URČOVANIE VIDITEĽNÉHO POVRCHU

Úlohou je nájsť tú časť povrchu telesa, ktorá je pre pozorovateľa viditeľná, teda časti objektov, ktoré nie sú zakryté žiadnymi inými telesami. V algoritmoch, ktoré si uvedieme, budeme väčšinou predpokladať, že namodelované objekty sú reprezentované mnohouholníkovými sieťami. Niektoré algoritmy však je možné bez výraznejších zásahov upraviť aj pre iné reprezentácie.

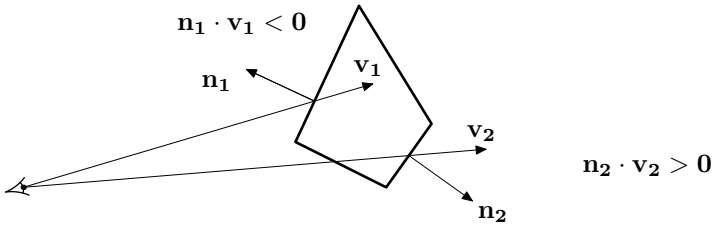
Algoritmy riešiace viditeľnosť v scéne delíme na dve skupiny, algoritmy v priestore objektov (*object space methods*) a algoritmy v priestore obrazu (*image space methods*). Uvedieme si príklady z oboch skupín.

1. Využitie normálových vektorov

Ide o metódu v priestore objektov. Algoritmus je vhodný pre reprezentáciu sieťami. Nech sú objekty popísané ako uzavrené orientované variety, čiže ku každej stene vieme rýchlo nájsť všetky s ňou incidentné vrcholy vymenované (pri pohľade zvonka) proti smeru hodinových ručičiek.

Všetky steny siete vieme po umiestnení pozorovateľa do scény rozdeliť na dve skupiny:

- odvrátené – normálový vektor a smer od pozorovateľa k stene zvierajú ostrý uhol,
- privrátené – normálový vektor a smer od pozorovateľa k stene zvierajú tupý uhol.



Obr. 1. Privrátené a odvrátené steny

Či je uhol ostrý alebo tupý, zistíme pomocou skalárneho súčinu, viď obrázok. Viditeľnými môžu byť iba privrátené steny, teda odvrátenými stenami sa ďalej už vôbec nemusíme zaoberať.

Ak scéna pozostáva z jediného konvexného mnohouholníka, prípadne z viacerých, ktoré sa však z hľadiska pozorovateľa neprekrývajú, tento algoritmus vyrieši problém viditeľnosti úplne. Bežne sú však scény omnoho komplexnejšie, a teda normálové vektory stien môžme použiť ako predspracovanie pre komplexnejšie algoritmy, ktoré sa potom už budú zaoberať iba privrátenými stenami.

Rozoberme si ešte detailnejšie postup hľadania normálových vektorov stien. Nech $A_0A_1 \dots A_{n-1}$ sú zaradom vrcholy mnohouholníka vymenované proti smeru hodinových ručičiek. Potrebujeme nájsť vektor kolmý na mnohouholník taký, aby smeroval von z mnohostenu. V ideálnom prípade, teda ak je mnohouholník konvexný a všetky vrcholy ležia v jednej rovine, stačí vziať smerové vektory ľubovoľných dvoch za sebou idúcich hrán a vypočítať ich vektorový súčin

$$\mathbf{n} = (A_i - A_{i-1}) \times (A_{i+1} - A_i).$$

Vektor $\mathbf{n}$ potom popisuje hľadaný smer normály. Teda ak $\mathbf{v}$ je vektor popisujúci smer od pozorovateľa ku stene, tak máme, že stena $A_0A_1 \dots A_{n-1}$ je privrátená, ak $\mathbf{n} \cdot \mathbf{v} < 0$, a je odvrátená, ak $\mathbf{n} \cdot \mathbf{v} > 0$.

Ak sme z nejakého dôvodu upustili od požiadavky konvexnosti, môže sa stať, že niektorý z uhlov mnohouholníka je nekonvexný. Pre nájdenie správneho smeru normály potrebujeme mať istotu, že počítame vektorový súčin z vektorov susedných hrán zvierajúcich konvexný uhol. To

zabezpečíme tak, že budeme uvažovať hrany, ktorých spoločný vrchol má extremálnu (najväčšiu alebo najmenšiu) niektorú súradnicu spomedzi všetkých vrcholov mnohouholníka.

Tiež sa môže stať, že vrcholy mnohouholníka neležia v skutočnosti v jednej rovine (chyby pri zaokrúhľovaní, principiálne nepresnosti pri konštrukcii siete), takže smer normálového vektora by závisel od toho, ktorú dvojicu susedných hrán by sme uvažovali. V takom prípade môžme za smer normály prehlásiť priemer alebo súčet všetkých možných „normál": Ak $A_0, A_1, \ldots, A_{n-1}$ sú vrcholy mnohouholníka vymenované v poradí proti smeru hodinových ručičiek, tak výsledný smer je

$$\mathbf{n} = \sum_{i=0}^{n-1} (A_i - A_{i-1}) \times (A_{i+1} - A_i),$$

kde indexy počítame modulo n.

Poznámka 1. Často pod normálou nejakého objektu (napr. mnohouholníka) rozumieme normovaný vektor špecifikovanej orientácie, čiže vektor jednotkovej dĺžky. V tejto časti sme vektory nenormovali, lebo pre účely viditeľnosti dĺžka vektora nehrá žiadnu úlohu. Preto je rozumné ušetriť si túto operáciu.

2. Z-buffer

V anglickej literatúre sa tiež nazýva *depth-buffer*. Je to príklad algoritmu v priestore obrazu. Ide o veľmi populárnu metódu, pretože sa jednoducho implementuje, a spolu s mnohouholníkovými reprezentáciami a Phongovym tieňovaním s ktorým sa zoznámite v nasledujúcom semestri) tvorí základ mnohých grafických aplikácií. *Z*-buffer je však dobre použiteľný aj v kombinácii s inými reprezentáciami.

2.1. **Základný algoritmus.** V algoritme sa každý objekt (napríklad mnohouholník) premietne v smere osi z do xy-roviny, teda akoby sa „zabudla" z-súradnica) a vykreslí sa do rastra. Pri každom mnohouholníku sa pritom počíta:

(1) súradnice (x, y) pre každý bod v rastri patriaci objektu (mnohouholníku),
(2) pôvodná z-súradnica pre každú (x, y)-hodnotu,
(3) farba (intenzita) pre každú (x, y)-hodnotu.

Pritom z-súradnice vypočítané v bode (2) sa zapisujú do *z-buffra*, čo je dvojrozmerné pole veľkosti okna, do ktorého sa vykresľuje, a farba vypočítaná v bode (3) sa zapisuje do *frame-buffra*, ktorý predstavuje samotné okno. Algoritmus v hrubých rysoch potom vyzerá nasledovne:

```
// inicializácia:
pre všetky x, y
   z-buffer[x, y] := maximálna hĺbka (nejaká inicializačná hodnota)
   voliteľne inicializuj frame-buffer (vyplň farbou pozadia)
// renderovanie:
pre každý mnohouholník v scéne
   nájdi všetky (x, y)-hodnoty v rastri, ktoré patria priemetu mnohouholníka
   pre každú relevantnú (x, y)-hodnotu
      vypočítaj príslušnú z-súradnicu (t.j. také z, že (x, y, z) leží v mnohouholníku)
      ak z < z-buffer[x, y]
         z-buffer[x, y] := z
         frame-buffer[x, y] := farba (intenzita) vypočítaná pre bod (x, y) mnohouholníka
```

Keď je jasná hrubá štruktúra algoritmu, môžme si uviesť algoritmus z-buffra v jemnejších krokoch. Výpočty vrámci jedného mnohouholníka sa robia pomocou vodorovnej zametacej priamky (*sweepline*) od jeho minimálnej hodnoty y až po maximálnu. Pre daný mnohouholník a pevné y táto priamka obsahuje x-súradnice koncových bodov prieniku priemetu mnohouholníka a príslušnej zametacej priamky, a pre každý koncový bod aj príslušné z-súradnice. Ak je mnohouholník nekonvexný, prienik môže pozostávať z viacerých úsečiek. V prípade, že sa celý mnohouholník

bude vykresľovať jedinou farbou, tieto údaje už stačia, ak by sme však chceli mnohouholník aj tieňovať, je potrebné hľadať aj farbu v každom bode, a v takom prípade by skenovacia priamka obsahovala aj údaje o farbách v koncových bodoch.

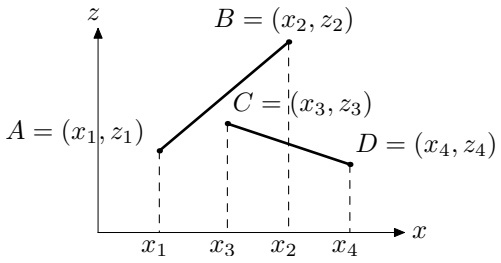
pre všetky x, y
  z-buffer$[x, y]$:= maximálna hĺbka (nejaká inicializačná hodnota)
  voliteľne inicializuj frame-buffer (vyplň farbou pozadia)
pre každý mnohouholník v scéne
  nech y_{min} a y_{max} sú minimálna a maximálna y-súradnica bodov rasterizovaného
      priemetu mnohouholníka
  // skonštruuj zoznam hrán:
  pre všetky $y \in \langle y_{min}, y_{max} \rangle$ v rastri
    nájdi prienik zametacej priamky (pre aktuálne y) a priemetu mnohouholníka
      (pôjde o jednu alebo viac úsečiek)
    do edge-list$[y]$ ulož x- a z-súradnice koncových bodov týchto úsečiek:
      $x_{left}, x_{right}, z_{left}, z_{right}$
    (ak sa bude tieňovať, ulož aj farby (intenzity) I v koncových bodoch: I_{left}, I_{right})
  // spracuj hrany zo zoznamu:
  pre všetky $y \in \langle y_{min}, y_{max} \rangle$
    pre každú úsečku v edge-list$[y]$
      prečítaj $x_{left}, x_{right}, z_{left}, z_{right}$ (prípadne aj I_{left}, I_{right})
      pre všetky $x \in \langle x_{left}, x_{right} \rangle$ v rastri
        urči z-súradnicu bodu, ktorý sa premieta do (x, y) (lineárnou interpoláciou)
        ak $z <$ z-buffer$[x, y]$
          z-buffer$[x, y] := z$
          frame-buffer$[x, y]$:= farba (intenzita) I

Ako už bolo spomenuté, výhodou tohto algoritmu riešenia viditeľnosti je jeho jednoduchosť. Preto je veľmi obľúbeným a často sa implementuje. Na druhej strane jednou z nevýhod je jeho pamäťová náročnosť. Na riešenie tohto problému sa niekedy scéna rozdelí na viac častí a každá z nich sa rieši zvlášť. Druhým problémom je, že pomocou z-buffra sa nedajú zobraziť priesvitné objekty. V z-buffri totiž nie je uložené, aký objekt sa nachádza prvý za priesvitným povrchom. Aby sme vyriešili tento problém, môžme algoritmus pozmeniť tak, že v každom poli z-buffra by sa nachádzal nie iba údaj o z-súradnici najbližšieho objektu, ale aj zoznam smerníkov na všetky objekty, ktorých priemet obsahuje príslušný bod, usporiadaných podľa z-súradnice v tomto bode. Je však zrejmé, že takáto verzia z-buffra je extrémne náročná na pamäť. Tiež si všimnime, že z-buffer často vykresľuje body aj celé objekty, ktoré v konečnom obrázku nie je vidieť, pretože sa prekryjú neskôr bližšími objektami. Táto vlastnosť charakterizuje aj niektoré iné algoritmy, ktorými sa ešte budeme zaoberať.

2.2. **z-buffer so zametacou (skenovacou) priamkou.** Ide o algoritmus z-buffra, keď sa scéna rozdelí na podscény tak, že každá sa bude zobrazovať do okna s výškou 1 pixel, šírka okna ostáva zachovaná. Máme teda z-buffer veľkosti šírky okna, pričom tento sa po spracovaní každého riadku znova inicializuje. Je to modifikácia z-buffra, ktorá šetrí pamäť. V implementácii sa vlastne zamieňa poradie for-cyklov: vonkajšia slučka nejde cez mnohouholníky, ale cez y-súradnice, pričom inicializácia buffrov sa nachádza vnútri tejto slučky.

2.3. **Zametacia priamka s intervalmi.** Myšlienkou tohto algoritmu je, že pri zametacej priamke netreba riešiť viditeľnosť v každom bode, ale stačí len raz, vo vnútri nejakého intervalu. Vysvetlíme se to presnejšie na príklade (viď obrázok).

Nech scéna pozostáva z dvoch mnohouholníkov. Zametacia priamka je priemetom roviny $y = k$ (konštantná hodnota). Jej prienikom so scénou môžu byť napríklad dve úsečky. Aby sme zistili, ktorá úsečka je pri pohľade v smere z-osi viditeľná, nájdeme si priemety koncových bodov, a tieto nám x-os rozdelia na niekoľko intervalov ($\langle x_1, x_3 \rangle, \langle x_3, x_2 \rangle, \dots$). Ak je v jednom bode vo



Obr. 2. Intervaly na zametacej priamke

vnútri niektorého intervalu viditeľná povedzme úsečka AB, bude úsečka AB viditeľná pozdĺž celého intervalu, ktorý je ohraničený priemetmi koncových bodov. V ostatných bodoch úsečky už teda nie je potrebné rozhodovať viditeľnosť, stačí už len vykresľovať.

// predspracovanie scény:
pre všetky mnohouholníky
  zaraď, pre ktoré zametacie priamky je zaujímavý (do ktorých y zasahuje)
// renderovanie:
pre každú zametaciu priamku (t.j. pre každé y v rastri)
  inicializuj z-buffer a frame-buffer
  pre každý relevantný mnohouholník
    nájdi prienik priemetu mnohouholníka a zametacej priamky (jedna alebo viac úsečiek)
  rozdeľ zametaciu priamku na intervaly
  pre každý interval
    nájdi najbližšiu úsečku (vyšetri napríklad bod v strede úsečky)
    zobraz úsečku, pripadne vytieňuj ju

Ukazuje sa, že tento algoritmus je rýchlejší než z-buffer so zametacou priamkou len ak scéna nie je príliš zložitá. Keďže v súčasnosti sú modelované scény veľmi komplikované, spravidla sa neoplatí toto zlepšenie implementovať.

2.4. **Ray casting.** S týmto algoritmom sme sa už stretli pri CSG reprezentácii. Teraz si len všimneme, že ide vlastne o z-buffer, kde scéna je rozdelená na podscény tak, že priemet každej má veľkosť jedného pixla, čiže obrazového bodu okna.

3. Maliarove algoritmy

Pojmom maliarov algoritmus rozumieme taký algoritmus, ktorý vykresľuje objekty asi tak, ako by ich vykresľoval maliar na plátno: najprv sa zobrazia najvzdialenejšie objekty a potom sa postupuje k bližším, takže vzdialené a neviditeľné sa postupne prekreslia bližšími. Tieto algoritmy si vyžadujú analýzu scény, radíme ich preto k algoritmom v priestore objektov.

3.1. **Algoritmus hĺbkového triedenia.** V prípade hĺbkového triedenia sa všetky objekty zoradia lineárne, zhruba podľa vzdialenosti od pozorovateľa, napríklad ak premietame scénu do xy-roviny (pohľad pozorovateľa v smere zápornej osi z), zoraďujeme podľa súradnice z. V nasledovnom budeme predpokladať, že objekty sú reprezentované mnohouholníkovými sieťami. Pri určovaní vzájomného poradia dvoch mnohouholníkov sa robia nasledujúce testy (stačí ak dvojica prejde jedným testom, aby sme vedeli určiť ich vzájomné poradie):

(1) ak z-obálky mnohouholníkov sa neprekrývajú, vieme ich navzájom jednoznačne zoradiť,
(2) ak sa xy-obálky mnohouholníkov neprekrývajú, môžme ich zoradiť ľubovoľne,
(3) ak jeden z mnohouholníkov je celý pred/za rovinou obsahujúcou druhý mnohouholník, vieme ich navzájom jednoznačne zoradiť,
(4) ak sa priemety mnohouholníkov neprekrývajú, môžme ich zoradiť ľubovoľne.

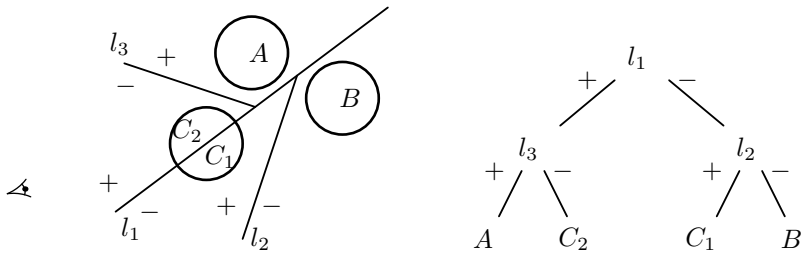
Vlastnosť (4) je len spresnením vlastnosti (2). Sú tu však uvedené oba testy, lebo test (2) je omnoho rýchlejší, a časovo náročnejší test (4) možno ani nebude potrebné vykonávať.

Pri teste (3) treba nájsť rovnicu roviny jedného mnohouholníka. Najprv nájdeme normálový vektor $\mathbf{n} = (n_1, n_2, n_3)$ roviny tak isto ako pri algoritme viditeľnosti využívajúcom normálové vektory. Potom môžme postupovať dvoma spôsobmi. V prvom prípade vypočítame koeficient d tak, aby $n_1x+n_2y+n_3z+d = 0$ bola hľadaná rovnica roviny – toto urobíme dosadením niektorého vrchola mnohouholníka ležiaceho v rovine. Nato postupne testujeme všetky vrcholy druhého mnohouholníka: súradnice dosadíme do rovnice roviny a v prípade, že všetky výsledné hodnoty majú rovnaké znamienko, celý mnohouholník leží na jednej strane od roviny a mnohouholníky vieme zoradiť.

Pri inom postupe budeme používať rovnicu roviny v tvare $\mathbf{n} \cdot (X - P) = 0$, kde P je bod patriaci rovine, $X = (x, y, z)$ je vektor premenných a $\cdot$ je skalárny súčin vektorov. Do tejto rovnice dosadzujeme za X súradnice vrcholov druhého mnohouholníka a sledujeme, či výsledné hodnoty majú rovnaké znamienko. Rozdiel medzi týmito dvoma prístupmi je v počte elementárnych operácií sčítania a násobenia dvoch čísel: v prvom prípade sme potrebovali urobiť niekoľko operácií, aby sme našli koeficient d. Avšak pri dosadzovaní súradníc jednotlivých vrcholov zisťujeme, že prvý postup vyžaduje operácií menej.

Nie všetky dvojice mnohouholníkov sa podarí pomocou uvedených testov navzájom usporiadať. Zložitejšie testy sa však už spravidla neimplementujú, pretože existujú tiež prípady, ktoré sa vôbec nedajú usporiadať (nekonvexné mnohouholníky, „zacyklené" mnohouholníky). Preto ak sa niektorý mnohouholník nedarí zaradiť do zoznamu, rieši sa to tým, že sa rozdelí rovinou iného mnohouholníka na dve časti, pričom výsledné mnohouholníky už zoradiť vieme (test (3)).

3.2. **Použitie BSP stromu.** Tentokrát objekty neusporiadame lineárne, ale do binárneho stromu. Scénu postupne rozdeľujeme a kódujeme do BSP stromu napríklad tak, že v listoch sa nachádzajú konvexné objekty, prípadne iné jednoduché objekty, ktoré vieme samostatne vykresliť (s viditeľnosťou) bez väčších problémov. Najťažšou úlohou je vtedy hľadanie vhodných deliacich rovín. Často nie je možné nájsť rovinu, ktorá všetky objekty porozdeľuje do dvoch polpriestorov (myslí sa netriviálne, t.j. aby neležali všetky objekty v jednom polpriestore). Vtedy je nutné niektorý objekt rozdeliť deliacou rovinou na dve časti.



Obr. 3. Usporiadanie scény do BSP stromu

V prípade polygonálnej reprezentácie sa v listoch nachádzajú jednotlivé mnohouholníky a deliace roviny volíme tak, aby tiež obsahovali steny reprezentácie. Taktiež sa stáva, že sa niektoré steny rozdelia na dve menšie a celkový počet sa teda zväčší. Prax ukazuje, že celkový počet objektov sa zvykne zhruba zdvojnásobiť, teda nepredstavuje to priveľký problém.

Po reprezentovaní scény BSP stromom si následne zvolíme polohu pozorovateľa a následne rekurzívne prechádzame stromom a vykresľujeme:

```
prečítaj deliacu rovinu v koreni stromu
ak je pozorovateľ v kladnom polpriestore
   vykresli najprv záporný podstrom
   potom vykresli kladný podstrom
ak je pozorovateľ v zápornom polpriestore
```

vykresli najprv kladný podstrom
potom vykresli záporný podstrom

V príklade na obrázku vykreslíme najprv scénu v zápornom podstrome od vrchola l_1, vo vrchole l_2 kreslíme tiež najprv záporný podstrom, a vo vrchole l_3 najprv kladný podstrom. Objekty sa teda vykresľujú v poradí B, C_1, A, C_2.

Ak sme scénu rozložili až na úroveň mnohouholníkov, t.j. v listoch BSP stromu sa nenachádzajú jednoduché objekty ale už len jednotlivé mnohouholníky, a ak sme mnohouholníky uložili aj do vnútorných uzlov stromu, pri vykresľovaní medzi záporným a kladným podstromom vykreslíme ešte mnohouholník z vnútorného uzla.

Veľkou výhodou tohto algoritmu je, že sa scéna usporadúva nezávisle od pozorovateľa. Veľmi lacno (rýchlo) sa preto počítajú animácie v statických scénach, keď sa pozorovateľ pohybuje v nejakom nemennom priestore, napríklad pri leteckých simuláciách alebo v niektorých počítačových hrách. Drahé predspracovanie (vytvorenie BSP stromu) sa vykoná na začiatku, a potom môže nasledovať interaktívna časť programu.

V maliarovom algoritme sa vykresľujú všetky objekty scény. V prípade, že vykreslenie každého bodu je náročná operácia (náročný výpočet farby v bode), je možné tiež postupovať opačným smerom: po zoradení sa objekty vykresľujú spredu dozadu, pričom si v nejakej dodatočnej maske o každom bode obrazovky udržiavame a obnovujeme informáciu, či už bol vykreslený, a ak áno, už ho viackrát nevykresľujeme.

3.3. **Použitie oktálneho stromu.** Je to takmer kópia predchádzajúceho algoritmu, scéna je tentokrát organizovaná v oktálnom strome. Táto metóda sa často používa sa často v prípade, že vstupné dáta sú už reprezentované oktálnym stromom. Vtedy po voľbe pozorovateľa (typicky mimo scény) si najprv usporiadame oktanty podľa viditeľnosti (podobne ako u BSP) a v tomto poradí potom prechádzame stromom a vykresľujeme jednotlivé podstromy.

4. Warnockov algoritmus delenia okna

Ide o stratégiu „rozdeľuj a panuj". V prípade, že scéna, ktorú chceme nakresliť do zobrazovacieho okna, je dostatočne jednoduchá, zobrazíme ju. V opačnom prípade okno rozdelíme na štyri kvadranty a každý riešime ako samostatné okno. Za jednoduché (riešiteľné) prípady sa považujú:

- okno je prázdne – okno sa vyplní celé farbou pozadia
- do okna zasahuje len jeden objekt – vykreslí sa objekt a zvyšok okna sa vyplní farbou pozadia
- do okna zasahuje viacero objektov, ale celé okno je prekryté nejakým najbližším objektom – okno sa vyplní farbou najbližšieho objektu

V každom inom prípade sa okno delí ďalej, až kým nenastane niektorý z jednoduchých prípadov, prípadne kým okno nemá veľkosť jedného pixla (vtedy sa vykreslí najbližší pixel).

5. Vykresľovanie drôteného modelu

V niektorých aplikáciách je žiadúce vedieť vykresliť nie vytieňovaný objekt, ale jeho drôtený model (napr. hrany polygonálnej reprezentácie) a to tak, že neviditeľné hrany sú odstránené. Na tento účel je možné použiť niektorý algoritmus riešiaci viditeľnosť stien, a pri samotnom vykresľovaní zobraziť len hranicu každého mnohouholníka a jeho vnútro premazať (vyplniť farbou pozadia).

Iný, špecifickejší prístup je, že nájdeme najprv viditeľné časti hrán, t.j. orežeme ich tak, že odrežeme neviditeľné časti, a potom vykreslíme všetky takto orezané hrany.

Hľadanie viditeľných častí hrán si môžme urýchliť podobne, ako sme pri polygónoch pomocou normál vylúčili odvrátené steny. Hrany rozdelíme do troch skupín:

- zadné – hrany, ktoré sú prienikom dvoch odvrátených stien,
- predné – hrany, ktoré sú prienikom dvoch privrátených stien,

- obrysové – hrany, ktoré sú prienikom privrátenej a odvrátenej steny.

Obr. 4. Objekt s dierou, vpravo sú zvýraznené obrysové hrany

Zadné hrany sú neviditeľné, preto sa nimi už ďalej nezaoberáme. Predné a obrysové hrany sú potenciálne viditeľné a musíme nájsť ich viditeľné časti. Tieto postupy si spomenieme len zbežne:

5.1. **Robertsov algoritmus.** Všetky potenciálne viditeľné hrany sa orezávajú všetkými stenami v scéne. Každú hranu v scéne porovnáme s každou stenou, nájdeme prieniky a zistíme, ktorá časť hrany nie je stenou zakrytá. Na záver vykreslíme všetky takto orezané hrany.

5.2. **Apelov algoritmus.** Potenciálne viditeľné hrany neorezávame stenami, ale obrysovými hranami, pretože viditeľnosť hrany sa môže zmeniť len v bode, kde priemet hrany pretína priemet niektorej obrysovej hrany. Tento algoritmus je pomerne náročný na implementáciu, lebo si vyžaduje ošetrovanie mnohých špeciálnych prípadov.

6. Zobrazovanie grafu funkcie

Ak chceme zobraziť graf funkcie $z = f(x, y)$ na nejakom intervale $\langle x_{min}, x_{max}\rangle \times \langle y_{min}, y_{max}\rangle$, je k dispozícii špeciálny a rýchly algoritmus využitia horizontu.

Každý bod $(x, y, f(x, y))$ na ploche premietame do dvojrozmerného obrazu, kde jeho nové súradnice (v priemetni, v okne) budeme označovať (x', y'). Najprv budeme vykresľovať izoparametrické krivky $(x, y_i, f(x, y_i))$ pre niekoľko pevných y_i postupne spredu dozadu. Nech $y_0, y_1, \ldots, y_n \in \langle y_{min}, y_{max}\rangle$ sú zoradené podľa vzdialenosti od pozorovateľa (t.j. bod $(0, y_0, 0)$ je najbližšie k pozorovateľovi, za ním nasleduje $(0, y_1, 0)$ atď. až po $(0, y_n, 0)$). Pre každú x'-súradnicu v okne si budeme pamätať dve extremálne vykreslené hodnoty y'_{min}, y'_{max}.

// vykresli krivku $(x, y_0, f(x, y_0))$, $x \in \langle x_{min}, x_{max}\rangle$
pre všetky $x \in \langle x_{min}, x_{max}\rangle$
  nech (x', y') je priemet bodu $(x, y_0, f(x, y_0))$
  $y'_{min}[x'] := y', y'_{max}[x'] := y'$
  vykresli (x', y')
// vykresli krivky $(x, y_i, f(x, y_i))$, $i = 0 \ldots, n$, $x \in \langle x_{min}, x_{max}\rangle$
pre $i = 1, \ldots, n$ (v tomto poradí)
  pre všetky $x \in \langle x_{min}, x_{max}\rangle$
    nech (x', y') je priemet bodu $(x, y_i, f(x, y_i))$
    ak $y' < y'_{min}[x']$
      $y'_{min}[x'] := y'$
      vykresli (x', y')
    ak $y' > y'_{max}[x']$
      $y'_{max}[x'] := y'$
      vykresli (x', y')

Následne podobne vykreslíme aj izoparametrické krivky $(x_i, y, f(x_i, y))$.

KAGDM FMFI UK Bratislava
Email address: jana.pilnikova@fmph.uniba.sk