Katedra Informatiky
Fakulta Matematiky, Fyziky a Informatiky
Univerzita Komenského, Bratislava

Všeobecná reprezentácia CSG objektov

(Bakalárska práca)

Michal Chládek

Vedúci: Dr. Tomáš Plachetka, PhD.

Bratislava, 2008

Čestne prehlasujem, že som túto bakalársku prácu vypracoval(a) samostatne s použitím citovaných zdrojov.

.....

Abstrakt

Cieľom tejto bakalárskej práce bolo navrhnúť reprezentáciu objektov na základe požadovaných vlastností (vracanie priesečníkov s polpriamkou a normál povrchu objektu v priesečníkoch). Na základe týchto vlastností sme s nimi schopní robiť operácie prieniku, rozdielu a zjednotenia pomocou Rothovho diagramu a takisto tieto objekty vykresľovať. Na ukladanie objektov používame serializáciu, ktorá je dôležitým konceptom v objektovo orientovaných programovacích jazykoch (v našom prípade Java). Súčasťou práce je aj program, ktorý dokáže načítať scénu z disku a následne ju vyrenderovať pomocou ray tracingu.

Kľúčové slová: ray tracing, Rothov diagram, serializácia

Obsah

Kapitola 1

Úvod

V súčasnosti je počítačová grafika stále používanejšia. Tým vyvstáva problém, ako vhodne reprezentovať objekty v počítači. Presadzuje sa prístup, pri ktorom sa objekty skladajú zo základných objektov(primitív). Avšak názor na to, ako tieto objekty reprezentovať a ktoré objekty to majú byť, nie je jednotný. Preto súčasne s narastajúcim počtom modelerov narastá aj počet formátov pre ukladanie 3d scén. Každý z týchto formátov dodržiava určité štandardy. Tie ale nie sú rovnaké pre všetky formáty. Pri vydaní novej verzie programu sa pozmenia aj štandardy na ukladania scén a už nemusia byť navzájom kompatibilné. Jednotlivé objekty môžu byť rôzne reprezentované a množina primitívnych objektov nemusí byť rovnaká. Toto spôsobuje problémy pri konverzii medzi formátmi. Navyše počet objektov v takýchto štandardoch je vždy obmedzený. Jedným z prístupov ako toto obísť, je vytvárať zložitejšie objekty z trojuholníkov, resp. polygónov. Nevýhoda takéhoto prístupu je, že sa jedná len o aproximáciu objektu. Teda množina vytvoriteľných objektov bude vždy obmedzená.

V tejto práci sa snažíme riešiť len časť problému, a to geometrickú reprezentáciu telies. Snažíme sa počítať priesečníky s polpriamkou, ktoré sú jedným zo vstupov do renderovacej rovnice. Množinu telies sa nesnažíme vymedziť ich vymenovaním, ale vlastnosťami, ktoré po nich požadujeme. Na základe týchto vlastností sme s nimi schopní robiť operácie zjednotenia,

prieniku a rozdielu. Taktiež sme schopní vykreslovať ich geometriu zjednodušeným ray tracingom.

Kapitola 2

Reprezentácia

Scénu vytvárame poskladaním zo základných objektov(primitív). Od každého objektu požadujeme, aby mal nami určené vlastnosti. Výhodou tohto prístupu je, že nemusíme dopredu určiť množinu objektov, ale len vlastnosti, ktoré musia spĺňať. Každý objekt musí vracať priesečník s polpriamkou a normálu povrchu objektu v priesečníku.

2.1 Lúč - polpriamka

V tejto práci sa jedná o triedu *Ray*. Polpriamka je reprezentovaná pomocou parametrického vyjadrenia priamky, resp. polpriamky

$$S(t) = A + t\mathbf{v};\ t \geq 0$$

Slúži na reprezentáciu lúčov v ray tracingu a reprezentáciu priesečníku, a normály. Bod A reprezentuje priesečník a vektor $\mathbf{v}$ reprezentuje smerový vektor normály.

2.2 Trieda Intersectable

Od všetkých primitív požadujeme, aby vracali všetky priesečníky s polpriamkou a normálu na povrch objektu v priesečníku. Toto zabezpečuje abstraktná

trieda *Intersectable*. Táto trieda obsahuje len dve metódy. Od každého objektu požadujeme, aby tieto dve metódy poskytoval.

Intersects (Ray ray) Parametrom je tu polpriamka, ktorej priesečníky hľadáme. Táto metóda vracia premennú boolean. True, ak existuje priesečník s danou polpriamkou a false, ak daný priesečník neexistuje.

getIntersects (Ray ray) Parameter je tu takisto ako pri metóde Intersects polpriamka, ktorej priesečníky hľadáme. V prípade, že polpriamka žiadny priesečník s objektom nemá, vracia hodnotu null. V ostatných prípadoch vracia zoznam priesečníkov a normál v usporiadanej postupnosti podľa vzdialenosti od koncového bodu polpriamky (bod A v triede *Ray*).

Všetky reprezentácie objektov sú odvodené od tejto triedy. Je jednoduché vytvárať nové objekty, stačí aby implementovali metódy tejto triedy. Na základe týchto metód sme schopní takéto objekty vykreslovať.

2.3 Rovina

Jednou z najjednoduchších primitív je rovina. V našej práci je reprezentovaná smerovým vektorom normály na rovinu a skalárnym násobkom bodu v rovine, a normálového vektora roviny. Táto reprezentácia vychádza zo známeho vyjadrenia roviny $\mathbf{n} \cdot (X - P) = 0$. Tento vzťah sa dá prepísať ako $\mathbf{n} \cdot X = d$ resp. $\mathbf{n} \cdot P = d$. Konštanta d sa dá zapísať aj v tvare $ax + by + cz - d = 0$, kde (a, b, c) je normálový vektor roviny. Pri takejto reprezentácii je počítanie priesečníkov s polpriamkov pomerne jednoduché. Potrebujeme nájsť prienik polpriamky $S(t) = A + t\mathbf{v};\ t \geq 0$ a roviny s vyjadrením $\mathbf{n} \cdot X = d$. Stačí vyriešiť rovnicu:

$$\mathbf{n} \cdot (A + t\mathbf{v}) = d$$

$$\mathbf{n} \cdot A + t\mathbf{n} \cdot \mathbf{v} = d$$

$$t\mathbf{n} \cdot \mathbf{v} = d - \mathbf{n} \cdot A$$

$$t = (d - \mathbf{n} \cdot A)/(\mathbf{n} \cdot \mathbf{v})$$

V prípade, že $t < 0$, polpriamka s rovinou priesečník nemá. Ak je $t \geq 0$, tak priesečník existuje a má vyjadrenie

$$Q = A + (d - \mathbf{n} \cdot A)/(\mathbf{n} \cdot \mathbf{v})\mathbf{v}$$

2.4 Trojuholník

Trojuholník je v našej práci reprezentovaný tromi nekolineárnymi bodmi. Pri zisťovaní priesečníku s polpriamkou sa najprv zistí priesečník polpriamky a roviny určenej vrcholmi trojuholníka. Ak existuje priesečník roviny s polpriamkou, overí sa, či tento priesečník leží v trojuholníku nasledujúcim spôsobom. Nech body A, B a C sú vrcholmi trojuholníka. Potom každý bod P v rovine určenej vrcholmi trojuholníka sa dá vyjadriť pomocou vektorov $B - A$, $C - A$ a bodu A.

$$P = A + u(C - A) + v(B - A)$$

$$P - A = u(C - A) + v(B - A)$$

$$\mathbf{v_0} = P - A,\ \mathbf{v_1} = C - A,\ \mathbf{v_2} = B - A$$

$$\mathbf{v_2} \cdot \mathbf{v_0} = (u\mathbf{v_0} + v\mathbf{v_1}) \cdot \mathbf{v_0},\ \mathbf{v_2} \cdot \mathbf{v_1} = (u\mathbf{v_0} + v\mathbf{v_1}) \cdot \mathbf{v_1}$$

Po vyriešení sústavy týchto dvoch rovníc vyjde

$$u = ((\mathbf{v_1} \cdot \mathbf{v_1})(\mathbf{v_2} \cdot \mathbf{v_0}) - (\mathbf{v_1} \cdot \mathbf{v_0})(\mathbf{v_2} \cdot \mathbf{v_1}))/((\mathbf{v_0} \cdot \mathbf{v_0})(\mathbf{v_1} \cdot \mathbf{v_1}) - (\mathbf{v_0} \cdot \mathbf{v_1})(\mathbf{v_1} \cdot \mathbf{v_0}))$$

$$v = ((\mathbf{v_0} \cdot \mathbf{v_0})(\mathbf{v_2} \cdot \mathbf{v_1}) - (\mathbf{v_0} \cdot \mathbf{v_1})(\mathbf{v_2} \cdot \mathbf{v_0}))/((\mathbf{v_0} \cdot \mathbf{v_0})(\mathbf{v_1} \cdot \mathbf{v_1}) - (\mathbf{v_0} \cdot \mathbf{v_1})(\mathbf{v_1} \cdot \mathbf{v_0}))$$

V prípade, že $v < 0$ alebo $u < 0$, je zrejmé, že bod leží mimo trojuholníka. Takisto v prípade, že $v > 1$ alebo $u > 1$, bod musí ležať mimo trojuholníka. Ak $u + v > 1$, bod leží až za stranou BC, a teda mimo trojuholníka. Takže na to, aby bod ležal v trojuholníku, musia byť splnené tri podmienky:

1. $u \geq 0$
2. $v \geq 0$
3. $u + v \leq 1$

2.5 Kocka

V tejto práci je reprezentovaná ako zoznam trojuholníkov. Pri zisťovaní priesečníkov sa otestuje každý trojuholník v zozname a vytvorí sa zoznam priesečníkov. Pri takomto prístupe sa priesečníky na hranách trojuholníkov pridajú do zoznamu viackrát. Preto je potrebné prejsť zoznam priesečníkov a vyhodiť z neho duplicitné záznamy. Keďže priesečníky je nutné vracať usporiadané od najbližšieho po najvzdialenejší, je potrebné ich ešte utriediť. Nakoľko priesečníky môžu byť najviac dva, je táto operácia triviálna. Konštruktor defaultne vytvorí kocku so stranou dĺžky jedna, jedným vrcholom so súradnicami $(0, 0, 0)$ a jedným so súradnicami $(1, 1, 1)$. Pri takejto reprezentácii odpadajú problémy s jej korektnosťou. Netreba testovať, či zadané parametre sú korektné, ako napríklad pri trojuholníku kolineárnosť bodov. Zmena veľkosti strany a zmena polohy v priestore sú zabezpečené transformáciami.

2.6 Guľa

Je reprezentovaná stredom a polomerom. Priesečníky sa počítajú pomocou rovnice $(X - C) \cdot (X - C) = r^2$, kde $C = (x_0, y_0, z_0)$ je stred gule a r je jej polomer. Táto rovnica sa dá prepísať do tvaru

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = r^2$$

Treba riešiť rovnicu, ktorú dostaneme po dosadení paramterického vyjadrenia priamky do rovnice gule.

$$(A + t\mathbf{v} - C) \cdot (A + t\mathbf{v} - C) = r^2$$

$$t^2 + 2((A - C) \cdot \mathbf{v})t + ((A - C) \cdot (A - C)) - r^2 = 0$$

Môžu nastať tieto tri prípady:

- Priesečníky sú dva v prípade, že rovnica má dve riešenia a obidve sú kladné ($t >= 0$ z definície polpriamky).

- Priesečník je jeden v prípade, že rovina má dve riešenia a z toho je jedno kladné a jedno záporné alebo rovnica má iba jedno riešenie. Prvý prípad znamená, že bod A z definície polpriamky leží vnútri kružnice. V takomto prípade je vrátený normálový vektor v priesečníku smerujúci von z kružnice.

- Priesečník neexistuje, ak rovnica nemá riešenie.

2.7 CSG objekt

Takýto objekt v sebe obsahuje CSG strom. Jeho reprezentácii a ray tracingu je venovaná samostatná kapitola (Kapitola 4).

Kapitola 3

Transformácie

Keďže je komplikované reprezentovať objekty aj s ich polohou v priestore, používame na tento účel transformácie. Toto jednak zjednodušuje reprezentáciu objektov, ale aj raytracovanie objektov je pri takomto prístupe jednoduchšie. Ďalšou výhodou transformácií je rozšírenie množiny objektov. Napríklad pri použití škálovania vieme reprezentovať kváder pomocou kocky, resp. elipsoid pomocou gule. Transformácia je reprezentovaná ako matica 4x4 zo vzťahu

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

kde (x', y', z') sú súradnice transformovaného bodu a (x, y, z) sú súradnice pôvodného bodu. Priesečník transformovaného objektu s polpriamkou je možné potom počítať nasledujúcim spôsobom. Najprv sa inverzne transformuje polpriamka. Následne sa vypočíta jej priesečník a normála v priesečníku s netransformovaným objektom. Potom sa tento priesečník a normála transformujú danou transformáciou. Pri takomto postupe nie je potrebné transformovať objekt a stačí transformovať len polpriamku, čo je rýchlejšie a jednoduchšie. Ďalšou výhodou je, že nepožadujeme žiadne ďalšie vlastnosti od objektu. Trnsformácia je implementovaná ako trieda, ktorá obsahuje maticu transformácie a maticu inverznej transformácie, teda inverznú maticu k

transformačnej matici. Táto trieda má dve metódy na transformáciu polpriamok. Jedna vracia transformovanú polpriamka a druhá inverzne transformovanú polpriamku. Presnejšia implementácia transformácií je popísaná v Kapitole 4.

3.1 Posunutie

Jedná sa o posunutie o vektor $\mathbf{v} = (v_1, v_2, v_3)$ vyjadrené vzťahom $t_v : X \to X + \mathbf{v}$ a reprezentované maticou

$$T_v = \begin{pmatrix} 1 & 0 & 0 & v_1 \\ 0 & 1 & 0 & v_2 \\ 0 & 0 & 1 & v_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Pri transformovaní polpriamky treba transformovať len bod a nie smerový vektor, lebo vektor sa posunutím nemení.

3.2 Škálovanie

Pomocou škálovania vieme zmeniť veľkosť objektu. Jedná sa o transformáciu s rovnicami

$$x' = s_1 x, \quad y' = s_2 y, \quad z' = s_3 z$$

a reprezentovanú maticou

$$\begin{pmatrix} s_1 & 0 & 0 & 0 \\ 0 & s_2 & 0 & 0 \\ 0 & 0 & s_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Pri škálovaní je nutné transformovať aj smerový vektor polpriamky, pretože škálovanie nemusí byť rovnaké vo všetkých smeroch, teda nemusí vracať lineárne závislý vektor.

3.3 Rotácia

Používame nasledovné tri rotácie:

1. $R_x(\varphi)$: otočenie okolo osi $x = (1, 0, 0)$ o uhol φ.

$$R_x(\varphi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi & 0 \\ 0 & \sin(-\varphi) & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2. $R_y(\varphi)$: otočenie okolo osi $y = (0, 1, 0)$ o uhol φ.

$$R_y(\varphi) = \begin{pmatrix} \cos\varphi & 0 & \sin(-\varphi) & 0 \\ 0 & 1 & 0 & 0 \\ \sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3. $R_z(\varphi)$: otočenie okolo osi $1 = (0, 0, 1)$ o uhol φ.

$$R_z(\varphi) = \begin{pmatrix} \cos\varphi & \sin\varphi & 0 & 0 \\ \sin(-\varphi) & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

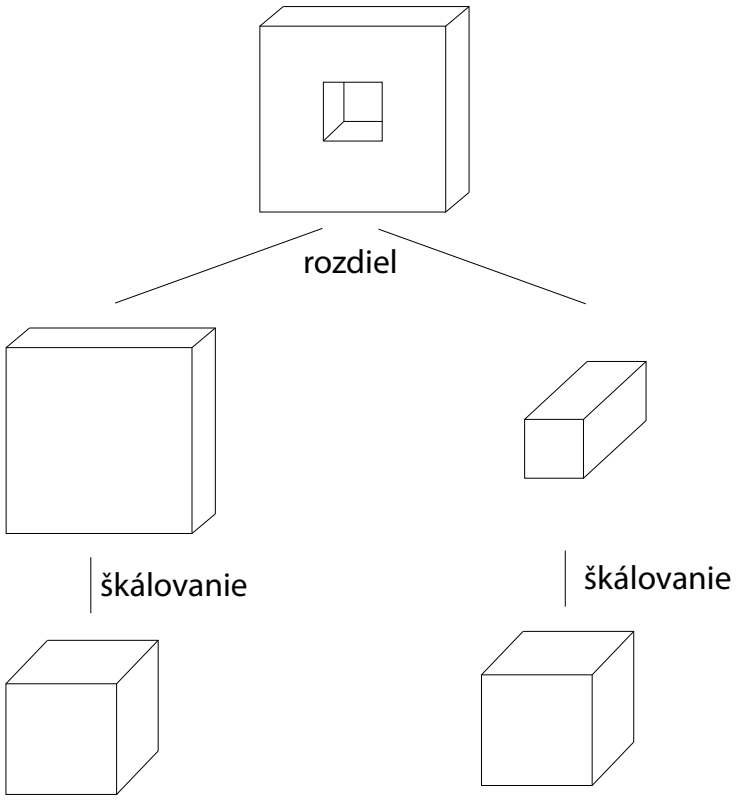
Pomocou týchto otočení vieme poskladať otočenie okolo ľubovolnej osi. Pri transformovaní polpriamky je potrebné transformovať bod aj smerový vektor polpriamky.

Kapitola 4

CSG - Constructive Solid Geometry

CSG je technika, ktorou môžeme vytvárať zložitejšie objekty pomocou booleovských operátorov zjednotenia prieniku a rozdielu. Objekty vytvárame z jednoduchých objektov (primitív), v našom prípade rovina, trojuholník, kocka a guľa. Najvhodnejšie je CSG objekt reprezentovať stromom, v ktorom každý vrchol, ktorý nie je list, reprezentuje booleovskú operáciu (prienik, zjednotenie, rozdiel) a má dvoch synov. Každý list reprezentuje jeden primitívny objekt. CSG reprezentácia je v našom prípade obohatená o transformácie. Transformáciu reprezentujeme v CSG strome vrcholom, ktorý nie je list a má jedného syna. Výhodou takéhoto prístupu je jednoduchosť, s akou sa dajú počítať priesečníky s polpriamkou. Výhodou je takisto zjednodušenie reprezentácie zložených objektov, ako aj rozšírenie množiny vytvoriteľných objektov. Ďalšou výhodou je oddelenie reprezentácie objektov od reprezentácie transformácií. Tieto dve triedy sú od seba úplne nezávislé. Príklad takéhoto stromu je na obrázku 4.1.

Pri zisťovaní priesečníku s polpriamkou potrebujeme rekurzívne prejsť celý strom. Keďže CSG objekt je tiež objekt, musí dediť metódy od triedy *Intersectable*. Teda musí vracať priesečníky s polpriamkou a normály povrchu v priesečníkoch, respektíve každý vrchol CSG stromu musí vracať priesečníky



Obrázok 4.1: CSG strom

a normály. Pri listoch je počítanie triviálne, lebo stačí vrátiť priesečníky a normály s objektom v liste.

Pri počítaní priesečníkov s vrcholom reprezentujúcim transformáciu postupujeme nasledovne:

1. Inverzne transformujeme polpriamku, aby sme mohli počítať priesečníky s netransformovaným objektom. Na to využijeme metódu transformácie, ktorá inverzne transformuje polpriamku.

2. Vypočítame priesečníky s inverzne transformovanou polpriamkou a normály v priesečníkoch pre podstrom, ktorého koreň je synom vrcholu reprezentujúceho transformáciu.

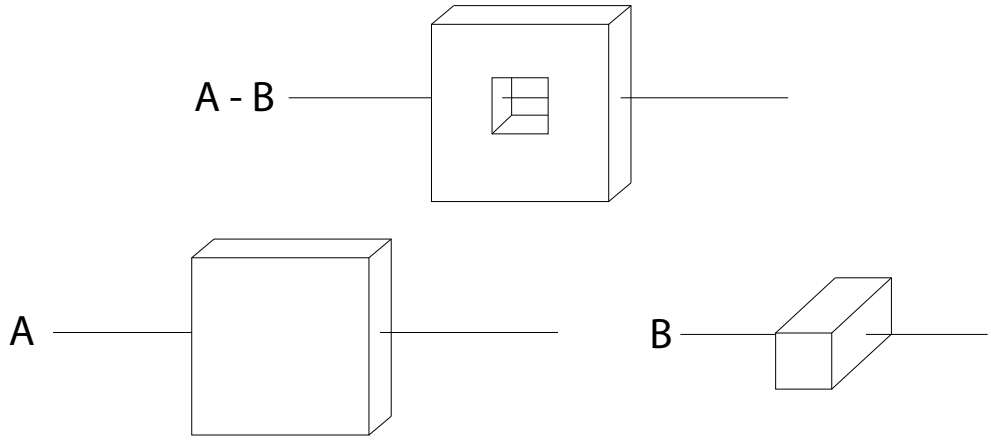
3. Nakoniec transformujeme priesečník a normálu, aby sme dostali priesečník a normálu s transformovaným objektom. Keďže priesečník a normála sú reprezentované triedou *Ray* takisto ako polpriamka, môžeme na to použiť metódu transformácie, ktorá vracia transformovanú polpriamku.

Na počítanie priesečníkov pre vrcholy, ktoré reprezentujú booleovský operátor, sa používa Rothov diagram.

4.1 Rothov diagram

Vo všeobecnosti je Rothov diagram definovaný pre priamku, v tomto prípade budeme uvažovať len o polpriamke, lebo pre naše účely je vhodnejšia. Rothov diagram je polpriamka od 0 po ∞, kde každý bod reprezentuje parameter polpriamky t (Kapitola 2). Na obrázku 4.1 je vidno časti Rothovho digramu polpriamky pre objekty z obrázku 4.1. Každý bod takejto polpriamky je buď vnútri alebo mimo objektu. Body na prechode medzi vnútrom a vonkajškom objektu sú priesečníky polpriamky s objektom. Keď ignorujeme singularity, tak úseky vnútri a mimo objektu sa striedajú. Ale na to, aby tento algoritmus dával korektné výsledky, je nutné, aby sa tieto úseky striedali. Inak nie

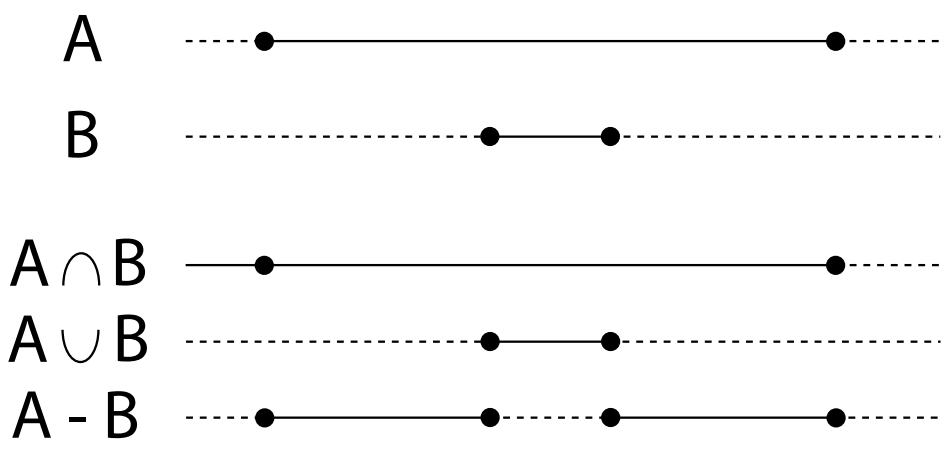
sme schopní určiť, či daný bod je vnútri alebo mimo objektu. My sme sa rozhodli ošetriť tento problém pridaním priesečníku ku každému singulárnemu priesečníku. Tento priesečník bude vzdialený od singulárneho priesečníku o nejaké malé ϵ. Normálový vektor povrchu v pridanom priesečníku bude opačný k normálovému vektoru v singulárnom bode. Týmto zabezpečíme, že sa oblasti vnútri a mimo objektu budú naozaj striedať. Už ostáva len určiť, ktoré priesečníky sú singulárne.



Obrázok 4.2: Rozdiel dvoch objektov.

Či v priesečníku polpriamka "vchádza" do objektu alebo z neho "vychádza", môžeme určiť pomocou skalárneho súčinu smerového vektora polpriamky a normálového vektora objektu v priesečníku. Všetky objekty v našom programe majú orientovaný povrch, takže ak je súčin kladný, polpriamka "vychádza" z objektu, a ak je záporný, tak doňho "vchádza". Túto vlastnosť využijeme pri kombinovaní jednotlivých polpriamok a pri hľadaní singulárnych priesečníkov. Ak bod nezachováva striedanie úsekov vnútri a mimo objektu, tak je singulárny a treba pridať priesečník.

Pri každej booleovskej operácii budeme kombinovať práve dve polpriamky. Keďže objekty nevracajú hodnotu parametra t, ale priesečníky, budeme prechádzať priesečníky od najbližšieho po najvzdialenejší. Prechádzanie nám uľahčí fakt, že objekty vracajú priesečníky usporiadané podľa vzdialenosti



Obrázok 4.3: Rothove diagramy pre objekty A, B z predchádzajúceho obrázku a pre prienik, zjednotenie a rozdiel týchto objektov. Čiarkovaná časť sa nachádza mimo objektu, neprerušovaná v objekte a krúžky sú priesečníky polpriamky s objektom.

od začiatku polpriamky. Teda priesečníky máme už utriedené.

Pri každom priesečníku sa rozhodneme, či ho pridáme do zoznamu výsledných priesečníkov. Pri zjednotení pridáme priesečník do zoznamu len v takom prípade, ak sa nenachádza vnútri druhého objektu. Pri prieniku pridáme priesečník do zoznamu len v prípade, ak sa nachádza vnútri druhého objektu. Pri rozdiele A-B pridáme do zoznamu všetky priesečníky objektu A a všetky priesečníky objektu B, ktoré sa nachádzajú vnútri objektu A. Navyše treba dávať pozor na to, že niektoré normály je potrebné otočiť o 180 stupňov, aby smerovali von z objektu. Sú to normály v priesečníkoch s objektom B. Tento postup je lepšie znázornený v nasledujúcom pseudokóde pre procedúry INTERSECTION, UNION a DIFFERENCE, ktoré vracajú priesečníky objektu po skombinovaní pomocou týchto booleovských operátorov.

INTERSECTION (*priesečníky_A*, *priesečníky_B*, *polpriamka*)
    odstránime singularity pridaním priesečníkov
    if(*priesečníky_A* == null **or** *priesečníky_B* == null)

```
        return null;
    a = priesečníky_A.First;
    b = priesečníky_B.First;
    vytvor zoznam výsledných priesečníkov
    while (a != null and b != null)
        if (VZDIALENOSŤ(polpriamka.bod, a.bod) ≤
         VZDIALENOSŤ(polpriamka.bod, b.bod))
            if (SKALÁRNY_SÚČIN(polpriamka.vektor,
             b.normálovýVektor) > 0)
                pridaj a do zoznamu výsledných priesečníkov
        if (VZDIALENOSŤ(polpriamka.bod, a.bod) >
         VZDIALENOSŤ(polpriamka.bod, b.bod))
            if(SKALÁRNY_SÚČIN(polpriamka.vektor,
             a.normálovýVektor) > 0)
                pridaj b do zoznamu výsledných priesečníkov
    return zoznam výsledných priesečníkov

UNION (priesečníky_A, priesečníky_B, polpriamka)
    odstránime singularity pridaním priesečníkov
    if (priesečníky_A == null or priesečníky_B == null)
        return null;
    if (priesečníky_A == null)
        return priesečníky_B;
    if (priesečníky_B == null)
        return priesečníky_A;
    a = priesečníky_A.First;
    b = priesečníky_B.First;
    vytvor zoznam výsledných priesečníkov
    while (a != null and b != null)
        if (VZDIALENOSŤ(polpriamka.bod, a.bod) ≤
         VZDIALENOSŤ(polpriamka.bod, b.bod))
            if (SKALÁRNY_SÚČIN(polpriamka.vektor,
```

```
                b.normálovýVektor) < 0)
                    pridaj a do zoznamu výsledných priesečníkov
            if (VZDIALENOSŤ(polpriamka.bod, a.bod) >
             VZDIALENOSŤ(polpriamka.bod, b.bod))
                if (SKALÁRNY_SÚČIN(polpriamka.vektor,
                 a.normálovýVektor) < 0)
                    pridaj b do zoznamu výsledných priesečníkov
    pridaj netestované priesečníky do výsledných priesečníkov
    return zoznam výsledných priesečníkov

DIFFERENCE (priesečníky_A, priesečníky_B, polpriamka)
    odstránime singularity pridaním priesečníkov
    if (priesečníky_A == null)
        return null;
    if (priesečníky_B == null)
        return priesečníky_A;
    a = priesečníky_A.First;
    b = priesečníky_B.First;
    vytvor zoznam výsledných priesečníkov
    while (a != null and b != null)
        if (VZDIALENOSŤ(polpriamka.bod, a.bod) ≤
         VZDIALENOSŤ(polpriamka.bod, b.bod))
            pridaj a do zoznamu výsledných priesečníkov
        if (VZDIALENOSŤ(polpriamka.bod, a.bod) >
         VZDIALENOSŤ(polpriamka.bod, b.bod))
            if (SKALÁRNY_SÚČIN(polpriamka.vektor,
             a.normálovýVektor) > 0)
                pridaj b do zoznamu výsledných priesečníkov
                 a otoč normálu
    pridaj netestované priesečníky zo zoznamu priesečníky_A
     do výsledných priesečníkov
    return zoznam výsledných priesečníkov
```

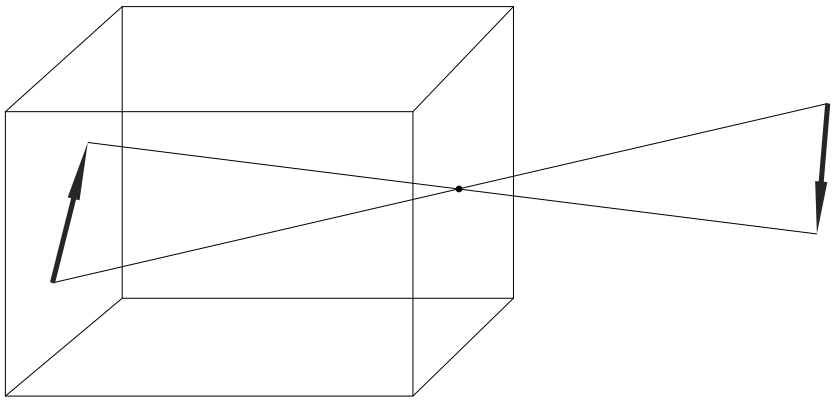
```
REMOVE_SINGULARITIES (priesečníky, polpriamka)
    if (priesečníky.size == 1)
        if (SKALÁRNY_SÚČIN(polpriamka.vektor,
         priesečníky.First.normálovýVektor) > 0)
              pridaj priesečník pred priesečníky.First s opačným
               normálovým vektorom k vektoru
               priesečníky.First.normálovýVektor
        else pridaj priesečník za priesečníky.First s opačným
               normálovým vektorom k vektoru
               priesečníky.First.normálovýVektor
        return priesečníky
    priesečník1 = null
    priesečník2 = priesečníky.First
    if(SKALÁRNY_SÚČIN(polpriamka.vektor,
      priesečník2.normálovýVektor) > 0)
        pridaj priesečník pred priesečník2 s opačným
         normálovým vektorom k vektoru
         priesečník2.normálovýVektor
    while (priesečníky.hasnext)
      priesečník1 = priesečník2
      priesečník2 = priesečníky.next
      if(SKALÁRNY_SÚČIN(polpriamka.bod,
        priesečník1.normálovýVektor) *
        SKALÁRNY_SÚČIN(polpriamka.bod,
        priesečník2.normálovýVektor) > 0)
          if(SKALÁRNY_SÚČIN(polpriamka.vektor,
            priesečník2.normálovýVektor) > 0)
              pridaj priesečník pred priesečník2 s opačným
               normálovým vektorom k vektoru
               priesečník2.normálovýVektor
          else pridaj priesečník za priesečník2 s opačným
```

normálovým vektorom k vektoru
priesečník2.normálovýVektor
return *priesečníky*

Kapitola 5

Renderovanie

Pri vykresľovaní vychádzame z tzv.“pinhole camera” modelu. Je to model z reálneho sveta, kde sa do krabice spraví diera a objekty sa premietajú cez túto dieru na zadnú stenu krabice. V ideálnom prípade je diera jeden bod, ktorým prechádzajú všetky lúče. Takýto model má tú nevýhodu, že objekty ním zobrazené sú otočené o 180^0.



Obrázok 5.1: "Pinnhole camera"model

V počítačovej grafike sa preto tento model modifikuje. Diera sa stotožní s okom pozorovateľa a premietacia rovina sa posunie pred oko pozorovateľa. Po takejto modifikácii sa jedná už o stredové premietanie so stredom v oku pozorovateľa. Odpadá potreba otáčať výsledný obraz, čo nie je taký problém.

Hlavný prínos tejto modifikácie je, že sa zjednodušia výpočty. Renderovanie obstarávujú dve triedy, PinholeCamera a Renderer. Camera počíta, aký lúč treba vyslať cez konkrétny bod obrazu a Renderer na základe tohoto lúča vypočíta priesečníky s objektami v scéne a priradí jednotlivým pixelom farbu.

5.1 Kamera

Kamera počíta, aký lúč treba vyslať cez konkrétny bod obrazu. Na to, aby to mohla vypočítať, potrebuje nasledujúce údaje.

- poloha kamery v priestore reprezentovaná bodom P
- bod T na ktorý sa pozerá
- zorný uhol φ (obr. 5.2)
- natočenie kamery okolo priamky PT reprezentované vektorom $\mathbf{u}$ (miesto neho sa može zadať aj vektor v rovine určenej bodmi P, T a vektorom $\mathbf{u}$ smerujúcim do rovnakého polpriestoru ako vektor $\mathbf{u}$)
- rozmery výstupného rastrového obrázku (x - šírka, y - výška)

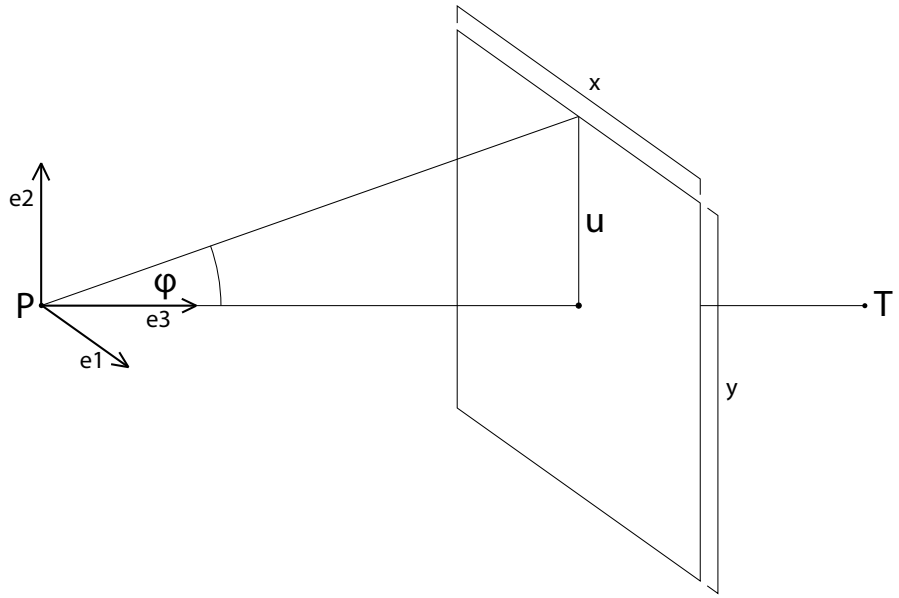
Pre uľahčenie výpočtu polpriamok sa zostrojí ortonormálna báza (e_1, e_2, e_3), ako je vidieť na obrázku 5.2.

$$\mathbf{e_3} = \frac{1}{|P-V|}(P-V), \quad \mathbf{e_1} = \frac{1}{|\mathbf{e_3} \times \mathbf{u}|}\mathbf{e_3} \times \mathbf{u}, \quad \mathbf{e_2} = \frac{1}{|\mathbf{e_1} \times \mathbf{e_3}|}\mathbf{e_1} \times \mathbf{e_3}$$

Súradnice pixela X sa vypočítajú v tejto báze a prevedú sa do referenčnej bázy. Potom polpriamka, ktorá prechádza týmto pixlom obrazu, je určená bodom P a vektorom $X - P$.

5.2 Renderer

Renderer na základe polpriamky, ktorú kamera priradí danému bodu obrazu, vypočíta priesečníky s objektami v scéne a priradí danému pixelu farbu. Pri



Obrázok 5.2: Reprezentácia kamery.

výpočte berie do úvahy len priesečník, ktorý je najbližšie k oku pozorovateľa, lebo len ten je viditeľný. Renderer neberie do úvahy žiadne osvetlenie ani textúry objektu. Vykresľuje len geometriu objektu. Ak priesečník s danou polpriamkou neexistuje, renderer nastaví pixelu, ku ktorému bola polpriamka priradená, čiernu farbu. Ak priesečník existuje, výpočíta uhol polprimaky s normálou objektu v priesečníku a priradí pixelu odtieň šedej. Je to jednoduchý prístup, ale sú pri ňom dostatočne viditeľné hrany objektu a jeho tvar je zreteľnejší, ako keby bol celý vykreslený v jednej farbe. Ukážky výstupu je možné vidieť v Kapitole 7.

Kapitola 6

Serializácia

Serializácia je proces, pri ktorom sa ukladá celý objekt alebo jeho časti na médium (hdd, CD, DVD,...), alebo sa posiela cez sieť po jednotlivých bitoch. Serializácia pozostáva z dvoch častí, z uloženia objektu (deflating) a z jeho spätného načítania (inflating). Jedným z dôvodov, prečo bola táto práca písaná v Jave, je práve serializácia. Objekty v Jave existujú len dovtedy, kým beží Java Virtual Machine a tento mechanizmus nám umožňuje uchovávať objekty aj po skončení behu Java Virtual Machine. Jednoducho uložíme ich aktuálny stav a po opätovnom spustení ho načítame z disku. Zmena renderovacieho algoritmu nevyžaduje zmenu dátového formátu za predpokladu, že nový renderovací algoritmus požaduje od objektov iba metódu getIntersections.

Na to, aby sme v jave vedeli serializovať objekt, musí tento objekt implementovať rozhranie *Serializable*. Pri ukladaní objektu treba vytvoriť najprv *ObjectOutputStream* a potom pomocou metódy tohoto streamu *writeObject(object)* zapísať objekt do tohoto streamu. Pri uložení na disk treba navyše použiť *FileStream*, ktorý je schopný písať a čítať dáta z disku. Pri čítaní z disku sa vykoná opačný postup. Namiesto *ObjectOutputStreamu* sa použije *ObjectInputStream* a namiesto *writeObject(object)* metóda *readObject(object)*. Teda uloženie objektu na disk by v skrátenej forme mohlo vyzerať takto:

```
ObjectOutputStream objstream =
new ObjectOutputStream (new FileOutputStream(filename));
objstream.writeObject(obj);
objstream.close();
```

a jeho načítanie takto:

```
ObjectInputStream objstream =
new ObjectInputStream (new FileInputStream(filename));
Object obj = objstream.readObject();
objstream.close();
```

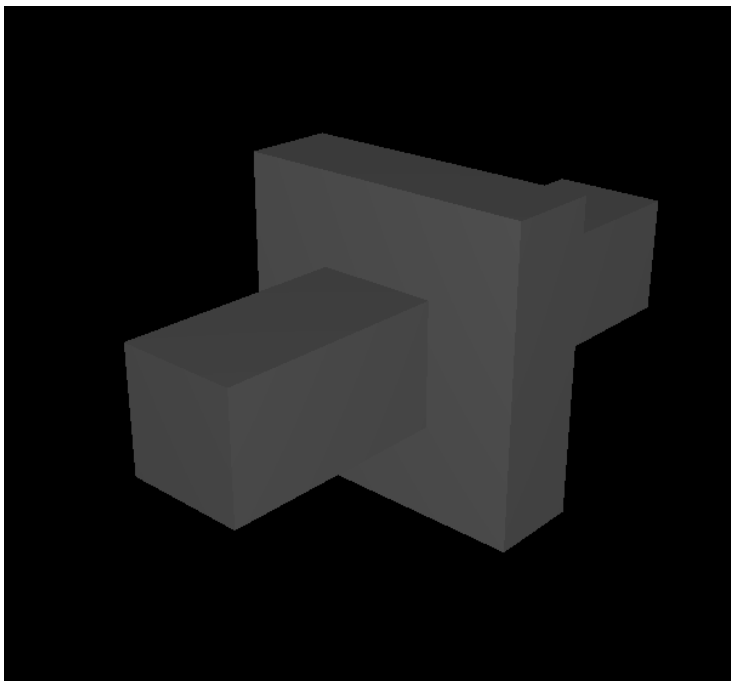
V našej práci neukladáme objekty jednotlivo na disk, ale ukladáme celú scénu. Scénu reprezentujeme triedou *Scene*. Táto trieda obsahuje zoznam objektov a kameru. Aby sme mohli objekt serializovať, musia aj všetky objekty v ňom implementovať rozhranie *Serializable*. Preto je potrebné, aby aj kamera, objekty a transformácie implementovali toto rozhranie. Pri serializácii sa potom celá scéna uloží ako súbor na disku s názvom *.ser. Po načítaní súboru je možné scénu vyrenderovať.

Kapitola 7

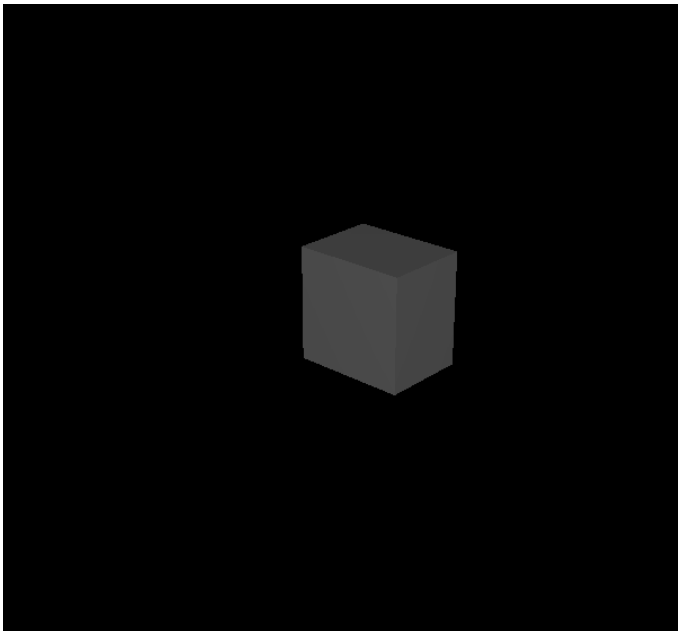
Testovanie

Táto kapitola je venovaná výsledkom vytvoreného programu. Na obrázkoch 7.1 až 7.3 je vidno zjednotenie, prienik a rozdiel dvoch transformovaných kociek. V týchto obrázkoch boli objekty vytvorené z rovnakých kvádrov, líšia sa len použitým booleovským operátorom. Na obrázku 7.4 je vidno zložitejšiu scénu.

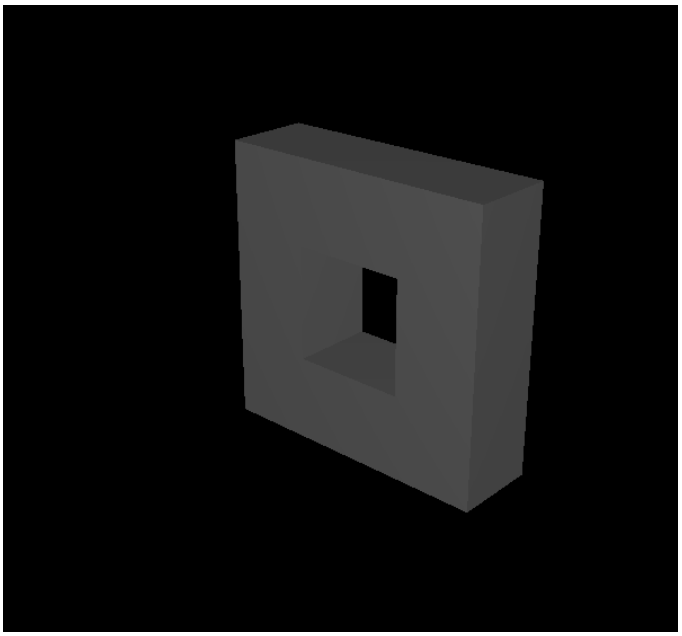
Všetky tieto obrázky vznikli po načítaní scény zo súboru a jej následnom vyrenderovaní.



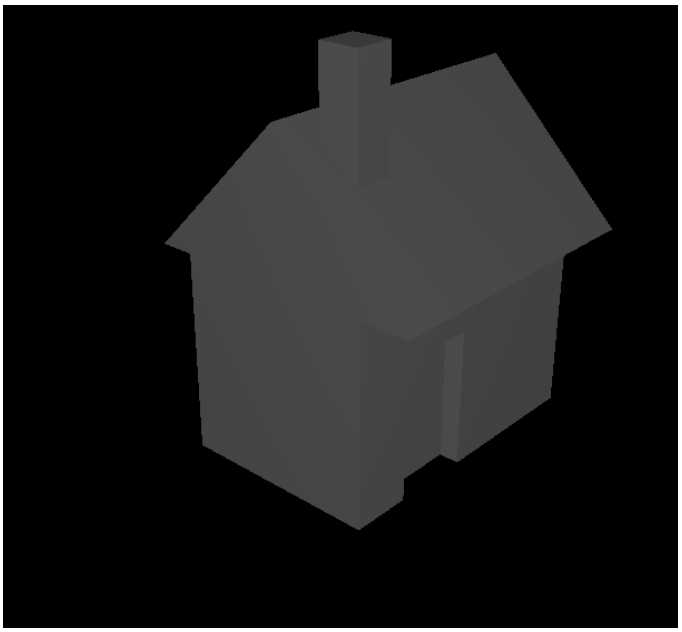
Obrázok 7.1: Zjednotenie dvoch telies.



Obrázok 7.2: Prienik dvoch telies.



Obrázok 7.3: Rozdiel dvoch telies.



Obrázok 7.4: Komplexnejšia scéna.

Kapitola 8

Záver

V tejto práci sme navrhli všeobecnú reprezentáciu objektov v počítači. Táto reprezentácia nám umožňuje vytvárať CSG objekty pomocou booleovských operácií. Na vypočítanie booleovských operácií sme použili Rothov diagram. Pôvodný článok o Rothových diagramoch sa zdá byť [Rot82], avšak nepodarilo sa nám ho zohnať. Algoritmus sme naviac rozšírili tak, že je schopný pracovať aj so singularitami. Ukázalo sa vhodné obohatiť CSG strom aj o transformácie, čo zjednodušilo reprezentáciu objektov, ako aj rozšírilo množinu vytvoriteľných objektov.

Dokázali sme implementovať serializáciu, takže objekty je možné ukladať na disk spolu s ich reprezentáciou a po ich načítaní s nimi opätovne pracovať. Toto bol jeden z dôvodov, prečo sme sa rozhodli vytvoriť bakalársku prácu v Jave. Tá totiž ponúka dobrú implementáciu serializácie. Taktiež sme tým docielili, že zmena renderovacieho algoritmu nevyžaduje zmenu vstupného formátu. Stačí, aby renderer využíval metódu getIntersects.

Nakoniec sme túto reprezentáciu využili na vykresľovanie objektov. Vytvorili sme program, ktorý načíta scénu z disku a následne ju vyrenderuje. Na vykresľovanie využíva ray tracing, ale neberie do úvahy žiadne modely osvetlenia ani textúr. Zobrazuje len geometrické vlastnosti objektov.

Literatúra

[Eri05] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2005.

[Gla89] Andrew S. Glassner. *An Introduction to Ray Tracing*. Morgan Kaufmann, 1989.

[HB97] Donald Hearn and M. Pauline Baker. *Computer graphics (2nd ed.): C version*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[Hli07] J. Hlinkova. Modeling bidirectional scattering distribution function. Bakalarska praca, Univerzita Komenskeho, Fakulta matematiky, fyziky a informatiky, 2007.

[JAV] http://java.sun.com/developer/technicalArticles/Programming/serialization.

[Rot82] S.D. Roth. Ray casting for modeling solids. *CGIP*, 18(2):109–144, February 1982.

[ser] http://java.sys-con.com/read/43802.htm.