VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

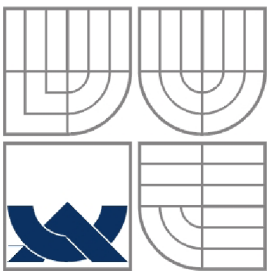
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ZOBRAZOVÁNÍ KOMPLEXNÍCH 3D SCÉN

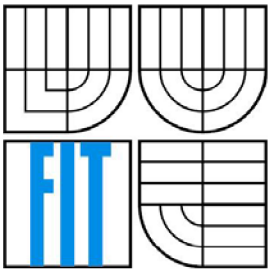
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. TOMÁŠ MRKVIČKA
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ZOBRAZOVÁNÍ KOMPLEXNÍCH 3D SCÉN

RENDERING COMPLEX 3D SCENES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ MRKVIČKA

VEDOUCÍ PRÁCE
SUPERVISOR

ING. ADAM HEROUT, Ph.D.

BRNO 2008

Zadání diplomové práce/2477/2007/xmrkvi03

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2007/2008

Zadání diplomové práce

Řešitel: **Mrkvička Tomáš, Bc.**
Obor: Počítačová grafika a multimédia
Téma: **Zobrazování komplexních 3D scén**
Kategorie: Počítačová grafika

Pokyny:

1. Seznamte se se současnými trendy v zobrazování 3D scén a popište je.
2. Prostudujte a popište problematiku datově řízeného zobrazování 3D scén.
3. Navrhněte způsob popisu součástí scény vhodný pro tento způsob zobrazovaní.
4. Navrhněte systém zobrazování respektující popis scény z předchozího bodu.
5. Implementujte datově řízené zobrazování 3D scény založené na návrzích z předchozích bodů.
6. Rozšiřte implementaci zobrazování tak, aby bylo možno získávat statistické informace o postupu zobrazování.
7. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; vytvořte plakátek pro prezentování projektu.

Literatura:

- dle pokynů vedoucího

Při obhajobě semestrální části diplomového projektu je požadováno:

- body 1.-4., dílčí experimenty vedoucí k řešení bodu 5.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVR-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Herout Adam, Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 24. září 2007
Datum odevzdání: 19. května 2008

L.S.

doc. Dr. Ing. Pavel Zemčík
vedoucí ústavu

Licenční smlouva

Abstrakt

Tato práce se zabývá problémem zobrazování rozsáhlých a obsahově velmi bohatých 3D scén, které jsou běžné např. pro moderní počítačové hry. Cílem práce je vytvoření tzv. datově řízeného zobrazovacího systému, který na základě popisu scény bude schopen sám scénu správně zobrazovat. Popis scény přitom musí být velmi jednoduchý tak, aby jej mohli vytvářet i lidé bez hlubších znalostí programování. První část této práce se zaměřuje především na navržení způsobu popisu scény a jeho následného využití při zobrazování scény. Druhá část práce se následně zabývá již vlastním využitím navrženého popisu scény při implementaci zobrazovacího systému.

Klíčová slova

Scéna, 3D grafika, virtuální realita, rozsáhlé scény, datově řízené zobrazování, konfigurace scény, popis modelu, geometrie, řešení viditelnosti, optimalizace zobrazování, popis scény, management scény.

Abstract

This thesis deals with representation of large and complex 3D scenes which are usually used by modern computer games. Main aim is design and implementation of data driven rendering system. Proper rendering is directed (driven) by scene description. This description is also designed with respect to scene creators whose typically do not have deep knowledge of programming languages in contrast to game programming developers. First part is focused on design of efficient scene description and its possible applications at scene rendering. Second part is focused on proper system implementation. Finally, consequently important system optimizations are mentioned too.

Keywords

Scene, 3D graphic, virtual reality, large scenes, data-driven rendering, rendering configuration, model description, geometry, visibility determination, rendering optimization, scene description, scene management.

Citace

Tomáš Mrkvička: Zobrazování komplexních 3D scén, diplomová práce, Brno, FIT VUT v Brně, 2008

Zobrazování komplexních 3D scén

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Adama Herouta, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Tomáš Mrkvička
8.5. 2008

Poděkování

Rád bych tímto poděkoval svému vedoucímu panu Ing. Adamu Heroutovi, Ph.D., za hodnotné odborné rady a přátelský přístup při řešení této diplomové práce. Také děkuji Ing. Jiřímu Krajíčkovi za pomoc při ověřování kvality práce a Jaroslavu Čermákovi za poskytnutí ukázkové scény. Dále bych chtěl poděkovat všem autorům a diskutujícím především ze serverů Gamasutra.com, GameDev.net a FlipCode, kde jsem získal spoustu cenných znalostí a nalezl spoustu zajímavých postřehů z oblasti práce se zobrazováním 3D scén. Závěrečný dík pak chci adresovat společnosti Bethesda Softworks, jejíž rozsáhlé a komplexní světy herní série The Elder Scrolls mi byly a stále jsou velkou inspirací při tvorbě této práce.

Obsah

1 Úvod

Tato práce se zabývá zobrazováním velmi rozsáhlých a obsahově komplexních 3D scén, které jsou i přes velmi rychle se rozvíjející schopnosti počítačového hardwaru stále velkým problémem 3D grafiky zobrazované v reálném čase. Takové scény se běžně vyskytují v aplikacích pro virtuální realitu jakými jsou dnes typicky počítačové hry nebo např. vojenské simulátory.

Obecným problémem takových scén je zejména velké množství objektů, vztahů mezi nimi a především grafických efektů, které jsou na objekty scény aplikovány. Dalším problémem práce s takto rozsáhlými scénami je postup tvorby scény, kdy obsah scény je tvořen lidmi (tvůrci modelů, textur, ...), jejichž znalosti programovacích jazyků a principů používaných v 3D grafice jsou často na dosti nízké úrovni. Z toho důvodu je nutné poskytnout k tvorbě scény takové prostředky, které zároveň umožní snadno scénu vytvářet a zároveň bude možné definovat scénu velmi komplexně. Přitom musí být ovšem stále zachován předpoklad, že tvůrci obsahu budou provádět pouze činnost nezbytně nutnou a všechna ostatní činnost bude maximálně automatizovaná.

Po stránce implementace zobrazování je velmi problematickou otázkou optimalizace zobrazení takto rozsáhlých scén. Vždy je nutné zobrazovat pouze skutečně viditelné části scény, přičemž je zároveň nutné udržovat zobrazení maximálně plynulé. Vzhledem k vždy omezeným dostupným hardwarovým prostředkům (procesorovému času, výkonu grafické karty, atd.), které mohou být (a často jsou) sdíleny dalšími částmi aplikace (např. umělá inteligence, výpočty fyzikálního modelu, atd.), je nutné hledat vhodný kompromis mezi kvalitou výsledného zobrazení a jeho rychlostí. Proto je nutné, aby navržený zobrazovací systém umožňoval např. snadno definovat parametry zobrazení scény v závislosti na dostupné hardwarové konfiguraci.

Velkou část zobrazení scény tvoří systém, který se stará o samotné zobrazení jednotlivých objektů ve scéně, tj. o aplikaci požadovaných grafických efektů na jednotlivých prvky scény. Vzhledem k tomu, že se neustále zvyšuje důraz na maximální realismus výsledné scény, zvyšují se i požadavky na použité zobrazovací techniky. Scéna dnes typicky obsahuje spousty světelných zdrojů, odrazových ploch a jiných objektů. Systém proto musí být sám schopen rozpoznat, že např. zadaný objekt leží v dosahu světla, nebo naopak že jiné objekty leží ve stínu vrhaném tímto objektem. Další ukázkovým (a problematickým) efektem jsou např. modely, jejichž povrch odráží obraz svého okolí. Zobrazení objektů tedy zahrnuje spoustu operací, které musí zobrazovací systém provést a k nimž potřebuje dostatek informací o scéně. Z těchto příkladů je tedy patrné, že objekty ve scéně mohou obsahovat spoustu přímých nebo nepřímých vazeb, které jsou buď budovány automaticky nebo definovány tvůrci scény. V každém případě je základním problémem při návrhu a tvorbě zobrazovacího systému minimalizovat množství těchto vazeb tak, aby tvůrce obsahu scény nemusel provádět žádná zbytečná manuální nastavení.

Práce se primárně zabývá datově řízeným zobrazováním velmi komplexních scén. Logicky se dělí na tři hlavní části. V té první (kapitoly 2 až 3) je nejprve proveden stručný rozbor možných přístupů k zobrazování 3D scén. Dále následuje rozbor technik datově řízeného zobrazování a návrh popisu scény vhodného k takovému způsobu zobrazování. Druhá část (kapitoly 4 až 7) se již zabývá samotnou implementací vlastního zobrazovacího systému, která je založena na poznatcích získaných v první části práce. Poslední část (kapitoly 8 až 10) se pak věnuje shrnutí vlastností aktuální implementace zobrazovacího systému a možnostem dalšího pokračování. Celá práce navazuje na poznatky získané během vypracovávání předcházejícího semestrálního projektu z něhož byly s drobnými korekcemi převzaty kapitoly 2 a 3.

2 Možnosti zobrazování 3D scén

Zobrazování a popis 3D scén lze logicky rozdělit na několik přístupů, jejichž vhodnost použití je závislá především na velikosti a obsahové složitosti scény. Z této složitosti také nepřímo vychází jak velký tým lidí bude takovou scénu vytvářet, což je druhý důležitý faktor, který ovlivňuje dělení typů systémů pro zobrazování 3D scén. Jedno z možných logických dělení zobrazování 3D scén je následující:

- naivní přístup
- systém založený na bázové abstraktních třídě
- datově řízené zobrazování

Jednotlivé typy budou rozebrány v následujících kapitolách.

2.1 Naivní přístup

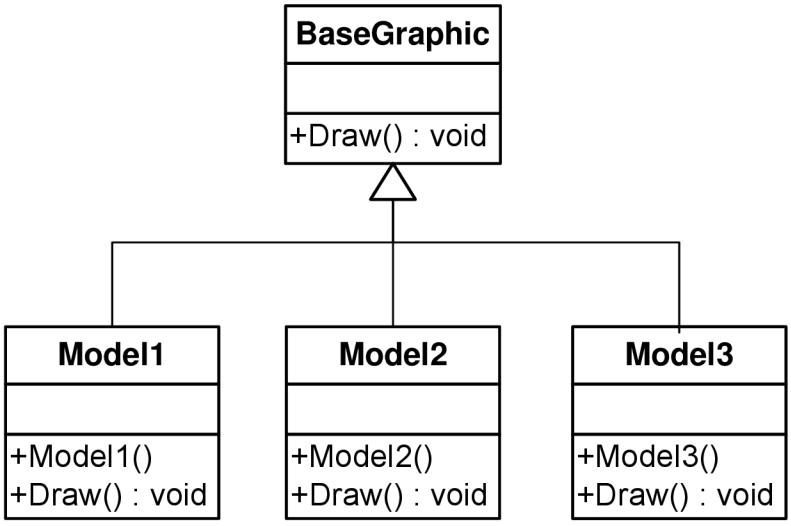
Tento přístup k zobrazování 3D scény se vyznačuje tím, že v něm neexistuje téměř žádná organizace zobrazovaných objektů ve scéně. V praxi vypadá tak, že se dostupné 3D zobrazovací rozhraní využívá přímo pro kreslení jednotlivých objektů scény právě tak jak je potřeba. Z hlediska struktury aplikace to znamená, že celý proces zobrazování je definován kódem programu uvnitř hlavní kreslící funkce (případně dalších funkcí volaných z této funkce). V tomto kódu je zahrnuta nejen definice zobrazovaných objektů scény, ale také např. definice světelných zdrojů, definice materiálů, reflexních ploch apod. Z toho vyplývá, že kompletní zobrazovací proces je pod přímou kontrolou pouze těch členů vývojového týmu, kteří znají příslušný programovací jazyk, dostupné 3D zobrazovací rozhraní a jsou schopni porozumět struktuře již fungujícího kódu.

Je zřejmé, že takový způsob zobrazování je zcela nevhodný pro tvorbu velmi rozsáhlých scén. Nicméně z hlediska rychlého prototypování nebo testování různých grafických algoritmů (plánovaných pro budoucí implementaci např. do složitějšího zobrazovacího systému) je tento postup zobrazování ideální, protože umožňuje velmi snadné provádění změn přímou úpravou programového kódu. Limitující je ovšem jeho použití členy vývojového týmu, kteří nemají dostatečné znalosti pro úpravu programového kódu. Právě ti často potřebují experimentovat se scénou a tedy je nutné jim poskytnout mocnější nástroj pro definici scény.

2.2 Systém založený na bázové abstraktní třídě

Zobrazovací systémy založené na bázové abstraktní třídě jsou již podle názvu systémy vyžadující pro návrh a implementaci objektově orientovaný přístup. Celá myšlenka spočívá v definici bázové abstraktní třídy, která definuje rozhraní pro všechny objekty scény, které budou zobrazovány. Od této

třídy jsou následně odvozovány jednotlivé objekty scény, které definují příslušné rozhraní a implementují v něm vlastní způsob zobrazování. Jednoduchý příklad takového systému je znázorněn na následujícím obrázku 2.1.



Obr. 2.1 - znázornění dědičnosti tříd v zobrazovacím systému založeném abstraktní bázové třídě

V tomto systému je pro všechny zobrazitelné objekty scény definována bázová třída `BaseGraphic` s abstraktní metodou `void Draw()`. Všechny objekty jsou odvozeny od této třídy, implementují povinnou metodu `void Draw()` a poté mohou být zobrazeny následujícím postupem:

1. Získej seznam všech objektů odvozených od bázové třídy `BaseGraphic`.
2. Vykresluj postupně objekty voláním polymorfní funkce `BaseGraphic::Draw`.

Díky polymorfnímu volání metody `BaseGraphic::Draw()` jsou jednotlivé objekty správně vykresleny podle příslušné implementace této metody. Z tohoto pohledu je tedy takový systém již lépe použitelný pro vykreslování scény, neboť existuje alespoň základní organizace objektů ve scéně.

Bohužel stejně jako v případě naivně navrženého systému, i zde platí, že systém je použitelný především pro lidi se znalostí příslušného programovacího jazyka a používaného 3D zobrazovacího rozhraní. V okamžiku přidání nového typu objektu je totiž nutné odvodit novou třídu od bázové třídy a implementovat chování její kreslící funkce. Teprve pokud jsou již všechny potřebné typy zobrazovacích tříd k dispozici, je možné systém poměrně dobře používat. Tvůrci scény v takovém případě mohou používat např. podpůrné nástroje, ve kterých vkládají objekty do scény pomocí výběru z nabízené množiny typů dostupných tříd.

Tento přístup však stále neposkytuje velké možnosti při ovládání tvorby obsahu scény. Primárně se totiž zabývá pouze viditelnými objekty scény (tedy tím co se skutečně zobrazuje). V úvahu ovšem nejsou brány další parametry scény, např. umístění světelných zdrojů. Konkrétně na našem příkladu byla definována abstraktní metoda `BaseGraphic::Draw()` sloužící pro vykreslení. Jak je vidět, tato metoda nepřijímá žádné informace o pozici světelných zdrojů ani jiných parametrech scény. Pokud bychom tedy chtěli mít možnost zobrazení daného objektu např.

s osvětlením, museli bychom do bázové třídy přidat další abstraktní metodu, např. `void DrawLight(LightInfo & light)`, která by definovala zobrazení objektu v přítomnosti zdroje světla s parametry `light`. Stejně tak bychom pak museli přidávat metody pro další specifické podmínky scény (např. pro objekty s odrazy okolních objektů apod.). Takové přidávání tříd by postupně mohlo vést k příliš komplikovanému kódu programu, který by se tak mohl stát snadno neudržovatelným.

Navíc má tento přístup další velký nedostatek. Z hlediska tvorby scény je často nutné měnit parametry zobrazení jednotlivých zobrazených objektů. Například v případě, kdy zobrazujeme objekt, který zároveň odráží své okolí na svém povrchu (např. kovově lesklá karosérie auta), bychom rádi definovali kvalitu těchto odrazů a to např. pro každý model zvlášť. Tak by zároveň bylo možné řídit náročnost zobrazení scény velmi jednoduchým způsobem. To však tento způsob zobrazování ve své základní podobě neumožňuje.

Narozdíl od naivního přístupu k zobrazování scény se tedy systém založený na bázové abstraktní třídě již dá použít i pro komplikovanější scény, nicméně stále není vhodný pro příliš rozsáhlé a obsahově bohaté scény. Z hlediska používání je stále málo vhodný pro tvůrce scény, kteří mají sice již více možností použití, ale stále mohou ovlivňovat pouze malou část vlastností scény.

2.3 Datově řízené zobrazování

Předchozí popsané přístupy k zobrazování scény se vyznačovaly velkým důrazem na potřebnou znalost konkrétního programovacího jazyka (případně 3D zobrazovacího rozhraní) od tvůrců obsahu scény. Při tvorbě skutečně rozsáhlých scén se však procesu tvorby scény téměř bez výjimky účastní pouze lidé, jejichž znalosti programovacích jazyků jsou často nulové (nebo jen velmi slabé), zatímco jejich schopnost pro výtvarnou činnost a estetiku jsou naopak na dosti značné úrovni. Z těchto důvodů je tedy nutné poskytnout pro takto nadané lidi takové prostředky k popisu scény, které jim umožní tvořit scénu s maximálním využitím svých schopností a zároveň nebudou vyžadovat hluboké znalosti principu 3D grafiky z hlediska algoritmů apod. Přístup, který definuje takový způsob tvorby a zobrazení scény, se obecně označuje jako datově řízené zobrazování scény.

Základní myšlenkou datově řízeného zobrazování je umožnit tvůrcům scény definovat obsah scény pomocí dostupných prostředků a tento vytvořený obsah potom použít jako vstupní data programové části zobrazovacího systému, která sama zajistí správné zobrazování celé scény.

Z praktického pohledu může být zobrazovací systém založený na datovém řízení velmi komplexní z hlediska práce programátora a celé aplikační logiky, kterou musí implementovat. V porovnání s předchozími popsanými systémy zobrazování může být složitost i mnohonásobně vyšší. Velkou výhodou takového systému však zůstává snadnější možnost tvorby obsahu scény, což je dnes primární požadavek, neboť výsledný dojem ze scény nakonec určuje celkový dojem z aplikace a tedy i její úspěch či neúspěch. Výhodou tohoto přístupu je také snazší modifikovatelnost

a rozšiřitelnost, neboť aplikační logika se může uvnitř systému neustále měnit a přesto může aplikace stále fungovat identicky na původních datech (aplikační logika se tedy z hlediska tvůrce systému chová jako černá skříňka poskytující určité rozhraní v podobě definice formátu vstupních dat).

Zobrazovací systémy založené na datovém řízení tedy přináší velkou výhodu při tvorbě scény a proto jsou ideálním prostředkem pro tvorbu aplikací s rozsáhlými a obsahově bohatými scénami. V následujícím textu se tedy budeme zabývat právě těmito systémy.

2.3.1 Problematika systémů pro datově řízené zobrazování

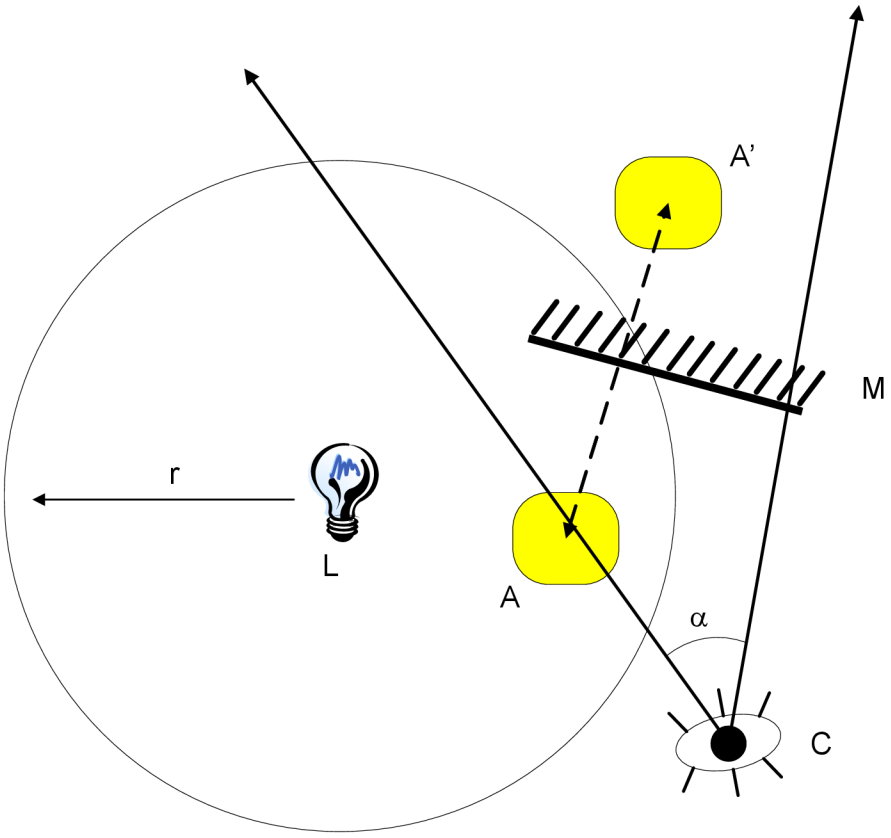
Jak již bylo zmíněno výše, je primárním cílem zobrazovacího systému založeném na datovém řízení poskytnout tvůrcům obsahu scény vhodné prostředky pro dostatečně kvalitní popis scény, který zároveň nevyžaduje vyšší znalosti z oblasti programování. Tento popis však musí být zároveň vhodný pro řízení procesu zobrazování, což může být často v rozporu s prvním požadavkem na jednoduchost popisu scény. Při vytváření návrhu celého systému je také nutné myslet na implementační detaily tak, aby byl celý systém vůbec realizovatelný.

Obecným problémem datově řízeného zobrazovacího systému bývá především určení všech jeho možností. Bohužel není možné nikdy navrhnout takový systém, který bude schopen zobrazovat libovolnou scénu s libovolnými grafickými efekty. Možnosti celého systému totiž vychází vždy z povoleného popisu scény, který je vždy konečný a tedy omezený. Z toho tedy vyplývá, že právě kvalita návrhu popisu scény hraje důležitou roli v datově řízených zobrazovacích systémech.

Pro ilustraci problému datově řízeného zobrazování zde uvádíme na obrázku 2.2 jednoduchý příklad scény a rozbor celé situace, který demonstruje ukázkový problém typického zobrazení scény. Na obrázku je znázorněn bodový světelný zdroj **L** s poloměrem dosahu osvětlení **r**. Ve scéně je umístěn objekt **A**, který se odráží v zrcadle **M** a vytváří se tak jeho virtuální obraz **A'**. Pozorovatel umístěný na pozici **C** sleduje scénu pod zorným úhlem **α**. Jak je vidět, v zorném poli pozorovatele se nachází nejen původní objekt **A**, ale také objekt zrcadla **M**, které zároveň zobrazuje obraz **A'** skutečného objektu **A**. Při zobrazování takové scény je nutné brát v úvahu několik základních požadavků na výsledné zobrazení. Především je nutné zajistit správné osvětlení všech viditelných objektů. To může působit problém zejména u objektu **A'**, který je ve své virtuální pozici mimo dosah světla, nicméně ve skutečnosti je osvětlen. Zobrazení tohoto objektu tedy nelze provést např. známou technikou přesunutí objektu podle osy zrcadla a následného vykreslení pomocí šablonového testu[1]. Další problém s objektem **A'** může vzniknout z hlediska kvality zobrazení. V případě rozsáhlejších scén totiž není možné zobrazovat všechny odrazy ve scéně stejně kvalitně jako primární objekty scény. Proto je vhodné zavést techniku, která umožní definovat kvalitu zobrazení objektů v zrcadlových plochách apod..

[1] Šablonovým testem se zde myslí testování pomocí paměti šablony (angl. „stencil buffer“) dostupné dnes téměř na každém grafickém hardwaru.

Na tomto příkladu je vidět, že i v tak jednoduché scéně může docházet k velmi komplikovaným situacím, které by za normálních okolností museli být řešeny poměrně složitým větvením kódu. Cílem datově řízeného zobrazování je najít takový popis scény, který umožní jednoduše definovat podobné situace a podle tohoto popisu dokáže vnitřní aplikační logika systému sama rozhodovat o procesu zobrazení.



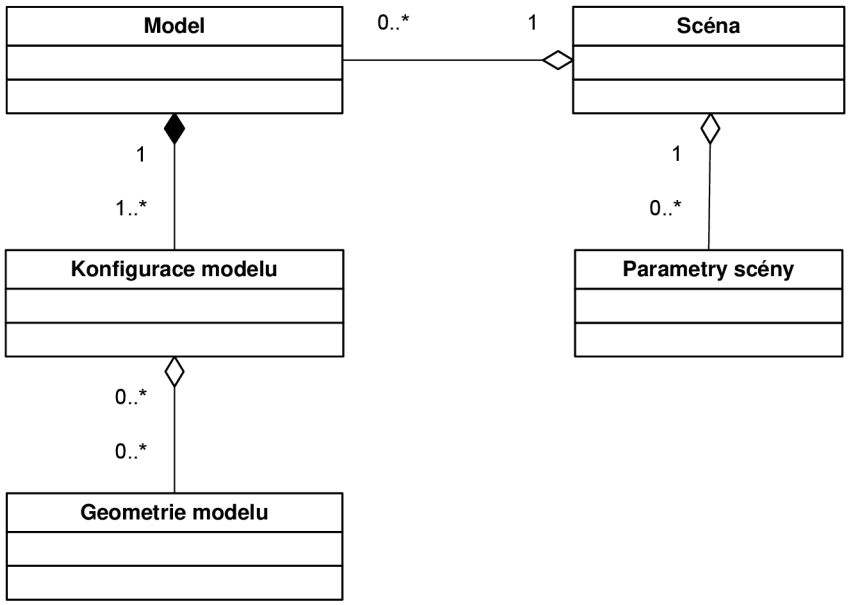
Obr. 2.2 – ukázka typické scény pro datově řízené zobrazování

Systém pro datově řízené zobrazování lze logicky rozdělit na dvě hlavní části. Tou první je samotný popis scény, jehož návrh velmi výrazně ovlivňuje kvalitu a možnosti celého systému. Druhou částí je pak samotná implementace zobrazovacího systému, která podle navrženého popisu scény provádí vlastní zobrazení. Tyto dvě části budou diskutovány v následujících kapitolách.

3 Návrh popisu scény

Popis scény hraje v případě datově řízeného zobrazovacího systému naprosto klíčovou roli, protože kvalita návrhu celého popisu definuje možnosti, které bude systém poskytovat tvůrcům obsahu scény. Dobře navržený popis scény musí především zohledňovat budoucí rozšíření svých možností. Ideální je případ, kdy jsou na počátku vytvořeny velmi jednoduché prostředky pro popis scény a ty jsou postupně modifikovány přidáváním nových možností. V této kapitole bude představen velmi jednoduchý návrh popisu scény tak, aby byl do budoucna snadno rozšiřitelný o další možnosti a aby byl zároveň použitelný pro další části zobrazovacího řetězce celého systému.

Vzhledem k tomu, že popisována scéna může obsahovat velké množství objektů, je velmi důležité navrhnout její popis jako maximálně srozumitelný a především jednoznačný, takže tvůrci obsahu scény budou scénu vytvářet pokud možno intuitivně. Z toho důvodu je výhodné pro popis scény použít textový formát, např. podobný formátu XML se zjednodušenou syntaxí nebo založený na typických konfiguračních INI souborech. Některé části scény je ale naopak vhodné popisovat binárními soubory (typicky geometrii modelu). Oba dva přístupy jsou ve zde představovaném návrhu použity a budou popsány dále. Zde navržený popis scény je přehledově znázorněn na obrázku 3.1.



Obr. 3.1 – znázornění návrhu popisu scény konceptuálním diagramem tříd

Celková scéna se tedy skládá z dvou základních typů entit - modelů a parametrů scény. Modely se dále skládají z takzvaných konfigurací modelů (bude vysvětleno dále), které mohou definovat geometrii modelu[2]. Parametry scény určují různé vlastnosti scény jakými je např. umístění světel, pomocných objektů pro určování viditelnosti apod. Všechny tyto součásti popisu scény budou popsány v následujících kapitolách. Celkový návrh bude tvořen metodikou odspodu nahoru, tedy od

[2] Tedy konfigurace nemusí vždy zahrnovat geometrii modelu.

jednotlivých součástí scény až po její kompletní popis, což přesně odpovídá modulární výstavbě scény používané v datově řízených zobrazovacích systémech.

Konkrétně zde bude rozebrán návrh souboru pro vhodný popis geometrie modelu, dále bude definován pojem „konfigurace modelu“, bude diskutován vhodný návrh popisu pro konfigurace modelu a nakonec bude navržen kompletní popis scény vhodný pro datově řízené zobrazování.

3.1 Geometrie modelu

Geometrie modelu je v rámci navrhovaného systému uvažována jako zobrazitelná[3] část každého modelu scény. Vzhledem k tomu, že se dnes pro zobrazování scén používají téměř výhradně polygonální modely (konkrétně modely složené z trojúhelníků), je i zde popisovaný způsob definice geometrie založen na sítí trojúhelníků. Konkrétně je geometrií v rámci systému myšlena množina vrcholů, které definují jednotlivé trojúhelníky daného modelu. Vzhledem k dnes používanému grafickému hardwaru jsou jednotlivé trojúhelníky z množiny vrcholů definovány pomocí pole indexů, kde vždy tři po sobě jdoucí indexy určují tři vrcholy tvořící jeden trojúhelník.

Pro uložení geometrie byl navržen vlastní (a velmi jednoduchý) binární souborový formát, který umožňuje uložit všechny potřebné informace o geometrii modelu. Binární formát byl vybrán především z důvodu snazšího (a rychlejšího[4]) načítání i ukládání geometrie a také částečně kvůli nižším nárokům na úložný prostor. Geometrie obsahuje pouze základní informace, tj. množinu vrcholů a indexů tvořících trojúhelníky, další informace už jsou součástí nadřazených datových struktur (model, konfigurace), proto není nutné používat textový formát, neboť zde uložené informace se považují za konstantní (jejich tvorbu obstarává např. externí modelovací aplikace a příslušný konvertor).

Pro jednoduchost používání geometrie bylo zavedeno pravidlo, které umožňuje uložit do jednoho souboru pouze jeden objekt geometrie modelu. Kompletní geometrie scény je tedy uložena v oddělených souborech, což je výhodné jak z hlediska budoucího použití při tvorbě scény, tak i z hlediska snadnější změny geometrie pro jednotlivé modely.

Základním problémem při návrhu binárního formátu pro uložení geometrie modelu bylo vyřešení otázky různorodosti formátu vrcholů jednotlivých modelů obsažených ve scéně[5]. Některé modely mohou obsahovat pouze vrcholy s definicí pozice, jiné ale ve vrcholech nesou i další informace jakými jsou např. texturovací souřadnice, normála, informace o barvě atd. Tento problém

[3] Prakticky však může být použita pro libovolný účel, např. k tvorbě pomocných těles pro určování viditelnosti apod.
[4] Rychlost načítání geometrie bude velmi významná především pro postupné načítání obsahu scény – viz kapitola 5.3.2.
[5] To byl také hlavní důvod zavedení nového formátu, neboť běžně dostupné formáty pro uložení geometrie jsou často velmi omezené z hlediska typů ukládaných vrcholů.

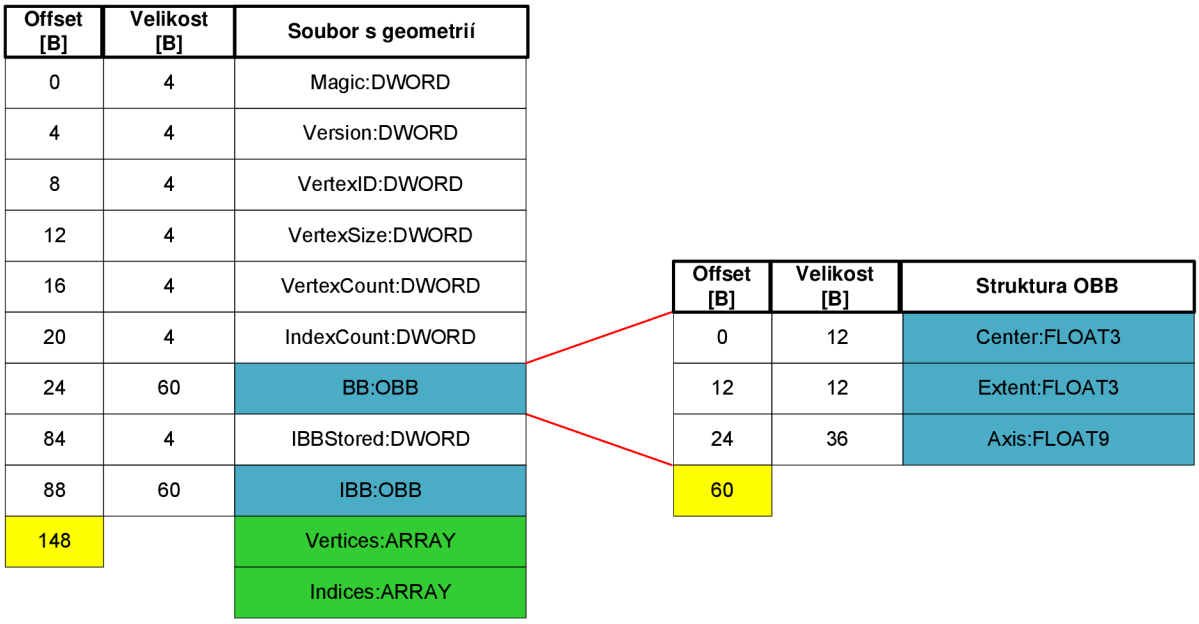
byl vyřešen definicí jednoduché hierarchie formátu vrcholů, která jako základní vrchol definuje následující datovou strukturu `TVertex3D`:

```
struct TVertex3D
{
    float x,y,z;
};
```

Tento typ vrcholu tedy obsahuje pouze informaci o pozici daného vrcholu v 3D prostoru. Jakýkoliv další typ vrcholu musí být odvozen od této datové struktury a tím je zajištěno, že bude s touto strukturou kompatibilní (tj. platí pravidlo OOP, že potomek může vždy nahradit předka). Takový přístup je výhodný především pro budoucí práci s geometrií modelu, kdy některé operace mohou probíhat na libovolné geometrii, protože vyžadují pouze informace o pozici vrcholů, která je obsažena povinně v každém vrcholu a to vždy na počátku vrcholu (tedy pozice se dá vždy získat při znalosti velikosti jednoho vrcholu).

Vzhledem k uložení vrcholů do binárního formátu bylo nutné k definici typu vrcholu přidat také informaci, která umožní jednoznačně rozeznat typ vrcholu v souboru s geometrií. Tj. po načtení souboru s geometrií modelu je nutné rozpoznat typ vrcholu, který tato geometrie používá a podle toho načíst příslušné vrcholy. Pro rozpoznání bylo zavedeno pro každý typ vrcholu jednoznačné identifikační číslo **VertexID**, které dovolí rozpoznat při načtení typ vrcholu.

Celá struktura binárního souboru pro uložení geometrie je znázorněna na následujícím obrázku 3.2.



Obr. 3.2 – formát souboru pro uložení geometrie modelu

V prvním sloupci je informace o posunutí dané položky od začátku souboru, v druhém sloupci pak velikost dané položky v bajtech. Datový typ **DWORD** reprezentuje 32-bitové číslo bez znaménka,

FLOAT3 (**FLOAT9**) je označení pole se třemi (devíti) čísly ve formátu 32-bitového čísla v plovoucí řádové čárce (dle normy IEEE 754 – viz [21]), **OBB** je datová struktura definovaná na vedlejším obrázku (dále viz popis v kapitole 3.1.1) a **ARRAY** definuje pole bajtů obecné velikosti. Význam jednotlivých atributů souboru s geometrií modelu je vysvětlen v následující tabulce 3.1:

Magic	Konstanta jednoznačně identifikující soubor s geometrií. Její hodnota je definována jako řetězec **GEOM**, tedy jako 32-bitové neznaménkové číslo **0x4d4f4547**.
Version	Definice verze souboru sloužící pro snadnější rozlišení starších verzí tohoto souborového formátu. Při změně definice souboru s geometrií je toto číslo zvýšeno. Výhodou tohoto přístupu je, že aplikace může pracovat jak s aktuálními verzemi souboru, tak s verzemi staršími. Za běhu aplikace je jednoduše podle čísla verze vybrána vhodná funkce pro načtení souboru.
VertexID	Identifikátor typu vrcholu. Musí být shodný s některým z platných identifikátorů vrcholu, které jsou aktuálně definovány v rámci celého systému. Hodnoty identifikátorů jsou určeny až pro konkrétní používaný systém.
VertexSize	Velikost jednoho vrcholu v bajtech. Tato informace je z pohledu systému redundantní, neboť může být získána pomocí atributu **VertexID**. Její použití je však výhodné v situacích, kdy libovolná jiná aplikace potřebuje pracovat se souborem a vyžaduje pouze zjistit velikost dat s vrcholy (viz dále).
VertexCount	Počet vrcholů uložených v poli **Vertices** (viz dále).
IndexCount	Počet indexů uložených v poli **Indices** (viz dále).
BB	V tomto atributu je uložen orientovaný ohraničující kvádr této geometrie (**OBB** – z anglického *Oriented Bounding Box*). Kvádr je uložen jako struktura typu OBB popsaná dále. Tento atribut obsahuje platnou strukturu OBB, která musí být vždy povinně v modelu uložena.
IBBStored	Tento atribut udává, zda jsou v atributu **IBB** uložena platná data. Hodnota 0 znamená, že atribut **IBB** obsahuje platná data. Libovolná jiná hodnota znamená, že v atributu **IBB** nejsou uložená platná data.
IBB	V tomto atributu je uložen tzv. vnitřní orientovaný ohraničující kvádr této geometrie (**IOBB** – z anglického *Inner Oriented Bounding Box*). Tento kvádr definuje těleso, které má tu vlastnost, že všechny vrcholy geometrie modelu leží mimo toto těleso a těleso je zároveň obsaženo geometricky uvnitř geometrie modelu (jedná se tedy v podstatě o těleso vepsané do geometrie modelu). Taková datová struktura je pak snadno použitelná v pozdějších fázích zobrazení při určování viditelnosti. Tento atribut obsahuje platnou strukturu OBB pouze pokud je atribut **IBBStored** nastaven na 0. Zadaná struktura OBB reprezentuje ohraničující kvádr v lokálních souřadnicích této geometrie. Více informací o struktuře OBB viz dále.
Vertices	Pole obsahující přímo data jednotlivých vrcholů. Velikost tohoto pole je určena atributy **VertexSize** a **VertexCount**. Tedy počet bajtů v tomto poli je roven **(VertexSize * VertexCount)**.
Indices	Pole obsahující přímo jednotlivé indexy, které pak určují jednotlivé trojúhelníky jako trojice vrcholů z pole **Vertices**. Velikost jednoho indexu je vždy rovna 2 bajtům. Každý index je tedy reprezentován jako 16 bitové číslo bez znaménka.

Tabulka 3.1 - význam jednotlivých položek v souboru s uloženou geometrií

Soubor s geometrií modelu tedy obsahuje všechny potřebné informace pro vykreslení modelu. Po načtení informací z hlavičky lze přečíst vrcholy i jejich indexy velmi snadno jako pole bajtů (jednotlivé položky jsou uloženy souvisle bez zarovnání). Povinně obsahuje ohraničující kvádr dané geometrie a volitelně může geometrie obsahovat i vepsaný **(IBB)** ohraničující kvádr. Ty lze

s výhodou využít především při řešení viditelnosti modelu s touto geometrií, případně např. při výběru modelu myší a podobných činnostech, kde je výhodné pracovat pouze o ohraničujícím kvádrem a ne všemi trojúhelníky geometrie modelu.

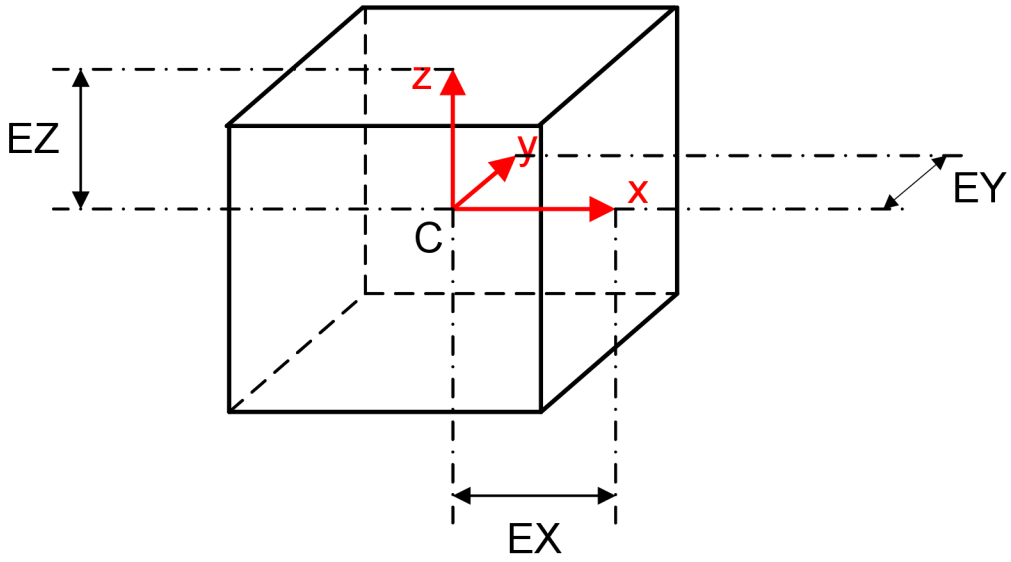
Protože zde použitá struktura OBB nemá svojí standardní definici, je vysvětlena v následujícím textu. Byla vybraná především kvůli její schopnosti podstupovat základní transformace (posunutí, rotace, změna velikosti), což znamená celkovou snazší práci s modelem bez nutnosti jakýchkoliv výpočtů nutných pro přepočet geometrie, a zároveň umožňuje stanovit pro geometrii optimální ohraničující kvádr.

Zde popsaná struktura souboru s geometrií je z hlediska návrhu považována za dostatečnou z hlediska budoucího použití a nepředpokládají se u ní žádné významnější změny. Další informace pro model lze vždy uložit do nadřazených struktur, které jsou uloženy v textové podobě (viz dále). Proto je zvolený binární formát považován za optimální.

3.1.1 Struktura OBB

Struktura OBB používaná v souboru s geometrií modelu obsahuje definici standardně používaného orientovaného ohraničujícího kvádru. Ten je pro ilustraci znázorněn na obrázku 3.3. Více informací o práci s takovou geometrickou strukturou lze získat např. v [5].

Struktura se skládá z umístění středu kvádru **Center (C)**, lokálních souřadných os kvádru **Axis (x, y, z)** a rozměrů kvádru **Extent (EX, EY, EZ)** podél jednotlivých os. Přesněji řečeno tyto rozměry udávají polovinu velikosti kvádru podél jednotlivých lokálních souřadných os.



Obr. 3.3 – znázornění struktury OBB

Přesné uložení hodnot ve struktuře OBB podle obrázku 3.2 je popsáno v následující tabulce 3.2:

Center	Umístění středu struktury **OBB** v 3D prostoru. Tento bod je definován v lokálních souřadnicích geometrie modelu. Bod je uložen standardně v pořadí **[cx,cy,cz]**.
Extent	Rozměry kvádru podle jednotlivých os lokálního souřadného systému (viz dále).

	Každý rozměr udává přesně polovinu velikosti kvádru podél příslušné osy. Rozměry jsou uloženy standardně jako **[EX,EY,EZ]**, tedy rozměr podél osy **X**, rozměr podél osy **Y** a rozměr podél osy **Z**.
Axis	V této položce jsou uloženy vektory definující osy lokálního souřadného systému. Tyto osy vychází ze středu kvádru a definují lokální souřadný systém. Vektory jsou uloženy v pořadí **[x,y,z]**, tedy nejprve je uložen vektor pro osu **X**, poté pro osu **Y** a nakonec pro osu **Z**.

Tabulka 3.2 - popis částí struktury OBB

Osy definované v atributu **Axis** tvoří lokální souřadný systém struktury **OBB**. Atribut **Axis** tvoří pole hodnot **[EXx, EXy, EXz, EYx, EYy, EYz,, EZx, EZy, EZz]**. Z těchto hodnot lze vytvořit rotační matici pro transformaci mezi světovým souřadným systémem a lokálním souřadným systémem OBB takto:

$$TransformMatrix = \begin{bmatrix} EXx & EYx & EZx \\ EXy & EYy & EZy \\ EXz & EYz & EZz \end{bmatrix}$$

Bod $POINT_{WORLD}$ ve světových souřadnicích (patřičně posunutý vzhledem ke středu **Center** struktury OBB) lze do bodu $POINT_{LOCAL}$ v lokálních souřadnic OBB pomocí této matice transformovat jako:

$$POINT_{LOCAL} = POINT_{WORLD} \times \begin{bmatrix} EXx & EYx & EZx \\ EXy & EYy & EZy \\ EXz & EYz & EZz \end{bmatrix}$$

Inverzní transformaci (tedy z lokálního souřadného systému do světových souřadnic) lze provést snadno pouhou transponací transformační matice *TransformMatrix*, protože ta je vždy ortonormální a tedy inverzní matice je shodná s maticí transponovanou.

3.2 Konfigurace modelu

Konfigurace modelu je další klíčová část popisu scény datově řízeného zobrazovacího systému. Zjednodušeně ji lze definovat jako předpis pro zobrazení geometrie modelu v určitých podmínkách scény[6]. Přitom podmínky scény jsou určeny parametry scény a zároveň okolními objekty ve scéně.

Podle výše uvedeného obrázku 3.1 je vidět, že celkový model ve zde představovaném návrhu popisu scény je tvořen množinou konfigurací, které mohou obsahovat množinu geometrií modelu. Tedy jeden model může být teoreticky tvořen více geometriemi. Tento přístup byl zvolen zcela záměrně a bude podrobně rozebrán v následujících odstavcích.

Pro ilustraci připomeneme opět situaci na výše uvedeném obrázku 2.2. Zde je zobrazena ukázková scéna, ve které je objekt **A**, který se zároveň odráží v zrcadle **M**. Tento objekt je osvětlen světelným zdrojem. Z toho tedy vyplývá, že v systému, který takovou scénu chce zobrazit, by měla

[6] Ve skutečnosti však nemusí obsahovat pouze informace pro zobrazení geometrie, ale téměř libovolné informace potřebné pro daný model, např. vhodné pro určování viditelnosti apod. To bude upřesněno až v kapitole 6.

existovat konfigurace pro zobrazení osvětleného modelu **A**, dále pak konfigurace pro zobrazení osvětleného modelu vyobrazeném v zrcadle **M** a např. konfigurace pro zobrazení objektu **A** v okamžiku, kdy světelný zdroj ze scény zmizí a objekt bude osvětlen pouze jednoduchým všudypřítomným světlem. Již dříve bylo zmíněno, že např. zobrazení modelu **A** v zrcadle nemusí být provedeno naprosto detailně (především z výkonnostních důvodů), je tedy logické, že konfigurace modelu **A** pro odraz by mohla definovat pro zobrazení poněkud méně náročnou geometrii modelu. To je právě důvodem, proč může model obsahovat definici více geometrií. Každá konfigurace zobrazení modelu totiž může vyžadovat jinou složitost geometrie v závislosti na požadované kvalitě. Dále pak může např. konfigurace modelu využívat více geometrií pro určování úrovně detailu založeném na vzdálenosti modelu od pozorovatele apod. Ve výsledku tedy každá konfigurace modelu definuje množinu geometrií podle svých potřeb.

Jak je vidět na předchozím odstavci, je pojem „konfigurace modelu“ poměrně široký, neboť konfigurace jako taková může popisovat téměř libovolný stav scény. Proto je vhodné jej nějak konkrétně definovat pro celý systém a určit pravidla, která pro konfigurace zobrazení modelů v systému platí. Tato pravidla pak přesně definují povolenou množinu konfigurací v systému, jejich přesnou podobu a význam.

Ve zde popisovaném návrhu bude pojem konfigurace modelu popisovat nejen informace o zobrazení daného modelu, ale zároveň bude chápán jako lokální stav scény, který určuje vlastnosti zobrazení jednotlivých modelů v dané lokalitě scény s určitými vlastnostmi. Takovou konfiguraci budeme nazývat konfigurací scény. Pokud model chce být správně zobrazen v takové lokalitě scény, musí obsahovat příslušnou korektní konfiguraci modelu pro specifickou konfiguraci scény. Z toho tedy vyplývá, že zobrazovací systém nejprve musí definovat všechny typy konfigurací scény a ty jsou potom definovány pro jednotlivé modely scény. Žádná jiná konfigurace, která není v systému definována, nemůže být pro model použita. To se samozřejmě může jevit jako jisté omezení zobrazovacího systému. Toto je naštěstí eliminováno možností přidávat do systému další potřebné konfigurace. Pokud např. potřebujeme přidat do systému podporu zobrazení stínů, jednoduše definujeme novou konfiguraci scény pro stíny apod. V takovém případě je ale jasné, že dříve definované modely neobsahují konfiguraci modelů pro stínů (neboť ta dříve neexistovala a tvůrce scény ji tedy pro model nemohl definovat) a tedy takové modely nemusí zobrazovat stíny korektně. Tento problém lze řešit několika způsoby. Jeden z nich situaci vyřeší například tak, že se chybějí konfigurace u modelu projeví např. chybovým hlášením do záznamu o běhu aplikace nebo zobrazením speciálního objektu ve scéně, který bude upozorňovat na chybějící konfiguraci. Další možností je např. pokusit se v rámci systému najít podobnou konfiguraci modelu, která obsahuje dostatečné množství informací a tuto podobnou konfiguraci použít jako náhradu za chybějící konfiguraci.

V následujícím textu se soustředíme na navržení vhodného popisu konfigurace modelu, přičemž bude diskutována především základní koncepce. Návrh konkrétních konfigurací určených

pro implementaci bude diskutován až v rámci implementace konkrétního zobrazovacího systému (viz kapitola 4).

3.2.1 Návrh formátu souboru pro uložení konfigurací modelu

Pro soubor s uloženou geometrií jsme navrhli vlastní binární formát souboru, protože geometrie je z pohledu tvůrce obsahu scény zajímavá pouze v konečném zobrazení a tedy není nutné mít detailní informace např. o každém uloženém vrcholu. Zároveň je tento způsob uložení efektivnější jak z hlediska použitého místa, tak z hlediska rychlosti načítání geometrie. U souboru s popisem konfigurací modelu je situace přesně opačná. Tvůrce obsahu scény často potřebuje měnit různé parametry modelů ve scéně a změna těchto parametrů znamená v datově řízeném zobrazovacím systému změnu příslušné konfigurace modelu. Proto je pro uložení konfigurací zobrazení vhodné použít strukturovaný textový formát, který bude snadno modifikovatelný a také dobře pochopitelný.

Zatímco navržený soubor pro uložení geometrie obsahoval pevně definovanou strukturu, soubor s definicí konfigurací je opět přesným opakem. Konfigurace modelu totiž obecně může obsahovat téměř libovolné položky (záleží vždy na typu konfigurace modelu). Především z toho důvodu je použití textového formátu téměř nutností. Pro popis konfigurace se zřetelně nabízí použití dnes populárního formátu XML (viz [18]). Ten však přes řadu nesporných výhod přináší několik základních komplikací. Především je obtížně zpracovatelný z hlediska implementace. Pro běžné vývojové platformy a programovací jazyky jsou typicky dostupné programové knihovny pro práci s tímto formátem, nicméně to neplatí vždy a to může přinést problém např. v případě, kdy bude systém portován na jiné platformy, kde dané knihovny nejsou k dispozici. Dalším (a pro nás podstatnějším) problémem formátu XML je pak jeho poměrně přísná syntaxe, která bývá v základní formě (tedy bez speciálního zobrazení) často považována za špatně čitelnou a náchylnou k tvorbě syntaktických chyb. To je v našem případě naprosto nepřípustné, protože vyžadujeme především snadnou srozumitelnost a modifikovatelnost s minimem pravidel tak, aby i lidé s nižší počítačovou gramotností dokázali s takovým formátem pracovat.

Z důvodů uvedených výše byl vytvořen vlastní jednoduchý textový formát pro uložení konfigurací, který vychází zejména z formátu konfiguračních souborů INI, ale zahrnuje zároveň možnost zanořování položek známou z XML. Formát povoluje následující tři konstrukce uvedené v tabulce 3.3:

Blok	Blok je reprezentován dvojicí hranatých závorek, ve kterých je uložen název bloku. Všechny znaky (včetně mezer) mezi hranatými závorkami jsou významné znaky jména bloku. Za uzavírací hranatou závorkou následuje otevírací složená závorka, která otvírá blok. V tom mohou být obsaženy další bloky nebo přímo jednotlivé atributy (viz dále). Blok je ukončen uzavírací složenou závorkou. **Příklad:**

	[test]{
	... obsah bloku
	}

Atribut	Atribut je složen ze jména a hodnoty. Jméno i hodnota musí být vzhledem ke snazšímu načítání umístěny do uvozovek a mezi jménem a atributem je povinně znaménko rovnosti. Každý atribut musí být vždy uložen na jednom řádku. **Příklad:** „jméno" = „hodnota"
Komentář	Za komentář se vždy považuje text od znaku „středník" až do konce příslušného řádku. **Příklad:** ;toto je komentář

Tabulka 3.3 - povolené prvky souboru pro popis konfigurace modelu

Výsledný soubor s konfigurací pak může vypadat např. takto:

```
[sekce1] {
      "atribut1" = "hodnota1" ; toto je komentář

      [sekce2] ; toto je prázdný vnořený blok
      {
      }
}
```

V souboru se tedy vyskytují jednotlivé bloky v nichž mohou být zanořené další bloky nebo obsahují přímo atributy. Každý atribut má své unikátní jméno v rámci bloku a povinně definuje svou hodnotu. Tato hodnota je vždy řetězec, jehož skutečný význam je interpretován vždy podle způsobu použití. Takto definovaný formát souboru umožňuje snadno vytvářet téměř libovolné konfigurace a zároveň je velmi snadno čitelný jak z hlediska tvůrce systému, tak z hlediska načítání v programu.

Výsledkem načtení souboru v tomto formátu je vždy hierarchická struktura v podobě stromu, kde listy tohoto stromu jsou atributy jednotlivých bloků. Díky povinnému použití uvozovek kolem jmen a hodnot atributu není problém definovat jednotlivé názvy s téměř libovolnými znaky. Jedinou výjimkou je zákaz použití znaků uvozovek.

Zde popsaný formát bude dále používán pro definici konfigurací jednotlivých modelů a také pro definici složení celé scény.

3.2.2 Definice konfigurací modelu

Vzhledem k různorodosti jednotlivých konfigurací modelu není možné naprosto přesně definovat, jak by měla konfigurace modelu vypadat. Každá konkrétní definice konfigurace má svoje vlastní požadavky na jednotlivé atributy. Proto lze o jednotlivých popisech konfigurací mluvit až v okamžiku implementace systému, kdy je jasné, jaké konfigurace bude tento systém obsahovat a co budou jednotlivé konfigurace vyžadovat za atributy.

Jedinou povinnou informací pro konfiguraci je vždy její jméno v rámci souboru s popisem modelu. Podle tohoto jména může zobrazovací systém rozpoznat, zda daná konfigurace je nebo není přítomna pro daný model. Jakmile systém požadovanou konfiguraci vyhledá, musí se obsah atributů této konfigurace shodovat s požadavky systému, jinak je konfigurace neplatná a zobrazovací systém to považuje za stejnou situaci, jako kdyby konfigurace pro zadaný model neexistovala.

Protože je konfigurace vždy součástí modelu, není pro ní definován typ souboru. Je vždy uložena v souboru s popisem modelu, který je ve formátu popsaném v kapitole 3.2.1.

3.3 Popis modelu

Model je podle uvažovaného návrhu vždy tvořen množinou konfigurací modelu, kde příslušná konfigurace modelu obsahuje svoje specifické atributy, kterými může být i geometrie modelu (případně jich může být i více). Model je definován v textovém souboru ve formátu popsaném v kapitole 3.2.1.

Již dříve bylo zmíněno, že jednotlivé konfigurace jsou vzájemně dosti odlišné a tedy jediný požadavek na soubor modelu je ten, že na nejvyšší úrovni jsou vždy správně uloženy bloky odpovídající jednotlivým konfiguracím.

Pro ilustraci definujme zobrazovací systém, který obsahuje konfiguraci zobrazení pro model v osvětlení bodovým světelným zdrojem (konfigurace *PointLight*), dále konfiguraci pro zobrazení modelu bez osvětlení (konfigurace *AmbientLight*) a například konfiguraci pro vykreslení modelu pouze se zapsáním do paměti hloubky (konfigurace *DepthPass*). Soubor s definicí takového modelu může vypadat například takto:

```
[PointLight] {
      ; zde jsou atributy a bloky specifické pro konfiguraci PointLight
      ; tedy pro zobrazení modelu v osvětlení bodovým světelným zdrojem
}
[AmbientLight] {
      ; zde jsou atributy a bloky specifické pro konfiguraci AmbientLight
}
[DepthPass] {
      ; zde jsou atributy a bloky specifické pro konfiguraci DepthPass
}
```

Zobrazovací systém může z takto definovaného souboru načíst všechny bloky a poté podle jména vyhledávat jednotlivé konfigurace, které potřebuje k zobrazení daného modelu. V zadané konfiguraci pak musí nalézt všechny potřebné informace pro zobrazení modelu. Např. v situaci, kdy potřebuje zobrazovací systém vykreslit všechny modely do paměti hloubky (např. při vykreslování stínů ve scéně), bude nutné vyhledat výše uvedenou konfiguraci *DepthPass* pro každý model. Ta může být definována např. následujícím jednoduchým způsobem:

```
[DepthPass] {
      ; zde jsou atributy a bloky specifické pro konfiguraci DepthPass
      ; tedy zobrazení modelu pouze do hloubkového bufferu

      ; tento atribut definuje jméno souboru s geometrií
      „geometry“ = “jmeno souboru s geometrií”
}
```

Zobrazovací systém očekává tento formát konfigurace, nalezne tedy požadovanou konfiguraci *DepthPass* a z ní získá z atributu „geometry“ informaci o geometrii modelu pro vykreslení. Tuto geometrii pak systém zobrazí podle svých vlastních pravidel.

Uvedená konfigurace *DepthPass* může být v systému definována i jiným způsobem. Vhodná definice by byla např. takto:

```
[DepthPass] {
      ;počet úrovní detailu geometrie - celkově dvě
      „lod_count“ = „2“

      ;tento blok určuje první úroveň detailu pro tuto konfiguraci
      [lod]{
            „level“ = „0“     ; definice první úrovně detailů
            „geometry“ = “jmeno souboru s geometrií”
      }

      ;tento blok určuje druhou úroveň detailu pro tuto konfiguraci
      [lod]{
            „level“ = „1“     ; definice druhé úrovně detailů
            „geometry“ = “jmeno souboru s geometrií”
      }
```

V této definici konfigurace systém vyžaduje zadání počtu úrovní detailu v atributu *lod_count*. Tento parametr určuje počet vnořených bloků, které pak systém vyhledává a podle vnitřního nastavení rozhoduje o tom, kterou úroveň detailu zvolit pro vykreslení geometrie. Popsanou konfiguraci lze dále obohatit o další atributy, např. by v každém bloku *[lod]* mohla být informace, která by určovala vzdálenost, od které vykreslovat model pomocí této úrovně detailu. Tím pádem by bylo možné řídit úroveň detailů zobrazování na základě datového popisu, což je přesně záměr datově řízeného zobrazovacího systému.

Z výše uvedených příkladů je patrné, že model definovaný množinou konfigurací je dostatečně popsaný. Nicméně nachází se stále v lokálních souřadnicích (tedy nemá svoji pozici ve scéně). Proto zbývá ještě popsat návrh celkového obsahu scény, která bude definovat umístění jednotlivých modelů.

3.4 Popis scény

Podle obrázku 3.1 se scéna skládá z modelů a parametrů. Model byl v přecházející kapitole definován jako textový soubor obsahující množinu konfigurací modelu, které jsou konkrétně definovány v implementaci systému. Zbývá tedy navrhnout způsob popisu parametrů scény. Scéna pak bude popsána množinou parametrů a modelů. Po dokončení návrhu scény již bude možné přistoupit k popisu vlastního zobrazovacího systému vycházejícího ze zde vytvořeného popisu scény.

3.4.1 Parametry scény

Vzhledem k tomu, že parametry scény mohou být opět velmi různorodé (podobně jako konfigurace modelu), není možné je přesně specifikovat dokud nebude specifikována konkrétní implementace zobrazovacího systému. Budeme se proto zabývat pouze způsobem definice parametrů scény.

Kvůli dodržení konzistentního přístupu při budování scény byl pro popis parametrů scény zvolen stejný textový formát jako pro definici jednotlivých konfigurací v rámci modelu. Narozdíl od konfigurace modelu je však každý parametr scény definován v jednom textovém souboru[7]. Takový soubor může obsahovat např. kompletní definici pro světelný zdroj (barvu, dosah světla, atd.), tento světelný zdroj je poté umísťován do scény (viz dále). Narozdíl od souboru s popisem modelu (který obsahoval množinu konfigurací modelu) smí soubor s informacemi o parametrech scény obsahovat pouze jeden hlavní blok, který identifikuje svým jménem přesně parametr scény. V tomto bloku smějí být samozřejmě obsaženy další bloky, takže z hlediska uložené informace není soubor s parametrem scény nijak omezen. Jméno hlavního bloku slouží systému jako jednoznačný identifikátor parametru scény. Ukázka možné definice souboru s popisem bodového zdroje světla je znázorněna zde:

```
;konfigurace bodového zdroje světla ve scéně
[PointLightParam] {
      ;difusní barva světla
      „diffuse_color“ = „255;255;255“

      ;maximální dosah světla
      „radius“ = „120“
}
```

Jak je vidět, obsahuje parametr scény základní informace, které zobrazovací systém očekává. Zde je to např. barva světla a maximální dosah. Z těchto informací je už zobrazovací systém schopen např. určit objekty v dosahu tohoto světelného zdroje a tyto potom řádně osvětlit.

Systém tedy obsahuje množinu povolených parametrů scény a ty jsou poté definovány v předepsaném tvaru v jednotlivých souborech. Z teoretického hlediska jsou parametry scény velmi podobné modelům, nicméně z pohledu výsledného zobrazování scény se jedná o množinu zcela odlišných prvků scény a proto je jejich definice také jiná. Definicí parametrů scény je již kompletně dokončena definice všech prvků scény a je tedy možné přistoupit k návrhu popisu rozmístění prvků ve scéně.

3.4.2 Popis obsahu scény

Scéna se skládá z výše definovaných modelů a parametrů (dále jen prvků) scény, které jsou definovány ve vlastních souborech ve formátu popsaném v kapitole 3.2.1. Pro popis scény je tedy

[7] To je logické, neboť parametr scény se neváže na žádný prvek scény.

pouze nutné definovat rozmístění jednotlivých prvků v rámci scény[8]. Narozdíl od definice konfigurací a parametrů je zde možné rovnou definovat výsledný návrh popisu scény, neboť ten se vždy omezuje pouze na rozmístění prvků ve scéně.

Scéna je definována opět textovým souborem, jehož formát je identický s formátem zavedeným v kapitole 3.2.1. V tomto formátu jsou podle aktuálního návrhu povoleny pouze tři typy (jména) bloků, které jsou popsány v následující tabulce 3.4:

model	Blok s názvem **model** obsahuje informaci o zadaném modelu scény.
param	Tento blok obsahuje informace o parametru scény.
scene	Pomocí tohoto bloku lze definovat zanoření scén a vytvářet tak scénu hierarchicky.

Tabulka 3.4 - základní bloky pro popis scény

Všechny tři bloky mají povinné atributy uvedené v tabulce 3.5:

name	Jméno tohoto prvku scény. Toto jméno slouží k identifikaci prvku pro tvůrce scény a mělo by být jedinečné, nicméně to není podmínkou, protože systém identifikuje prvky scény vlastním způsobem. „name" = „jméno prvku scény"
position	Umístění prvku v aktuální scéně. Hodnota atributu obsahuje pozici v trojrozměrném prostoru v tomto formátu: „position" = „pozice_x;pozice_y;pozice_z"
rotation	Rotace prvku ve scéně. Tato rotace udává lokální rotaci prvku, tedy rotaci aplikovanou na prvek ještě před posunutím na zadanou pozici **position**. Hodnota tohoto atributu je vyjádřena jako čtyřsložkový vektor reprezentující jednotkový kvaternion (více o kvaternionech viz příloha A.1) shodný s odpovídající rotací. Formát je následující: „position" = „qx;qy;qz;qw"
file	Soubor obsahující popis prvku (modelu, parametru, vnořené scény). Formát je následující: „file" = „jméno souboru s popisem prvku"

Tabulka 3.5 - atributy bloků pro popis scény

Jak je vidět, lze scénu poměrně dobře stavět hierarchicky pomocí bloku **scene**. Stačí vytvořit jednoduchou scénu v samostatném souboru a tu pak vložit do další scény pomocí tohoto bloku. Jediný problém tohoto přístupu je možné vytvoření cyklické závislosti, kdy např. soubor se scénou **A** obsahuje vloženou scénu ze souboru **B**, ve které je opět vložena scéna **A**. Taková situace je z hlediska popisu scény dost obtížně řešitelná a musí jít být zabráněno tvůrcem obsahu scény. V konkrétní implementaci systému pak taková situace může být ošetřena během načítání scény, kdy lze cyklickou závislost detekovat při procházení stromem scény.

[8] Pouze pokud to prvek vyžaduje, typicky některé parametry scény nejsou konkrétně umístěny ve scéně (např. informace o všudypřítomném světle apod.).

Zde představený popis scény se všemi podčástmi již tvoří požadovaný základ pro skutečný popis scény datově řízeného zobrazovacího systému. Tím je tedy dokončena fáze návrhu popisu scény a je možné se začít zabývat popisem vlastního zobrazovacího systému a jeho implementace.

4 Implementace zobrazovacího systému

V následujícím textu se již budeme zabývat popisem vlastní implementace zobrazovacího systému založeném na poznatcích a návrzích z předcházejících kapitol vzniklých v rámci semestrálního projektu, na nějž navazuje tato diplomová práce. Nejprve bude diskutována základní architektura celého systému a dále budou podrobně rozebrány její hlavní prvky, které jsou součástí vytvořené implementace. Zároveň budou také diskutována jejich další možná budoucí vylepšení a rozšíření.

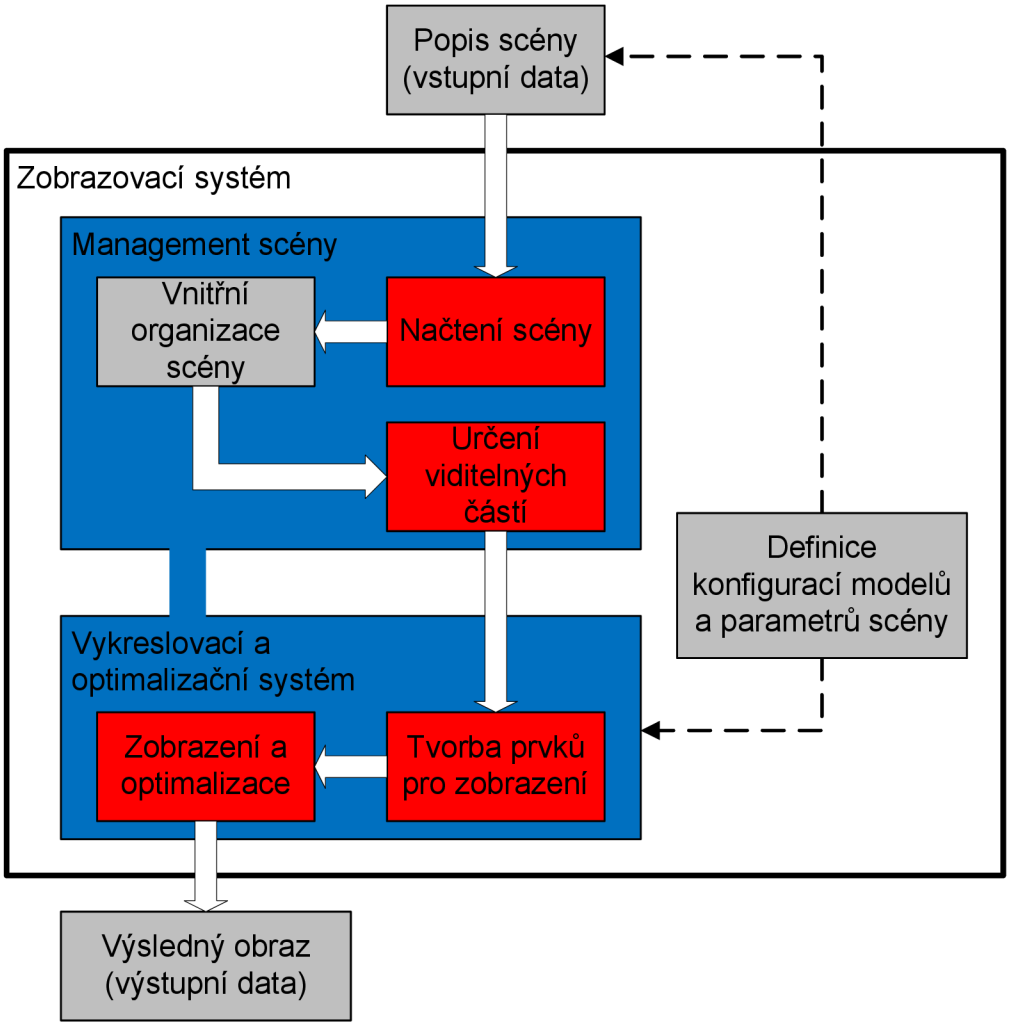
4.1 Základní architektura zobrazovacího systému

Zobrazovací systém je stejně jako každý jiný softwarový systém nutné před implementací správně analyzovat a navrhnout jeho základní architekturu. Zobrazovací systémy se dnes prakticky bez výjimky navrhují podle vzorové softwarové architektury, která má podobu „řetězce“, tedy několika softwarových celků spojených sériově. Ty postupně provádí jednotlivé činnosti nad vstupními daty, čímž je transformují na výsledný obraz. Aktuální implementace zobrazovacího systému také vychází z této softwarové architektury, přičemž ale její komponenty nejsou striktně spojeny pouze sériově. Základní zjednodušené schéma celé navržené architektury je na obrázku 4.1.

Na uvedeném obrázku je znázorněn celý zobrazovací systém, jehož vstupem je popis scény a výstupem vykreslený obraz. Zobrazovací systém transformuje vstupní data na výstupní data, přičemž tato transformace (tedy tok dat) je znázorněna bílými šipkami. Zobrazovací systém obsahuje množinu definic konfigurací modelů a parametrů scény, které byly popsány v předchozím textu a podle kterých se řídí vytváření vstupního popisu scény a zároveň její vykreslování.

Celý zobrazovací systém lze z logického pohledu rozdělit na dvě části, které jsou na obrázku 4.1 znázorněny modrou barvou a kterým se budeme důkladně věnovat ve velké části následujícího textu. První z nich je celkový management scény, který v sobě zahrnuje především problémy organizace scény, řešení viditelnosti a správu grafických datových zdrojů. Druhou část systému pak tvoří vlastní proces vykreslování scény, který má na starosti především správnou aplikaci grafických efektů a také optimalizaci celého procesu vykreslování.

Obě hlavní zmíněné části jsou součástí jednoho systému a tedy mezi sebou musí spolupracovat. Jako základní forma spolupráce se předpokládá, že management scény poskytuje vstupní data vykreslovacímu systému. V určitých situacích však potřebuje vykreslovací systém další informace udržované managementem scény. Z toho důvodu musí být oba celky propojeny (to na obrázku naznačuje modré přemostění), přitom základním požadavkem na toto propojení je, aby vazba mezi oběma celky byla minimální tak, aby úpravy v jedné části příliš neovlivňovaly druhou část, což je základní předpoklad každého správně navrženého softwarového systému. Proto i tomuto tématu byla při implementaci věnována pozornost a v textu na ní bude upozorněno.



Obr. 4.1 – schéma základní architektury zobrazovacího systému

4.2 Poznámka k názvosloví

V následujícím textu budou často používány některé výrazy, jejichž záměna může vést k snadnému zmatení čtenáře a proto je vhodné je pro jistotu důrazně vysvětlit. Jedná se především o pojem „konfigurace“ zavedený v kapitole 3.2. Tento pojem velmi jednoduše označuje jeden blok na hlavní úrovni souboru s popisem modelu (viz kapitola 3.3). Nicméně zde se bude používat ve třech možných významech. Pokud bude použito slovní spojení „konfigurace modelu“, pak se jedná o jednu sekci ze souboru s popisem modelu přesně ve shodě s kapitolou 3.2. Naopak slovní spojení „konfigurace zobrazení scény“ nebo krátce „konfigurace scény“ označuje jisté lokální podmínky scény, které určují, jakým způsobem se mají zobrazovat prvky v dané oblasti. Slovní spojení „zobrazovací konfigurace modelu“ pak označuje takovou konfiguraci modelu, která je svázána se specifickou konfigurací zobrazení scény a je tedy využívána pro zobrazení modelu v této konfiguraci scény[9].

[9] Tedy ne všechny konfigurace modelu jsou zobrazovací konfigurace modelu!

5 Management scény

Dobrý management scény je nutným základem pro správné fungování každého zobrazovacího systému pracujícího s rozsáhlými a komplexními scénami. Jedním z hlavních měřítek kvality systému pro management scény je především stupeň jeho oddělení od vlastního vykreslovacího procesu (popsaného v kapitole 6). Pokud je toto oddělení maximální (a tedy vazby mezi oběma celky jsou minimální), pak lze obě části snadno vylepšovat bez nutnosti neustále promítat změny provedené v jedné části do části druhé. Právě toto byl jeden z hlavních požadavků na zde popisovanou implementaci managementu scény, jejíž řešení bude popsáno v následujícím textu.

5.1 Načtení a organizace scény

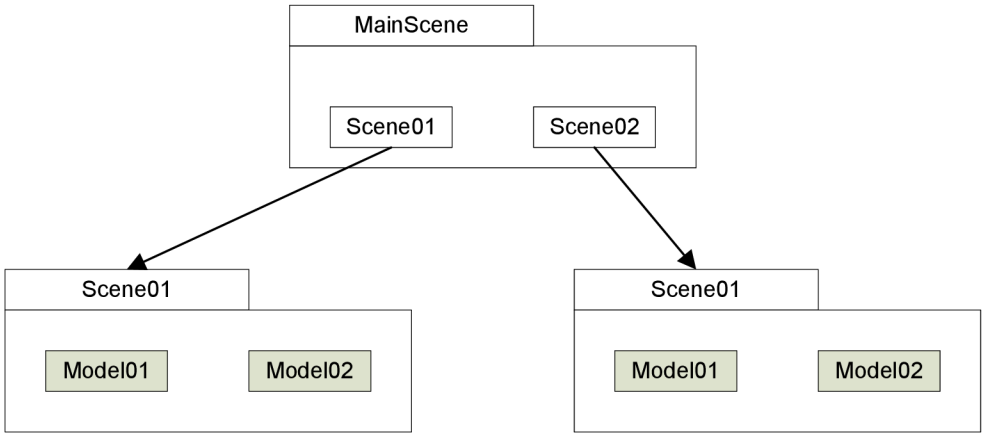
Pro management scény je nejprve nutné zvolit správnou organizaci scény. Z kapitoly 3.4 víme, jak vypadá vstupní popis scény, pomocí něhož tvůrci obsahu scénu definují. Tento popis umožňuje definovat scénu hierarchicky jako datovou strukturu typu strom (viz např. [23]), jehož koncové uzly (listy) obsahují odkazy na soubory s popisy modelů a parametrů scény (dále jen prvky). Pro potřeby vlastního zobrazování a kompletního managementu prvků je však výhodné převést stromovou strukturu popisu scény na vhodnější tvar, který bude popsán dále.

Aktuální implementace provádí načtení a převod ve třech fázích, první fáze převádějící strom na lineární strukturu prvků je následující:

1. Nejprve je načten hlavní soubor s popisem scény, který představuje kořenový uzel celého stromu.
2. Dále jsou iteračně načítány další soubory s popisem scény (pouze bloky `[scene]` v již načtených souborech) na které vedou odkazy z již načtených souboru s popisem scény. Při tomto iteračním načítaní je prováděna kontrola na cyklické vazby mezi soubory, takže je zamezeno zacyklení procesu načítaní. Výsledkem tohoto kroku je stromová struktura vzájemně odkazovaných souborů s popisem scény.
3. Ze stromové struktury získané v kroku 2 jsou získány seznamy (tj. již lineární struktury) odkazů na soubory s popisem modelů (bloky `[model]`) a parametrů (bloky `[param]`). Každý odkaz je přitom představován pouze jménem cílového souboru s popisem prvku a transformační maticí, která je získána průchodem stromu z kroku č.2 od kořene směrem k příslušnému prvku. Jedná se tedy prozatím o paměťově nenáročné prvky.

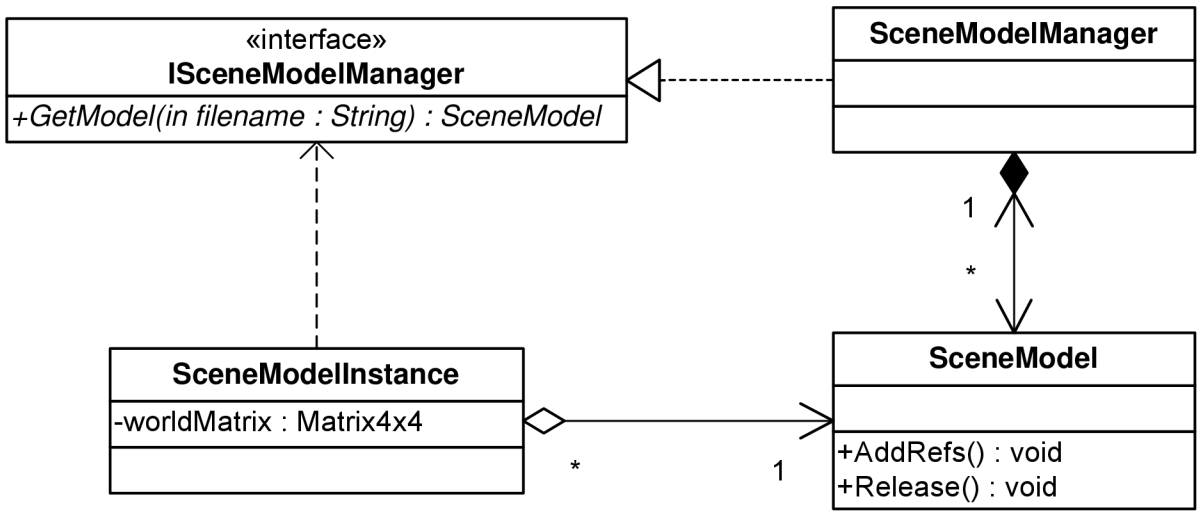
Po této první fázi máme tedy k dispozici seznam odkazů na soubory prvků včetně jejich umístění vyjádřeného transformační maticí. Zbavili jsme se tedy hierarchické struktury, která mohla být libovolně komplikovaná a máme jednoduchou lineární strukturu prvků. Je ale zřejmé, že díky způsobu popisu scény je možné, aby některé soubory prvků byly v této lineární struktuře odkazovány

vícekrát, pokaždé pouze s jinou transformační maticí[10]. Příklad takové situace je znázorněn na obrázku 5.1, kde soubor **MainScene** s popisem hlavní scény obsahuje odkazy na dvě podscény popsané v jenom identickém souboru **Scene01**. Výsledná scéna tedy bude mít celkem 4 modely, přitom celkový počet navzájem různých modelů bude 2.



Obr. 5.1 – ukázka redundance stromové struktury vzniklé načtením scény

Vzhledem k tomu, že modely často obsahují datově rozsáhlé prvky (parametrů se toto netýká, proto mohou a zatím jsou uloženy redundantně), je vhodné zamezit redundantnímu načítání modelů. K tomu se nabízí využít pro uložení modelů v organizaci scény klasické schéma instance-model-manažer, znázorněné konceptuálním diagramem tříd na obrázku 5.2, které zároveň definuje základní třídy použité pro organizaci scény.



Obr. 5.2 – organizace prvků scény pomocí schématu instance-model-manažer

V tomto schématu se do nově vytvářené organizace scény neukládají přímo modely (reprezentované třídou `SceneModel`), ale pouze instance modelu typu `SceneModelInstance`, které pomocí manažeru modelů získávají referenci na příslušný model typu `SceneModel`, jenž implementuje

[10] Ve skutečnosti je tento případ velmi častý, naopak jen velmi vzácně se ve scéně vyskytuje model, který je použit pouze jednou.

metody pro počítaní referencí (více např. v [22]) a může tak být efektivně spravován manažerem objektů typu `SceneModel`.

Třída `SceneModel` tedy představuje skutečný model načtený ze souboru s popisem modelu. V tomto okamžiku zatím budeme předpokládat, že obsahuje pouze načtený konfigurační soubor s popisem modelu, což je pro management scény zatím dostačující informace. Další upřesnění třídy `SceneModel` bude popsáno v kapitole 6.2.1.

Instance modelu typu `SceneModelInstance` musí obsahovat všechny informace specifické pro daný prvek, přičemž v aktuálně použitém popisu scény se jedná vždy pouze o transformační matici daného prvku určující umístění instance modelu v prostoru. Všechny ostatní informace jsou již společné pro celý objekt typu `SceneModel` a mohou tedy být získány pomocí reference na tento objekt. Vzhledem k možnosti postupného načítaní obsahu scény (viz dále v kapitole 5.3.2) a s tím související možností dočasně odstranit nevyužívané datově rozsáhlé prvky z paměti, je vhodné do datového typu `SceneModelInstance` uložit ještě další pomocné informace. V aktuální implementaci je vedle transformační matice uložen také ohraničující kvádr instance modelu transformovaný do světových souřadnic scény. Ten je načten ze souboru s popisem modelu, kde musí být uveden v konfiguraci modelu se jménem `[bounding-box]`[11], která je povinnou konfigurací modelu pro každý soubor s popisem modelu. Celá druhá fáze načítání a převodu tedy vypadá následovně:

1. Postupně procházej seznam odkazů na modely z první fáze.
2. Pomocí rozhraní manažeru modelů získej referenci na model typu `SceneModel` podle jména souboru. Pokud nelze referenci na model získat, pokračuj krokem č.1.
3. Z popisu modelu získej ohraničující kvádr v lokálních souřadnicích pomocí konfigurace modelu `[bounding-box]`. Pokud taková konfigurace modelu neexistuje, pokračuj krokem č.1.
4. Vytvoř objekt instance modelu typu `SceneModelInstance`, ulož do něj transformační matici, dále referenci na model získanou v kroku č.2 a ohraničující kvádr transformovaný transformační maticí této instance modelu. Vytvořený objekt pak ulož do seznamu instancí modelů a pokračuj krokem č.1.

Po provedení druhé fáze máme tedy k dispozici seznam instancí modelů typu `SceneModelInstance`, které obsahují reference na modely typu `SceneModel` spravované manažerem modelů. Nyní už můžeme přejít k třetí a poslední fázi, která uloží instance modelů do struktury vhodné pro management scény.

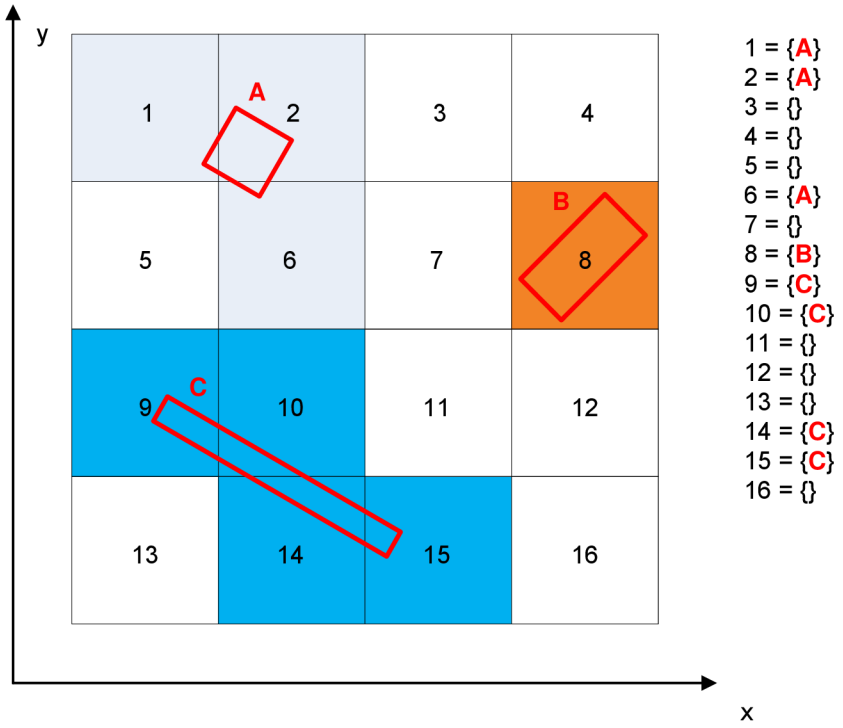
Pro toto uložení byla zvolena dvourozměrná dlaždicová struktura, která umožňuje scénu dělit na menší oblasti a s nimi později pracovat samostatně např. při určování viditelnosti. Rozdělení scény

[11] Popis této konfigurace modelu stejně jako dalších lze nalézt v externím dokumentu na přiloženém na CD.

na dlaždice je určeno parametrem `[scene-area]`[12] použitém v souboru s popisem hlavní scény. Tento parametr určuje velikost jedné dlaždice a počet dlaždic podél os X a Y[13]. Poslední fáze sloužící k uložení instancí modelu do dlaždicové struktury probíhá následovně:

1. Vytvoř všechny dlaždice tvořící celou scénu podle parametru `[scene-area]`.
2. Postupně procházej všechny instance modelů vytvořené v předchozí fázi.
3. Otestuj, zda instance modelu leží uvnitř definované oblasti scény, pokud leží mimo pak pokračuj krokem č.2.
4. Přidej referenci na instanci modelu do všech dlaždic, do kterých padne projekce ohraničujícího kvádru podle osy Z pro danou instanci modelu. Pokračuj krokem č.2.

Výsledkem poslední fáze je tedy seznam dlaždic, kde každá dlaždice má přiřazen seznam referencí na instance modelů, které k této dlaždici patří po projekci na rovinu XY. Tato situace je pro ilustraci znázorněna na obrázku 5.3.



Obr. 5.3 – znázornění umístění jednotlivých instancí modelů do dlaždicové struktury pomocí projekce

Na něm je zobrazena projekce ohraničujících kvádrů (A,B,C) příslušných instancí modelů do oblasti scény a následně jsou barevně vyznačeny dlaždice, ke kterým příslušné instance modelů patří. Vpravo je pak seznam dlaždic a pro každou dlaždici seznam instancí modelů, které k dlaždici náleží.

Po provedení poslední fáze máme tedy k dispozici kompletně načtenou scénu reprezentovanou seznamem dlaždic a k nim přiřazené reference na instance modelů, což je již velmi dobrá struktura

[12] Tento parametr hlavního souboru scény je povinný a pokud není nalezen, pak scénu nelze načíst! Oblast je definována pouze v osách X a Y, podél osy Z se oblast definuje až po načtení scény jednotlivě pro každou dlaždici. Jeho popis lze nalézt v externím dokumentu na přiloženém CD.
[13] Předpokládá se pravotočivý souřadný systém s osou Z směřující „nahoru“. Tedy osa X směřuje „doprava“, zatímco osa Y „dopředu“. Viz např. [20].

organizace scény. Ta zatím zároveň nemá žádné větší paměťové nároky, neboť obsahuje pouze instance modelů, přičemž samotné modely zatím obsahují pouze načtené konfigurační soubory, což jsou opět relativně paměťově nenáročné prvky. V rozšiřování této organizace scény budeme pokračovat v následujícím textu.

5.2 Řešení viditelnosti

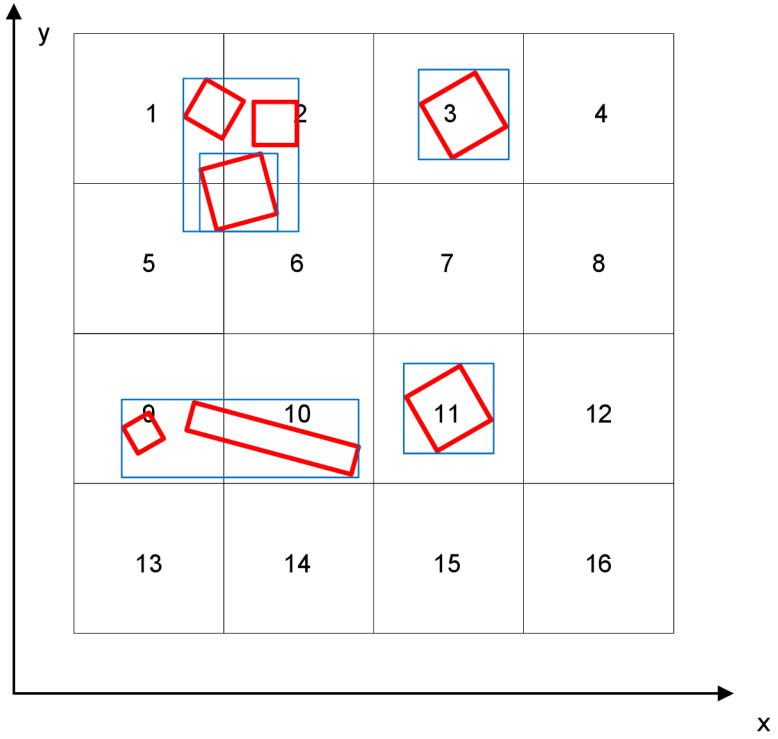
Problém řešení viditelnosti patří při zobrazování rozsáhlých scén k těm vůbec nejvážnějším, protože i přes rychlé tempo růstu výkonu dnešního hardwaru není možné plýtvat výkonem grafického subsystému zobrazováním těch částí scény, které stejně nejsou vidět. Stejně tak ale není možné provádět naprosto přesné testování viditelnosti jednotlivých prvků scény (např. každého trojúhelníku samostatně), které by sice ulehčilo práci grafickému subsystému, ale příliš by vytěžovalo centrální procesor potřebný i pro jiné úlohy. Proto je při řešení viditelnosti nutné nalézt rozumný kompromis, který bude ohleduplný jak na výkon procesoru, tak grafického subsystému. Rozumný systém pro řešení viditelnosti musí být také dostatečně univerzální, tj. musí být schopen rozhodovat o viditelných částech scény podle požadovaných kritérií (typicky podle základních objemových těles určujících aktuální zájmovou oblast ve scéně), a musí být schopen tuto činnost provádět například i opakovaně během jednoho snímku, aniž by se dotazy na viditelnost vzájemně ovlivňovaly. .

Zde popsané implementované řešení viditelnosti využívá dlaždicovou strukturu organizace scény vzniklou při načtení scény popsaném v kapitole 5.1. V jednotlivých dlaždicích jsou uloženy instance modelů, které k dlaždicím přísluší po projekci do roviny XY. Vzhledem k tomu, že při načítání scény jsou do instancí modelů načteny z modelů informace o ohraničujících kvádrech jednotlivých modelů, můžeme při řešení viditelnosti uvažovat pouze o instancích modelů a odkazovanými modely typu `SceneModel` se nemusíme zabývat. To se ukáže jako výhoda později, kdy se budeme zabývat otázkou managementu datových zdrojů a postupného načítání obsahu scény.

Prvním důležitým krokem při řešení viditelnosti je vybudování hierarchické struktury vhodné pro rychlé testování viditelnosti. Zde se nabízí několik tradičních a často používaných metod dělení prostoru, jakými jsou např. Octtree, Kd-Tree nebo BSP-Tree (viz např. [6]), nicméně v aktuální implementaci byla zvolena mnohem výhodnější[14] hybridní technika, která je založena na dělení prostoru a zároveň na shlukování instancí modelů. Dělení prostoru je uvažováno pouze v rovině XY a využívá dělení podle dlaždic vytvořených v kapitole 5.1, čímž tuto dlaždicovou strukturu dále rozšiřuje[15]. V každé dlaždici se viditelnost řeší na základě shlukování instancí modelů v ní obsažených. Konkrétně se po načtení scény pro každou dlaždici vypočítá z instancí modelů v ní obsažených ohraničující kvádr zarovnaný s osami souřadného systému XYZ, který bude dále

[14] Tato technika je výhodná tím, že narozdíl od uvedených algoritmů nemusí pokrývat celý prostor, ale pouze ty jeho části, které obsahují nějaké prvky.
[15] Prostor se tedy dělí pouze jednou, proto je důležité aby byla optimálně zvolena velikost dlaždice již při tvorbě scény, což je plně v odpovědnosti tvůrce scény.

používán pro rozhodnutí, zda se mají instance modelů uvnitř dlaždice vůbec testovat. Generování ohraničujících kvádrů pro dlaždice je znázorněno na obrázku 5.4 při pohledu podél osy Z. Na tomto obrázku je patrné, že dlaždice 3 a 11 vytvořily ohraničující kvádr, který je menší než celá dlažice, a tedy tím zvýhodní budoucí testování viditelnosti. Je také patrné, že dlaždice neobsahující žádný prvek negenerují žádný ohraničující kvádr a mohou být z testování viditelnosti vynechány. Problém však zřetelně nastává u dlaždic č. 1, 2, 5, 6, 9 a 10. Tam jsou jednotlivé instance modelů obsaženy ve více než jedné dlaždici a vygenerované ohraničující kvádry se tedy překrývají, což logicky vyplývá z toho, že vygenerovaný ohraničující kvádr pro instanci modelu může zasahovat i mimo dlaždici. Toto překrývání následně působí problémy při testování viditelnosti pomocí klasické metody, kterou lze popsat takto:



Obr. 5.4 – znázornění generování ohraničujících kvádrů pro jednotlivé dlaždice

1. Testuj každou dlaždici s vygenerovaným ohraničujícím kvádrem (tedy s alespoň jednou instancí modelu) na viditelnost pomocí jejího ohraničujícího kvádru.
2. Pokud je ohraničující kvádr dlaždice viditelný, pak testuj každou instanci modelu v této dlaždici na viditelnost.
3. Pokud je daná instance modelu viditelná, pak ji přidej do seznamu viditelných instancí modelů určených pro vykreslení.

Z obrázku 5.4 je vidět základní problém tohoto postupu. Ten se projeví např. u dlaždic č. 5 a 6, které obě sdílejí jednu stejnou instanci modelu a zároveň jeden stejný ohraničující kvádr. Je tedy zřejmé, že pro obě dlaždice bude testování viditelnosti vracet vždy stejné výsledky, což v konečném důsledku bude znamenat, že se v seznamu viditelných instancí modelů budou některé prvky opakovat, což způsobí, že vykreslování bude redundantní. Tento problém lze samozřejmě následně řešit

odstraněním redundantních položek ze seznamu viditelných instancí modelů, příp. lze kontrolovat redundantní položky už při přidávání instancí modelů do seznamu. Takové metody však mohou vést na zbytečně složité algoritmy opakovaného vyhledávání v poli (především pro velké množství položek mohou být časově velmi náročné). Proto je mnohem přijatelnější použít techniku, která zabezpečí, že každá instance modelu může být přidána do seznamu viditelných objektů pouze jednou. Takovou technikou může být např. speciální značka v objektu instance modelu, která jasně určí, zda byla právě testovaná instance modelu již použita při aktuálním testu viditelnosti. Bohužel použití značky s booleovskou hodnotou typu ANO-NE přináší nevýhodu v tom, že před každým testem je nutné vymazat tyto značky u všech instancí modelů, což i přes triviálnost této operace může např. ve scéně s milionem instancí modelů znamenat poměrně časově náročnou operaci[16]. Proto bylo pro značku použito místo booleovské hodnoty 32-bitové bezznaménkové číslo, jehož funkce je následující. Po načtení scény jsou všem instancím modelů nastaveny značky na hodnotu 0 a v objektu pro správu scény je nastavena tzv. globální značka na hodnotu 1. Při testování se pak postupuje následujícím způsobem.

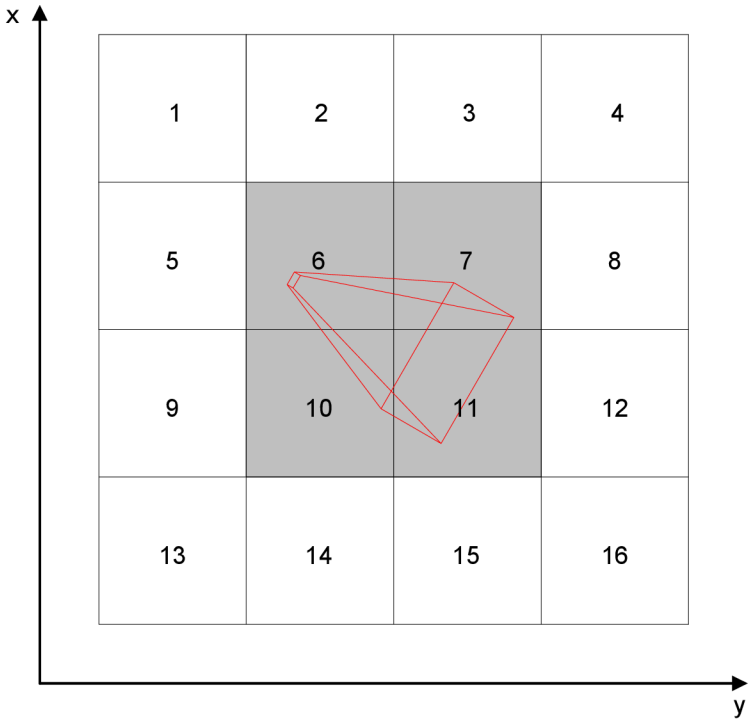
1. Testuj každou dlaždici s vygenerovaným ohraničujícím kvádrem (tedy s alespoň jednou instancí modelu) na viditelnost pomocí jejího ohraničujícího kvádru.
2. Pokud je ohraničující kvádr dlaždice viditelný, pak testuj každou instanci modelu v této dlaždici na viditelnost.
3. Před testem instance modelu porovnej její značku s globální značkou. Pokud jsou různé, pak tato instance modelu ještě nebyla testována. V takovém případě proveď testování viditelnosti pro tuto instanci modelu, pokud je viditelná tak ji zařaď do seznamu viditelných instancí modelu a nakonec nastav hodnotu značky instance modelu na hodnotu globální značky. Pokud je naopak globální značka shodná se značkou instance modelu, pak se žádné další testování neprovádí a pokračuje se další dostupnou instancí modelu.
4. Po otestování všech instancí modelů ve všech dlaždicích zvyš hodnotu globální značky o 1, což zajistí, že příští testování bude opět aplikováno na všechny instance modelů.

Pomocí tzv. globální značky jsme tedy bez zvýšených nároků na výpočet dosáhli jedinečnosti každé instance modelu ve výsledném seznamu viditelných instancí modelů. Tyto instance už lze dále považovat za viditelné a tedy je možné je přímo zaslat do vykreslovací části, která se postará o jejich správné vykreslení na základě modelů typu `SceneModel`, jejichž reference jsou v instancích modelů obsaženy. Další výhodou použití globální značky je také možnost provádět další pomocné testování viditelnosti mnohokrát během jednoho vykreslovaného snímku, což bývá užitečné především pro vytváření složitějších grafických efektů, jakými jsou např. stíny nebo odrazy okolního prostředí, které vyžadují pomocné testy viditelnosti objektů ve scéně. Globální značka umožňuje

[16] Navíc operace testování viditelnosti může probíhat během vykreslení jednoho snímku mnohokrát, což by náročnost celé operace nastavování značek ještě více zvýšilo.

provádět tyto pomocné testy viditelnosti bez zbytečných přípravných operací a tím opět pomáhá urychlit běh celého zobrazovacího systému.

Aktuálně implementovaný systém pro určování viditelnosti samozřejmě umožňuje detekovat viditelnost instancí modelů na základě různých objemových těles, která reprezentují aktuální zájmovou oblast scény. V aktuální implementaci je schopen detekovat viditelné instance modelů na základě průniku s běžně používanými objemovými tělesy, kterými jsou pohledová pyramida (vhodná pro běžné pozorování a světelné zdroje), libovolně orientovaný kvádr (vhodný pro směrová světla), kvádr zarovnaný s osami souřadného systému a koule (vhodná např. pro bodová světla s nastaveným rozsahem), přičemž pro každé toto těleso je detekce implementována maximálně efektivním způsobem[17]. Pro všechny prováděné detekce je také důležité, že se ještě před začátkem testování všech dlaždic vyřadí automaticky ty dlaždice, které v žádném případě nemohou být viditelné. To je např. pro test pohledovou pyramidou znázorněno na obrázku 5.5. Použité objemové těleso je jednoduše projektováno na osy X a Y souřadného systému a z této projekce jsou určeny minimální a maximální indexy testovaných dlaždic podél os X a Y. Tímto způsobem je možné okamžitě omezit počet testovacích dlaždic na minimum (na obrázku 5.5 jsou vybarveny šedou barvou).



Obr. 5.5 – znázornění rychlé eliminace dlaždic při testu viditelnosti pomocí projekce testovacího tělesa do roviny XY

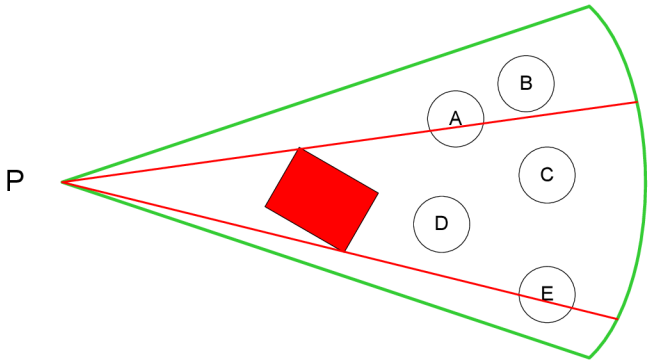
5.2.1 Pokročilé techniky řešení viditelnosti

Kapitola 5.2 popisuje snadný způsob řešení viditelnosti, jehož výstupem je množina viditelných instancí modelů. Aktuální implementace systému využívá pouze tento způsob detekce viditelnosti,

[17] Techniky řešení viditelnosti jsou založeny na určování kolizí mezi použitými geometrickými tělesy. O těchto technikách se lze dozvědět více např. [5] nebo přímo ve zdrojových kódech implementace.

který se dá považovat za dostatečný z hlediska vyváženosti zatížení grafického subsystému a centrálního procesoru. Nicméně je vhodné alespoň z teoretického hlediska popsat další techniky, které je možné použít pro detekci viditelných částí scény. V implementovaném systému připadají v úvahu dvě zajímavé techniky – modelové uzávěry a uzávěry parametrů[18]. Obě techniky jsou vesměs identické, rozdíl je pouze ve fázi, kdy se aplikují.

Modelový uzávěr je jednodušší technikou, neboť se aplikuje až po první fázi detekce všech viditelných instancí modelů popsané v kapitole 5.2 a tedy ji nijak neovlivňuje. Vstupem modelového uzávěru jsou všechny získané viditelné instance modelů, z nich jsou následně vybrány vhodné uzávěry, tj. takové objekty, které mají největší pravděpodobnost zakrýt objekty nacházející se za nimi směrem od pozorovatele. Poté následují testy, které mohou pomoci vyloučit takto zakryté objekty. Ukázka takového zakrytí je na obrázku 5.6. Na něm pozorovatel v pozici **P** pozoruje scénu s několika objekty, přičemž červený kvádr byl použit jako uzávěr. Tímto uzávěrem je pak možné eliminovat z procesu zobrazení objekty **C** a **D**. Je také zřejmé, že v případě přiblížení pozorovatele k uzávěru dojde ke zvětšení oblasti zakrývané uzávěrem a tedy je tento způsob velmi vhodné použít především v případě velkých dohledových vzdáleností ve scéně, kdy může být počet zakrytých objektů značně vysoký. Momentální implementace zatím bohužel tento způsob detekce viditelnosti neaplikuje, nicméně v popisu geometrie v kapitole 3.1 je možné uložit ke geometrii kromě ohraničujícího kvádru také tzv. vnitřní kvádr, který je nutný pro detekování viditelnosti pomocí uzávěru[19]. Protože testování pomocí modelového uzávěru probíhá až po základním testování, je integrace tohoto testu do stávajícího systému relativně jednoduchá a navíc neovlivňuje zbytek celého systému.



Obr. 5.6 – ukázka použití uzávěrového tělesa pro odstranění zakrytých částí scény

Druhou zmiňovanou technikou jsou uzávěry parametrů. Principiálně se jedná o stejný postup detekce viditelnosti, jako v případě modelových uzávěrů. Jediný rozdíl je pouze v místě použití. Zatímco modelový uzávěr je tradičně aplikován až po první fázi detekce viditelných objektů, uzávěr parametrů je používán přímo v první fázi, přičemž se snaží tuto fázi zefektivnit. Název techniky je odvozen z faktu, že pro uzávěr se nepoužívají modely (i když mohou být také použity), ale speciální parametry

[18] České slovo „uzávěr“ zde nahrazuje anglické slovo „occluder“, i když přesnější by bylo použití výrazu „pohlcovač“. Za uzávěr se považuje takový objekt scény, který způsobí zakrytí a tedy zneviditelnění jiných objektů ve scéně.
[19] Resp. je vyžadováno libovolné konvexní těleso, kvádr byl v souboru s geometrií zvolen kvůli jednoduchosti. Více lze nalézt např. v [2].

scény (tj. prvky scény definované blokem `[param]` v popisu scény). Takový parametr scény může např. definovat uzávěrové těleso, které se bude využívat už během první fáze viditelnost a to již během detekce viditelnosti jednotlivých dlaždic. Tím je možné zefektivnit celý proces detekce viditelnosti. Takový způsob detekce viditelnosti se používá např. ve scénách s městskou zástavbou, kdy je celá řada domů definována jedním (nebo více) parametrovým uzávěrem, který je použit pro rychlé odstranění všech dlaždic za tímto uzávěrem. Vzhledem k tom, že aktuální implementovaný systém je spíše zaměřený na otevřené scény, nebyla implementace tohoto způsobu detekce viditelnosti použita. Nicméně v případě orientace zobrazovacího systému např. pro zobrazování městské zástavby je možné tuto techniku do systému snadno implementovat, aniž by bylo nutné nějak zásadně zasahovat do zbývajících částí systému.

Další možnost rozšíření stávajícího řešení určování viditelnosti lze nalézt např. v lepším shlukování jednotlivých instancí modelů v rámci jedné dlaždice ve scéně. V aktuální implementaci je pro každou dlaždici vygenerovaný pouze jeden ohraničující kvádr, přičemž se předpokládá, že velikost dlaždic bude zvolena optimálně tvůrcem scény. V budoucích vylepšeních by tedy bylo možné uvažovat o možnosti generovat pro jednu dlaždici např. hierarchii ohraničujících kvádrů založenou např. na shlukování instancí modelů uvnitř dlaždice pomocí optimálně zvoleného shlukovacího algoritmu.

Problematiku řešení viditelnosti lze samozřejmě vylepšovat i dalšími metodami, které však již přesahují rámec této práce. Lze sem zařadit např. portálové algoritmy vhodné především pro prostředí uvnitř budov, které lze výhodně kombinovat např. s algoritmy pracujícími s potenciálně viditelnými množinami objektů (viz např. [3]). V dnešní době výkonného grafického hardwaru je také vhodné uvažovat o určování viditelnosti pomocí modelových uzávěrů implementovaném přímo v grafickém hardwaru (viz např. [7]).

5.3 Správa datových zdrojů

V kapitole 5.1 byla představena základní organizace scény pomocí dlaždicové struktury obsahující instance modelů. Dále bylo zjednodušeně předpokládáno, že samotné modely typu `SceneModel` odkazované z jednotlivých instancí modelů obsahují prozatím pouze načtený soubor s popisem samotného modelu. Tedy paměťová náročnost takto strukturované reprezentace scény byla relativně nízká. Je však zřejmé, že ve skutečnosti musí model obsahovat další datové položky nutné pro vykreslování, jakými jsou především textury a geometrie modelů[20]. Takové datové položky již mají mnohem větší paměťové nároky a i přesto, že jsme eliminovali vícenásobné načítaní jednotlivých modelů pomocí schématu instance-model-manažer, v rozsáhlých scénách bychom jistě brzy vyčerpali dostupnou paměť systému. Ještě větší problém s nedostatkem paměti se pak samozřejmě dá očekávat

[20] Více se těmito datovými položkami budeme zabývat v kapitole 6.2.1, prozatím pouze budeme předpokládat jejich existenci v rámci objektu modelu.

v případě grafického subsystému, jehož paměťové prostředky bývají mnohem menší než běžně dostupná operační paměť systému. Z těchto důvodů je nutné do celkového systému pro management scény zahrnout subsystém pro správu datových zdrojů, který bude za běhu aplikace efektivně postupně načítat potřebné datové zdroje. Takový systém byl použit i v aktuální implementaci a skládá se ze dvou hlavních částí – efektivní správy datových zdrojů a postupného načítání obsahu scény. Obě tyto části budou popsány v následujícím textu.

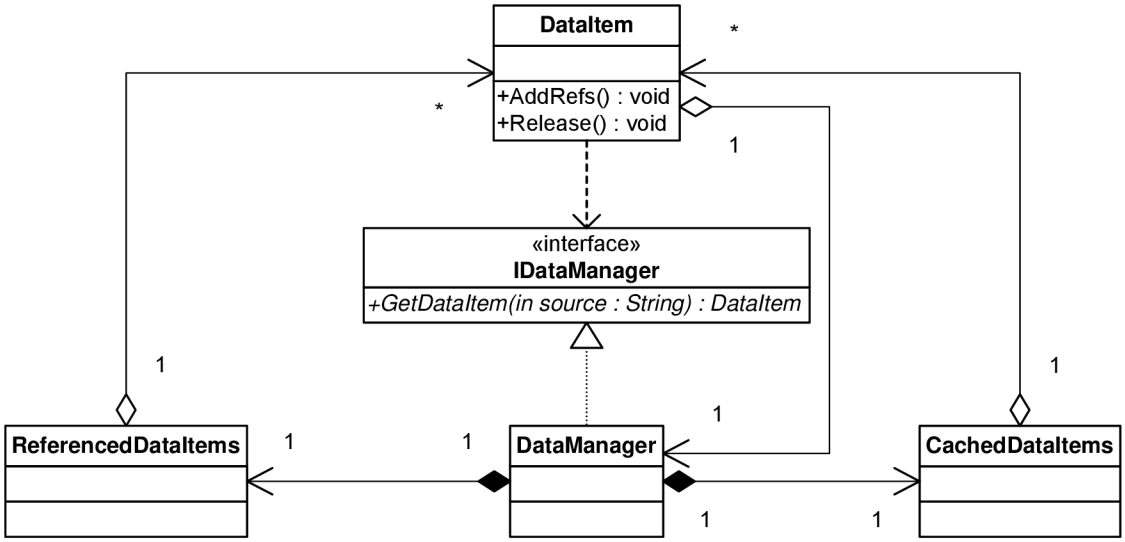
5.3.1 Efektivní správa datových zdrojů

Za datové zdroje jsou v aktuální implementaci považovány textury, geometrie modelů a zkompilované objekty programovatelných shaderů. Všechny tyto zdroje jsou považovány za paměťově náročné[21] a tedy je vhodné vytvořit systém, který bude tyto prvky efektivně spravovat.

Základním požadavkem takového systému je především zabránit redundanci při načítání datových zdrojů. Například po načtení textury ze souboru **A** se předpokládá, že další požadavek na texturu z tohoto souboru již nebude texturu znovu načítat, ale vrátí referenci na texturu načtenou dříve. Dalším požadavkem systému pro efektivní správu datových zdrojů je možnost využití vyrovnávací paměti pro zdroje, které sice nejsou aktuálně potřebné, ale v minulosti byly použity a vzhledem k aktuálnímu dostatku paměti a k tomu, že vytvoření datového zdroje je relativně náročná operace[22], je tyto prvky možné udržovat ve vyrovnávací paměti, přičemž mohou být v případě nedostatku paměti kdykoliv odstraněny.

Celý systém pro efektivní správu datových zdrojů je v aktuální implementaci založen na počítání referencí (viz např. [22]) na příslušné objekty, což zaručuje, že daný datový zdroj bude automaticky uvolněn v případě, že počet referencí klesne na nulu. V systému jsou k dispozici tři samostatné manažery datových zdrojů (pro textury, geometrii modelů a shadery), které realizují svojí činnost podle konceptuálního diagramu tříd na obrázku 5.7.

[21] Jedinou výjimku tvoří programovatelné shadery, které jsou sice paměťově nenáročné, ale z hlediska GPU je vhodné udržovat jejich počet minimální a vyhnout se jakékoli redundanci.
[22] To platí především v případě, kdy jsou data uložena na médiích s pomalým přístupem, jakými jsou např. optické disky typu CD nebo DVD.



Obr. 5.7 – schématické vyjádření manažeru pro správu datových zdrojů

Manažer příslušného datového zdroje `DataManager` jednoduše spravuje odděleně právě používané prvky pomocí objektu třídy `ReferencedDataItems`. V objektu třídy `CachedDataItems`, který představuje vyrovnávací paměť, jsou potom umístěné datové zdroje, které právě nejsou v aplikaci používány a mohou být kdykoliv odstraněny za účelem rychlého uvolnění paměti.

V případě požadavku na datový zdroj je nejprve prohledán objekt třídy `ReferencedDataItems` a pokud je v něm daný prvek nalezen, pak je mu zvýšen počet referencí a nová reference je vrácena aplikaci. Pokud není prvek nalezen v objektu třídy `ReferencedDataItems`, pak se prohledává objekt třídy `CachedDataItems`. V případě, že je v tomto objektu hledaný prvek nalezen, je zvýšen počet referencí na tento prvek, prvek je následně vyjmut z objektu třídy `CachedDataItems` a přesunut do objektu třídy `ReferencedDataItems`. Pokud není požadovaný datový zdroj nalezen ani v objektu třídy `CachedDataItems`, pak je nutné tento datový zdroj načíst přímo ze souboru na disku. Ještě předtím se však ověří, že aktuální velikost datových zdrojů spravovaných příslušným manažerem nepřesahuje povolenou hranici. Pokud ano, pak se vždy musí odstranit tolik prvků z objektu třídy `CachedDataItems`, aby byl splněn daný paměťový limit[23]. Potom je již možné načtený datový zdroj uložit do objektu třídy `ReferencedDataItems` a novou referenci vrátit aplikaci.

Aplikace používá datové zdroje potřebným způsobem a v okamžiku, kdy daný datový zdroj dále nepotřebuje, zavolá metodu pro uvolnění držené reference (`DataItem::Release`), čímž dojde k uvolnění reference na tento objekt. Objekt sám zkontroluje zbývající počet referencí a pokud je nulový, pak pomocí vazby na manažer provede přeřazení sebe sama v rámci manažeru z objektu třídy `ReferencedDataItems` do objektu třídy `CachedDataItems`.

[23] Odstranění všech prvků z objektu třídy `CachedDataItems` samozřejmě nezaručuje, že bude splněn daný paměťový limit daného manažeru. To je spíše otázkou správné tvorby obsahu scény, která musí být po stránce paměťové náročnosti optimálně vyvážena.

Zde popsaný způsob efektivní správy datových zdrojů je velmi dobře použitelný pro celý zobrazovací systém a v kombinaci s postupným načítáním obsahu scény popsaným v následující kapitole vytváří velmi mocný nástroj pro správu datových zdrojů zobrazovacího systému.

5.3.2 Postupné načítání obsahu scény

Organizace scény zavedená v kapitole 5.1 prozatím předpokládala, že zobrazovací systém má načtené všechny dlaždice se všemi jejich instancemi modelů. Je však logické, že ve velmi rozsáhlých scénách by bylo velmi neefektivní udržovat v paměti načtené všechny instance modelů s jejich referencemi na příslušné modely. Z kapitoly 5.1 víme, že instance modelů reprezentované třídou `SceneModelInstance` jsou paměťově relativně nenáročné a proto se o ně po načtení není nutné zvláště zajímat. Větší problém však působí modely typu `SceneModel` odkazované právě z instancí modelů. Ty totiž mohou obsahovat odkazy na paměťově rozsáhlé datové zdroje[24] spravované způsobem popsaným v kapitole 5.3.1. Proto byl v implementaci managementu scény zaveden systém pro postupné načítání obsahu scény[25], který umožňuje udržovat v paměti načtené pouze aktuálně potřebné části scény.

Tento systém je stejně jako velká část systému managementu scény založen na dělení scény na dlaždice popsaném v kapitole 5.1. Z této kapitoly víme, že po načtení scény je každé dlaždici přiřazen seznam instancí modelů k ní patřících (které obsahují vlastní ohraničující kvádry) a z nich je následně spočítaný ohraničující kvádr určující viditelnost celé dlaždice, který podmiňuje viditelnost všech instancí modelů v dlaždici obsažených. Pro určování viditelnosti jednotlivých částí scény tedy nejsou vůbec potřebné reference na modely uložené uvnitř instancí modelů. Z těchto faktů vyplývá základní princip systému pro postupné načítání obsahu scény.

Každá dlaždice může být označená buď jako načtená v paměti nebo naopak jako neinicializovaná, k čemuž slouží booleovský příznak nově přidaný do každé dlaždice. Dále je ke každé instanci modelu typu `SceneModelInstance` přidán booleovský příznak, který této instanci modelu určuje, zda vlastní platnou referenci na model. Třída `SceneModelInstance` je dále doplněna o možnost uvolnit referenci na její model a zase ji znovu načíst.

Na počátku ihned po načtení scény jsou všechny dlaždice označené jako neinicializované a ze všech instancí modelů jsou uvolněny reference na modely. Při procházení scénou se pak postupuje následovně. Je-li dlaždice detekována jako viditelná a zároveň je označena jako neinicializovaná, pak to znamená, že tato dlaždice předtím explicitně nenačetla modely pro své instance modelů. V takovém případě systém vyvolá metodu pro načtení této dlaždice, která způsobí, že pro každou instanci modelu v dlaždici bude načten její model. Po tomto kroku je dlaždici změněn příznak

[24] Stejně jako v kapitole 5.1, i zde zatím nebudeme konkrétně popisovat strukturu třídy `SceneModel`, pouze budeme předpokládat, že obsahuje paměťově náročné datové zdroje. Podrobnosti k této struktuře budou uvedeny v kapitole 6.2.1.

[25] V anglické literatuře se toto nazývá běžně termínem „data streaming“.

určující, že dlaždice je již načtená v paměti, a dlaždice je uložena do vyrovnávací paměti načtených dlaždic. Tímto způsobem je zajištěno, že po detekci viditelných instancí modelů obsahují tyto všechny načtený model a s viditelnými instancemi modelů je tedy možné dále pracovat (viz kapitola 7 o vykreslování). Po dokončení práce s viditelnými instancemi modelů (tedy typicky po dokončení vykreslování aktuálního snímku) systém provede úklid aktuálně načtených dlaždic ve vyrovnávací paměti. Ten probíhá jednoduchým způsobem, kdy má každá dlaždice nastavenu časovou značku[26], která určuje, kdy byla dlaždice naposledy použita, a systém má zároveň nastaven maximální možný počet dlaždic uložených ve vyrovnávací paměti. Pokud je tento počet překročen, pak se aktivuje mechanismus, který z vyrovnávací paměti dlaždic postupně odstraňuje dlaždice (počínaje tou s nejstarší časovou značkou) a při tomto procesu zároveň uvolňuje reference na modely z příslušných instancí modelů uvnitř odstraňovaných dlaždic. Tímto mechanismem je tedy zajištěno, že při průchodu scénou budou v paměti načtené skutečně jen prvky z lokálního okolí pozorovatele. Při implementaci však bylo nutno vyřešit několik zásadních problému celého systému pro postupné načítání obsahu scény.

Velkým problémem je především nové získaní reference na model v rámci opětovného načtení instance modelu. Zde může dojít k situaci, kdy soubor se zdrojovými daty modelu např. nemůže být nalezen a tedy se načítání modelu nezdaří. Tuto situaci není možné řešit žádným užitečným způsobem, aplikace však přesto musí pokračovat v běhu. Proto se ve stávající implementaci systému předpokládá, že ještě před vlastním zobrazením vykreslovací systém zkontroluje, zda je skutečně model v dané instanci modelu dostupný (více viz kapitola 7.2 o vykreslování).

Další velký problém celého mechanismu je patrný z faktu, že daná instance modelu může být sdílena více dlaždicemi, což v implementaci znamená, že více dlaždic sdílí jeden paměťový objekt pomocí reference na něj. To bohužel v praxi vytváří situaci, kdy např. dlaždice **A** a **B** obsahují stejnou instanci modelu **IM**, přičemž obě dlaždice jsou plně načtené v paměti. Kdykoliv se však může stát, že v rámci úklidu vyrovnávací paměti dlaždic dojde k odklizení dlaždice **B** a tedy k uvolnění všech modelů z instancí modelů v této dlaždici **B**, tedy včetně instance modelu **IM**. To ale znamená, že stejná instance modelu **IM** bude v ten okamžik bez platné reference na model i v dlaždici **A**, která je stále načtená ve vyrovnávací paměti a stejně tak i označená. Zabránit této situaci lze více způsoby, např. by bylo možné v každé instanci modelu udržovat seznam vazeb na dlaždice, ve kterých je instance modelu obsažena, nebo by bylo možné použít opět mechanismus počítání referencí aplikovaný i na instance modelů. Takové techniky jsou však poměrně zbytečně komplikované i vzhledem k faktu, že k popsané situaci nedochází příliš často. Je proto lepší využít pro řešení tohoto problému poměrně jednoduchou techniku, která kontroluje platnost modelů v rámci instancí modelů až v okamžiku, kdy je to skutečně nutné. Tento okamžik následuje ihned po určení, že daná instance modelu je skutečně viditelná (jak víme z kapitoly 5.2, model pro určování viditelnosti instance

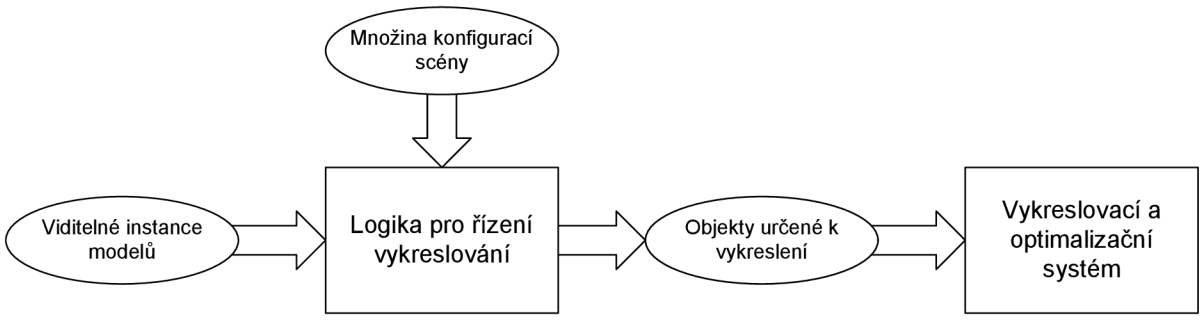
[26] Tato časová značka je ve skutečnosti 32-bitové neznaménkové číslo shodné s globální značkou používanou při určování viditelnosti – viz kapitola 5.2.

modelu není potřeba, takže je to možné). Tím pádem kontrolujeme platnost modelů v instancích modelů pouze v okamžiku, kdy danou instanci modelu skutečně budeme používat. Zde by však mohla vzniknout otázka, zda je tedy vůbec nutné provádět načítání modelů k jejich instancím pomocí dlaždic, když by celý mechanismus mohl fungovat jednoduše tak, že by se modely vždy načítaly k jejich instancím až poté, co by jejich instance byla určena jako viditelná. Takový mechanismus by jistě fungoval, ale nebylo by efektivně vyřešeno odstraňování modelů z těch instancí, které již nejsou aktivně používané. Přesněji řečeno, systém by nevěděl, které dlaždice má uklidit a tím pádem by se velmi zkomplikoval proces úklidu nepotřebných částí scény. Právě proto byl zvolen systém postupného načítání obsahu založený na dlaždicích, který spíše než aby pomáhal určovat, jaká část scény je právě načtená, pomáhá velmi jednoduše určovat, jakou část scény je možné (a nutné) odstranit z paměti.

K celé implementaci systému pro postupné načítání obsahu je ještě vhodné dodat, že díky využívání globální značky generované při určování viditelnosti jako časové značky dlaždic a díky možnosti provádět test viditelnosti vícenásobně během jednoho vykreslovaného snímku, je nutné vyčkat s úklidem vyrovnávací paměti dlaždic skutečně až do okamžiku, kdy je celý snímek vykreslen. V opačném případě by hrozilo, že budou instance modelů načtené v prvním testu viditelnosti v následném testu viditelnosti prováděném v jiné části scény odstraněny, což by se v scéně projevilo chybějícími objekty nebo grafickými artefakty způsobenými chybějícími objekty (např. chybnou aplikací stínů vrhaných do scény apod.).

6 Základy vykreslovacího systému

V této kapitole se budeme zabývat základy vlastního vykreslovacího systému, který spolupracuje se systémem managementu scény popsaném v kapitole 5. Přesněji řečeno, vstupem vykreslovacího systému je množina instancí modelů určených jako viditelné (pomocí postupu popsaného v kapitole 5.2) a cílem vykreslovacího systému je všechny tyto instance modelů správně zobrazit. Z toho tedy vyplývá, že vykreslovací systém je od systému managementu scény vysoce oddělen, neboť je spojuje pouze množina instancí modelů. Můžeme tedy říci, že se nám podařilo maximalizovat oddělení těchto dvou hlavních subsystému v rámci celého zobrazovacího systému, což byl také jeden z hlavních požadavků na celý zobrazovací systém. Zjednodušené schéma implementovaného a zde popsaného řešení vykreslovacího systému je na obrázku 6.1.



Obr. 6.1 – zjednodušené schéma vykreslovacího systému

Na tomto obrázku je vidět, že vstupem celého vykreslovacího systému je množina viditelných instancí modelů získaných pomocí postupu popsaného v kapitole 5.2. Dále do systému vstupuje množina tzv. konfigurací scény (budou popsány dále v této kapitole). Z těchto dvou množin poté logika pro řízení vykreslování musí získat skupinu objektů[27], které už jsou přímo vhodné pro vykreslování a ty následně poskytnout subsystému, který zajišťuje samotné vykreslování a to pokud možno optimálním způsobem. My se nyní budeme zabývat jednotlivými částmi naznačenými na uvedeném obrázku 6.1 s výjimkou samotného vykreslovacího a optimalizačního systému, který bude rozebrán v samostatné kapitole.

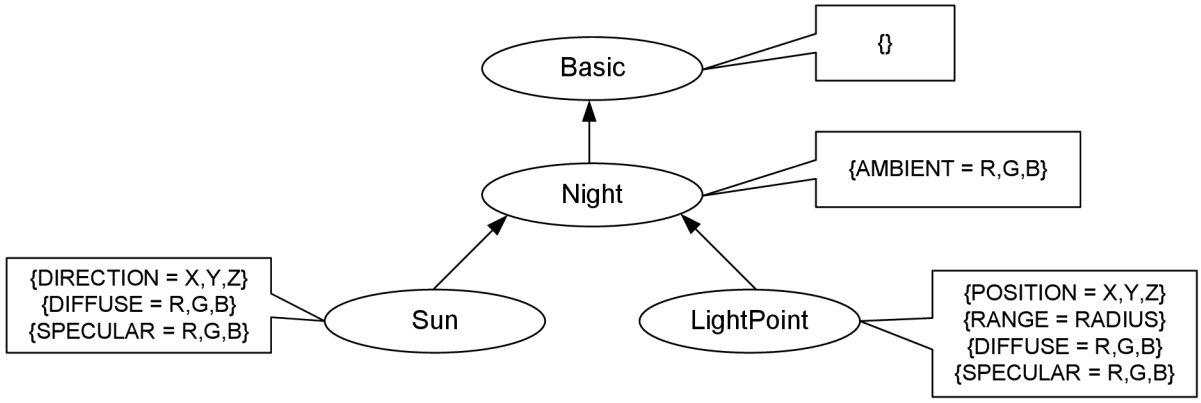
Ještě předtím si však ujasníme několik pojmů, které zde budou používány. Z kapitoly 3.3 víme, že každý soubor s popisem modelu obsahuje množinu konfigurací modelu, které určují, jak se má model zobrazit v určitých lokálních podmínkách scény. Toto tvrzení však není zcela přesné, protože některé konfigurace modelu v rámci souboru s jeho popisem nejsou určeny přímo k zobrazení, ale obsahují další důležité informace. Příkladem takové konfigurace modelu je např. konfigurace `[bounding-box]`, obsahující informaci o ohraničujícím kvádru tohoto modelu. Takové konfigurace nás však v tuto chvíli nebudou zajímat a zaměříme se pouze na ty, které se týkají zobrazení daného modelu v určitých lokálních podmínkách scény. Tyto konfigurace budeme dále

[27] Tyto objekty budou specifikovány dále v této kapitole.

nazývat „zobrazovací konfigurace modelu“ a lokální podmínky scény, ve kterých se příslušné zobrazovací konfigurace modelu budou aplikovat, budeme nazývat „konfigurace scény“. Konfiguracím scény se nyní budeme podrobně věnovat.

6.1 Konfigurace scény

Konfigurace scény představuje jistou množinu parametrů, které ovlivňují způsob zobrazení instance modelu v této konfiguraci scény. Pro představu může být takovým parametrem např. intenzita světelného zdroje, která určí, jak moc osvětleny budou instance modelů v okolí tohoto světelného zdroje. Základní verze implementace zobrazovacího systému by tedy mohla vypadat tak, že pro každou viditelnou instanci modelu nalezneme konfiguraci scény, ve které se instance modelu nachází, a aplikujeme parametry této konfigurace scény na model z této instance modelu, čímž bude instance modelu řádně zobrazena. Tato základní myšlenka však trpí několika nedostatky, přičemž ten úplně nejvážnější vychází z možnosti existence více konfigurací scény v rámci celé scény. To znamená, že instance modelu se může nacházet v dosahu více konfigurací scény a je proto nutné mezi nimi rozhodnout, která je pro vykreslení významnější. Ještě mnohem horší je situace, kdy je jedna konfigurace scény definována uvnitř jiné a například spolu sdílí určité parametry, pak je nutné rozhodnout, který z těchto společných parametrů vlastně použít. Z těchto důvodů bylo nutné pro konfigurace scény zavést pokročilejší způsob definice, který alespoň prozatím vyřeší představené problémy. Za takový způsob lze považovat v implementaci zvolenou hierarchii dědičnosti konfigurací scény, která umožní definovat jednotlivé konfigurace scény závisle na sobě podle potřeby. Pro názornost dále vysvětlovaných principů je na obrázku 6.2 prezentována hierarchie dědičnosti konfigurací scény, která je aktuálně implementována v zobrazovacím systému.



Obr. 6.2 – hierarchie dědičnosti konfigurací scény z aktuální implementace

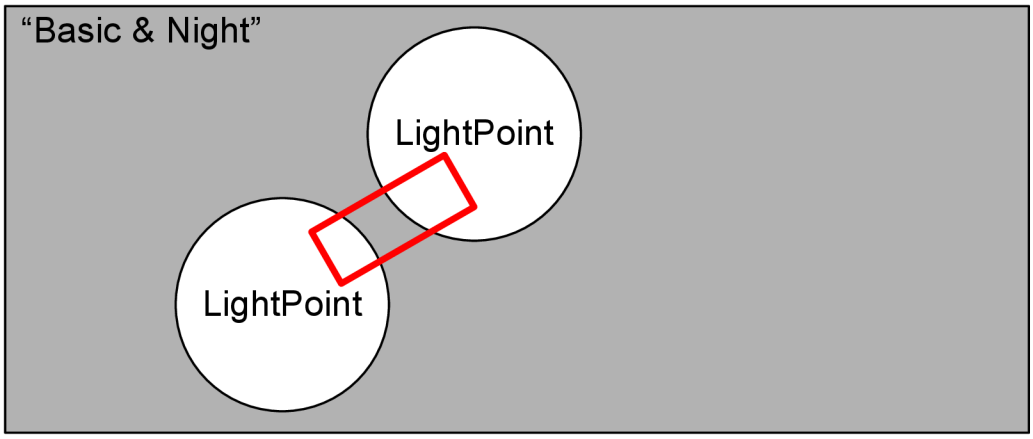
Zde můžeme vidět, že úplně základní konfiguraci scény v celém systému představuje konfigurace scény označená jako **Basic**. Ta neobsahuje žádné vlastní parametry, čímž umožňuje odvodit libovolnou další konfiguraci scény od této základní. Od této základní konfigurace scény jsou potom odvozovány další konfigurace scény, které postupně přidávají další parametry. V rámci této dědičnosti je důležité si uvědomit dvě základní fakta. Tím prvním vyplývajícím z dědičnosti

konfigurací scény je, že libovolná konfigurace scény na nižší úrovni hierarchie grafu dědičnosti obsahuje všechny parametry předchozích konfigurací scény. To je důležitý předpoklad pro budoucí vykreslování samotných instancí modelů (viz kapitola 6). Druhý pak říká, že dědičnost konfigurací scény není v implementaci nijak závislá na dědičnosti používané v implementačním jazyce[28], což přináší výhody především při samotné manipulaci s konfiguracemi scény a jejich parametry. Konkrétně je implementace jednotlivých konfigurací scény založena na bázové abstraktní třídě `SceneConfig`, od které jsou jednotlivé konfigurace scény odvozeny. Třída `SceneConfig` pak zavádí abstraktní metody `IsCompatible` a `GetCompatible`, které pomáhají určit, příp. získat jinou konfiguraci scény z aktuální konfigurace scény podle toho zda to je možné či nikoliv.

Máme tedy definovánu základní hierarchii dědičnosti konfigurací scény, ale zbývá ještě dále upřesnit její význam. Již bylo zmíněno, že konfigurace scény na nižších úrovních sdílí parametry se svými předky. To např. v případě konfigurace scény **LightPoint** znamená, že obsahuje parametr **AMBIENT**, určující intenzitu všudypřítomného světla, převzatý z konfigurace scény **Night**. Představme si na chvíli scénu, ve které je umístěno bodové světlo, které ale v jistém okamžiku zhasne. Je zřejmé, že na místě tohoto zhasnutého světla není prázdno, ale nachází se zde ještě všudypřítomné světlo. To platí i v rámci naší hierarchie dědičnosti. Z tohoto jednoduchého příkladu tak vyplývá další vlastnost hierarchie dědičnosti konfigurací, která předpokládá, že libovolná konfigurace scény vlastně určuje prostor scény (tedy objemové těleso), ve kterém parametry dané konfigurace scény platí. Na základě dědičnosti jednotlivých parametrů proto v hierarchii dědičnosti konfigurací scény musí vždy platit, že konfigurace scény na nižší úrovni musí být prostorově vnořená uvnitř všech svých předků, přičemž takový požadavek lze samozřejmě vždy splnit, neboť se předpokládá, že minimálně konfigurace typu **Basic** je prostorově nekonečná a tedy libovolná konfigurace odvozená přímo od ní je v ní prostorově vnořená. V námi uvedeném příkladě hierarchie konkrétně platí, že konfigurace scény **Basic**, **Night** a **Sun** jsou prostorově nekonečné a tedy splňují podmínku vnoření hierarchie dědičnosti. Konfigurace scény **LightPoint**, představující bodový světelný zdroj, je prostorově omezená umístěním a dosahem světelného zdroje, což lze vyjádřit koulí umístěnou v prostoru, která vždy splňuje podmínku vnoření do konfigurace scény **Night**, která je prostorově nekonečná.

Prostorové vnoření konfigurací scény definované v předchozím odstavci však není poslední vlastností hierarchie dědičnosti konfigurací scény. Představme si např. následující scénu znázorněnou na obrázku 6.3 a situaci v ní.

[28] Aktuální implementace řeší dědičností vztahem „má“ (angl. „has-a“) vyjadřujícím kompozici objektů, tedy ne klasicky vztahem „je“ (angl. „is-a“) vyjadřujícím dědičnost. Více např. v [8].



Obr. 6.3 – ovlivnění instance modelu více konfiguracemi scény

V této scéně je definována všudypřítomná konfigurace scény **Basic** a s ní zároveň konfigurace scény **Night**. Dále jsou definovány dvě lokální bodová světla, která tak vytváří dvě konfigurace scény **LightPoint**, obě správně prostorově vnořené. Červený obdélník představující instanci modelu tak v této situaci náleží celkem do čtyř konfigurací scény, přitom je ale logické, že obě konfigurace scény **LightPoint** obsahují parametry svých předků (**Basic** a **Night** – viz obrázek 6.2) a tedy při zobrazování dané instance modelu by se měly brát v úvahu pouze relevantní konfigurace scény. Z toho důvodu bylo nutné přidat do hierarchie dědičnosti mechanismus pro rozhodování, která z konfigurací scény je vlastně nejlépe použitelná. Logicky by se nabízelo využití acyklické grafové struktury vzniklé v rámci hierarchie dědičnosti konfigurací scény, bohužel tento přístup by mohl způsobit problémy v budoucím rozšiřování množiny konfigurací scény. Např. i v naší situaci (viz obrázek 6.3) není úplně jasné, jestli konfigurace scény **LightPoint** je při zobrazování použitelnější než konfigurace scény **Sun** ležící na stejné úrovni v grafu na obrázku 6.2. Z tohoto důvodu byla pro každou konfiguraci scény povinně přidána její priorita (reprezentovaná jednoduše celým číslem a v rámci třídy `SceneConfig` metodou `GetPriority`). Tato priorita umožňuje snadno rozlišit vícenásobný výskyt konfigurací scény pro vykreslovanou instanci modelu. Konkrétně je při vykreslování instance modelu nejprve určena množina všech konfigurací scény, které mohou danou instanci modelu ovlivnit. Z této množiny je následně vybrána konfigurace scény s nejvyšší prioritou, která bude použita pro vykreslení instance modelu. Ještě zbývá vyřešit otázku, co dělat v případě, kdy pro instanci modelu bude nalezeno více konfigurací scény se stejnou prioritou (viz situace na obrázku 6.3 se dvěma bodovými světly), touto situací se však budeme podrobně zabývat až v kapitole 7.2.

6.2 Základní entity logiky pro řízení vykreslování

Z obrázku 6.1 je patrné, že vstupem logiky pro řízení vykreslování je množina viditelných instancí modelů typu `SceneModelInstance` a dále množina konfigurací scény typu `SceneConfig` popsaných v minulé kapitole 6.1. Výstupem je pak množina zatím nespecifikovaných objektů, které budou použity pro vykreslování. Cílem logiky pro řízení vykreslování je tedy jistým způsobem

z instancí modelů a dostupných konfigurací scény vytvořit množinu objektů, které již bude snadné zobrazit a které budou zároveň poskytovat dostatek možností pro optimalizaci výsledného zobrazení.

Z entit používaných logikou pro řízení vykreslování jsme zatím popsali pouze instance modelů a konfigurace scény. Přitom víme, že instance modelu obsahuje referenci na model typu `SceneModel`, o kterém jsme však doposud předpokládali pouze to, že obsahuje načtený soubor s popisem modelu a zatím blíže neurčené grafické datové zdroje. Nyní je tedy vhodná příležitost definovat strukturu datového typu `SceneModel`. Dále je také nutné konkrétně specifikovat objekty, které jsou výstupem logiky pro řízení vykreslování a slouží jako základní jednotky pro proces vykreslování.

6.2.1 Struktura modelu

Jednoduše řečeno je struktura `SceneModel` v rámci celého systému zodpovědná za vykreslování jednotlivých instancí modelů a také za správu grafických datových zdrojů. Jak již ale víme, vykreslování je podmíněno konfigurací scény (nebo více konfiguracemi scény), ke které daná instance modelu náleží. Víme také, že k tomu aby model mohl být správně vykreslen v příslušné konfiguraci scény, musí mít ve svém popisu definovanou příslušnou konfiguraci modelu, která je v případě použití pro zobrazení v rámci konfigurace scény označována jako zobrazovací konfigurace modelu. Každý model tedy obsahuje pro každou konfiguraci scény požadovanou zobrazovací konfiguraci modelu. Ještě předtím než začneme definovat datový typ `SceneModel`, upřesníme popis každé zobrazovací konfigurace modelu v rámci souboru s popisem tohoto modelu. Její základní tvar je vždy následující:

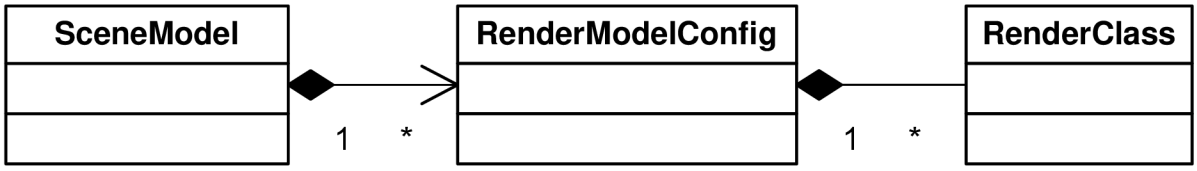
```
[%jméno zobrazovací konfigurace modelu%] {
      ; zde následuje libovolné množství podsekcí [lod] tohoto tvaru
      [lod]{
            "lod_start" = "%vzdálenost%"
            "class" = "%vykreslovací třída%"

            ;další pametry specifické pro danou vykreslovací třídu
      }
}
```

Každá zobrazovací konfigurace modelu má tedy své jméno, které je vždy shodné se jménem příslušné konfigurace scény. Struktura každé zobrazovací konfigurace modelu je pak pro všechny typy identická, obsahuje pouze seznam podsekcí `[lod]`. Libovolné jiné podsekce nebo atributy jsou jednoduše ignorovány. Jak je již z názvu patrné[29], představuje podsekce `[lod]` informace o jedné úrovni detailu pro danou zobrazovací konfiguraci modelu. To tedy znamená, že pro každou

[29] Zkratka LOD se běžně používá pro úroveň detailu, z angl. „Level-of-Detail“.

zobrazovací konfiguraci modelu můžeme specifikovat libovolný počet úrovní detailů. Každá úroveň detailu je jednoznačně určena atributem `lod_start`, který udává vzdálenost od pozorovatele, od které se daná úroveň detailu aplikuje. Systém sám všechny úrovně detailu načte a seřadí podle vzdálenosti, takže žádné další určování úrovní detailu není potřeba.

Mnohem důležitějším atributem v rámci každé sekce `[lod]` je však parametr `class`. Ten pro danou úroveň detailu určuje, jaká vykreslovací třída[30] bude použita pro vykreslení této úrovně detailu. Použitý atribut `class` také zároveň určuje, jaké další parametry jsou očekávány v aktuální sekci `[lod]`. To tedy znamená, že sekce `[lod]` může obsahovat libovolné další atributy v závislosti na použité vykreslovací třídě. Nyní by bylo vhodné definovat, co vlastně zobrazovací třída určená atributem `class` představuje. Nicméně pro lepší souvislost textu ji budeme definovat až v následující kapitole, nyní se spokojíme s tím, že vykreslovací třída je v rámci implementace představována abstraktní třídou `RenderClass`, od které se odvozují všechny konkrétní vykreslovací třídy. Třída `RenderClass` pak představuje důležitou součást výstupních objektů z logiky pro řízení vykreslování znázorněných na obrázku 6.1. Nyní již tedy můžeme definovat strukturu datového typu `SceneModel` diagramem na následujícím obrázku 6.4.



Obr. 6.4 – struktura datového typu SceneModel

Každý model typu `SceneModel` tedy obsahuje pro každou zobrazovací konfiguraci modelu reprezentovanou strukturou `RenderModelConfig` množinu vykreslovacích tříd reprezentovaných datovým typem `RenderClass`. Struktura `RenderModelConfig` tedy zjednodušeně definuje kolekci vykreslovacích tříd typu `RenderClass`, které představují jednotlivé úrovně detailu pro zobrazovací konfigurace modelu, přičemž každá tato kolekce je nepovinná a závisí plně na tom, zda byla v souboru s popisem definována příslušná zobrazovací konfigurace modelu.

6.2.2 Vykreslovací třída RenderClass

V předchozí kapitole jsme zavedli strukturu `SceneModel` s tím, že je složena z kolekcí vykreslovacích tříd typu `RenderClass`, kde každá kolekce představuje úrovně detailu pro jednu zobrazovací konfiguraci modelu. Nyní se zaměříme na vlastní popis třídy `RenderClass`, která je pro celý proces vykreslování velmi významná.

Tato třída slouží jako základní vykreslovaná jednotka v rámci celého vykreslovacího systému. Určuje způsob, jakým se má daný model zobrazit a je zároveň zodpovědná za správu grafických

[30] Pojem „vykreslovací třída“ zde nemá souvislost s třídami známými z objektově-orientovaného programování, pouze tak označujeme způsob vykreslování nějakého objektu – viz dále.

datových zdrojů nutných pro toto zobrazení. Tedy vlastní datový typ `SceneModel` vůbec nespravuje grafické datové zdroje, ale pouze jednotlivé vykreslovací třídy, které jsou za svoje grafické datové zdroje odpovědné. V této souvislosti je nutné upozornit na to, že jednotlivé vykreslovací třídy nejsou okamžitě přítomné v modelu po jeho načtení, ale naopak jsou vytvářeny podle potřeby, takže se není nutné obávat velké paměťové náročnosti tohoto řešení. Tomuto tématu se budeme podrobně věnovat až v kapitole 7.2, kde bude popsán kompletní proces zobrazení.

Z kapitoly 6.2.1 víme, že každá konkrétní vykreslovací třída je vždy povinně umístěna v rámci nějaké zobrazovací konfigurace modelu (a tedy konfigurace scény) a zároveň každá vykreslovací třída definuje svoje potřebné atributy v sekci `[lod]` souboru s popisem modelu v němž je obsažena. Vzhledem k tomu, že žádné další informace pro vykreslovací třídu nejsou potřebné, můžeme objekt dané vykreslovací třídy bez potíží vytvořit na základě jejího unikátního jména a všech potřebných parametrů určených atributy v příslušné sekci `[lod]`. K tomu můžeme použít dobře známý návrhový vzor „továrna“ (angl. „Factory“ – viz např. [9]), takže proces definice nové vykreslovací třídy a jejího vytvoření lze popsat takto:

1. Odvoď novou konkrétní vykreslovací třídu od abstraktní vykreslovací třídy `RenderClass` a specifikuj metodu pro vytváření instancí této nové třídy – metodu budeme nazývat `Create`.
2. Zaregistruj metodu `Create` z kroku č.1 do objektu reprezentujícího návrhový vzor „továrna“ se jménem příslušné vykreslovací třídy.
3. Při vytváření instance vykreslovací třídy získej její jméno z příslušné sekce `[lod]`.
4. Použij objekt reprezentující návrhový vzor „továrna“ a poskytni mu jméno vykreslovací třídy a celou `[lod]` sekci pro vytvoření požadované vykreslovací třídy. „Továrna“ podle jména vyhledá příslušnou metodu `Create` a předá ji celou sekci `[lod]` na základě které metoda `Create` vytvoří objekt vykreslovací třídy a vrátí aplikaci ukazatel na abstraktní vykreslovací třídu `RenderClass`. Metoda `Create` samozřejmě ví, jaké grafické datové zdroje příslušná vykreslovací třída potřebuje a podle potřeby je získá z datových manažerů.

Tímto postupem tedy lze dynamicky vytvářet vykreslovací třídy za běhu podle popisu daného modelu. Vykreslovací třída je tedy vždy zodpovědná za své správné vytvoření implementací své metody `Create`.

Pro ilustraci celého principu zde předvedeme proces vytvoření na vykreslovací třídě použité v aktuální implementaci. Jedná se o primitivní vykreslovací třídu, která jednoduše zobrazí model jako prostou geometrii s definovanou konstantní barvou. Její popis v souboru modelu je následující:

```
[basic] {
      [lod]{
            "lod_start" = "0"
            "class" = "basic_vt0_diffuse"
            "geom" = "model.geom"
            "color" = "1;0;0;1"
      }
}
```

Jak je vidět, soubor definuje zobrazovací konfiguraci modelu **Basic** a v ní jednu úroveň detailu. Ta začíná ve vzdálenosti 0 od pozorovatele a vyžaduje vytvoření vykreslovací třídy s názvem „**basic_vt0_diffuse**“. Tento název je tedy předán objektu továrny, který podle názvu vykreslovací třídy nalezne správnou metodu pro vytvoření konkrétního objektu. Ten vyhledá příslušnou metodu `Create`, které předá informace o celé sekci `[lod]`. Metoda `Create` pak očekává, že v předávaných parametrech nalezne požadované informace, kterými jsou zde název souboru s geometrií určenou pro vykreslení a barva použitá pro celý model. Pokud tyto informace obdrží správně, pak je objekt požadované vykreslovací třídy vytvořen a vrácen aplikaci. V opačném případě k vytvoření nedojde a aplikace na to musí reagovat.

Nyní je tedy jasné, jak lze snadno vytvořit podle popisu v souboru modelu příslušnou vykreslovací třídu za pomoci návrhového vzoru „továrna“. Zatím však nebylo definováno, jakým způsobem vlastně vykreslovací třída může vykreslovat instance modelů a zároveň nebyla popsána souvislost mezi vykreslovacími třídami a konfiguracemi scény. Právě této souvislosti se budeme věnovat v následující velmi podstatné kapitole.

6.2.3 Spolupráce vykreslovacích tříd a konfigurací scény při vykreslování

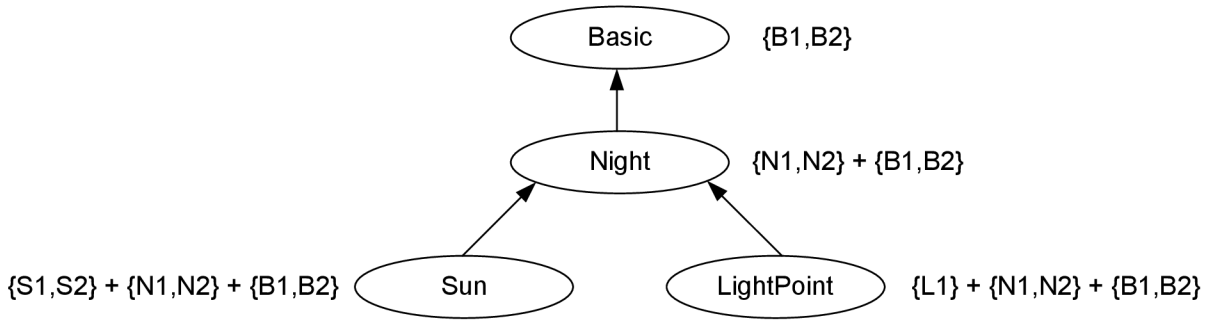
V kapitole 6.2.2 jsme začali s definicí vykreslovací třídy `RenderClass`, přičemž byl popsán princip jejího vytvoření i to, že je zodpovědná za grafické datové zdroje. Zatím však nebylo popsáno, jakým způsobem je tato třída použita při vykreslování, takže se nyní budeme věnovat tomuto tématu, které zároveň zahrnuje souvislost mezi vykreslovacími třídami a konfiguracemi scény.

Jak již víme, vykreslovací třída se vždy povinně nachází uvnitř nějaké `[lod]` sekce v souboru s popisem modelu, přičemž tato `[lod]` sekce je vždy součástí nějaké zobrazovací konfigurace modelu, která je vždy svázána s konfigurací scény stejného jména. Tato konfigurace scény poskytuje určité parametry, které se používají při vykreslování a tedy by měly být přístupné vykreslovací třídě. Z toho také vyplývá základní mechanismus vykreslovaní dané vykreslovací třídy. Ta jednoduše definuje metodu `Render`, která přebírá jako parametr objekt konfigurace scény typu `SceneConfig`, ve které je definována a z níž může následně číst požadované parametry. Je ale

zřejmé, že tento jeden parametr pro vykreslení nestačí. Vykreslovací třída je totiž vždy součástí nějakého modelu typu `SceneModel` (viz kapitola 6.2.1), který ale nemá žádné informace o právě vykreslované instanci modelu, stejně tak tedy tyto informace nemá ani vykreslovací třída. Proto je nutné metodu `Render` doplnit dalším parametrem a tím je objekt instance modelu typu `SceneModelInstance`, který již poskytuje všechny potřebné informace pro správné vykreslení dané instance modelu. Metoda `Render` vykreslovací třídy tedy vždy přebírá dva parametry – instanci modelu, jíž se vykreslování týká, a konfiguraci scény, v níž se instance modelu nachází. Z těchto informací už musí být implementace konkrétní vykreslovací třídy schopna provést správné vykreslení dané instance modelu. Z tohoto popisuje je tedy zřejmé, že vykreslovací třída se v podstatě chová podle klasického předpisu definovaného návrhovým vzorem „příkaz“ (viz např. [9]), tj. po vytvoření vykreslovací třídy už nejsou jednotlivé konkrétní typy vykreslovacích tříd rozlišovány a odpovědnost za vykreslení je plně v režii vykreslovací třídy a její metody `Render`.

S ohledem na metodu `Render` vykreslovací třídy `RenderClass` však zbývá vyřešit ještě jeden problém. Předchozí odstavec definoval metodu `Render` pro vykreslovací třídu, která předpokládá jako parametry instanci modelu a konfiguraci scény. Konkrétní vykreslovací třída pak tyto parametry musí použít pro správné zobrazení daného modelu. Konfigurace scény definované v kapitole 6.1 vytvářejí hierarchii dědičnosti, ze které vyplývá dědičnost jejich jednotlivých parametrů. Pokud se zamyslíme nad definicí vykreslovacích tříd v kapitole 6.2.2, zjistíme, že některé vykreslovací třídy mohou mít takové požadavky, které v dané konfiguraci scény jednoduše nejsou dostupné. Např. může existovat vykreslovací třída zobrazující modely ve scéně osvětlené bodovým světlem, taková vykreslovací třída tedy jistě potřebuje pro správné zobrazení pozici bodového světla. Je ale zřejmé, že ne každá konfigurace scény poskytuje tuto informaci. Konkrétně v naší ukázce na obrázku 6.2 poskytuje tuto informaci pouze konfigurace scény **LightPoint**. Je tedy logické, že pokud se objeví v jisté zobrazovací konfiguraci modelu taková vykreslovací třída, která v dané zobrazovací konfiguraci modelu nemůže získat dostatek informací, zobrazení modelu by nemělo vůbec proběhnout kvůli nedostatku informací.

Z těchto důvodů je nutné (stejně jako do konfigurací scény) přidat jistou hierarchii i do množiny vykreslovacích tříd. Vzhledem k hierarchii dědičnosti konfigurací scény se však tato úloha velmi zjednodušuje, neboť hierarchie vykreslovacích tříd tuto využije a to následujícím způsobem. Každá vykreslovací třída definuje minimální potřebnou konfiguraci scény pro své zobrazení. Vykreslovací třídu pak lze použít v této minimální konfiguraci scény a ve všech konfiguracích scény z ní odvozených. Jedná se tedy o klasickou vlastnost OOP, kdy potomek může zastoupit předka, ačkoliv jak již bylo zmíněno, v rámci konfigurací scény se dědičnost neřeší prostředky dědičnosti daného jazyka. Celá situace je pro názornost demonstrována na obrázku 6.5.



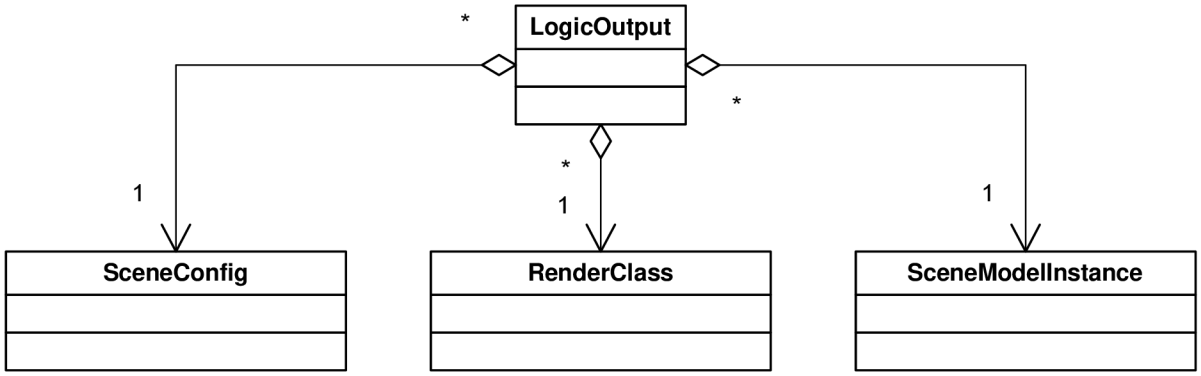
Obr. 6.5 – závislost vykreslovacích tříd a konfigurací scény

Zde jsou znázorněny jednotlivé konfigurace scény a k nim vykreslovací třídy, které s nimi lze použít. Např. vykreslovací třída **L1** může být použita pouze v konfiguraci scény **LightPoint**, protože ostatní konfigurace scény jí neposkytují všechny potřebné parametry. Naproti tomu vykreslovací třídy **B1** a **B2** mohou být využity v libovolné konfiguraci scény, neboť jsou definovány pro konfiguraci scény **Basic**, ze které povinně vychází všechny další konfigurace scény. Tímto jednoduchým mechanismem jsme tedy dokázali zajistit, že příslušná vykreslovací třída bude mít dostatek informací pro své správné zobrazení.

6.2.4 Výstupní objekty logiky pro řízení vykreslování

V předcházejících kapitolách jsme si popsali základy všech potřebných entit využívaných logikou pro řízení vykreslování. Stále však nebyly popsány objekty reprezentující výstup této části vykreslovacího systému, které jsou poslední neznámou entitou v celém systému a budou popsány právě v této kapitole. Poté již budeme moci přistoupit k popisu vlastního fungování logiky pro řízení vykreslování.

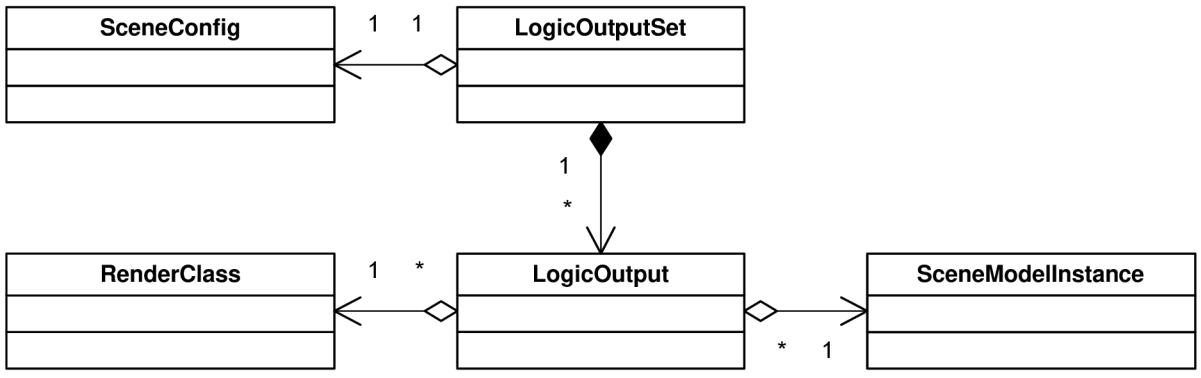
Při popisu vykreslovací třídy typu `RenderClass` jsme se zmínili o tom, že představuje de facto návrhový vzor „příkaz“, neboť po vytvoření instance konkrétní vykreslovací třídy již vykreslování necháváme plně v režii této vzniklé instance. Mohlo by se tedy zdát, že vhodným výstupním objektem logiky pro řízení vykreslování bude právě objekt vykreslovací třídy typu `RenderClass`. V kapitole 6.2.3 jsme se ale také zmínili o tom, že metoda `Render` vykreslovací třídy vyžaduje dva základní parametry – instanci modelu, které se vykreslování týká, a konfiguraci scény, v níž vykreslování probíhá. Z těchto znalostí tedy logicky vyplývá základní návrh složení výstupního objektu logiky pro řízení vykreslování. Jeho struktura je znázorněna konceptuálním diagramem na obrázku 6.6.



Obr. 6.6 – základní návrh struktury výstupních objektů z logiky pro řízení vykreslování

V tomto návrhu je tedy výstupní objekt reprezentovaný třídou `LogicOutput` reprezentován jako trojice (vykreslovací třída `RenderClass`, konfigurace scény `SceneConfig`, instance modelu `SceneModelInstance`). Zde by jistě mohla vzniknout otázka, proč vlastně není skutečně použita jako výstupní objekt samotná vykreslovací třída `RenderClass`, která by měla oba potřebné parametry uložené jako atributy. To by bohužel nebylo možné, protože nesmíme zapomínat, že samotný objekt vykreslovací třídy patří do modelu reprezentovaného třídou `SceneModel`, který ale může být využíván mnoha instancemi modelu typu `SceneModelInstance`. Z toho důvodu je nutné použít nový specializovaný objekt třídy `LogicOutput`.

Nicméně předchozí úvaha ukazuje na jiné možné vylepšení třídy `LogicOutput` reprezentující výstupní objekt, které se projeví především v příští kapitole. Uvažujme na chvíli o tom, že ve scéně existuje např. pouze jediná konfigurace scény typu **Basic**. Každý výstupní objekt typu `LogicOutput` tedy bude obsahovat vždy stejnou konfiguraci scény, což zavádí poměrně značnou redundanci. Na základě této úvahy tedy změníme podobu výstupního objektu tak, jak je prezentována v konceptuálním diagramu na obrázku 6.7.



Obr. 6.7 - optimální návrh struktury výstupních objektů z logiky pro řízení vykreslování

Nyní jsme tedy schéma změnili tak, že výstupními objekty logiky pro řízení vykreslování nebudou objekty typu `LogicOutput`, ale celé kolekce těchto objektů, reprezentované třídou `LogicOuputSet`. Přitom každá kolekce se vždy váže na jednu konkrétní konfiguraci scény typu `SceneConfig`, čímž jsme odstranili redundanci z předchozího návrhu. V dalším textu navíc

uvidíme, že uvedená reprezentace výstupních objektů má nesporné výhody při optimalizaci výsledného zobrazení.

6.3 Popis činnosti logiky pro řízení vykreslování

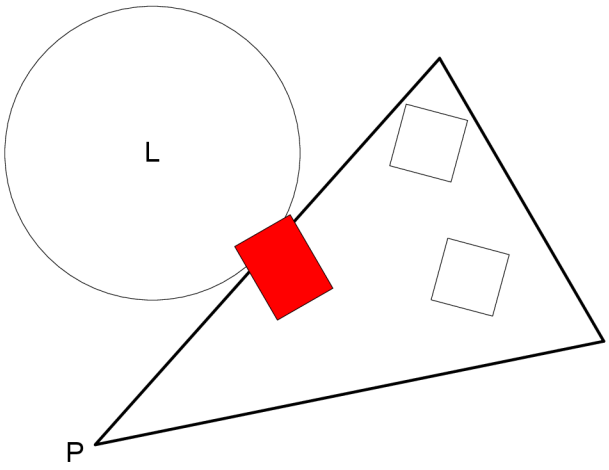
Jak již víme, základním úkolem logiky pro řízení vykreslování je transformovat vstupy reprezentované viditelnými instancemi modelů a množinou konfigurací scény na výstupní objekty typu `LogicOutputSet` popsané v předchozí kapitole. Tento proces bude podrobně popsán v této kapitole, přičemž zároveň s tím budeme postupně doplňovat dříve definované entity celého zobrazovacího systému.

6.3.1 Vyhledání použitelných konfigurací scény

Prvním důležitým krokem logiky pro řízení vykreslování je maximálně omezit počet konfigurací scény, které musí být uvažovány při vykreslování. V kapitole 6.1 s popisem konfigurací scény jsme se v souvislosti s konfiguracemi scény zmínili o jejich prostorovém významu, tj. že každá konfigurace scény reprezentuje nějaký prostor. Tím pádem je možné na konfigurace scény aplikovat techniky podobné těm pro detekci viditelnosti, které určí, zda je danou konfigurace scény nutné uvažovat. V souvislosti s tím je však nutné vyřešit dva základní problémy.

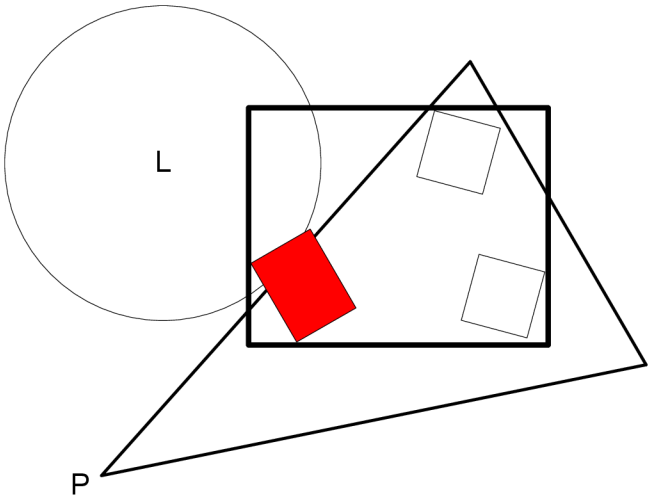
Tím prvním je fakt, že konfigurace scény může představovat libovolné prostorové těleso. Např. námi používaná konfigurace **LightPoint** je vždy představována v prostoru jako koule, zatímco konfigurace **Basic** představuje nekonečný prostor. Stejně tak by bylo možné vytvořit další konfigurace scény, které by mohly vytvářet libovolné další geometrické útvary. Proto bylo nutné vyřešit problém, jakým způsobem určit, zda daná konfigurace scény vůbec ovlivňuje viditelnou část scény. Z toho důvodu byla každé konfiguraci scény přidána sada metod na testování, zda dochází ke kolizi ze základními geometrickými útvary používanými v systému. Konfigurace scény tedy musí být vždy schopna určit, zda koliduje např. se zadaným kvádrem, koulí nebo pohledovou pyramidou. Výhodou tohoto přístupu je, že konfigurace scény tak může spolehlivě určit kolizi s aktuální viditelnou částí scény a logika pro řízení vykreslování se nemusí starat o složitost prostorového tělesa představovaného danou konfigurací scény. Názornou ukázkou může být např. konfigurace **Basic**, která je prostorově nekonečná a tedy na jakýkoliv dotaz ohledně kolize vždy odpoví kladně, tj. že ke kolizi došlo.

Druhým problémem při hledání použitelných konfigurací scény je pak fakt, že k nalezení těchto všech nestačí prohledávat pouze aktuálně viditelnou část scény. Celý problém je znázorněn na obrázku 6.8.



Obr. 6.8 – znázornění problému detekce použitelných konfigurací scény

Na něm pozorovatel v pozici **P** pozoruje scénu v níž je umístěno několik objektů, přičemž objekt reprezentovaný červeným obdélníkem je zcela určitě ovlivňován bodovým světlem **L**. Je ale zřejmé, že bodové světlo **L** reprezentované v prostoru koulí nekoliduje s aktuální pohledovou pyramidou a konfigurace scény vzniklá z tohoto bodového světla by tedy nebyla klasifikována jako použitelná, přestože pro správné osvětlení viditelného objektu je potřebná[31]. Proto je zřejmé, že oblast pro hledání použitelných konfigurací scény je nutné rozšířit tak, aby plně zahrnovala všechny aktuálně viditelné instance modelů. V aktuální implementaci byl pro toto rozšíření zvolen kvádr zarovnaný s osami souřadného systému a to především z důvodu urychlení celého systému. Z množiny viditelných instancí modelů lze takový kvádr získat velmi rychle hledáním maximálních a minimálních souřadnic podél jednotlivých os X,Y,Z. Navíc i následné testování na kolize tohoto kvádru s jednotlivými konfiguracemi scény je mnohem rychlejší než jejich testování s původní pohledovou pyramidou. Celá situace po vygenerování kvádru je znázorněna na obrázku 6.9.



Obr. 6.9 – správně vygenerovaný ohraničující kvádr pro nalezení použitelných konfigurací scény

[31] To vychází z toho, že vykreslování je prováděno po objektech a pokud bude objekt vykreslován např. pomocí Gouraudova stínování, pak dochází k interpolacím barev mezi viditelnými částmi objektu a těmi mimo pohledovou pyramidu. Pokud by objekt mimo pohledovou pyramidu nebyl osvětlen, pak by celý objekt nemohl být správně zobrazen.

Jak je vidět, výsledný prostor pro hledání použitelných konfigurací scény tedy nyní správně zahrnuje i konfiguraci scény vzniklou od bodového světla **L**. Vyřešili jsme tedy problém hledání použitelných konfigurací scény a můžeme přejít k vlastnímu procesu vytváření výstupních objektů typu `LogicOutputSet`.

6.3.2 Vytváření výstupních objektů typu LogicOutputSet

V předchozí kapitole jsme popsali postup pro nalezení všech použitelných konfigurací scény, vstupem celé logiky pro řízení vykreslování je pak množina všech viditelných instancí modelů. Z těchto prvků již nyní můžeme začít vytvářet výstupní objekty typu `LogicOutputSet` popsané v kapitole 6.2.4.

Proces vytváření výstupních objektů probíhá jednotlivě pro každou viditelnou instanci modelu. Ještě před spuštěním samotného algoritmu je pro každou použitelnou konfiguraci scény vytvořen výstupní objekt typu `LogicOutputScene`, k němuž je přiřazena reference na příslušnou konfiguraci scény. Poté již můžeme vytvářet výstupní objekty následujícím způsobem.

1. Procházej všechny instance modelů vstupující do logiky pro řízení vykreslování.
2. Pro každou instanci modelu nalezni všechny konfigurace scény z množiny konfigurací scény získaných podle postupu v kapitole 6.3.1, které aktuální instanci modelu ovlivňují.
3. Ze všech konfigurací scény získaných v kroku 2 ponech pouze ty s nejvyšší definovanou prioritou (viz kapitola 6.1).
4. Pro každou ponechanou konfiguraci scény nalezni v modelu typu `SceneModel` aktuální instance modelu potřebnou vykreslovací třídu s ohledem na úroveň detailu. Do objektu typu `LogicSceneOutput` odpovídajícímu aktuálně zpracovávané konfiguraci scény pak ulož nový objekt typu `LogicOutput`, který obsahuje referenci na aktuální instanci modelu a získanou vykreslovací třídu.

Algoritmus tedy jednoduše prochází jednotlivé instance modelů a postupně je přiřazuje odpovídajícím konfiguracím scény na základě priorit. Z algoritmu je také patrné, že instance modelu může být přiřazena do více konfigurací scény, což je ale v pořádku a bude to rozebráno podrobněji až v další kapitole zabývající se samotným zobrazením.

V uvedeném algoritmu je především důležité upozornit na fázi získávání vykreslovací třídy v kroku č.4. Již při představování struktury modelu typu `SceneModel` v kapitole 6.2.1 mohla vyvstat otázka, zda uložení všech vykreslovacích tříd pro všechny zobrazovací konfigurace modelu nebude příliš paměťově náročné. Odpověď by samozřejmě zněla, že ano. Na toto bylo naštěstí pamatována a proto byl datový typ `SceneModel` navržen tak, že ve skutečnosti nenačítá jednotlivé vykreslovací třídy přímo po svém vytvoření, ale až na vyžádání[32]. Po vytvoření příslušného objektu typu `SceneModel` jsou tedy načteny pouze potřebné informace pro vytváření jednotlivých

[32] Takový způsob práce se angl. nazývá „lazy evaluation“, neboli opožděné vyhodnocování.

vykreslovacích tříd, ale skutečná vykreslovací třída je vytvořena až v okamžiku, kdy o ní požádá logika pro řízení vykreslování. V takové situaci je vykreslovací třída vytvořena a uložena do příslušného modelu pro další použití. Jedná se tak vlastně o další rozšiřující stupeň systému pro postupné načítání obsahu popsaného v kapitole 5.3.2, který je ale řízen již vykreslovací částí celého zobrazovacího systému, jenž má pochopitelně mnohem lepší informace o potřebných částech scény.

V kroku č. 4 uvedeného algoritmu mohou nastat ještě dva poměrně vážné problémy, které se aktuální implementaci řeší vždy stejným způsobem. Díky systému pro postupné načítání obsahu scény se např. může stát, že model pro příslušnou instanci modelu nelze vůbec získat a tedy není možné získat žádnou vykreslovací třídu. Druhý problém pak může nastat v situaci, kdy sice instance modelu obsahuje správně načtený model, který ale nemá specifikovanou příslušnou zobrazovací konfiguraci modelu nutnou pro vykreslení. Oba tyto problémy se v aktuální implementaci řeší stejným způsobem, uvedená instance modelu je jednoduše zařazena do speciálního seznamu chybných instancí modelů, které jsou po dokončení vykreslování znázorněny ve scéně vykreslením speciálního objektu, který tak upozorní tvůrce scény na to, že něco není v pořádku v definici příslušného modelu. Je potom na tvůrci scény, případně na odpovědných testerech, zaregistrovat tento problém a opravit jej doplněním definice příslušného modelu.

Získáním výstupních objektů jsme úspěšně dokončili celou problematiku logiky pro řízení vykreslování a máme tak vše potřebné pro popis poslední a v podstatě nejdůležitější části systému, kterou je samotný vykreslovací a optimalizační systém.

7 Vykreslovací a optimalizační systém

V této kapitole se již budeme zabývat vlastním vykreslováním scény, jehož nedílnou součástí je také optimalizace celého vykreslovacího procesu. Vykreslovací a optimalizační systém předpokládá dle schématu na obrázku 6.1, že jeho vstupem je množina objektů určených k vykreslení. Ty jsme definovali jako objekty typu `LogicOuputSet` v kapitole 6.2.4 a získali postupem popsaným v kapitole 6.3.2.

7.1 Základní pravidla optimalizace vykreslování

Ještě předtím než se začneme věnovat vlastnímu procesu vykreslování, musíme si ujasnit pravidla, která je nutná dodržet, aby se vykreslování scény dalo považovat za optimální. Přitom předpokládáme, že se bavíme pouze o vykreslování viditelných instancí modelů, tedy základním předpokladem optimalizace, kterým je odstranění neviditelných částí scény, se již zabývat nebudeme.

Optimalizace vykreslování grafiky je velmi proměnlivá činnost vzhledem k časovému období, kterého se týká. Zatímco v určité době platí pro optimální vykreslování určitá pravidla, o pár let později může být všechno jinak. Proto je důležité upozornit, že zde popsané principy se týkají aktuální doby (tedy rok 2008) a časově přilehlých obdobích. Samozřejmě že v každé době vždy platilo, že čím méně je toho nutné vykreslovat, tím rychlejší vykreslení bude[33], my se zde však budeme spíš věnovat obecným principům, které nehledí na počet vykreslovaných entit, ale spíše na mechanismy vlastního vykreslovaní. Všechny zde popsané principy pak aplikujeme na implementovaný vykreslovací systém.

7.1.1 Dosažení maximálního paralelismu mezi GPU a CPU

Dnešní grafický hardware (dále jen GPU) je v rámci celého výpočetního systému možné považovat za naprosto samostatný čip, který se stará o vykreslování grafiky na základě příkazů od aplikace (tedy CPU). Přitom práce CPU a GPU není nijak svázána a závislá, naopak oba čipy pracují paralelně a čím více paralelní je jejich vzájemná práce, tím vyšší výkon mohou podávat (viz např. [4]). Bohužel, paralelismus mezi CPU a GPU je přerušen pokaždé, kdy CPU vyžaduje od GPU nějakou činnost, jejímž výsledkem má být okamžitá odpověď a která je zároveň závislá na procesu zobrazení. Takovou činností dnes bývá např. čtení z paměti hloubky (angl. „depth-buffer“) a další blokující operace. V takovém okamžiku je nutné činnost CPU a GPU synchronizovat, což v praktickém důsledku znamená, že CPU zahálí, zatímco GPU musí dokončit aktuální vykreslovací operace a poté poskytnout odpověď na zadaný dotaz. Pro oba čipy to většinou znamená velkou ztrátu výkonu.

[33] S tím se pojí vcelku často používaná, poměrně úsměvná, ale zároveň pravdivá poučka, že nejrychleji se zobrazí ty objekty, které se vůbec zobrazovat nebudou.

Samozřejmě existují i další operace, které způsobí porušení paralelismu mezi CPU a GPU, v podstatě jakákoliv komunikace mezi těmito čipy je nežádoucí. Proto dnes obecně platí, že tok dat mezi CPU a GPU by měl být vždy ve směru od CPU a GPU, v opačném případě dochází většinou k degradaci výkonu při vykreslování. Zároveň také platí, že komunikace mezi GPU a CPU by měla být minimalizována. Proto je prvním nutným pravidlem použitým při optimalizaci vyhnout se jakýmkoliv operacím, které mohou ovlivnit paralelní činnost CPU a GPU. Pokud je použití např. zmíněného čtení z paměti hloubky nezbytně nutné, pak existují způsoby, jak degradaci výkonu za určitých podmínek minimalizovat (viz např. [14]). Na problémy s paralelismem mezi CPU a GPU bude při popisu vykreslovacího systému upozorněno.

7.1.2 Minimalizace stavových změn GPU

V předchozí kapitole bylo uvedeno, že CPU řídí činnost GPU pomocí příkazů. Tyto příkazy jsou tradičně představovány voláním jednotlivých funkcí použitého grafického API (dnes typicky Direct3D nebo OpenGL). Samotné volání příkazů příslušného API ovšem není ekvivalentní příkazům, kterými se GPU skutečně řídí. Volání jsou v ovladačích GPU transformována na sadu jednodušších příkazů, které jsou specifické pro konkrétní grafický čip a ty teprve ovládají grafický čip. Základním problémem těchto volání bývá především vysoká režie s nimi spojená, ačkoliv v programu se typicky jedná o volání jedné funkce, její složitost může být velmi vysoká díky složitosti operací, které se pod danou operací skrývají (viz např. [12]). O to vyšší je cena takového volání v případě, že vyžaduje změnu nastavení GPU. Takovou změnou může být např. změna aktuální textury, vertex shaderu, pixel shaderu apod. Obecný trend dnešní doby říká, že počet změn stavu GPU by měl být v rámci jednoho vykreslovaného snímku minimální. To je taky základním požadavkem vykreslovacího systému a my se na něj zaměříme nejvíce.

7.2 Postup vykreslování

Nyní se již dostáváme k samotnému procesu vykreslování scény. Jeho vstupem jsou objekty typu `LogicOutputSet` získané postupem popsaným v kapitole 6.3.2, kde každý jeden objekt představuje kolekci objektů typu `LogicOutput` pro jednu konfiguraci scény. Nejjednodušší způsob, jak scénu pomocí těchto objektů zobrazit, by bylo jednoduše postupně využít jednotlivé vykreslovací třídy `RenderClass` uložené v objektech typu `LogicOutput` (viz kapitola 6.2.4) a pomocí metody `Render` je jednoduše vykreslovat. Takový způsob vykreslování by však nebyl příliš efektivní, především kvůli tomu, že každá vykreslovací třída může používat jiné grafické datové zdroje (shadery, textury, ...) a to by znamenalo příliš časté změny stavu GPU, což by odporovalo jednomu ze základních požadavků na optimalizaci popsaném v kapitole 7.1.2. Navíc by takový způsob kreslení v podstatě znehodnotil předchozí práci celého systému, protože s takovým přístupem

bychom mohli viditelné instance modelů zobrazovat přímo a nečekat s jejich vykreslením na pozdější dobu.

Vykreslování tedy musí být prováděno lepším způsobem, který minimalizuje množství stavových změn prováděných v GPU. To tedy znamená, že vykreslované objekty by měly být jistým způsobem seřazeny tak, aby stavové změny v GPU byly minimální. Zde je ovšem nutné vyřešit dva základní problémy – jakým způsobem jednotlivé vykreslované objekty seřadit a jaká jsou vlastně kritéria pro toto řazení.

Kritéria pro řazení vykreslovaných objektů typu `LogicOutput` jsou v aktuální implementaci vždy odvozeny od příslušné vykreslovací třídy typu `RenderClass` v nich obsažené. To je logické, neboť třída `RenderClass` vlastně jako jediná obsahuje v rámci celého systému grafické datové zdroje a také je jako jediná využívá. Jak ale zajistit, že při vykreslování jednotlivých objektů příslušných vykreslovacích tříd pomocí jejich metody `RenderClass::Render` nebude docházet k zbytečným změnám stavu GPU? K vyřešení tohoto problému byla každá vykreslovací třída povinně vybavena metodami, které mohou vykreslovací systém informovat o tom, jaké grafické datové zdroje používá. V aktuální implementaci to znamená, že každá vykreslovací třída musí povinně poskytnout informaci o používané geometrii, vertex shaderu a pixel shaderu. Další informace již poskytovat nemusí, neboť např. u textur není nikdy zajištěno, kolik jich daná vykreslovací třída používá, zatímco předchozí povinné prvky jsou vždy nutné pro vykreslování.

Vykreslovací systém je tedy schopen získat o každé vykreslovací třídě informaci o základních grafických datových zdrojích, které tato třída využívá. Na základě těchto informací je poté schopen seřadit vykreslované objekty v takovém pořadí, aby počet stavových změn v rámci GPU byl pokud možno minimální.

Kritéria pro řazení objektů při vykreslování máme tedy definovaná, nicméně ještě není jasné, v jakém pořadí je využít k řazení objektů. Každá vykreslovací třída totiž poskytuje hned několik kritérií pro řazení, z čehož vyplývá, že řazení musí probíhat postupně podle jednotlivých kritérií. Nyní je tedy nutné vyřešit, které kritérium použít pro řazení jako první, druhé apod. Na toto bohužel neexistuje jednoznačná odpověď, pouze z teoretického hlediska je možné konstatovat, že jako první by mělo probíhat řazení podle těch kritérií, která se mění nejméně (tedy je velká šance, že mnoho vykreslovacích tříd sdílí jedno kritérium), zatímco jako poslední by se mělo řadit podle toho kritéria, které se s vysokou pravděpodobností mění nejvíce. Na tomto teoretickém základu byla založena také aktuální implementace, která řazení provádí následujícím postupem.

Každá zobrazovací třída poskytuje informace o použité geometrii, vertex shaderu a pixel shaderu. Geometrie se vždy skládá z pole vrcholů a pole indexů, které tuto geometrii tvoří[34]. Protože

[34] České výrazy „pole vrcholů“ a „pole indexů“ zde označují tradičně používané angl. výrazy „vertex buffer“ a „index buffer“, které označují objekty uložené často přímo v paměti GPU a používané při vykreslování geometrie.

je aktuální implementace založena na Direct3D API, specifikuje každá geometrie také deklaraci[35] použitých vrcholů. Máme tedy celkem pět kritérií podle kterých lze vykreslovací třídy seřadit před vykreslením, jsou to deklarace vrcholů, vertex shadery, pixel shadery, pole vrcholů a pole indexů. Toto pořadí zároveň určuje nejlepší vhodné pořadí, v jakém objekty řadit, neboť se předpokládá, že např. počet různých deklarací vrcholů se pohybuje v řádu jednotek, počet vertex shaderů je v běžné aplikaci v řádu desítek, ale typicky jich bývá méně než počet různých pixel shaderů. Počet polí vrcholů pak bývá předem neurčený, v rozsáhlých scénách ale díky velkému množství různé geometrie přesahuje počet různých shaderů. Navíc změna pole s vrcholy není v dnešním grafickém hardwaru tak časově náročná. Posledním kritériem pro řazení pak zůstávají pole indexů, neboť u nich se předpokládá stejný nebo vyšší počet než v případě polí vrcholů, především pak z toho důvodu, že pole vrcholů může být snadno sdíleno a pomocí mnoha polí indexů používáno pro vykreslování různých objektů, které ukládají vrcholy do společného pole vrcholů. Tím jsme tedy definovali podle čeho a v jakém pořadí se vlastně mají řadit vykreslovací třídy a mohli bychom tedy přistoupit k vlastnímu algoritmu vykreslování. To by jistě bylo možné, přesto je mnohem lepší pro částečné zjednodušení procesu řazení provést ještě následující zjednodušení.

Víme, že vstupem vykreslovacího systému jsou objekty typu `LogicOuputSet`, kde každý představuje kolekci objektů typu `LogicOutput` pro jednu konfiguraci scény. Přitom se předpokládá, že počet objektů typu `LogicOuputSet` vstupujících do vykreslovacího systému je malý (typicky v jednotkách). Zároveň se dá předpokládat, že vykreslovací třídy použité v jednotlivých konfiguracích scény mají typicky více společných grafických datových zdrojů. Z těchto důvodu lze považovat za efektivnější provádět řazení (a tedy i vykreslování) jednotlivých vykreslovaných prvků po jednotlivých konfiguracích scény spíše než nad všemi prvky najednou. Po tomto zjednodušení již můžeme definovat celý proces vykreslení, ten vypadá takto:

1. Postupně procházej všechny vstupní konfigurace scény reprezentované vstupními objekty typu `LogicOutputSet`.
2. Získej deklaraci vrcholu používanou prvním prvkem ze seznamu všech objektů typu `LogicOutput`. Pokud je již kolekce prázdná, pokračuj dalším objektem typu `LogicOutputScene` z kroku 1. Jinak vyjmi všechny prvky využívající stejnou deklaraci vrcholu z aktuální kolekce `LogicOutputSet` a umísti je do dočasné kolekce prvků typu `LogicOutput`. Zároveň s tím nastav deklaraci vrcholu do grafického zařízení.
3. Z dočasné kolekce prvků získané v kroku 2 získej vertex shader používaný prvním prvkem této kolekce. Pokud je již kolekce prázdná, pokračuj krokem 2. Jinak vyjmi všechny prvky využívající stejný vertex shader z této dočasné kolekce a umísti je do další dočasné kolekce

[35] Jedná se o specifický prvek vyvinutý za účelem snazší komunikace mezi aplikací, ovladači grafického subsystému a GPU. Před každým vykreslováním vyžaduje Direct3D určit deklarací typ použitých vrcholů, čímž se urychluje proces navázání programovatelných shaderů na vstupy aplikace. Více informací o deklaraci vrcholů lze nalézt v dokumentaci k Direct3D SDK nebo v [17].

prvků typu `LogicOutput`. Zároveň s tím nastav tento vertex shader do grafického zařízení.

4. Z dočasné kolekce prvků získané v kroku 3 získej pixel shader používaný prvním prvkem této kolekce. Pokud je již kolekce prázdná, pokračuj krokem 3. Jinak vyjmi všechny prvky využívající stejný pixel shader z této dočasné kolekce a umísti je do další dočasné kolekce prvků typu `LogicOutput`. Zároveň s tím nastav tento pixel shader do grafického zařízení.
5. Z dočasné kolekce prvků získané v kroku 4 získej pole vrcholů používané prvním prvkem této kolekce. Pokud je již kolekce prázdná, pokračuj krokem 4. Jinak vyjmi všechny prvky využívající stejné pole vrcholů z této dočasné kolekce a umísti je do další dočasné kolekce prvků typu `LogicOutput`. Zároveň s tím nastav toto pole vrcholů do grafického zařízení.
6. Z dočasné kolekce prvků získané v kroku 5 získej pole indexů používané prvním prvkem této kolekce. Pokud je již kolekce prázdná, pokračuj krokem 5. Jinak vyjmi všechny prvky využívající stejné pole indexů z této dočasné kolekce a umísti je do další dočasné kolekce prvků typu `LogicOutput`. Zároveň s tím nastav toto pole indexů do grafického zařízení.
7. Vykresli všechny prvky z dočasné kolekce získané v kroku 6 voláním metody `Render` příslušné vykreslovací třídy uložené v daném prvku typu `LogicOutput`. Metoda `Render` může předpokládat, že je v grafickém zařízení správně nastavena deklarace vrcholu, vertex shader, pixel shader, pole vrcholů i pole indexů. Ostatní potřebné parametry grafického nastavení si již musí nastavit sama.
8. Po dokončení kroku 7 pokračuj krokem 6.

Tímto jednoduchým algoritmem se tedy velice efektivně vykreslují jednotlivé objekty. Příslušná metoda `Render` použité vykreslovací třídy musí nastavit pouze svoje specifické grafické zdroje, jakými jsou typicky textury nebo pro každou instanci specifické konstanty shaderových programů. Na celém algoritmu řazení je především důležitá jeho rychlost. Algoritmus totiž ve skutečnosti položky neřadí v pravém slova smyslu, tj. nedochází k žádnému přesouvání položek, které zbytečně zatěžuje běžné řadící algoritmy. Jedná se spíše o jakési třídění, kdy vybrané objekty mohou být bez problémů rychle vyřazeny z původní kolekce.

Popsaným algoritmem tedy jednoduše vykreslíme všechny vstupní objekty typu `LogicOuput` získané z logiky pro řízení vykreslování. Mohlo by se tedy zdát, že je tím samotný proces vykreslování ukončen. V jistých situacích tak tomu skutečně může být, nicméně nesmíme zapomenout na to, že jedna instance modelu mohla být v logice pro řízení vykreslování přiřazena k více konfiguracím scény a tedy je obsažena ve více objektech typu `LogicOuput` různých kolekcí typu `LogicOutputSet` (viz zmínka v kapitole 6.3.2). To se typicky stává když instance modelu leží v dosahu dvou a více světelných zdrojů apod. To ovšem znamená, že se popsaným postupem

objekt vykreslí vícenásobně a jeho výsledný grafický vzhled bude nesprávný! I na tuto situaci bylo naštěstí ve vykreslovacím systému pamatováno.

Dnešní grafické aplikace typicky aplikují více grafických efektů na jeden objekt pomocí víceprůchodového vykreslování daného objektu s využitím alfa míchání (viz např. [11]) v paměti snímku. Tato technika pracuje tak, že se nejprve vykreslí daný objekt naprosto běžným způsobem pro jeden grafický efekt a následné grafické efekty se již aplikují pomocí zapnutého alfa míchání k aktuálnímu výsledku. Tento postup byl aplikován i ve zde popisovaném vykreslovacím systému, nicméně narozdíl od jednoduché teorie, která počítá pouze s jedním objektem, se zde musí brát v úvahu fakt, že jednotlivé grafické efekty aplikované na jeden objekt nenásledují ihned po sobě (každý efekt vychází z jedné konfigurace scény a ty se vykreslují každá odděleně, viz algoritmus vykreslování), ale naopak ve dříve nespecifikovaném pořadí. Jak tedy rozlišit, který grafický efekt má být zobrazen jako první a které se mají následně přimíchat se zapnutým alfa mícháním? Odpověď je jednoduchá – na pořadí aplikace grafických efektů nezáleží, neboť alfa míchání více grafických efektů pomocí víceprůchodového vykreslování vždy probíhá aditivním skládáním barev a to je operace komutativní. Jedinou podmínkou celého procesu tedy zůstává, že první grafický efekt aplikovaný na vykreslovaný objekt musí být proveden bez alfa míchání. To bylo v aktuální implementaci vyřešeno snadno následujícím postupem. Každý vykreslovaný objekt představuje instanci modelu a ta je jedinečná. Pokud je na instanci modelu aplikováno více konfigurací scény (tedy více grafických efektů) pak je toto v objektu instance modelu typu `SceneModelInstance` poznamenáno zvláštní značkou booleovského typu, která udává, že daná instance modelu ještě nebyla vykreslena s prvním grafickým efektem. Při vykreslování jednotlivých instancí modelů je pak tato značka kontrolována a metoda `Render` příslušné vykreslovací třídy (která má instanci modelu k dispozici) podle této značky rozpozná, jakým způsobem má aplikovat vykreslování – zda běžným způsobem nebo pomocí alfa míchání. Pokud kreslí instanci modelu jako první, pak tuto značku jednoduše nastaví na opačnou hodnotu, což způsobí, že příští vykreslení stejné instance modelu proběhne se zapnutým alfa mícháním a jednotlivé grafické efekty ze všech konfigurací scény tedy budou správně aplikované. Tímto způsobem jsme tedy vyřešili problém víceprůchodového vykreslování některých objektů a tím dokončili popis fáze vykreslování. Nyní se ještě budeme zabývat některými jednoduchými technika zlepšujícími výkonnost vykreslovacího systému.

7.2.1 Techniky použité pro optimalizaci vykreslování

Aktuální implementace používá kromě řazení objektů před vykreslením také další techniky optimalizace zobrazení, které budou popsány v této kapitole.

První a zároveň velmi jednoduchá metoda optimalizace vyplývá ze zásady o dodržení maximálního paralelismu mezi GPU a CPU popsaného v kapitole 7.1.1. Při vykreslování 3D grafiky se dnes bez výjimky používá techniky „double-buffering“, kdy pro vykreslování existují dvě paměti

snímku[36], přičemž jedna (zadní paměť snímku) se používá pro vykreslování nového snímku, zatímco druhá (přední paměť snímku) zobrazuje snímek předchozí. Po vykreslení nového snímku se obě paměti snímku vymění a takto se celá situace opakuje. Zároveň však správné použití této techniky velmi ovlivňuje stupeň paralelismu mezi GPU a CPU. Celý proces totiž funguje tak, že aktuální snímek je vykreslován do zadní paměti snímku a po vykreslení je vyvolána příslušná metoda použitého grafického API, která způsobí výměnu obou pamětí snímku. Volání této metody má však jednu nepříjemnou vlastnost, narozdíl od většiny ostatních metod běžných v používaných grafických 3D API. Metoda pro výměnu paměti snímku je totiž logicky blokující, tj. po jejím zavolání není řízení vráceno aplikaci dříve,dokud není aktuální snímek dokončen tak, aby mohla dojít k záměně paměti snímků. Takové blokující volání má samozřejmě neblahý dopad na výkon celé aplikace, především pak působí špatné využití CPU. Naštěstí na řešení tohoto problému existuje velmi jednoduchá technika, kdy je výměna obou pamětí snímku provedena až před začátkem vykreslování příštího snímku namísto původní výměny ihned po dokončení snímku. Celý proces vykreslování jednoho snímku je pro názornost znázorněn na následujícím obrázku 7.1.

```
FrameRenderCommonBufferSwap()
{
   ...proveď úkoly aplikace …
   ...ZAČÁTEK SNÍMKU…
   ...vykresli snímek…
   ...KONEC SNÍMKU…
   ...VÝMĚNA PAMĚTÍ SNÍMKŮ...
}
```

```
FrameRenderOptimalBufferSwap()
{
   ...proveď úkoly aplikace …
   ...VÝMĚNA PAMĚTÍ SNÍMKŮ...
   ...ZAČÁTEK SNÍMKU…
   ...vykresli snímek…
   ...KONEC SNÍMKU…
}
```

Obr. 7.1 – znázornění vykreslení jednoho snímku, vlevo tradiční a neefektivní způsob, vpravo pak optimálnější varianta

Vlevo je znázorněn běžný postup při vykreslování snímku, kdy jsou nejprve provedeny úkoly vyžadované aplikací, které zatěžují CPU. Poté je vykreslen snímek a následně je ihned provedena výměna přední a zadní paměti snímku, přičemž právě v tomto okamžiku typicky dochází k pozastavení činnosti CPU a tedy plýtvání výkonu. Vpravo je pak znázorněna optimalizovaná technika výměny přední a zadní paměti snímku. Na ní se jako obvykle provedou úkoly aplikace, ale ihned poté je provedena výměna snímku, což napoprvé může vypadat nelogicky, ale při druhém průchodu se smysl této operace vyjasní. Po výměně se začne vykreslovat nový snímek, ale po jeho dokončení nedojde okamžitě k výměně přední a zadní paměti snímku, naopak znovu jsou zpracovávány úkoly aplikace. Právě v tom okamžiku má GPU dostatek času dokončit vykreslování snímku do zadní paměti snímku a tím se také maximálně paralelizuje činnost GPU a CPU. Jako jediná nevýhoda se samozřejmě může jevit zpožděné zobrazení o jeden snímek, to je však naštěstí při

[36] To není úplně přesné vyjádření, neboť paměť snímku (angl. „frame buffer“) je vždy pouze jedna a v ní jsou uloženy tzv. povrchy, které reprezentují např. paměť hloubky, vykreslovací paměť (angl. „back buffer“) a zobrazovací paměť (angl. „front buffer“). My zde však budeme pro zjednodušení předpokládat, že existují dvě oddělené paměti snímků, které se navzájem střídají.

běžně použitelné rychlosti zobrazování (25 snímků za vteřinu a více) naprosto nepostřehnutelná nevýhoda, takže ji lze akceptovat. V souvislosti s aktuální implementací je důležité poznamenat, že pod částí „...proveď úkoly aplikace...“ se skrývá také detekce viditelnosti, příp. načítání grafických datových zdrojů apod., tedy CPU je efektivně využíván právě v okamžiku, kdy je GPU zaměstnán dokončením vykreslení předchozího snímku.

Další optimalizační technika použitá v aktuální implementaci vychází ze zásady o minimalizaci počtu změn stavu GPU během vykreslování popsané v kapitole 7.1.2. Při samotném vykreslování je sice vykreslování optimalizováno seřazením podle základních používaných prvků (viz kapitola 7.2), nicméně změn stavu GPU typicky probíhá při vykreslování mnohem více. Velkým problémem mnoha grafických aplikací je především redundantní nastavování stavu GPU, kdy je např. již vypnut zápis do paměti hloubky, ale aplikace se přesto znovu pokouší toto nastavení nastavit. Ve většině případů takové redundantní volání zachytí ovladače grafického hardwaru a jednoduše ho ignorují, někdy to však nemusí platit. Navíc bohužel pro ovladače grafického hardwaru typicky platí, že běží v procesu mimo aplikaci a většinou také na jiné úrovni zabezpečení, což znamená, že je mnohem rychlejší odfiltrovat redundantní změny stavu GPU přímo v aplikaci než provádět volání směřující mimo kód aplikace. To provádí i aktuální implementace vykreslovacího systému, která kompletně abstrahuje jednotlivá volání pro změnu stavu GPU a filtruje redundantní volání. Tento způsob filtrace je výhodný především v grafickém API Direct3D, které umožňuje používat grafický hardware v módu „pure device“ (pokud to hardware podporuje, více informací např. v [17]) a který předpokládá, že aplikace je zodpovědná za filtraci stavových změn. V takovém případě se pro grafický hardware může použít mnohem jednodušší a tedy výkonnější[37] větev zpracování v ovladačích grafického hardwaru.

Poslední významná optimalizační technika použitá v aktuální implementaci systému nevychází z dříve popsaných zásad, přesto je v dnešních grafických systémech téměř bez výjimky používána. Jedná se o tzv. optimalizaci předčasným vykreslením hloubky[38]. Technika je podrobně popsána např. v [14]. Zjednodušeně řečeno je scéna vykreslena nejprve do paměti hloubky s vypnutým zápisem do jednotlivých barevných kanálů, přičemž se u moderní grafického hardwaru předpokládá, že tento způsob kreslení je několikanásobně rychlejší. Následně probíhá normální zobrazení scény, které však ještě před spuštěním programu pixel shaderu eliminuje zpracování všech fragmentů scény, které by stejně neprošly následným testem hloubky. Tím se výrazně šetří výkonem pixel shaderů, které často bývají obzvláště ve scénách s komplexním osvětlením úzkým hrdlem celé aplikace. Zároveň se první průchod se zápisem pouze do paměti hloubky dá využít i později, v aktuální implementaci je např. získaná hloubka scény použita pro vytvoření efektu hloubky ostrosti (angl. „depth-of-field“) po vykreslení celé scény.

---

[37] Výkonnost těchto ovladačů vychází z absence kontroly všech možných nastavení, vše až na nejnutnější kontroly je přenecháno aplikaci, která má globální přehled o všech potřebných stavech GPU.

[38] V angličtině se často označuje jako „z-only pass“, „depth-only pass“ nebo ještě častěji jako „lay down depth first“.

Celý zobrazovací systém samozřejmě obsahuje i další méně významné optimalizace, ty však nejsou natolik podstatné a proto již v tomto textu nebudou diskutovány. Většinou se jedná o optimalizace doporučené v [14], kde lze nalézt spoustu zajímavých informací.

8 Škálování výkonu vykreslovacího systému

Součástí každého vykreslovacího systému by měla být také možnost škálování výkonu takovým způsobem, aby bylo systém možné provozovat s různě výkonným hardwarovým vybavením. Tento problém odpadá např. v případě možnosti počítat s unifikovanou hardwarovou konfigurací, což je např. případ herních konzolí a podobných nemodifikovatelných zařízení. My se však zaměřujeme na použití zobrazovacího systému osobními počítači a proto musíme počítat i s proměnlivostí použitých hardwarových konfigurací.

V této kapitole si rozebereme možnosti, který nám poskytuje aktuální implementace systém. Přitom se nebudeme zabývat takovými triviálními možnostmi, jako je např. změna rozlišení výsledného zobrazení nebo snížení dohledové vzdálenosti, které se pokládají za samozřejmé.

8.1 Škálování na bázi odlišného popisu scény

Tento ne příliš dobrý způsob škálování výkonnosti je dnes přesto stále používán v mnoha systémech pro zobrazení scén. Scéna je jednoduše vytvořena v několika úrovních hardwarové náročnosti, kdy se jednotlivé úrovně liší především složitostí geometrie a velikostí použitých textur, případně aplikovanými shadery. Tento způsob je sice funkční a může být použit i ve zde představovaném systému, bohužel přináší zvýšenou pracnost při tvorbě scény, což bývá především u velmi rozsáhlých scén problém a proto je lepší se takovému způsobu škálování vyhnout.

Ve zde představovaném systému by takový způsob škálování bylo možné použít poměrně snadno díky oddělení popisu scény a jednotlivých datových zdrojů. Stačilo by definovat datové zdroje v různých formách složitosti a ještě před načtením scény přesměrovat[39] načítání zdrojů do příslušné složky.

8.2 Poloautomatické a automatické formy škálování

Implementovaný systém nabízí dvě další možnosti, jak škálovat svou výkonnost. Bohužel ani jedna z nich nebyla zatím implementována, nicméně jejich implementace je natolik triviální, že zde budou popsány, přičemž se s jejich implementací počítá v budoucích vylepšeních. Obě zde popsané možnosti lze samozřejmě kombinovat a to samozřejmě i s možností popsanou v předchozí kapitole.

[39] Ačkoliv to zatím nebylo zmíněno, implementovaný systém je postaven na výkonném frameworku, který obsahuje také implementaci tzv. virtuálního souborového systému, který umožňuje připojovat libovolné části souborového systému do své virtuální struktury a tím přesměrovat datové soubory podle potřeby bez nutnosti přesouvat je na skutečném disku. Více informací o tomto systému lze najít přímo v implementaci, kde jsou dostupné zdrojové kódy včetně kompletních komentářů.

První možnost škálování je založena na vytvoření několika různých množin programovatelných shaderů, které odpovídají různým způsobům náročnosti vykreslení. Tato technika je tedy podobná technice definice různých obsahů scény, nicméně hlavní rozdíl je v tom, že příslušná definice scény se vůbec nemusí měnit. To je způsobeno tím, že v definici scény se nikde nevyskytují zmínky o použitých shaderech, pouze se specifikuje vykreslovací třída, která určuje způsob zobrazení daného objektu. Nic nám tedy nebrání změnit např. na základě konfiguračního souboru celého systému vykreslovací shadery pro všechny vykreslovací třídy aniž by to nějak ovlivnilo použití specifikovaného obsahu scény. Proto lze tuto možnost škálování výkonu označit za tzv. poloautomatickou, neboť nevyžaduje tolik pozornosti jako kompletní úprava celé scény pro různé stupně škálování, ale přesto se vytváří několik různých množin grafických datových zdrojů, zde konkrétně shaderů.

Druhá možnost škálování už je narozdíl od té první naprosto automatická (ačkoliv vyžaduje určité předpoklady o vytvořené scéně). Její myšlenka i implementace je přesto naprosto triviální. Vzpomeneme si na popis jednotlivých modelu (viz kapitola 6.2.1), kde je součástí každé zobrazovací konfigurace modelu množina sekcí `[lod]`, kde každá tato sekce definuje jednu úroveň detailu modelu, přičemž se předpokládá, že vzdálenější úrovně detailu vykreslují modelu v podstatně horší kvalitě (a tedy méně náročně). Pokud si uvědomíme, že použitá úroveň detailu se vždy určuje podle vzdálenosti instance modelu od pozorovatele, logicky dojdeme k závěru, že automaticky lze snadno snížit kvalitu výsledného zobrazení tak, že vždy jednoduše vzdálenost mezi pozorovatelem a instancí modelu upravíme přidáním zadané konstanty, která logicky posune jednotlivé úrovně detailu. Pokud např. model definuje pro jistou zobrazovací konfiguraci modelu tři úrovně detailu začínající postupně ve vzdálenostech 0, 500, 1000, pak přidáním konstanty 500 dojde k posunutí o jednu úroveň detailu, takže první úroveň nebude nikdy použita. Pokud budeme např. chtít použít u všech modelů všechny poslední úrovně detailu, pak jednoduše definujeme dostatečně velkou konstantu, která zajistí, že vždy bude vybrána až poslední úroveň detailu. Jak je vidět, tento způsob je velmi efektivní a jediné, co vyžaduje, je definice všech modelů scény s více úrovněmi detailů od těch nejsložitějších až po ty jednoduché, přičemž pro jednotlivé úrovně detailů by měly být vhodně nastaveny jejich počáteční vzdálenosti.

9 Budoucí rozšíření zobrazovacího systému

V této kapitole se budeme zabývat možnými rozšířeními aktuálně implementovaného systému, která byla uvažována již při samotném návrhu, ale především z časových důvodů nebyla do implementace zařazena.

9.1 Dynamické konfigurace scény

V předchozím textu byl celý reprezentovaný systém založen na tzv. konfiguracích scény, které určovaly, jakým způsobem se instance modelů v jejich dosahu budou zobrazovat. Tento způsob vykreslování scény je velmi efektivní a produkuje kvalitní výsledky s využitím velmi jednoduchého popisu scény. Jediný problém při vykreslování scény tímto způsobem nastal v situaci, kdy na jednu instanci modelu působilo více konfigurací scény a vykreslování bylo prováděno víceprůchodovým vykreslováním. Tento způsob se sice dnes aktivně používá, přesto jej již nelze považovat za příliš efektivní, neboť zvyšuje množství komunikace mezi CPU a GPU, což je v dnešní době považováno za velký problém. Např. v případě, kdy bude na instanci modelu aplikováno 10 světel, je nutné tuto instanci vykreslit deseti průchody, přičemž na devět z těchto vykreslovacích průchodů se bude aplikovat alfa míchání, což je pro GPU zbytečně náročná operace narozdíl od běžného zápisu barvy do paměti snímku. V dnešní době velmi silných specifikací shaderů verze 3 (případně 4) se proto již doporučuje ustupovat od vykreslování víceprůchodovými algoritmy v aplikaci a provádět veškerou činnost pokročilými programovatelnými shadery (viz např. [10]), které podporují větvení a tedy mohou rozhodovat například o tom, kolik světel bude ovlivňovat daný objekt, jaké budou jejich parametry apod. Přitom je ale nutné kvůli efektivitě používat pro objekty ovlivňované pouze jednou konfigurací pokud možno jednoduché shaderové programy, které jsou vždy vykonávány rychleji. Z toho důvodu byla pro konfigurace scény navržena speciální technika tzv. dynamických konfigurací scény, jejíž teoretický základ zde bude popsán..

Dynamická konfigurace scény je svými vlastnostmi podobná běžným konfiguracím scény, jediný rozdíl je v tom, že není nikdy ve scéně specifikována tvůrcem scény, ale je vždy vytvářena za běhu celého systému. Okamžikem vytvoření dynamické konfigurace scény je situace, kdy je pro jeden objekt nalezeno více aplikovatelných konfigurací scény se stejnou prioritou, což by za normální situace znamenalo přiřadit danou instanci modelu všem potřebným konfiguracím scény a provést víceprůchodové vykreslování. V případě dynamické konfigurace se však postupuje jinak. Vykreslovací systém se v případě více aplikovatelných konfigurací scény jednoduše pokusí z nich sestavit co nejlepší známou dynamickou konfiguraci (nebo více) a tu následně použít při vykreslování. Pro ilustraci si představme situaci, kdy máme ve scéně definovanou dynamickou

konfiguraci scény pro libovolné množství bodových světelných zdrojů. Pro určitou instanci modelu bylo nalezeno celkem pět aplikovatelných konfigurací scény, přitom tři z nich byla bodová světla a další dvě měly stejnou prioritu, ale jiný typ (který teď není podstatný, mohou to být např. směrová světla apod.). Za normální situace by bylo použito vykreslování pomocí pěti průchodů, každý pro jednu konfiguraci scény. Systém s podporou dynamických konfigurací však bude fungovat tak, že správně rozpozná tři stejné konfigurace scény pro bodová světla a z nich vytvoří novou dynamickou konfiguraci, kterou lze pomocí složitějšího shaderu vykreslit jedním průchodem. Tím se tedy počet průchodů vykreslení dané instance modelu sníží na tři. Pokud však bude systém ještě propracovanější, pak bude zřejmě obsahovat i takovou dynamickou konfiguraci scény, která umožní kombinovat všechny nalezené konfigurace scény dohromady[40] a tedy umožní provést vykreslení objektu jedním průchodem.

Systém dynamických konfigurací scény je tedy dobrou cestou, kterou by se měla ubírat další vylepšení tohoto zobrazovacího systému. Jeho výhodu lze spatřit především v udržení kompatibility s předchozím popisem scény pro statické konfigurace scény a zároveň ve zvýšení efektivity procesu vykreslování při aplikaci náročnějších grafických efektů.

9.2 Využití výpočetní síly GPU

Moderní GPU dnešní doby je již možné považovat za téměř univerzální výpočetní systémy, přičemž se dá podle aktuálního dění předpokládat, že v budoucnu se GPU budou stále více přibližovat svou univerzalitou k schopnostem CPU. Proto je dobré uvažovat nad přesunutím některých úkolů z CPU přímo na GPU.

Zde se v úvahu nabízí především přesunutí všech výpočtů souvisejících s detekcí viditelnosti, neboť ty jsou typicky reprezentovány vektorovými operacemi, pro které jsou GPU vysoce optimalizovaná. Zde se např. ukazuje, jak výhodné bylo oddělit detekci viditelnosti od samotných grafických dat (viz kapitola 5.1), kdy jednotlivé instance modelu typu `SceneModelInstance` jsou jednoduché objekty, které mají všechny potřebné informace nutné přímo pro určování viditelnosti a tedy mohou být snadno použity přímo pro výpočty prováděné v GPU.

[40] To obzvláště u moderních grafických 3D API jakými je např. Direct3D 10 s podporou Shader Model 4.0 není vůbec problém. Rozhraní Direct3D 10 umožňuje ještě mnohem více, včetně vykreslování více objektů s mnoha různými aplikovanými grafickými efekty apod. Více lze najít přímo v [13] .

10 Souhrnné informace o aktuální implementaci

V předchozích kapitolách byla představena aktuální implementace zobrazovacího systému z pohledu základních mechanismů. Tato kapitola provede podrobnější shrnutí aktuální implementace celého zobrazovacího systému.

10.1 Základní organizace programového kódu

Celá implementace zobrazovacího systému je napsána v jazyce C++ a o aktuální vykreslovaní se stará 3D aplikační programové rozhraní (dále jen API) Direct3D9. Aktuální vykreslovací systém používá pro všechny vykreslovací operace programovatelné shadery napsané v jazyku HLSL.

Implementace je založena na vlastním aplikačním frameworku, který poskytuje základní nezbytné prvky pro práci s 3D grafikou a další podpůrné funkční celky. Mezi ně patří např. matematická knihovna obsahující základní prvky lineární algebry a skupina tříd pro detekci kolizí pro základní tělesa (3D i 2D) používaná v grafických aplikacích. Další důležitou částí celého aplikačního frameworku je pak skupina tříd pro abstrakci aktuálně používaného 3D API (momentálně Direct3D9), což umožňuje velké části aplikace pracovat nezávisle na příslušném zobrazovacím 3D API. Velmi důležitou součástí aplikačního frameworku je také implementace virtuálního souborového systému (zmíněný již v kapitole 8.2), který umožňuje pracovat s datovým zdroji nezávisle na jejich skutečném umístění díky možnosti přesměrovat jednotlivé části skutečného souborového systému na virtuální souborový systém. To je vůbec jedna z nejdůležitějších vlastností aplikačního frameworku, protože umožňuje tvůrcům obsahu scény definovat jména souborů s grafickými datovými zdroji, resp. jakékoliv jiné soubory, pomocí relativních cest virtuálního souborového systému a přitom není nutné přemýšlet o tom, kde jsou soubory skutečně umístěny. V praxi pak může být výsledný systém dodáván např. na disku CD, spustitelná aplikace je při instalaci umístěna na lokální pevný disk a v konfiguračním systému je pouze provedeno přesměrování virtuálního souborového systému na příslušný CD disk. Přitom názvy všech souborů použitých v aplikaci, příp. v popisu scény, zůstávají beze změny. Více o implementaci a činnosti virtuálním souborovém systému lze nalézt např. ve [19] nebo přímo v programové dokumentaci aplikačního frameworku.

Implementace vlastního managementu scény a jejího vykreslování (tedy práce s konfigurace scény, vykreslovacími třídami apod.) je postavena nad základním aplikačním frameworkem, takže je zcela odstíněna od příslušného 3D API, konkrétního souborového systému apod. Což je dobrá vlastnost především pro přenosnost na libovolnou jinou platformu, kdy stačí upravit pouze základní aplikační framework a celý management scény a vykreslovací systém nemusí být vůbec změněny.

Celý systém byl navržen tak, aby byl maximálně použitelný jako zobrazovací systém v jiné aplikaci (např. počítačové hře), z toho důvodu je celý kód pečlivě rozdělen do jmenných prostorů, což zabraňuje budoucím kolizím s kódem použitým v cílové aplikaci. Navíc je celý systém implementován tak, aby jeho použití bylo velice přímočaré. V rámci cílové aplikace, kde má být použit, je pouze nutné inicializovat aplikační framework (volání jedné metody po spuštění aplikace), což zahrnuje také napojení aplikačního frameworku na vybrané okno aplikace. Dále již stačí vytvořit objekt třídy `TScene`, která reprezentuje hlavní scénu a poté zajistit v metodě pro překreslování okna překreslování scény.

10.2 Aktuálně implementované prvky vykreslovacího systému

Celý implementovaný systém je založený na konfiguracích scény a k nim jsou přiřazeny příslušné vykreslovací třídy. Pro názornou ukázku byly implementovány základní konfigurace scény představené již v kapitole 6.1. Konfigurace **Basic** představuje základní konfiguraci scény bez parametrů, od ní je přímo odvozena konfigurace **Night**, obsahující pouze informaci o všudypřítomném světle. Od ní jsou odvozeny konfigurace scény typu **Sun** a **LightPoint**, kde konfigurace scény **Sun** je použitelná především pro scény s denním osvětlením, protože představuje směrové světlo. Konfigurace scény **LightPoint** pak představuje důležitý prvek pro noční scény.

K těmto konfiguracím scény bylo implementováno několik ukázkových vykreslovacích tříd, přičemž jsou k dispozici např. následující efekty:

- Phongův osvětlovací model s Gouraudovým i Phongovým (per-pixel) stínováním
- Lambertův osvětlovací model
- modulace textury pomocí světelných map (lightmap)
- bump mapping (viz např. [15])
- parallax mapping (viz např. [24])

Jednotlivé konfigurace scény a vykreslovací třídy jsou popsány v externím dokumentu na přiloženém CD, kde je zároveň umístěn návod pro přidávání nových konfigurací scény a vykreslovacích tříd.

Systém obsahuje také plnou podporu směrových a bodových světel, včetně plné podpory dynamických stínů vytvářených pomocí stínových map. Pro ukázku efektů zpracovávajících již vykreslený obraz (tzv. „post-processingové“ efekty) pak byla implementována podpora dnes populárního efektu hloubky ostrosti.

10.3 Vytvořené aplikace

Hlavní výsledek této práce je tvořen implementovaným aplikačním frameworkem a nad ním vystavěným zobrazovacím systém. Pro názornost použití celého systému pak byla vytvořena

jednoduchá aplikace s grafickým uživatelským rozhraním, která demonstruje běh celého systému zobrazováním zvolené scény. Tato aplikace také poskytuje statistiky o průběhu zobrazování, které zaznamenává a poskytuje přímo zobrazovací systém. Základní popis této aplikace je uveden v příloze A.2. Na přiloženém CD je také k dispozici kompletní programová dokumentace k celému aplikačnímu frameworku vygenerovaná z komentářů zdrojového kódu.

V rámci práce vznikla také jednoduchá aplikace umožňující získávat geometrii pro zobrazování modelů v potřebném datovém formátu. Její popis je obsažen v příloze A.3.

Závěr

V rámci práce se podařilo navrhnout a implementovat datově řízený zobrazovací systém spolu s vhodným popisem scény. Vytvořený zobrazovací systém lze považovat za velmi perspektivní z hlediska budoucího použití především díky jeho vysoké obecnosti a snadné modifikovatelnosti s ohledem na přidávání nových prvků (grafických efektů, konfigurací scény, ...). Jako součást práce vznikla také poměrně obsáhlá knihovna pro práci s 3D grafikou, která může být libovolně využita i v jiných projektech.

Obecný přínos práce spočívá především v popisu zobrazovacího systému založeném na myšlence definice konfigurací scény a k nim přidruženým vykreslovacím třídám. Práce také poukazuje na nutnost správného návrhu zobrazovacího systému z hlediska oddělení jeho jednotlivých částí a v neposlední řadě také na nutnost optimalizovaného vykreslování scény.

Pro autora je největším přínosem celé práce především získání nových zkušeností z oblasti tvorby zobrazovacích systému, které budou moci být využity buď při dalším rozšiřování implementovaného zobrazovacího systému nebo případně při návrhu zcela nového zobrazovacího systému.

Možnosti pokračování projektu byly průběžně zmiňovány v textu v podobě dalších možných rozšíření jednotlivých částí systému. Z pohledu autora se jeví jako nejzajímavější možnost rozšíření o podporu tzv. dynamických konfigurací scény (zmíněno v kapitole 9.1), což je záležitost, která dnes nemá ekvivalentní zastoupení ani v komerčních zobrazovacích systémech a tudíž představuje poměrně zajímavou výzvu pro implementaci. Z čistě technického hlediska by také jistě bylo zajímavé (v případně dostupnosti potřebných vývojářských nástrojů) přenesení stávající implementace na některou z konzolí nové generace (Xbox 360, Playstation 3), neboť podpora všech základních platforem je dnes pro každý zobrazovací systém klíčová.

Celkově lze výsledek práce považovat za uspokojivý, přesto je nutné říci, že k ověření skutečné použitelnosti, příp. pro hledání slabých míst celého zobrazovacího systému, by bylo nutné použít jej ve skutečné aplikaci (typicky počítačové hře), na jejímž vývoji by se podílel větší tým lidí, kteří by sami dokázali posoudit, zda zde vytvořený zobrazovací systém splňuje (nebo po úpravách je schopen splnit) všechny požadavky, které od zobrazovacího systému očekávají.

Literatura

[1] Angel, E.: *Interactive Computer Graphics: A Top-Down Approach using OpenGL (4thEdition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., čtvrté vydání, 2006, ISBN 0-321-32137-5.

[2] Bacik, M. : *Rendering the Great Outdoors: Fast Occlusion Culling for Outdoor Environments*. Gamasutra – The Art & Business of Making Games, [online], Prosinec 2007, [rev. 2007-12-30], [cit. 2007-12-30].
URL http://www.gamasutra.com/features/20020717/bacik_pfv.htm

[3] Bikker, J. : *Building a 3D Portal Engine*. flipcode – Daily Game Development News & Resources, [online], Květen 2008, [rev. 2008-05-05], [cit. 2008-05-05].
URL http://www.flipcode.com/archives/Building_a_3D_Portal_Engine-Issue_01_Introduction.shtml

[4] Huddy, R. : *DirectX8 Performance*. In Proceedings Game Developer Conference 2002, [online], Květen 2008, [rev. 2008-05-05], [cit. 2008-05-05].
URL http://developer.nvidia.com/object/gdc_d3dperf.html

[5] Eberly, D. H.: *3D Game Engine Design*. San Francisco, USA: Morgan Kaufmann Publishers, druhé vydání, 2001, ISBN 1-55860-593-2.

[6] Eberly, D. H.: *3D game engine architecture: engineering real-time applications with wild magic*. Boston, USA: Morgan Kaufmann Publishers, první vydání, 2005, ISBN 0-12-229064-X.

[7] Fernando, R. et all.: *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Boston, USA: Addison-Wesley Professional, první vydání, 2004, ISBN 0-321-22832-4.

[8] Fowler, M.: *UML Distilled*. Massachusetts, USA: Addison-Wesley Professional, druhé vydání, 2000, ISBN 0-201-65783-X.

[9] Gamma, E., Helm, R., Johnson, R., Vlissides J.M.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Massachusetts, USA: Addison-Wesley Professional, první vydání, 1997, ISBN 0-201-63361-2.

[10] Lacroix, J. : *Let There Be Light!: A Unified Lighting Technique for a New Generation of Games*. Gamasutra – The Art & Business of Making Games, [online], Květen 2008, [rev. 2008-05-05], [cit. 2008-05-05].
URL http://www.gamasutra.com/features/20050729/lacroix_pfv.htm

[11] McReynolds, T., Blythe, D.: *Advanced Graphics Programming Using OpenGL*. San Francisco, USA: Morgan Kaufmann Publishers, první vydání, 2005, ISBN 1-55860-659-9.

[12] Microsoft Corporation: *Accurately Profiling Direct3D API Calls*, [online], Květen 2008, [rev. 2008-05-05], [cit. 2008-05-05].
URL http://msdn2.microsoft.com/en-us/library/bb172234(VS.85).aspx

[13] Microsoft Corporation: *Direct3D 10 Graphics*, [online], Květen 2008, [rev. 2008-05-05], [cit. 2008-05-05].
URL http://msdn2.microsoft.com/en-us/library/bb205066(VS.85).aspx

[14] NVIDIA Corporation: *NVIDIA GPU Programming Guide*, [online], Květen 2008, [rev. 2008-05-05], [cit. 2008-05-05].
URL http://developer.nvidia.com/object/gpu_programming_guide.html

[15] Nguyen, T.: *Bump Mapping: How It Works,* 2008, [online], Květen 2008, [rev. 2008-05-05], [cit. 2008-05-05].
URL http://www.tweak3d.net/articles/bumpmapping/

[16] Schneider, P. J., Eberly, D. H.: *Geometric tools for computer graphics*. San Francisco, USA: Elsevier Science, první vydání, 2003, ISBN 1-55860-594-0.

[17] Thompson, R.: *The Direct3D Graphic Pipeline*, [online], Květen 2008, [rev. 2008-05-05], [cit. 2008-05-05].
URL http://www.xmission.com/~legalize/book/index.html

[18] Walsh, N.: *A Technical Introduction to XML,* 2008, [online], Květen 2008, [rev. 2008-05-05], [cit. 2008-05-05].
URL http://www.xml.com/pub/a/98/10/guide0.html

[19] Walter, M.: *Programming a Virtual File System.* flipcode – Daily Game Development News & Resources, [online], Květen 2008, [rev. 2008-05-05], [cit. 2008-05-05].
URL http://www.flipcode.com/archives/Programming_a_Virtual_File_System-Part_I.shtml

[20] Weisstein, E. W.: *Right-Handed Coordinate System*. MathWorld – A Wolfram Web Resource, 2008, [online], Květen 2008, [rev. 2008-05-05], [cit. 2008-05-05].
URL http://mathworld.wolfram.com/Right-HandedCoordinateSystem.html

[21] Wikipedia contributors: *IEEE 754-1985*. Wikipedia, The Free Encyclopedia, 2008, [online], Květen 2008, [rev. 2008-05-05], [cit. 2008-05-05].
URL http://en.wikipedia.org/wiki/IEEE_754

[22] Wikipedia contributors: *Reference Counting*. Wikipedia, The Free Encyclopedia, 2008, [online], Květen 2008, [rev. 2008-05-05], [cit. 2008-05-05].
URL http://en.wikipedia.org/wiki/Reference_counting

[23] Wirth, N.: *Algorithms + Data Structures = Programs.* Englewood, USA: Prentice-Hall, Inc, první vydání, 1976, ISBN 0130224189.

[24] Zink, J.: *A Closer Look At Parallax Occlusion Mapping.* GameDev.net – all your game development needs, [online], Květen 2008, [rev. 2008-05-05], [cit. 2008-05-05].
URL http://www.gamedev.net/reference/articles/article2325.asp

Seznam příloh

Příloha A.1 Popis jednotkového kvaternionu pro vyjádření rotace.

Příloha A.2 Popis aplikace pro prohlížení popisu scény.

Příloha A.3 Popis aplikace pro získávání souborů z geometrií.

Příloha A.4 Obrazová příloha.

Externí příloha Přiložený CD disk. Obsahuje zdrojový text této práce, zdrojové kódy k vytvořenému softwaru a také zkompilované programy vhodné pro spuštění. Dále obsahuje dokumenty popisující jednotlivé prvky zobrazovacího systému a úplnou programovou dokumentaci zobrazovacího systému. Na CD je také k dispozici ukázková scéna vhodná pro prohlížení vytvořeným softwarem.

Přílohy

A.1 Jednotkový kvaternion pro vyjádření rotace

Soubor s popisem obsahu scény definovaný v kapitole 3.4.2 obsahuje pro vyjádření rotací jednotkový kvaternion v následujícím tvaru:

$$q = [qx, qy, qz, qw]$$

Takový kvaternion představuje rotaci o daný úhel **α** kolem osy procházející počátkem souřadného systému, mající tento jednotkový směrový vektor:

$$dir = [dx, dy, dz]$$

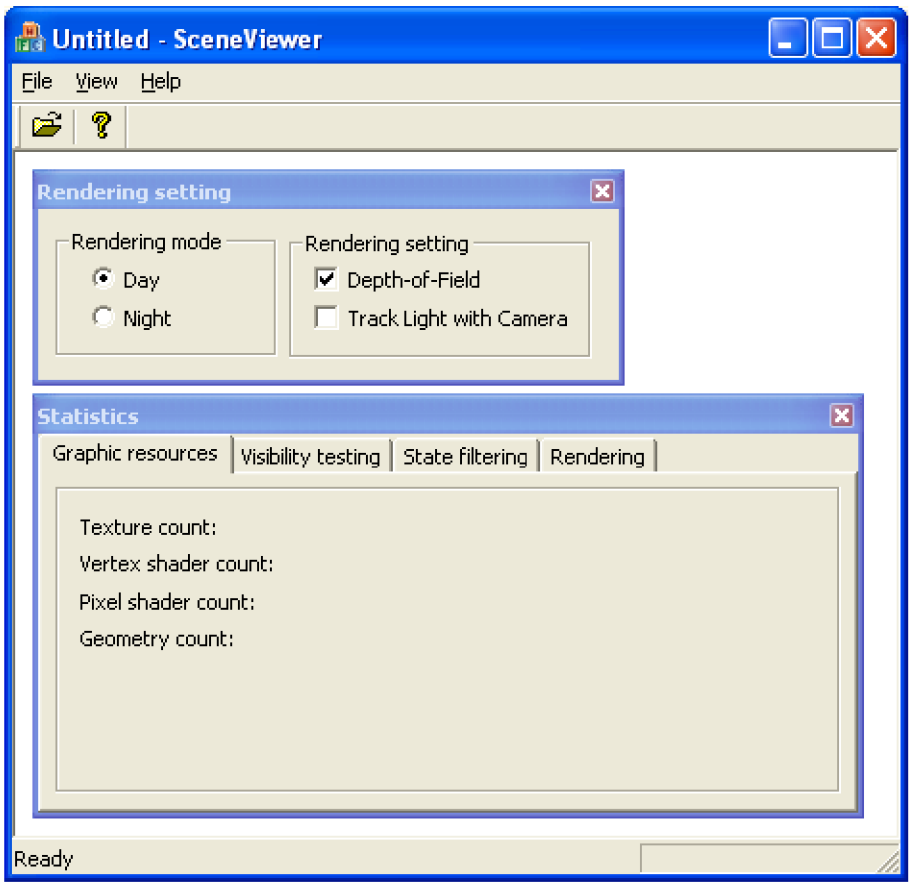
Z vektoru ***dir*** a zadaného úhlu **α** a lze pak složky kvaternionu spočítat takto:

$$qx = dx \times \sin(\alpha \div 2)$$
$$qy = dy \times \sin(\alpha \div 2)$$
$$qz = dz \times \sin(\alpha \div 2)$$
$$qw = \cos(\alpha \div 2)$$

Více informací o kvaternionech a operacích s nimi lze získat např. v [6].

A.2 Aplikace pro prohlížení scény - SceneViewer

Aplikace **SceneViewer** představuje jednoduchou ukázku použití aplikačního frameworku. Lze pomocí ní načíst soubor s popisem scény a následně scénou procházet, zároveň zobrazuje statistiky o průběhu zobrazování a také umožňuje měnit základní nastavení zobrazení. Aplikace má standardní grafické uživatelské rozhraní s nabídkou a panelem nástrojů v horní části aplikace, zbytek okna tvoří klientská oblast, ve které se zobrazuje právě načtená scéna. Okno aplikace je na obrázku A.2.1.



Obrázek A.2.10.1 – okno aplikace SceneViewer pro prohlížení scény

V aplikaci jsou k dispozici dva nemodální dialogy **Render setting** a **Statistics**, které lze zobrazit z nabídky **View**. Pomocí nabídky **File** lze načíst, příp. uzavřít, soubor s popisem scény. Ukázkové soubory s popisem scény jsou k dispozici na přiloženém CD, kde jsou také uvedené podrobné instrukce pro načtení scény. Po načtení scény jsou důležité především oba výše zmíněné nemodální dialogy, které budou popsány zde.

Dialog Render setting

V tomto dialogu lze přímo za běhu aplikace provádět změny ve zobrazování scény. Vzhledem k jednoduchosti aplikace jsou zde k dispozici pouze základní volby, které demonstrují možnosti zobrazovacího systému. V sekci **Rendering Mode** je možné vybrat aktuální způsob zobrazování

scény, k dispozici jsou volby denního a nočního režimu. Změna těchto položek způsobí změnu dostupných konfigurací scény a to se následně promítne do způsobu zobrazení celé scény. Tedy při změně tohoto nastavení je i přes změnu vzhledu zobrazení použita stále stejná scéna.

Sekce **Rendering Setting** obsahuje možnosti nastavení zobrazování, přičemž v této jednoduché aplikaci jsou k dispozici pouze dvě. Položka **Depth-of-Field** zapíná či vypíná efekt hloubky ostrosti aplikovaný na scénu. Položka **Track Light with Camera** je užitečná ve scéně v nočním režimu, pokud je aktivní, pak se spolu s pozorovatelem pohybuje i bodové světlo umístěné nad ním, což umožňuje lépe pozorovat některé efekty (zde např. stíny).

Dialog Statistics

Tento dialog poskytuje informace o průběhu vykreslování scény. Dialog je rozdělen na několik záložek, které budou popsány zde.

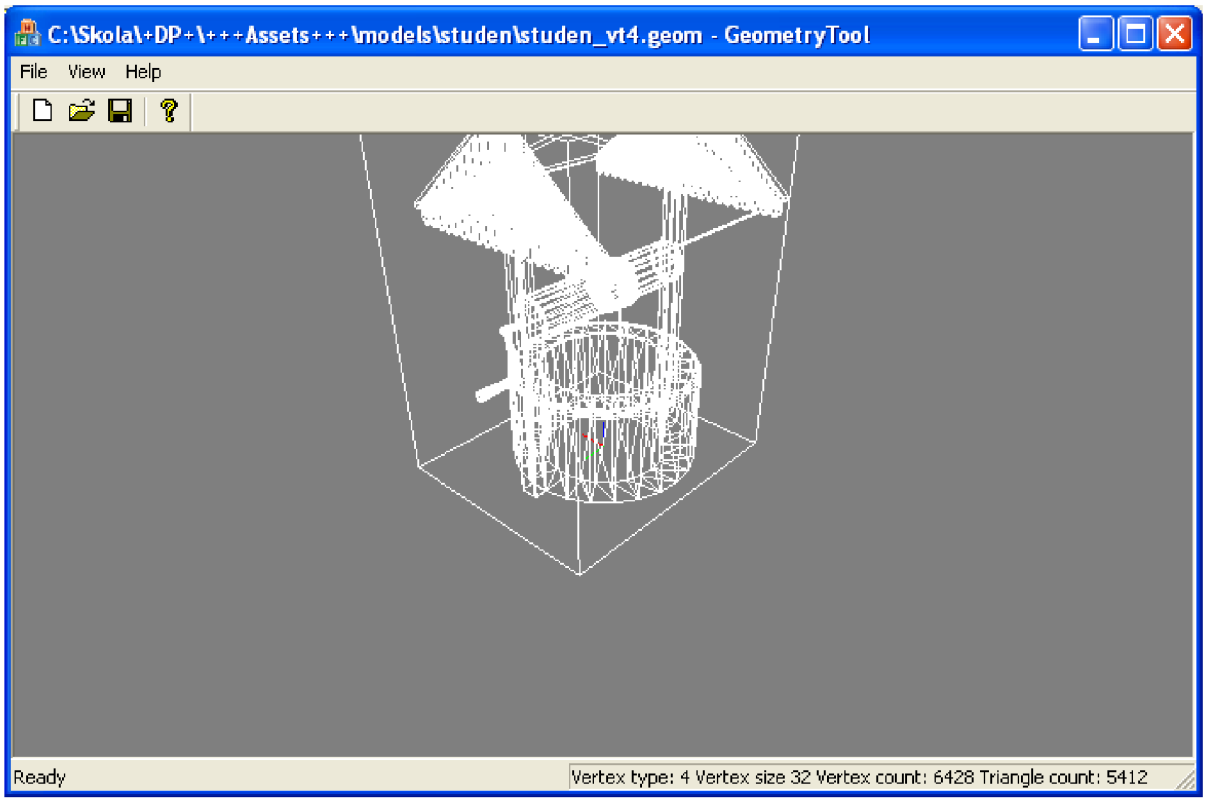
- záložka **Graphic resources** – tato záložka obsahuje informace o grafických datových zdrojích aktuálně obsažených v aplikaci. Význam jednotlivých položek je následující:
  - **Texture count** – počet jedinečných textur použitých aplikací.
  - **Vertex shader count** – počet jedinečných vertex shaderů použitých v aplikaci.
  - **Pixel shader count** – počet jedinečných pixel shaderů použitých v aplikaci.
  - **Geometry count** – počet jedinečných souborů s geometrií použitých v aplikaci.
- záložka **Visibility testing** – tato záložka obsahuje informace o provedených testech viditelnosti během posledního zobrazovaného snímku. Význam jednotlivých položek je následující:
  - **Tested tiles** – počet testovaných dlaždic v posledním snímku.
  - **Visible tiles** – počet nalezených viditelných dlaždic v posledním snímku.
  - **Tested instances** – počet testovaných instancí modelů v posledním snímku.
  - **Visible instances** – počet viditelných instancí modelů v posledním snímku, tedy celkový počet vykreslovaných instancí.
- záložka **State filtering** - tato záložka obsahuje informace o počtu změn stavu grafického zařízení, sekce **Changed states** obsahuje informace o provedených změnách, sekce **Filtered states** pak obsahuje informace o změnách stavu, které byly redundantní a tedy byly zachyceny zobrazovacím systémem. Význam položek je následující:
  - **Declarations** – počet změn deklarací vrcholů v grafickém zařízení.
  - **Vertex shaders** – počet změn vertex shaderů v grafickém zařízení.
  - **Pixel shaders** – počet změn pixel shaderů v grafickém zařízení.
  - **Vertex buffers** – počet změn polí s vrcholy v grafickém zařízení.
  - **Index buffers**– počet změn polí s indexy v grafickém zařízení.
  - **Textures** – počet změn textur v grafickém zařízení.

- **Sampler states** – počet změn nastavení texurovacích jednotek grafického zařízení (např. kvalita filtrování, zpracování texturovacích souřadnic apod.).
- **Render states** – počet změn různých nastavení grafického zařízení (např. vypnutí nebo zapnutí zápisu do paměti hloubky, alfa míchání apod.).

- záložka **Rendering** – tato záložka obsahuje informace o vykreslování aktuálního snímku. Význam položek je následující:
  - **Frame ID** – číslo aktuálně vykreslovaného snímku.
  - **Detected scene configs** – počet nalezených konfigurací scény v aktuálním pohledu.
  - **Used scene configs** – počet použitelných konfigurací scény pro vykreslení.
  - **1-pass models** – počet modelů vykreslovaných jedním průchodem.
  - **Multi-pass models** – počet modelů vykreslovaných více průchody se zapnutým alfa mícháním.
  - **Models with no config** – počet modelů, které nebylo možné zobrazit, protože nebyla nalezena žádná konfigurace, ke které by patřily. Tyto modely jsou znázorněny ve scéně zeleným vykřičníkem.
  - **Models with missing config** – počet modelů, které byly správně přiřazeny ke konfiguraci scény, ale neobsahují potřebnou konfiguraci modelu pro tuto konfiguraci scény. Tyto modely jsou znázorněny ve scéně červeným vykřičníkem.

A.3 Aplikace pro získávání geometrie - GeometryTool

Aplikace **GeometryTool** je jednoduchý nástroj pro tvorbu souborů s geometrií s požadovaným typem vrcholů. V současné implementaci podporuje pouze vstupní souborový formát typu DirectX, nicméně je možné ji rozšiřovat o podporu libovolných souborových formátu s popisem modelu. Výstupem této aplikace je vždy soubor ve formátu popsaném v kapitole 3.1. Okno aplikace je zobrazeno na obrázku A.3.1:

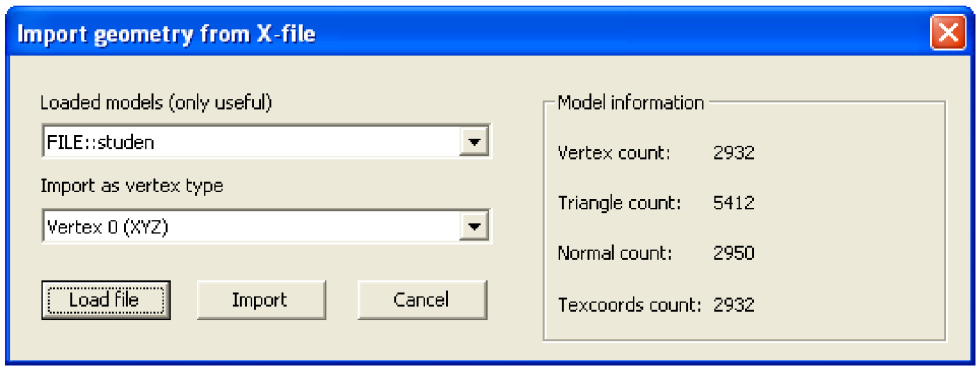


Obrázek A.3.1 – okno aplikace pro tvorbu souborů s geometrií

V klientské části aplikace je znázorněn model, kterým lze rotovat pomocí myši, příp. měnit vzdálenost pozorovatele od modelu pomocí kolečka myši. Ve stavovém panelu v dolní části aplikace jsou informace o aktuálně načteném modelu v tomto pořadí – typ vrcholů modelu, velikost jednoho vrcholu v bajtech, počet vrcholů a počet trojúhelníků.

Novou geometrii lze získat pomocí nabídky **File** a položky **Import X-file...**, která zobrazí dialog znázorněný na obrázku A.3.2. V tomto dialogu lze pomocí tlačítka **Load File** načíst soubor s modelem ve formátu DirectX. Pokud se soubor správně načte pak jsou jeho jednotlivé části obsaženy ve výběru **Loaded models**. Informace o vybrané části modelu jsou pak umístěny v sekci **Model information**. Po výběru požadované části modelu stačí vybrat příslušný typ vrcholu a následně pomocí tlačítka **Import** načíst model do hlavní aplikace. Tam je již možné model uložit

pomocí volby **Save** z nabídky **File**. Z nabídky **File** hlavní aplikace lze také načíst již uložený soubor s geometrií pomocí volby **Open**.



Obrázek A.3.2 – dialog pro import geometrie

A.4 Obrazová příloha

Obrázek A.4.1 – scéna definovaná s konfigurací Night a bodovým světlem

Obrázek A.4.2 – stejná scéna jako na obrázku A.4.1, ale v konfiguraci Sun