3D počítačová grafika na PC

Josef Pelikán, KSVI MFF UK Praha
http://cgg.ms.mff.cuni.cz/

Cílem této přednášky je nastínit některé základní principy zpracování trojrozměrné počítačové grafiky na soudobých počítačích třídy „PC”. Dnešní běžné domácí počítače obsahují tak výkonné grafické akcelerátory, že se hodí ke zpracování 3D grafiky lépe, než specializované profesionální grafické stanice před 5 až 10 lety. Hlavně za to vděčíme zábavnímu („hernímu”) průmyslu, který za poslední dekádu posunul hardware i software o velký kus dopředu – zejména je důležitý vliv na výrobu ve velkých sériích a neustále se snižující ceny grafických čipů („GPU”).

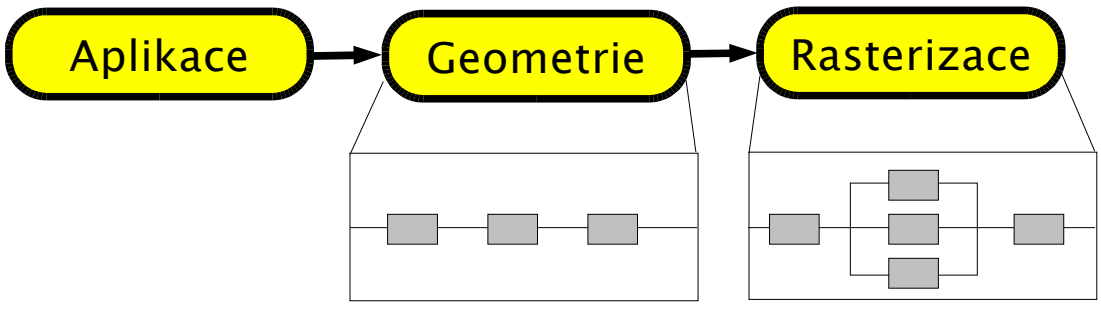
Někteří analytikové dokonce tvrdí, že vývoj grafických čipů je dynamičtější a přináší více inovací, než klasický průmysl univerzálních procesorů („CPU”). Rozhodně si pokroky ve vývoji GPU vyžádaly za poslední léta mnoho změn v architektuře i programovém vybavení PC počítačů: grafické sběrnice VLBus a AGP, speciální rychlé interní sběrnice uvnitř „chipsetu”, velmi rychlé paměťové čipy, programátorská rozhraní OpenGL nebo Direct3D, apod. Jenom na ukázku uveďme, že typický moderní procesor používá paralelní zřetězené zpracování („pipeline”) s délkou řádově 20 fází, grafické akcelerátory obsahují zřetězení až 800 fází[1] – podle typu zpracovávaných dat.

Předmětem našeho zájmu tedy bude nejen to, jak se 3D grafika v počítačových programech používá, ale též principy velmi výkonné akcelerace, kterou poskytují soudobé GPU. Abychom mohli porozumět principům 3D grafiky, musíme si nejprve zopakovat některé základní pojmy z matematiky, zejména jde o lineární algebru. Potom bude následovat stručný popis datových reprezentací pro klasickou 3D počítačovou grafiku (omezíme se zde pro jednoduchost na tzv. povrchové modely) a dále se již budeme zabývat jednotlivými technikami používanými při zobrazování 3D modelů. Neopomeneme zmínit nejdůležitější techniky zlepšující vzhled výsledného obrazu (průhlednost, textury, stínování, mip-mapping, neizotropické filtrování, anti-aliasing, mlha) nebo urychlující grafický výkon (Z-buffer, stencil-buffer), či prostředky umožňující pružně měnit konfiguraci GPU (programování GPU: „vertex-shaders” a „fragment-shaders”). Pro ty nejzvídavější budou připraveny ukázky programování 3D grafiky v prostředí MS Windows (za pomoci DirectX API).

Tento text nemá ambice být uceleným výkladem problematiky, vážné zájemce odkazuji na množství literatury – alespoň nejdůležitější zdroje informací najdete v posledním oddíle.

1 Fáze grafického zpracování

I když zatím neznáme podrobnosti o datech ani grafických algoritmech, je vhodné si uvést přehled typického 3D zobrazovacího řetězce (bez ohledu na to, zda/jak je podporován hardwarem). Na následujícím diagramu je hrubé rozdělení zobrazovacího řetězce:



1 Pentium 4 versus NVidia GeForce 3

Jednotlivé etapy zpracování mohou být samozřejmě velmi složité a často v sobě obsahují zřetězení mnoha dílčích kroků (jako v případě geometrického zpracování) nebo dokonce paralelní výpočet (jak bývá zvykem u finální rasterizace = převodu na pixely).

Typicky obsahuje fáze „**Aplikace**" reprezentaci 3D dat a jejich jakékoli aplikačně závislé zpracování: pohyby těles, fyzikální simulace virtuálního světa, rozhodování o tom, které části scény jsou důležitější (a které se tedy budou kreslit nejpřesněji – tzv. „Level of Detail" = „LoD"), interakce mezi dynamickými předměty, „umělou inteligenci" (zejména důležitá je ve video-hrách, ale též v ostatních realistických simulátorech), apod. Implementace je výhradně na straně software, další podrobnosti jsou zcela mimo rámec naší přednášky.

Fáze „**Geometrie**" je zodpovědná za většinu operací prováděných nad jednotlivými ploškami (obecně polygony, často jen trojúhelníky) a jejich vrcholy: geometrické transformace v 3D, osvětlení, projekce, ořezávání, transformace a ořezávání ve 2D. Často je implementována pomocí jednoho dlouhého řetězce („pipeline"), pokud je alespoň některá část přenesena na hardware (říká se tomu „hardware T&L"), uplatňuje se zde i paralelismus (nezávislé zpracování).

Finální fáze „**Rasterizace**" má za úkol převést geometrická data („primitiva" – nejčastěji body, čáry a trojúhelníky) do rastrové reprezentace, prostě řečeno: „vykreslit 2D/3D objekty na rastrový displej". Nejdůležitějšími technikami jsou: určování viditelnosti (Z-buffer), mapování textur, interpolace barev (stínování), zpracování průhledných objektů, mlha, práce s šablonou („stencil"), apod. Některé z těchto kroků jsou již přes deset let implementovány pomocí hardware, ve spotřební oblasti (domácí počítače) se GPU uplatňují zhruba sedm let[2]. Stupeň paralelního zpracování je zde velmi vysoký, protože se výsledky zapisují do poměrně pomalé video-paměti, do které musí mít současně přístup generátor řádkového rozkladu (dvoucestná paměť). Můžeme se setkat i s desítkou paralelně pracujících rastrovacích jednotek na jednom GPU čipu.

2 Teorie

V tomto oddíle si zopakujeme nejdůležitější pojmy z matematiky a optiky, které 3D počítačový grafik každodenně používá. Pro případné další studium doporučuji libovolnou vhodnou učebnici matematiky (lineární algebry, geometrie), základy bývají shrnuty i ve většině dobrých učebnic počítačové grafiky (např. [1], [2], [3]).

2.1 Lineární algebra a geometrie

Při popisu souřadnic 3D objektů se (kromě obyčejných kartézských souřadnic) používají tzv. **homogenní souřadnice**. Je to čtveřice [x, y, z, w], kde poslední složka „w" je nulová pro vektory (směry) a nenulová pro vlastní body trojrozměrného prostoru. Je-li homogenní složka nenulová, ale různá od 1, můžeme spočítat normalizované souřadnice [x/w, y/w, z/w, 1]. V normalizovaném tvaru [x, y, z, 1] jsou první tři složky identické s „obyčejnými" nehomogenními souřadnicemi bodu.

Jaký důvod máme pro používání homogenních souřadnic? Spolu s homogenními transformačními maticemi (rozměrů 4×4) nám to umožňuje jednotně reprezentovat nejen rotace, protažení (změnu měřítka), zkosení, apod., ale též posunutí. Díky tomu je snadné např. libovolně umístit osu otáčení, střed protažení nebo zkosení.

Transformační matice budeme zásadně chápat jako homogenní, jejich aplikace na souřadnice bodu spočívá v násobení **řádkového vektoru souřadnic** maticí **zprava**[3]:

2 Akcelerátor „3Dfx Voodoo 1" byl na trh uveden v roce 1996

3 to odpovídá variantě, kterou si zvolili návrháři knihovny Direct3D, jiné systémy (např. OpenGL) mohou používat úplně opačný přístup (matice jsou transponované, vektory sloupcové)

$$[x, y, z, w]\cdot\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = [x', y', z', w']$$

Nejběžnější transformační matice mají speciální tvar posledního sloupce – proto se také někdy ukládají jen jako třísloupcové:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ t_1 & t_2 & t_3 & 1 \end{bmatrix}$$

Zde levá horní submatice a_{11} až a_{33} vyjadřuje rozměr a orientaci, vektor [t_1, t_2, t_3] posunutí (translaci) a jednotkový čtvrtý sloupec naznačuje, že se jedná o **afinní transformaci** (zachovávající rovnoběžky). Matice **perspektivní projekce** má naopak netriviální poslední sloupec.

Několik ukázek elementárních transformačních matic – rotace kolem osy „z" o úhel „α", natažení objektu ve směru osy „x" s koeficientem 1.5 a posunutí o vektor [2, 1, -3]:

$$\begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1.5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & -3 & 1 \end{bmatrix}$$

Díky asociativitě maticového násobení lze posloupnost několika navazujících maticových transformací spočítat jako násobení jedinou (složenou, kompozitní) maticí:

$$(([x, y, z, w]\cdot M_1)\cdot M_2)\cdot M_3 = [x, y, z, w]\cdot(M_1\cdot M_2\cdot M_3)$$

Pomocí skládání maticových transformací lze snadno spočítat matice pro otáčení kolem libovolné osy, protažení podle libovolného vektoru, apod.

Významnou roli mezi transformacemi hrají tzv. **transformace tuhého tělesa** („rigid-body transform"), složené pouze z otáčení a posunutí. Pomocí jedné translace a tří otočení lze například sestrojit matici, která mezi sebou převádí dvě souřadné soustavy – ztotožňuje dva uživatelské systémy definované kartézskými osami (x, y, z) a (v, u, l). Při označení druhého systému jsme použili notaci souřadného systému spojeného s polohou pozorovatele: „v" je směr pohledu („view"), „u" vektor mířící vzhůru („up" – temeno hlavy pozorovatele) a „l" vektor levé upažené ruky („left hand").

Dále se budeme zabývat **projekčními transformacemi**: pro jednoduchost předpokládáme, že uživatel je umístěn v základní poloze – v počátku souřadnic – a dívá se ve směru osy „z".

Rovnoběžné promítání (odpovídající pohledu z nekonečna) se implementuje pouhým „zapomenutím" třetí souřadné složky (složky „z"). **Perspektivní projekce** musí navíc zohlednit vzdálenost pozorovaného objektu – čím je předmět dál, tím se zobrazí jako menší. Jednoduchý vzorec pro perspektivní projekci by mohl vypadat: [x/z, y/z], nebo se použije následující homogenní transformační matice („d" je vzdálenost projekční roviny od pozorovatele; připomínáme, že pozorovatel je v počátku souřadnic a dívá se ve směru kladné poloosy „z").

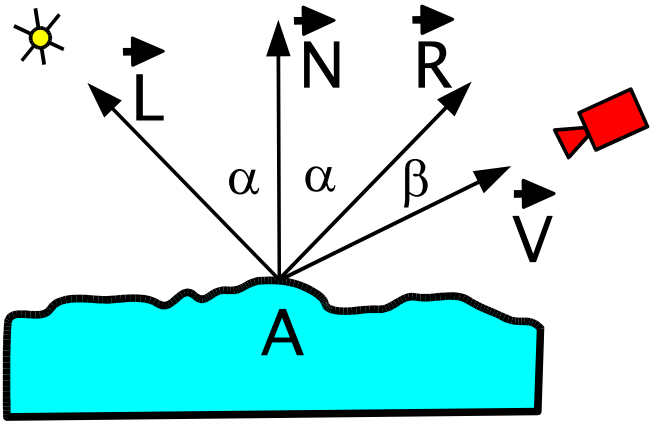
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1/d \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

V praxi se používají komplikovanější matice, které transformují celý objem zorného úhlu („**view frustum**") do kvádru nebo krychle, nejčastěji umístěných kolem počátku souřadnic. V systému Direct3D se například jedná o kvádr [-1, -1, 0] až [1, 1, 1].

Kompletní projekční transformace se tedy skládá z převodu souřadných soustav (souřadná soustava spojená s pozorovatelem se převede do světového systému souřadnic) následovaného některou z matic perspektivní projekce.

2.2 Osvětlení

Aby zobrazení trojrozměrných těles bylo více plastické, používají se v počítačové grafice různé **osvětlovací modely**. V nejjednodušší formě to jsou vzorečky popisující odraz světla na povrchu tělesa – podle orientace plošky vzhledem ke světelnému zdroji se spočítá barevný odstín a pozorovatel tak získá lepší představu o prostoru. Do „osvětlení" se obyčejně nezahrnuje výpočet stínů (které části předmětu jsou skutečně zdrojem osvětleny a na které vrhne stín nějaká překážka), to budeme diskutovat až později..



Na obrázku je znázorněna situace, kdy na povrch tělesa do bodu „A" svítí ze směru „L" světelný zdroj. Normálový vektor je označen „N", směr k pozorovateli „V" a vektor dokonalého (zrcadlového) odrazu „R". Vektory „L", „N" a „R" svírají úhel „α", vektory „R" a „V" úhel „β".

Jedním z nejjednodušších světelných modelů je **Phongův model**: světlo odrážející-se do směru „V" se skládá ze tří složek: okolní neboli **zbytkové světlo** („ambient light") „L_a", **difusní odraz** („diffuse") „L_d" a **lesklý odraz** („specular") „L_s". Zjednodušené vzorce pro výpočet těchto tří složek:

$$L_a = C \cdot k_a$$
$$L_d = (C \cdot C_L) \cdot k_d \cdot \cos\alpha$$
$$L_s = C_L \cdot k_s \cdot \cos^h \beta$$

„**C**" je vlastní barva povrchu tělesa, „$\mathbf{C_L}$" barva a intenzita zdroje, konstanty „$\mathbf{k_a}$", „$\mathbf{k_d}$" a „$\mathbf{k_s}$" slouží k nastavení vlastností materiálu (lesklý, matný, apod.) a exponent „**h**" ovlivňuje ostrost zrcadlového odlesku (součin dvou barevných vektorů se počítá po složkách). Nejvážnější

zjednodušení spočívají v tom, že barva zrcadlového odlesku je totožná s barvou světla a dále v tom, že „k_s" považujeme za konstantu. Přestože není tento elementární model fyzikálně věrný, pro použití v rychlém zobrazování většinou postačí.

Poznámka: kosinus úhlu, který svírají dva jednotkové vektory, se počítá velmi jednoduše – je to jejich **skalární součin**. To znamená ve 3D tři násobení a dvě sčítání, některé architektury (MMX SSE, 3DNow! nebo assemblery GPU) na to mají dokonce speciální instrukce!

Pokud se ve scéně vyskytuje více zdrojů světla, jejich „L_d" a „L_s" se sčítají („L_a" se započítá pouze jedenkrát). Je-li zdroj světla relativně blízko ve scéně (tj. není „v nekonečnu"), měl by se započítat útlum jeho intenzity podle převrácené hodnoty čtverce jeho vzdálehosti od bodu „A". V praxi se však tento vztah nahrazuje slabším vzorcem, kde se vzdálenost objevuje v menší mocnině.

2.3 Mlha

Při průchodu světla okolním prostředím (atmosférou) dochází ve skutečnosti k mnoha jevům – z nich se v počítačové grafice nejčastěji počítá pohlcení/rozptyl paprsku v mlze. Barva mlhy „**C_f**" (obvykle bílá) se mísí s vypočtenou barvou povrchu tělesa „**C_s**" podle vzorce:

$$C = f\,C_S + (1-f)\,C_f$$

kde „**f**" je koeficient vypočítaný podle některého z následujících vzorců (první je jednodušší lineární mlha, druhý vzorec popisuje fyzikálně věrnější exponenciální mlhu):

$$f = \frac{z_{end} - z}{z_{end} - z_{start}} \qquad f = \mathrm{e}^{-D \cdot z}$$

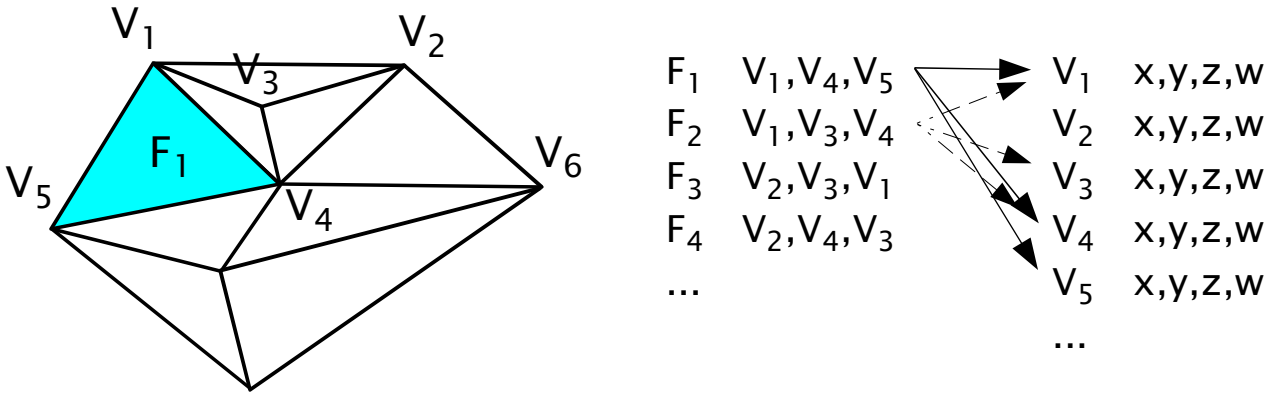
„**z_{end}**" a „**z_{start}**" udávají vzdálenosti začátku a konce mlhy od pozorovatele, „**z**" je vzdálenost vykreslovaného bodu od pozorovatele. „**D**" definuje „hustotu" mlhy – čím je toto číslo větší, tím je mlha hustší.

3 Reprezentace 3D scény

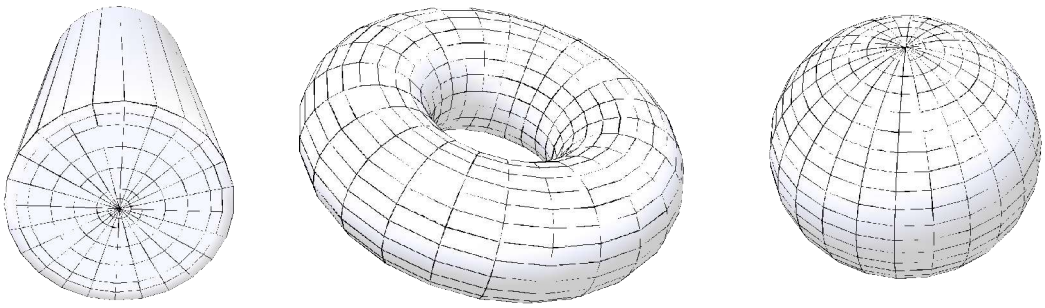
V hardwarově podporované 3D grafice se používá v podstatě jenom jedna reprezentace scény – všechny objekty jsou definovány pomocí svého povrchu (proto se této reprezentaci někdy říká povrchová, „**B-rep**" jako „boundary representation"). A jelikož je potřeba povrch těles definovat co nejúsporněji, používáme k tomu **síť mnohoúhelníků** (polygonů) – všechna naše tělesa jsou tedy mnohostěny. I když se při modelování někdy používají i složitější plošky, do grafického vykreslovacího systému se většinou už posílají jenom **trojúhelníky**.

Je třeba si uvědomit, že mnohostěn nemůže dokonale nahradit hladká tělesa – i u tak jednoduchého tělesa, jako je třeba koule, narážíme při aproximaci mnohostěnem na mnohé problémy. Obecně platí: čím přesnější (věrnější) aproximaci chceme dostat, tím větší počet trojúhelníků musíme použít. A tím pomalejší bude vykreslování modelu v grafické kartě.

V paměti počítače se síť trojúhelníků pamatuje snadno – stačí mít uložené 3D souřadnice všech vrcholů sítě a každý trojúhelník si pak už jen pamatuje, které tři vrholy mu patří (že se vrcholy mezi jednotlivými trojúhelníky sdílejí, je přirozené: ušetříme tak paměť i čas procesoru). Vrcholy tělesa mohou nést ještě další informace – například barvu, normálový vektor (pro stínování), texturovou souřadnici (bude vysvětleno později) atp. Běžné dnešní modely obsahují stovky vrcholů (menší tělesa) až milióny vrcholů (ohromné objekty, celá scéna).



Při práci s povrchovou reprezentací musíme pamatovat na to, že všechny plochy jsou implicitně **jednostranné**, tj. z každého trojúhelníka může být vidět pouze jeho lícová strana (která to je, bývá zadáno jeho orientací nebo orientací normálového vektoru). Vykreslovací systém obyčejně ignoruje ty trojúhelníky, které jsou k pozorovateli obráceny rubovou stranou. U regulárně uzavřených mnohostěnů se ani nesmí stát, že by bylo „vidět" dovnitř tělesa..



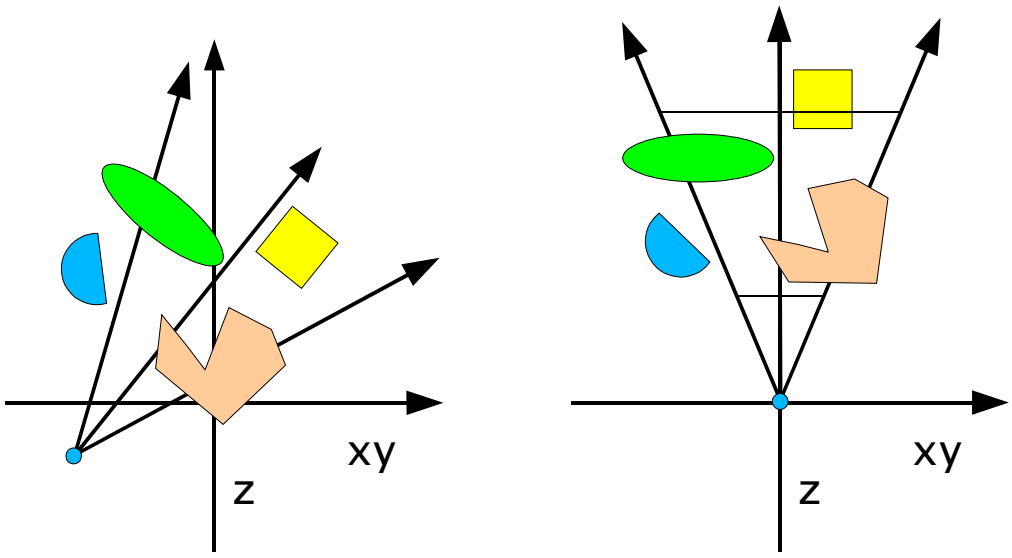
3.1 Level of Detail

Moderní grafické akcelerátory umějí zobrazit kolem 10^7 až 10^8 trojúhelníků za sekundu. To ale nestačí pro plynulý pohyb v případě, že naše scéna v plné přesnosti má 10^7 nebo více trojúhelníků. Nastupují techniky souhrnně nazývané „**Level of Detail**" (LoD): využívají toho, že pozorovatel si může dobře prohlížet pouze objekty, ke kterým má blízko. Vzdálené předměty se mohou kreslit zjednodušeně nebo se dokonce zcela eliminují. Systém LoD má za úkol automaticky připravovat různé stupně přesnosti modelu podle toho, jak blízko k danému objektu má pozorovatel. Primitivní algoritmy umějí pouze „přepínat" mezi předem připravenými modely (pokud není systém dobře vyladěn, můžeme pozorovat rušivé „přeskakování"), dokonalejší řešení obsahuje metody, jak adaptivně přepočítávat složitost trojúhelníkové sítě podle vzdálenosti k pozorovateli. Pokud se pozorovatel posune o malý kousek, bývá i změna sítě malá.

Kvalitní algoritmy pro LoD jsou poměrně komplikované a v rámci této přednášky se jimi nemůžeme podrobněji zabývat – zájemce nechť se poučí v literatuře.

4 Geometrické zpracování

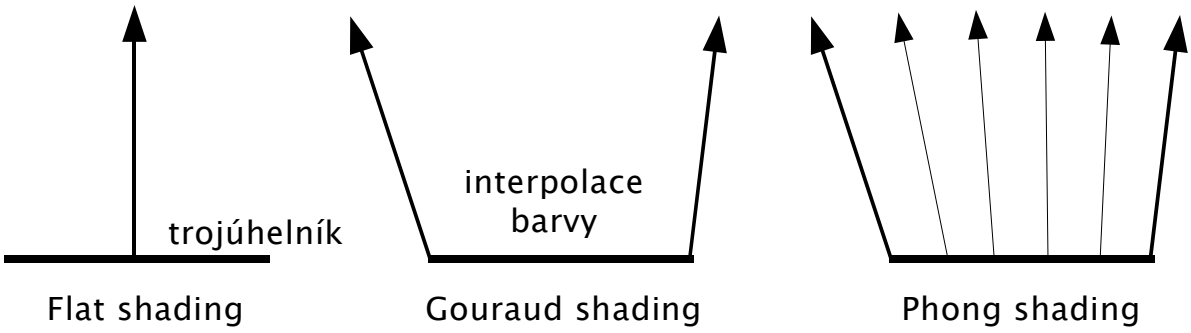
Máme-li povrchový model 3D scény připraven k zobrazení, musí nejprve projít geometrickým zpracováním. Tam se pracuje s celými trojúhelníky, resp. jejich vrcholy. Souřadnice vrcholů se transformují ze světových souřadnic (ve kterých je má uložena aplikace) nebo z lokálních relativních souřadnic (viz Hierarchické transformace) do takové souřadné soustavy, ve které se s nimi bude pohodlně pracovat. To bývá obvykle soustava spojená s pozorovatelem – směr pohledu leží na ose „z". Souřadnice „x" a „y" se ještě transformují (škálují) tak, aby kresba vyplnila požadovaný výřez na obrazovce a pak se již dvojice [x, y] dají použít přímo k vykreslení a zbylá složka „z" poslouží při výpočtu viditelnosti jako vzdálenost od pozorovatele (resp. vzdálenost od přední ořezávací roviny).



4.1 Stínování

Nezávisle na tom se může uplatnit výpočet osvětlení: podle zadaného zdroje světla se pro všechny plošky spočítají vektory „R", „L" a „V" (jsou-li zdroj i pozorovatel ve značné vzdálenosti, může se variabilita vektorů „L" a „V" zanedbat). V praxi se používá jedna ze tří metod interpolace:

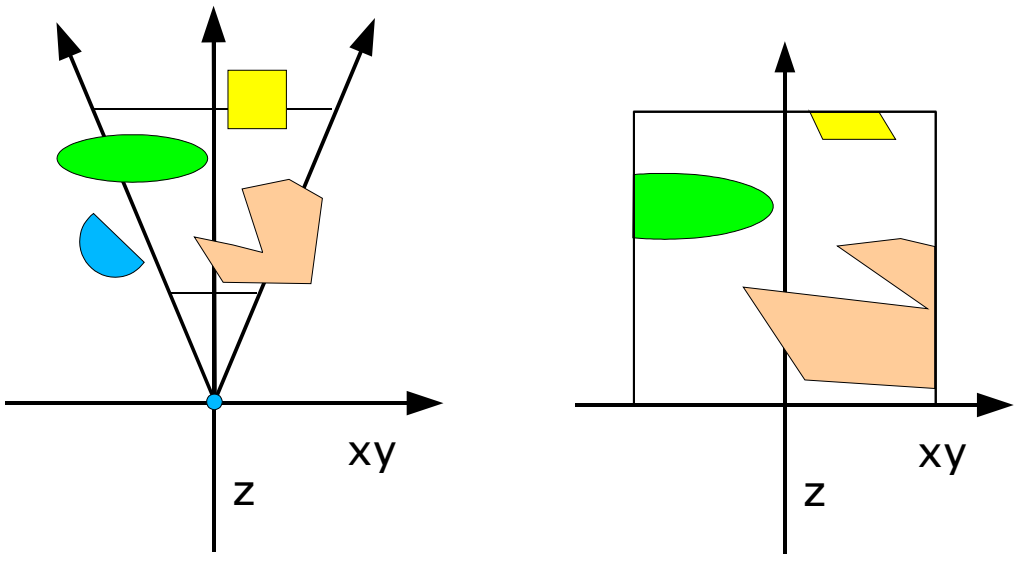
1. **Konstantní stínování** („Flat shading") je nejjednodušší a nejrychlejší, bohužel dává pro aproximace hladkých těles nejhorší výsledky. Osvětlení se spočítá jednou pro každý trojúhelník a celá plocha se vybarví daným odstínem
2. **Gouraudova interpolace barvy** („Gouraud shading") odstraňuje nepříjemně ostré přechody mezi jednotlivými ploškami, hodí se proto na objekty, které jsou aproximacemi hladkých těles. Osvětlení se počítá v každém vrcholu mnohostěnu a uvnitř trojúhelníka se odstín dopočítává lineární interpolací. Takovou interpolaci zvládnou běžné grafické akcelerátory. Při použití ostrých odlesků (velmi lesklé materiály) nedává ani interpolace barvy uspokojivé výsledky
3. **Phongova interpolace normály** („Phong shading") je nejdokonalejší, ale také nejpomalejší metoda: do každého kresleného pixelu se interpoluje normálový vektor (z daných „normál" ve vrcholech mnohostěnu). V každém pixelu se potom počítá osvětlení podle požadovaného modelu. Protože postup předpokládá schopnost hardware vykonávat v jednotlivých pixelech netriviální operace, je Phongova interpolace dostupná jen u nejmodernějších GPU (a jen pro omezenou množinu světelných zdrojů – viz další oddíly)



4.2 Projekce a ořezávání

Poslední fáze geometrického zpracování začíná výpočtem skutečné polohy objektů (vrcholů) v rovině průmětny. Pro rovnoběžnou projekci je to triviální operace, perspektivu vyřeší některá z transformačních matic uvedených dříve (převod komolého zorného jehlanu do viditelného kvádru/krychle – anglicky se mu říká „**view frustum**").

Aby se následující fáze zpracování („rasterizace") nezatěžovaly zbytečně daty, která nemohou podstatně přispět k výslednému obrázku (a aby se při promítání nemuselo počítat se zvláštními případy), aplikuje se na všechna grafická data algoritmus „**ořezávání**" („clipping"). To znamená, že jsou ze zpracování okamžitě odstraněny všechny objekty ležící celé mimo zorný jehlan (frustum) a objekty jen částečně zasahující do zorného jehlanu jsou „oříznuty". Protože se jedná o body, úsečky nebo trojúhelníky, není výpočet příliš složitý. Grafický akcelerátor ho samozřejmě umí provádět velkou rychlostí.



4.3 Hierarchické transformace

Ještě se vrátíme k první fázi geometrického zpracování 3D dat: reprezentace virtuálního světa obsahuje nezřídka **hierarchické principy** – scéna se skládá z objektů (často jsou objekty *instancemi* vytvořenými podle daných vzorů), samotné *objekty* se skládají z komponent, ty zase z menších dílů, atd. atd. Z mnoha praktických důvodů je vhodné, aby se v takovém modelu světa používaly **relativní transformační matice**.

Příklad: **kolečko** dětského kočárku je pevná součástka, která se však může otáčet kolem osy definované na podvozku. Pohybuje-li se **kočárek** (třeba rovnoměrným posuvným pohybem vpřed), u kolečka se skládá rotační pohyb s tímto celkovým posunem. Když jede kočárek **výtahem** a náhodou v něm popojede, u koleček se již skládají tři dílčí pohyby: vertikální posun výtahu, horizontální posun kočárku a rotační pohyb kolečka (ještě komplikovanější příklad mne napadl, když jsem uvažoval, že po takovém kolečku navíc leze *brouček* – to už nechám laskavě na čtenáři...).

Vtip hierarchických transformací je založen na tom, že žádný objekt nemá transformační matici ze své souřadné soustavy („local coordinate system") do světového systému souřadnic („world coordinate system"), ale pouze **relativní matici transformace** mezi svou lokální soustavou a soustavou spojenou s bezprostředním „nadřízeným". Tj. kolečko se otáčí vůči kočárku, kočárek se posunuje vůči podložce, podlaha výtahu se posunuje vzhledem k povrchu Země, atp. To však znamená, že při výpočtu skutečné polohy jakéhokoli objektu ve scéně potřebujeme počítat součin všech matic na cestě od objektu až k nejvyššímu uzlu hierarchie (ten používá světové souřadnice).

Zde se mohou uplatnit nejmodernější grafické akcelerátory se schopností vnějšího programování. Programovat se dá zpracování jednotlivých vrcholů (viz „vertex-shaders") a pokud je na grafickém čipu k dispozici dostatek místa na dílčí transformační matice, může se program aplikace značně zpřehlednit a především se usnadní práce hlavnímu procesoru (CPU). Je dobré si uvědomit, že všechny dříve citované pohyby kolečka kočárku, kočárku, výtahu, apod. se potom realizují pouhou změnou několika transformačních matic, největší objem dat (databáze scény = souřadnice vrcholů) zůstane konstantní!

5 Výpočet viditelnosti (Z-buffer)

Součástí poslední fáze grafického zpracování jsou algoritmy určující přesný vzhled výsledku – jednou z důležitých komponent je metoda výpočtu viditelnosti. V grafickém hardware se už dlouhá léta používá výhradně jeden algoritmus: **Z-buffer**. Idea je jednoduchá – zobrazované grafické objekty (čáry, trojúhelníky, apod.) se rozdělí na jednotlivé body (pixely) a pro každý pixel se jeho viditelnost určí samostatně. Protože se jedná o algoritmus typu „hrubá síla", je výpočet podpořen dvojrozměrným polem velikosti obrazovky. Pro každý pixel obrazovky si pamatujeme vzdálenost toho bodu 3D scény, který je v tomto pixelu zobrazen (tj. byl */zatím/* nejbližší ze všech). Dosud nepokreslené části obrazovky mají v poli uloženou nějakou velkou hodnotu („nekonečno"), touto hodnotou se pole na začátku musí inicializovat.

Protože se pro reprezentaci vzdálenosti bodu od pozorovatele používá složka souřadnic „z", dostala celá metoda jméno „Z-buffer". Vzdálenost se v bufferu ukládá nejčastěji jako 16- nebo 24-bitové číslo v pevné řádové čárce. Spotřeba paměti na Z-buffer (několik megabytů pro dnešní typická grafické rozlišení 1024×768 až 1600×1200) už není kritická, grafické akcelerátory běžně disponují řádově větším objemem paměti. Zpracování (testování) jednotlivých pixelů je sice časově náročné, protože všech pixelů je ohromné množství (třeba i 10^8 na snímek), pomůže zde však masivní paralelismus – jednotlivé HW obvody mohou pracovat nezávisle a na čip jich podle potřeby umístíme několik jednotek i desítek.

Starší implementace Z-bufferu byly přímo integrovány do „rasterizátoru" (obvod realizující rozklad trojúhelníka na jednotlivé pixely), dnes se kvůli větší flexibilitě „Z-test" zařazuje mezi ostatní kroky do zobrazovacího řetězce (další podrobnosti budou v oddílu „Fragment-shaders").

Velkými výhodami Z-bufferu jsou zejména: možnost vykreslovat objekty v **libovolném pořadí** (bez jakéhokoli prostorového třídění – výjimka viz Průhlednost) a **korektní kresba** většiny problematických situací, jako jsou protínající-se stěny, cyklické zákryty, apod. Naopak je třeba si dát pozor na omezenou přesnost výpočtu Z-testu – není vhodné umisťovat tenké předměty

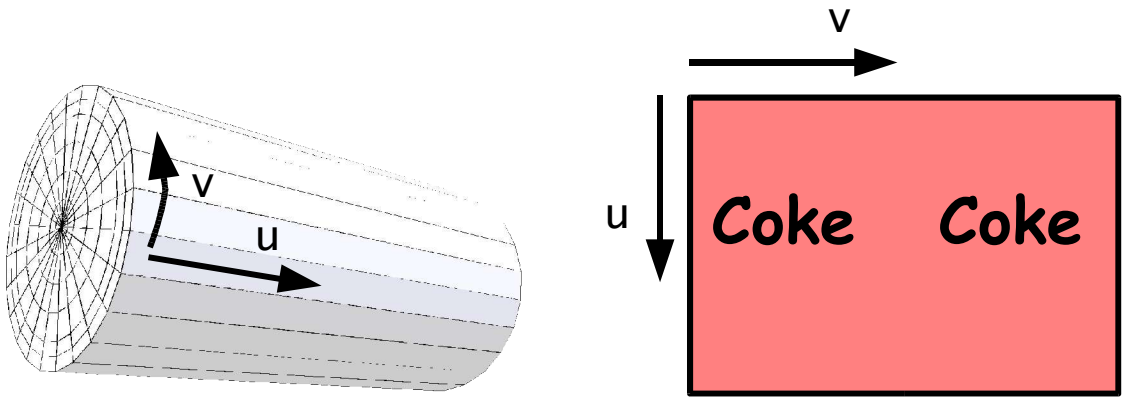
těsně k sobě (papír položený na stole), atd. Pokud bychom toto pravidlo porušili, objeví se při animaci náhodné „problikávání" nepravidelných částí jednoho z objektů („**Z-fighting**").

6 Textury

Vzhled nakreslených 3D objektů by bez textur byl velmi chudý (připomeňme, že zatím umíme počítat viditelnost ve scéně a víme o možnosti použít stínování – to ovšem umí modifikovat pouze odstín barvy předmětu) – povrch objektů by byl velmi hladký, nerealistický. Bez nadsázky lze říci, že techniky **mapování textur** mají největší zásluhu na tom, jak pěkně dnes počítačem generované obrázky vypadají (i když ještě dlouho zřejmě nebudou nerozeznatelné od přirozených snímků).

Definovat laikovi pojem textury není úplně jednoduché: musíme se spokojit s obecným vyjádřením, že „textura je technika modifikující pixel po pixelu vzhled kresleného povrchu". Nejčastěji se textura aplikuje jako **barevná** „tapeta", kterou lepíme na povrch tělesa. Ale mohou být i textury ovlivňující optické vlastnosti materiálu (**odrazivost**), zavádějící „hrbolatost" (populární „**bump-texture**") nebo nahrazující některý z obtížných algoritmických postupů (např. **Phongovo stínování** nebo **zrcadlový odraz** na hladkém povrchu tělesa).

Příklad: chceme-li nakreslit plechovku od Coca-coly, bylo by příliš pracné snažit se reprezentovat různé nápisy nebo loga algoritmicky. Místo toho prostě vezmeme skutečnou plechovku, její plášť rozvineme, zdigitalizujeme (třeba stolním scennerem) a získanou bitmapou potáhneme plášť válce (že je skutečný tvar pláště plechovky trochu jiný, nás v prvním přiblížení nemusí trápit). Data posílaná do grafického akcelerátoru se budou skládat z „**geometrie**" (reprezentace tvaru plechovky) plus „**textury**" (bitmapový obrázek).



Jak umí grafická karta texturu zobrazit? K jednotlivým vrcholům mnohostěnu musíme přiřadit tzv. **texturové souřadnice**: jsou to nejčastěji 2D souřadnice (budeme je označovat [u, v]) vztahující se k pixelům bitmapové textury. Těmto „pixelům" se obyčejně říká „**texely**", aby se nepletly se skutečnými pixely na obrazovce. Rasterizátor v GPU už bude umět texturu procházet a jednotlivé pixely na povrchu tělesa správně vybarvit.

Pokud vás napadlo, zda lze texturu kombinovat se stínováním, případně s přirozenou barvou povrchu tělesa, odpověď zní „ano". Plechovka od Coca-coly tedy bude nejen opatřena správnými nápisy, ale bude se i přirozeně lesknout na světle.

Dokonalejší grafické karty umějí kombinovat v jednom trojúhelníku několik textur (když ještě byl grafický řetězec pevně daný a nedal se konfigurovat, podle výrobce a typu GPU se dalo najednou použít až 4-8 textur). Jednotlivé výsledky se prostě slučují lineární kombinací („blending") nebo násobením. Dnešní programovatelné akcelerátory si může vývojář přizpůsobit dle potřeby, prostředky jsou ovšem někdy značně omezené (viz „Fragment-shaders").

7 Zpracování pixelů

Tento oddíl obsahuje několik poznámek o výpočtech spojených se zpracováním jednotlivých pixelů. Jak už bylo napsáno výše, rastrovací jednotka musí umět rozložit grafické primitivum (trojúhelník) na jednotlivé pixely. Takový rozklad se programově obvykle realizuje tzv. **řádkovým rozkladem** („scanline", „scan conversion") – řádek po řádku se určuje, jak dlouhý vodorovný úsek se má nakreslit. Hardwarové implementace jsou postavené na podobných principech: konstruktéři se snaží redukovat potřebu náročných operací (dělení, násobení) a převádět je na sčítání. Většina výpočtů se může uskutečňovat v pevné řádové čárce..

Protože při rasterizaci trojúhelníka je potřeba znát v každém pixelu další údaje (hloubku „z", texturovací souřadnice pro každou aktivní texturu, Gouraudovsky interpolovanou barvu, apod.), implementuje se v GPU řada interpolátorů. Snahou je, aby se maximální počet interpolací počítal lineárně.

Bohužel to není možné při interpolaci texturovacích souřadnic (ani při spojitém stínování, tam však nepřesnosti způsobené lineární interpolací nejsou tolik zřetelné). Postupem času bylo zavedeno několik interpolačních technik, které jsou korektní i v perspektivní projekci, jednou z nich je tzv. „**hyperbolická interpolace**"[4]. Na řádce sousedních pixelů se lineárně interpolují tři veličiny „**u/w**", „**v/w**" a „**1/w**". Jejich vydělením – pro první složku „(u/w) / (1/w)" – pak dostaneme požadované hodnoty [u, v]. Poznámka: při simultánní interpolaci několika textur na jednom trojúhelníku se společný člen „1/w" počítá pouze jednou.

Zdá se, že ve složitějším grafickém kontextu může být zpracování jednotlivých pixelů („**pixel fill-rate**") tím nejslabším článkem zobrazovacího řetězce. Opravdu: existují aplikace, ve kterých je tomu tak, ale naopak i jiné aplikace, kde je kritickým místem zpracování vrcholů („**transform-limited applications**") – scény s extrémně velkým počtem malých objektů. A samozřejmě může mít aplikace tak složitou vnitřní logiku, že vůbec nedokáže vytížit daný akcelerátor (masivní fyzikální simulace, částicové systémy, apod.). Ke které skupině váš program patří, zjistíte nejlépe jeho profilováním. Vývojové nástroje firem NVidia i ATI by na takové výzkumy měly být připraveny.

8 Průhlednost

První nadstandardní technika, o které si řekneme, ale kterou nemusí využívat každý grafický program, je **poloprůhlednost**. Mnoho (zejména geometrických) efektů se implementuje velmi snadno a elegantně, máme-li k dispozici poloprůhledné trojúhelníky. Dokonce je možné mapovat poloprůhledné textury – v některých místech jsou pak objekty (plochy) průhledné, jinde ne.

Průhlednost se v počítačové grafice obyčejně definuje pomocí parametru „α" („**α-channel**", „α-kanál"). Je to desetinné číslo mezi „0" a „1" udávající míru **neprůhlednosti** objektu – „0" znamená úplně průhledný, „1" zcela neprůhledný. Často se přidává jako čtvrtá složka k barevné trojici [R, G, B], pak mluvíme o **poloprůhledné barvě** [R, G, B, α].

Poloprůhlednými polygony nebo texturami lze mj. implementovat následující jevy: okenní tabulky, koruny stromů nebo keřů v dálce, plameny, kouř, výbuchy, apod.

Grafické akcelerátory umějí s poloprůhledností pracovat už delší dobu, je však třeba upozornit na omezení, které průhlednost obyčejně přináší: pracujeme-li s poloprůhlednými objekty v 3D scéně a používáme-li současně Z-buffer pro určení viditelnosti, musíme poloprůhledné polygony posílat do GPU setříděné **odzadu dopředu**. Jinak může vzniknout nekorektní výstup.

4 Heckbert, Moreton 1991, Blinn 1992

9 Šablony (stencils)

Inspirováni Z-bufferem vymysleli vývojáři grafických čipů další silný nástroj: **buffer šablon** („stencil buffer"). Jako se do Z-bufferu ukládá informace o hloubce nakresleného 3D bodu, stencil buffer slouží k uložení libovolné informace (atributu) daného pixelu. Typicky je programátorovi k dispozici (pro každý pixel) několik bitů v bufferu šablony. Grafický zobrazovací řetězec je navíc modifikován tak, aby se do šablony dalo **zapisovat** (dokonce pomocí libovolné booleovské operace) a naopak – před vlastním vykreslením (nebo Z-testem) je GPU připraven šablonu **otestovat** a podle výsledku kreslení provést či nikoli..

Tak dokážeme elegantně vykreslovat pouze do některých částí obrazovky (jako náhradu za poloprůhlednost tam, kde by nebyla dostatečně efektivní), selektivně vyřadit Z-testování, počítat vržené stíny, atd. Nejvíce triků umožňují šablony spolu s víceprůchodovými zobrazovacími algoritmy (viz oddíl Více průchodů).

10 Programování GPU

V posledních třech letech zažily technologie 3D grafického zobrazení revoluci (srovnatelnou snad jen s první HW implementací Z-bufferu). Vývojáři aplikací se již nemusí spoléhat na rigidní zobrazovací řetězec, který pro ně vymysleli konstruktéři grafického čipu (naneštěstí se ten řetězec lišil model od modelu a každý výrobce měl rovněž trochu jiný přístup). Soudobé 3D akcelerátory lze **programovat**, tj. za běhu konfigurovat postup jejich výpočtu – to se týká všech podstatných částí zobrazovacího řetězce. Mezinárodní (mezi-firemní) společnosti definují programovací prostředí pro GPU tak, aby bylo maximálně transparentní a umožňovalo psát snadno přenositelné programy (které poběží i na čipech v době vývoje aplikace neznámých). V softwarovám průmyslu je to běžná věc, zde se však jedná o jednu z prvních významných a úspěšných inter-disciplinárních aktivit (spolupráce konstruktérů grafického HW a vývojářů SW).

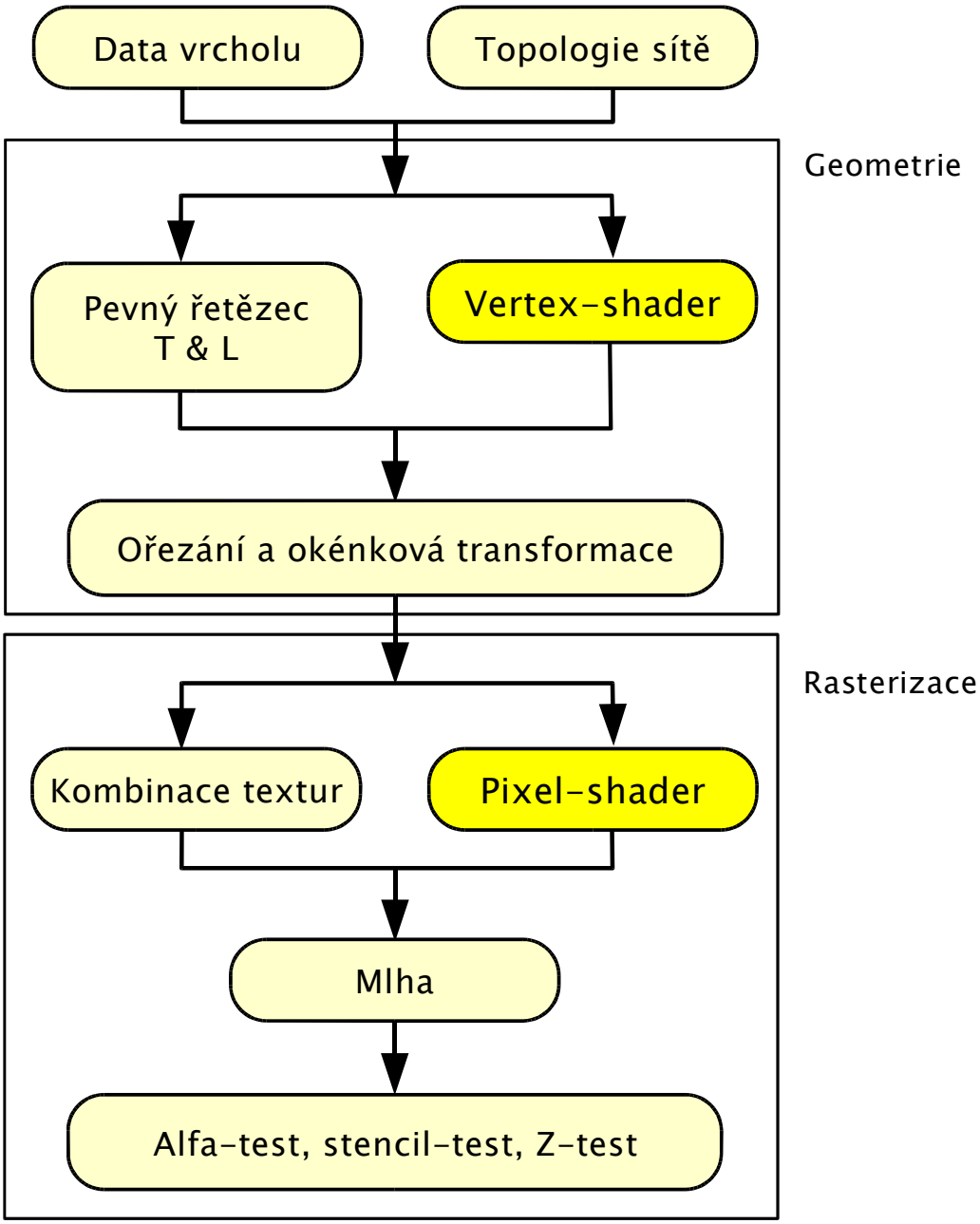
Průkopníky těchto nových technologií jsou zejména firmy **NVidia**, **ATI** (výrobci GPU), dále **Microsoft** jako garant **Direct3D** API a konsorcium **OpenGL** (starší konkurenční 3D API dostupné i na otevřenějších platformách – UNIX, Linux, ..).

Programovatelnost GPU se týká dvou klíčových fází zpracování grafických dat:

1. zpracování jednoho vrcholu mnohostěnu (data přidružená k vrcholu: souřadnice, normálový vektor, barva, texturové souřadnice, navíc může vývojář k vrcholu přidat několik dalších hodnot dle své potřeby)
2. zpracování viditelných částí trojúhelníka – jednotlivých pixelů (data spojovaná s pixelem: texturové souřadnice, souřadnice „z", interpolované barvy difuzního a leslého světla, interpolovaná průhlednost a normála, ..)

Program, prováděný v prvním případě, se nazývá „**vertex shader**" a jeho možnosti jsou dány podobou se strojovými jazyky moderních univerzálních CPU. Je možné provádět podmíněné skoky, volat podprogramy, apod., existují zde však omezení co do počtu instukcí nebo dat, nad kterými lze pracovat.

Program, prováděný při zpracování každého pixelu, se jmenuje „**pixel-shader**" (nebo „**fragment-shader**"). Zde jsou možnosti ještě více omezeny díky tomu, že je běh kódu příliš těsně svázán s pixelovými interpolátory (neefektivita v tomto místě by mohla velmi dramaticky snížit aktuální „fill-rate" grafického procesoru). I když jsou některé povolené instrukce dost komplexní, programátor se musí omezit na jednotky (maximálně několik málo desítek) takových příkazů. Spíš než jako univerzální algoritmus lze pixel-shader chápat jako skript pro konfiguraci několika přesně vymezených etap rasterizace (ale v budoucnosti se tato filozofie může jistě změnit).



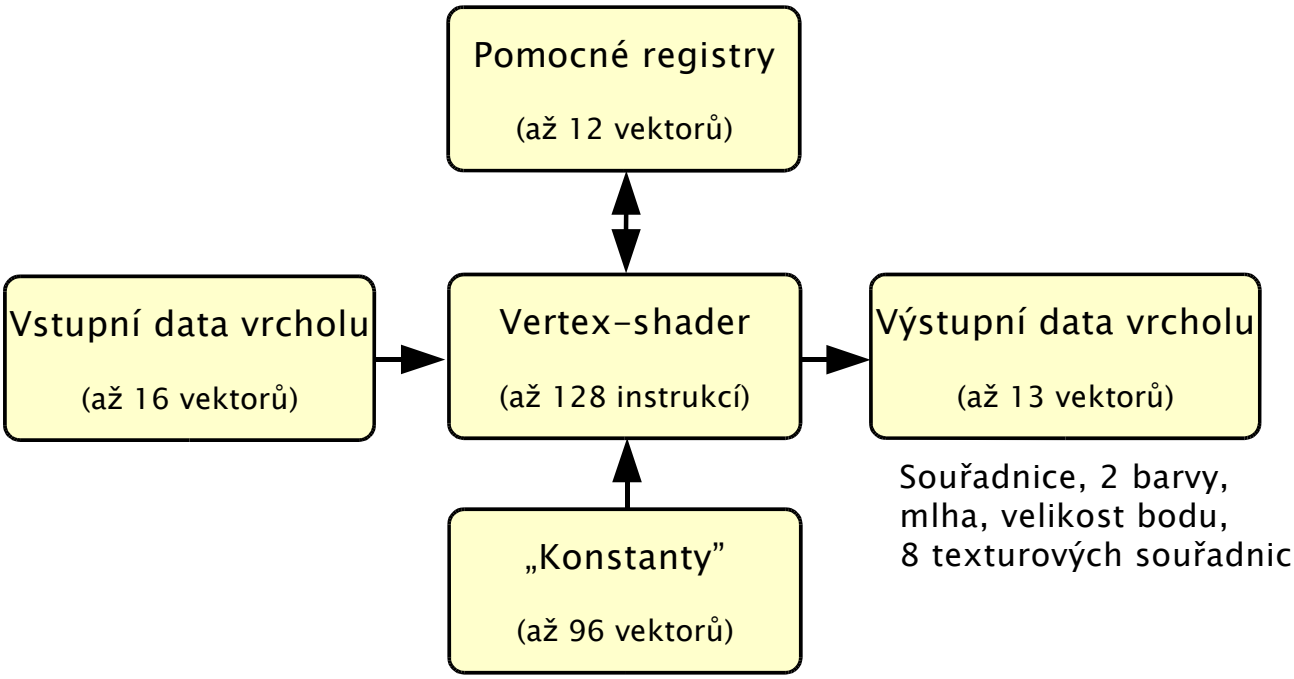
10.1 Vertex-shaders

Program se vyvolává na každý vrchol, který vstoupí do řetězce grafického zpracování. Výsledkem jeho práce je balíček dat, která pokračují v cestě řetězcem, některé hodnoty jsou automaticky interpolovány do všech pixelů, jiné mají předem definovaný význam. Není možné přidat nový vrchol nebo naopak existující vrchol zrušit!

Programovací jazyk: v současnosti je k dispozici několik vývojových prostředí včetně programovacích jazyků, které jsou pro člověka dostatečně srozumitelné. Jako datové typy se používají 4-složkové vektory 32-bitových čísel v plovoucí desetinné čárce (ekvivalent v jazyce C: „float[4]”). Lze adresovat jednotlivé složky, maticové operace pracují s posloupnostmi třech nebo čtyřech po sobě následujících vektorů (matice 4×3 či 4×4). Microsoft používá v Direct3D 9 pro oba

druhy programů tzv. „**HLSL**" („High-level Shading Language"). Podrobnější popis jazyků nemůže být do této přednášky zahrnut, zájemci nechť vyhledají literaturu na Internetu.

Prostředí, ve kterém vertex-shader pracuje (nejstarší specifikace VS 1.1):



Poměrně svazujícím omezením je počet „**konstant**" omezený na 96 čtyřsložkových vektorů. Po odečtení nutné režie (konstanty 0, 1, -1, projekční transformační matice, apod.) zbývá místo jen asi na 20 „rigid-body" transformací, což je velmi málo – nestačí to například ani na transformace hierarchického modelu jednoho člověka (systémem „artikulované hierarchie")..

Příklad velmi jednoduchého vertex-shaderu – každý vrchol je transformován projekční transformační maticí a navíc je na něj aplikována konstantní difusní barva. **Projekční matice** je uložena v konstantách „c0" až „c3", **difusní barva** v konstantě „c4" (o naplnění konstant se musí předem postarat aplikační program) :

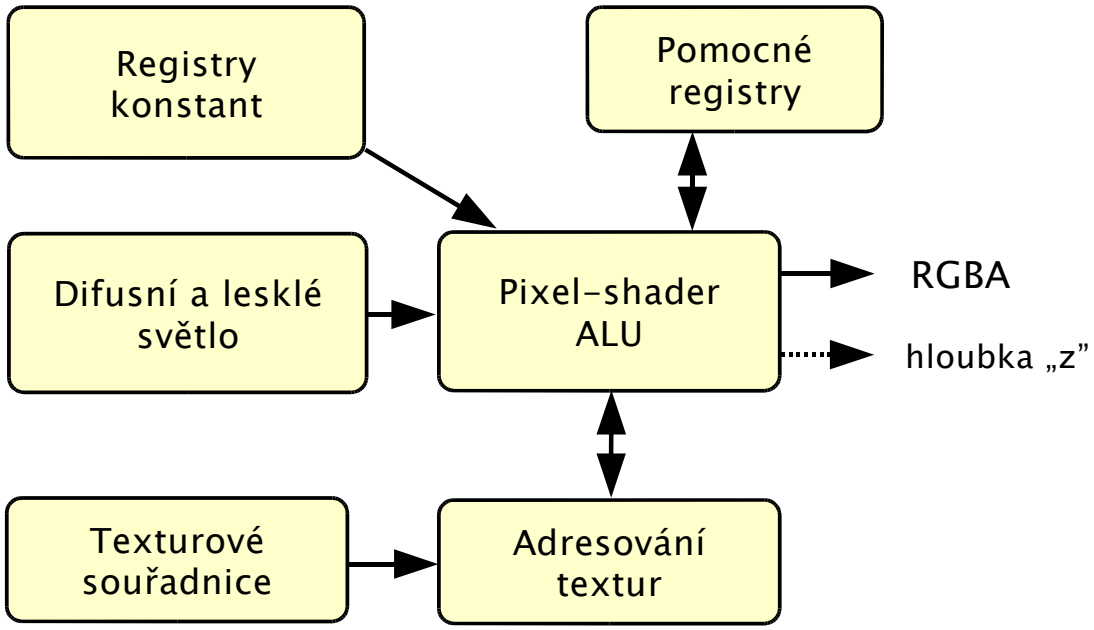
```
vs_1_1              // verze vertex-shaderu
dcl_position v0     // deklaruji, že souřadnice vrcholu mají být v registru v0
m4x4 oPos, v0, c0   // aplikuji projekční transformaci (výstup jde rovnou do oPos)
mov oD0, c4         // definuji konstantní difusní barvu vrcholu
```

10.2 Pixel-shaders (fragment-shaders)

Program se vyvolává na každý rasterizovaný pixel. Pixel-shader může pracovat s několika texturami, dvěma barvami (které mu spočítal vertex-shader), mlhou a v některých verzích i s hloubkovou souřadnicí „z". Na výstupu je vždy výsledná **barva pixelu** (a někdy též přepočítaná **hloubka** „z"), pozici pixelu pochopitelně modifikovat nelze. Množina konstant a pomocných registrů je zatím malá, malý je rovněž počet sériově prováděných instrukcí (některé instrukce lze při šikovném uspořádání spouštět paralelně).

Prostředí, ve kterém obecný pixel-shader pracuje, je uvedeno v následujícím diagramu. V současnosti se pixel-shadery ještě bouřlivě vyvíjejí, proto zde nebudeme popisovat velké podrobnosti. V DirectX 9 jsou definovány celkem čtyři verze specifikace PS – „1.x", „2.0", „2.0 extended" a „3.0", ta nejnovější obsahuje dokonce řízení programu, podprogramy, přes 200

konstant typu „float”, „int” a „bool”, výběr dat z 2D i krychlových textur, „swizzling” – libovolné permutace zapisovaných a čtených vektorů, apod.:



Příklad jednoduchého pixel-shaderu, který mixuje barvy dvou vstupních textur podle koeficientu zadaného v difusní složce barvy (spočítal vertex-shader):

```
ps_1_1              // verze pixel-shaderu
tex t0              // výběr barvy z textury podle texturovacího registru t0
tex t1              // to samé pro texturovací registr t1
mov r1, t1          // texturu t1 přesunu do výstupního registru r1
lrp r0, v0, t0, r1 // lineární interpolace mezi t0 a r1 podle koeficientu v0
```

11 Pokročilé techniky

Tento oddíl obsahuje ukázky některých zajímavých pokročilejších technik a algoritmů, které se používají v hardwarově podporované 3D grafice. Popisované efekty nemusí být založeny na programovatelných GPU (i když někdy je to podmínkou).

11.1 Anti-aliasing

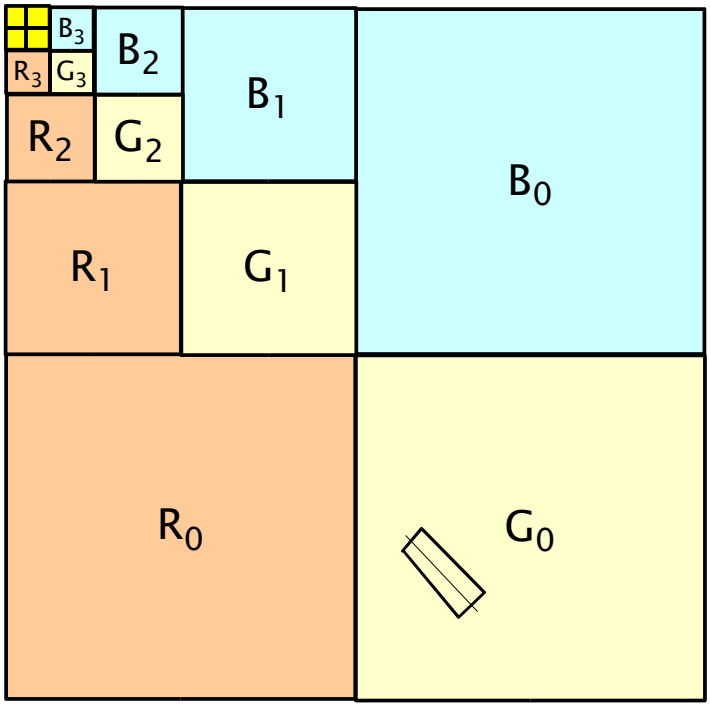
Při každém **pravidelném vzorkování** funkce spojitého argumentu (funkcí je zde tzv. „obrazová funkce” – abstraktní dokonalý obraz – a pravidelné vzorkování je způsobeno úplně pravidelnou sítí pixelů) může vznikat jev, kterému se říká „**alias**”. V 3D grafice se alias projevuje zubatými obrysy hladkých předmětů nebo barevným „zrněním” na vzdálenějších texturách. O texturách bude pojednávat následující oddíl, zde si řekneme o potlačení aliasu obecně.

Pokud alias vzniká nedostatečným vzorkováním (a ne interferencí), lze ho potlačit metodou nazývanou „**převzorkování**” („supersampling”). V modernějších grafických kartách bývá supersampling implementován. Metoda „hrubé síly” spočívá v nakreslení dvakrát většího obrazu do paměti (z hlediska počtu pixelů to dá čtyřnásobné množství práce), kde se potom čtyři sousední pixely průměrují do jednoho pixelu na obrazovce. U některých akcelerátorů lze konfigurovat

koeficient zvětšení, případně mají dokonce zabudovaný nějaký lepší (úspornější) vzorkovací algoritmus. Detaily o metodách vzorkování najde čtenář v [1] nebo [2].

11.2 Mip-mapping a neizotropické filtrování

U textur by dalo zbytečně mnoho práce opakované průměrování sousedních texelů. Již klasická technika, která využívá dopředu připravené zmenšené kopie textury, se jmenuje „**MIP-mapping**" (z latinského „multum in parvo"). Jsou-li data tříkanálová (např. RGB bez průhlednosti), lze předem připravenou MIP-mapu uložit elegantně v jediném poli:



Algoritmus použití MIP-mapy: při každém mapování se spočítá faktor zmenšení textury, najde se nejbližší předpřipravená úroveň a daný pixel se obarví spočítaným texelem nebo interpolací mezi čtyřmi sousedními texely. Důmyslnější metody umějí využívat dvou sousedních úrovní MIP-mapy (tj. interpolovat mezi pěti texely), ale nevím, zda se to v hardwaru vyplatí implementovat (srovn. s příštím odstavcem).

Použití MIP-mapy vede sice ke zlepšení vzhledu výsledného obrazu – interference textur přestane být rušivá – zaplatíme však za to menší ostrostí zobrazené textury. To je přímým důsledkem rovnoměrného zmenšování textury v obou směrech (vodorovném i svislém). Existuje několik technik, které řeší nerovnoměrné zmenšování (např. „**ripmap**" od Hewlett-Packard), mezi nimi je asi nejvýznamější „**neizotropické filtrování**". Při mapování textury se zjistí, jakou plochu textury pokryje zpětně promítnutý pixel (má tvar deformovaného čtverce). Vybere se nejvíce protažená osa a podél ní se z MIP-mapy vybere a zprůměruje několik vzorků (2 až 8, viz předchozí obrázek). Takový postup zaručí, že ani plochy pozorované pod velmi ostrým úhlem nebudou zbytečně rozmazané.

11.3 Více průchodů

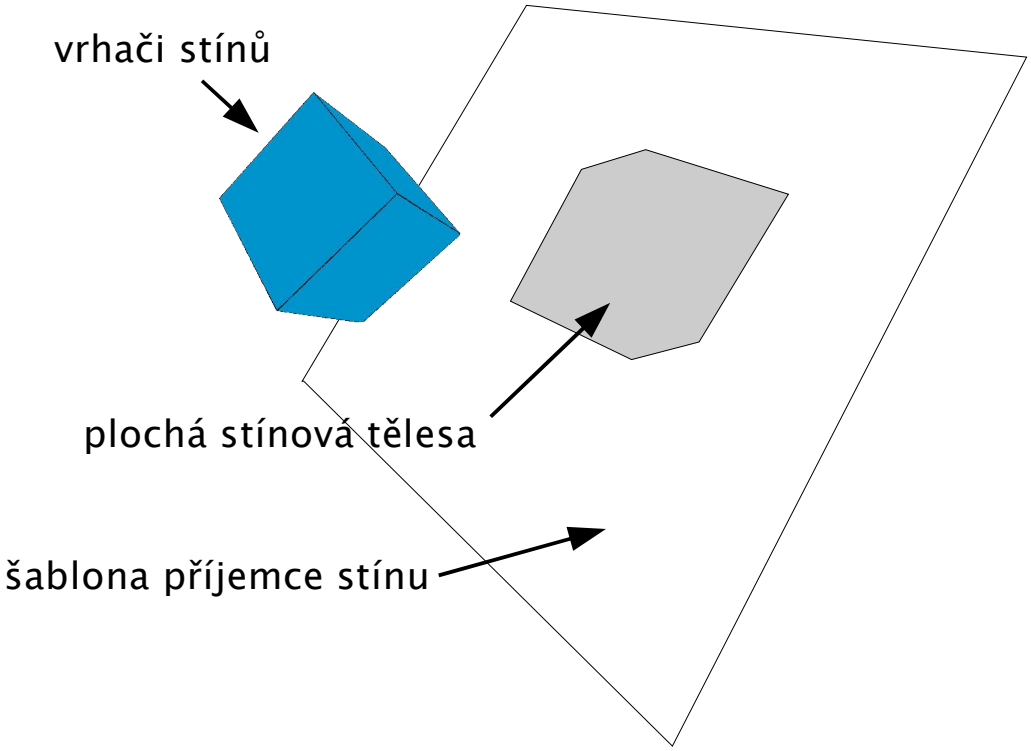
I když dosud probírané techniky (Z-buffer, poloprůhlednost, stencil-buffer, programovatelné shadery) jsou velkým přínosem k realistickému a efektivnímu zobrazení 3D grafiky, některé situace nelze jednoduše vyřešit obyčejným jednoprůchodovým zobrazením. Zejména pracujeme-li s šablonou (stencil-buffer), můžeme považovat za užitečné procházet zobrazovaná data několikrát – např. jedním průchodem se připraví šablona a spočítá mapa hloubky (Z-buffer), v druhém průchodu se již některé části scény vykreslují jen tam, kde je nastaven příslušný bit šablony. Obecně: víceprůchodové algoritmy využívají toho, že lze v paměti GPU uchovat color-buffer (obyčejná video-paměť) stencil-buffer a Z-buffer mezi jednotlivými fázemi. Topologie scény se mezi průchody nemění, geometrie a transformační matice někdy ano (například pro rozmazání pohybem nebo při výpočtu měkkých odrazů).

Klasickým příkladem použití šablony spolu s víceprůchodovým zobrazením, je výpočet vržených stínů uvedený v dalším oddíle.

11.4 Výpočet vržených stínů

Cílem je vytvořit obraz stínů, které vrhá dostatečně ostrý světelný zdroj (např. slunce v exteriéru). Jednodušší přístup uvažuje bodový zdroj světla, nejradši ve velké vzdálenosti. Uvedeme zde algoritmus, který umí vrhat stíny na **jedinou plochu** (například podlahu), dokonalejší postupy schopné stínit libovolná tělesa jsou popsány v literatuře.

Princip: pomocí **projekční matice** (pro rovnoběžnou projekci, leží-li zdroj dostatečně daleko, nebo pro perspektivu, je-li naopak blízko) se promítnou všechna tělesa, která mohou vrhat stín, do roviny „příjemce" stínu. Tyto průměty se zobrazí jako regulární tělesa v 3D světě, a tak vzniknou ostře ohraničené tmavé „stíny".



Problémy budeme řešit pomocí **stencil-bufferu** a jiných technik: aby stíny nepřesahovaly plochu, na kterou jsou vrhány, spočítáme v prvním průchodu do šablony oblast, kde je na obrazovce „příjemce" vidět (a stínové průměty omezíme jen na tuto šablonu). Potíže s přesností Z-testu se objevují vždy, když chceme umístit plochý objekt (stín) na povrch nějaké plochy (příjemce). Zde existuje několik řešení: stínové průměty posunout maličko *nad* plochu příjemce nebo využít opět šablonu a druhý průchod. Poslední nepříjemnost nás čeká tehdy, když bychom chtěli stínové průměty aplikovat jako poloprůhledné – například chceme, aby pod stínem prosvítala původní barva/textura plochy. Pouhé mechanické kreslení všech stínových průmětů přes sebe nevede k dobrému výsledku, protože některé části stínu budou započítány několikanásobně. Opět pomůže chytrá aplikace stencil-bufferu.

Shrnutí algoritmu:

1. celou 3D scénu vykreslíme v **běžném promítání**, přitom příjemce nastavuje stencil-bit a všechny ostatní plochy ho nulují (tak máme na konci první fáze k dispozici korektní masku ukazující, kde byl příjemce skutečně zobrazen)
2. **vypneme Z-testování** a všechny potenciální vrhače stínu vykreslíme pomocí speciálního promítání **do roviny příjemce**. Použijeme poloprůhledný tmavý materiál, který (je-li nanesen na plochu příjemce) vytvoří iluzi stínu. Aby nedocházelo k opakovanému stínění, kreslí se jen tam, kde je „stencil" nastaven a nakreslení viditelného stínového pixelu okamžitě daný bit vynuluje.

11.5 Bump-mapping

Poslední dva oddíly pojednávají o zvláštním použití texturovacích technik. „**Bump-map**" je populární metoda umožňující navodit **dojem nerovného povrchu** tam, kde si ve skutečnosti nemůžeme dovolit komplikovat geometrii 3D modelu. Trik je založen na modifikaci normálového vektoru – pozorovatel totiž maximum informace získává z osvětlovacích efektů (odlesků); pokud budou nakreslené odlesky vypadat tak, jako kdyby byl povrch hrbolatý, nikoho ani nenapadne uvažovat o tom, zda skutečně hrbolatý je!

Elegantní implementace bump-mappingu využívá zvláštní **textury** obsahující drobné změny normály (ve skutečnosti jsou v textuře uloženy obě parciální derivace „**h(u,v)**", což je funkce modelující zvrásnění simulovaného povrchu relativně k použité ideální plošce) a speciálně naprogramovatného **pixel-shaderu** (ten musí zajistit, že se v každém pixelu interpolovaná normála trochu „pokazí" danou texturou).

11.6 Environment-mapping

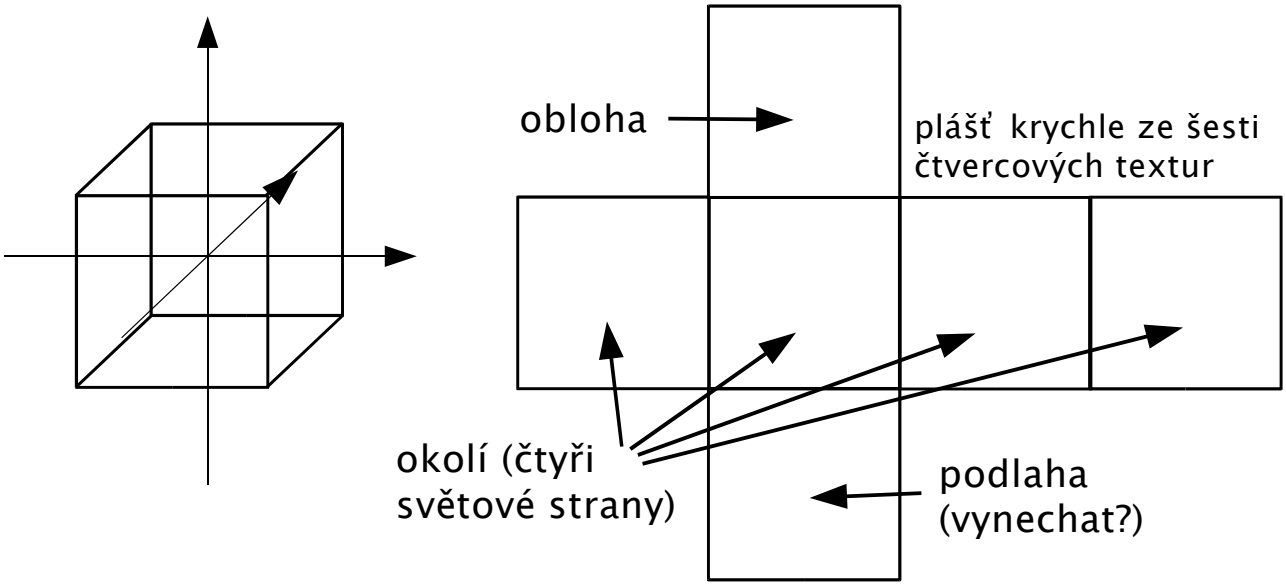
Často používaná technika dokáže napodobit odraz celé scény na lesklém povrchu nějakého malého tělesa (vánoční ozdoba), simuluje realistické měkké osvětlení (spočítané například radiační metodou), odrazy v zrcadlech nebo směrové Phongovo osvětlení. Příslušné algoritmy byly vyvinuty dávno před zavedením programovatelných GPU, plného uplatnění však dosahují až dnes (s pomoci vertex- a pixel- shaderů).

Základní myšlenka spočívá v tom, že se předem do nějaké textury připraví všesměrový pohled z jistého bodu ve scéně. Geometrie takového zobrazení se používají nejčastěji dvě: **krychlová** (textura obsahuje rozvinutý plášť krychle – šest čtverců) nebo **sférická** (povrch koule ve sférických souřadnicích). Proto se celé technice říká „mapování okolí".

„**Textura okolí**" může obsahovat buď kompletní **obraz zbytku scény** (pro odlesk scény na povrchu malého předmětu nebo v zrcadle) nebo třeba pouze předem spočítanou **mapu osvětlení** – závislost barvy a intenzity odlesku na normálovém vektoru.

Mapování takových textur je založeno na tom, že pixel-shader (nebo vertex-shader) dokáže indexovat texturu pomocí vhodně transformovaného normálového vektoru. Až budou v budoucnu

pixel-shadery méně omezené, jistě se najdou další možnosti mapování textur (třeba podle kombinace normály a nějakého explicitního datového pole).



12 Literatura

Nejlepším zdrojem informací jsou zahraniční publikace a články na Internetu, některé populárnější zdroje najdete i v českém jazyce:

1. Tomas Akenine-Möller, Eric Haines: *Real-time rendering*, 2nd edition, A K Peters, 2002, ISBN 1-56881-182-9, doplňky na http://www.realtimerendering.com/
2. J.D. Foley, A. van Dam, S.K. Feiner, J.H. Hughes: *Computer Graphics: Principles and Practice, Second Edition in C*, Addison-Wesley, 1996
3. Alan Watt, Mark Watt: *Advanced Animation and Rendering Techniques – Theory and Practice*, Addison-Wesley, 1992
4. Jiří Žára, Bedřich Beneš, Petr Felkel: *Moderní počítačová grafika*, Computer press, 1998, ISBN 80-7226-049-9
5. Peter Kovach: *Inside Direct3D*, Microsoft Press, 2000
6. Stránky pro vývojáře společnosti NVidia: http://developer.nvidia.com/
7. Stránky pro vývojáře společnosti ATI: http://www.ati.com/developer/
8. Stránky OpenGL konsorcia: http://www.opengl.org/
9. Stránky o DirectX společnosti Microsoft: http://msdn.microsoft.com/directx/
10. Stránky pro vývojáře her: http://www.gamasutra.com/