Západočeská univerzita v Plzni
Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

DIPLOMOVÁ PRÁCE

Plzeň, 2015

Kamil Rendl

Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Diplomová práce

Simulátor optických jevů

Plzeň, 2015 Kamil Rendl

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: **2014/2015**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Kamil RENDL**
Osobní číslo:	**A13N0123P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Název tématu:	**Simulátor optických jevů**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování:

1. Seznamte se s běžnými programy pro simulaci optických jevů paprskovou optikou a typickými úlohami, které řeší. Zaměřte se zejména na simulaci jevů souvisejících s holografií.
2. Navrhněte architekturu jádra interaktivního simulátoru, který bude umožňovat simulaci paprskové optiky ve 2-D i v 3-D, simulaci záznamu a nasvícení difraktivní struktury, bude uvažovat obecné optické členy (několik paprsků vstupních, několik paprsků výstupních), bude uvažovat základní fyzikální vlastnosti světla (energie, spektrum, polarizace, koherence).
3. Implementujte jádro simulátoru s ohledem na rozšiřitelnost a dlouhodobou udržovatelnost kódu. Simulátor by měl v ideálním případě fungovat jak v podobě webové (online), tak v podobě desktop (offline) aplikace.
4. Navrhněte architekturu grafického uživatelského rozhraní interaktivního simulátoru. Uživatelské rozhraní musí umět vytvořit, editovat, analyzovat a vizualizovat několik vzájemně souvisejících optických soustav (např. soustava pro záznam a soustava pro rekonstrukci hologramu), vizualizovat a analyzovat chod paprsků, umožňovat souborový vstup a výstup.
5. Implementujte základní podobu grafického uživatelského rozhraní umožňující využívání funkcí simulátoru.
6. Funkčnost implementace ověřte na realistických i netriviálních umělých optických soustavách.

Rozsah grafických prací: **dle potřeby**

Rozsah pracovní zprávy: **doporuč. 50 s. původního textu**

Forma zpracování diplomové práce: **tištěná**

Seznam odborné literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce: **Ing. Petr Lobaz**
Katedra informatiky a výpočetní techniky

Datum zadání diplomové práce: **1. září 2014**

Termín odevzdání diplomové práce: **14. května 2015**

Doc. RNDr. Miroslav Lávička, Ph.D.
děkan

L.S.

Západočeská univerzita v Plzni
Fakulta aplikovaných věd

Prof. Ing. Jiří Šafařík, CSc.
vedoucí katedry

V Plzni dne 15. září 2014

Prohlášení

Prohlašuji, že jsemdiplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 10. dubna 2015 ……..

Kamil Rendl

Abstrack

A new implementation of an optical simulator is presented. It is able to simulate behavior of light rays in an optical system. Its main goal is to simulate optical systems used for hologram recording and reconstruction.

The old version of the simulator was able to simulate rays in 2-D only and had several drawbacks. The new version has completely new, more flexible software architecture that allows to simulate rays in 3-D and allows easier optical system setup.

The thesis describes details of the new software architecture.

Poděkování

Tímto bych chtěl velice poděkovat ing. Petru Lobazovi, vedoucímu celého projektu,za jeho pomoc při řešení mnoha problémů, které se při vývoji a studiu vyskytly. Bez jeho pomoci a vstřícného přístupu bych jen velice těžko dokončil celou práci.
Také bych chtěl poděkovat slečně Lucii Herejtové, se kterou jsem spolupracoval při vývoji celého projektu. Na základě jejích zkušeností a inovativníchmyšlenek bylo v projektu realizováno mnoho různých vylepšení, za což jí velice děkuji.

Obsah

1 Úvod

Před několika lety byl založen projekt, který měl za cíl vytvořit výukový program (simulátor), jehož pomocí by se dal vizualizovat paprskový model holografie tak, aby bylo uživateli umožněno sledovat chování jednotlivých světelných paprsků, například při ohybu paprsků na difrakčním záznamovém materiálu(holografické desce). V následujících letech byl celý projekt úspěšně rozšiřován o další užitečné vlastnosti. Jelikož jde vývoj neustále dopředu, je nutné i tento projekt stále rozvíjet. Proto na základě uživatelských ohlasů bylo rozhodnuto, že celý vizualizační program, který byl až dosud realizován pouze v 2Dprostoru,bude převeden do 3D prostoru.

Tím vznikla otázka, zdali i nadále pokračovat ve vývoji stávajícího simulátoru, nebozda založit nový projekt, ve kterém budou využity veškeré, dosud získané poznatky. Po detailnějších úvahách se rozhodlo, že bude vytvořena nová verze simulátoru.

Celá tato práce je zaměřena napopis klíčových kroků, které byly podniknuty v rámci vývoje nového simulátoru pro vizualizaci paprskových modelů holografie.

V úvodních kapitolách jsou probrány různé simulátory, které se zabývají problematikou vizualizace paprskového modelu. Zároveň je zde uveden popis simulátoru, který se stal vzorem pro vývoj tohoto projektu a na jehož poznatcích byly realizovány mnohé vlastnosti.

Kapitoly 4,5,6,7 a 8 se zaměřují především na implementační problematiku celého projektu. Jsou zde popsány jednotlivé objekty a jejich vlastnosti, s tím, že je zde kladen důraz na popis klíčových metod.Součástí je také popis struktury celého projektu, kde jsou zároveň vysvětleny jednotlivé vazby mezi objekty. Závěrem této části je detailní výklad o implementaci nového optického členu, jenž je současně demonstrován na příkladu.

V poslední kapitole celé práce je popsán průběh testování, při kterém byla vytvořena jednoduchá holografická sestava. Součástí kapitoly je obrázková ilustrace, která doprovází výklad a umožňuje tak čtenáři udělat si přesnou představu o vytvořeném simulátoru.

2 Paprskové simulační programy

Před začátkem realizace celého projektu byl proveden průzkum různých programů a simulátorů, které by mohly být vhodné pro naše požadavky. Při vyhledávání bylo nalezeno několik obdobných simulátorů. Po detailnější analýze však byly zavrhnuty. Při ověřování jednotlivých vlastnostíbyla zaměřena pozornost především na možnosti sledování pohybu jednotlivých paprsků při simulaci, uživatelskou přívětivost a celkovou funkčnost.

2.1 Požadavky na simulátory

Jak již bylo zmíněno výše, základním stavebním kamenem pro vyhodnocování programů se stal fakt,želze snadným způsobem provést zkoumání pohybu a vlastností jednotlivých paprsků. Jednalo se především o možnost sledovat dráhu pouze jednoho, nebo několika málopaprsků, které vyšly ze zdroje světla a směřovaly do scény s ostatními objekty.

U jednotlivých optických prvků v simulátorechbyla stěžejní vlastností možnost sledovánílomu paprsku při přechodu do jiného prostředí, odraz paprsku od předmětu (úhel, směr, atd.),rozdělení paprsku a různé další metody, které ovlivňují chování a směry paprsků. Možnost pozorovat tyto aspekty a případně ovlivňovat jejich vlastnosti a chování, byl jednoznačněnejkritičtější parametr při vyhodnocování jednotlivých programů.

Další, neméně podstatnou vlastností, kterou jsem také zkoumal, byla možnost sledovat jednotlivé vlastnosti paprsků jako je vlnová délka, energie, spektrum, apod.

Kritérium pro uživatelskou přívětivost, snadnou obsluhu a přehlednost celé simulace již nebylo tak přísně vyžadováno jako u předchozích bodů, ale bylo k ní taktéž přihlédnuto.

2.2 Ray Optics Simulation

Ray Optics Simulation je simulátor optických paprsků, který je vyvíjen jako volně dostupné rozšíření webového prohlížeče Chrome.Při testování byl využit prohlížeč Chrome ve verzi 42.0.23 a Ray Optics Simulation ve verzi 2.0.

Po spuštění simulátoru se zobrazí základní okno, ve kterém je možné provádět jednotlivé simulace. Při horním okraji simulátoru je umístěno jednoduché a přehledné menu, pomocí kterého lze snadno přidávat jednotlivé optické prvky do simulované scény.

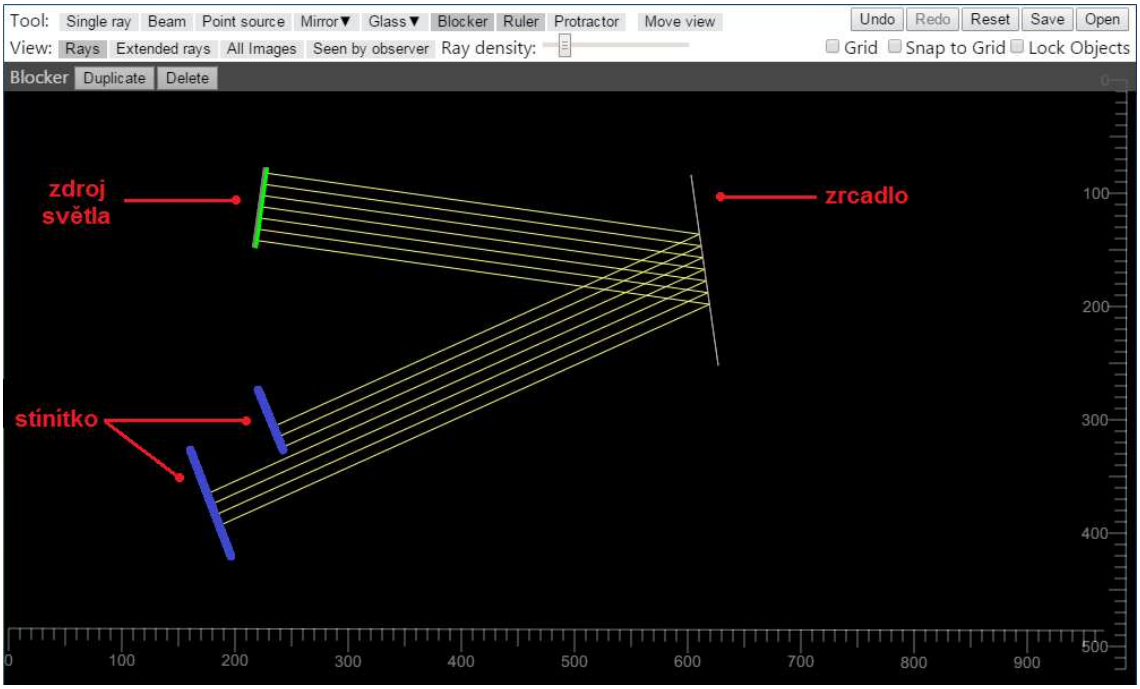
V rámci testování byla vytvořena v simulátoru jednoducháoptická sestava, která se skládala ze světelného zdroje (vyzařoval rovnoběžné paprsky), zrcadla a dvou stínítek. Při vytváření celé scény se velice snadno manipulovalo s jednotlivými optickými

prvky.Nastavení celé scény proto nebylo nikterak velkým problémem. K tomu také přispěl fakt, že simulátor pracuje pouze ve 2D prostoru.

Ve chvíli, kdy byla vytvořena celá simulace, byla pozornost zaměřena především na možnost sledování chování jednotlivých paprsků. Při této analýze jsem dospěl k závěru, že simulátor velice dobře realizuje chování (úhel, směr, …) a vykreslování jednotlivých paprsků. Oproti tomu při snaze sledovat vlastnosti jednotlivých paprsků jako jsou vlnová délka, spektrum, apod., bylo zjištěno, že simulátor tyto vlastnosti vůbec nenabízí.

O uživatelském rozhraní by se dalo říci, že je velice jednoduché a přehledné. Jednoduchost a přehlednost jsou však vykoupeny celkovou nízkou funkčností, kterou simulátor nabízí.

Na obrázku 2.1 si lze prohlédnout celou simulaci, kterou jsem při testování vytvořil. Zároveň je zde také vidět jednoduché menu, které ukrývá veškerou funkcionalitu celého simulátoru.



Obrázek 2.1 Ray Optics Simulation

2.3 VirtualLab 5

VirtualLab 5 je rozsáhlý program zabývající se simulací chování optických paprsků. Program vyvinula společnost LightTrans GmbH sídlící v Německu. V rámci testování mi byla společností LightTrans GmbH poskytnuta 30-denní testovací trial verze, kterou jsem následně otestoval. Konkrétně se jednalo o verzi simulátoru VirtualLab 5.11.1

Po spuštění programu se zobrazí rozměrná úvodní obrazovka, v níž se následně provádějí jednotlivé simulace. Po spuštění uživatel již na první pohled pozná, že se jedná o velice rozsáhlý program.

Pro otestování chování simulátoru jsem vytvořil jednoduchou optickou sestavu, která se skládala ze zdroje světla (laserový paprsek), zrcadla a clony. Jelikož jsem s tímto programem přišel do styku poprvé, vytvoření této jednoduché scény zabralo poměrně dost času. Tento fakt byl způsoben především velkým množstvím možností, které simulátor nabízí pro nastavení jednotlivých optických prvků.

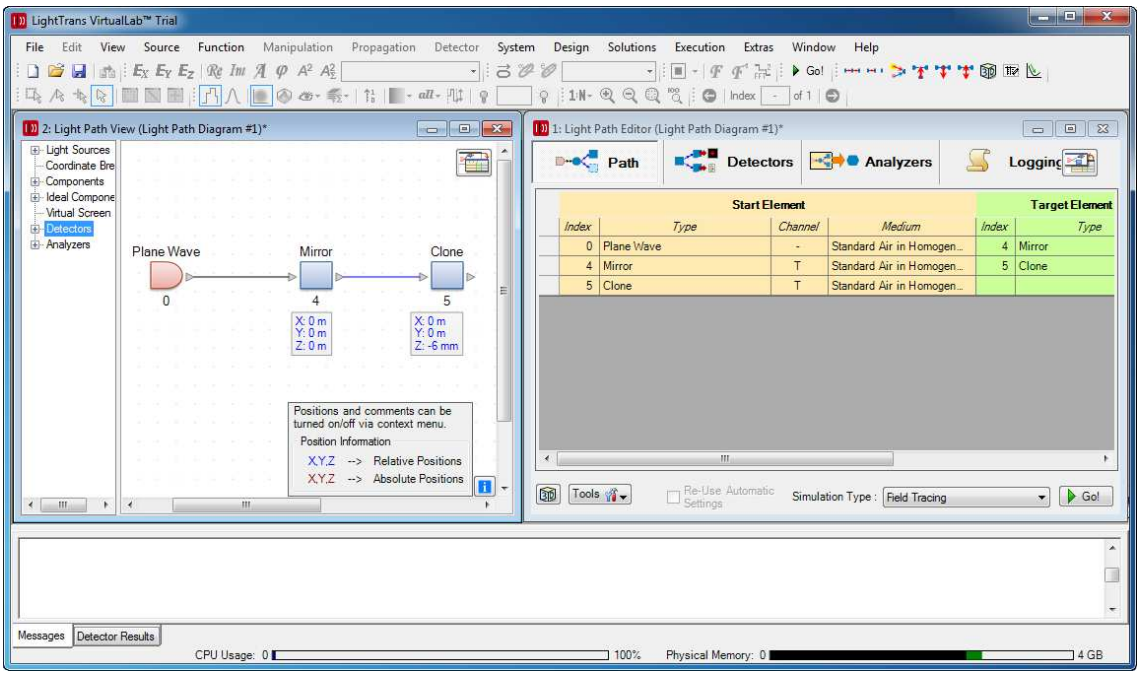
Před spuštěním samotné simulace je nutné, aby uživatel určil posloupnost jednotlivých optických členů, které se budou vzájemně osvětlovat. Na základě této posloupnosti je poté spuštěna celá optická simulace.

Možnost sledovat směry a pohyby jednotlivých paprsků, jak putují mezi optickými prvky, v tomto programu bohužel nejde. Na druhou stranu program umožňuje sledování jednotlivých detailních hodnot paprsků jako je vlnová délka, spektrum, apod.

Ohledně uživatelské náročnosti se mi tento program jeví jako velice komplikovaný a dosti nepřehledný pro jednoduché simulace.

Závěrem bych program zhodnotil takto: simulace jednotlivých vlastností a paprsků je na výborné úrovni. Ovšem vizualizační možnosti jednotlivých optických prvků a paprsků jsou naprosto nedostačující pro hledané účely. Složitost a mnoho různých funkcí, které simulátor nabízí, řadí tento simulátor spíše do kategorie pro uživatele, kteří přesně vědí, jaké sestavy chtějí simulovat a jak se budou optické sestavy chovat. A především proto se jeví tento program jako nevhodný pro výukové účely.

Na obrázku 2.2 je vidět hlavní okno simulátoru.



Obrázek 2.2 VirtualLab 5

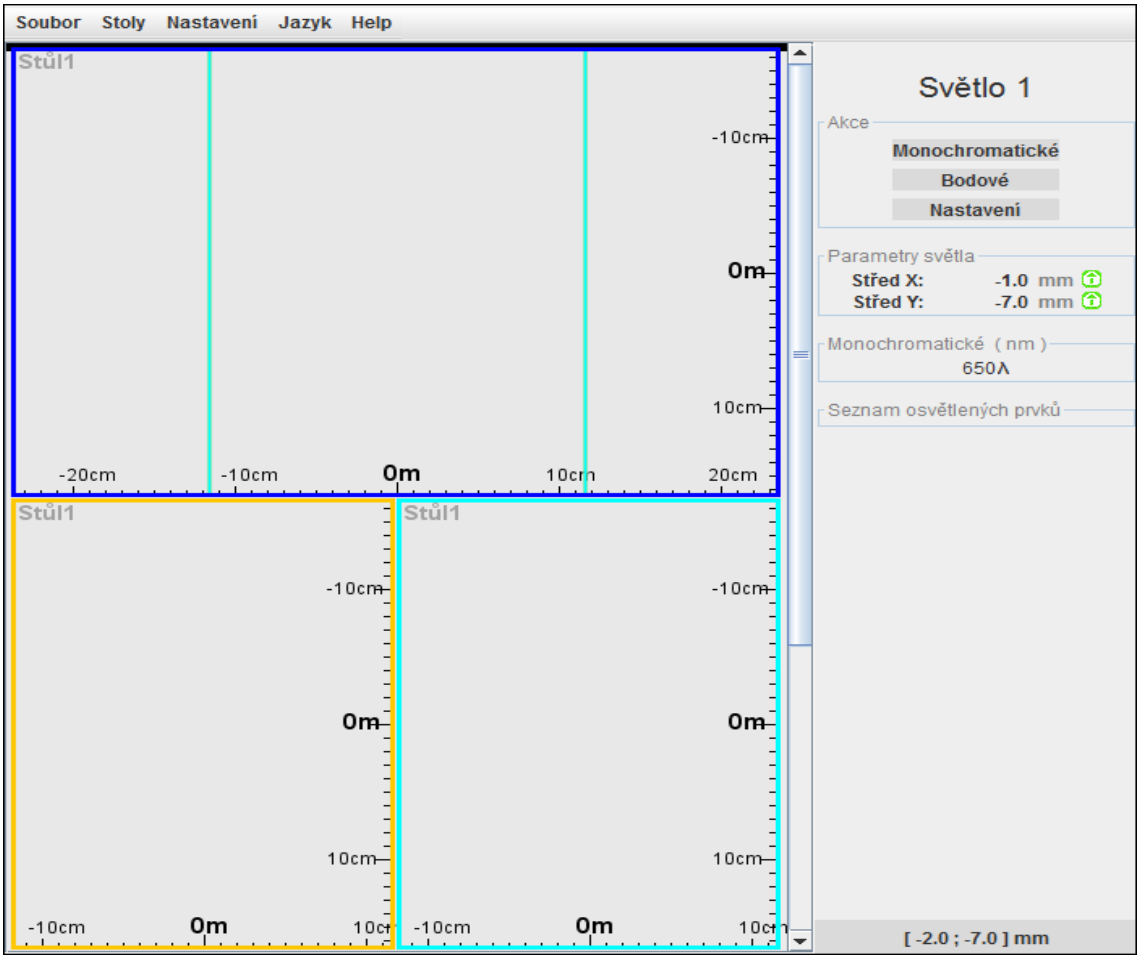
2.4 Holografický simulátor

Jedná se o simulátor, který byl vyvinut přesně pro účely, které nyní hledáme. Bohužel zatím pracuje pouze ve 2D prostředí.Simulátor byl vytvořen v rámci mé bakalářské práce„*Vizualizace principu hologramu*“ [4]. Následně byl studentemZápadočeské univerzity Michaelem Hadáčkem rozšířen v rámci jeho bakalářské práce „*Vizualizace principu hologramu*“ [5].

Simulátor umožňuje přesné sledování pohybu paprsků a zpětnou analýzu jejich chování.

Co se grafické a uživatelské přívětivosti týká, je program na dobré úrovni a lze se jím v některých částech inspirovat pro případný další vývoj. Na níže uvedeném obrázku 2.3 lze vidět grafickou koncepci jednotlivých prvků grafického uživatelského rozhraní.

Při návrhu a následném vývoji tohoto simulátoru se již předem počítalo s budoucím rozšířením scény z 2D do 3D. Proto by nebyl velký problém rozšířit tento program do 3D. Ovšem jak se pozvolna projekt vyvíjel, docházelo k objevování mnoha úskalí a problémů, které nebyly dobře řešeny, především z pohledu další rozšiřitelnosti projektu. Proto bylo rozhodnuto, že dalšírozšiřováníjiž nemá výrazný přínos, co se budoucnosti týká.



Obrázek 2.3 Vzhled holografického simulátoru

2.5 Získané poznatky

Shrneme-li si poznatky z jednotlivých prozkoumaných simulátorů, zjistíme, že Ray Optics Simulator má jednoduché a snadno ovladatelné GUI. Jedná se především o možnost volně rozmisťovat optické členy na plochu, bez nutnosti explicitně definovat, který optický člen kam svítí. Bohužel celý simulátor Ray Optics Simulator pracuje pouze ve 2D. Oproti tomu VirtualLab 5 umožňuje práci ve 3D. Navíc ještě podporuje tvorbu holografického záznamu a jeho následnou rekonstrukci. Jeho nedostatek však spočívá v nutnosti zadat explicitně definici optických členů o tom, který kam svítí, což je pro interaktivní návrh optické sestavy poměrně nešikovné.

Pro úspěšnou realizaci této práce se jeví jako nejlepší možnost naposledy zmíněný Holografický simulátor. Ten má veškeré vlastnosti, které bychom využili pro modelování 3D scény a zároveň umožňuje záznam a rekonstrukci hologramu. Koncepční návrh celého tohoto simulátoru je bohužel nevhodný z hlediska rozšiřování v budoucnosti. Protobylo přijato rozhodnutí o vytvoření zcela nové verzesimulátoru. Nová verze simulátoruby měla vycházet ze základních principů původníverze, ale v mnoha ohledech by mělo dojít k razantním změnám a tojak při návrhu celé aplikace, tak i při realizaci.

Využití všech poznatků získaných při vývoji a užívání předchozí verze by mělo vést k jednoznačně robustnějšímu programu a to jak po uživatelské stránce, tak i po implementační stránce.

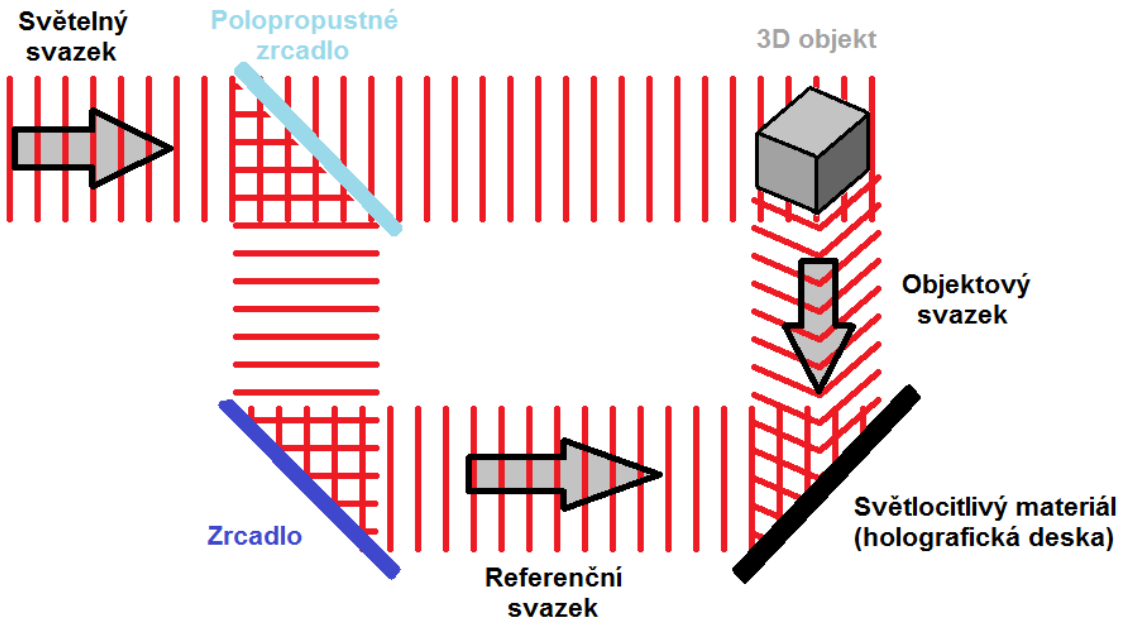
3 První verze

Kapitola je zaměřena na popis první verze Holografického simulátoru. Jsou zde popsány hlavní mechanizmy a vlastnosti, které simulátor nabízí. Zároveň je zde uvedena ukázka simulace tvorby a rekonstrukce holografického záznamu.

Funkci hologramu lze pochopit ukázkou dvou simulací chodu světelných paprsků: zaprvé tzv. záznam hologramu, zadruhé tzv. rekonstrukce hologramu. Detailně jsou oba procesy záznamu i rekonstrukce vysvětleny v publikacích „*Holographic imaging*“ [2] a „*Holografie: teoretické a experimentální základy a její použití*“ [3].

Zjednodušeně řečeno, během záznamu hologramu je světlocitlivý materiál osvětlen jak světlem odraženým od 3D objektu, tak i pomocným (tzv. referenčním) světlem. Tato dvě světla spolu interferují a interferenční obrazec je světlocitlivým materiálem zaznamenán.

Na obrázku 3.1 je vidět zjednodušený model záznamu hologramu. Světelný svazek je pomocí polopropustného zrcadla rozdělen na dva světelné svazky. Jeden (objektový) svazek dopadá přímo na 3D objekt, ze kterého se odráží a směřuje k holografické desce, na které následně interferuje s druhým (tzv. referenčním) svazkem. Výsledný interferenční obrazec je zaznamenán světlocitlivým materiálem (holografickou deskou).

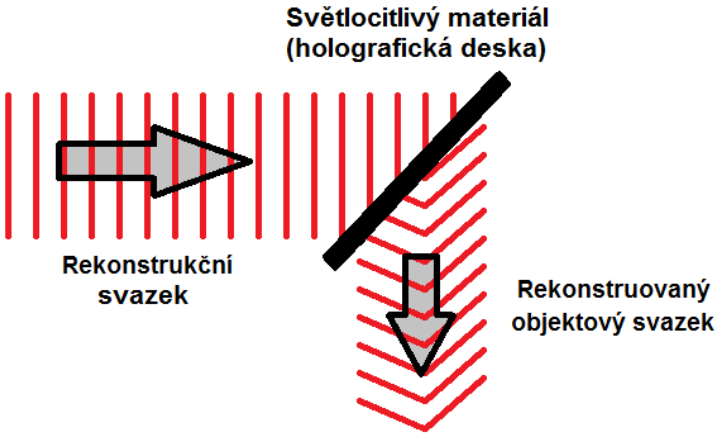


Obrázek 3.1 Zjednodušený záznam hologramu

Poznámka: V simulátoru je záznam hologramu ještě více zjednodušen. Zjednodušení spočívá především v tom, že objektový svazek nevzniká rozdělením hlavního svazku a následným odrazem od „nějakého“ 3D objektu, ale objektový svazekje složen z množiny světelných zdrojů (viz. Obrázek 3.3, kde jsou zobrazena tři světla tvořící objektový svazek), které se chovají jako by byly odraženy z 3D objektu. Vytváření referenčního svazku je také zjednodušeno, tudíž pro vytvoření takového svazku stačí pouze vytvořit zdroj světla s určitými vlastnostmi, na který bude nahlíženo jako na referenční zdroj.

V průběhu rekonstrukce hologramu je zaznamenaný interferenční obrazec (tzv. hologram) osvětlen světelným svazkem, tzv. rekonstrukčním. Na obrazci dochází k difrakci (ohybu) světla a hologram opouštějí jak paprsky přímo propuštěné, tak paprsky difraktované. Pokud má rekonstrukční svazek stejné vlastnosti jako svazek referenční, tvoří některé z difraktovaných paprsků dokonalou kopii objektového světla.

Na obrázku 3.2 je znázorněna zjednodušená rekonstrukce hologramu. Holografická deska je osvětlena tzv. rekonstrukčním svazkem, který má totožné vlastnosti jako původní tzv. referenční svazek, který byl použit při vytváření záznamu. Po osvícení holografické deskydochází k difrakci (ohybu) některých paprsků rekonstrukčního svazku, které následně tvoří původní zaznamenaný objektový svazek.



Obrázek 3.2 Zjednodušená rekonstrukce hologramu

Poznámka: V simulaci hologramu bychom chtěli pozorovat chování difrakce světelného svazku. Pokud se například při záznamu hologramu použilo objektové světlo šířící se od bodového světelného zdroje, chtěli bychom v simulaci rekonstrukce hologramu pozorovat difraktované paprsky, které zdánlivě začínají v bodu objektového světelného zdroje (Viz obrázek 3.7, kde jsou šedivými přímkami znázorněny výsledné rekonstruované paprsky a šedivými čárkovanými přímkami jsou znázorněny jejich zdánlivá prodloužení tak, aby bylo vidět, že rekonstruované paprsky jakoby vycházejí z původního bodu).

3.1 Vznik simulátoru

V době, kdy byl založen projekt na vývoj první verze simulátoru,nenašli jsme na trhu žádné optické simulátory, které by snadným způsobem dokázalysimulovat chování světelných paprsků v optických sestavách. Při hledání simulátorů jsme se tehdy zaměřili především na holografické simulátory, neboli simulátory, které dovedou vytvořit a následně rekonstruovat holografický záznam tak, aby bylo možné budoucím studentům demonstrovat základní vlastnosti a chování jednotlivých paprsků.

Jelikož tenkrát nebyl nalezen žádný vhodný simulátor, bylo rozhodnuto, že se vytvoří simulátor, na němž bude možné jednotlivé holografické simulace realizovat.

Při původním návrhu simulátoruještě nebylo zcela jasné, co vše lze od simulátoru očekávat. Na základě toho byla tedy první verze omezena pouze do 2D prostoru s ohledem na to, že bude do budoucna počítáno s rozšířením do 3D.

3.2 Základní popis vlastností

Celý simulátor je napsaný v jazyce Java jako appletová aplikace, která je spustitelná pomocí webového prohlížeče.

Simulátor je navržen tak, aby v něm bylo možno vytvářet několik vzájemně se neovlivňujících simulací najednou. Jednotlivé simulační scény obsahují pouze své vlastní optické členy, nad kterými jsou prováděny jednotlivé výpočty, díky čemuž je zaručena nezávislost simulací.

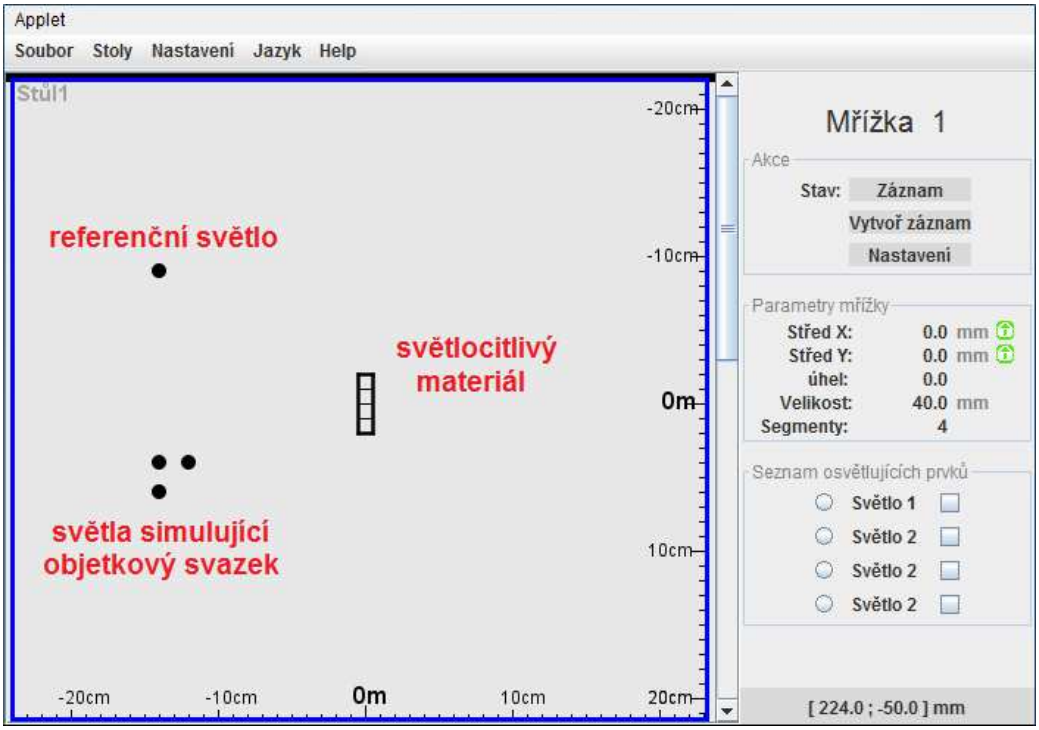
Další důležitou vlastností, kterou simulátor nabízí, je možnost vytváření různých pohledů na jednu nebo více simulovaných scén. Jinak řečeno,máme jednu simulovanou scénu (např. zdroj světla a čočku),kterou můžeme sledovat z několika různých pohledů. Například jeden pohled bude zobrazovat detailní chování paprsků na čočce, oproti tomu další pohled bude sledovat chování celé sestavy. Pro snadnější představu bude později ukázána práce s jednotlivými pohledy.

V simulátoru jsou implementovány prozatím tyto optické prvky: světelný zdroj, světlocitlivý materiál (holografická deska), zrcátko, optická čočka a dělič svazku paprsků.

3.3 Jednoduchá simulace

Nyní si ukážeme, jakým způsobem je možné vytvářet simulace v původním holografickém simulátoru. Celou ukázku budeme demonstrovat na vytvoření a následné rekonstrukci holografického záznamu.

Prvním krokem je rozmístění jednotlivých optických prvků. Na obrázku 3.3 je v modrém okně vidět základní rozmístění optických prvků, které jsem provedl. Přidávání jednotlivých optických prvků do scény se provádí pomocí menu, které sezobrazuje pomocí pravého tlačítka myši.



Obrázek 3.3 Rozmístění optických prvků

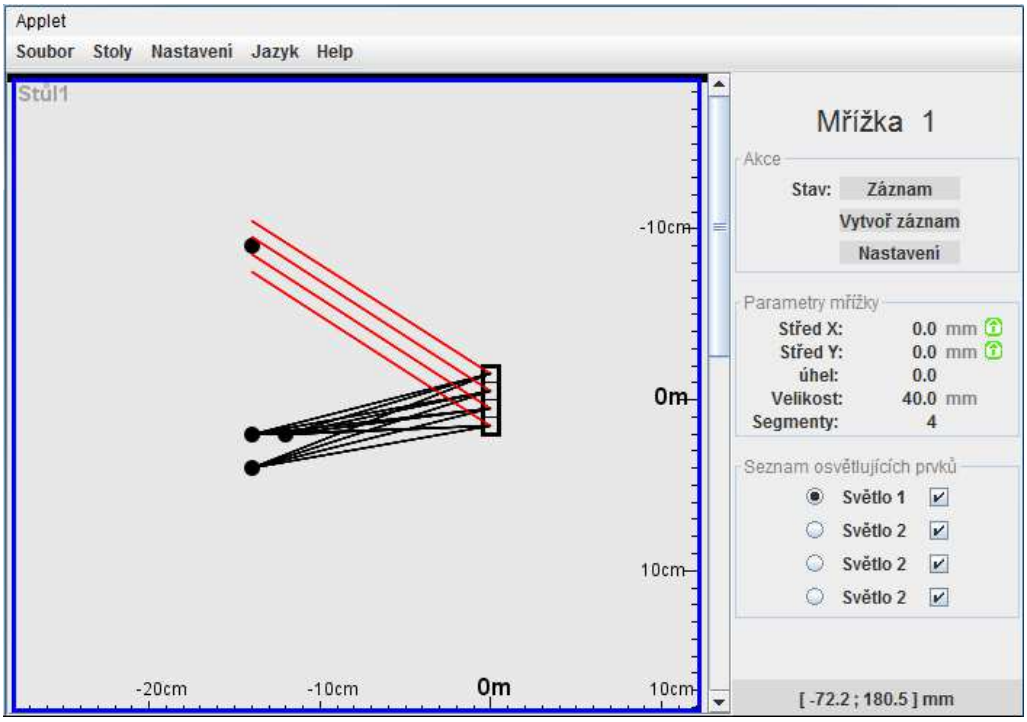
Máme tedy rozestavěnou scénu s jednotlivými optickými členy. Shluk tří světel,které jsou vidět na obrazovce, bude imitovat vlastnosti objektového svazku paprsků, které se budeme snažit zachytit do světlocitlivého materiálu (holografické desky). Samostatnésvětlo, které je umístěno nad nimi, bude představovat zdroj referenčního světla, pomocí kterého bude vytvořen záznam. Ve středu scény je umístěn světlocitlivý materiál, do kterého vytvoříme záznam.

Na obrázku 3.3je možné si všimnout panelu zobrazeného při pravém okraji. V tomto panelu jsou zobrazovány jednotlivé vlastnosti a nastavení, které je možné měnit v rámci aktuálně vybraného optického prvku. Aktuálně je zde zobrazeno nastavení pro světlocitlivý materiál (holografickou desku).

Dalším krokem simulace je nastavení posloupnosti tak, jak se budou jednotlivé optické členy vzájemně osvětlovat. Jelikož se v tomto případě nejedná o nikterak složitou optickou sestavu, bude stačit nastavit pouze optické prvky, které osvětlují světlocitlivý materiál.

Nastavení se provádí v pravém panelu v části „Seznam osvětlujících prvků“. V tomto seznamu se zobrazují všechny optické prvky, které jsou aktuálně vloženy v simulované scéně. My si zde vybereme pouze ty, které chceme, aby osvětlovaly nášsvětlocitlivý materiál (holografickou desku). Jelikož máme v simulaci pouze optické prvky, od kterých chceme, aby osvětlovalysvětlocitlivý materiál, vybereme vše. Ve chvíli, kdy vybíráme prvky, se v simulátoru již zobrazují jednotlivé optické paprsky.

Na obrázku 3.4 je vidět, jakým způsobem vedou paprsky z jednotlivých zdrojů světel až na holografickou desku (světlocitlivý materiál). Paprsky, které jsou zobrazeny červenou barvou,představují optické paprsky, které vycházejí z referenčního světla. Z fyzikálního pohledu je referenční světlo jako každé jiné, nicméně jak pro potřeby simulace, tak v praxi, je vhodné explicitně označit, které světlo chápeme jako referenční. Tento výběr může uživatel provést v dialogu „Seznam osvětlujících prvků“.



Obrázek 3.4 Zobrazení paprsků

Poznámka: U referenčního zdroje světla jsem nastavil rovnoběžné vyzařování paprsků (obdobným způsobem se šíří například paprsky u laserů), oproti ostatním světlům, kde jsem ponechal všesměrové vyzařování paprsků.

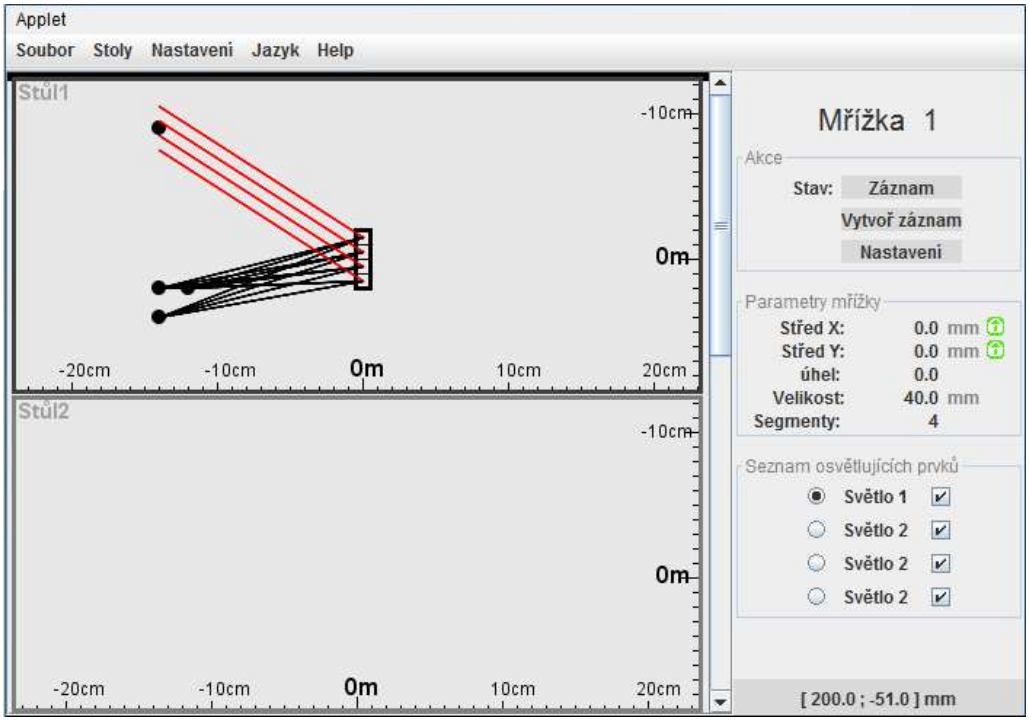
Třetím a závěrečným krokem pro vytvoření záznamu je samotné zaznamenání paprsků do světlocitlivého materiálu (holografické desky). Samotný záznam se provede tlačítkem „Vytvoř záznam“,který se nachází v pravém panelu. Po stisku tlačítka je záznam vytvořen, čímž je celá záznamová část hotova.

Aby více vynikly vlastnosti simulátoru, rekonstrukci záznamu provedeme v jiné simulaci a zároveň k tomu vytvoříme ještě okno s náhledem, který bude zobrazovat novou simulaci.

Nejprve je tedy nutné vytvořit novou simulační scénu, kde bude rekonstrukce provedena. Jednotlivé simulační scény jsou v tomto programu označovány jako stoly (Přirovnání k reálným stolům, na nichž se provádějí reálné optické pokusy). Vytvoření nové scény (stolu) se provádí pomocí menu, které je umístěno při horním okraji obrazovky.

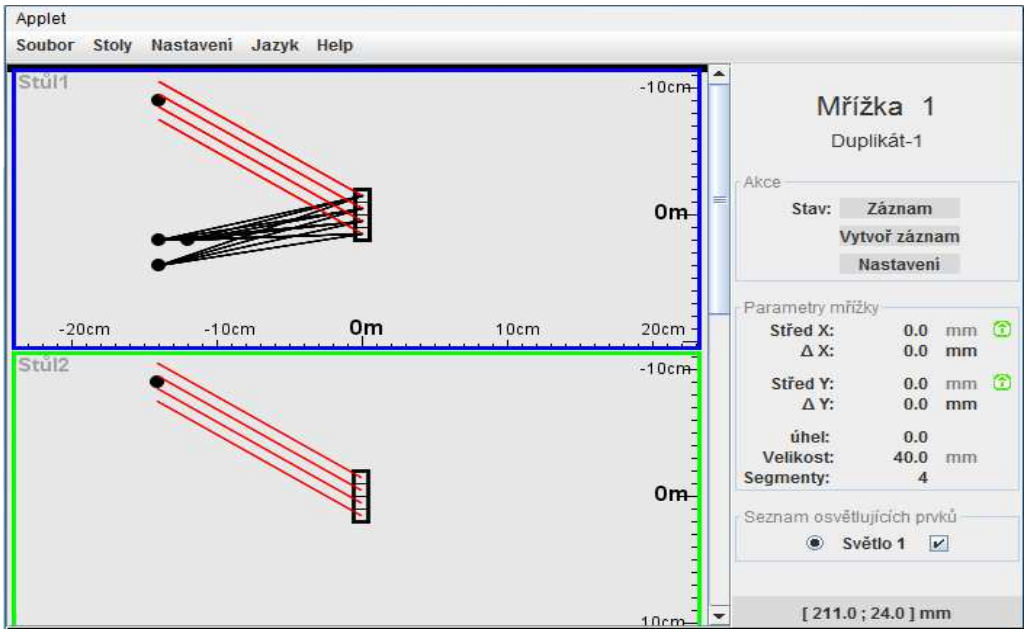
Dalším krokem je vytvoření náhledu, kterým budeme novou scénu (stůl) sledovat. Přidání nového okna se provádí tím stylem, že původní okno je rozděleno na dvě části, přičemž se každá chová jako samostatné nezávislé okno. Rozdělit okna je možné horizontálně nebo vertikálně. My zvolíme horizontální rozdělení.

Na obrázku 3.5 je vidět, jak bylo okno rozděleno. Zároveň je zde vidět, že každé okno zobrazuje jinou simulační scénu (stůl). Horní okno neustále vykresluje optickou sestavu pro vytvoření záznamu, kdežto dolní okno vykresluje novou scénu, která je prozatím prázdná.



Obrázek 3.5 Vytvoření nového okna

Abychom mohli provést jednoduchou rekonstrukci, stačí pouze vytvořit kopii světlocitlivého materiálu (holografické desky) a referenčního světla. Obě kopie vložíme do nově vytvořené scény, jak je vidět na obrázku 3.6.

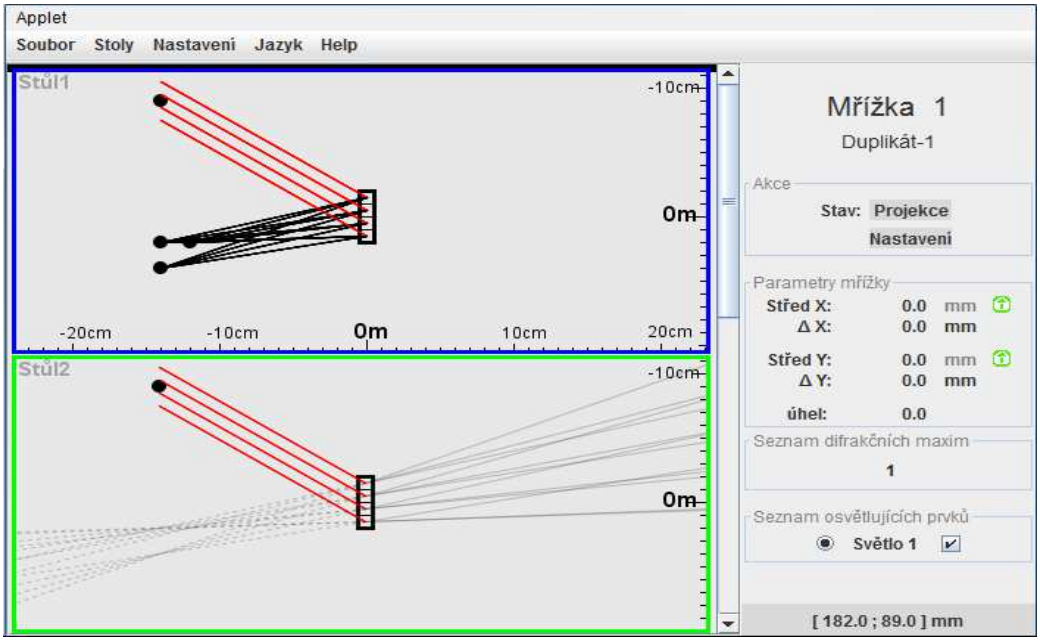


Obrázek 3.6 Kopie objektů

Poté již stačí jenpřepnoutholografickou desku (světlocitlivý materiál) ze záznamového režimu do režimu rekonstrukce. To lze provést tlačítkem „Záznam“, jenž se nachází v pravém panelu.

Pro přepnutí do režimu rekonstrukce simulátor začne vykreslovat jednotlivé paprsky, které budou odpovídat směrům a úhlům paprsků původního objektového paprskového svazku.

Na obrázku 3.7 je zobrazena výsledná rekonstrukce celého záznamu. Paprsky vpravo od holografické desky jsou výsledné paprsky. Takto by se v reálném světě normálně šířily. Čárkované paprsky na levé straně jsou pouze informativní a slouží pouze pro optické prodloužení paprsků z pravé strany. Díky tomuto prodloužení si uživatel může snadněji uvědomit závislosti mezi paprsky.



Obrázek 3.7 Výsledná rekonstrukce

4 Návrh architektury

Tato kapitola popisuje návrh architektury nového optického simulátoru. Většina úvah a navržených řešení vychází z poznatků předchozí verze. V kapitolebude tedy velmi často odkazováno na předchozí verzi optického simulátoru a její nedostatky.

4.1 Získané poznatky z předchozí verze

Tento bod shrnuje různé poznatky, které byly načerpány při tvorbě, testování, rozšiřování a užívání předchozí verze. Snahou je co nejlépe vystihnout všechny možné problematické části tak, aby se rapidně omezila možnost jejich opakovaného vzniku. Poznatky jsou rozděleny do několika částí podle logických souvislostí.

4.1.1 Poznatky související s architekturou programu

Předchozí verze vycházela z modelu MVC (Model-View-Controller). Postupným přidáváním funkčností však docházelo k smršťování celého programu více a více k sobě, až došlo prakticky ke ztrátě všech výhod MVC modelu. Proto je při návrhu a implementaci nutné dbát na dodržování zásad MVC.

Velkým úskalím se stal také fakt, že jednotlivé optické členy musely mít předem nadefinovány jinéoptické prvky, které je osvětlují. Neboli, již při modelování scény musel autorpředem vyznačit cestu paprskutak, jak se bude šířit přes jednotlivéoptické členy.Toto řešení bylo využito proto, aby simulátor snadno poznal,které optické členy jsou jak a čím osvětlovány. Z hlediska zkoumání složitějších optických je však tato vlastnost pro uživateledosti obtěžující. Proto v následující verzi bude tato vlastnost odstraněna.

Z poznatku také vyplývá fakt, že při vytváření nového optického členu je zapotřebí implementovat chování vůči všem již existujícím optickým členům v simulátoru. Což značně komplikuje rozvoj celého programu.

4.1.2 Programátorské poznatky

Celý projekt je veden jako appletová aplikace proto, aby jej bylo možné spustit jak v online, tak i v offline prostředí. Samotný applet se spouští standardně ve webovém prohlížeči. Každý prohlížeč má však definována různáfunkční tlačítka, která odchytává pro vlastní potřebu, proto je nutné při implementaci dbát na to, aby nebyla použita tato funkční tlačítka pro ovládání simulátoru.

4.1.3 Simulační poznatky

Velmi dobrou vlastností, která vycházela z celkového konceptu předchozí implementace, byla možnost pohybovat jedním objektem vůči jinému objektu, přičemž simulátor ihned reagoval na změnu pozic v simulaci a následně ihned dopočítával nově vzniklé paprsky nebo případné změny jejich směrů. Protože jednotlivé objekty vzájemně věděly o svých sousedech, nebyl problém dopočítávat jednotlivé paprsky mezi sousedními objekty.Znalost jednotlivých sousedů však v nové verzi zanikne, proto bude nutné vyřešit tuto interakci jiným způsobem.

Další funkčnost, která se setkala s úspěchem, je možnost kopírování jednotlivých objektů do stejných i jiných simulací (zkopírovaní do jiné scény). Kopírování je dvojího typu a to jako:

- **Nezávislá kopie** – tato kopie je vytvářena jako zcela totožný objekt se vzorem, ze kterého vzniká, avšak po vytvoření již nemá vůbec žádné propojení na svého předchůdce, ze kterého vznikla, ani on na ni. Veškerá parametrická nastavení a funkčnostjsou kopírovány naprosto totožně se vzorem.
- **Kopie se závislostí na předka –** tato kopie se vůči předchozí liší tím, že při vytvoření si uchovává odkaz na svůj vzorový optický prvek a vzorový optický prvek si uchovává odkazy na všechny své následníky. Díky této vlastnosti je možné provést změnu nějakého parametru ve vzorovém optickém prvku nebo v nějaké jeho kopii a provedená změna se automaticky nastaví všem následníkům i vzorovému optickému prvku. Toto řešení se v praxi však příliš neujalo, proto nemá význam pro následující verzi.

Dobrou vlastností je možnost zamknutí pozice optického prvku. Neboli optický prvek je možné zamknout v nějakém směru (x nebo y). Při pohybu s optickým prvkem se následně tato pozice nemění.

4.1.4 Uživatelské poznatky

Vzhled a rozložení komponent v hlavním okně předchozí verze bylo uživateli oceněno jako vhodné řešení, jednalo se především panel při pravém okraji, kde se zobrazují jednotlivé parametry optických členů.

S kladnými ohlasy bylo přijato řešení náhledů. A to především možnost vytvořit si několik nezávislých pohledů na jednu simulaci, které mohly zobrazovat simulovanou scénu z několika různých úhlů pohledu.

Úspěch také zaznamenalo řešení rozdělování vizualizačních oken s jednotlivými simulacemi. Možnost rozdělení okna vodorovně či svisle je hodně využívána při různých detailních simulacích.

4.2 Návrh architektury

Celý projekt bude postaven na MVC architektuře. Podobně jako tomu bylo v předchozí verzi. Jednotlivé objekty optických členů o sobě nebudou vzájemně vědět. Budou pouze zpracovávat poznatky a informace z jednotlivých paprsků, které se budou mezi nimi šířit obdobným způsobem jako je tomu v reálném prostředí.

4.2.1 Vrstva Model

Vrstva Model bude obsahovattřídy (struktury) jednotlivých optických členů, jako jsou zdroje světla, zrcátka, stínítka a další.

Tyto třídy budou vytvořeny jako následníci abstraktní rodičovské třídy, která bude zastřešovat většinu společných vlastností, jako jsou pozice, natočení, velikost, apod. Součástí rodičovské třídy bude také abstraktní metoda, kterou musí každý následník překrýt vlastní implementací. Tato metoda bude mít za úkol přijmout seznam vstupních paprsků, nějakým způsobem je zpracovat a vrátit seznam výstupních paprsků. Speciálním případem optických prvků budou třídy, které budou simulovat chování zdroje světla. Tyto třídy nebudou přijímat žádné vstupní paprsky, ale budou pouze vyzařovat výstupní paprsky.

Jednotlivé elementy budou skládány do optických sestav. Každá taková sestava bude obsahovat seznam se všemi svými optickými členy tak, aby bylo možné nad tímto seznamem spustit metodu pro sledování paprsků, která následně dopočítávšechny vzniklé paprsky v simulaci.

4.2.2 Vrstva Controller

Pomocí vrstvy Controller bude realizován přístup k datovému modelu. Každý optický prvek protobude mít vlastní třídu Controller, aby veškerá logika vztahující se k zadávání, změně nebo mazání jednotlivých hodnot uživatelem musela býtřešena pomocí této třídy. Na základě tohoto řešení dojde k částečné ochraně proti zadání nevalidních dat.

Jednotlivé třídy Controller budou vytvářeny jako následníci abstraktnírodičovské třídy, která bude zastřešovat většinu společných nastavení.

4.2.3 Vrstva View

Prezentační vrstva View bude zobrazovat simulace pomocí takzvaných náhledů. Pod náhledem si lze představit klasické okno (kameru), ve kterém je zobrazen podhled na simulační scénu v 3D prostoru. Náhledem (kamerou) bude možno posouvat, rotovat, přibližovat apodobně, čímž se docílí možnosti sledovat vykreslovanou simulaci ze všech stran a úhlů.

Náhledů (kamer) na jednu simulační scénu bude moci být více, aby si mohl uživatel vytvořit několik nezávislých pohledů na sledovanou simulační scénu.

4.3 Simulační prostředí

Simulační prostředí se v minulé verzi velice osvědčilo. Proto bude principielně zachováno prakticky v nezměněné formě.V simulátoru bude možné vytvářet několik simulací najednou, každá v jiné 3D scéně. Každá 3D scéna bude pozorovatelná z různých pohledů. Jednotlivé pohledy mohou být přepínány mezi různými scénami podle potřeb.

4.3.1 3D scéna

Pod pojmem 3D scéna si lze představit 3Dprostor, do kterého jsou vkládány jednotlivé optické členy. Tyto optické členy tvoří jednu optickou sestavu (jedna simulace). V reálném prostředí to lze přirovnat ke stolu (stůl rovná se 3D scéna), na kterém jsou vyskládány jednotlivé optické prvky (zdroj světla, zrcátka, apod.).

4.3.2 Ukládání a načítání simulací

Ukládání sestav bude řešeno pomocí textového souboru. Do souboru se budou ukládat všechny simulace, které jsou v simulátoru otevřeny. Struktura souboru bude řešena pomocí jednoduchých názvů proměnných, aby bylo možné snadným způsobem v tomto souboru provádět drobné změny.

Načítání simulací ze souboru bude řešeno vždy na nově vytvořené stoly, aby nedošlo k přepsání některých otevřených simulací.

4.4 Optické členy

Každý optický člen bude následovníkem abstraktní třídy, která bude zastřešovat společné parametry jednotlivých členů.

Optické členy budou realizovány jako zcela nezávislé objekty vůči ostatním optickým členům i vůči simulačnímu prostředí, ve kterém budou simulace probíhat. Neboli každý optický člen přijme veškeré paprsky, které jej osvětlují, a podle vlastní implementace tyto paprsky zpracuje a vrátí množinu výstupních paprsků zpět do imulačního prostředí.

Výčet hlavních společných parametrů:

- Pozice objektu (*x*, *y*, *z*)
- Natočení objektu (*rotační matice 3x3*)
- Rozměry objektu (*šířka*, *výška*, *hloubka*)
- Tvar objektu
- Seznam vstupních paprsků (*ArrayList*)
- Seznam výstupních paprsků (*ArrayList*)

Výčet hlavních společných metod:

- Metoda pro obsluhu vstupního paprsku
- Metoda pro kopírování celého objektu
- Metody pro uložení do souboru
- Metody pro načtení ze souboru

4.5 Paprsky

Oproti předchozí verzi nebudou nyní paprsky předem znát koncový objekt, který protínají. Objekty paprsků budou obsahovat pouze počáteční pozici paprsku (*x*,*y*,*z*) a směrový vektor (*x*,*y*,*z*). Na základě těchto dvou hodnot budou dopočítávat průsečík s koncovým objektem.

Každý paprsek bude obsahovat zároveň informace o svých vlastnostech. Zde je výčet nejdůležitějších hodnot:

- Počáteční pozice (*x*,*y*,*z*)
- Počáteční optický prvek (prvek, ze kterého paprsek vychází)
- Směrový vektor
- Seznam vlnových délek
- Vlastnosti polarizace (pro každou vlnovou délku)
- Vlastnosti koherence (pro každou vlnovou délku)
- Energie (pro každou vlnovou délku)
- Cílový optický prvek (po nalezení)
- Cílová pozice(*x*,*y*,*z*) (po nalezení)
- Maximální počet odrazů paprsku

Protnutý objekt se bude vyhledávat až při simulaci. Proto budou veškeré informace o vlastnostech paprsku přenášeny přímo v objektech paprsků. Dalo by se říci, že objekt paprsku bude jakýmsi rozhraním mezi jednotlivými elementy, čímž budou zcela odstraněny závislosti mezi jednotlivými elementy (optickými prvky).

4.6 Sledování paprsků

Sledování paprsků (neboli *„ray tracing“*) bude realizovánoprvotním průchodem seznamu všech elementů v simulaci, při němž se vyberou všechny světelné zdroje. Z těchto vybraných světelných zdrojů se vykopírují všechny výstupní paprsky, které tyto zdroje vyzařují do prostoru. Kopírování proběhne do předem připraveného seznamu „neprozkoumaných“ paprsků.

Následně se z tohoto seznamu budou postupně vybírat jednotlivé paprsky. U každého paprsku bude dopočteno,zda protíná či neprotíná některý z ostatních elementů. Pokud bude vybraný paprsek protínat některý element, tak nad tímto elementem spustí metodu, která na základě tohoto paprsku vypočítá výstupní paprsek/paprsky. Výstupní paprsek/paprsky následně přidá na konec seznamu „neprozkoumaných“ paprsků tak, aby mohly tyto paprsky být později také prozkoumány. Pokud paprsek nebude protínat žádný objekt, bude paprsek zrušen, jelikož nemá žádný vliv na simulaci. Takto se bude vše opakovat,dokud nedojde k prozkoumání všech paprsků v seznamu „neprozkoumaných“ paprsků.

Aby při vzniku nových paprsků nedocházelo k zacyklení (například dvě zrcadla se neustále vzájemně osvětlují nově vznikajícími paprsky), bude každý paprsek obsahovat celočíselnou proměnnou. Tato proměnná bude značit maximální zbývající počet odrazů paprsku. Světelný zdroj tedy při generování paprsků každému z nich nastaví maximální počet odrazů. Následněse budou paprsky šířit mezi jednotlivými optickými prvky a při průchodu každým prvkem bude snížen maximální počet odrazů o jeden. Ve chvíli, kdy dosáhne maximální počet odrazů hodnoty nula, paprsek již dále pokračovat nebude.

Důležitým bodem je určení způsobu,jakým bude probíhat výběr nezpracovaných paprsků ze seznamu; zda k seznamu přistupovat jako k zásobníku (LIFO) nebo jako k frontě (FIFO).Po důkladnější analýze jsemdospěl k názoru, že tento fakt nemá nikterak závažný vliv na funkčnost celého mechanizmu pro sledování paprsků, tedy nezáleží na tom, zda se použije zásobník nebo fronta. Při realizaci simulátoru, jsem se proto přiklonil k zásobníku (LIFO).

Po prozkoumání všech (seznam „neprozkoumaných“ paprsků) paprsků bude simulátor mít veškeré informace o chování celé simulace, a tak bude moci jednoduchým způsobem zobrazit celou scénu uživateli.

5 Vývojové prostředí

Aby byl program meziplatformně přenositelný, je celý projekt veden v jazyce Java, který v současné době spravuje společnost Oracle. Java je vydávána v několika knihovních verzích podle charakteru vytvářené aplikace a prostředí, ve kterém má být využívána. Pro tento projekt byla zvolena standardní knihovna Java SE (Standard edition), která plně vystačí veškerým požadavkům. Přesné označení vybrané knihovny je Java SE jdk 1.8.0_31.

Projekt byl založen a je veden ve vývojovém prostředí IntelliJ IDEA od firmy JetBrains. Jedná se o komerční vývojové prostředí, které je dostupné ve dvou licenčních verzích: Community Edition a Ultimate Edition. Hlavní rozdíly mezi těmito verzemi jsou především ve funkčnosti a ceně. Community Editon je volně přístupná free verze pro širokou veřejnost, ovšem s omezenou funkčností. Oproti tomu Ultimate edition je plně funkční, ale placená platforma. Zde v projektu je využívána licenční verze Community Edition. Konkrétní označení vývojového prostředí je IntelliJ IDEA Community Edition 14.0.3.

Aby byl možný paralelní vývoj projektu různými vývojáři, byl využit verzovací systém Git společně s webovou službou Bitbucket. Webová služba Bitbucket nabízí bezplatný hosting pro všechny open-source projekty. Zároveň také vyniká velice snadnou a přehlednou obsluhou.

6 Implementace

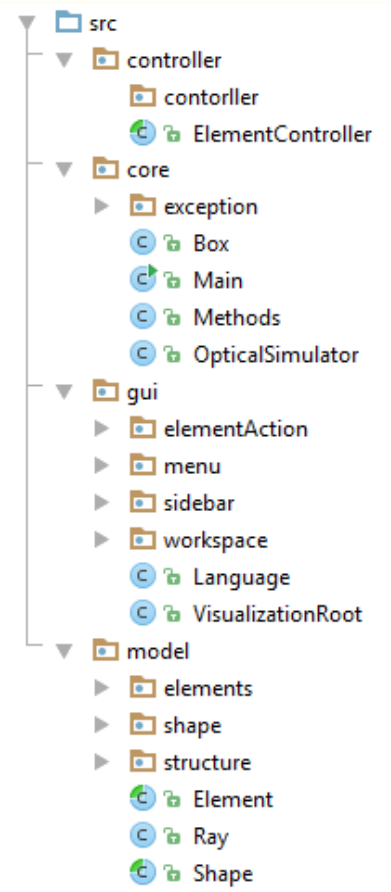
Kapitola popisuje strukturu celého projektu, a to hlavně rozložení jednotlivých tříd. Snahou je především nastínit vzájemné vazby a vztahy mezi objekty.Zároveň jsou u každé třídy vypsány nejdůležitější atributy a metody, které se v dané třídě nacházejí.

6.1 Popis jednotlivých tříd

Celý projekt je rozdělen z důvodu přehlednosti a logické návaznosti do čtyřhlavních balíků.

- *core* – obsahuje třídy obsluhující simulaci a jádro celého programu
- *gui* – obsahuje třídy pro obsluhu uživatelského rozhraní
- *model* – obsahuje datové struktury jednotlivých elementů
- *controller* – obsahuje třídy s metodami pro přístup k datům

Na níže uvedeném obrázku 6.1 je znázorněna částečná struktura a organizace jednotlivých podbalíků a tříd projektu.



Obrázek 6.1 struktura celého projektu

6.1.1 Balík Core

V tomto balíku se nacházejí především třídy, které se vztahují k jádru a hlavní funkcionalitě celého projektu. Třída *OpticalSimulator* je jádrem celé aplikace. Balík obsahuje jak hlavní spustitelnou třídu, tak i třídu s globálními funkcemi.

6.1.1.1 Třída Main

První třída, která je volána operačním systémem při spuštění aplikace, obsahuje pouze jednu metodu *„main()“*. Tato metoda vytváří instanci hlavní třídy *OpticalSimulator* a následně jí předává řízení.

6.1.1.2 Třída OpticalSimulator

OpticalSimulator je jádrem celého programu. Obsahuje seznam s jednotlivými aktuálně otevřenými simulacemi, ke kterým je možné pomocí této třídy přistupovat. Informace o jednotlivých otevřených simulacích jsou uloženy v seznamu *„listBox“*.

Ve třídě jsou zároveň implementovány funkce pro ukládání a načítání globální konfigurace celého programu. Jak ukládání, tak i načítání konfiguračních dat probíhá do samostatného konfiguračního souboru.

Dále jsou zde implementovány metody pro uložení a načtení jednotlivých simulací. Jelikož se v tomto případě jedná o data simulátoru, tak metody pracují s vlastním datovým souborem. Tento soubor je v textové podobě ačástečně strukturovaný tak, aby uživatel mohl snadno tento soubor přečíst a částečně upravit.

Přehled atributů:

- `ArrayList<Box> listBox` –seznam všech simulací, které jsou aktuálně otevřeny v programu. Každá nově vytvořená simulace (3D scéna) musí být přidána do tohoto seznamu.
- `VisualizationRoot visualizationRoot` – odkaz na hlavní vykreslovací třídu celého GUI.

Přehled metod:

- *void saveWorkspace(String fileName)* – uložení všech simulací do souboru
- *int loadWorkspace(String fileName)* – načtení simulací ze souboru
- *void saveConfig(String fileName)* – uložení konfigurace do souboru
- *void loadConfig(String fileName)* – načtení konfigurace ze souboru
-

6.1.1.3 Balík Exception

Tento balík obsahuje třídy výjimek, které jsou vytvářeny jako následníci knihovní třídy *Exception.* V balíku se zatím nachází pouze jedna třída *LoadWorkspaceException,* která je spouštěna ve chvíli, kdy dojde k chybě při načítání ze souboru.

6.1.1.4 Třída Box

Třída *Box* imituje reálnou 3D scénu (lze si pod tím představit stůl, na který se vkládají jednotlivé optické členy). Třída v sobě uchovává jednotlivé optické členy, ze kterých se skládá výsledná simulační scéna.

Dále je zde implementována hlavní metoda *„rayTracing()“* pro sledování jednotlivých paprsků. Všechny paprsky vedoucí mezi dvěma optickými členy jsou ukládány do seznamu *„listCalculateRay“*. Paprsky, které vycházejí z jednoho optického členu a zároveň žádný jiný neprotínají, jsou uloženy v seznamu *„listCalcalateRayNoIntersection“*.

Přehled atributů:

- `ArrayList<Element> listElements` – seznam elementů umístěných ve 3D scéně.
- `ArrayList<View> listViews`– seznam náhledů, které jsou aktuálně připojeny ke 3D scéně a zobrazují ji.
- `ArrayList<Ray> listNoCalculateRay` – seznam paprsků, u kterých ještě nebyl proveden výpočet koncového elementu.
- `ArrayList<Ray> listCalculateRay` – seznam paprsků, které vedou mezi dvěma elementy.
- `ArrayList<Ray> listCalculateRayNoIntersection`– seznam paprsků, které vedou do nekonečna

Přehled metod:

- *void rayTracing()* – metoda pro sledování paprsků
- *void save()* – metoda pro ukládání simulací do souboru
- *String load()* – metoda pro načítání simulací ze souboru

6.1.1.5 Methods

Methods je statická třída, která obsahuje pouze globální metody, které jsou využívány několika třídami. Jedná se především o metody zabývající se rotacemi jednotlivých bodů v prostoru.

Přehled metod:

- *Position rotatePointAroundCenterAxis()* – rotace bodu okolo počátku souřadnicového systému
- *Position revertRotatePointAroundCenterAxis()* – zpětná rotace bodu okolo počátku souřadnicového systému
- *double getDistancePointAandB()*– výpočet vzdálenosti mezi dvěma body

6.1.2 Balík Model

Tento balík obsahuje především datové struktury a struktury dílčích elementů (optických prvků). Proto je rozdělen do tří podsložek a tří hlavních tříd, od kterých dědí jednotlivé elementy (optické členy) nebo paprsky.

6.1.2.1 Třída Element

Jedná se o abstraktní třídu. Každý element (optický prvek) je vytvořen jako následník této třídy. Díky tomu je zaručeno, že každý následovník této třídy bude obsahovat stěžejní atributy a funkce pro správnou funkčnost celého simulátoru.

Důležité atributy, které tato třída obsahuje, jsou především pozice, rotace, velikost, apod. Přístup k těmto atributům probíhá klasicky přes *getry* a *setry*. Dále musí mít každý element (optický člen) nastaven příznak *„isLightSource“*. Pokud bude hodnota nastavena na hodnotu *„true“*, bude k tomuto elementu (optickému členu) přistupováno jako k aktivnímu zdroji světla. Oproti tomu hodnota *„false“* značí, že daný element (optický prvek) je pouze pasivním optickým členem a reaguje tedy až na paprsky, které jej osvětlí. Dále je nutné mít nastaven příznak *„isAllRay“*, který značí zda optický člen bude zpracovávat vstupní paprsky po jednom „*false*“, nebo je bude zpracovávat všechny najednou *„true“*.

Důležitá metoda *„isIntersectionBall()“*, která je předdefinována pro všechny elementy (optické členy) slouží ke zjištění průsečíku s obalovou koulí. Jedná se o rychlý test, který je prováděn vždy před tím, než je přistoupeno ke složitějším matematickým výpočtům průniku paprsku s objektem.

Přehled atributů:

- `int ID_number` – každáinstance má své vlastní *ID*
- `Position position` – struktura obsahující informace o pozici
- `Matrix matrix` – matice uchovávající informace o natočení
- `Dimension dimension` – struktura nesoucí informace o rozměrech
- `Shape shape` –objekt mající informace o tvaru elementu
- `boolean isLightSource` – příznak zda je element zdrojem světla
- `boolean isAllRay` – příznak zda element zpracovává všechny vstupní paprsky najednou nebo ne
- `ArrayList<Ray> listOutputRay` – seznam vystupujících paprsků
- `ArrayList<Ray> listInputRay` – seznam vstupních paprsků

Přehled abstraktních metod, které musejí následníci překrýt:

- *void init()* –slouží k inicializaci objektu a je spouštěna vždy po vytvoření objektu

- *Position getPositionIntersection(Ray)* – je využívána pro zjištění, zda vstupní paprsek, který je předán jako vstupní parametr,protíná daný element či nikoliv. Pokud paprsek element protíná, vrací metoda pozici průsečíku. Pokud ne,tak návratová hodnota je„*null*“. Metoda je volána při každém testování průsečíků paprsků. Proto je z výpočetního hlediska co nejvíce optimalizována.
- *ArrayList<Ray>calculation(Ray)* – metoda pro zpracování vstupního paprsku. Je použitave chvíli, kdy je objeven průsečík paprsku s daným elementem.Hlavní činností této funkce je zpracování vstupního paprsku a vytvoření nově vzniklých paprsků. Metoda vrací seznam nově vzniklých paprsků.
- *ArrayList<Ray>calculationAllRay(Ray)* – metoda pro zpracování všech vstupních paprsků, které se nacházejí v seznamu *listInputRay*. Je spouštěna ve chvíli, kdy má daný element v seznamu *listInputRay* všechny paprsky, které jej osvětlují. Metoda vrací seznam nově vzniklých paprsků.
- *void updatePosition()* – metoda je spuštěna pokaždé, když dojde ke změně pozice objektu.
- *void updateDimension()* – metoda je spuštěna pokaždé, když dojde ke změně velikosti objektu.
- *Element copy()* – metoda k vytvoření nezávislé kopie; je volána vždy při pokynu k vytvoření kopie daného objektu.
- *void save()* – metoda pro ukládání parametrů elementu do souboru
- *void load()* – metoda pro načítání parametrů elementu ze souboru

Přehled předdefinovaných společných metod:

- *boolean isIntersectionBall()* – metoda pro rychlý test průniku paprskus obalovou koulí. Vrací hodnotu„*true*“ nebo „*false*“.
- *Position getPositionIntersectionShapeSquare(Ray)* – jelikož většina elementů má tvar obdélníkové plochy, je v této metodě implementován test pro ověření, zda vstupní paprsek prochází danou obdélníkovou plochou či nikoliv. Metoda při nálezu průsečíku vrací danou pozici průsečíku. V opačném případě vrátíhodnotu „*null*“.Pokud bude element jiného než obdélníkového tvaru, nelze tuto metodu při implementaci takového elementu využít, a proto je nutné napsat specifickou metodu, která bude daný test řešit.
- *void saveGlobalSettings()* – metoda je určena pro ukládání globálního nastavení do souboru, tedy obstarává uložení všech atributů, které mají jednotlivé elementy (optické prvky) společné (pozice, natočení, …).
- *void loadGlobalSettings()* – metoda je určena k načítání globálních atributů

6.1.2.2 Třída Shape

Abstraktní třída *Shape* uchovává informace o tvaru elementu (optickém prvku). Třída obsahuje dva základní modely. Prvním modelem je matematický model, který popisuje tvar elementu (optického prvku) pro výpočetní procesy. Druhým modelem je síť bodů a ploch, které představují tvar elementu tak, jak se budou zobrazovat uživateli na monitoru. V následující kapitole 8.2 budou uvedeny konkrétní příklady obou sítí a zároveň bude popsána jejich detailnější funkčnost.

Následník třídy *Shape* je vždy pevně svázán s nějakým následníkem třídy *Element*, v poměru 1:1. *Shape* třída vznikla především pro usnadnění čitelnosti celého kódu a snadné změně tvarů pro různé elementy (optické prvky).

Přehled atributů:

- `ArrayList<Position> numerNet` – seznam (síť) bodů, která je využívána při matematických výpočtech. V celém seznamu je důležité zachovat přesně stanovené pořadí vkládání jednotlivých bodů. Jelikož každá pozice seznamu má speciální význam. Konstanty definující jednotlivé pozice v seznamu jsou uvedeny taktéž v této třídě.
- `Hashtable<View, ArrayList<GeneralPath>> faceView` – *„faceView“* tabulka v sobě uchovává před-vypočítané sítě bodů, ze kterých je objekt složen. Tato síť je využívána při vykreslování elementu (optického prvku) na monitor.

  Do *„hash“*tabulky jsou hodnoty ukládány pomocí odkazu na jednotlivé pohledy (instance třídy *View*). Lze tedy říci, existuje-li vícero pohledů,ve kterých se nachází daný objekt,je v této tabulce uložena před-vypočítaná síť pro každý jednotlivý pohled. Toto řešení značně urychluje překreslování jednotlivých náhledů.

Přehled abstraktních metod (potomci je musí překrýt):

- *void init()* –slouží pro inicializaci objektu, je volána vždy po vytvoření objektu
- *Shape copy()* –vytvoří totožnou kopii s daným objektem. Je volána vždy, když dochází ke kopírování elementu.

Přehled předdefinovaných metod:

- *boolean contains(View, x,y)* – metoda je využívána při výběru elementu kursorem. Pomocí zadaného pohledu(*View*), pozice kusoru (*x, y*) a hashtabulky *„faceView“* metoda dopočte, zda kursor ukazuje na objekt či nikoliv.

6.1.2.3 Třída Ray

Třída *Ray* reprezentuje jednotlivé paprsky. Každý paprsek je realizován jako úsečka z bodu A do bodu B. Z toho vyplývá, že simulace například lomu paprsku na zrcadle, je realizována jako soustava dvou objektů třídy *Ray*. Jeden objekt třídy *Ray* simuluje dopad paprsku na zrcadlo, a druhýobjekt simuluje odražení paprsku od zrcadla. Na tomto principu jsou vytvořena všechna zpracování paprsků na jednotlivých elementech.

Objekt *Ray* slouží jako rozhraní mezi jednotlivými elementy (optickými prvky), proto obsahuje mnoho atributů, pomocí kterých se přenáší jednotlivé informace. Pod tímto odstavcem je uveden seznam nejdůležitějších atributů. K nim v budoucnu přibudou ještě atributy pro polarizaci, koherenci apod.

Jelikož nelze vypočítávat odrazivost paprsků do nekonečna, tak je ve třídě vytvořen atribut *„maxPass“*, který řeší maximální možný počet odrazů paprsku. Tím pádem při každém zpracování paprsku na nějakém elementu (optickém prvku) dojde ke snížení tohoto čísla o jedna (*např.: vstupní paprsek: maxPass = 3, výstupní paprsek: maxPass = 2*). Ve chvíli, kdy hodnota *„maxPass“* dosáhne nuly, paprsek zanikne a již se dále nešíří.

Přehled atributů:

- `Element startElement` – odkaz na element, ze kterého paprsek vychází
- `Element endElement` – odkaz na element, který paprsek osvětluje
- `Position startPosition` – přesná pozice, ze které paprsek vychází
- `Position end Position` – přesná pozice, kde paprsek končí
- `Vector vector` – směrový vektor paprsku
- `int maxPass` – maximální počet odrazů paprsku
- `ArrayList<double> listWaveLength` – seznam vlnových délek, které paprsek obsahuje

Přehled metod:

- *Ray getCopy()* – metoda vrátí v nové instanci kopii daného paprsku
- *void translation()* – posune paprsek o danou hodnotu
- *void rotation(Matrix)* – natočí paprsek podle zadané rotační matice

6.1.2.4 Balík Structure

Balík*Structure*, obsahuje pouze jednotlivé datové struktury (třídy), které jsou využívány pro snazší manipulaci s jednotlivými daty.

Seznam jednotlivých tříd (struktur) z balíku:

- ***Position*** – datová třída, pomocí níž jsou uchovávány souřadnice v prostoru. Jednotlivé souřadnice jsou uloženy jako datový typ*double*. Označení souřadnic je *X, Y, Z*.
- ***Dimension*** – datová třída, uchovávající informace o rozměrech elementů (optických prvků). Rozměry jsou uloženy v datových typech *double* a jsou označeny jako *Width, Height, Depth*.
- ***Matrix*** – datová třída obsahující informace o natočení objektu vůči souřadnicovému systému. Hodnoty natočení jsou uloženy v rotační matici 3x3, která je součástí této třídy.
- ***Vector*** – datová třída, obsahující informace o směrovém vektoru. Třída je složena ze tří hodnot *X, Y, Z* typu *double*.
- ***Edge*** – datová třídauchovávající informace o jedné úsečce. Informace jsou uloženy jako počáteční bod„*startPosition*“ a koncový bod „*endPosition*“úsečky.
- ***Counter*** – datová třída sloužící jako počítadlo.

6.1.2.5 Balík Elements

Balík*elements* obsahuje již konkrétní implementace jednotlivých optických prvků. Veškeré zde obsažené třídy jsou vytvořeny jako následníci třídy *Element*.

Přehled tříd obsažených v balíku:

- ***Light*** – simuluje chování bodového nebo plošného světla
- ***Mirror*** – simuluje chování zrcadla
- ***Wall*** – simuluje chování stínítka
- ***HolographicPlate*** – simuluje chování světlocitlivého materiálu (holografické desky)

6.1.2.6 Balík Shape

V tomto balíku se nacházejí jednotlivé tvary elementů (optických prvků). Všechny zde přítomné třídy jsou zděděny od svého předchůdce *Shape*.

Přehled tříd v balíku:

- ***LightShape*** – uchovává informace o tvaru světla
- ***MirrorShape*** – uchovává informace o tvaru zrcadla
- ***WallShape*** –uchovává informace o tvaru stínítka
- ***HolographicPlateShape*** – uchovává informace o tvaru světlocitlivého materiálu (holografické desky)

6.1.3 Balík Controller

V tomto balíku se nachází abstraktní třída*ElementController*, od které musí každý následník zdědit vlastnosti. Proto jednotlivé třídyvycházejí z tohoto vzoru. Dále je zde umístěn balík *controllers*, kde jsou umístěnykonkrétní specifikacetříd *„controller“*, které se vztahují již na konkrétní elementy (optické prvky).

6.1.3.1 Třída ElementController

Třídu *ElementController* je možné chápat jako rozhraní mezi grafickým uživatelským rozhraním a datovým modelem.

Jedná se o abstraktní třídu, která slouží jako vzorová třída pro následníky, kteří musejí být dědici této třídy. Pro abstraktní metody, které jsou ve třídě nadefinovány, je nutné, aby každý z následníků překryl svou vlastní implementací.

Třída *ElementController* si uchovává odkaz *„dataElement“* na instanci třídy *Element*, sníž jsou vzájemně úzce spjaty. Přes tento odkaz jsou posléze prováděny veškeré přístupy k datové třídě (objekt typu *Element*).

Každý uživatelský požadavek na změnu v datovém modelu (např. změna pozice, natočení, apod.) musí být prováděn přes metody implementované v jednotlivých třídách tohoto balíku. Proto třída *ElementController*obsahuje mnoho metod sloužících k nastavování jednotlivých hodnot elementů.

Při implementaci grafického rozhraní je nutné zachovat základní konvence a k jednotlivým elementům (optickým prvkům) a jejich hodnotám přistupovat pouze pomocí metod, které jsou součástí třídy *ElementContorller*.

Přehled atributů:

- `Element dataElement`– odkaz na datovou třídu element (optický prvek)
- `boolean isActiveElement`– příznak pro určení, zda je objekt aktivní
- `boolean lockPositionX`– příznak, zda je povoleno/zakázáno měnit pozici elementu (optického prvku) po ose X
- `boolean lockPositionY`– příznak, zda je povoleno/zakázáno měnit pozici elementu (optického prvku) po ose Y
- `boolean lockPositionZ`– příznak, zda je povoleno/zakázáno měnit pozici elementu (optického prvku) po ose Z

Přehled abstraktních metod:

- *ElementController copy()* – metoda vrátí nezávislou kopii objektu
- *void save()* – metoda implementuje ukládánídat do souboru
- *String load()* – metoda implementuje načtení dat ze souboru.

Přehled společných metod:

- *void init()* – metoda pro inicializaci objektu pro jeho vytvoření. Pokud ji bude chtít následník změnit, stačí ji pouze překrýt vlastní implementací.
- *void changePosition()* – metoda volána při každé změně pozice daného elementu(optického členu).
- *void changeRotation()* – metoda volána při každé změně natočení daného elementu (optického členu).
- *void changeDimension()* – metoda volána při každé změně rozměrů elementu (optického členu)

6.1.3.2 Balík Contollers

Tento balík obsahuje jednotlivé dědice abstraktní třídy *ElementController*. Každý následník v tomto balíku se vztahuje ke konkrétnímu elementu (optickému členu) z balíku *„model.element“* o němž byla zmínka výše.

Seznam tříd, které jsou v balíku obsaženy:

- ***LightController*** – vztahuje se k optickému prvku světla (element *Light*)
- ***MirrorController*** –vztahuje se k optickému prvku zrcadla (element *Mirror*)
- ***WallController*** – vztahuje se k optickému prvku stínítka (element *Wall*)
- ***HolographicPlateController*** – vztahuje se k světlocitlivému materiálu (element *HolographicPlate*)

6.1.4 Balík Gui

Ve chvíli, kdy vzniklo zadání této práce, tak ještě nebylo známo, že bude na projektu semnou někdo spolupracovat. Proto bylo do zadání této diplomové práce zakomponováno vytvoření jednoduchého grafického uživatelského rozhraní, aby byl simulátor publikovatelný.Avšak po několika dnech od vzniku zadání mé diplomové práce přišla slečna Lucie Herejtová a vyjádřila zájem o spolupráci na projektu a to především v oblasti grafického uživatelského rozhraní. Jelikož celý projekt je rozsáhlý, nebylo důvodu, proč kolegyni nepřizvat ke spolupráci. Po zaškolení do celé problematiky projektu, ihned začala s vývojem celého grafického rozhraní. Z toho důvodu již nebylo zapotřebí, abych realizoval jednoduchou implementaci grafického rozhraní, které je uvedeno v zadání mé diplomové práce.

Autorem tohoto celého balíku„*gui*“ je spoluautorka projektu Lucie Herejtová.Proto bude popis grafického uživatelského rozhraní uveden spíše rámcově. Popis jednotlivých částí tohoto balíku bude orientován spíše z funkčního pohledu vůči celému projektu. *Poznámka: Detailní popis celého grafického uživatelského rozhraní bude uveden v bakalářské práci Lucie Herejto*vé.

Balík „*gui*“ se vztahuje kompletně celý ke grafickému rozhraní programu. V tomto balíku jsou obsaženy jak vykreslovací metody, tak metody pro obsluhu uživatelských akcí.

6.1.4.1 Třída VisualizationRoot

Hlavní třída pro vytvoření a spuštění grafického rozhraní. Instance je vytvářena jako následník knihovní třídy *JFrame*. Třída *VisualizationRoot* je vytvářena při každém spuštění programu. V jejím konstruktoru jsou uvedeny dílčí inicializační metody nutné pro vytvoření základního grafického okna.

Základní okno je děleno do tří hlavních oblastí. První je při horním okraji, kde je umístěna úzká lišta s hlavním menu. Druhá oblast se nachází při pravém okraji základního okna, kde je umístěn panel pro zobrazování informací ojednotlivých elementech (optických prvcích). Zbytek okna připadá třetí oblasti, ve které je umístěn panel, kde se vykreslují jednotlivé simulace. Detailnější informace o funkcionalitě jednotlivých panelů jsou uvedeny v následujících odstavcích. Na obrázku 9.1, který je uveden v deváté kapitole, je vidět rozložení jednotlivých panelů, které byly právě popsány.

Přehled nejdůležitějších atributů:

- `OticalSimulator opticalSimulator` – odkaz na hlavní třídu aplikace
- `Workspace workspace`– odkaz na třídu řešící vykreslování simulací
- `ArrayList<Box> boxArrayList`–seznam s jednotlivými simulacemi.
- `int width, height`– rozměry hlavního okna (v pixelech)

6.1.4.2 Třída Language

Třída *Language* zpracovává požadavky na jednotlivé jazykové mutace. V současné době je k dispozici pouze mutace pro český a anglický jazyk.

Součástí třídy jsou dvě hlavní metody„*changeLanguage()*“, která se stará o změnu jazykové mutace, a metoda *„getText()“*, která vrací text v požadovaném jazyce podle aktuálně nastavené jazykové mutace.

6.1.4.3 Balík Workspace

V tomto balíku je umístěna funkcionalita vztahující se k vykreslování simulací (optických sestav) na monitor.

Třída Workspace - je výchozí třídou pro celý balík. Tato třída má za úkol uchovávat seznam aktuálně otevřených náhledů na jednotlivé simulace. Je vytvořena jakonásledníkknihovní třídy *JTabbedPane*, díky čemuž je tříděumožněnografické zobrazení a obsluha seznamu náhledů, které spravuje. Vypsání seznamu náhledů je realizováno v podobě jednotlivých záložek při horním okraji okna. Při výběru některé záložky je zobrazen aktuální náhled, ke kterému se daná záložka vztahuje.

Třída View-realizuje zobrazení jednotlivých náhledů na simulaci. Obstarává vykreslování elementů (optických prvků), přičemž zároveň implementuje rozhraní *ViewListener*, díky čemuž je umožněna manipulace s jednotlivými elementy (optickými prvky).

Třída ViewListener- obsluhuje veškeré události, které se týkají vstupů od uživatele, a to jak událostí z klávesnice, tak i myši.

Třída Split- umožňuje rozdělení náhledu (instance třídy *View*) na nějakou simulaci na dva menší vzájemně nezávislé náhledy (Vykreslovací plocha původního náhledu, je rozdělena na dvě menší vykreslovací plochy, kde budou následně vykreslovány dva nově vzniklé nezávislé pohledy na simulaci).

6.1.4.4 Balík ElementAction

V tomto balíku je umístěna abstraktní třída *AddElement*, která slouží jako vzorový předpis pro dědice. Třída je implementována jako následník třídy *AbstractAction*, díky čemuž obsahuje metody, které reagují na uživatelské požadavky. Konkrétně se jedná o uživatelský požadavek na přidání nového elementu.

Ostatní třídy (*AddHolographicPlate, AddMirror, AddLight, AddWall*) jsou vytvořeny jako následníci třídy *AddElement*, proto každý z nich implementuje vlastní metodu *„addElemet()“*, ve které je obstaráno správné založení a vytvoření nového objektu.

6.1.4.5 Balík Sidebar

V tomto balíku jsou uloženy třídy vztahující se k zobrazení a obsluze nastavení jednotlivých elementů (optických prvků). Zobrazení nastavení je umístěno při pravém okraji hlavního okna. O vykreslování informací se stará třída*InfoPanel*, která by se dala charakterizovat jako obalovýpanel, ve kterémje zobrazovánonastavení pro jednotlivé optické členy.

V balíku je dále umístěna abstraktní třída *ElementPanel*, která zastřešuje většinu společných nastavení. Zároveň obsahuje abstraktní metodu *„updateElementSpecific-Info()“*, kterou musí každý následník implementovat. Pomocí ní jsou následně zobrazována specifická nastavení pro konkrétní elementy (optické prvky).

Seznam jednotlivých tříd zděděných od třídy *ElementPanel*:

- ***LightPanel*** – nastavení pro optický prvek světla (element *Light*)
- ***MirrorPanel*** – nastavení pro optický prvek zrcadla (element *Mirror*)
- ***WallPanel*** – nastavení pro optický prvek stínítka (element *Wall*)
- ***HolographicPlatePanel*** – nastavení prosvětlocitlivý materiál (element *HolographicPlate*)

6.1.4.6 Balík menu

V tomto balíku jsou umístěny veškeré uživatelské akce, které jsou nějakým způsobem spojeny s hlavním menu nebo s popup menu (menu zobrazované při stisku pravého tlačítka myši). Jednotlivé akce jsou rozděleny do několika tříd podle akce, ke které se vztahují.

Seznam jednotlivých tříd:

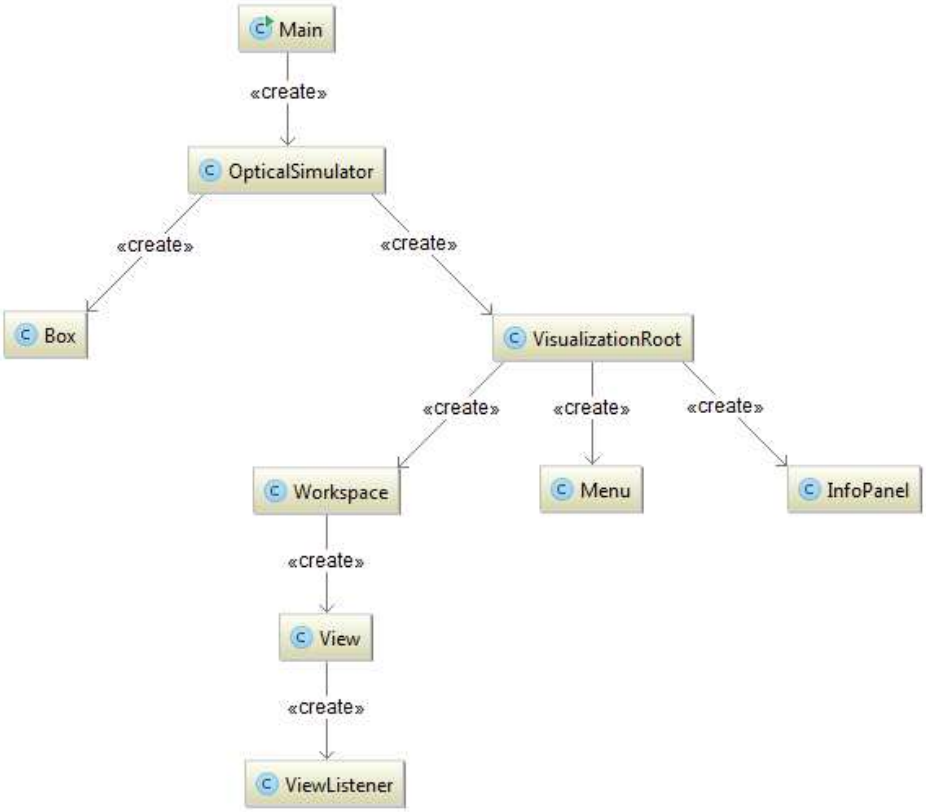
- ***AddTable*** – akce pro přidání nové simulace (nový stůl)
- ***AddView*** – akce pro přidání nového náhledu na nějakou simulaci
- ***ChangeTable*** – akce pro nastavení náhledu na jinou simulaci
- ***CopyComponent*** – akce pro kopírování elementu
- ***DeleteComponent*** – akce pro smazání elementu
- ***DeleteSplit*** – akce pro odstranění náhledu
- ***DiffMaximsPopUp*** – akce pro výběr difrakčních maxim
- ***LoadWorkspace*** – akce pro načtení simulace ze souboru
- ***SaveWorkspace*** – akce pro uložení simulace do souboru
- ***Menu*** – třída obsluhující funkcionalitu hlavního menu
- ***PasteComponent*** – akce pro vložení kopírovaného elementu
- ***PopupListener*** – třída obsluhující události související s popup menu
- ***SplitHorizontali*** – akce pro rozdělení náhledu horizontálně
- ***SplitVerticali*** – akce pro rozdělení náhledu vertikálně
- ***ToggleButton*** – akce pro události při přidávání nových elementů
- ***GeneralSettings*** – třída pro zobrazení a obsluhu globálního nastavení.

6.2 Struktura celého projektu

Tato podkapitola pojednává o vzájemných závislostech jednotlivých tříd. Zároveň se snaží vyzdvihnout nejdůležitější vztahy a procesy, ke kterým dochází v průběhu simulace i mimo ni.

6.2.1 Vazby při spuštění programu

Na obrázku 6.2 je vyobrazeno postupné vytváření jednotlivých tříd tak, jak se vytvářejí při spuštění simulátoru. Při pokynu spuštění operační systém načte třídu *Main*, kde se nachází stejnojmenná spouštěcí funkce, kterou spustí. Tato funkce se dále postará o načtení celého optického simulátoru.Jednotlivé vazby a závislosti mezi třídami simulátoru jsou přehledně zobrazeny na obrázku 6.2.



Obrázek 6.2vazby objektů

6.2.2 Vazby v datovém modelu

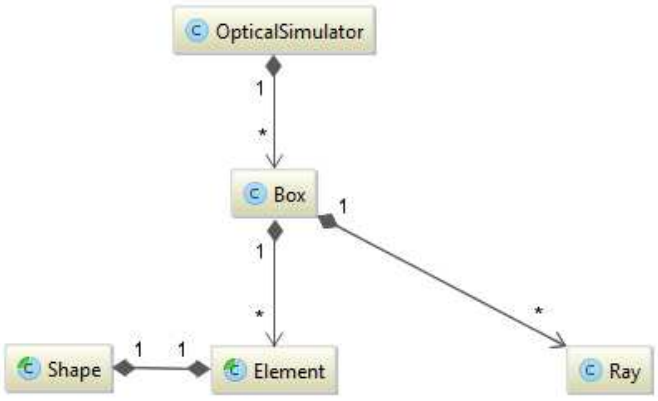
Na obrázku 6.3 jsou vidět vztahy mezi základními třídami, které reprezentují jednotlivé objekty. Z obrázku je patrné, že hlavní třída *OpticalSimulator* uchovává znalost jednotlivých simulací (instancí třídy *Box*). Informace o těchto simulacích jsou uloženy v seznamu *„OpticalSimulator.listBox“*.

Z vazby mezi třídami *Box* a *Element* vyplývá, že třída *Box* obsahuje seznam *„listElements“*jednotlivých elementů (optických prvků). Jedná se prakticky o nejdůležitější seznam celého simulátoru, jelikož obsahuje všechny elementy (optické prvky), které se vztahují ke konkrétní simulaci.

Třída *Box* dále obsahuje dva seznamy paprsků, které v simulaci vznikají. V jednom seznamu jsou uvedeny paprsky vedoucí mezi dvěma optickými prvky*(element1 „zdrojový optický prvek“ osvětluje element2 „osvětlený optický prvek“).*Ve druhém seznamu jsou paprsky, které obsahují pouze zdrojový optický prvek, ale žádný jiný optický prvek neosvětlují*(element1 „zdrojový optický prvek“ neosvětluje nic).*

Vazba 1:1 mezi objekty *Shape* a *Element* je především z důvodu přehlednosti celého kódu. Veškerá funkcionalita ze třídy *Shape* by mohla být vložena do třídy *Element* a nevznikl by žádný problém. Jelikož třída *Elemet* a její následníci jsou již tak dosti rozsáhlé třídy, bylo rozhodnuto, že bude kód rozdělen do dvou tříd *Shape* a *Element*.

Díky tomuto rozdělení zároveň vznikla nová vlastnost simulátoru, která sice ještě není implementována, ale do budoucna jistě bude. Toto rozdělení do dvou tříd umožní při simulaci měnit předdefinované tvary (třídy *Shape*) jednotlivým optickým prvkům (třídy *Element*) přímo při běhu simulace. Snadno si lze tuto skutečnost vysvětlit na příkladu. Například při simulaci je zkoumán odraz paprsku od rovného zrcadla, které má tvar čtverce, následně chceme změnit tvar zrcadla ze čtvercového na kulaté (rozměry, pozice, natočení, apod. chceme zachovat naprosto totožné). Při této konstrukci, která je nyní implementována, bude stačit pouze vyměnit objekt *Shape*čtvercového zrcadla za objekt *Shape*kulatého zrcadla, čímž získáme potřebný efekt.

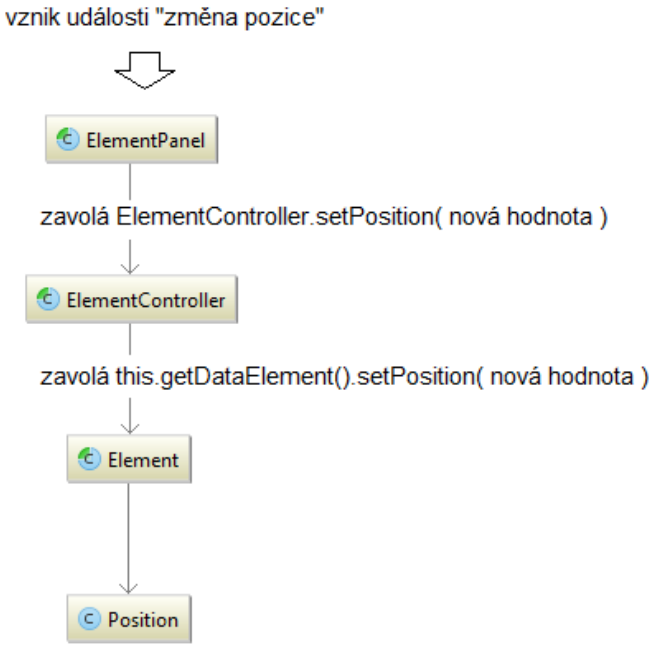


Obrázek 6.3 vazby v datovém modelu

6.2.3 Vazby mezi třídami z pohleduzměny dat v GUI

Komunikace mezi grafickým uživatelským rozhraním a datovým modelem probíhá přes třídu *ElementContoroller*. Tato třída společně se svými následníky vytváří vrstvu oddělující datový model od uživatelského rozhraní. Díky tomu nebude v budoucnu nikterak problematická případná změna grafického rozhraní bez nutnosti zásahu do datového modelu a naopak. Třída *ElementController* a její následníci se starají o částečnou validitu zadávaných dat.

Na obrázku 6.4 je vidět jak by probíhala případná změna pozice elementu (optického prvku), kterou zadal uživatel pomocí grafického uživatelského rozhraní.



Obrázek 6.4

6.3 Sledování paprsků (ray tracing)

Sledování paprsků je implementováno v metodě *„rayTracing()“*, která je umístěna ve třídě *Box*. Metoda obstarává nejdůležitější část celého simulátoru, jelikož je zodpovědná za zpracování všech paprsků, které se v jednotlivých simulacích vyskytují.

Metoda na zpracování paprsků vychází z teorie prohledávání grafu do hloubky. Celou optickou sestavu je možné přirovnat ke grafu, jehož uzly jsou jednotlivé elementy (optické prvky) a hrany jsou jednotlivé paprsky. Následně se prochází grafem od vstupních uzlů (optických prvků, z nichž vychází paprsky, např. element *Light*) a prozkoumávají se jednotlivé dostupné hrany (paprsky). Dočasná fronta objevených, ale neprozkoumaných hran (paprsků) je realizována zásobníkem.

Informace o chování a principu prohledávání grafů do hloubkybyly čerpány ze skript *„Teorie grafů, diskrétní optimalizace a výpočetní složitost 1“* [1], kde je uvedena celá jedna kapitola zabývající se problematikou prohledávání grafů. Zároveň je tam také uvedena ukázka algoritmu, z něhož jsem vycházel.

Jak algoritmus funguje? Ihned po spuštění metody *„rayTracing()“* dojde k vyčištění seznamů paprsků *„listCalculationRay“* a *„listCalculatinRayNo-Intersection“*. Následně se nadefinují a vynulují pomocné proměnné, které jsou při výpočtu využívány.

Poté se projde seznam*„listElements“*všech elementů (optických prvků), které jsou součástí simulované scény. Při průchodu seznamem je u každého optického členutestován příznak *„isLightSource“*,jehož pomocí se rozlišuje, které elementy (optické členy) jsou počátečním zdrojem světla. U elementů, které jsou světelnými zdroji, se zkopíruje seznam výstupních paprsků *„listOutpuRay“* do seznamu nezpracovaných paprsků *„listNoCalculationRay“*.

Následně je tento seznam *„listNoCalculationRay“* procházen a postupně jsou zpracovávány jednotlivé paprsky, které jsou v něm obsaženy.

Při průchodu se vybere vždy poslední paprsek ze seznamu a uloží se dopomocné proměnné *„actualRay“*.

Poté je znovu procházen celý seznam *„listElements“* s jednotlivými elementy (optickými prvky) a jsou testovány průsečíky elementů (optických prvků)s aktuálně vybraným paprskem *„actualRay“*. Pokud je nalezen průsečík, je automaticky uložen do pomocného seznamu *„listPositionAndElement“*, ve kterém je uložena jak přesná pozice průsečíku, tak odkaz na element (optický prvek), ke kterému se daný průsečík vztahuje. Tímto způsobem je otestován celý seznam *„listElements“*. Výsledkem tohoto testování je seznam všech protnutých objektů, aktuálně vybraným paprskem *„actualRay“*.

Jelikož však paprsek může osvětlovat pouze jeden element (optický prvek), je nutné najít nejbližší optický prvek od počátečního bodu paprsku *„startPosition“*. Proto následuje zpracování pomocného seznamu*„listPositionAndElement“*, kde jsou uloženy jednotlivé průsečíky a elementy (optické prvky), ke kterým se vztahují.

Seznam je tedy postupně procházen a na základě nejmenší vzdálenosti mezi aktuálně vybraným průsečíkem z *„listPosititonAndElement“* a počátečním bodem aktuálního paprsku *„startPosition“* je vybrán nejbližší průsečík.

Je-li tedy nalezen nejbližší průsečík, je zároveň nalezen i nejbližší element (optický prvek), který je paprskem osvícen. Může tedy dojít k nastavení koncového bodu *„endPosition“* a koncového elementu *„endElement“* u aktuálně zpracovávaného paprsku *„actualRay“*.Poté je aktuální paprsek přidán do seznamu vypočtených paprsků *„listCalculationRay“*a u protnutého elementu (optického prvku) je spuštěnametoda*„calculation(Ray)“*,které je předán aktuální osvětlující paprsek.

Jelikož metodu *„calculation(Ray)“* implementuje každý element (optický prvek) různým způsobem, nelze jednoduše popsat, jak probíhá zpracování vstupního (osvětlujícího) paprsku. Lze pouze podotknout, že metoda *„calculation(Ray)“* po provedení výpočtů se vstupním paprskemvrátí seznam nově vzniklých paprsků, které se uloží do seznamu *„listNoCalculationRay“*, kde budou později zpracovány.

Poznámka: Jedná-li se o optický prvek, který vyžaduje zpracování všech vstupních paprsků najednou, tak metoda „calculation(Ray)“ provádí pouze uložení vstupního paprsku do seznamu „listInputRay“ a následný výpočet výstupních paprsků proběhne až později.

Po dokončení metody *„calculation(Ray)“* dojde k odebrání aktuálního paprsku *„actualRay“* ze seznamu *„listNoCalculationRay“* a přistoupí se ke zpracovávání dalšího paprsku z tohoto seznamu. Cyklus *do-while* je prováděn do té doby, dokud nebude seznam *„listNoCaluculationRay“* zcela vyprázdněn.

Ve chvíli, kdy je seznam *„listNoCalculationRay“* zcela vyprázdněn, tak je přistoupeno ke zpracování vstupních paprsků, na optických členech, které mají nastaven příznak *„isAllRay“*. Proto je znovu prozkoumán seznam všech elementů (optických prvků) a jsou vybrány jen ty, které mají nastaven příznak *„isAllRay“* na hodnotu *„true“*. U těchto vybraných elementů je následně spuštěna metoda *„calculationAllRay()“*.

Jelikož metodu *„calculationAllRay()“* implementujekaždý element (optický prvek) různým způsobem, nelze ji tedy jednoduše popsat. Lze pouze podotknout, že metoda *„calculationAllRay(Ray)“* po provedení výpočtů se všemi vstupními paprsky ze seznamu *„listInputRay“* vrátí seznam nově vzniklých paprsků (*Pozor:Metoda „calculationAllRay(Ray)“ může vracet také prázdný seznam.*). Vrácený seznam je následně uložen do seznamu *„listNoCalculationRay“*.

Poté následuje znovu otestování, zda není seznam *„listNoCalculationRay“* prázdný. Je-li seznam prázdný, tak je metoda pro sledování paprsků u konce *(Byly zpracovány všechny paprsky)*. Není-li seznam *„listNoCalculationRay“* prázdný, tak opět následuje procházení a postupné zpracování jednotlivých paprsků z tohoto seznamu až do té doby dokud není seznam zcela prázdný. Poté jsou opět spuštěna metoda *„calculationAllRay()“*u všech elementů (optických členů), které vyžadují

zpracování všech paprsků najednou. Tento způsob se neustále opakuje, dokud nejsou zpracovány všechny paprsky v simulaci.

Poznámka: pokud nebude nalezen průsečík aktuálně zpracovávaného paprsku „actualRay“ s žádným elementem (optickým prvkem), bude paprsek přidán do seznamu „listCalculationRayNoIntersection“.

6.4 Ukládání simulací do souboru

Ukládání simulací do souboru, je implementováno na vodopádovém principu. Tedy jednotlivé objekty, které obsahují data pro uložení, se vzájemně hierarchicky volají, podle závislostních vazeb.

V metodě, která je volána jako první, je při ukládání vytvořen objekt, ve kterém je otevřen a načten soubor, do něhož bude ukládání probíhat. Odkaz na tento objekt si metody předávají jako vstupní parametr. Soubor tedy není nutné pokaždé znovu načítat a otvírat při volání jednotlivých metod *„save()“*.

Metody si jako druhý parametr předávají textový řetězec, ve kterém je obsažena sekvence tabulátorů. Tento řetězec je vkládán vždy před začátek každého řádku tak, aby vznikalo různé hloubkové odsazení pro jednotlivé třídy a jejich ukládané hodnoty. Díky tomuto řešení je výsledný uložený soubor přehlednější. V příloze č. 1 je uveden příklad jedné uložené simulace do souboru, která obsahuje světlo, zrcadlo a stínítko.

Hlavní metoda pro ukládání je umístěna ve třídě*OpticalSimulator*a je pojmenována *„saveWorkspace()“*. V metodě je implementováno nejprve uložení základních informací vztahujících se k aktuálně spuštěnému simulátoru.Až poté jsou ukládány informace o jednotlivých simulacích.

Realizace uložení jednotlivých simulací je vytvořena tak, že je procházen seznam otevřených simulací*„listBox“* a u každé simulace je spuštěna metoda*„save()“*, která je součástí třídy*Box*.Tato metoda již dále sama zajistí uložení veškerých informací o dané simulaci.

Metoda *„save()“* ze třídy *Box*, nejprve uloží veškeré informace o dané simulaci, a poté přikročí k uložení jednotlivých elementů (optických prvků). Uložení probíhá tak, že je procházen seznam elementů (optických prvků) *„listElements“* a u každé třídy *Controller* aktuálně vybraného elementu je spuštěna metoda *„save()“*.

Tato metoda nejprve uloží veškeré informace obsažené ve třídě *Controller* a poté zavolá metodu *„save()“* pro uložení dat že třídy *Element*.

Ve třídě *Element* je uložení implementováno tímzpůsobem, že nejprve je zavolána metoda pro uložení globálního nastavení *„saveGlobalSettings()“* (proměnné, které mají všechny elementy (optické prvky) stejné např. *pozice*, *velikost* atd.). Poté je zavolána metoda *„saveLocalSettings()“*, která uloží specifické proměnné pro konkrétní elementy (optické členy). A jako poslední je volána metoda *„save()“*ze třídy *Shape*, která má za úkol uložit informace o tvaru elementu (optického prvku).

6.5 Načítání simulací ze souboru

Načítání simulací ze souboru je realizováno obdobným způsobem jako ukládání simulací do souboru. Simulátor načte a otevře soubor s uloženou simulací do knihovní třídy Scanner, která umožňuje čtení souboru po řádcích. Jednotlivé metody si odkaz na tento objekt předávají tak, aby nemusely neustále otvírat a načítat soubor se simulací.

Obecně by se nechal princip načítání popsat takto: metoda načte řádek a rozdělí jej na dvě části. Poté vezme první část a otestuje, zdali je načtený řádek určen pro zpracování touto metodou či nikoliv. Pokud ano, metoda zpracuje i druhou část řádku, kterou uloží do požadované proměnné, nebo vytvoří nový objekt, u kterého následně zavolá metodu pro pokračování v načítání. Pokud je zjištěno, že daný řádek není určen pro zpracování touto metodou, je celý aktuálně načtený řádek vrácen jako návratová hodnota o úroveň výše. Tam ji buď metoda zpracuje, nebo ji předá opět o úroveň výše.

Ve třídě *OpticalSimulator* je metoda *„loadWorksapace()“*, která je spouštěna jako první při načítání uložených simulací ze souboru. V této metodě je nejprve otevřen soubor se simulacemi. Následně je načten první řádek, který je rozdělen na dvě části. První část se otestuje vůči vícenásobnému if příkazu, který obsahuje jednotlivé hodnoty, na které má daná metoda reagovat. Pokud je nalezená hodnota například *„SAVE_CONSTANT_BOX,“* tak je vytvořen nový objekt typu *Box*, do kterého se budou nahrávat následující načtené elementy. Proto následuje zavolání metody *„load()“* u nově vytvořené instance *Box* a ta se již o zbylé načtení postará sama.

Metoda *„load()“* ve třídě *Box* nejprve načte všeobecné informace o simulaci a následně začne načítat informace o jednotlivých objektech. Tedy načte-li metoda například hodnotu *„SAVE_CONSTATNT_LIGHT_CONTROLLER“*, tak vytvoří nový objekt *LightController*, u kterého následně zavolá metodu *„load()“*, která začne načítat hodnoty pro daný objekt. Narazí-li tato metoda při načítání na hodnotu *„SAVE_CONSTANT_LIGHT“*, vytvoří objekt *Light* a spustí u něho metodu *„load()“*, která se bude starat o načítání dále. Tímto způsobem je provedeno i následné načtení tvaru (třída *Shape*).

Pokud kterákoliv z metod *„load()“*, které byly zmíněny výše, načte řádek s hodnotou, kterou nezná, vrátí celý načtený řádek jako svou návratovou hodnotu té metodě, která ji spustila. Tato metoda se pokusí řádek zpracovat. Pokud se jí to povede, pokračuje se klasickým způsobem v načítání. Nepovede-li se jí to, tak metoda vrátí celý řádek také jako svoji návratovou hodnotu své nadřazené metodě. Tímto způsobem se pokračuje do té doby, dokud není nalezena metoda, která načtený řádek dokáže zpracovat. Stane-li se, že nebude žádná metoda umět zpracovat daný řádek, hlavní metoda *„loadWorkspace()“* spustí výjimku.

7 Implementace optických členů

Tato kapitola je zaměřena na konkrétní popis implementace jednotlivých optických členů. V kapitole se zaměříme především na popis nejdůležitějších metod, které jsou pro jednotlivé optické prvky charakteristické. V celém simulátoru jsou prozatím implementovány čtyři základní optické členy.

7.1 Světelný zdroj (Light)

Světelný zdroj je implementován jako následník třídy *Element*, čímž je zaručeno, že bude nově vzniklá třída obsahovat všechny důležité metody pro správnou funkcionalitu celé třídy.

Aby světelný zdroj správně fungoval, jsou v této třídě vytvořeny dvě důležité pomocné metody *„setVectors()“* a *„createListOutputRay()“*.

První zmíněná metoda zajišťuje vygenerování seznamu *„listRayVectors“*, jehož každá položka bude obsahovat bod (*Position*) a směrový vektor (*Vector*). Tento seznam reprezentuje počáteční body a směrové vektory, ze kterých budou následně generovány jednotlivé výstupní paprsky.

Druhá metoda *„createListOutputRay()“*zajišťuje vygenerování již konkrétních paprsků, které se budou ze zdroje šířit. K vytváření jednotlivých paprsků využívá seznam *„listRayVectrors“*, ze kterého převede každou položku na jeden konkrétní paprsek. Zároveň tato metoda nastaví každému paprsku jednotlivé vlastnosti (intenzitu, vlnovou délku, apod.), které paprsek ponese.

Proč je vytváření paprsků dvoukrokové? Na tuto otázku je odpověď celkem snadná. Jde především o výpočetní optimalizaci.

První metoda,která vytváří seznam *„listRayVectors“*, vytvoří jednotlivé body a směrové vektory vůči základní poloze zdroje světla. Světelný zdroj je tedy v základní poloze *(pozice 0,0,0 a všechny osy objektu jsou rovnoběžné s osami souřadnicového systému)* a jednotlivé body a směrové vektory jsou dopočítávány vůči této poloze. Seznam*„listRayVectors“* tak stačí vytvořit pouze jednou a to při vytváření světelného zdroje, nebo při změně vstupních dat *(například: změna rozložení paprsků, změna počtu paprsků, apod…).*

Druhá metoda *„createListOutputRay()“*, která je volána častěji,využívá předdefinovaný seznam *„listRayVektors“* a již jen transformuje jednotlivé body a vektory podle pozice a natočení zdroje světla. Následně z těchto hodnot vytváří jednotlivé výstupní paprsky, které jsou ukládány do seznamu *„listOutputRay“*.

Ostatní metody, které se v této třídě nacházejí, nejsou nikterak zvláštní, a jejich funkčnost je prakticky totožná s funkčností metod,které budou popsány v následující kapitole, která se zabývá implementací nového optického členu.

7.2 Světlocitlivýmateriál (HolographicPlate)

Třída *HolographicPlate* simuluje chování světlocitlivého materiálu (Holografické desky), který dokáže zaznamenat vzor vznikající interferencí světla dopadajícího na světlocitlivý materiál z několika zdrojů. Tento vzor (hologram) lze následně osvítit a pozorovat difrakci světla, resp. hologram lze rekonstruovat. Třída obsahuje čtyři hlavní metody, které charakterizují chování celého optického prvku.

První metodou je *„createRecord()“*, která zajišťuje vytvoření záznamu. Tato metoda je spouštěna ve chvíli, kdy uživatel stiskne tlačítko pro vytvoření holografického záznamu.

Po spuštění metoda nejprve vytvoří seznam všech objektů, které osvětlují světlocitlivý materiál (HolographicPlate). Následně je tento seznam předložen uživateli, který vybere optický prvek, ke kterému se bude přistupovat jako k referenčnímu zdroji světla. Poté, co uživatel vybere referenční zdroj, metoda pokračuje.

Při vytváření záznamu je světlocitlivý materiál (dále jen holografická deska) rozdělen na jednotlivé segmenty, podle zadaného rozlišení. Následně jsou tyto segmenty postupně procházeny.

U každého segmentu je vždy prozkoumán seznam*„listInputRay“* vstupních paprsků a jsou vybrány všechny paprsky, které dopadají na plochu daného segmentu. Vybrané paprsky jsou následně vzájemně porovnávány, a pokud pocházejí nějaké dva paprsky ze stejného elementu (optického prvku), jsou sloučeny (zprůměrovány) do jednoho. Tímto mechanizmem se zajistí, že vybrané paprsky, které dopadají do daného segmentu, jsou každý z jiného optického prvku.

Je-li vytvořen seznam zprůměrovaných paprsků pro daný segment, může se přistoupit k vytvoření interferenčního záznamu.

Při vytváření interferenčního záznamu se nejprve vybere referenční paprsek. Vybírá se podle optického prvku, ze kterého vychází (referenční optický člen byl vybrán na začátku metody). Následně je procházen zbylý seznam zprůměrovaných paprsků a s každým z nich je vypočten koeficient, který je prostorovou frekvencí vzoru vzniklého interferencí dvou paprsků. Jednotlivé vypočtené koeficienty jsou následně uloženy.

Tímto způsobem je proveden výpočet všech segmentů a paprsků, které na ně dopadají.

Druhoumetodou je *„calculation(Ray inputRay)“*. Tato metoda se pouze zajišťuje uložení vstupního paprsku *„inputRay“* do seznamu *„listInputRay“*. Jako svou návratovou hodnotu vrací prázdný seznam typu *„ArrayList<Ray>“*.

Třetímetodou je *„calculationAllRay()“*, která zajišťuje zpracování všech vstupních paprsků. Metoda je volána až ve chvíli, kdy element (*HolographicPlate*) má k dispozici všechny vstupní paprsky.

Pokud je holografická deska (*HolographicPlate*) bez záznamu, tak tato metoda vrací prázdný seznam typu *„ArrayList<Ray>“*.

Obsahuje-li holografická deska již nějaký záznam, takdojde ke spuštění metody*„play()“*, kterázpracuje všechny vstupní paprsky, které se nacházejí v seznamu *„listInputRay“*. Výsledné výstupní paprsky metoda *„calculationAllRay()“* vrací jako seznam typu *„ArrayList<Ray>“*.

Asi se ptáte, proč tato metoda vyčkává s rekonstrukcí paprsků, až budou načteny do seznamu *„listInputRay“* všechny paprsky. Tato metodika je použita,jelikož do jednoho segmentu holografické desky může dopadnout několik rekonstrukčních (referenčních) paprsků (z jednoho optického členu) najednou. Jelikož jsme při vytváření záznamu tyto paprsky slučovali (průměrovali) do jednoho, musíme totéž provést i při rekonstrukci. Proto se s vytvořením rekonstrukčních paprsků čeká, až budou známy všechny vstupní paprsky.

Poslední zajímavou metodou, která je umístěna v této třídě, je metoda *„play()“*, kterázajišťuje rekonstrukci záznamu. Jak již bylo uvedeno výše, je tato metoda spouštěna až ve chvíli, kdy jsou v seznamu *„listInputRay“* načteny všechny paprsky, které osvětlují tento element (*HolographicPlate*).

Po spuštění metody je holografická deska rozdělena do jednotlivých segmentů podle rozlišení. Následně jsou procházeny jednotlivé segmenty a u každého segmentu je prohledán seznam *„listInputRay“*, z něhož se vyberou všechny paprsky, které dopadají do daného segmentu.

Tyto vybrané paprsky jsou poté sloučeny (zprůměrovány) do jednoho výsledného paprsku. Tento paprsek můžeme nazvat jako rekonstrukční.

Je-li tedy znám rekonstrukční paprsek, můžeme provést výpočet výstupních paprsků pomocí jednotlivých koeficientů, které byly zaznamenány při vytváření záznamu. Nově vzniklé paprsky metoda vrací jako seznamu typu *„ArrayList<Ray>“*.

7.3 Zrcadlo (Mirror)

U tohoto optického prvku je zajímavá pouze metoda *„calculation(Ray inputRay)“*, která zpracovává vstupní paprsek, na jehož základě následně generuje výstupní paprsek.

Při spuštění metody je vytvořena kopie paprsku. Zkopírovaný paprsek je následně přetransformován do nové pozice vůči základní poloze a natočení zrcadla. Poté je paprsku otočen směrový vektor a je provedena zpětná transformace paprsků vůči pozici a natočení zrcadla. Touto zpětnou transformací doje k přemístění paprsku do správné pozice v simulované scéně. Výsledný paprsek je následně vrácen v seznamu typu *„ArrayList<Ray>“*.

7.4 Stínítko (Wall)

Jedná se o optický element, který neobsahuje žádnou zvláštní metodu, ani speciální konstrukci. Jediná vlastnost, která stojí za zmínku je, že metoda *„calculation(Ray inputRay)“* negeneruje žádné výstupní paprsky, a proto vrací prázdný seznam typu *„ArrayList<Ray>“*.

8 Rozšíření (doplnění nových elementů)

Kapitola pojednává o tom, jakým způsobem lze do simulátoru implementovat další optické prvky. V následujícím textu se předpokládá alespoň částečná znalost struktury tříd v celém projektu, která je popsána v předchozích kapitolách. Zároveň je nutná alespoň částečná znalost jazyka Java.

Při popisu implementace nového optického členu bude vše vysvětlováno na praktickém příkladu, a zároveň s výkladem bude do projektu doplňován prvek optickéčočky (anglicky *Lens*).

8.1 Třída Lens

Nejprve je nutné vytvořit novou třídu, která bude následníkem abstraktní třídy *Element* (*src.model.Element*). V nově vzniklé třídě musí být implementovány všechny funkce, jejichž předpisy jsou definovány v rodičovské třídě.Dále je také nutné nadefinovat specifické atributy (hodnoty), které bude třída *Lens* využívat navíc oproti předdefinovaným atributům ve třídě *Element*.

Jelikož pro správnou implementaci optické čočky je kromě globálních atributů (*pozice, natočení, velikost,* …) nutné znát také ohniskovou vzdálenost, musí být v této třídě vytvořen atribut, jenž tuto hodnotu bude uchovávat.

V následující tabulce je uveden příklad jakým by mohl být atribut realizován.

Lens.java

```
private double focusLength;
```

Nyní bude vysvětlen význam jednotlivých metod, které je nutné implementovat pro správný chod simulátoru.

8.1.1 Metoda init

Metoda *„init()“* jespouštěna ihned po vytvoření objektu a slouží pro inicializaci parametrů. V této metodě může být například realizováno nastavení jednotlivých atributů z nějakého globálního nastavení apod.

Jelikož čočka neobsahuje žádné speciální parametry, které by musely být inicializovány touto metodou, není zapotřebí v této metodě cokoliv implementovat.

Zde je ukázka implementace metody „init()“

Lens.java

```
public void init(){}
```

Poznámka: Pokud bychom vytvářeli například nějaký zdroj světla, tak by se v této metodě implementoval aparát pro vygenerování jednotlivých paprsků, které by z daného zdroje světla vycházely.

8.1.2 Metoda saveLocalSettings

Metoda *„saveLocalSettings(PrintWriter writer, String indentation)“*je spouštěna vždy, když dochází k ukládání simulací. Konkrétně tato metoda by měla mít na starosti uložení veškerých informací, které se vztahují pouze k tomuto objektu (čočce).

Jinak řečeno, uložení atributů (*pozice, natočení, velikost, …*) je implementováno v metodě *„saveGlobalSettings()“*, která se nachází ve třídě *Element*, proto již není nutné tyto atributy ukládat. Oproti tomu ostatní atributy, které jsou specifické pouze pro tento optický prvek (čočku), by však měly být ukládány pomocí této metody „saveLocalSettings()“. V našem případě se jedná konkrétně o uložení ohniskové vzdálenosti.

Před samotným uložením atributu s ohniskovou vzdáleností do souboru je dobré vytvořit konstantu, která bude představovat jedinečný identifikátor pro daný atribut *focusLength*(ohniskovou vzdálenost).

Ideální místo pro vytvoření konstanty je ve třídě *OpticalSimulator*, kde se už podobné konstanty nacházejí. Konstanta by měla mít naprosto jedinečnou hodnotu vůči všem obdobným konstantám. Hodnota konstanty zároveň nesmí obsahovat znak, jenž je uveden v konstantě *„OpticalSimulator.SAVE_PLUS“*.

Zde je uvedena část kódu, jak by mohl vypadat zápis konstanty ve třídě *OpticalSimulator*.

OpticalSimulator.java

```
 public static String SAVE_CONSTANT_LENS_FOCUS_LENGTH= "FOCUS_LENGTH";
```

Nyní tedy lze přistoupit k uložení atributu*focusLength* (ohniskové vzdálenosti) do souboru.

Ve funkci je dobré nejprve vytvořit celý řádek (řetězec *String*) s ukládanou hodnotou. Jako první by se mělo do řetězce vložit odsazení (odsazení začátku řádku), které je obsaženo ve vstupní proměnné *indentation.* Poté by měla být vložena jednoznačná identifikační konstanta (byla vytvořena v předchozím kroku *„SAVE_CONSTANT_LENS_FOCUS“*). Následně je nutné vždy vložit konstantu *„OpticalSimulator.SAVE_PLUS“*, pomocí níž se odděluje identifikátor a hodnota atributu. Poté je na konec celého řetězce přidána hodnota atributu*focusLength* (ohniskové vzdálenosti).

Je-li tedy vytvořena celá řádka, stačí ji pouze uložit do výstupního souboru. Proto je zde druhý vstupní parametr *writer*, který obsahuje odkaz na objekt, pomocí jehož metody *„println()“* lze snadno uložit celý řádek do souboru.

Na následujících řádcích je uvedeno, jak by mohl vypadat kód pro uložení ohniskové vzdálenosti do souboru.

Lens.java

```
public void saveLocalSettings(PrintWriter writer, String indentation){
String line = indentation+
               OpticalSimulator.SAVE_CONSTANT_LENS_FOCUS_LENGTH+
               OpticalSimulator.SAVE_PLUS+ focusLength;
       writer.println(line);
}
```

8.1.3 Metoda loadLocalSettings

Metoda *„loadLocalSettings(Scanner input, Counter numberLine)“* je využívána pro načítání specifického nastavení elementu (čočky) ze souboru.

Jelikož je soubor strukturovaný, dojde nejprve k načtení globálních parametrů (*pozice, natočení, velikost,* …) metodou *„loadGlobalSettings()“*. Následně poté je spuštěna metoda *„loadLocalSettings()“*, která obstará načtení specifických atributů pro tento daný optický prvek (čočku). V tomto případě se jedná o načtení ohniskové vzdálenosti.

Jak by měla metoda správně fungovat? Ve chvíli, kdy dojte ke spuštění metody, by měla metoda převzít veškerou iniciativu nad načítáním jednotlivých řádek. To znamená, že by měla metoda načítat řádky tak dlouho, dokud je bude umět zpracovat. Pokud narazí na řádek, který neumí zpracovat, předá jej jako svou návratovou hodnotu a metoda se ukončí.

V našem případě je nejlepší vytvořit cyklus, který bude neustále načítat jednotlivé řádky. Při každém cyklu vezme aktuálně načtený řádek a rozdělí jej podle konstanty *„OpticalSimulator.SAVE_PLUS“* (tím bude řádek rozdělen na dvě části; identifikační část a část s hodnotou).

Následně vezme první (identifikační) část řádku a porovná ji s identifikačními konstantami atributů, které umí zpracovat. V případě optické čočky se bude porovnávatpouze s konstantou *„OpticalSimulator.SAVE_CONSTATNT_LENS_FOCUS“*. Budou-li si identifikátory rovny, přistoupí ke zpracování druhé části řádku a to tak, že bude spuštěna metoda *„Double.valueOf(druhá_část_řádky)“*, které je předána druhá část načtené řádky. Tato metoda převede vstupní řetězec na číslo typu *double* a vrátí jej jako návratovou hodnotu, která se již jen uloží do příslušného atributu *focusLength*.

Při načítání každé nové řádky je nutné zvýšit hodnotu objektu *Counter*, který je druhým vstupním parametrem *„numberLine“*. Zvýšení se provádí pomocí metody *„addOnes()“*. Toto počítadlo je zavedeno především proto, aby bylo při případné chybě v načítání snadné identifikovat vadný řádek. Při implementaci této metody není zapotřebí ošetřovat jakékoliv výjimky, které by mohli při načítání řádků nastat. Veškerý mechanizmus pro ošetření výjimek je implementován na vyšší úrovni.

Příklad jak by mohla vypadat implementace:

Lens.java

```
public String loadGlobalSettings(Scanner input, Counter numberLine){
    String line;
    String[] parseLine;

while (true){
        line = input.nextLine();
        numberLine.addOnes();
        parseLine = line.split(OpticalSimulator.SAVE_PLUS);

if(parseLine[0].trim().equals(
               OpticalSimulator.SAVE_CONSTANNT_LENS_FOCUS_LENGTH)){

 focusLength= Double.valueOf(parseLine[1]);

       }else{
return line;
        }
    }
}
```

8.1.4 Metoda copy

Metoda *„copy()“* je spouštěna při kopírování elementu (čočky). Při implementaci je důležité dbát na to, aby byly veškeré atributy dobře zkopírovány, a to především atributy, které uchovávají odkaz na nějaký objekt. Pokud se nějaký takový atribut vyskytne, je nutné vytvořit i kopii onoho odkazovaného objektu. Pokud by se toto pravidlo nedodrželo, budou si jednotlivé zkopírované elementy (optické členy) vzájemně měnit hodnoty a simulátor nebude fungovat dle očekávání.

Jak správně implementovat tuto metodu? Nejprve je nutno vytvořit nový objekt stejného typu (v našem případě objekt typu*Lens*). Poté se do tohoto objektu nakopírují jednotlivé atributy.

Pro zkopírování globálních atributů (pozice, natočení, apod.) je ve třídě *Element* vytvořena metoda *„copyGlobalAttributes(Element, ElementControler)“*, kterou stačí pouze zavolat. Metoda zkopíruje všechny globální atributy, do objektu, který je jí předán jako první vstupní parametr, a zároveň nastaví tomuto objektu odkaz na třídu typu *Controller*, který je jí předán jako druhý parametr.

Nakonec je nutnoprovést zkopírování všech specifických atributů, které jsou charakteristické pro daný optický prvek (v našem případě se jedná o zkopírování ohniskové vzdálenosti *focusLength*).

Zde je uveden kód, jak by mohla případně vypadat implementace celé metody.

Lens.java

```
public Element copy(ElementController elementController){
Lens copy = new Lens();
    copyGlobalAttributes(copy, elementController);
    copy.setFocusLength(this.getFocusLength());
return copy;
}
```

8.1.5 Metoda updatePosition

Metoda *„updatePosition()“*, je spuštěna pokaždé, když dojde ke změně pozice daného elementu (optického prvku). Je proto nutné, aby každý následník (v tomto případě optická čočka) implementoval tuto metodu tak, aby byl schopen reagovat na změnu pozice.

V našem případě optické čočky není nutné na změnu pozice nikterak reagovat. Proto může zůstat celá metoda prázdná, jak je vidět v tabulce níže.

Lens.java

```
public void updatePosition(){}
```

Poznámka: Kdybychom implementovali například nějaký zdroj světla, tak by bylo nutné v této metodě vytvořit aparát, kterým by se zajistila změna pozice i pro jednotlivé paprsky, které by tento zdroj světla opouštěly.

8.1.6 Metoda updateRotation

Metoda *„updateRotation()“*, se spouští ve chvíli, kdy dojde k pootočení elementu (optického prvku). Každý následník třídy *Element* musí tuto metodu překrýt svou vlastní implementací tak, aby byl schopen reagovat na pootočení.

V případě optické čočky není opět nutné nikterak reagovat na změnu natočení.Proto může zůstat i tato metoda prázdná.

Lens.java

```
public void updateRotation(){}
```

Poznámka: Metoda by se mohla využít opět při implementaci nějakého zdroje. V této metodě by se tak vytvořit kód, který by aktualizoval směry jednotlivých paprsků, které ze zdroje vycházejí.

8.1.7 Metoda updateDimension

Metoda „updateDimension()“ je obdobná jako předchozí dvě metody. Je spouštěna ve chvíli, kdy dojde ke změně rozměrů daného elementu (v tomto případě čočky). Proto je nutné, aby každý následník zde implementoval vlastní reakci na tuto událost.

V případě optické čočky není nutné implementovat žádné speciální činnosti, které by měly na tuto událost reagovat. Proto metoda zůstane prázdná.

Lens.java

```
public void updateDimension(){}
```

Poznámka: Metoda by byla opět využita například při implementaci plošného zdroje světla, kdy by tato metoda řešila změnu rozmístění jednotlivých paprsků v závislosti na rozměrech optického prvku.

8.1.8 Metoda getPositionIntersection

Metoda *„getPositionIntersection(Ray)“* je velice důležitou, jelikož testuje, zda vstupní paprsek, který je metodě předán jako parametr, protíná daný element (čočku) či nikoliv.

Při implementaci je dobré využít nejprve testu obalové koule,neboli spustit rychlý test, zda paprsek protíná obalovou kouli, ve které je uzavřen celý element (čočka). Metoda pro rychlý test je již vytvořena v rodičovské třídě *Element*, takže ji stačí pouze spustit *„Element.isIntersectionBall(Ray inputRay)“*.

Je-li obalová koule paprskem protnuta, nastupují na řadu přesnější výpočty na určení průniku paprsku s daným elementem.

Jelikož testovaný element (čočka) může být umístěn v různé pozici s různým natočením, je dobré nejprve přetransformovat testovaný paprsek a element (čočku)do pozic, aby se element (čočka) nacházel v počátku souřadnicového systému a jednotlivé osy elementu byly rovnoběžné s osami souřadnicového systému. Díky této transformaci se výrazně usnadní následující výpočty pro zjištění průsečíku.

V simulátoru jsou jednotlivé elementy (optické členy) uloženy v základní poloze, jsou tedy umístěny v počátku souřadnicového systému a osy elementu jsou rovnoběžné s osami souřadnicového systému. To znamená, že konkrétní pozice a natočení objektu je uložené v samostatných objektech (*Position* a*Matrix*). Díky této vlastnosti při transformaci testovaného paprsku a elementu (čočky) stačí tedy transformovat pouze testovaný paprsek.

K tomuto účelu jsou ve třídě *Ray* implementovány metody *„revertTranslation(Position)“* a *„revertRotation(Matrix)“*, kterým stačí předat pouze pozici*„Element.getPosition()“* a rotační matici*„Element.getMatrix()“* testovaného elementu (čočky). Tyto dvě metody na základě zadaných parametrů přetransformují testovaný paprsek do požadované pozice.

Jelikož jde v této kapitole především o ukázku implementace nového optického prvku, není na místě pouštět se do složitějších matematických výpočtů průniku paprsku s objektem určitého tvaru. Proto bude přijat fakt, že nyní by měl v implementaci metody následovat detailní výpočet průsečíku.

Existuje-li průsečík paprsku s elementem (čočkou), je nutné výsledný průsečík přetransformovat zpět do pozice původní scény. K tomuto účelu jsou ve třídě *Methods* implementovány metody *„translationPoint(přesouvaný_bod, Position)“* a *„rotatePointAroundCenterAxis(rotovaný_bod, Matrix)“*, které vrátí zadaný bod v požadované pozici.

Neexistuje-li průsečík paprsku s elementem (čočkou), metoda by měla vrátit hodnotu *„null“*.

Poznámka: Příklad implementace celé této metody je například ve třídě Mirror.

8.1.9 Metoda calculation

Metoda *„calculation(Ray inputRay)“* je určena pro výpočet paprsků, které vzniknou při dopadu nějakého paprsku na element (čočku). Je spuštěna pokaždé, když metoda *„rayTracing()“* (ze třídy *Box*) zjistí, že vstupní paprsek protíná daný element (čočku).

Při implementaci je opět doporučeno přetransformovat vstupní paprsek podle elementu (čočky) tak, aby byl element (čočka) umístěn v počátku souřadnicového systému a osy elementu (čočky) byly rovnoběžné s osami souřadnicového systému.

Následně by měla být implementována logika, která obstará výpočet směru paprsku, který vznikne na elementu (čočce) a bude simulovat výstupní paprsek. Detailní implementace této pasáže nemá opět nikterak přínosný význam k probíranému tématu, proto bude vycházeno z faktu, že nový (výstupní) paprsek byl nějakým způsobem dopočítán. Jakým způsobem již není podstatné.

Nově vzniklý (výstupní) paprsek má tedy bod, ze kterého vychází a směr, kterým vede. Jelikož ale byla před tím provedena transformace vstupního paprsku podle pozice a natočení elementu (čočky), je nutné na výstupní paprsek aplikovat také zpětnou transformaci. K tomuto účelu jsou ve třídě *Ray* implementovány metody*„rotation(Matrix)“* a *„translation(Position)“*, kterým stačí předat pouze pozici *„Element.getPosition()“* a rotační matici *„Element.getMatrix()*“, na základě kterých bude dopočítána transformace paprsku do původní scény.

Následně je nutné, abymetoda vrátila výsledný výstupní paprsek/yv seznamu typu *„ArrayList<Ray>“*, tak aby mohli být tyto paprsky simulátorem dále zpracovány.

8.1.10 Metoda calculationAllRay

Metoda *„calculationAllRay()“*, slouží pro zpracování všech vstupních paprsků ze seznamu *„listInputRay“* najednou. Tato vlastnost však pro realizaci optické čočky nebude potřeba, proto metoda zůstane prázdná.

Lens.java

```
public ArrayList<Ray> calculationAllRay(){
     return null;
}
```

Poznámka: Tato metoda je například využita při implementaci světlocitlivého materiálu (HolographicPlate), kde je nutné zpracovávat všechny paprsky najednou.

8.2 Třída LensShape

Aby byla zachována konvence programu je nově vytvářená třída nazvána jako *LensShape*.

Třída *LensShape* musí být vytvořena jako následník abstraktní rodičovské třídy *Shape*. Díky tomu bude zaručeno, že třída *LensShape* bude muset implementovat vlastní funkcionalitu metod, jenž jsou předdefinovány ve třídě *Shape*.

8.2.1 Metoda init

Metoda „*init()*“ je volána ihned po vytvoření objektu a slouží pro inicializaci parametrů třídy.V tomto konkrétním případě (vytváření čočky) se jedná o vytvoření matematické sítě bodů, a sítě bodů pro zobrazovací účely.

Pod pojmem matematická síť si lze představit množinu bodů, které charakterizují tvar optického prvku (čočky).Z jednotlivých bodů této sítě sevytvářízákladní tvar, který charakterizuje celý optický prvek, se kterým se poté provádějí matematické operace, jako jsou například nalezení průsečíku paprsku s optickým prvkem (čočkou) a podobně.Oproti tomu síť bodů pro zobrazovací účely charakterizuje tvar přesně tak, jak se bude optický prvek vykreslovat na obrazovce.

Matematická síť je realizována jako „*ArrayList<Position>*“, kde podle umístění v seznamu jsou rozlišovány jednotlivé stěžejní body matematické sítě. Ve třídě *Shape* jsou předdefinovány konstanty specifikující význam jednotlivých bodů v seznamu. Pro implementaci čočky bude nutné k těmto konstantám dodefinovat ještě jednu, která bude udávat pozici ohniska. Implementace by mohla vypadat například takto:

```
Shape.java

public final static int NUMER_NET_FOCUS = 5;
```

Vytvoření matematickésítě by poté mohlo být realizováno tímto způsobem:

```
LensShape.java

public void init(){
        numerNet = new ArrayList<Position>();
        numerNet.add(NUMER_NET_ORIENTATION, new Position(-1, 0, 0));
        numerNet.add(NUMER_NET_CENTER, new Position(0, 0, 0));
        numerNet.add(NUMER_NET_HALF_WIDTH, new Position(
                element.getDimension().getWidth() / 2), 0, 0);
        numerNet.add(NUMER_NET_HALF_HEIGHT, new Position(0,
                element.getDimension().getHeight() / 2, 0));
        numerNet.add(NUMER_NET_FOCUS,
                new Position(element.getFocusLength(), 0, 0));
}
```

Poznámka: Rozměry šířky a výšky jsou v této síti uloženy jako poloviční hodnoty. Neboli v těchto hodnotách není uložena celková šířka a výška objektu, ale je zde

uložena pouze poloviční šířka a výška (šířka a výška je brána od středu objektu do jednotlivých krajů). Tato realizace byla zvolena především kvůli snazší manipulaci s celou sítí.

Nyní se můžeme rozhovořit o tom, jak vytvořit co nejrealističtější síť bodů pro vykreslování tvaru objektu. To však není hlavním záměremtéto kapitoly. Proto zde bude pouze poukázáno, jakým způsobem se taková síť vytváří. Následně bude ukázka implementace sítěpro jednoduchý krychlový objekt.

Prvním krokem je definice jednotlivých bodů, ze kterých se bude tvar objektu skládat. Tyto body by měly být uloženy do pole *„vertices“*. V druhém kroku se z těchto bodů tvoří jednotlivé plochy, které budou uloženy do seznamu *„indices“*. Každá takto vytvořená a uložená plocha bude tvořena seznamem bodů, které ji budou vytyčovat.

V níže uvedeném rámečku je naznačen způsob, jakým by mohl být realizován tvar kvádru. Na stejném principu lze však vytvořit jakýkoliv požadovaný tvar.

```
// definice jednotlivých bodů sítě
vertices = new Position[8];
vertices[0] = new Position(-width, -height, +depth);
vertices[1] = new Position(+width, -height, +depth);
vertices[2] = new Position(+width, +height, +depth);
vertices[3] = new Position(-width, +height, +depth);
vertices[4] = new Position(-width, -height, -depth);
vertices[5] = new Position(+width, -height, -depth);
vertices[6] = new Position(+width, +height, -depth);
vertices[7] = new Position(-width, +height, -depth);

// definice jednotlivých poloch
indices = new ArrayList<int[]>(6);
indices.add(new int[]{0, 1, 2, 3});
indices.add(new int[]{1, 5, 6, 2});
indices.add(new int[]{5, 4, 7, 6});
indices.add(new int[]{4, 0, 3, 7});
indices.add(new int[]{3, 2, 6, 7});
indices.add(new int[]{4, 5, 1, 0});
```

8.2.2 Metodacopy

Metoda *„copy()“*jespouštěna ve chvíli, kdy dochází ke kopírování celého elementu (čočky). Návratovou hodnotou této funkce by měla být naprosto stejná kopie této třídy bez jakýchkoliv závislostí na vzorové třídě.

8.2.3 Metoda updateNumerNet

Tato metoda je spouštěna, dojde-li ke změně rozměrů elementu (čočky). Metoda by proto měla upravit obě sítě bodů, realizujících tvar elementu (čočky)v závislosti na nových rozměrech. Implementace metody je obdobná jako při vytváření matematické a vykreslovací sítě bodů.

8.3 Třída LensController

Další třída, kterou je nutné vytvořit, je třída *LensController*. Tato třída bude zprostředkovávat komunikaci mezi datovým modelem a grafickým rozhraním.Proto je nutné, aby byla vytvořena jako následník abstraktní třídy *ElementController* (*src.controller*).

8.3.1 Metoda copy

Metoda je volána při požadavku na vytvoření kopie elementu (optického prvku). Při kopírování elementů (optických prvků) je nejprve volána metoda *„copy()“* ze třídy *ElementController*,a až následně z této metody je volána metoda *„copy()“*, která je umístěna ve třídě *Element*.

Návratovou hodnotou této metody by měla být nově vytvořená kopie. Zde je uvedena případná možnost implementace.

```
LensController.java

public ElementController copy(){
    LensController newCopy = new LensController();
    newCopy.setDataElement(this.getDataElement().copy(newCopy));
return newCopy;
}
```

8.3.2 Metoda save

Metoda*„save()“*, slouží kuložení nastavení do souboru. Zde je důležité, aby byl do souboru zapsán řádek, jenž bude značit přesný typ třídy (v tomto případě *LensController*). To z důvodu, aby při načítání bylo snadno rozpoznatelné, o jakou konkrétní třídu se jedná.

Ideálním způsobem je vytvořit konstantu ve třídě *OpticalSimulator*, která bude obsahovat jedinečný identifikátor pro tuto konkrétní třídu.

```
OpticalSimulator.java

public final static String SAVE_CONSTANT_LENS_CONTROLLER
       = "LENS_CONTROLLER";
```

Při spuštění metody*„save(PrintWriter writer, String indentation)“* jí jsou předány dva vstupní parametry. První vstupní parametr je odkaz na knihovní třídu *PrintWriter*, jenž obsahuje otevřený soubor, do kterého jsou ukládány jednotlivé informace. Druhým parametrem je řetězec obsahující sekvenci tabulátorů, které je dobré umístit na začátek každého řádku. Díky tomu je výsledný výstupní soubor částečně strukturován a lze se v něm tedy lépe orientovat.

Co se týče samotné implementace, ta by mohla vypadat nějak podobně, jako je tomu v rámečku pod tímto odstavcem. Na první řádce je provedeno uložení jedinečného identifikátoru pro tuto třídu a na další řádce je zavolána metoda *„save()“* pro uložení dat ze třídy typu *Element* (konkrétně třída *Lens*).

```
LensController.java

public void save(PrintWriter writer, String indentation){
    writer.println(indentation+
                OpticalSimulator.SAVE_CONSTANT_LENS_CONTROLlER +
                OpticalSimulator.SAVE_PLUS);
    getDataElement().save(writer, indentation+"\t");
}
```

8.3.3 Metoda load

Metoda *„load(Scanner input, Counter numberLine)“*realizuje načítání zesouboru. Metoda by měla postupně procházet jednotlivé řádky, a ve chvíli, kdy narazí na řádek obsahující informace vztahující se k datové třídě (třída *Lens*), by měla vytvořit novou instanci této třídy. Zároveň byměla zavolat metodu *„load()“*z této nově vytvořené třídy. Tato metoda by pak již veškerou obsluhu načítání parametrů řešila sama. Zde je nástin, jak by případně mohla metoda vypadat.

```
LensController.java

public String load(Scanner input, Counter numberLine){

    String line,returnLine = "";

while (true){
if(returnLine == "") {
            line = input.nextLine();
            numberLine.addOnes();
        }else{
            line = returnLine;
            returnLine = "";
        }
        String[] parseLine = line.split(OpticalSimulator.SAVE_PLUS);

       if(parseLine[0].trim().equals(
                       OpticalSimulator.SAVE_CONSTANT_LENS)){
            Lens lens = new Lens();
            setDataElement(lens);
            returnLine = lens.load(input,numberLine);
        }else{
return line;
        }
    }
}
```

Poznámka: Při implementaci této metody, není nutné ošetřovat případné výjimky, které by mohli při načítání vzniknout. Veškerý mechanizmus pro ošetření výjimek je realizován na vyšší úrovni.

8.4 Třída AddLens

Dále je nutné vytvořit třídu *AddLens*, která bude reagovat na události tlačítek pro přidání nového elementu (v tomto případě čočky). Třídu *AddLens* je dobré vytvořit v balíku *„src.gui.elementAction“*, kde se zároveň nachází rodičovská třída *AddElement*, od níž bude třída *AddLens* vytvořena jako její následník.

Třída *AddLens* musí mít implementovánu vlastní metodu *„addElement(View view, Position position)“*. Tato metoda je spuštěna pokaždé, když přijde pokyn od uživatele na přidání nového elementu (čočky).

Doporučené chování metody je ve vytvoření nového objektu *LensController* a jeho přidání do seznamu *„listElements“* dané simulace. Následně by měla být spuštěna metoda pro překreslení simulace na obrazovce.

Další důležitou součástí třídy je konstruktor, který by měl mít v sobě nadefinován text, který bude zobrazován na tlačítkách. Zde je také možné nadefinovat klávesové zkratky, na které bude tato třída reagovat.

V níže uvedeném rámečku je příklad, jak by mohla realizace třídy vypadat.

AddLens.java

```
public class AddLens extends AddElement {

public AddLens(View view) {
super(view, getText("menu_add_lens", "Lens"));
        putValue(MNEMONIC_KEY, new Integer(KeyEvent.VK_E));
        putValue(ACCELERATOR_KEY,KeyStroke.getKeyStroke(
                                KeyEvent.VK_E, ActionEvent.CTRL_MASK));
    }

public void addElement(View view, Position position) {
        LensController newLens = new LensController(view.getBox(),
                position, Matrix.getDefaultMatrix(),
                new Dimension(1, 100, 100),
                getText("menu_add_lens", "Lens") + " " +
                Workspace.lensCount++);
        view.getBox().addElementController(newLens);
        view.repaintAllViews(true, true);
        view.setLastActive(newLens);
        VisualizationRoot.INFO_PANEL.refreshInfo(view);
    }
}
```

8.5 Třída LensPanel

Třída *LensPanel* realizuje zobrazení hodnot aktuálně vybraného elementu (čočky) v panelu nastavení, jenž je zobrazován při pravém okraji. Aby byly zachovány standardy rozmístění tříd, měla by být třída *LensPanel* vytvořena v balíku *„src.gui.sidebar“*. V tomto balíku se zároveň nachází třída *ElementPanel*, která bude rodičovskou třídou pro třídu *LensPanel*.

Při implementaci této třídy, je nutné v konstruktoru vytvořit jednotlivé ovládací prvky grafického rozhraní. Tím, že je třída následníkem třídy *ElemenPanel*, není již nutné starat se o ovládací prvky nastavující pozici a natočení tohoto optického členu. Ty jsou do každého ovládacího panelu přidávány automaticky.

Pro čočku bude tedy nutné vytvořit ovládací prvky pro *výšku*, *šířku* a *ohniskovouvzdálenost*. Pro větší přehlednost kódu je lepší implementaci prvků provést v samostatné metodě, a tu poté již jen spustit z konstruktoru.

Zde je nástin, jak by mohla vypadat část kódu pro vytvoření ovládacího prvku.

```
LensPanel.java (část kódu pro implementaci jednoho prvku)

JLabelheight= new JLabel(getText("height", "Height"));
setConstraints(0, lastInRow(), 2, 1, GridBagConstraints.CENTER);
add(height);

JLabel heightLabel = new JLabel(getText("height", "Height"));
setConstraints(0, currentRow(),1,1,GridBagConstraints.FIRST_LINE_END);
add(heightLabel);

heightSpin= new JSpinner(newSpinnerNumberModel(
                        element.getDimensionHeight(),START_RESOLUTION,
                        END_RESOLUTION, STEP_RESOLUTION));

((JSpinner.NumberEditor) heightSpin.getEditor()).getTextField().
                        setBackground(getBackground());
((JSpinner.NumberEditor) heightSpin.getEditor()).getTextField().
                        setColumns(TEXTFIELD_WIDTH);

setConstraints(1,lastInRow(),1,1,GridBagConstraints.FIRST_LINE_START);

heightSpin.addChangeListener(new ChangeListener() {
public void stateChanged(ChangeEvent e) {
element.setDimensionHeight(heightSpin.getValue());
view.repaintAllViews(true, true);
    }
});
add(heightSpin);
```

8.6 Úprava třídy InfoPanel

Aby mohlo být nastavení realizováno a zobrazováno v postraním pravém panelu, je nutné přidat rozhodovací podmínku do třídy *InfoPanel*, ve které je na 55. řádku umístěn vícepodmínkový příkaz *if*, do kterého je nutné přidat podmínku pro zobrazování panelu *LensPanel*.

V rámečku je zeleně vyznačen náznak, jak by mohla vypadat nově přidaná podmínka.

```
InfoPanel.java

if (view.getLastActive() instanceof MirrorController) {
elementPanel = new MirrorPanel(view);
}else if (view.getLastActive()instanceof HolographicPlateController){
elementPanel = new HolographicPlatePanel(view);
}else if (view.getLastActive() instanceof LightController) {
elementPanel = new LightPanel(view);
}else if (view.getLastActive() instanceof WallController) {
elementPanel = new WallPanel(view);
}else if (view.getLastActive() instanceof LensController) {
elementPanel = new LensPanel(view);
}
```

8.7 Výchozí barva elementu

Dalším důležitým aspektem při vytváření nového elementu (optického prvku) je nastavení výchozí barvy pro vykreslování.

Ve třídě *View* je na začátku uveden seznam jednotlivých výchozích barev pro různé elementy (optické prvky). Ideální způsob je zde vytvořit specifickou barvu pro nově vzniklý element (čočku) tak, aby byly při vykreslování jednotlivé elementy (optické prvky) snadno rozpoznatelné.

Následně je zapotřebí přiřadit barvu k vykreslovanému elementu (čočce). Na řádce 401 třídy *View* je uveden rozhodovací strom, ve kterém je nutné přidat další rozhodovací podmínku.

V tabulce pod textem je zobrazena celá rozhodovací podmínka, ve které je zelenou barvou vyznačena nově přidaná část kódu.

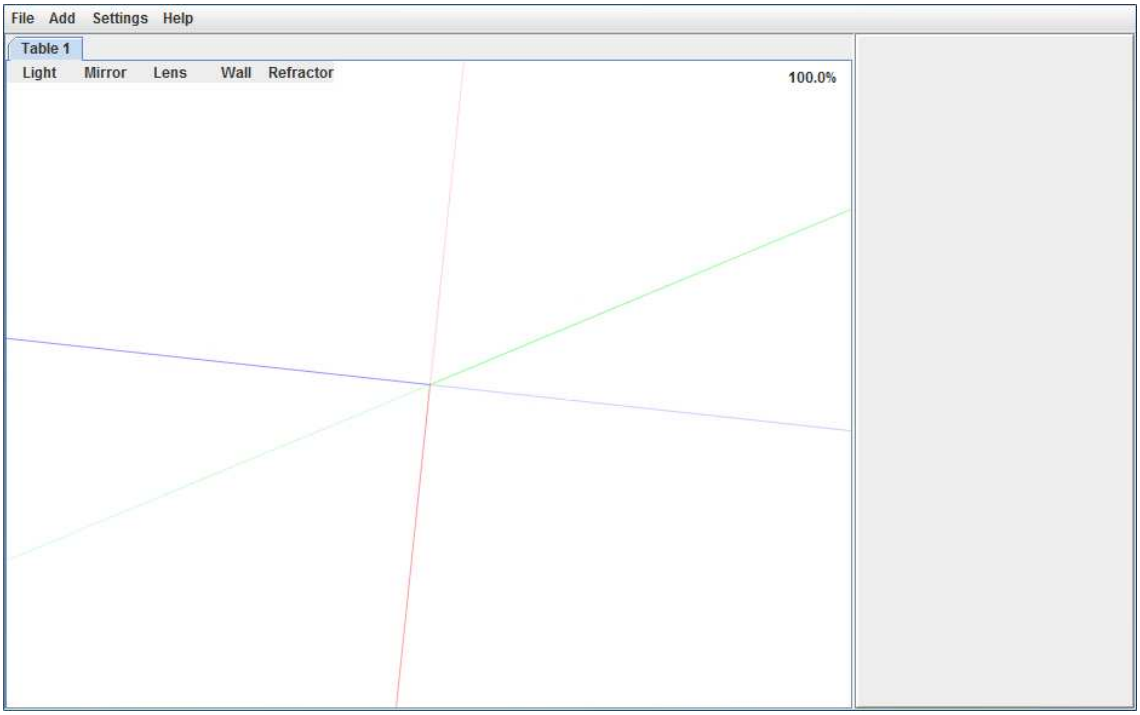
```
View.java

if (face.getElement() instanceof LightController) {
drawer.setColor(LIGHT);
} else if (face.getElement() instanceof MirrorController) {
drawer.setColor(MIRROR);
} else if (face.getElement() instanceof WallController) {
drawer.setColor(WALL);
}else if (face.getElement() instanceof LensController) {
drawer.setColor(LENS);
}else {
drawer.setColor(Color.BLACK);
}
```

9 Ověření funkčnosti

Tato kapitola se zaměřuje na otestování výpočtů a vizualizace jednotlivých simulací. Grafické uživatelské rozhraní prozatím není ještě plně funkční, jelikož se na něm neustále pracuje.Simulátor je tedy v některých případech špatně ovladatelný. Ovšem tato skutečnost by neměla být zas až tak velikou překážkou pro toto testování.

Abychom měli představu, jak vypadá nová verze simulátoru, je na obrázku 9.1 zobrazen simulátor těsně po spuštění. Rozložení jednotlivých komponent zůstalo prakticky zachováno jako v předchozí verzi. Při horním okraji je umístěna lišta, ve které je umístěno základní menu. Pod touto lištou se nachází panel, ve kterém jsou vykreslovány jednotlivé optické simulace. Při pravém okraji je umístěn panel, v němž se zobrazují a nastavují hodnoty jednotlivých optických prvků.



Obrázek 9.1 základní okno simulátoru

Poznámka: V panelu, kde se zobrazují simulace, jsou aktuálně vidět tři čáry (modrá, červená, zelená). Tyto čáry značí jednotlivé osy souřadnicového systému. Modrá osu X, červená osu Y a zelená osu Z.

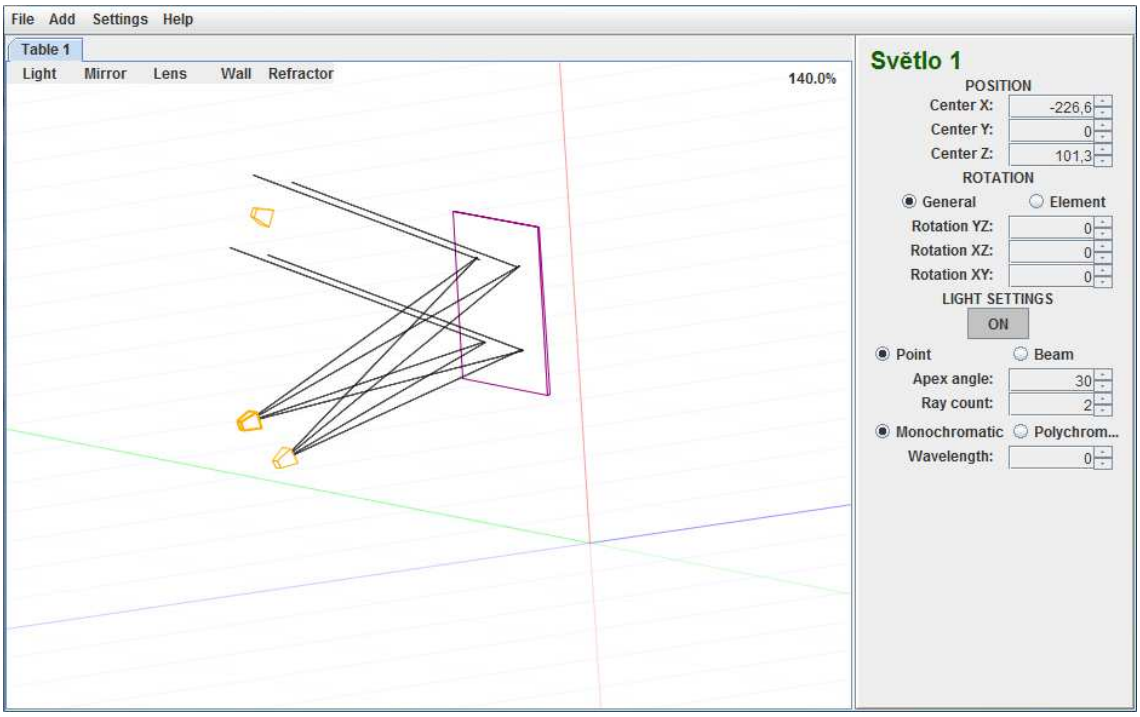
9.1 Simulace hologramu H1

Prvním testem bude vytvoření simulace pro záznam a rekonstrukci hologramu H1. Při testovánívytvořím podobný scénář,který byl použit v kapitole č.2, kde byla demonstrována funkčnost první verze simulátoru. Z důvodu přehlednosti však pro simulaci objektového svazku použiji pouze dva světelné zdroje.

9.1.1 Vytvoření záznamu

Prvním krokem je rozmístění a natočení jednotlivých optických prvků po scéně. Přidáme tedy tři světla a světlocitlivýmateriál (dále jen holografickou desku).

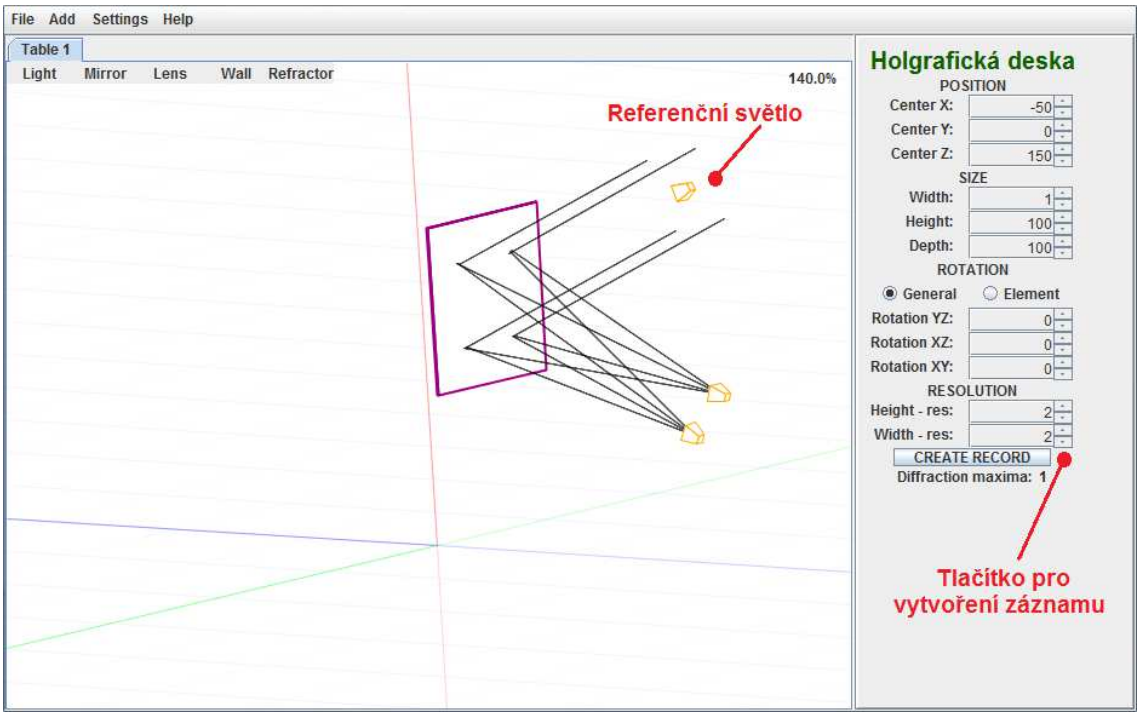
Na obrázku 9.2 je zobrazené rozmístění optických prvků. Jelikož jsou jednotlivé optické členy natočeny tak, aby se vzájemně osvětlovaly, může simulátor dopočítat a zobrazit jednotlivé paprsky, které putují mezi optickými členy.



Obrázek 9.2 Rozmístění komponent

V dalším kroku již můžeme přistoupit k samotnému vytvoření záznamu. V postraním panelu u nastavení holografické desky je umístěno tlačítko „Create Record“, pomocí něhož se vytvoří a uloží záznam interferenčníhovzoru. Po stisku tlačítka se zobrazí tabulka, v níž je uživatel vyzván k výběru referenčního světla. Po výběru referenčního světla je vytvořen a uložen celý záznam, čímž je simulace vytvoření hologramu hotova.

Na obrázku 9.3 je celá simulace zobrazena z jiného úhlu pohledu. Dále je na obrázku vidět panel s nastavením pro holografickou desku, na kterém se nachází tlačítko pro vytvoření záznamu, o němž byla zmínka výše.

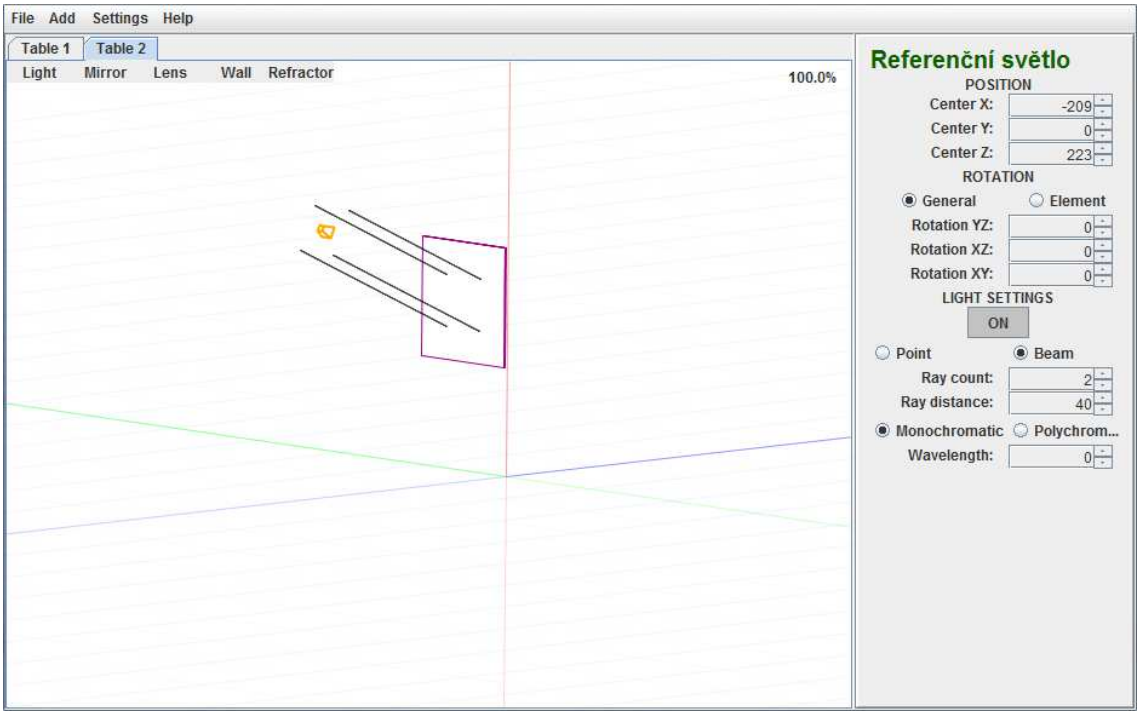


Obrázek 9.3 Vytvoření záznamu

9.1.2 Rekonstrukce záznamu

Pro rekonstrukci holografického záznamu si vytvoříme novou prázdnou scénu, do které zkopírujeme holografickou desku se záznamem a referenční zdrojové světlo z první simulace.

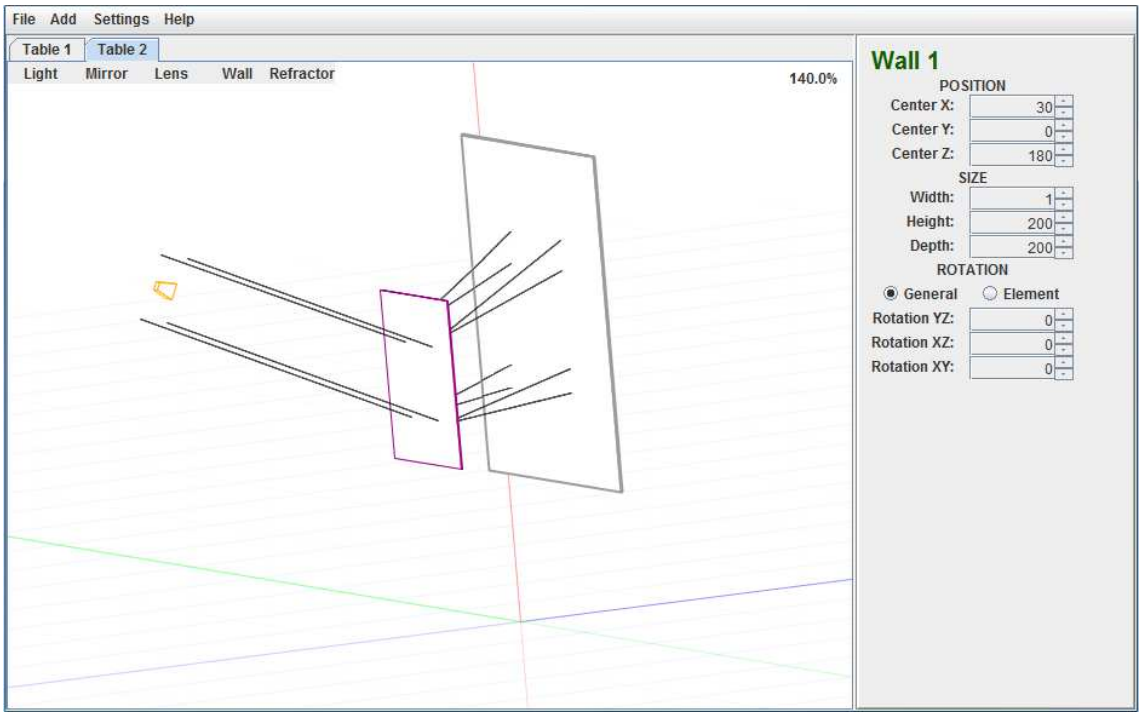
Na obrázku 9.4 je zobrazena nová simulační scéna s nakopírovanými optickými prvky.



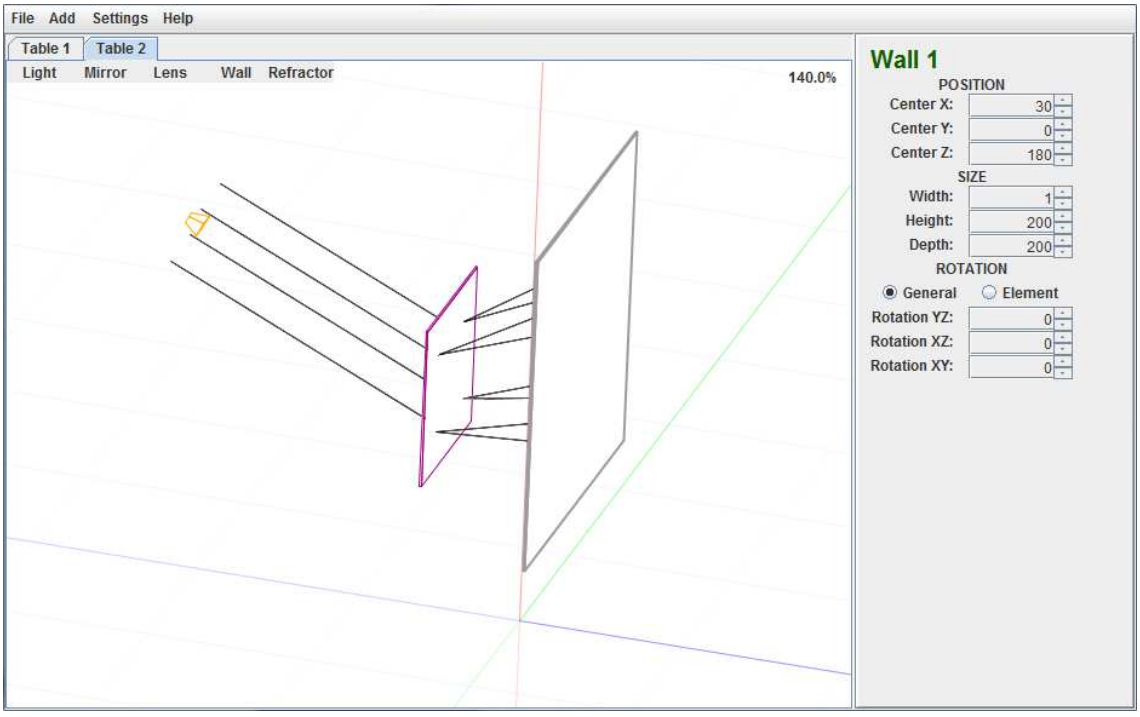
Obrázek 9.4 Zkopírování optických prvků

Tím, že zdrojové světlo již osvětluje holografickou desku, jsou již nyní generovány rekonstrukční optické paprsky. Jelikož ale simulátor nevykresluje optické paprsky, které neprotínají žádný jiný optický prvek, nejsou tyto paprsky vidět. Proto je nutné do scény ještě vložit stínítko, na které se nám promítnou výsledné paprsky.

Na obrázku 9.5 je zobrazena celá rekonstruovaná scéna i se stínítkem, na které jsou promítnuty jednotlivé výstupní paprsky z holografické desky. Pro snadnější představu celé simulace je na obrázku 9.6 zobrazena ta samá rekonstruovaná scéna, ale z jiného úhlu.



Obrázek 9.5 Výsledná rekonstruovaná scéna



Obrázek 9.6 Výsledná rekonstruovaná scéna z jiného úhlu

10 Závěr

Cílem projektu bylo vytvořit jednoduchý a přehledný simulátor vizualizující 3D prostor, ve kterém by bylo možné simulovat chování optických paprsků při různých optických pokusech, které souvisejí především s holografií. Záměrem práce však nebylo vytvořit dokonalý simulátor vlnové optiky, ale pouze simulátor, pomocí kterého lze vizualizovat vlastnosti paprsků při průchodu jednotlivými optickými prvky.

V začátcích celého projektujsem prostudoval různé podobné optické simulátory, které jsou na trhu dostupné. Při jejich zkoumání jsem dospěl k závěru, že žádný zkoumaný simulátor nenabízí vlastnosti, které bychom potřebovali. Z toho důvodu začala práce na novém simulátoru, který by lépe vyhovoval našim potřebám.

Dalším krokem byl návrh architektury interaktivního simulátoru, který umožňuje simulaci paprskové optiky. Zde bylo stěžejním bodem vytvoření metody pro sledování jednotlivých paprsků. Přes implementaci jádra simulátoru jsem se dostal až k návrhu grafického uživatelského rozhraní, na kterém jsem spolupracoval s kolegyní Lucií Herejtovou.

Výsledkem celé práce bylo ověření funkčnosti nově vytvořeného simulátoru na realistických optických soustavách. Během ověřování jsem dospěl k závěru, že vše funguje dle našich představ.

Co se týče projektu jako takového, i nadále se na něm intenzivně pracuje. Přidělávají se nové vlastnosti, které rozšíří stávající uživatelnost. Aktuálně se zaobírámekonkrétně problematikou Java Appletů. Především přemýšlíme, zda neustoupit od myšlenkypublikacesimulátoru jako Java Applet a nepřistoupit k publikaci pomocí spustitelných jar souborů.

Seznam použité literatury

[1] RYJÁČEK, Zdeněk;*Teorie grafů, diskrétní optimalizace a výpočetní složitost 1*, Skripta ZČU Plzeň, 2007

[2] BENTON, Stephen; BOVE, V. Michael.*Holographic imaging*. Hoboken, N.J.: Wiley-Interscience,c2008, xxiii, 261 p. ISBN 04-700-6806-X

[3] MILER, Miroslav;*Holografie: teoretické a experimentální základy a její použití*/1. vyd. Praha: 1974

[4] RENDL, Kamil;*Vizualizace principu hologramu*. Plzeň, 2012. Bakalářská práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd. Vedoucí ing. Petr Lobaz.

[5] HADÁČEK, MICHAEL;*Vizualizace principu hologramu*. Plzeň, 2012. Bakalářská práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd. Vedoucí ing. Petr Lobaz.

Příloha 1

Struktura souboru se simulací.

```
OPTICAL_SIMULATOR=
        BOX=
                NAME=Table 1
                LIST_CONTROLLERS=
                        LIGHT_CONTROLLER=
                                LIGHT=
                                        GLOBAL_SETTINGS=
                                                NAME=Referenční světlo
                                                ID=4
                                                POSITION=-209.0;0.0;223.0
                                                MATRIX=3;3
                                                        0.9063077870;0.0;0.4226182617
                                                        0.0;1.0;0.0
                                                        -0.4226182611;0.0;0.906307787
                                                DIMENSION=10.0;10.0;10.0
                                                IS_LIGHT_SOURCE=true
                                        LOCAL_SETTINGS=
                                                LIGHT_ON_OFF=true
                                                PLANE_OR_POINT=true
                                                LIGHT_ACTUAL_FORM=0
                                                LIGHT_ANGLES_LIGHT=30.0
                                                LIGHT_BEAMWIDTH=45.0
                                                LIGHT_COUNT_RAY=2
                                                LIGHT_MAX_JUMPS=5
                        LIGHT_CONTROLLER=
                                LIGHT=
                                        GLOBAL_SETTINGS=
                                                NAME=Světlo 1
                                                ID=2
                                                POSITION=-226.6;0.0;101.3
                                                MATRIX=3;3
                                                        0.9659258262;0.0;-0.25881904510
                                                        0.0;1.0;0.0
                                                        0.25881904510;0.0;0.9659258262
                                                DIMENSION=10.0;10.0;10.0
                                                IS_LIGHT_SOURCE=true
                                        LOCAL_SETTINGS=
                                                LIGHT_ON_OFF=true
                                                PLANE_OR_POINT=false
                                                LIGHT_ACTUAL_FORM=0
                                                LIGHT_ANGLES_LIGHT=30.0
                                                LIGHT_BEAMWIDTH=40.0
                                                LIGHT_COUNT_RAY=2
                                                LIGHT_MAX_JUMPS=5
```

LIGHT_CONTROLLER=
LIGHT=
GLOBAL_SETTINGS=
NAME=Světlo 3
ID=3
POSITION=-204.0;0.0;76.0
MATRIX=3;3
0.9063077870;0.0;-0.42261826174
0.0;1.0;0.0
0.42261826174;0.0;0.9063077870
DIMENSION=10.0;10.0;10.0
IS_LIGHT_SOURCE=true
LOCAL_SETTINGS=
LIGHT_ON_OFF=true
PLANE_OR_POINT=false
LIGHT_ACTUAL_FORM=0
LIGHT_ANGLES_LIGHT=30.0
LIGHT_BEAMWIDTH=40.0
LIGHT_COUNT_RAY=2
LIGHT_MAX_JUMPS=5
HOLOGRAPHIC_PLATE_CONTROLLER=
HOLOGRAPHIC_PLATE=
GLOBAL_SETTINGS=
NAME=Holgrafická deska
ID=1
POSITION=-50.0;0.0;150.0
MATRIX=3;3
1.0;0.0;0.0
0.0;1.0;0.0
0.0;0.0;1.0
DIMENSION=1.0;100.0;100.0
IS_LIGHT_SOURCE=false
LOCAL_SETTINGS=
HOLOGRAPHIC_PLATE_RESOLUTION_Y=2
HOLOGRAPHIC_PLATE_RESOLUTION_Z=2
HOLOGRAPHIC_PLATE_IS_RECORD=true
HOLOGRAPHIC_PLATE_LIST_DIFFRACTION=1
HOLOGRAPHIC_PLATE_RECORD=2;2;2
-1.62123668948E-4;8.48408286995E-4
-1.561292035298E-4;6.48096224067E-4
1.62123668948E-4;8.48408286995E-4
1.561292035298E-4;6.48096224067E-4
-1.832363512823E-4;0.001131577599988
-1.675284149295E-4;9.49892762225E-4
1.832363512823E-4;0.001131577599988
1.675284149295E-4;9.4989276222E-4
OPTICAL_SIMULATOR_END=