FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Vizualizace rozsáhlých urbanistických scén
Student:	Giang Chau Nguyenová
Vedoucí:	Ing. Jiří Chludil
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

1. Analyzujte problematiku vizualizace velmi rozsáhlých urbanistických scén a technik pro její optimalizaci. Zaměřte se zejména na LOD, blokové načítání a viditelnost.
2. Na základě analýzy vybraných technik navrhněte a implementujte prototyp pro vizualizaci rozsáhlých urbanistických scén v Unreal engine.
3. Zaznamenávejte hlavní vizualizační parametry u navrženého prototypu (framerate, bandwidth, ...) a vyhodnoťte účinnost optimalizačních technik.
4. Prototyp podrobte uživatelskému testování (zaměřte se na vnímání kvality zobrazení uživatelem).

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 13. prosince 2017

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Vizualizace rozsáhlých urbanistických scén

Giang Chau Nguyenová

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jiří Chludil

14. května 2018

Poděkování

Ráda bych poděkovala Ing. Jiřímu Chludilovi za odborné vedení a konzultace po celou dobu mé práce. Dále děkuji své rodině a přátelům za neustálou podporu během celé doby mého studia.

Prohlášení

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

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

V Praze dne 14. května 2018 .....

Odkaz na tuto práci

Nguyenová, Giang Chau. *Vizualizace rozsáhlých urbanistických scén.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato práce je zaměřená na optimalizaci vizualizace rozsáhlých urbanistických scén a obsahuje přehled několika metod, které postupně rozebírám a porovnávám podle zvolených kritérií. Zaměřuji se především na metody úrovně detailu modelu a viditelnost ve scéně. Na základě podrobné analýzy zkoumaných algoritmů jsem implementovala mnou vybrané metody a prototyp zobrazila pomocí herního jádra Unreal Engine. U navrženého prototypu jsem zaznamenala hlavní vizualizační parametry a vyhodnotila účinnost jednotlivých technik.

Klíčová slova rozsáhlé urbanistické scény, optimalizační metody, LOD – level of detail, zjednodušování sítě, viditelnost, Unreal Engine

Abstract

This thesis focuses on optimizing the visualization of large urban scenes and contains an overview of several methods, which I analyze and compare by chosen criteria. I mainly focus on level of detail methods and visibility of the scene. On the basis of the detailed analysis of examined algorithms, I have implemented selected methods and visualized the prototype using the game engine Unreal Engine. I noted rendering parameters of designed prototype and evaluated the efficiency of each used technique.

Keywords large urban scenes, optimization methods, LOD – level of detail, simplifying mesh, visibility, Unreal Engine

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Vykreslování rozsáhlých a komplexních scén v reálném čase se v posledních letech stává důležitou součástí počítačové grafiky a jejím hlavním (a ne jediným) cílem je navodit vizuální dojem skutečnosti. Bohužel pro vizualizaci takto velkých prostředí je potřeba se vypořádat se spoustou problémů. 3D interaktivní scény často vyžadují složité výpočetní modely, které je potřeba nasimulovat a poté vhodně zobrazit tak, aby zajistily plynulý pohyb ve scéně. Každý počítačový grafik se musí vypořádat se záležitostmi jako je reálnost zobrazovaného prostředí na úkor rychlosti jeho vykreslování, scény by navíc měly být věrohodné a frekvence snímků by ideálně neměla klesnout pod určitý práh. Přestože jsou v současnosti grafická vybavení pro počítač na vysoké úrovni, virtuální scény jsou pořád složitější (hlavně co se týče počtu polygonů modelu) a efektivnost jejich vykreslování pomocí grafických karet je stále nižší, než je vyžadována.

Naštěstí existují metody a algoritmy pro snížení výpočetní složitosti bez větších ztrát na vizuální kvalitě scén a každá technika se snaží uchopit problém z jiného pohledu. Jeden ze způsobů je například zaměřen na předpovídání pohybu pozorovatele. Pokud se pozorovatel nachází v nějaké části virtuálního města a chodí (nebo se dívá) určitým směrem, dává smysl si předem načíst další oblasti, které mají vysokou šanci být v nejbližší době navštíveny. Díky tomu je možné zajistit i fixní rychlost aktualizace a procházení městem je o něco plynulejší. Tento přístup je hodně spjatý i s viditelností různých objektů. Části, které uživatel vůbec nevidí – je k nim otočený zády, nebo jsou zakryté jiným objektem – se vůbec nevykreslí a tím se zmenší výpočetní náročnost scény. Další z nejčastějších metod je LOD – *level of detail*. Již od poloviny sedmdesátých let minulého století používali programátoři techniky pro zjednodušování modelů ke zvýšení efektivity svých grafických aplikací. Kromě toho se vyskytly i další výhody jako jsou snížené náklady pro ukládání nebo rychlejší přenos po síti.

Způsobů je tedy opravdu mnoho a v mojí bakalářské práci bych se chtěla zaměřit na několik z nich, jak ze stránky teoretické, tak i z praktického po-

hledu a získat si tak obecný přehled o nejmodernějších poznatcích v oblasti počítačové grafiky. Teoretická část práce je také součástí projektu VHP – Virtuální historický průvodce, kde hlavním přínosem mojí práce je prozkoumání různých možností optimalizace, které by mohly být v budoucnu použity.

Kapitola 1

Cíl práce

Cílem mojí bakalářské práce je prozkoumání stávajících technik pro optimalizaci vizualizace rozsáhlých scén a navzájem porovnat účinnost a výhody i nevýhody jejich použití. V práci se hlavně zaměřím na metody úrovně detailu (*level of detail*) a na viditelnosti objektů ve scéně.

Analytická část by mi měla pomoci v návrhu vizualizace města v projektu VHP (Virtuální historický průvodce) a také k vytvoření prototypu v herním jádru Unreal Engine. V prorotypu budou použity mnou vybrané optimalizační algoritmy. Na základě vyrobeného prototypu vyhodnotím účinnost jednotlivých optimalizačních technik a zaznamenám jeho hlavní vizualizační parametry.

Kapitola 2

Optimalizace 3D scén

Všechny simulace rozsáhlých virtuálních scén sdílejí jeden zásadní problém a tím je složitost jejich zobrazovaných dat. Množina dat je poměrně velká a složitá na to, aby je hardware zvládl zobrazit v reálném čase bez nějakých komplikací. Dosavadní řešení se snaží zredukovat vykreslované prvky (např. geometrii) scény na úrovni programu, aby snížily nároky na výkon samotného hardwaru. Redukování a omezování prostředků se však musí provádět s určitou rezervou (s ohledem na estetiku), aby měl pozorovatel dobrý zážitek z procházení virtuální scénou. Tudíž je potřeba prozkoumat různé aspekty ve 3D scéně, jež mohou ovlivňovat výkon celé simulace a rozhodnout se pro nejlepší způsob optimalizace. Kombinování různých optimalizačních metod umožní uživateli si tyto virtuální místa zobrazit i na běžné domácí počítačové sestavě.

Pro urychlení zobrazování velkých 3D scén bylo vyvinuto mnoho metod, které jsou podrobněji popsané v dalších kapitolách (kapitoly 3 a 4). Metody lze rozdělit do dvou kategorií: metody zjednodušující scénu a metody, jež vypočítávají viditelnost objektů ve scéně [1].

První skupina vychází z myšlenky, že pro pozorovatele vzdálené nebo nedůležité části scény není nutné zobrazit v nejvyšších detailech. Tyto detaily jsou při určité velikosti nerozlišitelné nebo jim uživatel nevěnuje pozornost, protože vnímá především objekty v popředí. Druhá skupina se soustřeďuje na rychlé určení viditelných částí scén. Z pozice, kde pozorovatel stojí, lze obvykle vidět jen některé objekty z celé scény. Velkou množinu objektů není třeba zpracovávat, ať již proto, že se nacházejí mimo zorný úhel nebo jsou zakryty (zastíněny) objekty v popředí [1].

S metodami určení viditelnosti jsou těsně spjaté další možnosti optimalizace, jež se zaměřují hlavně na způsob zacházení s obrovskými daty (metody jsou popsány v kapitole 5). Tyto techniky si rozdělí rozsáhlý prostor scény, který poté uloží do vhodné datové struktury. Využitím těchto datových struktur lze určení viditelnosti ve scéně značně urychlit.

2.1 Počet polygonů ve scéně

Vše co se ve virtuální scéně vykreslí záleží na pohledu uživatele a jeho perspektivě. Při modelování scén je doporučeno odstranit zbytečné části modelu, které ve výsledku nebudou vůbec vidět, a zvážit zda jsou jednotlivé části modelu natolik důležité aby zabíraly prostředky (paměť, výpočetní čas...). Modely objektů by neměly mít přehnaně vysoký počet polygonů, ne všechny detaily na modelu musí být vymodelované (můžou se jednoduše nahradit namapováním textur) [2]. Cílem optimalizace je snížit počet vykreslených polygonů na aktuální scénu bez výrazného omezení vizuální kvality.

2.2 Zátěž hardwaru při vykreslování scény

Vykreslování (*rendering*) 3D scén hodně záleží na výkonu hardwaru, zejména grafické karty (GPU), procesoru (CPU) a velikosti paměti. Podle Sjoerda De Jonga (2006) [3] je při zobrazování scény důležité prostředky zatížit zhruba stejně (jak procesor, grafickou kartu tak i paměť).

Procesor je především zodpovědný za aktualizaci a výpočet snímků 3D scény. Grafický procesor se stará o vykreslení polygonů a textur, manipuluje s podprogramy *shaders* v zobrazovacím řetězci (*vertex shader, fragment shader...*) a zajišťuje další doplňující efekty (*post-processing effects*) [4]. Paměť omezuje počet různých originálních modelů k vykreslení, zatímco opakované modely zatěžují spíše procesor a grafickou kartu. De Jong na své webové stránce píše, že paměť přispívá k hladké aktualizaci snímků (bez rušivých záseků při pohybu virtuální scénou) a CPU/GPU zase zvyšuje rychlost jejich vykreslení [3].

Pokud jde o samotné vykreslení objektu do scény v reálném čase pomocí herního jádra (*game engine*), tak se objekt nejprve rozdělí podle materiálů aplikované na model a jádro jednotlivé části vykreslí postupně. Model s jedním materiálem se vykreslí jako jeden celek oproti modelu se dvěma nebo více materiály, který se musí vykreslit vícekrát, pokaždé s jiným materiálem. Způsob určení rozdělení a pořadí vykreslování jednotlivých části vyžadují výkon a čas ze strany procesoru a grafické karty. Další důležitou věcí je si uvědomit, že každá síť modelu se vykresluje samostatně. Obojí, síť a materiály, se rozdělí do sektorů a neustále je požadováno volání vykreslovací funkce (*draw calls*), což značně zatěžuje hardware. Z tohoto důvodu je dobré (pokud možno) si objekty zkombinovat do většího celku [3].

Regulování rozumného množství textur je dalším způsobem optimalizace, jelikož každá textura vyžaduje jedno zavolání vykreslení. Někdy jsou textury náročnější na vykreslení než samotná síť modelu. Unikátní textura v paměti také zabírá více místa než jedna síť modelu, proto je vhodné unikátní modely ukládat v menším počtu polygonů a opakující se objekty ve scéně ukládat ve větším rozlišení. Přestože se jedná o snížení výpočetního času v řádu

milisekund, pro hardware (pracující se stovkami modelů) je to ušetřený čas potřebný k vykreslení požadovaného množství snímků za sekundu (*frames per second* – FPS) [3].

2.3 Rozsah optimalizace

Rozsah optimalizace je dán především velikostí scény a spektrem vidění. Příkladem může být rozlehlé prostředí zahrady (terénu), které potřebuje víc optimalizace, jelikož je vykreslená scéna mnohonásobně větší než třeba scéna v uzavřené místnosti, kde není optimalizace až tak důležitá, pokud obsahuje menší počet objektů (tudíž lze ve scéně využít detailnější modely). Guillaume Provost (2003) toto popisuje v návrhu scén (jednotlivých úrovní) pro počítačové hry [5].

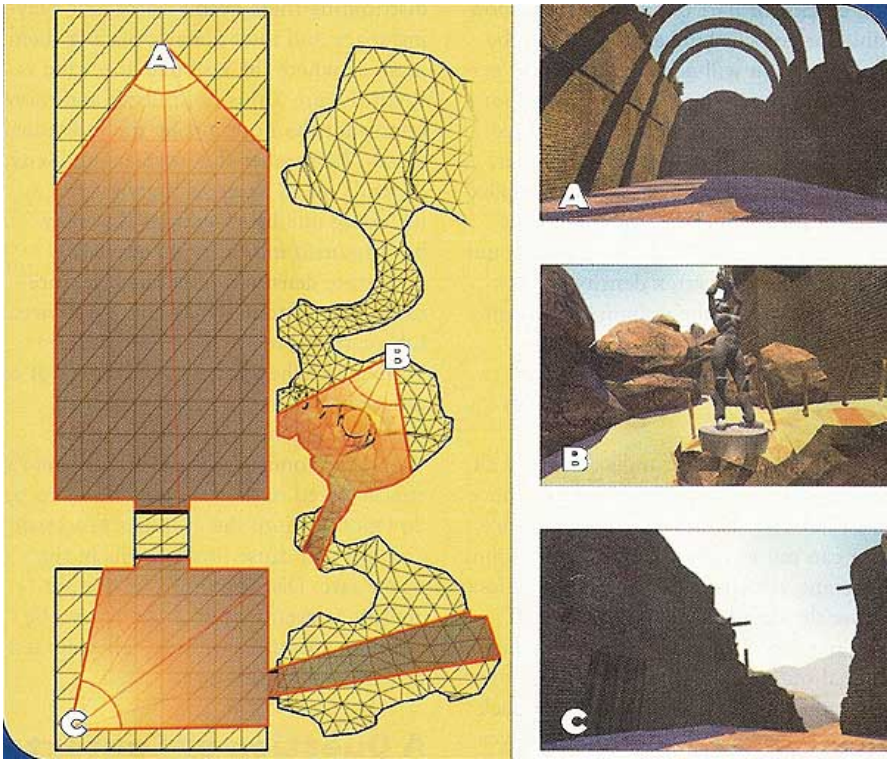
Pro dosažení dobrého konstantního výkonu při vykreslování scén je zapotřebí brát v úvahu její hustotu vrcholů, hustotu textur a viditelnost. Hustota vrcholů je dána počtem vrcholů na model (síť). Malá oblast ve scéně zaplněná vysokým počtem modelů s hustou sítí sníží výkon vykreslování, proto je nutné tyto modely rovnoměrně rozprostřít po celé oblasti. Pokud se model s hustou sítí vykreslí v menší oblasti (s omezeným spektrem viditelnosti), nebude potřeba optimalizaci více řešit (obrázek 2.1) [5].

Hustotou textur je myšlen počet textur uložené v paměti pro aktuální oblast ve scéně. Nehodí se soustředit mnoho textur na jednu oblast, jelikož se opět může snížit výkonnost vykreslování kvůli neustálému načítání a mazání textur do a z paměti (hlavně v případech s omezenou pamětí) – tomuto jevu se říká *bottleneck* (nestíhají se dodávat data ke zpracování). V UDK (*Unreal Development Kit*) je například omezený počet pro použité textury (ukládané do paměti), tzv. *texture pool*, tudíž nelze texturami zbytečně plýtvat [6].

Spektrum viditelnosti je všechno co lze v určité scéně vidět (zobrazí se jako pixel na obrazovku). Počet vrcholů a počet textur, jež je možné vložit do scény je konstantní, tudíž čím větší je scéna tím nižší je hustota detailů ve scéně a naopak. Jedná se o jednu z nejdůležitějších složek ovlivňující výkon vykreslování. Běžné techniky, které snižují spektrum viditelnosti jsou například blokování výhledu nebo mlha s efektem hloubky ostrosti (na externí neuzavřená prostředí) [5]. Optimalizování scén na základě viditelnosti je dále rozebíráno v kapitole 4.

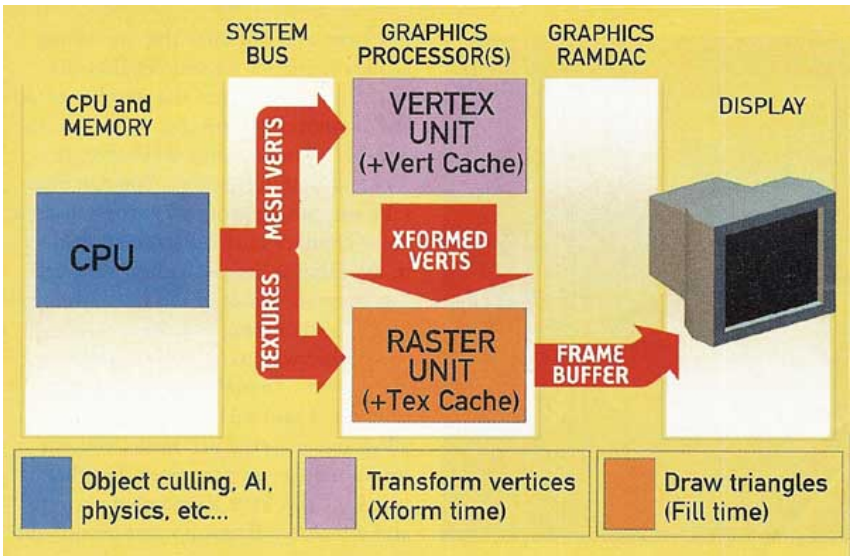
2.4 Nutnost optimalizace

Během renderování scén se transformují vrcholy v síti modelu a vykreslují se trojúhelníky. Tyto dvě operace se dějí paralelně, pomalejší z nich určuje rychlost vykreslení celého objektu. K posouzení toho, zda je nutné optimalizovat, se určí podle jevu *bottleneck* a nákladů na vykreslení. Náklady na vykreslení rozhodují o nutnosti optimalizace, zatímco *bottleneck* určí co se má optimalizovat.



Obrázek 2.1: Kamera B má narozdíl od kamery A a C menší spektrum viditelnosti a díky tomu je možné zobrazit sochu ve vysokých detailech do scény bez nutnosti větší optimalizace [5]

Pokud *bottleneck* způsobuje pomalá transformace vrcholů (při vykreslování) pak se jedná o *vertex-bound* síť (rychlost vykreslování je omezená kvůli transformaci vrcholů) – nákladem je pak vynaložený čas pro transformaci. Naopak pokud *bottleneck* způsobuje vykreslování trojúhelníků označuje se tato síť jako *fill-bound* (rychlost je omezená kvůli vyplnění sítě povrchem) – nákladem je čas pro vykreslení těchto trojúhelníků (znázorněné na obrázku 2.2). Sítě typu *fill-bound* vyžadují jiná pravidla a způsob optimalizace než sítě typu *vertex-bound*. Zjednodušeně je tedy buď optimalizována geometrie modelu (primárně u *vertex-bound*) nebo materiál a textura namapovaná na model (*fill-bound*) [5].

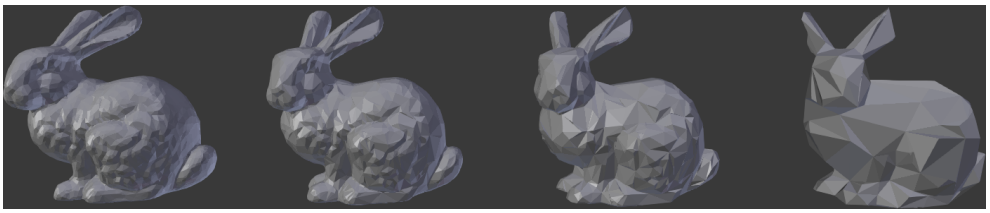


Obrázek 2.2: Typická hardwarová roura pro vykreslování a její možný *bottleneck*: (**1**) *Vertex-bound*: vrcholy se nestíhají transformovat, (**2**) *Fill-bound*: rasterizér nestíhá vykreslovat trojúhelníky, (**3**) *Data-bound*: sběrnice nestíhá dodávat data ke zpracování, (**4**) *CPU-bound*: procesor pracuje s příliš mnoho objekty nebo je zaneprázdněn jinou logikou v programu [5]

Kapitola 3

Level of detail

Metody LOD (*level of detail*) jsou optimalizační techniky, které regulují množství detailu v modelech vykreslených do virtuálního prostředí. Každý 3D objekt je reprezentován v různých úrovních detailu (obrázek 3.1). V této hierarchii je obvykle pět až deset úrovní, nejvyšší LOD reprezentuje objekt se všemi detaily, nejnižší LOD pak co nejvíce zjednodušenou reprezentaci, která je ještě vizuálně akceptovatelná jako náhrada původního objektu. Úrovně LOD se vytvářejí především kvůli snížení nákladů na vykreslení malých, vzdálených nebo nepodstatných objektů ve scéně.



Obrázek 3.1: Model stanfordského králíka: (**1**) 7638 polygonů, (**2**) 3846 polygonů, (**3**) 1527 polygonů, (**4**) 371 polygonů

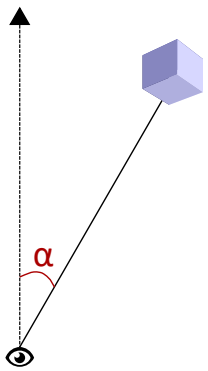
Co je vlastně myšleno detailem modelu? Podle Jamese H. Clarka (1976) je to počet polygonů reprezentující objekt [7], pak by se jednalo o geometrické LOD. Čím víc polygonů model obsahuje, tím vyšší úroveň detailu má a naopak. Avšak geometrie není jediný způsob vyjádření detailu trojrozměrného objektu. Existují například úrovně detailu v animacích, kdy pro pohyb méně důležitých objektů se používají jednodušší výpočty, což samozřejmě vede k hrubším výsledkům. Další aproximace se můžou provádět při osvětlování scény. Bodová světla jsou v určité vzdálenosti simulována světly směrovými či dokonce okolními (*ambient*) bez významnějších vizuálních změn pokud jsou osvětlované objekty dostatečně malé [8]. Zmíněné příklady zdaleka netvoří úplný seznam reprezentací detailů modelu.

Přechod mezi jednotlivými úrovněmi detailu je založen na měnících se podmínkách při zobrazování scény. Hlavní podmínkou pro změnu LOD byla původně vzdálenost mezi objektem a středem pohledu, v současnosti však existuje mnoho kritérií, z nichž hlavních pět je uvedeno v následujícím seznamu [9]:

vzdálenost LOD objektu je určeno podle euklidovské vzdálenosti oka pozorovatele od předdefinovaného bodu uvnitř objektu. Jedná se o nejběžnější a nejjednodušší použití LOD, kdy objekt dále od pozorovatele se vykresluje v hrubějších detailech než bližší objekty.

velikost Kritériem určení LOD je počet pixelů, v nichž je daný objekt na obrazovce vidět (velikost projekce). Jednoduše se jedná o další způsob realizace LOD na základě vzdálenosti, hojně využívaný ve vizualizaci terénů v reálném čase. Jeden z mnoha přístupů je popsán Peterem Lindstromem (1995). Jeho model terénu se rekurzivně dělí na čtvrtiny pomocí *quadtree* struktury a podle předem nastaveného prahu (jedná se o promítnutou vzdálenost mezi sousedními vrcholy, jež je nižší než nějaký počet pixelů) se rozhodne, do jaké hloubky strom poroste pro danou oblast terénu (což určuje „poskytnuté množství“ detailu pro tento region) [10].

výstřednost Bez vhodného systému pro sledování pohybu očí uživatele se obecně může předpokládat, že se uživatel bude dívat spíše doprostřed obrazovky (obrázek 3.2). Detailněji se tedy zobrazují objekty blíže ke středu a objekty směrem ke stranám obrazovky se vykreslí v nižším rozlišení. Další techniky mimo jiné detailněji zobrazují nějakou oblast zájmu (*area of interest*) než aby se držely centra obrazovky (např. metoda zaměřená na pohled pozorovatele od Marca Levoye a Rosse Whitakera [11]).



Obrázek 3.2: Výstřednost objektu je dána úhlem od směru pohledu pozorovatele

rychlost LOD objektu se mění podle rychlosti jeho pohybu v zorném úhlu pozorovatele. Pomalé či statické objekty jsou detailnější než rychle se pohybující objekty, které uživatel vidí jen na krátký okamžik. Thomas A. Funkhouser a Carlo H. Séquin (1993) snižovali stupně LOD podle poměru rychlosti objektu ku průměrné velikosti polygonu v modelu objektu [12]. Toto regulování množství detailu na modelu se zdá být rychlé a efektivní, i když je to samozřejmě vjemově nepřesné [9].

stálá frekvence snímků Toto hledisko pro posuzování LOD objektu se liší od dříve vyjmenovaných tím, že se spíše zaměřuje na výpočetní optimalizaci a ne na optimalizaci vnímání. Detaily modelu se regulují hlavně kvůli dosažení a udržení předepsané rychlosti aktualizace snímků scény.

Jakýkoliv systém, který implementuje tento způsob určení LOD musí obsahovat plánovač, jehož úkolem je analyzovat zatížení scény a přiřadit LOD hodnotu každému objektu odpovídajícím způsobem. Existují dva hlavní typy plánovačů: reaktivní a prediktivní. Reaktivní plánovač je ten jednodušší a nezaručuje omezenou rychlost snímků – zkrátka jen upravuje hodnoty LOD objektu podle toho, zda se poslední vykreslený snímek ještě započítal do cílového počtu snímků nebo jestli byl vykreslen až po uplynutí doby. V tom prvním případě se detaily mohou navýšit (protože se snímky stíhají vykreslovat), v opačném případě se detaily musí omezit. Naproti tomu je prediktivní plánovač, který u následujícího snímku odhadne jeho složitost a podle toho přiřadí hodnotu LOD objektům tak, aby nedošlo k překročení limitu počtu snímků za určitou dobu [9].

Příkladem reaktivního plánovače je systém Viper, popsaný Richardem L. Hollowayem v roce 1992, ve kterém dojde k ukončení vykreslování, jakmile se systém přetíží [13]. Kromě toho se ještě definuje jednoduché prioritní schéma určující důležitost objektu a pokud dojde k nechtěnému přerušení, bude tento prioritní objekt vykreslen vždy a to v nejvyšších možných detailech. Funkhouser a Séquin (1992) zase použili ve svém systému (pro průchod architektonickou scénou [12]) prediktivní plánovač vyvažující poměr přínosu vykreslení daného snímku a vynaloženého výpočetního úsilí na jeho zobrazení.

3.1 Typy LOD

Kromě způsobů určení stupně LOD objektu existují v současné době čtyři druhy geometrického LOD: diskrétní, spojitý, závislý na pohledu a hierarchický [14]. Tyto typy pracují s polygonální reprezentací objektů, například s tělesy či terénem. Druhá skupina LOD (pohledově závislé a hierarchické LOD) využívá i obrazové, tedy především dvojrozměrné informace [1].

3.1.1 Diskrétní stupně LOD

Metodu navrhl v roce 1976 James H. Clark a je používána bez větších modifikací až dodnes. Tento tradiční diskrétní přístup předem vytvoří sadu LOD reprezentací pro každý objekt zvlášť. Objekty se zjednodušují na úrovni celého povrchu objektu (nezávisle na budoucím úhlu pohledu pozorovatele) a proto se často diskrétní LOD označuje jako izotropní nebo jako LOD nezávislé na pohledu (*view-independent LOD*) [15].

Každé diskrétní úrovni modelu je přiřazeno číslo představující vzdálenost a při zobrazování dochází k přepnutí na další stupeň. Nevýhodou je redundance dat a skokové přecházení z jedné reprezentace do druhé kvůli chybějícím informacím o vztahu jednotlivých částí modelu [1]. Naopak podstatná výhoda diskrétního LOD spočívá v oddělení dvou operací: zjednodušování modelu a následně jeho vykreslení do scény. Algoritmus zjednodušování trvá tak dlouho dokud nevytvoří potřebná LOD a během vykreslování scény se pak jen podle definovaných podmínek vybere příslušná úroveň detailu pro každý objekt. Současný grafický hardware si tyto statické modely předem sestaví a zpracuje do optimálního vykreslovacího formátu, s použitím trojúhelníkových pruhů (*triangle strips*) a polí vrcholů (*vertex arrays*), což je výrazně rychlejší než neseřazená množina polygonů. Díky snadné implementaci a jednoduchosti při rozhodování se o změně úrovní je tento postup stále využíván, hlavně v aplikacích virtuální reality [9].

3.1.2 Spojité stupně LOD

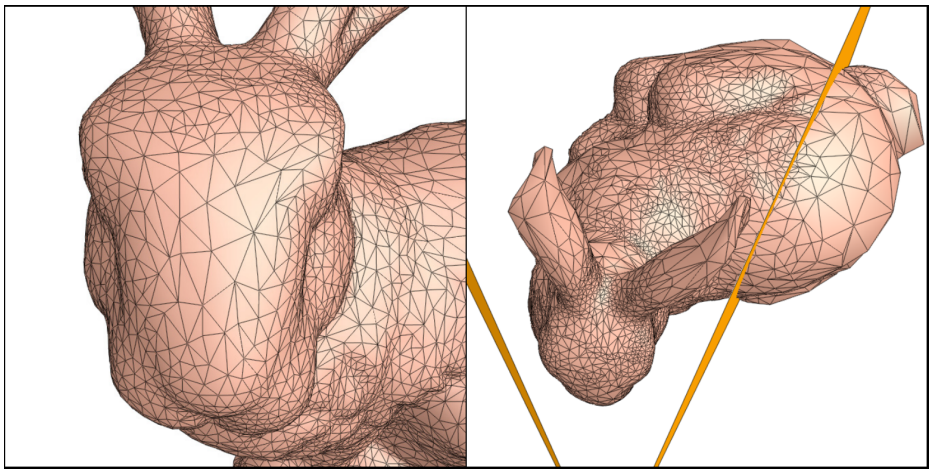
Spojité LOD na rozdíl od diskrétního uchovává datovou strukturu, ze které lze získat požadovanou úroveň detailu přímo za běhu programu, typicky přidáním či odebráním trojúhelníků z reprezentace. Hlavní výhodou tohoto přístupu je lepší granularita: úroveň detailu je u každého objektu určena přesně, nemá přebytečné trojúhelníky a tudíž se „ušetří“ pro vykreslování dalších objektů – to samozřejmě vede k efektivnějšímu využití zdrojů a celkové vizuální věrohodnosti. Kromě toho, pokud je nutné načíst nějaký rozsáhlý model z disku nebo přes síťové spojení, podporuje tato metoda postupný přenos jednotlivých polygonů (*progressive transmission*) a model je postupně dynamicky zlepšován a vykreslován [15].

Algoritmus pro zjednodušování sítě (*mesh*) je vhodné provést již ve fázi předzpracování a jeho výsledky uložit do vhodné datové struktury tak, aby při vykreslování nebylo nutné provádět složité geometrické výpočty, ale pouze jen měnit počet trojúhelníků. Protože se uchovávají různé reprezentace, vyžaduje tato datová struktura větší paměťové nároky než statická síť trojúhelníků [1].

Spojitému LOD se také říká progresivní, podle datové struktury *progressive mesh*, navržená Huguesem Hoppem (1996) [16]. Struktura zachycuje všechny jednotlivé kroky pro zjednodušení modelu, tudíž lze během zobrazování postupovat oběma směry (tedy zkvalitnit nebo zjednodušit popis modelu).

3.1.3 LOD závislé na pohledu

Toto LOD rozšiřuje spojité stupně LOD, používá různé parametry závislé na pohledu k volbě nejvhodnější úrovně detailu pro aktuální scénu. Počítá se i s periferním viděním pozorovatele a dobře pracuje s velkými objekty. Části modelu blíže k pozorovateli jsou zobrazovány s více detaily než vzdálenější části. Jeden objekt tedy může mít několik přechodů LOD zároveň (obrázek 3.3). Výhodou tohoto LOD je opět lepší granularita, tentokrát už i na úrovni jednoho objektu [15].



Obrázek 3.3: LOD závislé na pohledu: síť ve směru pohledu je hustší [17]

3.1.4 Hierarchické LOD

Samotné LOD závislé na pohledu není dostačující pro zobrazení velice malých objektů, k tomu je vhodné použít hierarchické LOD. Tento postup spojuje menší objekty do společného útvaru a určuje úroveň detailu celého shluku, nikoliv jednotlivě po objektech. Celou scénu zpracovává jako jeden objekt, který pak zjednodušuje pomocí pohledově závislého LOD, což umožňuje lepší škálovatelnost rozsáhlých strukturovaných modelů – snižuje počet vykreslovaných objektů v celé scéně (nemusí se tak často volat vykreslovací funkce na každý snímek) [14]. Kromě toho je toto LOD použito i v diskrétním LOD, kde se jednotlivé úrovně detailu mohou seskupit do hierarchie.

3.2 LOD používaná v praxi

Tradiční způsob diskrétního LOD zůstává stále nejčastějším přístupem v praxi, i přes výhody spojitého a pohledově závislého LOD. Diskrétní stupně LOD se vytvoří předem (*preprocessing*) a současný grafický hardware si s tím pak jednoduše poradí. Naopak LOD závislé na pohledu vyžaduje další zpracování

navíc pro zjednodušení a zdokonalení modelu v době běhu programu a potřebuje další operace k vyhodnocení vizuálních parametrů. Pokud je tedy systém například omezený rychlostí procesoru tak toto vedlejší zatížení může snížit frekvenci vykreslování snímků a přínos použití LOD se tolik neprojeví [15].

Datové struktury spojitého LOD a LOD závislého na pohledu mají také vyšší paměťové nároky než struktura ukládaná v diskrétním LOD, což spoustu vývojářů může odradit (například ve videohrách se přístup pohledově závislého LOD nepoužije dokud je to nezbytně nutné pro vykreslování rozsáhlých terénů) [15]. Spojité LOD je ale na rozdíl od pohledově závislého LOD populárnější, používá ho například herní jádro Unreal Engine pro správu objektů ve hře.

3.3 Nevýhody LOD

Technika LOD je jeden z hlavních nástrojů pro snížení výpočetní náročnosti a regulování detailů ve scéně. Bohužel s jeho použitím se pojí několik problémů, které je potřeba nějakým způsobem řešit [9]:

- V současné době neexistuje jednoznačný mechanismus pro výběr optimálního LOD. Většina systémů využívá metody pokus-omyl v kombinaci s podmínkami přechodu úrovní LOD (popsané v předešlé části) nebo používá specifickou heuristiku pro konkrétní případ. Například Funkhouser a Séquin [12] uplatnili při procházení architektonické scény heuristiku, která odhadovala důležitost modelu podle lidského vnímání. Autoři tvrdí, že některé typy objektů mohou mít ve scéně neodmyslitelnou důležitost a pokud by kvalita těchto objektů měla být nějakým způsobem degradována, drasticky by to ovlivnilo zážitek pozorovatele.

  Ačkoliv jsou tyto metody dostačující pro implementaci rozumného LOD, určitě by bylo dobré mít obecný mechanismus pro výběr LOD vycházejícího z formálnějšího řešení.

- Přímým důsledkem chybějícího jednotného postupu pro výběr úrovně LOD je výrazné „vyskakování“ částí objektu (tzv. *popping effect*) při neustálém přepínání mezi dvěma reprezentacemi objektu [1]. Tento nežádoucí účinek způsobuje vizuální artefakty a může pozorovatele značně rozptylovat. V sekci 3.5 jsou popsána možná řešení blikajícího efektu při přechodu z jedné úrovně detailu do druhé.

- Samotné generování různých úrovní detailu určitého modelu je složitým procesem. Je spousta decimačních postupů, které z komplexního polygonálního modelu vytvoří sadu jednodušších modelů, které se podobají originálu (jednotlivé techniky jsou dále rozebírány v sekci 3.4). Bohužel nejsou zcela bezchybné. Na automaticky vygenerovaných modelech lze pozorovat nejen geometrické chyby ale i chyby jako jsou špatné barvy

nebo jinak nakloněné normály namapovaných textur. Tyto modely je potřeba dále zpracovat nebo ručně upravit, což může být pro rozsáhlejší modely docela náročné. Tato práce je převážně zaměřená na upravování geometrie v síti modelu, materiály a texturami modelu se příliš nezabývá.

3.4 Zjednodušování sítě modelu a generování stupně LOD

V moderní počítačové grafice jsou objekty reprezentovány hlavně pomocí geometrických útvarů, a to především kvůli jejich částečné lineární aproximaci tvaru. Jedná se o nejčastější reprezentaci modelu pro všechny možné aplikace, od počítačových her až po vizualizaci lékařských, vědeckých a CAD (computer-aided design) dat [15].

Tyto modely objektů, získané například převodem objemové reprezentace do povrchové nebo vzniklé trojrozměrným snímáním, je potřeba v LOD nějakým způsobem zjednodušit – zmenšit jejich obrovské množství polygonů pro případnou další manipulaci. Při snižování počtu trojúhelníků v síti však dochází většinou ke zhoršení kvality, neboť decimační algoritmy postupně vynechávají méně důležité prvky modelu [1].

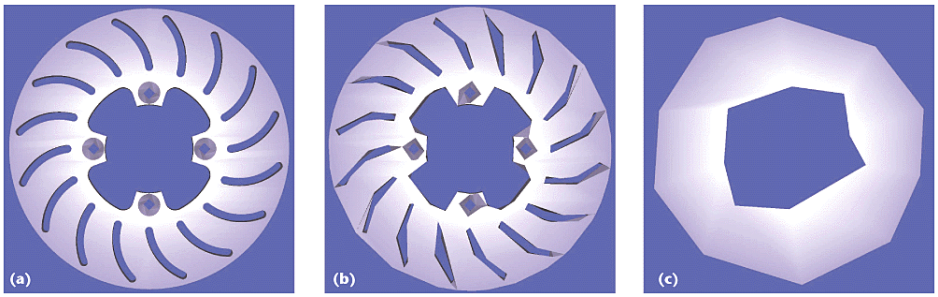
Metody zjednodušování povrchu lze dělit podle toho zda zachovávají topologii (údaje o struktuře sítě modelu – například které vrcholy tvoří trojúhelník, jaké trojúhelníky spolu sousedí, zda síť má otvory atp.), jestli ponechávají podmnožinu primitiv objektu (vrcholy, hrany, trojúhelníky...) nebo zda je zjednodušená síť nově vytvořená převzorkováním té původní [14].

Algoritmy jež nemění topologii sítě, zachovávají manifoldy[1] v každém kroku, neuzavírají ani nevytvářejí nové otvory v síti a tím pádem jsou zjednodušené objekty relativně vizuálně věrohodné. Toto „omezení“ nicméně limituje možnosti zjednodušování, jelikož objekty s mnoha štěrbinami nemohou být zmenšeny pod nějaký počet polygonů, aniž by nějaké původní otvory nezmizely (obrázek 3.4) [15].

Na druhou stranu, algoritmy měnící topologii povolují tyto drastická zjednodušování provádět za cenu horší vizuální kvality a problikávajících efektů způsobené objevováním a zase mizením děr v síti [15]. Většina těchto algoritmů však nevyžadují upravenou topologickou síť (obvykle je vyžadována trojúhelníková síť a aby hrany v síti byly sdílené maximálně dvěma trojúhelníky atd.), což výrazně zvyšuje jejich použitelnost.

Mnoho zjednodušovacích algoritmů však předpokládají na vstupu sítě složené z trojúhelníků. Trojúhelníky jsou oproti obecným polygonům výhodnější

[1] Pojem *manifold* (někdy také *2-manifold*) popisuje takové modely těles, které odpovídají nějakému skutečnému tělesu. Obdobně *non-manifold* označuje tělesa, které nelze ve skutečném světě vyrobit.



Obrázek 3.4: (**a**) originální model brzdového rotoru: 4736 trojúhelníků, 21 děr, (**b**) zachovaná topologie při zjednodušování: 1006 trojúhelníků, 21 děr, (**c**) nezachovaná topologie při zjednodušování: 46 trojúhelníků, 1 díra [18]

kvůli jejich konstantní velikosti v paměti a zaručené plošnosti [15]. Tyto modely lze získat například triangulací jejich sítě. Jeden z triangulačních algoritmů je například Seidelova iterační triangulace polygonů (které samy sebe nikde neprotínají) pracující s n vrcholy v čase $\mathcal{O}(n \log * n)$ – v praxi je algoritmus pro jednoduché objekty skoro lineární [15].

Algoritmy zjednodušování sítě bohužel neberou v potaz materiál na povrchu sítě. V moderní grafice se objekty obvykle nezobrazují jako jednoduché drátěné modely nebo nevýrazné jednobarevné útvary. Model objektu je většinou složen z barevných trojúhelníků nebo může mít na svém povrchu namapovanou barevnou texturu. Zkrátka metody pracují jen s geometrií modelu a vůbec se nehledí na jejich povrchové vlastnosti. Jeden z důvodů může být skutečnost, že většina metod byla vyvinuta hlavně pro lékařské aplikace, kde monochromatické struktury jsou velice běžné. Oproti tomu, oblast počítačové grafiky plně využívá barevnosti scén a zajímá se spíše o výkon vykreslení než o vizuální přesnost. Pro účely grafiky jsou určitě vhodnější algoritmy, jež topologii sítě nezachovávají [9].

Existuje různorodá škála zjednodušovacích algoritmů, mnoho z nich sdílejí společný základ a proto je těžké tyto metody nějakým způsobem systematicky klasifikovat. Carl Erikson (1996) tyto algoritmy dělí do tří obecných kategorií [19], příklady k vyjmenovaným kategoriím jsou dále v sekcích 3.4.1 až 3.4.3:

1. Odstraňování geometrie (*Geometry removal*) – jedná se o algoritmus, který odstraňuje vrcholy nebo celý trojúhelník z reprezentace objektu.

2. Vzorkování (*Sampling*) – metoda si navzorkuje geometrii modelu a poté se pokusí vygenerovat zjednodušený model odpovídající navzorkovaným datům.

3. Adaptivní rozdělení (*Adaptive subdivision*) – tento algoritmus začíná u jednoduchého základního modelu, rekurzivně ho rozděluje a v každé další iteraci se doplní nové detaily.

Cílem zjednodušování je vytvořit hierarchii sítí modelu s různým počtem trojúhelníků. Algoritmy, které zmíněnou hierarchii budují lze klasifikovat podle toho, jakým směrem postupují – odshora dolů (*top-down*) nebo zdola nahoru (*bottom-up*). V terminologii stromů vede směr odshora dolů od kořene k listům. Zjednodušování odshora dolů začíná od zjednodušené verze sítě a postupně přidává detaily podle určitých upravovacích pravidel (princip adaptivního rozdělení). Zjednodušování zdola nahoru naopak začíná u modelu s nejvyššími detaily (listy hierarchie) a opakovaně aplikuje modifikační operátory k získání sekvence postupně zjednodušených sítí (princip odstraňování geometrie) [15].

3.4.1 Princip odstraňování geometrie

Následující část práce popisuje několik algoritmů zjednodušování sítě založené na principu odstraňování geometrie. Tyto algoritmy zjednodušují síť odstraňováním vrcholů nebo celých trojúhelníků z reprezentace daného objektu.

Geometrická optimalizace

Příkladem zjednodušovacího algoritmu, který je založen na principu odstraňování geometrie, je geometrická optimalizace (*Geometric Optimisation*, Paul Hinker a Charles Hansen, 1993).

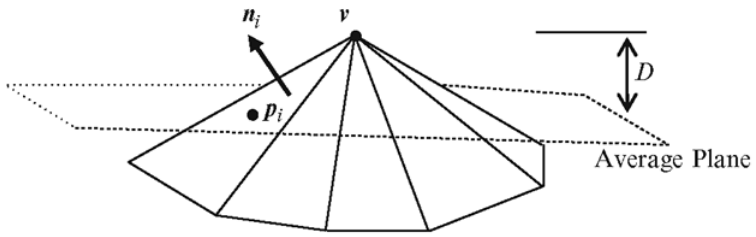
Metoda se pokouší sloučit skupinu koplanárních[2] trojúhelníků do jednoho. Nejdřív se seskupí trojúhelníky s přibližně stejnými normálami (normála je „stejná“ pokud je v nějakém akceptovatelném rozmezí úhlu vůči průměrné normále ostatních trojúhelníků ve skupině). Z okrajů (hran) nově vzniklé skupiny trojúhelníků se vytvoří nový útvar, který je pak triangulován aby byl povrch modelu jednotně teselován (pokryt) [20].

Tento přístup je vhodný pro modely s velkým počtem trojúhelníků a neovlivňuje jejich celkový vzhled. Na druhou stranu se tento algoritmus nehodí pro velmi zakřivené modely, protože by obsahovaly menší počet koplanárních instancí [9].

Decimace sítí trojúhelníků

Další metoda odstraňování geometrie je známá pod názvem *Decimace sítí trojúhelníků* (Schroeder a kolektiv, 1992). Tento decimační algoritmus odstraňuje vybrané vrcholy ze sítě postupným procházením vrcholů. Pro každý vrchol zjišťuje, jak moc je důležitý pro zachování tvaru v okolí vrcholu (podle vzdálenosti daného vrcholu od průměrné roviny jeho sousedních vrcholů, znázorněné na obrázku 3.5). Pokud je vrchol označen jako zbytečný (vzdálenost vrcholu je kratší než hranice stanovená uživatelem), tak je odstraněn (společně se všemi trojúhelníky, které obsahují tento vrchol) a vzniklá díra je vyplněná

[2] ležící ve stejné rovině



Obrázek 3.5: Vrchol v je nad průměrnou rovinou o vzdálenost D, tudíž splňuje decimační kritérium a je odstraněn [23]

triangulací. Tento proces se opakuje tak dlouho dokud se nedojde ke stanovenému procentuálnímu snížení (počtu trojúhelníků) nebo už nejsou v modelu vrcholy, jež by splňovaly decimační kritérium [21]. Autoři uvádějí, že tato metoda dobře pracuje s modely převedené z objemových dat pomocí algoritmu *Marching Cubes* (neboli „Pochodující kostky“, William E. Lorensen a Harvey E. Cline, 1987) [9].

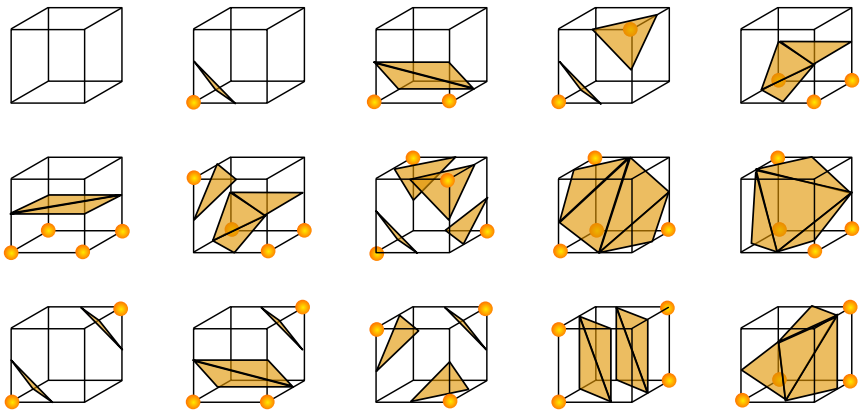
Iterační algoritmus Pochodujících kostek vytváří polygonální síť z implicitní funkce ve 3D skalárním poli. Algoritmus „pochoduje“ (iteruje) po celém 3D regionu, který byl předtím rozdělen do kostek (označují se jako voxely[3]). Pro každou kostku se vypočítá, zda jí povrch modelu (trojúhelník) prochází nebo ne. Pokud je všech 8 vrcholů kostky (voxely) rovno 1, znamená to, že kostka leží celá uvnitř povrchu modelu. V obdobném případě, pokud je všech 8 voxelů rovno 0, kostka leží mimo povrch modelu.

Cílem algoritmu je vypočítat trojúhelníkový povrch (v kostkách) v případech, kdy jsou některé voxely rovné 1 a jiné zase 0. Všech možností pro jednu kostku je 2^8 (pro 8 voxelů, nabývajících hodnot 0 nebo 1), z nichž mnoho je redundantních (kvůli rotační a zrcadlové symetrii) – ve výsledku existuje jen 15 jedinečných případů (ukázané na obrázku 3.6) [22].

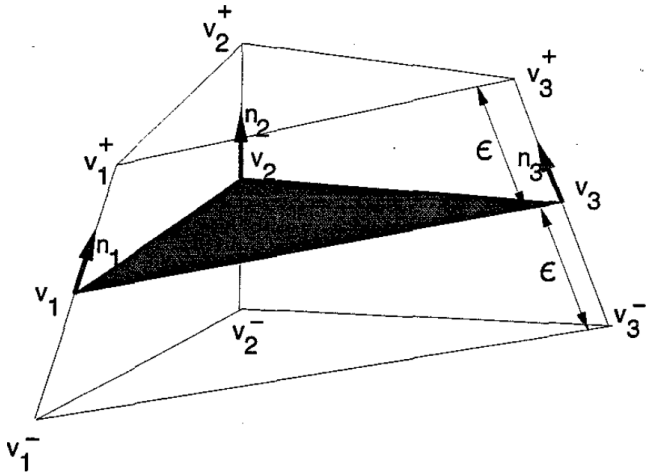
Hierarchická geometrická aproximace

Hierarchická geometrická aproximace (*Hierarchical Geometric Approximation*, Amitabh Varshney), navržená v roce 1994, odstraňuje geometrii pomocí odsazených ploch (*offset surfaces*). Tyto plochy se generují z pozic vrcholů, které jsou vyvýšené ve směru normál a velikost škálování (udávající toleranci chyby při zjednodušování) je předem definovaná uživatelem. Použijí se dvě odsazené plochy: jedna plocha je uvnitř vstupního modelu a druhá je vně modelu (znázorněné na obrázku 3.7). Poté se vygeneruje seznam trojúhelníků ležící uvnitř těchto ploch (s použitím jen vrcholů originálního modelu). Podle Varshneye je

[3]Pojem *voxel* vznikl jako analogie dvourozměrného pixelu. Označuje nejmenší element ve trojrozměrném diskrétním prostoru. Voxely mají tvar kvádru nebo krychle a jsou uspořádány do pravoúhlé mřížky.



Obrázek 3.6: 15 unikátních případů intersekce trojúhelníkové plochy s kostkou [24]



Obrázek 3.7: Trojúhelník (v_1, v_2, v_3) a jeho dvě odsazené plochy ve směru normál: (v_1^+, v_2^+, v_3^+) je vnější plocha, (v_1^-, v_2^-, v_3^-) je vnitřní plocha, ϵ znázorňuje vzdálenost odsazení (tolerance chyby, kterou definuje uživatel) [25]

tento problém NP-úplný[4], ale prezentuje hladový (*greedy*) algoritmus jako jeho alternativu, kde se stanoví které vrcholy jsou pokryté danými trojúhelníky a poté se vyberou optimální trojúhelníky ze seznamu kandidátů [25].

Podle Reddyho [9] představuje tahle aproximace pozoruhodný způsob jak zredukovat síť trojúhelníků a mnoho dalších lidí se pokusilo o jeho rozšíření nebo zobecnění (příkladem je metoda *Simplification envelopes* od Jonathana Cohena a kol., 1996 [26]).

[4]Jedná se o NP-těžký problém, který je zároveň NP.

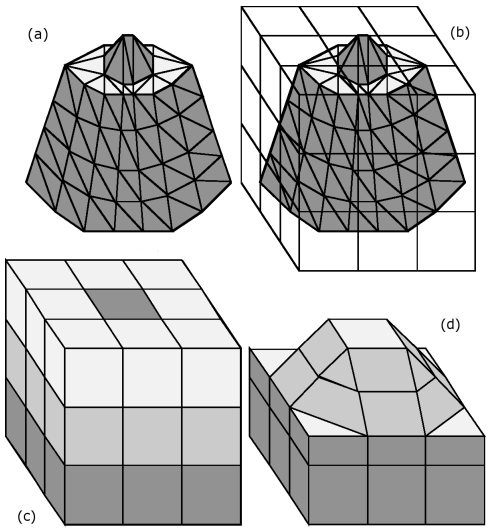
3.4.2 Princip vzorkování

V této části je popsáno několik metod, které zjednodušují síť modelu převzorkováním té původní. Metody si nejprve navzorkují geometrii modelu a poté se snaží vygenerovat model, jenž odpovídá navzorkovaným datům.

Zjednodušování pomocí voxelů

Zjednodušování objektů na základě voxelů (*Voxel Based Object Simplification*, Taosong He a kol., 1995 [27]) je jeden z přístupů využívající vzorkování k redukci trojúhelníkovité sítě.

Model se nejprve převede pomocí 3D mřížky do reprezentace voxelů (libovolného rozlišení) a pro každý z nich se vypočte hustota vzorkováním trojúhelníků uvnitř tohoto voxelu (obrázek 3.8). Nakonec se spustí algoritmus Pochodujících kostek, který z těchto dat zrekonstruuje zjednodušený model. Při konverzi se však může stát, že vzniknou redundantní trojúhelníky, v takovém případě se na model aplikuje algoritmus odstranění geometrie [9].



Obrázek 3.8: (**a**) originální model, (**b**) model je vložen do 3D mřížky, (**c**) určení hustoty voxelů, (**d**) zjednodušený model (po použití algoritmu Pochodujících kostek) [19]

Hustota 3D mřížky (určující objem voxelů) má podstatný vliv na míru zjednodušení modelu: objemnější voxely způsobí drastičtější redukci sítě než voxely menšího rozměru [27]. Tato technika voxelů se v žádném případě nesnaží zachovávat topologii sítě (předpokládá, že síť je uzavřená, bez děr), což ve výsledku vede k velice zjednodušeným modelům [9].

Optimalizace sítě modelu

Hugues Hoppe a kol. [28] v roce 1993 představili další proces zjednodušení sítě pomocí vzorkování, který je založen na konceptu energetické funkce (*Mesh Optimisation*). Tato funkce je používaná k měření odchylky mezi originální a zjednodušenou verzí sítě (vytvořenou zjednodušovacími operátory popsané v sekci 3.4.4). Problém zjednodušení sítě je tudíž převeden na optimalizaci této energetické funkce (hledá se optimální distribuce vrcholů pro konkrétní funkci). Energie sítě modelu se stanovuje např. v závislosti na čtverci vzdáleností původních bodů od upravené sítě, na celkovém počtu vrcholů a nebo na vzdálenosti vrcholů sítě navzájem [1].

Autoři poznamenali, že jejich technika síťové optimalizace pravidelně distribuuje pozice vrcholů podle zakřivení v modelu (oblast s velkým zakřivením je reprezentována vyšším počtem vrcholů než relativně rovná oblast), a proto jsou jednotlivé aproximace sítě tvarově konzistentní [9].

Existuje vylepšená verze metody energetických funkcí, zvaná *Progressive meshes* (nebo Progresivní sítě, Hugues Hoppe, 1996) [16], založená jen na operátoru kolapsu hrany. Algoritmus umožňuje vizuálně hladký přechod mezi dvěma sítěmi modelu, protože snižuje složitost modelu postupným odmazáváním hran v síti. Hrany se odstraňují podle jejich váhy (např. podle koeficientu zakřivení – čím menší je koeficient, tím vyšší je její šance na odebrání z modelu [29]). Metoda má i svou inverzi (použitím operátoru rozdělení vrcholu), jelikož si ukládá pořadí odstraněných hran (společně s údaji o pozici smazaných vrcholů).

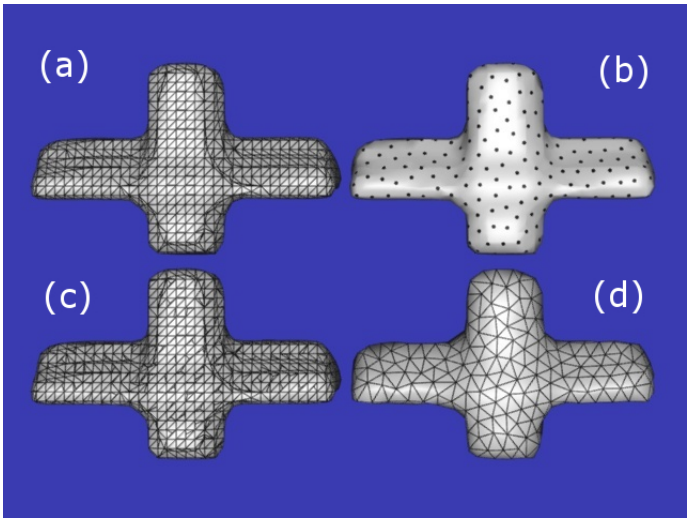
Retiling

Jedna z dalších technik je formulována Gregem Turkem (1992), nazývána *Retiling polygonal surfaces* („znovupokrytí“ nebo „opětovné pokrytí“ polygonálního povrchu), či jen metoda *Retiling* [30].

Tato metoda začíná s obecnou sítí modelu a provede její počáteční triangulaci s předem definovaným počtem vrcholů od uživatele. Tyto vrcholy jsou pseudonáhodně umístěné na povrchu dosavadního modelu a pomocí relaxační procedury (*relaxation*) se každý bod odsune dál od sousedních (*point repulsion*) tak, aby byly na povrchu sítě vrcholy uniformně rozdělené. Staré i nové vrcholy formulují přechodný tvar modelu (tzv. *mutual tessellation*). Postupně jsou původní vrcholy sítě po jednom likvidovány a síť se v každém kroku znova lokálně pokryje pro zachování původního tvaru objektu (kroky algoritmu jsou znázorněné na obrázku 3.9) [30].

3.4.3 Princip adaptivního rozdělení

Metody založené na tomto principu rekurzivně rozdělují a upravují základní model – pracují s nějakou hrubou aproximací originálního modelu. Postupně



Obrázek 3.9: (**a**) originální model, (**b**) nové vrcholy po relaxaci, (**c**) přechodný tvar modelu, (**d**) finální rozdělení [30]

se v každé další iteraci rozdělování doplní více detailů, hlavně v částech sítě, kde se nový model nejvíce liší od toho původního.

Superfaces

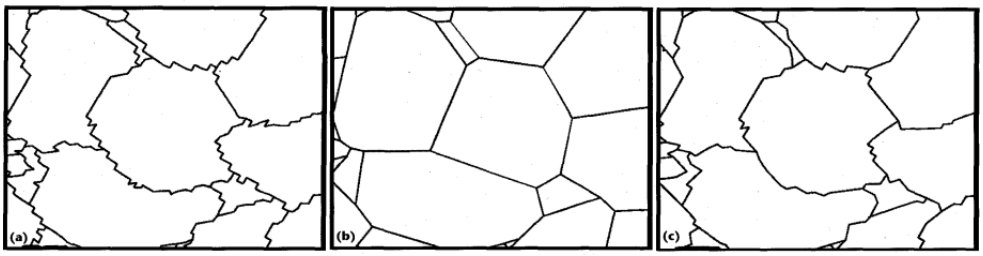
Metoda *Superfaces* od Alana D. Kalvina a Rusella H. Taylora (1996) [31] spojuje polygony v síti modelu do větší skupiny zvané „superfaces“, které postupně dělí dokud není splněna uživatelem zadaná hranice chyby [9].

Nejprve se zvolí počáteční polygon (*seed polygon*) na síti modelu a následně je sloučen se sousedními polygony podle určitých kritérií (např. se nepřesáhne hranice chyby stanovená uživatelem, nezmění se tvar objektu...). Algoritmus se poté pokusí spojit hranice mezi sousedními skupinami (*border straightening*) do jedné společné rovné hrany (*superedge*) a tím se okraje skupiny polygonů narovnají. Pokud se při procesu narovnávání okrajů poruší nějaké podmínky, tak se plocha opět rekurzivně rozdělí (ukázka na obrázku 3.10). V posledním kroku se skupiny polygonů „ztriangulují“ [9].

Metoda je rychlá a zaručí, že nové vrcholy jsou do určité vzdálenosti od původních vrcholů (přibližují se k podmnožině původních vrcholů). Má i řadu nevýhod, například si neumí poradit s dírami v síti.

3.4.4 Zjednodušovací operátory

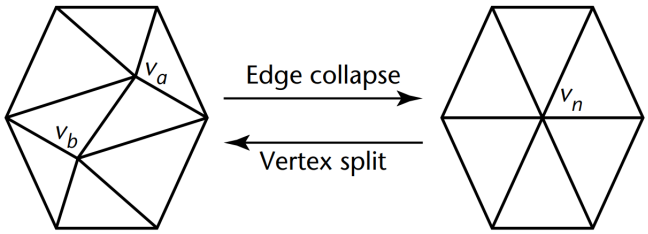
Zjednodušovací operátory jsou metody, které pracují na nižší úrovni (operují přímo s vrcholy, hranami sítě). Tyto operátory (nebo jejich kombinace) jsou používány ve zjednodušovacích algoritmech vyšší úrovně popsané v předcho-



Obrázek 3.10: (**a**) *s*kupiny spojených polygonů (*superfaces*), (**b**) narovnání hranic skupin (*border straightening*), (**c**) rekurzivně rozdělené hranice (*edge splitting*) [31]

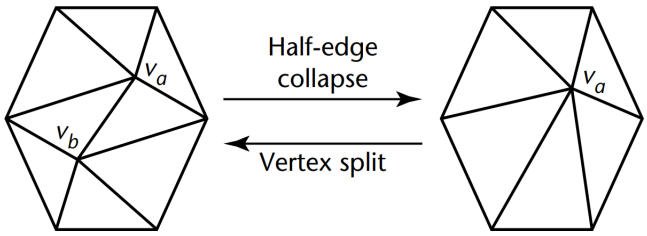
zích částech 3.4.1 až 3.4.3. Operátory jsou dvojího typu: lokální a globální. Lokální operátory snižují složitost sítě po menších částech (lokálně) zatímco globální modifikují síť řízeným způsobem, neboť považují síť za jeden ucelený útvar [15]. Následující seznam v žádném případě neobsahuje všechny příklady existujících operátorů.

Edge collapse Kolaps (sražení) hrany je jeden z lokálních zjednodušovacích operátorů. Byl poprvé použit Huguesem Hoppem (1993, popsané v sekci 3.4.2) v algoritmu *Mesh Optimization*. Operátor srazí hranu (v_a, v_b) do jednoho nového vrcholu v_n (obrázek 3.11), kromě hrany (v_a, v_b) zmizí i trojúhelníky, které tuto hranu sdílí. Inverzním operátorem je rozdělení vrcholu (*vertex split*), jenž přidává novou hranu a přilehlé trojúhelníky [15].

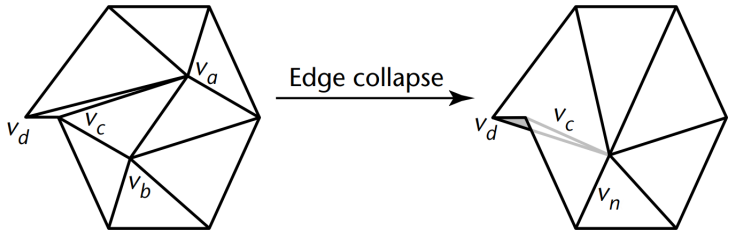


Obrázek 3.11: Operátor kolapsu hrany (v_a, v_b) a jeho inverzní operace [15]

Existují dvě varianty pro kolaps: vrchol, ke kterému se hrana srazí je jeden z koncových bodů hrany (*half-edge collapse*), tj. $v_n = v_a$ nebo v_b (obrázek 3.12). V obecnějším případě (*full-edge collapse* nebo jen *edge collapse*) může v_n být nově vypočítaný vrchol. *Full-edge* kolaps má větší flexibilitu ve vytváření nových vrcholů, tudíž vytváří vizuálně věrnější modely. Na druhou stranu je *half-edge* kolaps efektivnější, zachovává



Obrázek 3.12: Kolaps hrany (v_a, v_b) do vrcholu v_a [15]



Obrázek 3.13: Kolaps hrany (v_a, v_b) do vrcholu v_a způsobí přehyb v síti kvůli nově vzniklému trojúhelníku (v_d, v_c, v_n) [15]

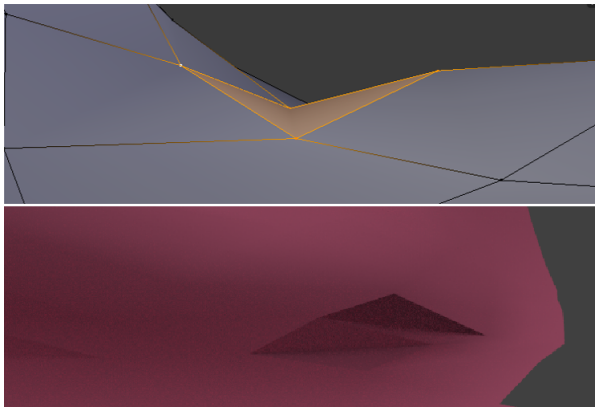
původní množinu vrcholů, což ulehčuje ukládání (není potřeba ukládat nové vrcholy) a celkovou manipulaci s trojúhelníky (oproti *full-edge* kolapsu aktualizuje menší počet trojúhelníků v síti) [15].

Během kolapsu hrany mohou vzniknout chyby v síti. Na obrázku 3.13 je síť špatně ohnutá (*non-manifold*), kvůli kolapsu hrany (v_a, v_b). Tuto chybu lze detekovat změřením normál odpovídajících trojúhelníků před a po kolapsu. Pokud se nově naměřené normály významně liší od původních (zhruba o 90°) jedná se o přehyb v síti. Přehyby způsobují vizuální artefakty jako například nesouvislé osvětlení modelu nebo nekonzistentní textura (obrázek 3.14).

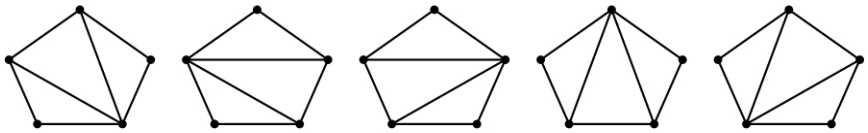
Vertex removal Lokální operátor odstranění vrcholu zanechá otvor v síti (odstraní i všechny trojúhelníky spolu s hranami, které tento vrchol sdílí – ukázka na obrázku 3.16). Operátor byl poprvé představen Schroederem a kol. (1992, popsáno v sekci 3.4.1) v metodě decimace sítí trojúhelníků. Otvor se vyplní triangulací plochy (polygonu), která díru nejdříve zakryje. Počet možností triangulace polygonu o $i + 2$ stranách lze popsat Catalanovým číslem [15]:

$$C_i = \frac{1}{i+1}\binom{2i}{i} \tag{3.1}$$

Výběr jedné z možností se týká problému diskrétní optimalizace. Na obrázku 3.15 jsou znázorněné možnosti triangulace pětiúhelníku.



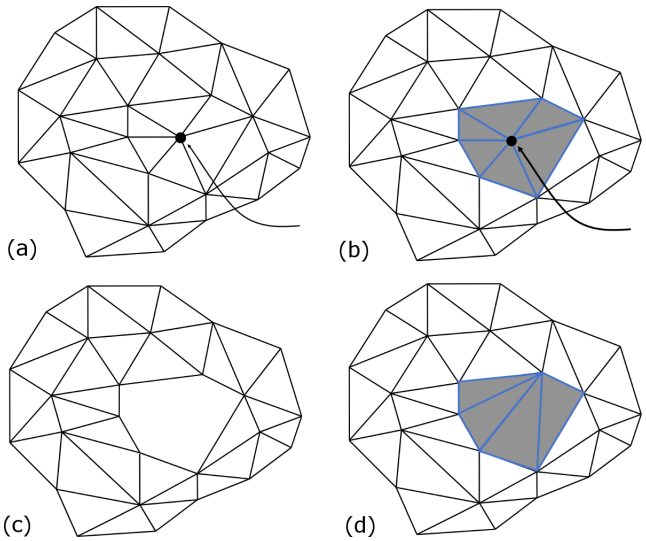
Obrázek 3.14: Model je nesouvisle osvětlen kvůli ostrým přehybům v síti



Obrázek 3.15: Možnosti triangulace pětiúhelníku

Polygon merging Paul Hinker a Charles Hansen poprvé použili ve své geometrické optimalizaci (1993, popsáno v sekci 3.4.1) lokální operátor sloučení ploch. Spojené plochy (nebo polygony) se opětovně „ztriangulují“ do nové sítě modelu. Při těchto krocích dochází k odstranění starých vrcholů a k vytvoření nových trojúhelníků – podobný proces jako při odstranění vrcholu. Nicméně operátor sloučení je obecnější, pracuje s obecnou sítí (nemusí být jen z trojúhelníků), odstraňuje několik vrcholů najednou a může propojovat i plochy s otvory [15]. Tento operátor byl použit i v algoritmu Alana D. Kalvina (1996) pod názvem *Superfaces* (více v sekci 3.4.3).

Low-pass filtering Tento globální operátor („*úzkopásmové filtrování*“) byl mezi prvními, který použil objemovou reprezentaci pro zjednodušení modelu. Představili ho Taosong He a kol. (1995, popsáno v sekci 3.4.2). Vstupní model je vložen do 3D mřížky, podle které se pak navzorkuje do objemové reprezentace (pro každý voxel v mřížce se spočítá průnik primitiv modelu a označí se hodnotou hustoty). Pomocí úzkopásmového filtrování (*low-pass filtering*, například s použitím gaussovského filtru) se postupně odstraní vysoké frekvence (tj. detailní prvky, včetně malých otvorů) navzorkované struktury. Odfiltrovaná struktura je zrekonstruována do izoplochy s použitím algoritmu Pochodujících kostek (k aproximaci povrchu ve voxelu se používá až 5 trojúhelníků) [15].



Obrázek 3.16: (**a**) vrchol, který bude odstraněn, (**b**) výběr všech trojúhelníků sdílící tento vrchol, (**c**) po odstranění vznikne v síti díra, (**d**) díra je vyplněná novými trojúhelníky [32]

Operátor kolapsu je oproti operátoru odstranění vrcholu lehčí na implementaci. Operátor odstranění vrcholu totiž navíc vyžaduje triangulaci vzniklé díry v síti (jedná se o diskrétní optimalizační problém). Schroeder a kol. (1992) popisují rekurzivní metodu „dělení smyčky“ (*loop-splitting*), která triangulaci v jistých okolnostech zjednodušuje (s použitím Seidelovy randomizované metody) [21]. Existuje však několik veřejně dostupných algoritmů na generování LOD založených právě na tomto operátoru, například v knihovně *Visualization Toolkit* neboli VTK[5].

Oba tyto operátory mají ale i své výhody, například operátor kolapsu postupně upravuje příslušnou geometrickou síť modelu, což se dá využít pro hladký přechod z jedné úrovně LOD do druhé (tzv. *geomorphing*). Operátor odstranění vrcholu zase ovlivňuje během modifikace menší počet trojúhelníků (oproti operátoru kolapsu). Zajímavostí je, že některé popsané vlastnosti obou operátorů sdílí operátor *half-edge* kolaps (má společnou podmnožinu operací obou operátorů) [15].

3.4.5 Porovnání algoritmů zjednodušování sítě

Tabulka 3.1 vystihuje základní charakteristiky jednotlivých algoritmů, které jsem popisovala v sekcích 3.4.1 až 3.4.3. Algoritmy jsou v tabulce popsány příslušným názvem, jejich autorem (či autory) a rokem publikování.

[5] https://www.vtk.org

	trojúhelníková síť	podmnožina vrcholů	zachovává topologii
Geometrická optimalizace (Hinker a Hansen, 1993)	✓	✓	✓
Decimace sítě trojúhelníků (Schroeder a kol., 1992)	✓	✓	✓
Hierarchická geom. aprox. (Varshney, 1994)	✓	✓	✓
Využití voxelů (He a kol., 1995)	✓	✗	✗
Optimalizace sítí (Hoppe a kol., 1993)	✓	✗	✓
Retiling (Turk, 1992)	✓	✗	✓
Superfaces (Kalvin a Taylor, 1996)	✓	✓	✓

Tabulka 3.1: Sloupce: *trojúhelníková síť* – algoritmus vyžaduje na vstupu trojúhelníkovou síť, *podmnožina vrcholů* – vytvořený zjednodušený model obsahuje podmnožinu původních vrcholů, *zachovává topologii* – algoritmus při zjednodušování zachovává topologii původního modelu

Všechny popsané metody předpokládají na vstupu síť z trojúhelníků. Vzhledem k tomu, že modely, které v prototypu použiji budou z větší části hlavně budovy (s rovnými fasádami) je toto omezení v rámci práce nevýznamné. Budovy s obecnou polygonální síti lze bez problémů (ovšem za předpokladu rovnosti v modelu) triangulací převést na síť s trojúhelníky.

Zjednodušené sítě, vytvořené algoritmy založených na principu vzorkování, obsahují většinou kompletně nové vrcholy. Metody se nesnaží zachovat původní podmnožinu vrcholů (na rozdíl od algoritmů s principem adaptivního rozdělení či odstraňování geometrie). V projektu VHP (Virtuální historický průvodce) se však předpokládá velká redundance dat, hlavně kvůli rychlosti načítání (všechny modely budou uložené v databázi a na strukturovaný dotaz by měly být ihned k dispozici), tudíž to nevadí.

Topologii sítě zachovávají všechny popsané algoritmy až na algoritmus zjednodušování pomocí voxelů (*Voxel Based Object Simplification*, Taosong He a kol. [27]). Pokud bude v práci nutné výrazně zdecimovat určitý model, je tento algoritmus nejvhodnější.

Greg Turk ve své práci poznamenal [30], že jeho metoda *Retiling* nejlépe funguje se zakřivenými objekty (například izoplochy z lékařských dat nebo molekulární grafika) a nehodí se pro hranaté modely jako jsou budovy, nábytek či stroje [9]. Tento algoritmus tedy není pro vizualizaci modelů budov moc příhodný.

Pro zjednodušení budov se spíše hodí použít metoda geometrické optimalizace (*Geometric Optimisation*, Paul Hinker a Charles Hansen [20]). Modely budov budou zaručeně mít vyšší počet koplanárních ploch, které se pak v této metodě sloučí.

Další použitelný algoritmus je *Progressive meshes* (Hugues Hoppe, 1996) – upravená (vylepšená) metoda *Mesh optimisation* z roku 1993. Metodu lze jednoduše implementovat pomocí operátoru *edge collapse* (popsáno v sekci 3.4.4). Váhy hran (energie) se určí podle koeficientu zakřivení (čím víc jsou plochy navzájem koplanární, tím menší je koeficient a při smazání této hrany mezi nimi zachová objekt svůj tvar). Existují i lepší způsoby pro výpočet vah hran, které dávají lepší výsledky, tento je ale na výpočet docela rychlý a jednoduchý, hodí se na generování sítě za běhu [29]. Metoda umožňuje zjednodušovat jen některé části sítě a zachovává důležité tvary modelu (např. rohy) [33].

3.5 Přechod mezi úrovněmi LOD

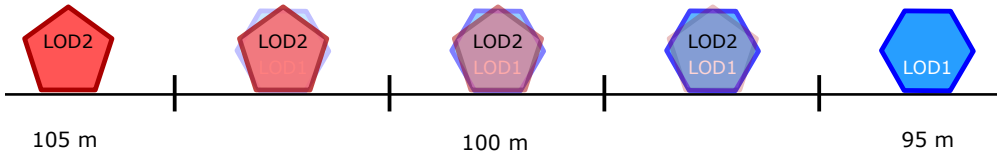
Během přepínání mezi úrovněmi LOD se často stává, že se objevuje nechtěný efekt problikávání objektu. Kvůli tomu vznikla řada metod, jež tento rušivý přechod mírní. Následující část popisuje dva nejběžnější způsoby „míchání“ (*blending*) dvou úrovní LOD: *alpha blending* a *geomorphing*.

3.5.1 Alpha blending

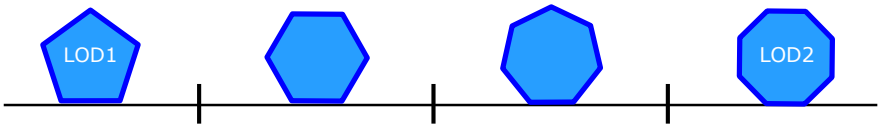
Metoda mírní prudký přechod mezi LOD pomocí alfa hodnoty (hodnota průhlednosti objektu) obou úrovní. Alfa hodnota je v rozmezí 0 až 1. Pokud je alfa hodnota rovna nule, je síť modelu zcela neviditelná a nemusí se vykreslovat (analogicky pro alfa hodnotu rovno jedné).

Pro každý objekt ve scéně se kromě vzdálenosti pro změnu LOD stanoví i interval blednutí (*fade range*) pro jejich úrovně detailu. Interval blednutí určuje oblast (*transition region*), kdy se dvě úrovně (označeno LOD1 a LOD2) budou vykreslovat zároveň, každá se svou lineárně interpolovanou hodnotou alfa (konkrétněji znázorněno na obrázku 3.17). Interval blednutí se může stanovit i jako časový interval (místo intervalu vzdálenosti od pozorovatele). Výhodou je, že se míšení dvou LOD provede za daný časový okamžik a nemusí se pořád vykreslovat obě úrovně LOD zároveň (ušetří to zbytečné vykreslování pokud uživatel stojí v oblasti přechodu dvou LOD delší dobu). Doba intervalu přechodu úrovní by měla být co nejkratší, protože se během ní vykresluje větší počet polygonů. Pokud program snižuje úroveň LOD právě kvůli počtu vykreslovaných polygonů, může delší doba přechodu způsobit problémy. V takových případech se výhoda využití LOD neprojeví a aplikaci to bude nadále zpomalovat [15].

Ideálně by se měly jednotlivé úrovně LOD (během přechodu) vykreslit zvlášť (jako plně viditelné objekty s alfa hodnotou rovné 1) a interpolovat až



Obrázek 3.17: Na obrázku je vzdálenost pro přepínání mezi úrovněmi LOD 100 m a interval blednutí je 10 m. Pozorovateli se v intervalu vzdálenosti (od objektu) mezi 105 m a 95 m vykreslují obě sítě LOD. Ve vzdálenosti 100 m od pozorovatele je hodnota alfy obou úrovní rovna 0.5.



Obrázek 3.18: Geomorphing vytváří podobné sítě jako mezistupeň během přechodu mezi úrovněmi LOD

výsledné obrazy. Vykreslování dvou úrovní modelu na sebe (s různou průhledností) pravděpodobně způsobí, že se obě úrovně budou navzájem zakrývat (*self-occlusion*) [34].

3.5.2 Geomorphing

Geomorphing (nebo „geomorfování“) míchá stupně LOD na úrovni prostoru objektu a mění samotné vrcholy v síti během jejího přechodu (znázorněno na obrázku 3.18). Jedná se o složitou ale velmi účinnou techniku pro plynulé přecházení z jednoho LOD do druhého. V praxi se při použití mohou objevit téměř nepostřehnutelné chyby (několik málo pixelů na obrazovce), jinak technika dává dobré výsledky [15]. *Geomorphing* podporuje mnoho komerčních (ale i veřejně dostupných) systémů, například herní jádro Unreal Engine používá geomorphing jako součást spojitých stupňů LOD.

První ukázku „geomorfování“ provedl Greg Turk ve své metodě *Retiling* (popsáno v sekci 3.4.2). Na přechodné síti (obsahující původní i relaxované vrcholy) provedl lineární interpolaci jako přechod z původního modelu (s vysokými detaily) do nového (s nižšími detaily) [30].

Hugues Hoppe ve své metodě *Progressive meshes* (popsáno v sekci 3.4.2) také použil termín *geomorphing* k označení podobné geometrické interpolace mezi úrovněmi LOD. Metodu použil ke zjemnění vizuálního přechodu při ko-

lapsu hrany (nebo při inverzní operaci rozdělení vrcholu) sítě. „Geomorfní“ kolaps hrany lineárně interpoluje pozici nového vrcholu s pozicí odpovídajícího vrcholu v síti původní (s vyššími detaily). Později se metoda *geomorphing* používala přímo během spouštěného programu (*run-time*) ale jen na oblasti, které pozorovatel viděl (pro neviditelné části by to bylo zbytečné plýtvání) [15].

Kapitola 4

Viditelnost

Věrohodné a realisticky vypadající virtuální scény obsahují většinou modely s vysokými detaily. Prostředí virtuálních měst mohou tudíž obsahovat i přes několik miliónů polygonů, což je množství, které nelze vykreslit v rozumném čase (v řádu sekund) aby byla umožněna rozumná interakce pozorovatele s prostředím (například jednoduché procházení městem bez zasekávání obrazu). K omezení počtu zpracovaných elementů se využívají různé techniky na vyřazení neviditelných částí ve scéně. Jak efektivně určit tyto neviditelné polygony z určitého úhlu pohledu je jeden z klíčových problémů počítačové grafiky.

Jedno z kritérií jak dělit algoritmy určení viditelnosti je forma jejich využití v aplikaci [1]. Algoritmy se dělí na:

- algoritmy pracující v reálném čase (během *run-time*),
- algoritmy předzpracování.

U algoritmů pracujících v reálném čase je důležité, aby byla doba jejich výpočtu co nejkratší. Kvalita výsledku je určena rychlostí zpracování. Tyto algoritmy musí opakovat výpočet viditelnosti pokaždé, když se změní směr pohledu uživatele, jelikož neobsahují informace o celkové (globální) viditelnosti prostředí [35]. Představiteli takových algoritmů jsou například metody odstraňování zastíněných objektů (*occlusion culling*, popsané v sekci 4.3). Druhá skupina algoritmů nemusí být výpočetně rychlá, ale je žádoucí, aby poskytovaly co nejpřesnější výsledky [1].

Algoritmy lze dělit i podle kvality jejich poskytovaných výsledků [1]:

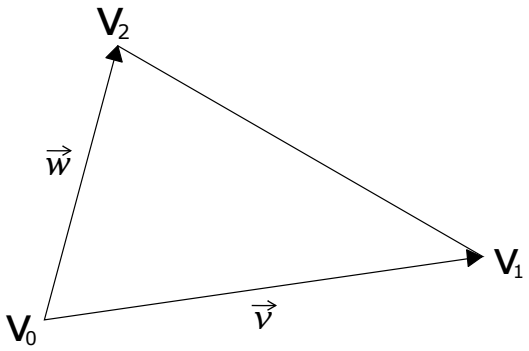
- přesné,
- konzervativní,
- agresivní,
- přibližné.

Exaktní řešení je poskytováno přesnými algoritmy – nalezne se právě ta množina objektů, kterou je nutné zobrazit. Konzervativní algoritmy mírně „nadhodnocují“ viditelnost objektů (bodů, ploch...) ve scéně [1]. Tato nadmnožina viditelných objektů se označuje jako potenciálně viditelná množina (*potentially visible set* – PVS) a může obsahovat objekty, které v konečné scéně viditelné nejsou. Naopak agresivní algoritmy udávají podmnožinu viditelných objektů. Přibližné algoritmy pak pouze určitým způsobem aproximují exaktní řešení [36].

4.1 Odstraňování odvrácených stěn

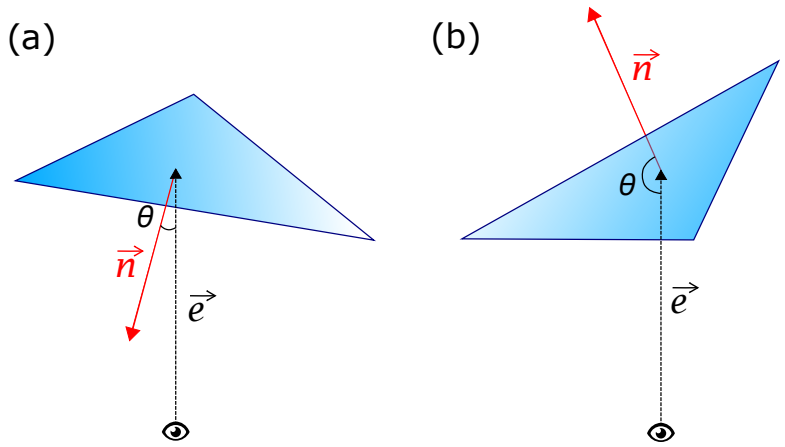
Každá síť modelu má dvě strany: vnější a vnitřní. Pozorovatel nevidí odvrácené (vnitřní) polygony v síti, obvykle vidí jen polovinu vnější části modelu zepředu. Druhá polovina modelu je zakrytá a lze ji tedy bezpečně odstranit (vymazat ze seznamu polygonů, které se mají vykreslit). Jsou dva způsoby jak určit, zda je polygon vůči pohledu (kameře) vnější nebo vnitřní [37]:

orientace trojúhelníků Neviditelnost se určí na základě orientace trojúhelníků v síti modelu. Pokud jsou vrcholy trojúhelníku (V_0, V_1, V_2) očíslovány v pořadí proti směru hodinových ručiček (obrázek 4.1), je vypočítaný obsah kladný (pro 2D trojúhelník, předpokládá se že souřadnice z je rovna nule). V opačném případě je obsah negativní. Pokud je hodnota obsahu menší než 0, jedná se o zadní stranu trojúhelníku, která se nemusí vykreslit.



Obrázek 4.1: Vektory $\vec{v} = V_1 - V_0$ a $\vec{w} = V_2 - V_0$, obsah se vypočítá jako $S = \frac{1}{2}\|\vec{\mathbf{v}} \times \vec{\mathbf{w}}\|$

použití normál Pomocí normály plochy se zjistí, zda je polygon odvrácený od pozorovatele či ne. Stačí vypočítat úhel mezi vektorem pohledu (označeno $\vec{e}$) a normálou ($\vec{n}$) trojúhelníku. Plocha je odvrácená pokud je úhel větší než 90° (obrázek 4.2 za b). Platí, že pokud je $\vec{n} \cdot \vec{e} < 0$, je trojúhelník odvrácený.



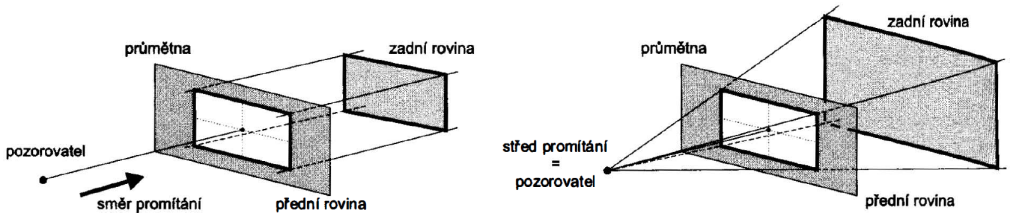
Obrázek 4.2: Vektory $\vec{n}$ (normála trojúhelníku) a $\vec{e}$ (vektor pohledu), θ je úhel mezi nimi, (**a**) trojúhelník není odvrácený od pozorovatele, (**b**) trojúhelník je odvrácený

Odstraňování odvrácených stěn (*backface culling*) eliminuje v průměru 50 % polygonů. K efektivnějšímu eliminování odvrácených polygonů se seskupí polygony, jejichž normály jsou orientovány podobným směrem a vytvoří se binární strom těchto skupin. Během zobrazování se projde strom od kořene a podle směru pohledu se vyberou skupiny uzlů představující přivrácené polygony [1]. Základní algoritmus pro odstranění odvrácených stěn je už v současné době implementovaný přímo v grafických procesorech (GPU).

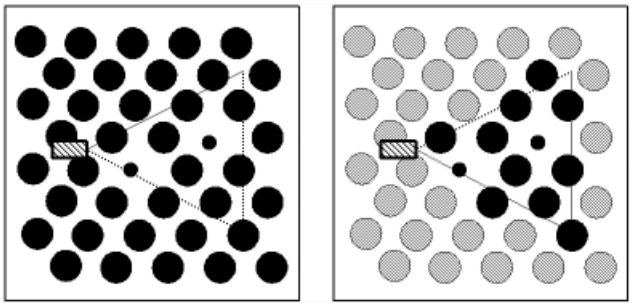
4.2 Odstraňování pohledovým objemem

Cílem této metody (*view frustum culling*) je zjištění, jaké objekty alespoň částečně leží v pohledovém objemu (*viewing frustum/volume*), někdy též záběr (na obrázku 4.3 jsou dva základní druhy promítání). Jedná se o region v prostoru virtuálního prostředí, ohraničující ty objekty, které mají být podrobeny promítání, všechno ostatní mimo region se před dalším zpracováním odstraní (resp. ořežou). Tento pohledový objem je definovaný šesti omezujícími rovinami (*clipping planes*), z nichž nejvýznamnější dvě jsou přední (*near*) a zadní (*far*) omezující roviny. Při ořezávání zajišťují tyto roviny odstranění příliš blízkých objektů (bránící ve výhledu) nebo příliš vzdálených objektů (pro pozorovatele nezajímavé) [1]. Grafický procesor zpracovává právě jen ty objekty, které jsou uvnitř tohoto objemu (obrázek 4.4), což potenciálně zlepšuje výkon aplikace. Výpočet objektů, které leží uvnitř jehlanu musí být rychlý a aplikovatelný na velkou skupinu polygonů [38].

Pro efektivní realizaci této techniky se využívá graf scény nebo hierarchické datové struktury [1]. Hierarchická struktura je reprezentována stromem (hie-



Obrázek 4.3: Pohledový objem při rovnoběžném (vlevo) a středovém (vpravo) promítání [1]

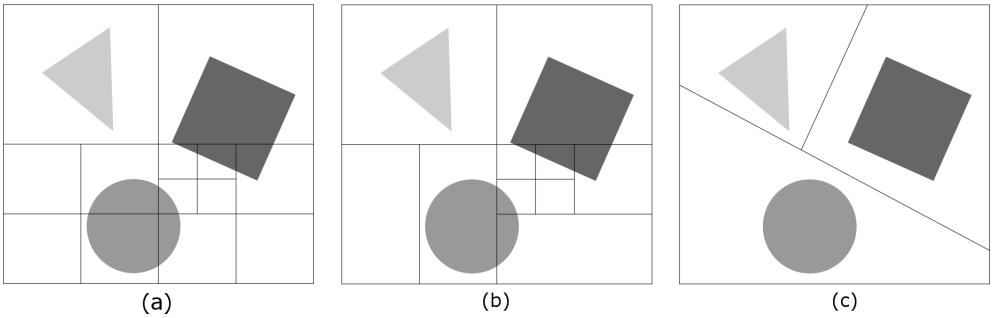


Obrázek 4.4: Ukázka odstraňování pohledovým objemem: šedé objekty jsou vyřazené a nebudou se dále zpracovávat [39]

rarchie obálek[6], oktalové stromy, BSP stromy – *binary space partitioning tree*, či k-dimenzionální stromy – *k-d tree*), jehož kořen znázorňuje celou scénu. Další možnosti dělení prostoru scény jsou popsány v kapitole 5 o blokovém načítání (v sekci 5.1).

Prostor se rekurzivně dělí (dělení je prováděno rovinným řezem), dokud v dané části scény ještě leží nějaké objekty nebo není dosažena definovaná hloubka stromu (obrázek 4.5 znázorňuje způsoby dělení prostoru některých stromů). Každý objekt je propojený s nejmenším uzlem, který jej obsahuje. Při procházení stromu se zjišťuje, zda pohledový objem obsahuje (či protíná) příslušnou část prostoru (obálku) a pokud ne, je celý podstrom neviditelný. V opačném případě se dále rekurzivně testují potomci uzlu [1]. Hlavní výhodou struktur je schopnost přizpůsobit se hustotám objektů ve scéně [40].

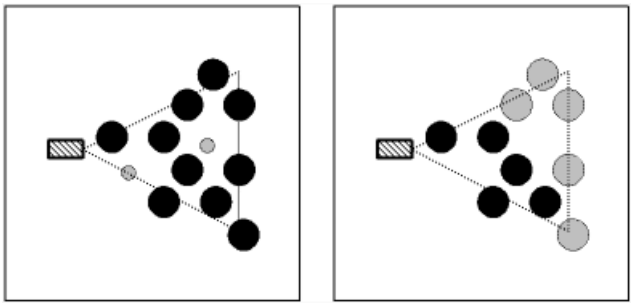
[6]Ohraničení objektu obálkou (*bounding volume/box*) je jeden ze základních způsobů urychlení v grafice vůbec. Testy (např. test polohy objektu) lze realizovat mnohem rychleji s obálkou, než se samotným objektem, který obálka obklopuje.



Obrázek 4.5: Způsoby dělení prostoru do hierarchických struktur: (**a**) oktalový strom (dělení třemi řezy kolmými na souřadnicové osy a umístěnými v polovině daného prostoru – uzel může mít 0 až 8 poduzlů), (**b**) k-d strom (speciální případ BSP stromu k popisu libovolného k-rozměrného prostoru), (**c**) BSP strom (binární strom s obecně umístěnými řezy) [40]

4.3 Odstraňování zastíněných objektů

Obě předchozí metody nejsou schopné vyloučit veškeré neviditelné polygony, které jsou z pohledu pozorovatele schované za jinými objekty. Do grafické karty jsou tudíž zbytečně posílány i zastíněné polygony, které se musí pomocí paměti hloubky[7] eliminovat až později, při zápisu do obrazové paměti. Například v městské scéně jsou viditelné nejbližší přilehlé ulice, které tvoří jen malou část celého prostředí (zbytek je jimi zastíněn). V následující části bude představeno několik metod (*occlusion culling methods*), které řeší uvedený problém tak, že vyloučí zmíněné polygony ještě dříve, než jsou podrobeny dalšímu zpracování (obrázek 4.6).



Obrázek 4.6: Ukázka odstraňování zastíněných objektů: šedé objekty jsou vyřazené a nebudou se dále zpracovávat [39]

[7]Paměť hloubky (*z-buffer, depth-buffer*) tvoří 2D pole, jehož rozměry odpovídají velikosti obrazu. Každá položka v poli obsahuje souřadnici z bodu, který je nejblíže pozorovateli a jehož průmět je odpovídající pixel v rastru (obrazové paměti).

Algoritmy si nejdříve připraví strukturu, která zachycuje možné zastínění scény jednotlivými polygony – tzv. strukturu zastínění (v pseudokódu 1 jako proměnná *occlusionRepresentation*), která mívá často hierarchický charakter. Zároveň je nutné vytvořit prostorové třídění scény (v pseudokódu 1 jako proměnná *scene*), tzv. prostorovou hierarchii (pomocná datová struktura, která má vhodně uspořádané informace o umístění objektů ve scéně – obrázek 4.5). Při vyhodnocování viditelnosti se prochází obě struktury a testuje se viditelnost [1]. Zde je pseudokód pro obecný algoritmus určující viditelnost objektů:

Pseudokód 1 Algoritmus odstraňování zastíněných objektů [41]

```
occlusionRepresentation ← empty
for each object g in scene do
    if isOccluded(g, occlusionRepresentation) then
        Skip(g)
    else
        Render(g)
    Update(occlusionRepresentation)
```

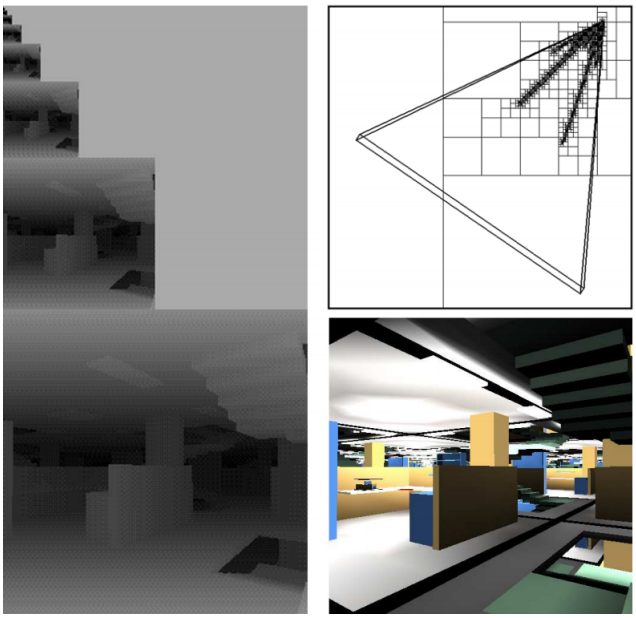
4.3.1 Hierarchický z-buffer

Hierarchický z-buffer (ukázka metody je na obrázku 4.7), neboli hierarchickou paměť hloubky, představili Ned Green a kol. (v roce 1993) a používá pyramidu z-bufferů (z-pyramidu) [42]. Nejnižší patro odpovídá klasické paměti hloubky, vyšší patra jsou převzorkována do poloviční rozlišení tak, aby odpovídala nejvzdálenějšímu záznamu ze čtveřice z nižšího patra z-pyramidy [1]. Metoda není efektivní, pokud není z-pyramida podporována grafickým procesorem (v takovém případě lze použít mapy zastínění, popsané v sekci 4.3.2).

Nejprve se vytvoří prostorová hierarchie pomocí oktalového stromu. Během fáze vykreslování se pro každý uzel (část rozděleného prostoru – obálka) uvnitř pohledového jehlanu porovná hloubka obálky s nejvyšším patrem pyramidy. Pokud je hodnota souřadnice z obálky vzdálenější než hodnota v z-pyramidě, je obálka zakrytá. Pokud ne, testování pokračuje v nižších patrech z-pyramidy [43]. Hloubky nově zapsaných pixelů v nižších patrech se použijí k aktualizaci hodnot ve vyšších patrech z-pyramidy [1].

4.3.2 Hierarchické mapy zastínění

Metodu navrhli v roce 1997 Zhang a kol. (*Hierarchical occlusion maps*) [45]. Její výhodou je, že se dá použít na všechny typy scén. Algoritmus k práci vyžaduje množinu objektů, které jsou dostatečně velké a blízko pozorovatele (tzv. *occluders* – stínící objekty/polygony). Tyto stínící polygony jsou poté vykresleny do nejnižší mapy zastínění (patro s nejvyšším rozlišením). Vyšší



Obrázek 4.7: Ukázka metody hierarchického z-bufferu na složité scéně (kancelář vpravo dole) a její odpovídající z-pyramida (nalevo). Vpravo nahoře je prostor scény rozdělen do prostorové hierarchie a znázorněn průchod od kořene hierarchie a současně od pozorovatele směrem do scény [44]

úrovně mapy jsou vytvořeny filtrováním té originální[8] a výsledkem je hierarchie sestávající z několika map zastínění rozdílné velikosti [40].

Každý záznam v mapě reprezentuje úroveň zastínění v intervalu 0 až 1 (0 – žádné zastínění, 1 – úplné zastínění). Pixel, jenž je zakrytý alespoň jedním polygonem má hodnotu jedna, jinak obsahuje hodnotu nula. Hierarchická mapa zaznamenává relativní podíl pixelů, do kterých se v odpovídající oblasti obrazu promítá alespoň jeden stínící polygon. Mapa zastínění sama o sobě neobsahuje žádné informace o hloubce. Hloubky stínících polygonů jsou odděleně uloženy v paměti přibližné hloubky (*depth estimation buffer*) s menším rozlišením (obvykle 64 $\times$ 64) [1].

Při testování objektu musí algoritmus zkontrolovat, zda je dostatečně daleko od pozorovatele aby byl zastíněn vybranými stínícími objekty (*occluders*) [40]. Rekurzivně se testují buňky v mapě zastínění (obsahující průmět objektu) dokud se nenajde buňka, která zastíněná není (hodnota < 1). V tomto případě je objekt (jeho obálka) označen za viditelný a test se ukončí. Pokud test skončil u plně zastíněných buněk mapy, porovnají se ještě hloubky pixelů se záznamy v paměti přibližné hloubky. V případě, že obálka leží za všemi porovnávanými záznamy, je označena za neviditelnou [1].

[8] tento proces je součástí moderních grafických procesorů

4.3.3 Stromy zastínění

Techniku prezentovali Jiří Bittner a kol. v roce 1998. Výhodou algoritmu je, že odhalí i zastínění způsobené kumulativním efektem několika stínících těles [1]. Ve fázi předzpracování (*preprocessing*) se označí možné stínící polygony (tělesa) a vytvoří se prostorová hierarchie pro scénu (jedná se o k-d strom, specializovaný BSP strom se zarovnanými řezy podle os). Autoři se zabývají hlavně architektonickými modely, takže stínící polygony jsou obvykle stěny budov. Zvolená stínová tělesa jsou seřazena podle jejich vzdálenosti od pozorovatele, aby se z nich vytvořila hierarchická reprezentace – neboli strom zastínění (*occlusion tree*) [46]. Jedná se o BSP strom, jehož uzly odpovídají rovinám procházející hranami vybraných stínících polygonů [1]. Testování viditelnosti probíhá testováním skupiny objektů, uspořádané v prostorové hierarchii pomocí k-d stromu [36]. Pokud je uzel hierarchie obsažen ve stromě zastínění, je označen za neviditelný.

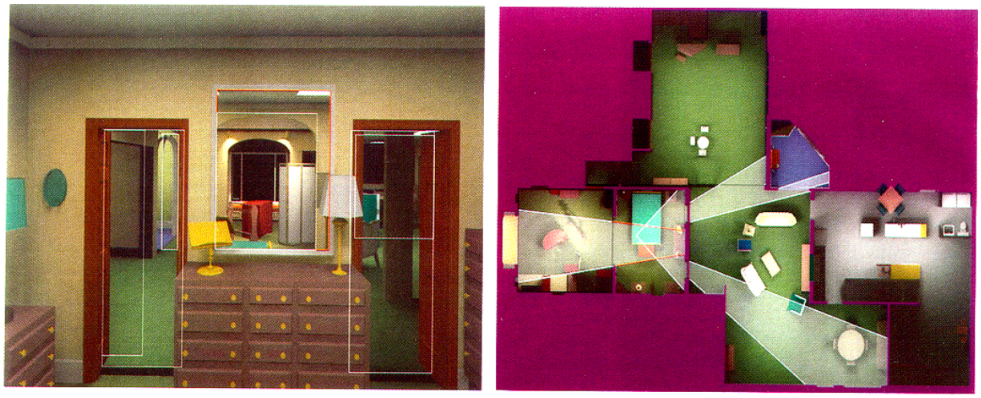
4.3.4 Portály a buňky

Metoda byla popsána Davidem Luebkem a Chrisem Georgesem v roce 1995 (*Portals and mirrors* [47]). V dnešní době je velice známá a často se používá v počítačových hrách, například v interiéru budov (kde jsou scény především statické). Tato jednoduchá a rychlá technika v praxi dosahuje velmi dobrých výsledků. Metoda se však nehodí pro rozsáhlé venkovní prostředí. Další nevýhodou může být nutnost delšího předzpracování (pro vytvoření buněk a portálů) [40].

Scéna se nejprve rozdělí do konvexních buněk (zhruba odpovídající jednotlivým místnostem), které jsou průstupné jedině přes portál (dveře a okna). Začíná se procházením scény od buňky, kde se nachází pozorovatel a pro každou navštívenou buňku se zobrazí všechny objekty v ní obsažené. Algoritmus postupně rekurzivně zpracuje propojené místnosti (pomocí portálových masek, jež jsou průnikem doposud zpracovaných portálů) dokud se nezpracují všechny viditelné portály v pohledovém objemu (počítají se i portály odražené v zrcadlech) [1]. Na obrázku 4.8 je ukázka scény místnosti používající právě tuto metodu.

4.3.5 Horizont zastínění

Algoritmus byl představen v roce 2001 Laurou Downsovou a kol. Metoda je speciálně navržena pro urbanistická prostředí. Horizont zastínění (na obrázku 4.9) se vytváří za běhu a ukládá se ve vyváženém binárním stromě. Horizont se aktualizuje přidáváním zpracovaných polygonů (např. stěny budov) procházením scény zepředu směrem dozadu [48]. Pro každý uzel hierarchie scény se testuje, zda uzel (obálka) je pod aktuálním horizontem, a pokud ano, je uzel neviditelný. V opačném případě se pokračuje níže k listům a horizont se aktualizuje pomocí polygonů v těchto listech [1].



Obrázek 4.8: Nalevo jsou označené průstupné portály ve scéně, vpravo je znázorněn pohled uživatele do dalších místností [47]

Obrázek 4.9: Scéna města s označeným horizontem zastínění (vlevo), část obrázku vpravo ukazuje skutečný počet vykreslených objektů (pohled ze shora) [48]

Na základě tohoto algoritmu [48] byl vytvořen další způsob využití horizontu zastínění, zaměřený především na terénní oblasti. Metodu představili Brandon Lloyd a Parris K. Egbert v roce 2002 [49]. Autoři navíc používají speciální typ kvadrantového stromu (*quadtree*), který umožní každé části terénu (zvané *gridlets*) mít různé úrovně detailu (*level of detail*). Úrovně detailu mezi jednotlivými části terénu lze bez problikávání měnit pomocí *quadtree morphing* (podobné „geomorfování“ v metodě Progresivních sítí).

4.4 Předzpracování viditelnosti

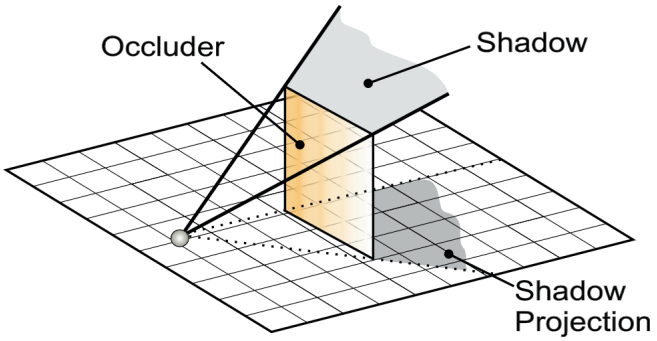
V určitých prostředí scén (jako je například město) je známá jejich vlastnost (vysoké budovy, neprůhledné fasády...). Výpočet viditelnosti se může tedy na základě těchto předpokládaných vlastností provést předem. Scéna se rozdělí do menších oblastí (buněk/bloků) a viditelnost objektů se vypočítá pro každou buňku. Předzpracování může u rozsáhlejších scén nějakou chvíli trvat, protože se musí zkontrolovat každý objekt ve všech existujících buňkách scény. Výsledky předzpracování se pak uloží a při následném (rychlém) vykreslování scény se pouze identifikuje množina předem vypočítaných objektů v buňce, ve které se pozorovatel zrovna nachází [40]. V době předzpracování chybí informace o pohybu pozorovatele (v dané scéně) a proto se viditelnost hodnotí „globálně“ s ohledem na různé situace, které mohou během vykreslení nastat [1].

V následující části je uvedena metoda založená právě na předzpracování viditelnosti. Jinou možností je využít geometrického popisu zastíněného objemu k rychlejšímu (a mnohdy přesnějšímu) určení potenciálně viditelné množiny (PVS) [1].

4.4.1 Objektové stíny

Peter Wonka a Dieter Schmalstieg ve své metodě *Occluder shadows* použili převážně 2.5D charakteru městského prostředí (v podstatě 2D s výškovou mapou), neboť většinu ploch v urbanistických modelech představují vertikálně orientované stěny budov.

Metoda v každé buňce (oblast prostředí) vybere určitý počet stínících objektů, které mají vysokou šanci zastínit značnou část scény. Pro vybrané stínící objekty se shromáždí půdorysné projekce jejich stínů (obrázek 4.10) do tzv. *cull maps*. Množina potenciálně viditelných objektů se poté získá z výsledného obsahu mapy (*cull* mapy) a na těchto objektech se nakonec testuje viditelnost [50].



Obrázek 4.10: Na obrázku je znázorněna stínící plocha (*occluder*), její stín (*shadow*) a půdorysná projekce stínu (*shadow projection*) [50]

4.5 Porovnání algoritmů určení viditelnosti

Metody odstraňování odvrácených stěn (*backface culling*) a odstraňování na základě pohledového objemu (*view frustum culling*) jsou zásadními technikami pro vyřazení neviditelných částí scén. Metody fungují rychle, spolehlivě a jsou (ve většině případů) efektivní. Obě tyto techniky jsou dnes většinou už implementovány na úrovni polygonů přímo v grafickém procesoru. Jedná se o nejčastěji používané optimalizace vykreslování v grafice vůbec [51].

Metody, které předem vypočítávají viditelnost scény byly velice populární v raném období vývoje 3D enginů. Jejich hlavní výhodou je, že v době běhu programu požadují menší náklady na vykreslení scény, což nyní na modernějších sestavách počítačů není až tak moc velkým přínosem. Čas ušetřený při vykreslování se zase využije ve fázi předzpracování k vytvoření potenciálně viditelné množiny objektů. Pro velká virtuální prostředí jsou tyto metody špatně škálovatelné, předzpracování by trvalo dlouhou dobu a informace o viditelnosti by zabíraly velké místo v paměti. Pro rozsáhlé a dynamické scény by tento způsob nemohl být dlouho udržitelný. Vypočítávání viditelnosti předem se spíše hodí pro aplikace na mobilních zařízení, protože mají omezenější výpočetní výkon (oproti počítači nebo konzoli) [52].

Všechny algoritmy pro odstranění zastíněných objektů uvedené v této práci jsou spíše konzervativní, tj. regiony ve scéně určí za neviditelnými, jen pokud jsou v konečné scéně opravdu zastíněné. Nevýhodou konzervativních algoritmů je, že mohou označit část objektů za viditelné, i když ve výsledku vidět nejsou. Některé aplikace preferují vynechat i ty objekty, které mohou – s malou pravděpodobností – být viditelné (aplikace je pak výpočetně výkonnější). Agresivní algoritmy mají však větší sklony k chybám při určení viditelnosti a scény by obsahovaly vizuální nedostatky, což je naprosto nepoužitelné, pokud záleží na kvalitě konečného obrazu.

Hierarchický z-buffer je jeden ze základních technik pro určení zastíněných částí ve scéně. Přestože má tato technika jasnou implementační strukturu, její efektivita hodně závisí na grafickém procesoru a podpoře z-bufferu. Komerční grafické procesory mohou mít například pomalejší zpětnou vazbu nebo nepodporují veškeré čtení ze z-bufferu [36].

Metody hierarchických map zastínění a stromů zastínění naopak nemají speciální požadavky na hardware ale zase záleží na správném výběrů stínících objektů. Nevhodně zvolené stínící objekty mohou způsobit pokles efektivity algoritmu. Nicméně jsou obě metody docela rychlé [40].

Speciálně pro urbanistické scény lze použít metody horizontu zastínění. Městské budovy jsou zpravidla jinak vysoké a díky tomu lze lépe stanovit hranice horizontu zastínění (na rozdíl od výšek v terénu, které jsou spojitější a hladší – neobsahují prudké skoky a zarovnání do pravého úhlu) [53].

Pro pevně uzavřené prostory s velkým počtem objektů, je vhodné použít metodu portálů a buněk. Ručně umístěné portály snižují čas strávený ve fázi předzpracování a zabírají jen o trochu víc paměti [51]. Pro vizualizaci roz-

sáhlého města se metoda ale použít nedá. Město nelze výhodně rozdělit do oddělených buněk a tím pádem by se musel projít každý objekt v prostoru scény.

Kapitola 5

Blokové načítání

Rozsáhlé scény je těžké zpracovávat najednou jako celek a navíc je velice nepraktické (a často nemožné) mít načtenou celou scénu do paměti. Z tohoto důvodu je doporučeno rozdělit si prostor scény do menších částí (bloků) a zpracovávat scénu po blocích stejné velikosti. Potřebné bloky se načtou do paměti a jsou neustále dynamicky aktualizovány podle průchodu pozorovatele (a podle směru pohledu) ve scéně, aby měl pozorovatel dojem, že se pohybuje v uceleném virtuálním světě.

Dynamické načítání bloků zrychluje nejen vykreslení scény ale snižuje i paměťové nároky během vizualizace [54]. Datová struktura, uchovávající tento virtuální prostor scény, musí umožňovat přístup k datům po jednotlivých blocích [55].

5.1 Hierarchie dat

Prostor scény se běžně ukládá do hierarchických datových struktur, které urychlují přístup k jednotlivých datům v prostoru a ulehčují celkovou manipulaci se scénou. Hierarchicky rozdělená scéna se používá při určování viditelnosti a vymezuje velikost bloků při dynamickém načítání dat do paměti (popsané v další sekci 5.2).

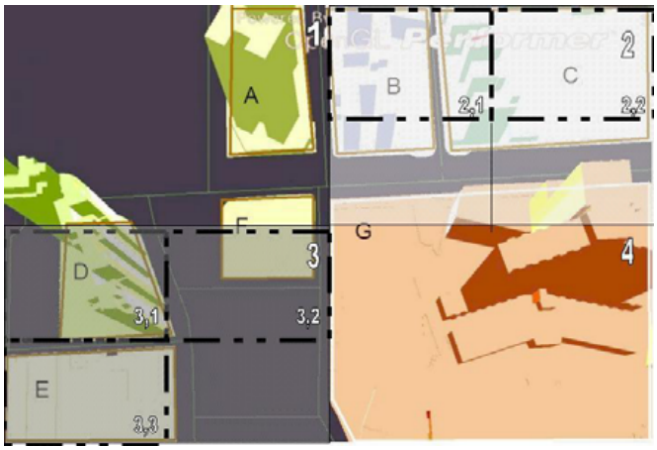
Hierarchických datových struktur je hned několik, základními typy jsou hierarchie obálek, kvadrantový strom, oktalový strom, BSP strom (*binary space partitioning tree*) a k-d strom. Stromy, na rozdíl od hierarchie obálek (které jsou vlastně také seskupeny do stromové struktury), dělí prostor scény systematicky [1]. Kromě základních typů se používají i různé (vylepšené) verze hierarchie, vhodné právě pro specifický typ scén (město, terén, uzavřená místnost...).

Z podrobného výzkumu, porovnávající různé datové struktury pro ukládání prostorových dat, od Volkera Gaedeho a Olivera Günthera [56] vychází, že pro urbanistické scény jsou nejvhodnější kvadrantové stromy (*Quadtree*) a R-stromy (*R-tree*) [57]. R-stromy rekurzivně dělí prostor scény do minimál-

ních ohraničujících obdélníků (*minimum bounding rectangles*). Účelem takového rozdělení (kromě urychlení operací prováděné nad scénou) je poskytnutí dat k určení LOD (úrovní detailu) za běhu programu [58].

Často se však místo klasických stromů používají „vylepšené“ verze. Jose Luis Pina a kol. (2008) používají například datovou strukturu *B-Quadtree* (*Block-Quadtree* – blokový kvadrantový strom), vytvořenou rekurzivním rozdělováním prostoru (regionu) města do kvadrantů. Pro autory je nejmenší jednotkou scény jeden blok – neboli shluk budov obklopenými ze všech stran ulicemi (*urban block*). Scéna se neustále rekurzivně dělí do kvadrantů, dokud není každý blok přiřazen právě jednomu kvadrantu. Na konci dělení obsahuje každý kvadrant jeden blok nebo je prázdný [57].

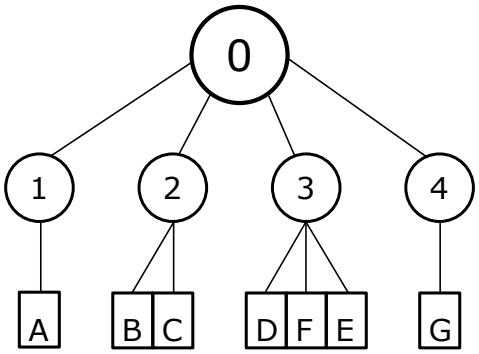
Na obrázcích 5.1 a 5.2 je znázorněn příklad rozdělení scény do kvadrantů a příslušný strom k danému rozdělení. Složitost vyhledávání v tomto adaptivním kvadrantovém stromě je v nejhorším případě rovno $\mathcal{O}(n)$, kde n značí hloubku stromu (která je většinou malá).



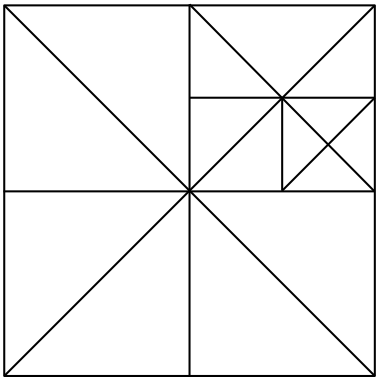
Obrázek 5.1: Příklad rozdělení scény do kvadrantů: A-G popisují bloky, čísla znázorňují kvadranty [57]

Kvadrantové stromy se používají i pro data velkých terénních oblastí. Příkladem je vizualizace terénu Petera Lindstroma a kol. (1995) [10]. Každá část rozděleného terénu je sama o sobě nezávislá. Algoritmy viditelnosti (odstraňování neviditelných částí) nebo LOD (úroveň detailu) se určí pro každý blok zvlášť, což zvyšuje efektivitu zpracování celé scény.

Renato Pajarola (1998) také použil kvadrantový strom – *restricted quadtree triangulation*. Kvadranty však dále rozdělil do trojúhelníků, aby mohl lépe aplikovat metodu Progresivních sítí (*Progressive meshes*) [55], příklad rozdělení prostoru je na obrázku 5.3.



Obrázek 5.2: Stromová reprezentace scény z příkladu 5.1, prázdné kvadranty se do stromu neukládají (adaptivní kvadrantový strom)



Obrázek 5.3: *Restricted quadtree triangulation* – kvadranty se ještě rozdělí na trojúhelníky

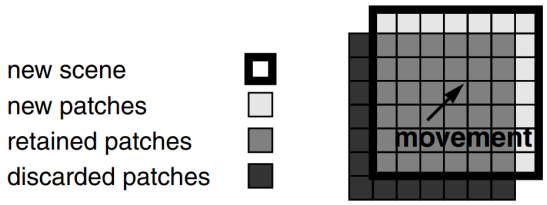
5.2 Správa paměti při vizualizaci

Manipulace se scénou po blocích umožní lepší využití paměti a celkově stabilnější vykreslování. Do paměti se načítají jednotlivé části prostoru scény a ty se dále zpracovávají. Minimálně by se měly ukládat ty části, které budou vykresleny v dalším snímku [12].

Příkladem může být interaktivní průchod architektonickou scénou Thomase A. Funkhousera a Carla H. Séquina (1992) [12]. Svou scénu si hierarchicky rozdělí do buněk (do struktury k-d stromu) a z buněk si sestaví segmenty (bloky), které uloží do zobrazovací databáze (*display database*). Segmenty se adresují jen ukazatelem do paměti. Pro rychlejší načtení dat se ukládají incidentní segmenty do databázového souboru hned za sebou. Do paměti se pak načítá segment (a další blízké segmenty), který má velkou šanci na vykreslení v příštím snímku. Zpracování viditelnosti a určení úrovní detailu se provádí právě nad těmito načtenými daty.

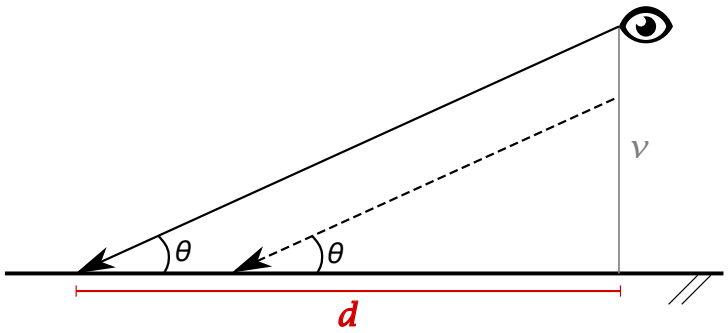
Xiang Li a kol. (2011) zase třídí scénu do hierarchických obálek [59]. Jednotlivé obálky jsou seskupeny do větší části (nazývané dlaždice), které jsou opět uloženy do databáze. Do paměti se z databáze načtou jen ty dlaždice (bloky), které je potřeba v následujícím vykreslování snímku zpracovat.

Renato Pajarola (1998) ve vizualizaci terénu načítá data do paměti po kvadrantech (*windowing*) [55]. Kvadrant je aktualizován podle pohybu kamery po menších kusech (*patches*). Některé kusy se zahodí, jiné zůstávají v paměti i po aktualizaci a potřebné kusy se nově načtou (znázorněné na obrázku 5.4).



Obrázek 5.4: Dynamické načítání (*windowing*) [55]

Počet aktivně načtených bloků dat do paměti se může měnit podle výšky pohledu pozorovatele. Čím výš stojí pozorovatel, tím víc bloků je potřeba zpracovat. Na obrázku 5.5 je znázorněn výpočet maximální vzdálenosti dohledu pozorovatele [54].



Obrázek 5.5: Vzdálenost d dohledu pozorovatele se vypočítá jako $d = v \cot\theta$, kde θ je úhel pohledu a v je výška pozorovatele

Kapitola 6

Shrnutí analýzy

Následující kapitola obsahuje stručný popis a odůvodnění výběru optimalizačních technik, které budou v prototypu (vizualizace scény) obsaženy. Rozhodnutí vychází z analýzy, napsané v předchozích kapitolách. Druhá část kapitoly pak popisuje požadavky kladené na prototyp.

6.1 Výběr optimalizačních technik

Metody LOD jsou důležitým a silným nástrojem pro optimalizaci vizualizace rozsáhlých scén jako jsou například simulace velkých měst. Taková urbanistická scéna může obsahovat obrovský počet objektů a opravdu není nutné zpracovávat a vykreslovat celé prostředí do nejmenších detailů. Vizuální vnímání objektů daleko od kamery (pozorovatele) klesá, ale počet trojúhelníků (polygonů), reprezentující tyto objekty (modely), by zůstal stejný. Bez regulování úrovně detailu jednotlivých objektů ve scéně by se jen zbytečně plýtvalo výpočetními prostředky (procesorový čas, paměť...).

Úrovně detailu modelu se budou v prototypu měnit v závislosti na velikosti projekce určitého objektu na obrazovku. Kritérium velikosti (počet pixelů, které model zabírá) je běžným způsobem použití LOD, dává dobré výsledky, je rychlý a jednoduchý na realizaci. Pro demonstraci algoritmů zjednodušující síť modelu (*mesh*) použiji v prototypu diskrétní stupně LOD. Úrovně modelu budu generovat pomocí algoritmu *Progressive meshes*, jelikož používá operátor kolaps hrany, který nepotřebuje dodatečnou triangulaci k pokrytí děr v síti (vzniklé po odstranění trojúhelníků). Algoritmus se hodí pro generování sítě i za běhu programu, při správném použití se lze vyhnout efektu problikávání při přepínání mezi sousedními úrovněmi LOD (popsáno v sekci 3.5). Další typ LOD, který bych ráda použila je hierarchické LOD. Spojením objektů do většího celku a úprava detailu nad seskupeným modelem by mělo ovlivnit nejen rychlost vykreslování snímků ale i počet volání vykreslovací funkce (*draw calls*).

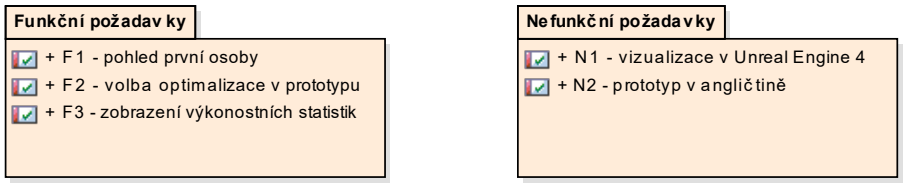
Dalším způsobem, jak zredukovat počet polygonů ke zpracování, je ignorování částí ve scéně, které pozorovatel ve výsledku neuvidí. Urbanistická scéna bude obsahovat hlavně modely budov, které jsou z větší části neprůhledné, a tudíž budou zakrývat objekty za nimi [60]. Zastíněné části ve scéně budu vymazávat pomocí základního algoritmu hierarchické paměti hloubky (nebo hierarchického z-bufferu), jelikož dokáže obstojně zpracovat komplexní scénu (nehledě na závislost na určitém hardwaru).

Další neviditelné části scény jsou například objekty mimo zorné pole uživatele nebo druhá (odvrácená) strana sítě modelu reprezentující objekt. Techniky pro odstraňování odvrácených stěn a odstraňování pohledovým objemem se dnes používají snad úplně všude, jelikož jsou velice efektivní. V prototypu budou obě metody, ale nebudu měřit jejich efektivitu.

Poslední způsob optimalizace, který bych chtěla použít je dynamické blokové načítání prostoru scény do a z paměti podle pohybu pozorovatele ve scéně.

6.2 Model požadavků

Sekce obsahuje shrnutí požadavků (obrázek 6.1), které jsou kladeny na prototyp pro vizualizaci scény. Požadavky jsou rozděleny na dvě části a to požadavky funkční a nefunkční (obecné).



Obrázek 6.1: Přehled požadavků na prototyp

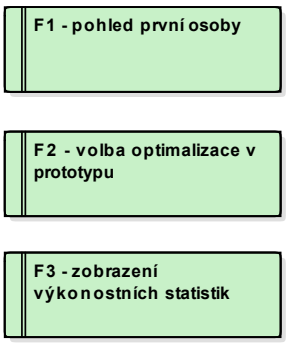
6.2.1 Funkční požadavky

Na obrázku 6.2 je seznam funkčních požadavků.

F1 Uživatel uvidí scénu v prototypu z pohledu první osoby (*first-person*). Ve scéně se pohybuje klávesami W (dopředu), A (doleva), S (dozadu) a D (doprava). Svůj pohled (otáčení se) ve virtuálním prostředí mění použitím myši.

F2 Uživateli bude umožněno si prototyp spouštět s různými optimalizačními technikami.

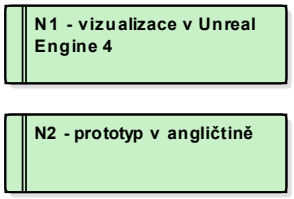
F3 V prototypu bude možné si zobrazit výkonnostní statistiky prototypu.



Obrázek 6.2: Funkční požadavky na prototyp

6.2.2 Nefunkční požadavky

Na obrázku 6.3 je seznam nefunkčních požadavků.



Obrázek 6.3: Nefunkční požadavky na prototyp

N1 Virtuální urbanistická scéna bude vizualizována v herním jádře Unreal Engine 4. V tomto enginu se budou také měřit vizualizační parametry daného prototypu.

N2 Prototyp bude zobrazovat výkonnostní statistiky v angličtině.

Kapitola 7

Návrh

Na základě předchozí analýzy a diskuzi s mým vedoucím práce jsem se rozhodla pro vizualizaci jedné 3D virtuální scény pro demonstraci vybraných optimalizačních technik (popsané v kapitole 6). Scénu bude možné vizualizovat s různými kombinacemi optimalizačních technik. Pro jednotlivé způsoby optimalizace zaznamenám hlavní vizualizační parametry scény, výsledky navzájem porovnám a nakonec použité techniky pomocí prototypu vyhodnotím.

7.1 Struktura prototypu

Scéna v prototypu bude poskládaná z volně dostupných modelů na internetových stránkách. Mým cílem je vytvořit jednoduchou oblast s velkým počtem městských budov. Touto oblastí se bude uživatel pohybovat z pohledu první osoby (*first-person*). Prostředí bude obsahovat velký počet polygonů. Modely se mohou opakovat, jednak kvůli omezeným prostředkům (zřídkakdy je každý objekt ve scéně originální) tak i kvůli demonstraci optimalizačních metod. Uživateli mohou být zobrazeny výkonnostní statistiky prototypu a bude si moci zapínat a vypínat jednotlivé způsoby optimalizace scény.

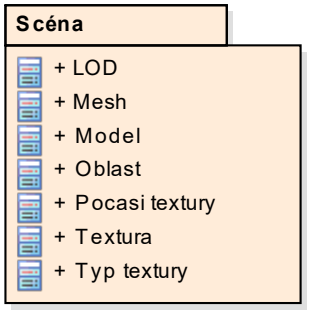
Neoptimalizovaná scéna bude výhradně obsahovat modely budov v nejvyšších možných detailech a to po celou dobu její vizualizace. Předpokládám, že v této scéně klesne výkonnost vykreslování snímků a celkově vzrostou požadavky na hardware. Tento postup mi umožní lepší porovnání parametrů různě optimalizovaných scén, jelikož by mezi nimi měly být rozdíly (týkající se vizualizačního výkonu).

Optimalizace scény by se měla provádět v omezeném měřítku. Při vytváření optimalizací se pokusím zohlednit estetickou složku scény, aby měl uživatel co nejlepší vizuální zážitek během procházení městského prostředí. Vyhodnocení vizuální stránky prototypu je velice subjektivní. Spoléhám se zde na své umělecké cítění a podle toho se pak rozhodnu o možném limitu optimalizování. Hlavní podmínkou pro stanovení hranice optimalizace pro mě bude zachování tvaru objektu (při snižování počtu polygonů v síti modelu). Ideálně

by optimalizovaná scéna měla vypadat podobně jako ta neoptimalizovaná ale samozřejmě s lepším vykreslovacím výkonem a efektivnějším využíváním prostředků (např. snížení počtu volání vykreslovací funkce).

7.2 Doménový návrh

Následující diagram (obrázek 7.2) znázorňuje návrh jednotlivých tříd ve vizualizaci města projektu VHP a zobrazuje vztahy mezi nimi.



Obrázek 7.1: Přehled tříd v prototypu

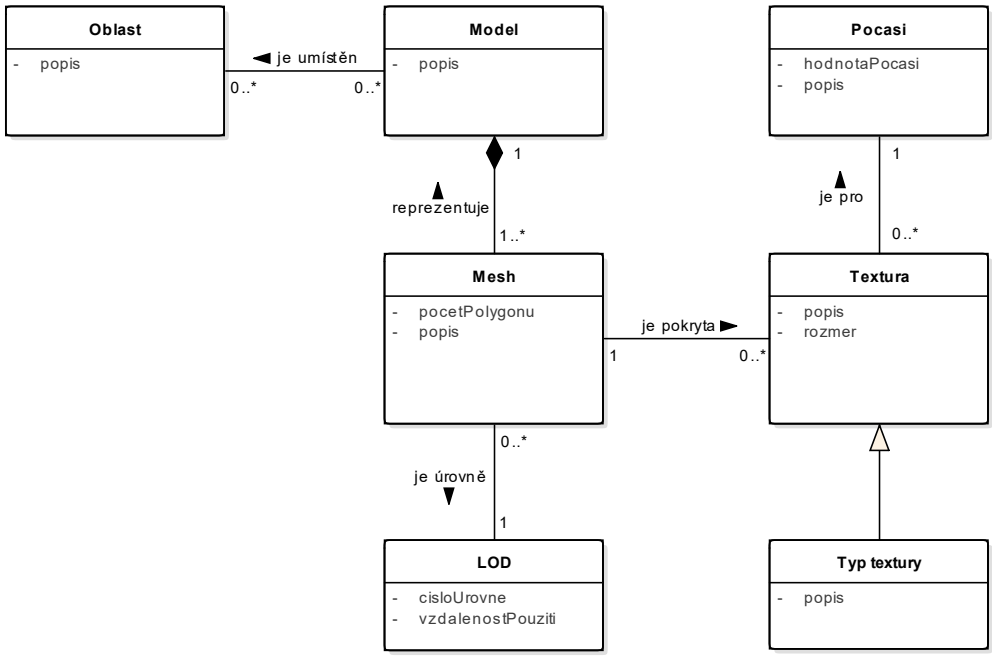
7.2.1 Popis modelu

Prvotní myšlenkou je rozdělit si město do jednotlivých oblastí. Každá oblast bude obsahovat několik modelů, jeden model se může vyskytovat ve více oblastech (opětovné použití modelu). Modely budou mít několik geometrických sítí, určující úroveň detailu modelu (LOD). Úrovně LOD se budou měnit (pro každý model) v závislosti na vzdálenosti od pozorovatele.

Sítě modelů mohou (ale nemusí) obsahovat namapované textury. Textury mohou být různého typu (difúzní, hrbolaté – *bump*, normálové...) a patří právě jedné síti modelu. U některých typů textur (např. difúzních) lze udržovat jejich informaci o vhodnosti použití podle aktuálního počasí ve scéně. Počasí je definované jako desetinné číslo v intervalu od 0 do 1 (0 značí „světlé“ počasí – například slunečno a 1 určuje „tmavé“ počasí – například zataženo). Pokud se nenajde přesná shoda textury pro dané počasí, použije se textura s nejbližší hodnotou (odchylka bude stanovena uživatelem).

Následující seznam obsahuje stručný přehled tříd a krátký popis atributů v jednotlivých třídách:

Oblast Město bude rozděleno do oblastí, každá oblast může obsahovat nějaký popis. Prototyp bude načítat město po oblastech podle toho, kde se uživatel zrovna nachází.



Obrázek 7.2: Doménový model pro VHP

Model Modelem se myslí nějaký objekt ve scéně. Třída obsahuje stručný popis modelu (např. budova, strom...).

Mesh (síť modelu) *Mesh* neboli síť modelu je 3D počítačová reprezentace určitého modelu ve scéně. Třída obsahuje atribut `pocetPolygonu`, udávající počet polygonů (trojúhelníků), ze kterých se síť skládá.

LOD Třída definuje úroveň detailu dané sítě modelu, atribut `cisloUrovne` značí danou úroveň (0 – nejvyšší úroveň). Atribut `vzdalenostPouziti` je vzdálenost od uživatele, od které se model přepne na tuto úroveň detailu.

Textura Textura popisuje povrch právě jedné sítě modelu (*mesh*). Textura má dané rozměry (např. 128×128) a obsahuje stručný popis.

Typ textury Třída znázorňuje typy textur (difúzní, normálové...).

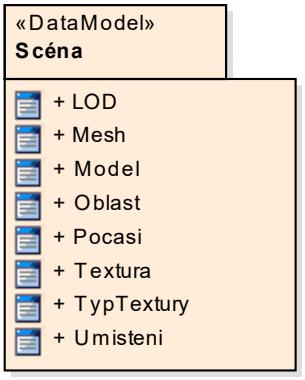
Pocasi Třída definuje hodnotu počasí dané textury. Textura není závislá na počasí pokud je atribut `hodnotaPocasi` záporný. Taková textura se vždy namapuje na síť modelu (*mesh*).

7.3 Úložiště modelů

Všechny objekty (3D modely) použité v projektu VHP, jejich textury a další metadata budou ukládané v databázi PostgreSQL[9]. Na databázový SQL dotaz by se měla vracet přesná struktura daného modelu v určité oblasti scény (jasně určená síť modelu, vhodně zvolené namapování textur podle počasí atp.).

7.3.1 Databázový model

Tato sekce obsahuje navržený způsob ukládání dat do objektově-relační databáze PostgreSQL. Všechny tabulky budou uložené v jedné databázi (přehled je na obrázku 7.3). Databázový model je na obrázku 7.4.



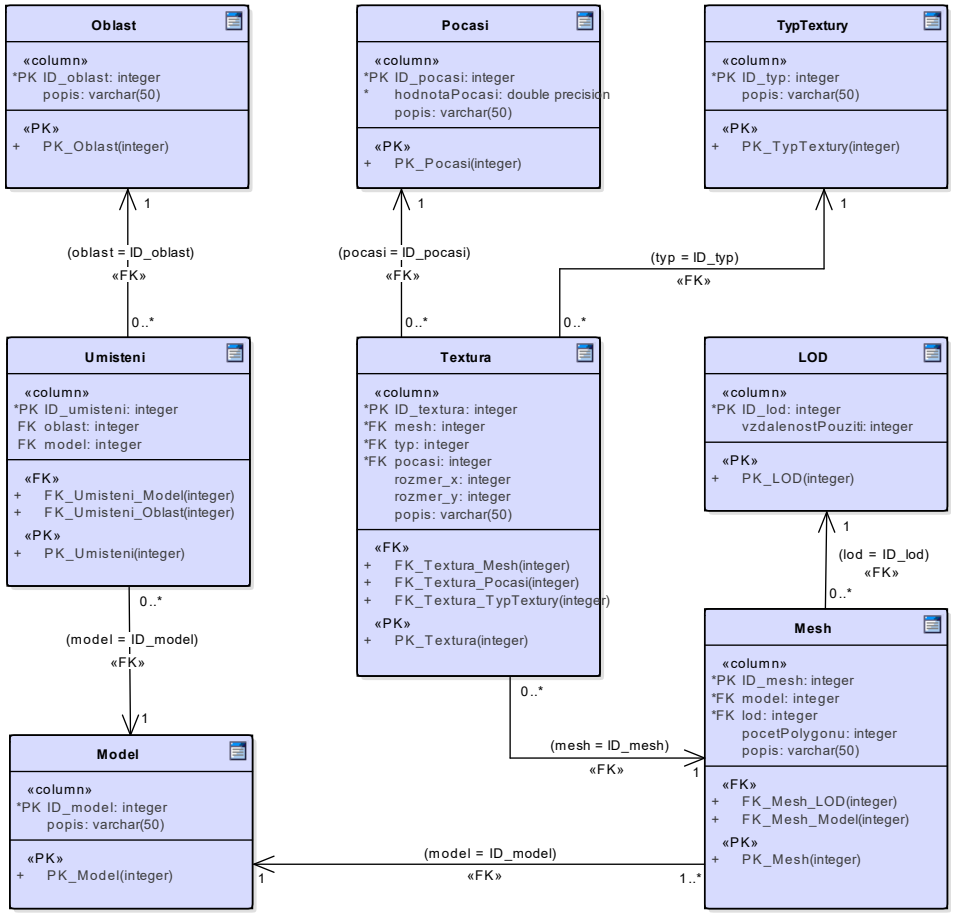
Obrázek 7.3: Přehled tabulek v databázy

7.3.2 Formát ukládání 3D modelů

Herní jádro Unreal Engine podporuje dva formáty 3D modelů: FBX (*filmbox*) a OBJ (*object file*). FBX je proprietární formát (s příponou `fbx`), jenž je kompatibilní s mnoha 3D nástroji třetích stran. Původně byl vyvinutý firmou Kaydara, nyní je ve vlastnictví korporace Autodesk. Formát OBJ (přípona `obj`) pochází od firmy Alias Wavefront. Jedná se o jednoduchý textový soubor, reprezentující 3D geometrii modelu. Soubor si ukládá pozice vrcholů a jejich normály, souřadnice textur, stěny (*faces*) sítě apod.

3D modely a jejich úrovně LOD budu generovat v aplikaci Blender. Tento 3D editor má svůj vlastní interní formát (s příponou `blend`), který lze bez problémů exportovat do obou formátů (jak FBX tak i OBJ). Pro své modely použiji spíše formát OBJ, jelikož se v aplikaci Blender lépe exportuje a pro potřeby vytvoření prototypu je formát obecnější.

[9] https://www.postgresql.org



Obrázek 7.4: Databázový model pro projekt VHP

7.4 Použité technologie k vytvoření prototypu

Zde jsou sepsány všechny programy a technologie použité k vytvoření prototypu (a jeho návrhu).

Blender [61] Blender je otevřený (*open-source*) software pro modelování a vykreslování 3D počítačové grafiky. V této aplikaci budu provádět úpravu použitých 3D modelů. Skripty pro Blender jsou napsané v jazyce Python a jsou spustitelné přímo v editoru Blender.

Unreal Engine [62] Unreal Engine (UE) je hlavní vizualizační nástroj mé práce. Jedná se o herní jádro (engine) společnosti Epic Games. Software je od verze 4 také s otevřeným zdrojovým kódem. Veškeré programování v tomto jádře je založené na jazyce C++. V enginu je navíc dostupné vizuální skriptování (v šablonách – *Blueprints*) pomocí uzlů (*Nodes*).

Microsoft Visual Studio [63] Jedná se o vývojové prostředí (IDE – *integrated development environment*) od společnosti Microsoft. Používám ho zejména pro otevírání a psaní kódu v jazyce C++ z Unreal Engine.

Enterprise Architect [64] Enterprise Architect (EA) je nástroj pro systémovou analýzu a návrh. V tomto programu jsem vytvářela hlavně UML diagramy pro návrhovou část své práce.

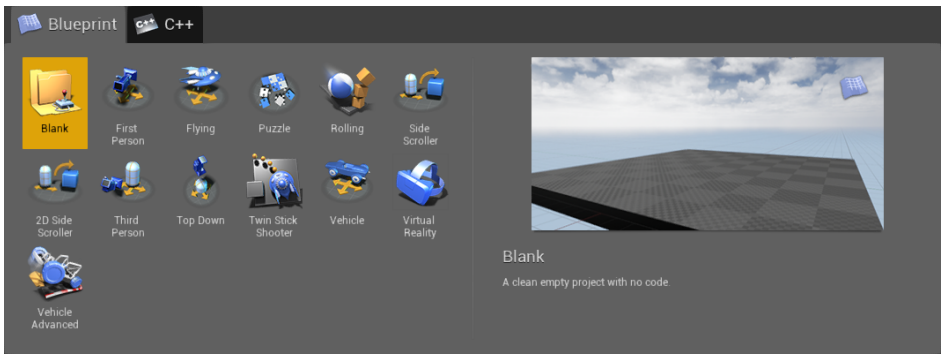
Kapitola 8

Implementace prototypu

Prototyp je vytvořen pomocí softwaru Unreal Engine od společnosti Epic Games, který převážně slouží k vývoji her [62]. Další možné využití herního enginu (jádra) je vizualizace různých scén (například urbanistických) a následné procházení těmito scénami.

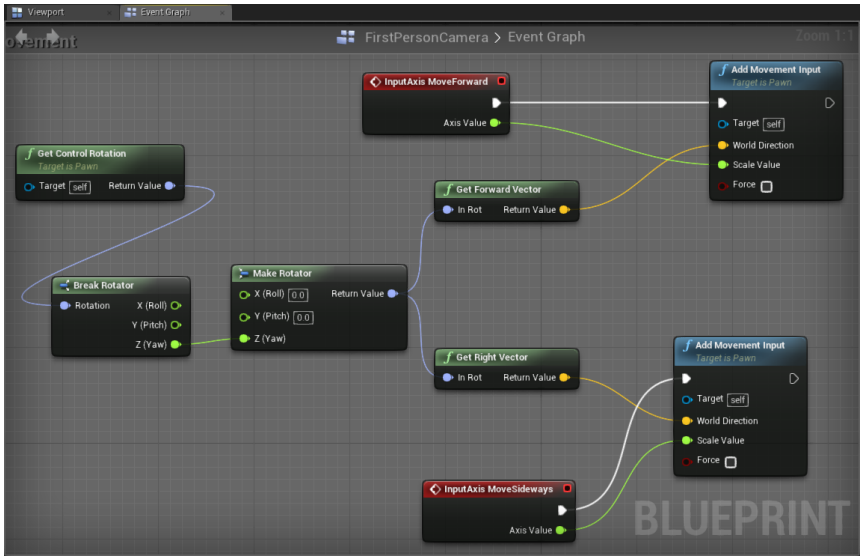
Pro práci jsem využila jádro ve verzi 4.17.2, později jsem přešla na jádro verze 4.19.2 kvůli některým chybám při měření statistických dat v prototypu. Obě verze lze obecně označit jako Unreal Engine 4 (UE4). Engine je volně stažitelný a obsahuje všechny dostupné funkce herního jádra až do nejnovější verze 4. Od této verze je možné implementovat kód v C++, což umožňuje vývojářům mít větší kontrolu nad svým projektem. Unreal Engine má i otevřený zdrojový kód[10], který je přístupný po vytvoření uživatelského účtu na hlavních stránkách enginu.

V programu využívám především šablony (*Blueprints*, obrázek 8.1), ve kterých je možnost vizuálního skriptování za použití uzlů (*nodes* – ukázka na obrázku 8.2). Uzly slouží hlavně k definování a práci s objektově orientovanými třídami.



Obrázek 8.1: Poskytnuté šablony v Unreal Engine 4

[10] https://github.com/EpicGames



Obrázek 8.2: Ukázka použití uzlů (*nodes*) v Unreal Engine 4

8.1 Implementační zajímavosti

Pro vytvoření městské scény jsem použila víceméně prázdnou šablonu (*Blank blueprint*). Jedná se o minimalistickou šablonu, která obsahuje jen základní prostředí (*skybox*[11], několik typů osvětlení a hudbu v pozadí). V šabloně bylo potřeba vytvořit novou kameru z pohledu první osoby, jelikož stará kamera umožňovala pozorovateli volný pohyb v prostoru scény (pozorovatel v ní mohl „létat“).

Městskou scénu v prototypu negeneruji ale skládám ji ručně. Celý prostor je rozdělen do čtyř částí (městských bloků), které se načítají postupně podle pohybu pozorovatele. Část města se sestává z vysokých budov a druhá část zase obsahuje nižší rodinné domky. Zvolila jsem tuto sestavu, abych mohla demonstrovat algoritmy viditelnosti ve scéně (především *occlusion culling* – odstraňování zastíněných částí scény). Město obsahuje kolem 120 budov, které dohromady dávají zhruba dva miliony trojúhelníků.

8.1.1 Použité modely

Modely budov je možné získat dvěma způsoby. První možností je vytvořit si model v nějakém 3D editoru, například editory 3ds Max, Maya či Blender (a další). Tento proces je ale složitý, zdlouhavý a výsledek záleží na dovednosti

[11] Metoda 3D virtuálních prostředí, která vytváří iluzi otevřeného světa. Celé prostředí scény je uzavřené v geometrickém útvaru (obvykle krychle), která má na svých stěnách namapovanou texturu pozadí.

modeláře. Vzhledem k tomu, že modelování budov není hlavní náplní této práce, rozhodla jsem se využít druhý způsob, a to získat již hotové modely od jiných uživatelů internetu. Jak už to většinou bývá, modely zdarma nejsou úplně dokonalé, a proto je ještě upravuji v 3D editoru.

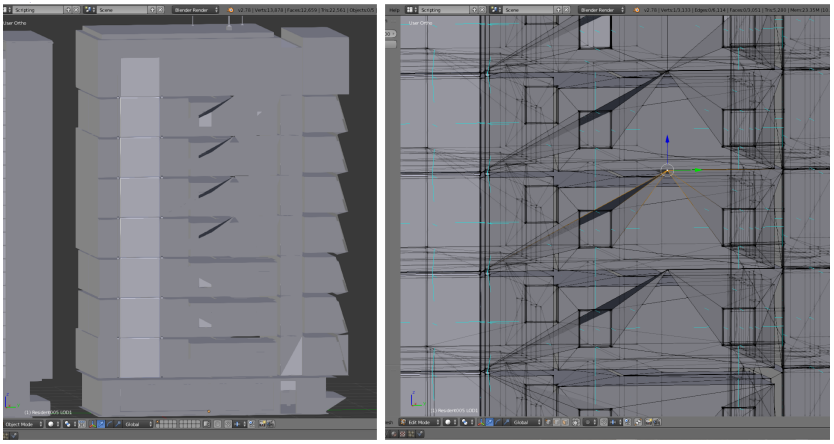
Prototyp se skládá z několika typů budov, z nichž některé jsou zobrazeny na obrázku 8.3. Všechny použité 3D modely a jejich materiály i textury upravuji v programu Blender. Použité modely mají většinou počet polygonů v rozmezí od 3 500 (rodinné domky) do 75 000 (panelové domy), zhruba tři až sedm materiálů a většinou jednu nebo žádnou texturu.

Obrázek 8.3: Ukázka některých modelů, které jsou v prototypu používány

V rámci ukázky diskrétních stupňů LOD jsem pro každou budovu vytvořila čtyři úrovně detailu pomocí metody Progresivních sítí (popsáno v další sekci 8.1.2). Nultá úroveň modelu, označená LOD0, reprezentuje originální nedecimovaný model. LOD1 má přibližně 75 % vrcholů z původního modelu, LOD2 pak 50 % a poslední LOD3 má jen cca 35 % (obrázek 8.4).

Obrázek 8.4: Ukázka čtyř úrovní LOD: úplně vlevo je síť úrovně LOD0 a úplně vpravo je síť stejného modelu v nejnižších detailech (LOD3)

Během generování decimovaných sítí z originálního modelu jsem se občas setkávala se špatně ohnutou sítí (obrázek 8.5), způsobenou operátorem kolapsu hrany. Tyto chyby jsem ve své implementaci progresivních sítí neřešila a manuálně je odstranila, jelikož jich mnoho nebylo.



Obrázek 8.5: Chyby v decimované síti (vzniklé použitím metody progresivních sítí)

8.1.2 Progresivní sítě

Metodu Progresivních sítí jsem naimplementovala v jazyce Python jako zásuvný modul do editoru Blender. Algoritmus snižuje složitost modelu opakovaným odstraňováním hran (pomocí operátoru *half-edge collapse*, popsaný v sekci 3.4.4). Jako odstraňování (kolaps) hrany je tedy myšleno spojení jednoho vrcholu ke druhému. Odstraňování probíhá tak, aby nevznikla velká „vizuální změna“ oproti originální síti.

V algoritmu si nejprve vypočítám váhy (energetickou funkci jako koeficient zakřivení) incidentních hran ke všem vrcholům v modelu. Váha (funkce *cost*) hrany (u, v) se počítá podle následujícího vzorce [29]:

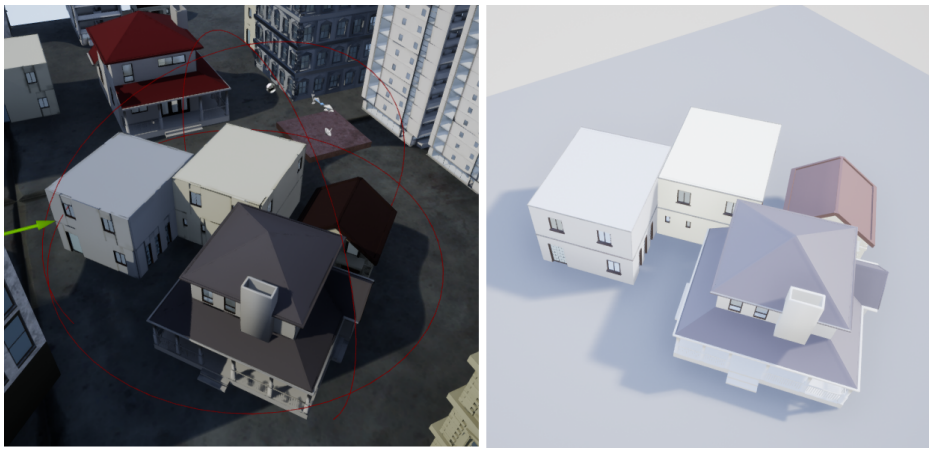
$$cost(u,v) = \|u\ -v\ \| \times max\{min(1 - \frac{\vec{n_t} \cdot \vec{n_s}}{2})\} \tag{8.1}$$

Vektory $\vec{n_t}$ jsou normály všech trojúhelníků s vrcholem u a $\vec{n_s}$ jsou normály všech trojúhelníků s hranou (u, v). Pro každou hranu se váha spočítá dvakrát (jednou z vrcholu v do u a podruhé z u do v) a může mít pokaždé rozdílnou hodnotu (tj. posun vrcholu v do u vede k jinému tvaru sítě než posun vrcholu u do v). Hrany s nejmenší váhou vkládám do minimové binární haldy. Postupně z této haldy hrany odebírám a odstraňuji, dokud nemá síť modelu menší počet vrcholů (rozsah decimace je určený uživatelem).

8.1.3 Hierarchické LOD

Kromě diskrétního LOD lze v UE4 použít nástroj pro vytvoření hierarchického LOD. Hierarchické LOD (nebo HLOD) v určité vzdálenosti od pozorovatele spojí několik statických sítí objektů do jedné větší sítě (*cluster* – shluk). Takový shluk obsahuje všechny materiály a textury jednotlivých objektů a

přitom je potřeba jen jedno volání vykreslovací funkce nad celou skupinu objektů (ukázka shluku na obrázku 8.6). Generování hierarchického LOD pro velmi rozsáhlé scény může trvat dlouhou dobu, proto jsem si rozdělila město do jednotlivých bloků a pro každý blok vytvářím HLOD zvlášť (popsané v další sekci 8.1.4).



Obrázek 8.6: Červená drátová koule znázorňuje jeden shluk budov, nad kterým se vytvoří hierarchické LOD

8.1.4 Městské bloky

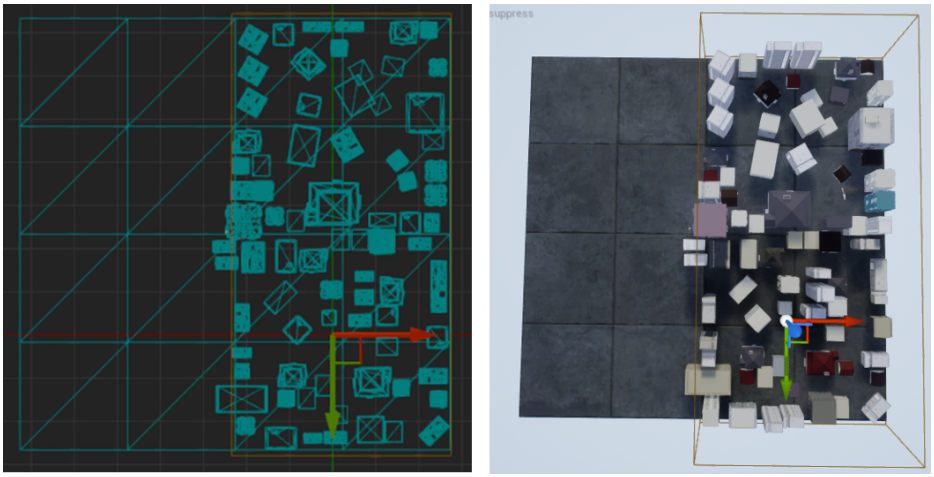
Scénu lze rozdělit do bloků pomocí nástroje *Levels*, který poskytuje lepší orientaci při práci s celou scénou města. Pro městské bloky je možné dodefinovat způsob jejich načítání do paměti pomocí nástroje *Streaming volumes* (použití je znázorněno na obrázku 8.7).

8.1.5 Viditelnost

V enginu je k dispozici několik metod pro určení viditelnosti objektů ve scéně. K dosažení optimálních výsledků se metody obvykle používají kombinovaně. Metody eliminování neviditelných částí scény jsou ve výchozím nastavení enginu UE zapnuté.

Pro každou primitivní komponentu scény se sleduje její stav viditelnosti a na základě toho se vytvoří *occlusion query* (struktura reprezentující způsob, jak určit, zda je objekt viditelný[12]). Tato struktura se čte z grafického hardwaru o jeden snímek později, tudíž se určení viditelnosti může trochu zpozdit. Zpožděné čtení může v některých případech vyvolat „vyskočení“ objektu do scény (např. když pozorovatel zahne do rohu nebo příliš rychle otáčí kamerou).

[12] *Occlusion query* vrací počet fragmentů (trojúhelníků), které se mají vykreslit podle ohraničující obálky objektu.



Obrázek 8.7: Obalení dvou bloků města do načítávacího objemu (*streaming volume*)

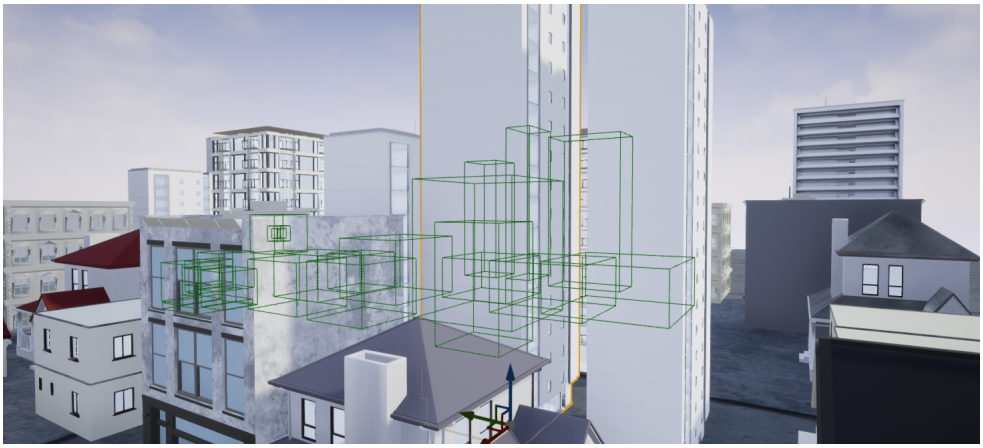
V UE4 je možné vynutit si vytváření *occlusion queries* pomocí klasické paměti hloubky (rasterizací), což je výchozí systém v hardwaru. Metoda je efektivní a dosahuje vysoké rychlosti zpracování, zvlášť když je podporována většinou dnešních grafických procesorů. Bohužel metoda klade určité nároky na paměť.

Implicitně se v UE4 viditelnost určí pomocí pohledového objemu kamery a ohraničující obálky objektu. Na obrázku 8.8 lze vidět ohraničující obálky objektů, které byly zastíněny vyššími budovami. Přesnost ohraničující obálky lze měnit podle relativní velikosti objektu v prostoru scény využitím hierarchické paměti hloubky (hierarchického z-bufferu). Metoda pomáhá redukovat efekt „vyskakování“ objektů do obrazovky. V UE4 je implementace metody hierarchického z-bufferu (označeno Hi-Z nebo HZB) stále považována za experimentální, a proto se doporučuje jen pro rozsáhlé scény s velkým množstvím stínících objektů.

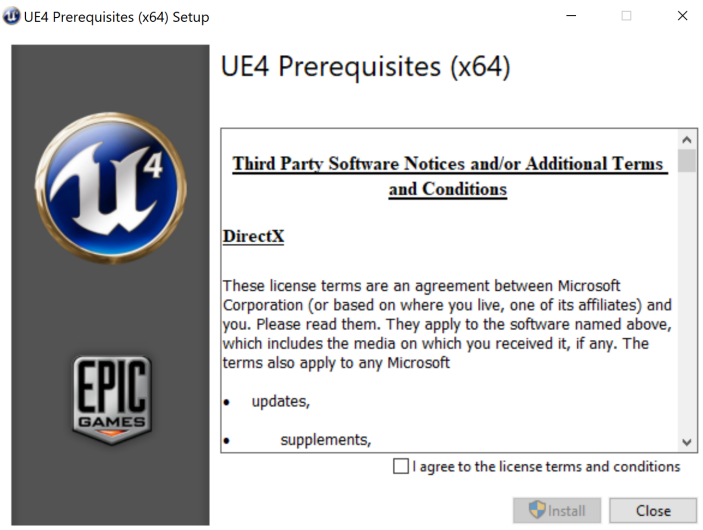
Pro scénu lze také nastavit hranice vzdálenosti (v jednotkách Unreal enginu) pro zpracování a vykreslení objektů ve scéně. Jedná se o nastavení přední a zadní omezující roviny pohledového objemu kamery. Vzdálenost (v prostoru scény – *world space*) se měří od středu ohraničující obálky objektu do pozice kamery.

8.2 Instalační příručka

Ke spuštění prototypu je třeba mít nainstalovaný operační systém Windows 8 či Windows 10 s podporou DirectX alespoň verze 11. Doporučuji alespoň 2 GB volného místa na disku a 2 GB RAM.



Obrázek 8.8: Vysoké budovy zakrývají objekty za nimi (ohraničující obálky těchto objektů jsou vyznačeny zeleně)



Obrázek 8.9: Okno pro instalaci prerekvizit UE4 pro spuštění prototypu

Z přiloženého DVD pak stačí spustit příslušný soubor s příponou `exe` z adresáře `bin`. Pokud v počítači chybí modul DirectX Runtime, zobrazí se požadavek (s názvem *Error*) na jeho doinstalování. Modální okno stačí odsouhlasit a po následování pokynů na obrazovce by mělo dojít k úspěšnému nainstalování potřebných modulů a dalších prerekvizit UE4 (obrázek 8.9). Po chvilce se prototyp spustí a uživatel se objeví na startovní pozici městské scény (na měděné ploše zobrazené na obrázku 8.10).

Obrázek 8.10: Po spuštění prototypu se zobrazí tato scéna

8.3 Návod k použití a uspořádání prototypu

Prototyp se ovládá převážně klávesnicí a myší. Myší lze měnit úhel pohledu kamery, klávesnicí lze (kromě pohybu kamery do stran) spínat optimalizační techniky a zobrazovat tabulky statistik. Následující seznam obsahuje popis funkcí jednotlivých kláves (uspořádání pro anglickou klávesnici):

W, A, S, D klasický pohyb kamery v prostoru scény,

F zobrazí/skryje aktuální snímkovou frekvenci (FPS) scény,

L vypne/zapne metodu diskrétního LOD ve scéně,

H vypne/zapne metodu hierarchického LOD ve scéně,

O vypne/zapne metodu eliminace neviditelných částí scény,

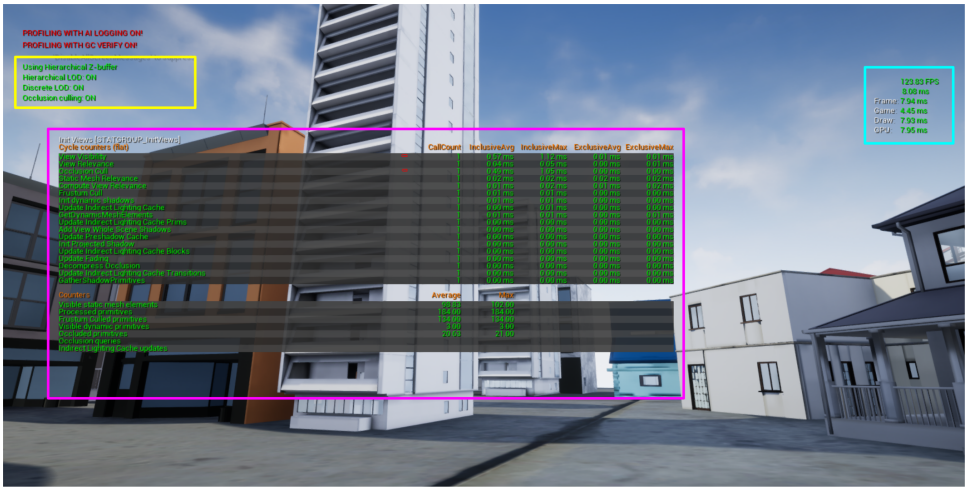
Z vypne/zapne metodu eliminace neviditelných částí scény pomocí hierarchické paměti hloubky (pokud se vypne, použije se standardní systém v hardwaru – *hardware occlusion queries*),

1/Num 1 zobrazí/skryje informace z příkazu `stat unit`,

2/Num 2 zobrazí/skryje tabulku příkazu `stat sceneRendering`,

3/Num 3 zobrazí/skryje tabulku příkazu `stat initViews`,

4/Num 4 zobrazí/skryje tabulku příkazu `stat rhi`,



Obrázek 8.11: Ukázka uspořádání informací v prototypu: žluté ohraničení obsahuje informace o spínání optimalizačních technik, růžové ohraničení definuje místo pro statistické tabulky, modré ohraničení znázorňuje informace o rychlosti vykreslování a frekvenci snímků

5/Num 5 zobrazí/skryje tabulku příkazu `stat memory`,

6/Num 6 zobrazí/skryje tabulku příkazu `stat gpu`,

‘ (zpětný apostrof) spustí v prototypu příkazovou řádku,

ESC ukončí spuštěný program.

Při přepínání jednotlivých optimalizací se vlevo nahoře obrazovky zobrazí informace o zapnutí či vypnutí dané metody. Statistické tabulky se zobrazí přes celou levou část obrazovky podle pořadí stisknutých kláves. Všechny tabulky se na jednu obrazovku nevejdou, je nutné je zase skrýt stiskem stejné klávesy. Informace o snímkové frekvenci je napravo obrazovky. Celé uspořádání prototypu je znázorněno na obrázku 8.11.

Prototyp se může spustit přímo z terminálu zadáním názvu prototypu (`city_3`) z příslušného adresáře (`bin`). Pro dodatečné sledování využité paměti lze použít přepínač `memreport -full`. Výsledky monitorování se uloží do logu (žurnálu), který se po každém spuštění prototypu ukládá automaticky do adresáře `bin\city_3\Saved\Logs`.

Kapitola 9

Vizualizační parametry prototypu

Následující kapitola popisuje zkoumané vizualizační parametry prototypu a způsob jejich měření. Na základě naměřených dat poté vyhodnocuji použité techniky optimalizace.

9.1 Zkoumané parametry

Ke sledování vizualizačních parametrů (*profiling*) během spuštěného prototypu používám nástroje přímo zabudované v Unreal Engine 4. Nástroj *Profiler* umožní načíst uložený soubor, který zachycuje (*capture*) hodnoty zkoumaných parametrů během vizualizace scény. V nástroji lze přesně definovat časový úsek nahrávání hodnot.

Parametry, kterými se budu v prototypu zabývat jsou:

1. snímková frekvence (*frame rate*),
2. počet volání vykreslovací funkce (*mesh draw calls*),
3. počet viditelných elementů statických sítí (*visible static mesh elements*),
4. čas pro vyřazení neviditelných částí objektů (*visibility culling time*).

Snímková frekvence udává rychlost vykreslování snímků za sekundu (FPS – *frames per second*). Frekvence se zjistí pomocí příkazu `stat fps`. Tento příkaz zobrazí i čas potřebný k vykreslení jednoho snímku v milisekundách (snímková frekvence se spočítá vydělením 1000 touto hodnotou – takže čím déle trvá vykreslení jednoho snímku, tím méně se jich vejde do rozmezí jedné sekundy).

Pro přesnější měření hodnoty snímkové frekvence se v UE4 vypíná vlastnost *Smooth frame rate*, která má tendenci zamezit prudkým skokům v hodnotách FPS. Snímková frekvence se hodí pro různá výkonnostní porovnávání

(*benchmark*), ale hodnota sama o sobě nic neříká o efektivnosti použitých technik optimalizace. Proto se v prototypu zabývám dalšími vizualizačními parametry.

Příkaz `stat unit` ukazuje, kolik času (v milisekundách) zabírá každá komponenta ke zpracování jednoho snímku:

- Frame: Celkový čas pro zpracování jednoho snímku (velmi podobný hodnotě v příkazu `stat fps`).
- Game: Čas pro CPU (vykonávání kódu, fyzika ve hře...).
- Draw/GPU: Čas pro GPU (vykreslování scény). Pro zobrazení podrobnější statistiky ohledně grafického procesoru lze použít příkaz `stat gpu`.

Pro získání statistik souvisejících přímo s vykreslováním scény se používá příkaz `stat sceneRendering`. Nejdůležitější z toho je parametr `InitViews`, jenž udává, jak dlouho trvá provést kontrolu viditelnosti (k eliminaci neviditelných částí ve scéně). Pro zjištění tohoto parametru lze použít i příkaz `stat initViews`, který navíc obsahuje informaci o konečném počtu viditelných elementů statických sítí (`Visible static mesh elements`).

Příkaz `stat rhi` zobrazí informace z renderovacího rozhraní (DirectX). Z příkazu jsou důležité hodnoty `DrawPrimitive calls` (počet volání vykreslovací funkce i s pomocnými sítěmi modelů) a `Triangles drawn` (počet vykreslených trojúhelníků). Počet vykreslených trojúhelníků se může lišit od skutečného počtu trojúhelníků ve scéně, protože se modely mohou několikrát zpracovat (pro výpočet vyřazení kvůli viditelnosti, osvětlení, vykreslení modelu s několika materiály apod.).

Dalším zajímavým parametrem může být využití paměti při blokovém načítání. Pro zobrazení, kolik paměti se využije při vizualizaci, se použije příkaz `stat memory`. Protože budou ve scéně hlavně sítě modelů (s minimálním množstvím klasických textur[13]), je tato hodnota čistě orientační a lze použít jen příkaz `stat memoryStaticMesh` pro demonstraci funkce blokového načítání.

9.2 Výsledky měření prototypu

V prototypu jsem změřila vybrané parametry pro jednotlivé optimalizační techniky. Data z měření mohou být na každé počítačové konfiguraci rozdílné. Následující výsledky z měření jsem získala na této počítačové sestavě:

- procesor: osmijádrový AMD FX-8350 s frekvencí 4.0 GHz,
- grafická karta: NVIDIA GeForce GTX 960 4 GB,

[13]Scéna obsahuje zejména *lightmap* textury, jež ukládají informace o osvětlení objektu ve scéně.

- velikost RAM: 16 GB,
- disk: SSD.

Vizualizační parametry jednotlivých technik (a jejich další kombinace) budu měřit v následujícím pořadí (seznam slouží pro identifikaci typu měření v tabulkách 9.1 až 9.3):

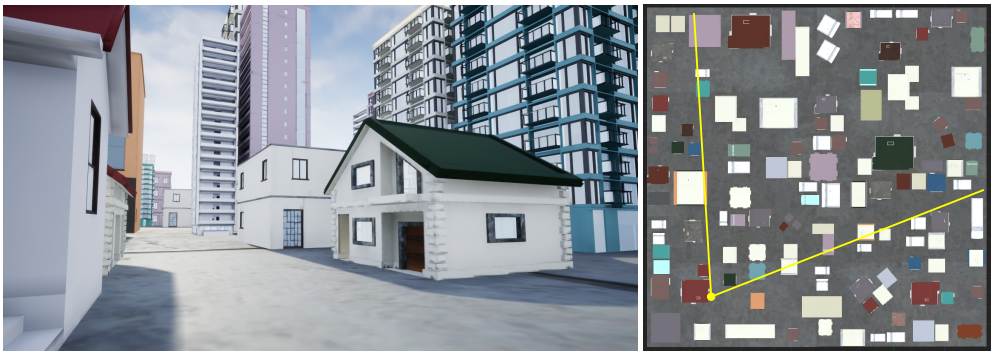
1. bez optimalizace,
2. diskrétní LOD,
3. hierarchické LOD,
4. odstraňování zastíněných částí scény jen pomocí výchozího systému v hardwaru (*hardware occlusion queries*),
5. odstraňování zastíněných částí scény pomocí hierarchického z-bufferu (hierarchické paměti hloubky),
6. kombinace diskrétního a hierarchického LOD,
7. kombinace diskrétního a hierarchického LOD, odstraňování zastíněných částí scény jen pomocí *hardware occlusion queries*,
8. kombinace diskrétního a hierarchického LOD, odstraňování zastíněných částí scény pomocí hierarchického z-bufferu.

Každý typ měření probíhá ve stylu zachycení zkoumaných parametrů scény v rozmezí 20 sekund. Výsledek, který zaznamenám je průměr všech zaznamenaných hodnot během tohoto intervalu. Celkem měřím hodnoty pro tři různé scény v prototypu.

9.2.1 První měření

Jako první jsem v prototypu zvolila tuto scénu na obrázku 9.1 ke změření vybraných parametrů. Scéna obsahuje právě několik malých domečků a vysoké budovy, které záměrně zakrývají pokračování pravé části scény. Scéna naopak pokračuje vlevo do dáli kde stojí několik vzdálených budov. Naměřená data z první scény jsou vypsané v tabulce 9.1.

Z měření lze zaznamenat určité zrychlení ve vykreslování pokud se na objekty ve scéně použijí diskrétní či hierarchické úrovně detailu. Diskrétní LOD je na tom o něco lépe, protože se ve výsledku zpracovává menší počet polygonů (měření typu 2). Hierarchické LOD přesto docela efektivně snižuje počet volání vykreslovací funkce, i když je snímková frekvence ve výsledku o něco nižší než při aplikování diskrétního LOD (měření typu 3). Hodnoty v měření typu 6 zase ukazují kombinované zlepšení při používání obou metod úrovní detailu



Obrázek 9.1: První měření proběhlo na této scéně, napravo je znázorněn pohled ze shora pro první scénu

typ měření	snímková frekvence (FPS)	volání vykreslovací funkce	viditelné elementy sítě	čas eliminace (ms)
1	100,25	466,9	270	0,178
2	104,35	424,92	249	0,178
3	101,49	373,92	217	0,185
4	106,7	144,97	87	0,388
5	99,42	218,96	131	1,053
6	104,16	341,95	201	0,183
7	107,67	158,97	94	0,379
8	101,61	192,96	116	0,921

Tabulka 9.1: Průměr naměřených hodnot z první scény

na modely budov ve scéně (vysoká snímková frekvence při využití diskrétního LOD a méně časté volání vykreslovací funkce během hierarchického LOD).

Čas eliminace neviditelných částí ve scéně se u měření typu 1 až 3 a 6 týká především odstraňování odvrácených stěn a odstraňování pohledovým objemem, jelikož odstraňování zastíněných objektů je v těchto měření vypnuté. Z tabulky je zřejmé, že jsou všechny zmíněné časové hodnoty podobné.

Pozoruhodným jevem je rozdíl v časech při odstraňování zastíněných částí ve scéně (optimalizační metoda *occlusion culling*). Hierarchický z-buffer (měření typu 5) je oproti klasickému systému v hardwaru (*hardware occlusion queries*, měření typu 4) pomalý a neefektivní. Podle předpokladů by měl hierarchický z-buffer snižovat nároky kladené na procesor a grafickou kartu, což v tomto případě není tak úplně pravda. Čas strávený nad eliminací neviditelných částí je skoro třikrát delší než u klasické paměti hloubky. Metoda hierarchického z-bufferu navíc konzervativně „nadhodnocuje“ viditelnost objektů,

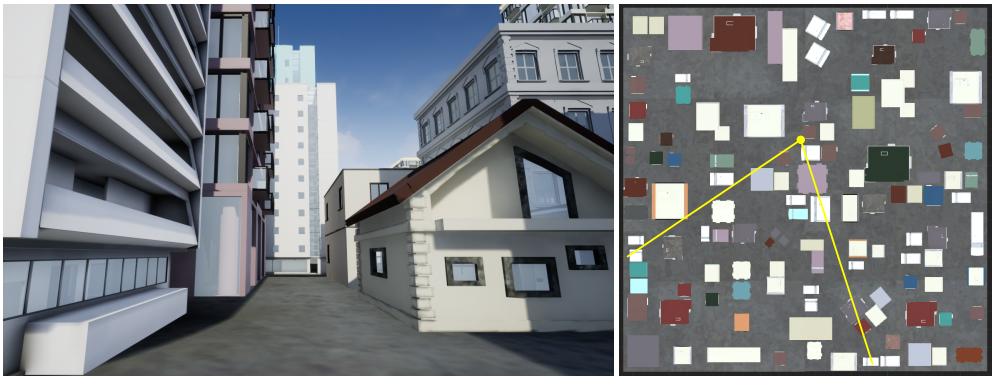
což lze vidět v rozdílu počtu viditelných elementů statických sítí v měření typu 4 a 5. Tento konzervativní přístup výpočtu viditelnosti má snižovat efekt „vyskakování“ objektů do scény.

Prvním důvodem neefektivity HZB (hierarchického z-bufferu) může být rozsah prostoru scény. Scéna neobsahuje dostatečný počet objektů pro plné využití potenciálu hierarchického z-bufferu. Ve scéně navíc nepoužívám efekty zastínění okolím (*ambient occlusion*, neboli AO) ani odrazy v prostoru obrazu (*screen space reeflections* – SSR). Obojí (AO a SSR) spoléhá na hierarchický z-buffer, který nakonec jen zbytečně plýtvá výpočetním časem grafického procesoru.

Z těchto důvodů nakonec vychází měření typu 7 nejlépe, jelikož standardní systém na použitém grafickém procesoru dokáže přesněji určit viditelnost objektů. Díky efektivnějšímu odstraňování zastíněných částí také prudce spadl počet volání vykreslovací funkce a snímky se začaly rychleji zpracovávat.

9.2.2 Druhé měření

Druhá scéna, na které jsem měřila hodnoty parametrů je z velké části zakryta budovami z obou stran. Pohled do dáli je blokován vysokou budovou (scéna je na obrázku 9.2). Výsledky druhého měření jsou zaznamenány v tabulce 9.2.



Obrázek 9.2: Druhé měření proběhlo na této scéně, napravo je znázorněn pohled ze shora pro druhou scénu

Výsledky druhého měření jsou nakonec velmi podobné prvnímu měření. Znovu byla zaznamenána menší efektivita metody HZB, a tudíž je nejvýhodnější postup optimalizace v měření typu 7. Jediným rozdílem v druhém je vyšší efektivita při odstraňování zastíněných objektů (zejména v měření typu 4), protože je zpracovaná scéna „menší“ (pohledový objem je blokován ze všech směrů).

V konečné optimalizaci (bez využití hierarchického z-bufferu, v měření typu 7) se metody úrovní detailů tolik neprojeví. Počet volání vykreslovací

typ měření	snímková frekvence (FPS)	volání vykreslovací funkce	viditelné elementy sítě	čas eliminace (ms)
1	101,4	406,92	238	0,192
2	105,75	380,93	225	0,197
3	101,54	366,93	216	0,195
4	112,33	82,98	47	0,277
5	104,81	120,97	71	0,868
6	105,92	346,96	206	0,194
7	112,89	80,98	46	0,287
8	106,35	116,98	69	0,845

Tabulka 9.2: Průměr naměřených hodnot z druhé scény

funkce je skoro stejný jako bez použití LOD v měření typu 4. Daná scéna totiž obsahuje malý počet objektů, na kterých lze zmenšit počet polygonů nebo je sloučit do skupiny.

9.2.3 Třetí měření

Třetí scéna, kterou jsem zvolila, obsahuje vepředu spoustu nižších domečků a vyšší budovy stojí spíše v zadní části. Ukázka scény je na obrázku 9.3. Třetí měření je zaznamenáno v tabulce 9.3.

Obrázek 9.3: Třetí měření proběhlo na této scéně, napravo je znázorněn pohled ze shora pro třetí scénu

Ve třetí scéně je nutné zpracovat a vykreslit velký počet budov, a proto je metoda diskrétního LOD natolik efektivní, aby zvýšila rychlost vykreslování snímků o více než 7 FPS (měření typu 2). Nicméně celkové měření dopadlo obdobně jako první a druhé měření.

typ měření	snímková frekvence (FPS)	volání vykreslovací funkce	viditelné elementy sítě	čas eliminace (ms)
1	95,9	702,86	418	0,219
2	103,59	630,88	382	0,23
3	98,66	404,92	243	0,206
4	103,15	306,96	185	0,629
5	94,95	453,91	281	1,264
6	101,78	380,95	231	0,201
7	106,03	259,73	156	0,402
8	98,56	340,98	209	0,994

Tabulka 9.3: Průměr naměřených hodnot ze třetí scény

Na naměřených hodnotách lze opět zpozorovat, že metody LOD a odstraňování zastíněných objektů pomocí klasického systému v hardwaru, se navzájem dobře doplňují a efektivně zrychlují vykreslování snímků (měření typu 7). Metoda hierarchického z-bufferu (měření typu 5 a 8) má stále nízký výkon, přestože se ve scéně zpracovává více budov než v první scéně.

Kapitola 10

Uživatelské testování

Testování prototypu bude v první řadě zaměřeno na vizuální vnímání městského prostředí pozorovatelem. Z testů by se mělo zjistit, zda měl pozorovatel „dobrý“ zážitek během průchodu scénou a zda byla scéna celkově vizuálně příjemná (např. pozorovatel nebyl zbytečně rušen problikávajícími efekty).

Testování by mělo odhalit různé nedostatky prototypu, například v rychlosti posunu kamery, kdy příliš rychlá kamera může ve speciálních případech způsobit i pocity nevolnosti. Další problém, který se může při testování objevit je špatně definovaná úroveň detailu budov. Tento nedostatek může celkově snížit estetickou složku scény a pozorovatele to může v některých případech i úplně odradit. V druhé řadě je také potřeba ověřit, zda pozorovatel dokáže rozeznat drobné detaily objektů ve scéně i při použití různých optimalizačních technik.

10.1 Popis testování

Prototyp se bude testovat především na hromadných školních akcích. Testování proběhne ve stylu pozorování krátkých záznamů prototypu (videí), které obsahují předdefinované průchody scénou. Tento styl testování donutí pozorovatele se více zaměřit na samotnou městskou scénu a nemusí se tím pádem soustředit na ovládání prototypu pomocí klávesnice a myši.

Všichni účastníci testování (kromě moderátora) nejdřív vyplní vstupní dotazník. Podle výsledků z dotazníku se mohou zvolit určitá videa, která se budou na konkrétním testování pouštět. Poté bude pozorovatelům (testerům) postupně zobrazeno pár videí, s různými průchody městem a optimalizačními technikami prototypu. Videa se mohou spouštět několikrát za sebou (záleží na délce jednoho videa). Na opětovné spuštění stejného videa může moderátor upozornit. Ke konci všichni testeři vyplní výstupní dotazník.

10.2 Podkladové materiály

V této sekci jsou stručně popsány použité podkladové materiály pro otestování prototypu (kromě samotných záznamů z prototypu).

10.2.1 Dotazníky

Tester během celého testování prototypu vyplní dva dotazníky: vstupní a výstupní dotazník.

Vstupní dotazník slouží k získání základních informací o účastnících testování (testerech) a ke zjištění potenciální cílové skupiny jednotlivých testerů. Díky tomu lze testery jednodušeji identifikovat a v jiných případech je i možné testování určitým způsobem individuálně upravit (pokud například tester trpí epilepsií).

Vstupní dotazník obsahuje otázky:

1. Do jaké následující věkové kategorie spadáte?
2. Máte nějaké zkušenosti s virtuálními 3D scénami (mapy, hry, virtuální realita...)?
3. Máte nějaké oční problémy (krátkozrakost, dalekozrakost, tupozrakost, jiné)?
4. Trpíte epilepsií?

Výstupní dotazník zachycuje pocity a názory testera z pozorovaných záznamů (videí) prototypu. V dotazníku se může pozorovatel (tester) vyjádřit k následujícím vlastnostem scény:

- počet budov a jejich umístění v prototypu (zda byly budovy ve scéně vhodně umístěné a nebyly moc nahuštěné k sobě),
- detailnost budov,
- rychlost kamery a celkový pohyb ve scéně (především plynulost scény bez záseků a problikávání),
- velikost procházeného prostoru (zda scéna působila rozsáhle a nebo naopak byla vjemově malá),
- další připomínky a komentáře testera.

10.2.2 Testovací scénáře

Testovací scénáře se týkají různě optimalizovaných urbanistických scén v prototypu. Základní scény pro testovací scénáře jsou:

- Neoptimalizovaná scéna – scéna obsahuje budovy v nejvyšších detailech a neřeší neviditelnost zastíněných objektů.
- Scéna s LOD – budovy ve scéně respektují diskrétní a hierarchické LOD.
- Optimalizovaná scéna – ve scéně jsou definované úrovně detailů budov a navíc se odstraňují zastíněné části scény (*occlusion culling*).

Scénář 1 Základní průchod scénou. Cílem testovacího scénáře je odhalení problému v rychlosti kamery a problikávání. Podle průchodu lze také zjistit vnímání velikosti celé městské scény.

1. záznam: Kamera (pozorovatel) se ve scéně volně pohybuje a simuluje turistu, který má spoustu času. Kamera se pomalu otáčí do stran (pomalý pohled).
2. záznam: Kamera se ve scéně volně pohybuje ale prudce se otáčí do stran (rychlý pohled).

Scénář 2 Detailnost scény. Cílem scénáře je zjistit, zda jsou definované úrovně detailu jednotlivých objektů ve scéně dostačující.

1. záznam: Kamera se ve scéně pomalu přibližuje k jedné nebo více budovám.
2. záznam: Statický záznam budovy z určité dálky od kamery.

Závěr

V rámci této práce jsem popsala různé metody optimalizace rozsáhlých 3D virtuálních scén a zmínila jsem používaná kritéria pro klasifikaci těchto metod. V analytické části práce jsem také v obou kategoriích optimalizace (úrovně detailu a viditelnost) porovnala techniky mezi sebou. Přestože jsem se u metod úrovní detailu soustředila více na geometrii a na generování zjednodušených sítí modelů, věřím, že jsem poskytla užitečný přehled používání technik LOD.

U kategorie metod pro určování viditelnosti jsem se zaměřila na odstraňování zastíněných částí ve scéně, jelikož ostatní algoritmy (například eliminace odvrácených stěn) jsou postupně přesouvány do grafického procesoru. Téma určování viditelnosti v rozsáhlých scénách je stále aktuálním předmětem výzkumu v počítačové grafice, proto jsem v práci algoritmy popsala více přehledově, bez podrobnějších implementačních detailů.

Problematika optimalizace rozsáhlých scén je značně obsáhlá a pro budoucí studie doporučuji zaměřit se spíše na konkrétnější analýzu optimalizace textur namapované na modely či způsobu výpočtu osvětlení 3D scén.

Další důležitou částí mé práce byl návrh vizualizace pro projekt Virtuální historický průvodce. Návrh vycházel nejen z mé vlastní analýzy ale i z prací dalších členů týmu. Mým hlavním cílem v projektu bylo prozkoumání optimalizačních technik, které by mohlo ostatním členům do budoucna pomoci s návrhem a implementací celkové struktury projektu.

Přínosem analýzy a návrhu bylo též vytvoření jednoduchého prototypu, který demonstruje efektivitu několika způsobů optimalizace. Na základě použití nástrojů a skriptování herního jádra Unreal Engine 4 nakonec vznikla urbanistická scéna, ve které se lze pohybovat a zobrazovat si statistické tabulky během její vizualizace. Pomocí tohoto prototypu jsem změřila vybrané vizualizační parametry, jež poukazovaly na efektivitu použitých technik optimalizace. Výsledky porovnávání naznačují předpokládaný pozitivní vliv optimalizací na výkonnost zpracování celé scény.

Literatura

[1] Žára, J.; Beneš, B.; Sochor, J.; aj.: *Moderní počítačová grafika*. Brno: Computer Press, druhé vydání, 2008, ISBN 80-251-0454-0.

[2] Omernick, M.: *Creating the Art of the Game*. NGR (Series), New Riders, 2004, ISBN 9780735714090. Dostupné z: https://books.google.cz/books?id=JJ5RAAAAMAAJ

[3] Jong, S. D.: Hourences. [online], 2006, [cit. 7. 4. 2018]. Dostupné z: http://www.hourences.com/tutorials

[4] Unreal Developer Network (Epic Games): Unreal Development Kit/Unreal Engine 3: Performance, Profiling, and Optimization. [online], 2012, [cit. 8. 4. 2018]. Dostupné z: https://api.unrealengine.com/udk/Three/PerformanceHome.html

[5] Provost, G.: Beautiful, Yet Friendly Part 1: Stop Hitting the Bottleneck. [online], Jun 2003, [cit. 7. 4. 2018]. Dostupné z: http://www.ericchadwick.com/examples/provost/byf1.html

[6] Unreal Developer Network (Epic Games): Unreal Development Kit/Unreal Engine 3: Memory usage and Profiling. [online], 2012, [cit. 8. 4. 2018]. Dostupné z: https://api.unrealengine.com/udk/Three/MemoryProfilingHome.html

[7] Clark, J. H.: Hierarchical Geometric Models for Visible Surface Algorithms. *Commun. ACM*, ročník 19, č. 10, 08 1976: s. 547–554, ISSN 0001-0782, doi:10.1145/360349.360354. Dostupné z: http://doi.acm.org/10.1145/360349.360354

[8] Johansson, P.: *Perceptually Modulated Level of Detail in Real Time Graphics*. Dizertační práce, School of Computer Science and Communication (CSC), 2013. Dostupné z: http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A680183&dswid=4994

[9] Reddy, M.: *Perceptually modulated level of detail for virtual environments*. Dizertační práce, University of Edinburgh, 1997. Dostupné z: https://www.era.lib.ed.ac.uk/handle/1842/505

[10] Lindstrom, P.; Koller, D.; Hodges, L.; aj.: Level-of-Detail Management for Real-Time Rendering of Phototextured Terrain. Technická zpráva, Georgia Institute of Technology, 01 1995. Dostupné z: https://www.researchgate.net/publication/27521574_Level-of-Detail_Management_for_Real-Time_Rendering_of_Phototextured_Terrain

[11] Levoy, M.; Whitaker, R.: Gaze-directed Volume Rendering. In *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, I3D '90, New York, NY, USA: ACM, 1990, ISBN 0-89791-351-5, s. 217–223, doi:10.1145/91385.91449. Dostupné z: http://doi.acm.org/10.1145/91385.91449

[12] Funkhouser, T. A.; Séquin, C. H.; Teller, S. J.: Management of Large Amounts of Data in Interactive Building Walkthroughs. In *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, I3D '92, Cambridge, Massachusetts, USA: ACM, 1992, ISBN 0-89791-467-8, s. 11–20, doi:10.1145/147156.147158. Dostupné z: http://doi.acm.org/10.1145/147156.147158

[13] Holloway, R. L.: Viper: A Quasi-Real-Time Virtual-Environment Application. Technická zpráva, Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, 1992. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.1145

[14] Heok, T. K.; Daman, D.: A review on level of detail. In *Proceedings. International Conference on Computer Graphics, Imaging and Visualization, 2004. CGIV 2004.*, 07 2004, s. 70–75, doi:10.1109/CGIV.2004.1323963. Dostupné z: http://ieeexplore.ieee.org/document/1323963

[15] Luebke, D.: *Level of Detail for 3D Graphics*. Morgan Kaufmann series in computer graphics and geometric modeling, Morgan Kaufmann Publishers, 2003, ISBN 9781558608382. Dostupné z: https://books.google.cz/books?id=CB1N1aaoMloC

[16] Hoppe, H.: Progressive Meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, New York, NY, USA: ACM, 1996, ISBN 0-89791-746-4, s. 99–108, doi:10.1145/237170.237216. Dostupné z: http://doi.acm.org/10.1145/237170.237216

[17] ACM SIGGRAPH: *View-Dependent Refinement of Progressive Meshes*, 1997. Dostupné z: http://hhoppe.com/vdrpm.pdf

[18] Luebke, D. P.: A Developer's Survey of Polygonal Simplification Algorithms. *IEEE Computer Graphics and Applications*, ročník 21, 05 2001: s. 24–35, ISSN 0272-1716, doi:10.1109/38.920624. Dostupné z: http://doi.ieeecomputersociety.org/10.1109/38.920624

[19] Erikson, C.: Polygonal simplification: An overview. *Dept. of Computer Science, TR96-016*, 1996: s. 1–32. Dostupné z: https://www.semanticscholar.org/paper/Polygonal-Simplification%3A-An-Overview-Erikson/0ccbdcd6e51cf07204a1ffcaa7d6a0b2332a24b1

[20] Hinker, P.; Hansen, C.: Geometric Optimization. In *Proceedings of the 4th Conference on Visualization '93*, VIS '93, Washington, DC, USA: IEEE Computer Society, 1993, ISBN 0-8186-3940-7, s. 189–195. Dostupné z: http://dl.acm.org/citation.cfm?id=949845.949882

[21] Schroeder, W. J.; Zarge, J. A.; Lorensen, W. E.: Decimation of Triangle Meshes. *SIGGRAPH Comput. Graph.*, ročník 26, č. 2, Červenec 1992: s. 65–70, ISSN 0097-8930, doi:10.1145/142920.134010. Dostupné z: http://doi.acm.org/10.1145/142920.134010

[22] Sikand, S.: Voxel to Mesh Conversion: Marching Cube Algorithm. [online], 07 2017, [cit. 9. 4. 2018]. Dostupné z: https://medium.com/zeg-ai/voxel-to-mesh-conversion-marching-cube-algorithm-43dbb0801359

[23] The-Crankshaft Publishing: what-when-how: Mesh Processing (Advanced Methods in Computer Graphics) Part 3. [online], [cit. 8. 4. 2018]. Dostupné z: http://what-when-how.com/advanced-methods-in-computer-graphics/mesh-processing-advanced-methods-in-computer-graphics-part-3/

[24] Fisher, M.: Marching Cubes. [online], 2014, [cit. 9. 4. 2018]. Dostupné z: https://graphics.stanford.edu/~mdfisher/MarchingCubes.html

[25] Varshney, A.: *Hierarchical geometric approximations*. Dizertační práce, University of North Carolina at Chapel Hill, 1994. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.6121

[26] Cohen, J.; Varshney, A.; Manocha, D.; aj.: Simplification envelopes. In *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, New Orleans, LA, USA, 1996, s. 119–128. Dostupné z: https://uncch.pure.elsevier.com/en/publications/simplification-envelopes

[27] He, T.; Hong, L.; Kaufman, A.; aj.: Voxel Based Object Simplification. In *Proceedings of the 6th Conference on Visualization '95*, VIS '95, Washington, DC, USA: IEEE Computer Society, 1995, ISBN 0-8186-7187-4, s. 296–. Dostupné z: http://dl.acm.org/citation.cfm?id=832271.833850

[28] Hoppe, H.; DeRose, T.; Duchamp, T.; aj.: Mesh Optimization. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, New York, NY, USA: ACM, 1993, ISBN 0-89791-601-8, s. 19–26, doi:10.1145/166117.166119. Dostupné z: http://doi.acm.org/10.1145/166117.166119

[29] Melax, S.: A Simple, Fast, and Effective Polygon Reduction Algorithm. *Game Developer*, 11 1998: s. 44–49, [cit. 13. 4. 2018]. Dostupné z: www.melax.com/gdmag.pdf

[30] Turk, G.: Re-tiling Polygonal Surfaces. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, New York, NY, USA: ACM, 1992, ISBN 0-89791-479-1, s. 55–64, doi:10.1145/133994.134008. Dostupné z: http://doi.acm.org/10.1145/133994.134008

[31] Kalvin, A. D.; Taylor, R. H.: Superfaces: polygonal mesh simplification with bounded error. *IEEE Computer Graphics and Applications*, ročník 16, č. 3, May 1996: s. 64–77, ISSN 0272-1716, doi:10.1109/38.491187.

[32] Ben-Chen, M.; Lai Lin, A.: Mesh Simplification. [online], 2010, [cit. 10. 4. 2018]. Dostupné z: http://graphics.stanford.edu/courses/cs468-10-fall/LectureSlides/08_Simplification.pdf

[33] Cignoni, P.; Montani, C.; Scopigno, R.: A comparison of mesh simplification algorithms. *Computers & Graphics*, ročník 22, č. 1, 02 1998: s. 37–54. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0097849397000824

[34] Helman, J.; Bigos, A.; Tarbouriech, P.; aj.: Designing Real-Time 3D Graphics for Entertainment. In *SIGGRAPH 96*, 1996, s. 1–19. Dostupné z: https://www.semanticscholar.org/paper/Designing-Real-Time-3-D-Graphics-for-Entertainment-Helman-Bigos/967358e11cde083dc6daee6fc66aa9de92cf7478

[35] Bittner, J.; Wonka, P.; Wimmer, M.: Visibility preprocessing for urban scenes using line space subdivision. In *Proceedings Ninth Pacific Conference on Computer Graphics and Applications. Pacific Graphics 2001*, 2001, s. 276–284, doi:10.1109/PCCGA.2001.962883. Dostupné z: https://ieeexplore.ieee.org/document/962883

[36] Pantazopoulos, I.; Tzafestas, S.: Occlusion Culling Algorithms: A Comprehensive Survey. *Journal of Intelligent and Robotic Systems*, ročník 35, č. 2, Oct 2002: s. 123–156, ISSN 1573-0409, doi:10.1023/A:1021175220384. Dostupné z: https://doi.org/10.1023/A:1021175220384

[37] Power, K.: Backface culling. [online], 12 2012, [cit. 14. 4. 2018]. Dostupné z: http://glasnost.itcarlow.ie/~powerk/GeneralGraphicsNotes/HSR/backfaceculling.html

[38] Lighthouse3D: View Frustrum Culling. [online], 2011, [cit. 14. 4. 2018]. Dostupné z: http://www.lighthouse3d.com/tutorials/view-frustum-culling/

[39] Silicon graphics Inc.: OpenGL Optimizer 1.0. [online], 1997, [cit. 29. 4. 2018]. Dostupné z: http://titan.cs.ukzn.ac.za/opengl/opengl-d5/trant.sgi.com/opengl/docs/white_papers/optimizer_wp.html

[40] Kovalčík, V.: *Occlusion Culling in Dynamic Scenes*. Dizertační práce, Masarykova univerzita, Brno, 2007, [cit. 14. 4. 2018]. Dostupné z: https://is.muni.cz/th/4269/fi_d/thesis.pdf

[41] Haines, E.; Möller, T.: Occlusion Culling Algorithms. [online], 11 1999, [cit. 19. 4. 2018]. Dostupné z: https://www.gamasutra.com/view/feature/131801/occlusion_culling_algorithms.php

[42] Greene, N.; Kass, M.; Miller, G.: Hierarchical Z-buffer Visibility. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, New York, NY, USA: ACM, 1993, ISBN 0-89791-601-8, s. 231–238, doi:10.1145/166117.166147. Dostupné z: http://doi.acm.org/10.1145/166117.166147

[43] Germs, R.; Jansen, F. W.: Geometric Simplification for Efficient Occlusion Culling in Urban Scenes. In *WSCG '2001: Conference proceedings: The 9-th International Conference in Central Europe on Computer Graphics*, WSCG '2001: Conference proceedings, University of West Bohemia, 2001, ISBN 80-7082-713-0, ISSN 1213-6972, s. 291–298. Dostupné z: https://dspace5.zcu.cz/handle/11025/11281

[44] Greene, N.; Wilhelms, J.; Pang, A.; aj.: *Hierarchical rendering of complex environments*. Dizertační práce, University of California, Santa Cruz, 1995. Dostupné z: https://www.soe.ucsc.edu/sites/default/files/technical-reports/UCSC-CRL-95-27.pdf

[45] Zhang, H.; Manocha, D.; Hudson, T.; aj.: Visibility Culling Using Hierarchical Occlusion Maps. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, ISBN 0-89791-896-7, s. 77–88, doi:10.1145/258734.258781. Dostupné z: http://dx.doi.org/10.1145/258734.258781

[46] Bittner, J.; Havran, V.; Slavik, P.: Hierarchical Visibility Culling with Occlusion Trees. In *Proceedings of the Computer Graphics International 1998*, CGI '98, Washington, DC, USA: IEEE Computer Society,

1998, ISBN 0-8186-8445-3, s. 207–. Dostupné z: http://dl.acm.org/citation.cfm?id=792757.792949

[47] Luebke, D.; Georges, C.: Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, I3D '95, New York, NY, USA: ACM, 1995, ISBN 0-89791-736-7, s. 105–ff., doi:10.1145/199404.199422. Dostupné z: http://doi.acm.org/10.1145/199404.199422

[48] Downs, L.; Möller, T.; Séquin, C. H.: Occlusion Horizons for Driving Through Urban Scenery. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, I3D '01, New York, NY, USA: ACM, 2001, ISBN 1-58113-292-1, s. 121–124, doi:10.1145/364338.364378. Dostupné z: http://doi.acm.org/10.1145/364338.364378

[49] Lloyd, B.; Egbert, P.: Horizon occlusion culling for real-time rendering of hierarchical terrains. In *IEEE Visualization, 2002. VIS 2002.*, Nov 2002, s. 403–409, doi:10.1109/VISUAL.2002.1183801. Dostupné z: https://ieeexplore.ieee.org/document/1183801/

[50] Wonka, P.; Schmalstieg, D.: Occluder Shadows for Fast Walkthroughs of Urban Environments. *Computer Graphics Forum*, ročník 18, č. 3, 1999: s. 51–60, doi:10.1111/1467-8659.00327. Dostupné z: https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.00327

[51] Johnson, A.: Culling explained. [online], 06 2013, [cit. 22. 4. 2018]. Dostupné z: http://docs.cryengine.com/display/SDKDOC4/Culling+Explained

[52] Hobson, T.: Precomputed Visibility Volume. [online], 2016, [cit. 22. 4. 2018]. Dostupné z: http://timhobsonue4.snappages.com/culling-precomputed-visibility-volumes

[53] Bittner, J.: *Hierarchical Techniques for Visibility Computations*. Dizertační práce, České Vysoké Učení Technické v Praze, 2002. Dostupné z: http://dcgi.fel.cvut.cz/home/bittner/diss.html

[54] Tseng, D.-C.; Huang, C.-C.: Dynamic View-Dependent Multiresolution Terrain Visualization. In *2006 IEEE International Conference on Multimedia and Expo*, July 2006, ISSN 1945-7871, s. 193–196, doi:10.1109/ICME.2006.262415. Dostupné z: https://ieeexplore.ieee.org/document/4036569

[55] Pajarola, R.: Large scale terrain visualization using the restricted quadtree triangulation. In *Visualization '98. Proceedings*, Oct 1998, ISSN 1070-2385, s. 19–26, doi:10.1109/VISUAL.1998.745280. Dostupné z: https://ieeexplore.ieee.org/document/745280

[56] Gaede, V.; Günther, O.: Multidimensional Access Methods. *ACM Comput. Surv.*, ročník 30, č. 2, Červen 1998: s. 170–231, ISSN 0360-0300, doi:10.1145/280277.280279. Dostupné z: http://doi.acm.org/10.1145/280277.280279

[57] Pina, J.; Serón, F.; Cerezo, E.: Accelerating Urban Scenes Visualization Using a Quadtree Decomposition of Urban Blocks. *WSCG '2008: Communication Papers: The 16-th International Conference in Central Europe on Computer Graphics*, 2008. Dostupné z: https://dspace5.zcu.cz/handle/11025/11109

[58] Zlatanova, S.: *3D GIS for urban development*. Dizertační práce, ITC – University of Twente's Faculty of Geo-Information Science and Earth Observation, 2000, [cit. 27. 4. 2018]. Dostupné z: https://3d.bk.tudelft.nl/szlatanova/PhDthesis/pdf/ch7.pdf

[59] Li, X.; Wan, W.; Wang, R.; aj.: A dynamic loading mechanism for large-scale 3D visualization. In *IET International Communication Conference on Wireless Mobile and Computing (CCWMC 2011)*, Nov 2011, s. 89–92, doi:10.1049/cp.2011.0853. Dostupné z: https://ieeexplore.ieee.org/document/6194810

[60] Teller, S. J.; Séquin, C. H.: Visibility Preprocessing for Interactive Walkthroughs. *SIGGRAPH Comput. Graph.*, ročník 25, č. 4, Červenec 1991: s. 61–70, ISSN 0097-8930, doi:10.1145/127719.122725. Dostupné z: http://doi.acm.org/10.1145/127719.122725

[61] Blender Foundation: Blender 2.78. [software], 1995, [přístup 1. 3. 2017]. Dostupné z: https://www.blender.org

[62] Epic Games: Unreal Engine 4, 4.17.2. [software], 1998, [přístup 2012]. Dostupné z: https://www.unrealengine.com

[63] Microsoft: Visual studio 2015, 14.0. [software], 1997, [přístup 20. 7. 2015]. Dostupné z: https://www.visualstudio.com

[64] Sparx Systems: Enterprise Architect, 11.0. [software], 2000, [přístup 2014]. Dostupné z: http://sparxsystems.com

Příloha **A**

Seznam použitých zkratek

AO Ambient occlusion

BSP Binary space partitioning

CAD Computer-aided design

CPU Central processing unit

DVD Digital versatile/video disc

EA Enterprise Architect

FBX Filmbox (formát souboru)

FPS Frames per second

GPU Graphic processing unit

Hi-Z Hierarchical z-buffer

HLOD Hierarchical level of detail

HZB Hierarchical z-buffer

IDE Integrated development environment

LOD Level of detail

OBJ Object file (formát souboru)

PVS Potentially visible set

RAM Random access memory

SQL Structured query language

SSD Solid-state drive

SSR Screen space reflections

UDK Unreal development kit

UE Unreal Engine

UE4 Unreal Engine 4

UML Unified modeling language

VHP projekt Virtuální historický průvodce

VTK Visualization Toolkit

Příloha **B**

Obsah přiloženého DVD

- `readme.txt` ..... stručný popis obsahu DVD
- `bin` ..... obsahuje spustitelný `exe` soubor pro platformu Windows
- `src`
  - `impl` ..... zdrojové kódy
  - `proj` ..... projekt Unreal Engine
- `text`
  - `pdf` ..... text bakalářské práce ve formátu PDF
  - `latex` ..... zdrojová forma bakalářské práce ve formátu LaTeX