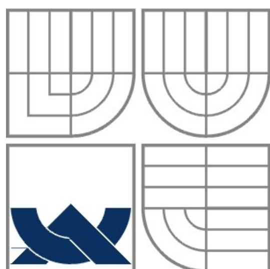
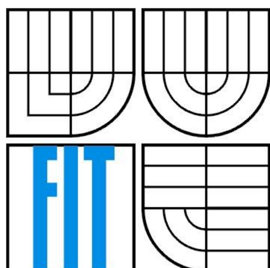


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VÝUKOVÝ PROGRAM PRO DEMONSTRACI ŘEŠENÍ VIDITELNOSTI 3D OBJEKTŮ

EDUCATION COMPUTER PROGRAM FOR DEMONSTRATION OF 3D MODELS VISIBILITY ALGORITHMS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE DALIBOR JUŘICA
AUTHOR

VEDOUCÍ PRÁCE ING. PŘEMYSL KRŠEK, Ph. D.
SUPERVISOR

BRNO 2008

Abstrakt

Cílem tohoto projektu není vytvoření aplikace schopné řešení viditelnosti objektů 3D scény, ale aplikace, jenž dokáže principy řešení viditelnosti 3D objektů zobrazit. Demonstrace budou navrženy tak, aby byly i pro neznalého uživatele zcela zřetelné, nedělaly uživateli problém principy demonstrovaných algoritmů pochopit a následující dostudování látky na základě znalostí z demonstrací bylo pro uživatele plně pochopitelné.

Klíčová slova

demonstrace, viditelnost, Malířův algoritmus, hloubkové třídění, Z-Buffer, Ray-Casting, výukový program, vyřazení odvrácených stěn, renderování

Abstract

The aim of this project isn't build the application, whick is able to hande visibility problem of three dimensional scenes, but application, that can demonstrate idea of this visibility problem. Separate scenes will be designed in way, that even the amateur will not have any problem to understand the principle of demonstrate algorithms. Subsequent graduation of the visibility problem will be then fully comprehensible.

Keywords

demonstration, visibility, Painter's algorithm, Depth sort algorithm, Z-Buffer, Ray-Casting, education computer program, back-face culling, rendering

Citace

Juřica Dalibor: Výukový program pro demonstraci řešení viditelnosti 3D objektů.
Brno, 2008, bakalářská práce, FIT VUT v Brně

Výukový program pro demonstraci řešení viditelnosti 3D objektů.

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Přemysla Krška.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Jméno Příjmení
Datum

Poděkování

Chtěl bych poděkovat panu Ing. Přemyslu Krškovi za profesionální přístup při vedení mé bakalářské práce, taktéž za motivaci a poskytnutí odborných rad.

© Dalibor Juřica, 2008.
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

Obsah....1
Úvod....3
1 Řešení viditelnosti....4
1.1 Vytváření prostorových objektů....4
1.2 Základní myšlenky řešení viditelnosti....5
1.2.1 Odstranění neviditelných ploch....5
1.2.2 Ohodnocení hran objektů....6
1.3 Dělení algoritmů dle výpočtové složitosti....7
1.3.1 Objektově orientované algoritmy....7
1.3.2 Obrazově orientované algoritmy....7
1.3.3 Kombinované algoritmy....8
1.4 Vektorové algoritmy....8
1.4.1 Robertsův algoritmus....8
1.4.2 Plovoucí horizont....9
1.5 Rastrové algoritmy....11
1.5.1 Malířův algoritmus....11
1.5.2 Warnockův algoritmus dělení obrazu....14
1.5.3 Z-Buffer....15
1.5.4 Ray-Casting....17
2 Návrh demonstrační aplikace....19
2.1 Definice pojmu „demonstrační aplikace“....19
2.2 Základní rozhraní aplikace....20
2.2.1 Výsledný pohled....21
2.2.2 Pracovní pohled....21
2.2.3 Časový proklad mezi kroky algoritmu....22
2.2.4 Krokování algoritmu....23
2.2.5 Řazení polygonů....23
2.3 Polygon scény....24
2.3.1 Barevné odlišení....24
2.3.2 Znázornění vykresleného/nevykresleného polygonu....25
2.3.3 Znázornění pořadí polygonů ve scéně....25
2.4 Návrh demonstrací....25
2.4.1 Demonstrace řešení viditelnosti polygonů....25
2.4.2 Demonstrace Malířova algoritmu....26

2.4.3 Demonstrace Z-Bufferu....27
2.4.4 Demonstrace Ray-Castingu....29
3 Implementace....31
3.1 3D API Java 3D....31
3.2 UML návrh aplikace....31
3.3 Základní struktura scén....32
3.3.1 Scenegraph pro výsledný pohled....33
3.3.2 Scenegraph pro pracovní pohled....33
3.3.3 Implementace základní struktury demonstrací....35
3.4 Implementace zvolených algoritmů....37
3.4.1 Vyřazení odvrácených stěn....37
3.4.2 Malířův algoritmus....39
3.4.3 Z-Buffer....40
3.4.4 Ray-Casting....42
4 Dosažené výsledky....46
5 Závěr....49
Literatura....50
Seznam příloh....51

Úvod

Obor počítačová grafika se bezesporu v poslední dekádě stal velkým fenoménem a jedním ze stěžejních oborů informační technologie. Jeho nástup odstartoval zejména velký boom v oblasti počítačových her. Využití počítačové grafiky si však našlo své místo i v dalších oblastech softwarových produktů. Tím bylo umožněno vytvořit mnohem více atraktivnější, přehlednější, efektivnější uživatelská prostředí.

Obraz takové grafické aplikace se skládá z komplexní prostorové scény. Aby však tato scéna mohla být zobrazena na monitor, musí podstoupit složitý postup zpracování. V postupu zpracování scény dochází ke krokům jako ořezávaní obrazu, rasterizace, projekce, osvětlení scény, a taktéž i k řešení viditelnosti objektů.

Viditelnost objektů se začala řešit zejména s nástupem 3D grafiky. Scéna je uložena v datové struktuře, která však nenese žádné informace o viditelnosti jednotlivých objektů. Ukládá pouze údaje o tvaru objektů, vlastnostech materiálů, texturách, transformacích objektů atd. Aby z datového modelu vznikl přirozený obraz scény, je potřeba vyřešit viditelnost jednotlivých objektů. To znamená vynalézt prostředky, které se o řešení viditelnosti objektů scény postarají a vykreslí tak scénu do přirozené podoby.

V reálném světě takové procesy probíhají přirozeně, což je dáno fyzikálními vlastnostmi materiálů objektů, nacházející se v prostředí, jenž pozorujeme. Ve virtuálním světě sice scéna taky vypadá tak, jako kdyby vše probíhalo přirozeně a samovolně, zdání však klame. Řešení viditelnosti totiž probíhá v tichosti a uživateli je poskytnut již čistý výsledek scény obrobené procedurou pro řešení viditelnosti. Výsledkem může být například nový snímek dynamické scény, nebo jediný vyrenderovaný obraz. Ve finálním bodě uživatel způsob zpracování scény nepociťuje a ani o něm vědět nepotřebuje.

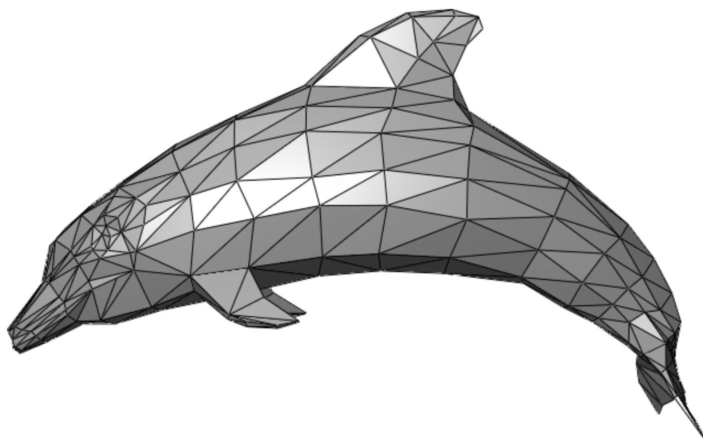
Kdyby se však chtěl dozvědět, jak se scéna vykresluje, jeho jedinou možností je si o této problematice přečíst ve specializované literatuře. Nemá ale žádnou možnost pozorovat řešení viditelnosti přímo na scéně. Cílem projektu je tak poskytnout uživateli takovou aplikaci, ve které bude možné sledovat principy řešeni viditelnosti. Uživatel tedy bude moci pozorovat na samotné scéně, jak je zpracováváno překrývání objektů scény, jakých prostředků je k tomu využíváno a na závěr i co je výsledným obrazem scény. Bude kladen důraz na to, aby bylo uživateli nastíněno chování vybraných algoritmů, které problematiku viditelnosti objektů prostorové scény řeší. Uživatel by měl být obeznámen i se základní myšlenkou viditelnosti, tedy jak dochází k vyřazení odvrácených ploch scény.

1 Řešení viditelnosti

V následujících bodech bude prováděn sběr důležitých informací o současných algoritmech řešící viditelnost 3D objektů. V každém algoritmu bude vytyčena základní myšlenka, na níž je algoritmus postaven. Popsány zde budou základní druhy členění algoritmů, a to z pohledu způsobu průchodu scénou i z pohledu, výsledku algoritmu. Důraz bude kladen na sběr informací zejména o rasterových algoritmech, které jsou masově využívány velkou částí současných aplikací pracující s prostorovou grafikou. Algoritmy vektorové budou v dokumentaci taktéž rozebrány, jelikož i ty tvoří nedílnou součást mnohých mainstreamových aplikací.

1.1 Vytváření prostorových objektů

Aby mohly počítače sloužit k vizualizaci prostorových scén, musí existovat prostředky pro vytvoření 3D grafické reprezentace objektů. Pokud se porozhlédneme kolem sebe, je tak možno spatřit kvantum objektů, těmi můžou být mobilní telefon, kalkulačka, talíř, křeček, cokoliv. Takové objekty jsou seskládány z atomů (neživé objekty), popř. z buněk (živé objekty). Tvar objektů je pak výsledkem struktury seskládaní stavebními prvky, které objekt tvoří. Když objekt chceme graficky ztvárnit, nezbývá nám jiná možnost, než jej vymodelovat. Asi těžko by jsme libovolný objekt modelovali na úrovni buněk, resp. atomů. Model objektu je tak oproti jeho reálné verzi značně zjednodušen. Modelování fyzikálních a jiných vlastností teď nechme stranou, jelikož podstatný je pro nás pouze tvar vymodelovaného objektu. Při vytváření tvaru objektu je objekt vykreslen za pomocí mnoha malých trojúhelníků, resp. čtverců. V počítačové grafice tak říkáme, že je objekt seskládán z *polygonů*. Příklad polygonálního modelu objektu můžete vidět na obrázku 1.1.



obr 1.1: Příklad objektu složeného z polygonů.

1.2 Základní myšlenky řešení viditelnosti

Vytvoření přirozeného obrazu ze scény je poměrně zdlouhavým procesem zpracování počítačové grafiky, jelikož povětšinou pracuje nad komplexními scénami složené klidně i z desetitisíců polygonů. Navíc scéna může být nepřetržitě vykreslována, proto je důležitým bodem při řešení viditelnosti 3D scény zpracovávat jenom to, co je v danou chvíli opravdu důležité. Opačný případ by vedl ke zbytečnému plýtvání výpočetního výkonu procesoru počítače resp. grafického procesoru.

Grafické scény jsou tvořeny z jednotlivých objektů a ty musí splňovat určité požadavky. Jedním z požadavků na objekt je regulérnost. Jinými slovy, aby objekt nebyl vytvořený takovou reprezentací, kterou by v reálu nebylo možno realizovat. Regulérní objekty mají specifické vlastnosti, na kterých lze stavět základní myšlenky v problematice řešení viditelnosti objektů. Základní myšlenky budou rozebrány v následujících dvou kapitolách.

1.2.1 Odstranění neviditelných ploch

Pokud si jako objekt přestavíme libovolné těleso z reálného světa, víme že vždycky na takovém tělesu existují části, jenž z daného místa nemůžeme vidět, jelikož jsou od nás odvráceny. Budeme-li chtít odvrácené části pozorovat, musíme se přemístit na místo, ze kterého je toto možné. Avšak tím že těleso již pozorujeme z jiné pozice, zakryjí se nám ty části, které jsme předtím viděli.

Tohoto faktu využívá myšlenka odstranění neviditelných ploch, známá taktéž pod názvem *back-face culling*. Algoritmy pro řešení viditelnosti zpracovávají postupně všechny polygony nehledě na to, jsou-li z daného místa pozorovatele viditelné či ne. Uvážíme-li, že odlehlých polygonů je ve scéně zhruba polovina, v případě scény složené z tisíce polygonů je jich zbytečně zpracováváno přibližně pět set. Z čehož vyplývá, že scéna před samotným započetím řešení viditelnosti musí být nejdříve předzpracována tak, aby algoritmy zpracovávaly pouze již ty polygony, jenž z daného místa pozorovatele má smysl zpracovávat. Polygony odlehlé od pozorovatele musí být ze scény nejdříve vyloučeny, čímž dojde k zefektivnění práce algoritmů, jelikož ty pracují s menším počtem polygonů než v případě nepředzpracování scény.

Zjištění přilehlých/odlehlých polygonů se provádí na základě společného vztahu normálových vektorů polygonů s vektorem definující směr pozorovatele na scénu. Normálový vektor je přímka kolmá k danému n-1 dimenzionálním podprostoru v n-dimenzionálním prostoru. V případě tří rozměrného prostoru je to vektoru kolmý na rovinu. Rovina je definována následující rovnicí:

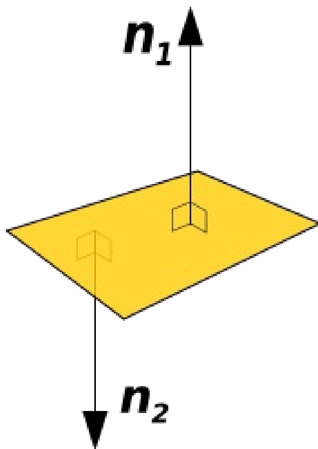
$$ax + by + cz + d = k, \qquad \textit{Vzorec 1.1}$$

kde $[x, y, z]$ jsou parametry roviny a (a, b, c) je právě normálový vektor roviny. Dosadíme-li do takové rovnice plochy za $[x, y, z]$ libovolný bod ležící na rovině, potom platí že $k = 0$. Každá rovina ale může být definována dvěma normálovými vektory, jak vyplývá z obrázku 1.2. V takovém případě

je pak potřeba určit, která z dvojice normál je platná. Pokud je z plošek seskládáno těleso, platná normála musí být ta, která z tělesa vystupuje.

Výpočet vztahu mezi plochou a vektorem pozorovatele se provádí dosazením vektoru pozorovatele do rovnice dané plochy za hodnoty *x*, *y*, *z*. Mohou nastat následující dvě možnosti:

1. jestliže $k > 0$, rovina je přilehlá k pozorovateli a
2. jestliže $k < 0$, rovina je odlehlá od pozorovatele

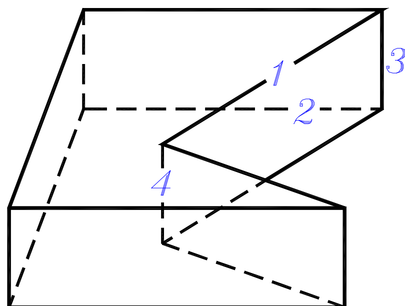


obr 1.2: Normálové vektory plochy

1.2.2 Ohodnocení hran objektů

Princip který ohodnocuje jednotlivé hrany objektů na základě zjištění viditelnosti dvojice sousedících stěn, které hrana spojuje. Ohodnocení hran objektů si našlo uplatnění např. v Robertsově algoritmu, který je popsáno v kapitole 1.4.1. V závislosti na viditelnosti ploch mohou nastat následující 3 situace, které ilustruje obrázek 1.3:

1. Hrana dvou přilehlých ploch je *potenciálně viditelná*. (Hrana 1)
2. Hrana dvou odlehlých ploch je *neviditelná*. (Hrana 2)
3. Hranu tvoří přivrácená a odvrácená plocha. V případě, že úhel normál mezi plochami je ostrý, hrana *neviditelná* (Hrana 4). Pokud je úhel je tupý, hrana je *obrysová* (Hrana 3).



obr 1.3: Ohodnocení hran objektu

1.3 Dělení algoritmů dle výpočtové složitosti

Podle způsobu, kterým zpracovávají scénu, je lze rozdělit do 3 základních kategoriím, které budou popsány v následující odstavcích.

1.3.1 Objektově orientované algoritmy

Objektové algoritmy pracují s objekty scény (stěny, hrany), u kterých porovnávají vzájemnou viditelnost. Výsledkem objektových algoritmů je seznam viditelných a neviditelných hran. Nevýhodou těchto algoritmů je velká citlivost na chyby při výpočtu. Časová náročnost objektových algoritmů je

$$O(n)=n^2 ,$$ *Vzorec 1.2*

kde n je počet objektů scény. Postup, kdy se orientujeme na objekty lze obecně charakterizovat algoritmem:

```
for každý objekt do
  begin
    Urči části (hrany resp. stěny) objektu,
    které nejsou zakryté ostatními objekty;
    Nakresli tyto nezakryté části objektu jejich barvou;
  end;
```

Kód 1.1: Pseudokód objektového algoritmu.

1.3.2 Obrazově orientované algoritmy

Obrazové algoritmy sice taktéž pracují se seznamem objektů, tímto seznamem však neprocházejí jako v případě objektových algoritmů. Obrazové algoritmy procházejí jednotlivými pixely obrazu, pro které ze seznamu objektů hledají ten objekt scény, který je danému pixelu mu nejblíže. Tyto algoritmy jsou méně citlivé na chyby, které mohou nastat ve výpočtu. Jejich nevýhodou je ale pevný rozměr výsledného rasteru. Časová složitost obrazových algoritmů je

$$O(n)=n\cdot p ,$$ *Vzorec 1.3*

kde n je počet objektů scény a p je počet pixelů, pro které je obraz zpracováván. Obrazově orientovaný postup lze charakterizovat algoritmem:

```
for každý pixel do
  begin
    Urči objekt, který je nejblíže pozorovateli
    ve směru pohledu vedeného každým pixelem;
    if paprsek vedený pixelem prochází objektem then
      Obarvi pixel barvou příslušnou k objektu
    else
      pixel obarvi barvou pozadí
  end;
```

Kód 1.2: Pseudokód obrazového algoritmu.

1.3.3 Kombinované algoritmy

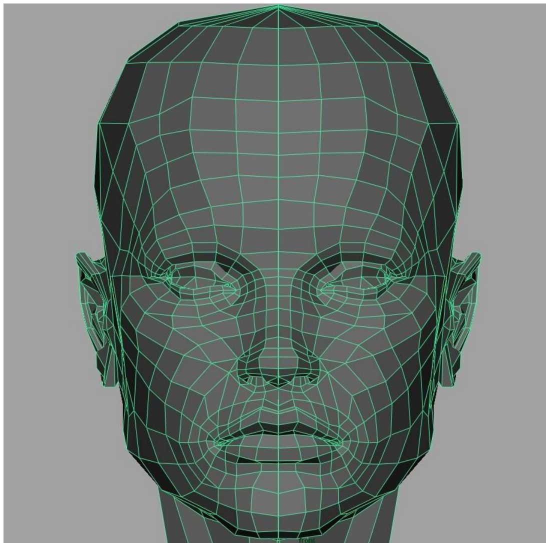
Kombinované algoritmy využívají principů obou předchozích typů. Využití kombinovaných algoritmů může být např. k předzpracování scén a třídění dat.

1.4 Vektorové algoritmy

Vektorové algoritmy (HLE – hidden line elimination) jsou takové algoritmy, jejichž výstupem je vektorový výsledek souboru viditelných a neviditelných hran. V následujících bodech budou popsány dva základní vektorové algoritmy řešení viditelnosti objektů.

1.4.1 Robertsův algoritmus

Publikován L.G. Robertsem v roce 1963. Cílem Robertsova algoritmu je správné vykreslení drátěného modelu objektů scény. Drátěný model potřebujeme vykreslovat tam, kde je důležité znát konstrukci polygonů objektů, tedy jak tyto polygony uspořádány. Zobrazit drátěný model je tak vhodný při modelování, proto si Robertsův algoritmus našel své uplatnění v aplikacích pro modelování prostorové počítačové grafiky, jakými jsou např. CAD systémy, 3D Studio Max apod.

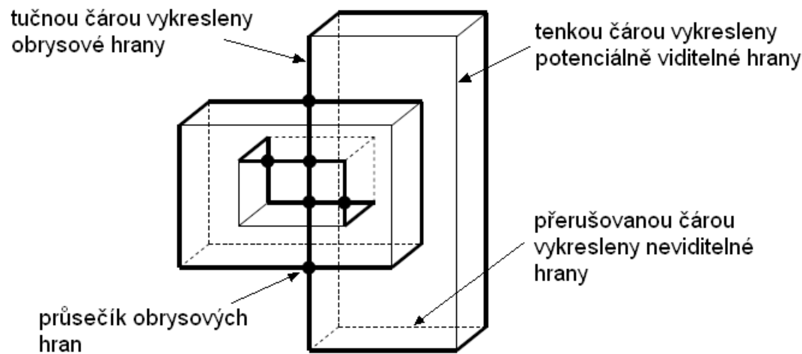


obr 1.4: Silueta obličeje s vykresleným drátěným modelem.

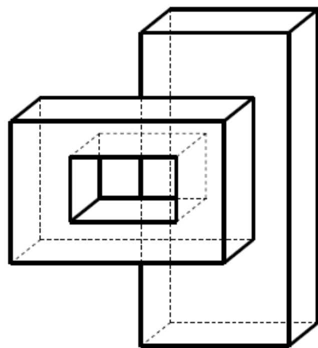
Robertsův algoritmus využívá myšlenky ohodnocení hran popsáné v kapitole 1.2.2. Algoritmus zpracovává scénu s konvexními mnohostěny a vytváří z nich seznam viditelných a neviditelných hran. Robertsův algoritmus má následující postup:

- Získání *potenciálně viditelných* a *obrysových* hran objektů scény
- Rozdělení všech získaných hran na úseky konstantní viditelnosti:
    - Podle průsečíků s obrysovými hranami.

    - Včetně hran stejného tělesa (pro nekonvexní objekty)
- Testování viditelnosti úseků podle vzdálenosti průsečíků a zakrytí:
    - Viditelnosti se mění v obrysových hranách.
    - Střídají se úseky viditelné a zakryté.
- Vykreslení viditelných úseků hran.



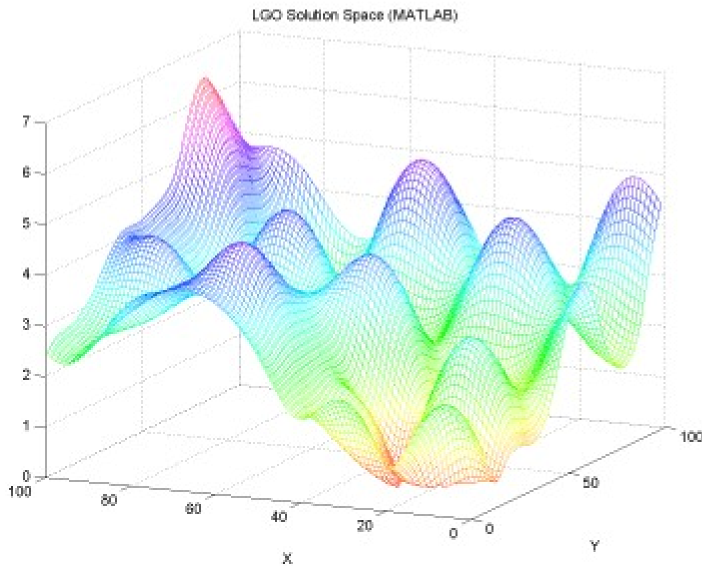
obr 1.5: Ohodnocení hran objektů.



obr 1.6: Výsledek scény po zpracování Robertsovým algoritmem.

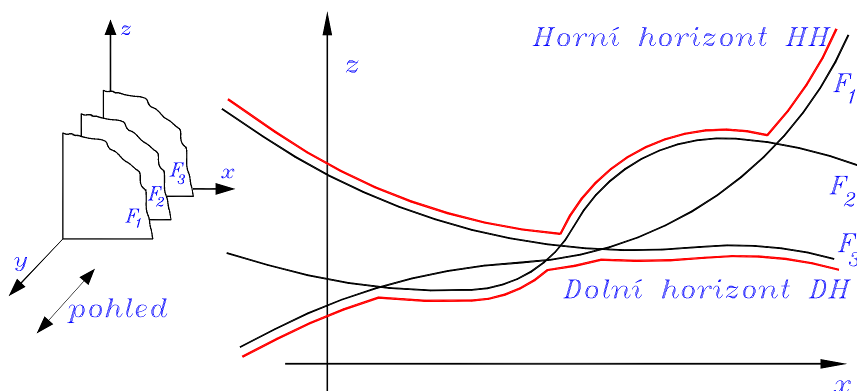
1.4.2 Plovoucí horizont

Plovoucí horizont je algoritmus sloužící k vizualizaci matematických funkcí za pomocí 3D grafů. Své využití si tedy našel zejména v technicky výpočetních aplikacích, jako jsou např. Matlab, Maple apod.



obr 1.7: Ukázka vizualizovaného grafu v programu MATLAB

Graf je vykreslen pomocí řezů v rovinách XZ a YZ. Ty jsou vykresleny od předu dozadu, přičemž jsou zaznamenávány informace o horním a dolním horizontu řezu (viz. obr. 1.8). Horním/dolním horizontem je pomocné pole, obsahující maximální/minimální vykreslenou hodnotu řezu. Zvolený řez je porovnáván s aktuálním stavem horního a dolního horizontu a vykresleny jsou pouze ty části, které zasahují mimo oblasti horizontů.



obr 1.8: Znázornění stavu horního/dolního horizontu po vykreslení třech řezů

Algoritmus je vhodný zejména pro zobrazení funkcí dvou proměnných. Ty jsou nejčastěji vyjádřeny v explicitním tvaru:

$z = F(x, y)$, *Vzorec 1.4*

pro $x \in \langle x_a, x_b \rangle$ a $y \in \langle y_c, y_d \rangle$.

Zpracování grafu probíhá dle následujícího postupu:

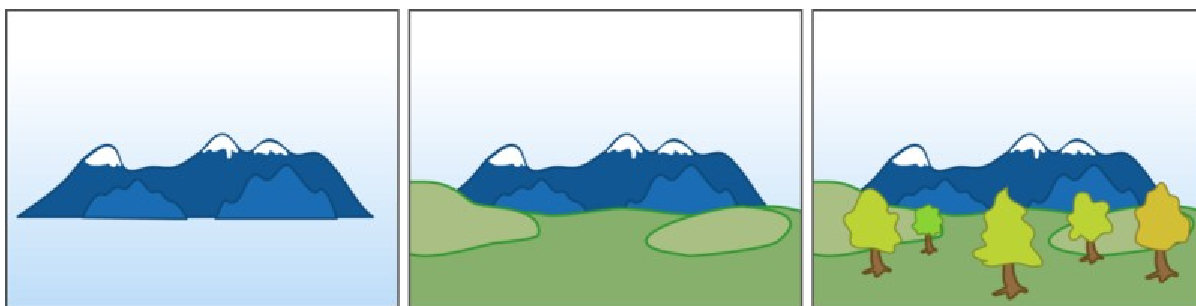
1. Inicializace masky horizontu:
   - nastavení dolní hranice masky (dolní okraj kreslící plochy).
   - nastavení horní hranice masky (horní okraj kreslící plochy).
2. Průběh vykreslování mimo hranice nastavené masky.
3. Přestavení horizontů.
4. Návrat k bodu 2., dokud nedojde k vyčerpání intervalu $\langle y_c, y_d \rangle$

1.5 Rastrové algoritmy

Za rastrové algoritmy (HSE – hidden surface elimination) jsou považovány takové algoritmy, jejichž výsledkem rastrový obraz scény, popř. obraz viditelných ploch. V následující kapitolách budou popsány 4 nejzákladnější rastrové algoritmy, jimiž jsou Malířův algoritmu, dělení obrazu, Z-Buffer a Ray-Casting.

1.5.1 Malířův algoritmus

Objektový algoritmus postavený na metodě *hloubkového třídění* scény. Algoritmus převádí řešení viditelnosti na úlohu uspořádání ploch podle jejich vzdálenosti od pozorovatele (obr. 1.10), což v praxi znamená setřídění dle z-tových souřadnic, nebo také hloubkových souřadnic (odtud tedy varianta názvu „algoritmus hloubkového třídění“). Setříděné plochy jsou vykreslovány v pořadí od nejvzdálenějšího k nejbližšímu, tím dochází k překreslování vzdálených objektů objekty bližšími. Princip překreslování je zobrazen na obrázku 1.9. Vykreslování scény je tak obdobný způsobu kreslení obrazu malířem, který taktéž vykresluje zprvu vzdálené objekty a ty posléze překreslí další vrstvou bližších objektů. Malířův algoritmus se svými vlastnostmi hodí zejména pro vykreslení scén složených z velkého množství malých plošek. Naopak je nevhodný pro zpracování scén s rozsáhlými plochami, které vykresluje nesprávně.

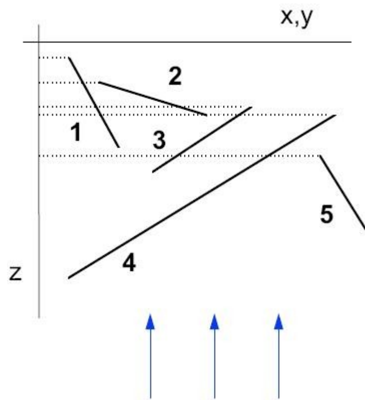


obr 1.9: Ukázkový příklad průběhu vykreslení scény Malířovým algoritmem.

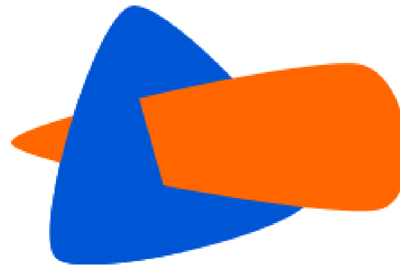
Algoritmus má na charakteristiku scény 2 základní předpoklady, jimiž jsou:

1. scéna musí být složená pouze z *rovinných plošek*

2. plošky mají společné body pouze na obvodu, nesmí se tedy navzájem prosekávat, jak je ukázáno na obrázku 1.11

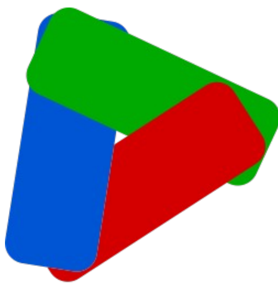


obr 1.10: Setřídění polygonů.

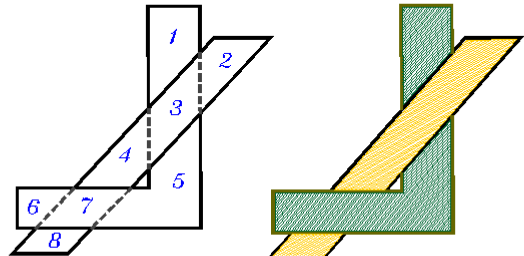


obr 1.11: Příklad konfliktu scény, který může vést k nespravenému vykreslení

Kromě konfliktu ukázaného na obrázku 1.11 může ve scéně vzniknout ještě jeden konflikt, který má za následek zacyklení algoritmu. Tímto konfliktem je vzájemné překrývání ploch mezi sebou. Konflikt je zobrazen na obrázku 1.12 a řeší se pomocí rozdělení ploch na jednotlivé části tak, aby se konflikt cyklického překrývání odstranil (viz. obr 1.13).



obr 1.12: Ukázka vzájemného překrývání ploch, které může vést k zacyklení algoritmu.



obr 1.13: Princip řešení vzájemného překrývání ploch.

Navíc Malířův algoritmus musí provádět *kontrolu pořadí* ploch, tj. zjistit, jestli v setříděném seznamu ploch nevznikají určité konflikty, které mohou nastat při překrývání ploch mezi sebou. Vzniklé konflikty by měly za následek nesprávné vykreslení obrazu, proto algoritmus navíc prochází seznam a testuje, jestli kandidátní plocha není v konfliktu s jinou. Kandidátní plocha je tak podrobena čtveřicí testů pro otestování překrývání vůči ostatním plochám. Kandidátní plocha je označena symbolem **P**, plochy následující v seznamu označeny symbolem $\mathbf{Q}_i$. Zmíněnými testy jsou:

1. **Minimax test**: testuje, jestli se plochy **P** a $\mathbf{Q}_i$ vůbec protínají v průmětu os *x* a *y* (viz. obr 1.14). Minimax test porovnává obdélníky opsané daným plochám. Pokud není zjištěný průnik obdélníků, test uspěje, jinak se pokračuje následujícím testem.

2. **P versus rovina Q**: testování, jestli plocha **P** neleží celá za rovinou danou polygonem $\mathbf{Q}_i$ (viz. obr 1.15). Výpočet prováděn dle vzorce

   $ax+by+cz+d<0$,   *Vzorec 1.5*

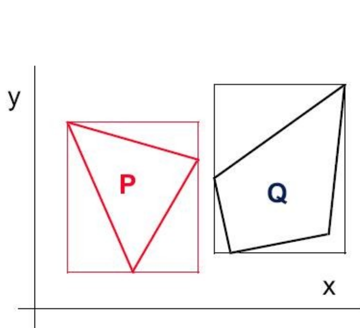
   kde a, b, c, d jsou parametry plochy $\mathbf{Q}_i$. Pokud je výraz platný, test uspěje.

3. **Q versus rovina P**: podobný případ předchozího testu. Zde však testujeme naopak, jestli plocha $\mathbf{Q}_i$ neleží celá před rovinou danou polygonem **P** (viz obr. 1.16). Výpočet prováděn vzorcem

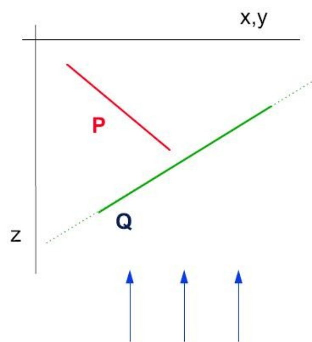
   $ax+by+cz+d>0$,   *Vzorec 1.6*

   kde a, b, c, d jsou parametry plochy **P**. Pokud je výraz platný, test uspěje.

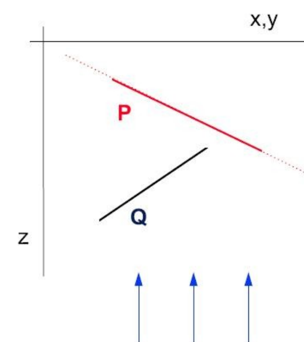
4. **Úplný test v průmětu**: když neuspěje ani jeden z výše zmiňovaných testů, jsou plochy **P** a $\mathbf{Q}_i$ testovány v průmětu os x a y. Zjišťuje se, jestli některá část **P** nepřekrývá plochu $\mathbf{Q}_i$. V takovém případě by **P** nešel vykreslit před $\mathbf{Q}_i$. Proti sobě testujeme všechny hrany **P** a $\mathbf{Q}_i$ a hledáme jejich průsečíky (viz. obr 1.17, 1.18). Při nalezení průsečíku porovnáváme z-tové souřadnice. Musí platit, že z-tová souřadnice polygonu **P** je vzdálenější než z-tová souřadnice $\mathbf{Q}_i$. Není-li průsečík nalezen, testuje se, jestli **P** leží celý uvnitř $\mathbf{Q}_i$ (viz. obr 1.19, 1.20).



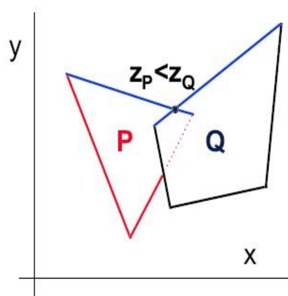
obr 1.14: Minimax test



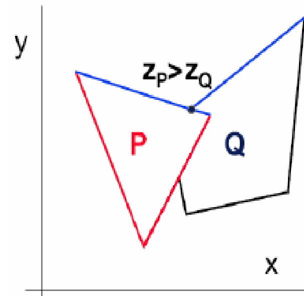
obr 1.15: P vs. rovina Q



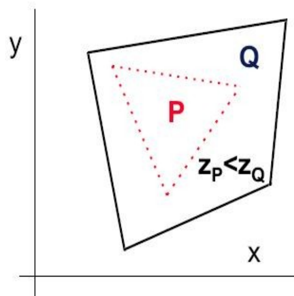
obr 1.16: Q vs. rovina P



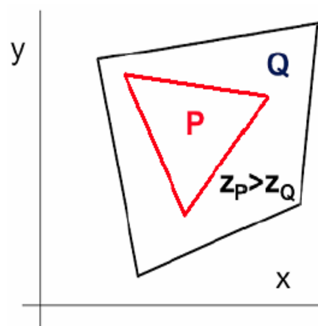
obr 1.18: Úplný test v průmětu. P leží za Q.



obr 1.17: Úplný test v průmětu. P leží před Q.



obr 1.19: P leží celý uvnitř a za Q



obr 1.20: P leží celý uvnitř a před Q

Neuspěje-li ani jeden z předchozích testů kontroly pořadí, nemůže dojít k vykreslení kandidátní plochy. V takovém případě musíme prohodit plochy **P** a $\mathbf{Q}_i$ mezi sebou a test provést znovu.

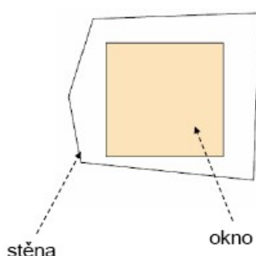
1.5.2 Warnockův algoritmus dělení obrazu

Objektový algoritmus představený J. Warnockem v roce 1969. Myšlenka algoritmu spočívá v rozdělení složitého problému na řadu menších, snáze řešitelných. Je přitom využito rekurzivního dělení obrazu. Každý pixel obrazu je kreslen pouze jednou. Warnockův algoritmus pracuje se scénou složenou z rovinných plošek, které se mohou i libovolně mezi sebou prosekávat.

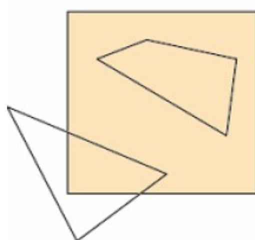
Tento algoritmus zjišťuje, jestli lze viditelnost objektů spadající pod plochu obrazu vyřešit jednoduše. V případě, že o viditelnosti objektů nelze rozhodnout, rozdělí obraz dvěma příčnými řezy a stejný postup provádí pro jednotlivé řezy. Dělení může probíhat klidně až na úroveň jednoho pixelu.

Ve vztahu k oknu může plocha vhledem nabýt celkem třech stavů:

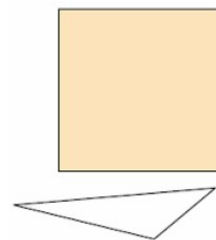
1. plocha **pokrývá** celou oblast okna (obr. 1.21)
2. plocha **zasahuje** do okna (obr. 1.22)
3. plocha **nezasahuje** do okna (obr. 1.23)



obr 1.21: Plocha pokrývá okno



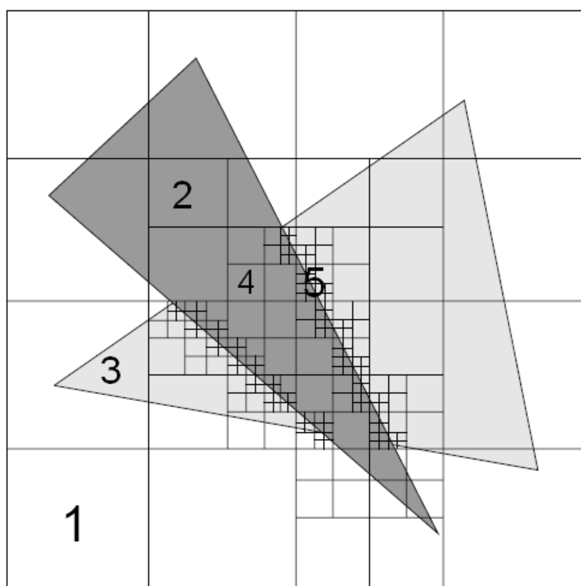
obr 1.22: Plocha zasahuje do okna



obr 1.23: Plocha nezasahuje do okna

Pro okno obrazu mohou nastat následující případy. všechny tyto případy jsou znázorněny na obrázku 1.24:

1. Žádná plocha *nezasahuje* ani *nepokrývá* okno. Okno je vyplněno barvou pozadí.
2. Pouze jediná plocha *pokrývá* okno. Ostatní plochy do něj nezasahují. Okno se vyplní barvou plošky.
3. Pouze jediná plocha *zasahuje* do okna. Okno se vyplní barvou pozadí, poté se vykreslí plocha ořezaná vhledem k oknu.
4. Několik ploch *zasahuje* nebo pokrývá okno, avšak jedna z ploch leží před všemi ostatními. Okno se vyplní barvou nejbližší plochy.
5. Nenastává žádný z výše uvedených případů. V takovém případě je okno rozděleno na čtyři shodné části a algoritmus je volán zvlášť pro jednotlivé části. Dostane-li se algoritmus až na úroveň pixelu, je do něj vykreslena barva nejbližší plochy.

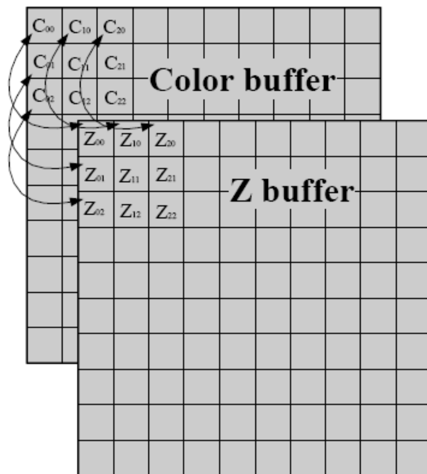


obr 1.24: Princip Warnockova algoritmu

1.5.3 Z-Buffer

Z-Bufferování obrazu je nejznámější a nejefektivnější metodou pro řešení viditelnosti obrazu. Jedná se o obrazový algoritmus realizovaný zejména hardwarově jako komponenta grafických akcelerátorů. Z-Buffer lze však realizovat i softwarově. Takové řešení ale není plně optimalizované pro výkon, který je pro Z-Buffer největším kladem. Dalšími výhodami Z-Bufferu je nepotřeba třídění ploch a správné vykreslení i nestandardních situací z obrázků 1.21 a 1.22, které v případě Malířova algoritmu musí být speciálně řešeny. Plošky dokonce nemusí být ani rovinného tvaru, jelikož dochází k rozložení objektu na pixely. Z-Buffer navíc umožňuje i vyplňování ploch a taktéž stínování. Snad jedinou nevýhodou metody Z-Buffer je vysoká náročnost na paměť, které ale v současnosti z důvodu dobré dostupnosti velkých pamětí již delší dobu není aktuální.

Základem metody je dvourozměrné pole (tzv. De*pth buffer*), jehož rozměry se shodují s rozměry zobrazovacího okna. Principem Z-Bufferu je uchovávání si nejmenších z-tových souřadnic pro jednotlivé obrazové body a taktéž vytvářet i paměť barev pro tyto body (tzv. Colo*r buffer*). Vazba mezi Z-Bufferem a Color bufferem je znázorněna na obrázku 1.25. Každá plocha scény je rastrována a jednotlivé body rastru jsou porovnávány s odpovídajícím bodem v Z-Bufferu. Když je hodnota hloubky bodu menší než hodnota v Z-Bufferu, je hloubka tohoto bodu rastru zaznamenána do Z-Bufferu a bod je vykreslen do výsledného obrazu na patřičnou pozici.



obr 1.25: Znázornění vazby Z-Bufferu a Color bufferu

Z-Bufferování je popsáno následujícím postupem:

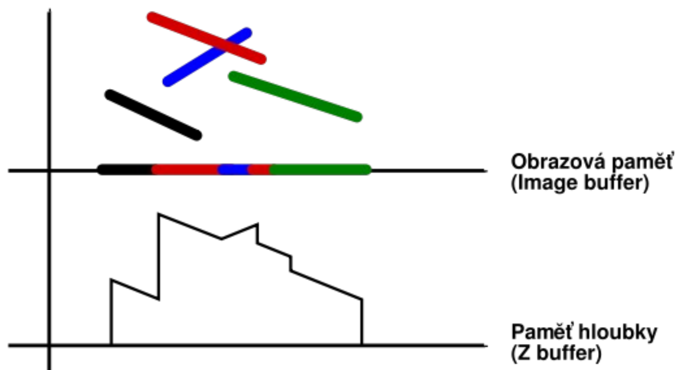
1. *Inicializace*:
   - color buffer je vyplněn barvou pozadí
   - hodnoty hloubky Z-Bufferu jsou nastaveny na největší možné číslo, tj. nekonečno
2. Každý objekt je rozložen na jednotlivé pixely, přičemž rozklad lze provést v libovolném pořadí. Pro každý pixel $[x_i, y_i]$ je stanovena jeho hloubka z_i
3. Pro každý pixel je proveden test hloubky. Test hloubky může být popsán následujícím jednoduchým kódem:

```
for všechny objekty do
  rasterizuj objekt;
  for všechny pokryté pixely(x,y) do
    begin
      if hloubku pixelu je menší než hodnota v Z-Bufferu then
        ulož hloubku pixelu do Z-Bufferu;
        Vykresli pixel(x,y);
    end;
```

Kód 1.3: Pseudokód testování hloubky pixelu se Z-Bufferem

Z kódu jednoznačně vyplývá, že má-li z_i menší hodnotu než položka [x_i, y_i] v Z-Bufferu, tak dojde k:

- obarvení pixelu [x_i, y_i] barvou nového pixelu
- aktualizování položky [xi, yi] Z-Bufferu na novou hodnotu Zi

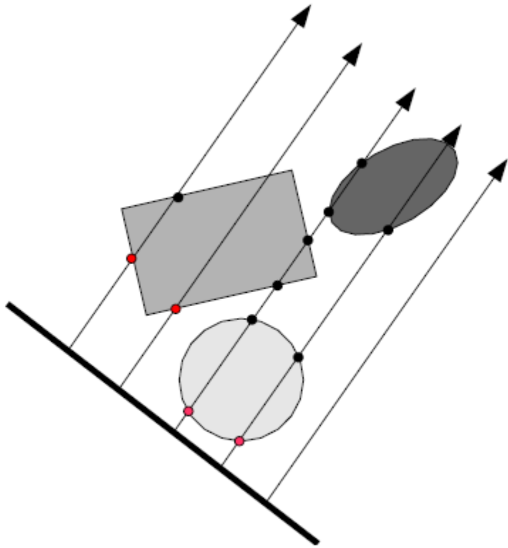


obr 1.26: Princip fungování Z-Bufferu

1.5.4 Ray-Casting

Ray-Casting je jistým zjednodušením metody Ray-Tracing. Tato metoda využívá principu vrhání paprsku do scény a sledování jeho průchodu scénou dle toho, jak se od jednotlivých objektů scény odráží. Takhle lze docílit velmi realistických výsledků modelu scény avšak při vysoké časové náročnosti na výpočet obrazu. Proto nelze při současné situaci výkonu počítačů nasadit metodu Ray-Tracing pro renderování realtime scén. Metoda Ray-Casting zjednodušuje Ray-Tracing tím, že nerealizuje odrazy paprsků od těles, pouze vyhodnocuje primární paprsky a jejich dopady na nejbližší objekt. Tím je dosaženo stále kvalitních výsledků řešení viditelnosti prostorových scén při daleko menší výpočetní náročnosti algoritmu než je tomu v případě metody Ray-Tracing. Metodu Ray-Casting tak lze v současnosti již používat i pro renderovaní realtime scén, čímž se hodí pro nasazení do 3D počítačových her a počítačových animací.

Metoda je tedy založená na principu obrazových algoritmů. Pro každý pixel grafického okna metoda zpracovává obraz scény tak, že v místě pixelu počítá průsečíky se scénou. Algoritmus prochází všemi polygony scény a pro ně počítá, jestli paprsek, procházející aktuálním pixelem, prochází některou z ploch scény. Z vypočtených průsečíků poté algoritmus vytváří seznam, ze kterého vybírá průsečík nejblíže pozorovateli a taktéž barevnou hodnotu nejbližšího průsečíků pro aktuální pixel vykreslí na plátno. Ray-Casting je možné nasadit i na scény složené ze zaoblených ploch.



obr 1.27: Znázornění principu metody Ray-Casting

2 Návrh demonstrační aplikace

V části pro návrh aplikace bude rozebrán koncept programu a jakým způsobem bude pro daný koncept probíhat demonstrace zvolených algoritmů.

Vhodným stavem pro aplikaci je, aby uměla demonstrovat všechny výše zmíněné základní algoritmy řešení viditelnosti. Bohužel z důvodu časové tísně musela být vybrána pouze část těchto algoritmů. Rozhodl jsem se proto vybrat ze škatulky algoritmů rastrových, se kterými je běžný uživatel přeci jenom více ve styku, než s algoritmy vektorovými. Demonstrace rastrových algoritmů tak může být zajímavá i pro zmíněného běžného uživatele. Pro demonstrování byly vybrány *Malířův algoritmus*, *Z-Buffer* a *Ray-Casting*, jejichž návrh bude zde podrobně rozebrán. Samozřejmostí je také demonstrování základní myšlenky *vyřazení odvrácených stěn*.

2.1 Definice pojmu „demonstrační aplikace“

Než se však vrhneme do návrhu jednotlivých demonstrací, je důležité si upřesnit zadání projektu a toho se držet, čímž by se při návrhu scén mělo předejít návrhu takové aplikace, která je v kontextu zadání zcela zbytečná. Ze zadání bych se pozastavil zejména nad pojmem *demonstrace*.

Co to vůbec demonstrace je? V oboru lingvistiky nabírá tento pojem dokonce několika významů. Známými významy demonstrace může být například demonstrování vojenských sil, demonstrování vlastností výrobků. Význam demonstrace má své postavení i v politice, kdy může znamenat demonstrování lidu jako výraz nespokojenosti s politikou. Avšak my se budeme zabírat pouze tím pojmem, který má z kontextu naší aplikace smysl.

V kontextu výukové aplikace pojem demonstrace nabývá významu pro postup předvádění, pomocí něhož je student obeznámen s určitou látkou. Demonstrace probíhá názornou ukázkou principů, funkce, vlastností, možností popřípadě jiných věcí, které se pro daný studijní okruh hodí demonstrovat. Demonstrování by mělo být zejména srozumitelné a hlavně by mělo probíhat na jednoduchých příkladech, aby se tak studijní látka neztrácela v rozsahu složitosti demonstrace.

V případě demonstrací algoritmů pro řešení viditelnosti tak není cílem navrhnout objemné, složité scény využívající těch nejmodernějších postupů a principů a ukázat co v současnosti je v počítačové grafice realizovatelné. Demonstrační scény by měly být pokud možno co nejjednoduššího charakteru a aby prováděly opravdu jenom demonstrování problematiky viditelnosti. Neměla by proto obsahovat spoustu zbytečných věcí navíc, díky čemuž by došlo pouze jen ke snížení výpovědní hodnoty scény o daném algoritmu. Principy demonstrovaných algoritmu musí být ve scéně samozřejmě viditelné a zřetelné.

Na druhou stranu však není cílem aplikace předvést doslovný popis jednotlivých algoritmů. Demonstrace by měly být stavěny pouze na základních principech algoritmů. Uživatel těmito

principy obeznámený si přesné znění algoritmů může, za pomocí odborné publikace, popřípadě internetu, kdykoliv dohledat. Jelikož takto seznámený uživatel bude mít o algoritmech vytvořenou představu, bude pro něj studium tak snáze pochopitelné.

Výukové aplikace mají navíc tu nevýhodu, že nedokáží sledovat odezvu v uživatelovi. V reálu může vyučující pozorovat reakce studentů a dle těch přizpůsobovat výklad učiva popř. svůj vlastní styl přednesu. V oblasti výukových aplikací toto není doposud realizovatelné. Proto musí aplikace výuku provádět takovým způsobem, který i zapojí uživatelovu aktivitu a nebyla jenom pouhou ukázkou něčeho. Jedním z nejobvyklejších způsobů pro zapojení uživatele do výuky je vytvoření *interaktivity* mezi programem a uživatelem.

2.2 Základní rozhraní aplikace

Jak již bylo řečeno, hlavní částí projektu je vytvořit aplikaci pro demonstraci algoritmů řešení viditelnosti. Aplikace tak musí umět jednotlivé algoritmy provádět, popřípadě alespoň věrohodně odsimulovat. Tohle však není vše, co by měla zvládat. Kdyby uživatel viděl pouhý výstup daného algoritmu, z práce algoritmu by se nedozvěděl zcela nic. Je potřeba tedy aplikaci přizpůsobit tak, aby mohly být pozorovány jednotlivé kroky algoritmů pro řešení viditelnosti.

Další důležitou věcí je zabývat se způsobem, kterým bude uživatel práci algoritmů pozorovat. Na jednu stranu je potřeba, aby šlo pozorovat postupné výsledky jednotlivých kroků algoritmů přesně pro ten pohled, ze kterého uživatel scénu pozoruje. Tahle myšlenka je logická, avšak uživatel by po celou dobu práce algoritmu musel být přichycen k tomu úhlu pohledu na scénu, pro který by vykreslování probíhalo. To znamená že by tím uživatel ztratil 3D rozhled na scénu a i přesto že by viděl jednotlivé kroky, podstatu algoritmů by vůbec nemusel pochopit.

Na druhou stranu by tedy měl být v programu umožněn právě 3D rozhled na scénu, ve kterém by uživatel mohl pozorovat vykreslování i z jiné pozice ve scéně než z té, pro kterou probíhá vykreslování scény. Takový pohled poskytuje mnohem větší přehled o scéně během procesu vykreslování. Tím že tento způsob rozhledu nabyl zpátky svého třetího rozměru, může uživatel taky sledovat jak jsou objekty vzdáleny vůči pozorovateli, vůči kterému vykreslování probíhá. Což je jedna z nejstěžejnějších vlastností scény, protože právě vzdálenost jednotlivých objektů od pozorovatele je nejdůležitější vlastností při řešení problému viditelnosti objektů.

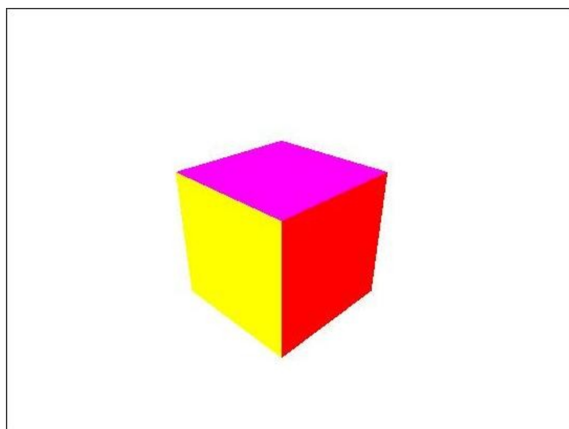
Avšak tím že jsme zavedli 3D rozhled po scéně i během vykreslování, uživatel zase ztratil možnost sledovat skutečný výsledný obraz. Z tohoto konceptu pohledu na scénu je sice možné si o výsledném obrazu udělat svojí představu, ale tohle není zrovna nejvhodnějším řešením, protože aplikace nemá za úkol testovat uživatelovu představivost. Jelikož se oba dva koncepty takto dokáží vzájemně doplňovat, je nejlepší možností, aby program umožňoval náhled na scénu oběma způsoby. Uživatel tím může libovolně nahlížet do jednoho ze dvou oken, porovnávat mezi sebou výsledky jejich scény a přemýšlet o principu algoritmu, který má být v aplikaci aktuálně demonstrován. Zcela

logicky si musí obsahy scén obou pohledů mezi sebou korespondovat. Jednotlivé pohledy budou více rozebrány v následující části.

2.2.1 Výsledný pohled

Je to právě ten pohled na scénu, do kterého se vykresluje výsledek scény - buď úplný, nebo postupný - zde záleží na charakteru demonstrace. Všechny demonstrované algoritmy vykreslují scénu právě vůči výslednému pohledu. Z výše zmíněných důvodů výsledný pohled neumožňuje žádnou manipulaci ohledně rotací a posuvů se scénou. Vyplývá to tak z podstaty situace. I když algoritmy ve skutečnosti vytvářejí pohyblivý obraz, výpočty stále provádějí pro scénu, která je v daný moment ukotvená. To znamená, že pro jeden cyklus výpočtu algoritmu je scéna statická (nepohyblivá). Kdyby byla v průběhu výpočtu scéna libovolným způsobem pozměněná (rotace, posun), algoritmus by pak jednotlivé objekty vykreslil z jiného úhlu pohledu a scéna by tak byla vykreslena nesprávně.

Tuto vlastnost tak musí přejímat i výsledný pohled, jelikož v něm je demonstrováno vykreslení právě jednoho cyklu algoritmu. Je proto důležité, aby s výsledným pohledem nešlo nijak manipulovat.



obr 2.1: Příklad scény výsledného pohledu.

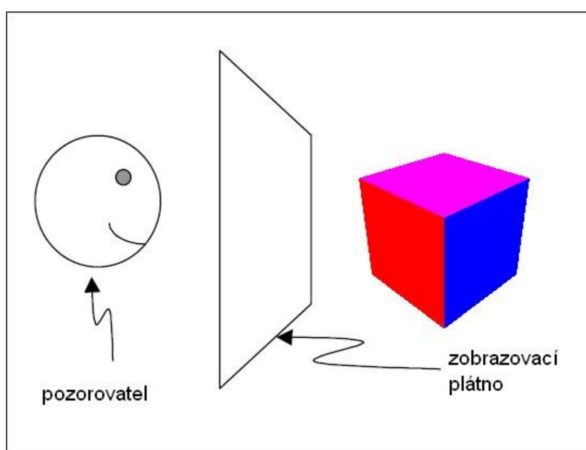
2.2.2 Pracovní pohled

Tento pohled na scénu je druhý z výše zmiňovaných. Zde algoritmy nepracují vůči poloze skutečného pozorovatele pohledu, ale pracují vůči jistému relativnímu bodu, jehož poloha ve scéně se mění zároveň s tím, jak je se scénou v tomto pohledu rotováno. Tímto relativním bodem je pozice, ze které uživatel pozoruje scénu výsledného pohledu. Poté je důležitým kritériem, aby pracovní pohled uměl korespondovat výslednému pohledu a tyto dva pohledy si svým obsahem scén odpovídaly i s ohledem na pozici pozorovatele. Kdyby tak uživatel pozoroval scénu z daného relativního bodu ve scéně pracovního pohledu, viděl by zcela shodnou scénu, kterou mu umožňuje pozorovat výsledný pohled. Aby ale uživatel v pracovním pohledu věděl o přítomnosti a poloze onoho relativním bodu pozorovatele, musí se o tomto bodu dát uživateli najevo. Nabízí se tedy možnost bod pozorovatele do

scény zřetelně vykreslit jako objekt, čímž dojde k vizuálnímu zobrazení relativního bodu pozorovatele ve scéně.

Relativní bod pozorovatele ale není jediným aspektem pro zpřehlednění orientace v pracovním pohledu. Během vykreslování objektů scény na grafický výstup totiž dochází k tzv. *projekci scény*. Jednotlivé objekty jsou „obtisknuty“ na pomyslné virtuální plátno, které znázorňuje plochu grafického výstupu. Zvýraznění obrysu virtuálního plátna do pracovního pohledu taktéž zpřehlední orientaci uživatele ve scéně, proto obrys zobrazovacího plátna bude do scény pracovního pohledu taktéž vykreslen.

Pracovní pohled by logicky umožňovat uživateli základní transformace nad scénou, jakými jsou rotace, posuv scény a zoomování. Díky těmto transformacím může uživatel nahlížet na scénu z libovolného úhlu pohledu a pozorovat demonstraci z pozice, která je mu právě nejpříjemnější. Taktéž může důkladněji prozkoumat obsah scény.



obr 2.2: Příklad pracovního pohledu s obsahem scény korespondující výslednému pohledu z obr. 2.1 a se znázorněním polohy pozorovatele a zobrazovacího plátna

2.2.3 Časový proklad mezi kroky algoritmu

V praxi probíhá zpracování viditelnosti scény v tak krátkém časovém intervalu, že během procesu vykreslování lze okem zcela těžko postřehnout jakýkoliv mezistav vzniklý mezi prvním a posledním krokem algoritmu. Kdyby byla demonstrace algoritmů vykonána v nejmenším možném časovém intervalu, tj. tak jak probíhá v praxi, uživatel by nezpozorovat žádnou práci algoritmu. Viděl by pouze již samotný výsledek algoritmu, což by ztratilo jakoukoliv výpovědní hodnotu o těchto algoritmech.

Je tedy důležité, aby aplikace jednotlivé kroky výpočtu mezi sebou časově prokládala a umožnila tak uživateli pozorovat samotné kroky algoritmu. Uživatel tedy bude v přijatelných časových intervalech pozorovat změny, které ve scéně nastanou vykonáním jednoho kroku algoritmu.

2.2.4 Krokování algoritmu

V části pro definování pojdu „demonstrační aplikace“ (kapitola 2.1) byl kladen důraz na interaktivitu. V bodě pro krokování programu si představíme první interaktivní část programu.

Kdyby byl uživatel odkázaný ke sledování sekvence kroků, nedostalo by se mu tak možnosti k důkladnějšímu prozkoumání jednotlivých mezikroků, jelikož by tížen časem, za který nastane následující krok algoritmu. Problém by byl vyřešen stanovením poměrně velké časové prodlevy mezi následujícími kroky, kde by uživatel měl dost prostoru na prozkoumání scény po zpracování dalšího kroku. Nastává ale otázka, jak velká časová prodleva je pro demonstrace ideální? Tak jako každý člověk zpracovává informace jiným způsobem a za různě dlouhou dobu, tak se jednomu uživateli může zdát zpracování rychlé a jinému zase pomalé. Rozhraní programu by tak mělo navíc umožňovat individuální krokování algoritmu, aby si uživatel mohl řídit posuny na další kroky algoritmů dle vlastní libosti.

Rozhraní pro krokování bude vykonávat následující funkce:

- *změna kroku*: nebo-li změna časové prodlevy mezi jednotlivými kroky algoritmu při simulaci algoritmu.
- *spuštění* demonstrace: provádí simulaci algoritmu s časovým prokladem mezi kroky algoritmu. Scéna je spuštěna od pozice, ve které se aktuálně nachází. Běh je ukončen buď v případě přerušení uživatelem, nebo v případě, že algoritmus došel na konec.
- *pozastavení* běhu: pro případ, že uživatel bude potřebovat zastavit demonstraci, aniž by došla nakonec. Pozastavená scéna si pamatuje svoji poslední pozici. Při pokračování na následující krok pokračuje stejně, jako kdyby běh této scény nebyl přerušen.
- *následující krok*: vhodná funkce v případě, že uživatel si přeje krokovat průběh demonstrace sám. Uživatel není tížen časovým intervalem mezi jednotlivými kroky a má tak více prostoru pro pořádné prozkoumání aktuálního stavu scény.
- *inicializace*: důležitá funkce v případě, že uživatel požaduje nové vykonání demonstrace, popřípadě se v průběhu demonstrace rozhodne inicializovat scénu do počátečního stavu.

2.2.5 Řazení polygonů

Při pozorném pročtení specifikací algoritmů byla zmínka o předpokladu na setřídění scény v průběhu zpracování scény algoritmem. Bylo zmíněno že Malířův algoritmus předpokládá scénu seřazenou a Z-Buffer krok seřazení scény nepotřebuje, jelikož se dokáže vypořádat s libovolně seřazenou scénou. Požadavek na seřazení scény je poměrně důležitý, jelikož výrazně ovlivňuje rychlost algoritmu. Seřadit velký počet polygonů je poměrně dosti časově náročný proces, který se taktéž připočítává k času zpracování scény. Požadavek na setřídění polygonů scény musí být proto pro Malířův algoritmus a Z-Buffer demonstrován, aby uživatel mohl sledovat, jestli má setřídění vliv na správné vykreslení scény algoritmem.

Je tedy důležité v demonstraci Malířova algoritmu a Z-Bufferu umožnit uživateli zadávat různá seřazení polygonů. Aplikace proto bude realizovat rozhraní pro změnu pořadí polygonů dle následujících kritérií:

- seřadit jako *rostoucí* posloupnost: tedy od nejbližšího polygonu k nejvzdálenějšímu
- seřadit jako *klesající* posloupnost: tedy od nejvzdálenějšího polygonu k nejbližšímu. Klesající posloupnost je vstupním kritériem pro Malířův algoritmus.
- seřadit *náhodně*: jelikož v praxi je nepravděpodobné, že polygony budou jakkoliv seřazeny. Náhodné seřazení tedy nejvíce zrcadlí stav, který nastává v praxi. Proto i náhodné seřazení polygonů je důležité do aplikace zanést. Pro případ, že uživatel bude chtít shlédnout demonstraci znovu, ale pro jinou náhodnou posloupnost by mělo rozhraní umožňovat také i *vygenerování* nové náhodné posloupnosti.

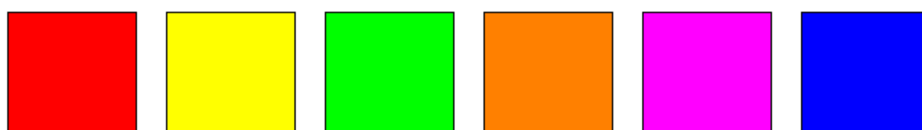
2.3 Polygon scény

Demonstrační scény by měly být co nejvíce jednoduché, proto je důležité se zabývat i otázkou, z čeho se jednotlivé scény budou skládat. Jelikož všechny algoritmy pracují s plochami resp. polygony, bohatě postačí průběh demonstrací provádět na obyčejných polygonech. Oproti demonstrování na složených objektech to tak bude jistým zjednodušením scény, které taktéž povede k větší přehlednosti při demonstrování. V následující části proto bude takový polygon popsán a taktéž budou nadefinovány vlastnosti, které polygon realizuje.

Ve skutečnosti se polygony nepočítají jako objekty scény, pouze jsou prostředkem pro vytváření objektů, nebo-li tvarů. Pro jednoduchost aplikace je však postačující vytvořit jediný tvar, pomocí něhož budou demonstrovány všechny tyto algoritmy. Jako vhodným řešením se pro demonstrační polygon jeví použít tvar čtverce.

2.3.1 Barevné odlišení

Jelikož se polygon bude ve scéně nacházet více než jedenkrát, je vhodné, aby uživatel byl schopný polygony mezi sebou navzájem rozeznat. Pokud by byla scéna složena pouze z jednobarevných polygonů, výsledný obraz vykreslený z těchto polygonů by ztrácel na přehlednosti a demonstrace tak svojí výpovědní hodnotu. Proto by polygon měl umožňovat vytvoření ve více barevných variací. Nahlížením na obsahy pracovního a výsledného pohledu tak uživatel bude jasně vidět, co z polygonu bylo do výsledného pohledu vykresleno.



obr 2.3: Použité barevné kombinace polygonů.

2.3.2 Znázornění vykresleného/nevykresleného polygonu

V aktuálním stavu je už uživatel schopný se orientovat mezi jednotlivými pohledy a pozorovat jednotlivé kroky algoritmu. Avšak nikde není uživateli zvýrazněna informace o tom, který z polygonů byl již vykreslený a naopak. Důležitou vlastností pracovního pohledu je tedy schopnost odlišení mezi vykreslenými a doposud nevykreslenými polygony.

Odlišení na základě stavu zpracování polygonů algoritmem je proto provedeno nastavením průhlednosti polygonu. Vykreslený polygon pracovního pohledu je nastaven jako *neprůhledný*, nevykreslený jako *průhledný*. Takový způsob odlišení vykreslených/nevykreslených polygonů je zcela intuitivní a uživateli by nemělo dělat problém takový způsob odlišení pochopit.

2.3.3 Znázornění pořadí polygonů ve scéně

V případě Malířova algoritmu a Z-Bufferu je potřeba uživateli sdělit, jak jsou polygony ve scéně seřazeny. Tento problém by šel vyřešit například vypsáním pořadí pro jednotlivé polygony, popřípadě vykreslení pořadí polygonů. Problém znázornění pořadí je však vyřešen zavedením rozlišení mezi vykresleným a nevykresleným polygonem a není třeba jej v aplikaci speciálně vytvářet. Tím že aktuálně vykreslený polygon změní svojí průhlednost, uživatel si tak intuitivně vytvoří představu o seřazení scény podle toho, jak jednotlivé polygony přicházejí na řadu.

Jedinou nevýhodou využití konvence znázornění vykreslených polygonů je ta, že uživatel zná pořadí pouze do pozice posledně vykresleného polygonu, nezná pořadí následujících polygonů. Tohle platí samozřejmě pouze v případě náhodně seřazené scény, kdy uživatel nemůže seřazení očekávat.

2.4 Návrh demonstrací

Pro tuto kapitolu bude rozebrán návrh demonstrací pro jednotlivé metodiky, které byly pro aplikaci vybrány.

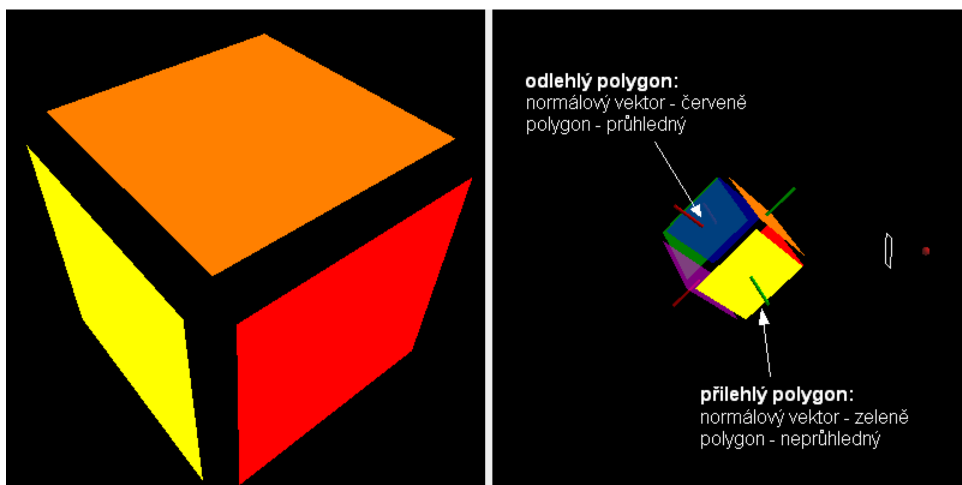
2.4.1 Demonstrace řešení viditelnosti polygonů

Scéna pro demonstraci základní myšlenky odstranění odlehlých stěn by měla předvádět, co je výsledkem vztahu natočení polygonů scény s polohou jejího pozorovatele. Dle vztahu natočení se polygony scény třídí na přilehlé a odlehlé. Tato demonstrace bude tedy provádět třídění polygonů a pro polygony vyznačovat, jestli jsou přilehlé či odlehlé.

Aby uživateli bylo zřejmé, že myšlenka třídění polygonů je postavena na výpočtech z normálových vektorů ploch, musí být pro každý polygon znázorněn i jeho normálový vektor. I když důležitou roli při výpočtu hraje i vektor směru pozorovatele, tento vektor ve scéně nemusí být zvýrazněn, jelikož je díky vyznačení místa pozorovatele ve scéně zřetelný. Proto postačí do pracovního pohledu zvýraznit pouze normálové vektory polygonů. Aby bylo možné normálové

vektory přilehlých a odlehlých polygonů rozeznat, budou tyto vektory mezi sebou barevně odlišeny. Pro přilehlé polygony je vhodné použít barvu zelenou, pro odlehlé polygony barvu červenou. Tato barevná specifikace je všeobecně chápána jako výraz ano/ne, čili uživatel intuitivně pochopí význam obarvení vektoru.

Následně už pouze zbývá si nadefinovat vzhled scény. Pro tuto scénu volím demonstrování na krychli, která patří mezi nejjednodušší prostorové objekty. Pochopení způsobu, jakým dochází k vyřazování polygonů krychle povede k pochopení celé problematiky, jelikož u složitějších objektů probíhá vyřazování zcela stejným způsobem. Avšak i tak jednoduchý objekt má stále tu nevýhodu, že odlehlé stěny jsou vždy překryty stěnami přilehlými. Z čehož vlastně vychází podstata myšlenky odstranění odvrácených polygonů, ale má to za následek, že krychle se všemi vykreslenými stěnami je ve výsledku vzhledově stejná jako krychle s vykreslenými pouze přilehlými stěnami. V takovém případě by nebylo možné ve výsledném pohledu pozorovat rozdíly mezi těmito dvěma případy. Proto navrhuji krychli poupravit tak, aby mezi sousedními stěnami byla vždy drobná mezera, skrz kterou bude možno nahlédnout do krychle a pozorovat jí tak i zevnitř. Navíc krychle by měla být uvedena přinejmenším do rotačního pohybu, protože na statické scéně by nešly pozorovat žádné rozdíly mezi jednotlivými snímky. Krychle proto bude rotovat tak, aby ve scéně docházelo k obměně přilehlých a odlehlých stěn.



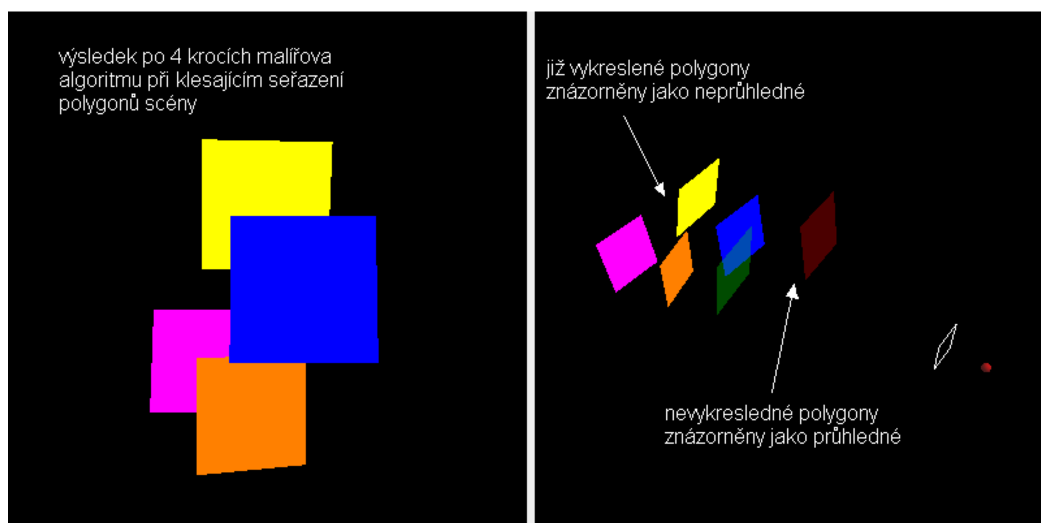
obr 2.4: Ukázka demonstrace třídění polygonů při vyřazení odlehlých stěn.

2.4.2 Demonstrace Malířova algoritmu

Tato scéna by měla demonstrovat zejména hlavní pointu Malířova algoritmu, čímž je vykreslení od nejvzdálenějšího polygonu k nejbližšímu. Uživatel by měl pozorovat, jak se mění výsledné vykreslení v závislosti na seřazení objektů scény, jelikož seřazení polygonů je základním kritériem Malířova algoritmu. Zřetelné musí být zejména to, že scéna je správně vykreslena pouze v případě, kdy objekty jsou seřazeny od nejvzdálenějšího k nejbližšímu a vykresleny v tomto pořadí. Proto scéna realizuje

rozhraní pro řazení polygonů, pomocí něhož může uživatel měnit pořadí polygonů a porovnávat výsledky Malířova algoritmu pro různé stavy seřazení.

Demonstrace začíná s prázdnou scénou. To znamená, že výsledný pohled je zcela prázdný, vyplněn pouze barvou pozadí scény, a pracovní pohled má všechny polygony nastavené jako průhledné. Pomocí krokovacího rozhraní je uživateli umožněno buď krokovat scénu individuálně, nebo spustit krokování s určitým časovým prolínáním. Postupně jak jednotlivý polygony přicházejí na řadu, je do výsledného obrazu vykreslen aktuální polygon, který celou svojí plochou překreslí obsah scény. Polygon, který je aktuálně vykreslený je v pracovním pohledu nastaven jako neprůhledný.



obr 2.5: Ukázka výstupu Malířova algoritmu.

2.4.3 Demonstrace Z-Bufferu

Jak jste se mohli výše dozvědět, na algoritmu Z-Bufferu není absolutně nic složitého. Algoritmus pro Z-Buffer zabírá opravdu jenom minimum programovacího kódu. Aby ale uživatel byl schopný pochopit těchto několika řádků programovacího kódu, musí znát princip, kterým je se Z-Bufferem scéna vykreslována. Uživatel se z demonstrace proto musí dozvědět, jaká oblast dat se do Z-Bufferu ukládá.

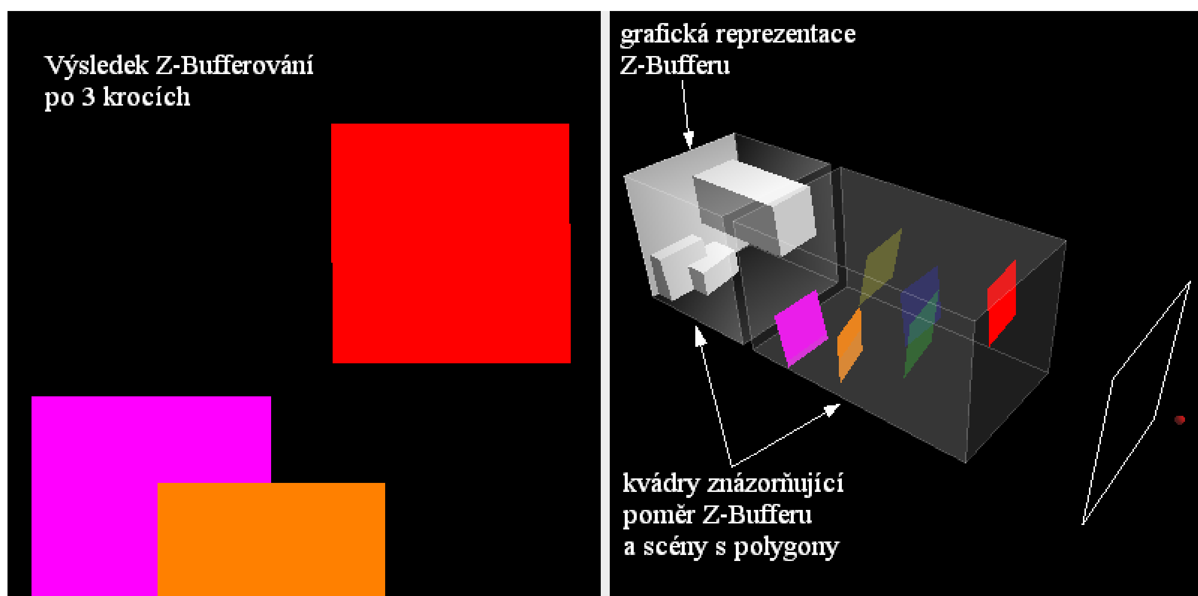
Jelikož je Z-Buffer hloubkovým obrazem vykreslených polygonů, nejvhodnější způsobem je tento hloubkový obraz scény vykreslit. Proto bude Z-Buffer demonstrován vykreslením své plastické podoby do scény. Při průchodem scény tak bude uživatel sledovat, jak se hloubkový obraz mění v závislosti na přibývajících polygonech, které jsou do scény vykresleny.

Bylo tedy stanoveno, že Z-Buffer bude do scény vykreslen. Důležité je si však upřesnit, do kterého místa ve scéně bude umístěn. Jako možným způsobem by šlo Z-Buffer vykreslit do scény přesně v pozici demonstrované scény, tedy aby byl tisknut přesně na ty části polygonů, pro něž uchovává jejich hloubku. Tento způsob by byl sice nejvíce přesný principu Z-Bufferu, ale scéna by se

tím stala nepřehlednou, jelikož by se Z-Buffer navzájem překrýval s polygony demonstrační scény. Musí se proto nalézt taková pozice, ve které Z-Buffer nebude kolidovat s polygony scény, ale stále bude scéně natolik blízko, aby šlo v jednom okamžiku pozorovat scénu zároveň s tvarem Z-Bufferu. I když nemůže být Z-Buffer vykreslen přesně do oblasti s demonstračními polygony, uživatelsky příjemné se jeví jej vykreslit přinejmenším souměrně do prostoru za těmito polygony. Myšleno tím je Z-Buffer vykreslit se stejnými x a y-ovými souřadnicemi, jenom posunutý ve směru z-tové souřadnice o takovou hodnotu, aby ke kolizi se scénou nedocházelo. Z-Buffer bude tedy posunutý za demonstrovanou scénu. Uživatel si může takto vykreslený Z-Buffer porovnávat s aktuálním stavem vykreslených polygonů a snažit se tím zachytit princip, který má být touto scénou demonstrován.

Scéna pro demonstraci Z-Bufferu bude vypadat obdobně jako scéna Malířova algoritmu. Ve scéně se bude nacházet několik polygonů, které dle seřazení postupně přicházejí na řadu a za pomocí Z-Bufferu jsou do scény vykresleny. Grafická reprezentace Z-Bufferu bude přitom ihned překreslena tak, aby byla pro novou podobu scény stále aktuální. Ačkoliv v demonstraci Z-Buffer může taktéž zpracovávat libovolně seřazenou posloupnost polygonů, výsledek vykreslení scény je vždy stejný. Je to dáno faktem, který byl řečen v kapitole 1.5.3. Pozorný uživatel si tak dokáže z výsledků scény jednoduše odvodit nároky na seřazení scény obou algoritmů, jelikož výsledky Malířova algoritmu a Z-Bufferu se budou značně lišit.

Nepříjemnou situací pro aplikaci je věc, že obsahuje dvojici grafických oken, které zabírají poměrně velkou část obrazovky, přitom jednotlivé okna nejsou natolik velká, aby aplikace mohla jakkoliv plýtvat prostorem scény. Pokud by Z-Buffer byl do scény vykreslen ve stejném poměru velikosti jako demonstrační scéna, uživatel by musel pohled poměrně dost oddálit, aby plocha okna zasáhla jak demonstrační polygony, tak i grafické znázornění Z-Bufferu. Oddálenou scénu by ale nešlo pozorovat tak detailně, jako v případě normální velikosti. Aplikace si nemůže dovolit takto plýtvat prostorem scény ani zrakem uživatele, je proto nutné Z-Buffer „uskromnit" a vykreslit jej poměrově smrštěný. Smrštění ale postihne Z-Buffer v oblasti z-tové souřadnice. V osách x a y se žádné transformace provádět nebudou. Aby však uživatel neztratil orientaci ve změně poměru, scéna bude navíc obsahovat dva kvádry, které mají za úkol znázornit poměr prostoru demonstračních polygonů s oblastí pro vykreslení Z-Bufferu. Scéna pro demonstraci Z-Bufferu s poměrovými kvádry je ukázána na obrázku 2.6.



obr 2.6: Ukázka vzhledu scény pro demonstraci Z-Bufferu.

2.4.4 Demonstrace Ray-Castingu

Z podstaty metody Ray-Casting vyplývá, že do demonstrace je kromě vlastním demonstrační polygonů do scény znázornit následující věci:

- *paprsek scény*: znázorňuje dráhu, po které jsou zjišťovány průsečíky se scénou, tj. přímka procházející místem pozorovatele a aktuálním pixelem zobrazovacího plátna. Paprsek je do pracovního pohledu vykreslen pomocí jednoduché čáry, s počátkem v místě pozorovatele a pokračující skrz aktuální pixel až do vzdálenosti, která přesahuje nejvzdálenější hodnotu vůči pozici pozorovatele.
- *průsečíky se scénou*: tj. zvýraznit místa, ve kterých byl zjištěn průsečík s paprskem. Průsečík může být do scény vykreslen pomocí koule, která je dost malá na to, aby nepřekrývala demonstrovanou scénu a dost velká na to, aby šla ve scéně zpozorovat.
- *výběr nejbližšího průsečíku*: nebo-li znázornění uživateli, který ze zjištěných průsečíků byl vybrán pro vykreslení do výsledného pohledu. Nejbližší průsečík znázorněn zelenou barvou, průsečíky další v pořadí znázorněny červenou barvou.

Metoda Ray-Casting je poměrně náročnou a zdlouhavou metodou, jelikož pro jediný snímek zpracovává ohromné množství pixelů, pro něž je potřeba vyhledat barvu. Uvážíme-li že velikost plátna výsledného pohledu je pouhých 400x400 pixelů, znamená to, že metoda zpracovává celkem 16000 pixelů. Protože platí že jeden pixel je zpracováván v jednom kroku, tak nechat uživatele pozorovat takové množství kroků by nebylo příliš zábavným způsobem výuky. Aplikace proto bude

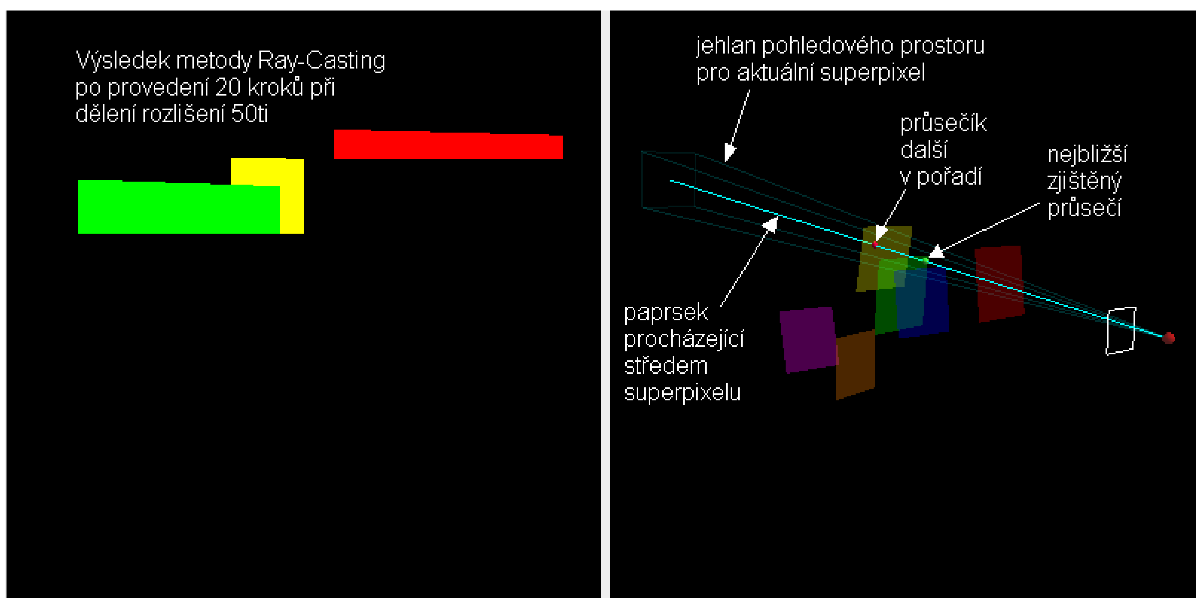
umožňovat degradování rozlišení do tzv. superpixelů. Časová náročnost průběhu metody Ray-Casting degradovaného rozlišení se tak kvadraticky sníží.

Při použití degradace rozlišení musí být ale ustanoveno, jak bude probíhat vykreslování scény pro aktuální superpixel. Superpixel se skládá z $D_x \cdot D_y$ pixelů, kde D značí velikost dělitele původního rozlišení a příslušné indexy pouze zvýrazňují, že superpixel je složen ze stejného počtu *x* a *y*-ových pixelů. Protože paprsek pracovního pohledu může být v jednom kroku vržen pouze na jediný bod scény, nutné je taky ustanovení, do kterého místa bude paprsek vržen. Superpixel si můžeme představit jako plochu čtvercového tvaru. Nejvhodnějším řešením je tak paprsek vrhnout do místa těžiště tohoto čtverce, kterým je pixel nejblíže středu superpixelu. Souřadnice středového pixelu jsou jednoduše vypočteny dle vzorců:

$$x = X \cdot D + D/2$$
$$y = Y \cdot D + D/2 \quad , \qquad \textit{Vzorec 2.1}$$

kde *x, y* jsou souřadnice středového pixelu, *X, Y* jsou souřadnice aktuálního superpixelu a *D* je dělitel rozlišen. I když se zavedením degradace provádí pouze jediný krok metody Ray-Casting, demonstrace bude přesto navržena tak, že kromě středového pixelu budou zpracovány všechny pixely aktuálního superpixelu. Výsledný obraz tak nabude podoby jako v případě nedegradovaného rozlišení.

Zavedení superpixelu má dále za následek to, že pro jeden krok se již nezpracovává oblast ležící na jediné přímce, ale oblast, spadající pod určitý objem. Používá-li metoda perspektivní projekci, je tímto objemem čtyřstranný jehlan s vrcholem v místě pozorovatele a stěnami náležícími okrajům superpixelu. Pokud je uživatelem zvoleno degradování rozlišení, musí se v pracovním pohledu znázornit i tento pohledový objem právě pomocí vykreslení čtyřstranného jehlanu do pracovního pohledu.



obr 2.7: Ukázka vzhledu scény pro demonstraci Ray-Castingu

3 Implementace

V kapitole *implementace* bude představeno prostředí, ve kterém probíhalo vytvoření demonstrační aplikace a taktéž zde budou popsány podstatné části při implementaci programu a jednotlivých demonstrací ve zvoleném prostředí.

3.1 3D API Java 3D

Java 3D tvoří 3D grafickou nadstavbu jazyku Java, stavěnou na existujících technologiích jako je DirectX a OpenGL. Dodávané je v podobě přídavných knihoven, které lze jednoduše doinstalovat. Následující verze Javy je dokonce plánovaná se zavedením Javy 3D jako standardní součást distribuce. Java 3D je volně ke stažení na stránkách www.java3d.dev.java.net.

Knihovna tříd Java 3D poskytuje rozhraní, které je jednodušší než rozhraní mnoha jiných grafických knihoven, ale zároveň je dostačující například k tvorbě dobrých her a animací. Rozhraní Java 3D i přes svoji jednoduchost plně vystačuje požadavkům na aplikaci, proto volím právě jej jako prostředek pro implementaci demonstrační aplikace. Nároky programu totiž po 3D API nevyžadují vysoký výkon, ani pokročilé možnosti v programování počítačové grafiky, navíc jednoduchost tohoto rozhraní může být dokonce i výhodou při objevování světa počítačové grafiky. Je tak jednodušší se více soustředit na základní a podstatné věci oproti věcem, které jsou spíše záležitostí moderní doby. Java 3D taktéž poskytuje kvalitní podporu z hlediska dokumentace a tutoriálu.

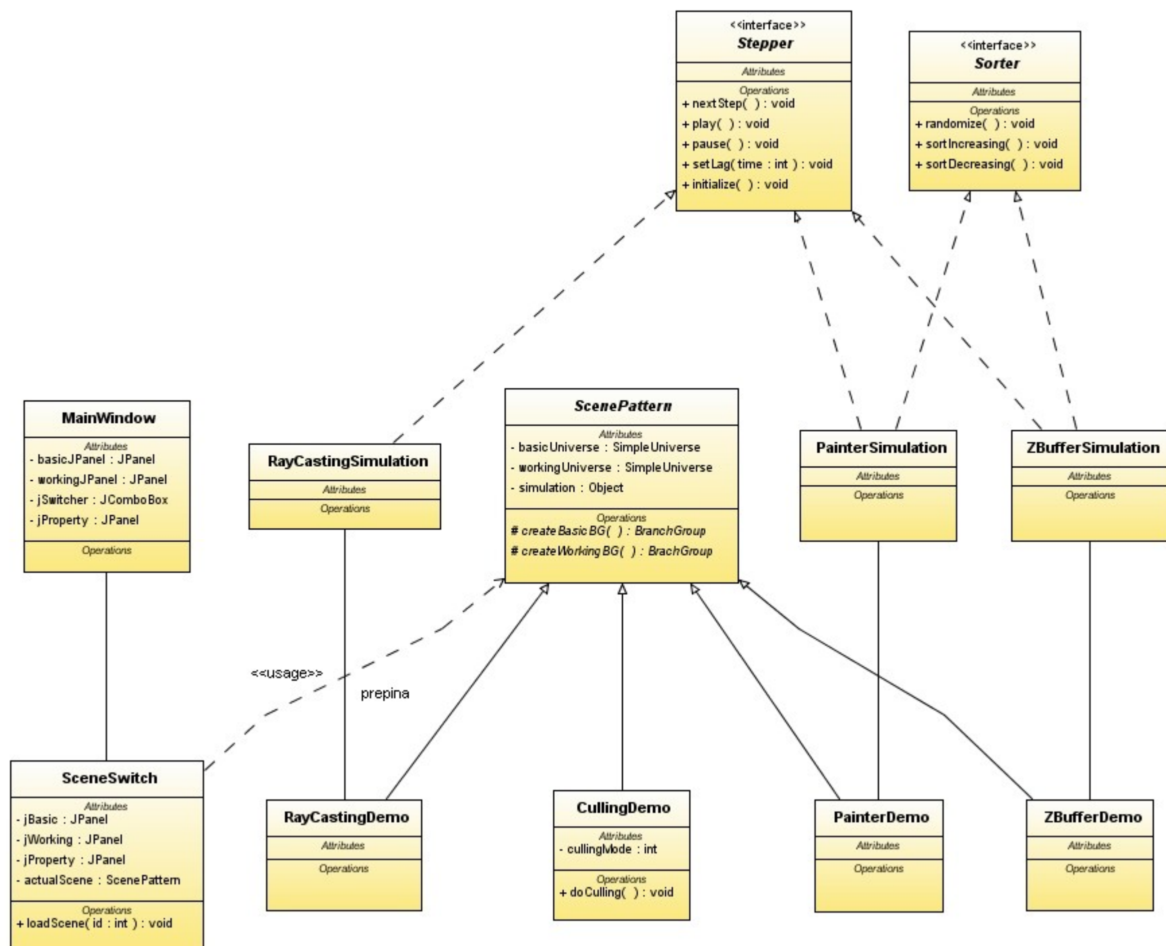
3.2 UML návrh aplikace

Hlavní grafické okno aplikace tvoří třída `MainWindow`. To je složeno ze dvojice kontejnerů pro výsledný a pracovní pohled, jeden kontejner pro ovládací část aplikace a jeden objekt typu `JComboBox` pro přepínání jednotlivých scén.

Třída `MainWindow` vytváří instanci třídy `SceneSwitch`, které se staré o přepínaní jednotlivých scén mezi sebou. Třída `SceneSwitch` přebírá od třídy `MainWindow` trojici zmíněných kontejnerů, do kterých vkládá potřebné objekty. Přepínání scén je prováděno funkcí `loadScene(int id)`. Aktuální scéna je uložena v proměnné `actualScene`, do které aplikace přiděluje instance jedné ze zvolených tříd `CullingDemo`, `PainterDemo`, `ZbufferDemo` nebo `RayCastingDemo`. Tyto třídy vytvářejí demonstrační scény a dále spouštějí spouštějí simulaci nad vytvořenou scénou. Všechny 4 třídy povinně dědí od třídy `ScenePattern`, která vytváří všeobecný vzor pro jakoukoliv demonstrační scénu v aplikaci, tj. nastavuje vlastnosti renderování pro výsledný pohled a provádí režie pracovního pohledu ohledně zobrazení obrysu okna, zobrazení oka, přidání světla do scény a vytvoření ovládání pohledu pomocí myši. Zděděné třídy musí povinně implementovat abstraktní funkce

`createBasicBG()` a `createWorkingBG()`, které vytvářejí `BranchGroup` s obsahem demonstrační scény. Tyto 2 funkce jsou volány v konstruktoru třídy `ScenePattern`. Vytvořené scény jsou uloženy do instancí třídy `SimpleUniverse`, která je v Java3D API základem pro vytváření struktury SceneGraphu.

Kvůli oddělení logiky vytváření demonstrační scény a simulace nad vytvořenou scénou, aplikace vytváří třídy `PainterSimulation`, `ZbufferSimulation` a `RayCastingSimulation`, které simulaci provádí. Tyto zmíněné tři třídy podle potřeby implementují rozhraní `Stepper` a `Sorter`. Rozhraní `Stepper` definuje funkce pro krokování programu a rozhraní `Sorter` definuje funkce pro třídění polygonů scény. Každá ze tříd `PainterDemo`, `ZbufferDemo` a `RayCastingDemo` logicky vytváří instanci pouze té odpovídající simulace, která jí náleží.



obr 3.1: UML diagram základní struktury programu.

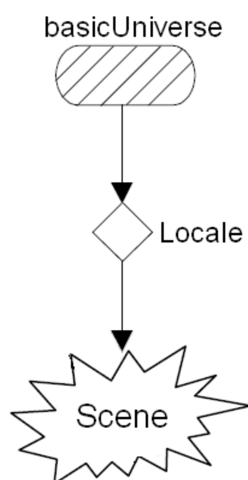
3.3 Základní struktura scén

Základní strukturu scén poskytuje výše zmíněná třída s názvem `ScenePattern`. Ta pro výsledný a pracovní pohled aplikace vytváří struktury scenegraphů, do kterých pak zděděné třídy přidávají

vlastní demonstrační scény. V této kapitole budou ukázány a popsány struktury a implementace scenegraphů. Jednotlivá místa pro přidání scény jsou ve scenegraphu taktéž vyznačena.

3.3.1 Scenegraph pro výsledný pohled

Scenegraph pro výsledný pohled vytváří strukturu složenou čistě jenom ze samotné demonstrační scény, jak vyplývá z obrázku 3.2. Realizování takové struktury je provedeno pouhým vytvořením instance třídy `SimpleUniverse` a vložením demonstrační scény do instance funkcí `addBranchGraph()`.



obr 3.2: Struktura scenegraphu pro výsledný pohled.

3.3.2 Scenegraph pro pracovní pohled

Scenegraph pracovního pohledu je strukturován už o něco více složitěji. Kromě samotné demonstrační scény, která svojí strukturou odpovídá demonstrační scéně výsledného pohledu, dále obsahuje:

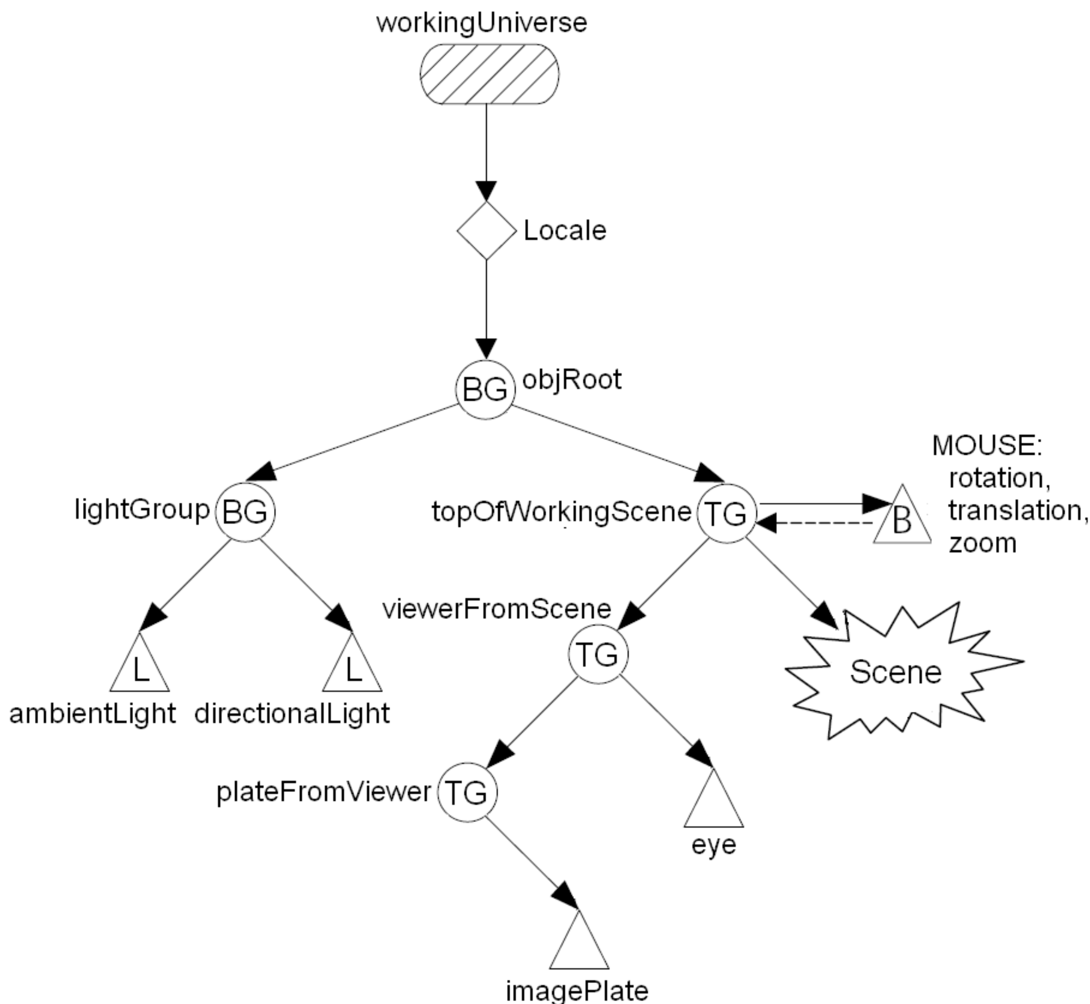
- *ambientní a směrové světlo*: toho využívá zejména scéna s demonstrací Z-Bufferu. Vykreslený Z-Buffer nabývá komplexnějšího tvaru a bez použití světla ve scéně by tento tvar nebyl zřejmý. Tím že je Z-Buffer za pomocí světla vystínován, nabývá teprve dojmu plastické podoby. Demonstrované polygony však světla ve scéně nevyužívají. Tyto polygony mají mezi sebou odstup a navíc jsou mezi sebou barevně odděleny, uživatel nabude informace o jejich tvaru a poloze přirozeně z perspektivy.
- *behavioury pro ovládání myší*: kvůli rotaci pracovního pohledu. Tyto behavioury umožňují libovolný pohyb nad pracovní scénou za pomocí myši. Pracují nad `TransformGroup`[1] s

1 `TransformGroup` je název třídy v API Java 3D, která ve scenegraphu uchovává transformace aplikované na všechny objekty náležící pod touto `TransformGroup`. Dále v textu bude použito pojmenování „skupina transformací“.

pojmenovaním `topOfScene`, pod který se přidávají všechny objekty, nacházejí v pracovním pohledu. Behavioury jsou celkem 3 a jedná se o:

1. **rotace scény**: standardní rotační behaviour. Při stisku levého tlačítka a pohybu myši je scéna intuitivně rotována kolem počátku souřadnic.
2. **posun scény**: umožňuje posouvat celkový obraz scény ve směrech nahoru, dolů, popřípadě doleva, doprava. Provádí se stisknutím pravého tlačítka a pohybem myši ve zvolených směrech.
3. **zoom**: kvůli přiblížení/oddálení pohledu na scénu. Zoomuje se pomocí pohybu scroolovacího kolečka myši.

- *bod pozorovatele*: nebo-li místo, ze kterého uživatel nahlíží na scénu výsledného pohledu. Bod pozorovatele je znázorněn koulí a jeho transformace od scény je ekvivalentní transformaci výsledného pohledu pro pozici pozorovatele. Transformaci provádí transformační skupina `viewerFromScene`. Transformace pozorovatele je uložena ve třídě `ViewingPlatform` daného `SimpleUniverse`. Transformační matici si lze přečíst funkcí `getViewPlatformTransform()`.
- *zobrazovací plátno*: to je znázorněno jednoduchým obrysem stěn čtverce. Velikost stěn zobrazovacího plátna si lze jednoduše odvodit, jelikož Java3D standardně nastavuje jeho velikost tak, že čtverec postavený do počátku souřadnic o straně délky 2 metry při projekci zaplní celou plochu plátna. Pro paralelní projekci je strana plátna jednoduše délky 2m. Pro perspektivní projekci se velikost však musí dopočíst. Zobrazovací plátno je od pozorovatele transformováno ve vzdálenosti dané nastavení prostředí Java 3D. Provádí se funkcí `getNominalEyeOffsetFromNominalScreen()` instance třídy `PhysicalBody` daného `SimpleUniverse`.



obr 3.3: Struktura scenegraphu pro pracovní pohled.

3.3.3 Implementace základní struktury demonstrací

Zde budou popsány nejdůležitější rysy abstraktní třídy `ScenePattern`, tvořící vzorovou šablonu pro vytváření tříd demonstračních scén.

Jako první bude rozebrán konstruktor, který je napsán v bloku 3.1 pro kód. Na řádcích 1-14 probíhá inicializace a vytvoření `SimpleUniverse` pro výsledný pohled. Vytvoření `SimpleUniverse` je provedeno na řádku 1. Následně, na řádcích 3-5 je provedeno rozhodování o způsobu projekce výsledného pohledu. Rozhoduje se dle proměnné `projection`, kterou konstruktor přejímá jako parametr. Jelikož projekce je standardně nastavena jako perspektivní, volání funkce `setProjectionPolicy()` probíhá pouze při zjištění požadavku na paralelní projekci. Na řádcích 6-13 probíhá inicializace způsobu renderování do výsledného pohledu. Ta umožňuje výsledný pohled ponechat v klasickém renderovacím módu, nebo jej přepnout do tzv. immediate módu, ve kterém je renderování scény pozastaveno. Proces renderování je pak řízen samostatně. Immediate mód je

aktivován voláním funkce stopRenderer() dané instance třídy Canvas3D. I v případě požadavku na immediate mód je na řádcích 10-12 do SimpleUniverse vložena demonstrační scéna.

Po inicializaci výsledného pohledu pak následně na řádcích 14-18 probíhá inicializace pracovního pohledu. Ta už neumožňuje nastavení projekce pohledu, ani nastavení immediate módu, takže je kromě výše zmíněných operací v podstatě totožná s vytvářením výsledného pohledu. Je zde ale navíc na řádku 16 volána funkce prepareWorkingUniverse(), která má za úkol vytvoření struktury scenegraphu pro pracovní pohled, známé z obrázku 3.3. Vytvořená demonstrační scéna pracovního pohledu je záhy přidána pod transformační skupinu topOfWorkingScene, jejíž inicializace a strukturace probíhá právě ve funkci prepareWorkingUniverse().

```
1.      Canvas3D c = createUniverse(BASIC_SCENE);
2.      BranchGroup bg;
3.      if (projection == View.PARALLEL_PROJECTION) {
4.        c.getView().setProjectionPolicy(projection);
5.      }
6.      switch (mode) {
7.        case BASIC_IMMEDIATE_MODE:
8.          c.stopRenderer();
9.        case BOTH_RENDER_MODE:
10.         bg = createBasicBG();
11.         jBasic.add(c, BorderLayout.CENTER);
12.         basicUniverse.addBranchGraph(bg);
13.     }
14.     c = createUniverse(WORKING_SCENE);
15.     jWorking.add(c, BorderLayout.CENTER);
16.     prepareWorkingUniverse();
17.     bg = createWorkingBG();
18.     topOfWorkingScene.addChild(bg);
```

Kód 3.1: Konstruktor třídy ScenePattern

Funkce prepareWorkingUniverse() sice vytváří podstatnou část při konstrukci struktury scenegraphu pracovního pohledu, avšak její popis by zde byl zbytečný a pouhým plýtváním místem, protože by zde bylo řečeno to samé, co v kapitole 3.3.2. I přesto se hodí si aspoň ukázat, jakým způsobem jsou zjišťovány potřebné metriky, tj. odstupy a rozměry virtuálního plátna a pozice pozorovatele od scény.

V první části je zjištění transformace, se kterou budou oko pozorovatele a obrys plátna transformovány od scény. Průběh zjišťování je proveden na řádcích 1-4 bloku 3.2. Funkcí getViewPlatformTransform() se zjistí nastavení transformace v prostředí výsledného pohledu a dle stejné transformační matice jsou oko s plátnem transformovány od demonstrační scény. Na řádku 5 se zjišťuje nastavení projekce pro výsledný pohled. Toto nastavení hraje důležitou roli při výpočtu délky strany rámu. Pokud je použita paralelní projekce, délka strany je jednoduše 2[2], v případě perspektivní

2 Takové nastavení je standardní v prostředí Java 3D. To totiž definuje velikost plátna tak, že čtverec délky strany 2 vyplní celou plochu obrazu.

projekce se musí délka strany ještě vypočíst. Vypočtení délky strany probíhá na řádcích 9-10. Nejdříve je však na řádku voláním funkce `getNominalEyeOffsetFromNominalScreen()` zjištěna vzdálenost mezi pozorovatelem a plátnem. Při použití perspektivní projekce Java 3D nedefinuje šířku zobrazovacího plátna, ale zorný úhel pohledu. Ten je přečten na řádku 9 za voláním funkce `getFieldOfView()`. Délka strany plátna se vypočte za pomocí údajů vzdálenosti pozorovatele od plátna a zorného úhlu pohledu použitím goniometrické funkce tangens a výpočetním vztahem mezi předponami v pravoúhlém trojúhelníku.

$$\tan(\gamma/2)=\frac{c/2}{v},$$ *Vzorec 3.1:*

kde γ je zorný úhel pohledu, v je vzdálenost mezi pozorovatelem a plátnem a c je délka strany zobrazovacího plátna. Tu si ze vzorce jednoduše vytkneme, takže nám vznikne vzorec pro výpočet délky strany.

$$c=\tan(\gamma/2)\cdot v\cdot 2,$$ *Vzorec 3.2:*

```
1.      TransformGroup basicTG = basicUniverse.getViewingPlatform().
                                 getViewPlatformTransform();
2.      Transform3D transformViewerFromScene = new Transform3D();
3.      basicTG.getTransform(transformViewerFromScene);
4.      viewerFromScene = new TransformGroup(transformViewerFromScene);
5.      int projectionPolicy = basicUniverse.getCanvas().
                               getView().getProjectionPolicy();
6.      double distanceBetweenViewerAndPlate =
          basicUniverse.getCanvas().getView().getPhysicalBody().
          getNominalEyeOffsetFromNominalScreen();
7.      double side;
8.      if (projectionPolicy == View.PERSPECTIVE_PROJECTION) {
9.        double fieldOfView = basicUniverse.getCanvas().
                               getView().getFieldOfView();
10.       side = Math.tan(fieldOfView/2)*distanceBetweenViewerAndPlate*2;
11.     } else { side = 2.0; }
```

Kód 3.2: Zjištění a výpočet potřebných metrik výsledného pohledu.

3.4 Implementace zvolených algoritmů

Nyní si probereme, jak jsou implementovány jednotlivé principy zvolených algoritmů.

3.4.1 Vyřazení odvrácených stěn

Poměrně jednoduchý problém pro implementaci, jelikož možnost vyřazení odvrácených / přivrácených / žádných stěn je standardní vlastností většiny grafických prostředí.

Aplikace z jednotlivých ploch seskládá tvar pootevřené krychle, která je navíc vložena pod rotační behaviour. Rotaci s objekty poskytuje třída `RotationInterpolator`, která rotuje vždy kolem

osy *y* s přidělenou transformační skupinou. Rotování probíhá v časových intervalech definovaných třídou `Alpha`. Vytvoření instance třídy `TransformGroup` je provedeno na řádku 1. Třída `Alpha` umožňuje mj. i nastavení doby, za jakou dojde k jednomu orotování dokola, tj. o 360°. Instance třídy `Alpha` je na řádku 2 vytvořena pod názvem `rotationAlpha` s vlastností neustálého rotování (-1 v prvním parametru) a jedním cyklem za 6 sekund (tj. 6000ms). Na řádcích 3-7 je nad zvolenou transformační skupinou `objSpin` vytvořena instance třídy `RotationInterpolator`, která je přidána pod `objRotate`. Cyklus `for` v bloku 8-11 postupně načítá jednotlivé plošky pod `objSpin`, který je vzápětí na řádku 12 přidány pod `objRotate`. Načítaným typem jsou instance třídy `TransformGroup`, které uchovávají informace o transformacích do požadovaného tvaru.

```
1.      TransformGroup objSpin = new TransformGroup();
2.      Alpha rotationAlpha = new Alpha(-1, 6000);
3.      RotationInterpolator rotator =
4.         new RotationInterpolator(rotationAlpha, objSpin);
5.      BoundingSphere bounds = new BoundingSphere(new Point3d(), 100.0);
6.      rotator.setSchedulingBounds(bounds);
7.      objRotate.addChild(rotator);
8.      for (int i = 0; i < 6; i++) {
9.        objSpin.addChild(tg[i]);
10.     } //end for
11.     objRotate.addChild(objSpin);
```

Kód 3.3: Část kódu pro vytvoření rotačního behaviouru nad zvolenou transformační maticí

Taková krychle je vytvořena jednou pro pracovní a jednou pro výsledný pohled. Pro pracovní pohled pole transformačních skupin `tg[i]` ze řádku 9 navíc obsahuje i graficky znázorněný vektor pro danou plochu.

Třídění ploch ve výsledném pohledu probíhají autonomně dle nastavení vlastností těchto ploch. To se provádí nastavením `PolygonAttributes` pro daný objektu. Nastavení vlastnosti třídění polygonů se provádí funkcí `setCullFace(int cullFace)` pro danou instanci `PolygonAttributes`. O zbytek se už stará renderovací funkce Javy 3D. Proto stačí pouze zařídit průběh třídění v pracovním pohledu. To probíhá každých 5ms, v nichž jsou pro aktuální pozici natočení vypočteny hodnoty směrového vektoru pozorovatele a normálové vektory ploch, ze kterých je zjištěno, jestli je plocha odlehlá či přilehlá. Kód pro třídění polygonů je ukázán v bloku 3.4. V kódu jsou z důvodu úspory vynechány části pro vypočtení směrového a normálového vektoru. Jejich pozice je vyznačena komentáři na řádkách 1,2. Úhel mezi směrovým a normálovým vektorem je vypočten na řádku 3. Na základě vypočteného úhlu je vzhled pracovní scény uveden do stavu odrážející aktuální stav výsledného pohledu. To provádí pro jednotlivé plochy dvojice funkcí `isFrontFace(int i)` a `isBackFace(int i)`, které jsou volány na základě velikosti úhlu. Úhel je porovnáván na řádku 4. Pokud je úhel menší než 90° (plocha je přilehlá), volá se funkce `isFrontFace(int i)` (řádek 5). V opačném případě je plocha odlehlá a volá se funkce `isBackFace(int i)`.

```
1.      // vypocet smeroveho vektoru: directionVector
2.      // vypocet normaloveho vektoru: normalVector
3.      angle = directionVector.angle(normalVector);
4.      if (angle > Math.PI / 2) { // prilehla
5.        isFrontFace(i);
6.      } else { // odlehla
7.        isBackFace(i);
8.      }
```

Kód 3.4: Třídění polygonů podle hodnot úhlů normály a vektoru pozorovatele.

3.4.2 Malířův algoritmus

Implementace demonstrace Malířova algoritmus není zase tak složitým problémem, avšak si vyžaduje vlastní řízení renderování polygonů do scény. Třída pro demonstraci Malířova algoritmu proto přepíná výsledný pohled do immediate módu voláním požadovaného tvaru konstruktoru nadřazené třídy `ScenePattern`. Přepnutí výsledného pohledu do immediate módu však není jedinou záležitostí ohledně nastavení prostředí. Defaultně totiž Java 3D využívá renderování do depth bufferu. To by mělo na následek sice správné vykreslení polygonů v relaci jejich vzdálenosti, ale nesprávné chování metody Malířův algoritmus. Depth buffer se tak musí taktéž vypnout, aby prostředí Java 3D umožnilo věrohodnou simulaci Malířova algoritmu. Proces vypnutí depth bufferu je proveden na řádkách 1-7. Na řádcích 1-2 se načítá grafický kontext canvasu výsledného pohledu. Řádky 3-6 vytvářejí `Appearance` s požadovanou vlastností vypnutého depth bufferu. Řádek 7 přiřazuje tuto `Appearance` načtenému grafickému kontextu. Ve výsledku tak při renderování v immediate módu nebude prostředí Java 3D řešit problém viditelnosti objektů a přenechá tento problém samotnému programátorovi.

```
1.      Canvas3D canvas = basicUniverse.getCanvas();
2.      GraphicsContext3D gc = canvas.getGraphicsContext3D();
3.      RenderingAttributes ra = new RenderingAttributes();
4.      ra.setDepthBufferEnable(false);
5.      Appearance ap = new Appearance();
6.      ap.setRenderingAttributes(ra);
7.      gc.setAppearance(ap);
```

Kód 3.5: Vypnutí depth bufferu grafického kontextu výsledného pohledu.

Tato demonstrace si dále vyžaduje režie ohledně definování pořadí polygonů ve scéně. Definování pořadí umožňuje třída `WaitingList` (tzv. pořadač), která je součástí aplikace. Třída `WaitingList` v parametru přebírá pole s uchovávanými objekty a také pole s definujícím pořadím uchovávaných objektů. Demonstrace Malířova algoritmu pro zadávání pořadí ale využívá třídy `RenderingSequence`, která ze třídy `WaitingList` dědí a dále ji rozšiřuje o podpůrné prostředky pro renderování objektů. Jelikož je v demonstraci Malířova algoritmu řízeno vlastní renderování, pojďme

se podívat, jak toto renderování vypadá. Pro renderování je využita třída `Renderer`, ta je taktéž součástí aplikace. Tato třída spouští vlastní vlákno, ve kterém v krátkých časových intervalech (cca 50ms) volá funkci `render()` provádící renderování sekvence polygonů do grafického kontextu požadovaného okna. Na řádku 2 je obsah grafického kontextu `gc` nejprve před dalším cyklem renderování vymazán. Cyklus `while` na řádcích 4-10 postupně načítá ty polygony z pořadače, které mají být do okna vyrenderovány. Pro ty si zjistí jejich celkovou transformaci ve scéně, kterou uloží do transformační matice `wholeTransform`. Blok pro zjištění transformace se provádí na pozici řádku 6 a v kódu je vynechán. Na řádku 8 je grafickému kontextu zadána požadovaná transformace a řádek 9 už provádí vlastní vykreslení aktuálního objektu do grafického kontextu výsledného pohledu.

```
1.      void render() {
2.        gc.clear();
3.        Shape3D sh;
4.        while ((sh = sequencer.getShapeSetNext()) != null) {
5.          wholeTransform.setIdentity();
6.          // zjisteni transformace polygonu ve scene, ulozena v wholeTransform
7.          }
8.          gc.setModelTransform(wholeTransform);
9.          gc.draw(sh.getGeometry());
10.       }
11.       canvas.swap();
12.     }
```

Kód 3.6: Jednoduchá renderovací funkce.

Při provedení jednoho kroku Malířova algoritmu pak stačí vykonat následující blok 3.7. Na řádku 1 je zjištěno pořadí následujícího polygonu v posloupnosti. V případě že se již prošlo všemi polygony, indexem je -1. Index se testuje na řádku 2. V případě, že posloupnost obsahuje ještě další polygon v pořadí, je patřičný polygon na odpovídajícím indexu nastaven jako neprůhledný (řádek 3) a renderování povoleno o další polygon ve posloupnosti (řádek 4).

```
1.      int indexOfNext = renderedScene.getIndexOfNext();
2.      if (indexOfNext != -1) {
3.        workingSquares[indexOfNext].setTransparency(BasicShape.OPAQUE);
4.        renderedScene.renderNext();
5.      }
```

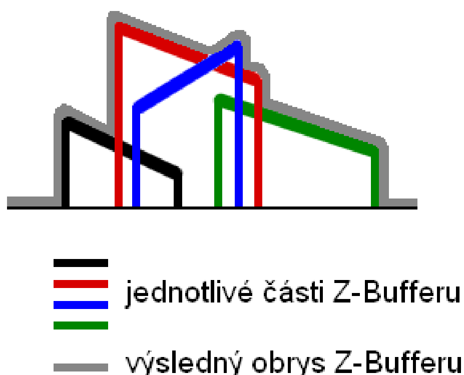
Kód 3.7: Režie při provádění následujícího kroku Malířova algoritmu.

3.4.3 Z-Buffer

Průběh demonstrace Z-Bufferování je proces velmi podobný procesu demonstrování Malířova algoritmu. Nebude tu proto rozebírán z pohledu, ze kterého je popisována implementace Malířova algoritmu. Demonstrace Z-Bufferu nevyužívá immediate módu pro výsledný pohled, renderování tak

ponechává v režii Javy 3D. Má-li být vyznačeno vykreslení polygonu do výsledného pohledu, je tak provedeno řízením viditelnosti polygonu, který se nachází ve scéně výsledného pohledu.

Důležitějším bodem je však zabývat se, jak probíhá vykreslení Z-Bufferu do pracovního pohledu. Uvážíme-li, že Java 3D poskytuje standardní prostředky pro vytváření 3D objektů, naskytuje se možnost si pro aktuální stav scény Z-Buffer přečíst a podle hodnot hloubek jednotlivých pixelů vytvořit jeho 3D podobu. Bylo by to možné řešení, akorát v případě rozlišení 400x400 pixelů by nám tak vznikl objekt složený ze 32000 trojúhelníků popřípadě 16000 čtverců. Je zcela jasné, že vykreslit Z-Buffer z tak obrovského počtu polygonů by bylo neskutečným plýtváním pamětí a výkonu počítače. Navíc Z-Buffer by musel být nejprve přečten, což by se taktéž podepsalo na ztrátě výkonu. Naštěstí je tu možnost Z-Buffer vykreslit jednodušeji a to dokonce i bez znalosti jeho aktuálního stavu. Demonstrační scéna je složena z pouhé hrstky polygonů, o kterých známe jak jejich souřadnice bodů lokálního prostoru, tak i transformace souřadnic do virtuálního prostoru. Můžeme tak zvlášť pro každý polygon vytvořit jeho odpovídající vzhled Z-Bufferu. Končená podoba Z-Bufferu se pak jednoduše seskládá z těch částí pro polygony, které jsou ve scéně již vykresleny. Více však řekne obrázek 3.4.



obr 3.4: Princip vykreslení Z-Bufferu do scény.

Aby mohl být Z-Buffer takto jednoduše vykreslen, musí být výsledný pohled přepnut do paralelní projekce. V perspektivní projekci by se navíc musely dopočítávat počáteční souřadnice, Z-Buffer by totiž nevystupoval kolmo ke svému počátku. To by pouze vedlo ke snížení přehlednosti demonstrace, proto demonstrace Z-Buffer jako jediná ve výsledném pohledu používá paralelní projekci. Vytvoření části Z-Bufferu z polygonu provádí funkce `processZBuffPart()` ukázána ve bloku 3.8. Ta předpokládá 2 parametry, těmi jsou polygon a jeho transformace ve scéně. Výpočet souřadnic části Z-Bufferu probíhá v cyklu `for` na řádcích 3-10. Cyklus postupně prochází všemi body zvoleného polygonu, které transformuje do globálních souřadnic. Načtení a transformace souřadnic probíhá na řádcích 4-5. Z-tová souřadnice se však musí nejprve transformovat z prostoru demonstrační scény do prostoru pro vykreslení Z-Bufferu. Oba dva prostory jsou definovány mezními okraji, do proměnné `ratio` na řádku 6 je vypočten poměr souřadnice z v prostoru demonstrační scény, ze kterého je na řádku 7 vypočtena nová z-tová souřadnice. Zde však ještě

nedochází ke transformaci části Z-Bufferu do oblasti pro její vykreslení. Transformaci provádí až nadřazená funkce. Zde je pouze přepočtena z-tová hodnota mezi lokálními souřadnicemi obou prostorů s ohledem na poměr délky mezi oběma prostory. Na řádku 11 pak funkce navrací tvar části Z-Bufferu. Pro vytvoření částí Z-Bufferu slouží třída `ZbuffPart`, která je součástí aplikace.

```
1.      ZBuffPart processZBuffPart(Transform3D t, EasySquare e) {
2.        // inicializace promennych
3.        for (int i = 0; i < count; i++) {
4.          qa.getCoordinate(i, p);
5.          t.transform(p);
6.          ratio = ((float) p.z - BACK_Z_LIMIT) / LENGTH_OF_AREA;
7.          p.z = ratio * LENGTH_OF_ZBUFF;
8.          // osetreni souradnic x,y zasahujici mimo prostor Z-Bufferu
9.          pts[i] = new Point3f(p);
10.       }
11.       return new ZBuffPart(pts);
12.     }
```

Kód 3.8: Funkce pro vytvoření grafické části Z-Bufferu.

Samotná simulace Z-Bufferování už má podobný průběh jako simulace Malířova algoritmu s tím, že v režii simulace je i kromě nastavování průhlednosti polygonů pracovní scény i nastavování průhlednosti částí Z-Bufferu.

3.4.4 Ray-Casting

Tato demonstrace sice vytváří úplně stejnou demonstrační scénu jako demonstrace Malířova algoritmu a Z-Bufferu, avšak pro výsledný pohled nepoužívá seznam polygonů takovým způsobem, jako tyto dvě předchozí scény. Pro výsledný pohled musí být znázorněno, jak jsou vykreslovány samotné pixely, popřípadě shluky pixelů. Musí tak existovat možnost, jak si hodnotu barvy průsečíku pro daný pixel přečíst. Java 3D naštěstí poskytuje prostředky pro zjišťování průsečíků se scénou. Takovou schopnost umožňuje například třída `PickCanvas`. Zjišťování průsečíků se provádí zadáním *x* a *y* souřadnice, pro něž se průsečíky počítají.

Za pomocí třídy `PickCanvas` se tak počítají průsečíky výsledného pohledu s demonstrační scénou nacházející se v tomto pohledu. I když tedy výsledný pohled slouží k vykreslení jednotlivých bodů, musí i tak obsahovat demonstrační polygony. Kdyby se ve scéně výsledného pohledu demonstrační polygony nenacházely, třída `PickCanvas` by pak stěží vypočetla nějaké průsečíky. Aby tyto polygony ve výsledném pohledu nijak nepřekážely, jsou nastaveny jako neprůhledné. Samotné body se pak vykreslují voláním funkce `drawLine(x1,y1,x2,y2)`[3]. Část pro vykreslení bodu je ukázána ve bloku 3.9. Na řádku 1 je přečtena barva aktuálního pixelu, která je nastavena pro kreslení

3 Bod je funkcí `drawLine` vykreslen jako přímka s počátečním bodem shodným s bodem koncovým. Nepodařilo se mi totiž nalézt způsob, jak se vykreslují samotné body.

grafického kontextu (proměnná `g2D`) výsledného pohledu. Na řádku 2 je vykreslen bod na souřadnice tohoto pixelu.

```
1.      g2D.setPaint(colorBuffer[actY][actX].get());
2.      g2D.drawLine(actX, actY, actX, actY);
```

Kód 3.9: Vykreslení bodu do výsledného pohledu.

Průběh simulace tak postupně projíždí jednotlivými pixely, pro které zjišťuje a vykresluje průsečíky se scénou. Proces výpočtu průsečíků třídou `PickCanvas` má však jednu nevýhodu a tou je značně velký nárok na výpočet, který při volbě použití superpixelu může být uživatelsky poměrně znatelný. Průsečíky proto nejsou počítány v průběhu demonstrace, ale při spuštění demonstrace je na začátku vytvořen obraz průsečíků celé scény a při průchodu jednotlivými kroky se průsečíky čtou z vytvořeného obrazu. Barevné složky obrazu jsou uloženy do dvojrozměrného pole `colorBuffer[y][x]`, hloubky průsečíků uchovává trojrozměrné pole `depthBuffer[y][x][i]`. Pole je trojrozměrné z důvodu potřeby uložit hloubky všech průsečíku, nejenom těch nejbližších. Blok 3.10 ukazuje způsob naplnění položky depth a color bufferu. Řádek 1 nastavuje aktuální pozici pixelu, pro kterou jsou na řádku 2 počítány průsečíky se scénou. Pokud je vypočten aspoň jeden průsečík, pokračuje se na řádcích 4-11. Řádek 4 načítá nejbližší průsečík, tj. z indexu [0], jehož barvu na řádku 6 ukládá do color bufferu. Řádky 7-11 jsou už spojeny s režií načítání všech hloubkových údajů průsečíků do depth bufferu.

```
1.      pickCanvas.setShapeLocation(x, y);
2.      PickResult[] result = pickCanvas.pickAllSorted();
3.      if (result != null) {
4.        PickIntersection intersection = result[0].getIntersection(0);
5.        Color4f color4f = intersection.getPointColor();
6.        colorBuffer[y][x] = new Color3f(color4f.x, color4f.y, color4f.z);
7.        depthBuffer[y][x] = new float[result.length];
8.        for (int i = 0; i < result.length; i++) {
9.          intersection = result[i].getIntersection(0);
10.         depthBuffer[y][x][i] = (float) (-intersection.getDistance());
11.       }
12.     }
```

Kód 3.10: Výpočet průsečíků pro aktuální pixel obrazu.

Tento blok je pro jednotlivé pixely obrazu proveden pouze jedenkrát. Následně se čte pouze pouze z polí `colorBuffer` a `depthBuffer`. Při zpracování jednoho kroku jsou tedy čteny údaje o pixelu (popř. superpixelu) a taktéž je i natočen paprsek do odpovídající pozice. Způsob implementace a zpracování paprsku bude popsána v následujícím odstavci.

Paprsek je vytvořen pomocí třídy `Ray`, která je součástí aplikace. K paprsku je také vykreslen pohledový jehlan. Ten je implementovaný třídou `Pyramid`. Pohledový jehlan je zobrazen v případě použití superpixelu, jinak je transparentní. Taktéž umožňuje nastavení pohledového úhlu, který

provádí funkce `setfieldOfView(double fieldOfView)`. Paprsek i pohledový jehlan jsou uloženy pod stejnou transformační skupinu, která je pouze rotována kolem os *x* a *y* tak, aby tuto dvojici natočila na aktuální pozici. Počátek paprsku i jehlanu je samozřejmě umístěn do místa bodu pozorovatele scény a obojí směřují do scény souměrně s osou *z*, tzn. protínají střed virtuálního plátna. Natočení paprsku provádí funkce `processRay()`, jehož část je ukázaná v bloku 3.11. Řádky 1-2 počítají středovou hodnotu souřadnice aktuálního superpixelu, tj. určeného souřadnicemi `superX` a `superY`. Na řádcích 3-5 je vypočtena rotační matice pro paprsek. Hodnoty `initXAngle` a `initYAngle` značí úhel horního levého rohu plátna s osou *z*. Z pozice aktuálního pixelu a velikostí úhlu mezi dvěma sousedními pixely je tak odvozena hodnota aktuálního natočení paprsku. Rotační matice je na řádku 6 zapsána do transformační skupiny, pod kterou se nacházejí paprsek i jehlan.

```
1.      actX = superX * divisor + (int) divisor / 2;
2.      actY = superY * divisor + (int) divisor / 2;
3.      yAngle.rotY(initXAngle - actX * delta);
4.      xAngle.rotX(initYAngle - actY * delta);
5.      xAngle.mul(yAngle);
6.      rotateRayTG.setTransform(xAngle);
```

Kód 3.11: Výpočet natočení paprsku v pracovním pohledu.

Doposud však nebyl zmíněný způsob vykreslení průsečíků do scény pracovního pohledu. Abychom mohli vykreslit průsečíky na patřičné pozice, potřebujeme znát jejich polohu v lokálních souřadnicích demonstrační scény pracovního pohledu. V současném bodě ale informace o průsečících není známa. Jsou známy pouze hloubky průsečíků pro jednotlivé pixely a barva nejbližšího průsečíku. Ke zjištění polohy průsečíku můžeme tak využít dvojích nejvíce se naskytujících způsobu:

1. Odvodit průsečíky z hodnot souřadnic pixelů a jejích hloubek
2. Použít třídu PickCanvas pro výpočet průsečíků

A nebo také nemusíme využít ani jednoho způsobu, jelikož hodnotu průsečíků znát vůbec ani nemusíme. Je zřejmé, že průsečíky budou vždy ležet na paprsku, jelikož ty vyjadřují průsečíky tohoto paprsku se scénou. V lokálním prostoru paprsku vede přímka paprsku z bodu `[0,0,0]` do bodu `[0,0,délka]`. O natočení paprsku se pak stará transformační skupina `rotateRayTG`. Budou-li se průsečíky nacházet na souřadnicích `[0,0,z]` a umístěny pod transformační skupinu `rotateRayTG`, rotace paprsku způsobí i rotaci průsečíků, takže tyto průsečíky budou vždy ležet na demonstračním paprsku. Pro tyto průsečíky se musí pouze zadat hodnota souřadnice *z*, kterou tvoří jednoduše hloubka bodu z depth bufferu. Do depth bufferu tak byly ukládány všechny průsečíky z důvodu, jelikož do pracovního pohledu chceme vykreslit všechny průsečíky pro aktuální pozici paprsku, ne jenom průsečík nejbližší. Kód vykreslující průsečíky paprsku do scény je zobrazen v bloku 3.12. Řádek 1 pouze načítá hloubky aktuálního pixelu do pomocného pole, aby se následně nemuselo přistupovat přes dvojici indexů *x* a *y*. Řádek 2 zjišťuje, jestli aktuální pozice má vůbec nějaké

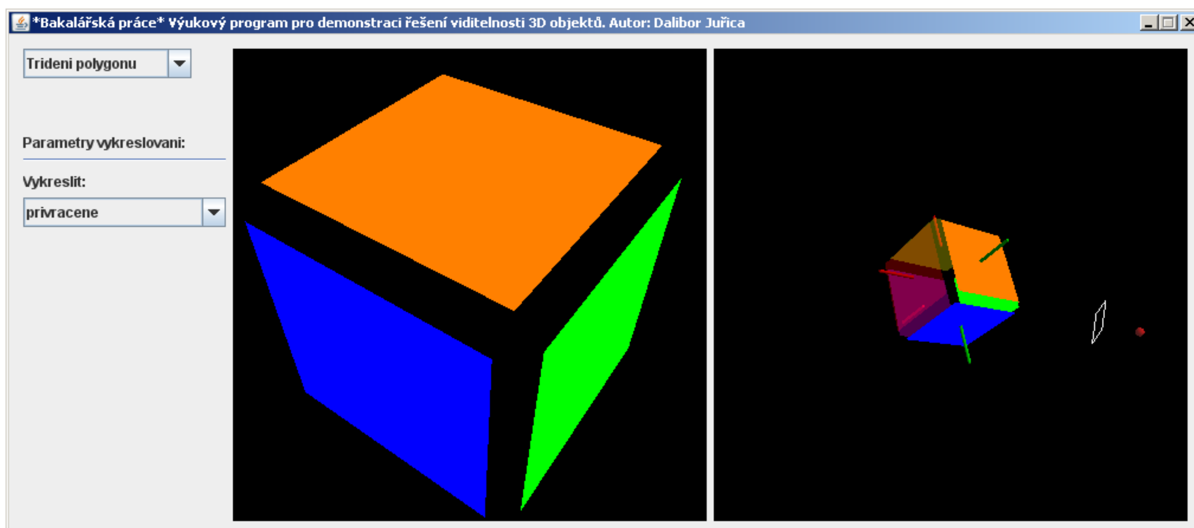
průsečíky. Pokud ano, jednotlivé průsečíky se zpracovávají v cyklu `for` na řádku 3-8. První je vytvořen vektor, který průsečík přesouvá z bodu `[0,0,0]` do bodu `[0,0,depth]`. Posunutí pro bod je nastaveno na řádku 6. Pole `translatePointsTG[]` obsahuje transformační skupiny s průsečíky pro každou položku zvlášť. Průsečíky ztvárňuje třída `IntersectionPoint`. Řádek 7 už pouze jenom zviditelní aktuální bod tak, aby šel ve scéně pozorovat. Rozlišení mezi prvním a ostatními průsečíky je provedeno tak, že první průsečík v poli průsečíku má zelenou barvu, ostatní průsečíky mají barvu červenou. Jelikož je pole `depths[]` automaticky seřazenou dle vzdáleností, první na řadu přijde vždy průsečík nejbližší. Průsečíků je v poli `intersectionPoints[]` tolik, kolik scéna obsahuje demonstračních polygonů.

```
1.      float[] depths = depthBuffer[actY][actX];
2.      if (depths == null) { return; }
3.      for (int i = 0; i < depths.length; i++) {
4.        vectorTranslate.set(0.0, 0.0, depths[i]);
5.        translatePoint.setTranslation(vectorTranslate);
6.        translatePointsTG[i].setTransform(translatePoint);
7.        intersectionPoints[i].setTransparency(BasicShape.OPAQUE);
8.      }
```

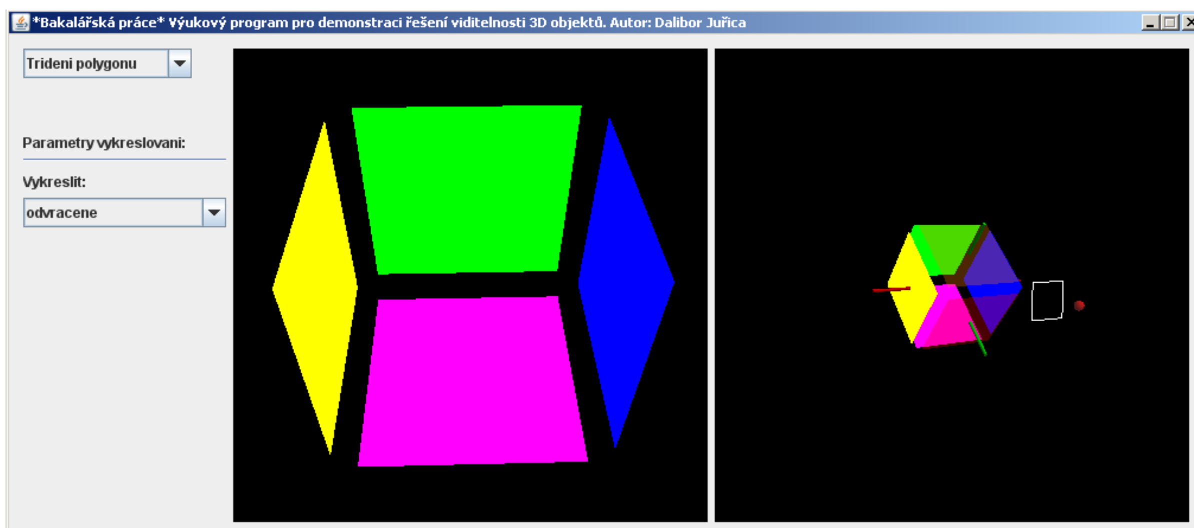
Kód 3.12: Vykreslení průsečíků paprsku se scénou do pracovního pohledu.

4 Dosažené výsledky

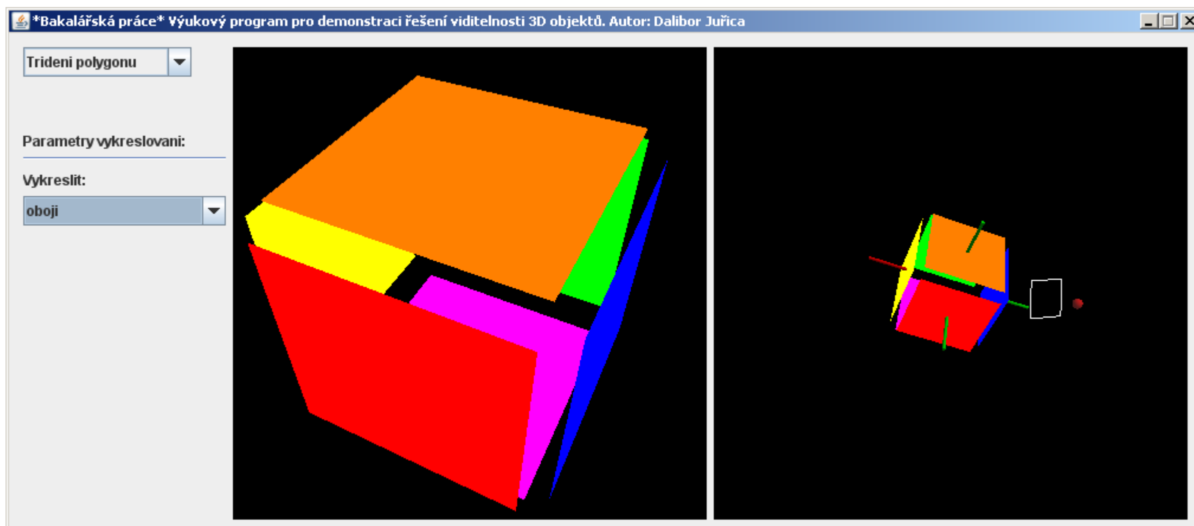
Byla vytvořena aplikace pracující v realtime režimu, která jednoduchým způsobem demonstruje vybrané algoritmy. Zde budou ukázány dosažené výsledky.



obr 4.1: Vyřazení odvrácených stěn.



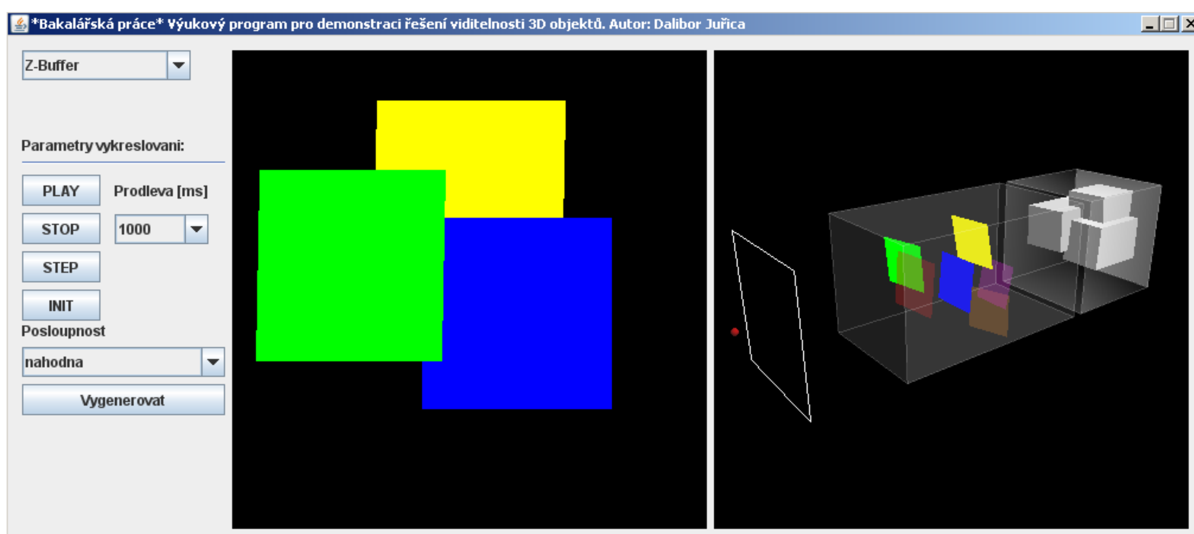
obr 4.2: Vyřazení přivrácených stěn.



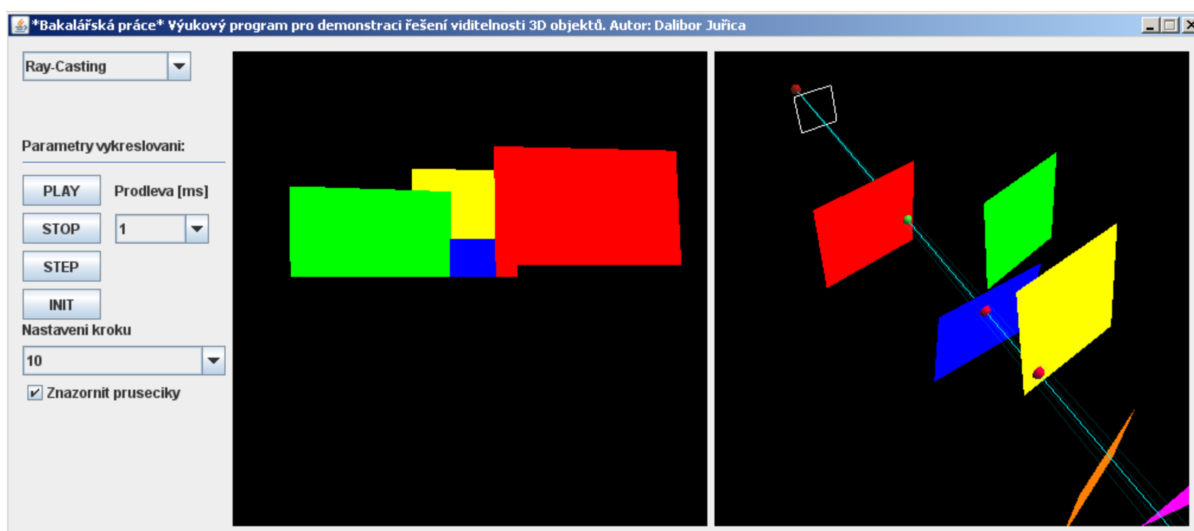
obr 4.3: Vykreslení všech stěn objektu.

obr 4.4: Vykreslení scény Malířovým algoritmem v nesprávném seřazení.

obr 4.5: Vykreslení scény Malířovým algoritmem ve správném seřazení.



obr 4.6: Vykreslení scény v náhodném pořadí pomocí Z-Bufferu.



obr 4.7: Vykreslení scény metodou Ray-Casting.

5 Závěr

Dosavadní prací bylo dosaženo implementování demonstrace vyřazení odvrácených stran, Malířova algoritmu, Z-Bufferu a Ray-Castingu. Za pomocí použití prostředí Java 3D se tak implementování problému stalo poměrně jednoduchou a značně dosažitelnou záležitostí, ve které nejsložitější částí bylo zejména nastudování požitého prostředí a první seznámení se s ním. Složitým bodem byla spolupráce s oblastí pro nastavení grafického prostředí, poskytované třídami `Canvas3D` a `View3D`, se kterou jsem se z počátku jako málo znalý domluvil poměrně těžko. Tato část není v základním tutoriálu Javy 3D dostatečně popsána a počáteční potřeba pozměnit defaultní nastavení prostředí probíhala spíše metodou pokus-omyl a ne přehledným zkušeným stylem. Je to dáno také tím, že Java 3D byla mým prvním 3D API, se kterým jsem kdy pracoval. Jelikož je první krok do světa počítačové grafiky opravdu značným nárazem informací, dá se v aplikaci očekávat, že není navržena nejideálněji a nevyužívá nejvhodnější prostředků. Prostředí aplikace jsem také značně zjednodušil a zaměřoval jsem se více na implementaci cílových problémů než na zkrášlování aplikace a vytváření sice uživatelsky příjemného prostředí, které je však k danému problému nijak nepřínosné. Oproti vytváření prostředí jsem se raději více zaměřil na optimalizaci demonstrací z hlediska rychlosti a paměťových nároků.

Dvířka pro pokračování v implementaci nových problematik a rozvíjení rozhraní programu jsou však stále otevřené. V současnosti existuje spousta metodik pro řešení viditelnosti, které by bylo přínosné do aplikace dále zanést. Dokonce ani některé základní algoritmy, jakými jsou Dělení obrazu a Robertsův algoritmus, se mi do aplikace z časových důvodů a značného vytížení ostatními předměty nepodařilo naimplementovat. Navíc demonstrační scény pro všechny 3 algoritmy jsou značně zjednodušené. Vhodný by byl návrh nových scén, popřípadě aby prostředí umožňovalo aspoň základní interakci při manipulaci s objekty scény, jako jsou posuvy, vkládaní nových objektů, mazání apod. Uživatel by si takto mohl se scénou více pohrát a zkoumání této problematiky by bylo pro něj ještě více atraktivním způsobem.

Pracování na projektu a programování v Javě 3D pro mne bylo zejména radostí a zábavou, jelikož až na několik výjimek se při implementaci nevyskytly žádné značnější problémy s neschopností něco naprogramovat. Je tudíž radostí pracovat s něčím, co rádo spolupracuje a nejsou s tím větší problémy. Samozřejmě byl pro mě projekt i obrovským přínosem zkušeností programování počítačové grafiky, která je pro mne nejatraktivnější oblastí z celého okruhu informačním technologií. Projekt jsem zvolil také i na základě mého zájmu k dané oblasti a potvrdil jsem si jím, že bych své znalosti rád rozvíjel tímto směrem.

Literatura

[1] Žára J. a kolektiv. *Moderní počítačová grafika*. Computer Press 2005.

[2] Němec, M. Materiály k předmětu *Základy počítačové grafiky*.

[3] Pelikán, J. Materiály k předmětu *Počítačová grafika I*.

[4] http://cs.wikibooks.org/wiki/Viditelnost

[5] http://www.java3d.org/czech.html

[6] Kršek, P. Materiály k předmětu *Základy počítačové grafiky* (IZG).

Seznam příloh

Příloha 1. Manuál

Příloha 2. Zdrojové texty

Příloha 3. Javadoc