Určovanie viditeľného povrchu

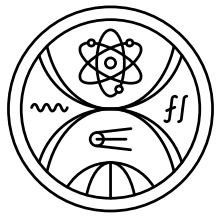
Algoritmy pre určenie viditeľnosti objektov a ich častí

Róbert Bohdal

robert.bohdal@fmph.uniba.sk
https:\\flurry.dg.fmph.uniba.sk\webog\bohdal-vyucba

Katedra algebry a geometrie
Fakulta matematiky fyziky a informatiky
Univerzita Komenského v Bratislave

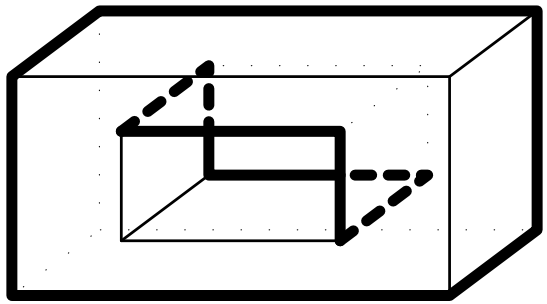
Počítačová grafika (1)
prednáška č. 5



Obsah

Viditeľnosť častí objektov v scéne

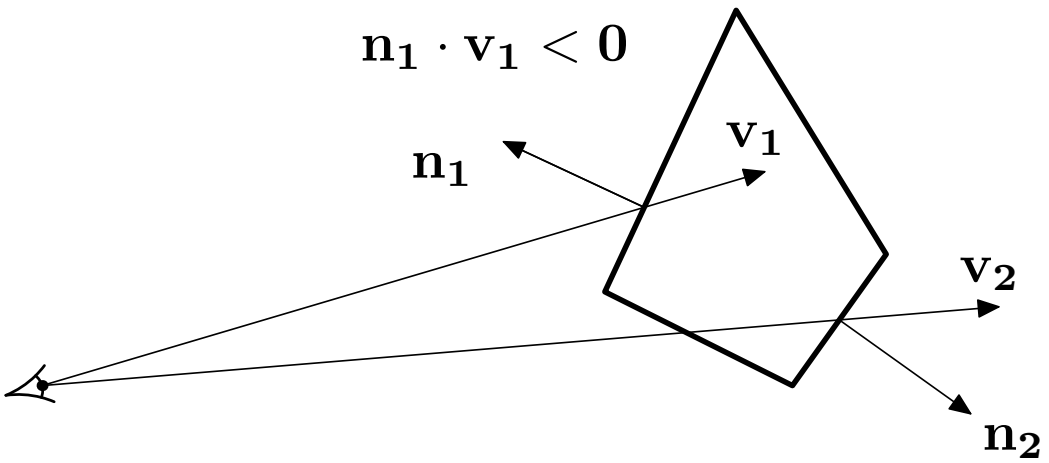
- Úlohou je nájsť tú časť povrchu telesa, ktorá je pre pozorovateľa viditeľná, t.j. časti objektov, ktoré nie sú zakryté žiadnymi inými telesami
- Väčšinou budeme predpokladať, že namodelované objekty sú reprezentované mnohouholníkovými sieťami
- Algoritmy riešiace viditeľnosť v scéne delíme na dve skupiny
  - algoritmy v priestore objektov (*object space methods*)
  - algoritmy v priestore obrazu (*image space methods*)



Príklad riešenia viditeľnosti nekonvexného objektu s dierou

Určenie viditeľnosti pomocou normálových vektorov stien

- Metóda v priestore objektov
- Algoritmus využíva reprezentáciu objektu pomocou uzavretých, orientovateľných, 2-manifold mnohouholníkových sietí. Vrcholy každej steny objektu sú indexované (pri pohľade zvonka) proti smeru hodinových ručičiek
- Všetky steny siete vieme po umiestnení pozorovateľa do scény rozdeliť pomocou skalárneho súčinu na:
  - **odvrátené** – normálový vektor a smer od pozorovateľa zvierajú ostrý uhol
  - **privrátené** – normálový vektor a smer od pozorovateľa zvierajú tupý uhol
- Viditeľné sú privrátené steny, odvrátené netreba pri vykresľovaní brať do úvahy (pozor na priesvitnosť)



Určenie viditeľnosti pomocou normálových vektorov stien

- Pri jednom konvexnom mnohouholníku, alebo viacerých neprekrývajúcich, vyrieši tento algoritmus problém viditeľnosti úplne
- Pri komplexnejších objektoch môžeme normálové vektory stien použiť ako predspracovanie, aby sme eliminovali odvrátené steny
- Nech sú vrcholy $A_0A_1 \ldots A_{n-1}$ uvažovanej steny naorientované proti smeru hodinových ručičiek, potom normálový vektor $\mathbf{n}$ mnohouholníka určíme nasledovne:
  - ak je mnohouholník konvexný, vypočítame normálu pomocou vektorového súčinu $\mathbf{n} = (A_i - A_{i-1}) \times (A_{i+1} - A_i)$
  - inak nájdeme 2 hrany, ktorých spoločný vrchol má extremálnu (najväčšiu alebo najmenšiu) súradnicu z vrcholov mnohouholníka a opäť použijeme vektorový súčin
- Ak vrcholy mnohouholníka neležia v jednej rovine, tak za smer normály použijeme priemer alebo súčet všetkých možných normál: $\mathbf{n} = \sum_{i=0}^{n-1}(A_i - A_{i-1}) \times (A_{i+1} - A_i)$

z-buffer algoritmus

- Algoritmus sa radí do metód v priestore obrazu
- Veľmi populárny, pretože sa jednoducho implementuje
- Je dobre použiteľný v kombinácii s rôznymi reprezentáciami objektov
- Spolu s Goraudovym a Phongovym tieňovaním je realizovaný v HW na grafických procesoroch

z-buffer - základný algoritmus

- Každý mnohouholník objektu sa premietne v smere osi z do roviny xy 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.
- z-súradnice vypočítané v bode 2 sa zapisujú do tzv. *z-buffra* (2D pole veľkosti okna, do ktorého sa scéna vykresľuje)
- Farba vypočítaná v bode 3 sa zapisuje do tzv. *frame-buffra*, ktorý predstavuje samotné okno s vykreslenou scénou

z-buffer - základný algoritmus

// inicializácia
pre všetky body x, y okna:
    z-buffer$[x, y]$:= maximálna hĺbka (nejaká inicializačná hodnota)
    voliteľne inicializuj frame-buffer s 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 vypočítaná pre bod (x, y) mnohouholníka

z-buffer - detailnejší algoritmus

- Výpočty vrámci jedného mnohouholníka sa robia pomocou vodorovnej zametacej priamky (*sweepline*) od jeho minimálnej hodnoty y až po maximálnu
- Táto priamka obsahuje x-súradnice koncových bodov prieniku priemetu mnohouholníka a príslušnej zametacej priamky. Pre každý koncový bod aj príslušné z-súradnice, voliteľne aj farbu
- Pre nekonvexný mnohouholník môže prienik pozostávať z viacerých úsečiek

z-buffer - detailnejší algoritmus

pre všetky x, y:
  z-buffer$[x, y]$:= maximálna hĺbka
  voliteľne inicializuj frame-buffer s 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)
    pre tieto úsečky ulož do poľa edge-list$[y]$ hodnoty $x_{left}, x_{right}, z_{left}, z_{right}$ a aj farby v koncových bodoch (vrcholoch) úsečky 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 vypočítaná pre bod (x, y) mnohouholníka

z-buffer - zhodnotenie

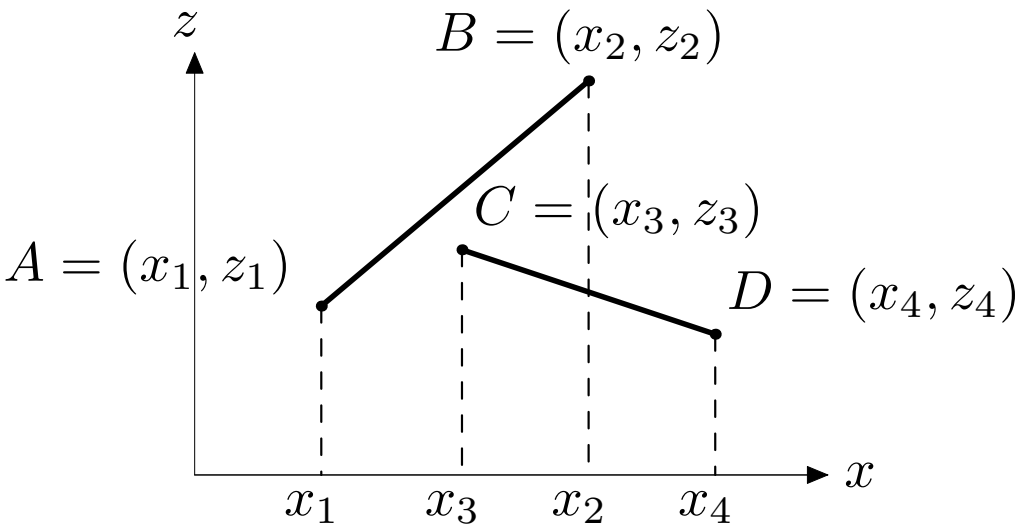
- Nevýhodou je vysoká pamäťová náročnosť. Riešením je rozdelenie scény na viac častí, pričom sa každá z nich rieši zvlášť
- Nedajú sa zobraziť priesvitné objekty. V z-buffri nie je uložené, aký objekt sa nachádza za prvým priesvitným povrchom. Riešením je ukladať zoznam smerníkov na všetky bezprostredne nasledujúce priesvitné objekty
- Vykresľujú sa body aj pre objekty, ktoré v konečnom obrázku nie je vidieť, pretože sa neskôr prekryjú bližšími objektami

z-buffer so zametacou (skenovacou) priamkou

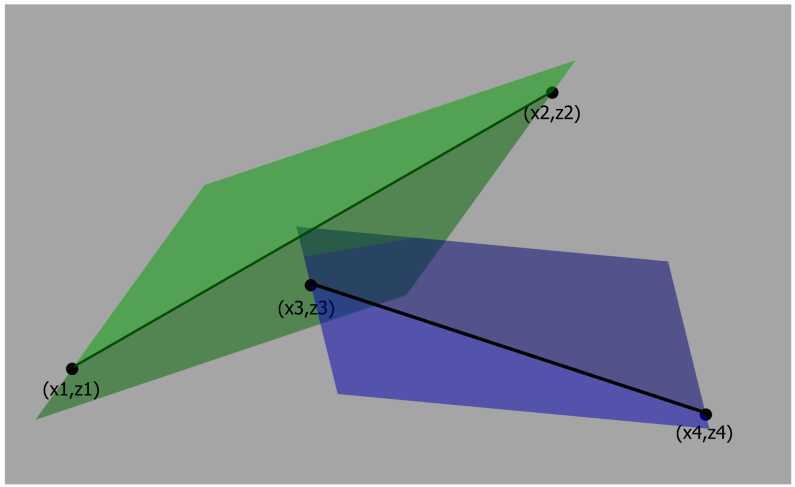
- Verzia z-buffra, v ktorej sa scéna rozdelí na podscény (okná) s výškou 1 pixel a šírkou pôvodného okna
- *z*-buffer sa po spracovaní každého riadku znova inicializuje
- V implementácii sa zamení poradie for-cyklov, kde vonkajší cyklus nejde cez mnohouholníky, ale cez *y*-súradnice a inicializácia frame a z-buffra sa nachádza vnútri tohto cyklu

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
- Aby sme zistili, ktorá úsečka je pri pohľade v smere z-osi viditeľná, nájdeme priemety koncových bodov, ktoré nám x-os rozdelia na niekoľko intervalov. Ak je v jednom bode vo vnútri niektorého intervalu viditeľná nejaká úsečka, bude viditeľná pozdĺž intervalu ohraničeného priemetmi koncových bodov. V ostatných bodoch úsečky nie je potrebné nanovo určovať viditeľnosť



Intervaly na zametacej priamke



Príklad dvoch polygónov, pohľad spredu

Ray casting a ray tracing

- Tieto algoritmy prirodzene implementujú viditeľnosť. Viac sa o nich dozvieme v kurze *Počítačová grafika (2)* v nasledujúcom semestri
- Ide vlastne o *z*-buffer, kde scéna je rozdelená na podscény tak, že priemet každej podscény má veľkosť jedného pixla – obrazového bodu okna

Maliarove algoritmy

- Pojmom maliarov algoritmus rozumieme algoritmus, ktorý vykresľuje objekty tak, ako by ich vykresľoval maliar na plátno
- Najprv sa zobrazia najvzdialenejšie objekty, potom sa postupuje k bližším, takže vzdialené a neviditeľné sa postupne prekreslia bližšími
- Radíme ich k algoritmom v priestore objektov, nakoľko vyžadujú analýzu scény, t.j. vzájomné usporiadanie objektov

Algoritmus hĺbkového triedenia

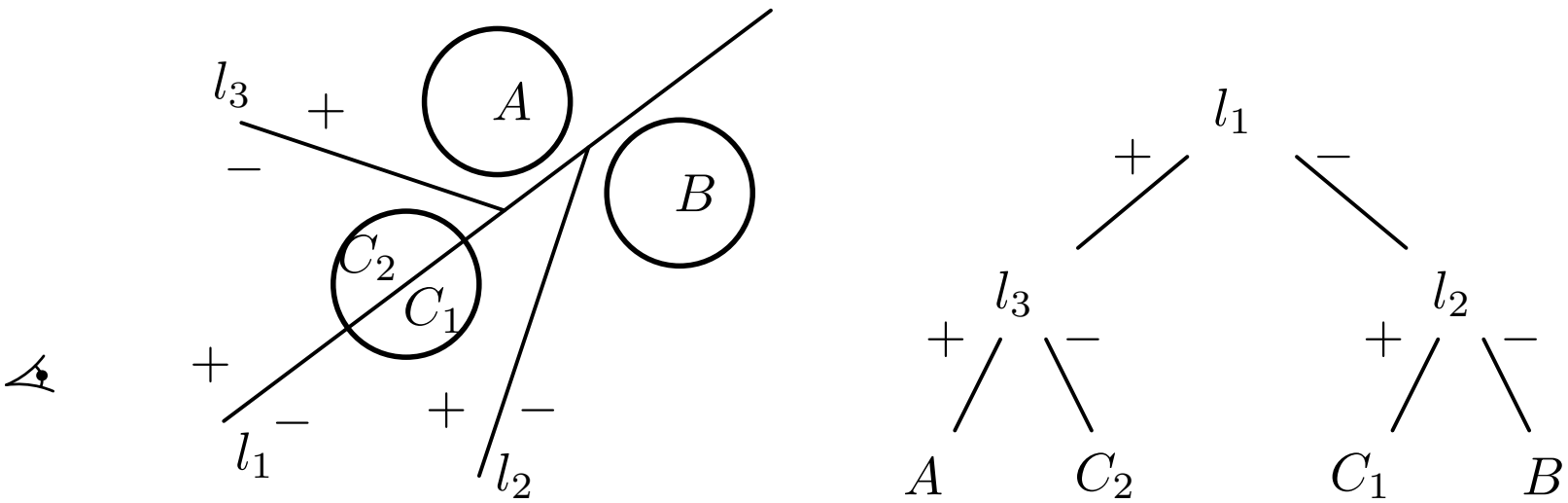
- Budeme predpokladať, že objekty sú reprezentované mnohouholníkovými sieťami
- Všetky objekty zoradia lineárne, „približne“ podľa vzdialenosti od pozorovateľa
- Ak premietame scénu do roviny xy (pohľad pozorovateľa v smere zápornej osi z), zoraďujeme podľa súradnice z
- Pri určovaní vzájomného poradia dvoch mnohouholníkov sa robia nasledujúce testy (stačí, ak dvojica prejde jedným testom):
  1. ak sa z-obálky mnohouholníkov 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 č. 4 je len spresnením vlastnosti č. 2. Navyše test č. 2 je rýchlejší a niekdy sa takto eliminuje nutnosť vykonať test č. 4

Algoritmus hĺbkového triedenia

- Test č. 3 vyžaduje nájsť rovnicu roviny $n_1x + n_2y + n_3z + d = 0$ (použijeme napr. vektorový súčin a dosadíme niektorý vrchol) jedného mnohouholníka a otestovať body druhého mnohouholníka vzhľadom na túto rovinu. Môžeme to urobiť dvoma spôsobmi:
  1. Súradnice vrcholov druhého mnohouholníka dosadíme do rovnice roviny a v prípade, že všetky výsledné hodnoty majú rovnaké znamienko, celý mnohouholník leží v jednej polrovine a mnohouholníky vieme zoradiť
  2. Použijeme rovnicu roviny v tvare $\mathbf{n} \cdot (X - P) = 0$, kde P je bod patriaci rovine. Do tejto rovnice dosadzujeme za X súradnice vrcholov druhého mnohouholníka a zisťujeme, či výsledné hodnoty majú rovnaké znamienko
- Nie všetky dvojice mnohouholníkov je možné pomocou uvedených testov navzájom usporiadať
- Zložitejšie testy sa zvyčajne neimplementujú, pretože existujú prípady, ktoré sa nedajú usporiadať (napr. 3 „zacyklené“ mnohouholníky)
- Ak sa niektorý mnohouholník nedarí zaradiť do zoznamu poradia, rozdelí sa rovinou iného mnohouholníka na dve časti tak, aby sa výsledné mnohouholníky zoradiť dali

Použitie BSP stromu

- Scénu postupne rozdeľujeme a kódujeme do BSP stromu
- V listoch sa nachádzajú konvexné či nejaké jednoduché objekty, ktoré vieme samostatne s potrebnou viditeľnosťou vykresliť
- Najťažšia časť je nájsť vhodné deliace roviny
- Často nie je možné nájsť rovinu, ktorá netriviálne rozdelí všetky objekty do dvoch polpriestorov. Vtedy je nutné niektorý objekt rozdeliť deliacou rovinou na dve časti (na obrázku je to objekt $C = C_1 \cup C_2$)



Usporiadanie scény do BSP stromu

Použitie BSP stromu

- V prípade polygonálnej reprezentácie sa v listoch nachádzajú jednotlivé mnohouholníky. Deliace roviny často volíme tak, aby tiež obsahovali niektoré steny objektov
- Po reprezentovaní scény BSP stromom sa zvolí poloha pozorovateľa, rekurzívne sa prechádza BSP stromom a vykresľuje podľa algoritmu:

  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 sa najprv vykreslí scéna v zápornom podstrome od vrchola l_1. Vo vrchole l_2 sa najskôr kreslí záporný podstrom, a vo vrchole l_3 najprv kladný podstrom. Objekty sa teda vykresľujú v poradí B, C_1, A, C_2
- Ak sa v listoch stromu nachádzajú už len jednotlivé mnohouholníky, a ak sú mnohouholníky uložené aj v uzloch stromu, pri vykresľovaní medzi záporným a kladným podstromom sa vykreslí aj mnohouholník z vnútorného uzla

Použitie BSP stromu

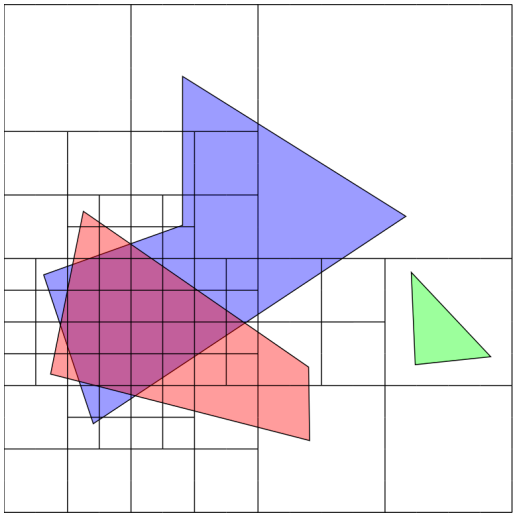
- Výhodou tohto algoritmu je, že sa scéna usporadúva nezávisle od pozorovateľa
- V statických scénach, kde sa nehýbu objekty ale iba pozorovateľ, sa náročné vytvorenie BSP stromu vykoná na začiatku. Následne je už možné veľmi rýchlo zobrazovať scénu, napr. aj v animáciach
- V maliarových algoritmoch sa vykresľujú všetky objekty scény. V prípade, že je vykreslenie každého bodu výpočtovo náročné, je možné postupovať opačne, t.j. po zoradení sa objekty vykresľujú spredu dozadu, pričom si v nejakom dodatočnom poli udržiavame a obnovujeme informáciu či už daný bod obrazovky bol vykreslený. Ak áno, tak ho už viackrát nevykresľujeme

Použitie oktálneho stromu

- Algoritmus je podobný ako u BSP, avšak scéna je rozdelená do oktálneho stromu
- Používa sa najmä v prípade, že vstupné dáta sú už reprezentované oktálnym stromom
- Po voľbe pozorovateľa sa najprv usporiadajú oktanty podľa viditeľnosti (podobne ako u BSP) a v tomto poradí sa potom prechádza stromom a vykresľujú jednotlivé objekty

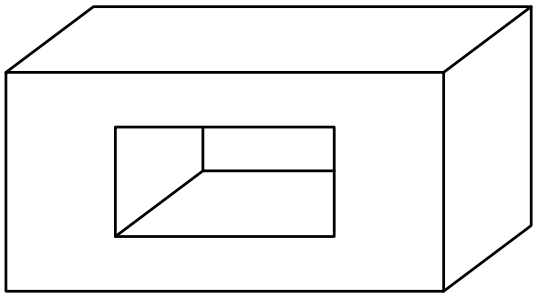
Warnockov algoritmus delenia okna

- Využíva myšlienku „rozdeľuj a panuj“, t.j. ak je scéna dostatočne jednoduchá, tak sa rovno zobrazí, inak sa rozdelí okno na štyri kvadranty a každý sa rieši samostatne
- Jednoduché (riešiteľné) prípady sa rovno vykreslia:
  - okno je prázdne – celé okno sa vyplní 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
- Inak sa okno delí ďalej, až kým nenastane niektorý z jednoduchých prípadov, prípadne kým okno nemá veľkosť jedného pixla, vtedy sa vykreslí najbližší pixel

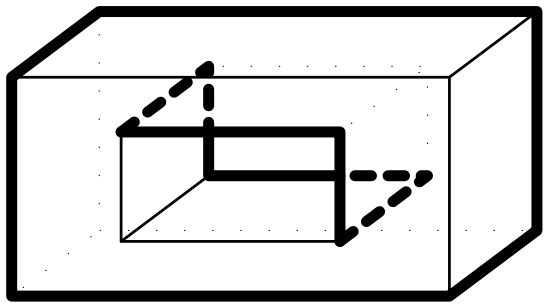


Vykresľovanie drôteného modelu

- Drôtený model (hrany polygonálnej reprezentácie) sa vykreslí tak, že neviditeľné časti (aj celé) hrán sa odstránia
- Môžeme použiť niektorý algoritmus riešiaci viditeľnosť stien, a pri samotnom vykresľovaní zobraziť len hranicu každého mnohouholníka a jeho vnútro premazať (vyplniť farbou pozadia)
- Špecifickejší prístup nájde najskôr viditeľné časti hrán a potom ich vykreslí
- Hľadanie viditeľných častí hrán využíva rozdelenie hrán 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.
- Zadné hrany sú neviditeľné, predné a obrysové hrany sú potenciálne viditeľné a preto musíme nájsť ich viditeľné časti



Objekt s dierou



Zvýraznené obrysové hrany

Robertsov algoritmus

- Všetky potenciálne viditeľné hrany sa orezávajú všetkými stenami objektov v scéne
- Každú hranu v scéne porovnáme s každou stenou, nájdeme prieniky a zistíme, ktorá časť hrany nie je stenou zakrytá
- Nakoniec sa vykreslia všetky orezané hrany

Apelov algoritmus

- Potenciálne viditeľné hrany sa neorezávajú stenami, ale obrysovými hranami, keďže viditeľnosť hrany sa môže zmeniť len v bode, kde priemet hrany pretína priemet niektorej obrysovej hrany
- Algoritmus je pomerne náročný na implementáciu, keďže vyžaduje ošetrovanie mnohých špeciálnych prípadov

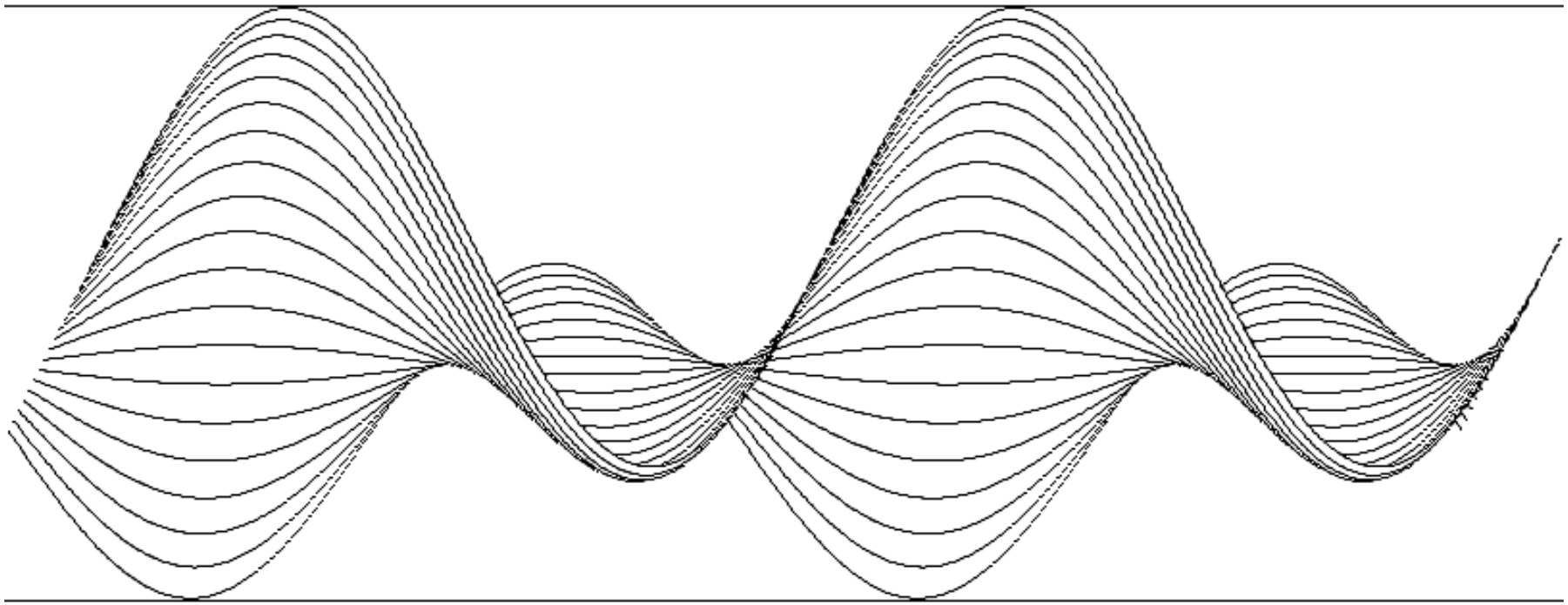
Zobrazovanie grafu funkcie

- Ak chceme zobraziť graf funkcie $z = f(x, y)$ na intervale $\langle x_{min}, x_{max} \rangle \times \langle y_{min}, y_{max} \rangle$, je možné použiť algoritmus využívajúci „obálku horizontu“
- Každý bod $(x, y, f(x, y))$ na ploche premietame do 2D obrazu, kde jeho nové súradnice v priemetni (okne) označíme (x', y')
- Najskôr sa vykreslia izoparametrické krivky $(x, y_i, f(x, y_i))$ pre $y_0, y_1, \ldots, y_n \in \langle y_{min}, y_{max} \rangle$ postupne odzadu dopredu
- Pre každú x'-súradnicu v okne si budeme pamätať v pomocnom poli dve extremálne vykreslené hodnoty y'_{min}, y'_{max}
- Následne sa rovnakým postupom vykreslia aj izoparametrické krivky $(x_i, y, f(x_i, y))$

Zobrazovanie grafu funkcie – algoritmus

// 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 $(x, y_i, f(x, y_i))$, $x \in \langle x_{min}, x_{max} \rangle$, $i = 1 \ldots, n$
pre $i = 1, \ldots, n$
    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')

Zobrazovanie grafu funkcie – ilustrácia



Časť vykresleného grafu funkcie $z = sin(x) \cdot cos(y)$ metódou obálky horizontu