Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Metody stínování a stereoprojekce

Plzeň, 2007 Karel KLOUDA

Prohlašuji,
že jsem bakalářskou práci vypracoval samostatně
a výhradně s použitím citovaných pramenů.

V Plzni, dne 18. července 2007

Karel Klouda

Obsah

Abstract

Shading algorithms and stereoscopy

This work presents the readers with overview of most common shading algorithms for 3D graphics. Also included is a thorouhg introduction into color spaces and their application, and into stereoscopy as a means of inducing more realistic impression for the viewer.

The acompanying computer software demonstrates the use of the discussed shading algorithms, implemented in hardware, and the application of stereoscopy. The second provided program lets the user analyse the impact of a color space other than the most commonly used RGB on dithering methods. Programming language used is C# for the programs and HLSL for the shaders. The Microsoft .NET Runtime libraries (version 2.0), as well as the Microsoft DirectX runtime libraries (version 9.0) are required to run the programs.

1 Úvod

Počítačová grafika patří k těm oblastem výpočetní techniky, se kterými se setkáváme velmi často a možná si to ani neuvědomujeme. Od dob svých začátků prošla obrovským vývojem. Mnohé z někdejších špičkových technologií jsou nyní běžně používány, a mnohé již byly překonány. Běžně máme v počítačích grafické karty pro třídimenzionální grafiku, které se dokáží velmi rychle vypořádat s často používanými operacemi, výsledky výpočtů můžeme zobrazovat ve vysokém rozlišení pomocí digitálního projektoru na plátno nebo je vytisknout na barevné laserové tiskárně.

V průběhu vývoje počítačové grafiky bylo nutné vyřešit některé základní problémy. Bylo nutné vyřešit, jakým způsobem co nejlépe „uvnitř počítače“ reprezentovat reálný svět se všemi jeho fyzikálními aspekty. Zobrazení barev již bylo do jisté míry vyřešeno od dob barevné televize, tedy od 50. let 20. století. Bylo ale nutné najít způsob zobrazení barevných obrazů na zařízeních, která dokázala zobrazit jen některé barvy, například na kartě standardu VGA, která na ploše 640 x 480 pixelů dokázala zobrazit 16 pevně definovaných barev. Získané algoritmy se dodnes běžně používají při tisku obrazů na domácích tiskárnách.

Stejně tak nástup trojrozměrné grafiky s sebou přinesl problémy, které bylo nutno vyřešit. Stejně jako u dvourozměrné grafiky požadujeme, aby výsledný obraz vypadal co nejpřirozeněji a bylo v něm co nejméně rušivých elementů při, pokud možno, co nejnižší výpočetní složitosti použitých algoritmů.

Moderní počítačová grafika se samozřejmě stále vyvíjí. Na Západočeské univerzitě probíhá výzkum, mimo jiné, v oblasti stereoskopie a jejího praktického využití, především v oblasti televize – projekty 3DTV a MUTED.

Hlavním cílem mé bakalářské práce je demonstrace stínovacích metod pro trojrozměrné objekty pro aplikace stereoskopického zobrazení. Demonstrační program by měl být dostatečně jednoduchý a přehledný a měl by efektivně využívat možností moderní výpočetní techniky – stínovací algoritmy jsou tedy implementovány v hardwaru.

Jako další cíl jsem si vytýčil prozkoumat vliv barevných prostorů na metody pro redukci počtu použitých barev v oblasti 2D grafiky. Program by opět měl být myšlenkově přímočarý a zároveň výpočetně jednoduchý.

K oběma těmto cílům potřebujeme znát určitou teorii, která je velmi podobně rozebrána v kapitole 2, realizační část je pak včetně stručné programátorské dokumentace popsána v kapitole 3. V kapitole 4 zkoumám vliv použitého barevného prostoru na rozptylové metody pro 2D obrazy a také vliv počtu trojúhelníků, definujících povrch trojrozměrného tělesa na snímkovou frekvenci v aplikaci 3D grafiky.

Byl bych rád, kdyby výsledky mé práce, teoretické i praktické, pomohly studentům bakalářského studijního programu Informatika na naší univerzitě a obdobných studijních programů na jiných univerzitách k pochopení základních principů použití stereoskopie, metod stínování a barevných prostorů.

2 Teoretická část

2.1 Barevné systémy

Barevných systémů existuje nepřeberné množství. Můžeme provést jejich základní rozdělení na systémy vědecké (CIE), technické (RGB, CMYK, …) a uživatelsky orientované (HSV a jiné).

Z fyzikálního hlediska je světlo vlněním. Jedná se o speciální případ vlnění, které lze detekovat lidským okem. Lidské oko má receptory pro krátké, střední a dlouhé vlnové délky, které fungují jako receptory modré, zelené a červené barvy. To znamená, že pro popsání barevného vjemu potřebujeme tři parametry. Specifická metoda, která každé trojici hodnot přiřadí jednu určitou barvu, se nazývá barevný prostor. Barevných prostorů existuje více, jedním z nich je např. CIE_{XYZ}.

V technických barevných prostorech může být většina reálných barev vytvořena mísením základních barev. V nejčastěji používaném prostoru RGB je to červená (R – red), zelená (G – green) a modrá (B – blue), v prostoru CMY potom azurová (C – cyan), purpurová (M – magenta) a žlutá (Y – yellow).

Vidíme-li světlo o určité vlnové délce, získáme vjem určité barvy. Přirozené zdroje světla neobsahují jen jednu vlnovou délku, i když my je vnímáme jako určitou barvu. Mají-li dva světelné zdroje stejnou viditelnou barvu, získáme stejný barevný dojem

bez ohledu na to, z jakých barev jsou složena světla.

2.1.1 CIE

CIE, plným názvem *Commission internationale de l'éclairage*, česky Mezinárodní komise pro osvětlení, je mezinárodně uznávanou autoritou přes světlo, barvy a barevné systémy. Sídlí ve Vídni v Rakousku a od dob svého založení na počátku 20. století vydala již přes sto publikací a doporučení, které mají status technické normy. Mezi jinými také vlnové délky základních barev systému RGB – viz související odstavec.

Barevný prostor CIE 1931 (CIE_{XYZ})

CIE 1931 byl jeden z prvních matematicky definovaných barevných prostorů. Je založen na přímém měření lidského oka, a slouží jako základ pro ostatní barevné prostory.

Lidské oko má receptory pro tři základní barvy – červenou, zelenou a modrou. Pokud bychom chtěli vyjádřit všechny viditelné barvy, potřebovali bychom trojrozměrný obraz. Informaci o barvě ale můžeme rozdělit na dvě části – jas a *chromacitu* (tón). Například bílá, šedá a černá jsou vlastně jedna a ta samá barva, liší se jen úrovní jasu.

Prostor XYZ eliminuje problém záporných hodnot *trichromatického spektrálního činitele* (viz obr. 4), na druhou stranu v něm mohou vznikat „neskutečné" barvy se *sytostí* (viz str. 13) vyšší než 100%, které nedokážeme získat rozkladem bílého světla. Parametr Y zhruba odpovídá úrovni jasu, barevný tón udávají parametry *x* a *y*, které lze odvodit ze složek *tristimulu*, což je vlastně „výstup ze sítnice" v lidském oku:

$$x = \frac{X}{X+Y+Z} \tag{1}$$

$$y = \frac{Y}{X+Y+Z} \tag{2}$$

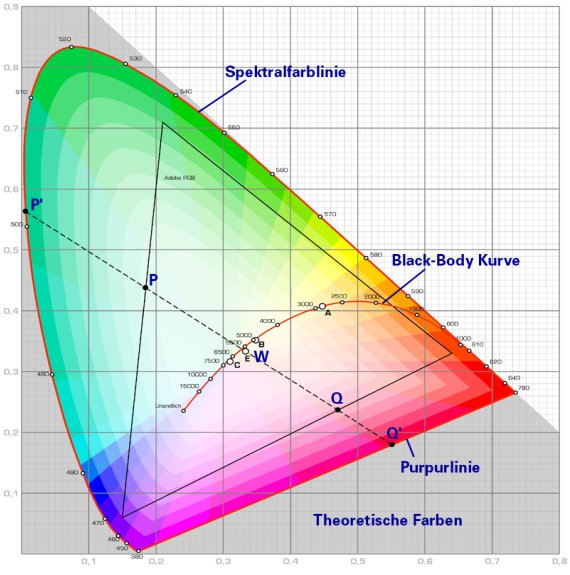
Hodnoty X a Z lze za podmínky znalosti hodnot Y, x a y dopočítat zpátky:

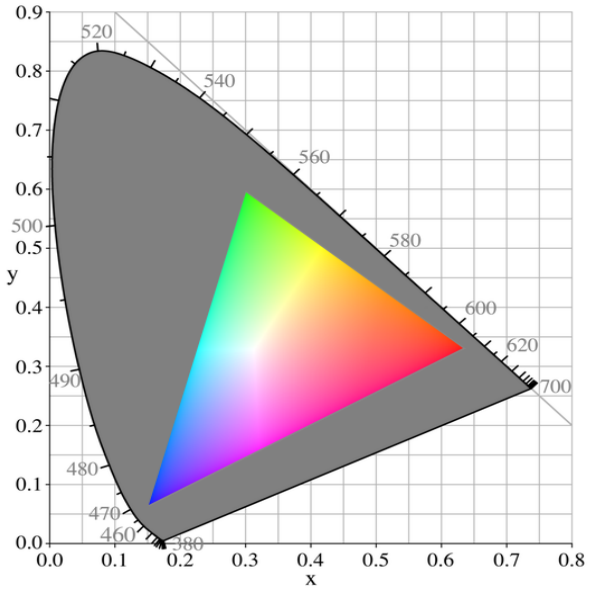
$$X = \frac{Y}{y} x \tag{3}$$

$$Z = \frac{Y}{y} (1 - x - y) \tag{4}$$

Na základě hodnot x a y můžeme sestavit diagram tónů – viz obr. 1 níže. Vnější křivka na horní části pestrobarevné oblasti se nazývá čára spektrálních barev, na jejím obvodu jsou vyznačeny odpovídající vlnové délky; na spodku oblasti je čára nespektrálních barev, také nazývaná čára fialových barev. Pestrobarevná plocha je podplochou oblasti, kde x + y ≤ 1. Mimo pestrobarevnou oblast leží teoretické barvy, které nedokážeme zobrazit.

Bílé izoenergické světlo D_{5000}, označované také jako D_{50} a v diagramu písmenem W, má v diagramu CIE_{xy} souřadnice x = y = $^1/_3$. Leží na linii bílých barev, která je v diagramu okótována stupnicí teploty barvy. V praxi – v televizních obrazovkách – se za bílé považuje normalizované světlo D_{6500}, mající v CIE_{xy} souřadnice x = 0.31271, y = 0.32902.





Ilustrace 1 (vlevo): Diagram CIE-xy.
Zdroj: [CIE_{XYZ}wiki]

Ilustrace 2 (vpravo): Tentýž diagram se zvýrazněným gamutem – oblastí, kterou dokáže zobrazit konkrétní zařízení.
Zdroj: [GamutWiki]

Nyní můžeme také zavést pojem sytost barvy. Uvažujme například spektrální barvu, označenou na diagramu písmenem P. Zavedeme si myšlenou polopřímku, vycházející z bodu W a procházející bodem P a konstruujeme její průsečík P' s čarou spektrálních barev. Sytost barvy definujeme:

$$H = \frac{|W\,P|}{|W\,P'|} \tag{5}$$

Z uvedeného je vidět, že mísením spektrálních barev získáváme tutéž barvu, ale s menší sytostí, které odpovídá určitý bod na spojnici příslušné barvy a bodu W. Postup funguje i obráceně: Tón libovolné barvy zjistíme pomocí průsečíku polopřímky, vycházející z bodu W a procházející vybraným bodem diagramu, s čarou spektrálních barev, ze které odečteme vlnovou délku.

Na diagramu z obrázku je také vidět ohraničená trojúhelníková oblast, zřetelněji znázorněná na obrázku 2 na předchozí straně. Nazývá se gamut a zachycuje oblast barvev – podmnožinu diagramu CIE_{XY} – které dokáže reprodukovat nějaké konkrétní zařízení. Jiný gamut bude mít barevný televizor Tesla Rubín[1] ☺, jiný gamut bude mít soudobý kancelářský LCD panel, odlišný gamut bude mít profesionální grafický monitor[2] a ještě jiný bude mít barevná inkoustová tiskárna.

Existuje lineární transformace z barevného prostoru CIE_{XYZ} do nejčastěji používaného RGB. Standardizovaná podoba vypadá takto:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.49 & 0.31 & 0.20 \\ 0.17697 & 0.81240 & 0.01063 \\ 0.00 & 0.01 & 0.99 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{6}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 2.36461 & -0.89654 & -0.46807 \\ -0.51517 & 1.42641 & 0.08876 \\ 0.00520 & -0.01441 & 1.00920 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{7}$$

1 Československý výrobek z přelomu 70. a 80. let 20. století, vyznačoval se intenzivnějším podáním červených odstínů v poměru k ostatním barvám.

2 Profesionální grafické monitory jsou vysoce kvalitní CRT displeje s prodejní cenou v řádech mnoha set, nebo i tisíců dolarů.

CIE$_{Lab}$

CIE$_{Lab}$ je barevný prostor, odvozený z CIE$_{XYZ}$ tak, aby byl *perceptuálně lineární* ve větší míře než CIE$_{Luv}$, probraný dále. CIE se snažila o to, aby velikost rozdílu složek barvy byla úměrná rozdílu barevného dojmu. Nelineární převodní vztahy mají napodobovat logaritmický průběh reakce lidského oka. Je to nejkompletnější barevný model, popisující všechny barvy, viditelné lidským okem. Je založen na třech níže popsaných parametrech. Pro jeho zobrazení by byl zapotřebí trojrozměrný obraz, v praxi se to ale řeší řezem při konstantním parametru L*.

- L* … světlost barvy, L* = 0% označuje černou, L* = 100% bílou
- a* … pozice mezi zelenou (a* = −128) a purpurovou (a* = 127)
- b* … pozice mezi modrou (b* = −128) a žlutou (b* = 127)

Model byl vyvinut také s úmyslem, aby sloužil jako referenční model nezávislý na zařízení. Proto je důležité uvědomit si, že vizuální reprezentace gamutu se všemi barvami nemůže být věrohodná, slouží pouze jako pomůcka k pochopení podstaty.

Transformace CIE$_{XYZ}$ → CIE$_{Lab}$:

$$L^* = 116 \cdot f\left(\frac{Y}{Y_n}\right) - 16 \tag{8}$$

$$a^* = 500 \cdot \left[f\left(\frac{X}{X_n}\right) - f\left(\frac{Y}{Y_n}\right) \right] \tag{9}$$

$$b^* = 200 \cdot \left[f\left(\frac{Y}{Y_n}\right) - f\left(\frac{Z}{Z_n}\right) \right] \tag{10}$$

$$f(t) = \sqrt[3]{t}, t > 0.008856 \tag{11a}$$

$$f(t) = 7.787t + \frac{16}{116}, t \le 0.008856 \tag{11b}$$

Hodnoty X_n, Y_n a Z_n jsou hodnoty tristimulu pro referenční bílé světlo. Rozdělení funkce f(t) na dva funkční předpisy jsme provedli, aby funkce neutekla do nekonečna pro t = 0.

Transformace $CIE_{Lab} \rightarrow CIE_{XYZ}$:

$$Nechť\ f_y = \frac{L^* + 16}{116};\quad f_x = f_y + \frac{a^*}{500};\quad f_z = f_y - \frac{b^*}{200};\quad \delta = \frac{6}{29} \tag{12}$$

$$Je-li\ f_y > \delta : Y = Y_n f_y^{\,3};\quad jinak : Y = \frac{f_y - 16}{116} 3\delta^2 Y_n$$

$$Je-li\ f_x > \delta : X = X_n f_x^{\,3};\quad jinak : X = \frac{f_x - 16}{116} 3\delta^2 X_n$$

$$Je-li\ f_z > \delta : Z = Z_n f_z^{\,3};\quad jinak : Z = \frac{f_z - 16}{116} 3\delta^2 Z_n$$

CIE_{Luv}

Systém CIE_{XYZ} je sice výpočetně nenáročný, má ale určitou nevýhodu: na některé barevné přechody – typicky mezi modrou a žlutou – lidské oko reaguje lépe než na jiné. Ze znalosti souřadnic dvou barev A a B v diagramu CIE_{xy} nedokážeme říci, zda bude rozdíl „velký“ nebo „malý.“ Proto se zavádějí tzv. perceptuálně rovnoměrné barevné systémy, jako např. CIE_{Luv} nebo CIE_{LAB}.

Pro převod ze systémů CIE_{XYZ}, resp. CIE_{xy} byly definovány následující vztahy:

$$u' = \frac{4X}{X + 15Y + 3Z} = \frac{4x}{-2x + 12y + 3} \tag{13}$$

$$v' = \frac{9Y}{X + 15Y + 3Z} = \frac{9y}{-2x + 12y + 3} \tag{14}$$

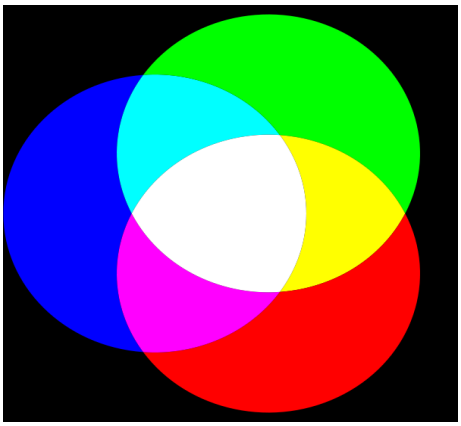
Hodnota L se vypočítá stejně, jako ve výše popsaném prostoru CIE_{Lab}. Rozšířením CIE_{Luv} je prostor CIE_{UVW}, definovaný transformací:

$$\begin{aligned} U &= 13W(u' - u'_w) \\ V &= 13W(v' - v'_w) \\ W &= 25\sqrt[3]{Y} - 17 \end{aligned} \tag{15}$$

… kde u'_w a v'_w jsou hodnoty u', v' získané pro smluvní bílé světlo.

2.1.2 RGB

RGB je aditivní barevný systém vyvinutý pro účely zpracování obrazu pomocí výpočetní techniky a částečně také pro barevnou televizi. Funguje na principu sčítání barev jednotlivých zdrojů v absolutní tmě – viz obrázek 3. Vlnové délky základních barev stanovila na základě měření a referenčních pozorovatelů komise CIE. Pro jednotlivé barvy: červená = 780 nm, zelená = 546.1 nm, modrá = 435.8 nm.



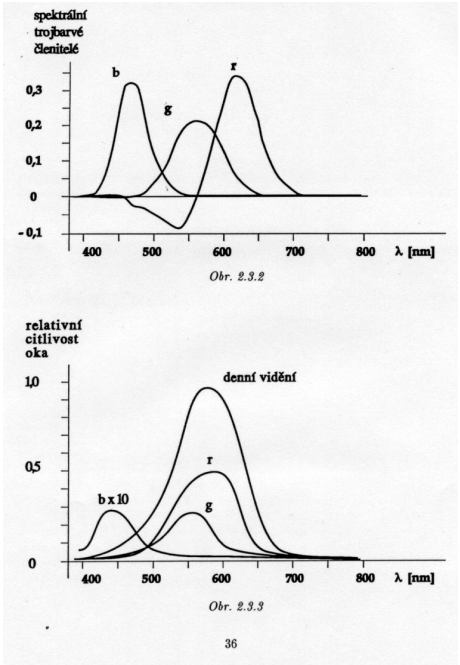
Ilustrace 3: Aditivní míchání barev v systému RGB. Zdroj: [RGBwiki]

Vezmeme-li v úvahu viditelné spektrum vlnových délek, potom každé vlnové délce odpovídá určitá barva. Jsou-li za základní barvy vybrány červená, zelená a modrá, potom lze ostatní barvy vyjádřit pomocí váhového součtu jednotlivých složek:

$$\begin{aligned} c_1 &= \begin{bmatrix} r_1 & g_1 & b_1 \end{bmatrix} \\ c_2 &= \begin{bmatrix} r_2 & g_2 & b_2 \end{bmatrix} \\ c = c_1 + c_2 &= \begin{bmatrix} r_1 + r_2 & g_1 + g_2 & b_1 + b_2 \end{bmatrix} \end{aligned} \tag{16}$$

Relativní citlivost oka je znázorněna na obrázku 4 dole. Lidské oko je schopno rozlišit asi 350 tisíc odstínů. Na obrázku 4 nahoře jsou ukázány průběhy jednotlivých váhových koeficientů, které se též nazývají trichromatické spektrální činitele.

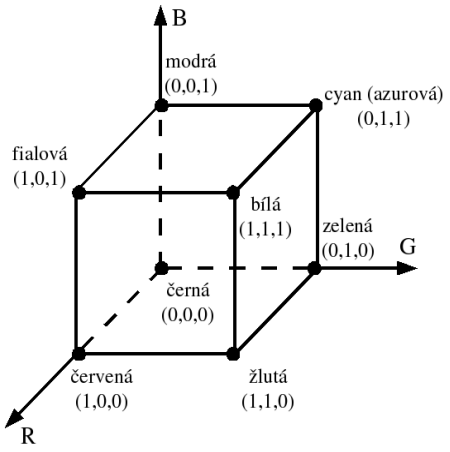
Z obrázku je vidět, že některé barvy nedokážeme reprezentovat součtem základních barev – nedokážeme totiž technicky vyrobit zápornou intenzitu osvětlení. Nejviditelnější je



Ilustrace 4: Průběh spektrálních činitelů a relativní citlivosti oka. Zdroj: [Ska93]

to na příkladu červené barvy. Je-li barva c tvořena vektorem trichromatických činitelů $[r, g, b]^T$ jehož složka r má zápornou hodnotu, potom barva získaná součtem barvy c a vektoru $[-r, 0, 0]^T$ odpovídá barvě $c' = [0, g, b]^T$.

Barevný prostor RGB můžeme zobrazit jako trojrozměrnou jednotkovou krychli – viz obr. 5. Každé barevné složce přiřadíme jednu osu, označíme je R, G, B. Z obrázku je vidět, že každou barvu, kterou dokážeme vyjádřit součtem základních barev, dokážeme zobrazit jako bod v prostoru, respektive jako vektor vycházející z počátku souřadného systému. Z obrázku je také vidět, že změnou délky vektoru při zachování směru dostaneme stejnou barvu, ale s jiným jasem. Je ale nutné zdůraznit, že tato úvaha opačným směrem nefunguje. Stejně dlouhé vektory dvou různých barev neodpovídají stejnému jasu a jednotková rovina není rovinou konstantního jasu. Jsou-li všechny tři barevné složky v poměru 1 : 1 : 1, je poměr skutečných jasů 1 : 4.6 : 0.06. Jas se udává v jednotkách cd/m^2.



Ilustrace 5: Jedna z reprezentací barevného systému RGB. Zdroj: [RGBwiki]

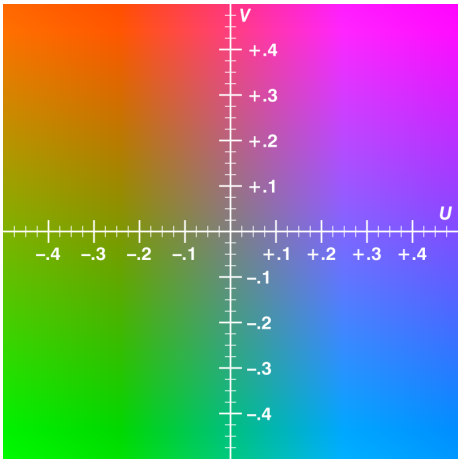
V počítači barvy z modelu RGB reprezentujeme jako trojici intenzit jednotlivých základních barev. Intenzity se mohou udávat reálným číslem z intervalu <0,1>, procentuálně (0% … 100%) nebo přirozeným číslem z intervalu <0, 255>. Použití posledního způsobu nám dává možnost reprezentace 16.7 milionu různých barevných odstínů, ale jak bylo zmíněno výše, lidské oko jich dokáže rozlišit jen asi 350 tisíc.

RGBA

RGBA je rozšířením modelu RGB o informaci o průhlednosti – takzvaný α-kanál. Hodnota 0 označuje zcela neprůhledný barevný bod, nejvyšší hodnota (1, 255 nebo 100%) znamená naopak zcela průhledný pixel. S výhodou se používá všude tam, kde se z nejrůznějších příčin obrazy na sebe vrství. Typickou ukázkou jsou moderní grafické editory jako GIMP nebo Adobe Photoshop, ve kterých úpravami α-kanálu můžeme vyrábět barevné přechody. S vrstvením pixelů na sebe se ale také musíme vypořádat při renderování trojrozměrné scény, když se v ní nacházejí průhledné objekty, např. skleněné okno ve zdi.

2.1.3 YIQ a YUV

Jedná se o barevné systémy, vyvinuté pro účely barevného televizního vysílání a řešící také kompatibilitu černobílých a barevných televizních přijímačů. YIQ je určen pro televizní standard NTSC, používaný zejména v severní Americe, v Japonsku a v Jižní Koreji, YUV je určen pro systém PAL, který se používá v Evropě, v Africe, v Austrálii a v jižní části Asie. Složka Y v obou barevných systémech nese informaci o jasu příslušného bodu, informace o odstínu barvy jsou obsaženy ve zbývajících dvou složkách. V systému YIQ je ve složce I pozice barvy mezi modrou a oranžovou a ve složce Q pozice mezi fialovou a zelenou. V systému YUV jsou složky U a V počítány z rozdílu mezi jasem a barvami.



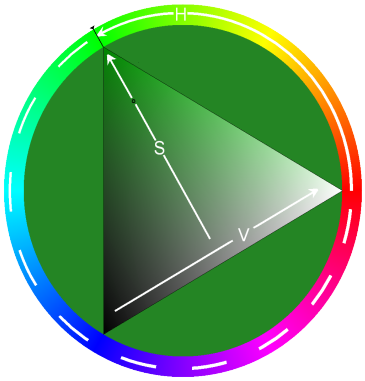
Ilustrace 6: Znázornění roviny UV v barevném systému YUV. Zdroj: [YUVwiki]

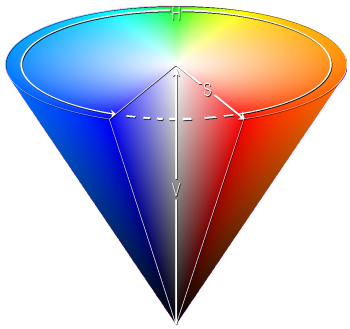
- $U = k_1 * (B - Y)$ … modrá – jas
- $V = k_2 * (R - Y)$ … červená – jas

Přesné převodní vztahy pro oba systémy lze najít např. v [YUVwiki], [YIQwiki]. Z těchto barevných systémů byly menšími nebo většími modifikacemi vytvořeny další barevné systémy pro některé specifické aplikace, jako je digitální video nebo televizní standard SECAM.

2.1.4 HSV

Model HSV definuje barevný prostor, specifikovaný třemi komponentami, H, S a V – viz níže. Vznikne nelineární transformací systému RGB a často se používá v grafických programech, kde je nejčastěji reprezentován kruhem barev, viz obr. 7. Na obvodu kruhu se vybírá tón barvy, v trojúhelníku uprostřed potom kombinace jasu a sytosti. Jiným způsobem vizualizace je kuželové zobrazení – viz obr. 8. HSV se s výhodou používá při výrobě barevných přechodů.





Ilustrace 7 (nahoře): Kruhová reprezentace barevného modelu HSV

Ilustrace 8 (dole): Kuželová reprezentace.

Zdroj (oba obrázky): [HSVwiki]

- H … tón barvy (hue), nejčastěji 0 – 360, někdy též 0 – 100%. Udává dominantní vlnovou délku barvy, vyjma intervalu 240 – 360 (oblast fialových barev)
- S … sytost (saturation) barvy, 0 – 100% (viz CIE_{XYZ})
- V … jas (value), 0 – 100%.

Grafici často upřednostňují model HSV, díky jeho podobnostem s lidským vnímáním barev. V modelech RGB a CMYK se výsledné barvy dosahuje aditivním, respektive

subtraktivním, mícháním barev. Naproti tomu HSV obsahuje takovou informaci o barvě, která je člověku srozumitelná – jaký tón barva má, jak je sytá a jak je světlá / tmavá. V tomto ohledu lze poukázat také na podobný barevný model HSL, který – podle některých názorů – tyto, pro člověka srozumitelné, informace dokáže reprezentovat ještě lépe.

Transformace

Uvažujme: $$H \in (0,360)$$ $$S, V, R, G, B \in (0,1)$$

RGB → HSV:

Nechť MAX = max{R, G, B}, MIN = min{R, G, B}. Potom:

$$H = \begin{cases} nedefinováno & když\ MIN = MAX \\ 60 \cdot \frac{G-B}{MAX-MIN} + 0 & když\ MAX = R\ a\ G \geq B \\ 60 \cdot \frac{G-B}{MAX-MIN} + 360 & když\ MAX = R\ a\ G < B \\ 60 \cdot \frac{B-R}{MAX-MIN} + 120 & když\ MAX = G \\ 60 \cdot \frac{R-G}{MAX-MIN} + 240 & když\ MAX = B \end{cases} \quad (17)$$

$$S = \begin{cases} 0 & když\ MAX = 0 \\ 1 - \frac{MIN}{MAX} & když\ MAX \neq 0 \end{cases} \quad (18)$$

$$V = MAX \quad (19)$$

RGB → HSV:

$$H_i = \left\lfloor \frac{H}{60} \right\rfloor mod\ 6; \quad f = \frac{H}{60} - H_i \quad (20)$$

$$p = V(1-S); \quad q = V(1-fS); \quad t = V(1-(1-f)S)$$

$$\begin{array}{llll} H_i = 0 & \rightarrow & R = V & G = t \quad B = p \\ H_i = 1 & \rightarrow & R = q & G = V \quad B = p \\ H_i = 2 & \rightarrow & R = p & G = V \quad B = t \\ H_i = 3 & \rightarrow & R = p & G = q \quad B = V \\ H_i = 4 & \rightarrow & R = t & G = p \quad B = V \\ H_i = 5 & \rightarrow & R = V & G = p \quad B = q \end{array}$$

Hovoříme-li o systému HSV, můžeme se také zmínit o tzv. doplňkových barvách – viz obr. 9. Dvě barvy jsou doplňkové právě tehdy, když jejich smísením dostaneme nějaký odstín šedé. Pro barvu danou koeficienty (H, S, V) existuje doplňková barva (H', S', V') taková, že jejich smísením ve stejném poměru dostaneme barvu, jejíž saturace S = 0.

Ilustrace 9: Diagram doplňkových barev v modelu HSV. Barvy ležící na protějších stranách kruhu jsou navzájem doplňkové. Zdroj: [CCwiki]

Doplňkové barvy:

$$H' = \begin{matrix} H-180 \Leftrightarrow H \geq 180 \\ H+180 \Leftrightarrow H < 180 \end{matrix} \quad (21)$$

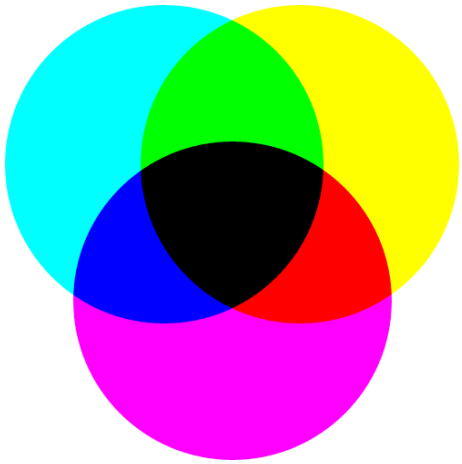
$$S' = \frac{VS}{V(S-1)+1} \quad (22)$$

$$V' = V(S-1)+1 \quad (23)$$

2.1.5 CMY a CMYK

Model CMY je subtraktivní barevný model. Mícháním barev omezujeme barevné spektrum, které se odráží od povrchu, nejčastěji bílého. Uplatnění nalézá především v oblasti barevného tisku. Stejně jako model RGB i model CMY pracuje se základními barvami:

- Azurová – C (cyan)
- Purpurová – M (magenta)
- Žlutá – Y (yellow)



Ilustrace 10: Mísení barev v subtraktivním barevném systému CMY. Zdroj: [CMYKwiki]

V systému RGB platilo, že smísením všech základních barev dostaneme bílou. Systém CMY je ale subtraktivní, mícháním vždy dostáváme tmavší barvy. Limitním případem je černá barva, kterou získáme smísením všech tří základních barev – viz obr. 9. Výroba černé barvy tímto způsobem je ale velmi drahá a v praxi má často nežádoucí barevný nádech. Proto se častěji používá systém CMYK, který je rozšířen právě o černou barvu.

Při tisku za použití čtyř barev dostaneme relativně dobré a kontrastní výsledky. Barvy ale nebudou a nemohou přesně odpovídat, protože modely RGB a CMYK mají každý jiný gamut – viz odstavec CIE. Například nedokážeme na papír dostat jasně modrou barvu (RGB: [0, 0, 255]), při tisku bude nahrazena nepříliš podobným odstínem modro-fialové, která je nejbližším ekvivalentem.

Konverze mezi RGB a CMY

Konverze mezi těmito barevnými prostory existují, ale nejsou vratné, takže po převodu tam a zpět nemusíme dostat přesně původní barvu.

CMYK → RGB

- Krok 1: CMYK → CMY:
  $B_{CMY} = \{C', M', Y'\} =$
  $B_{CMY} = \{C(1 - K) + K, M(1 - K) + K, Y(1 - K) + K\}$
- Krok 2: CMY → RGB
  $B_{RGB} = \{1 - C', 1 - M', 1 - Y'\}$
  $B_{RGB} = \{(1 - C)(1 - K), (1 - M)(1 - K), (1 - Y)(1 - K)\}$

RGB → CMYK

- Krok 1: RGB → CMY:
  $B_{CMY} = \{C', M', Y'\} =$
  $B_{CMY} = \{1 - R, 1 - G, 1 - B\}$
- Krok 2: CMY → CMYK
  $K = \min\{C', M', Y'\}$
  Jestliže $K = 1 \rightarrow B_{CMYK} = \{0, 0, 0, 1\}$
  Jinak

$$B_{CMYK} = \left\{ \frac{C' - K}{1 - K}, \frac{M' - K}{1 - K}, \frac{Y' - K}{1 - K}, K \right\}$$

Rozšíření CcMmYK

V praxi se také občas používá rozšíření modelu CMYK o světlejší odstín azurové a purpurové barvy. Setkat se s tím můžeme u domácích inkoustových tiskáren, optimalizovaných pro tisk fotografií.

2.1.6 Další barevné systémy

Výčet barevných systémů není výše uvedenými v žádném případě vyčerpán, Rádlová jich ve své bakalářské práci [Rad02] rozebírá více než 100. Můžeme namátkou zmínit systém SΘW, založený na přepisu CIE UVW do polárních souřadnic, nebo barevný systém Opponent, modelující skutečné lidské vidění.

Každý barevný systém byl vytvořen se zřetelem na svůj specifický účel, největší rozdíl v myšlenkovém přístupu je mezi modely technickými (RGB a další) a uměleckými (HSV a jeho variace). Je tedy vidět, že oblast světla a barev, ačkoli se může zdát snadno zvládnutelná, nabízí velké pole působnosti pro zkoumání.

2.2 Barevné systémy v praxi: Půltónování

Praktická aplikace nabytých poznatků se – dle očekávání – neobejde bez problémů. Nejčastěji musíme řešit situaci, kdy chceme obraz, např. fotografii, zobrazit nebo vytisknout na zařízení, které nedokáže přesně zobrazit některé barvy, např. černobílá laserová tiskárna. Samozřejmě chceme, aby výsledek vypadal co nejvěrněji.

2.2.1 Metoda vzorů

Metoda vzorů je jedním ze základních algoritmů pro půltónování černobílých obrazů. Jednotlivé pixely obrazu jsou nahrazeny vzorem, jehož body navozují dojem požadovaného jasu. Rozměry vzorů volíme tak, aby počet jejich polí zvětšený o jeden odpovídal požadovanému počtu úrovní jasu.

Nejjednodušším případem je požadavek na dvě úrovně jasu – černou a bílou. Rozměr vzoru bude 1 x 1. Všechny hodnoty jasu větší nebo rovné definované

prahové hodnotě nahradíme bílým bodem, všechny menší hodnoty nahradíme černým bodem. Této metodě se také říká metoda konstantního prahu. Při požadavku na 5 úrovní jasu potřebujeme vzor 2 x 2:


0% černé
100% jasu


25% černé
75% jasu


50% černé
50% jasu


75% černé
25% jasu


100% černé
0% jasu

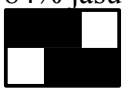
Lidské oko je citlivé na souvislé vodorovné a svislé pruhy, proto by se ve vzorech neměly vyskytovat, a jsou nahrazovány diagonálními šrafami.

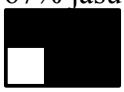
Samozřejmě lze vyrobit vzory vyšších rozměrů. Mohou mít i obdélníkový tvar, jako např. vzor 2 x 3 pro 7 úrovní jasu – viz tabulka níže (úroveň 100% jasu neuvádím ☺). Použitím obdélníkových vzorů lze někdy částečně odstranit problémy při zpracování, pokud má zařízení jiné rozlišení v horizontálním a vertikálním směru.

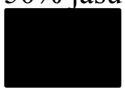

16% černé
84% jasu


33% černé
67% jasu


50% černé
50% jasu


67% černé
33% jasu


84% černé
16% jasu


0%

Vzory lze formálně reprezentovat pomocí matice se stejnými rozměry, jako má vzor. Prvky matice jsou přirozená čísla, udávající úroveň, ve které se na odpovídajícím místě vzoru objeví černý bod. Výše popsaný vzor pro 7 úrovní bude mít matici:

$$ {}^{(3,2)}T = \begin{bmatrix} 4 & 1 & 5 \\ 6 & 3 & 2 \end{bmatrix} \tag{24} $$

Používáním vzorů dochází ke zvětšení obrazu, protože každý jeho pixel nahrazujeme několika pixely vzoru. Nedochází zde ke ztrátě obrazové informace, a na zařízeních s vysokým rozlišením dostáváme dobré výsledky.

2.2.2 Floyd-Steinbergův algoritmus

Floyd-Steinbergův algoritmus je rozšířením výše zmíněné metody konstantního prahu o distribuci chyby do sousedních pixelů. Na rozdíl od metody vzorů rozměry obrazu tedy nevzrůstají. Princip algoritmu je zřejmý z následujícího schématu:

$$\begin{bmatrix} I(x,y) & +\frac{3}{8}\epsilon \\ +\frac{3}{8}\epsilon & +\frac{1}{4}\epsilon \end{bmatrix} \tag{25}$$

Floyd-Steinbergův algoritmus dle [FSpri] a [Ska93]:

```
Prah = (Imin + Imax) / 2;
Pro všechny řádky v obrázku dělej:
      Pro všechny body na řádku dělej:
            Jestliže I(x,y) < Prah:
                  VykresliBod(x, y, Cerna);
                  Chyba = I(x,y) - Cerna;
            jinak:
                  VykresliBod(x, y, Bila);
                  Chyba = I(x,y) – Bila;


            I(x+1,y) += 3/8 * Chyba;
            I(x,y+1) += 3/8 * Chyba;
            I(x+1,y+1) += 1/4 * Chyba;
```

Modifikací výše uvedeného algoritmu je *algoritmus chybové difuze*, kde jsou chyby distribuovány v jiných poměrech a navíc do pole (x-1, y+1):

$$\begin{bmatrix} & I(x,y) & +\frac{7}{16}\epsilon \\ +\frac{3}{16}\epsilon & +\frac{5}{16}\epsilon & +\frac{1}{16}\epsilon \end{bmatrix} \tag{26}$$

Floyd-Steinbergův algoritmus vykazuje proti prosté prahové funkci lepší zobrazení detailů. Navíc další malou modifikací (viz [Ska93]) můžeme zvýraznit změny kontrastu.

2.2.3 Adaptivní dithering

Další metodou, která umožňuje zobrazení při nezměněné rozlišovací schopnosti, je dithering, kterému se také někdy říká rozmývání. Je založen na principu vkládání náhodné chyby do obrazu. Chyby nelze vkládat zcela náhodně, takový přístup nedává dobré výsledky. Místo toho existuje adaptivní vzor, pomocí kterého dostaneme uspokojivé výsledky. Rozhodnutí, zda rozsvítit pixel na souřadnicích [i,j] tedy nezávisí jen na jeho intenzitě, ale také na čtvercové matici ${}^{(n)}D$ řádu n, obsahující hodnoty 0 až n^2-1. Po určení odpovídajícího prvku na souřadnicích [i,j] kde i = y mod n + 1, j = x mod n +1, je daný pixel aktivován jen tehdy, je-li ${}^{(n)}D(i,j) < I(x,y)$. Podle [Ska93] je nejmenší vzor definován takto:

$$ {}^{(2)}D = \begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix} \tag{27} $$

Vzory větších rozměrů (n je rozměr matice) lze odvodit z rekurentního vztahu:

$$ {}^{(2n)}D = \begin{bmatrix} 4\,{}^{(n)}D & 4\,{}^{(n)}D + 2\,{}^{(n)}U \\ 4\,{}^{(n)}D + 3\,{}^{(n)}U & 4\,{}^{(n)}D + 1\,{}^{(n)}U \end{bmatrix} \tag{28} $$

$$ {}^{(n)}U = \begin{bmatrix} 1 & \dots & 1 \\ \dots & & \dots \\ 1 & \dots & 1 \end{bmatrix} \tag{29} $$

Pro n=4 dostaneme:

$$ {}^{(4)}D = \begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix} \tag{30} $$

Algoritmus ([Ska93]):

```
Vhodně zvol n, vyrob matici;
Pro všechny řádky obrazu dělej:
      i = index_radku mod n;
      Pro všechny pixely na řádku dělej:
            j = index_sloupce mod n;
            Jestliže D(i,j) < I(x,y):
                  VykresliBod(x, y, Cerna);
            Jinak:
                  VykresliBod(x, y, Bila);
```

2.3 Osvětlení a stínování

Ve předchozím oddílu jsem rozebíral především dvourozměrnou grafikou. Ta má sice stále nezanedbatelný podíl v četnosti použití, ale již delší dobu se většímu zájmu grafiků, vývojářů i spotřebitelů těší trojrozměrná grafika. Z určitého pohledu je tato kapitola doplněním a rozšířením předchozího oddílu, ale může být nahlížena i zcela samostatně.

V oboru trojrozměrné grafiky vzniká obraz procesem renderingu, kdy se matematicky popsaná scéna spolu s definicí materiálů, světel a pozice a směru kamery převede na výsledný obraz na obrazovce. Při tom musíme spočítat osvětlení objektů a jejich stíny. Cílem snažení je dosažení realisticky vypadající scény.

2.3.1 Osvětlení

Pokud bychom se snažili do počítače převést skutečný fyzikální model osvětlení, brzo bychom dospěli k závěru, že je to nereálné. Světlo je totiž elektromagnetické vlnění. Na scéně může být mnoho zdrojů světla o různých barvách (vlnových délkách) i intenzitách (amplitudách), jejich vlnění se mohou různě skládat a interferovat. Je vidět, že bychom se nejspíš nedopočítali. Proto se pro osvětlení zavádí různá, často velmi tvrdá, zjednodušení; např. Newtonův geometrický model světla, které se šíří rovnými paprsky.

Po zjednodušení osvětlovacího modelu nám zbude poměrně málo parametrů, ovlivňující výsledný jas plošky v prostoru. Jsou to: směr, vzdálenost a svítivost světla v prostoru a směr pohledu pozorovatele. Funkce, která na základě těchto hodnot určí odražené světlo se jmenuje BRDF *(bidirectional reflectance distribution function)*. Podle [GRG5] je její tvar:

$$f_r(x, \boldsymbol{\omega}_r, \boldsymbol{\omega}_i) = \frac{d\, L_r(x, \boldsymbol{\omega}_r)}{d\, L_i(x, \boldsymbol{\omega}_i)(\boldsymbol{\omega}_i \cdot \boldsymbol{n})\, d\, \boldsymbol{\omega}_i} \tag{31}$$

kde ω_i je směr dopadu světla, ω_r směr směr odrazu světla a dL_i a dL_r dopadající, respektive odražená, radiance.

Samostatnou problematikou jsou světelné zdroje, kterých existuje několik druhů: bodové zdroje, rovnoběžné zdroje, plošné zdroje, reflektory a obloha – pokud je ve scéně přítomna. Zdroje už ze své podstaty emitují energii.

Nejjednodušší je bodový zdroj, který vyzařuje energii rovnoměrně do celého okolí. Fotony se nikam neztrácejí, vlnoplocha „obsahuje“ stejný výkon v libovolné vzdálenosti od zdroje. Mění se tedy plošná hustota výkonu, která musí zohlednit vzdálenost od zdroje. Na element plochy dP proto dopadne pouze poměrná část výkonu. Rovněž musíme zohlednit fakt, že vnímaný jas by neměl souviset s velikostí plochy – poštovní známka by měla být stejně jasná jako list papíru A4. Zavedeme tedy „hustotu světla,“ jako poměr intenzity dopadajícího světla a velikosti elementární plochy:

$$(L \approx) \qquad E = \frac{F}{dP} = \frac{I \cdot \cos\alpha}{4\pi d^2} \tag{32}$$

kde I je intenzita světla, α úhel, který svírají paprsek a normálový vektor plošky dP a d vzdálenost plošky od zdroje světla. Za předpokladu matného povrchu, kde se světlo odráží do všech směrů stejně, lze výše uvedenou rovnici považovat za rovnici

světelnosti plošky (L). Pro přehlednost lze vzorec (32) přepsat takto:

$$L = k_d I_d \frac{(N \cdot S)}{d^2} \tag{33}$$

kde jsme kosinus úhlu nahradili skalárním součinem normálového vektoru plošky a směrového vektoru odraženého paprsku, výsledek je tentýž. 4π můžeme promítnout do difuzního koeficientu k_d. Tímto způsobem spočítáme tzv. difuzní (rozptylovou) světelnost od jednoho světelného zdroje. Zdrojů může být obecně ve scéně více, což zohledníme v následujícím vzorci (34).

Kromě difuzní odrazové složky z předcházejícího vzorce, která se též nazývá Lambertova, je celková světelnost plošky tvořena také ambientní složkou (indexy a) a Phongovou spekulární složkou, která definuje odraz světla od lesklého povrchu a zohledňuje míru odchylky od odraženého paprsku (skalární součin odraženého paprsku a vektoru se zadanou odchylkou). Parametr n v mocnině určuje rozptyl odrazu.

Phongův osvětlovací model

Osvětlovací model, založený na těchto složkách se nazývá Phongův osvětlovací model. Vyvinul ho vietnamský vědec Bui Tuong Phong a představil ho ve své disertační práci na univerzitě v Utahu v USA v roce 1973. Světelnost plošky má podle [ZPGcv5] následující tvar:

$$L = k_a I_a + \sum_{\forall \text{světla}} \left(k_d I_d \frac{N \cdot S}{d^2} \; + \; k_s I_s \frac{(R \cdot V)^n}{d^2} \right) \tag{34}$$

kde R je směrový vektor odraženého paprsku a V směrový vektor k pozorovateli.

Z výše uvedených vzorců by stále mělo být zřejmé, že takto definovaný osvětlovací model je stále pouze hrubou aproximací reality a má některé nectnosti. Vykreslené

povrchy vypadají, jako by byly z plastu. Větší věrnosti dosáhneme sofistikovanějšími výpočty jednotlivých parametrů, např. Oren-Nayarovým výpočtem difuzní složky nebo výpočtem spekulární složky podle Warda nebo Torrance-Sparrowa.

Osvětlovací model Torrance-Sparrow

Model Torrance-Sparrow se snaží více přiblížit realitě a fyzikálním jevům, které vznikají při odrazech světla. Vychází se z předpokladu, že povrch tělesa je tvořen velkým množstvím ideálně rovných mikroplošek. Plošky ale mají různou orientaci, čímž simulujeme nepravidelnou fyzikální strukturu reálných povrchů. Torrance a Sparrow předpokládali pro orientaci plošek Gaussovo rozdělení. Součtem odraženého světla od všech relevantních plošek tedy vznikne rozptýlený zrcadlový odraz.

Zavedeme distribuční funkci hrubosti povrchu, která ve výpočtu zohlední výše zmíněné Gaussovo rozdělení orientace plošek:

$$D = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{\alpha^2}{2\sigma^2}} \tag{35}$$

kde α je úhel mezi skutečnou normálou dané plošky a průměrnou normálou ze všech plošek a σ směrodatná odchylka daného rozložení. Dále do modelu zavedeme funkci zeslabení intenzity, kterou modelujeme sebe-zastiňování a maskování plošek:

$$G = min\left[1.0,\ \frac{2(N\cdot H)(N\cdot E)}{(E\cdot H)},\ \frac{2(N\cdot H)(N\cdot L)}{(E\cdot H)}\right] \tag{36}$$

kde N je průměrný normálový vektor, H normálový vektor konkrétní plošky, E směrový vektor odraženého paprsku a L směrový vektor ke zdroji světla.

Posledním činitelem je funkce odrazivosti, která respektuje Fresnelův zákon lomu a odrazu světla. Dává informaci o tom, jaká část světla se od plošky odrazí, než aby byla absorbována:

$$F=\frac{(g-c)^2}{2(g+c)^2}\left[1+\frac{(c(g+c)-1)^2}{(c(g-c)+1)^2}\right] \tag{37}$$

$$c=(H\cdot L) \tag{38}$$

$$g=\sqrt{n^2(\lambda)+c^2-1} \tag{39}$$

Význam H a L je stejný jako ve vzorci výše, n je index lomu materiálu v závislosti na vlnové délce světla. Vypočítá se z naměřené hodnoty odrazu F_0 pro kolmo dopadající světlo:

$$n=\frac{1+\sqrt{F_0(\lambda)}}{1-\sqrt{F_0(\lambda)}} \tag{40}$$

Hanák dává v [GRG6] návod, jak vztah (37) aproximovat, abychom se vyhnuli dělení a odmocňování a tím urychlili výpočet:

$$F\approx(1-(1-(N\cdot E)^5))+n(1-(N\cdot E))^5 \tag{41}$$

Nyní již nezbývá nic jiného, než nadefinované a vypočítané parametry použít. Dostaneme alternativní podobu spekulární složky Phongova osvětlovacího modelu. Získané koeficienty D, E a F ještě vydělíme skalárním součinem průměrné normály a odraženého paprsku, tím ošetříme skutečnost, že s rostoucí odchylkou od normály pozorovatel vidí víc plošek. Finální vzorec modifikovaného Phongova modelu po dosazení vypadá takto:

$$L=k_a I_a+\sum_{\forall \text{světla}}\left(k_d I_d\frac{N\cdot S}{d^2}\ +\ k_s I_s\frac{D\cdot G\cdot F}{d^2\cdot(E\cdot N)}\right) \tag{42}$$

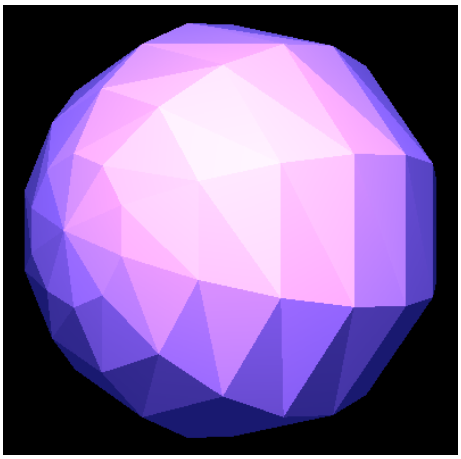
Takto modifikovaný osvětlovací model umožňuje simulovat nepravidelnost reálného povrchu. Dobře dokáže zobrazit zrcadlové odrazy, rozptýlené právě na základě nepravidelného povrchu a umožňuje vykreslovat materiály kovového i nekovového charakteru. Přestože poskytuje velmi dobré výsledky, častěji je používán původní Phongův model, protože je výpočetně jednodušší a vyrenderované scény stále působí z hlediska fotorealismu uspokojivě.

2.3.2 Stínování

Z předcházejícího odstavce nyní máme dostatek informací, jak spočítat osvětlení pro jednu každou konkrétní plošku tělesa. Zbývá nám doladit výsledek tak, aby byl co nejvíce fotorealistický i při použití objektů s nižším počtem polygonů.

Konstantní stínování

Nejjednodušším řešením je ponechat výsledek tak, jak přijde z Phongova osvětlovacího modelu. Zobrazujeme-li např. krychli, je výsledek uspokojivý a nepotřebujeme žádné další výpočty. Složitější geometrická tělesa, jako např. koule by ale vypadaly velice kostrbatě. Mohli bychom použít více polygonů, tím by ale stoupla výpočetní náročnost renderovacího procesu. Proto se snažíme světelnost jednotlivých plošek nějakým způsobem interpolovat tak, aby přechody byly co nejplynulejší.

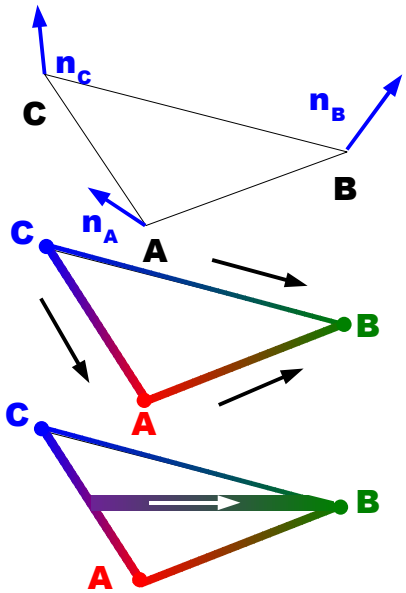


Ilustrace 11: Koule s nízkým počtem polygonů, stínovaná metodou konstantního stínování. Zdroj: vlastní produkce.

Gouraudovo stínování

První ze stínovacích modelů stínování pro tělesa s nízkým počtem polygonů odvodil v roce 1971 francouzský vědec Henri Gouraud, tehdy studující na univerzitě v Utahu v USA.

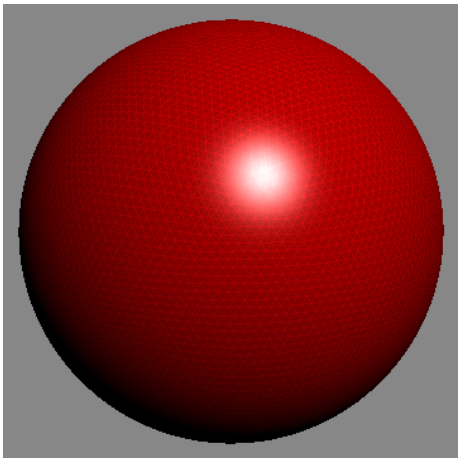
Princip je následující: Mějme model tělesa, v jehož každém vrcholu je definován normálový vektor. Pomocí Phongova (nebo obecně libovolného) osvětlovacího modelu



Ilustrace 12: Průběh Gouraudova stínování. Každému vrcholu určíme intenzitu osvětlení (barvu), hodnoty napřed interpolujeme po hranách, po promítnutí na obrazovku také po řádcích. Zdroj: vlastní produkce.

každému vrcholu přiřadíme výslednou intenzitu světla a provedeme interpolaci intenzity, nejdříve po hranách, následně (po promítnutí na stínítko obrazovky) také po jednotlivých řádcích obrazu.

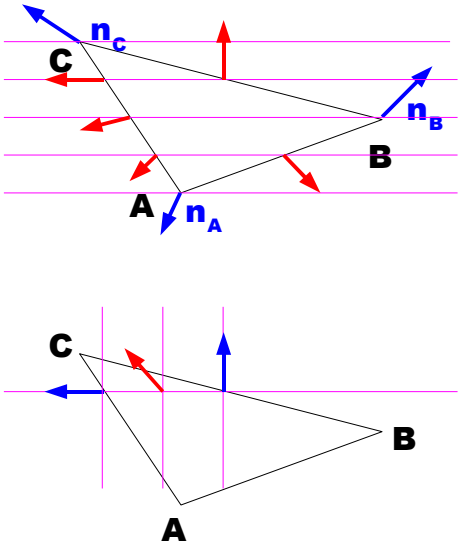
Výhodou algoritmu je jeho výpočetní jednoduchost, pocházející právě z interpolace intenzity. Výsledky (viz obr. 13) jsou sice o poznání lepší než u konstantního stínování, ale při nízkém počtu a velkých rozměrech polygonů jsou nepřirozené, velký důraz totiž dostanou odrazové efekty.



Ilustrace 13: Koule s vysokým počtem trojúhelníků, stínovaná Gouraudovou metodou s interpolací intenzit. Zdroj: [GSwiki]

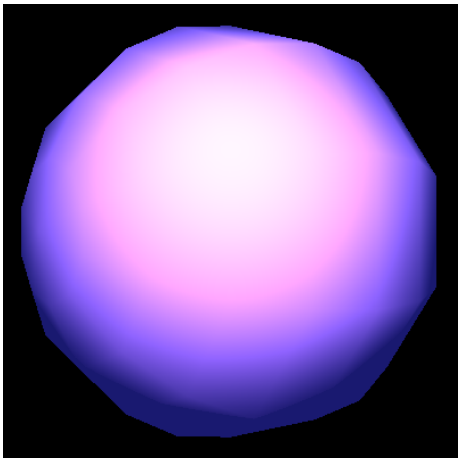
Phongův model

Phongův model stínování může být nahlížen jako vylepšení předcházejícího algoritmu. Na rozdíl od intenzity odraženého světla jsou interpolovány normálové vektory povrchu. Stejně jako u Gouraudova modelu jsou normály interpolovány napřed po hranách a po promítnutí na stínítko obrazovky také pro každý pixel na řádku, který náleží zobrazovanému polygonu. K získané normále v každém bodu na obrazovce je potom osvětlovacím modelem přiřazena intenzita světla a bod je zobrazen.



Ilustrace 14: Interpolace normálových vektorů povrchu po hranách a po obrazových řádcích ve Phongově stínovacím algoritmu. Zdroj: vlastní produkce.

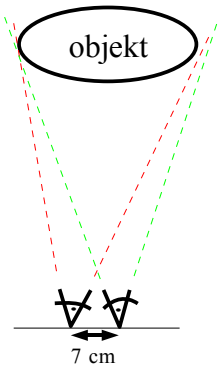
Právě díky interpolaci normál dává pro hladké a pravidelné povrchy Phongův stínovací model lepší výsledky než Gouraudův – viz obr. 15. Varianty tohoto algoritmu jsou interpretovány v hardwaru pod názvem „pixel / fragment shading.“ Osvětlení a stínování scény tedy můžeme provádět přímo na kartě, čímž pochopitelně ušetříme výpočetní čas na procesoru.



Ilustrace 15: Phongovo stínování na kouli definované nízkým počtem polygonů. Srovnej s obrázkem 11 na straně 32. Zdroj: vlastní produkce.

2.4 Stereoskopické promítání

Stereoskopie, doslova *dvojí vidění*, je způsob zobrazování trojrozměrné reality dvojrozměrnými prostředky. Snaží se ještě více přiblížit prostorovému lidskému vidění. Zdravý člověk má dvě oči, umístěné ve vzdálenosti asi 7 cm od sebe. Každé oko tedy dostává poněkud odlišný vjem, o jejich korektní spojení se stará mozek.



Ilustrace 16: Základní koncepce stereoskopie. Zdroj: vlastní produkce.

Nejstarší používanou stereoskopickou technikou jsou *stereogramy*. Jsou to dvojice obrázků pro levé a pravé oko, tištěné často na průsvitném papíře. K jejich prohlížení sloužil stereoskop, do kterého se karta s obrázky zasunula a při pohledu proti světlu získal uživatel dojem trojrozměrného obrazu.

Ani moderní výpočetní technika nezůstává ve stereoskopii stranou, právě naopak – ve vývoji jsou projekty trojrozměrné televize, se stereoskopií částečně souvisí také projekt víceuživatelské 3D televize.

Při technické aplikaci stereoskopie si musíme vystačit s jedinou zobrazovací plochou, na kterou nějakým způsobem potřebujeme dostat obrazy pro obě oči. Existuje několik způsobů realizace, nejjednoduššími anaglyfickými brýlemi počínaje přes závěrkové brýle (shutter glasses) a polarizované světlo, až po nové technologie, jako jsou hologramy.

2.4.1 Anaglyfy

Nejjednodušší stereoskopickou technikou jsou *anaglyfy*, a proti ostatním způsobům je také nejlevnější, kromě speciálních brýlí totiž nepotřebuje žádnou speciální techniku. Obrazy pro pravé a levé oko se pomocí speciálního softwaru transformují do jediného výsledného obrazu. Z jednoho obrazu se vezmou barvy podobné červené, ze druhého barvy podobné modré.

K prohlížení se používají jednoduché papírové nebo plastové brýle, které místo čoček mají barevné plastikové fólie. Před levým okem bývá červená fólie, před pravým zelenomodrá. Skrz červený filtr projde jen modré světlo, přes modrý naopak všechny barvy kromě modré. Mozek si pak oba vjemy spojí a získá iluzi barevného třírozměrného obrazu.

Tento způsob trojrozměrného zobrazování nalézá omezené uplatnění v počítačových hrách a v kinematografii, větší úspěchy ale sklízí technologie polarizovaného světla.

2.4.2 Systémy založené na polarizovaném světle

Technologie polarizovaného světla samozřejmě vykazuje vyšší kvalitu uživatelského dojmu než anaglyfy, stále je to ale dostatečně jednoduchá technologie, kterou si uživatel, má-li hluboko do kapsy, může snadno pořídit domů.

Obrazy pro levé a pravé oko jsou promítány na tutéž zobrazovací plochu, ale každý obraz má jinou polarizaci světla – pro jedno oko je polarizace horizontální, pro druhé vertikální. Toho lze dosáhnout například použitím dvou digitálních projektorů a polarizačních předsádek před čočku projektoru, na tomto principu funguje stereostěna na katedře informatiky Západočeské univerzity.

K prohlížení jsou zapotřebí speciální brýle s polarizačními filtry, které propustí pouze světlo se stejnou polarizací. Každé oko tak dostane správný obraz, který není zatížen barevnými deformacemi, jako je tomu v případě anaglyfů.

2.4.3 Další technologie

Za zmínku stojí systém Infitec, který funguje na podobném principu, jako systém s polarizovaným světlem. Základní barvy (červená, zelená a modrá) ale mají posunuté vlnové délky, zpětná korekce se provádí opět speciálními brýlemi, které vlnové délky barev upraví zpět na správné hodnoty. Za stereoskopickou technologii můžeme považovat také holografii.

3 Realizační část

Úkolem pro realizační část bylo vytvořit demonstrační systém pro demonstraci jednotlivých metod, včetně stereoskopie. Úlohu jsem rozdělil do dvou hlavních částí: Zvlášť je realizována část pro metody osvětlení a zvlášť je realizována část pro barevné systémy.

3.1 Použité programové nástroje

3.1.1 Programová část

Pro programovou část bakalářské práce jsem zvolil objektově orientovaný programovací jazyk C# a prostředí Microsoft .NET. Výhodou této kombinace je snadná výroba programového vybavení pro nejčastěji používanou platformu Windows, samozřejmě včetně uživatelského rozhraní u programu pro barevné systémy. Samotný jazyk C# je navíc velmi podobný jazyku Java, vyučovanému v prvním ročníku v předmětech *Počítače a programování 1* a *2*.

Kombinace jazyka C# a prostředí .NET je velmi vhodným pracovním nástrojem pro programování aplikací trojrozměrné grafiky v proprietárním prostředí DirectX od Microsoftu. DirectX je vysoce standardizované, a díky oficiální podpoře ze strany Microsoftu bez problémů spolupracuje s prostředím .NET, ze stejného důvodu je také nejpoužívanějším nástrojem pro 3D grafiku v prostředí Windows.

Jazyk C# a prostředí .NET jsou do značné míry platformově nezávislé – pro ne-Windowsové platformy se používá implementace *Project Mono*. Hlavní překážkou pro přenositelnost kódu je ale právě proprietární prostředí DirectX, které je pochopitelně k dispozici pouze pro Windows. Pro použití kdekoli jinde by bylo nutné program přizpůsobit pro alternativní standard OpenGL společnosti SGI, který v současné době spravuje Khronos Group.

Samozřejmě by nebyl problém program vyvinout v libovolném moderním jazyku, včetně zmíněné Javy. Mnou zvolená kombinace (C# a .NET pod Windows) má však řadu předností, z nichž některé již byly vyjmenovány výše.

- Uživatel dostane spustitelný *exe* soubor, na který stačí dvakrát kliknout,
- chybějící podporu pro .NET a DirectX si může snadno a bezbolestně doinstalovat,
- oficiální podpora a bezproblémová spolupráce s prostředím DirectX pro trojrozměrnou grafiku.

Verze použitého software

- Microsoft .NET SDK 2.0
- Microsoft Visual C# Express 2005
- Microsoft DirectX 9.0 SDK Update (October 2005)
- High-Level Shading Language – Shader Model 2.0
- Microsoft Windows XP Home

3.1.2 Dokumentace

Dokumentace k programu vznikla v open-source kancelářském balíku OpenOffice.org[3], verze 2.0.4. Jeho výhodou je – kromě nulových pořizovacích nástrojů – používání otevřeného standardu pro vytvářené dokumenty. Program navíc umí exportovat dokumenty do formátu PDF, jehož prohlížeč je dnes na většině počítačů.

Programátorskou dokumentaci jsem vyrobil pomocí open-source nástroje Doxygen[4]. Podle parametrů v textovém konfiguračním souboru generuje dokumentaci v mnoha formátech, včetně HTML.

3.2 Program pro demonstraci metod stínování

3.2.1 Obecný popis

Jedná se o program pro názorné předvedení základních stínovacích metod, konkrétně plochého stínování, Gouraudova stínování a Phongova stínování. Stínovací algoritmy jsou realizovány hardwarově, přímo na grafické kartě. V souladu s požadavky zadání je počítáno s možností stereoskopického zobrazení.

3.2.2 Analýza

Od programu se požaduje, aby po jednotlivých primitivech vykreslil nějaký zvolený demonstrační objekt. Každé primitivum je zapotřebí odpovídajícím způsobem vystínovat, k tomu potřebujeme znát jeho umístění v prostoru, normálový vektor ve vrcholech (případně průměrný normálový vektor) a umístění světelného zdroje.

3 Dostupné na http://www.openoffice.org/
4 Dostupné na http://www.doxygen.org/

V případě stereoskopického zobrazení je nutné vyrenderovat scénu ze dvou bodů, ležících 7 cm od sebe, tato hodnota odpovídá vzdálenosti očí.

Vykreslované objekty

Jako demonstrační objekty pro stínování jsem použil předdefinované objekty ve struktuře `Mesh` (namespace `Microsoft.DirectX.Direct3D`), konkrétně `Mesh.Sphere`, `Mesh.Box`, `Mesh.Torus` a `Mesh.Teapot`. Tím jsem si ušetřil práci při definování jednotlivých vrcholů a indexů pro trojúhelníkovou síť.

Samotná struktura `Mesh` je bez problémů použitelná pro Gouraudovo a Phongovo stínování, problém ale nastane u konstantního. Vertex-shader na grafické kartě totiž nemá žádné informace o kontextu renderovaného objektu, trojúhelníky zpracovává po vrcholech, takže si nemůže sám spočíst průměrný normálový vektor pro konstantní stínování. Jako rozumné řešení se jeví tyto normálové vektory předpočítat již při tvorbě vykreslovaného objektu a předat je do zařízení jako doplňkovou informaci.

Tím se dostáváme k dalšímu problému: Jeden vrchol může být sdílen mnoha trojúhelníky a pro každý trojúhelník chceme do bodu doplnit jinou dodatkovou informaci. To jsem vyřešil převodem objektu `Mesh` na množinu vzájemně nezávislých trojúhelníků, které pak lze vykreslit jednoduchým zavoláním `device.DrawPrimitives(...)`.

K doplňkové informaci o normále je také vhodné připojit určitý referenční bod v trojúhelníku, aby plocha byla vystínována konstantně i za zvlášť nepříznivých okolností. Mám na mysli příliš velký trojúhelník a světelný zdroj umístěný velmi blízko jednoho vrcholu. Osvětlovací model (viz dále) totiž k výpočtu potřebuje více údajů, než pouhou normálu, konkrétně směrový vektor ke světelnému zdroji a směrový vektor k pozorovateli. Ve výše zmíněném případě se u jednotlivých vrcholů trojúhelníka mohou tyto údaje velmi lišit, a proto bychom (i při konstantní normále)

mohli pro každý vrchol trojúhelníka obdržet jinou intenzitu osvětlení, které by tedy nebylo pro celou plochu konstantní. Na druhou stranu ale musím připustit, že pro malé trojúhelníky, dostatečně vzdálené od zdroje světla, je tento přístup tak trochu „s kanónem na vrabce," protože rozdíly v osvětlení budou stěží postřehnutelné.

Pro názornější demonstraci jsem se rozhodl, že bude dobré, když se objekt bude otáčet kolem svislé osy. To by samozřejmě šlo zajistit pomocí manipulace s daty objektu, ale v tomto jednoduchém případě, kdy je na scéně pouze jediný objekt, by to nebylo efektivní. Zvolil jsem jiný způsob – kamera se pohybuje po kruhové trajektorii a objektivem sleduje objekt. Výsledný vjem je stejný, ale bez zbytečných výpočtů.

Osvětlení a stínování

Moderní grafické karty disponují technickými prostředky pro hardwarovou realizaci algoritmů pro manipulaci s obrazovými daty, a stínování je jen jednou z mnoha možných aplikací. V prostředí DirectX se pro programování grafického procesoru používá vysokoúrovňový stínovací jazyk HLSL.

Pro výpočet osvětlení se používá varianta Phongova osvětlovacího modelu implementovaná v hardwaru, konkrétně HLSL funkce `lit(...)`, která provádí výpočet osvětlení na základě znalosti umístění světelného zdroje, normálového vektoru daného bodu a pozice pozorovatele. Výsledná barva pixelu je součtem ambientní, difusní a spekulární složky, které do shaderu předá program.

K výpočtu tedy potřebujeme znát prostorové souřadnice kamery a světelného zdroje. Obě informace můžeme kartě předat přímo, např. pomocí sdílených proměnných v hardwarovém stínovacím programu. Informace o umístění kamery je však také obsažena v tzv. *pohledové matici*, která je součástí transformačního procesu mezi „světovými souřadnicemi" vykreslované scény a souřadnicemi na stínítku obrazovky. Navíc, každé předávání dat kartě vyžaduje určitou časovou režii, a proto předávání duplicitních informací není nejlepším možným řešením.

V praxi se osvětlení scény počítá v souřadnicovém systému kamery, čímž se eliminuje nutnost předávat do stínovacího algoritmu její souřadnice. Naopak se všechny objekty ve scéně převedou do jejího souřadnicového systému, ve kterém kamera leží v počátku a „hledí“ ve směru osy z.

Transformace mezi světovými souřadnicemi a souřadnicemi kamery se podle [MSDXCST] provádí vynásobením světovou maticí a pohledovou maticí.

$$\boldsymbol{X_C} = \boldsymbol{X_W} \cdot M_W \cdot M_V \tag{43}$$

… kde X je uvažovaný bod ve světových (index W), respektive kamerových (index C) souřadnicích a M jsou transformační matice – světová (W) a pohledová (V).

V případě normálového vektoru je ale situace složitější. Lze ho sice převést do souřadnic kamery stejným způsobem jako bod, ale pokud bychom kromě této transformace prováděli ještě nějaké další, např. zvětšení pouze po jedné ose, mohlo by se stát, že po nich už nebude normálový vektor kolmý k povrchu transformovaného tělesa. Proto se používá následující transformace:

$$\boldsymbol{N_C} = \boldsymbol{N_W} \cdot \left((M_W \cdot M_V)^{-1} \right)^T \tag{44}$$

… kde N je normálový vektor ve světových (index W) nebo kamerových (index C) souřadnicích. Samotný jazyk HLSL ale nemá prostředky pro výpočet invertované matice, proto je nutné předat ji z hlavního programu.

Po transformaci souřadného systému má kamera souřadnice $[0, 0, 0]_C$. Světelný zdroj jsem pro zjednodušení položil dvě jednotky nad kameru, má tedy pevné souřadnice $[0, 2, 0]_C$. Jejich neměnné pozice využívám při výpočtu směrových vektorů ke světelnému zdroji a k pozorovateli.

3.2.3 Realizace

Aplikační logiku programu jsem vytvořil standardními prostředky prostředí .NET a DirectX, samotné stínování se provádí hardwarově na kartě za použití tzv. *shaderů* vytvořených v jazyce HLSL.

Jak jsem popsal výše, stínovaný objekt převádím na sekvenci trojúhelníků. Trojúhelníky jsou definovány trojicemi vrcholů. Žádný předdefinovaný formát vrcholu však nevyhovoval mým potřebám, tedy jsem si podle návodů (viz [MSDX9]) vytvořil vlastní strukturu `VertexPosNormPosNorm`, obsahující souřadnice bodu v prostoru (sémantika `POSITION0`), příslušný normálový vektor (`NORMAL0`) a dále referenční bod trojúhelníka (těžiště, sémantika `POSITION1`) a referenční normálu (`NORMAL1`). Použitý způsob deklarace přes `VertexElements` a `VertexDeclaration` je v DirectX dostupný od verze 9.

Stereoskopie

Pokud jde o realizaci stereoskopie, použil jsem kostru vzorového programu StereoWallTurorial ze serveru http://herakles.zcu.cz[5], který jsem ve značném rozsahu zmodifikoval a doplnil tak, aby splňoval požadavky zadání.

V principu jde o to, že je zapotřebí vyrenderovat scénu ze dvou bodů a získaný obraz zobrazit na dvou výstupních zařízeních, pokud jsou k dispozici. Je-li přítomno jen jedno zobrazovací zařízení (což je případ většiny domácích uživatelů), bude se scéna pochopitelně renderovat jen z jednoho bodu.

Přítomnost dvou a více renderovacích zařízení detekuji pomocí `Screen.AllScreens.Length`. Je-li tato hodnota vyšší než 1, potom je k dispozici stereo režim, ale dávám uživateli také možnost ho v nastavení odmítnout, potom se bude renderovat monoskopicky.

5 http://herakles.zcu.cz/education/zpg/down/StereoWallTutorial.zip [online]

Pozice jednotlivých kamer pro stereoskopické vykreslování se počítají ze souřadnic pozorovacího bodu na kružnici. Známe pohledový vektor, kamera vždy směřuje do počátku souřadného systému, kde je umístěn vykreslovaný objekt. Proto není problém vyrobit jednotkový kolmý vektor rovnoběžný „s rovinou podlahy“ - rovinou x-z. Stačí přehodit hodnoty x a z, jednu z nich vynásobit -1 a výsledek znormalizovat pomocí `Vector3.Normalize(...)`. Pozici kamery pak spočítáme jako součet původního pozorovacího bodu s lineárním násobkem kolmého vektoru:

$$\begin{gathered} Vstup: \boldsymbol{x_0}, \vec{v} \\ Hledáme\ \vec{s}: \vec{s} \in \rho_{xz} \wedge \vec{s} \perp \vec{v} \\ \vec{s} = [s_x, s_y, s_z] = [-v_z, v_y, v_x] \\ \boldsymbol{x_L} = \boldsymbol{x_0} - 0.035 \cdot \vec{s} \\ \boldsymbol{x_P} = \boldsymbol{x_0} + 0.035 \cdot \vec{s} \end{gathered} \tag{45}$$

Stínování

Jak již jsem zmínil, stínování se provádí hardwarově na grafické kartě pomocí tzv. *shaderu*. Shader je jednoduchý program, určený přímo pro grafickou kartu. Standardem pro tyto programy je vysokoúrovňový stínovací jazyk (high-level shading language, HLSL), který je syntaxí velmi podobný jazyku C.

Kód shaderu se skládá ze dvou hlavních částí: vertex-shaderu a pixel-shaderu. Již jejich názvy naznačují dopad jejich činnosti, vertex-shader pracuje s vrcholy, kdežto pixel-shader s jednotlivými pixely. Aby mohly spolupracovat, předává vertex-shader pixel-shaderu data, která „po cestě“ prochází přes interpolátor. Vstupním bodem do programu je definice techniky, která říká, který konkrétní vertex-shader a pixel-shader bude na vykreslovaný objekt použit.

Vertex-shader musí do pixel-shaderu předat alespoň polohu vrcholu, transformovanou do souřadnic *viewportu*[6], ale může předávat i další data, jako je normála ve vrcholu nebo intenzita osvětlení.

6 Pohledové okno kamery, pro jednoduchost si jej lze představit jako hledáček kamery.

Do pixel-shaderu přijdou předané hodnoty interpolované, interpolace se provádí po hranách a následně po jednotlivých řádcích obrazu, což velmi dobře vyhovuje požadavkům implementovaných stínovacích algoritmů. Povinným výstupem z pixel-shaderu je barva pixelu, která může (a nemusí) pokračovat do dalších částí grafické *pipeline*[7].

Data jsou do shaderu předávána jednak implicitně (program si o ně řekne ve vstupní datové struktuře vertex-shaderu), a také explicitně, příkazem `effect.SetValue(název_handleru, předaná_data)`. Druhým způsobem se předávají data, která jsou konstantní pro více běhů shaderu, tedy transformační matice a koeficienty materiálu. Příliš časté nastavování konstant aplikaci zpomaluje.

Můj shader implementuje tři základní stínovací algoritmy: Konstantní stínování, Gouraudovo stínování, které interpoluje barvu pixelu a Phongovo stínování, které interpoluje pozici a normálu ve vrcholu, a teprve v pixel-shaderu se z ní spočte barva pixelu.

Kompozice programu

Třída `MujMesh`

Jak bylo zmíněno a odůvodněno výše, pro reprezentaci stínovaného objektu používám separátní třídu `MujMesh`. Její konstruktor na základě předaných parametrů (odkaz na zařízení, druh objektu a pole číselných parametrů) vytvoří objekt typu `Mesh`, získá z něj počty vrcholů a trojúhelníků, dále získá data z vertex-bufferu a z index-bufferu, převede je na sekvenci trojúhelníků a vloží je do vlastní instance vertex-bufferu. Doplňkový normálový vektor je normalizovaným součtem normálových vektorů ve vrcholech, doplňkový referenční bod je těžiště trojúhelníka.

Po dokončení potřebných výpočtů v konstruktoru je objekt `Mesh` zničen a odklizen

7 Doslovný překlad zní „roura," na jednom „konci" je geometrický objekt v prostoru, na druhém jeho obraz na stínítku obrazovky.

(`Mesh.Dispose()`). Po celou dobu existence instance třídy se uchovává vertex-buffer a počet trojúhelníků.

Struktura VertexPosNormPosNorm

Obsahuje definici vlastního formátu vrcholu s jedním vektorem pro prostorové souřadnice bodu, jedním vektorem pro normálu příslušnou danému vrcholu a dvěma doplňkovými vektory s referenčním bodem a referenční normálou, které uchovávají informace pro konstantní stínování. Ve třídě je také obsažena definice uspořádání dat a jejich významu pro shadery.

Třída Stinovani

Obsahuje aplikační logiku programu, zajistí vytvoření a inicializaci potřebných objektů a řídí proces renderingu. Je-li aktivován stereoskopický režim, zajišťuje přepínání mezi virtuálními kamerami, které renderují obraz pro pravé a levé oko.

Třída RidiciOkno

Definuje řídicí okno pro výběr vykreslovaného objektu a pro zadávání souvisejících parametrů. Po kliknutí na příslušné tlačítko se pokusí vybrat data ze vstupních polí, pokud se zdaří, zavolá se zjištěnými parametry metodu `Stinuj(...)` ve třídě Program.

Třída Program

Řídí běh programu. Tvoří řídicí okno aplikace a na požádání také novou instanci třídy Stínování. Při spuštění zkontroluje, zda přítomné DirectX zařízení má schopnosti Shader Modelu 2.0.

Shadery – soubor bpini_efekty.fx

Shadery jsou zřejmě nejdůležitější (anebo druhou nejdůležitější) součástí tohoto programu, dodávají mu požadovanou funkčnost.

Pro *konstantní stínování* z programu do shaderu implicitně vstupují prostorové souřadnice vrcholu a doplňková data - referenční bod trojúhelníka (`POSITION1`) a jeho referenční normála (`NORMAL1`). Z nich se po transformaci do souřadného systému kamery (viz výše) spočítá barva – intenzita osvětlení daného vrcholu. Díky předanému referenčnímu bodu trojúhelníka máme zaručeno, že barva bude za všech okolností na celé ploše trojúhelníka stejná.

Poloha vrcholu se ze „světových“ souřadnic do souřadnic viewportu transformuje postupným vynásobením světovou, pohledovou a projekční maticí. Transformované souřadnice vrcholu jsou spolu se spočítanou barvou pixelu předány do pixel-shaderu, který pouze přiřadí barvu na výstup. Protože všechny tři vrcholy mají stejnou barvu, interpolace ji nezmění.

V případě *Gouraudova stínování* vstupují do vertex-shaderu souřadnice vrcholu a jeho normálový vektor. Po transformaci vstupů do souřadného systému kamery se z nich (stejně jako v případě konstantního stínování) určí směrový vektor z bodu směrem ke světlu a k pozorovateli, které spolu s normálovým vektorem poslouží k výpočtu osvětlovacího modelu a barvy pixelu. Barva pixelu je předána do pixel-shaderu, kde je opět pouze přiřazena na výstup. Výpočet tedy proběhne pro každý trojúhelník jen třikrát, mezilehlé hodnoty se lineárně interpolují.

Teoreticky vzato, moje implementace konstantního stínování je speciálním případem Gouraudova stínování. Nenašel jsem totiž žádné schůdné a zároveň jednoduché řešení, kterak odpovídající barvu *hardwarově(!)* spočítat *jednou* a výsledek pouze použít v pixel-shaderu. Jistě by bylo možné barvu spočítat softwarově v programu a předat ji do shaderu přes `effect.SetValue(...)`, ale to by znamenalo mít stejný kód v programu na dvou místech, výpočet by byl také pomalejší – zkrátka nebylo by to efektivní.

Vertex-shader *Phongova stínování* má stejné vstupy jako vertex-shader Gouraudova

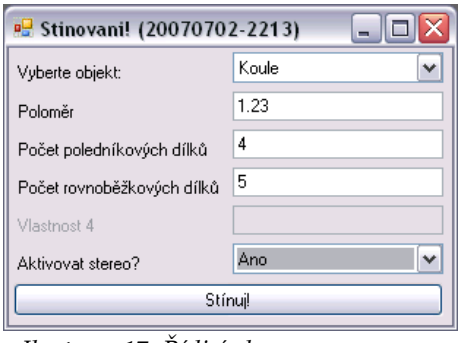
stínování – souřadnice vrcholu a normálu. Obě hodnoty jsou po převodu do souřadného systému kamery předány do pixel-shaderu, kde se na jejich základě spočítá osvětlovací model.

Dovolím si zde poznámku, která laskavému čtenáři jistě neunikla: U konstantního a Gouraudova stínování se výpočet barvy prováděl pro každý trojúhelník jen třikrát, vypočtené hodnoty se potom interpolovaly tak, aby rovnoměrně pokryly celý trojúhelník. Phongovo stínování barvu počítá pro každý výsledný pixel zvlášť, má tedy proti předcházejícím dvěma algoritmům mnohem vyšší výpočetní složitost – O(mn) kde m a n odpovídají základně a výšce ve vzorci pro výpočet plochy trojúhelníka.

Přeje-li si laskavý čtenář znát implementační detaily, dovolím si jej odkázat na zdrojové kódy.

Chování programu

Metoda `Main(...)` ve třídě Program vytvoří řídicí okno (instance třídy `Form1`), které zjistí od uživatele všechny potřebné informace.



Ilustrace 17: Řídicí okno programu pro demonstraci metod stínování. Zdroj: vlastní produkce.

Po kliknutí na tlačítko *Stínuj!* přečte řídicí okno hodnoty ze vstupních polí a zajistí vytvoření nové instance třídy `Stinovani`, která řídí kreslicí proces. V přípravné fázi, volané konstruktorem třídy `Stinovani()` se postupně provedou následující kroky:

- inicializace oken – výroba potřebného počtu celoobrazovkových oken, jejich aktivace, definice reakcí na události (stisknutí kláves, zavírání okna),
- inicializace Direct3D zařízení, výroba a přiřazení prezentačních parametrů

zařízení, výroba a přiřazení potřebného počtu *swap-chainů* zařízení,

- inicializace geometrie – výroba a umístění demonstračního objektu, definice materiálu, transformačních matic a konstantních parametrů kamery,

- inicializace shaderu – načtení FX souboru a zajištění jeho externí kompilace, navázání proměnných programu na proměnné shaderu

- další pomocné kroky a zobrazení okna.

Po zavolání metody `Run()` se začne provádět neblokující smyčka, spočívající ve výpočtu pozorovacího bodu, vyrenderování pohledů na scénu (`RenderViews()`) a zpracování událostí aplikace. Provádí se do té doby, než uživatel zmáčkne klávesu Esc a booleovská proměnná `quit` nabude hodnoty `true`.

Metoda pro renderování pohledů na scénu zajistí výpočet skutečného umístění kamery a renderování pohledu na scénu z daného bodu metodou `RenderScene()`. Demonstrační objekt je vykreslován po jednotlivých trojúhelnících, algoritmus lze shrnout do následujícího pseudokódu:

```
Pro všechny trojúhelníky prováděj:
      Má-li být efekt aktivní:
            Nastav parametry pro efekt;
            Vyber techniku efektu;
            Spusť efekt a jeho základní průchod;

      Vykresli trojúhelník (DrawPrimitives(...));

      Byl-li efekt aktivní:
            Ukonči efekt;
```

Po dokončení renderovacího procesu se provede zpracování událostí aplikace (`Application.DoEvents()`), v případě tohoto programu je to zpracování reakcí na zmáčknuté klávesy. Je-li stisknuta klávesa Esc, uloží se do řídicí proměnné neblokující smyčky `quit` hodnota `true` a dále se renderovat nebude, běh instance třídy `Stinovani` se ukončí a znovu se odkryje řídicí okno.

3.2.4 Funkčnost řešení

Funkčnost řešení jsem ověřil na objektech koule (`Mesh.Sphere`) a konvice (`Mesh.Teapot`). V obou případech stínovací algoritmy fungovaly dle očekávání a objekty stínované Phongovým stínováním vypadaly opticky nejlépe.

Zjistil jsem, že rychlost renderování objektu závisí na použité stínovací technice a na počtu trojúhelníků, které tvoří objekt.

Konvička (Utah Teapot) se 2256 trojúhelníky se na domácím testovacím počítači[8] vykreslovala velmi ochotně: Phongovo stínování kreslilo 340 snímků za sekundu, konstantní a Gouraudovo dokonce 480 snímků za sekundu. Na mnohem výkonnějším počítači[9] v laboratoři se stereostěnou jsem s Gouraudovým stínováním dosáhl dokonce 6800 snímků za sekundu.

Na objektu koule jsem zkoumal závislost snímkové frekvence na počtu trojúhelníků, které tvoří objekt. Výsledky jsou v sekci 4.1.

Ilustrace 18, 19 a 20 (shora dolů) ukazují konstantní, Gouraudovo a Phongovo stínování na čajové konvici, vymodelované na univerzitě v Utahu.

Zapnuté nebo vypnuté stereoskopické promítání se na celkové snímkovací frekvenci obou vykreslovacích ploch neprojevilo, ale hodnota na každém displeji bude pochopitelně poloviční.

8 AMD Duron 1.1 GHz, základní deska MSI-745 Ultra (SiS 745), 256 MB DDR-SDRAM, grafika nVidia GeForce 6200 (128 MB) s ovladači verze 6.14.10.9371, operační systém Windows XP

9 AMD Athlon 64 3000+, grafika nVidia GeForce 8800, operační systém Windows XP

3.3 Program pro demonstraci vlivu barevných systémů

3.3.1 Obecný popis

Jedná se o program, demonstrující použití různých rozptylových metod v různých barevných prostorech. Je to klasická „okenní“ klikací aplikace s přehledným menu, která uživateli dovoluje vybrat libovolnou kombinaci barevného prostoru a rozptylu a pozorovat výsledek.

3.3.2 Analýza

Myšlenkové úvahy vycházejí z premisy, že vstupem i výstupem je obraz – bitmapa. Při práci potřebujeme zpracovat každý pixel původního obrazu a vytvořit z něj pixel výstupního obrazu. K tomu potřebujeme provést následující kroky:

- převést pixel z bitmapy do vnitřní reprezentace z prostoru RGB
- převést bod do zvoleného cílového prostoru
- provést nad získaným bodem požadovanou operaci rozptylu
- výsledek převést zpět do RGB
- zapsat pixel do výsledného obrazu

Samozřejmě potřebujeme výše popsaný cyklus provádět velmi rychle a s minimálními paměťovými nároky, tedy bez zbytečných mezistavů a bez zbytečných datových struktur.

Jedním z dílčích úkolů tedy bylo navrhnout společnou datovou strukturu, dostatečně obecnou na to, aby pojala informace o všech použitých barevných prostorech a

zároveň dostatečně jednoduchou, aby s ní použité algoritmy dokázaly snadno pracovat. Jako nejvýhodnější se ukázalo použití struktury se čtyřmi čísly s plovoucí řádovou čárkou a veškeré hodnoty normovat na rozsah 0 až 1.

Původní plán počítal s individuálním cyklem a paměťovou strukturou pro každý jednotlivý transformační krok, ale program byl neúnosně pomalý a paměťově nenasytný. Proto bylo nezbytné aplikační logiku zjednodušit a zracionalizovat používání paměťových zdrojů. Úspory operací a paměti je dosahováno používáním statických proměnných – jakýchsi pevných registrů, které se vytvoří jednou při spuštění programu. Data se do nich přiřazují referencí – opět odpadá tvorba zbytečných datových struktur.

Při implementaci rozptylových metod jsem bral v potaz také barevný prostor, ve kterém rozptyly provádím. Jak jsem se v průběhu vývoje přesvědčil, u barevných prostorů s oddělenými složkami jasu a barvy je nutno prahování provádět jen na složce obsahující jas a složky s barvami nechat na pokoji.

3.3.3 Realizace

Program je koncipován podle *třívrstvé aplikační architektury* s důsledným oddělením tzv. programových vrstev – podprogramů, které nějakým způsobem logicky patří k sobě. Programy, realizované s tímto myšlenkovým přístupem, jsou vnitřně rozděleny na vrstvu datovou, která načítá a ukládá data, vrstvu aplikační, obsahující aplikační logiku a vrstvu prezentační, která zpracované výsledky předvede uživateli.

Také je v rámci možností důsledně aplikován objektově-orientovaný přístup, který značně zpřehledňuje a zjednodušuje kód. Při pojmenovávání tříd a identifikátorů jsem se snažil používat výstižné české názvy, podle potřeby oddělené velkými písmeny.

Pro jednoduchost a myšlenkovou přímočarost jsem program implementoval pouze

s využitím prostředků standardní verze prostředí .NET, tedy zejména jmenných prostorů `System.Drawing` a `System.Windows.Fonts` a veškeré operace jsou implementovány softwarově. Softwarovou implementaci kompenzuji důslednou racionalizací zacházení s pamětí. Kde je to možné, používám sdílené statické proměnné a parametry a výsledné hodnoty předávám odkazem.

Dovolím si laskavého čtenáře, pokud si přeje znát konkrétní implementační detaily, odkázat na zdrojový kód.

Datové struktury

Třída `Float4`

`Float4` je základní pracovní objekt programu. Jak jeho název napovídá, obsahuje čtyři proměnné typu `float`. `Float` jsem zvolil proto, že nepotřebujeme žádnou závratnou přesnost výpočtů, takže nám stačí kratší, 32-bitová reprezentace.

Třída `Float4` také obsahuje užitečné statické utility, jako je součet a rozdíl složek dvou předložených proměnných Float4, také umí lineární násobek, který všechny čtyři složky vynásobí zadaným číslem, a konečně také vyrobit hlubokou kopii předloženého Float4. Poslední vlastnost je implementována, protože .NET u objektů, které nejsou základní datové typy, dokáže vytvořit pouze tzv. *mělkou kopii*, několik proměnných pak odkazuje na jednu a tutéž paměťovou „buňku," a změny, provedené v jedné složce, se tak automaticky odrazí ve všech proměnných, které na danou buňku odkazují.

Třída `Obrazek`

Tato třída obsahuje dvě proměnné typu `Bitmap` (namespace `System.Drawing`) pro uchovávání původního a zpracovaného obrazu. Kromě nich jsou zde také pomocné proměnné typu Float4, které mají funkci pracovních registrů.

Třída stojí na pomezí mezi aplikační a prezentační vrstvou, kromě zmíněných proměnných s obrázky obsahuje metodu `ZpracujObrazek(...)`, která na základě dvou předložených parametrů, identifikátorů barevného prostoru a rozptylu, provede transformaci původního obrazu do výsledného. Dá se o ní říci, že je jedním z hlavních stavebních prvků aplikační logiky programu.

Algoritmus, který zmíněná metoda provádí, se dá velmi zjednodušeně zapsat do následující posloupnosti kroků:

```
Podle předložených parametrů a velikosti vstupního obrazu urči
velikost výsledného obrazu;

Je-li potřeba, inicializuj pomocná pole;

Pro všechny řádky původního obrazu:
      Pro všechny pixely na řádce:
            Převeď pixel z obrazu do vnitřní reprezentace;
            Převeď bod do požadovaného barevného prostoru;
            Nad cílovým barevným prostorem proveď požadované rozmytí;
            Výsledek převeď zpět do RGB;
            Pixel(y) zapiš do výsledného obrazu;
```

Pomocné proměnné se tvoří a inicializují mimo cyklus, čímž se pochopitelně náročnost programu snižuje na minimum. Navíc s transformačními rutinami (budou popsány v oddíle Aplikační vrstvy) se pracuje pomocí referencí, netvoří se tedy stále nové a nové objekty. Tím odpadají nepříjemné prostoje při alokaci a dealokaci místa na paměťové haldě.

Datová vrstva

Datová vrstva programu je obsažena ve třídě `DatovaVrstva`, a stará se o načtení a uložení obrazů, k čemuž slouží příslušně pojmenované metody.

Aplikační vrstva

Do aplikační vrstvy patří kromě stejnojmenné třídy `AplikacniVrstva` také *statické* třídy `BarevneProstory` a `Rozptyly` se statickými metodami. Instance těchto tříd

se nevytváří, naopak mají po spuštění programu v paměti alokováno konkrétní místo.

Statická třída `BarevneProstory`

Třída obsahuje statické transformační metody pro převod bodu, daného ve vstupní proměnné `Float4` do požadovaného barevného prostoru. Vstupní a výstupní parametry jsou zásadně předávány referencí. Odpadá tak alokace a dealokace místa na paměťové haldě programu, protože reference odkazují na existující, inicializované objekty.

Implementovány jsou následující barevné transformace:

- Pixel (`System.Drawing.Color`) na RGB (a opačně)
  - Paměťové nároky:
    - Pixel na RGB: 1 × float na konstantu
    - RGB na Pixel: 6 × float, 3 × int
  - Výpočetní složitost:
    - Pixel na RGB: 3 × násobení
    - RGB na Pixel: 3 × násobení, 3 × zaokrouhlení, 3 × převod na int
- RGB na CMYK (a opačně)
  - Paměťové nároky:
    - RGB na CMYK: 2 × pomocný Float4, 1 × Float4 na konstantu
    - CMYK na RGB: 1 × Float4 + 1 × Float4 na konstantu

- Výpočetní složitost:
  - RGB na CMYK: 8 × sčítání, 4 × porovnávání, 3 × násobení, 1 × dělení
  - CMYK na RGB: 4 × sčítání, 3 × násobení
- RGB na HSV (a opačně)
  - Paměťové nároky
    - RGB na HSV: 1 × pomocný Float4, 1 × float
    - HSV na RGB: 1 × pomocnýFloat4, 4 × float
  - Výpočetní složitost
    - RGB na HSV: 11 × porovnání, 4 × násobení, 5 × sčítání, 2 × dělení
    - HSV na RGB: 7 × porovnání, 6 × násobení, 6 × sčítání
- RGB na CIE_{XYZ} (a opačně)
  - Paměťové nároky: Obojí: 3 × float
  - Výpočetní složitost:
    - RGB na CIEXYZ: 8 × násobení, 5 × sčítání
    - CIEXYZ na RGB: 9 × násobení, 6 × sčítání
- CIE_{XYZ} na CIE_{Lab} (a opačně)
  - Paměťové nároky: Obojí 2 × pomocnýFloat4, 1 × float

- Výpočetní složitost:
  - CIEXYZ na CIELab: 3 × porovnání, 9 × násobení, 6 × sčítání, 3 × mocnina
  - CIELab na CIEXYZ: 16 × násobení, 9 × sčítání, 8 × porovnání, 3 × mocnina

- RGB na YUV (a opačně)
  - Paměťové nároky: Obojí 1 × pomocnýFloat4, 1 × float
  - Výpočetní složitost:
    - RGB na YUV: 11 × násobení, 8 × sčítání
    - YUV na RGB: 6 × násobení, 6 × sčítání

Rozsahy barevných prostorů jsou, kvůli zjednodušení dalších algoritmů, normovány na rozsah [0; 1].

Třída `Rozptyly`

Obsahuje statické metody rozmývání. Implementovány jsou následující:

- Metoda konstantního prahu
  - Paměťové nároky: 2 × float na konstanty
  - Výpočetní složitost: nejvýše 8 × porovnání, 4 × přiřazení
- Rozptyl Floyd-Steinbergovou metodou a (podobně definovaná) chybová difuze
  - Paměťové nároky: 2 × pomocnýFloat4

    - Výpočetní složitost: nejvýše 20 × sčítání, nejvýše 16 × násobení, 4 × porovnání

- Rozptyl metodou adaptivního vkládání chyb s řídicí maticí 2 x 2
    - Paměťové nároky: pouze přiřazujeme, tedy minimální
    - Výpočetní složitost: nejvýše 20 × porovnání, nejvýše 16 × přiřazení
- Metoda vzorů (2x2)
    - Paměťové nároky: 1 × pomocnýFloat4
    - Výpočetní složitost: nejvýše 14 × porovnání, nejvýše 8 × přiřazení, nejvýše 2 × dělení

Všechny metody respektují význam barevných složek v různých barevných prostorech. V RGB, CIE_{XYZ} a CMYK se prahují všechny barevné kanály, v HSV se prahují pouze kanály S a V, protože právě v nich je obsažena informace o jasu barvy. Ze stejného důvodu se v CIE_{Lab} prahuje pouze L a v YUV pouze Y.

Metody Floyd-Steinberg, chybová difuze a metoda vzorů přijímají při volání jako jeden ze svých parametrů pomocné pole ze třídy `Obrazek`. V případě prvních dvou metod slouží pole k distribuci chyby do dosud nezpracovaných sousedních pixelů a má rozměry [2; šířka_obrazu]. Po dokončení zpracování jednoho řádku obrazu se prohodí v pomocném poli první řádek se druhým a nový spodní řádek se vynuluje.

V případě metody vzorů jsou do pole podle definice umístěny plné a nulové hodnoty, podle toho, zda si přejeme příslušný pixel rozsvítit, nebo nikoliv.

Třída `AplikacniVrstva`

Hlavní třída aplikační vrstvy v sobě obsahuje především sadu zpracovávaných obrázků, reprezentovanou jednorozměrným polem objektů `Obrazek`. Dále obsahuje

metody, reagující na požadavky prezentační vrstvy, které mají vztah k aplikační vrstvě a k datové vrstvě, na kterou má aplikační vrstva odkaz.

Pro prezentační vrstvu má největší význam metoda `zpracujPozadavekNaTransformaciObrazku(String druhPozadavku, byte id_cile)`, která vyhodnotí požadavek na transformaci obrazů a zajistí jeho vykonání a vrácení zpracovaných obrazů prezentační vrstvě. Dále je přítomna metoda `NactiObrazky(String[] nazvySouboru)` pro načtení obrazů do prezentační vrstvy a metoda `ZaridUlozeniObrazku(String[] nazvySouboru, Bitmap[] predaneObrazky)` pro uložení obrazů do souborů na disku.

Prezentační vrstva

Stejnojmenná třída obsahuje deklaraci oken programu – hlavního okna a jednoho nebo dvou celoobrazovkových oken, dále jsou obsaženy metody pro řízení obsahu celoobrazovkových oken, metody pro spolupráci s aplikační vrstvou a metody pro zobrazení souborových dialogů a hlášek programu.

Konstrkuktor třídy PrezentacniVrstva provede, kromě inicializace používaných objektů, také spuštění *okenní části* programu.

Třída `HlavniOknoProgramu`

Obsahuje definici grafického uživatelského rozhraní programu, včetně definice reakcí na události, a zřejmě je tak jeho nejkomplexnější součástí.

Transformaci obrazů podle právě vybraných parametrů zajišťuje metoda `zaridZpracovaniObrazku(String druhPozadavku, byte id_cile)`, kterou po kliknutí zavolají relevantní položky menu, samozřejmě s relevantním parametrem.

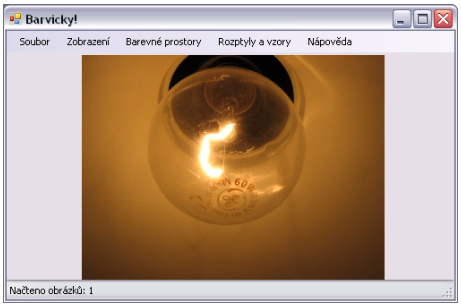
Na kliknutí je také zařízeno načítání a ukládání souborů. Relevantní metody napřed zjistí prostřednictvím souborových dialogů z prezentační vrstvy názvy souborů, a jsou-li všechny názvy neprázdné, zařídí zavolání metody `nactiObrazky(String[] nazvySouboru)`, respektive `ulozObrazky(String[] nazvySouboru)`.

Obrazy jsou zobrazovány v komponentách `PictureBox[] picBoxy`, které jsou vždy v okně umístěny dva, přičemž jeden z nich může být skrytý – to nastává v případě práce s jediným obrázkem a v případě „simulované stereoskopie“ při práci se dvěma obrázky, kdy na tutéž plochu vykreslujeme dva obrazy.

Při požadavku na uložení souborů jsou ukládány právě ty obrazy, které jsou právě v daném okamžiku zobrazeny v okně programu.

Chování programu

Spouštěč, třída Program, vytvoří instance tříd `DatovaVrstva`, `AplikacniVrstva` a `PrezentacniVrstva` a zajistí jejich vzájemné propojení. Prezentační vrstva vytvoří všechny používaná okna a spustí okenní aplikaci, uživateli se zobrazí okno.



Ilustrace 21: Snímek okna programu Bavicky!
Zdroj: vlastní produkce.

Uživatel zadá v menu otevřít soubory. Jeho požadavek je propagován do aplikační vrstvy, která z datové vrstvy načte požadovaný počet obrázků, naplní vhodným způsobem své datové struktury a načtené obrázky okamžitě vrátí hlavnímu oknu.

Po kliknutí na relevantní položky v menu programu je do aplikační vrstvy poslán požadavek na provedení transformace. Transformace se provede tak, že se u všech existujících obrazů v příslušné instanci třídy `Obrazek` zavolá metoda

`zpracujObrazek(int cilovyProstor, int cilovyRozptyl)`. Obrazy, vzniklé transformací, jsou vráceny zpět do hlavního okna ke zobrazení.

Byl-li načten jeden obraz, zobrazuje se pouze jeden obraz. Jestliže byly načteny dva obrazy, lze je v okně zobrazovat buď vedle sebe, anebo přes sebe – pracovně to nazývám „simulovaná stereoskopie," k jejímu dosažení jsou hodnoty barev u každého pixelu sečteny a vynásobeny jednou polovinou.

Stereoskopie

Program disponuje celoobrazovkovým výstupem na dvě zařízení v prostředí Microsoft Windows. To odpovídá technickým požadavkům stereostěny v grafické laboratoři katedry informatiky a výpočetní techniky ZČU v Plzni.

Jsou-li k dispozici dvě grafická zařízení (dva monitory) a jsou-li zároveň zobrazovány dva obrázky vedle sebe, je na jednu zobrazovací plochu promítnout jeden obrázek a na druhou druhý. Je-li v okně zobrazován jen jeden obrázek (včetně případu „simulované stereoskopie"), bude zobrazen na obou zařízeních.

4 Experimenty

Pro experimentální část bakalářské práce jsem si stanovil úkol, prozkoumat, jakým způsobem závisí výsledná snímkovací frekvence na počtu trojúhelníků, kterými je vykreslovaný a stínovaný objekt tvořen.

Také jsem prozkoumal vliv použitých barevných prostorů na metody ditheringu v oblasti dvojrozměrné grafiky, jakým způsobem se projeví vliv barevného prostoru ve srovnání se standardním prostorem RGB.

4.1 Vliv počtu trojúhelníků objektu na počet snímků za sekundu

4.1.1 Metodika a realizace testování

Za referenční objekt jsem si zvolil kouli a postupně jsem jí nastavoval vyšší a vyšší počty dílků ve směru os. Jako referenční stínovací model jsem si zvolil Phongovo stínování, protože tam se výpočet provádí pro každý pixel individuálně.

Testování jsem realizoval tak, že jsem opakovaně spouštěl program a postupně zvyšoval hodnotu definičních parametrů koule po násobcích dvaceti. Poloměr koule jsem ve všech případech ponechal roven jedné. Naměřenou hodnotu jsem odečetl z výpisu v levém horním rohu obrazovky, uvažoval jsem nejčastější hodnotu.

4.1.2 Výsledky měření

Měření jsem prováděl na referenčním počítači[10]. Naměřené hodnoty jsem zaznamenal do následující tabulky.

Počet dílků koule v každém směru	20	40	60	80	100	120	140	160	180	200
Počet trojúhelníků	760	3120	7080	12640	19800	28560	38920	50880	64400	N/A
FPS	340	300	260	230	200	160	135	110	90	N/A
Počet zpracovaných trojúhelníků za sec.	258400	936000	1.84 M	2.9 M	3.96 M	4.57 M	5.25M	5.6 M	5.8 M	N/A

První pohled nám naznačuje, že čím méně trojúhelníků je v zorném poli kamery, tím lépe. Na druhou stranu – tvrzení, že nejlepší trojúhelník je ten, který nemusíme kreslit se v tomto konkrétním případě příliš neprokázalo. I přesto, že je ve shaderu zapnuté ořezávání tak, aby se kreslily pouze trojúhelníky, které mohou být vidět – jejich normála svírá s pohledovým vektorem ostrý úhel, naměřené hodnoty jsou téměř stejné, jako v případě vypnutého ořezávání.

Snímková frekvence klesá pomalejším tempem než jakým roste počet trojúhelníků objektu. To je samozřejmě dobře, a lze to zdůvodnit pomalým růstem časové režie karty při přípravě kreslení.

Konečně ze zpomalujícího se růstu počtu zpracovaných trojúhelníků za sekundu můžeme usoudit, že se testovací systém blížil k jisté kritické mezi. Tento úsudek ale nemůžeme exaktně prověřit bez určitého zásahu do programu. Konstrukce detailnějších koulí totiž ztroskotala na rozměrech interního pole sousednosti vrcholů ve struktuře Mesh. Jedním z možných řešení by bylo vykreslování více objektů. V každém případě lze očekávat, že s dalším růstem počtu zpracovávaných trojúhelníků bude efektivita dále klesat.

10 AMD Duron 1.1 GHz, základní deska MSI-745 Ultra (SiS 745), 256 MB DDR-SDRAM, grafika nVidia GeForce 6200 (128 MB) s ovladači verze 6.14.10.9371, operační systém Windows XP

4.2 Vliv barevných prostorů na metody rozmývání ve 2D grafice

Metody rozmývání implementované v předloženém programu pracují s prahovými hodnotami v nějakém uvažovaném barevném prostoru. Prahovat lze všechny barevné kanály, anebo pouze ty relevantní, které obsahují informaci o jasu barvy.

Prahujeme-li všechny kanály, získáme po převodu zpět do RGB určitou množinu barev, a tedy každý pixel výsledného obrazu bude mít jednu z těchto barev. Prahujeme-li pouze relevantní kanály, mohou být získané RGB barvy v celém spektru.

4.2.1 Metody a průběh testování

Nejprve jsem experimentálně zjistil, na jaké RGB barvy se převedou všechny možné prahové barvy(např. trojice [0, 0, 0], [0, 0, 1], [0, 1, 0] atd.) v každém barevném prostoru. Tím jsem dostal sadu 20 různých RGB barev, které se mohou vyskytnout v obrazech při použití rozmývacích metod. Pak jsem napsal jednoduchý program, který v předloženém obraze získané barvy hledá.

Pro druhý způsob rozmývání, kdy bereme v úvahu význam jednotlivých složek, jsem program doplnil o výpočet histogramů.

Obrazy ke zkoumání jsem vytvořil ve svém programu *Barvičky!*. Data získaná prahováním všech kanálů v prostorech CIE_{Lab}, HSV a YUV byla naměřena ve vývojové verzi programu, která ještě nezohledňovala význam jednotlivých složek.

Realizace

Jak jsem už zmínil, pro analýzu obrazů jsem si vyrobil pomocný program, který hledá v obraze předdefinované prahové hodnoty. Provádí následující algoritmus:

```
Pro všechny řádky obrazu:
      Pro všechny pixely:
            Získej hodnotu pixelu;
            Sekvenčně projdi pole předdefinovaných barev a porovnávej;
            Jestliže se barva pixelu rovná předdefinované na pozici k
                  četnosti[k]++; // Zvětš hodnotu na pozici k
                                 // v poli četností předdefinovaných barev
            Jinak                // je nedefinovaná
                  četnosti[20]++;
Vytvoř textový soubor pro zápis výsledků;
Pro všechny předdefinované barvy:
      Zapiš do souboru hodnoty barvy, naměřenou četnost a relativní četnost
```

Program také umí vytvořit histogram předloženého obrazu a rozdílový histogram pro dvojici předložených obrazů. Při tom provádí následující algoritmus:

```
Pro všechny řádky obrazu:
      Pro všechny pixely obrazu:
            Získej pixely na pozici [i,j] v obou předložených obrazech;

            histogramR[pixel.R]++;
            histogramG[pixel.G]++;
            histogramB[pixel.B]++;

            Spočti jejich rozdíl;
            Normuj rozdíl pixelů na rozsah [0; 255];
            Zapiš znormovaný rozdíl do výsledné bitmapy;

Zapiš výslednou rozdílovou bitmapu na disk;
Vytvoř textový soubor pro zápis výsledků;
Pro všechny úrovně [0 .. 255]
      Zapiš do souboru úroveň intenzity, počet červených,
            zelených a modrých pixelů s touto intenzitou
            a poměrné zastoupení v obraze.
```

Oba dva podprogramy zapisují výsledky analýz na disk, používá se textový formát CSV s hodnotami oddělenými středníky. Získané soubory se pak dají snadno načíst do tabulkového procesoru.

Zajímavým (ale do jisté míry „vedlejším“) produktem je *rozdílová bitmapa*, která vizualizuje rozdíl předložených souborů. Oblasti, ve kterých se obrazy zcela shodují, mají šedivou barvu (R, G, B = 128). Tam, kde se obrazy liší, se úměrně velikosti rozdílu zmenšuje, resp. zvětšuje intenzita příslušné barevné složky.

Uživatelský manuál

Analyzér bitmap má dva pracovní režimy, mezi kterými se volí pomocí přepínačů **-h** pro tvorbu histogramů a **-p** pro analýzu prahových hodnot. Za tímto přepínačem se jako další parametry předávají názvy jednoho nebo dvou obrazových souborů. Syntaxe spouštěcího příkazu vypadá takto:

```
analyzer.exe {-h | -p} soubor_1 [soubor_2]
```

Rozdílová bitmapa (jen při tvorbě histogramů) se zapíše do souboru, jehož název je dán složením řetězců „DIFF-“ + název prvního souboru + „-“ + název druhého souboru + „.BMP.“ Zprávy z analýzy se zapíší do souborů s názvy „HIST-“ + název prvního souboru [+ „-“ + název druhého souboru] + „.CSV,“ resp. „PRAHY-“ + název prvního souboru [+ „-“ + název druhého souboru] + „.CSV.“

Interpretace předložených výsledků

Výsledky jsou reprezentovány dvojí formou. Jestliže prahujeme *pouze relevantní* barevné kanály, může zpětným převodem do RGB vzniknout téměř jakákoli barva. Proto uvádím barevné histogramy pro každý barevný kanál.

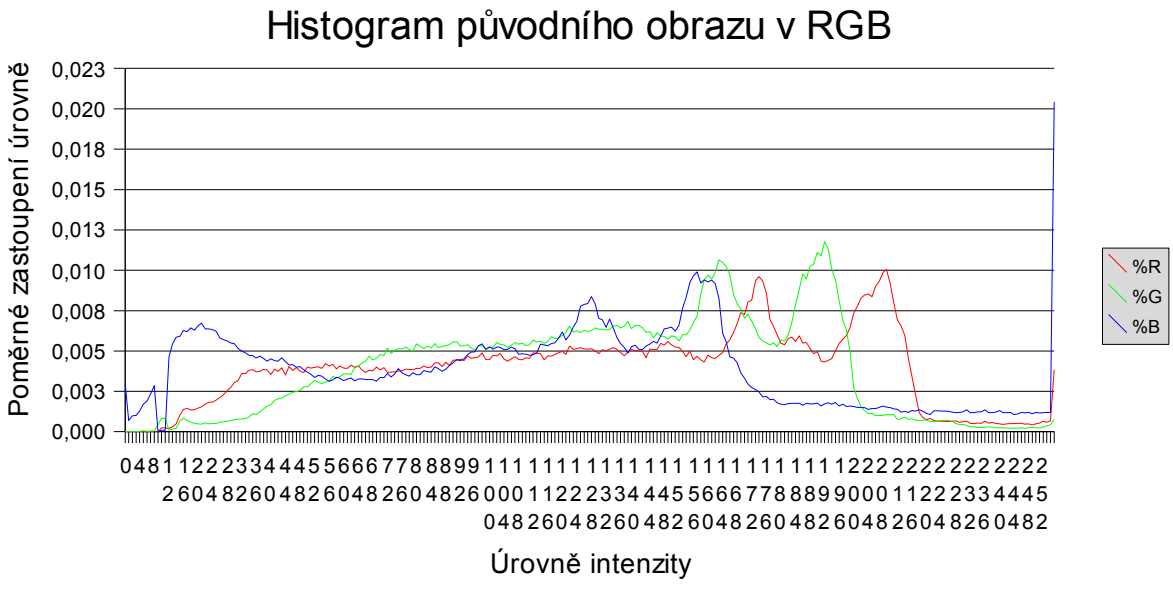
Při měření jsem ale zjistil, že se hodnoty v histogramech velmi často pohybují těsně kolem nuly a pouze v krajních bodech „vystřelí“ na hodnotu několika desítek procent zastoupení. Proto si nejméně zajímavé histogramy dovolím vynechat a uvedu pouze ty nejzajímavější, jejich absenci ale budu suplovat dostatečným komentářem.

V případech, kdy prahujeme *všechny* barevné kanály, dostaneme celkem 20 různých prahových barev. Výsledky této varianty zobrazuji sloupcovými grafy. V grafech jsou vedle sebe n-tice sloupců, každý z nich zastupuje jednu prahovou barvu, přesný popis je v legendě u každého grafu. Výška sloupce odpovídá zastoupení té které prahové barvy v transformovaném obraze.

Zdrojová data

Všechny experimenty jsem prováděl na referenční fotografii `Referencni_01.jpg`, jejíž jsem autorem. Výsledky transformací jsou na přiloženém CD.

Ilustrace (22): Vzorová fotografie pro výzkum vlivu barevných prostorů na metody rozmývání. Zdroj: vlastní fotografie.



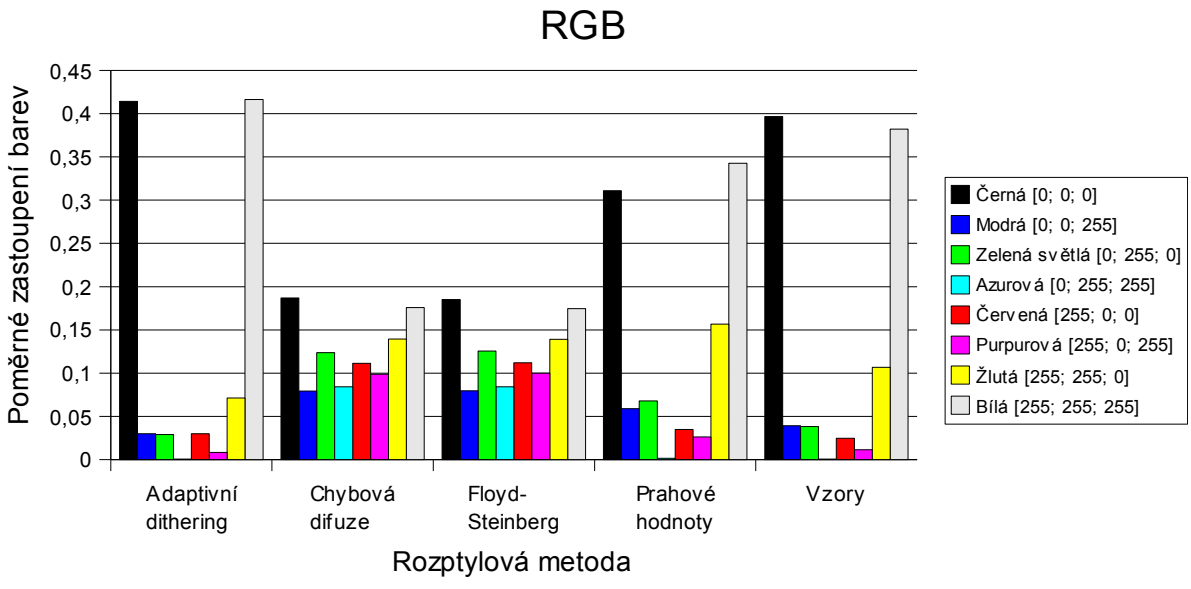
4.2.2 Barevný prostor RGB

Prahové hodnoty v prostoru RGB si můžeme představit jako vrcholy krychle na obrázku 5 na straně 17 (znázornění barevného prostoru RGB pomocí jednotkové krychle). Prahových barev je celkem 8:

- Černá [0; 0; 0],
- Modrá [0; 0; 255],
- Zelená [0; 255; 0],
- Azurová [0; 255; 255],
- Červená [255; 0; 0],
- Purpurová [255; 0; 255],
- Žlutá [255; 255; 0],
- Bílá [255; 255; 255].

Statistiku relativní četnosti prahových barev v různých rozmývacích algoritmech jsem zachytil do následujícího grafu. Lze z něj vyčíst, že u adaptivního ditheringu, metody prahových hodnot a metody vzorů mají velký důraz černá a bílá. Obrazy získané metodou Floyd-Steinbergovy distribuce chyby a metodou chybové difuze jsou proti nim barevnější.

V grafech jsem neuvažoval variantu bez použití rozptylové metody, na její vizualizaci by byly vhodnější histogramy jednotlivých barevných kanálů. Zjistil jsem však, že pouhá aplikace barevného prostoru obraz nijak nezmění, a proto nemá smysl zahrnovat do práce několik sad zcela shodných histogramů.

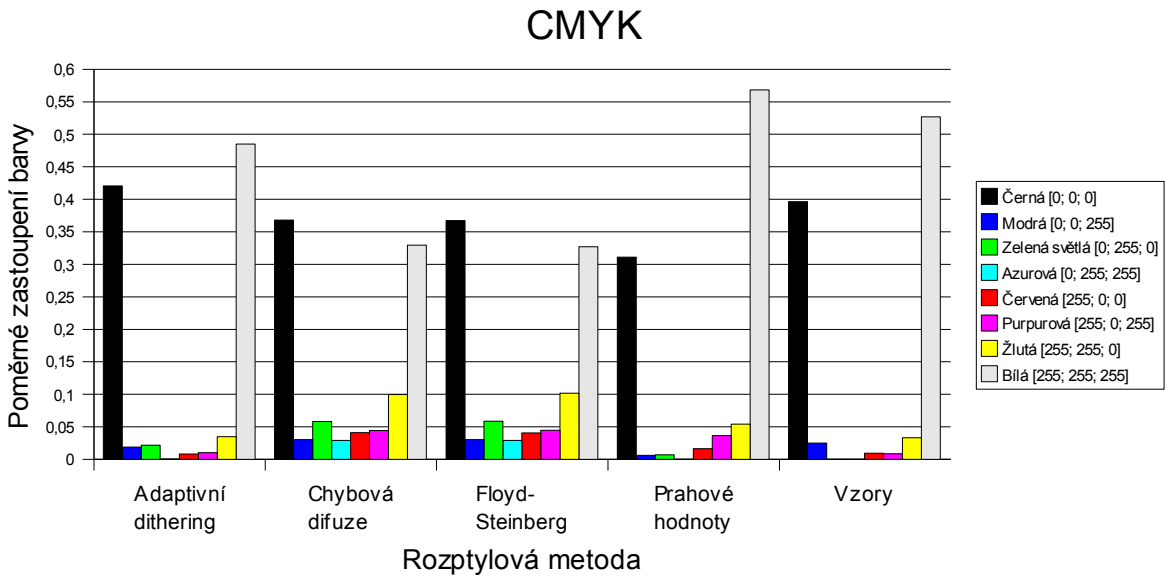


4.2.3 Barevný prostor CMYK

V barevném prostoru CMYK pracujeme se čtveřicemi barev. Jestliže každá složka nabývá prahových hodnot 0 a 1, dojdeme k 16 možným „prahovým barvám“ v prostoru CMYK. Po převodu do RGB se jejich počet sníží na 8 a odpovídají prahovým barvám v prostoru RGB.

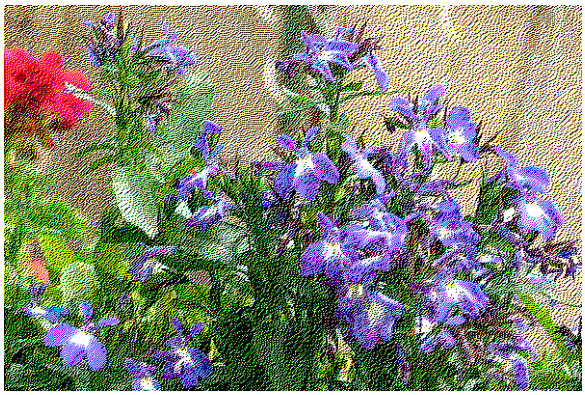
C	M	Y	K	RGB
0	0	0	0	Bílá [255; 255; 255]
0	0	0	1	Černá [0; 0; 0]
0	0	1	0	Žlutá [255; 255; 0]
0	0	1	1	Černá [0; 0; 0]
0	1	0	0	Purpurová [255; 0; 255]
0	1	0	1	Černá [0; 0; 0]
0	1	1	0	Červená [255; 0; 0]
0	1	1	1	Černá [0; 0; 0]
1	0	0	0	Azurová [0; 255; 255]
1	0	0	1	Černá [0; 0; 0]
1	0	1	0	Zelená světlá [0; 255; 0]
1	0	1	1	Černá [0; 0; 0]
1	1	0	0	Modrá [0; 0; 255]
1	1	0	1	Černá [0; 0; 0]
1	1	1	0	Černá [0; 0; 0]
1	1	1	1	Černá [0; 0; 0]

Z grafu dole je vidět, že u všech rozptylových metod dominují černá a bílá, ostatní barvy je pouze doplňují. Přesto lze původní obraz ještě snadno identifikovat. Na následující straně jsou výsledné obrazy z prostoru CMYK, výsledky z dalších barevných prostorů jsou na přiloženém CD.



(23)

(24)



(25)

(26)

(27)

(28)

Ilustrace 23 – 28 ukazují vliv barevného prostoru CMYK na metody rozmývání.
23 – bez rozmývání, 24 – metoda konstantního prahu, 25 – Floyd-Steinberg,
26 – chybová difuse, 27 – metoda vzorů, 28 – adaptivní dithering.

V obrazové příloze najdete tytéž obrázky rozmývány v obvyklém prostoru RGB,
data z ostatních barevných prostorů naleznete na přiloženém CD.
Zdroj: vlastní produkce (všechny).

4.2.4 Barevný prostor HSV

Varianta s prahováním relevantních barevných kanálů

Prahoval jsem složky S a V, ve kterých je obsažena informace o jasu barvy, složku H jsem ponechal beze změny, neboť nese informaci o barvě.

Ze všech tří barevných prostorů, ve kterých jsem prahoval pouze relevantní barevné složky, vyšel v HSV relativně nejlépe algoritmus chybové difuze a Floyd-Steinbergův algoritmus. Viz obr. 29 na této straně nahoře.

Ilustrace 29: Floyd-Steinbergův rozptyl na relevantních barevných kanálech v barevném prostoru HSV.

Za zmínku stojí také adaptivní dithering (viz obr. 30), v jehož histogramu jsou nenulové hodnoty pouze na pozicích násobků šesti (a samozřejmě na intenzitě 255). Celkově vyšel obraz celkem nepřirozeně: Zelené listy rostlin sice zůstaly zelené, červený květ na levé straně lze také ještě identifikovat, ale to je asi vše. Je to důsledek způsobu, jakým probíhá transformace při aplikaci adaptivního vzoru.

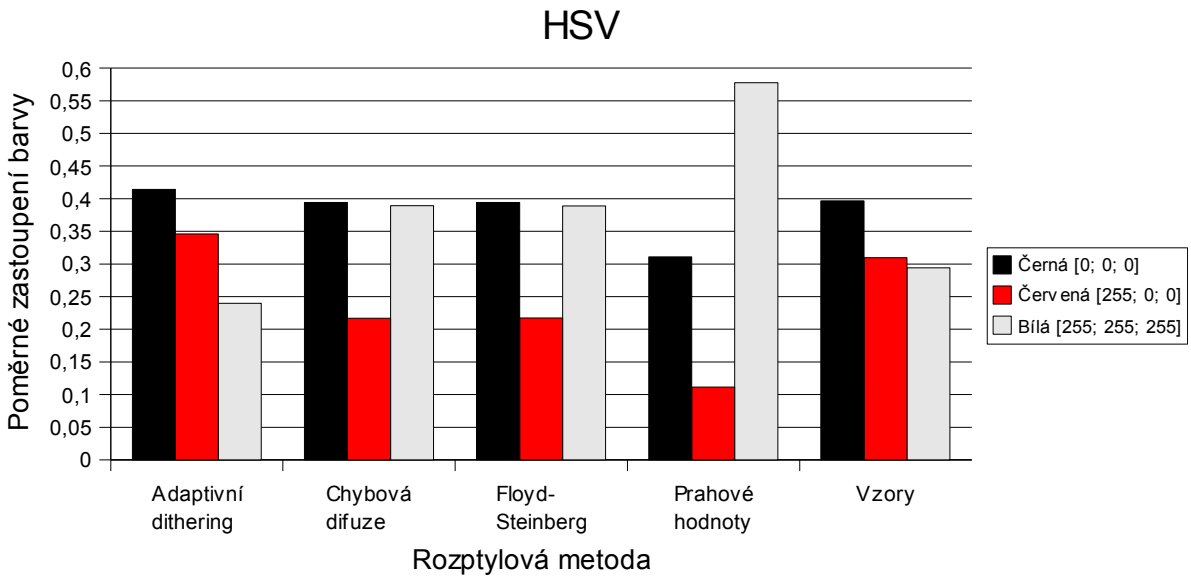
Ilustrace 30: Adaptivní dithering na relevantních barevných kanálech v barevném prostoru HSV. Zdroj: vlastní produkce (oba).

Varianta s prahováním všech barevných kanálů

V dále analyzovaných barevných prostorech pracujeme se trojicemi hodnot, z nichž každá může nabývat prahovou hodnotu 0 nebo 1.

H	S	V	RGB
0	0	0	Černá [0; 0; 0]
0	0	1	Bílá [255; 255; 255]
0	1	0	Černá [0; 0; 0]
0	1	1	Červená [255; 0; 0]
1	0	0	Černá [0; 0; 0]
1	0	1	Bílá [255; 255; 255]
1	1	0	Černá [0; 0; 0]
1	1	1	Červená [255; 0; 0]

Celkem dostaneme 3 unikátní prahové barvy v RGB, což je ze všech uvažovaných barevných prostorů nejméně. Při použití algoritmu prahových hodnot dominuje bílá, u ostatních algoritmů černá. Červená barva má ve všech obrazech (mimo algoritmu prahových hodnot) také zřetelnou intenzitu.



4.2.5 Barevný prostor YUV

Varianta s prahováním relevantních kanálů

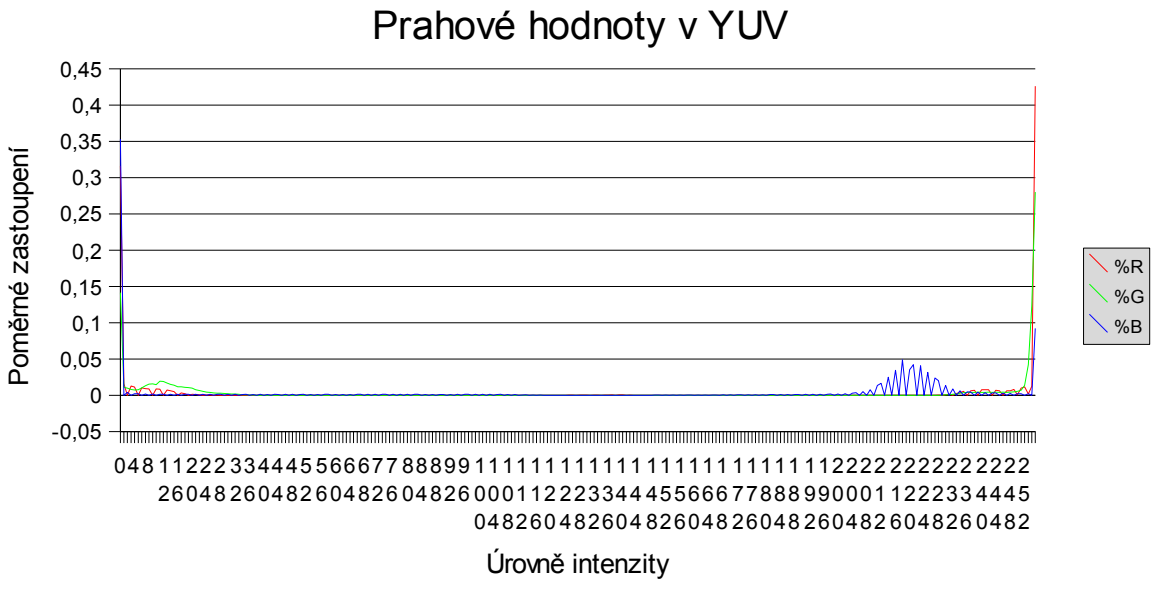
V prostoru YUV nesou informaci o barvě složky U a V, složka Y nese informaci o jasu, a právě tu jsem prahoval.

Floyd-Steinbergův algoritmus (a také podobně definovaná chybové difuse) vyšly nepříliš výrazně, naproti tomu při prahování všech barevných kanálů vyšel dosti výrazně, ale také dosti do zelena. Viz obr. 31 a 32.

Algoritmus adaptivního rozptylu opět působí mimozemským dojmem s převládajícími odstíny zelené a růžové.

Ilustrace 31 (nahoře):
Chybová difuse s prahováním kanálu Y.

Ilustrace 32 (dole):
Prahování všech tří kanálů.
Zdroj: vlastní produkce (oba).

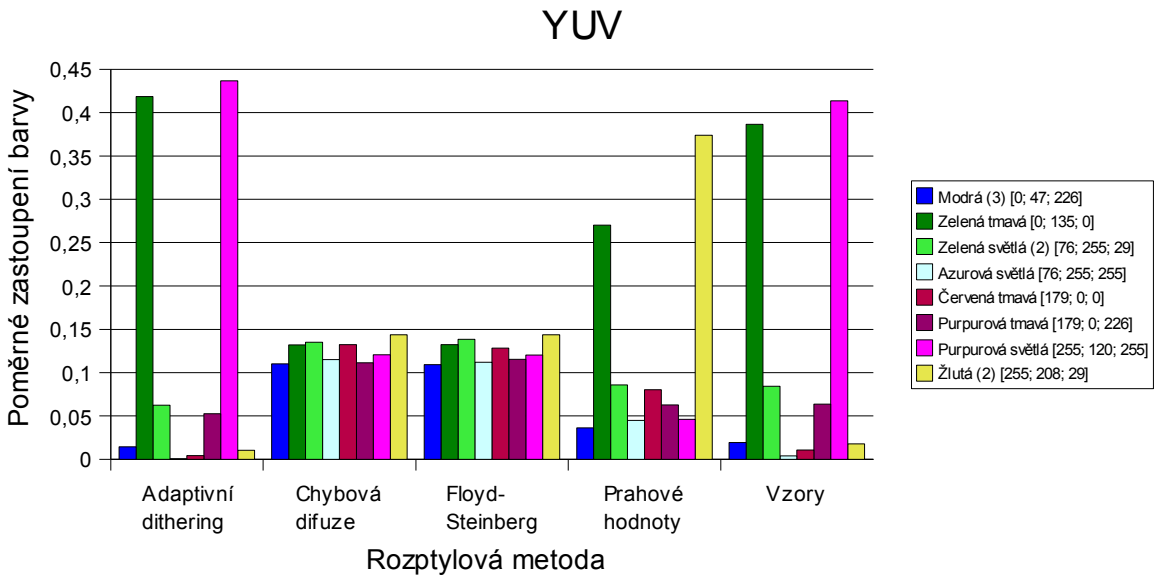


Varianta s prahováním všech kanálů

V barevném prostoru YUV se prahové hodnoty transformují na následující barvy:

Y	U	V	RGB
0	0	0	Zelená tmavá [0; 135; 0]
0	0	1	Červená tmavá [0; 179; 0]
0	1	0	Modrá [0; 47; 226]
0	1	1	Purpurová tmavá [179; 0; 226]
1	0	0	Zelená světlá [76; 255; 29]
1	0	1	Žlutá [255; 208; 29]
1	1	0	Azurová světlá [76; 255; 255]
1	1	1	Purpurová světlá [255; 120; 255]

Při adaptivním ditheringu a při metodě vzorů dostanou velký důraz tmavá zelená a světlá purpurová, u prahových hodnot jsou vysoké počty žlutých a tmavě zelených pixelů. Výsledky z Floyd-Steinbergova algoritmu a z chybové difuse jsou podobné, všechny prahové hodnoty v nich mají přibližně stejné zastoupení, ale vzhledem ke dvěma zeleným odstínům v prahových barvách „působí“ tyto obrazy zeleně.



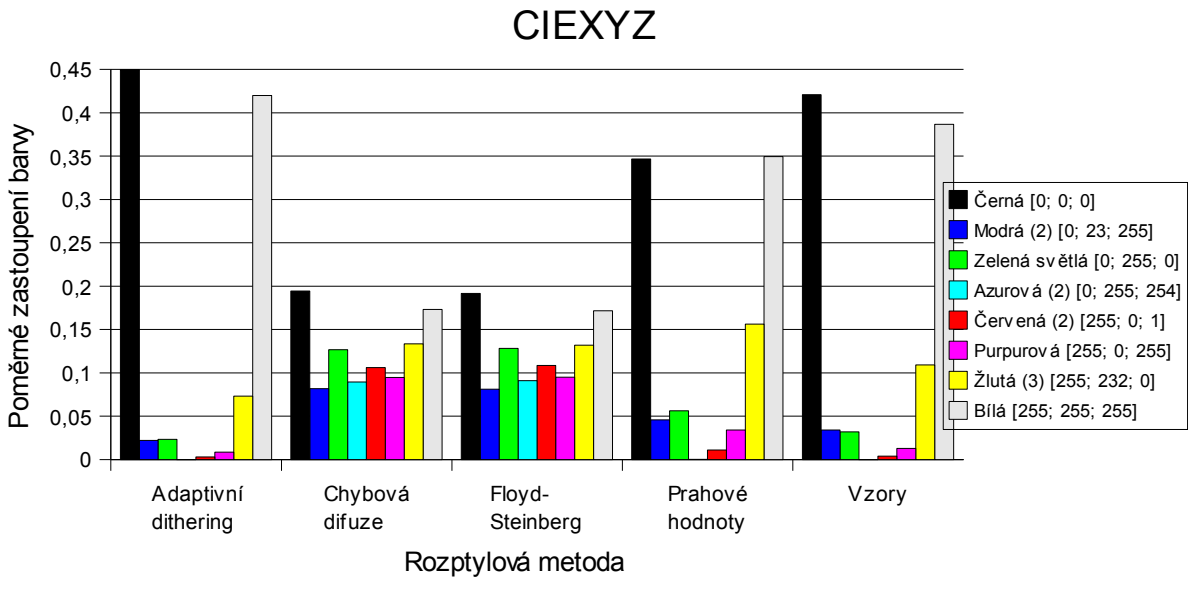
4.2.6 Barevný prostor CIE_{XYZ}

V prostoru CIE_{XYZ} se prahové hodnoty transformují takto:

X	Y	Z	RGB
0	0	0	Černá [0; 0; 0]
0	0	1	Modrá [0; 23; 255]
0	1	0	Zelená [0; 255; 0]
0	1	1	Azurová [0; 255; 254]
1	0	0	Červená [255; 0; 1]
1	0	1	Purpurová [255; 0; 255]
1	1	0	Žlutá [255; 232; 0]
1	1	1	Bílá [255; 255; 255]

Na první pohled překvapivé hodnoty u azurové a červené vznikly proto, že transformací z XYZ do RGB vznikají „teoretické“ barvy s hodnotami mimo povolený rozsah. V případě červené jsme se dostali k RGB trojici [2.36431; -0.51517; 0.0052]. Nejjednodušší ošetření tohoto případu je položit příslušnou hodnotu rovnou nejbližší hranici, tedy [1; 0; 0.0052]. K hodnotám v tabulce jsme potom dospěli transformací na rozsah <0; 255>.

Ve všech algoritmech, vyjma Floyd-Steinbergova a chybové difuze, mají dominanci černá a bílá, silnou pozici má také žlutá barva.



4.2.7 Barevný prostor CIE_{Lab}

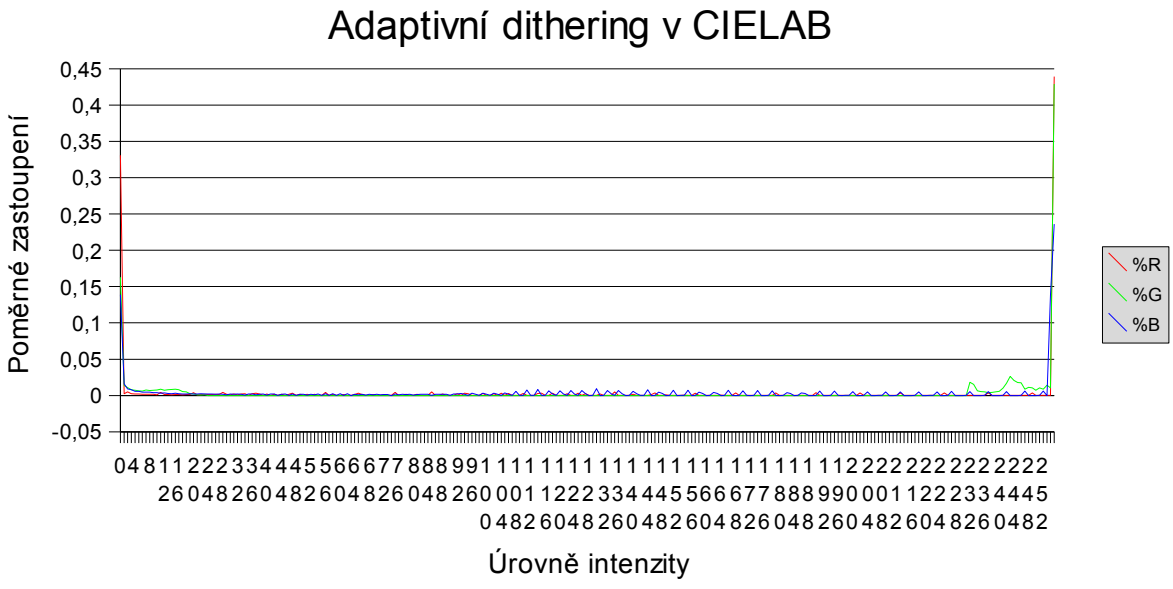
Varianta s prahováním relevantních barevných kanálů

Informaci o jasu barvy v tomto barevném modelu nese složka L, zbylé dvě nesou informaci o barevném odstínu. Proto prahujeme pouze složku L.

Proti variantě s prahováním všech kanálů dostaneme mnohem kvalitnější výstupy. Zatímco obrazy prahované ve všech třech složkách se utápí v zelené barvě a jí podobných odstínech, zde dostáváme výsledky, které se mnohem více podobají původnímu obrazu. Stejně jako u CIE_{XYZ} i zde dochází ke ztrátě části barevné informace, protože se u některých složek můžeme dostat mimo rozsah RGB barev.

Opět si dovolím demonstrovat výsledek adaptivního ditheringu, který ani tentokrát nezklamal a „uchvátil“ moji pozornost. Totéž ale nelze říci o dalších algoritmech: Floyd-Steinberg a chybová difuse jsou velmi nevýrazné, zcela se ztratila informace o červeném odstínu květiny na levém okraji snímku.

Ilustrace 33: Adaptivní dithering v prostoru CIELab s prahováním složky L. Zdroj: vlastní produkce.

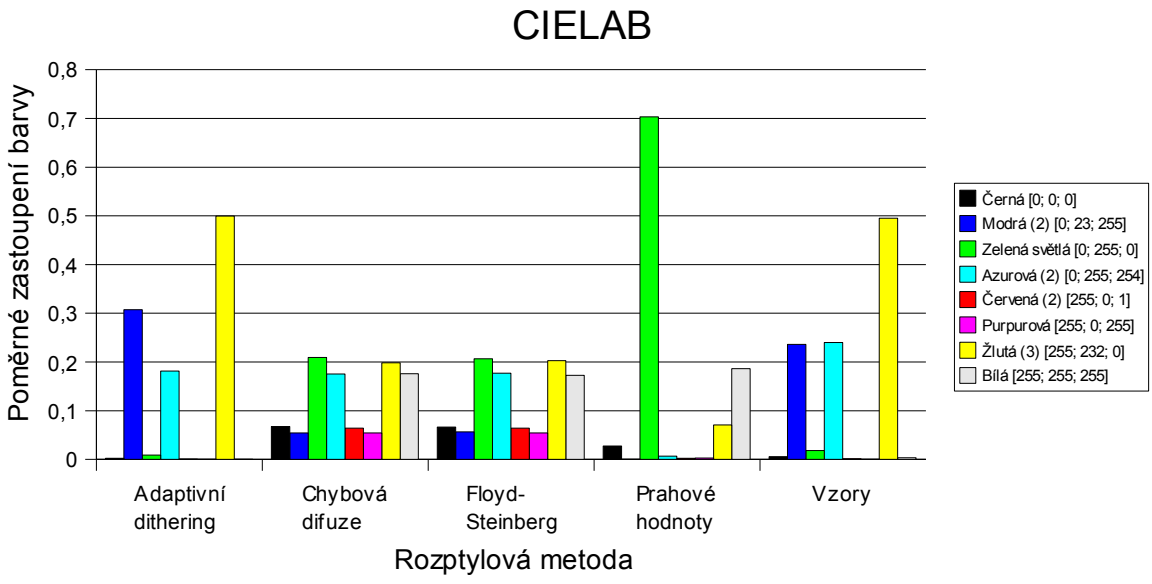


Varianta s prahováním všech barevných kanálů

Při použití barevného prostoru CIE_{Lab} se prahové hodnoty transformují na následující RGB hodnoty. Je vhodné poznamenat, že transformace neprobíhá přímo, ale přes prostor CIE_{XYZ}.

L	a	b	RGB
0	0	0	Modrá [0; 23; 255]
0	0	1	Černá [0; 0; 0]
0	1	0	Purpurová [255; 0; 255]
0	1	1	Červená [255; 0; 1]
1	0	0	Azurová [0; 255; 254]
1	0	1	Zelená [0; 255; 0]
1	1	0	Bílá [255; 255; 255]
1	1	1	Žlutá [255; 232; 0]

Dostali jsme stejnou množinu RGB barev jako v prostoru CIE_{XYZ} , ale v jiném pořadí. Obrazy transformované v prostoru Lab působí velmi zeleně, nejviditelnější to bylo u algoritmu prahových hodnot. V sadě prahových barev jsou relativně podobné odstíny zelené a azurové, a také modrá a žlutá, které, pokud spolu sousedí, dají při pohledu z dálky zelenou.



5 Závěr

Předkládám laskavému čtenáři ke zhodnocení dílo, které shrnuje problematiku světla a barev ve 2D i 3D grafice, včetně velmi aktuální technologie stereoskopie, která umožňuje celkem snadno navodit dojem hloubky prostoru v obraze. Je velmi pravděpodobné, že stereoskopické zobrazování bude v budoucnu standardem, jeho masovému rozšíření zatím brání vysoké pořizovací náklady současných technických realizací. Lze očekávat, že s masovou produkcí půjdou ceny dolů.

V bakalářské práci jsem zúročil znalosti nabyté dosavadním studiem informatiky na Západočeské univerzitě, zejména z předmětů *Základy počítačové grafiky* a *Grafická rozhraní a GPU*. Asi bych neměl opomenout ani předmět *Objektově orientované programování*, který mi dal praktický základ pro tvorbu grafického uživatelského rozhraní, a ostatně také *Počítače a programování 1* a *2*, kde se studenti učí základům algoritmizace. Při tvorbě dokumentace k bakalářské práci se mi rovněž velmi hodily znalosti z předmětu *Příprava textu počítačem*.

V souladu se zadáním jsem implementoval systém pro demonstraci stínovacích metod a stereoskopie. Pomocí hardwarových shaderů jsem implementoval konstantní, Gouraudovo a Phongovo stínování, stereoskopie (renderování scény ze dvou exaktně daných pozic) je realizována v kódu programu.

Při řešení se vyskytlo pár obtíží technického rázu, ale také několik problémů spočívajících v organizaci vlastní práce. Problémy z první kategorie se mi podařilo eliminovat díky dodatečné analýze, jak je možné, že vůbec vznikly.

Jedním z nich byl formát vertex-bufferu objektu `Mesh`, se kterým jsem se pokoušel pracovat přímo. Vykreslení celého objektu najednou fungovalo bez problémů, ale když jsem jej chtěl kreslit po jednotlivých trojúhelnících, což bylo nutné kvůli předávání doplňujících dat pro konstantní stínování, dostal jsem pouze černou obrazovku. Využil jsem tedy znalostí nabytých v předmětu *Základy počítačové grafiky* a převedl jsem tento problém na jiný, jehož řešení jsem už z minula znal. Z objektu `Mesh` jsem přečetl data vertex-bufferu a index-bufferu, vyrobil z těchto dat novou instanci tříd `VertexBuffer` a naplnil ji posloupností trojúhelníků.

Druhým technickým problémem bylo předávání doplňkových informací kartě pro konstantní stínování, protože vzhledem ke způsobu zacházení s vykreslovanými objekty si je nedokáže zjistit sama. Prvním řešením, které mě napadlo, bylo předávat tato data přes uniformní proměnné shaderu pro každý trojúhelník. Toto řešení se ukázalo jako technicky proveditelné, ale prakticky nepoužitelné. Konvička z Utahu se při použití tohoto způsobu kreslila rychlostí 15 snímků za sekundu.

Vyhovujícím řešením se ukázala být konstrukce vlastního formátu vrcholu, který vedle pozice a normály pro bod v prostoru obsahuje pozici a normálu určitého referenčního bodu. Normála vzniká zprůměrováním normál ve vrcholech, předáváním referenčního bodu zajišťuji, aby byla spočítaná barva skutečně konstantní za všech možných okolností. Data se tedy předávají ve společném datovém proudu, rozlišení jejich významu se provádí přiřazenou sémantikou pro vertex-shader.

Touto úpravou jsem se na svém referenčním počítači[11] u konvičky stínované Gouraudovým algoritmem dostal ke 340 snímkům za sekundu. Na řádově výkonnějším počítači v laboratoři počítačové grafiky jsem s Gouraudovým stínováním dosáhl dokonce 6800 snímků za sekundu.

Problém druhé kategorie (organizační) se ale ukázal jako mnohem bolestivější. Nedlouho před řádným termínem odevzdání jsem si uvědomil, že jsem celou dobu vlastně řešil něco jiného než mi bylo zadáno – místo metod stínování pro 3D grafiku jsem řešil metody rozmývání pro 2D grafiku. Nasměroval jsem tedy své úsilí správným směrem a dosud dosažené výsledky jsem v práci ponechal.

Těmi výsledky je právě analýza dopadu barevného prostoru na metody stínování, na kterou nyní mohu také nahlížet jako na analýzu dopadu metod stínování na použitý barevný prostor – obě označení jsou, pokud jde o dosažené výsledky, stejně výstižná.

Pokud jde o použitelnost obrazů stínovaných v různých barevných prostorech, pak často používaná Floyd-Steinbergova metoda dopadla nejlépe v barevných prostorech HSV s prahováním relevantních kanálů S a V a CMYK, ostatní metody vykázaly úbytek barevnosti. Relativně slušně vyšla tato metoda také v prostoru HSV s prahováním všech barevných kanálů, její výsledek by bylo možné použít pro dvoubarevný tisk se dvěma přímými barvami – černou a červenou.

Zaujaly mě také obrazy rozmývané metodou adaptivního ditheringu. Prakticky použitelné jsou však jen výsledky získané v barevných prostorech RGB, CMYK a CIE_{XYZ}, a také v HSV s prahováním všech kanálů ze stejného důvodu, jako je popsáno výše. Naproti tomu v CIE_{Lab}, HSV a YUV jsem získal obrazy s nepřirozenými barvami, které však nebyly „výtvarně nezajímavé," nejvíce mě fascinoval výsledný obraz z CIE_{Lab}. - viz obr. 32 na straně 76.

11 AMD Duron 1.1 GHz, základní deska MSI-745 Ultra (SiS 745), 256 MB DDR-SDRAM, grafika nVidia GeForce 6200 (128 MB) s ovladači verze 6.14.10.9371, operační systém Windows XP

Ke zhodnocení předkládám dílo, o kterém se domnívám, že svým rozsahem překračuje požadavky jedné „standardní“ bakalářské práce. Jsem přesvědčen o tom, že splňuje požadavky zadání, a věřím, že může být k užitku nejen studentům počítačové grafiky, ale i laické veřejnosti – komukoli, kdo má chuť se o řešené problematice něco dozvědět.

Literatura

[Ska93] Skala, V.: *Světlo, barvy a barevné systémy.* Praha, Academia, 1993

[Rad02] Rádlová, R.: *Barevné systémy.* Plzeň, 2002. Bakalářská práce na Fakultě aplikovaných věd Západočeské univerzity, na katedře informatiky a výpočetní techniky. Vedoucí práce Václav Skala.

[FSpri] *Image quantization, Halftoning and Dithering* [online]. c1999 [cit. 2007-06-15]. Dostupné z URL < http://www.cs.princeton.edu/courses/archive/fall99/cs426/lectures/dither/index.htm >

[ZPGcv4] *Barvy a fotorealismus I* [online]. c2007 [cit. 2007-06-15]. Dostupné z URL < http://herakles.zcu.cz/education/zpg/cviceni.php?no=4 >

[ZPGcv5] *Barvy a fotorealismus II* [online]. c2007 [cit. 2007-06-15]. Dostupné z URL < http://herakles.zcu.cz/education/zpg/cviceni.php?no=5 >

[GRG5] Hanák, I.: Světlo a stín. *Přednášky z předmětu KIV/GRG* [online]. [cit. 2007-01-04]. Dostupné z URL < http://herakles.zcu.cz/education/Grg/2006/lects/06-grg-05.pdf >

[GRG6] Hanák, I.: Materiály. *Přednášky z předmětu KIV/GRG* [online]. [cit. 2007-01-04]. Dostupné z URL < http://herakles.zcu.cz/education/Grg/2006/lects/06-grg-06a.pdf >

[JTS] Jirka, T.: *Model Torrance-Sparrow* [online]. [cit. 2007-01-14]. Dostupné z URL < http://herakles.zcu.cz/~tjirka/Doc/lighting_model%20-%20Torrance-Sparrow.ppt >

[MVE2] *Modular Visualization Environment – 2* [online]. c2007. [cit. 2007-06-15]. Dostupné z URL < http://herakles.zcu.cz/research/projects/11/index.php >

[FSwiki] Přispěvatelé Wikipedie[12]: *Floyd-Steinberg dithering* [online]. [cit. 2007-01-14]. Dostupné z URL < http://en.wikipedia.org/wiki/Floyd-Steinberg_dithering >

[CIE_{XYZ}wiki] Přispěvatelé Wikipedie: *CIE 1931 color space* [online]. [cit. 2007-01-14]. Dostupné z URL < http://en.wikipedia.org/wiki/CIE_1931_color_space >(anglická verze) a < http://de.wikipedia.org/wiki/CIE-Normvalenzsystem > (německá verze)

[GamutWiki] Přispěvatelé Wikipedie: *Gamut* [online]. [cit. 2007-01-14]. Dostupné z URL < http://en.wikipedia.org/wiki/Gamut >

[CIELabwiki] Přispěvatelé Wikipedie: *Lab Color Space* [online]. [cit. 2007-01-14]. Dostupné z URL < http://en.wikipedia.org/wiki/Lab_color_space >

12 Wikipedie je otevřená internetová encyklopedie s volně šiřitelným obsahem pod licencí GNU/FDL.

[RGBwiki] Přispěvatelé Wikipedie: *RGB* [online]. [cit. 2007-01-14]. Dostupné z URL < http://en.wikipedia.org/wiki/Rgb > (anglická verze) a < http://cs.wikipedia.org/wiki/RGB > (česká verze)

[YIQwiki] Přispěvatelé Wikipedie: *YIQ* [online]. [cit. 2007-01-14]. Dostupné z URL < http://en.wikipedia.org/wiki/YIQ >

[YUVwiki] Přispěvatelé Wikipedie: *YUV* [online]. [cit. 2007-01-14]. Dostupné z URL < http://en.wikipedia.org/wiki/YUV >

[HSVwiki] Přispěvatelé Wikipedie: *HSV color space* [online]. [cit. 2007-01-14]. Dostupné z URL <http://en.wikipedia.org/wiki/HSV_color_space >

[CCwiki] Přispěvatelé Wikipedie: *Complementary color* [online]. [cit. 2007-01-14]. Dostupné z URL <http://en.wikipedia.org/wiki/Complementary_color >

[CMYKwiki] Přispěvatelé Wikipedie: *CMYK* [online]. [cit 2007-01-14]. Dostupné z URL < http://cs.wikipedia.org/wiki/CMYK > (česká verze) a < http://en.wikipedia.org/wiki/CMYK_color_model > (anglická verze)

[PRMwiki] Přispěvatelé Wikipedie: *Phong reflection model* [online]. [cit 2007-01-14]. Dostupné z URL < http://en.wikipedia.org/wiki/Phong_reflection_model >

[FlShWiki] Přispěvatelé Wikipedie: *Flat shading* [online]. [cit 2007-01-14]. Dostupné z URL < http://en.wikipedia.org/wiki/Flat_shading >

[GSwiki] Přispěvatelé Wikipedie: *Gouraud shading* [online]. [cit. 2007-01-14]. Dostupné z URL < http://en.wikipedia.org/wiki/Gouraud_shading >

[PSwiki] Přispěvatelé Wikipedie: *Phong shading* [online]. [cit 2007-01-14]. Dostupné z URL < http://en.wikipedia.org/wiki/Phong_shading >

[AGwiki] Přispěvatelé Wikipedie: *Anaglyf* [online]. [cit 2007-05-12]. Dostupné z URL < http://cs.wikipedia.org/wiki/Anaglyf >

[SteWiki] Přispěvatelé Wikipedie: *Stereoskopie* [online]. [cit 2007-05-12]. Dostupné z URL < http://cs.wikipedia.org/wiki/Stereoskopie >

[KaiStereo] Kaiser, J.: *Software pro prohlížení obrázků na stereo stěně* [online]. c2004, [cit. 2007-06-15]. Dostupné z URL < http://home.zcu.cz/~kaiserj/temp/apg/StereoViewer.pdf >

[ZPGstereo] *StereoWallTutorial* [online]. [cit. 2007-06-15]. Demonstrační program stereoskopického renderování, pro Windows XP a DirectX. Dostupné z URL <http://herakles.zcu.cz/education/zpg/down/StereoWallTutorial.zip >

[ERGBcf] *EasyRGB - Color mathematics and conversion formulas* [online]. c2007, [cit. 2007-06-15]. Dostupné z URL < http://www.easyrgb.com/math.html >

[Lbloom] Lindbloom, B. J.: Useful Color Equations [online]. c2007, [cit. 2007-06-15]. Dostupné z URL < http://www.brucelindbloom.com/Math.html >

[VishCCA] Vishnevsky, E.: *Color Conversion Algorithms* [online]. [cit. 2007-06-15]. Dostupné z URL < http://www.cs.rit.edu/~ncs/color/t_convert.html >

[GlynnHSV] Glynn, E.F.: *efg's HSV Lab Report* [online]. c2005, [cit. 2007-06-15]. Dostupné z URL < http://www.efg2.com/Lab/Graphics/Colors/HSV.htm >

[MSDN2] *Microsoft Development Network Library* [online]. c2007, [cit. 2007-06-15]. Dostupné z URL < http://msdn2.microsoft.com/en-us/library/default.aspx >

[MSDX9] Mirza, Y. H., da Costa, H.: *DirectX 9.0: Introducing the New Managed Direct3D Graphics API in the .NET Framework* [online]. c2003, [cit. 2007-07-02]. Dostupné z URL < http://msdn.microsoft.com/msdnmag/issues/03/07/DirectX90/ >

[MSDXCST] *Camera Space Transformations (Direct3D 9)* [online]. c2007, [cit. 2007-07-11]. Dostupné z URL < http://msdn2.microsoft.com/en-us/library/bb172390.aspx >

Přílohy

A. Uživatelská dokumentace

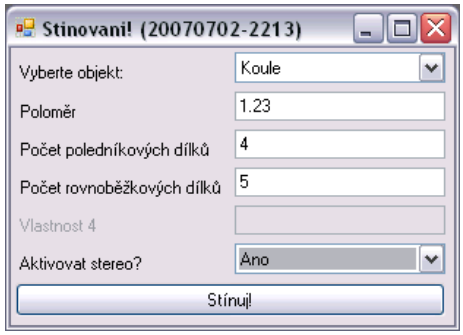
Program pro demonstraci metod stínování

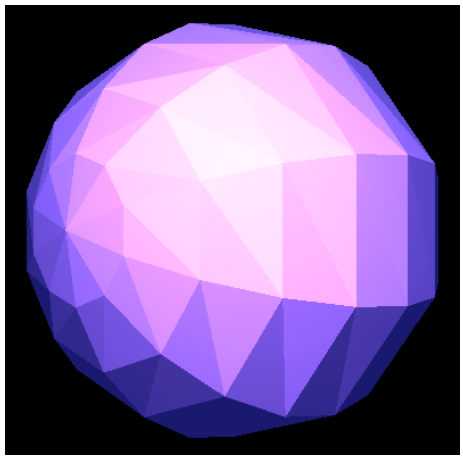
Program vyžaduje pro bezproblémový běh následující:

- Microsoft .NET Runtime ve verzi 2.0 nebo vyšší
- Microsoft DirectX Runtime ve verzi 9.0 (October 2005)
- Microsoft Windows XP

Obojí jsou dostupné ke stažení na http://www.microsoft.com .

Pro spuštění programu poklepejte myší na jeho ikonku ve vašem oblíbeném správci souborů. Preferujete-li příkazovou řádku, donavigujte do adresáře s programem a napište jeho název:





Ilustrace 34 (nahoře) a 35 (dole) k programu Stínování. Zdroj: vlastní produkce.

```
C:\X:
X:\cd public
X:\public\stinovani.exe
```

… kde X je písmeno, které vaší optické mechanice přiřadil operační systém, typicky D nebo E. Program není třeba instalovat, spouští se přímo z CD.

Po spuštění program zobrazí okno uživatelského rozhraní, které požaduje zadání relevantních informací – co se má kreslit a jak to má být velké.

Po zodpovězení otázek se objeví celoobrazovkové okno aplikace, které vykresluje zvolený objekt. V tomto stavu program reaguje na povely z klávesnice:

- Escape – konec programu,
- S – změna stínovací techniky (konstantní stínování, Gouraudovo stínování a Phongovo stínování, dále opět konstantní),
- E – zapnutí / vypnutí efektu, umožní porovnat hardwarové a softwarové stínování,
- X – posune kameru dolů,
- Y – posune kameru nahoru (na QWERTZ klávesnici vlevo dole)
- U – uloží snímek obrazovky do kořenového adresáře disku C.

Program pro demonstraci vlivu barevných systémů

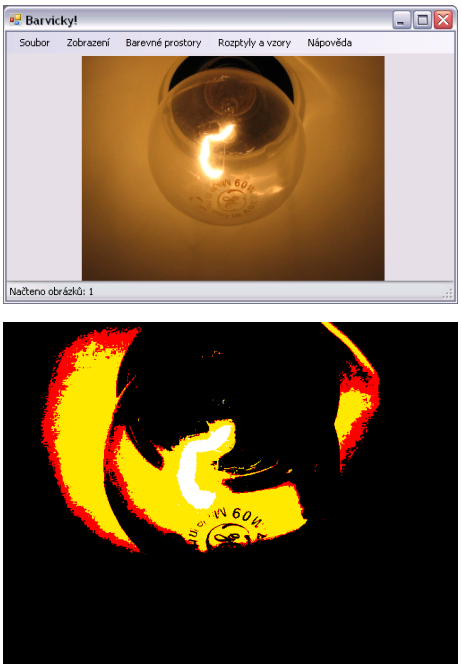
Pro bezproblémový běh program vyžaduje:

- Microsoft .NET Framework Runtime ve verzi 2.0 nebo novější
- Microsoft Windows XP

.NET framework je k dispozici k bezplatnému stažení na http://www.microsoft.com/net/ .

Pro spuštění programu poklepejte myší na jeho ikonku ve vašem oblíbeném správci souborů. Preferujete-li příkazovou řádku, donavigujte do adresáře s programem a napište jeho název:

```
C:\X:
X:\cd public
X:\public\barvicky.exe
```



Ilustrace 36 a 37 k programu Barvičky. Na vrchním snímku okno programu s načteným obrazem Referencni_02.jpg, na spodním snímku výsledek metody konstantního prahu v barevném prostoru CIE_{XYZ}. Zdroj: vlastní produkce, včetně fotografie.

… kde X je písmeno, které vaší optické mechanice přiřadil operační systém, typicky D nebo E. Program není třeba instalovat, spouští se přímo z CD.

Zobrazí se okno programu, odpovídající současnému standardu aplikací pro Windows. Jeho některé prvky jsou zatím deaktivovány.

Zvole menu Soubor – Otevřít - Jeden obrázek nebo Dvojici obrázků. V souborovém dialogu vyberte soubor, na kterém chcete zkoumat vliv barevných prostorů a klikněte na OK. Zvolený obrázek se načte a zobrazí se v okně aplikace.

S obrázkem (nebo obrázky) se manipuluje zvolením kombinace barevného prostoru a metody rozptylu z příslušného menu. Po vybrání libovolné položky začne program

okamžitě počítat obraz, který odpovídá právě zvolené kombinaci hodnot.

V menu Zobrazení lze zvolit zvětšení obrazů (100% a automatické zvětšení) a také aktivovat celoobrazovkový stereoskopický režim. V něm lze klávesou Z nastavit zvětšení/zmenšení obrazu (stejně jako v hlavním okně). Celoobrazovkový režim se ukončuje klávesou Esc.

Program se ukončí zavřením jeho hlavního okna způsobem obvyklým pro aplikace v prostředí Windows nebo pomocí menu Soubor – Konec.

B. Obrazová příloha

(38)

(39)

(40)

(41)

(42)

(43)

Ilustrace 38 – 43 zachycují dopad rozmývacích metod na obrázek v prostoru RGB:
38 – bez rozptylu, 39 – prahové hodnoty, 40 – Floyd-Steinbergův algoritmus,
41 – chybová difuze, 42 – metoda vzorů 2×2, 43 – adaptivní dithering.

(44)

(45)

(46)

(47)

(48)

(49)

Ilustrace 44 – 49 ukazují vliv metody konstantního prahu, je-li provedena v různých barevných prostorech. 44 – RGB, 45 – CMYK, 46 – HSV s prahováním kanálů S a V, 47 – HSV s prahováním všech kanálů, 48 – YUV s prahováním kanálu Y, 49 – YUV s prahováním všech kanálů. Zdroj: vlastní produkce (všechny).

(50)

(51)

Ilustrace 50 – 52 ukazují vliv metody konstantního prahu, je-li provedena v různých barevných prostorech.

50 – CIE_{Lab} s prahováním kanálu L,
51 – CIE_{Lab} s prahováním všech kanálů,
52 – CIE_{XYZ}.

Zdroj: vlastní produkce (všechny).

(52)

C. Programátorská dokumentace

Programátorská dokumentace byla vygenerována open-source nástrojem Doxygen, který je volně k dispozici ke stažení na http://www.doxygen.org.

Vzhledem k velkému rozsahu je text programátorské dokumentace umístěn v samostatné příloze. Jeho plné znění je k dispozici také na přiloženém CD.