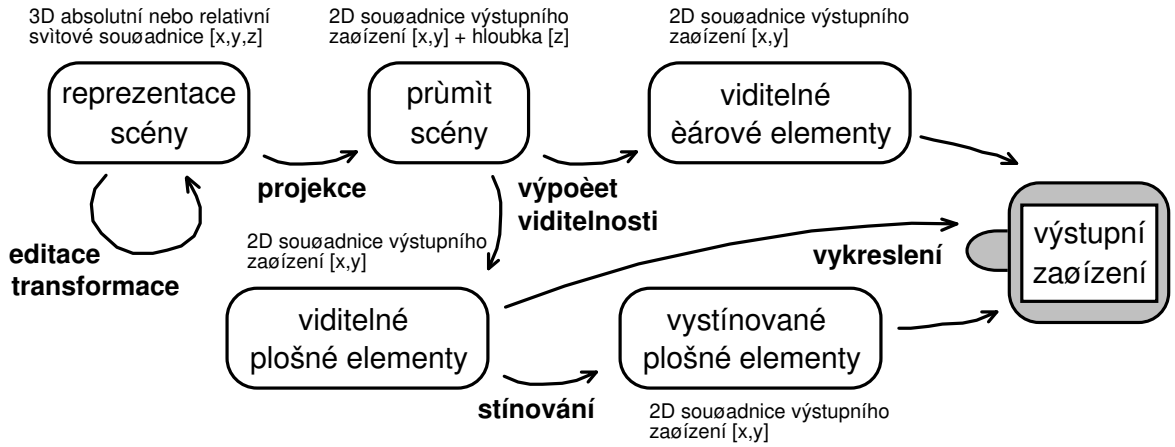
3D počítačová grafika

Josef Pelikán
UK Praha, matematicko-fyzikální fakulta*

1. Struktura 3D grafického systému

Tento materiál se zabývá základy 3D grafiky s důrazem na zobrazování 3D scén. Jde o část grafického systému, která má za úkol zobrazit (co nejrychleji a/nebo co nejvěrněji) prostorový model reprezentovaný v počítači. Další velmi významná část grafického systému spravuje tento model a uživateli umožňuje snadno jej vytvářet a editovat. Modelovacími funkcemi se však zde zabývat nebudeme.

Základní schéma klasického 3D grafického systému a jeho zobrazovací části může vypadat takto:



V dalších kapitolách se budeme postupně zabývat jednotlivými částmi zobrazovacího řetězce.

2. Reprezentace a modelování 3D scén

V této kapitole si povíme základy nejdůležitějších metod pro reprezentaci 3D scén v paměti počítače. Nejprve trochu matematiky potřebné při modelování i zobrazování:

2.1. Lineární transformace

V počítačové grafice používáme k popisu polohy bodů v rovině nebo prostoru nejčastěji tzv. **kartézskou** (pravoúhlou) **soustavu souřadnic**, poloha každého bodu je zadána dvojicí nebo trojicí čísel, kterou píšeme formálně jako vektor: [x,y] nebo [x,y,z].

Jestliže budeme chtít změnit polohu nějakého bodu nebo tělesa, musíme jeho souřadnice přepočítat. Tomuto kroku říkáme **rovinná** nebo **prostorová transformace**. Půjde nám především o jednoduché transformace, jako je posunutí (translace), otáčení (rotace), změna měřítka (podobnost, stejnolehlost), zrcadlové převrácení nebo zkosení.

Abychom dokázali nějakým jednotným způsobem tyto transformace reprezentovat a provádět, používáme čtvercové **matice 2×2** nebo **3×3**, kterými vektory souřadnic **násobíme zprava**. Mluvíme tedy pak o lineárních transformacích. Příklad obecné transformace bodu v rovině $[x,y]$ do nové polohy $[x',y']$ pomocí matice $\mathbf{M}$:

* **KSVI MFF UK, Malostranské nám. 25, 11800 Praha 1, tel: (02)24510286/l.240,214, fax: (02)535222, e-mail: Josef.Pelikan@mff.cuni.cz, WWW: http://sun3.ms.mff.cuni.cz/~pepca/**

$$[x', y'] = [x, y] \cdot \mathbf{M}, \quad \text{kde } \mathbf{M} = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}$$

Uveďme si matice jednoduchých transformací v rovině: $\mathbf{R}_\alpha$ **- otočení** okolo počátku o úhel α, $\mathbf{S}_\mathbf{a}$ **- stejnolehlost** se středem v počátku a koeficientem a, $\mathbf{M}_\mathbf{y}$ **- zrcadlové převrácení** kolem osy **y**, $\mathbf{H}_{\mathbf{xab}}$ **-** zkosení zachovávající osu **x**:

$$\mathbf{R}_\alpha = \begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix} \quad \mathbf{S}_a = \begin{bmatrix} a & 0 \\ 0 & a \end{bmatrix} \quad \mathbf{M}_y = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \quad \mathbf{H}_{xab} = \begin{bmatrix} 1 & a \\ b & 1 \end{bmatrix}$$

Možná jste si všimli, že zde chybí matice pro **posunutí**, tu totiž nelze v kartézském prostoru sestavit. Budeme-li chtít reprezentovat posunutí maticovým násobením, musíme zavést tzv. **homogenní soustavu souřadnic**. Ke dvěma složkám x,y přidáme ještě třetí, homogenní složku w. Pro obyčejné body bude homogenní složka nenulové (většinou kladné) číslo a převod mezi kartézskými a homogenními souřadnicemi budeme počítat jednoduchými vzorci

$$[x, y] \rightarrow [x, y, 1] \qquad [x, y, w] \rightarrow \left[\frac{x}{w}, \frac{y}{w}\right] \quad \text{pro } w \neq 0$$

Rovinné lineární transformace v homogenním prostoru se provádějí násobením zprava homogenní maticí velikosti 3×3:

$$[x', y', w'] = [x, y, w] \cdot \mathbf{M}, \quad \text{kde } \mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

Homogenní matice základních transformací dostaneme z těch obyčejných přidáním prvků $m_{13} = m_{23} = m_{31} = m_{32} = 0$, $m_{33} = 1$. Matice posunutí $\mathbf{T}_{xy}$ bude mít naopak ve třetím řádku zapsán vektor posunutí $[x,y]$. Příklady homogenních matic:

$$\mathbf{R}_\alpha = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{S}_a = \begin{bmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{M}_y = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{T}_{xy} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x & y & 1 \end{bmatrix}$$

Při sestavování **složitějších transformací** se využívá distributivního zákona **-** matici složené transformace dostaneme jako součin matic dílčích kroků v pořadí zleva**-**doprava. Tak lze sestavit např. matici otočení kolem libovolného bodu $[x,y]$ o úhel α jako součin $\mathbf{R}_{xy\alpha} = \mathbf{T}_{-x\,-y} \cdot \mathbf{R}_\alpha \cdot \mathbf{T}_{x\,y}$.

Lineární transformace v prostoru se konstruují a užívají obdobně **-** liší se pouze rozměrem: **3×3** pro obyčejnou nebo **4×4** pro homogenní matici. Příklady homogenních matic prostorových transformací: $\mathbf{R}_{\mathbf{x}\alpha}$ **- otočení** okolo osy **x** o úhel α, $\mathbf{R}_{\mathbf{y}\alpha}$ **- otočení** okolo osy **y** o úhel α, $\mathbf{T}_{xyz}$ **- posunutí** o vektor $[x,y,z]$:

$$\mathbf{R}_{x\alpha} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_{y\alpha} = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{T}_{xyz} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{bmatrix}$$

2.2. Objemový model - CSG

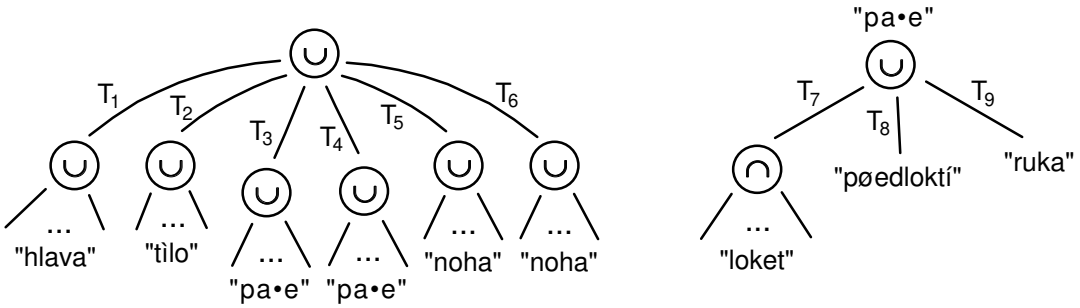
Tato metoda modelování těles je velice oblíbená ve strojírenství i deskriptivní geometrii. Umožňuje uživateli sestavovat předměty z jednoduchých tvarů (tzv. **elementárních těles**) pomocí **množinových operací**. Nejčastěji používaná elementární tělesa jsou: poloprostor ohraničený rovinou, kvádr, válec, kužel, hranol, koule, jehlan. Základní množinové operace: sjednocení, průnik, rozdíl (napodobuje obrábění **-** např. vrtání), doplněk.

Celé výsledné těleso nebo scéna jsou pak reprezentovány tzv. **CSG stromem** (Constructive Solid Geometry). Ve vnitřních uzlech jsou ukládány množinové operace, v listech jednotlivá použitá elementární tělesa. S každým odkazem na podstrom může být

spojená **transformační matice** určující, jak se má použité těleso transformovat před tím, než se použije jako argument množinové operace.

To odpovídá **hierarchickému členění scény**: např. objekt robota se může skládat z hlavy, těla, paží a nohou. Všechny tyto části mohou být spojeny operací sjednocení uložené v kořeni CSG stromu, každá část se může obdobně skládat z menších součástek, ty zase z menších, atd. Změníme-li jednu transformaci, patřící např. k levé paži robota (matice T_3), můžeme jednoduše otočit celou paží, aniž bychom se museli starat o součástky tvořící tuto paži a jejich vzájemné polohy (ty jsou dány maticemi T_7 až T_9). Transformace tedy mají **relativní** charakter.

Příklad zjednodušeného CSG stromu robota a jeho paže:

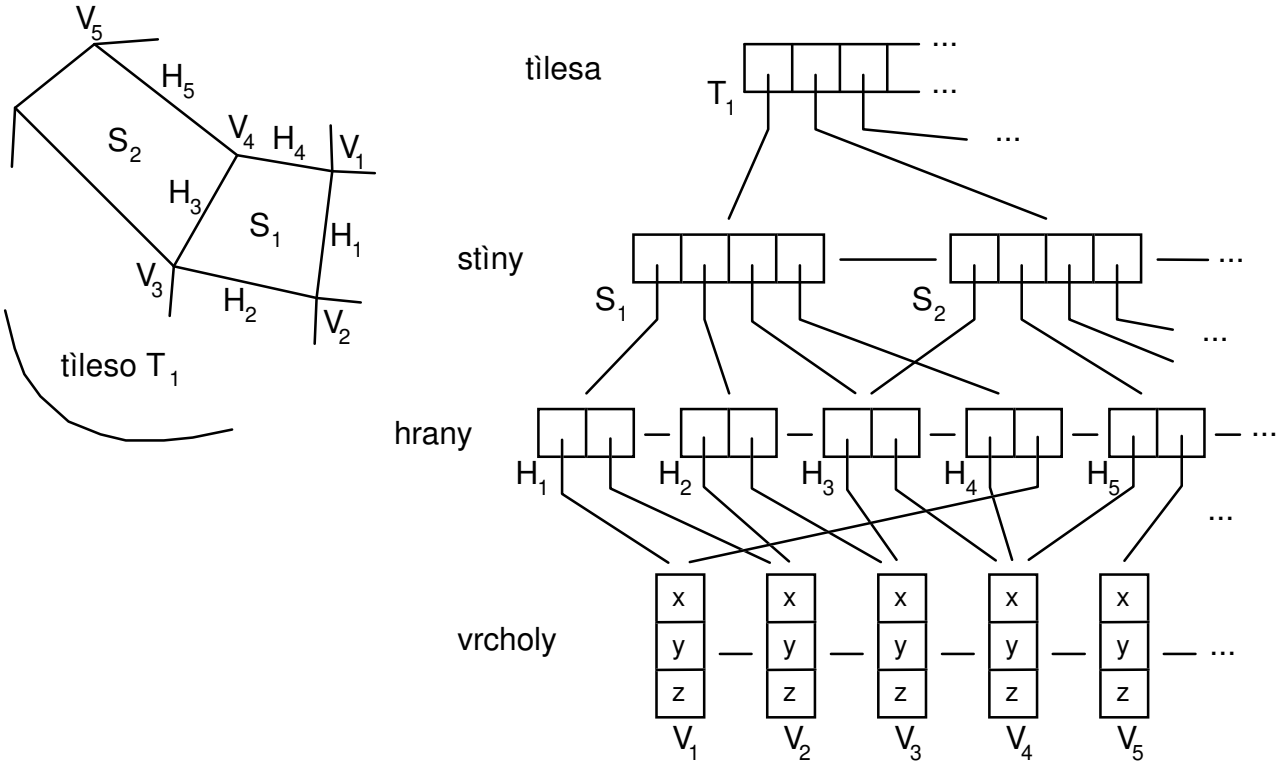


U této první reprezentace hned poznamenáme, jak ji lze **zobrazovat**. Přímé přesné zobrazení se dá dělat metodou **vrhání** nebo **sledování paprsku**, o kterých bude řeč v poslední kapitole. Rychlejší ale méně přesná metoda převádí CSG strom na **povrchovou reprezentaci** (bude popsána v následujícím oddíle), ta se potom zobrazuje klasickým postupem (viz kapitola 1).

2.3. Povrchový model

Povrchová reprezentace je nejčastěji používaná metoda jednoduchého ukládání těles. Jak již název říká, pamatuje se pouze **povrch předmětů** a není možné vždy jednoduše zjistit, které části prostoru předměty zabírají. Pro snadné uložení i výpočty se předpokládá, že povrch těles je tvořen **rovinnými stěnami**, křivé plochy se musí aproximovat (viz následující oddíl).

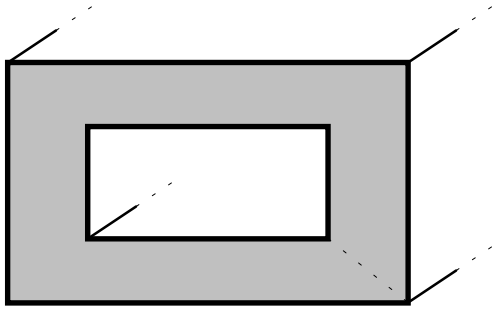
Nejjednodušší povrchová reprezentace se skládá ze čtyř seznamů (polí), ve kterých jsou uloženy: vrcholy, hrany, stěny a tělesa. Někdy se jí proto říká **VHS(T)** reprezentace. Záznam **vrcholu** obsahuje pouze jeho souřadnice **[*x*,*y*,*z*]**, záznam **hrany** je tvořen dvojicí odkazů na vrcholy **(v_1,v_2)**, záznam **stěny** musí obsahovat seznam hran tvořících její obvod **$(h_1, h_2, .., h_n)$** a konečně **těleso** si pamatuje stěny tvořící jeho povrch **$(s_1, s_2, .., s_m)$**:



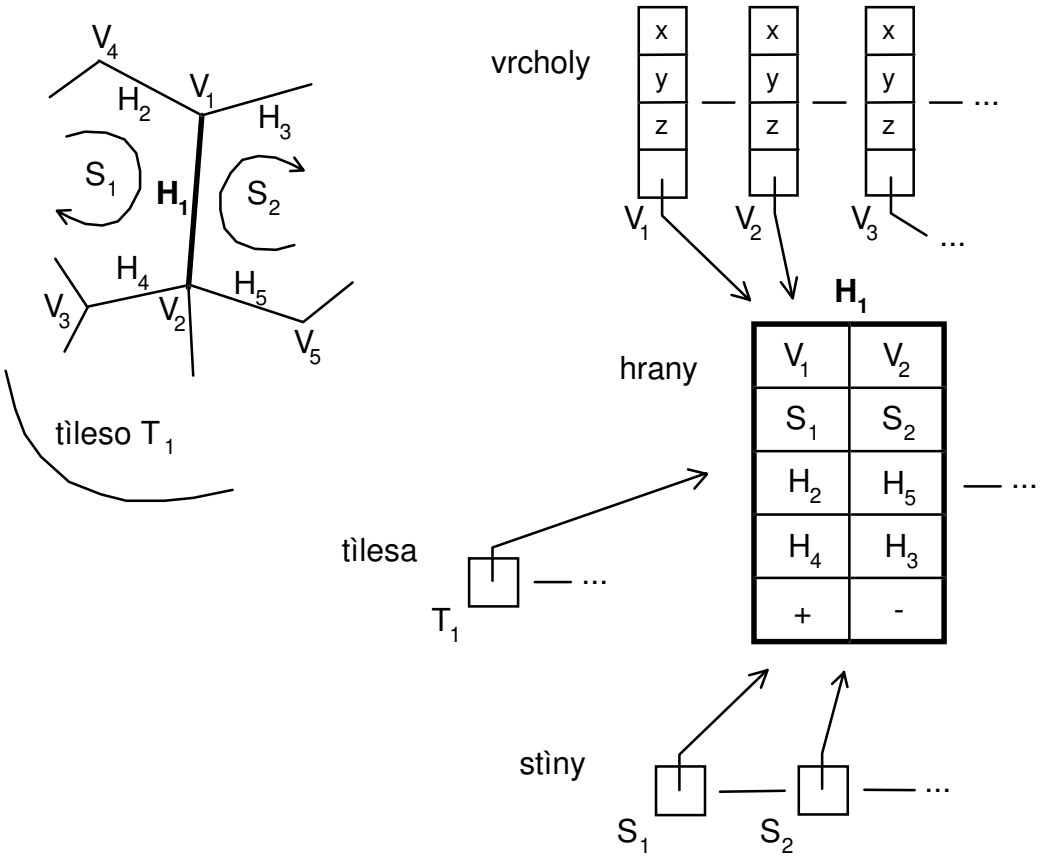
Když potřebujeme procházet graf hran nebo hledat sousední stěny, zjistíme, že to nelze efektivně udělat bez přidání dalších (redundantních) údajů. Jiná povrchová reprezentace již všechny tyto pomocné informace obsahuje: říká se jí **okřídlená hrana**, protože jejím základním prvkem je hrana odkazující se na dvě sousední stěny - křídla. Tato reprezentace umožňuje efektivně vyhledávat sousední hrany a stěny i přecházet od vrcholů k incidentním hranám a stěnám. Má jediné omezení - každá hrana musí být obsažena maximálně ve dvou stěnách (např. dvě krychle dotýkající se jednou hranou nelze uložit bez zdvojení této hrany).

Záznam **okřídlené hrany** se skládá z odkazů na dva vrcholy, odkazů na max. dvě sousední stěny a pro každou z nich jsou tam ještě odkazy na předcházející a následující hranu v té stěně. Záznam **vrcholu** obsahuje opět souřadnice **[*x,y,z*]** a dále odkaz na libovolnou incidentní hranu (z té lze snadno dohledat např. incidentní stěny). **Stěna** a **těleso** mohou mít pouze odkaz na svoji libovolnou hranu.

Pokud bychom potřebovali reprezentovat **děravá tělesa**, musíme zobecnit údaje o stěnách a dovolit děravé stěny. Nejjednodušší řešení je založené na tom, že přidáme do děravé stěny **umělou** (neviditelnou) **hranu** tak, aby spojila oba mnohoúhelníky tvořící obvod stěny:

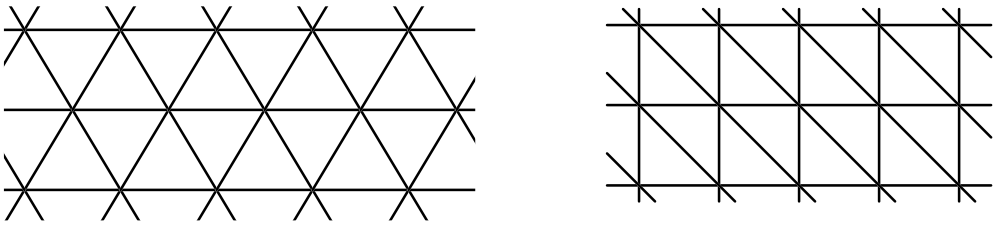


Na následujícím obrázku najdete schema části paměťové struktury pro uložení tělesa metodou okřídlené hrany:



2.4. Křivé plochy

Povrchový ploškový model se hodí nejlépe k reprezentaci hranatých těles. **Hladké křivé plochy** se musí v tomto modelu nahradit povrchem složeným z rovinných plošek. Nejčastěji se používají trojúhelníkové nebo čtyřúhelníkové sítě, protože při výpočtu nemusejí vycházet pouze rovinné čtyřúhelníky, většinou se všechny čtuřúhelníky také dělí na trojúhelníky. Následující schema ukazuje dvě trojúhelníkové sítě - pravou (souměrnou) a nepravidelnou, která vznikla z čtyřúhelníkové:



Při dělení křivých ploch je třeba volit vhodný kompromis mezi **přesností aproximace** (čím větší je počet plošek, tím bude kresba věrnější) a **efektivitou** (spotřebou výpočetního času a paměti). Pokud kreslíme na obrazovku jedinou průměrně členitou křivou plochu, postačí většinou její dělení na 20×20 až 40×40 čtyřúhelníků (to odpovídá 800 až 3200 trojúhelníkům).

Existují i algoritmy, které adaptivně přizpůsobují jemnost dělení plochy její členitosti - v místech s větší křivostí plochy se používá jemnější dělení a naopak.

3. Projekce

V prvním článku klasického zobrazovacího řetězce se musí prostorové souřadnice těles převést do plošných souřadnic na kreslicí rovině. Zobrazování prostorových scén se provádí už od rané historie lidstva, a tak se postupně vyvinulo a začalo používat několik druhů rovinných projekcí (projekcemi na nerovinné plochy se zde nebudeme zabývat):

Nejjednodušší je **rovnoběžná projekce -** obraz v průmětně se vytváří soustavou rovnoběžných promítacích paprsků. Podle toho, pod jakým úhlem dopadají promítací paprsky na průmětnu, mluvíme o **kolmé** nebo o **kosoúhlé projekci**. Kolmá projekce odpovídá pohledu člověka (nebo objektivu fotoaparátu) z velmi velké vzdálenosti - z nekonečna. Kosoúhlá projekce byla vytvořena uměle a nepodobá se žádnému reálnému pohledu, je však výhodná pro technické kreslení.

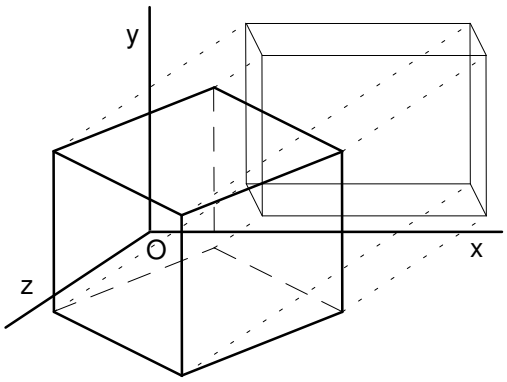
Věrnější obrázky dostaneme použitím tzv. **středové** neboli **perspektivní projekce**. U té vychází svazek promítacích paprsků z jednoho bodu - středu promítání. Perspektiva napodobuje pohled člověka (nebo objektivu fotoaparátu) na těleso z blízké vzdálenosti; při tom vzniká perspektivní zkreslení - bližší části se zobrazují jako větší a naopak.

V následujících dvou oddílech se budeme zabývat implementacemi rovinných projekcí:

3.1. Rovnoběžné promítání

Nejjednodušší rovnoběžné promítání **- Mongeovu projekci -** dostaneme tak, že pouze vynecháme příslušnou souřadnici. Tím vznikne půdorys a nárys (příp. též bokorys). Pro potřeby výpočtu viditelnosti však tu třetí souřadnici (hloubku) budeme potřebovat a Mongeova projekce se tak zredukuje na pouhou permutaci složek souřadnice.

Složitější případy **rovnoběžné kolmé projekce** se budeme snažit převést na předcházející jednoduchý případ transformací soustavy souřadnic. Takové převedení jsme schopni sestavit z posunutí a otáčení (viz oddíl 2.1). Konkrétní výsledné postupy jsou velmi jednoduché a nebudeme je zde uvádět. Domluvme se jenom, že budeme všechny pohledy převádět do situace, kdy se pomyslný pozorovatel dívá proti směru osy **z** a průmětnou je rovina **xy**:



Kosoúhlá projekce se od předchozích případů liší tím, že po převedení průmětny do roviny **xy** musíme ještě aplikovat transformaci **zkosení** zachovávající rovinu **xy** beze změny a pak teprve zanedbat souřadnici *z*. Homogenní matice zkosení bude mít pro **kavalírní** (na všech souřadných osách má stejné měřítko) a **kabinetní projekci** (na ose z se nanášené vzdálenosti zkracují na polovinu) tvar:

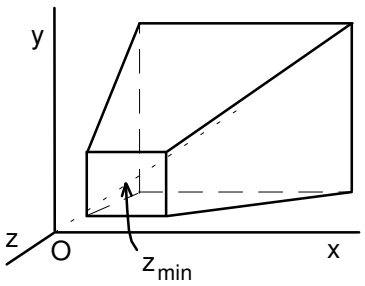
$$\mathbf{H}_{KAV} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\cos\alpha & -\sin\alpha & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad \mathbf{H}_{KAB} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{1}{2}\cos\alpha & -\frac{1}{2}\sin\alpha & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

kde α je úhel, který má v průmětu svírat průmět osy **z** s vodorovným směrem.

3.2. Středové promítání

Aby byly naše výpočty co nejjednodušší, budeme i při perspektivní projekci transformovat souřadné soustavy - a to tak, aby ležel **střed projekce** (poloha pomyslného pozorovatele) **v počátku** a **průmětna** byla **kolmá na osu z**. Představme si opět, že pozorovatel se dívá proti směru osy **z**, tj. do záporných hodnot souřadnice *z*. Označme si jako $\mathbf{z_{min}}$ polohu nejbližší roviny, kterou

jsme schopni zobrazit. Pak můžeme promítnout všechny body se zorného komolého jehlanu zobrazeného na obrázku pomocí uvedené homogenní matice ***PER***:



$$\mathbf{PER} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \dfrac{1}{1+z_{min}} & -1 \\ 0 & 0 & \dfrac{-z_{min}}{1+z_{min}} & 0 \end{bmatrix}$$

Matice **PER** má tu výhodu, že přepočítává i souřadnici *z* tak, aby ve výsledném prostoru byla zachována **linearita útvarů** (úsečky zůstanou úsečkami a roviny rovinami).

4. Výpočet viditelnosti

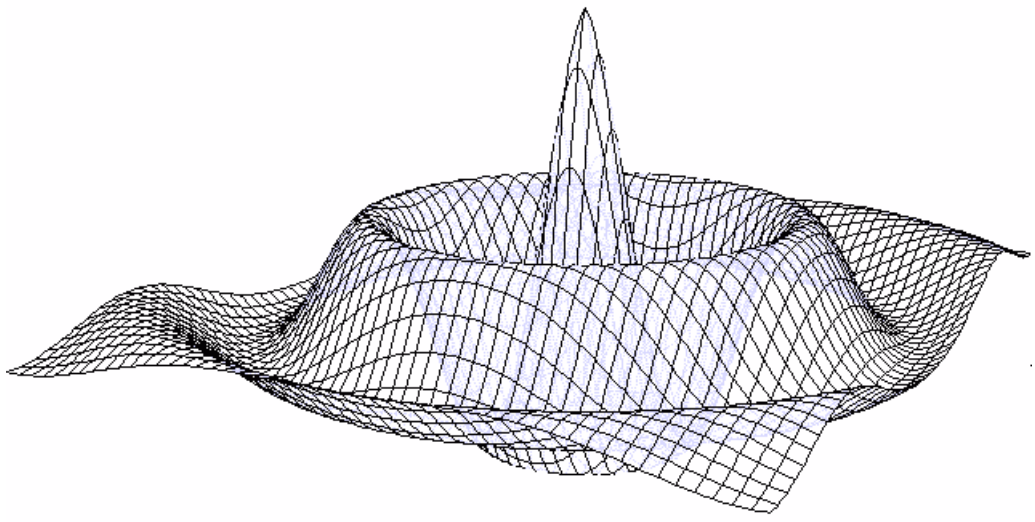
Výpočet viditelnosti je nejzajímavějším a nejsložitějším článkem klasického zobrazovacího řetězce. Jde nám o to, aby uživatel viděl na výsledném obrázku pouze ty objekty a jejich části, které nejsou zakryté jinými předměty. Přitom rozlišujeme algoritmy **čárové**, schopné spočítat vektorově reprezentovanou kresbu vhodnou pro výstup např. na plotr, a algoritmy **plošné** (rastrové), schopné vybarvovat povrchy těles a obvykle podstatně využívající rastrový charakter výstupního zařízení (obrazovky nebo rastrové tiskárny).

Existuje velké množství metod výpočtu viditelnosti, my se budeme v následujících oddílech zabývat čtyřmi nejužívanějšími principy, které se uplatňují při klasickém výpočtu viditelnosti. Poslední kapitola pak bude pojednávat o přímé metodě zobrazování, která v sobě též implicitně zahrnuje výpočet viditelnosti, ale ke klasickým metodám ji nezařazujeme.

4.1. Metoda plovoucího horizontu

Jediná ze čtyř metod, která se může implementovat též v čárové variantě. Není zcela univerzální metodou, neboť vyžaduje, aby zobrazovaný objekt splňoval speciální vlastnosti **-** my ji budeme používat ke kreslení **grafu spojité funkce dvou proměnných**.

Graf spojité funkce dvou proměnných ***f(x,z)*** je plocha složená z bodů, pro něž platí podmínka ***y=f(x,z)***. Zobrazuje se obyčejně pomocí **sítě povrchových křivek** sestavených z bodů, které mají zafixovanou souřadnici ***x*** nebo ***z***. Příklad takového grafu vidíme na obrázku:



Jak zjistíme, které části čar mají být vidět a které ne? Plochu kreslíme odpředu (od větších hodnot souřadnice *z* k menším) a při kreslení si udržujeme **horní a dolní obrys** dosud nakreslené plochy (horní a dolní horizont). Každou kreslenou čáru testujeme proti těmto obrysovým křivkám a nakreslíme z ní pouze ty části, které leží nad horním nebo pod dolním obrysem. Současně obrysy aktualizujeme.

Existuje několik variant tohoto algoritmu lišících se metodou reprezentace obrysových křivek. V nejpřesnější a zároveň nejpracnější variantě bychom si museli pamatovat obrysové křivky přesně - jako **hladké křivky**. Častěji se používá algoritmus, který povrchové křivky kreslí jako lomené čáry, tedy i oba obrysy mohou být reprezentovány **lomenými čarami**.

Nejsnadnější a nejrychlejší varianta (která však nemůže generovat vektorovou kresbu) využívá **rastru pixelů**, do kterého se obrázek vykresluje. Pro každou hodnotu vodorovné souřadnice v rastru (tj. pro každý svislý sloupec pixelů) si pamatujeme minimální a maximální dosud nakreslenou výšku ***y***. Každý pixel vykreslované čáry testujeme proti těmto hodnotám a nevykreslíme ho, pokud leží mezi nimi. Současně minimální a maximální hodnoty upravujeme, nejlépe však až po dokreslení celé čáry, aby nedošlo k přerušení jejích svislých úseků.

Všechny varianty algoritmu plovoucího horizontu můžeme upravit pro libovolný typ a směr projekce nebo pro vykreslování skrytých čar (např. jinou barvou nebo přerušovaně).

4.2. Malířův algoritmus

V tomto oddílu budeme mluvit o celé skupině algoritmů rastrového výpočtu viditelnosti, které využívají toho, že se výstup kreslí nejprve do nějakého **bufferu**, odkud je až na konci zobrazen (např. videopaměť grafického adaptéru nebo tisk na rastrové tiskárně s vyrovnávací pamětí).

Idea těchto metod spočívá v tom, že se scéna vykresluje **odzadu dopředu**, přičemž dříve nakreslené objekty mohou být částečně nebo úplně překryty pozdější kresbou. Samozřejmě musíme **vybarvovat** vnitřky kreslených ploch - jedná se tedy o plošné algoritmy. Vše, co potřebujeme spočítat, je vhodné **pořadí vykreslování** objektů scény.

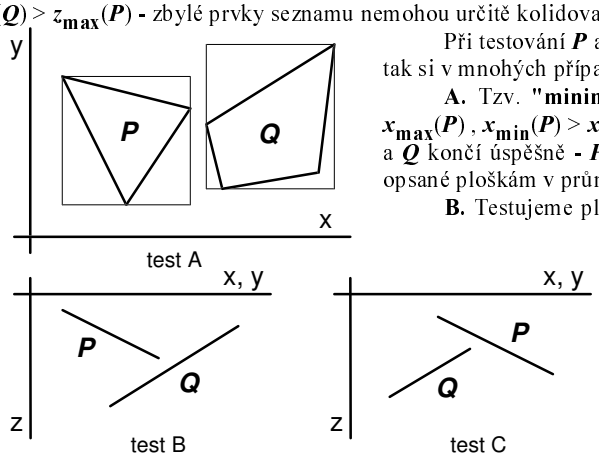
I. V některých případech lze snadno **explicitně** zjistit správné pořadí kreslení jednotlivých plošek; jako příklad nám opět může posloužit graf funkce dvou proměnných ***y=f(x,z)***: v definiční rovině ***xz*** snadno určíme správné pořadí vykreslování jednotlivých plošek (odzadu-dopředu).

II. Další variantou je tzv. **třídění podle hloubky**, kde se kreslené plochy pouze setřídí podle souřadnice ***z*** (hloubky) - uvažuje se nejvzdálenější bod nebo těžiště každé plochy. Tato metoda je nenáročná a rychlá, není však zcela korektní. Můžeme ji použít u scén tvořených velkým množstvím malých objektů: třeba při kreslení parametrických ploch rozdělených na tisíce až stotitisíce plošek.

III. Univerzální korektní varianta **malířova algoritmu** je poměrně složitější a my si ji zjednodušíme dodatečnými podmínkami: budeme předpokládat, že zobrazovaná scéna je složena z **rovinných plošek** (mnohoúhelníků) a navíc zakážeme jejich **vzájemné protínání** (prosekávání). Dvě různé plošky tedy budou smět mít společné body pouze po obvodu.

Celý algoritmus můžeme rozdělit na dvě fáze: v první z nich plošky setřídíme podle hloubky sestupně (k tomu použijeme minimální ***z***-souřadnici každé plošky) a uložíme do seznamu ***S***. Tím jsme dostali hrubé pořadí plošek, ze kterého budeme vycházet ve druhé fázi:

Postupně budeme odebírat plošky ze začátku seznamu ***S*** a každou před vykreslením otestujeme. Testovaného kandidáta označíme ***P***. Je třeba zjistit, zda můžeme tuto plošku nakreslit aniž bychom porušili správný vzhled kresby - musíme se přesvědčit, jestli ***P*** nezakrývá část nějaké dosud nenakreslené plošky (ty čekají v seznamu ***S***). Budeme tedy kontrolovat ***P*** proti všem ostatním ploškám ***Q*** z ***S***, se kterými by mohla kolidovat. Využijeme setřídění seznamu ***S*** podle rostoucího z_{min} a skončíme na plošce, která má $z_{min}(Q) > z_{max}(P)$ - zbylé prvky seznamu nemohou určitě kolidovat s ***P***.



Při testování ***P*** a ***Q*** budeme postupovat od nejjednodušších testů ke složitějším, a tak si v mnohých případech ušetříme práci. Obyčejně se popisují čtyři druhy testů:

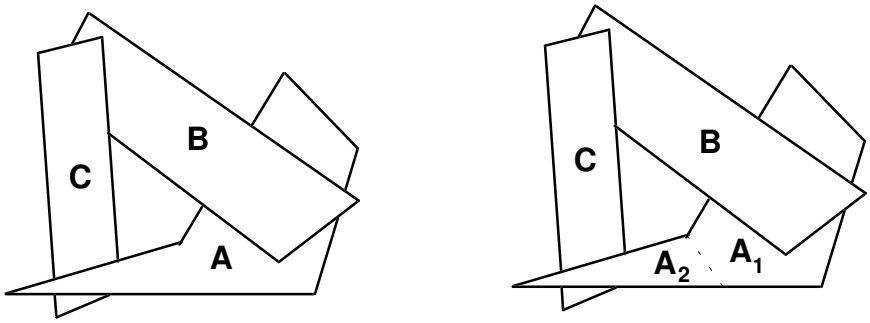
A. Tzv. **"minimax test"** - je-li splněna některá ze čtyř podmínek: $x_{min}(Q) > x_{max}(P)$, $x_{min}(P) > x_{max}(Q)$, $y_{min}(Q) > y_{max}(P)$ nebo $y_{min}(P) > y_{max}(Q)$, testování ***P*** a ***Q*** končí úspěšně - ***P*** může být nakreslena před ***Q*** (otestovali jsme vlastně obdélníky opsané ploškám v průmětu).

B. Testujeme plošku ***P*** proti **rovině plošky *Q***. Jestliže leží celá ***P*** za rovinou ***Q***, testování končí úspěšně.

C. Testujeme plošku ***Q*** proti **rovině plošky *P***. Jestliže leží celá ***Q*** před rovinou ***P***, testování končí úspěšně.

D. Musíme provést **úplný test** obou plošek **v průmětu** - bereme postupně všechny hrany plošky ***P*** proti všem hranám ***Q***. Jestliže se v průmětu žádná dvojice hran neprotíná a ani nenastává situace, že by průmět jedné z plošek ležel celý uvnitř druhého (stačí zkonstrolovat jeden vrchol pro každý z obou případů), testování končí úspěšně. Jestliže mají plošky ***P*** a ***Q*** společný bod v průmětu a ***P*** je v něm dále od pozorovatele, test též končí úspěchem.

V opačném případě jsme zjistili, že mají plochy ***P*** a ***Q*** špatné uspořádání. Provedeme s nimi obrácené testy **B**, **C** a pokud je to možné, přesuneme plochu ***Q*** před ***P*** (***Q*** se tedy stane novým kandidátem místo ***P***). Aby nemohlo dojít k zacyklení, musíme si u plošky P označit, že byla při tomto výběru již jednou kandidátem. Pokud se plocha stane kandidátem poprvé v rámci jednoho výběru, svědčí to o existenci **cyklického zákrytu** plošek. Takovou situaci řešíme rozštěpením některé plošky (viz obrázek - cyklus A - B - C byl rozštěpen a správné pořadí nakreslení je A_1 - B - C - A_2).



4.3. Paměť hloubky (Z-buffer)

Z-buffer je jeden z nejrozšířenějších algoritmů pro výpočet plošné viditelnosti, zejména pro svoji jednoduchost a možnost hardwarové implementace. Nevýhodou malířova algoritmu byla nutnost uspořádat plošky před vykreslováním - a právě tuto vlastnost z-buffer nemá, elementy se do něj mohou zapisovat v libovolném pořadí. Výpočet viditelnosti se provádí zvlášť pro **každý pixel** kresleného výřezu: k tomu slouží pomocná **paměť hloubky** (po které byl algoritmus pojmenován).

Paměť hloubky obsahuje jednu buňku pro každý pixel kresleného výřezu - v této buňce se pamatuje hloubka (*z*-souřadnice) bodu ve scéně, který byl do tohoto pixelu nakreslen. Pro uložení hloubky je nejpohodlnější použít reálný typ, výpočty jsou však potom pomalé a spotřeba paměti velká. Nejčastěji se místo toho používá 16- až 24-bitová celočíselná hodnota, i tak může dosáhnout velikost z-bufferu několika megabytů. Příklad: kreslíme na obrazovku v rozlišení 800×600. Při použití 16-bitových hodnot v paměti hloubky budeme potřebovat 937.5 KB.

Jak bude vypadat algoritmus výpočtu viditelnosti: předpokládejme, že máme k dispozici dostatečné množství paměti pro celý zobrazovaný výřez. Nejprve inicializujeme **obrazovou paměť** (video-RAM, buffer tiskárny, ..) barvou pozadí a **z-buffer** naplníme hodnotou "−∞" (větší hloubka, než může mít jakýkoliv objekt ve scéně). Pak již můžeme brát objekty scény v libovolném pořadí a zapisovat je do z-bufferu - každý objekt rozložíme na **pixely** (to bychom museli stejně udělat při kreslení do obrazové paměti) a u každého pixelu otestujeme jeho hloubku (*z*-souřadnici) s hodnotou zapsanou v příslušné buňce z-bufferu. Pokud je nová hodnota větší (nový pixel leží blíž než ten nakreslený), buňku z-bufferu přepíšeme a pixel zakreslíme do obrazové paměti; v opačném případě neděláme nic.

Díky své jednoduchosti a přímočarosti se tato metoda umí dobře vypořádat se všemi nestandardními situacemi: nevadí jí protínající se plochy ani cyklické zákryty, objekty ve scéně nemusí být rovinné (pouze je musíme umět rozložit na pixely = nakreslit). Již dlouho se vyrábí grafický hardware pro pracovní stanice, který právě pomocí z-bufferu řeší viditelnost 3D scén (stanice Silicon Graphics, Sun, Apollo, ..), v poslední době se začínají objevovat podobné 3D urychlovače i pro PC.

Na závěr si uveďme pár triků, které pomohou ušetřit paměť: pokud nám paměť nestačí na uložení z-bufferu pro celý obrázek, rozdělíme si jej na několik **vodorovných pruhů** a v každém z nich použijeme z-buffer zvlášť. Kreslení bude pomalejší, protože budeme muset procházet seznam objektů tolikrát, kolik máme těch pruhů. Extrémní úsporu paměti přináší tzv. **řádkový z-buffer**, kdy jako pruhy bereme jednotlivé řádky rastru (tehdy se nám vyplatí vymyslet nějakou rychlou metodu na vyloučení objektů, které nemohou určitě do dané řádky zasáhnout).

4.4. Watkinsův algoritmus řádkového rozkladu

Poslední algoritmus výpočtu viditelnosti, o kterém se zde zmíníme, je nejkomplikovanější. Odstraňuje však podstatnou nevýhodu malířova algoritmu i z-bufferu: nutnost použití obrazové paměti pro výstupní obrázek a především vícenásobné překreslování pixelů (všechny objekty, které jsou později překresleny, jsme vlastně kreslili zbytečně!). To vyjde najevo především při stínování, o kterém budeme mluvit v další kapitole.

Watkinsův algoritmus vychází z metody vyplňování obecného mnohoúhelníka v rovině tzv. řádkovým rozkladem. Na rozdíl od 2D případu se však počítá i se **souřadnicí *z*** a všechny plošky scény se zpracovávají **současně**. Existuje několik variant tohoto algoritmu, které se však liší pouze v metodě určení viditelnosti na aktuální řádce, základní princip algoritmu je společný.

Vstupní data: předpokládáme, že je 3D scéna uložena ve formě **rovinných plošek** (mnohoúhelníků), každá ploška je zadána posloupností svých vrcholů (3D souřadnice). Dále předpokládáme, že výstupní zařízení má rastrový charakter - obrazovka, rastrová tiskárna (jehličková, laserová), není vhodný plotr (souřadnicový zapisovač). Algoritmus rozdělíme do několika kroků:

1. **předzpracování**: ze scény odstraníme odvrácené plošky (ty, které určitě nemohou být vidět - např. u krychle najdeme vždy alespoň tři odvrácené stěny), zbylé plošky rozložíme na jednotlivé hrany, dále už budeme pracovat hlavně s hranami.

Vodorovné hrany odstraníme a pro všechny ostatní spočítáme některé pomocné údaje: záznam pro uložení **hrany** může vypadat třeba takto:

```
x : real;        { x-souřadnice horního koncového bodu, později souřadnice průsečíku s aktuální řádkou }
y : integer;     { y-souřadnice horního koncového bodu }
z : real;        { z-souřadnice horního koncového bodu }
dy : integer;    { výška hrany v pixelech: |y2-y| }
dxy : real;      { změna x-souřadnice při posunutí na následující řádku (směrnice pro x): (x2-x)/dy }
dzy : real;      { změna z-souřadnice při posunutí na následující řádku (směrnice pro z): (z2-z)/dy }
st : ^stena;     { odkaz na stěnu, do které patří }
```

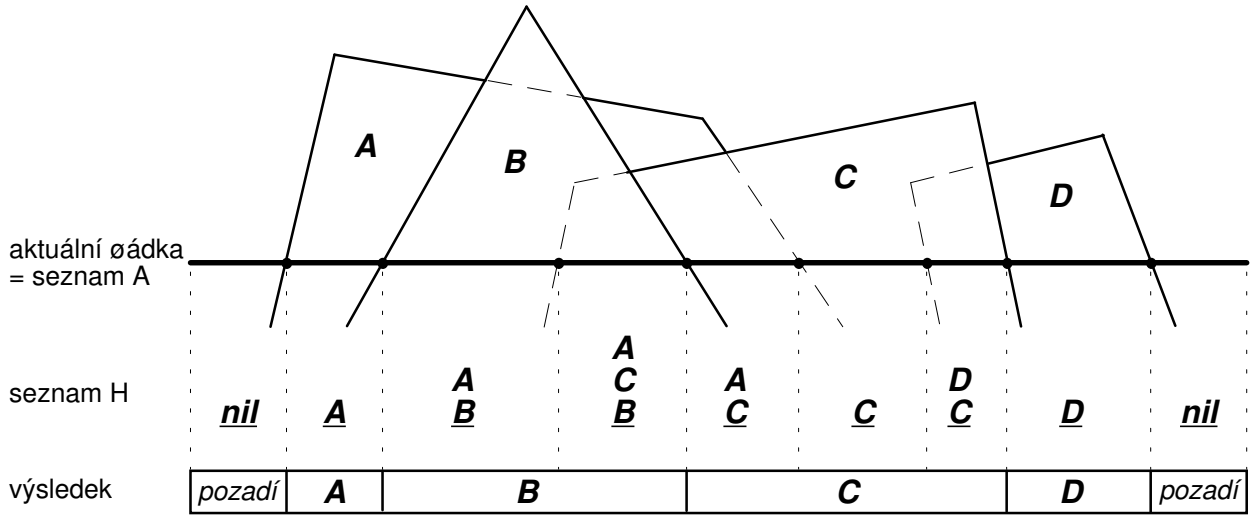
Záznam pro **stěnu** může vypadat takto:

```
dzx : real;      { změna z-souřadnice při posunutí na následující pixel (směrnice pro z): dz/dx }
barva : barvy;   { barva stěny }
...;             { další pomocné údaje pro výpočet viditelnosti }
```

2. **inicializace seznamu *S***: všechny předzpracované hrany setřídíme do **vstupního seznamu *S*** podle těchto kritérií:
   i. vzestupně podle ***y***
   ii. vzestupně podle ***x***
   iii. vzestupně podle ***dxy***
3. **inicializace seznamu *A***: v tzv. **aktuálním seznamu *A*** budeme mít uloženy všechny hrany, které budou protínat právě zpracovávanou řádku. Seznam budeme udržovat setříděný podle kritérií **ii.** a **iii.** z předchozího kroku. Na začátku zařadíme do ***A*** počáteční úsek seznamu ***S*** **-** hrany se shodným (tj. minimálním) ***y***.
4. **výpočet viditelnosti na aktuální řádce**: je třeba projít aktuální seznam ***A*** a určit viditelnost **-** k tomu můžeme například použít pomocný seznam stěn ***H*** setříděný podle hloubky. Na začátku řádky bude tento seznam prázdný. Budeme postupně procházet seznam aktuálních hran ***A*** a pro každou hranu se podíváme, zda její stěna je v ***H***. Pokud tam není, zařadíme ji tam a v opačném případě ji odstraníme (pro zatřiďování stěn do seznamu ***H*** podle hloubky použijeme pomocné údaje *z*, ***dzx***). **První prvek *H*** určuje **viditelnou stěnu** a pokud je seznam ***H*** **prázdný**, kreslíme **barvu pozadí** (viz obrázek).
5. **přechod na další řádku -** je třeba aktualizovat seznam ***A***, pro každý prvek ***A*** provedeme:

```
dy := dy - 1;
if dy=0 then "odstraň hranu ze seznamu A"
x := x + dxy;
z := z + dzy;
```

   Dále musíme zkontrolovat setřídění seznamu ***A*** (viz krok **3**) a zatřídit do něj hrany začínající na následující řádce **-** ty budou opět tvořit počáteční úsek vstupního seznamu ***S***.
6. **podmínka cyklu**: jestliže je alespoň jeden ze seznamů ***A***, ***S*** neprázdný, výpočet pokračuje krokem **4**. Jinak algoritmus končí.



Složitější varianta algoritmu musí počítat i s možností vzájemného **průniku dvou plošek**. My bychom tento případ poznali podle toho, že by si v seznamu ***H*** vyměnily dvě plošky svá pořadí. Pak by bylo potřeba dopočítat zvláštní umělou hranu, zařadit ji do seznamu ***A*** a vrátit se k jejímu předchůdci.

5. Stínování

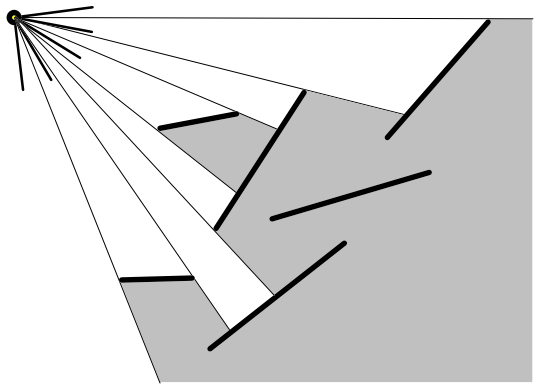
Odstraněním zakrytých hran a stěn těles podstatně zvětšíme přehlednost obrázku. Dalšího zlepšení kresby můžeme dosáhnout napodobením reálných světelných poměrů v prostorové scéně **- stínováním**. Jedná se v zásadě o dva principy: 1. kreslení stínů vržených neprůsvitnými tělesy ("**vrhané stíny**") a 2. změny ve vybarvení povrchu tělesa v závislosti na jeho osvětlení ("**stínování**").

5.1. Vrhané stíny

Začneme principem myšlenkově jednodušším **-** jde pouze o to, zjistit, zda je příslušný bod na povrchu tělesa osvětlen daným světelným sdrojem (obvykle předpokládáme pouze ideální bodové zdroje světla). Podle výsledku pak použijeme jeden ze dvou odstínů barvy tělesa a tím zvýrazníme stíny.

Existuje mnoho metod, jak vrhané stíny počítat. My si jen stručně popíšeme několik jednodušších:

Pomocí vhodného **algoritmu viditelnosti** spuštěného z pozice světelného zdroje lze pro každou plošku scény určit, která její část (pokud vůbec nějaká) je osvětlena zdrojem. Za předpokladu mnohoúhelníkové scény a bodového zdroje světla jsou obrysy stínů lomenými čarami, které si můžeme pro každou plošku zapamatovat a později použít při vybarvování.



Pojem **stínového tělesa** zahrnuje část prostoru, která není osvětlena daným bodovým světelným zdrojem. Jestliže máme scénu složenou z mnohoúhelníků, bude její stínové těleso soustavou (nekonečných) mnohostěnů. Pokud bychom uměli stínové těleso efektivně spočítat, mohli bychom jej použít při vykreslování plošek scény.

Jedna metoda výpočtu stínového tělesa předpokládá implementaci **operace sjednocení** nad (nekonečnými) mnohostěny: je jednoduché určit stínové těleso pro jednotlivou plošku ("elementární stínové těleso" je zkoseným nekonečným jehlanem), celkové stínové těleso dostaneme sjednocením všech elementárních stínových těles.

Další metoda využívá **prostorového třídění** podobně jako to dělal malířův algoritmus. Seřaďme všechny plošky scény od nejbližší k nejvzdálenější vzhledem ke světelnému zdroji (můžeme k tomu použít "obrácený" malířův algoritmus a perspektivní projekci). Potom budeme brát plošky v tom pořadí **-** o každé budeme vědět, že ji mohou zastínit pouze plošky předcházející. Pro každou plošku spočítáme její zastínění a případně aktualizujeme stínové těleso (místo složitého nekonečného mnohostěnu si můžeme pamatovat pouze soustavu jehlanů).

Pomocný z-buffer pro výpočet vrhaných stínů pracuje takto: nejprve provedeme kompletní výpočet viditelnosti pomocí z-bufferu, pozorovatele přitom umístíme do pozice světelného zdroje. Místo barev budeme do výstupního bufferu zapisovat čísla plošek. Potom již kreslíme scénu normálním způsobem s tím rozdílem, že každý kreslený pixel otestujeme: jeho skutečnou polohu v prostoru $[x,y,z]$ promítneme do pomocného výstupního bufferu a podíváme se, jaká ploška je v něm zobrazena. Pokud je to ta plocha, kterou právě kreslíme, znamená to, že kreslený pixel byl ze světelného zdroje vidět, a proto je osvětlený. A naopak.

5.2. Modely osvětlení

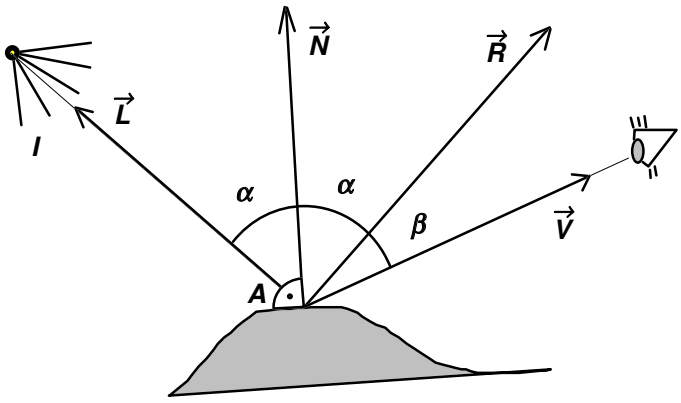
Dále se již budeme zabývat **stínováním**, tj. počítáním kvantitativních světelných poměrů na povrchu těles. Počítačová grafika se již desítky let snaží napodobit reálné světelné poměry v prostorových scénách. My si v následujících oddílech uvedeme jednoduché metody použitelné při klasickém výpočtu viditelnosti a v poslední kapitole budeme mluvit o modernějším a komplexnějším přístupu k řešení tohoto problému.

Základem stínování je tzv. **model osvětlení -** vhodná aproximace fyzikální reality, která musí být dostatečně věrná a přitom efektivně spočítatelná. Z mnoha užívaných si vybereme jeden z nejjednodušších **-** tzv. **Phongův model osvětlení**.

Bude nás zajímat jeden bod na povrchu tělesa (označme jej A), na který se pozorovatel dívá z daného směru a který je osvětlen jedním bodovým světelným zdrojem pod daným úhlem (viz obrázek). Nám půjde o to, jak spočítat barvu a intenzitu světla, které je z bodu A vyzářeno směrem k pozorovateli:

Pro **ideálně matné těleso** bychom mohli intenzitu odraženého světla spočítat jako přímou úměrnost $E = K \cdot \cos\alpha$, kde K je konstanta vyjadřující intenzitu a vzdálenost světelného zdroje a schopnost povrchu odrážet rozptýlené světlo. Pro odvrácené části tělesa bychom museli dodefinovat $E = 0$ a díky tomu by neosvětlené části zůstávaly zcela černé.

Z praktických důvodů se k předchozí **- difusní -** složce přičítá ještě tzv. **okolní světlo** ('ambient light'), které nahrazuje špatně vyčíslitelné nepřímé osvětlení.



Počítáme-li tedy s barevným ideálně matným tělesem ozářeným jediným bodovým zdrojem, můžeme použít kompletních vzorců:

$$E = E_A + E_D, \quad E_A = k_A \cdot C, \quad E_D = k_D \cdot I \cdot C \cdot \cos\alpha,$$

kde E_A je okolní světlo, E_D difusní světlo, k_A resp. k_D koeficienty odrazu okolního resp. difusního světla, C barva povrchu tělesa a I intenzita (a barva) světelného zdroje opravená případně i o jeho vzdálenost.

K těmto vztahům doplnil v roce 1975 Japonec Phong Bui-Tuong ještě vzorec pro **zrcadlově odražené světlo E_S**, které se projevuje ve formě polomatných odlesků na povrchu většiny skutečných těles:

$$E = E_A + E_D + E_S, \quad E_S = k_S \cdot I \cdot W(\alpha) \cdot \cos^n \beta,$$

kde k_S je koeficient zrcadlového odrazu, $W(\alpha)$ funkce aproximující Fresnelovy vzorce pro odraz světla (lze ji nahradit konstantou 1) a n exponent ovlivňující velikost odlesku (typicky několik desítek až stovek).

Na závěr použité vzorce upravíme do podoby, ve které se často používají - budeme předpokládat, že všechny vektory jsou normalizované a že nám ve scéně svítí N světelných zdrojů:

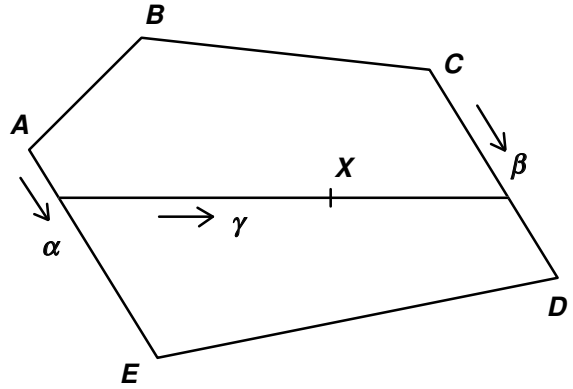
$$E = k_A \cdot C + \sum_{i=1}^{N} I_i \cdot \left(k_D \cdot C \cdot \left(\vec{N} \cdot \vec{L}_i \right) + k_S \cdot \left(\vec{V} \cdot \vec{R}_i \right)^n \right),$$

Konstanty popisující světelné vlastnosti povrchu tělesa: C, k_A, k_D, k_S, n, jediná veličina charakterizující světelný zdroj (jeho intenzitu, barvu a případně i vzdálenost) je I_i.

5.3. Stínovací algoritmy

Nejsnazší je stínovat scénu složenou z mnohostěnů a osvětlenou vzdáleným světelným zdrojem (nebo více vzdálenými zdroji). To pak můžeme po aplikaci některého z výše uvedených vzorců výslednou barvou vybarvit celou plošku. Říká se tomu **konstantní stínování** a je to nejrychlejší stínovací metoda. Bohužel se příliš nehodí pro stínování hladkých těles, která byla aproximována množstvím malých plošek, protože příliš zvýrazní přidané hrany (Machův efekt z psychologie vidění).

H. Gouraud navrhnul v roce 1971 metodu **interpolace barvy**, která se od té doby nazývá jeho jménem a která potlačuje nepříjemné zvýraznění umělých hran u aproximovaných těles. Předpokládejme, že chceme vystínovat hladké těleso aproximované mnohostěnem. Zavedeme si normálové vektory ve vrcholech mnohostěnu (buď je dostaneme z tvaru původního ideálního tělesa nebo je spočítáme z normál sousedních stěn pomocí aritmetického průměru). Pro každý vrchol spočítáme osvětlení podle příslušného vzorce a nakonec interpolujeme barvy z vrcholů do všech vnitřních bodů stěn. Nejčastěji se k tomu používá bilineární interpolace při kreslení řádkovým rozkladem (viz obrázek):



$$f_X = (1-\gamma) \cdot \left((1-\alpha) \cdot f_A + \alpha \cdot f_E \right) + \gamma \cdot \left((1-\beta) \cdot f_C + \beta \cdot \right.$$

kde koeficienty lineárních kombinací α, β a γ leží mezi 0 a 1 a spočítáme je všechny podobně, např.:

$$\alpha = \left(y_X - y_A \right) / \left(y_E - y_A \right).$$

Gouraudova metoda se dobře hodí pro stínování matných těles ($k_S=0$), ale tvar odlesků je při jejím použití velmi nepřesný. Proto Phong Bui-Tuong navrhnul současně se svým osvětlovacím modelem i dokonalejší stínovací metodu - tzv. **interpolaci normál**. Opět nejdřív spočítáme normálové vrcholy ve všech vrcholech aproximačního mnohostěnu. Potom však při kreslení řádkovým rozkladem interpolujeme tyto vektory do vnitřních pixelů plošek a teprve tam počítáme osvětlení podle příslušného vzorce. Tento postup se vyznačuje vysokou věrností barevných přechodů na zdánlivě hladkých tělesech, ale díky výpočtu osvětlení v každém pixelu plochy je mnohem pomalejší než např. interpolace barvy.

Několik poznámek k **bilineární kombinaci** používané při interpolaci barev i normál: interpolaci lze snadno zabudovat do algoritmu řádkového rozkladu při poměrně vysoké efektivitě výpočtu (pouze jedno sečtení a zaokrouhlení na každou interpolovanou veličinu a jeden průchod vnitřní smyčkou) - stínování lze v takovém případě přímo zabudovat do algoritmu na výpočet viditelnosti (např. z-buffer nebo Watkinsův algoritmus). Nepříjemná vlastnost: tato interpolace není obecně invariantní k otáčení, což se může nepříznivě projevit zejména při animacích (toto lze odstranit používáme-li jen trojúhelníkové stěny).

6. Rekurzivní sledování paprsku (Ray-tracing)

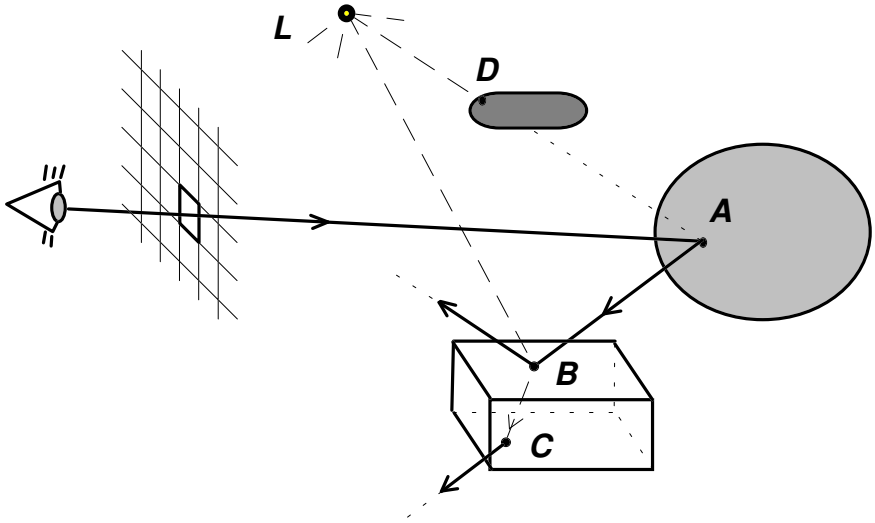
Poslední metoda, kterou si zde popíšeme, se staví k problému realistického zobrazení scény mnohem komplexněji. Snaží se o napodobení přírodních podmínek tím, že přímo **sleduje dráhy světelných paprsků**. Naneštěstí je distribuce světla ve scéně tak složitým problémem, že ani tato metoda nemůže být v mnohých případech úspěšná. Každopádně s ní můžeme dosáhnout velmi dobrých výsledků při zobrazování brilantních lesklých scén bez matných povrchů a velkého podílu rozptýleného světla. Dokážeme s ní rovněž věrně zobrazit průhledná tělesa včetně nejdůležitějších jevů provázejících odraz a lom barevného světla.

První publikovaná verze algoritmu je již z roku 1968, výpočty se tehdy prováděly na velkém sálovém počítači a výsledky se zobrazovaly na kreslicím stolu (plotru). Rozvoj metody však nastal až v osmdesátých letech s nástupem rychlých pracovních stanic vybavených kvalitními grafickými terminály. V odborných kruzích se dnes již metodě sledování paprsku tolik pozornosti nevěnuje, protože jsou známy kvalitnější algoritmy syntézy obrazu (bohužel jsou též mnohem výpočetně náročnější).

Základní myšlenka vychází ze simulace tzv. **dírkové kamery**, která zachycuje obraz pomocí malé dírky umístěné místo čoček objektivu. Kdybychom chtěli zjistit, jaké světlo dopadlo do určitého bodu stínítka, stačilo by sledovat zpětně chod paprsku skrz dírku kamery. Metoda sledování paprsku napodobuje ideální "dírkový objektiv" bez rušivého ohybu světla. Ve scéně se pak snaží nasbírat co nejvíce informací o světelných paprscích, které by mohly ovlivnit barvu v daném bodě stínítka. Místo "bod stínítka" budeme již teď říkat "pixel obrazovky" a na "dírkový objektiv" také zapomeneme.

Předpokládejme, že sledujeme zpětně dráhu paprsku ve scéně a narazíme při tom na povrch nějakého tělesa. Pokud je těleso **matné**, nemůžeme již dál pokračovat ve sledování nějakého paprsku, ale snažíme se co nejlépe odhadnout, jakou intenzitu a barvu by měl paprsek přicházející od toho tělesa **-** k tomu můžeme použít některého modelu osvětlení. Pokud však narazíme na **lesklý povrch** (bod ***A*** na obrázku), pokusíme se kromě klasického výpočtu osvětlení ještě o napodobení lesklého odrazu a budeme sledovat paprsek odražený podle známého pravidla elementární optiky. Na lesklém povrchu by se tak měl objevit více či méně silný odraz okolních předmětů, což se v reálném světě také stává. Úplně nejsložitější výpočet nás čeká, když narazí náš paprsek na **lesklý průhledný povrch** (bod ***B*** na obrázku) **-** tam bychom měli kromě odraženého sledovat i lomený paprsek světla, aby se objevila kresba, která je vidět skrz průhledné těleso.

Protože budeme chtít zachytit i vícenásobné odrazy a lomy světla, bude nutné formulovat algoritmus **rekurzivně**, tj. každý paprsek (primární = poslaný z pixelu obrazovky i ten jedou odražený) se bude moci dále odrážet nebo zalamovat (viz obrázek).



Jádrem algoritmu sledování paprsku je rekurzivní funkce **Sleduj()**, jejíž hrubé schema si teď ukážeme:

```
function Sleduj ( P0, P1, hloubka ) : RGB;   { P0..počátek paprsku, P1..směr, hloubka..počet odražení }
var     A, R, T : Point3D;                   { pomocné body a vektory }
        B : RGB;                             { výsledná barva }
begin
  A := Prusecik(Scena,P0,P1);                { průsečík paprsku se scénou }
  if A=0 then Sleduj := Pozadi               { paprsek na nic nenarazil }
         else
    begin                                    { paprsek narazil na těleso }
      B := 0;
      for i := 1 to N do                     { příspěvky od světelných zdrojů }
        if Prusecik(Scena,A,L[i]-A)=0 then
          B := B + kL*Svetlo(A,L[i]);
```

```
      hloubka := hloubka + 1;
      if hloubka <= maxhloubka then   { konec rekurze }
        begin
          if "A je odrazivé" then
            begin
              "spočítej R"              { odražený paprsek }
              B := B + kR*Sleduj(A,R,hloubka);
            end;
          if "A je průhledné" then
            begin
              "spočítej T"              { lomený paprsek }
              B := B + kT*Sleduj(A,T,hloubka);
            end;
        end;
      Sleduj := B;                      { nastřádaná návratová hodnota }
    end;
end;
```

V kódu funkce **Sleduj()** jsme použili jednoduchý způsob **řízení hloubky rekurze** (tzv. statickou metodu) - předem byla dána konstanta **maxhloubka**, která omezuje počet sledovaných odrazů/lomů.

Dokonalejší (dynamická) metoda sleduje "**důležitost paprsku**", číslo udávající procentuální příspěvek právě počítaného paprsku k výsledné barvě pixelu. Pokud tato veličina klesne pod předem danou mez (2-10%), rekurze se zastaví.

6.1. Výpočet průsečíků

Výpočet průsečíku daného paprsku se scénou (funkce **Prusecik()** z předchozího příkladu) je jednou z časově nejnáročnějších operací celého algoritmu (udává se číslo 80-90%), zejména proto, že se volá poměrně často - mnohokrát pro každý pixel. Proto je velmi důležité, aby byl výpočet průsečíku co nejefektivnější.

Průsečíky s geometricky jednoduchými tělesy se často počítají **analyticky** (metodami analytické geometrie). Jedná se například o útvary: obdélník, kvádr, koule, válec, kužel. Pro některá tělesa (např. pro složité mnohostěny) potřebujeme provést velké množství takových výpočtů. Jako příklad si uvedeme výpočet průsečíku s nejpopulárnějším tělesem Ray-tracingu: **koulí**. Předpokládáme vyjádření paprsku v parametrickém tvaru $P_0 + t \cdot P_1$, kde $0 \le t < \infty$. Do rovnice koule se středem $\left[S_x, S_y, S_z\right]$ a poloměrem $\boldsymbol{R}$: $\left(x - S_x\right)^2 + \left(y - S_y\right)^2 + \left(z - S_z\right)^2 - R = 0$ dosadíme za x, y a z jednotlivé složky z rovnice paprsku a dostaneme tak kvadratickou rovnici s neznámou t, jejímž řešením spočítáme průsečík(y) paprsku s tělesem. Podobně bychom postupovali u ostatních kvadratických ploch.

Pro složitější tělesa - například obecné parametrické plochy nebo rotační tělesa, aproximační pláty - není možné sestavit analytický výpočet průsečíku s paprskem. V takovém případě je nutné použít **numerických metod** a počítat polohu průsečíku iteračně - přibližně.

Objemový model (**CSG** - viz oddíl 2.2) je nejpopulárnější metodou používanou k reprezentaci scén pro rekurzivní sledování paprsku. Předpokládáme, že umíme spočítat průniky paprsku se všemi používanými elementárními tělesy - jde pak pouze o to, jak uplatnit množinové operace. A zde se ukazuje, že můžeme množinové operace nad elementárními tělesy převést pomocí distributivního zákona na množinové operace na přímce (polopřímce). Příklad: spočítat přímo průsečík paprsku $\boldsymbol{P}$ s tělesem $A \cap \left(B \cup C\right)$ je obtížné. Naštěstí platí $P \cap \left[A \cap \left(B \cup C\right)\right] = \left(P \cap A\right) \cap \left[\left(P \cap B\right) \cup \left(P \cap C\right)\right]$ a my můžeme provádět množinové operace až na polopřímce paprsku. Průniky paprsku s tělesy jsou pouze množiny intervalů na polopřímce, se kterými se dá snadno počítat. Obsahuje-li CSG-strom i prostorové transformace, můžeme použít další trik: nebudeme transformovat tělesa v CSG-stromu, ale inverzní transformací přepočítáme polohu paprsku. Tak lze mít všechna elementární tělesa v **základních polohách**, kde se průsečíky počítají ještě snadněji (např. koule je vždy jednotková a se středem v počátku,..).

6.2. Textury

Přirozenějšího vzhledu těles dosahujeme nanášením tzv. **textur** na jejich povrch. Textura je jakýsi vzorek, který může ovlivnit barvu, lesk nebo směr odrazu paprsků a slouží k napodobení reálně se vyskytujících povrchů jako je: kůra stromu, vyleštěné dřevo nebo mramor, slupka jablka, banánu, pomeranče, atd. Používají se dvě odlišné metody nanášení textur, říká se jim plošné nebo prostorové (objemové) textury.

Plošnou texturu si můžeme představit jako tapetu, kterou polepíme povrch daného tělesa. Z matematického hlediska se vlastně jedná o zobrazení, které dvojici povrchových souřadnic [u,v] přiřazuje hodnotu barvy, odrazivosti, atd. Při výpočtu metodou

rekurzivního sledování paprsku je při tom nutné provádět tzv. **inverzní mapování** textury - 3D souřadnice průsečíku [***x***,***y***,***z***] převádět na povrchové souřadnice [***u***,***v***]. To nemusí být vždy triviální - například na kulovou plochu nelze bez zkreslení nanést čtvercovou texturu (kouli nemůžeme jednoduše oblepit tapetou), v některých případech však lze použít (zkreslující) polární souřadný systém.

Prostorové textury vystihují objemovou strukturu materiálu - např. rozložení letokruhů ve dřevě nebo vrstviček nerostů v nějaké hornině (mramoru). Najít správné vzorce nebo algoritmy pro výpočet objemové textury může být velmi obtížné, při zobrazování se zase vyhneme nutnosti mapovat texturu: spočítané 3D souřadnice průsečíku [***x***,***y***,***z***] se přímo použijí k výpočtu textury. Při napodobování přírodních materiálů jsou často využívány pseudonáhodné **šumové funkce** pro dosažení přirozeného - nepravidelného vzhledu. Příklad: dřevěný materiál můžeme napodobit funkcí

$$Barva(x,y,z) = L\Big(\big[D(x,y,z) + S_1(x,y,z)\big] \cdot \big[1 + S_2(x,y,z)\big]\Big),$$

kde ***D*** je vzdálenost bodu od středu kmene, ***L*** cyklická funkce modelující změnu barvy přirostlého dřeva v průběhu roku a S_1, S_2 dvě nezávislé spojité šumové funkce.

6.3. Vyhlazování (anti-aliasing)

Při kreslení na obrazovku bývají šikmé linie často znehodnoceny jemným zoubkováním. Tento jev se nazývá **alias** a vyskytuje se v každém rastrovém prostředí. Při použití textur se můžeme setkat s ještě výraznějším artefaktem - v pravidelné mřížce ubíhající do dálky dochází k nepříjemným interferencím (textura interferuje s rastrem obrazovky).

Při výpočtu obrázku metodou sledování paprsku je alias způsoben tím, že se barva určuje v **pravidelné mřížce pixelů** - pro každý pixel vrhneme jeden primární paprsek. Většinu nepříjemných jevů můžeme potlačit pomocí následující úvahy: pixely si nebudeme představovat jako body, ale jako malé čtverečky nebo obdélníčky. Barvu každého pixelu pak budeme chtít spočítat jako **průměrnou hodnotu barvy** na ploše pixelu (pro matematiky - integrální průměr obrazové funkce na ploše pixelu). Poněvadž nelze spočítat průměr z nekonečně mnoha hodnot (nelze analyticky vyčíslit hodnota tohoto integrálního průměru), používají se v praxi nejčastěji metody typu **Monte-Carlo**:

Plocha pixelu se pokryje množstvím náhodně rozmístěných vzorků tak, aby byly rozmístěny relativně rovnoměrně. Pak se každým vzorkem vyšle jeden paprsek a ze všech získaných hodnot se spočítá aritmetický průměr. Čím víc vzorků (paprsků) použijeme, tím bude výsledek kvalitnější, ale výpočet bude pomalejší (rozumné maximum je 100 paprsků/pixel).

Šikovný kompromis se jmenuje **adaptivní zjemňování**: každým pixelem obrázku pošleme relativně malý počet paprsků (2-4) a potom paprsky přidáváme jen v těch pixelech, ve kterých se "něco mění" (jedna barevná plocha přechází v druhou, mění se barva textury, atd.).

7. Literatura

[Foley90] Foley J., van Dam A., Feiner S., Hughes J.: Computer Graphics, Principles and Practice, 2nd edition, Addison-Wesley, Reading, 1990

[Glassner89] ed. Glassner A. S.: An Introduction to Ray Tracing, Academic Press, London, 1989

[Glassner95] Glassner A. S..: Realistic Image Synthesis, vol. 1 & 2, ?, 1995

[NewSpr79] Newmann W., Sproull R. F.: Principles of Interactive Computer Graphics, 2nd edition, McGraw-Hill, New York, 1979

[Skala91] Skala V.: Počítačová grafika I-III, skriptum VŠSE Plzeň, 1991

[SoŽá92] Sochor J., Žára J.: Algoritmy počítačové grafiky, skriptum ČVUT FEL Praha, 1992

[Žára90] Žára J. a kol.: Počítačová grafika, principy a algoritmy, Grada a.s., Praha, 1990