VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

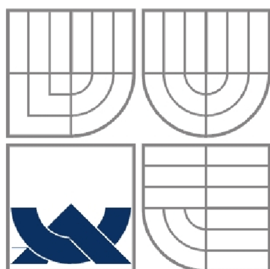
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

VÝPOČET VIDITELNOSTI V 3D BLUDIŠTI

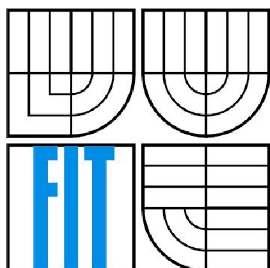
DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE BC. JIŘÍ PETRUŽELKA
AUTHOR

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

VÝPOČET VIDITELNOSTI V 3D BLUDIŠTI

VISIBILITY DETERMINATION IN 3D MAZE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — BC. JIŘÍ PETRUŽELKA
AUTHOR

VEDOUCÍ PRÁCE — ING. LUKÁŠ POLOK
SUPERVISOR

BRNO 2014

Abstrakt

Cílem práce je prezentace metod pro určování viditelnosti, návrh a implementace aplikace, prezentující výpočet viditelnosti ve 3D bludišti.

Abstract

The purpose of this thesis is to present methods for visibility determination and to design and implement an application to demonstrate visibility determination in a 3D maze.

Klíčová slova

viditelnost, portály, binární rozdělování prostoru, potenciální sada viditelnosti, occlusion query, OpenGL

Keywords

visibility, portal culling, binary space partitioning, potentially visible sets, occlusion query, OpenGL

Citace

Petruželka Jiří: Výpočet viditelnosti v 3D bludišti, diplomová práce, Brno, FIT VUT v Brně, 2014

Výpočet viditelnosti v 3D bludišti

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Lukáše Poloka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Jiří Petruželka
27.května. 2014

Poděkování

Rád bych poděkoval vedoucímu práce panu Ing. Lukáši Polokovi za poskytnuté informace a rady a své rodině za všechny formy podpory.

© Jiří Petruželka, 2014.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod ... 3
2 OpenGL ... 4
2.1 Vykreslovací řetězec a viditelnost ... 4
2.2 Occlusion Query ... 5
2.2.1 Conditional Rendering ... 5
2.3 Immediate a retained módy ... 5
3 Viditelnost v počítačové grafice ... 7
3.1 Potenciální sada viditelnosti ... 7
3.2 Viewing frustrum culling ... 8
3.3 Backface culling ... 9
3.4 Z-culling ... 9
3.5 Occlusion culling ... 10
3.6 Portal culling ... 11
3.7 Zjednodušování scény ... 12
4 Dělení scény a pomocné objekty ... 13
4.1 Bounding volume (hierarchy) ... 13
4.2 BSP strom ... 15
4.3 Octree ... 16
5 Návrh ... 18
5.1 Hierarchizace scény a stanovení buněk ... 18
5.2 Generování portálů a přiřazení portálů ... 19
5.3 Částečné stanovení PVS ... 20
5.4 Detekce viditelných portálů ... 20
5.5 Zaměření a hodnocení efektivity ... 22
6 Implementace ... 23
6.1 Návrh programu ... 23
6.1.1 Popis a konfigurace ... 23
6.1.2 Zdrojová data ... 24
6.1.3 Datový návrh ... 24
6.1.4 Třídy ... 25
6.2 Dělení scény ... 28
6.3 Generování portálů ... 32
6.3.1 Ořezávání polygonů ... 34
6.4 Běh a vykreslování ... 41

6.4.1 Příprava dat ... 41
6.4.2 Hlavní fáze ... 41
6.5 Detekce viditelnosti ... 42
6.6 Uživatelské rozhraní ... 44
7 Vyhodnocení ... 47
8 Závěr ... 48
Manuál instalace ... 51
1. Hlavní program ... 51
2. Uživatelské rozhraní ... 51
Příloha B ... 52
Manuál konfigurace a ovládání ... 52
1. Hlavní program ... 52
2. Uživatelské rozhraní ... 53
Příloha C ... 55
Obsah CD ... 55
Příloha D ... 56
Popis přiložených scén a konfigurací ... 56
Scény ... 56
Statické objekty ... 56
Trajektorie ... 56
Příloha E ... 57
Vývojová a testovací prostředí a software ... 57
Použitý software ... 57
Příloha F ... 58
Poznámky ke zdrojovým datům ... 58

1 Úvod

Cílem této práce je popsat základní přístupy stanovování viditelnosti v počítačové grafice. Dále navrhnout, vytvořit a zhodnotit algoritmus pro řešení viditelnosti v bludišti a jiných interierních prostorách prezentovaný formou programu.

Při vykreslování scén počítačem je třeba rozvrhnout dostupné (a značně konečné) výpočetní prostředky na jednotlivé úkony. Pokud je scéna součástí komplexnějšího programu, musí se dále dělit o zdroje i s dalšími prvky tohoto systému. Na jedné straně je poptávka po zobrazování komplexních a rozsáhlých scén, proti tomu jde požadavek na co nejvyšší výkon, plynulost výsledku a nejlépe i uspořenou paměť. Jako projev efektivity řešení můžeme zahrnout i zpracování pouze těch částí scény, které může uživatel v daném okamžiku vidět. Bude tedy snaha co nejrychleji rozlišit viditelnou část scény v daném okamžiku a pouze tu zadat ke zpracování a zobrazení grafické kartě.

Práce se zaměřuje na scény s vysokou mírou zastínění jednotlivých částí, tato charakteristika koresponduje s vlastnostmi např. interiérů budov, letadel, strojů, husté městské zástavby a je význačnou vlastností bludišť. Počítá se s možností rozdělení scény na konvexní prostorové části, spojené průhlednou plochou, průchodem či oknem.

Výsledkem budou algoritmy demonstrované aplikací, prezentující procházení trojrozměrnou scénou, a jejich srovnání. Aplikace bude psána v jazyce C++ a bude využívat rozhraní OpenGL. V závěru práce se porovná a zhodnotí efektivita algoritmu a jeho nastavení ve scénách a nastíní se další možné směry vývoje.

2 OpenGL

OpenGL je multiplatformní, jazykově nezávislé nízkoúrovňové programové rozhraní ke grafickému hardwaru [9]. Bylo vydáno v roce 1992 firmou Silicone Graphics Inc. [15], nyní spravované konsorciem Khronos Group. Uplatňuje se v oblastech počítačem podporovaného designu, vizualizacích v lékařství a vědě, zábavním a herním průmyslu, simulacích a virtuální realitě. Je zpětně kompatibilní a i řadu let staré aplikace mají stále zaručený korektní běh. Na druhé straně je možností používat rozšíření oficiálně uznaná i jiná, především od výrobců hardwaru a grafických karet. [17]

Sestává se ze stovek funkcí, sloužících k definici objektů a řízení generování obrázků do obrazové paměti, zejména barevných obrázků trojrozměrných scén. Je možné implementovat OpenGL téměř kompletně softwarově, v praxi je však značná část akcelerována hardwarem. V OpenGL hraje důležitou roli stav, k němuž se uchovává velké množství dat. Mnoho příkazů tak slouží pouze ke změně stavu, a tím modifikují či specifikují následující příkazy. [9] Následující odstavce více osvětlují motivaci a popisují funkcionality, relevantní dále v návrhu.

2.1 Vykreslovací řetězec a viditelnost

OpenGL využívá vykreslovací řetězec (rendering pipeline), sérii procesů, jimiž zpracovává poskytnutá data a lze je ovlivňovat příkazy. Chování některých částí je pak možno přímo řídit pomocí programů zvaných „shader“. Programátor pomocí příkazů definuje data, která jsou předána řetězci ke zpracování.

V první fázi jsou zpracovány jednotlivé vertexy, aplikován vertex shader, volitelně tessellace a geometry shader. V následující fázi jsou vertexy upraveny, a mimo pohledové transformace také proběhne ořezání pohledových tělesem (viewing frustrum). Z vertexů jsou sestavena geometrická primitiva a ta jsou rasterizována, výsledkem jsou fragmenty. Před jejich zápisem do framebufferu je na nich vykonána řada testů, ze kterých je pro naše téma důležitý především test hloubky (depth test), který zkoumá pořadí a překryvy fragmentů, aby byly viditelné pouze nezakryté plochy. [9]

Pipeline tedy obsahuje řadu procesů, jak odstranit zakryté části scény. Proč je tedy třeba řešit viditelnost z pozice programátora? Veškerá tato ořezání probíhají až v pokročilých částech vykreslovacího řetězce, depth test je až jeden z posledních úkonů. Zbytečně tedy byl spotřebován čas a zbytečně zabrána paměť grafické karty částmi scény, které byly následně vyřazeny. Tato práce řeší, jak vyřadit co největší možné množství dat ještě před vložením do vykreslovacího řetězce. Tento úkon má pochopitelně také svůj výkonnostní dopad, je tedy nutno najít rovnováhu a stanovit situace, kdy bude navrhovaný algoritmus výhodné použít.

2.2 Occlusion Query

Occlusion queries jsou funkcionalita umožňující se dotázat, zda byly vykresleny (a tedy jsou viditelné) fragmenty určitého objektu. Lze také volitelně získat přesný počet fragmentů. Jedná se o rasterizaci (a tudíž metodu pracující v prostoru obrazu), která je podporována hardwarem. [8] Vykreslené fragmenty jsou stanoveny pouze pomocí depth testu, ostatní testy nejsou brány v potaz [16]. Výpočet je možno provádět i při vypnutí funkcionalit, irelevantních z hlediska viditelnosti, typicky je při použití occlusion queries vypnut zápis do frame a depth bufferu. S výhodou lze také testovat pouze obalující objekt, bounding volume, cílového objektu. Je ovšem nutno počítat, že obálka může zabírat výrazně více pixelů, než původní objekt. Jelikož occlusion queries pracují s rasterizací, záleží doba výpočtu na zabrané ploše. Samotný dotaz také představuje určitou režii, kterou je nutno brát v potaz.

Dotaz se zpracovává asynchronně a nemusí být okamžitě k dispozici. Při použití za běhu výsledné aplikace by čekání na výsledky mohlo způsobit zpomalení, je proto třeba použít prostředky, které toto omezení zmírní. Jedná se například o prostorovou a časovou lokalitu, dělení scény a využití předchozích výsledků. [8]

2.2.1 Conditional Rendering

Díky Conditional rendering lze podmínit vykonání určitých vykreslovacích příkazů výsledky stanovených dotazů (queries). Programátor aktivuje podmiňované vykreslování, stanoví dotaz, ke kterému se váže, a nastaví způsob získání dotazu. Příkaz uzavírající podmíněné vykreslení pak vyznačí pořadavky mezi úvodním a uzavíracím příkazem jako podmíněné. Dle způsobu získání dotazu pak vykreslování buď čeká na výsledek dotazu (viz asynchronnost dotazů v occlusion query), vyhodnotí dotaz okamžitě (může tedy vykreslit nehledě na výsledek, pokud není k dispozici), nebo způsoby shodnými s předchozími s tím, že se vykreslí pouze region, označená výsledkem dotazu jako viditelná (tj. nebyly viditelné v případě occlusion query). [9][16]

Kombinace occlusion query a conditional rendering je jedním z možných způsobů řešení viditelnosti. Jsou výhodné pro rozsáhlejší scény s velkým množstvím zakrytých objektů a (narozdíl od některých jiných přístupů) je použitelný i pro dynamické scény [8]. Je možné ho také kombinovat s jinými principy a mechanismy, což je prvkem návrhové části této práce.
Podmíněné renderování (kromě omezení regionu) je možno emulovat programově, což má výhodu flexibility (lze použít starší query nebo výsledek podmínit ještě jiným způsobem).

2.3 Immediate a retained módy

Tyto módy popisují rozdílné způsoby, jakými program zadává data grafické kartě.

U „immediate“ módu se pro každý snímek zvlášť předají veškerá data. Jsou volány funkce rozhraní, specifikující pozice, barvy, normály a texturovací souřadnice jednotlivých vertexů, pro každý jednotlivý vertex scény. Pokud nenastala žádná změna v zobrazovaných datech, program i přesto musí opětovně odeslat všechny údaje. Pro malý počet vertexů nemusí tento fakt hrát roli, v opačném případě se graduálně začíná projevovat řada nevýhod. Je zaměstnán procesor, který nemůže vykonávat jinou činnost a musí odesílat data. Grafická karta musí zpracovávat příjem dat a čekat na označení konce odesílaných dat (glEnd()).

U „retained“ módu nastal posun od imperativního k deklarativnímu přístupu. Místo zadávání dat jako posloupnosti příkazů jsou požadovaná data jako celek odeslána do paměti grafické karty. K tomuto se v OpenGL používá objekt Buffer, který zapouzdřuje pole dat. Tento úkon je proveden jednou, tedy po dobu, dokud není scéna změněna. Pro každý snímek pak stačí programu označit:

- ze kterých bufferů se má renderovat
- jaký typ dat obsahují
- jaký typ objektů specifikují (úsečky, trojúhelníky...)
- kolik těchto objektů se má vykreslit

Snímek je tak možno zadat k renderování pomocí jednotek příkazů. Pokud je scéna změněna, není třeba znovu nahrát celou scénu, postačí aktualizovat konkrétní úseky paměti a při vykreslení na toto bude brán ohled.

I tak lze odtušit, že je zbytečné nahrát do grafické karty celou scénu. Při dostatečně velkém množství dat a malé paměti karty dokonce nemožné. Určování viditelnosti umožňuje výrazně omezit velikost dat, nutných k přenosu na kartu, předat jen ty prvky scény, které jsou vidět.

3 Viditelnost v počítačové grafice

Oblast viditelnosti v počítačové grafice se zabývá rozlišením viditelných a neviditelných objektů v určitém čase a směru z určitého bodu scény. Objekty mohou být mimo aktuální pohled, zakryty jinými objekty, případně zcela průhledné. Poslední případ je mimo zaměření práce.

Algoritmy této oblasti se dělí podle reprezentace prostorových objektů, většina pracuje s ploškovou reprezentací. Podle tvaru výsledných dat se dělí na:

- Vektorové algoritmy – výstupem je soubor geometrických prvků, nezávislých na rozlišení. Dělí se dále na algoritmy vracející úsečky, anebo plochy. I menší odchylka může mít značný vliv na celý výsledek.
- Rastrové algoritmy – výsledkem je obraz vyplněný pixely s barvami dle viditelných ploch.

Další dělení je z hlediska prostoru, v jakém se odehrává, se zaměřením na rychlost zpracování:

- Prostor objektů – porovnání poloh a rozměrů objektů mezi sebou. Může být výpočetně velmi náročné (každý objekt s každým v triviálním případě), ovšem lze urychlit řadou technik, například seskupením do obálek nebo hierarizací, jako je BSP strom. Složitost bývá kvadratická, potenciálně linearitmická.
- Prostor obrazu – pro každý pixel obrazu se detekuje objekt, který se na něj promítne. Složitost je lineárně úměrná počtu pixelů.

Dělení není exkluzivní, existují algoritmy, které kombinují přístupy více kategorií. Pro zvýšení efektivity je možno data předzpracovat, zaměnit, seskupin, či hierarchizovat. [4]
Odstraňovaní elementů, označených za neviditelné, se nazývá (visibility) culling. Tyto metody jsou spojeny s určitým principem či premisou a typicky se jich v procesu vykreslování uplatní více. Ať už explicitně jako samostatný krok stanovení viditelné sady, nebo jako princip v podobě součásti jiného algoritmu (například kombinace viewing frustrum culling a portal culling, kdy ignorujeme portály mimo pohledové těleso).

3.1 Potenciální sada viditelnosti

Potenciální sada viditelnosti (potentially visible set, PVS) je sada prvků scény, které mohou (ale ne nezbytně musí) být z určitého bodu či regionu v daný moment viditelné. Termín poprvé použil Airey v roce 1990 [10][6]. PVS lze vyhodnotit buď před zobrazováním scény (před během aplikace), nebo až za běhu.

Výpočet za běhu má výhodu v možnosti zpracování dynamických objektů, na druhé straně je zvýšeno množství vykonávaných výpočtů.
Při předpočítání není běžně pozice pozorovatele známa, je však možno provést výpočet pro celou oblast (např. buňku). Alternativně lze potom vypočítat viditelnost v množství vzorových pozic a při

výpočtu pak vybírat nejbližší pozici a data z ní. Datová náročnost pak může být ještě vyšší a navíc půjde (bez dalších úprav) o metodu pouze aproximační, ne konzervativní. Další nevýhodou je samotný výpočet, především v případech, kdy je buď dostupný výkon více než dostatečný pro průběžný výpočet; dále v případech kdy očekáváme, že v samotném běhu bude shlédnuta pouze malá část scény a většina výpočtu tak bude provedena zbytečně. Zásadní výhodou je pak existence výsledku již na začátku výpočtu scény, namísto potenciálně zdlouhavého počítání hodnotu jednoduše vybereme z dostupných dat.

Potenciální sady viditelnosti se dělí dle typu výsledku (toto dělení lze použít i obecně na algoritmy viditelnosti) [11]:

- Konzervativní – konzistentně vrací nadmnožinu skutečně viditelných elementů. Je třeba uchovat a následně zpracovat více dat, všechna viditelná data jsou ovšem správně označena. Nadbytečná data pak mohou být redukována aplikací některé z eliminačních technik.
- Agresivní – konzistentně vrací podmnožinu viditelných elementů a výsledné zobrazení tak je nepřesné. Je vhodné pro případy s důrazem na maximální rychlost výpočtu, také v případě velmi detailních, dynamických scén, nebo při značném pohybu pozorovatele, kdy uživatel není dostatečně schopen nepřesnosti vnímat.
- Přibližné (aproximační) – nelze konzistentně stanovit relace k množině skutečně viditelných prvků scény, výsledek může být podmnožina i nadmnožina. Opět vhodné pro případy s důrazem na rychlost výpočtu.
- Přesné (exaktní) – vypočítají vždy přesnou množinu viditelných prvků. Tento typ může být výpočetně náročný, pro větší scény může trvat mnoho hodin [11].

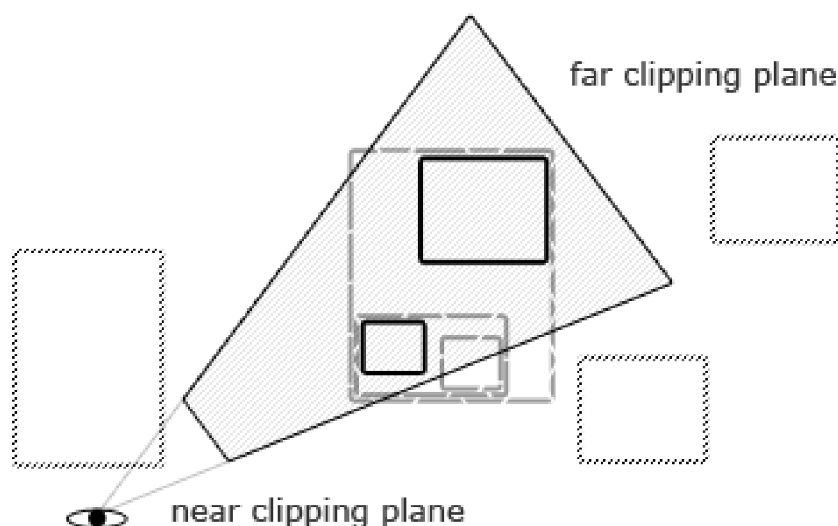
Co přesně je elementem uvažovaným v PVS za prvek výsledku záleží na konkrétní variantě, potenciálně viditelná sada je obecný termín, je tak možno identifikovat polygony, objekty, skupiny objektů, nebo např. buňky (v případě portálů, viz dále).

3.2 Viewing frustrum culling

Scéna je renderována z pohledu pozorovatele určitým směrem. Ve většině případů je alespoň část scény zcela mimo tuto oblast pohledu a tudíž nemůže být vidět, a to ani při ignorování možnosti zastínění nebo průhlednosti.

Takto viditelný prostor lze vyznačit objektem, nazývaným pohledové těleso, viewing frustrum. Jedná se o čtyřboký komolý jehlan vzniklý s pomyslným vrcholem v bodě pozorovatele. Boční strany označují hranice rozhledu do stran; vrchní a spodní strany vymezují prostor, který je příliš blízko nebo daleko k pozorování (tzv. near a far clipping plane). Viewing frustrum culling odstraňuje plochy a objekty scény, které jsou mimo toto těleso. [4]

Efektivitu lze zvýšit např. seskupením do obalových těles nebo využitím časové koherence [4], viz obrázek 1. Tento algoritmus je možno předřadit před jiné (např. occlusion culling) a snížit tak počet operací a porovnání v jejich procesech.



Obrázek 1. Viewing frustrum culling a bounding volume (hierarchy). Tečkovaně neviditelné obálky, čárkovaně částečně viditelné, plnou čarou viditelné obálky.

3.3 Backface culling

Tento princip využívá orientace ploch u ploškové reprezentace a pracuje s premisou, že plochy odvrácené od pozorovatele nejsou viditelné. U koule je takto možno odstranit až polovinu plošek, v praxi efektivita závisí na druhu objektů, u interiérových scén v případě stěn bude odstraněných ploch značně méně [1].

Charakteristika takové plošky je, že její normála svírá s vektorem pohledu ostrý úhel. Skalárním součinem lze triviálně stanovit orientaci plošky přivrácená/odvrácená. [4] Podmínkou je, že kamera je vně objektu a nezasahuje do něj. Backface culling lze také provádět po skupinách (tzv. clustered backface culling), kde jsou sdruženy plošky sdílející stejné nebo dostatečně podobné směry normál, do určité míry odchýlení. To je implementováno pomocí normálového kuželu (normal cone) se stanoveným poloúhlem α. [1]

V OpenGL se backface culling activuje jednoduše voláním příkazu glCullFace s příslušným nastavením. Pro korektní výpočet je třeba dbát na definici vertexů primitiva ve správném pořadí. [16]

3.4 Z-culling

Jedná se o rastrovou metodu v prostoru obrazu. Z-culling je klasický a rychlý algoritmus pro odstraňování zakrytých pixelů, dostupný na většině grafických procesorů.

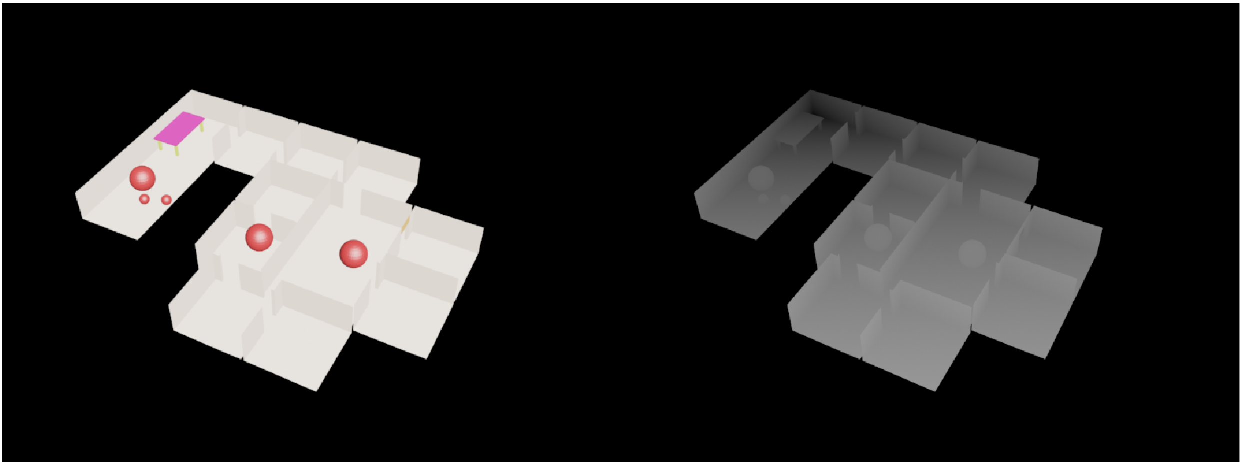
Metoda používá paměť hloubky (z-buffer / depth-buffer), což je dvourozměrné pole velikosti výsledného obrazu, uchovávající souřadnici z nejbližšího bodu, který je na daný pixel obrazu promítnut. Základní algoritmus má pak jednoduchý princip – pro každý pixel obrazu nalezneme

plochy, které se na něj promítají, stanoví se jejich vzdálenosti a z té nejbližší se určí výsledná barva a hodnota v depth buffer.

Každá plocha je zpracována pouze jednou a tudíž je složitost metody lineární. Vylepšením je hierarchizace depth bufferu. Ten je rozdělený na stupně (z-pyramida), každý s polovičním rozlišením oproti předchozího (až po 2x2). Scéna je také hierarchizována (obvykle do octree). Zapisované z-depth hodnoty se propagují do vyšších (méně detailních) stupňů z-pyramidy. V algoritmu se poté porovnává obálka octree se z-pyramidou postupně k nejvyššímu rozlišení, pokud je shledána viditelná, postupuje se recurzivně v octree dále podobným principem. [1] Pro zlepšení efektivity lze využít temporální koherence. Pokud však není hierarchická paměť hloubky implementována na grafickém procesoru, není výrazně efektivnější. Z-culling jde také s výhodou paralelizovat rozdělením na regiony. [4]

Je standardní součástí vykreslovacího řetězce a lze jej aktivovat příkazem `glEnable` s parametrem `GL_DEPTH_TEST`. [16]

Nachází se ovšem až u konce vykreslovacího řetězce, což omezuje jeho vliv na rychlost. Také průhledné objekty nezapřičiňují eliminaci vzdálenějších pixelů a je třeba je zpracovávat ve správném pořadí.



Obrázek 2. Nalevo vykreslená scéna a napravo zkonstruovaná paměť hloubky.

3.5 Occlusion culling

Podobně jako z-culling řeší zakrytí elementů jinými elementy, zde se však jedná o vektorový algoritmus, v jakém prostoru se odehrává pak záleží na jeho konkrétní variantě – obrazová porovnává dvourozměrně po projekci, objektová porovnává polohy a rozměry objektů ve scéně. Existuje také hybridní paprsková varianta, konvertující zájmové body na paprsky pro jednodušší porovnávání. [1] Efektivita také záleží na pořadí objektů, pokud jsou objekty seřazeny od nejbližších ke vzdáleným, lze jich mnohem více vyloučit a dále neuvažovat [1]. V tom může pomoci již zmíněná hierarchizace.

Výhodou occlusion culling je, že ho lze provést dříve, než jsou zastíněné objekty vloženy do vykreslovacího řetězce a tak je zcela vyloučit ze všeho zpracovávání. Opět lze využít hierarchické shlukování objektů do skupin (a vyloučit jedním testem celé skupiny objektů), u obrazových variant pak hierarchizací podobnou z-pyramidě popsané v části z-culling.

Occlusion culling není standardní metodou přítomnou v OpenGL, implementace je ponechána na programátorovi.

3.6 Portal culling

Portal culling je pro tuto práci zásadní technika určování viditelnosti. Obsahuje řadu principů, zmíněných v předchozích odstavcích a poprvé ji popsal pan John Airey roku 1990 [10]. Primárním zaměřením této metody jsou interiérové scény, kdy stěny mezi jednotlivými místnostmi stíní prostor za nimi (podobně jako u occlusion culling). Jednotlivé místnosti (buňky, cells) jsou propojeny portály, kterými v reálu mohou být průchody či okna. Portálová struktura může být pochopitelně vytvořena obecně na jakémkoli prostoru, který je možno rozdělit na mnohostěnné prostory, kde některé z nich sdílejí jednu či více průhledných ploch. Portál je tedy region na rozhraní buňky, který tuto buňku propojuje se sousední [7], přičemž standardně netvoří celou projektovanou plochu mezi buňkami a tak vždy alespoň část sousedících buněk je zastíněna.

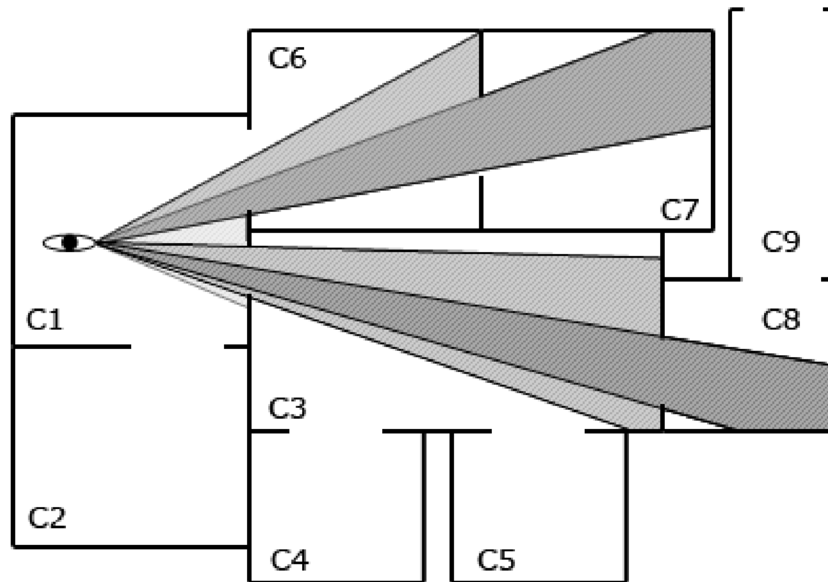
Uvažuje se tedy určitá forma předzpracování scény, dělení na buňky, následné generování portálů a uložení vazeb mezi nimi [1]. Není však principielně vyloučen předpoklad manuálního dělení scény a ručního definování portálů. Čas vývojářů a architektů je však něco, co se snažíme minimalizovat, a vyloženě prakticky vyloučeno je to v případě velkých scén, např. vizualizace průchodu mrakodrapem.

Na této struktuře je následně stanovována potenciální sada viditelnosti, ať už při předpočítání (Airey, Teller, Ranta-Eskola aj.) či za běhu aplikace (Jones, Lubke&Georges). Buňka je potenciálně viditelná, pokud sdílí portál, ve kterém se nachází pozorovatel. Buňka je pak skutečně alespoň částečně viditelná, pokud je alespoň jakákoli část portálu viditelná z pozice pozorovatele. Zde se uplatní princip podobný frustum culling, kdy původní viewing frustrum pozorovatele je postupně ořezáváno a děleno procházejícími portály, až je nakonec zcela ukončeno v určitých buňkách. Také se mohou lišit metody podle přístupu k statickým a dynamickým objektům, zda tyto mohou zakrývat portály a jak je řešeno stanovení jejich viditelnosti v následujících buňkách.

V procesu renderování se pak často aplikuje rekurzivní přístup, kdy se v první buňce (tj. buňce s pozorovatelem) stanoví viditelnost portálů a pokračuje se v buňkách asociovaných s viditelnými portály stejným způsobem dále, dokud je množina viditelných portálů v buňce nenulová. Výpočet končí po ukončení výpočtu ve všech větvích. [5]

Portal Culling dobře spolupracuje s jinými principy, jako je například dělení scény (BSP, Octree aj.), kde dělení scény může být použito pro stanovení buněk a hierarchizace umožní vyhledání buňky v pouze logaritmickém čase.

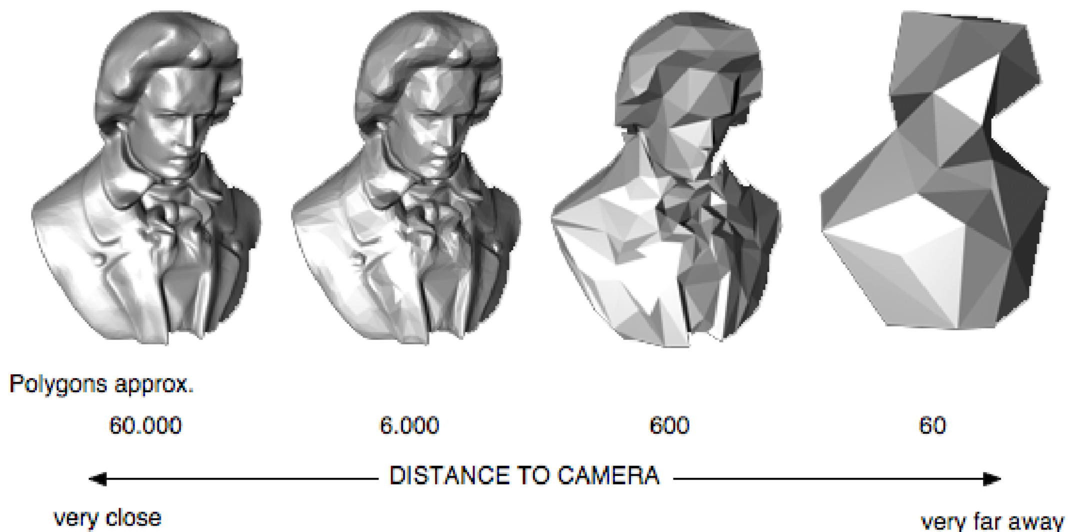
Mechanismus portálů je určitě příliš sofistikovaný, aby byl součástí nízkoúrovňového OpenGL, je tedy na vývojářích ho vytvořit.



Obrázek 3. Portal Culling. Buňky označeny jako C1 až 9. Buňky C1, C3, C6, C7 a C8 jsou algoritmem označeny jako viditelné.

3.7 Zjednodušování scény

Další ze skupiny urychlujících nebo eliminujících algoritmů je například detail culling. Ten během transformace odstraňuje objekty, projevující se jen malým počem pixelů v projekci. Populárnějším zástupcem je level of detail, který funguje obdobně, místo odstranění ovšem pouze nahradí daný objekt jednodušším. Algoritmus je tak flexibilnější a funguje i v okamžicích, kdy je pozorovatel statický, avšak je třeba řešit momenty přepnutí modelů (lod switching), kdy při změně úrovně detailů může pozorovatel vidět skokovou změnu, řada technik se věnuje otázce plynulejšího přechodu mezi modely. [1]



Obrázek 4. Ukázka principu Level of detail. Vlevo maximálně detailní model blízko kamery. Vpravo nejvíce zjednodušený model daleko od kamery. Zdroj: http://www.opensg.org/projects/opensg/wiki/Tutorial/OpenSG2/NodeCores.

4 Dělení scény a pomocné objekty

Vytváří datovou strukturu, vztahující se ke scéně a objektům v ní. Organizuje geometrii scény n-rozměrného prostoru a slouží k urychlení výpočtů a dotazů na překryv geometrických entit. Jsou častým prvek při eliminaci neviditelných částí scény, ray tracingu, při detekci kolizí.
Struktury jsou obvykle hierarchické, kde nejvyšší úroveň obsahuje úrovně nižší. Často má formu stromu, kdy je v listech uchována geometrie a ostatní uzly representují skupiny těchto geometrických množin. Díky hierarchizaci se obvykle očekává zlepšení složitosti dotazů z O(n) na O(log n).
Podobně jako bylo uvedeno u dělení prostoru na buňky a portály, tyto obecné struktury mohou být vytvořeny buď předpočítáním, nebo průběžně za běhu po částech. Zvyšují paměťovou náročnost ale i efektivitu. Je možné je komprimovat, což ovšem efektivitu opět sníží, je tedy nutné nalézt rovnováhu mezi kritérii. [1]

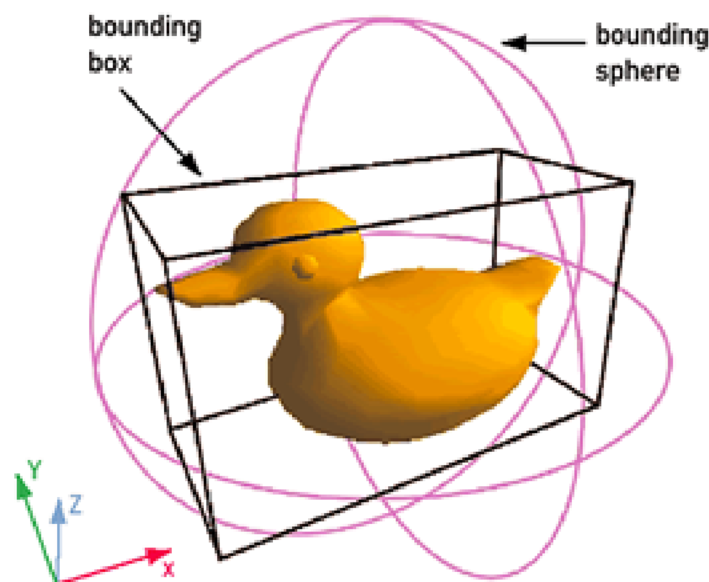
Významnými technikami jsou hierarchie obálek, BSP stromy, oktalové stromy.

4.1 Bounding volume (hierarchy)

V mnoha případech, kdy často jde o úvodní kroky algoritmů, není třeba práce s exaktními modely. Ty mohou mít tisíce i miliony polygonů a přitom v případě potřeby (např. pouze hrubé siluety) si můžeme vystačit s modelem nesrovnatelně jednodušším. Také shluk objektů, složený z řady menších či nespočtu částic či bodů, je možné nahradit jediným podobným tělesem.

Bounding volume je tedy zástupný objekt, který se používá při výpočtech namísto původního složitějšího. Je běžné využívat dokonce až základní prostorová tělesa, bounding box (krychle nebo kvádr) a bounding sphere patří mezi nejklasičtější. Popularita bounding boxu je dána tím, že poměrně přijatelně reprezentuje velice často zobrazované objekty, jako jsou osoby, zvířata, budovy, auta aj. Termín bounding box se mnohdy v praxi nesprávně používá i pro dvourozměrné objekty.
Pod pojmem bounding volume je zpravidla myšleno minimum bounding volume, tedy objekt nejmenší velikosti takové, že v sobě ještě obsáhne celý zastupovaný objekt či objekty. Rozlišuje se také oriented bounding volume, axis-aligned (ortogonální) bounding volume a jiné [1].

Bounding volume může být hierarchizované, tzn. bounding volumes jednotlivých zastoupených objektů mohou být obsaženy v nadřazeném bounding volume. Tuto strukturu lze reprezentovat jako strom, kde kořenový uzel reprezentuje BV obsahující celou scénu; nemusí jít jen o binární strom, efektivnějších výsledků se pro některé aplikace dosáhlo použitím stromů se 4 nebo 8 potomky. Tímto způsobem lze například provádět výpočty nad objekty, reprezentujícími vysoce detailní zástup lidí, za použití pouze pár stovek polygonů. Tato technika se využívá například v některých algoritmech pro occlusion culling [4]. BVH lze využít i v dynamických scénách, kde se při změně bounding volumes buď přepočítají, rodičovské BV se zvětší, nebo se využije temporal bounding volume [1].



Obrázek 5. Ukázka obalujících těles. Zdroj: http://www.tcs.fudan.edu.cn/rudolf/Courses/Algorithms/Alg_ss_07w/Webprojects/Chen_hull/applications.htm.

4.2 BSP strom

Binary space partitioning tree je datová struktura, která rekurzivně člení celý prostor scény pomocí rovin. Sjednocení listů stromu dá dohromady opět celou scénu.

První výhodou BSP stromů je hierarchizace. Pokud provedeme určitý test na vyšší úrovni stromu, na uzlu, který obsahuje řadu podúrovní a listů, a tento test selže, můžeme tímto vyloučit a netestovat i všechny potomky. V určitých příznivých situacích, při velkém množství geometrických objektů a negativních výsledcích dotazů, tak můžeme ušetřit i miliony operací na dotaz. Také urychluje procházení struktury a testování průniků [1].

Druhou výhodou BSP stromů je možnost traverzovat ho způsobem, kdy se projde geometrie seřazená podle vzdálenosti z libovolného bodu. Vzhledem k principu stínění objektů jinými objekty (viz occlusion culling) a způsobu výpočtu některých algoritmů (viz z-culling) dokáže tato struktura urychlit výpočet těchto operací. U polygon-aligned BSP stromu je toto seřazení přesné, u axis-aligned pouze přibližné [1].

Lze ho rozdělit na dva typy, první dělí rovinami souběžnými se souřadnými osami (axis-aligned, ortogonální), druhá rovinami souběžnými s určitými polygony ve scéně (polygon-aligned).

Ortogonální BSP začíná s ortogonálním bounding boxem celé scény. Ten je rozdělen po určité ose (ne nutně středem) a boxy vzniklé tímto dělením jsou opět rekurzivně rozděleny stejným principem. Toto dělení pokračuje do splnění stanovené podmínky. Strategie pro volbu osy, kterou povede dělící rovina, mohou být:

- Cyklování os v pořadí x,y,z: pak jde o variantu zvanou kD-tree
- Řez podél nejdelší strany boxu
- Použitím teorie geometrické pravděpodobnosti

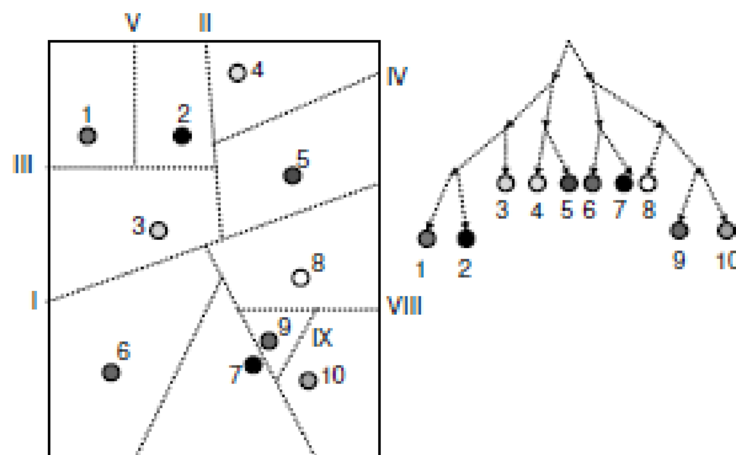
Objekty, které by měly být rovinou rozděleny, buď mohou být zaznamenány do dané úrovně stromu (a díky tomu ponechat pouze jednu instanci ve stromu), přiřadit je oběma potomkům, nebo objekty rozdělit na dva samostatné (zde se generují nové polygony podél dělící roviny). [1]

U polygon-aligned BSP jsou dělící roviny dány určitým polygonem scény. Tato rovina dělí geometrii do dvou skupin a všechny protnuté polygony jsou rozpůleny. Provede se stejný výpočet s výběrem jiného polygonu a takto se rekurzivně pokračuje, dokud nejsou všechny polygony přiřazeny do stromu. Podobně jako u ortogonální varianty je nutno zvolit správnou strategii pro výběr polygonu, výsledné stromy mají různou míru efektivity, přičemž je vhodné se snažit o dosáhnutí vyváženého stromu (tj. hloubky jednotlivých větví se liší maximálně o jednu úroveň). [1] Významný parametr je množství polygonů, rozdělených při tvorbě stromu. V horším případě může počet polygonů narůst až na $O(n^2)$, v praxi je obvykle $O(n)$, přičemž nárůst bývá přibližně na dvojnásobek. Jedna ze strategií je tzv. least-crossed criterion, kde se z náhodného výběru polygonů vybere ten, který dělí nejméně polygonů. [12]

Díky přesnému seřazení geometrie podle vzdálenosti z daného bodu je tato varianta výhodná pro použití se statickou geometrií. Mechanismus procházení je takový, že zjišťujeme, na které straně dělící roviny je kamera a tuto detekci rekurzivně opakujeme pro další úrovně zanoření do BSP stromu. [1]

Dělení a generování scény je možno provádět i za běhu programu, kde toto řešení pro ortogonální BSP bylo prezentováno (Aila, Miettinen [14]). Optimalizací by mohlo být stanovení počátečních bodů pozorovatele, blízko kterých by prioritně probíhal výpočet. Tento způsob ovšem omezuje, až znemožňuje, výpočet předběžné potenciální sady viditelnosti.
BSP strom může být výhodné doplnit dalšími mechanismy. Pokud máme strom, kde je mnoho těles přiřazeno více buňkám, lze využít tzv. poštovní schránku (mail box), která uchovává seznam již testovaných objektů, pro zamezení duplicitního testování. BSP strom také sice dělí prostor, ale není zaručeno, že prostorově sousedící objekty budou vloženy do podobné části stromu, tím méně přímo do sousedního uzlu. Je tak vhodné tyto vazby (provazy, ropes) do listů doplnit. To znamená i delší čas předzpracování a více uložené paměti. [4]

Nevýhodou BSP stromu je určitá rigidita, kdy pro změny ve scéně (změna pozice elementů, nový element, odstraněný element) je nutno provést rozsáhlé výpočty, při vkládání případně může docházet k degradaci. [21]



Obrázek 6. Schéma dělení prostoru do BSP stromu. [21]

4.3 Octree

Další populární variantou je oktalový strom (octree). Je podobný ortogonálnímu BSP stromu, kdy je trojrozměrný prostor dělen všemi osami, protínajícími se ve středu prostoru, rekurzivně až do splnění určitého kritéria. Toto dělení vytváří boxy (osmici), boxům v listech stromu je pak přiřazena geometrie. Pokud je objekt příliš velký, lze ho přiřadit do vyšší úrovně (vnitřního uzlu). Výhodné je, že je tato struktura pravidelnější a umožňuje rychlejší dotazování [1]. Na druhé straně je méně

schopno se přizpůsobit geometrii v obraze než ortogonální BSP strom, má obecně více buňek, více prázdných buněk a protnutých těles [4].

Existuje varianta tzv. loose octree, kde velikost boxů není rigidně stanovená, je dán koeficient zvětšení boxů a boxy se tak překrývají. Díky této flexibilitě lépe dělí tělesa a sníží se také četnost přegenerovávání octree v dynamických scénách. Flexibilita je i nevýhodou, je nutno provést více výpočtů, jelikož obálky vyšší úrovně obsahují více než jen své potomky. Loose octree se používá např. při detekci kolizí nebo occlusion culling. [13]

5 Návrh

Tato kapitola pojednává o obecném navrženém řešení, konkrétní postupy a algoritmy budou následovat v kapitole Implementace.

Cílem je vytvořit algoritmus, efektivně vyhodnocující viditelnost v prostoru, a vytvořit program, jenž ho bude prezentovat. Jako prostor, ve kterém bude algoritmus využíván, bude uvažován interiér s vysokým stupněm zastínění, jako je například bludiště. Scéna bude algoritmu dostupná před jeho vlastním spuštěním, což znamená možnost využít předzpracování. Toto předzpracování není principielně časově neomezeno, z hlediska využitelnosti algoritmu však nelze tolerovat mnohahodinový běh takového programu. Algoritmus bude za běhu stanovovat viditelnost ve scéně z aktuální pozice a efektivně předávat tato data k vykreslení na obrazovku. V potaz budou brány statické předměty umístěny ve scéně, ne ovšem objekty dynamické.

Hlavními technikami použitými v řešení budou portály, BSP stromy a occlusion queries. Portal culling potřebuje ke svému použití identifikovat buňky a portály. K tomu využijeme procesu hierarchizace prostoru polygon-aligned BSP stromem, díky kterému budeme zároveň moci procházet buňky seřazeně dle vzdálenosti od daného bodu. Occlusion queries pak budou sloužit jako doplněk určení konkrétní viditelnosti portálů za běhu i s ohledem na statické objekty a potenciálně i dynamické objekty.

Řešení bude probíhat v následujících fázích:

1. Hierarchizace scény a stanovení buněk
2. Generování portálů a přiřazení portálů buňkám
3. Částečné stanovení PVS
4. Detekce viditelných portálů
5. Nastavení vykreslovacího řetězce a předání dat k vykreslení

První tři zmíněné kroky budou provedeny před během programu a výsledek uložen do paměti. Třetí krok je v předzpracování pouze částečně, stanovování potenciální sady viditelnosti bude probíhat primárně během čtvrtého kroku.

Jednotlivé kroky budou rozepsány v následujících kapitolách.

5.1 Hierarchizace scény a stanovení buněk

Tato práce využívá algoritmus, prezentovaný Brettem Wadem [20] a další části Samuelem Ranta-Eskolou [5]. Jedná se o konstrukci BSP strom ve variantě polygon-aligned.

Při generování stromu je třeba stanovit klasifikaci vztahů mezi polygony [5][19]:

- Množina polygonů je konvexní, pokud mezi každou dvojicí platí konvexnost. Tu můžeme definovat tak, že jeden polygon dvojice leží v pozitivním poloprostoru druhého a i druhý leží v pozitivním poloprostoru druhého, nebo sdílí stejnou rovinu.
- Jeden polygon se nachází před druhým, pokud leží v pozitivním poloprostoru stanoveným tímto druhým polygonem

Iterativní algoritmus vezme aktuální sadu polygonů, která je na počátku rovna všem polygonům scény. Z této sady vybere dělící polygon, který rozdělí prostor na dva poloprostory. Ty se rozlišují jako pozitivní a negativní poloprostor podle orientace polygonu (v pozitivním směru se nachází normála polygonu) [19]. Polygony pozitivního poloprostoru jsou pak zpracovávány v jedné větvi stromu (a výpočtu), polygony negativního pak ve druhé. Pokud určitý polygon leží částečně v pozitivním i negativním poloprostoru, pak je rozdělen na dva polygony.

Pokud je množina polygonů konvexní, je výpočet v této větvi ukončen. Všechny polygony z množiny jsou uloženy do listového uzlu stromu a jsou označeny za buňku. Jinak výpočty probíhají stejným způsobem v obou vzniklých podvětvích.

Při výběru dělících polygonů je třeba brát v úvahu dva různé cíle, a to finální počet polygonů a vyváženost stromu, kde obojí má vliv na výkon. Vyvážený strom lze procházet paralelně a dělit sadu vždy na dvě podobně početné množiny. Vyšší počet polygonů pak zvyšuje náročnost všech operací, které se provádí na úrovni polygonů, hran a vrcholů. V práci bude použita mírně upravená strategie prezentovaná p. Eskolou [5], kdy se bude dělící polygon vybírat podle nejvyváženějšího poměru rozdělení scény na dvě části a nejmenšího množství řezů. Pokud bude množina dostatečně velká, nebude se ale vybírat ze všech polygonů, ale z náhodné podmnožiny. Při neúspěchu se vybere jiná podmnožina, při určitém počtu selhání teprve dojde k výběru z celé množiny a ke snížení nároků na daný dělící polygon. Pokud ani pak žádný nevyhoví, výpočet v podvětvi se ukončí.

5.2 Generování portálů a přiřazení portálů

V následující fázi automatické přípravy scény přichází generování portálů. Jde o označení průhledné plochy mezi jednotlivými buňkami, stanovenými v předchozí fázi. Algoritmus je postavený na principu od Andrease Brincka [5].

Základním předpokladem je, že portály leží ve stejné rovině jako dělící polygony. Je tedy vytvořen polygon v dané rovině, jehož všechny body leží uvnitř nebo na obálce scény hierarchizované BSP. Tento polygon je pak předáván skrze BSP strukturu od kořenového prvku. V každém kroku je ořezán dělícím polygonem a u listového uzlu (tj. buňky) potom obalujícím kvádrem buňky a všemi souběžnými polygony (tedy ležícími ve stejné rovině). Výsledný polygon je označen za portál a přiřazen dané buňce. Jakmile je mu přiřazena i druhá, vytvoří se tak prostorová relace mezi buňkami, které nemusí ležet v sousedních listech stromu.

5.3 Částečné stanovení PVS

V této fázi (stále v rámci předzpracování scény) se stanovuje potenciálně viditelná sada pro každou buňku. Toto je možno řešit např. pomocí metody ray tracing [5][10], provést analytický výpočet přesné potenciální viditelnosti [2][3], nebo stanovit pouze částečnou PVS a odložit výpočet na běh programu [7].

Tato práce vychází a doplňuje verzi představenou pány Luebke a Georges [7]. PVS bude řešeno na úrovni buněk, kdy pokud je buňka viditelná, je za viditelné prozatím označeno i vše v ní. Viditelné objekty mohou být dále zpřesněny v dalších fázích, případně v rámci vykreslovacího řetězce.

Z toho vyplývá, že buňka, ve které se nachází pozorovatel, je určitě viditelná. Tuto skutečnost není třeba ukládat do paměti. Toto řešení PVS dále stanovuje pouze tou měrou, že potenciálně viditelná sada je přibližně určena formou množiny portálů dané buňky, kdy sousedící buňky – při uvažování pouze konvexních buněk – jsou jistě viditelné.

5.4 Detekce viditelných portálů

V průběhu vykreslování bude stanovována množina potenciálně viditelných prvků scény a tyto průběžně předávány k vykreslení. Inspirací je práce Luebke a Georges [7], kteří popisují a upravují princip Cliffa Jonese [18] pro implementaci portal culling. Na prostor obrazu bude projektován bounding rectangle portálů (označovaný jako cull box) výchozí buňky (buňky, ve které se nachází pozorovatel). Portál je považován za viditelný, pokud se alespoň jeden vrchol jeho bounding recangle projektuje do prostoru obrazu. V případě, že má portál v sekvenci portálů, viditelných z výchozí buňky, nějaké předky, musí alespoň jedna jeho část zasahovat do průniku projekcí předchozích portálů. Pro aktuální portál bude vždy platný ten průnik, který náleží předchozímu portálu v sekvenci portálů mezi jím a výchozí buňkou.

Při procházení jsou testovány následující portály oproti dosavadnímu průniku sekvence cull boxů portálů. Pokud bude alespoň část portálu zasahovat do tohoto regionu, bude jemu příslušná buňka označena jako viditelná.

Práce doplňuje formu occlusion cullingu portálů, implementovanou pomocí hardware occlusion query. Bude tak možno vyloučit portály, které jsou zakryty jiným tělesem. Zkoumaný portál bude nejdříve otestován porovnáním s průnikem (viz úvodní odstavec této části), poté bude jeho neprůhledná reprezentace použita jako subjekt v occlusion query. Pokud bude jakýkoli pixel viditelný, bude portál označen jako viditelný. Alternativně lze stanovit minimální počet pixelů, nutných pro označení portálu za viditelný, což změní klasifikaci této části z konzervativní na přibližnou, avšak pro velmi nízký limit pixelů nemusí být vizuální kvalita rozlišitelná a získáme tím možnost vyloučit danou buňku, a všechny v její větvi následující, z výpočtu.

S occlusion queries se pojí řada skutečností, které je nutno vzít v potaz [8]:

1. Jedná se o asynchronní funkcionalitu, je vkládána do fronty příkazů a její vyhodnocení trvá určitý čas. Pokud v tento čas procesor neprovádí další zpracování a nepředává grafické kartě další data k vykreslení, je tento čas vynaložen neúčelně
2. Z důvodu předchozí skutečnosti navíc může dojít k vyprázdnění vyrovnávací paměti grafické karty, pak bude muset čekat i ona.
3. V neposlední řadě je tu určitá režie spojená se zpracováním dotazu a samotné vyhodnocení dotazu, náročnost rasterizace

Problémy uvedené v prvních dvou bodech lze alespoň částečně řešit pomocí temporální a prostorové koherence a fronty příkazů, mezi kterými je (namísto čekání) zařazeno zpracování jiných částí scény. Dále je doporučena hierarchizace scény, ta je již dostatečně provedena rozdělením na portály pomocí BSP stromu. Pokud je jedna buňka označena za neviditelnou, další nejsou zpracovávány (a jejich neviditelnost je označena pro účely koherence). U temporální a prostorové koherence čekáme pouze malou změnu viditelnosti mezi jednotlivými snímky a proto výsledky předchozích výpočtů uchováváme. Pokud se dříve viditelný portál stal neviditelným, tuto skutečnost u něj poznačíme. Pokud se dříve neviditelný portál stane viditelným, je nutno ho ihned vykreslit. U neviditelných portálů je tedy nutno provést ověření. [8]

Řešení bude prezentovat dva přístupy použití occlusion query. Ten první je samostatné testování viditelnosti polygonální reprezentace portálu, která nebude zapsána do framebufferu. Occlusion query dotaz se pošle ještě před testováním cull boxu portálů. U druhého algoritmu se bude testovat viditelnost samotné buňky (která by se jinak vykreslovala každopádně). Jelikož viditelná plocha je nanejvýš stejná jako plocha portálu a vykreslení by bylo zadáno v algoritmu bez occlusion query i tak, výkonnostně může být tato varianta paradoxně lepší. Tato varianta může pak pouze vyloučit buňku, která byla označena za viditelnou (a ne naopak). Klasifikace výsledku obou variant (zda je konzervativní, agresivní apod.) pak záleží na jejich použití a podmínění použití (tj. například zda je přijímáno pouze vyhodnocení ze stejné pozice). Ze zmíněných důvodů ale tato varianta může být potenciálně agresivnější.

Bude možno nastavit, zda se bude occlusion query test aplikovat již od výchozí buňky, nebo až od určité úrovně zanoření.

Obrázek 7. Projekce axis-aligned bounding rectangle portálů; zobrazení ořezu frustrum portály. [7]

5.5 Zaměření a hodnocení efektivity

Po dokončení implementace bude vytvořeno několik testovacích scén a vybrány množiny nastavení algoritmu, které budou použity. Hlavními kritérii (dle důležitosti) budou:

1. Přesnost detekce a korektnost výsledného zobrazení
2. Paměť potřebná pro vygenerování snímku (resp. množství či poměr vyřazených polygonů)
3. Čas potřebný pro zpracování algoritmu
4. Čas potřebný pro vygenerování snímku scény

6 Implementace

Implementace se uživatelsky dělí na dvě části: hlavní program, který spustí průchod jednou určitou scénou v zadané konfiguraci a obsahuje konkrétní algoritmy pro dělení scény a detekci viditelnosti; dále uživatelské rozhraní, které umožňuje procházet všechny námi vytvořené konfigurace, spouštět je a prohlížet výsledky. Následující kapitoly se budou věnovat výhradně hlavnímu programu, s výjimkou části Uživatelské rozhraní.

6.1 Návrh programu

6.1.1 Popis a konfigurace

Program představuje průchod scénou. Každá scéna obsahuje:

- Prostorová data prostředí (stěny), povinně
- Prostorová data statických objektů
- Trajektorii

Prostorová data popisují vlastní prostředí, které je zpracováno BSP a portály. Statické objekty slouží k doplnění objektů a zvýšení náročnosti výpočtu, ale jsou vkládány až po předzpracování scény.

Standardně se lze po scéně volně pohybovat, trajektorie umožňuje projití po stále stejné trase pro zjednodušení práce uživatele, jednotné podněty pro testování algoritmů a prezentační účely.

Pro zmenšení počtu argumentů příkazové řádky je většina údajů načítána z konfiguračního souboru s navrhovanou příponou .portal-config. Zde lze nastavit rozměry okna, cesty k souborům definujícím data scény a trajektorii, použité algoritmy a další (viz Příloha B).

Příkazovou řádkou specifikujeme cestu k tomuto souboru a volitelně další nastavení:

- Počet znovuspuštění vykreslovací fáze programu
- Cesta k výstupnímu souboru
- Aktivování prezentačního módu

  nebo
- Aktivování benchmark módu

V prezentačním módu je kamera umístěna mimo pozorovatele scény (více viz podkapitola Běh a vykreslování). Benchmark mód, výstupní soubor a znovuspuštění je vysvětleno dále v kapitole Vyhodnocení.

6.1.2 Zdrojová data

Pro definici geometrie používá aplikace formát Wavefront .obj. Jedná se o jednoduchý textový soubor s pozic vertexů a normál jednotlivých elementů scény. Umožňuje i definici skupin vyhlazení (smoothing groups), texturovací koordináty a materiály, tyto položky práce nevyužívá. [22]

Pro načítán používá práce hlavičkovou knihovnu Gerharda Reitmayra objload. Ten poskytuje načtená data v objektu:

```
struct Model {
	vector<float> vertex;
	vector<float> texCoord;
	vector<float> normal;

	map<string, vector<unsigned short> > faces;
};
```

kde *faces* má pro každý pojmenovaný objekt scény seznam indexů do ostatních kontejnerů.

Podrobnosti ohledně .obj souborů jsou popsány v Příloze F.

Soubor trajektorie je textový soubor, kde jsou na každém řádku mezerou odděleny rámec, prostorové souřadnice kamery, souřadnice směrového vektoru kamery (počátek v souřadnici kamery).

6.1.3 Datový návrh

Data ze zdrojových souborů představují převažující část dat, zpracovávaných aplikací. Ta jsou načtena pomocí knihovny objload a potom konvertovány na vlastní strukturu, do kontejneru vector objektů Polygon:

```
struct Polygon {
	vector<glm::vec3> vertexes;
	glm::vec3 normal;
	glm::vec4 color;
/* ... */
}
```

V této fázi jsou ještě podporovány obecné polygony s N vertexy. Knihovna sice importuje polygony jako množinu trojúhelníků, ale při dělení polygonů kandidátním polygonem u BSP algoritmu a při generování portálů z obálek a ořezávání polygonů mohou vzniknout polygony i s více než čtyřmi vertexy. Před renderováním je pak proveden převod na trojúhelníky, což je efektivnější struktura pro vykreslování na grafickém akcelerátoru. Typy elementů GL_QUAD (čtyři vrcholy) a GL_POLYGON (obecný počet polygonů) byly již v OpenGL označeny jako zastaralé (deprecated), jsou tedy podporované, ale jejich použití není doporučeno [16].

Během generování BspStromu se pak vytváření uzly BspNode.

```
class BspNode {
	vector<Polygon*> polygons;
	vector<Portal*> portals;
/* ... */
};
```

Zde vypsána pouze datová část. Všechna práce se děje s odkazy na původní polygony, data jsou tedy v aplikaci přítomna jednou a nejsou kopírována. Výjimkou je část BspNode, kde je z optimalizačních důvodu provedena také konverze na množinu hodnot typu float.

```
class BspNode {
	vector<float> vertices;
	vector<float> colors;
/* ... */
};
```

6.1.4 Třídy

V následující části je stručný přehled tříd aplikace v chronologickém pořadí aktivace jejich činnosti v programu. Na konci je potom vyobrazeno schéma tříd a jejich dynamické vztahy.

6.1.4.1 Config

Třída config obsahuje a verifikuje data příkazové řádky.

6.1.4.2 Logger

Řídí zápis výstupu do souboru.

6.1.4.3 Stats

Shromažďují statistiky z hlavního běhu programu. Pro každý množstevní i časový parametr udržuje minimální, maximální a průměrnou hodnotu a množství vzorků. Má úzký vztah se třídou Renderer.

6.1.4.4 BspGenerator

Slouží ke konstrukci BSP stromu z dat získaných ze zadaného souboru. Obsahuje metody pro generování stromu, výběr dělícího polygonu, dělení polygonu polygonem a metody pro načtení statických objektů a dělení na trojúhelníky. Poslední tři by stálo za úvahu přesunout jinam.

6.1.4.5 BspTree

Reprezentuje BSP strom. Má odkaz na kořenový prvek a BspGenerator. Obsahuje metody pro akci se stromem, které deleguje na kořenový uzel, dokáže vrátit listové uzly nebo dělící polygony.

6.1.4.6 BspNode

Reprezentuje uzel BSP stromu. Kromě dat primárně poskytuje metodu na vyhledání zadaného vertexu (který případně posílá níže stromem).

6.1.4.7 PortalPlacer

Zapouzdřuje algoritmus generování portálů a jejich vkládání do BSP stromu.

6.1.4.8 Portal

Reprezentuje portál s odkazy na spojené buňky.

6.1.4.9 RenderConfig

Udržuje konfiguraci vykreslování, zadaných dat a detekce viditelnosti. Kompletní výpis hodnot je popsán v Příloze B.

6.1.4.10 Vertex

Reprezentuje jeden bod v prostoru, určený třemi souřadnicemi kartézské soustavy.

6.1.4.11 Polygon

Reprezentuje polygon určený N souřadnicemi a normálovým vektorem. Mezi hlavní metody patří:

- Vrácení polygonu na stejné rovině, vyplňujícího určitou obálku
- Stanovení relace k určitému bodu či jinému polygonu
- Ořezání souběžných polygonů
- Stanovení zda obsahuje daný vertex

6.1.4.12 Edge

Reprezentuje hranu určenou dvěma body. Umí určit relaci s jinou hranou, a zda obsahuje určitý vertex.

6.1.4.13 EdgeCollection

Kolekce hran, většinou určující polygon. Umí najít ekvivalentní kolekci polygonu vyplňujícího obálku, vrátit průsečíky s hranami nebo určit, zda obsahuje uvnitř určitý bod.

6.1.4.14 BoundingRectangle2D

Určuje obdélník ve dvourozměrném prostoru (v práci konkrétně v obrazových souřadnicích).

6.1.4.15 PolarCoordinates

Reprezentují polární souřadnice zadaného bodu a umí porovnat pozici a úhel s jinou polární souřadnicí.

6.1.4.16 Renderer

Třída, která nastavuje vykreslování, řídí volání vykreslovací metody, upravuje pozici dle uživatelského vstupu nebo dané trajektorie. Významná je její vazba s některou s implementací ContentRenderer, jejíž metodu volá pro vykreslení konkrétních dat.

6.1.4.17 SceneObserver

Obsahuje údaje o pozici a směru pohledu pozorovatele.

6.1.4.18 SceneState

Obsahuje SceneObserver a dále index aktuálního snímku scény (tedy nejen pozici, ale i čas z pohledu aplikace).

6.1.4.19 ContentRenderer

Abstraktní třída, téměř vlastně rozhraní. Deklaruje rozhraní metody, která slouží k vykreslení konkrétních dat scény.

6.1.4.20 SimpleRenderer

Triviální implementace renderování, popsána dále v podkapitole Běh a vykreslování.

6.1.4.21 PortalRenderer

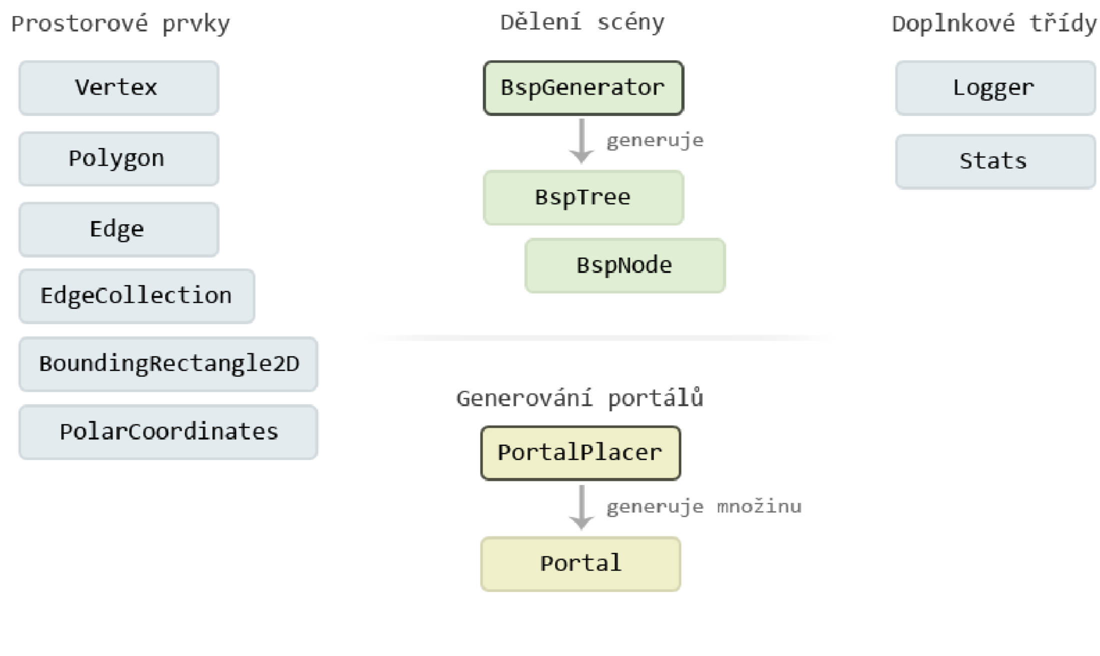
Třída, která detekuje viditelnost pomocí portálů a případně i OcclusionQuery a zadává její vykreslení třídě BufferedPortalRenderer. Popsána dále v podkapitole Běh a vykreslování.

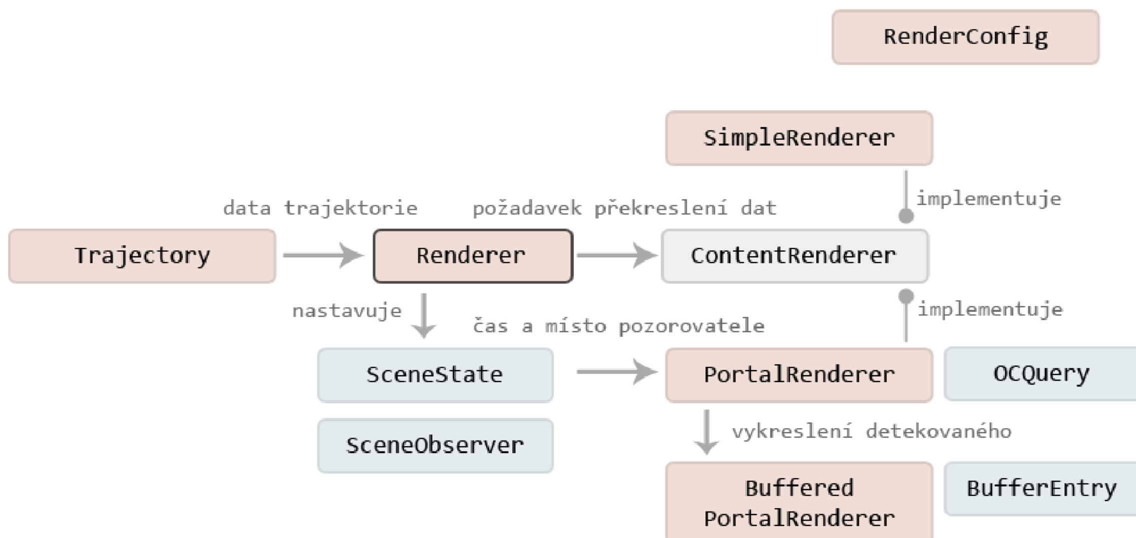
6.1.4.22 BufferedPortalRenderer

Zadává data polygonů buněk k jejich vykreslení grafické kartě. Provádí to efektivním způsobem udržováním kolekce BufferEntry s údaji, které buňky už jsou nahrány.

6.1.4.23 Trajectory

Udržuje data trajektorie pozorovatele ve scéně. Dokáže vrátit data pro konkrétní snímek. Pokud tento snímek nemá a je někde mezi počátečním a koncovým, dokáže údaje interpolovat.





Obrázek 9. Dynamické vztahy mezi třídami, chronologicky od spuštění aplikace.

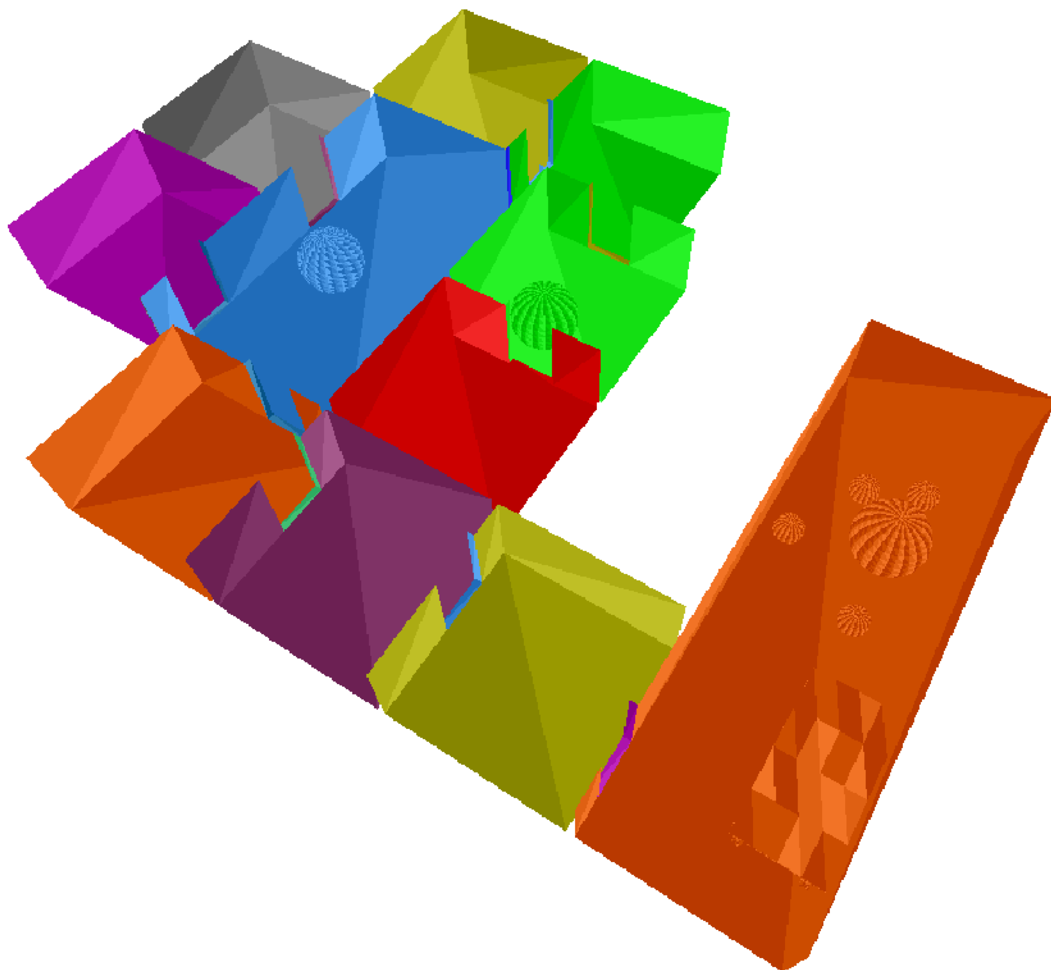
6.2 Dělení scény

Dělení scény zajišťuje třída BspGenerator. Kromě obecného požadavku na vytvoření BSP stromu chceme, aby listy stromu byly vždy konvexní množiny polygonů. To znamená, že mezi každým polygonem je relace taková, že leží buď ve stejné rovině, nebo jsou oba v pozitivním podprostoru toho druhého. Ve zdrojových kódech je pozitivní označena jako levá a negativní jako pravá strana.

Dělení scény vychází z algoritmů popsaných Ranta-Eskolou [5]. Následuje iterativní verze základní části algoritmu v pseudokódu:

```
VytvořPrázdnýStrom()
BspNode kořenovýUzel, aktuálníUzel

PřiřaďVšechnyPolygony(kořenovýUzel)

List<BspNode*> SeznamNezpracovanýchUzlů
Vector<Polygon> PozitivniStrana, NegativniStrana
Polygon dělícíPolygon

do {
	aktuálníUzel = SeznamNezpracovanýchUzlů.vyjmiPrvni()

	if( jeKonvexni(aktuálníUzel.polygony) ) {
		continue;
	}

	dělícíPolygon = najdiDělícíPolygon(aktuálníUzel.polygony)
	for(Polygon p z aktuálníUzel.polygony) {
		er = dělícíPolygon.relace(p)

		if( er == Pozitivni || er == Souběžný ) {
			PřiřaďDoPozitivní(p)
		} else if( er == Negativní ) {
			PřiřaďDoNegativní(p)
		} else {
			Polygon p1, p2 = RozdělPolygonNaPoloviny(dělícíPolygon);
			PřiřaďDoPozitivní(p1)
			PřiřaďDoPozitivní(p2)
		}
	}

	aktuálníUzel.dělícíPolygon = dělícíPolygon
	if( PozitivníStrana.neprázdá ) {
		BspNode pozitivní(PozitivníStrana)
		aktuálníUzel.pozitivní = pozitivní
		if( NegativníStrana.neprázdná ) {
			SeznamNezpracovanýchUzlů.vlož(pozitivní)
		}
	}

	if(NegativníStrana.neprázdá ) {
		BspNode negativní(NegativníStrana)
		aktuálníUzel.negativní = negativní
		if( PozitivníStrana.neprázdná ) {
			SeznamNezpracovanýchUzlů.vlož(negativní)
		}
	}

} while(SeznamNezpracovanýchUzlů je neprázdný)
```

Výsledkem je BspTree s hierarchií stromově propojených elementů BspNode. Náhled rozdělení scény BSP je v následujícím obrázku.



Obrázek 10. Scéna po rozdělení pomocí BSP na konvexní oblasti. Zelené buňky pouze dostaly stejný index barvy, v hierarchii jde o odlišné objekty.

Algoritmus výběru dělícího polygonu také vychází z algoritmů pana Eskoly [5]. Na základě toho, že bylo prokázáno, že pro výběr polygonu scény s nejnižším počtem průniků s ostatními polygony stačí prozkoumat 5 náhodně vybraných polygonů [19], nevybírají se zkoumané polygony ze všech polygonů, ale z náhodně vybrané podmnožiny.

Pseudoalgoritmus výběru unikátních náhodných prvků:

```
VyberNáhodnéPrvky(VstupníVektor, VýstupníVektor, početPrvků) {
        For( i ... početPrvků ) {
                Index = NáhodnéČíslo(0 ...VstupníVektor.délka)
                if( VýstupníVektor.obsahuje(VstupníVektor[index]) == false ) {
                VýstupníVektor.přidej(VstupníVektor[index])
                }
}
```

Jediná výkonnostní nevýhoda algoritmu je detekce unikátnosti vybraného prvku pro cílovou kolekci. V algoritmu je tak provedeno navíc PočetPrvků^2 / 2 porovnání.Výběr náhodného čísla má ale

konstantní složitost, stejně tak přístup na prvek při výběru vhodného kontejneru (například std::vector).

Pseudokód výběru je potom:

```
VyberDělícíPolygon(Polygony) {
	if( jeKonvexni(Polygony) ) {
		return NULL
	}

	Vector<Polygon> Selekce;
	if( Polygony.počet > 2N ) {
		VyberNáhodnéPrvky(Polygony,Selekce,N)
	} else {
		Selekce = Polygony
	}

	Polygon* kandidát = NULL
	PozitivníStrana = NegativníStrana = PočetDělení = 0
	NejmenšíDělení = NejlepšíPoměr = Polygony.počet
	PřijatelnýPoměr = 4.9 /* vybráno po testování */
	Limit = 4
	do {
		for(Polygon s z Selekce) {
			/* relace s polygony scény */
			for(Polygon p z Polygon) {
				er = s.relace(p)
				if( er == Pozitivni || er == Souběžný ) {
					PozitivníStrana++
				} else if( er == Negativní ) {
					NegativníStran++
				} else {
					PočetDělení++
				}

			}

			/* všechny polygony na jedné straně – maximálně nevyvážené,
není to kandidát */
			if(PozitivníStrana == 0 || NegativníStran == 0) {
				continue;
			}

			/* vždy větší / menší */
			Poměr = poměr(PozitivníStrana, NegativníStrana)
			if(
		(Poměr < PřijatelnýPoměr && PočetDělení < NejmenšíDělení) ||
		(PočetDělení == NejmenšíDělení && Poměr < NejlepšíPoměr)
			) {
				kandidát = s
			}
		}

		// neuspěli jsme – snížíme kritéria a vybereme novou podmnožinu
		PřijatelnýPoměr *= 1.3
		if( --Limit > 0 && Polygony.počet > 2N ) {
			VyberNáhodnéPrvky(Polygony,Selekce,N)
		}
```

```
	} while(kandidát == NULL);

	// neuspěli jsme s výběrem dle poměru
	if( kandidát == NULL ) {
		NejmenšíDělení = Polygony.počet
		for(Polygon p1 z Polygony) {
			for(Polygon p2 z Polygony, p2 != p1) {
				if( p2.relace(p1) == Dělící ) {
					PočetDělení++
				}
			}

			if(PočetDělení == 0) {
				return p1
			} else if(PočetDělění < NejmenšíDělení) {
				NejmenšíDělení = PočetDělění
				kandidát = p1
			}
		}
	}
}
```

kde N je počet náhodných prvků. Nebyla stanovena přímo na 5, ale 25, pro jistější detekci správného polygonu.

Nyní je již BSP strom hotový. Data jednotlivých uzlů budou ještě později upravena pro lepší následné zpracování, o BspNode se tedy bude psát ještě dále na začátku podkapitoly Běh a vykreslování.

```
class BspTree {
	BspNode* root;
	BspGenerator* generator;
}
```

6.3 Generování portálů

Z pohledu generování portálů je nyní scéna rozdělena na množinu konvexních útvarů. Potřebujeme nyní nalézt plochy, které se nachází na rozhraní mezi těmito algoritmy. Tato práce používá lehce upravený algoritmus navržený Andreasem Brinckem a prezentovaný v [5]. Uvažuje, že portály leží na stejné rovině jako dělící polygony (kterými jsme vlastně předtím tu scénu dělili). Každý dělící polygon je tedy poslán strukturou dolů jako portálový polygon. Rozlišujeme termíny portál a portálový polygon v tom, že portálový polygon je čistě definice prostorového útvaru, který reprezentuje daný portál. Portál je pak struktura, která obsahuje tento útvar, odkaz na buňky z jejíž dělícího polygonu byla vytvořena a především pak odkazy na buňky, které propojuje.

```
class BspNode {
	vector<Portal*> portals;
/* ... */
};
```

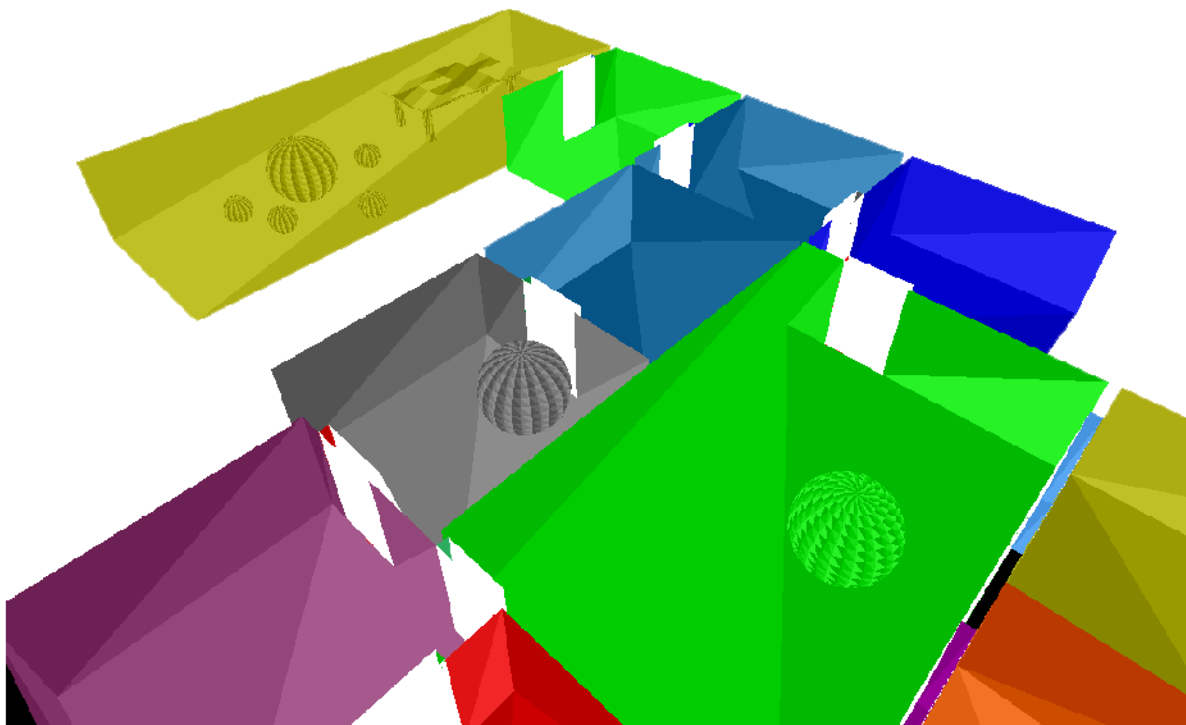
```
struct Portal {
	Polygon* shape;
	BspNode* cell1;
	BspNode* cell2;
	BspNode *originNode;
	unsigned id;
}
```

Portál vždy propojuje právě buňky (cells), jejich počet ve struktuře je tedy dán napevno. Naopak jedna buňka může obsahovat libovolný počet portálů. Aby mělo ale použití portálů smysl, měl by tento počet být poměrně malý.

V každém kroku pokračuje v té větvi, na kterou vzhledem k dělícímu polygonu (pozitivní či negativní strana) tento polygon leží. Pokud je tímto polygonem dělen, je rozdělen na dva portály (a portálové polygony) a tyto pokračují strukturou dále.

V listové buňce je pak zkoumáno, zda není nějakým z polygonů dělen. Pokud se tak stane, je polygon rozdělen a poslán od kořenové buňky stromem znovu. To je z důvodu, že v tomto případě může být buňka spojena s více jinými buňkami na stejné stěně. Pokud žádné dělení neproběhlo, je portálový polygon ořezán souběžnými polygony. Pokud se tak vytvoří více polygonů, jsou opět poslány od kořenového prvku. V případě, že zbude jediný prvek (což může být i jeden z polygonů odeslaných znovu od kořene), je mu přiřazena aktuální listová buňka. Toto řešení přiřazuje daný portál do kolekce výsledných portálů pouze v momentě, kdy je mu přiřazena i druhá buňka a daný polygon tedy skutečně leží na rozhraní dvou buněk. Před přiřazením buňce se také kontroluje, jestli daný polygon skutečně leží uvnitř konvexní množiny, v určitých kombinacích dělení BSP stromu a stanovování dělících polygonů může v některých scénách portálový polygon zasahovat mimo hranice polygonu.

Algoritmus byl také přepracován z rekurzivního na iterativní, jelikož má tendenci velmi zanoření a při menší velikosti limitu zásobníku mohlo dojít k problému.



Obrázek 11. Vyobrazení vložených portálů (v aplikaci skryto).

6.3.1 Ořezávání polygonů

V předchozí části bylo zmíněno použití algoritmu pro ořezavání polygonů (polygon clipping). V tomto případě tím myslíme vytvoření doplňku plochy původního polygonu oproti všem polygonům, které leží na stejné rovině a které do něj zasahují, překrývají ho.

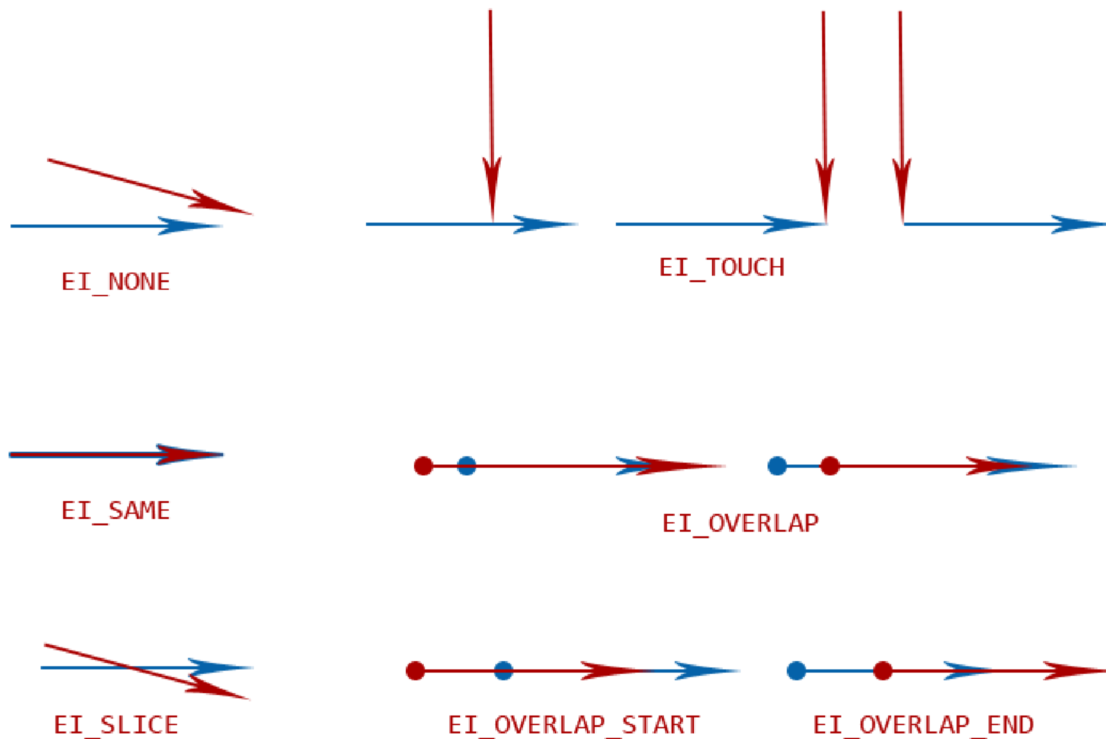
Tato práce používá vlastní algoritmus pro ořezávání polygonů založený na reprezentaci polygonů jako množiny orientovaných hran, nikoli množiny bodů. Orientace hrany je určena stejným způsobem, jako OpenGL automaticky určuje normálu elementu, tedy po směru hodinových ručiček. Bylo třeba definovat řadu tříd a funkcí určujících relace a operujících nad prostorovými prvky.

Pro algoritmus je tedy nejpodstatnější struktura hrana:

```
class Edge {
	glm::vec3 start;
	glm::vec3 end;
	glm::vec3 markedPoint;
	Edge(glm::vec3 start, glm::vec3 end): start(start), end(end) {}
	EdgeRelation getEdgeRelation(Edge& other, glm::vec3& result1, glm::vec3& 
result2);
	bool containsVertex(glm::vec3 position);
	bool equals(Edge& other);
	inline float length();
```

```
	static float getAngle(glm::vec3 center, glm::vec3 a, glm::vec3 b);
}
```

Zásadní metoda třídy Edge je getEdgeIntersection, která kromě zjištění případných průniků především celkově klasifikuje vztah jednotlivých hran, tedy segmentů přímky v prostoru s určitým počátečním a koncovým bodem. Následující obrázek popisuje všechny uvažované relace:



Obrázek 12. Klasifikace relace červené hrany ke hraně modré. Šipka ukazuje konec hrany, kruh v nutných případech začátek hrany.

Algoritmus pro zjištění relace v pseudokódu je pak následující. Hodnota EPS je velmi malé desetinné číslo, sloužící k řešení pouze přibližné reprezentace desetinných čísel. V přesné matematické reprezentaci by byla nahrazena hodnotou 0.

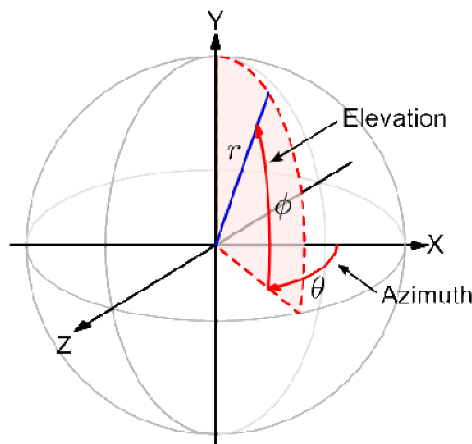
```
EdgeRelation getEdgeRelation(Edge& dva, glm::vec3& prunik1, glm::vec3& prunik2) {
	if( jedna.shodnáS(dva) ) {
		return EI_SAME
	}

	vektorJedna = jedna.konec - jedna.start
	vektorDva = dva.konec - dva.start
	vektorMezi = dva.start - jedna.start
	kolmice = dot(vektorJedna, vektorDva)

	// neleží ve stejné rovině
	if( dot(vektorMezi, kolmice) > EPS ) {
```

```
		return EI_NONE
	}

	směr = dot( cross(vektorMezi, vektorDva), kolmice ) / dot(kolmice, kolmice)

	// stejný, případně potenciálně stejný směr
	if( směr is NaN || abs(směr) < EPS || abs(směr) > (1 – EPS) ) {
		// přesuneme hrany tak, aby první byla ve středu soustavy
		jednaStartCentrovaný, jednaKonecCentrovaný
		dvaStartCentrovaný, dvaKonecCentrovaný

		/* pokud se jedna z nich nachází v záporném poloprostoru
		jsou obě posunuty aby byly ve stejném */

		// získáme polární souřadnice bodů
		polarJednaStart, polarJednaKonec, polarDvaStart, polarDvaKonec

		if(	polarDvaStart.stejnyUhel(polarJednaKonec) &&
			polarDvaKonec.stejnyUhel(polarJednaKonec) ) {

			/* nyní víme, že:
				- leží ve stejné rovině
				- mají stejný úhel
				- leží na stejné přímce */

			minr = min(polarJednaStart.r, polarJednaKonec.r)
			maxr = max(polarJednaStart.r, polarJednaKonec.r)

			if(	polarDvaStart MEZI minr,maxr &&
				polarDvaKonec MEZI minr,maxr &&
			) {
				prunik1 = dva.start
				prunik2 = dva.end
				return EI_OVERLAP
			}elif( polarDvaStart MEZI minr,maxr && polarDvaKonec > maxr ){
				prunik1 = dva.start
				return EI_OVERLAP_START
			}elif( polarDvaKonec MEZI minr,maxr && polarDvaStart < minr ){
				prunik1 = dva.konec
				return EI_OVERLAP_END
			}

		}

		if(	stejnýBod(jedna.konec, dva.konec) ||
			stejnýBod(jedna.start, dva.start) ) {
			prunik1 = start
			return EI_TOUCH
		}
		if(	stejnýBod(jedna.konec, dva.konec) ||
			stejnýBod(jedna.start, dva.start) ) {
			prunik1 = konec
			return EI_TOUCH
		}

	}

	if( směr >= 0.0 && směr <= 1.0 ) {
		prunik1 = jedna.start + vektorJedna * vec3(směr, směr, směr)
```

```
            delkaDva = delka(dva.start – dva.end)
            vzdalenostPruniku1 = delka(start – prunik1)
            vzdalenostPruniku2 = delka(konec – prunik1)

            if( abs(vzdalenostPruniku1) < EPS || abs(vzdalenostPruniku2) < EPS){
                  // protly by se, ale končí v bodu průniku
                  return EI_TOUCH
            } elif( vzdalenostPruniku1 - vzdalenostPruniku2 > delkaDva ) {
                  // protly by se, ale nejsou dostatečně dlouhé
                  return EI_NONE
            }

            return EI_SLICE
      }

      return EI_NONE
}
```

Polární souřadnice jsou vytvářeny následovně:

```
PolarniSouradnice(glm::vec3& v) // bod
      r = sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
      if( r != 0 ) {
            theta = acos(vector[2] / r);
            if( theta != theta ) {
                  theta = 0.0f;
            }
      } else {
            theta = 0;
      }

      /* arcus tangens s pomocí dvou algoritmů umožňuje určit znaménko,
      tedy kvandrant, ve kterém se výsledek nachází */
      phi = atan2(vector[1], vector[0]);
      if( phi != phi ) {
            phi = 0.0f;
      }
}
```



Obrázek 13. Vyobrazení polárních souřadnic. Zdroj: http://www.labbookpages.co.uk/audio/beamforming/coordinates.html

Dále potom významná struktura EdgeCollection, která reprezentuje skupinu hran, tedy obvykle polygon:

```
class EdgeCollection {
	EdgeCollection(Polygon* fromPolygon);

	int findStart(glm::vec3& vertex);
	bool containsPoint(const glm::vec3& vertex);
	bool containsPolygon(const Polygon& p);

	void getIntersections(Edge& edge, vector<Edge>* result);
	void makeBasePoint(Vertex* out);
	void makeBoundCollection(EdgeCollection& out);
	void makePolygon(Polygon& out);
}
```

Ta se dokáže vytvořit z polygonu a opět ze sebe vytvořit polygon.

Další vybraná funkce je pro jednoduchou detekci, zda je bod obsažen v obalovém polygonu, jiného polygonu. Předpokládá se, že bod leží ve stejné rovině.

```
Bool containsPoint(const glm::vec3& vertex) {
	Vertex v = /* bodVeStředuPolygonu() */
	Edge hrana(vertex, v)

	// kolekce z obalového polygonu této kolekce
	EdgeCollection kolekceHran
	EdgeCollection::makeBoundCollection(ec)
	EdgeRelation ei

	for(Edge h z ec) {
		ei = h.getEdgeRelation(e, null, null)
		if( ei == EI_SLICE ) {
			return false
		}
	}
	return true
}
```

S těmito metodami (a dalšími, jednoduššími) pak můžeme sestrojit algoritmus pro subtrakci souběžných polygonů:

```
Void getSubtraction(vector<Polygon> polygony, vector<Polygon>* vysledky) {
	EdgeCollection ec(this)
	EdgeCollection ecPuvodni(this) // ec bude modifikovana

	/* následuje část 1: zkonstruujeme množinu hran */

	// každý polygon
	for(Polygon p z Polygony) {
		if( /* žádný bod není obsažen v ecPuvodni */ ) {
			continue
		}

		EdgeCollection ecp(p)

		// projdeme jeho hrany
		for(Edge ep z ecp) {
```

```
			// vůči všem hranám aktuální kolekce

			// fáze 1 „overlap“
			for(Edge e z ec) {
				switch(e.getRelation(ep, p1, p2)) {
					case default: break
					case EI_OVERLAP:
					// zaměníme p1 a p2 aby p1 bylo blíž e.start
						e.konec = p1
						// vložíme novou úsečku
						ec.vlož(Edge(p2, e.původníKonec))
						// odstraníme vzniklé úsečky nulové délky
					case EI_OVERLAP_START:
						e.konec = p1
						// odstraníme vzniklé úsečky nulové délky
					case EI_OVERLAP_END:
						e.konec = p1
						// odstraníme vzniklé úsečky nulové délky

				}
			}

			// fáze 2 „shodné“
			for(Edge e z ec) {
				if( e.shodnáS(ep) ) {
					// odstraň
				}
			}

			// fáze 3 „pouze uvnitř“
			if( /* ve fázi 1 nebyl detekován průnik/dotek */ ) {
				ec.vlož(ep)
			}

			// fáze 4 „dotek a průnik“
			if(Edge e z ec)  {
				switch(e.getRelation(ep, p1, p2)) {
					case EI_NONE: break
					case EI_SLICE:
						uprostřed = // je p1 mezi e.start a e.end?
						if( uprostřed ) {
							e.konec = p1
						}

						if(/* tato hrana ještě nebyla vložena */){
							ec.vlož(ep)
						}

						if( uprostřed ) {
							ec.vlož(Edge(p1, e.původníKonec))
						}
						break
				}
			}

			// fáze 5: smaž hrany nulové délky
		}
	}
```

```
	/* následuje část 2: vytvoříme kolekci polygonů z jedné kolekce hran */
	if( ec.prazdné ) {
		return
	}

	Polygon p
	Vec3 point
	e = minuleE = dalšíIndex = 0
	while( ec.neníPrázdné ) {
		// přidáme aktuální bod
		if(p.neprázdné && ec[e].konec == point) {
			// .konec je dalším bodem
			p.přidejPokudNeobsahuje(ec[e].konec)
			point = ec[e].start
		} else {
			// hrana je obráceně
			p.přidejPokudNeobsahuje(ec[e].start)
			point = ec[e].end
		}

		/* pokud je 3+ bodů na stejné přímce,
		odstraň kromě prvního a posledního */

		ec.odstraň(e) // hranu z kolekce

		// přidali jsme start hrany, hledáme v kolekci konec hrany
		dalšíIndex = ec.najdiBod(point)
		if( /* nenalezeno */ ) {
			/* pokud je 3+ bodů na přímce mezi prvním a posledním
			bodem, odstraň kromě prvního a posledního */

			if( p.početBodů > 2 ) {
				vysledky.přidej(p)
			}

			p = novýPolygon
			e = 0 /* v algoritmu odstraňujeme,
				takže 0 je jiný bod než na začátku */
		} else {
			e = dalšíIndex
			/* v další iteraci přidáme tento bod */
		}


		minuleE = e
	}
}
```

Algoritmus je poměrně náročný, ale poskytuje dobré výsledky. Je nutno počítat s velkým množstvím kombinací situací. Celkově situaci zesložiťuje reprezentace desetinných čísel, u kterých se musí počítat, že výsledky některých funkcí nemusí být přesné. Zásadní problém je to u polárních souřadnic, které jsou na hodnoty velmi citlivé, a malá chyba může zcela změnit výsledky. Pomůže ale vhodná úprava dat, kdy konkrétní hodnoty (zvláště parametr vzdálenost) mohou být odlišné, ale při srovnávacích operacích půjde o ekvivalentní případy. Další úpravu bylo nutno zadat, že bod s polárními souřadnicemi (0,0,0) bude vracet stejný úhel s jakoukoli jinou souřadnicí.

6.4 Běh a vykreslování

6.4.1 Příprava dat

Před renderovací fází je u portálového algoritmu ještě provedena úprava dat listů BSP stromu. Z důvodů uvedených v části Datový návrh se polygony listů rozdělí na trojúhelníky. Pro vyšší efektivitu jsou pak hodnoty souřadnic všech polygonů uloženy do uzlu jako jedno dlouhé pole bez struktury.

Pro urychlení výpočtu detekce pozice pozorovatele je do uzlu vložena i jeho osově orientovaná obálka. V původním algoritmu se detekovalo, zda je skutečně pozorovatel uvnitř konvexní množiny (BSP strom není konstruován na úroveň jednotlivých polygonů a proto pouze klasické binární vyhledání nestačí) porovnání relace se všemi polygony uzlu. Toto lze zjednodušit rozlišením polygonů prostředí a polygonů statických objektů a porovnáváním pouze s prostředím, ještě rychlejší je ale porovnání šesti čísel, které definují obálku.

Dále jsou vloženy statické objekty. Jednotlivé polygony jsou vkládány do příslušné buňky podle toho, na kterých stranách dělících polygonů při průchodu od kořene stromem jsou nalézány. Algoritmus je založen na algoritmu popsaném v [5].

Nakonec jsou vloženy hodnoty barev pro každý pixel (aplikace pro demonstraci obarvuje každý úzel jinou barvou, s různými odstíny).

6.4.2 Hlavní fáze

Po inicializaci třídy Renderer je již připraven OpenGL kontext a SDL2 okno a sestaven glProgram z vertex a fragment shaderu. Je inicializována pozice pozorovatele. Spouští se hlavní smyčka. Pokud je program v bechmark módu, Renderer pouze spouští renderovací metodu. Jinak mezi spuštěními smyčky čeká na události, v případě definované trajektorie a prodlevy mezi snímky trajektorie jen do doby, dokud nenastane čas přejít na další krok této trajektorie.

Ve vykreslovací metodě může nastavit pozici pozorovatele podle aktuální trajektorie. Jinak nastaví příslušné matice (Projection, Model a View) a pokud je aktivní portálový algoritmus, tak je také předá ContentRendereru. U prezentačního módu musí udržovat dvě sady matic View, jednu pro zobrazení uživateli, druhou pro výpočet viditelnosti z pohledu pozorovatele ve scéně. Potom volá hlavní vykreslovací metodu ContentRenderer, ta vykreslí scénu, sám případně vykreslí pozorovatele ve scéně (s vynulování Depth Bufferu aby byl vždy nahoře) a uloží uplynulou dobu do objektu třídy Stats.

Nakonec provede nastavení další události trajektorie, případně čekání na další snímek.

Pokud má program nastaveno více spuštění (argument -n), je po ukončení hlavní fáze celý proces spuštěn znovu až do provedení **n** opakování nebo do ukončení programu uživatelem.

6.5 Detekce viditelnosti

6.5.1 SimpleRenderer

U triviální implementace renderu vlastní detekce viditelnosti neprobíhá. Tento renderer se pouze spoléhá na Z-culling poskytovaný OpenGL a jinak zadává k vykreslení celou scénu.

6.5.2 Portal culling

Detekci viditelnosti portály řeší třída PortalRenderer. Jako první krok vyhledá buňku, ve které se pozorovatel nachází. Pokud se nachází mimo jakoukoli buňku, je vykreslena celá scéna[1]. Jinak se zadá vykreslení aktuální buňky (tu vždy vidíme) a zavolá se její zpracování s předáním aktuálního stavu scény (snímek, pozice a směr pozorovatele) a projekcí portálu označenou jako celou obrazovku.

Ve zpracování buňky je nejdříve tato buňka uložena mezi již zpracované, aby se zabránilo zacyklení. Pak se prochází jednotlivé portály vedoucí z této buňky do jiných.

Portál nám vrátí buňku na opačné straně. Pokud je již zpracována, portál nezpracováváme. Potom se rychlým testem na úhly mezi body portálu, pozicí pozorovatele a pozicí pohledu se určí, zda se portál celou plochou nachází za pozorovatelem. Pokud ano, portál nezpracováváme. V následujícím kroku porovnáme vzdálenost středového bodu portálu a pozorovatele. V případě, že se portál nachází těsně před pozorovatelem, označíme ho za viditelný a jako průnik označíme celou rodičovskou projekci. Pokud ne, provedeme vytvoření projekce obdélníkové obálky následujícím algoritmem:

```
BoundingRectangle2D getRectangle(Portal portal) {
	bool found = false
	int minX = INT_MAX, minY = INT_MAX, maxX = 0, maxY = 0
	glm::vec4 viewport = glm::vec4(0, 0, šířkaObrazovky, výškaObrazovky)

	for(Vertex v z Polygonu portálu) {
		// získáme projektované souřadnice
		projected = project(v, View * Model, Projection, viewport)

		// pokud je z mimo clipping planes, zahodíme
		if( projected[2] < 0 || projected[2] > 1.1 ) {
			continue
		} else {
		// jinak nalezeno
			found = true
		}
	}
```

[1] Tady je třeba dávat pozor při tvorbě trajektorie, pohyb mimo scénu může znamenat velké snížení rychlosti a znehodnocení výsledků. Pokud je maximální počet vykreslených trojúhelníku ve výstupním souboru uvedený jako 100 %, může to znamenat tuto situaci.

```
	if( x < minX ) {
		minX = x
	}

	if( x > maxX ) {
		maxX = x
	}

	if( y < minY ) {
		minY = y
	}

	if( y > maxY ) {
		maxY = y
	}

	if( /* nenalezeno */ ) {
		return null
	}


	br.x = minX
	br.y = minY
	br.width = maxX - minX
	br.height = maxY - minY

	return br
}
```

Poté se vyhledá průsečík s předaným rodičovským obdélníkem, a pokud alespoň částečně zasahuje a průnik nemá nulovou velikost, je příslušná buňka označena za viditelnou a je rekurzivně zavoláno její zpracování, s průsečíkem jako novým rodičovským obdélníkem. Takto se postupně ořezává viewing frustrum, dokud je jakákoli buňka viditelná.

Na konci se do objektu Stats uloží počet skutečně vykreslených trojúhelníků.

6.5.3 Occlusion Queries

Při použití occlusion queries se odesílají dotazy grafické kartě a záznamy těchto dotazů se pro asynchronní zpracování udržují v kolekci PortalRendereru. Struktura záznamu je následující:

```
struct OCQuery {
	Portal* portal;
	BspNode* node;
	SceneState state;
	GLuint query;
	GLuint buffer;
	bool issued;
	unsigned id;
}
```

Jak je vidno, záznam udržuje také informaci o čase, pozici a směru pozorování v momentě odeslání. Pokud se výsledek kontroluje v čase, místě nebo směru odlišném více, než je nastavená hranice, výsledek se nepoužije.
Podle nastavení limitu viditelných pixelů se buď volí GL_SAMPLES_PASSED (pokud je třeba znát počet), nebo GL_ANY_SAMPLE_PASSED_CONSERVATIVE (pokud dostupno), jinak GL_ANY_SAMPLE_PASSED.

6.5.3.1 Varianta 1

V této variantě se před projekcí portálu odešle occlusion query na objekt vlastního portálu, s vypnutým zápisem do color a depth bufferu. Pokud byl ze stejné buňky na stejný portál dotaz již odeslán, dostatečně dávno a nebyl vyřízen, původní dotaz se zahodí a vytvoří nový. Informace ohledně dotazu a jeho okolností jsou ve formě struktury OCQuery uloženy do interní kolekce.

Následně se neblokujícím způsobem zkontrolují odeslané query, zda byly dokončeny. Výsledky dokončených query se uloží a tyto query se smažou.

V posledním kroku se zkontroluje, zda je aktuální buňka mezi označenými za neviditelné, v kladném případě se zpracování aktuální buňky ukončí. V záporném se pokračuje detekcí pomocí projekce, jak popsáno výše.

6.5.3.2 Varianta2

Tato varianta využívá fakt, že danou buňku v aktuálním kroku stejně vykreslovat budeme, takže se pouze vykreslení obalí požadavkem na uložení výsledku query. Dotaz tedy neodesílá hned, jen až pokud je portál označený za viditelný předchozím algoritmem. Provádí se ale kontrola výsledků předchozích dotazů, stejně jako ve variantě 1.

6.5.4 BufferedPortalRenderer

Tato třída poskytuje funkcionalitu efektivního vykreslování výsledků PortalRendereru. Aby se zabránilo odesílání stejných dat znovu a využila se temporální a lokační koherence, ukládá si tato třída strukturu

```
struct BufferEntry {
	BspNode* node;
	GLuint buffer;
	unsigned countUnused;
}
```

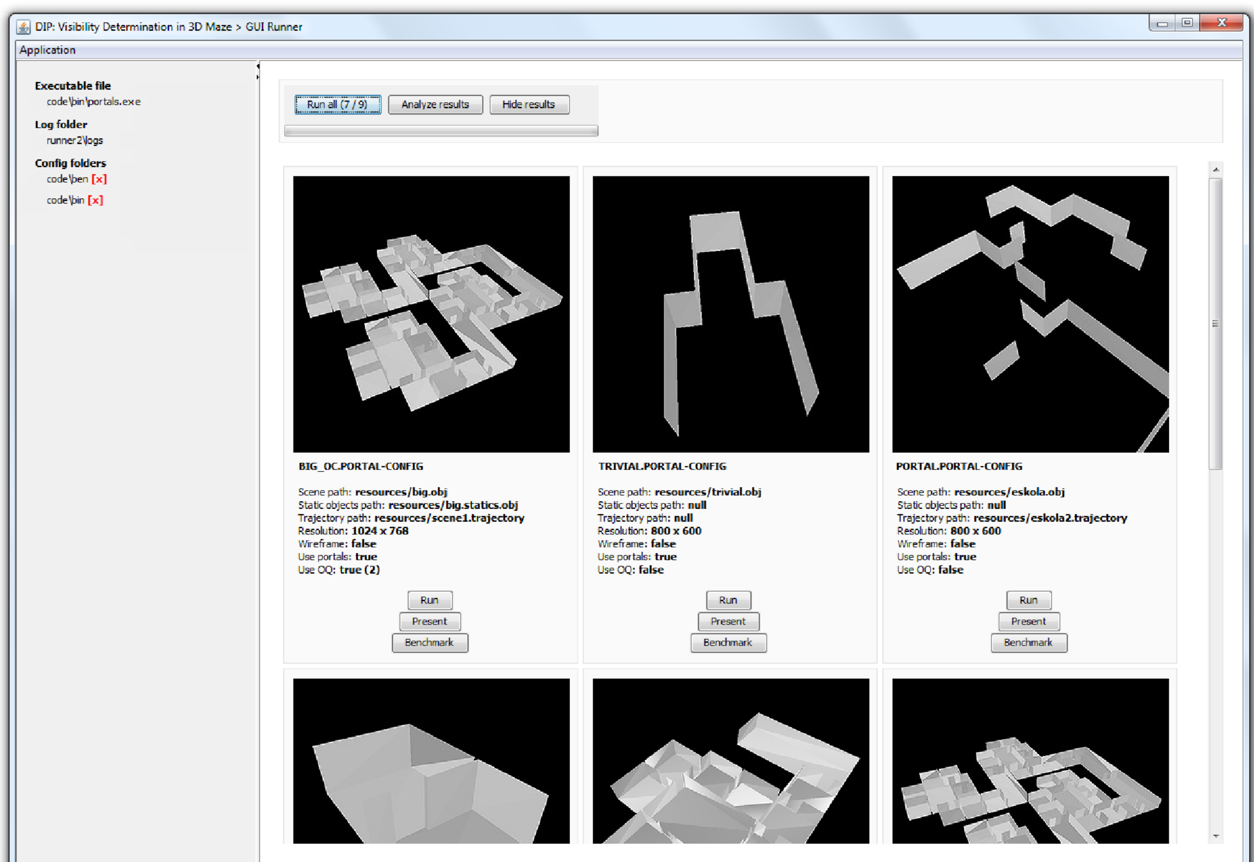
Při požadavku na buňku, která není v záznamech, BPR vytvoří na grafické kartě buffer a naplní ho příslušnými daty, následně zadá vykreslení těchto dat a uloží si tento záznam do paměti. V dalším přístupu už pak jen volá vykreslení z daného bufferu. Jelikož je tento případ nastává velmi často, vykreslení se tím značně zrychluje.

Na konci celého vykreslovacího procesu projde všechny záznamy a těm, které nebyly renderovány, je zvýšen parametr countUnused o 1. Pokud tímto překročí nastavený limit, je záznam i připojený buffer smazán. Záznamům s vykreslenými buňkami je nastaven countUnused na 0.

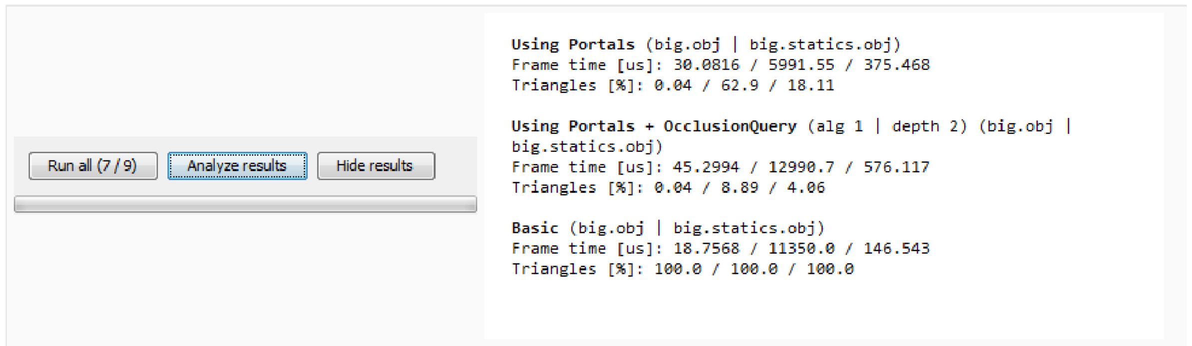
6.6 Uživatelské rozhraní

Jako doplněk bylo k práci vytvořeno uživatelské rozhraní v jazyce Java, které umožňuje spouštět předdefinované konfigurace. Jde o samostatný program, je třeba zadat cestu k hlavnímu programu. Dále složku pro výstupní soubory a složky, které chceme prohledávat pro nalezení konfigurací.

Aplikace zobrazí všechny konfigurace s náhledem (ručně vytvořené JPG soubory) a základními parametry. Jednotlivé konfigurace je možno spouštět v klasickém režimu, prezentačním nebo benchmarkovém. Lze také spustit všechny konfigurace s trajektorií v benchmarkovém módu s čtyřnásobným opakováním (akce **Run all**). Výstupy pro jednotlivé konfigurace a scény lze nechat shromáždit a vypsat.



Obrázek 11. Snímek obrazovky spouštěcího a analyzačního programu.



Obrázek 12. Detail shrnutí výsledků běhů programu s různými algoritmy a nastaveními. Pro konkrétní kombinace scény, statických objektů a trajektorie a nastavení algoritmu jsou vypsány výsledky doby vykreslování a množství vykreslovaných dat. Jednotka µs je vypsána jako us.

7 Vyhodnocení

Za běhu programu byly shromažďovány údaje o počtu vykreslených trojúhelníků a době detekce a celkové době vykreslení. Do výstupních souborů se pak zapisuje min/max/průměr vykreslených trojúhelníků a min/max/průměr celkového renderování. Vizuálně byla kontrolována kvalita výsledku. Výsledky měření v rámci této práce přinesly několik poznatků.

Jednak se ukazuje, že výkon dnešních karet je enormní a proto je třeba komplexní scény s texturami, výpočty v shaderech a velikým množstvím polygonů, aby kladl na akcelerátor nápor. Schematické scény zaměřené primárně na množství geometrie nejsou velkým problémem a je možné je zpracovat v zanedbatelném čase i bez sofistikovaných přístupů k viditelnosti. Tím spíše pokud dokáží plně využít a minimalizovat přenos dat a přepínání bufferů.

Naproti tomu režie occlusion queries je i při asynchronním přístupu překvapivě vysoká. Tento poznatek se lehce podcení, ale už jen samotné odeslání query zabere nemalý čas. Pokud chceme renderovat mimo color a depth buffer, vypínání a zapínání zápisu také způsobí podstatné zpomalení. Je také třeba je použít jen v situacích s vysokou koherencí, v opačných případech použití výsledků může způsobit nepřesnosti v obraze, což bývá obvykle nepřijatelné.

Výsledky jsou součtem dvaceti běhů na konfiguraci pro scénu big.obj.

scéna	trojúhelníků celkem	min/max/prům trojúhelníků [%]	min/max/prům času [µs]	prům. čas detekce viditelnosti [µs]
triviální	99464	100 / 100 / 100	18.8 / 11350 / 146.5	-
portály	102170	0.04 / 62.9 / 18.11	30 / 5991.5 / 379.5	0.75
+ OQ v1	99524	0.04 / 18.89 / 9.06	45.3 / 12990.7 / 576.1	302.4
+ OQ v2	102170	0.04 / 25.78 / 11.2	42.2 / 7590.9 / 465.4	50.9

Tabulka 1. Výsledky měření běhů scény s různými konfiguracemi

Portálová implementace poskytuje kvalitní detekci viditelnosti, vrací výsledky adekvátní viditelnosti. U occlusion query je pro adekvátní výsledek třeba vhodně nastavit prahy koherence, aby dávaly dobré výsledky. Při rychlých změnách kamery ale dojde k zahazování výsledků.

8 Závěr

V předcházejících kapitolách bylo uvedeno krátké představení OpenGL, prezentovány vybrané významné algoritmy pro stanovení viditelnosti a eliminaci neviditelných elementů. Byl popsán obecný návrh řešení. Ten se primárně zaměřuje na dělení a hierarchizaci scény pomocí BSP stromu a aplikaci metody portal culling, se kterou spolu s funkcionalitou occlusion queries bude sloužit ke stanovení viditelnosti.

Tento algoritmus byl následně implementován v podobě aplikace v jazyce C++, demonstrující algoritmus na průchodu zadanou scénou buď po předpřipravené trajektorii, nebo volně pomocí klávesnice a myši. Programu je možno zadat použitý algoritmus, data scény, data statických objektů, trajektorii. Jako doplněk byla vytvořena aplikace s grafickým uživatelským rozhraním v jazyce Java pro spouštění hlavního programu a zobrazení výsledků.

Řešení viditelnosti pomocí portálů při předzpracování scény do struktury BSP stromu může být velmi rychlé. Při posuzování vlivu je třeba mít na paměti celý proces vykreslení scény, kdy je nejen třeba stanovit viditelné části, ale také je efektivně předat aplikaci, což má samo o sobě také vliv na výkon. Při triviální práci s vertexy a maximálním využití retained módu specifikace dat do určitého množství může být překvapivě výhodnější použít naivní přístup, což ale není případ například realistických prezentací.

Mechanismus occlusion query s sebou nese řadu úskalí a jeho režie je překvapivě vysoká, pro efektivní využití je tedy třeba maximálně využívat časové a prostorové koherence. Dává také přesnější výsledky u dynamických scén, které implementace pouze s pomocí portálů nedokáže brát v potaz.

Dále bylo předvedeno, že samotná detekce viditelných portálů projekcí a průnikem je natolik rychlá (a přesto dostatečně spolehlivá), že dokáže konkurovat předpočítané potenciální sadě viditelnosti a můžeme tak vynechat jeden z náročných kroků předzpracování.

Pro další vývoj bych se soustředil na výrazně důkladnější měření u složitějších a komplexnějších scén a na větší zaměření na adresování úskalí occlusion queries pro různé situace po straně návrhu i implementace, aby bylo lépe možno zhodnotit jejich využitelnost ve vybraných podmínkách.

Literatura

[1] AKENINE-MOLLER, TOMAS – MOLLER, TOMAS – HAINES, Eric. Real-time rendering. AK Peters, Ltd., 2008.

[2] TELLER, SETH J. Visibility computations in densely occluded polyhedral environments. 1992.

[3] TELLER, SETH J. – SÉQUIN, CARLO H. Visibility preprocessing for interactive walkthroughs. In: ACM SIGGRAPH Computer Graphics. ACM, 1991. p. 61-70

[4] ŽÁRA, JIŘÍ, et al. Moderní počítačová grafika, druhé, přepracované a rozšířené vydání, kompletní průvodce metodami 2D a 3D grafiky. 2004.

[5] RANTA-ESKOLA, SAMUEL – OLOFSSON, Erik. Binary space partioning trees and polygon removal in real time 3d rendering. Uppsala master's thesis in computing science, 2001.

[6] LAAKSO, MIKKO. Potentially visible set (pvs). Helsinki university of technology, 2003.

[7] LUEBKE, DAVID – GEORGES, CHRIS. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In: Proceedings of the 1995 symposium on Interactive 3D graphics. ACM, 1995. p. 105-ff.

[8] PHARR, MATT – FERNANDO, Randima. Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation. Addison-Wesley Professional, 2005.

[9] SEGAL, MARK – AKELEY, Kurt. The OpenGL Graphics System: A Specification. 2010.

[10] AIREY, JOHN M. Increasing update rates in the building walkthrough system with automatic model-space subdivision and potentially visible set calculations. NORTH CAROLINA UNIV AT CHAPEL HILL DEPT OF COMPUTER SCIENCE, 1990.

[11] NIRENSTEIN, SHAUN, et al. Rendering Techniques 2002: Proceedings of the 13th Eurographics Workshop on Rendering. Springer-Verlag. p. 191-202. Pisa. Itálie.

[12] FUCHS, HENRY – KEDEM, ZVI M. – NAYLOR, BRUCE F. On visible surface generation by a priori tree structures. In: ACM Siggraph Computer Graphics. ACM, 1980. p. 124-133.

[13] ULRICH, THATCHER. Loose octrees. Game Programming Gems, 2000, 1: 434-442.

[14] AILA, TIMO – MIETTINEN, VILLE. dpvs: An occlusion culling system for massive dynamic environments. Computer Graphics and Applications, IEEE, 2004, 24.2: 86-97.

[15] Silicone Graphics Inc. OpenGL Overview. http://web.archive.org/web/20090304111652/http://www.sgi.com/products/software/opengl/overview.html

[16] OpenGL. Wiki. https://www.opengl.org/wiki [cit. 2013-12-26]

[17] OpenGL. https://www.opengl.org [cit. 2013-12-26]

[18] JONES, CLIFF B. A new approach to the 'hidden line'problem. The Computer Journal, 1971, 14.3: 232-237.

[19] JAMES, ADAM. Binary space partitioning for accelerated hidden surface removal and rendering of static environments. 1999. PhD Thesis. University of East Anglia.

[20] WADE, BRETTON. BSP Tree Frequently Asked Questions (FAQ). ftp://ftp.sgi.com/other/bspfaq/faq/bspfaq.html [cit. 2014-05-26]

[21] KOLÁŘ, DUŠAN. Pokročilé databázové systémy (prostorové databáze). Ústav informačních systémů. Fakulta informačních technologií. VUT Brno. 2006.

[22] REDDY, MARTIN. The Graphics File Formats Page. *Obj files (.obj).* http://www.martinreddy.net/gfx/3d/OBJ.spec [cit. 2014-05-26]

Seznam příloh

A. Manuál instalace
B. Manuál konfigurace a ovládání
C. Obsah CD
D. Popis přiložených scén a konfigurací
E. Vývojová a testovací prostředí a software
F. Poznámky ke zdrojovým datům

Příloha A

Manuál instalace

1. Hlavní program

Program napsaný v jazyce C++, ve zdrojových souborech (viz Příloha C) se nachází soubor Makefile. Definuje následující příkazy:

- make
  provede překlad programu
- make clean
  smaže soubory objektového kódu a spustitelný soubor programu
- make help
  vypíše nápovědu (nevyžaduje přeložený program)
- make run[1-6]
  spustí program s předem definovanou konfigurací
- make drrun[1-6]
  spustí kontrolu programu (s danou konfigurací) programem Dr. Memory (MS Windows) nebo valgrind (Linux)

Překlad byl testován na systémech zmíněných v Příloze E, systémy Windows i Linux. Spuštění bylo testováno na systému Windows. Pro úspěšnou kompilaci je třeba mít instalované a konfigurované knihovny:

- GLUT (např. freeglut, použita verze 2.8.1)
- GLEW (použita verze 1.10)
- SDL (použita verze 2.0.3)

Pro spuštění je třeba verze OpenGL 4.4, resp. podpora occlusion queries (pokud spouštíme povolené) ve standardních funkcionalitách a GLSL verze 3.3 (tj. OpenGL verze 3.3, nutno vždy).

Cesty k hlavičkovým souborům a knihovnám, používané překladačem, lze případně v Makefile upravit na řádcích 10 a 13 (Windows) nebo 20 a 23 (Linux).

2. Uživatelské rozhraní

Program uživatelského rozhraní je napsán v Javě, verze JDK 1.7. Ve zdrojových souborech (viz Příloha C) se nachází soubor Makefile. Definuje příkazy **make run**, který spustí rozhraní.

Příloha B

Manuál konfigurace a ovládání

1. Hlavní program

a. Spuštění

Program přijímá argumenty příkazové řádky, jako jediný povinný argument potom cestu ke konfiguračnímu souboru, kde je možné další nastavení.

-c	řetězec	cesta ke konfiguračnímu souboru, nutný parametr
-n	kladné celé číslo	(volitelné) počet spuštění akční části programu (tj. po předzpracování scény)
-l	řetězec	cesta k výstupnímu souboru
-b	-	aktivuje benchmark mód (nelze kombinovat s -p) tj. vypíná se čekání mezi kroky trajektorie a reagování na uživatelské vstupy
-p	-	aktivuje prezentační mód (nelze kombinovat s -b) pohled uživatele je oddělen od kamery entity procházející scénu a je tak možno sledovat průběh z nadhledu
-h	-	zobrazí nápovědu

Při případné úpravě Makefile souboru je možné dopsat tyto argumenty buď jednotlivě k cílům, nebo globálně na řádcích 40 až 43.

Konfigurační soubor je textový soubor o pevném formátu. Každý řádek obsahuje položku ve formátu:

```
KLÍČ[dvojtečka][mezera]HODNOTA[konec řádku]
```

Všechny klíče kromě staticObjectsPath a trajectoryPath musí být přítomny. Pořadí je také nutno dodržet:

scenePath	řetězec	cesta k souboru .obj s definicí scény
staticObjectsPath	řetězec	(volitelné) cesta k souboru .obj s definicí statických objektů
trajectoryPath	řetězec	(volitelné) cesta k souboru s trajektorií
ambientIllumination	desetinné číslo	globální úprava osvětlení

width	kladné celé číslo	šířka vykreslované plochy
height	kladné celé číslo	výška vykreslované plochy
wireframe	0 nebo 1	zda kreslit celé plochy, nebo pouze hrany
portals	0 nebo 1	zda použít algoritmus portálů, nebo triviální
occlusionQuery	0, 1 nebo 2	zda použít doplňkový algoritmus používající Occlusion Queries. Portals musí být aktivováno.
occlusionQueryDepth	nezáporné celé číslo	od jaké úrovně vzdálenosti od buňky pozorovatele aktivovat occlusion query algoritmus
exitAfterTrajectory	0 nebo 1	zda ukončit aplikaci poté, co trajektorie dorazila do posledního bodu
trajectoryDelay	kladné celé číslo	prodleva mezi přechody mezi kroky trajektorie. V Benchmark módu je vždy 0.

Zpracovaná konfigurace je vždy před vykreslováním zapsána do výstupního souboru (definovaného přepínačem -l), takže lze zkontrolovat její zadání.

b. Ovládání

Pokud je nastavena trajektorie, kamera následuje pozorovatele, při použití některého z následujících ovládacích prvků se toto sledování přeruší (dokud není uživatelem znovu aktivováno). To neplatí pouze u benchmark módu, který nereaguje na žádné uživatelské vstupy.

Šipka vlevo/vpravo		otočení o 5º kolem osy Z
	CTRL	otočení o 22.5º
	SHIFT	otočení o 90º
Šipka nahoru/dolů		přesun o 0.09 jednotek vpřed/vzad
	CTRL	přesun o 0.2 jednotek
	SHIFT	přesun o 1.6 jednotek
Pohyb myší	**Levé tlačítko**	otáčení kolem os Y a Z
Klávesa S		zapne/vypne nahrávání trajektorie
Klávesa F		zapne/vypne volný pohyb při prezentačním módu
Klávesa ESC		ukončí aplikaci (i při nastavení více běhů, ukončí se zcela)

2. Uživatelské rozhraní

Rozhraní slouží ke spouštění hlavního programu s vybranou konfigurací, hromadně, či analýzu výsledků. Po spuštění je třeba nastavit (přes menu položky Application) následující:

- **Set executable**
  cesta ke spustitelnému souboru (např. zdroj_program/bin/portals.exe)
- **Log folder**
  složka, do které se budou ukládat výsledky
- **Add scan folder**
  přidá do seznamu složku, kde se budou vyhledávat konfigurační soubory .portal-config

Aplikace kontroluje obsah složek s konfiguracemi pouze při změně nastavení, pokud byla po přidání cesty konfigurace přidána či odebrána, je nutno použít akci **Refresh** z menu aplikace. Program očekává příponu souborů .portal-config, jinak soubory nenačte.
Hromadně jsou spustitelné pouze konfigurace s nastavenou trajektorií. První číslo v tlačítku **Run all** udává počet těchto konfigurací, lomeno počet celkových konfigurací. Hromadné spuštění je voláno s argumenty **benchmarkMode** a **numberOfRuns=4** (v aktuální verzi není konfigurovatelné).

Příloha C

Obsah CD

dp.pdf	Technická zpráva diplomové práce ve formátu PDF
zdroj_program/	Zdrojové soubory hlavního programu
bin/	Složka, do které se generuje spustitelný soubor a výstup programu
config/	Složka s předdefinovanými konfiguracemi
obj/	Dočasné soubory pro linkovací program
libraries/	Hlavičkové soubory třetích stran
resources/	Zdrojová data programu – soubory s definicí scény a trajektorie
shaders/	Shadery programu
zdroj_gui/	Soubory uživatelského rozhraní
dist/	Složka s interpretovatelným souborem a uloženou konfigurací programu
logs/	Složka s uloženými výstupy spuštění (možno změnit v programu)
src/	Zdrojové soubory uživatelského rozhraní
media/	Složka se snímky obrazovky programu a videa prezentující běh programu

Příloha D

Popis přiložených scén a konfigurací

Scény

big.obj	Větší scéna složená z propojených krychlových místností, celkem 2000 polygonů.
scene1.obj	Menší scéna složená z propojených krychlových místností
cross.obj	Čtyři jednoduché místnosti spojené s centrální místností průchody
eskola.obj	Několik složitějších místností, bez podlahy a střechy. Vychází z nákresu v [5]
two_room.obj	Dvě místnosti propojené průchodem
trivial.obj	Dvě přímo spojené místnosti, jeden portál
single_room.obj	Jediná místnost, žádný portál

Statické objekty

big.statics.obj	Mnoho kulovitých předmětů s mnoha polygony, celkem 196800 polygonů.
scene1.statics.obj	Kulovité předměty, celkem 7680 polygonů.

Trajektorie

scene1.trajectory	Průchod celou **scene1.obj**, průchod částí **big.obj**
cross.trajectory	Průchod celou **cross.obj**
eskola.trajectory	Průchod celou **eskola.obj**

Příloha E

Vývojová a testovací prostředí a software

1. OS: MS Windows 7 64-bit SP1
   Procesor: Intel Core2 Duo 8500 3.16 GHz
   RAM: 4.0 GB 800 MHz 5-5-5-12 DDR2
   Grafický akcelerátor: NVIDIA GeForce GTX 260
2. OS: Ubuntu 12.04.4 LTS
   Procesor: Intel Core2 Duo 8500 3.16 GHz
   RAM: 4.0 GB 800 MHz 5-5-5-12 DDR2
   Grafický akcelerátor: NVIDIA GeForce GTX 260
3. OS: MS Windows 7 64-bit SP1
   Procesor: Intel Pentium G850 2.9 GHz
   RAM: 4.0 GB 1333MHz CL9 DDR3
   Grafický akcelerátor: NVIDIA GTS450 512MB DDR5

Použitý software

- Netbeans IDE 8.0
- Sublime Text 2
- DrMemory
- Autodesk 3DS Max 2014 trial s gw::ObjIO exportérem
- Adobe Photoshop CS4
- Debut Video Capture Software 1.94
- XMedia Recode 3.1
- Dropbox
- MS Word 2007 student
- Shapeshifter

Příloha F

Poznámky ke zdrojovým datům

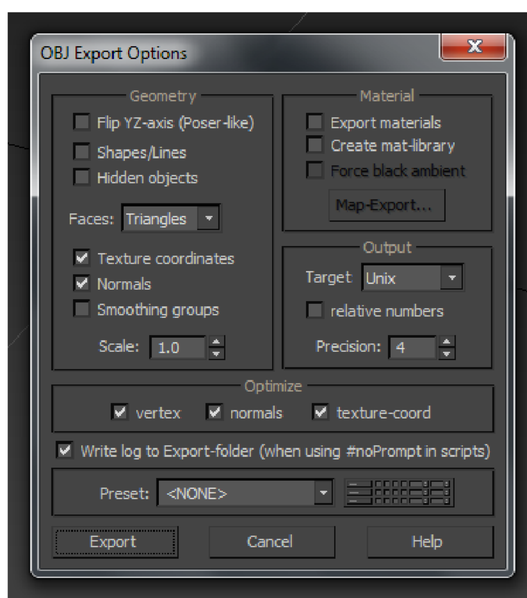
Zdrojová data jsou ve formátu Wavefront .obj. Podrobný popis formátu je k dispozici v [22]. Podporovány a vyžadovány jsou údaje pozic a normál vertexů. Texturovací souřadnice se mohou v souboru vyskytovat. Polygony mohou být triangulizované, ale není to vyžadované. Stejně tak optimalizace zdrojových dat je možná.

Pokud má pozorovatel ve scéně reprezentovat člověka v interiéru klasické budovy, pak jeden metr odpovídá zhruba 1.0 délky (což je pro běžné nastavení kamery editorů poměrně málo, je nutno přiblížit).

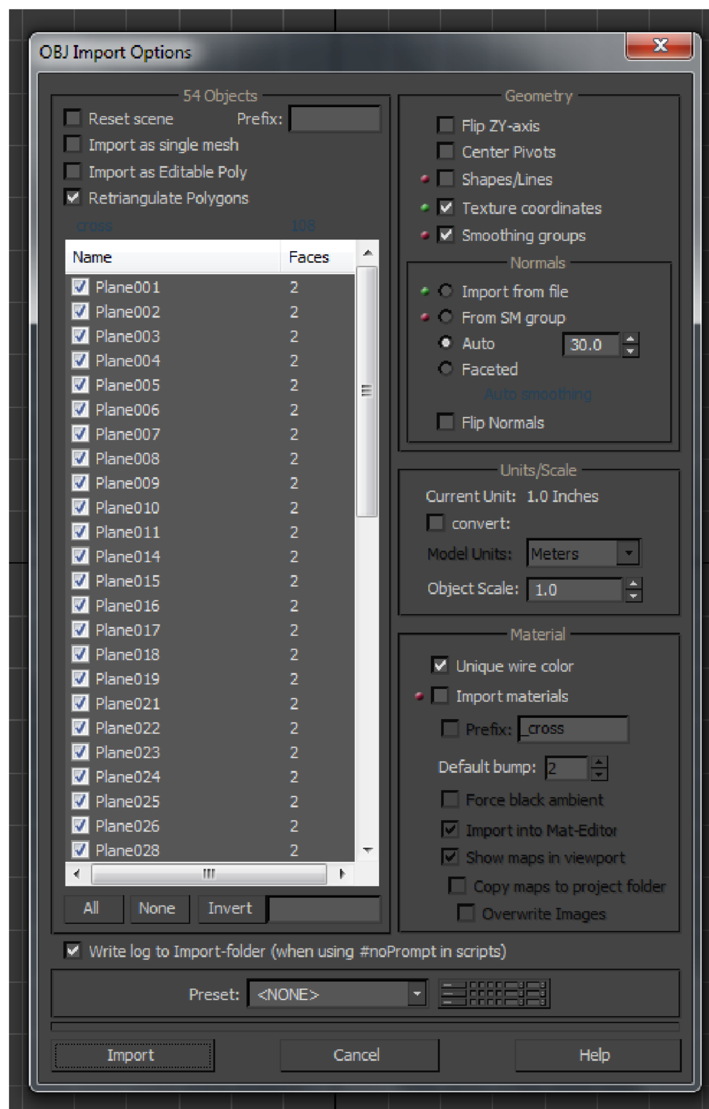
Počítá se s neprohozením Y a Z souřadnic (minimálně z pohledu editoru 3DS Max), pokud je tedy model převrácený, měla by být možnost u exportu zaměnit (nebo zrušit záměnu) těchto souřadnic.

Při nesprávném BSP rozdělení modelu nebo špatném vložení statických objektů je třeba zkontrolovat správnost normál v exportovaném souboru. V některých případech je normála souběžná s polygonem, nikoli kolmá. Pokud to editor umožňuje, většinou k opravě vede importovat chybný .obj soubor zpět do editoru s tím, že nenačítáme normály z něj, ale zvolíme automatické stanovení normál. Po importu tato data exportuje zpět do daného souboru. V ten moment by měly být normály opraveny, někdy bylo třeba postup opakovat.

Následující obrázky představují náhled nastavení exportu a opravného importu.



Obrázek 13. Doporučené nastavení exportu zdrojových dat.



Obrázek 14. Doporučené nastavení importu pro opravu zdrojových dat.