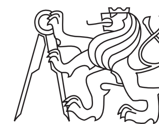
Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra Softwarového Inženýrství



Bakalářská práce

# Jádro systému pro vývoj MMORPG her - uživatelské rozhraní

***Lukáš Vilím***

Vedoucí práce: Ivan Ryant Ing. Bc.

27. června 2014

# Poděkování

Kolegovi Tomáši Bařtipánovi a vedoucímu Ivanu Ryantovi za skvělou spolupráci a trpělivost.

# Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 27. června 2014 . . . . . . . . . . . . . . . . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2014 Lukáš Vilím. Všechna práva vyhrazena.
*Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.*

### Odkaz na tuto práci

Vilím, Lukáš. *Jádro systému pro vývoj MMORPG her - uživatelské rozhraní.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2014.

# Abstrakt

Má práce analyzuje a hodnotí trendy v moderních MMORPG hrách a zahrnuje návrh originálního herního systému v tomto stylu. Implementační část je zaměřena na 3D grapfické uživatelské rozhraní editoru tohoto systému s využítim technologií Qt, OpenGL a metod Cel Shadingu.

**Klíčová slova** MMORPG, Herní Engine, Editor, OpenGL, Cel Shading, Qt.

# Abstract

My thesis analyzes and evaluetes trends in mondern MMORPG games and contains draft of original gaming system in this style. Implementation part is mainly focused on 3D graphical user interface of editor of this system using Qt, OpenGL technologies and method of Cel Shading.

**Keywords** MMORPG, Game Engine, Editor, OpenGL, Cel Shading, Qt.

# Obsah

**Úvod** 1

**1 Cíl práce a projektu** 3

**2 Popis struktury práce ve vztahu k vytyčeným cílům** 5

**3 Rešerše existujících herních MMORPG systémů** 7
3.1 World of Warcraft . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
3.2 Realm of the Mad God (dále již jen RotMG) . . . . . . . . . . . 10
3.3 Lineage 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.4 Shrnutí rešerše . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

**4 Grafické Pojmy** 17

**5 Volba programovacího jazyka a knihoven** 19
5.1 OpenGL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
5.2 Qt . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

**6 Návrh herního konceptu se zaměřením na grafickou podobu hry** 23

**7 Analýza a návrh uživatelského rozhraní Editoru** 25
7.1 Návrh grafického uživatelského rozhraní Editoru . . . . . . . . 26

**8 Realizace** 31
8.1 Implementace GUI pomocí nástrojů Qt . . . . . . . . . . . . . 31
8.2 Třída `Log` . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
8.3 Třída `Grid` . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
8.4 Třída `Camera` . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
8.5 Třída `OpenGLWindow` . . . . . . . . . . . . . . . . . . . . . . . 37
8.6 Třídy `ImporterWindow`, `RoomWindow` a `PreviewWindow` . . . . . 38

8.7 Třída `EditorMainWindow` . . . . . . . . . . . . . . . . . . . . . . 40
8.8 Třída `InstanceConfigurationDialog` . . . . . . . . . . . . . . . 42
8.9 Třída `Model` . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
8.10 Třída `EditorGraphics` . . . . . . . . . . . . . . . . . . . . . . . . 45
8.11 Cel Shading . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
8.12 Translace bodů a objektů pomocí myši . . . . . . . . . . . . . . . 47

**9 Testování** **49**

**Závěr** **51**

**Literatura** **53**

**A Seznam použitých zkratek** **55**

**B Obsah přiloženého CD** **57**

# Seznam obrázků


3.1 World of Warcraft – modifikované uživatelské UI . . . . . . . . . . . 10
3.2 Realm of the Mad God UI . . . . . . . . . . . . . . . . . . . . . . . . 12
3.3 Lineage 2 UI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

7.1 Use case diagram nástroje Importer . . . . . . . . . . . . . . . . . . 27
7.2 Use case diagram nástroje Room Editor . . . . . . . . . . . . . . . . 28
7.3 Use case diagram nástroje pro konfiguraci instance . . . . . . . . . . 29

8.1 Příklad práce s Qt Designerem – ConfigurationDialog . . . . . . . . 31
8.2 Sekvenenční diagram delegace událostí mezi třídami . . . . . . . . . 38
8.3 Diagram provázanosti tříd . . . . . . . . . . . . . . . . . . . . . . . 40
8.4 Struktura typu Mesh . . . . . . . . . . . . . . . . . . . . . . . . . . 43
8.5 Příklad deformace koule pomocí dragged ball shaderu . . . . . . . . 44
8.6 příklad Wireframe-zobrazení . . . . . . . . . . . . . . . . . . . . . . 45
8.8 Pohyb bodem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

# Úvod

Herní Engine je pojem užívaný v souvislosti se softwarovým frameworkem pro tvorbu hry a ovládání jejích periferií, jako je zvuk, zobrazování, síťová komunikace, fyzika, animace, umělá inteligence atd.

MMO (Massive Multiplayer Online) je typ her, které jsou orientované na interakci mezi hráči ve velkém počtu převážně přes internet. Mohou to být dokonce stovky či tisíce a herní servery na tento nápor musí být připravené, poskytovat hráčům informace včas a minimalizovat znevýhodnění mezi hráči rozdílnou kvalitou připojení.

RPG (Role Playing Game) často překládáno do češtiny jako „hra na hrdiny". Jedná se spíše o hraní rolí, jak říká anglický název. Hráči se zde stylizují do postav, které si zvolí a jednají jako ony v herním světě. Na tento svět vždy dohlíží nějaká entita, která vymezuje pravidla a omezuje hráče, aby nejednali v rozporu s nimi. Než se tento koncept převedl do počítačových her, předcházel mu známý koncept deskových her podobných tradičnímu D&D (Dungeon And Dragons), kde na hráče dohlížel další hráč v roli takzvaného „Dungeon Mastera — Pána jeskyně", který hráče prováděl herním příběhem a světem. V této roli se v počítačových hrách ocitá herní engine, který musí kontrolovat, zda se všichni hráči chovají správně a případně některé trestá. Samozřejmostí je, že sám engine není neomylná entita a tudíž když se prvky RPG spojí s výše zmíněným MMO je doplněn o hráče v roli administrátorů, kteří pomáhají normálním hráčům řešit jejich problémy s enginem a eliminovat ty, kteří se ilegální cestou snaží engine zneužít.

Důležitou součástí RPG her byly vždy jeskyně či jiné prostory, které hráči prozkoumávali a plnili v nich úkoly. V MMORPG hrách se vyskytují také, ale vznikl zde jeden nový druh takovýchto míst, kterému se říká instance. Jedná se o druh prostředí, který není společný pro všechny hráče. Pokud se skupina hráčů rozhodne na takové místo vystoupit, připraví se pro ně jejich vlastní prostředí, které nesdílí se žádnou další skupinou hráčů. Může se například jednat o jeskyni, která bude pokaždé vypadat jinak, nebo také o stále stejnou jeskyni. Většinou jsou tato místa omezena maximálním počtem hráčů,

kteří mohou vstoupit. Mnoho MMORPG her takto řeší problém, který přináší mnoho hráčů v herním světě a odděluje je tímto způsobem od sebe.

FPS (First Person Shooter) je koncept akčních her z pohledu první osoby. Tento koncept klade důraz na reflexy hráče a jeho rychlost. Prvními zástupci tohoto žánru jsou například Doom nebo Wolfenstein 3D.

Kapitola 1

# Cíl práce a projektu

Cílem projektu, jehož součástí je tato bakalářská práce, je návrh a implementace funkčního enginu pro MMORPG s důrazem na modularitu světa ve kterém se bude hra odehrávat pomocí architektury klient — server — editor. Mou rolí je návrh a implementace grafického uživatelského rozhraní a kolega Bařtipán pracuje výhradně na aplikační logice.

Grafická část projektu je zaměřená na nefotorealistické zobrazování scény stylizované do komiksové podoby pomocí OpenGL.

Cíle projektu jsou následující:

- Rešerše existujících herních MMORPG systémů
- Prototyp Editoru
  - Analýza způsobu užití a návrh uživatelského rozhraní
  - Práce s 3D modely a jejich obalovými tělesy
  - Vytváření herního obsahu používaného herním enginem
- Prototyp Serveru
  - Návrh a implementace
  - Síťová komunikace
- Prototyp Klienta
  - Návrh a implementace

Kapitola 2

# Popis struktury práce ve vztahu k vytyčeným cílům

Během tvorby prototypu editoru jsem si neuvědomil, že splnit zadání v plném rozsahu v daném časovém rámci bude nad mé možnosti. Nebyl jsem si vědom celkové složitosti projektu do doby, než jsem začal navrhovat jednotlivé herní mechanismy a způsoby jejich realizace.

Obsahem mé práce tedy bude analýza, návrh a implementace uživatelského prostředí editoru herního obsahu, rešeršní zpracování existujících herních systémů pro hraní MMORPG her a koncept herních mechanismů, který má dopad na celkovou podobu projektu a jeho realizaci.

Nejprve se budu věnovat existujícím herním systémům, jejich grafické podobě a mechanismům pro ně charakteristické. Na základu této rešerše navrhnu koncept herních mechanismů, které bych chtěl uplatnit v mém projektu.

V druhé části vymezím pojmy týkající se grafického uživatelského prostředí a dále se zaměřím na jeho analýzu, návrh a implementaci.

Kapitola **3**

# Rešerše existujících herních MMORPG systémů

V dnešní době se na trhu objevuje stále větší množství MMORPG her, které jsou si v mnohém velice podobné a snaží se přiblížit trendům několika málo nejúspěšnějších. Zvláště v poslední době je opravdu velký boom her tohoto typu, které trh spíše pohltil.

Budu zde zmiňovat hry, které mají úspěch a svým způsobem byly jako první v něčem unikátní. Toto srovnání vyzdvihuje herní mechanismy které danou hru charakterizují a ani při delším setrvávání v herním světě neustoupí do pozadí. Tyto mechanismy se také projevují v nástrojích používaných při tvorbě obsahu těchto her.

S těmito nástroji se bohužel nemáme možnost seznámit, jelikož nebývají dostupné veřejnosti, ale pouze vývojářům pracujícím na projektu. Tedy se můžeme pouze domnívat, jak dané nástroje vypadají a fungují. Při návrhu podoby nástroje pro tvorbu herního obsahu budu vycházet ze vztahů určených konceptem.

V následující rešerši se zaměřím na následující charakteristiky:

- Uživatelské rozhraní a ovládací prvky
- Důležité a zajímavé herní mechanismy
- Reprezentace postav
- Zhodnocení nároků na hráče

Jako vhodné kandidáty z kterých vycházím jsem zvolil World of Warcraft, Realm of the Mad God a Lineage 2.

Série Lineage byla jednou z prvních, která zavedla MMORPG hry, jak je dnes známe a tak jsem ji vybral jako klasického zástupce tohoto žánru.

World of Warcraft byl prvním konceptem, který dovolil hráči úplně volný pohyb po herním světě. Postava zde konečně mohla skákat a dobývat tak

ukryté, těžko dostupné lokace a v neposlední řadě létat. Přenesly se tak některé herní prvky FPS her do MMORPG.

Realm of the Mad God je zde v roli nováčka, který není produkován velkým herním studiem a přes svoji jednoduchost si rychle získal širokou hráčskou základnu.

## 3.1 World of Warcraft

Byl jedním z prvních MMORPG, které přineslo rozsáhlou strukturu herního světa a nepřeberné množství možných činností. Příběhově vycházel ze světa svých předchůdců a nabídl hráčům velký svět plný úkolů, monster, měst a ras, které mohli hráči objevovat.

Je klasickým zástupcem lineárních MMO, které předkládají hráči svět plný úkolů vedoucích vždy k jasnému cíli a postupně ke zvýšení úrovně hráče na maximum. Dosáhnout v takovéto hře vyšší úrovně pomocí jiné činnosti než je plnění zadaných úkolů je takřka nemožné a ničí to tak celkový zážitek ze hry hráčům, kteří rádi přemýšlí a dělají věci po svém.

Grafika hry je v plném rozsahu 3D a umožňuje hráči skákat, létat a jezdit na zvířatech, to vše v relativně pěkném zpracování s jednou výtkou, toto zpracování se již delší dobu radikálně nezměnilo a až na pár detailů vypadá stále stejně již deset let.

Ovládací prvky, které hráč používá k interakci se svou postavou jsou přehledně rozmístěny po obrazovce a hráč si jejich převážnou většinu může sám rozvrhnout jak potřebuje. Důležité plus jsou v tomto ohledu takzvané addony, programy které umožňují ovlivňovat podobu a rozmístění ovládacích prvků a dokonce přidat i některé nadstandardní funkce usnadňující hráči hru. Díky nim si každý hráč muže sestavit takové prostředí, které bude odpovídat jeho potřebám.

Struktura ovládacích prvků i přes všechny modifikace je stále velice podobná. Jedná se o prostředí poskytující hráči informace o jeho postavě, například kolik má životů či many, a také o prostředí ve kterém se pohybuje, jako je mapa lokace, pozice protivníků a další.

Nutno podotknout, že zde hráč přijde do styku i s dalšími nabídkami zahrnujícími informace o hráčích, klanech, obchodu a také různých úkolech. Jedná se o velice komplikovaný systém, který vyžaduje vlastní framework na práci s okny a ovládacími prvky.

Jako většina moderních MMORPG využívá konceptu označených cílů, takže se hráč nemusí příliš starat o směr, kterým by měl střílet své projektily či kouzla. Vystačí si s označením na protivníka a stiskem příslušných kláves či klikáním na ikony ovládacích prvků.

Jak jsem již zmínil, hráč si veškeré své schopnosti může dle libosti organizovat pomocí addonů či nastavení základního prostředí do různých skupin

a kolonek s ikonami. Takto uspořádané schopnosti pak může používat pomocí modifikovatelných klávesových zkratek.

Celkově je tento přístup k uživatelskému rozhraní opravdu příjemný. Uživatel může jinak všední prostředí přetvořit v rozhraní, které se mu bude líbit. Tento přístup je u herní komunity velice ceněný.

Nyní přejdu k herním mechanismům. World of Warcraft obsahuje v současné době 13 ras, které může hráč zvolit pro svou postavu. Volba rasy určuje hráčovu příslušnost k jedné ze dvou znepřátelených frakcí, Aliance a hordy.

Jako další zvolí povolání, které určuje již po zbytek jeho herní kariéry styl, kterým hru hraje. Povolání by se zde dala rozdělit do pěti typových kategorií a to na Tank, Melee DPS, Ranged DPS, Healer a Off Tank. Kdo se již s podobným systémem setkal, tuší o čem je řeč. Důležité je, že po zbytek herní kariéry je hráč zaškatulkován jako jeden, maximálně dva z těchto typů, protože vhodnou volbou takzvaných talentů se postava může změnit z jednoho typu na druhý, ale za relativně vysokou cenu. Hráč pak také musí mít příslušné vybavení pro hraní dané kategorie a jeho získání bývá často velice časově náročné.

Jak již jsem zmínil, hru ovlivňují talenty, které říkají, jaké může postava používat schopnosti a jaké dostává pasivní bonusy. To vše organizované do tří stromů, ve kterých se člověk snaží vybalancovat své talenty tak, aby byl v boji co nejefektivnější jako daná kategorie.

Dalším aspektem je to, že hra je orientovaná na dva druhy boje. Jeden je Player versus Enviroment (PvE), kde se hráči utkávají s monstry volně po světě, nebo organizovaně v unikátních lokacích, které se příliš nemění a maximálně se mění rozmístění soupeřů a strategie nutné k jejich zdolání. Druhým typem boje je Player versus Player (PvP). Jak název napovídá jedná se o souboje jednoho nebo více hráčů proti sobě.

Za první i druhý typ boje je hráč řádně odměněn, ale pravidlem bývá, že odměny budou opět typově orientované. Tudíž za zabití těžkého bosse hráč získá blyštivé brnění, které se hodí při dalším souboji s jiným těžkým bossem, ale na boj hráče proti hráči se nehodí. Stejně tak v PvP.

Když všechno shrneme a pomineme pár okrajových činností jako je zpracování materiálů a vyrábění lektvarů, je World Of Warcraft pouze o tom jak zabít efektivně dané monstrum nebo protihráče a o tom jak se co nejrychleji dostat na nejvyšší úroveň. Což ostatně může trvat několik měsíců a nejsem si jist, jestli si hráči stále tolik užívají prvních 50 úrovní svých postav z celkových 100. Mám dojem, že spíše ne a možná se kýžený zážitek dostaví až na zmíněné stovce.

Dalším mínusem v zábavnosti hry je fakt, že jakmile postava umře, tak krom jistého poničení jejích věcí vlastně neutrpí žádnou citelnou ztrátu a může tak vesele pokračovat v hraní bez strachu z jisté smrti.

Obrázek 3.1: World of Warcraft – modifikované uživatelské UI

## 3.2 Realm of the Mad God (dále již jen RotMG)

Je webová hra, která svou zábavností a chytlavým konceptem v mnohém předčí i výše zmíněný World of Warcraft.

Grafické zpracování navozuje jistou iluzi 3D, ale veškerá herní logika se třetím rozměrem nepracuje. Uživatelské rozhraní je zde sice strohé a nemodifikovatelné, ale nabídne hráči přesně to co potřebuje. Všechny interaktivní grafické prvky jsou zde umístěny v pravé části obrazovky členěné do jednotlivých skupin podle významu. Úplně nahoře je jednoduchá mapa, podle které se hráč orientuje a může i využít možnosti výběru jednoho či skupiny hráčů kliknutím myši a případně se k nim teleportovat. Dále jsou zde vlastnosti postavy, pod nimi se nachází jednoduchý inventář věcí, které má postava na sobě a u sebe a také seznam nejbližších šesti postav.

S okolními postavami může hráč interagovat jen omezeně a to převážně za účelem obchodu. V dolní části obrazovky je pak okno s veškerou komunikací v lokaci ve které se hráč nachází.

Ovládání je triviální a během chvíle si ho osvojí téměř každý. Jedná se o klasický koncept WASD zajišťující pohyb nahoru, vlevo, dolů a doprava, kde hráč myší míří projektily, které jeho postava střílí.

Toto vše se dá libovolně nastavit, včetně jedné jediné schopnosti, kterou má každá postava.

Tento systém není příliš komplikovaný, důvodem je jednoduchost hry, která je zaměřena pouze na zabíjení monster.

Svět je složený ze čtvercových ploch a celá hra je zabalena do osmibitového kabátu, kde se jako postavy prohánějí $32 \times 32$ sprity střílející různé projektily.

Svět je náhodně generovaný a téměř každou hodinu se celý přegeneruje.

V tomto světě hráči objevují vstupy do různých generovaných instancí, které většinou prozkoumávají ve větších skupinách, protože monstra jsou v nich opravdu silná. Zkušenější hráč však dokáže většinu těchto instancí dokončit úspěšně i sám.

Po smrti některého z nepřátel je šance že se na zemi objeví pytel s náhodně generovaným vybavením podle typu nepřítele.

Další možnosti pohybu se otevřou pokud hráč zvolí režim odemčené kamery a může poté ještě otáčet kamerou doprava a doleva okolo své postavy, což mu při správném použití poskytuje jistou výhodu.

Hra obsahuje čtrnáct postav, kde má každá unikátní schopnost. Tuto schopnost může vylepšovat tím, že najde lepší verzi předmětu, který má ve svém slotu pro předmět schopnosti.

Jedná se o takzvanou plně kooperativní hru, kde se hráči v normálním herním módu nemohou navzájem ohrozit a sdílejí jistou porci vybavení, které získají za zabíjení monster.

Na začátku si zvolíme postavu za kterou chceme hrát. A že na začátku moc velký výběr není, protože jediná odemčená postava je čaroděj a k odemčení dalších musíme dosáhnout určité úrovně s již odemčenou postavou. Každá postava může dosáhnout maximální úrovně 20 a to zkušenější hráč stihne během hodiny a někdy i daleko rychleji.

Nutno ale zmínit, že pokud postava dostane smrtelný zásah, umírá. A to bez nároku na záchranu věcí které má u sebe. Tomu se dá předcházet včasným útěkem z boje, což umožňuje rychlý teleport do hlavního města. Tento teleport může využívat každý hráč kdykoli a kdekoli.

Další novátorská funkce tohoto herního systému je, že hráči nacházející se ve stejném světě či instanci se k sobě mohou volně a instantně teleportovat. To přináší úplně jiný herní styl, který je rychlejší a zábavnější.

Nejsou zde vůbec žádné úkoly ani války mezi hráči. Co tedy udržuje hráče ve hře i po dosažení dvacáté úrovně se všemi postavami? Jakmile hráč dosáhne úrovně dvacet, začíná sbírat takzvanou slávu za každé zabité monstrum a lepší nalezený předmět. Všechny postavy mají jisté atributy, které se během hry mění a na úrovni 20 většinou hráči začínají hledat lektvary, které trvale zvyšují dané atributy a snaží se dosáhnout maxima u všech atributů.

Hra je velmi akční a koncept trvalé smrti může část herní komunity odradit. Na druhou stranu dělá hru zajímavou a hráči si tedy mohou vychutnat hru na nízkých úrovních vícekrát. Nenastane tedy stav, kdy hráče přestane hra bavit, protože ji již dokončil se všemi postavami a získal nejlepší vybavení.

Obrázek 3.2: Realm of the Mad God UI

## 3.3 Lineage 2

První díl této MMORPG hry byl jedním z prvních her tohoto žánru vůbec.

Grafické zpracování je opět ve 3D, ale s podstatnými omezeními. Postava se zde nepohybuje pomocí kláves WASD, ale pomocí kliknutí na příslušné místo. Postavy zde nemohou skákat a tak i nízké překážky s většími kolizními obálkami jsou nepřekonatelnou překážkou.

Dalo by se možná říct, že tento systém je horší a čím větší volnost je hráčům dána, tím lépe, ale já si myslím opak. Když si vzpomenu na jiné herní koncepty jako je třeba proslulé Diablo, tak k němu neodmyslitelně patří pohyb pomocí myši a nikoli kláves.

Uživateli jsou zde nabídnuty ovládací prvky, které jsou do jisté míry nastavitelné. Opět se jedná o informace o zvoleném cíli a postavě samotné. Také je zde nabídka do které mohou, podobně jako u World of Warcraft, hráči umístit schopnosti svých postav. Tato nabídka však již příliš nastavitelná není a klávesové zkratky na její ovládání jsou fixně nastavené na F1 až F12. Celkově je zde k dispozici až 36 volných pozic na schopnosti postavy, které se dají ovládat zkratkami kombinující CTRL, SHIFT a již zmíněné funkční klávesy.

Hráč se zde setká také s komplikovanými nabídkami zabývajícími se jeho schopnostmi, zkušenostmi, úkoly a politikou. Celkově je interakce s prostředím

a ostatními hráči na vysoké úrovni.

Tento systém je opět velmi komplexní, ale na rozdíl od World of Warcraft nenabízí takovou flexibilitu.

Hra je stylizována do anime fantasy prostředí, kde si hráč vybírá ze šesti ras. Každá rasa má svá specifická povolání, která se na určitých úrovních dále větví. Hráč, který dosáhne maximální úrovně má jednu jedinou specializaci a zvýšit variabilitu své postavy může pouze pomocí systému vedlejších povolání. Tento systém povoluje naučit se nové povolání od úrovně 40. Takže můžete ve světe potkat trpaslíka který kouzlí, i když trpaslíci žádné kouzelníky nemají. Každá postava může dosáhnout maximální úrovně 90 a to může hráči zabrat i půl roku.

Další zajímavou vlastností hry je postih za smrt. Pokaždé když postava zemře, bez ohledu na to jak, dostane postih v podobě odebraných zkušeností. Pokud postava zemře vícekrát, může sestoupit i o úroveň níže.

Hra umožňuje útočit na ostatní hráče a pokud se napadený hráč nebrání, tak útočník dostane postih v podobě karmy. Karma indikuje že hráč zabil bezbranného protivníka a ostatní hráči ho mohou do doby, než se karmy zbaví bez postihu napadat. Pokud má postava karmu, je zde možnost, že při smrti nepřijde pouze o zkušenosti, ale i o část výbavy. Čím více karmy má, tím větší je šance ztráty. Postava může postupně snižovat množství karmy zabíjením monster, nebo tím, že ji někdo zabije.

Herní koncept neklade velký důraz na plnění úkolů, takže hlavním zdrojem zkušeností pro hráčovu postavu je zabíjení monster ve velkých počtech. Většinou se tomu věnují větší skupiny hráčů, které se organizují tak, aby zabili za co nejkratší dobu co nejvíce monster.

Schopnosti jednotlivých postav také nabádají ke spolupráci s ostatními hráči, což je i autorský záměr. Pokud se hráči dostatečně zorganizují, mohou dosáhnout daleko většího úspěchu, než jednotlivci. I když je jednotlivců opravdu mnoho, jednoznačně podlehnou organizované skupině.

Hra navíc obsahuje velké množství měst, která mají hrad a o ten se může jednou týdně bojovat. Klan který vlastní město má výhody a také zisk z obchodů v něm, jednoduše se vyplatí alespoň zkusit o město bojovat. Tady přichází spousta dalších mechanik jako jsou trpasličí obléhací stroje, či najatá garda počítačem ovládaných jednotek, které budou bránit hrad s bránícím klanem.

Obrázek 3.3: Lineage 2 UI

## 3.4 Shrnutí rešerše

Příklady produktů výše vykazují diverzitu v přístupu ke hráči, herním mechanikám i koncepci světa. A že myšlenky v nich obsažené nejsou zrovna nové, dokazuje to to, že World of Warcraft i Lineage 2 jsou na trhu již více než deset let. Ve srovnání s nimi je graficky daleko hůře vybavený Realm of the Mad God nováčkem v tomto oboru a i přesto si rychle získal velkou a stálou komunitu.

Bohužel je RotMG omezený dvěma aspekty, které spolu úzce souvisí.

První z nich je platforma na které je realizovaný a tou je platforma flash. Flash je sice dobrý na jednoduché webové hry a aplikace, ale napsat v něm MMO je opravdu odvážné.

Druhým aspektem je přístup vývojářů, kteří nedrží krok s komunitou a přidávají nový obsah jen zdrženlivě. Je vidět, že omezení, která plynou z platformy či implementace, jsou tak veliká, že udělat větší zásah do projektu bez jeho zhroucení je opravdu tvrdý oříšek. K těmto domněnkám přispívají časté pády serverů při přechodech na nové verze a velice špatný výkon herního klienta na naplněných serverech.

Došel jsem se k závěru, že hra s koncepcí světa podobnou RotMG by měla velkou šanci na úspěch. Z mého pohledu je RotMG jen prototypem otevírajícím možnosti dalším hrám podobného ražení. Na komunitě je vidět, že o takové produkty stojí.

Poukazuje na to nejen rostoucí zájem o RotMG, ale i klesající popularita World of Warcraft a Lineage 2. Tyto projekty již tolik nové hráče nepřitahují a ti spíše hledají alternativy vybočující ze standardů těchto her.

Kapitola 4

# Grafické Pojmy

**Scéna** — je množinou objektů rozestavěných, rotovaných a zvětšených v prostoru, která je zobrazována grafickým rozhraním zařízení. Objekty jsou většinou definovány množinou bodů v lokálním prostoru modelu, které jsou rozmístěny co nejblíže středu a posléze přeneseny pomocí modelové transformace na pozici ve scéně ve světových souřadnicích.

**GPU** (Graphic Processing Unit) — grafický procesor umístěný na grafické kartě sloužící speciálně k vykreslování dat uložených v operační paměti zobrazovacího zařízení.

**GUI** (Graphical User Interface) — je typem uživatelského rozhraní, které uživateli umožní ovládat program či zařízení pomocí interaktivních grafických prvků.

**Shader** — počítačový program sloužící k řízení jednotlivých částí programovatelného zobrazovacího řetězce grafické karty (GPU).

**Cel Shading** — typ nefotorealistického vykreslování používaného v počítačové grafice k vytvoření iluze komiksového obrazu. Skládá se většinou ze dvou částí. První je vykreslení siluety objektu a druhá je vykreslení osvětleného objektu s limitovaným počtem přechodů mezi jednotlivými odstíny barev.

**Widget** — neboli ovládací prvek je základní element pro interakci programu s uživatelem. Je vizuálně ztvárněn a slouží k manipulaci s daty daného programu.

**Container Widget** — typ ovládacího prvku, který zahrnuje jeden nebo více ovládacích prvků do skupiny.

**3D model** — soubor bodů, normál, texturových koordinát a indicií podle kterých se jednotlivé body spojují v roviny. Modely jsou většinou uloženy v 3D grafických souborech různých formátů a mohou obsahovat i další informace o animacích či materiálech.

**Obalové těleso** — slouží k co nejpřesnější aproximaci objektu za využití co nejmenšího počtu primitiv s účelem co nejvíce zjednodušit výpočty potřebné k interakci mezi objekty.

**Sprite** — název pro malý 2D obrázek integrovaný do větší scény. Pokud je sprite animovaný, jedná se o sekvenci stejně velký obrázků střídajících se tak, aby budily dojem pohybu.

**Mesh** — 3D modely z důvodu přehlednosti a logického rozdělení často dělí na menší části, meshe.

**Vertex** — bod v prostoru určený souřadnicemi.

Kapitola 5

# Volba programovacího jazyka a knihoven

Volba programovacího jazyka byla rychlá. OpenGL sice podporuje i jiné jazyky než C++, jako třeba jazyk Java, ale po předchozích zkušenostech s OpenGL podporou v Javě jsem se raději uchýlil k C++, které OpenGL nativně podporuje.

Při návrhu implementace GUI jsem musel zvolit knihovny se kterými budu pracovat. Potřeboval jsem dobrou knihovnu na podporu OpenGL a knihovnu, která by mi zjednodušila načítání 3D modelů z používaných souborových formátů.

Pro načítání modelů jsem zvolil knihovnu Assimp, kterou jsem již dříve používal v semestrálním projektu na předmět BI-PGR zabývajícím se počítačovou grafikou a OpenGL. Plně splňovala mé požadavky a nemusel jsem se učit nové postupy. Dalším plusem je, že tato knihovna umí pracovat nejen s daty modelu, ale i s animacemi. A to jak s meshovými, tak i skeletonovými, které bych později v projektu chtěl využít. Také umí k načítaným modelům předpřipravit některá data navíc, jako třeba spočítat hladké normály, otočit pořadí bodů, nebo rozdělit model na trojúhelníky.

Poté jsem vybíral mezi knihovnami, které by mi usnadnily práci s OpenGL a vytvářením GUI, SDL a Qt.

SDL (Simple DirectMedia Layer) je stejně jako Qt multiplatformní multimediální knihovnou podporující nízkoúrovňové programování v OpenGL. Jenže SDL, která je vývojáři her široce využívána, nemá vyšší podporu GUI. Pro implementaci prototypu editoru je stěžejním kritériem rychlý návrh přehledného GUI.

Takže zvítězil balík Qt, který má knihovny a nástroje pro jeho tvorbu.

## 5.1 OpenGL

OpenGL je standard poskytující multiplatformní rozhraní pro tvorbu grafických aplikací. Používá se při vývoji CAD programů, vizualizací, počítačových her a dalších aplikací, které vyžadují podporu znázorňování grafických primitiv. Je podporováno téměř na každém zařízení umožňujícím vykreslovat grafiku a pokud není, umožňují to některé softwarové implementace, ovšem s nižším výkonem.

OpenGL není objektově orientované a pracuje převážně s 3D modely. Samo o sobě je stavovým automatem a tudíž se s ním musí také tak zacházet. Celý systém je sekvenční a pokud programátor nastaví, že má přejít do nějakého stavu, tak zůstane v tomto stavu dokud jej programátor opět nezmění. Jedná se o architekturu jednotlivých příkazů, které se posléze uskutečňují na GPU.

Poskytovaný přístup ke grafickému rozhraní je opravdu nízký a tudíž se nehodí k realizaci menších projektů u kterých programátor rychle potřebuje vysokou funkčnost a vystačí si s nižším výkonem.

Jakmile ale vyžadujeme vysoký výkon a velkou modularitu výstupu, například při vykreslování modelů složených ze statisíců bodů a jejich následné deformace, tak je OpenGL ideálním nástrojem.

Také poskytuje podporu programů, které běží na GPU. Tyto programy jsou psány v jazyce GLSL (OpenGL Shading Language), který je velice blízký klasickému C, ale obsahuje i syntaxe, které jsou pro něj specifické. Tyto programy umožňují kontrolovat zobrazovací řetězec GPU a jmenují se shadery. Základní typy shaderů jsou následující:

- Vertex shader – Zpracovává vždy v jeden čas jeden vrchol a data s ním související. Spočítá souřadnice jeho zobrazení a pošle data dál fragment shaderu.

- Fragment shader – Pracuje vždy s jedním texelem (pixelem) a počítá jeho barvu z různých dat poskytnutých předešlými shadery, nebo uživatelem.

- Geometry shader – Na rozdíl od vertex shaderu může pracovat s více vrcholy najednou a dokonce i vrcholy generovat a tím vytvářet různé efekty či další primitiva.

- Tesselation shader – Zajišťuje podporu tesselace rovnou v shader programech. Tento celkem nový typ shaderu generuje z daných dat další geometrii podobně jako geometry shader. Používá se převážně ke zvětšení detailu modelů přidáváním dalších bodů a zpřesňování cílové podoby modelu interpolací.

Typický postup vykreslování modelu se skládá z:

- Inicializace
- Přepočtu řídícich dat zobrazení
- Vykreslení

Inicializace sestává převážně z kompilace shader programů, načtením dat modelu ze souboru a nahrání do paměti GPU. Toto umožňují vertexové buffery, do kterých může programátor uložit téměř jakákoli data a posléze specifikovat jak je má OpenGL interpretovat a předávat jednotlivým shaderům.

Základní řídící data jsou matice jednotlivých zobrazení. A pokud se objekty nebo jejich části ve scéně hýbají a deformují, tak i jejich data se mění a musí se přepočítávat.

Před kreslením se vždy nastaví stavový automat do stavu, který vyhovuje předpokládanému výsledku, třeba že se mají zobrazovat jenom plochy otočené směrem k pozorovateli nebo naopak od něj. Poté se zvolí způsob, kterým se mají dané vrcholy v bufferech zobrazit a zbytek se děje na GPU v předpřipravených shader programech. Příkladem takového způsobu je třeba `GL_TRIANGLES` , který bude spojovat trojce bodů a vytvoří mezi jejich spojnicemi rovinu.

Více zde technologii OpenGL rozvádět nebudu. Výše zmíněná fakta by měla postačovat k jejímu částečnému porozumění potřebnému k pochopení mé práce.

Je to komplexní obor, kterému se věnují daleko obsáhlejší práce a k jejímu plnému pochopení musí čtenář dospět jinou cestou. Dále se o něm budu zmiňovat jako o standardu který využívám k zobrazování grafiky a budu brát jako samozřejmost jistou úroveň orientace v této problematice.

## 5.2 Qt

Jedna z nejpopulárnějších multiplatformních knihoven pro vytváření programů s grafickým uživatelským rozhraním. Knihovna existuje pro jazyk C++, Python, Ruby, Perl, Pascal, C♯, Java a Haskell.

Aplikace vytvořené pomocí této knihovny se budou vždy chovat podle právě používané platformy a přebírají její vzhled.

Funkčnost obsáhlá touto knihovnou je opravdu obrovská. Obsahuje knihovny pro práci se soubory, tvorbu grafického rozhraní, stylizování jednotlivých oken pomocí CSS stylesheetů, matematické operace, OpenGL grafiku, síťové operace, databázové operace, práci s webem a periferiemi zařízení nezávisle na platformě.

V neposlední řadě tvůrci Qt zavedli vlastní programovací jazyk QML, který umožňuje spojovat všechny výše zmíněné jazyky a využívat funkčnost v nich

implementovanou při tvorbě grafických aplikací. Další knihovní částí úzce svázanou s QML je QtQuick 2, knihovna zaměřená na grafiku a vysokoúrovňové programování pomocí QML, kdy se různé části QtQuick 2 dají podědit například v C++, nadeklarovat vlastní funkci a opět používat v QML.

V mém projektu jsem použil knihovní verzi pro C++. Hlavními přednostmi využití Qt je rozšíření klasického C++ o meta třídy, které jsou generovány pomocí metaobject kompilátoru. Tento kompilátor detekuje v C++ kódu určitá makra, z nich poté generuje takzvané MOC soubory obsahující další data o použitých třídách a zajišťující funkci, která není v C++ nativně podporována.

Jedním z příkladů jsou signály a sloty, které se stanou přístupnými kterékoli třídě dědící od jakékoli třídy obsahující `QObject`, nebo dědící od `QObjectu` samotného a obsahující `Q_OBJECT` makro. Signály jsou typy metod, které jsou implementovány pouze v headeru v sekci „signals:“, jako argument mají hodnotu signálu, která se má předávat slotu a volány jsou kdekoli v kódu třídy, když je potřeba daný signál emitovat.

Sloty jsou deklarované jako typické metody třídy, jenže v sekci „slots:“, která může být stejně jako ostatní metody třídy private, public nebo protected. Slot může být volán stejně jako každá metoda, ale navíc může být spojen se signálem pomocí funkce `connect()`. Na jeden signál může být navázáno více slotů a stejně tak na jeden slot více signálů. Dalo by se říct, že sloty a signály nejsou nic jiného, než obyčejné metody, ale to je velký omyl.

Hlavní výhodou signálů a slotů je, že spojují objekty na úrovní instancí a nikoli tříd, takže se program může za běhu rozhodnout jaké instance k sobě naváže.

Další funkcí zajištěnou meta informacemi z MOC souborů je introspekce. Jedná se o schopnost programu zjistit typ vlastností objektu za běhu.

Kapitola 6

# Návrh herního konceptu se zaměřením na grafickou podobu hry

Mou ideou je hra podobná RotMG, stylizovaná do komiksového světa fantasy ve 3D.

Interakci s postavou bych zajistil pomocí kláves WSAD, myši a klávesových zkratek závislých na rozložení schopností a akcí ve fixních nabídkách na obrazovce.

Místo velkého množství postav s jedinou schopností bych volil jednu postavu s modifikovatelným vzhledem a větší diverzitou schopností zajištěnou rozsáhlým stromem schopností, kde by pro dosažení další úrovně schopnosti bylo potřeba schopnost předešlou co nejvíce efektivně využívat.

Nutností bude propracovaný dialog s tímto stromem, kde budou hráči nabídnuty jednotlivé schopnosti a požadavky k jejich získání. Interaktivně si pak vybere jakou cestou se bude ubírat a jak příslušné schopnosti získá.

Koncept by neměl klást vysoké nároky na získávání zkušeností a zdlouhavé postupování po úrovních. Zlepšování atributů postavy by hráč mohl ovlivnit jen minimálně, takže by se tyto informace daly shrnout například do dialogu s inventářem postavy a největší vliv na hru by pak měl výše zmíněný strom a vybavení, které postava používá.

Nesmí zde samozřejmě chybět interakce s entitami ovládanými počítačem. Ty bych řešil dialogem obsahujícím text konverzace mezi hráčem a entitou a případné další ovládací prvky.

Hlavní část herního světa by se měla pseudonáhodně generovat z předem vymodelovaných částí. Instance by se pak měly generovat plně náhodně tak, aby hráči nikdy netušili, co je čeká.

Tím se dostáváme k tomu, že celý tento systém bude muset obsahovat události, spouštěče událostí a jejich následky. Všechny tyto věci budou pevně

svázané s instancí či částí světa. Tato funkce bude asi nejnáročnější částí práce, protože abych zaručil modularitu událostí, musím do projektu integrovat skriptovací jazyk.

Ve hře by měly hrát velkou roli takzvané sféry. Jednalo by se o úrovně světa, ve kterých by se chování postav hráčů jistým způsobem změnilo. Některé postavy by dostali speciální schopnost, jiné postih či bonus. Mezi některými z nich by se hráči mohli pohybovat svobodně a existovaly by tedy souběžně s normálním herním světem. Pokud by se postava koncipovaná jako zloděj schovala ve sféře stínů, mohla by být objevena postavou, která přenese všechny z ostatních sfér v určitém okruhu do normálního světa.

Jiné by představovaly vyšší úroveň bytí na kterou se hráči mohou postupem po úrovních a plněním úkolů dostat. V těchto sférách by pak hráč mohl zemřít a znamenalo by to pro něj pouze sestup o sféru níže. Také by se zde dalo najít unikátní vybavení, speciální monstra a instance.

Vstupy do jednotlivých instancí by se náhodně generovaly po mapě. Každý průchod instance by měl být náhodně generovaný. Tuto část pokrývá práce kolegy Bařtipána. Instance by se měla skládat z pokojů různých typů, jako jsou chodby, slepé místnosti a normální místnosti. Z těchto primitiv se při hře algoritmicky vygeneruje náhodné seskupení.

Některé postavy by se pak specializovaly na přecházení mezi sférami. Takto specializovaní hráči by pak mohli dočasně přemisťovat postavy do různých sfér a získávat taktické výhody. Například přenést protivníka na okamžik někam kde je slabý, nebo schovat skupinu postav před nájezdem monster.

Tento koncept dělení světa je unikátní a nabízí velký potenciál. Na sférách a pohybu mezi nimi se dá postavit dynamický systém schopností postav a nových povolání.

Většina výše uvedených požadavků se přímo netýká prototypu editoru herního obsahu, ale bude mít velký dopad na podobu serverové a klientské části aplikace. Další důležité herní mechanismy jsou součástí práce mého kolegy, který implementuje logickou část projektu.

Tento koncept je můj pohled na projekt a jeho realizaci. Jelikož se ale jedná o týmovou práci, bude podléhat úpravám a diskusi s kolegou Bařtipánem.

Kapitola 7

# Analýza a návrh uživatelského rozhraní Editoru

Editor by se měl chovat jako současné 3D modelovací programy, jako je například Blender.

Bylo jednoznačně řečeno, že systém musí umět pracovat s 3D grafikou a tudíž i s modely objektů a jejich daty. Dále musí podporovat vlastní souborové formáty a jejich vytváření. Tuto problematiku definuje a řeší práce kolegy Baťtipána.

Skládá se ze tří částí, které byly identifikované z návrhu konceptu hry a následné diskuze s kolegou.

Nástroj věnující se integraci samotného modelu do projektu se jmenuje Importer, bude podporovat známé formáty 3D modelů, přidá k nim data důležitá pro další práci s nimi a zabalí je do jiného souborového formátu. Jeho hlavním úkolem je manuální přidělení množiny obalových těles importovanému modelu.

Tato tělesa budou určena body a poloměry a uživatel s nimi bude moct jednoduše a přirozeně pohybovat ve 3D prostoru scény. Pomocí ovládácích prvků umožníme uživateli přidávat a odebírat kontrolní body obalovím tělesům.

Další potřebnou funkcí je vytváření pokojů, ze kterých se posléze generuje instance. Tento nástroj se bude jmenovat Room Editor a bude pracovat s již importovanými modely. Uživatel bude moci pomocí tohoto nástroje nadefinovat vlastnosti daného pokoje, přidat jednotlivé objekty do jeho scény a uložit tato data ve vlastním souborovém formátu.

Poslední důležitou funkcí je vytváření konfigurace jednotlivých instancí. Instance je množinou pokojů, u kterých je uveden maximální a minimální počet jejich výskytů, velikost mřížky a další důležitá data. Uživatel tak vytvoří nový soubor obsahující tato data a Instance Editor je uloží opět ve vlastním souborovém formátu.

## 7.1 Návrh grafického uživatelského rozhraní Editoru

Obecné nároky na vzhled tohoto nástroje:

- Přehlednost
- Srozumitelnost
- Dělení do logických sekcí

Sekce by měly být přímo graficky rozděleny tak, aby uživateli bylo ihned jasné, ve kterém nástroji se nachází a jaké jsou jeho možnosti.

Organizací nástrojů a informací by se měl nástroj podobat ostatním grafickým nástrojům, se kterými přicházíme běžně do styku při tvorbě 3D modelů.

Nástroje Room Editor a Importer vyžadují zobrazení 3D modelů, takže jejich hlavní částí bude OpenGL-kontext ve kterém se bude odehrávat většina činností uživatele, a jejich realizace bude nejobtížnější.

Sekce vytváření konfigurace instance vyžaduje pouze komponenty, které umožní výběr množiny pokojů a jejich nastavení. Takže zde žádný OpenGL-kontext nebude třeba.

Každá takováto aplikace také musí obsahovat vrchní kontextovou nabídku, ve které většinou bývají nástroje na otevírání a ukládání souborů, konfiguraci aplikace, nebo změny aktuálního nástroje.

### 7.1.1 Importer

Načte data modelu, uloží je do souborového formátu projektu, přidá k nim množinu obalových těles vytvořených uživatelem a nastaví příslušné hodnoty jako je jméno objektu a jeho ID. Podporovány budou dva druhy obalových těles:

- koule – určena bodem a poloměrem.
- kapsule – určená dvěma body tvořícími úsečku a vzdáleností od ní

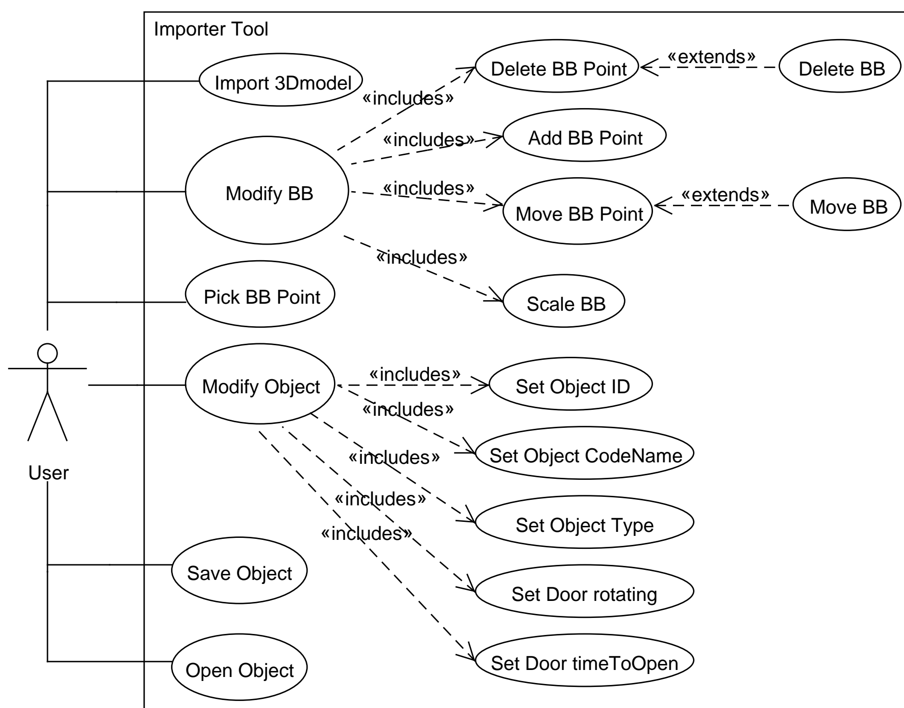
Nástroj by měl obsahovat container widget, do kterého se budou soustředit záznamy o datech importovaného modelu a kontrolních bodech obalových těles. Dále OpenGL-kontext, ve kterém bude probíhat samotná editace přidávaných obalových těles.

V tomto ohledu se inspiruji Qt Creatorem, ve kterém jsou vlastnosti objektů a ovládací prvky organizovány v `QTableWidgetu` (dále jen tabulky). Do této tabulky umístím všechna data a přidám k nim příslušné popisky.

Z požadavků se dá usuzovat, že kontrolní body obalových těles se budou muset jednoduše přemisťovat po scéně. Tento požadavek zajistím pomocí myši

a data aktivního kontrolního bodu vypíši do tabulky s aktuálními daty importovaného objektu. Tam také umístím ovládací prvky na přidávání a odebírání jednotlivých kontrolních bodů nebo celých obalových těles.

Vše by tedy mělo být na jednom místě přehledně seřazeno do skupin podle souvislostí s importovaným souborem, kontrolními body a dalšími funkcemi. Pohyb po scéně s importovaným modelem zajistím pomocí kláves WASD a pohybem myši.




Obrázek 7.1: Use case diagram nástroje Importer

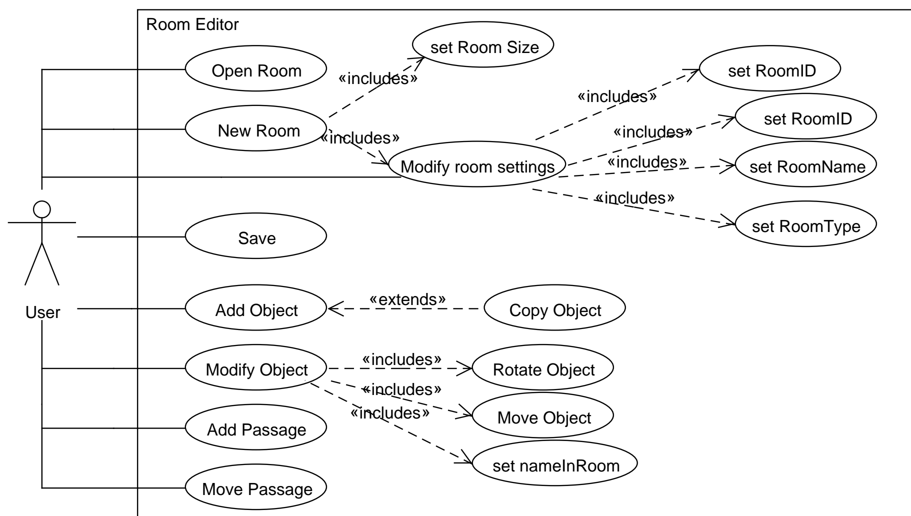
### 7.1.2 Room Editor

Pracuje již s importovanými 3D modely a seskupuje je do jednotlivých místností. Také pracuje s daty o daném pokoji:

- Rozměry
- Jméno
- ID pokoje
- Typ
- Místa, kde se dá přejít do dalšího pokoje

Opět by tento nástroj měl obsahovat tabulku a OpenGL-kontext, aby nástroje vypadaly jednotně a uživatel netápal.

V tabulce, stejně jako u Importeru, budou veškerá data o objektech ve scéně a obecné informace o pokoji. Navíc by se tam také měl nacházet strom již importovaných objektů seřazený podle typů objektů. Z něj by se měly jednotlivé objekty přidávat do pokoje metodou Drag and Drop.




Obrázek 7.2: Use case diagram nástroje Room Editor

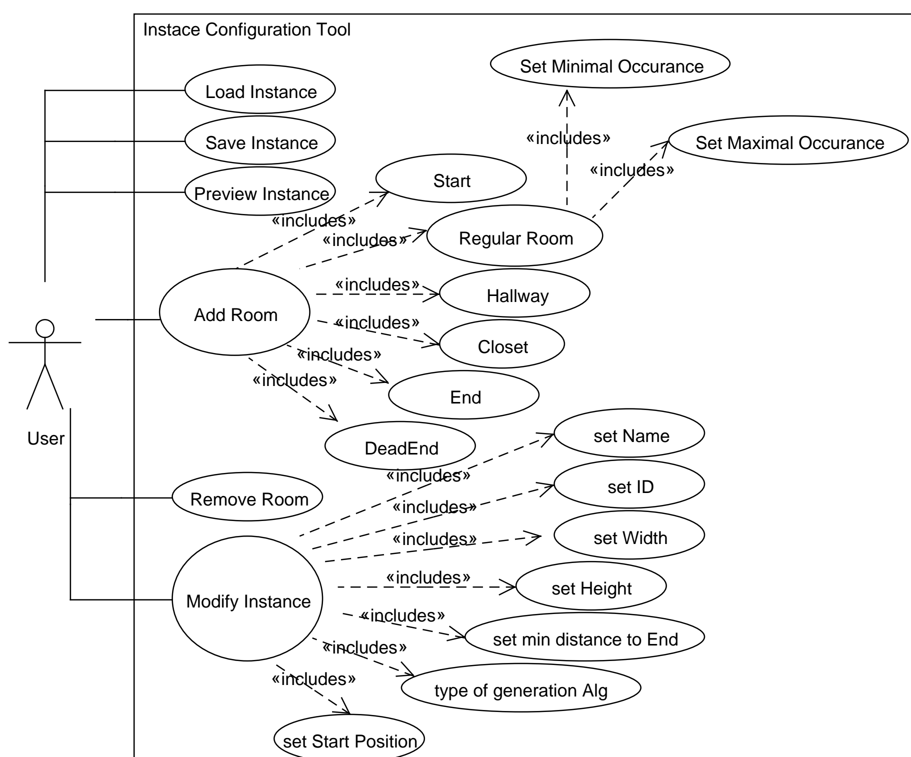
### 7.1.3 Editor Konfigurace instance

Třetí část editoru je vytváření takzvaných instancí. V tomto projektu je instance soubor pokojů a jejich povolených počtů. Pokoje se v mapě seskupí náhodně za sebe podle zadaných parametrů.

Jelikož zde již není žádný OpenGL-kontext a tato část se výrazně liší od Impoteru a Room Editoru, musím ji vyčlenit. Vytvořím ji tedy jako dialog, ve kterém uživatel vybere z vytvořených místností podmnožinu, která se pak uloží do souboru s konfigurací instance.

Měl by tedy obsahovat ovládací prvky pro přidávání a odebírání místností s dvěma container — widgety zobrazujícími množinu vybraných pokojů a pokojů které je možné do instance přidat. Tuto funkci bych zajistil stromy s adresářovou strukturou, kde budou pokoje rozděleny po složkách podle svých typů.

Pro modifikaci parametrů jako je velikost instance, pozici startovního pokoje a další funkce použiji, jako v jiných nástrojích, sadu Labelů, tlačítek, ComboBoxů a LineEditů.




Obrázek 7.3: Use case diagram nástroje pro konfiguraci instance

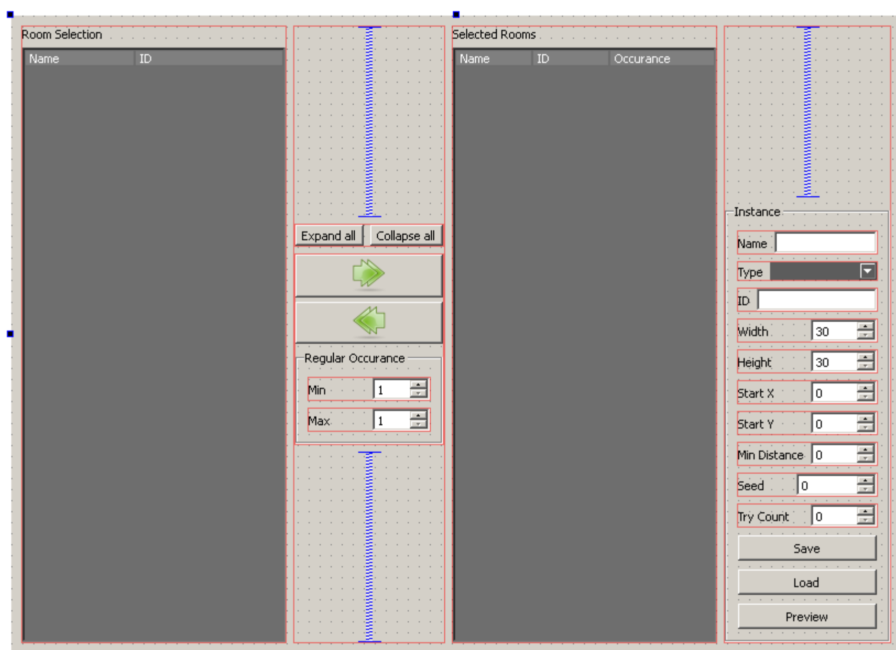
Kapitola 8

# Realizace

## 8.1 Implementace GUI pomocí nástrojů Qt

[1]

Qt IDE obsahuje takzvaný Qt Designer, který umožňuje vytváření GUI metodou „Drag and Drop“ a měnit vlastnosti jednotlivých widgetů jako v každém WYSIWYG — editoru. Funguje pouze jako základ pro tvorbu GUI, většinu složitějších prvků musí programátor sám implementovat. V Qt Designeru se dá rychle vytvořit funkční prototyp GUI aplikace a zbytek funkcí se doladí




Obrázek 8.1: Příklad práce s Qt Designerem – ConfigurationDialog

v kódu korespondující třídy.

Příklad na obrázku ukazuje jak může vypadat výsledek.

Tento vzor je z mé práce a jedná se právě o `InstanceConfigurationDialog`.

Jsou zde viditelné červené boxy, které znázorňují takzvané Layouty a modré prvky, které se nazývají Spacery. Takto vytvořený dialog bude správně reagovat na zvětšení, zmenšení a nebo přemístění okna právě díky Layoutům a Spacerům.

Krátce teď vysvětlím k čemu tyto prvky slouží a jak ovlivňují program za běhu.

Všechny widgety v Qt mají jistou geometrii a tou základní je minimální a maximální výška a šířka, vertikální a horizontální stretch, pozice a aktuální velikost. Stretch je důležitá hodnota, která udává, jak moc se bude widget v Layoutu zvětšovat či zmenšovat oproti ostatním.

Layoutů je více druhů a ve zkratce určují, jak se widgety umístěné v nich budou chovat, když se změní velikost okna nebo rozmístění či velikost jiných widgetů v tomto okně. Základní a nejpoužívanější v mé práci jsou Horizontal a Vertical Layout.

Horizontal Layout můžeme v ukázce vidět například na tlačítkách Expand all a Collapse all. Říká jim, že budou seřazeny horizontálně za sebe podle jejich geometrie. VerticalLayout je vidět například uprostřed. Zahrnuje celou skupinu widgetů, layoutů a spacerů, obsahující opět tlačítka Expand all a Collapse all. Tento Layout jim říká, že budou seřazeny vertikálně nad sebe opět podle geometrie.

Spacerů jsou dva druhy, Horizontal a Vertical spacer. Spacery oddělují skupiny widgetů, nebo widgety od sebe tak, aby mezi nimi vznikla viditelná mezera. Pokud by v příkladu nahoře nebyly, tak by se všechny widgety roztáhly tak, aby vyplnily celý layout, nebo na jejich maximální velikost danou geometrií.

V příkladu jsou vidět dvě tmavě šedivá pole. Tato pole jsou instance třídy `QTreeWidget`, klasické stromové struktury, jak je známe například z průzkumníku, ale zde jsou prázdné. Samotné skupiny a prvky do nich vkládám až v implementaci a dokonce některé z prvků vkládaných do této struktury jsou mého vlastního typu, který dědí od třídy `QTreeWidgetItem`.

Proč do nich vkládám své vlastní typy a nepoužívám `QTreeWidgetItem`? Protože když uživatel klikne na položku ve stromu, nebo klikne na tlačítko přidat, musím vzít příslušnou položku a například ji odebrat, nebo zavolat funkci logiky na přidání nového pokoje. Logika editoru pracuje většinou s ukazateli na datové struktury našich typů a tak se nehodí parsovat string z textové hodnoty položky a hledat ji v našich strukturách, když položka sama už může nést hledaný ukazatel.

V sekci pojmů jsem vysvětloval výhody plynoucí z metainformací o třídách, které umožňuje Qt využívat. Nejedna z nich je využívána i při tvorbě GUI.

V příkladu dialogu je vložené velké množství různých widgetů. Otázkou je, jak vlastně využívat tyto widgety v metodách související třídy. Dialog vy-

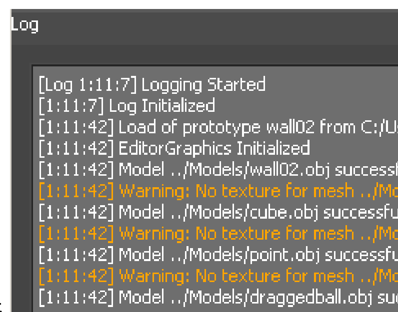
tvořený v Qt designeru je ve své podstatě pouze XML-soubor, který obsahuje všechna nastavení a vlastnosti přidaných widgetů. Při kompilaci se z něj vytvoří .cpp soubor, který se includuje do headeru naší třídy. To vše automaticky a na nás je tedy pouze tento kód využívat v naší třídě.

Přístup k widgetům je zajištěn přes proměnnou `ui`. Takže všechny widgety přidané pomocí Qt Designeru skrývá tato magická proměnná v podobě ukazatelů. Můžeme tedy rovnou v kódu měnit jejich nastavení a hodnoty.

Stejně tak jsem zmiňoval systém slotů a signálů, který se dá využívat pomocí metody `connect()` a nebo také pomocí takzvaných jmenných konvencí. To znamená, že pokud v našem UI máme widget se jménem `pushButton_klikni`, tak je automaticky propojen podobně jako u `connect()` se slotem naší třídy, který se musí specificky jmenovat `on_pushButton_klikni_clicked()`.

Tato vlastnost často usnadňuje práci při vytváření generických funkcí programu a Qt Designer má dokonce i kontextové menu, z kterého vytvoří daný slot automaticky. Ale pokud ji programátor nezná a nevědomky si pojmenuje slot podobně jako nějaký objekt v jeho ui, může to způsobit nemalé potíže a těžko se tato chyba hledá. Sám jsem se jí dopustil.

## 8.2 Třída Log




Tato třída slouží k obsloužení widgetu `QTextEdit`, který přijímá při líné inicializaci jako parametr metoda `init()`. Její hlavní funkcí je výpis akcí aplikační logiky a usnadnit uživateli řešení případných problémů. Většina hlášení je koncipována tak, aby jim porozuměl i uživatel a nejedná se tedy pouze o diagnostický nástroj.

Paralelně s výpisem do instance `QTextEdit` se výstup ukládá do textového souboru EditorLog.txt, kde se dají zpětně zjistit akce uživatele.

Výstup zobrazovaný instancí `QTextEdit` je ve formátu HTML, který umožňuje jednotlivá varovná či kritická hlášení barevně odlišit. Zvolil jsem tedy tři typy hlášení:

- Message
- Warrning
- Error

Každý z typů má jinou barvu, která se dá nastavit v headeru constants.h a platí, že čím je hlášení kritičtější, tím výraznější je jeho barva.

Třída funguje jako fronta, do které se střádají hlášení, ke kterým je přidán čas zařazení, a v zadaném intervalu se všechna vypíší.

## 8.3 Třída `Grid`

Konstruktor této třídy má dva parametry. Prvním je délka strany čtverce, který má mřížka vyplnit. Druhým je počet předělů na jednotkové vzdálenosti. Konstruktor pak připraví data, která se odešlou na GPU. Takto inicializovaná mřížka má svůj střed umístěný ve středu souřadného systému.

Před vykreslováním se pak pomocí setteru `setModelMatrix()` nastaví modelová matice určující umístění a natočení mřížky.

K samotnému vykreslení mřížky jsou zapotřebí další dva parametry, kde jedním je projekční, a druhým zobrazovací matice.

Tuto třídu v projektu používám pro znázornění vzdálenosti v nástrojích Importer i Room Editor. V nástroji Importer je také požita na znázornění roviny, ve které je přemisťován vybraný bod.

## 8.4 Třída `Camera`

Třídou řešící pohyb v trojrozměrném prostoru je třída `Camera`. Tato třída je použita ve všech částech Editoru obsahující OpenGL-kontext.

Obsahuje soubor dat potřebných k využití funkce `glLookAt()`, která vypočítá pohledovou transformaci ze světového souřadného systému do kamerového souřadného systému. Zjednodušeně zajišťuje pohyb uživatele po scéně.

Nejdříve tedy vysvětlím co pohledová matice vlastně je a začnu citátem z mého oblíbeného seriálu Futurama:

“The engines don’t move the ship at all. The ship stays where it is and the engines move the universe around it.”

Volně přeloženo: „Motor lodí vlastně vůbec nehýbe. Loď zůstává kde je a motor hýbe vesmírem kolem ní.“.

Přesně takto funguje i OpenGL. Pohledová matice pouze otočí a posune všechny objekty stejně, takže výsledek budí dojem, že se hýbe kamera a scéna zůstává na stejné pozici i když tomu tak vůbec není.

K výpočtu pohledové matice pomocí `glLookAt()` potřebuji:

- Pozici, kde se právě kamera nachází
- Bod vzdálený od kamery, na který se dívám
- Up vektor určující natočení kamery

Nevhodnou volbou Up-vektoru můžu dokonce docílit, že bod na který se dívám vůbec neuvidím.

Nyní přejdu k teorii pohledu, kterou jsem zvolil za vhodnou pro prototyp editoru.

Představme si, jaké možnosti dává výše popsaná technika výpočtu pohledové matice. Umožňuje takové Up-vektory, že otočíme celý pohled vzhůru nohama a nebo jej natočíme, jako když dáme hlavu k rameni. To by se možná v jisté míře hodilo při znázorňování lidského vnímání prostoru, ale na ovládání pomocí co nejmenšího množství kláves a na zajištění dobré orientace v prostoru je využití všech možností úplně nevhodné.

Rozhodl jsem se tedy Up-vektor omezit na pohyb v takové rovině, která obsahuje vektor směru pohybu, je kolmá na rovinu obsahující osy $XZ$, a úhel sevřený s touto rovinou bude z rozsahu $\pm\pi/2$ radiánů. Toto omezení zajistí pohyb kamery, jaký známe z her z pohledu první osoby.

K zajištění těchto omezení jsem se rozhodl, že třída bude obsahovat směrový vektor, pozici kamery, Up-vektor, horizontální a vertikální úhel ve sférických souřadnicích. Takto pokaždé, když uživatel stiskne klávesy na pohyby dopředu, dozadu, vlevo a nebo vpravo, mohu jednoduše jen posunout pozici kamery ve směru – proti směrovému vektoru – ve směru vektoru kolmého na směr pohybu a Up-vektor (či proti směru tohoto vektoru pro opačný pohyb) a vše je hotovo.

Přepočet rotace kamery je trochu složitější. Pokaždé když je rotace kamery aktivní a uživatel pohne myší, uloží se poslední pozice myši a při dalším pohybu se z této pozice spočítá rozdíl vzdáleností mezi posledním a aktuálním stavem.

Protože souřadnice, které získám z pohybu myši, jsou dvourozměrné a z omezení pohybu kamery vyplývá, že osa $X$ zobrazovacího zařízení bude vždy v rovině rovnoběžné s rovinou určenou osami $XZ$, mohu použít rozdíl pozice myši na ose $X$ jako inkrement horizontálního rotačního úhlu. Podobně tak pozice myši na ose $Y$ bude vždy odpovídat inkrementu vertikálního úhlu, protože rovina určená pozicí kamery, ve které leží směr pohybu a Up-vektor bude vždy kolmá na rovinu zobrazovacího zařízení.

V implementaci dále dělím rozdíl mezi pohyby myši konstantou, protože úhly jsou v radiánech, pozice v pixelech a tudíž pohyb myší o 10 pixelů otočil kameru o nepřijatelných 10 radiánů.

Pokaždé když dojde k inkrementaci některého z úhlů, musí se přepočítat jak Up-vektor, tak směr pohybu. Jinak by se dalo kombinací pohybů docílit nedefinovaného chování aplikace.

Horizontální a vertikální úhel je ve sférických souřadnicích. Jejich přepočet do kartézských souřadnic na směrový vektor se provede pomocí následujícího vzorce:

$$\begin{aligned} direction = (&\cos(verticalTilt) \cdot \sin(horizontalTilt), \sin(verticalTilt), \\ &\cos(verticalTilt) \cdot \cos(horizontalTilt)) \end{aligned} \tag{8.1}$$

To znamená, že pro nulový vertikální i horizontální úhel bude kamera směřovat rovnoběžně s osou $Z$.

Využívám zde vztahu pro výpočet pozice na kružnici, kde se střídají vertikální a horizontální úhel v závislosti na právě počítané složce, nazývaném mapování na kouli.

Nyní si analogicky spočítám takzvaný Right-vektor, který je kolmý na Up-vektor a směrový vektor:

$$Right(\sin(horizontalTilt - \pi/2), 0, \cos(horizontalTilt - \pi/2)) \tag{8.2}$$

$Y$ bude vždy konstantní díky omezující podmínce pohybu kamery. Nikdy se nedostaneme do stavu, kdy by right vektor neležel v rovině $XZ$ a $\pi/2$ se odečítá z důvodu rotace tohoto vektoru vůči směru pohybu. Kdyby se $\pi/2$ neodečetlo, jednalo by se o stejný vektor jako je směrový vektor, jenže ochuzený o složku $Y$.

Teď, když mám směrový vektor a Right-vektor, mohu přesně určit Up-vektor pomocí vektorového součinu, jehož výsledkem je vektor kolmý na oba vektory.

S takto připravenými daty mohu kdykoli, když si o to jiná třída řekne, přepočítat pohledovou matici tak, že použiji `glLookAt()` s hodnotami pozice kamery, Up-vektoru a bod, na který se dívám, vypočítám součtem pozice kamery se směrovým vektorem.

Moje implementace třídy `Camera` se chová líně co se týče přepočtu aktuálních hodnot při každé změně rotace a pohybu kamery. Pokud uživatel zavolá metody na pohyb kamery, tak se aktuálně přepočítá pozice a všechny vektory zůstávají stejné.

To je v pořádku, ale když uživatel inkrementuje rotaci kamery, pouze se změní příznak `m_needs_update` na `TRUE`. To znamená, že když si uživatel příště řekne o pohledovou matici, nejdříve se matice přepočítá a poté teprve vrátí hodnotu.

To ale také znamená, že když se inkrementuje rotace, směrový vektor a Up-vektor se nezmění, a tak následující volání pohybů ve směru směrového vektoru nebo Up-vektoru bude přepočítáváno podle starých směrů těchto vektorů.

Když ale vezmeme v úvahu, že při každém zobrazovacím cyklu si program řekne o aktuální pohledovou matici, což je každých 12–50ms, uživatel nemůže zaregistrovat jakkoli chybu ve funkci a ušetříme tím CPU nezanedbatelné množství výpočtů. Tím spíše není v silách uživatele zajistit aby se potkaly dva zobrazovací cykly, ve kterých se provede nejprve rotace, poté translace a chyba byla rozpoznatelná. Sekvenční dialog níže ukazuje, jak jsou jednotlivé uživatelské vstupy řešeny.

Jediný rozdíl mezi vstupem, který má za následek translaci pozice kamery a rotaci kamery, je ten, že translace kamery se volá při překreslování scény a rotace se průběžně inkrementuje pokaždé, když je to potřeba.

Důvod je prostý. Každá stisknutá klávesa se zaznamená do hash-tabulky a pokud je během zobrazovacího cyklu aktivní, tak se kamera přesune. Tento

přístup zajišťuje plynulost pohybu. Kdybych využíval opakovaného volání stisknuté klávesy, tak by se kamera musela pohybovat o větší krok, aby se uživatel vůbec někam pohnul. U rotace kamery toto není potřeba, protože události pohybu myší se objevují tak často, že rozdíl mezi jednotlivými pozicemi je často jen jeden pixel.

Tento způsob může způsobovat jisté nepřesnosti, ale pouze takové, které uživatel nemůže zaregistrovat.

## 8.5 Třída `OpenGLWindow`

Dalším tématem jsou veškerá okna sdílející jeden jediný OpenGL-kontext. Všechna okna zobrazující 3D grafiku v projektu jsou potomky třídy `OpenGLWindow` a jako třídní proměnnou mají výše zmíněnou třídu `Camera` a další dvě důležité proměnné, `m_context` typu `QOpenGLContext` a `m_animationTimer` typu `QTimer`.

První z nich, `m_context`, je statický,protože veškeré shader programy, `QVertexArrayBuffery` a jiné typy proměnných užívané ke komunikaci s GPU jsou striktně vázané na jeden kontext. Takže pokud bych chtěl recyklovat již inicializovanou třídu z Importeru v Room Editoru, musel bych veškeré prostředky na GPU uvolnit a znovu naalokovat při přemisťování třídy. To by mělo za následek nežádoucí režii.

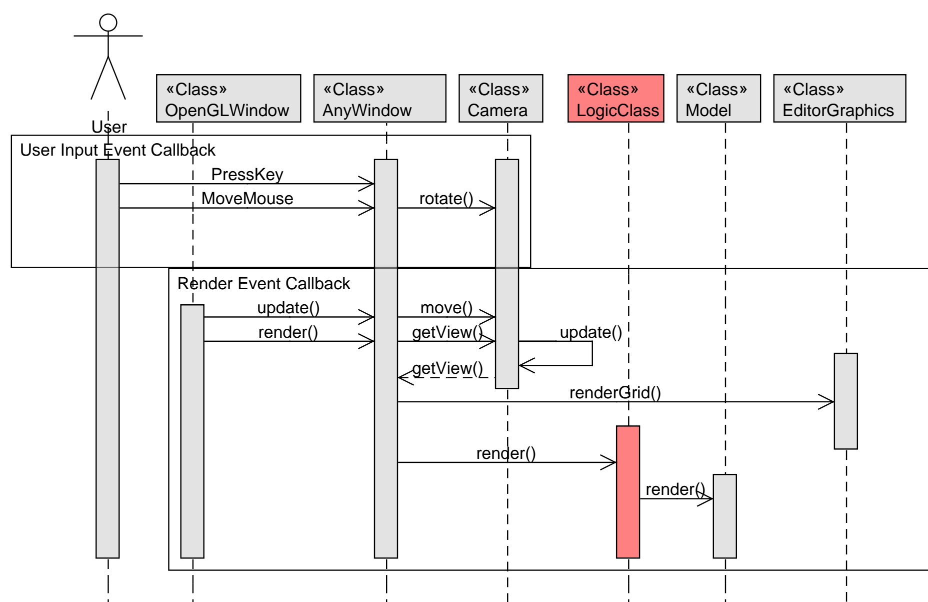
Proměnná `m_animationTimer` má za úkol volat překreslování kontextu v daném intervalu pomocí metody `renderer()`, která volá abstraktní metody `initialize()`, `update()` a `render()`.

Na začátku je interval v `m_animationTimer` nastaven na 12ms, což je 60 zobrazení za sekundu. V každém zobrazovacím cyklu se měří jak dlouhou dobu zabere přepočet a vykreslení scény a pokud zabere více než je interval timeru, tak se interval inkrementuje o 1ms.

Tento postup je nezbytný. Pokud bychom pořád dokola volali překreslování scény s intervalem 0, což je v podstatě stejné jako volat překreslení hned na konci každého zobrazovacího cyklu, došlo by k částečnému zablokování fronty událostí, kterou Qt využívá k řešení interních událostí, jako je například uživatelský vstup nebo překreslování GUI. Fronta událostí je v Qt sekvenční a následekem je totální nefunkčnost programu.

Jelikož je `m_animationTimer` třídní proměnnou každého okna, tak pokud máme aktivních více oken s různou zátěží, tak při přechodu z okna s vysokou zátěží do okna s nižší nezpůsobí přepnutí zbytečné snížení kvality zobrazování.

K diskuzi se nabízí, zda je správné řešit překreslování okna, přepočet dat a uživatelský vstup v jednom vlákně. Samozřejmě by bylo lepší separovat vykreslování grafiky od logických operací a zajistit tak plynulejší běh, jenže v prototypu editoru nepočítám s tak vysokou výpočetní náročností, aby se rozdíl projevil. U prototypu klienta budu určitě od samotného začátku koncipovat aplikaci jako vícevláknovou a přizpůsobím tomu její strukturu.




Obrázek 8.2: Sekvenenční diagram delegace událostí mezi třídami

## 8.6 Třídy `ImporterWindow`, `RoomWindow` a `PreviewWindow`

Tyto třídy obsluhují vždy příslušný OpenGL-kontext a odezvy na uživatelský vstup.

Všechny dědí od třídy `OpenGLWindow` a tudíž se chovají jako okna. `ImporterWindow` a `RoomWindow` pak používám v projektu jako widgety, tudíž k nim vytvořím container widget, který je ovládá. Dále přetěžují následující metody poděděné od `QWindow`:

- `keyPressEvent()`
- `keyReleaseEvent()`
- `mouseMoveEvent()`
- `mousePressEvent()`
- `mouseReleaseEvent()`

Jejich pomocí obsluhuji uživatelský vstup z klávesnice a myši. U obsluhy kláves a tlačítek myši se jedná o zaznamenání stisknuté klávesy do hash-tabulky obsahující dvojice v podobě identifikátoru a příznaku. Hodnota příznaku říká zda je klávesa stisknuta nebo uvolněna. Odezvy na stavy v hash-tabulce se řeší vždy před překreslením scény.

Odezvy na pohyb myši řeším okamžitě při změně pozice.

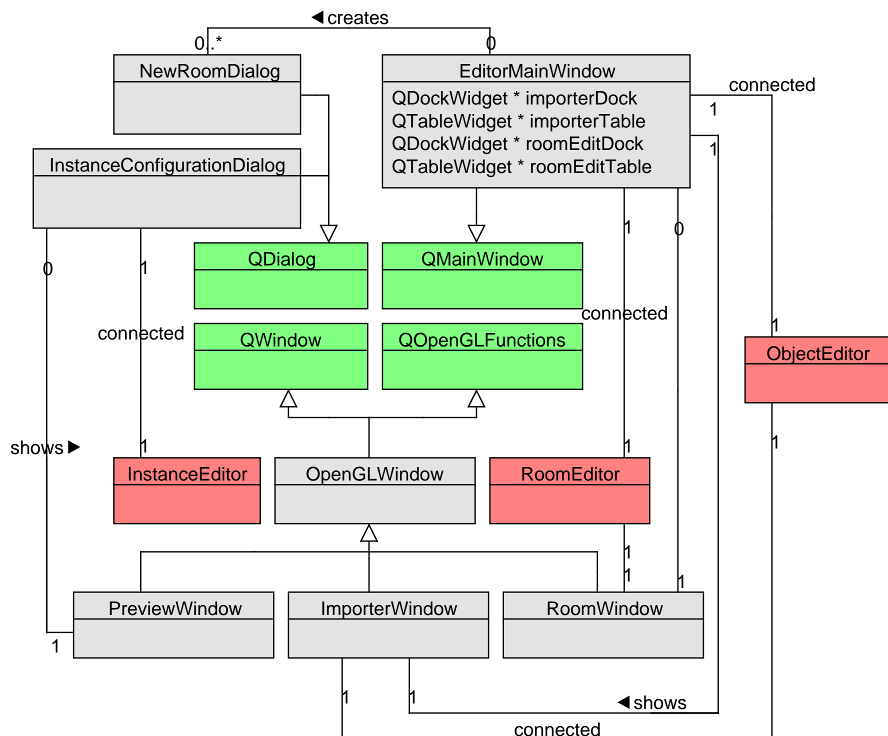
Důvod, proč se odezvy stisků kláves odehrávají před překreslením scény, je následující. Stisk klávesy vyvolá událost a tato událost se začne opakovat, až když uživatel drží klávesu déle. Interval, ve kterém se tato událost opakuje, záleží na nastavení systému. Pro pohyb po scéně by nebylo vhodné převádět událost stisku klávesy na inkrement pozice kamery, výsledný pohyb by byl trhaný.

Odezvy na změnu pozice kamery přicházejí vždy ve velice krátkém intervalu a zde se tyto události, díky jemnosti rozdílů mezi pozicemi, dají jako inkrement použít. Realizoval jsem tyto třídy odděleně, protože se jedná o části různých nástrojů a jejich funkce se v některých případech liší. Například `PreviewWindow` umožňuje uživateli pouze pohyb po scéně a nikoli interakci s ní. Naopak `ImporterWindow` a `RoomWindow` umožňují jak pohyb, tak i přidávání objektů do scény pomocí klávesových zkratek.

Každé přetížení výše zmiňovaných funkcí je tedy částečně unikátní a realizovat tedy tuto funkci pomocí jediné velké třídy je nevhodné.

Rozdíly v realizaci pohybu s body a objekty se budu věnovat na konci realizační části mé práce. Jedná se o částí `ImporterWindow` a `RoomWindow`.

Třídy také interagují s korespondující částí aplikační logiky, kterou dostávají v podobě ukazatele při vytváření nové instance a pomocí aplikační logiky vykreslují scénu do OpenGL-kontextu svého předka. Následující diagram zachycuje propojení jednotlivých tříd. Zelené jsou třídy Qt, červená je aplikační logika a šedé jsou mnou realizované třídy.




Obrázek 8.3: Diagram provázanosti tříd

## 8.7 Třída `EditorMainWindow`

Třída představující hlavní okno editoru. Jejími třídními proměnnými jsou veškeré widgety přidávané manuálně v kódu a také dvě proměnné typu `QWidget`, které jsou použity jako containery pro `ImporterWindow` a `RoomWindow` třídy.

Proměnné jsou typu `QWidget`, protože obě třídy dědí od třídy `QWindow`, tudíž se jedná o okna a měla by tedy být zobrazena zvlášť. To se mi nehodí a tak používám postup popsaný v dokumentaci Qt, který vytvoří z oken widgety, pomocí statické metody `QWidget::createWindowContainer()`. Widgety pak umístím do připravených Layoutů, takže budou umístěny v aplikaci přesně na svém místě.

Hlavní funkcí této třídy je spojit veškerou aplikační logiku a poskytnout jí data od uživatele. Obsahuje tedy třídy vytvořené kolegou, `ObjectEditor` a `RoomEditor`, kterým podle zrovna aktivního okna předává data z widgetů umístěných v `QDockWidgetech`.

Sama o sobě není tak zajímavá, ale spíše pracná, protože se zde odehrává kompletní inicializace GUI pro všechny nástroje a také obsluhuje veškeré jejich

odezvy pomocí příslušných slotů.

Kvůli přehlednosti jsem zde zvolil jmennou konvenci podobnou funkcím generovaným z Qt Designeru. Pro sloty obsluhující jednotlivé widgety je následující `jmenoNastroje_typWidgetu_jmenoWidgetu_TypUdalosti()`. Nejedná se tedy o typickou velbloudí notaci, ale kód je čitelný a na první pohled je vidět, co daný slot obsluhuje. Jinak je tomu u akcí, které jsou převážně volány z kontextových menu ve vrchní části aplikace.

Jejich jmenná konvece je následující `on_actionNazev_typAkce()`. Zdánlivě podobná konvence, ale akce nejsou vázané k žádnému nástroji, jsou tedy globální, takže se i konvence liší. Akce zde představují požadavky na změnu aktivního nástroje nebo vytvoření nového dialogu.

Zajímavě jsem zde vyřešil problém, který se objevil, když jsem začal pracovat s instancemi třídy `QTableWidget`, tedy tabulkami. V nich jsou umístěny veškeré informace a nástroje, které může uživatel používat, a dají se do nich umístit jak třídy `QTableWidgetItem`, tak veškeré třídy dědící od `QWidget`. Tabulky tedy obsahují jak tlačítka, ComboBoxy a různé další widgety, tak normální tabulkové záznamy. Je zbytečné zde používat widget `QLineEdit`, protože pouze znepřehlední tabulku a vyplatí se tedy použít již existující záznam typu `QTableWidgetItem`.

Jenže tyto záznamy se nedají omezit validátory jako `QLineEdit`. Abych omezil typ dat, které lze do jejich kolonek vepsat, musel jsem napsat funkci, která reaguje na typ záznamu, který kolonka dostane při inicializaci a příslušně upraví její data.

Typy jsou následující:

- `QTWI_UNCHANGEABLE`
- `QTWI_FLOAT`
- `QTWI_INT`
- `QTWI_TEXT`

Jmenná konvence je opět nasnadě. `QTWI` zkráceně znamená `QTableWidgetItem` a část za podtržítkem označuje typ.

Většina widgetů v Qt umožňuje ukládat do jejich struktur data libovolného typu pomocí třída `QVariant`. Tato data jsou ve třídě uložena v podobě řetězce bytů. Pokud známe typ dat, můžeme je pak jednoduchou konverzí z widgetu opět získat.

Pokud je typ neměnný, tak se při inicializaci nastaví hodnota variantu v příslušné instanci `QTableWidgetItem` na žádanou hodnotu a při každé změně se textová hodnota v kolonce změní zpět na hodnotu variantu. Tento typ používám pro veškeré kolonky v tabulce, které slouží pouze jako popisky.

Pro typy `FLOAT` a `INT` funkce nejdříve zjistí, zda je text v kolonce opravdu typu `integer` či `float`. Kontrola probíhá pomocí regulárních výrazů a pokud

hodnota odpovídá, nastaví se opět hodnota variantu na textovou hodnotu kolonky. Pokud nedopovídá, nastaví se textová hodnota kolonky na současnou hodnotu variantu.

Typ `TEXT` nepodléhá žádným omezením. Povoluje veškeré znaky.

Také zde rozpoznávám typy vlastních souborových formátů při akci `Open`. Pokud uživatel vybere v kontextovém menu akci `Open`, zobrazí se mu klasický `QFileOpenDialog` závislý na platformě. Uživatel vybere soubor, aplikace zkontroluje typ souboru a podle něj otevře příslušný nástroj či oznámí uživateli, že daný souborový formát není podporován.

Pokud uživatel zvolí akci `Save`, nebo `Save as`, program reaguje na aktivní nástroj. Aplikační logika je oboustranně propojena s touto třídou pomocí signálů a slotů. `RoomEditor` emituje signály informující o změně selekce, přidání objektu, smazání objektu a načtení nového souboru. Stejně tak `Importer`, který informuje o načtení modelu a změně aktuálních dat. Nemůže se tak stát, že se data v aplikační logice změní a uživatel nezaznamená změnu v GUI.

## 8.8 Třída `InstanceConfigurationDialog`

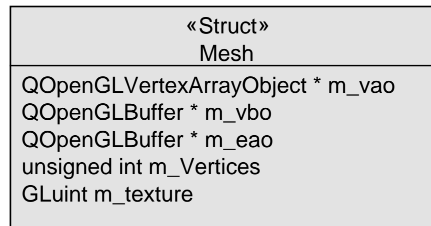
Veškeré widgety použité v této třídě jsou vytvořeny v Qt Designeru.

Při vytvoření nové instance třídy `InstanceConfigurationDialog` se vytvoří také instance aplikační logiky třídy `InstanceEditor`, která načte z adresářové struktury veškeré vytvořené validní místnosti. Tyto místnosti se uloží do stromu obsahující použitelnou množinu místností do jednotlivých složek podle jejich typů.

Přidávání místností do množiny vybraných pak probíhá poklepáním na příslušnou místnost ve stromu dostupných místností, nebo jejím výběrem a použitím tlačítka přidat. Analogicky se pak místnosti ze stromové struktury vybraných místností odebírají. Implementačně to znamená přesunout instanci `QTreeWidgetItem` z jedné stromové struktury do druhé a nebo její úplně smazat. Tyto uživatelské akce vyvolávají vždy reakci ve formě volání příslušných metod aplikační logiky instance třídy `InstanceEditor`.

Dialog nadále obsahuje tlačítka umožňující uložení a načtení dat ze souboru a widgety na modifikaci parametrů instance.

Navíc tento dialog umožňuje pomocí tlačítka Preview zobrazit okno `PreviewWindow`, ve kterém se zobrazí náhodně vygenerovaný náhled instance podle zadaných parametrů.




Obrázek 8.4: Struktura typu Mesh

## 8.9 Třída Model

Model je základní grafická třída využívající OpenGL k zobrazování svého obsahu. Jednotlivé modely, složené z částí, kterým se říká meshe, jsou při načítání rozděleny do stejnojmenných struktur. Ty obsahují proměnnou typu QOpenGLVertexArrayObject, která z pohledu OpenGL zahrnuje dva či více následujících bufferů. Když se při inicializaci modelu provede vše správně, stačí před vykreslováním zavolat metodu této proměnné bind() a OpenGL bude připravené na vykreslování daného objektu.

Zmíněné dva buffery jsou typu QOpenGLBuffer. Jeden obsahuje vertexy, normály, texturové souřadnice a druhý indicie používané na identifikaci jednotlivých stěn. Oba tyto buffery se při inicializaci alokují až po volání metody bind() instance třídy QOpenGLVertexArrayObject.

Konstruktor nastaví výchozí hodnoty některých proměnných a jméno souboru ze kterého se má načítat model na prázdný řetězec. Důležité je nastavit hodnotu proměnné m_fileName pomocí setteru ještě před voláním metody initialize(). Tato metoda kontroluje, zda není jméno načítaného souboru prázdné a poté se pokusí načíst daný 3D model.

Samo načítání je jednoduchý proces, kdy se pomocí knihovny Assimp vytvoří instance typu aiScene. Tato instance po načtení obsahuje veškerá data načítaného souboru.

Následně iteruji přes všechny meshe instance aiScene, vytvářím příslušné struktury v mé třídě Model a kopíruji data z assimpových struktur do vlastních. Postupně tato data přenáším do paměti GPU, kde pro každý typ shaderu nastavím správné atributy.

Pokud v assimpových strukturách najdu texturu asociovanou s daným meshem, načtu ji pomocí statické metody singletonu EditorGraphics, loadTexture(). Tato metoda vrací GLuint jako identifikátor textury v paměti GPU.

Jakmile se podaří úspěšně načíst data ze souboru, je instance třídy Model připravena k vykreslování.

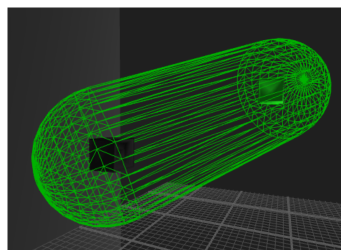
Protože se jedná o obecnou třídou editoru, představující 3D modelové grafické primitivum, je možné ji zobrazit v různých módech.

Třída `Model` využívá čtyři typy shaderů:

- `PHONG`
- `DRAGGED_BALL`
- `TOON`
- `PICK`

Phongův osvětlovací model počítá osvětlení per pixel a je jedním z nejpoužívanějších typů osvětlení. Jednoduše aproximuje přirozené světlo. Používám ho na zobrazování obalu scény a bodů.

Dragged ball je skupina shader-programů používaná pro deformaci speciálního modelu koule k zobrazení obalového tělesa tvaru kapsule, tedy tažené koule, jak napovídá název.



Obrázek 8.5: Příklad deformace koule pomocí dragged ball shaderu

Toon shader používá metody Cel Shadingu k docílení komiksového vzhledu modelu. Tímto shaderem se zobrazují všechny modely v editoru.

Pick shader se v modelu používá při vykreslování do off screen bufferu v módu picking. Před vykreslováním modelu se nastaví příslušné ID a on se pak vykreslí barvou zakódovaného ID.

Před vykreslováním instance třídy `Model` by se měla nastavit všechna data potřebná ke správnému vykreslení.

V jednom zobrazovacím cyklu se může metoda `render()` jedné instance volat vícekrát, zvláště u primitiv umístěných v singletonu `EditorGraphics`.

Metoda `setPickingPass()` uloží svůj argument do třídní proměnné `m_ID` a nastaví třídu do stavu určeného příznakem `m_picking_pass`. Pokud se třída nachází v tomto stavu při volání metody `render()`, tak dojde k použití picking shaderu. Protože je `m_ID` integer a barva je trísložkový vektor typu `float`, kde jsou všechny složky z rosahu 0–1, musím `m_ID` zakódovat do barevných složek RGB následujícím způsobem.

$$\begin{aligned} R &= (ID/(256 \cdot 256))/255.0f \\ G &= ((ID - R \cdot 256 \cdot 256)/256)/255.0f \\ B &= ((ID - R \cdot 256 \cdot 256 - G \cdot 256))/255.0f \end{aligned} \tag{8.3}$$

To zaručí přesné vykreslení dané barvy do bufferu. Následně při čtení hodnoty z bufferu na pozici pixelu pod kurzorem dostanu zpětným roznásobením a součtem opět ID objektu pod kurzorem. Tento způsob pickingu objektů není nejrychlejší, zabere celý jeden zobrazovací cyklus navíc a k tomu se musí před čtením z bufferu čekat, než GPU naplní celý buffer. Tento, i když pomalý, přístup pro potřeby editoru bohatě postačuje.

Samotné vykreslování pomocí funkce `render()` má jako argumenty tři matice

- Projekční
- Pohledovou
- Modelovou

Tato funkce předá hodnoty shader programům na GPU a iteruje přes všechny meshe modelu. V každé iteraci zavolá metodu `bind()` pro `OpenGLVertexArrayObject` daného meshe. Následně, pokud má mesh texturu, předá shaderu identifikátor textury a zbytek vykreslování provede GPU.

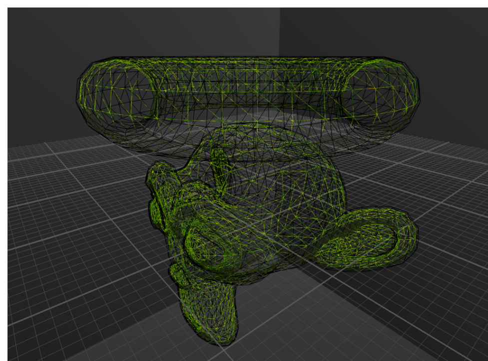
Zajímavou funkcí je u třídy `Model` wireframe-mód, který se dá zapnout pomocí příslušného setteru. V tomto módu se při vykreslování vypne culling, takže se budou vykreslovat všechny plochy modelu, bez ohledu na otočení k pozorovateli. Také se přepne `glPolygonMode( GL_FRONT_AND_BACK, GL_LINE)`.

Výsledek je takový, že OpenGL vykreslí všechny plochy pouze pomocí linek a vytvoří tak drátěný model. Tato možnost je přístupná v nástroji Importer, stačí zaškrtnout checkBox a načtený model se změní v drátěnou kostru, jak jí známe třeba z Blenderu.

## 8.10 Třída EditorGraphics

[1]

Je to singleton poskytující statické výpočetní metody, metody pro měření času, modely používané k vykreslování primitiv a předkompilované shader-programy sdílené všem grafickým třídám. V takto rozsáhlém projektu jsem se chtěl vyhnout globálním proměnným a singleton je tedy ideálním řešením.



Obrázek 8.6: příklad Wireframe-zobrazení

Tato třída se inicializuje při startu aplikace a hned v konstruktoru předkompiluje veškeré shader programy používané v projektu, které si při načítání modelu berou instance třídy `Model`. Také se inicializuje čas, kdy byla aplikace spuštěna a ukazatele na veškerá primitiva se nastaví na `NULL` hodnoty.

Modelová primitiva, jako jsou body obalových těles, koule a další, se inicializují až když jsou potřeba, nebo když je zavolána metoda `gInit()` této

třídy. Tato metoda inicializuje veškerý obsah instance třídy `EditorGraphics`, tento přístup se nazývá líná inicializace.

Hlavní funkcí jsou ale statické metody pro výpočty potřebné při operacích s modely.

Implementace těchto výpočtů je všeobecně známá a jedná se převážně o analytickou geometrii. Používám výpočty průsečíku paprsku s rovinou, také jsou zde metody na výpočet paprsku ve světových souřadnicích. Pro tento výpočet potřebujeme informace o projekční a pohledové matici se známou velikostí okna a souřadnice myši.

Za zmínku také stojí metoda `loadTexture()`, která má jako argument cestu k textuře a načte její data pomocí triku s Qt třídou `QImageReader` a `QImage`. `QImageReader` načte texturu a vrátí třídu `QImage`, ze které se pomocí statické metody třídy `QGLWidget::convertToGLFormat()` dají jednoduše získat načtená a předzpracovaná data pro použití v OpenGL.

`EditorGraphics` také umožňuje pomocí statické metody `renderPlane()` vykreslit rovinu se zadaným ID textury a modelovou maticí. Modelová matice umožňuje rovinu libovolně zvětšit, přesunout či natočit.

## 8.11 Cel Shading

[2] Cel Shading jsem implementoval pomocí dvoufázového vykreslování. Nejprve se vykreslí silueta objektu. Před prvním vykreslováním se použije `glCullFace(GL_FRONT)`. To nastaví OpenGL tak, aby zobrazovalo pouze ty polygony, které jsou odvrácené od pozorovatele. Poté se model vykreslí barvou siluety, v mém případě černou, a všechny jeho body se posunou ve směru normál o tloušťku siluety. To zajišťuji v shader-programu, kde jsou k dispozici všechny potřebné informace k této operaci. Při druhém vykreslování se nastaví `glCullFace(GL_BACK)`. To je výchozí hodnota, při které se zobrazují polygony otočené směrem k pozorovateli a vykreslí se barevný model v normální velikosti. Aby výsledek vypadal komiksově, tak při výpočtu osvětlení barevného modelu používám phongův osvětlovací model, ale výslednou sílu světla limituji skokovou funkcí.

$$Light = \frac{\lceil phongIntensity \cdot numberOfShades \rceil}{numberOfShades} \tag{8.4}$$

Tento postup zaručí maximálně takový počet přechodů z osvětlené do neosvětlené části modelu, jako je počet přechodů. Tato jednoduchá metoda vytváří iluzi komiksového obrazu, ale stále se potýká s jistými nedostatky.

Jedním z nich je to, že sice vykresluji siluetu objektu a limituji na něm osvětlení, což budí komiksový dojem. Jenže pokud textury objektu nejsou stylizovány do komiksové podoby, nebude tak vypadat ani výsledek zobrazení.

(a) Příklad limitování barev na fotografii

(b) Příklad herní komiksové grafiky

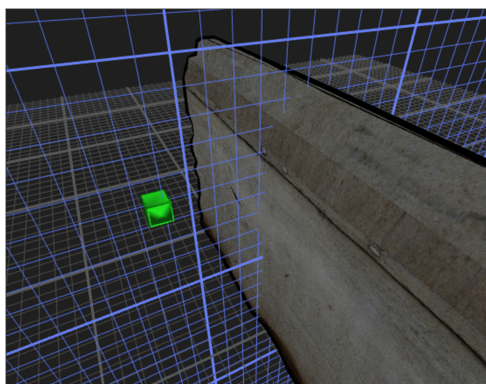
Toto se dá vyřešit metodou limitování barev podobně jako u fotek. Nejdříve se vše zobrazí s nelimitovaným osvětlením a siluetou a následně se na výsledek aplikuje limitace barev pouze na daný rozsah.

Tento postup ale produkuje výsledky, které se komiksové podobě současných her ani vzdáleně nepodobají a tudíž je žádoucí vytvářet objekty již v komiksové podobě.

## 8.12 Translace bodů a objektů pomocí myši

[3]

Při implementaci prvního prototypu nástroje Importer jsem se pouze snažil objevit chyby, kterých jsem se dopouštěl při zobrazování jednotlivých modelů a pohybu ve scéně. Navrhl jsem jednoduchý systém pohybování body, který sice fungoval, ale později se ukázal jako nevhodný.



Obrázek 8.8: Pohyb bodem

Vždy, když jsem označil bod a táhl myší, spočetl jsem diferenci předešlé pozice myši a aktuální pozice a o tuto diferenci jsem posunul bod v rovině kolmé na směrový vektor kamery. Metoda fungovala, ale když jsem se později pokusil s nástrojem pracovat, ukázalo se, že přesunout bod přesně na cílovou pozici je takto téměř nemožné kvůli nepřehlednosti pohybu bodu ve scéně.

Také s vyšší vzdáleností bodu od kamery se zvětšoval rozdíl v posunutí kurzoru po okně a posunutí bodu v dáli, které bylo téměř nepatrné.

Navrhl jsem tedy nový způsob pohybu bodem založený na výpočtu směru paprsku z pozice kamery. Směr paprsku se spočte ze souřadnic na obrazovce inverzní projekční a pohledovou transformací. Postup je následující. Naleznu osu, od které je směrový vektor kamery nejméně odchýlen. Vytvořím rovinu určenou bodem se kterým se pohybuje a touto osou. Naleznu dva průsečíky paprsků, kde oba mají počátek v pozici kamery a směr určený přepočtem pozice myši na paprsek do scény ve světových souřadnicích. První paprsek je přepočten z předešlé pozice myši a druhý z aktuální pozice myši. Rozdílem průsečíků získám výslednou translaci bodu v této rovině.

Tato metoda zajistí přesný a plynulý pohyb bodu myší a zvýší přehlednost této operace. Takový přístup je pro uživatele přirozenější a zvyšuje přesnost pohybu bodů po scéně.

V nástroji Room Editor jsem zvolil stejnou metodu, jenom hledám vždy průsečíky s rovinou určenou osou $Y$ a pozicí taženého objektu. Rozhodl jsem se tak po testování, kdy jsem vytvářel vzorová data a zjistil jsem, že uživatele v Room Editoru přepínaní mezi režimy tří rovin spíše brzdí.

Na pohyb objektů po ose $Y$ jsem v tomto nástroji zvolil jiný přístup, kdy rovina, se kterou se hledají výše zmíněné průsečíky, je definována směrovým vektorem kamery ochuzeným o složku $Y$. Z translačního vektoru mezi průsečíky se pak k výsledné translaci použije jen složka $Y$.

Při dalším testování jsem pak dospěl k názoru, že podporovat možnost pohybu po jedné ose by bylo dobré jak v režimu Importeru tak Room Editoru po všech osách a doplnil tedy její plnou podporu.

Zvětšení objektu v Room Editoru a poloměr určující obalové těleso v Importeru jsem se rozhodl počítat také technikou paprsků vrhaných do scény z pozice kamery. V tomto případě je rovina určena směrovým vektorem kamery a zvětšení nebo poloměr jsou rovny délce vektoru mezi průsečíky.

Kapitola 9

# Testování

Vzhledem ke grafické podobě aplikace bylo nutné kontrolovat výsledky testů vizuálně. Forma testů byla převážně uživatelská. Při testování triviálních částí systému jsem také využíval Log, do kterého jsem vypisoval akce aplikace a kontroloval jejich správnost.

Při testování částí aplikace využívající technologii OpenGL jsem vytvářel vlastní testy, které indikovaly špatné výstupy pomocí zbarvení celé scény jedinou barvou. GPU je uzpůsobené na přijímání dat z CPU a předávání dat zobrazovacímu zařízení. Zpětné získávání dat vypovídajících o výpočtech či datech, které zpracovává, není standardní postup.

Scénáře pro testování spojovaly aplikační logiku i grafickou část aplikace. Jednalo se o komplexní testy funkčnosti. Nyní popíši podobu scénářů testů.

- Nástroje Importer a Room Editor
  - Načtení 3D modelu ze souboru / Vytvoření nového pokoje
  - Vložení několika obalových těles / objektů
  - Výběr a pohyb s obalovými tělesy / objekty
  - Smazání několika obalových těles / objektů
  - Uložení importovaného 3D modelu / pokoje
  - Načtení importovaného 3D modelu / uloženého pokoje
- Nástroj Konfigurace Instance
  - Výběr množiny pokojů instance
  - Nastavení parametrů
  - Zobrazení náhledu
  - Uložení instance
  - Načtení instance

- Zobrazení náhledu

Nejdůležitější částí testovacího procesu bylo zatížení systému velkým objemem dat a vytváření vzorových dat pro demonstraci funkčnosti systému. Vytváření vzorových dat mi pomohlo doladit aplikaci do finální podoby.

# Závěr

Aktuální stav projektu hodnotím jako úspěch. V krátkém časovém rámci jsem se naučil pracovat s novými technologiemi a s kolegou jsme dokázali vytvořit funkční a uživatelsky příjemný prototyp editoru. Znalost těchto nových technologií je nedocenitelnou zkušeností a jsem si jist, že se uplatní i nadále.

Jelikož jsme museli s kolegou spolupracovat jako malý tým, získal jsem také zkušenosti týkající se koordinace týmové práce. Ke konci implementační části jsme již opravdu efektivně spolupracovali a doufám že v tomto duchu budeme i pokračovat.

Návrh i implementační struktura projektu je čistá, snažili jsme se zachovávat veškeré konvence, aby byl projekt čitelný a dále rozšiřitelný. Bakalářská práce poslouží jako základní kámen projektu, ve kterém hodláme pokračovat.

Během návrhu jsme stanovili, o jakou další funkčnost se musí editor rozšířit, a při finálním testování se objevilo pár chyb v návrhu, které budou muset být v nejbližší době odstraněny. Tyto nedostatky se týkají přímého propojení aplikační logiky s grafickou částí, která pak nemůže být realizována v samostatném vlákně a nejdou tak uplatnit některé optimalizační metody.

V následující fázi projektu se tedy budou muset tyto návrhové chyby odstranit a do projektu integrujeme skriptovací jazyk pro obsluhu uživatelem nadeklarovaných událostí a reakcí. [**?**]

# Literatura

[1] Trolltech: *Qt Project Documentation*. Dostupné z: http://qt-project.org/doc/

[2] Nvidia: *The Cg Tutorial*. Dostupné z: http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html

[3] OpenGL, O. E.; community, W.: *OpenGL Insights*. CRC Press, Červenec 2012.

[4] Khronos Group: *OpenGL 4 Reference Pages*. Dostupné z: http://www.opengl.org/sdk/docs/man/

[5] Assimp Development Team: *Assimp Documentation*. Dostupné z: http://assimp.sourceforge.net/lib_html/index.html

Příloha A

# Seznam použitých zkratek

**CAD** Computer-aided Design

**DPS** Damage per second

**GPU** Graphic Processing Unit

**GUI** Graphical User Interface

**GLSL** OpenGL Shading Language

**IDE** Integrated Development Environment

**MMO** Massive Multiplayer Online

**PvE** Player versus environment

**PvP** Player versus player

**QML** Qt Meta Language - User Interface Markup Language

**RPG** Role Playing Game

**RotMG** Realm of the Mad God

**SDL** Simple DirectMedia Layer

**UI** User Interface

**XML** Extensible markup language

Příloha B

# Obsah přiloženého CD

```
readme.txt ................................. Stručný popis obsahu CD
Documentation ....... Doxegenová dokumentace s uživatelskou příručkou
Executables ............ Spustitelná forma implementace s knihovnami
Models ........................... Vzorové modely a grafická primitiva
Sources
  Implementation ......................... zdrojové kódy implementace
  Thesis ..................... zdrojová forma práce ve formátu LaTeX
text
  BP - Lukáš Vilím 2013-2014.pdf ...... text práce ve formátu PDF
```