automatizovaně testovali uživatelské rozhraní a zkoumali, zda například nastavení hodnoty v krabičce přestavující translaci posune 3D model na správné místo. Pro webové technologie existuje například velmi známý nástroj Selenium, který uživatelské rozhraní tímto způsobem otestuje. Pro knihovnu Dear ImGui jsem žádný takový nástroj nenašel. Autor knihovny publikované na webu github.com různé animace, na kterých je zachycen průchod automatizovanými testy uživatelského rozhraní. Vypadá to ale, že se jedná o soukromý kód, který nikdy nebude určen k zveřejnění.

3.6 CI/CD

CI/CD (*Continuous Integration* a *Continuous Delivery*) je soubor technik a operací vedoucích k automatizaci a zkvalitnění vývoje software. Koncept *Continuous Integration* spočívá v rychlé integraci změn od členů týmu do hlavní vývojové větvě [16]. Každá taková změna je ověřena pomocí automatizovaného sestavení a spuštění testů. CI je jednou z dvanácti základních praktik metodiky extrémního programování, které publikoval Kent Beck [9].

V současné době na nové verzi I3T pracuje pět vývojářů. Každý z nich několikrát denně nahraje svůj kód do sdíleného repozitáře. Pro naše účely bude naprosto dostačující ověření příspěvku rozdělit do dvou operací. První fází kontroly bude, zda daný kód neobsahuje chyby, které je možné odhalit ve fázi kompilace. Pokud první fáze úspěšně projde, nastane fáze druhá – kontrola chyb, které mohou zachytit jednotkové testy. Pokud i ty projdou, prohlásíme, že dané změny můžeme bezpečně integrovat do hlavní vývojové větve.

Dalším benefitem by mohlo být zavedení *continous delivery*. V tomto procesu dochází k automatickému sestavení verze programu určené k publikování. Bylo by výhodné mít zajištěný automatický *build* I3T po nahrání změn do hlavní vývojové větve (*master*). Existují on-line služby a prostředí (například AppVeyor), které se zaměřují přímo na tento úkon. Často přímo podporují sestavení hned pro několik platforem, díky čemuž je proces publikování nové verze aplikace značně zjednodušen.

Pro současné potřeby vývoje I3T je naprosto dostačující zavést pouze *continuous integration*. Existuje celá řada služeb, které umí provést tyto aktivity. Většinou jsou ale zpo-

platněné nebo nějak jinak omezené, nejčastěji nabízejí maximálně desítky stovek minut výpočetního času na jeden měsíc. Pro potřeby vývoje I3T jsme zvolili Gitlab, který nám Fakulta elektrotechnická poskytuje neomezeně a zcela zdarma. Jeho hlavní výhodou je, že nebudeme muset do vývoje zapojovat žádnou další externí službu, kterou by bylo nutné spravovat. Gitlab již využíváme ke správě kódu I3T.

Kapitola 4

Návrh architektury a implementace

Následující kapitola detailněji popisuje jednotlivé kroky návrhu architektury a implementace nového I3T. Jedná se o návrh struktury adresářů nového projektu, návrh klíčových částí aplikace – *Core* – a jejich otestování. Jednotlivé sekce jsou doplněny o UML diagramy vypracované podle specifikace OMG UML [31].

4.1 Struktura adresářů projektu

Před samotným vytvářením souborů s kódem a dalších souborů projektu bylo nutné navrhnout srozumitelnou strukturu adresářů projektu (viz kód 4.1). Adresář `Dependencies` obsahuje zdrojový kód potřebných knihoven. Kód I3T včetně hlavičkových souborů je zahrnut v adresáři `Source`.

```
1  Dependencies/        - Externí knihovny.
2  Docs/                - Soubory pro generování dokumentace.
3  Source/              - Zdrojový kód I3T.
4      Commands/
5      Core/
6      GUI/
7      Logger/
8      Rendering/
9      Tutorial/
10     Utils/
11 .clang-format        - Formátovací pravidla.
12 .gitignore           - Soubory, které verzovací systém nespravuje.
13 .gitlab-ci.yml       - Konfigurace CI.
14 .gitmodules          - Seznam externích modulů.
15 CMakeLists.txt       - Pravidla pro sestavení projektu.
```

Kód 4.1: Příklad nové struktury projektu.

Vzhledem k tomu, že se na práci na I3T podílí i další studenti, je také dobré určit konvence psaní kódu nebo způsob, jak do projektu přispívat. V návaznosti na původní I3T jsme zvolili formátování kódu ve stylu Microsoft s tím rozdílem, že operátory `*`, `&` píšeme nalevo. Všechna pravidla formátování jsou definována v souboru `.clang-format`.

Prvním krokem je vytvoření zcela nového projektu využívajícího CMake. Každý takový projekt musí obsahovat ve svém kořenovém adresáři soubor `CMakeLists.txt`, který definuje hlavní pravidla sestavení [23]. V ukázce kódu 4.2 je znázorněn minimální možný `CMakeLists.txt` pro sestavení I3T.

```
1  project(I3T VERSION 2.1.0)
2
3  set(CMAKE_CXX_STANDARD 20)
4  set(CMAKE_CXX_STANDARD_REQUIRED True
5
6  include_directories(Dependencies/assimp/include)
7  include_directories(Dependencies/glfw/include)
8  include_directories(Dependencies/spdlog/include)
9  include_directories(Dependencies/yaml-cpp/include)
10
11 add_subdirectory(Dependencies/assimp)
12 add_subdirectory(Dependencies/glfw)
13 add_subdirectory(Dependencies/spdlog)
14 add_subdirectory(Dependencies/yaml-cpp)
15
16 add_executable(${PROJECT_NAME} ${I3T_SOURCE})
17
18 target_link_libraries(
19         ${PROJECT_NAME}
20         PUBLIC
21         ${OPENGL_LIBRARIES}
22         assimp
23         glfw
24         spdlog
25         yaml-cpp
26 )
```

Kód 4.2: Jednoduchý `CMakeLists.txt` pro I3T.

Makro `include_directories` specifikuje, v jakých dalších adresářích má CMake hledat hlavičkové soubory. Další makro `add_subdirectory` zahrne hlavní `CMakeLists.txt` nacházející se v adresářích zdrojových souborů vkládaných knihoven. Na řádku 16 se nachází specifikace cíle sestavení. Makro `add_executable` vytvoří cíl s názvem I3T, jehož výstupem je spustitelný binární soubor a pro jehož sestavení je nutné zkompilovat takové soubory, jejichž výčet je určen proměnnou `I3T_SOURCE`. Řádek 18 určuje, s jakými knihovnami je třeba daný cíl propojit. Názvy knihoven (assimp, glfw, ...) odpovídají názvům cílů ze zahrnutých `CMakeLists.txt`.

Správa externích knihoven

Jak již bylo zmíněno v předchozí kapitole, správa závislostí je realizována pomocí submodulů systému pro správu verzí Git [10]. To umožní zahrnout zdrojový kód potřebných knihoven přímo do zdrojového kódu I3T tak, že adresář s kódem jedné knihovny bude nadále vystupovat jako samostatný repozitář. Následující příklady popisují, jak se se submoduly pracuje.

```
$ git submodule add https://github.com/glfw/glfw.git \
    Dependencies/glfw
```

Kód 4.3: Knihovna GLFW jako submodul.

Tento příkaz (ukázka 4.3) naklonuje repozitář knihovny GLFW do specifikované lokace `Dependencies/glfw`. Informace o tom, v jakém adresáři se submodul nachází a jaký je jeho vzdálený repozitář, jsou uloženy do souboru `.gitmodules` v kořenovém adresáři zdrojového kódu I3T. Systém git ignoruje adresáře, jejichž cesty se nacházejí v tomto souboru, a změny, které v nich nastanou, nezahrnuje ke změnám provedeným v hlavním repozitáři.

```
$ git clone --recursive https://git.example.com/repo.git
```

Kód 4.4: Naklonování repozitáře se submoduly.

Při klonování repozitáře využívajícího submoduly je nutno provést `clone` s parametrem `--recursive` (ukázka kódu 4.4). Tento přepínač zajistí, že git spolu s obsahem hlavního repozitáře stáhne také obsah submodulů a navíc rekurzivně stáhne obsah jejich případných dalších submodulů. Pokud bychom na uvedený přepínač zapomněli, adresáře, ve kterých se má nacházet obsah submodulů, budou prázdné. Ty lze naplnit provedením následujícího příkazu (ukázka 4.5).

```
$ git submodule update --init --recursive
```

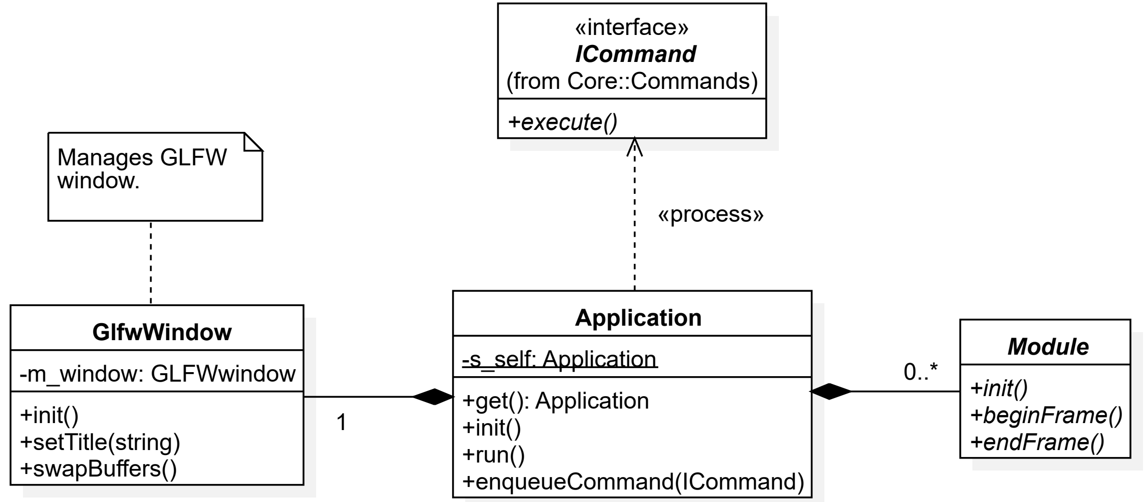
Kód 4.5: Inicializace nenaklonovaných submodulů.

4.2 Core

Aplikace se skládá ze tří komponent – *Core*, *GUI* a *Rendering*. Komponentě *Core* se stará se o klíčové funkcionality, jako je vytvoření okna, zpracování vstupních a dalších událostí a inicializaci ostatních dvou komponent. Ústředním bodem celého *Core* je třída `Application`.

4.2.1 Třída Application

Třída `Application` (viz obr. 4.1) představuje vstupní bod do I3T. Má jen jednu možnou instanci a je inicializována ve funkci `main`. Jejím hlavním smyslem je držet pohromadě jednotlivé části *Core* a spravovat je.



Obrázek 4.1: Diagram třídy `Application`.

Třída se stará o prvotní inicializaci prostředí (funkce `init`). Při inicializaci je vytvořena instance třídy `GlfwWindow`, která má na starosti vytvoření okna a OpenGL kontextu pomocí GLFW (viz podkapitola 3.4 kapitoly 3). Proběhne-li vše bez chyby, zaregistrují se pro okno funkce *(callback)*, které mají být vykonány po vzniku události v okně aplikace. Knihovna GLFW nám poskytuje celou řadu funkcí s jejichž pomocí můžeme reagovat na vstupní akce. Za zmínění stojí funkce `glfwSetKeyCallback`, `glfwSetCursorPosCallback` a `glfwSetMouseButtonCallback`. První funkce umožní reagovat na vstup z klávesnice. GLFW nás informuje prostřednictvím argumentů, se kterými je naše zaregistrovaná funkce zavolána, o tom, jaká klávesa byla stisknuta nebo uvolněna. Ostatní uvedené funkce slouží ke zpracování událostí generovaných myší. Jedná se o informace o pozici kurzoru a o stavu tlačítek myši. Vstupní události jsou zpracovány třídou `InputManager`, která je uvedena v sekci 4.2.3.

```
1 double lastFrameSeconds = 0.0;
2
3 void Application::run()
4 {
5     while (m_shouldRun)
6     {
```

```
7           // Wait for input.
8           glfwWaitEvents();
9
10          // Process commands.
11          for (auto& command : m_commands)
12          {
13            command->execute();
14          }
15          m_commands.clear();
16
17          double current = glfwGetTime();
18          double delta = current - lastFrameSeconds;
19
20          // Update and display.
21          logicUpdate(delta);
22          onDisplay();
23
24          lastFrameSeconds = current;
25      }
26  }
```

Kód 4.6: Kód třídy `Application`.

Třída `Application` obsluhuje hlavní smyčku (znázorněna ukázce kódu 4.6). Ta, dokud GLFW nezachytilo událost zavření okna, neustále běží. Program do ni vstoupí po zavolání funkce `run`. Nejprve jsou zpracovány vstupní události funkce `glfwWaitEvents`, ta uspí hlavní vlákno aplikace a probudí ho ve chvíli, kdy GLFW obdrželo od okenního subsystému vstupní událost. Jedná se o úspornější alternativu k `glfwPollEvents`, která je vhodnější například pro hry, kde je potřeba celou scénu překreslovat nezávisle na vstupních událostech [2, Input guide]. Po obdržení vstupní události GLFW zavolá příslušný registrovaný *callback*. Na zpracování událostí navazuje zpracování příkazů, které se nashromáždily během předchozího snímku (ř. 10–15). Příkazy jsou speciální události specifické pro I3T. Detailům jejich implementace se věnuje podkapitola 4.2.2.

```
1   void Application::logicUpdate(double deltaSec)
2   {
3       InputManager::preUpdate();
4
5       for (auto* module : m_modules)
6           module->update(deltaSec);
7
8       GraphManager::update(deltaSec);
9
10      InputManager::update();
11  }
12
13  void Application::onDisplay()
14  {
15      for (auto* module : m_modules)
16          module->beginFrame();
17
18      m_world->render();
```

```
19
20      for (auto* module : m_modules)
21          module->endFrame();
22
23      m_window->swapBuffers();
24  }
```

Kód 4.7: Kód funkcí `logicUpdate` a `onDisplay`.

Po zpracování všech událostí následuje na ř. 21 v ukázce 4.6 logický *update* (jeho implementace, viz ukázka 4.7). Ten mění stav namapovaných kláves a aktualizuje jednotlivé zapojené moduly. Po logickém *update* následuje samotné vykreslení. To je rozděleno do tří fází. V první fázi jsou všechny moduly informovány o začátku snímku, což může sloužit například k přípravě dat potřebných k vykreslení. Poté následuje vykreslení scény, kterou představuje proměnná `m_world`. Nyní je možné provést takové operace, které je třeba vykonat ke konci snímku. Jedná se především o vykreslení uživatelského rozhraní realizovaného modulem `UIModule`.

4.2.2 Příkazy

Příkazy jsou speciální typy událostí, které mohou být zpracovány asynchronně. Jejich účelem je umožnit libovolně mnoha částem I3T, aniž by o sobě navzájem věděly, reagovat na určité typy akcí, jako je například zavření okna. Ukázka kódu 4.8 ilustruje způsob vytvoření definice příkazu. Každý konkrétní příkaz lze definovat jako novou třídu, která je odvozena od šablony třídy `Command`. Do jejích parametrů stačí předat název našeho konkrétního příkazu a typy dat, která může příkaz předat.

Technika, kdy je název odvozené třídy parametrem šablony třídy základní, se v C++ nazývá Curiously Recurring Template Pattern (CRTP) [12]. Třída `Command` obsahuje statické členské funkce `addListener`, `dispatch` a statickou členskou proměnnou `s_listeners` (obr. 4.2 na str. 25, podtržené položky). Mohlo by se zdát, že budou v celém programu existovat právě jednou. Použití techniky CRTP ale zajišťuje vygenerování výše uvedených funkcí a proměnné pro každý jeden specifický typ příkazu.

```
1 class BeforeCloseCommand : public Command<BeforeCloseCommand>
2 {};
3
4 class ConsoleCommand : public Command<ConsoleCommand, std::string>
5 {};
```

Kód 4.8: Definice konkrétních příkazů.

Samotné používání příkazů je uvedeno v následující ukázce kódu (kód 4.9). Ta ilustruje zaregistrování funkce, která má být zavolána, je-li příkaz `BeforeCloseCommand` vykonán. Tímto způsobem je možné k jednomu příkazu přidružit více funkcí. Příkaz je vložen do fronty příkazů zavoláním funkce `BeforeCloseCommand::dispatch`, jak je uvedeno v těle funkce `GlfwWindow::init`. To má za následek to, že je aplikaci předán příkaz spolu se svými parametry do fronty příkazů a ty jsou v tomto nebo v příštím snímku vykonány, jak bylo uvedeno v ukázce kódu 4.6.[1]

[1]Pořadí vykonání ovlivňuje to, zda příkaz byl přidán do fronty před fází zpracování všech příkazů, nebo až

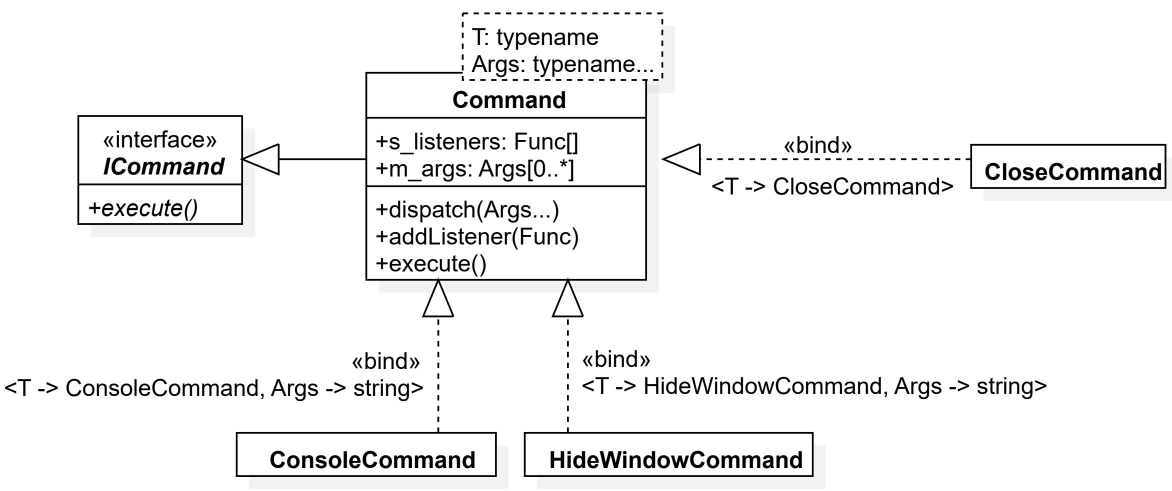
```
void Application::init()
{
    BeforeCloseCommand::addListener(
        std::bind(&App::onBeforeClose, this)
    );
}

void GlfwWindow::init()
{
    // ...

    // Execute command on window close event.
    glfwSetWindowCloseCallback(m_mainWindow,
        [](GLFWwindow* window) {
            BeforeCloseCommand::dispatch();
    });
}
```

Kód 4.9: Používání příkazů.

Instance konkrétních typů příkazů se liší ve své velikosti, a není je tedy možné uložit do jednoho pole. Proto třída `Application` pracuje s ukazatelem na základní abstraktní třídu `ICommand` (obr. 4.2), která definuje nejmenší nutné rozhraní pro provedení příkazu. Třída `ICommand` obsahuje čistě virtuální funkci `execute`, kterou implementuje šablona třídy `Command<T, Args...>`. Její implementace se stará o zavolání všech registrovaných funkcí (kód 4.10).



Obrázek 4.2: Hierarchie příkazů.

Šablona třídy `Command<T, Args...>` obsahuje veškerou logiku, která je potřebná pro vytváření a odběr příkazů. Jak je z ukázky kódu 4.10 patrné, třída se stará prostřednictvím funkce `addListener` (ř. 15) o registrování funkcí, které mají být zavolány, je-li příkaz vykonán (*callback*). Třída má také na starosti předání kopie příkazu třídě `Application` spolu s argumenty funkce `dispatch` (ř. 25) uloženými v členské proměnné `m_args`, jejímž zavoláním byl příkaz vytvořen a předán do fronty příkazů instanci třídy `Application`.

poté.

Ta, jak již bylo uvedeno (4.2.1), zavoláním virtuální funkce `execute` v hlavní smyčce jeden příkaz po druhém vykoná.

```
template <typename Type, typename... Args>
class Command : public ICommand
{
public:
    using Callback = std::function<void(Args&...)>;

    Command() = default;

protected:
    explicit Command(const std::tuple<Args...>& args)
        : m_args(args) {}

    virtual ~Command() = default;

    static void addListener(Callback function);

    void execute() override
    {
        for (Callback callback : s_listeners)
            std::apply([callback](auto... args) {
                callback(args...);
            }, m_args);
    }

    static void dispatch(Args... args)
    {
        std::tuple<Args...> m_args(args...);
        App::get().enqueueCommand(
            new Command<Type, Args...>(m_args)
        );
    }

private:
    static std::vector<Callback> s_listeners;
    std::tuple<Args...> m_args;
};

template <typename Type, typename... Args>
std::vector<Command::Callback>
Command<Type, Args...>::s_listeners;
```

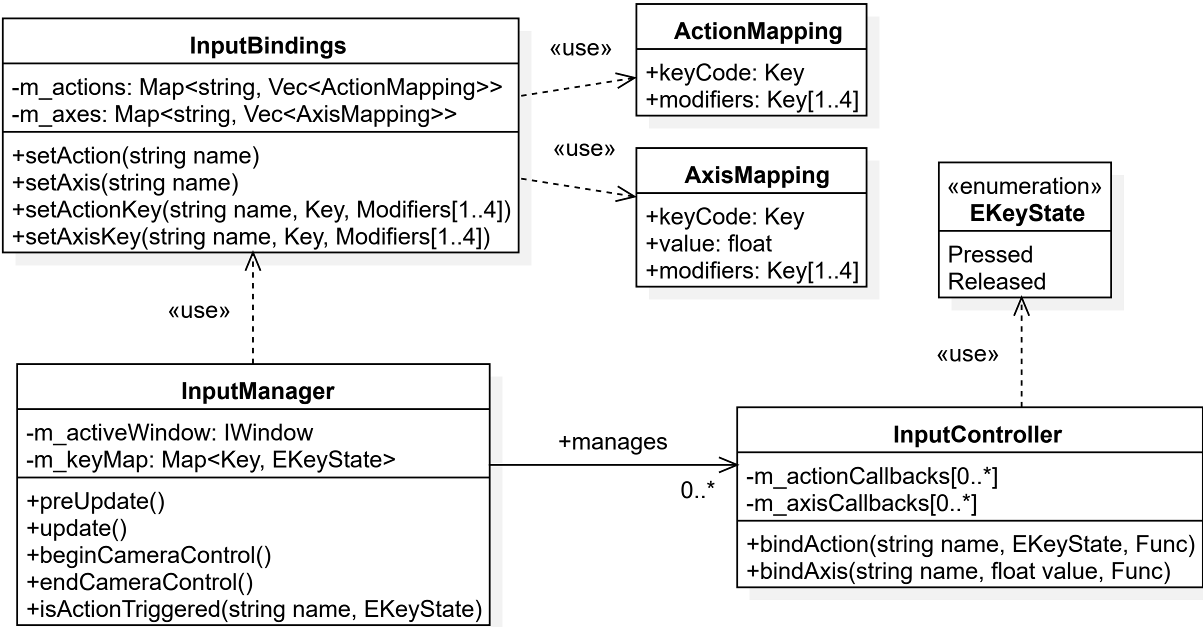
Kód 4.10: Kód třídy `Command`.

4.2.3 Obsluha vstupu

V původním I3T zpracovávala vstup výhradně třída `InputManager`. Ta zachycovala ve svých datových strukturách stavy jednotlivých kláves a tlačítek myši. Každá klávesa či tlačítko mohlo nabývat čtyř stavů – *JUST_DOWN*, *DOWN*, *JUST_UP* a *UP*. Tyto stavy byly v každém snímku modifikovány ve funkci `preUpdate`, která měnila stav *JUST_UP* na *UP* a

stejnou operaci prováděla i se stavem *JUST_DOWN*. V novém návrhu převzala zpracování vstupních akcí knihovna Dear ImGui. Ta se stará o obsluhu všech prvků UI.

Ne všechny události jsou ale přímo spojené s prvky UI. Jedná se například o ovládání kamery v 3D scéně. Pro potřeby zpracování vlastních vstupní akcí, Dear ImGui umožňuje se dotazovat na stavy jednotlivých kláves. Psaní takových dotazů do kódu uživatelského rozhraní může být být nepřehledné. Dále tento přístup není příliš vhodný, pokud chceme podporovat změnu kláves pro jednotlivé akce bez nutnosti rekompilace kódu.



Obrázek 4.3: UML diagram tříd zodpovědných za obsluhu vlastního uživatelského vstupu v I3T.

Novým přístupem je zavedení centrální struktury, která umožňuje dané vstupní akci určené unikátním jménem přiřadit libovolné klávesy včetně specifických kombinací kláves *ctrl*, *alt* nebo *shift*. Při implementaci jsem vycházel z toho, jak funguje obsluha vstupu ve hře tvořené pomocí Unreal Engine. Pro hru je možné definovat dva typy vstupu – *ActionMappings* a *AxisMappings* [13].

Mapování *ActionMapping* slouží k definování ovládání jednorázové akce, jakou může být například otevření inventáře. Chceme, aby akce byla vykonána pouze jednou po dobu stisknutí klávesy. Varianta *AxisMapping*, kterou Unreal Engine definuje, přestavuje kontinuální akci, jako je například pohyb hráče, kdy chceme, aby pozice hráče byla měněna po celou dobu stisknutí dané klávesy. Takové mapování přiřazuje dané akci uživatelem definovanou hodnotu a klávesu či jejich kombinaci. Můžeme tak jedné akci typu *AxisMappings* s námi zadaným jménem *MoveForward*, přiřadit například dvě následující mapování. Jedno na klávesu *W* s hodnotou `1.0f`, které pohne s hráčem vpřed, a druhé na klávesu *S* s hodnotou `-1.0f`, které pohne hráčem vzad. Posledním krokem je v C++ kódu hry vytvořit funkci, kupříkladu `APlayer::move(float)`, a pomocí volání

```
InputComponent->BindAxis(
    "MoveForward", this, &APlayer::MoveForward
);
```

ji zaregistrovat ke zpracování. Díky nadefinování specifických hodnot pro různé klávesy můžeme pro pohyb vpřed i vzad použít stejnou funkci.

V případě I3T (obr. 4.3 na straně 27) můžeme jednotlivá mapování vytvořit díky třídě `InputBindings` a to za použití funkcí `setAction(const char* name)` nebo `setAxis(const char* name)` a za použití funkcí `setActionKey` a `setAxisKey` přiřadit k jednotlivým akcím, příslušné klávesy. Spolu s hlavní klávesou je možné specifikovat i klávesy modifikační. Klávesy jednotlivých akcí je možné za běhu měnit a tím umožnit jejich konfiguraci například v okně nastavení I3T.

```
1  void InputBindings::init()
2  {
3      setAction("Fire");
4      setActionKey("Fire", Keys::t, { Keys::ctrll });
5
6      setAxis("MouseScroll");
7      setAxisKey("MouseScroll", 1.0f, Keys::mouseScrlUp);
8      setAxisKey("MouseScroll", -1.0f, Keys::mouseScrlDown);
9  }
10
11 // ...
12
13 void onScroll(float val)
14 {
15     // Do scroll ...
16 }
17
18 // InputController Window::Input;
19 Window::Window()
20 {
21     Input.bindAction("Fire", EKeyState::Pressed, []()
22     {
23         Log::info("Action fired.");
24     });
25
26     Input.bindAxis("MouseScroll", onScroll)
27 }
```

Kód 4.11: Příklad registrace vstupní akce v okně.

Pro registrování funkcí pro dané vstupní akce slouží třída `InputController`. Použití jejích funkcí je vyobrazeno v ukázce kódu 4.11. Instanci třídy `InputController` obsahuje každé okno. Reference na všechny takové třídy jsou drženy v poli třídy `InputManager`. Ta zajišťuje zpracování vstupních akcí jen pro okno, které je aktivní.

4.2.4 Moduly

Část *Core* dále poskytuje třídu `Module`. Ta představuje rozhraní jakékoliv nezávislé jednotky I3T. Díky třídě `Module` mohou programátoři aplikaci snadno rozšířit o další funkcionality. Třída `UIModule`, která slouží jako kontejner pro jednotlivá okna, je odvozená od této třídy (viz podkapitola 4.5).

```
1  class Module
2  {
3  public:
4      Module() {};
5      virtual ~Module() {}
6      virtual void init() {}
7      virtual void update(double deltaSec) {}
8      virtual void beginFrame() {}
9      virtual void endFrame() {}
10 };
```

Kód 4.12: Třída `Module`.

Třída `Module` zajišťuje redukci závislosti mezi jednotlivými objekty i jinými částmi kódu. Bude-li nějaký vývojář chtít I3T v budoucnu rozšířit o nějakou funkcionalitu, postačí, když vytvoří vlastní implementaci třídy `Module` a tu prostřednictvím funkce `Application ::addModule` do I3T zapojí. Při zapojení bude automaticky provolána funkce `init` a poté periodicky se začátkem a koncem každého snímku funkce `beginFrame` a `endFrame`.

Vlastní implementaci třídy `Module` je teoreticky možné skrýt do samostatné dynamicky linkované knihovny a tu pak použít jako jednoduše distribuovatelný *plug-in*. Ten bude I3T moci snadno za běhu automaticky načíst do paměti a uvést ho do provozu.

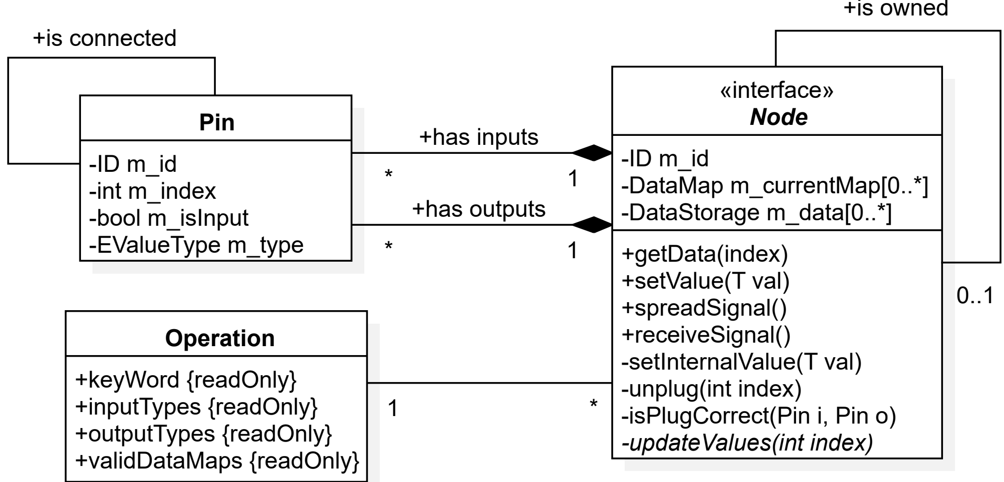
4.3 Datová reprezentace krabiček

Hlavním účelem aplikace I3T je výuka transformací a jejich hierarchie ve 3D grafice. Samotnou reprezentaci jednotlivých matematických operací obsahuje adresář *Nodes*. Každá taková operace představuje uzel orientovaného grafu. Michal Folta tyto uzly nazval krabičkami [15, s. 26]. Jak jsem již v úvodu zmínil, budu se držet jeho názvosloví.

I3T obsahuje několik typů krabiček. Nejdůležitějšími jsou transformace. Ty jsou reprezentovány jako čtvercové matice řádu čtyři. Jedná se především o rotace, translace, změny měřítka nebo projekce. Transformace lze interaktivně editovat a tím pozorovat jejich vliv na objekty ve scéně. Transformace je možné skládat do sekvencí a jejich propojením reprezentovat graf scény. Dalším typem krabiček jsou operátory. Ty slouží jako pomocné krabičky pro provádění různých matematických operací, které se mohou hodit při práci s transformacemi. Jedná se například o inverzi matice, transpozici, běžné operace s vektory, sestavení matic dle parametrů a podobně. Pro všechny krabičky je společné, že mohou mít 0 až N vstupů a výstupů.

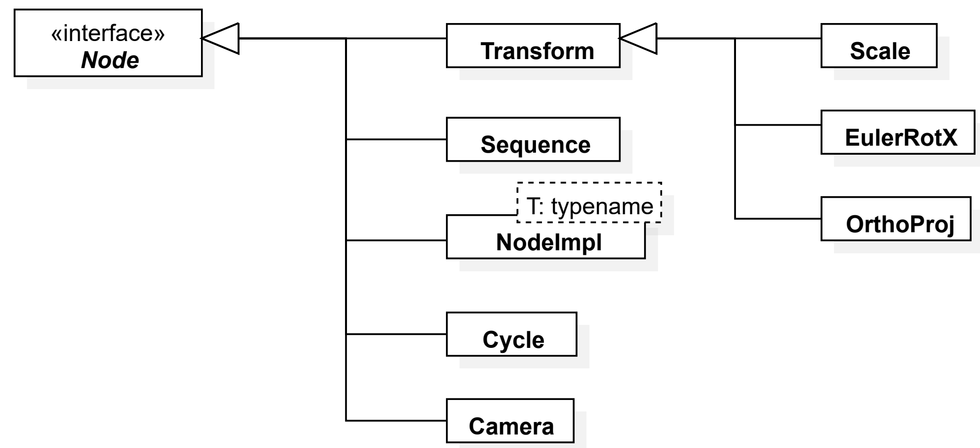
Každý vstup i výstup má pevně určený typ hodnoty, kterou může konzumovat nebo poskytnout. Propojením vstupu a výstupu vzniká v grafu krabiček orientovaná hrana, která má za důsledek to, že z výstupu jedné krabičky doputuje hodnota, kterou výstup poskytuje, do vstupu té druhé. Vstup druhé krabičky hodnotu konzumuje. S takto obdrženou hodnotou krabička provede své operace a jejich výsledek poskytne ostatním zapojeným krabičkám na svém příslušném výstupu. Detaily procesu propojování a přenášení hodnot v grafu popisuje následující text.

4.3.1 Třída `Node`



Obrázek 4.4: Detail třídy Node.

Každá krabička je implementací třídy `Node`. Ta tvoří základní datovou reprezentaci uzlu grafu. Všechny druhy krabiček jsou od ní odvozeny, neboť je pro ně společné, že mají vstupní a výstupní piny a mohou obsahovat nějaká data. O vlastní zpracování vstupních hodnot se stará funkce `updateValues`, která je virtuální a každá odvozená krabička ji



Obrázek 4.5: Hierarchie typů krabiček. Šablona třídy `NodeImpl<T>` představuje operátory a krabičky *Model*, *Pulse* a *Screen*. Výčet transformací je zde omezen jen na tři typy. Úplný seznam všech krabiček transformací je uveden v tabulce 4.1.

implementuje. Do vstupu krabičky může být zapojen nejvýše jeden výstup. Nelze do něj zapojit libovolně mnoho výstupů, protože by nebylo možné jednoznačně určit, ze kterého výstupu má krabička číst hodnoty. Výstupní hodnotu jednoho pinu může číst libovolně mnoho krabiček.

Propojení jsou realizována skrze třídu `Pin`. Ta obsahuje informace o tom, jakého typu daný vstup nebo výstup je, jaké má pořadí v rámci všech vstupů nebo výstupů krabičky a do jakých jiných pinů je zapojen. Přes spojené piny je možné libovolně procházet celou souvislou komponentu grafu.

Vlastní hodnoty krabičky jsou uloženy ve struktuře `DataStorage`. Práce s různými typy dat je reprezentována pomocí typu `std::variant` ze standardu jazyka C++17 [29]. Například, proměnné typu `std::variant<float, vec3>` můžeme přiřadit pouze jednu z hodnot typu uvedeného v parametru šablony. Jedná se tedy o typově bezpečný `union`, který garantuje, že nebude obsahovat jiný typ hodnoty, než který je přesně specifikován při inicializaci. Do této datové struktury je v I3T možné uložit jedno desetinné číslo, tří nebo čtyř složkový vektor, matici nebo ukazatel na `void`. Díky použití `std::variant` struktura `DataStorage` v paměti programu zabírá blok o velikost matice obsahující šestnáct třiceti dvou bitových desetinných čísel typu IEEE 754 `float`.

Každá krabička obsahuje zpravidla tolik úložišť, kolik má výstupů. Výjimku tvoří krabičky, které nemají žádný výstup (všechny transformace, *Model* a *Screen*). Jakého typu úložiště, vstupy a výstupy pro daný typ krabičky budou, se rozhodne na základě vlastnosti *Operation* (definované v souboru `Core/Nodes/Operations.h`). Stejnojmenná struktura obsahuje záznamy o názvu krabičky, jmenovky vstupů a výstupů, a především informace o tom, jaké typy hodnot vstupy či výstupy přijímají. Pro každý typ krabičky existuje záznam o jejích vlastnostech, podle kterého je vytvořena.

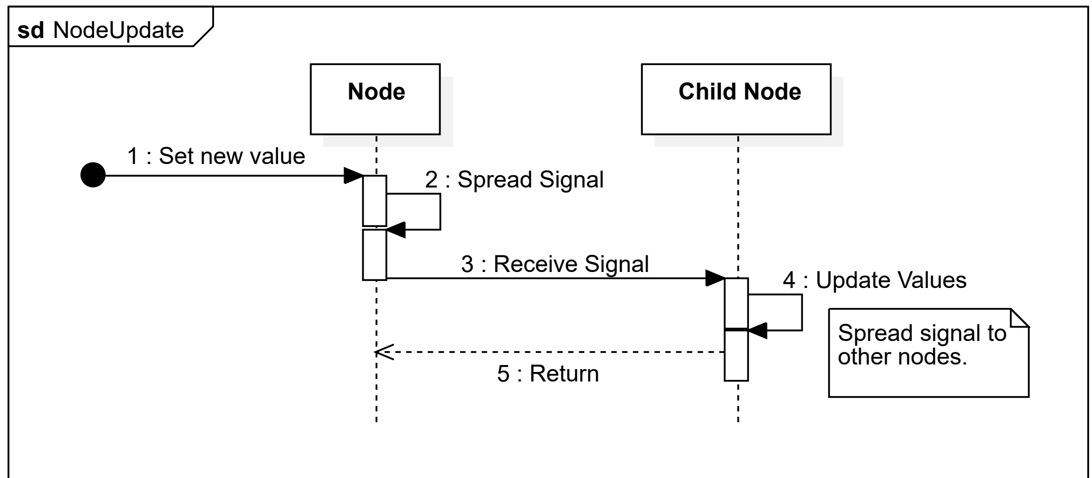
Jak je z diagramu 4.4 patrné, třída `Node` poskytuje rozhraní pro správu svých hodnot. Funkce `setInternalValue` má na starosti zapsání hodnoty do příslušného úložiště určeného jeho indexem. Tato funkce ale není dostupná veřejně (modifikátor `protected`). Pro uživatelské nastavení hodnoty slouží funkce `setValue`. Každá krabička, které lze přiřadit hodnotu zvenčí, garantuje, že v případě předání nesprávné hodnoty nebude její hodnota

změněna. Každý typ odvozený od třídy `Node` (viz obr. 4.5) může funkci `setValue` přepsat a zavést svůj způsob validace nastavovaných hodnot. To zajistí např. platné hodnoty v matici, pokud někdo edituje jednotlivé hodnoty (synergie). Například krabička `EulerRotX`, která představuje rotaci kolem osy x o úhel α, má hodnotu ve tvaru čtvercové matice uvedené ve vztahu 4.1.

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.1}$$

Vidíme zde jasná omezení. Nastavitelné mohou být pouze prvky r_{22}, r_{23}, r_{32} a r_{33}, a to tak, že mohou nabývat pouze hodnot z intervalu $\langle -1, 1 \rangle$. Validace se týká především transformací. To, jakým způsobem jsou nastavované hodnoty validovány, je vysvětleno v sekci věnované transformacím (4.3.3).

Pokud byla hodnota uzlu úspěšně nastavena, je třeba o její změně informovat všechny krabičky (děti), které jsou připojené na její výstupy. Přepočítání jejich hodnot probíhá rekurzivně a má podobu prohledávání grafu do hloubky. Celý proces probíhá tak, že při nastavení hodnoty krabička sama na sobě zavolá funkci `spreadSignal`, které předá index úložiště, jehož hodnota byla nově nastavena. Tato funkce projde seznam všech krabiček připojených do příslušného výstupu (označeného indexem) a zavolá na nich funkci `receiveSignal`, které předá informaci, jakého vstupního pinu se změna týká. Zapojená krabička díky tomu pozná, odkud změna pochází. To je užitečné především tehdy, má-li více vstupů. Poté následuje samotný výpočet nové hodnoty pomocí funkce `updateValues`. Pokud tato změna nějak ovlivní interní hodnotu krabičky, celý proces se opakuje, dokud nenarazíme na krabičku, která nemá žádný výstup. Postup je přehledně ilustrován v sekvenčním diagramu 4.6. Tento postup navrhl Michal Folta [15, s. 29] a nebylo na něm třeba nic měnit.



Obrázek 4.6: Popis průběhu aktualizace hodnot v grafu.

4.3.2 Operátor

Operátory jsou základním typem krabiček. Slouží jako jednoduché funkce, které zpracují vstupní hodnotu a provedou s ní nějaký výpočet. Můžeme si uvést například operátor *Transpose*, který přijímá do vstupu libovolnou čtvercovou matici řádu čtyři. Pokud má vstup zapojen, na svém výstupu poskytuje ke čtení matici transponovanou. Pokud ho zapojený nemá, nabízí výchozí hodnotu, v tomto případě jednotkovou matici.

I3T obsahuje celkem osmdesát sedm operátorů, není účelné je zde všechny uvádět. Co ale za zmínku stojí, je způsob jejich implementace. Marek Nechanský ve své práci zmínil, že pro každý operátor mělo I3T speciální třídu [30, s. 32]. V mnoha třídách se kód opakoval, a jediné, co bylo pro každý operátor odlišné, byla funkce `updateValues`. Proto navrhl nový postup. Vytvořil šablonu třídy `OperatorNode`, která byla odvozená od třídy `Operator` (dnešní `Node`). Pro každý typ operace vytvořil záznam ve výčtovém typu `OperationType`. Každý takový záznam sloužil jako specializace šablony, pro kterou speciálně vytvořil implementaci funkce `updateValues`. Tohoto přístupu jsem se držel i já, jen jsem mírně pozměnil názvy typů. Kód 4.13 obsahuje deklaraci třídy `NodeImpl`, která představuje operátor, a specializaci funkce `updateValues` pro typ operátoru *Transpose*.

```
1  template <ENodeType T>
2  class NodeImpl : public Node
3  {
4  public:
5      ~NodeImpl() override = default;
6      void updateValues(int inputIndex) override;
7  }
8
9  template <>
10 inline void
11 NodeImpl<ENodeType::Transpose>::updateValues(int inputIndex)
12 {
13     if (m_inputs[0].isPluggedIn())
14     {
15         setInternalValue(
16             glm::transpose(m_inputs[0].getStorage().getMat4())
17         );
18     }
19     else
20     {
21         setInternalValue(glm::mat4(1.0f));
22     }
23 }
```

Kód 4.13: Kód operátorů. Implementace operátorů a dalších jednoduchých krabiček s příkladem kódu funkce `updateValues` operátoru *Transpose*.

4.3.3 Transformace

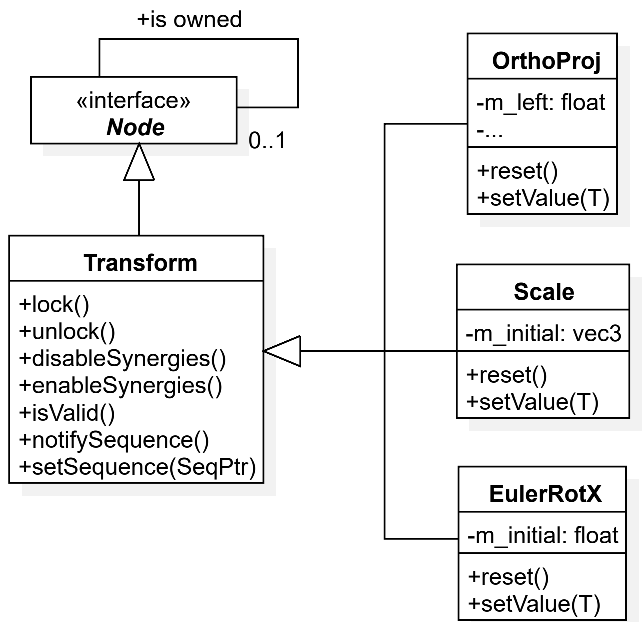
Transformace jsou speciálním typem krabiček, které nemají žádné vstupy nebo výstupy. Mohou obsahovat jedinou hodnotu, a to čtvercovou matici řádu čtyři. Tento typ krabiček slouží výhradně k umisťování do sekvencí. I3T disponuje šesti typy transformací v různých provedeních (viz tabulka 4.1).

Tabulka 4.1: Všechny typy transformací, které I3T nabízí.

Typ transformace	Krabička	Poznámka
Volná transformace	*Free*	
Translace	*Translate*	o vektor $\mathbf{v} = (v_1, v_2, v_3, 1)$
Škálování	*Scale*	vektorem $\mathbf{v} = (v_1, v_2, v_3)$
	UniformScale	ve všech směrech stejné
Rotace	*EulerAngleX*	kolem osy x o úhel α
	EulerAngleY	kolem osy y o úhel α
	EulerAngleZ	kolem osy z o úhel α
	AxisAngle	kolem obecné osy o úhel α
	Quat	pomocí kvaternionu
Pohledová transformace	*LookAt*	
Projekce	*Ortho*	
	Perspective	
	Frustum	

Transformace se od ostatních typů krabiček liší také tím, že podporují několik režimů nastavení hodnoty. Ve výchozím stavu má matice transformace zamčené takové hodnoty, které nedává smysl v rámci daného typu transformace nastavovat. Kupříkladu máme-li krabičku translace, ve výchozím stavu ji lze nastavit jen první tři hodnoty v posledním sloupci. Toto chování lze obejít odemknutím transformace. Tímto se dostaneme do stavu volné transformace a je možné změnit libovolnou hodnotu v matici. Pokud tak učiníme, krabička transformace se dostane do nevalidního stavu. Z takového stavu se snadno můžeme přepnout zpět pomocí zavolání funkce `reset`, která nastaví krabičce poslední validní hodnotu.

Jak je z diagramu na obr. 4.7 patrné, společným rozhraním pro všechny transformace je třída `Transform`. Ta je odvozena od společného rozhraní všech krabiček `Node`. Třída `Transform` navíc obohacuje běžnou krabičku o funkce zajišťující zamykání a odmykání hodnot na vyžádání (`lock` a `unlock`) a o funkce, které vypínají a zapínají synergie (`enableSynergies` a `disableSynergies`), tj. dopočítávání další hodnot matice tak, aby byla konsistentní se svou definicí.

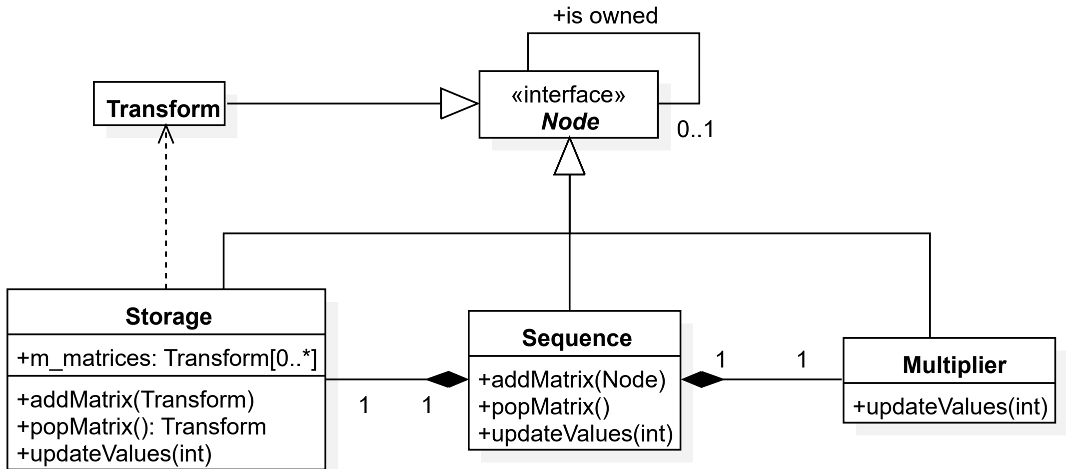


Obrázek 4.7: Detail implementace vybraných transformací.

Jednotlivé transformace přetěžují funkce `setValue`, pro které zavádějí validaci nastavovaných hodnot nebo jejich přepočet v případě provázaných hodnot při zapnutých synergiích. Pro transformace je dále typické, že v sobě drží poslední validní hodnotu, kterou je možné pomocí funkce `reset` v případě porušení správnosti transformace znovu nastavit.

4.3.4 Sekvence

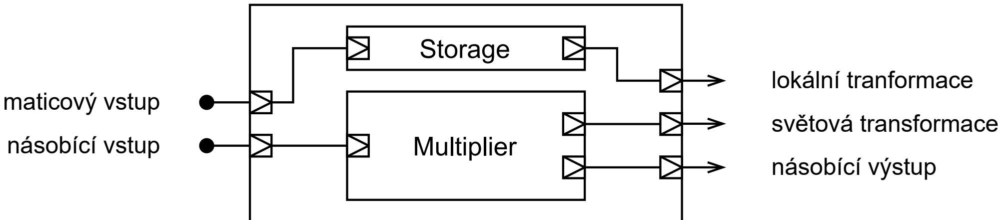
Při vytváření scény je nutné transformace spolu skládat, tedy matice, kterými jsou popsány, mezi sebou násobit. K tomuto účelu slouží krabička sekvence. Ta umožňuje do sebe vkládat krabičky s transformacemi a jednoduše měnit jejich pořadí. Propojováním sekvencí uživatel tvoří graf scény.



Obrázek 4.8: Detail třídy `Sequence`.

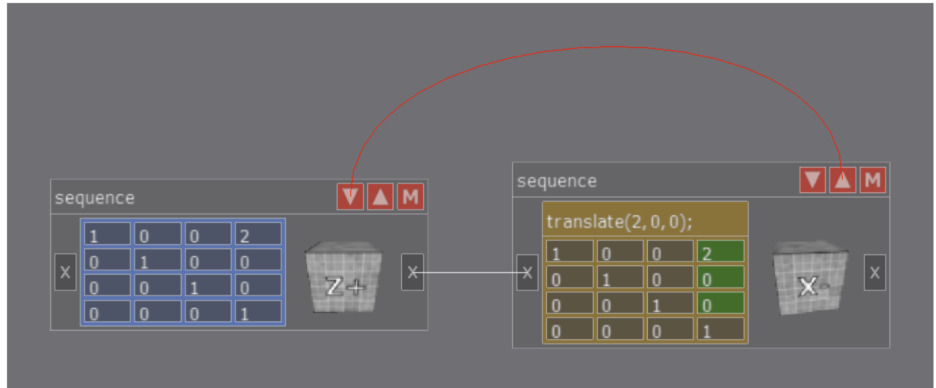
V původní verzi I3T sekvence měla pouze čistě grafickou reprezentaci v podobě třídy `TransformationForm`. Jednalo se o speciální typ, který neměl mnoho společného s ostatními krabičkami. V novém I3T je sekvence implementací třídy `Node` (viz obr. 4.8), díky čemuž poskytuje úplně stejné rozhraní jako všechny operátory nebo transformace.

Třída `Sequence` slouží jako jakási fasáda pro skrytí vnitřní logiky fungování sekvence transformací. Stejně jako v I3T Michala Folty, i v nové verzi se sekvence skládá ze dvou uzlů grafu (viz obr. 4.9). Prvním uzlem je struktura `Storage`. Jejím hlavním úkolem je v sobě držet jednotlivé transformace. Vždy, když se nějaké z nich změní hodnota, provede se jejich přenásobení. Komponenta `Storage` má jeden vstup a jeden výstup. Na svém výstupu poskytuje výslednou matici transformace vzniklou složením matic, které v sobě drží. Je-li do jejího vstupu zapojena krabička poskytující na svém výstupu matici, transformace, které `Storage` v sobě drží, jsou ignorovány, a místo nich je použita matice ze vstupu.



Obrázek 4.9: Vnitřní struktura sekvence. Komponenty `Storage` a `Multiplier` nejsou spojeny, `Multiplier` vyčítá hodnoty lokálních transformací skrze svého vlastníka – sekvenci (ta obsahuje referenci na `Storage`).

Druhou komponentou sekvence je `Multiplier`. Jedná se o strukturu, jejímž hlavním účelem je obsluha násobících vstupů a výstupů. Má v sobě uloženou hodnotu, která odpovídá složení všech transformací v grafu scény od kořene do tohoto uzlu. Je-li zapojen násobící vstup sekvence, komponenta `Multiplier` složí transformace všech svých předchůdců (vyčte je ze sekvence zapojené do násobícího vstupu) se svými lokálními transformacemi, jejichž složení vyčte z výstupu komponenty `Storage`, a výslednou hodnotu si uloží.



Obrázek 4.10: Zdánlivě cyklické zapojení dvou sekvencí.

Díky uvedeným skutečnostem je možné sekvence zapojit tak, aby vznikl zdánlivý cyklus (obr. 4.10). Protože se ale každá sekvence skládá ze dvou navzájem nepropojitelných komponent (obr. 4.9), cyklus takovýmto zapojením ve skutečnosti nevznikne.

4.3.5 Ostatní krabičky

Je třeba zde uvést zbývající krabičky. V podkapitole věnované operátorům (4.3.2) je zmíněno, že všechny operátory jsou specializací šablony třídy `NodeImpl<T>`. To se týká i posledních krabiček I3T – *Model*, *Pulse* a *Screen*. Jsou až na jeden rozdíl totožné s operátory – nemají předlohu ve funkcích knihovny GLM.

4.3.6 Správa grafu krabiček

Třída `GraphManager` poskytuje programátorům rozhraní pro vytváření a manipulaci s krabičkami. Umožňuje je propojovat a rozpojovat (viz tabulka 4.2). Dále je také možné se jejím prostřednictvím dotazovat, zda libovolná krabička má v rámci celého grafu nějaké předchůdce nebo následníky, tedy je-li její vstup zapojen do výstupu jiné krabičky a naopak.

Tabulka 4.2: Rozhraní třídy `GraphManager`.

Specializace	Funkce

Vytváření uzlů	`NodePtr createTransform<T>(Args...)`
	`SequencePtr createSequence()`
	`NodePtr createNode<ENodeType>()`
	`CyclePtr createCycle()`

Správa uzlů	`EResult plug(const NodePtr&, const NodePtr&, from, to)`
	`void unplugAll(const NodePtr&)`
	`void unplugInput(const NodePtr&, int index)`
	`void unplugOutput(const NodePtr&, int index)`

Dotazování	`EResult isPlugCorrect(const Pin* in, const Pin* out)`
	`NodePtr getParent(const NodePtr& node, int index)`
	`vec<NodePtr> getOutputNodes(const NodePtr&, int index)`
	`vec<NodePtr> getAllInputNodes(const NodePtr&)`
	`vec<NodePtr> getAllOutputNodes(const NodePtr&)`

Velmi důležitá je funkce `plug`, která zajišťuje propojení vstupu a výstupu dvou krabiček. To, které jejich piny mají být spojeny, je specifikováno číselnými hodnotami `from` a `to`, které určují jejich index v daném uzlu. Propojovací funkce vytvoří potřebné asociace mezi danými krabičkami jen v případě, že je možné je spojit. Jak bylo výše uvedeno, krabičky tvoří orientovaný acyklický graf. Nemůžeme připustit, aby v grafu vznikl cyklus, pokud by se tak stalo, algoritmus 4.6 by nikdy neskončil, protože navštívené uzly nijak neoznačuje. Dále není možné spojit piny různých typů (například *float* do *vec3*) nebo stejných druhů (například vstup do vstupu).

```
1 EResult GraphManager::isPlugCorrect(const Pin* input, const Pin* output)
2 {
3     if (input->getType() != output->getType())
4         return ENodePlugResult::Err_MismatchedPinTypes;
5
6     if (!input->m_isInput || output->m_isInput)
7         return ENodePlugResult::Err_MismatchedPinKind;
```

```
8
9       if (input->m_master == output->m_master)
10          return ENodePlugResult::Err_Loopback;
11
12      auto toFind = input->m_master;
13
14      std::stack<NodePtr> stack;
15      stack.push_back(output->m_master);
16
17      while (!stack.isEmpty())
18      {
19          auto act = stack.pop();
20          if (act == toFind) { return ENodePlugResult::Err_Loop; }
21
22          for (auto& inp : act->inputs)
23          {
24              if (inp->isPluggedIn())
25              {
26                  Pin* ct = pin.m_input;
27                  stack.push_back(ct->m_master);
28              }
29          }
30      }
31
32      return ENodePlugResult::Ok;
33  }
```

Kód 4.14: Mírně modifikovaný původní algoritmus pro detekci přítomnosti cyklu v grafu [15, s. 30].

V ukázce kódu 4.14 je zobrazena mírně modifikovaná kontrola propojení. Funkce `isPlugCorrect` v původní verzi I3T informovala pouze o tom, zda je propojení možné nebo není. To nemusí být dostačující, neexistuje totiž žádný způsob, jak informovat uživatele o tom, proč není propojení možné. Upravená implementace používá stavové kódy ve formě výčtového typu `ENodePlugResult`.

K vytváření krabiček slouží funkce `createTransform`, `createSequence`, `createNode` a `createCycle`. Všechny vrací *chytrý* ukazatel typu `std::shared_ptr` [4, §iso.20.7.2.2] na nově vytvořenou krabičku. *Chytrý* ukazatel spravuje alokovaný zdroj a hodnotu udávající počet použití tohoto zdroje. Vždy, když vytvoříme kopii daného objektu, je počet užití zvýšen o jedna. Zanikne-li objekt typu `std::shared_ptr`, počet se o jedna sníží. Zaniká-li poslední existující objekt odkazující na daný zdroj (má počet používání roven jedné), je zdroj uvolněn. To nám umožní bezpečněji pracovat s pamětí, jelikož se nám nemůže nikdy stát, že není *raw* ukazatel na zdroj nikým vlastněn [39, §13].

Zajímavostí je, že *chytré* ukazatele mají přetížené operátory `operator*` a `operator->`, díky čemuž se s nimi pracuje úplně stejným způsobem jako s klasickými ukazateli. Alternativou k `std::shared_ptr` by mohl být `std::unique_ptr`, který může v danou chvíli mít jen jednoho vlastníka. Nelze kopírovat, lze pouze přesouvat. Tento typ ukazatele nezavádí žádné počítání referencí a je tedy, co se režie týče, stejně efektivní jako klasický ukazatel. Práce s ním je ale z důvodů nutnosti přesouvání mnohem složitější, proto jsem zvolil typ ukazatele `std::shared_ptr`.

4.3.7 Přidání nového typu krabičky

Z pohledu kódu existují tři typy krabiček: a) jednoduché krabičky, které jsou odvozeny od třídy `NodeImpl<T>`, u nichž lze modifikovat pouze chování funkce `updateValues`, b) transformace, které umožňující validaci nastavovaných hodnot – ty jsou odvozené od třídy `Transform`, c) ostatní krabičky. Těmi se rozumí speciální typy krabiček, které mají velmi specifické chování. Jedná se například o krabičky sekvence, kamera nebo cyklus.

Pokud chceme I3T rozšířit o jednoduchou krabičku nebo transformaci, je třeba provést následující kroky:

1. **Definovat typ operace.**

   V souboru `Core/Nodes/Operations.h` je nutné vytvořit položku ve výčtovém typu `ENodeType` nebo `ETransformType`, která bude určovat pořadí definice operace v seznamu `g_operations` nebo `g_transforms`. Záznam musí být v následujícím tvaru:

   ```
   {"NodeKeyword", "NodeLabel", numberOfInputs, inputTypes,
       numberOfOutputs, outputTypes},
   ```

2. **Definovat chování nové krabičky.**

   Pokud definujeme novou jednoduchou krabičku, je třeba doplnit specializaci funkce `NodeImpl<T>::updateValues` (soubor `Core/Nodes/NodeImpl.h`) pro daný typ operace určený v předchozím kroku definovaném `ENodeType::NewOpName`. Implementace funkce `updateValues` třídy `NodeImpl<ENodeType::NewOpName>` může vypadat následovně:

   ```
   template <> FORCE_INLINE void
   NodeImpl<ENodeType::NewOpName>::updateValues(int inputIndex)
   {
       if (m_inputs[0].isPluggedIn())
       {
           auto newValue = glm::transpose(
               m_inputs[0].getStorage(inputIndex).getMat4()
           );
           setInternalValue(newValue);
       }
   }
   ```

   Pokud definujeme novou transformaci, je třeba vytvořit novou třídu odvozenou od třídy `Transformation` (soubor `Core/Nodes/Transform.h`). Nové takové třídě můžeme přepsat funkce `setValue`, `isValid` a `reset`.

3. **Vytvořit grafickou reprezentaci nové krabičky.**

   Aby mohla být krabička zobrazena, je třeba vytvořit její grafickou reprezentaci. Toto ale není předmětem této práce a je také možné, že kód grafické reprezentace krabiček, který tvoří další členové týmu, bude v budoucnu značně modifikován, proto není možné zde uvést jasný postup. Dostatečnou inspirací k tomu, jak postupovat, by mohly být soubory v adresáři `GUI/Elements/Nodes`. Jakmile má krabička svou grafickou reprezentaci, lze ji přidat do nabídky v okně *Workspace* (viz soubor `GUI/Elements/Windows/WorkspaceWindow.h`) a vytvořit její instanci.

4.3.8 Jednotkové testy

K otestování funkčnosti krabiček, zejména nastavování hodnot a jejich propojování, slouží sada jednotkových testů umístěných v adresáři `Test`. Nové I3T vzniká pod rukama několika vývojářů. Jednotkové testy jsou vhodné především pro otestování funkcionalit, které ve fázi vývoje nemůžeme otestovat prostým spuštěním aplikace. Pro psaní testů byl zvolen framework Google Test [20].

```
1 // Tests factorial of positive numbers.
2 TEST(FactorialTest, HandlesPositiveInput) {
3   EXPECT_EQ(Factorial(1), 1);
4   EXPECT_EQ(Factorial(2), 2);
5   EXPECT_EQ(Factorial(3), 6);
6   EXPECT_EQ(Factorial(8), 40320);
7 }
```

Kód 4.15: Podoba jednoduchého jednotkového testu (převzato z `https://google.github.io/googletest/primer.html`).

Ukázka 4.15 ilustruje, jak vypadá jednotkový test psaný za použití frameworku Google Test. Makro `TEST` vygeneruje třídu, která má název složený ze *test suite name*, což je první parametr makra – `FactorialTest`, a *test case name*, tedy `HandlesPositiveNumber`. Veškerý kód, který píšeme za makro do bloku ohraničeného znaky `{}`, je tělo testu. Google Test následně veškeré definované testy automaticky spustí.

Tento framework umožňuje definovat vlastní třídu odvozenou od třídy `testing::Test` . Té lze přepsat členské funkce `SetUp` a `TearDown` a tím dosáhnout nastavení testovacího prostředí a následného úklidu. Název třídy lze následně předat jako první parametr makru `TEST` a díky tomu proběhne test v definovaném prostředí.

```
 1 TEST(TranslationTest, InvalidValue_ShouldNotBePermitted)
 2 {
 3   auto translationNode = Builder::createTransform<Translation>();
 4
 5   // invalid coordinates
 6   auto result = translationNode->setValue(generateFloat(), {0, 3});
 7
 8   EXPECT_EQ(
 9     EStatus::Err_ConstraintViolation, result.status
10   );
11   EXPECT_EQ(ETransformState::Valid, translationNode->isValid());
12 }
```

Kód 4.16: Jednotkový test v I3T. Test testuje, zda nastavení nevalidní hodnoty bude zakázáno.

I3T obsahuje 85 jednotkových testů, jejich počet se v budoucnu může zvýšit. Není třeba zde uvádět přesný výčet všech testů, vše lze velmi dobře dohledat v kódu. Testy testují správnost propojování a odpojování krabiček, správnost nastavování hodnot, tedy zda krabička po nastavení hodnoty provede správně výpočet, který má provést, nebo zda krabička nepřijme nevalidní hodnotu. Pro transformace je také typickou vlastností provázanost jednotlivých hodnot v matici transformace. Například pro uniformní škálování chceme, aby se při nastavení jedné hodnoty z prvních tří hodnot na hlavní diagonále matice mě-

nily i hodnoty zbývající. Toto chování je u příslušných transformací (uniformní škálování a rotace kolem všech os otestováno.

Také je důležité otestovat chování sekvencí. Ty umožňují přidávat a odebírat transformace a provádět jejich interpolace. Jsou testovány i další krabičky, například cyklus, který obsahuje velké množství funkcí měnících jeho stav (`play`, `stop` a další).

V kódu 4.16 je vyobrazen jednoduchý test, který testuje případ, kdy nastavení nevalidní hodnoty na pozici v matici $(0, 3)$ (první sloupec, čtvrtý řádek) nebude krabičkou představující translaci povoleno, a nedojde tedy ke změně hodnot matice, tudíž krabička zůstane ve validním stavu.

4.4 CI/CD

Pro potřeby automatizovaného otestování přispívaného kódu jsme se rozhodli využít platformu Gitlab, která nám již slouží ke správě verzí kódu aplikace. Gitlab nám umožňuje definovat jednotlivé kroky sestavení a příslušné akce, které mají být v rámci kroku provedeny.

```
1  image: gcc
2  before_script:
3    - apt update && apt -y install cmake git libgl-dev libxrandr-dev
       libxinerama-dev libxcursor-dev libxi-dev libopengl0
4
5  stages:
6    - build
7    - test
8
9  build:
10   stage: build
11   script:
12     - mkdir out && cd out
13     - cmake ..
14     - cmake --build . -j 8 --target I3T
15     - cmake --build . -j 8 --target I3TTest
16   artifacts:
17     paths:
18       - Binaries
19     expire_in: 1 week
20   cache:
21     paths:
22       - Binaries
23   variables:
24     GIT_SUBMODULE_STRATEGY: recursive
25
26 # run tests using the binary built before
27 test:
28   stage: test
29   script:
30     - Binaries/Debug/I3TTest
```

Kód 4.17: CI/CD skript pro I3T.

V ukázce kódu 4.17 je vyobrazena konfigurace CI pro I3T. Všechny kroky skriptu jsou vykonávány v Docker kontejneru izolovaně od operačního systému. První řádka skriptu určuje, jaký kontejner má být pro CI použit. My jsme použili již připravený kontejner dostupný na webu *Docker Hub* obsahující GCC *toolchain* [6]. Na dalším řádku se nachází definice akcí, které mají být provedeny před zahájením CI. Pro sestavení I3T je nutné nainstalovat nástoje CMake a Git a knihovny pro práci s OpenGL a okny v prostředí Linuxu (X Window System). Dále následuje definice jednotlivých kroků CI. Nejprve je nutné definovat, z jakých kroků se celý proces bude skládat (blok `stages`), a následně definovat akce těchto kroků. Proces CI pro I3T se skládá z kroků `build` a `test`.

V kroku `build` je celé I3T spolu s jednotkovými testy sestaveno pomocí CMake. Výsledné soubory sestavení nacházející se v adresáři `Binaries` (jedná se o slinkované spustitelné soubory a binární soubory knihoven) jsou k dispozici ke stažení po dobu jednoho týdne. Pravidlem `cache` můžeme určit, které soubory mají být uchovány pro další kroky CI. Všechny výsledné binární soubory (adresář `Binaries`) uchováme pro krok `test`.

Pokud jsou I3T i jednotkové testy úspěšně sestaveny, následuje krok `test`. V něm jsou všechny testy, které byly sestavené v předchozím kroku, spuštěny. Proběhnou-li všechny úspěšně, je proces CI zdárně dokončen a změny mohou být bezpečně integrovány. Stane-li se, že nějaký krok neproběhne bezchybně, Gitlab prostřednictvím e-mailu informuje uživatele, který nahrál změnu do repozitáře, o nezdaru.

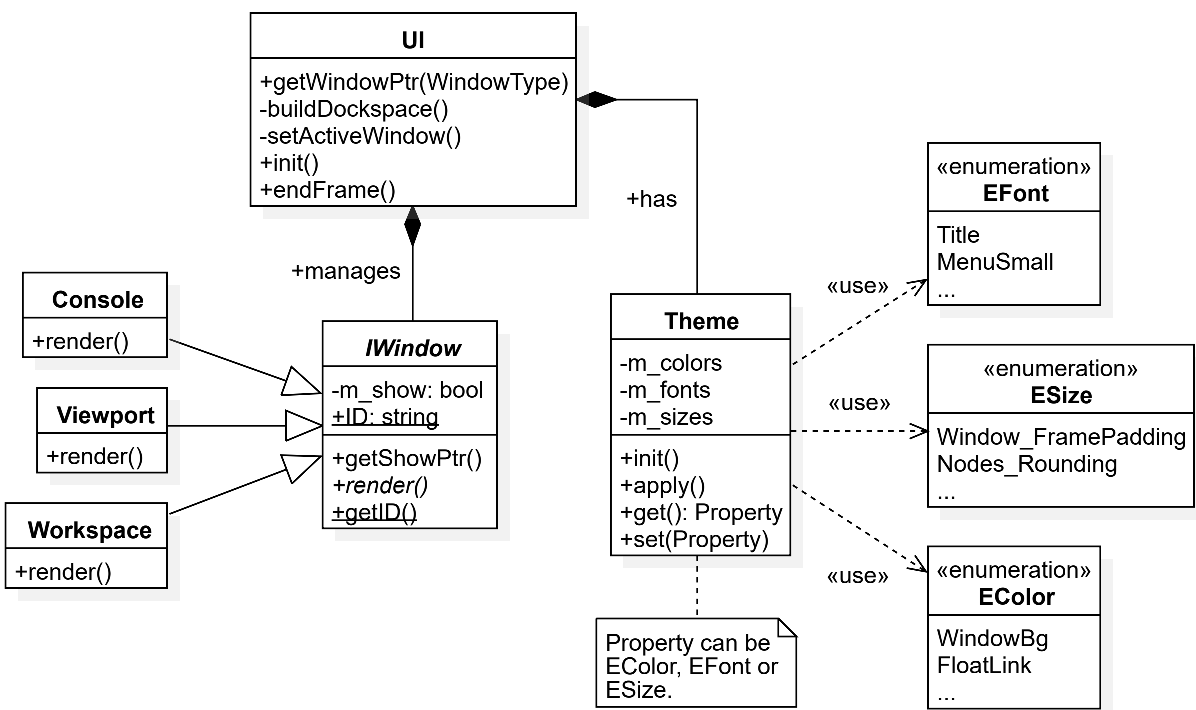
4.5 Uživatelské rozhraní

Uživatelské rozhraní aplikace je tvořeno za pomoci knihoven GLFW a Dear ImGui. Knihovna GLFW se stará o inicializaci OpenGL kontextu a správu oken. Umožňuje nám snadným způsobem zaregistrovat funkce, které jsou volány po provedení nějaké vstupní akce, jako je například stisknutí klávesy či myši nebo pohyb kurzoru myši. Knihovna Dear ImGui se stará o vyváření a obsluhu prvků uživatelského rozhraní.

4.5.1 Napojení Dear ImGui na vstup okna

Uživatelské rozhraní bude vytvořeno za pomoci knihovny ImGui. Aby vše dobře fungovalo, je třeba propojit okno, ve kterém chceme zachytávat události, s knihovnu ImGui. Knihovně GLFW je možné předat ukazatele na funkce, které mohou reagovat na vstupní akce. ImGui není pevně vázaná na žádný okenní systém, proto jejich napojení musíme realizovat sami.

Jelikož jsme se rozhodli využít možnosti dokovat okna, není snadné připojení ImGui na vstup realizovat ručně vlastním kódem. Je třeba do hloubky pochopit, jak GLFW funguje, včetně oblastí, které většina uživatelů nevyužije, jako je například vykreslování do více oken. Pro uživatele, kteří nepotřebují kód ušitý na míru, poskytuje adresář `backends` v repozitáři knihovny příklady propojení s různými knihovnami. Pro naše účely jsme použili soubory `backends/imgui_impl_glfw.[cpp|h]` a `backends/imgui_impl_opengl3.[cpp|h]` [32].



Obrázek 4.11: UML diagram modulu UI.

4.5.2 UI modul

Třída `UIModule` je implementací třídy `Module`. Slouží jako kontejner pro jednotlivá okna, jejichž kód implementuje rozhraní `IWindow`. `UIModule` také spravuje všechny načtené styly (obr. 4.11). Třída se stará o vykreslování prvků uživatelského rozhraní, v každém snímku vystaví nové uživatelské rozhraní a provolá funkce `render` všech oken. Ty obsahují kód, který vykreslí specifický obsah jednotlivých oken.

V průběhu práce s knihovnou ImGui se ukázalo, že ImGui učiní okno aktivním, jen pokud do něj bylo kliknuto levým tlačítkem myši. To se nám ale jevilo jako nepraktické, neboť by bylo možné se po 3D scéně rozhlížet až po kliknutí levým tlačítkem, což je neintuitivní. Rozhodli jsme tedy, že aktivní bude nejsvrchnější okno, nad kterým se nachází kurzor myši. O nastavení aktivního okna se stará funkce `setActiveWindow`. Spolu se změnou aktivního okna se také aktivuje `InputController` nového aktivního okna.

4.5.3 Node editor

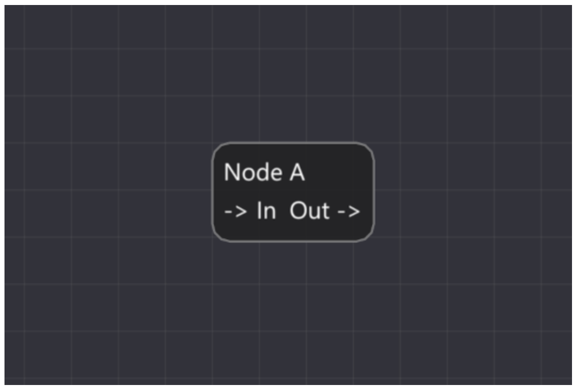
Základní uživatelské rozhraní je tvořeno pomocí knihovny ImGui. Co ale knihovna neposkytuje, je vytváření uzlů a krabiček v node editoru. ImGui je snadno rozšířitelná a je možné s její pomocí vytvořit vlastní prvky UI. Abychom ale neřešili něco, co někdo určitě již vyřešil před námi, sáhli jsme po existující knihovně ImGui Node Editor [11], která nám umožní snadné vytváření a obsluhu jednotlivých krabiček.

```
1  static ed::EditorContext* g_Context;
2
3  void render()
4  {
5      ed::SetCurrentEditor(g_Context);
6      ed::Begin("My Node Editor", ImVec2(0.0, 0.0f));
7
8      int uniqueId = 1;
9
10     ed::BeginNode(uniqueId++); // Start drawing node.
11         ImGui::Text("Node A");
12         ed::BeginPin(uniqueId++, ed::PinKind::Input);
13             ImGui::Text("-> In");
14         ed::EndPin();
15         ImGui::SameLine();
16         ed::BeginPin(uniqueId++, ed::PinKind::Output);
17             ImGui::Text("Out ->");
18         ed::EndPin();
19     ed::EndNode(); // End drawing node.
20
21     ed::End();
22 }
```

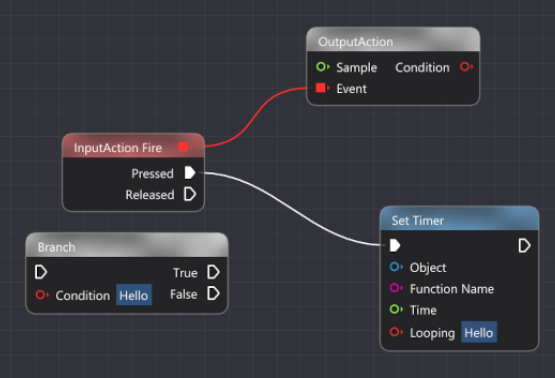
Kód 4.18: Ukázka vytvoření uzlu s pomocí knihovny ImGui Node Editor (převzato z příkladů knihovny *imgui-node-editor* [11]).

Kód 4.18 popisuje průběh vykreslení jednoduchého uzlu s jedním vstupem a jedním výstupem. Výsledkem tohoto kódu je jednoduchá krabička, která je vyobrazena na obr. 4.12a.

Použitá knihovna sama zajišťuje interakci s uživatelem. Krabičky lze vybírat pomocí kliknutí nebo pomocí obdélníkového výběru. K výběru lze další krabičku přidat pomocí kliknutí myši spolu s klávesou ctrl. S vybranými krabičkami lze volně pohybovat po celém prostoru *node* editoru.



(a) Výsledek jednoduchého příkladu.

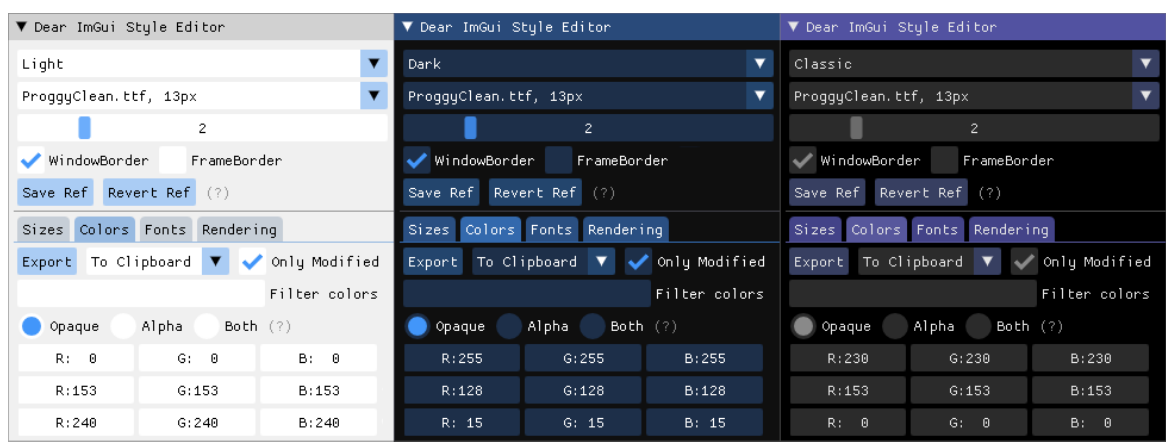


(b) Knihovna umožňuje vytvořit krabičky ve stylu *bluprints* z Unreal Editoru.

Obrázek 4.12: Příklady uzlů node editoru (převzato z příkladů knihovny *imgui-node-editor* [11]).

4.5.4 Vlastní styly

I přes svou jednoduchost je knihovna ImGui velmi dobře konfigurovatelná. Přichází se třemi zabudovanými barevnými styly (obr. 4.13), které lze jednoduše modifikovat pomocí ImGui Style Editoru obsaženém v ImGui Demo Window. Ten umožňuje měnit jak nastavení barev oken, tlačítek a dalších prvků, tak různé velikosti výplní a ohraničení elementů GUI. Vlastní modifikovanou paletu barev a velikostí lze pomocí tlačítka Export exportovat do C/C++ kódu, který příslušné vlastnosti nastaví. ImGui neumožňuje tyto vlastnosti uložit nebo načíst ze souboru.

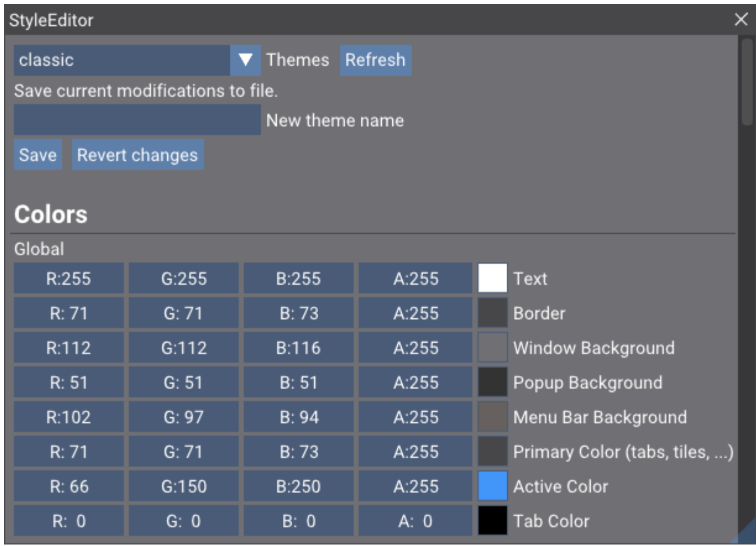


Obrázek 4.13: Výchozí styly poskytované knihovnou Dear ImGui (upravený Dear ImGui Style Editor).

Ne všechny vlastnosti spojené se vzhledem lze nastavit a aplikovat globálně. Často může nastat případ, že například v jednom okně potřebujeme mít jinou barvu pozadí než v tom druhém. Pro lokální aplikování jistých vlastností stylu knihovna ImGui poskytuje

funkce `ImGui::PushStyleVar`, `ImGui::PopStyleVar`, které aplikují určitou vlastnost určenou výčtovým typem ImGuiStyleVar a příslušnou hodnotou pouze v bloku kódu ohraničeném voláními `Push` a `Pop`.

Pro zapouzdření vlastností specifických pro náš projekt slouží třída `Theme`. V ní jsou uloženy všechny hodnoty barev, velikostí a použitá písma (viz obr. 4.11). Všechny tyto vlastnoti jsou indexované pomocí výčtových typů `EColor`, `EFont` a `ESize`. Hodnoty lze číst a nastavovat pomocí funkcí `get*` a `set*`. Pro snadnější dotazování se na hodnoty příslušných vlastností slouží funkce `I3T::getColor`, `I3T::getFont` a `I3T::getSize`, které delegují dotaz na aktivní styl spravovaný třídou `UIModule`.



Obrázek 4.14: Editor stylu I3T.

I3T obsahuje dva výchozí styly. Jedná se o styl *Classic* (obr. A.1), který má předlohu v původní podobě I3T Michala Folty, a styl *Modern* (obr. A.2). Ten je odvozen od grafického návrhu Michala Pilky [35] a Marka Nechanského [30]. Pro účely snadných úprav výchozích stylů byl vytvořen vlastní editor stylů (obr. 4.14), který umožňuje nastavit vlastnosti jednotlivých oken, jako jsou barva pozadí, barva ohraničení, barva hlavní nabídky a kontextových menu; a vlastnosti spojené s krabičkami, tedy velikost políček pro číselné hodnoty, jejich barvy, barvy vstupů a výstupů pro příslušné typy hodnot. Uživatelům je umožněno pomocí tohoto editoru vytvořit libovolný vlastní styl, který mohou uložit do souboru, ze kterého je při spuštění aplikace načten.

Kapitola 5

Diskuze

I3T je užitečný nástroj usnadňující výuku transformací v 3D grafice. Budoucím cílem zdokonalování nástroje je publikování jeho kódu prostřednictvím známé platformy Github. Aby bylo možné tento záměr uskutečnit, je třeba ještě zkvalitnit kód a vylepšit uživatelské rozhraní. Následující podkapitoly shrnou možná vylepšení, která by dopomohla zdokonalit kvalitu kódu I3T nebo nástroj samotný.

5.1 Kvalita kódu

Prvním důležitým krokem je sjednotit C/C++ kód. Ne všichni členové týmu byli dostatečně ochotni zbavit se svých zvyklostí zápisu kódu a dodržovat ve prospěch projektu smluvené konvence tak, aby všechen kód vypadal jednotně. Lze to jednoduše vyřešit naformátováním souborů s C++ kódem pomocí nástroje `clang-format`. Neshoda se ale netýká pouze formátování.

C++ je poměrně starý jazyk, který má mnoho podobného s jazykem C, ten se dá chápat jako jeho podmnožina [37, s. 21]. Jazyk C++ je ale neustále zdokonalován, přibývají nové funkcionality do STL i nové funkcionality jazyka samotného. V praxi se často setkáme s tím, že různé projekty obsahují kód psaný v různých stylech. Například knihovna ImGui je psána v mezích standardu jazyka C++98 a její autor se víceméně vyhýbá STL[1]. Na druhou stranu jiné projekty využívají *moderního C++*, pak hovoříme o standardech jazyka C++11 a novějších. Aby se šířila osvěta mezi programátory, v průběhu vývoje jazyka C++ vznikají nová doporučení (C++ Core Guidelines), která dávají návod, jak jazyk co nejlépe používat [38]. Pro potřeby našeho projektu je nejdůležitější shodnout se na verzi standardu jazyka, protože nelze, aby kód jednoho programu byl psán nejednotně.

Z výše uvedených poznatků plyne do budoucna doporučení, že je třeba provádět důkladnější kontrolu přispívaného kódu a více vyžadovat dodržování smluvených konvencí o podobě a stylu kódu. Dojde-li navíc k zveřejnění kódu, bude umožněno komukoliv, kdo bude mít zájem, do projektu přispět. V tu chvíli bude ještě důležitější řídit se smluvenými pravidly.

[1] Například namísto `std::string` používá `const char*` a další funkce z jazyka C (například pro práci s řetězci – `strlen`, `strcpy`).

Pro další zkvalitnění kódu I3T by bylo vhodné otestovat i další části aplikace. Poměrně snadné by bylo otestovat část, která se týká skriptování nebo tutoriálů. Užitečné by bylo také navrhnout komplexní testovací scénáře průchodem aplikací.

Protože C++ je komplexní jazyk, který není snadné rychle ovládnout, nezamezíme některým chybám jen použitím testů. Je nutné vyvarovat se dalších chyb, jako je například špatná práce s pamětí, kterou jednoduše pomocí jednotkového testu neodhalíme. Proto by bylo vhodné rozšířit proces *Continuous Integration* o další krok, a to o analýzu správné práce s pamětí pomocí nástroje Valgrind.

5.2 Použitelnost

Použitelnost a přívětivost nového uživatelského rozhraní tvořeného s pomocí knihovny Dear ImGui by mohla být zdokonalena nezávislým uživatelským testováním. Dalším rozšířením by také mohla být podpora jazykových mutací, která by umožnila nástroj rozšířit mezi jiné uživatele, než jsou studenti vysokých škol nebo univerzit.

Pokud by byl jazyk UI modifikovatelný za běhu, mohli bychom o I3T hovořit jako o internacionalizovaném software (i18n). Toho by bylo možné dosáhnout překladem zobrazovaných zpráv. Každá informace by měla svůj unikátní identifikátor využívaný v kódu a pomocí prostředníka (např. třída `Locale`), který by našel překlad v slovníku příslušného jazyka, by byla přeložena. Jednotlivé překlady by mohly být například umístěny v souborech, jejichž název by odpovídal kódu jazyka dle normy ISO 639-1.

```
{
  hello: "Hello!"
}
```

```
{
  hello: "Ahoj!"
}
```

```
{
  hello: "Salut!"
}
```

Kód 5.1: Hypotetické slovníky ve formátu JSON pro jednotlivé jazykové mutace. Vlevo anglická mutace (`en.json`), uprostřed česká mutace (`cs.json`) a vpravo mutace francouzská (`fr.json`).

Takové soubory by mohly být umístěny například v adresáři `data/i18n` a po startu aplikace by jejich obsah byl načten do operační paměti. Aby prostředník, který překlad realizuje, věděl, jakou jazykovou verzi použít, je nutné zjistit, jaký jazyk využívá operační systém.[2] To lze realizovat následujícím způsobem:

```
#include <locale>

void init()
{
  auto localeName = std::locale("").name(); // cs_CZ.UTF8

  auto localeId = std::string(
    localeName.begin(), localeName.begin() + 2
  ); // cs

  Locale::setLocale(localeId);
}
```

[2]Výchozí nastavení pro první použití aplikace. Kód vlastní rozdílné jazykové mutace by mohl být po nastavení uložen v konfiguračním souboru.

Poté, co je aplikace inicializovaná a víme, jaký jazyk operační systém používá, můžeme překládat zobrazované zprávy. Následující kód zobrazí okno s přeloženým textem *Ahoj!* v titulku a v těle okna:

```
void showHello()
{
  ImGui::Begin(Locale::translate("hello"));  // window title
  ImGui::Text(Locale::translate("hello"));   // text
  ImGui::End();
}
```

5.3 Webová verze

I přes to, že pro první použití I3T není vyžadována instalace – stačí, když je adresář spolu se spustitelným souborem (`I3T.exe`), sdílenými knihovnami a daty extrahován do uživatelem oblíbené lokace – mohl by se způsob distribuce programu o něco zjednodušit. Existuje totiž technologie WebAssembly, která umožňuje programům napsaným v kompilovaných jazycích, běžet v prostředí webového prohlížeče. Stačí je jen zkompiloval speciálním kompilátorem, jehož výstupem bude binární soubor obsahující instrukce ve formátu WASM [5]. I3T by pak mohlo být „rychle po ruce" například v případě, pokud by bylo nutné okamžitě demonstrovat nějakou operaci na počítači, který I3T nemá připravené k užívání.

Existence webové verze by mohla také usnadnit sdílení výukových tutoriálů a scén. Uživatelé by mohli své scény sdílet na webu podobně, jako je možné sdílet *shadery* na webu *Shadertoy* [3].

5.4 Sjednocení změn

Na předchozím vývoji I3T se podílelo více studentů. Některé námi provedené změny ale odstranily jejich příspěvky. Kvůli použití nové technologie pro realizaci grafické podoby krabiček (knihovna *imgui-node-editor*) bylo zcela vyřazeno automatické rozmisťování krabiček od Marka Nechanského [30]. Dále kvůli modifikaci zpracování vstupních událostí není zatím plně funkční zapisování interakce uživatele s aplikací od Filipa Uhlíka [44] (na příslušných místech chybí volání logovacích maker, viz soubor `Logger/LoggerInternal.h`). V budoucnu je nutné tyto nedostatky odstranit.

Kapitola 6

Závěr

Nástroj I3T je unikátní aplikací, která umožňuje interaktivním způsobem získat vhled do fungování geometrických transformací užívaných nejen v 3D grafice. V průběhu několikaletého používání nástroje se objevilo několik nedostatků. Bylo nutné především vylepšit jeho uživatelské rozhraní, ale také ho připravit na snadné rozšíření o další funkcionality. Stávající kód se pro tyto změny jevil jako nepružný. Proto bylo nutné navrhnout novou architekturu aplikace s ohledem na principy OOP. Což především znamenalo oddělit kód logiky aplikace od kódu uživatelského rozhraní a upravit kód uživatelského rozhraní tak, aby namísto vlastního vykreslování prvků UI byla použita knihovna Dear ImGui takovým způsobem, aby bylo možné uživatelsky přívětivě přizpůsobit vzhled aplikace. Od aplikace se dále požadovalo, aby byl její kód přenositelný na jiné platformy.

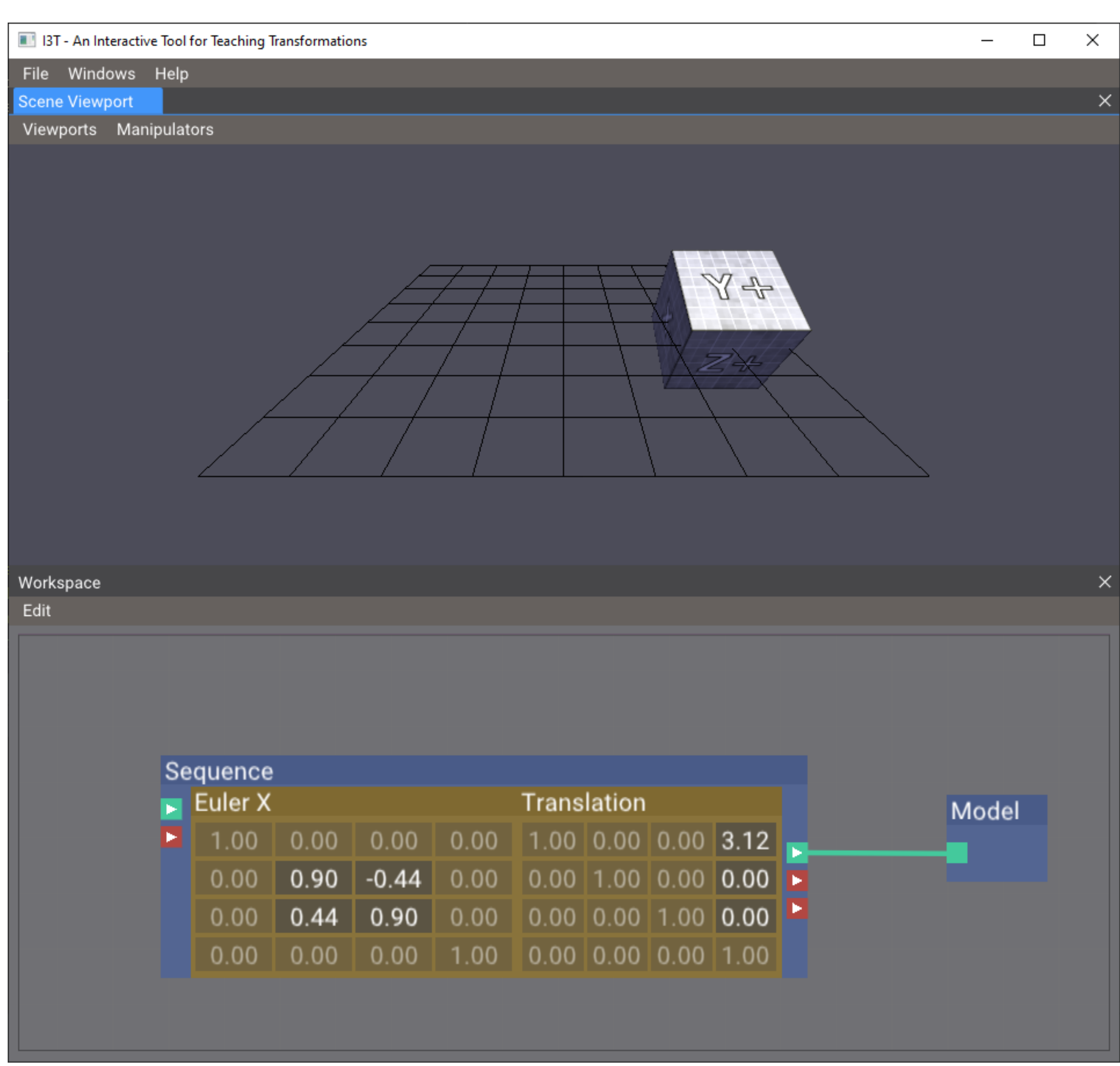
Samotným výstupem této práce je nové I3T. Ve Foltově I3T [15] byly krabičky, pomocí kterých uživatel nastavuje transformace objektům ve scéně, odvozeny od grafického uživatelského rozhraní. Tuto původní reprezentaci krabiček nahradila zcela nezávislá na způsobu jejich zobrazení. Výhodou oproti původnímu řešení je také to, že všechny krabičky mají jednotné rozhraní. Veškeré jejich funkcionality, jako je propojování a nastavování hodnot, jsou otestovány pomocí jednotkových testů. Celé prostředí GUI je rozděleno do samostatných dokovatelných oken a díky použití knihovny *imgui-node-editor* lze jednoduše vytvářet grafickou reprezentaci krabiček. Všechny vstupní akce je možné na jednom místě v kódu konfigurovat. Vzhled prvků uživatelského rozhraní je snadno přizpůsobitelný pomocí vlastního editoru stylů. V něm je možné upravit barvy a rozměry prvků UI a jejich výplní.

Mým příspěvkem do projektu bylo tedy poskytnutí základů, na kterých se dá I3T dále rozvíjet. Prostředí aplikace dotváří další členové týmu, kteří mají na starosti vytvoření grafické reprezentace krabiček, podporu adaptivních výukových tutoriálů a manipulaci s objekty v 3D scéně.

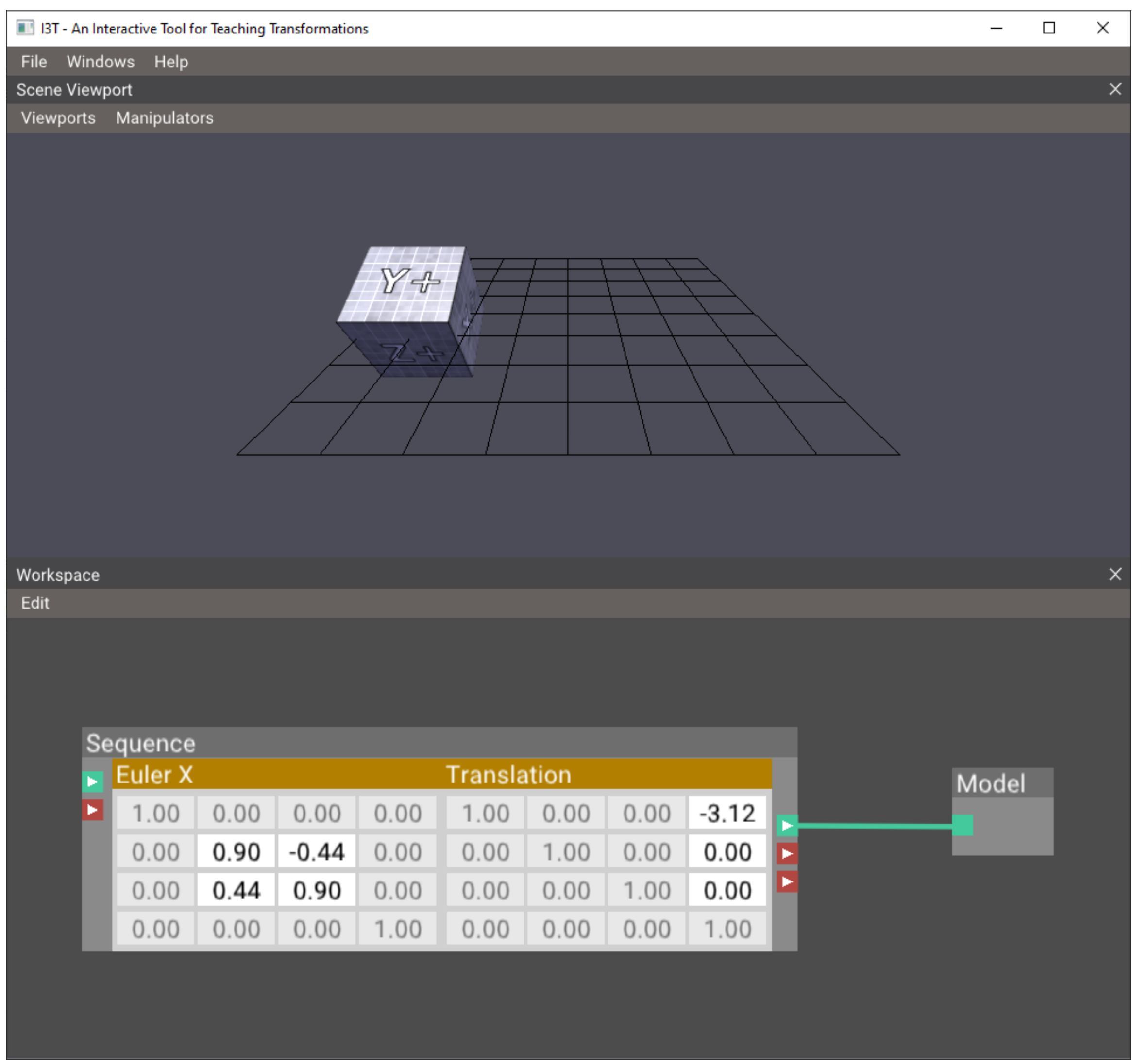
Věříme, že naše společné úsilí pomůže ke zkvalitnění nástroje a k jeho snadnějšímu rozšíření nejen mezi studenty počítačové grafiky.

Příloha A

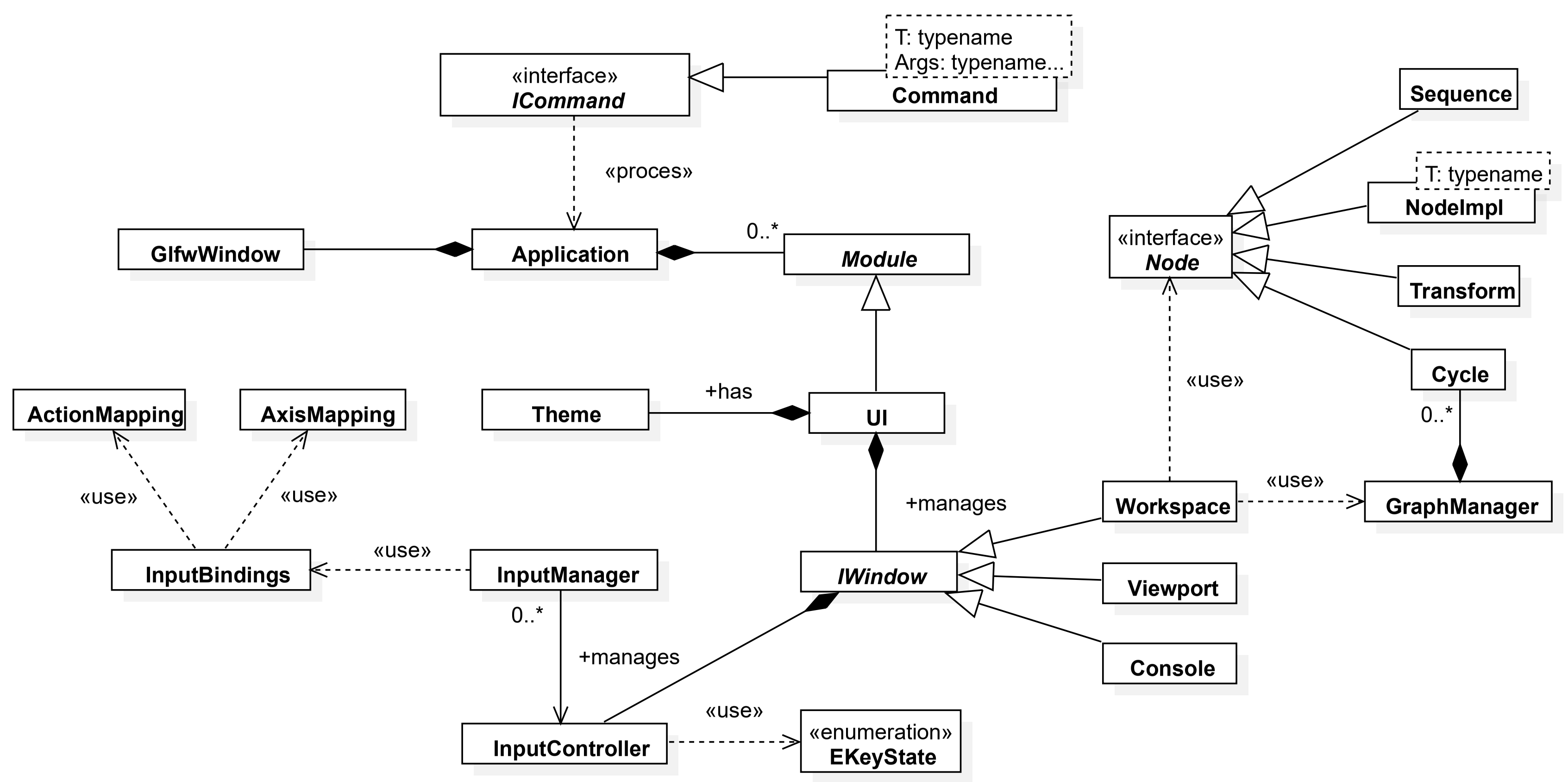
Obrázky a diagramy



Obrázek A.1: Uživatelské rozhraní ve stylu *Classic*, který je založen na původní podobně I3T.



Obrázek A.2: Uživatelské rozhraní ve stylu *Modern*, který je založen na návrzích Lukáše Pilky a Víta Zadiny [35, 45].



Obrázek A.3: Diagram hlavních tříd aplikace. Vyobrazeny jdou zde pouze názvy jednotlivých tříd a jejich vzájemné vztahy. Detailní popis tříd poskytují diagramy v kapitole 4.

Příloha B

Přiložené soubory

Přiložené CD obsahuje spustitelnou verzi programu (adresář `bin`), zdrojový kód I3T (adresář `src`) a LaTeX kód tohoto textu (adresář `thesis`), ze kterého je možné pomocí programu **`pdfLaTeX`** vygenerovat PDF soubor. Podrobnější informace o obsahu jednotlivých adresářů a o sestavení I3T obsahuje soubor `README.md`.

```
.
|-- bin
|-- src
|-- thesis
‘-- README.md
```

Kód B.1: Obsah přiloženého CD.

Kód I3T je dostupný na fakultním Gitlabu na adrese `https://gitlab.fel.cvut.cz/i3t-diplomky/i3t-bunny`. Repozitář je neveřejný, přístupný pouze na vyžádání.

Obrázek B.1: Odkaz na repozitář s kódem.

Zkratky

API Application Programming Interface. 6, 9, 14, 15

CD Continuous Delivery. 7, 16

CI Continuous Integration. 7, 16, 42, 48

CRTP Curiously Recurring Template Pattern. 24

GLM OpenGL Mathematics. 4, 37

GLSL OpenGL Shading Language. 4

GLX OpenGL Extension to the X Window System. 14

GUI Grafické uživatelské rozhraní. 22, 45, 51

OOP Objektově orientované programování. 6, 51

STL The C++ Standard Template Library. 12, 47

UI Uživatelské rozhraní. 8, 15, 16, 27, 44, 48, 51

Literatura

[1] Freeglut: The Free OpenGL Utility Toolkit. http://freeglut.sourceforge.net/ (Navštíveno: 23. 5. 2021).

[2] GLFW: A multi-platform library for OpenGL, OpenGL ES, Vulkan, window and input. https://www.glfw.org/ (Navštíveno: 20. 5. 2021).

[3] Shadertoy. www.shadertoy.com (Navštíveno: 9. 8. 2021).

[4] ISO/IEC 14882:2011. *Programming languages — C++*, 2011. Draft N3337 dostupný z https://github.com/cplusplus/draft.

[5] Emscripten: An LLVM-to-WebAssembly Compiler. *GitHub*, 2021. https://github.com/emscripten-core/emscripten (Navštíveno: 16. 6. 2021).

[6] Gcc – Official Image. *Docker Hub*, 2021. https://hub.docker.com/_/gcc (Navštíveno: 9. 8. 2021).

[7] SCons: A software construction tool. *GitHub*, 2021. https://github.com/SCons/scons (Navštíveno: 13. 7. 2021).

[8] Sean Barrett. Immediate Mode GUIs. *Game Developer*, 12(8):34–36, 2005.

[9] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.

[10] Scott Chacon and Ben Straub. *Pro git*. Springer Nature, 2014.

[11] Michał Cichoń. Node Editor using ImGui. *GitHub*, 2020. https://github.com/thedmd/imgui-node-editor (Navštíveno: 4. 2. 2021).

[12] James O. Coplien. Curiously Recurring Template Patterns. *C++ Report*, pages 24–27, 1995.

[13] Epic Games. Unreal Engine – Input (Version 4.22). *Unreal Engine Documentation*, 2021. https://docs.unrealengine.com/en-US/InteractiveExperiences/Input/index.html (Navštíveno: 15. 6. 2021).

[14] Brian Fitzgerald. A Survey of Programming Language Package Systems. *Blog – Some Things Are Obvious*, 2016. http://neurocline.github.io/papers/survey-of-programming-language-packaging-systems.html (Navštíveno: 11. 4. 2021).

[15] Michal Folta. Systém pro výuku transformací. Diplomová práce, FEL ČVUT, 2016. http://hdl.handle.net/10467/64836.

[16] Martin Fowler. Continuous Integration. *Blog – Martin Fowler*, 2006. https://www.martinfowler.com/articles/continuousIntegration.html (Navštíveno: 5. 7. 2021).

[17] G-Truc Creation. GLM – OpenGL Mathematics. https://glm.g-truc.net/0.9.9/.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[19] Google and Contributors. Bazel. https://bazel.build/ (Navštíveno: 9. 1. 2021).

[20] Google and Contributors. GoogleTest - Google Testing and Mocking Framework. *GitHub*, 2021. https://github.com/google/googletest (Navštíveno: 26. 5. 2021).

[21] Daniel Gruncl. Manipulátory pro editaci transformačních matic a skriptování v I3T. Bakalářská práce, FEL ČVUT, 2021. http://hdl.handle.net/10467/94657.

[22] JFrog. Conan, the C/C++ Package Manager. https://conan.io/ (Navštíveno: 9. 1. 2021).

[23] Kitware and Contributors. *CMake Reference Documentation*. https://cmake.org/cmake/help/latest/index.html (Navštíveno: 8. 1. 2021).

[24] Craig Larman. *Applying UML and Patterns*, volume 2. Prentice Hall Englewood Cliffs, NJ, 1998.

[25] Microsoft Corporation. MSBuild reference. *Microsoft Docs*, 2016. https://docs.microsoft.com/cs-cz/visualstudio/msbuild/msbuild-reference?view=vs-2019 (Navštíveno: 8. 1. 2021).

[26] Microsoft Corporation. CRT library features - Compatibility (MSVC 160). *Microsoft Docs*, 2020. https://docs.microsoft.com/en-us/cpp/c-runtime-library/compatibility (Navštíveno: 20. 7. 2021).

[27] Microsoft Corporation. Vcpkg: C++ Library Manager for Windows, Linux, and MacOS. *GitHub*, 2020. https://github.com/microsoft/vcpkg (Navštíveno: 9. 1. 2021).

[28] Miroslav Müller. Windowing system for the I3T Tool. Semestrální projekt, FEL ČVUT, 2020.

[29] Axel Naumann. *Working Draft, Variant: a type-safe union, N4542*. International Organization for Standardization (ISO), Geneva, Switzerland, 2015. Draft N4542 dostupný z http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4542.pdf.

[30] Marek Nechanský. Automatické rozmisťování propojených modulů v nástroji I3T. Bakalářská práce, FJFI ČVUT, 2019. http://hdl.handle.net/10467/85183.

[31] Object Management Group. The Unified Modeling Language – UML, 2011.

[32] Aymar Ocornut. Dear ImGui Bloat free Graphical User interface for C++. *GitHub*, 2020. https://github.com/ocornut/imgui (Navštíveno: 6. 1. 2021).

[33] Jussi Pakkanen. The Meson Build System – A simple comparison. https://mesonbuild.com/Simple-comparison.html (Navštíveno: 13. 7. 2021). Version 4.10.

[34] Jason Perkins and Individual Contributors. Premake. https://premake.github.io/ (Navštíveno: 8. 1. 2021).

[35] Lukáš Pilka. Grafické návrhy rozhraní pro nástroj I3T. Interní komunikace, 2018.

[36] Richard M. Stallman, Roland McGrath, and Paul D. Smith. Quick Reference. *GNU Make Manual*, 2020. https://www.gnu.org/software/make/manual/html_node/Quick-Reference.html (Navštíveno: 8. 1. 2021).

[37] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.

[38] Bjarne Stroustrup and Herb Sutter. *C++ Core Guidelines*. 2021. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines (Navštíveno: 1. 8. 2021).

[39] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Pearson Education, 2004.

[40] Tanenbaum, Andrew S. and Woodhull, Albert S. *Operating Systems: Design and Implementation*, volume 68. Prentice Hall Englewood Cliffs, 1997.

[41] The Khronos Group. *The OpenGL Shading Language, version 4.50*, 2017. Rev. 7.

[42] The Khronos Group. OpenGL Context. *OpenGL Wiki*, 2019. https://www.khronos.org/opengl/wiki/OpenGL_Context (Navštíveno: 17. 4. 2021).

[43] The Qt Company. Qt for Beginners. *Qt Wiki*, 2021. https://wiki.qt.io/Qt_for_Beginners (Navštíveno: 9. 1. 2021).

[44] Filip Uhlík. Logovací systém pro nástroj na výuku transformací I3T. Bakalářská práce, FEL ČVUT, 2020. http://hdl.handle.net/10467/87647.

[45] Vít Zadina. Testování užitečnosti nástroje pro výuku transformací. Bakalářská práce, FIT ČVUT, 2020. http://hdl.handle.net/10467/83400.

[46] Sofie Šašorina. Převedení krabiček v interaktivním nástroji na výuku transformací do knihovny ImGui. Bakalářská práce, FEL ČVUT, 2021. http://hdl.handle.net/10467/94770.