Abstract

Image synthesis in the computer graphics is usually performed using trichromatic color systems, e.g. RGB. That is sufficient for lots of applications but is insufficient in the areas where accurate color computations, which involve the whole range of the visible light wavelength, are required. These computations are many times more expensive and therefore some acceleration has to be introduced to make them practical to use. Increasing power of graphics hardware accelerators may bring new possibilities into the full spectral rendering.

Contents

Declaration

I hereby declare that this diploma thesis is completely my own work and that I used only the cited sources.

Martin Janda

Pilsen, April 2004

1 Introduction

An important application of the computer graphics (CG) is synthesis or rendering of the images of the real world. These images are then used for preview purposes in architecture and design or for entertainment in movies. Obviously in an interest of all who is working with the synthesized images is a maximal level of photorealism. This implies a correct light transport simulation and correct calculation of colors.

In the process of rendering is used a mathematical description of a scene's geometry as a starting point for the light transport simulation. The light from light sources propagates through space until it reaches some surface. The following simulation of the light's interaction with a surface then determines a surface's apparent color. The simulation is driven by an illumination model which uses material's properties, positions of light sources and a position of a viewer to compute the final color and its intensity.

The precise and exact calculation of light's interaction with a surface is physically and computationally very difficult task and there are lots of simplifications made to speed things up. One such simplification is use of the light sources which emit only in three wavelengths of light. As a consequence, material's reflectance is needed to be defined only at those three wavelengths. This simplification is possible to make because of the principle a human eye perceive color information.

There are two types of receptors in the human eye, cones and rods. While all rods are of the same type and thus provide only achromatic information, there are three types of the cones. Each type is sensitive to a different wavelength, which roughly corresponds to a red, a green and a blue color respectively. This organization of the human eye allows representing each color as weighted mix of sources of a red, a green and a blue light.

From the text above one could think that sampling a scene only at three wavelengths is sufficient. But it is not true. The spectral reflectance of materials and the spectral power distribution of light sources are continuous with respect to a wavelength. For this reason the color calculations based only on three wavelengths cannot accurately simulate an interaction of light with a surface.

Full spectral information must be used in order to calculate colors accurately. As one might expect, this makes rendering more computationally expensive and also other problems are introduced. Spectral data are seldom known and reliable measurement is a quite difficult task. Data mixing problem arise when spectral information and RGB information is used in one scene and once rendering is done, the compensation of the human eye's adaptability must be made to match computed colors with perceived ones. On the other hand, the advantage of full spectral information is that it enables more accurate simulation of wavelength dependent effects like fluorescence, chromatic dispersion or diffraction.

If other problems are omitted then the main problem of efficiency can be solved by utilization of more efficient spectral calculation methods such as linear color representation, which uses ortho-normal basis functions to express all spectral power distributions within a scene or a color pre-filtering, which computes the color of a surface for a dominant light and then uses standard RGB illumination computations. Both methods are tested and verified in this thesis.

Another way of speeding things up is to transfer computation from a CPU to a GPU. The graphic processing units are equipped with a programmable pipeline and with sufficient computation power to overtake the spectral computations. For the programming of the pipeline a special high level shading language was developed and one of its implementation is included in the graphics library DirectX.

To conclude, full spectral rendering brings great possibilities for photorealistic rendering. The tradeoff is increased computation time and data requirements. More efficient spectral data representation and hardware acceleration can reduce the computation time. Exploration of these methods and their efficiency is the main issue of this thesis.

2 Illumination Models

Once light hits a surface then very complicated interaction occurs. Some light is reflected; some light is absorbed and eventually transmitted. The reflected light then determines the surface's apparent color, which is the required information when a synthetic image is calculated in the CG.

This interaction of light and a surface is in the CG described by an illumination model. The illumination model is usually simplified version of the real phenomena in order to lower computation demands, which would be very high for a physically exact solution. But even the simplified models are sufficient if they provide visually believable result, for in many cases visual believability is more important than physical correctness, see Phong's illumination model. Alternatively, in some cases physically correct values are required as well and therefore more accurate illumination model should be used, see Torrance & Cook's illumination model.

The essence of each illumination model is a bi-directional reflectance distribution function or shortly BRDF which relates luminance in an examined outgoing direction to illumination reaching the point of a surface from a light source, therefore the bi-directional labeling. This function can be computed analytically or can be measured and tabulated. Difference between a computed BRDF and a measured BRDF could be used as a reference of an illumination model's accuracy.

A problem of shading is related with the illumination models. By the shading is meant computations of values inside a polygon when the values are known only at vertices. The used method has a great effect on the appearance of a rendered polygonal mesh and although a good illumination model is used bad shading can spoil the realism of a final image.

2.1 Bi-Directional Reflectance Distribution Function

A color of a surface is determined by light reflected from it. The reflected fraction of incident light is given by a reflection function. Because the function is relating the amount of incident light from a direction ω_i and amount of light reflected in a direction ω_o the function is labeled as a bi-directional reflectance distribution function abbreviated as a BRDF.

A similar function can be defined for transparent materials. This function relates the amount of light incident on a surface and the amount of light transmitted by the surface. Each of these functions is defined in a different half-space. A reflectance function is defined in a half-space over the surface and a transmittance function is defined in a half-space under the surface. However, the transmittance function is seldom used. Transparent materials are usually treated as perfectly transparent and homogenous, which means that light is transmitted in one direction according to the Snell's refraction law and the intensity loss at interface does not depend on an incident direction.

Some radiometric quantities are needed for a proper definition of a BRDF and therefore they are introduced first.

2.1.1 Radiometry

Radiometry is the measurement of optical radiation. The optical radiation is electromagnetic radiation in a frequency of range from 3×10^{11} Hz to 3×10^{16} Hz. This interval includes infrared, visible and ultraviolet regions. Presented information was taken from [Palm03].

The basic quantity of the radiometry is energy. Energy is labeled as Q and its units are Joules [J]. Radiant energy is the sum of photons' energy (photon is a quantum of electromagnetic radiation). Energy of one photon is $Q_P = h.f$, where $h = 6.626 \times 10^{-34}$ is the Planck's constant, $f = c/\lambda$ is the frequency of the photon, c is the speed of light and λ denotes a wavelength. The amount of energy radiated by a radiation source over time is called power or a radiant flux. Power is labeled as Φ, its unit is Watt [W] and it is a derivative of energy with respect to time $\Phi = dQ/dt$.

There are two geometric quantities used in the definition of the following radiometric quantities. One of them is area A and its projected version $A\cos\theta$, where θ denotes an angle between the normal of an area and the direction onto which is the area projected. Another important geometric quantity is a solid angle Ω, which is measured in steradians [sr]. The Ω is defined as the ratio of a spherical area ant the square of the radius of the projection sphere. The spherical area is a projection of the object of interest onto a unit sphere and the corresponding solid angle Ω is the area of that projection. There are maximally 4π steradians of a solid angle Ω in a sphere.

Irradiance M is defined as power per unit area incident from all direction in a hemisphere onto surface which coincides with the base of the hemisphere. Irradiance $M = d\Phi/dA$ and its unit is $[Wm^{-2}]$. Power leaving a surface into all direction in a hemisphere is the same quantity but for practical reasons is called a radiant exitance and is labeled as E.

Radiant intensity I is power per unit solid angle. Radiant intensity $I = d\Phi/d\omega$ and its unit is $[Wsr^{-1}]$. This quantity is introduced mainly for its meaning in photometry.

Radiance L is defined as power per projected unit area per unit solid angle. Its unit is $[Wm^{-2}sr^{-1}]$. Similarly to the irradiance distinction the radiance incident on a surface is referred as field radiance $L_f = M/(d\omega\cos\theta)$ and the radiance leaving a surface as surface radiance $L_s = E/(d\omega\cos\theta)$. Radiance alias luminance in photometric terminology is the quantity usually computed in the CG.

All these quantities may vary along a wavelengths range so their spectral versions are marked with a subscript λ. But in this thesis the spectral versions are taken as implicit so the subscript will be omitted in the following text.

2.1.2 Photometry

Photometry is related with the radiometry that was described briefly in the previous section. While the radiometry concerns with electromagnetic radiation the photometry targets only the narrow range of a visible part from 360 nm to 830 nm that is called light. The photometry has the same quantities as radiometry except that they are weighted with eye's spectral response and their units have different names. Presented information was taken from [Palm03].

Radiant intensity in photometry is called luminous intensity I_v. Its unit is a candela [cd]. This unit is one of the SI units. Its definition changed during time but the most recent definition is that one candela is the luminous intensity in a given direction of a light source of monochromatic radiation with a frequency 540 GHz and that has radiant intensity 1/685 watts per steradian.

Other photometric quantities based on luminous intensity are:

- luminous flux Φ_v – corresponds with power in radiometry, its unit is lumen [lm],
- illumination E_v – corresponds with irradiance in radiometry, its unit is lux [lx],
- luminance L_v – corresponds with radiance in radiometry, its unit is nit [cd/m^2].

2.1.3 BRDF Definition

With defined irradiance and radiance the BRDF is then expressed as a ratio E[2.1-1]. This definition is simplified for clearing the basic idea. The BRDF is actually dependent also on a wavelength, position, polarization and eventually on orientation if material is anisotropic. The positional dependence is usually simulated by a texture, the wavelength and the anisotropy are considered only in advanced illumination models and the polarization is ignored almost in all cases.

$$\rho(k_i, k_o) = \frac{L_s}{M} \qquad \text{E[2.1-1]}$$

L_s	Radiance leaving a surface in a small solid angle around a direction k_o
M	Irradiance reaching a surface from a light source in a direction k_i

When an image is calculated the required information is the radiance that reaches an eye or a camera. The radiance is a universal quantity because if the radiance is known then all other radiometric quantities can be calculated from it. If irradiance expressed using radiance E[2.1-2] is substituted into the BRDF's definition E[2.1-1] then one gets E[2.1-3] and after rearranging E[2.1-4]. Once the BRDF is known then the total radiance leaving a surface in a direction k_o is computed as an integral E[2.1-5]. The equation E[2.1-5] is the basis for so called rendering equation and the calculation of its solution or at least its approximation is in the CG called rendering.

$$H = L_i \cos\theta_i d\omega_i \qquad \text{E[2.1-2]}$$

$$\rho(k_i, k_o) = \frac{L_o}{L_i \cos\theta_i d\omega_i} \qquad \text{E[2.1-3]}$$

$$L_o = \rho(k_i, k_o) L_i \cos\theta_i d\omega_i \qquad \text{E[2.1-4]}$$

$$L_s(k_o) = \int_{2\pi} \rho(k_i, k_o) L_f(k_i) \cos\theta_i d\omega_i \qquad \text{E[2.1-5]}$$

$$R(k_i) = \int_{2\pi} \rho(k_i, k_o) \cos\theta_o d\omega_o \qquad \text{E[2.1-6]}$$

$$\rho(k_i, k_o) = \rho(k_o, k_i) \qquad \text{E[2.1-7]}$$

The properly defined BRDF should fulfill the condition of energy conservation which states that that the amount of the energy reflected from a surface must be smaller or equal to the amount of the incident energy. The BRDF fulfills this condition if directional

hemispherical reflectance E[2.1-6] is smaller or equal to one for all directions k_i. Another notable property of the BRDF is that equation E[2.1-7] should apply due to the Helmholtz reciprocity rule.

2.2 Illumination Models Overview

The illumination models are the essence of each rendering system. Their purpose is the simulation of a light's reflection form a surface. The accuracy of this simulation directly influences the photorealism of a rendered image. The evaluation of illumination models takes most of the computation time because they are often very computationally expensive and are evaluated many times for a single image. The requirements of accuracy and a fast evaluation are obviously contradictory. For this reason illumination models can be separated into two groups.

The first group consists of the empirical illumination models which are computationally effective but are simplified against the reality. They are physically inaccurate but provide visually acceptable results. These models are incorporated into the applications where visual believability and a fast evaluation are more important then physically accurate values. The most common representative of this group is the Phong's illumination model. This model is used in 3D accelerators even though a programmable pipeline enables more complex models.

The second group consists of the physically based illumination models which are more physically accurate. They are more complex then the empirical ones and their use is in the applications where higher accuracy is required even at the expense of longer evaluation times. The most common illumination model of this group in the CG is the Blinn's illumination model.

Another aspect of classification is the implementation of the BRDF described in [Schl94]. The original BRDF, described in section 2.1.2, is a ratio of radiance leaving a point on a surface and irradiance reaching that point. As a consequence, values of the BRDF are not bounded to an interval $\langle 0, 1 \rangle$. The BRDF has to be weighted by the solid angle subtending incident irradiance to get a proper value. It is a problem whenever that information is unavailable such as in the case of a ray tracer. Afterwards the different definition E[2.2-1] of a BRDF has to be used. The important property of this BRDF is a normalization property E[2.2-2] which implies that the reflected radiance cannot be larger then the incident radiance. The total surface radiance due to *S* light sources has the form of the finite sum E[2.2-3].

$$\rho(k_i, k_o) = \frac{L_o}{L_i} \qquad \text{E[2.2-1]}$$

$$\forall k_i, \forall k_o : \rho(k_i, k_o) \leq 1 \qquad \text{E[2.2-2]}$$

$$L_s(k_o) = \sum_{i=1}^{S} \rho(k_i, k_o) L_f(k_i) \qquad \text{E[2.2-3]}$$

$\rho(k_i, k_o)$	Alternative BRDF of a surface.
$L_s(k_o)$	Radiance leaving the point of a surface in a direction k_o.

$L_f(k_i)$	Radiance reaching the point of a surface from a direction k_i.

In the following section the most common illumination models in the CG are described. Also their usability in the context of a spectral rendering is discussed. Vectors' and angles' labeling as shown in Fig. 2-1 is used for their mathematical description.

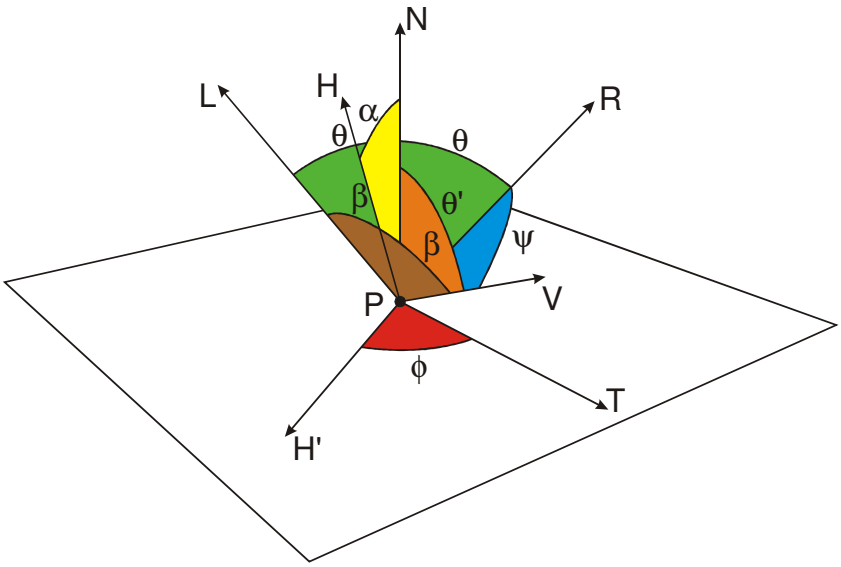

Fig. 2-1: Vectors' and angles' labeling used in the descriptions of illumination models.

L	A direction to a light source.	$\alpha = \cos^{-1} h$	$h = \mathbf{H.N}$
N	The normal of a surface.	$\beta = \cos^{-1} i$	$i = \mathbf{H.V}$
R	The mirror reflection of incident light.	$\theta = \cos^{-1} j$	$j = \mathbf{N.L}$
V	A direction to a viewer.	$\theta' = \cos^{-1} k$	$k = \mathbf{N.V}$
H	A bisector of L and V.	$\psi = \cos^{-1} l$	$l = \mathbf{R.V}$
T	The tangent of a surface.	$\phi = \cos^{-1} m$	$m = \mathbf{H'.T}$
H′	The projection of H on a surface		
P	The point on a surface.		

2.2.1 Phong's Illumination Model

Phong proposed his illumination model in 1975. This model falls into the group of the empirical models and is defined using the simplified BRDF. The BRDF of this model is expressed as the linear combination of a diffuse and a specular term. According to [Phong75], the BRDF is expressed as E[2.2-4]. The definition of the BRDF as the linear combination of a specular and diffuse term is based on the idea that there are two types of reflection which depends on surface properties.

If a surface is smooth then light is reflected specularly. This reflection is highly directional and obeys the law of mirror reflection. The light reflected in this way can be recognized as a bright spot or highlight on a surface. If a surface is rough then a diffuse reflection occurs. Light is reflected multiple times on the irregularities of a surface and some fraction is also absorbed and partially reemitted. Such light is reflected in all directions with approximately same probability.

Reflected light is always the mixture of the specular and the diffuse reflection in the real world. The separation of the diffuse and the specular term is very common and appears also in the descriptions of other illumination models.

$$\rho = C_p(\mathbf{N.L}(1-d)+d)+W(\theta_i)(\mathbf{R.V})^n \qquad \text{E[2.2-4]}$$

C_p	The reflection coefficient of a surface at a point P.
d	An environmental diffuse reflection coefficient. It simulates ambient term.
$W(\theta_i)$	The function that relates an incident angle and the ratio of specularly reflected light and incident light.
n	The shininess of a specular reflection.

The term $\mathbf{N.L}$ in E[2.2-4] simulates diffuse reflection which is based on the Lambert's law of illumination which states that the amount of illumination is proportional to the cosine of the incidence angle. The cosine of the angle subtended by two normalized vectors equals to the dot product of those vectors and therefore the cosine function does not have to be evaluated.

The specular highlight term $\mathbf{R.V}$ is based on empirical observations. The amount of specularly reflected light is greatest in the mirror reflection direction $\mathbf{R}$. The intensity in other directions then fades with an increasing deviation from the mirror direction. Phong took a cosine function as approximation of this behavior. The power n of the specular term then increases speed of its falloff.

$$\rho = dC^D\mathbf{N.L} + sC^S(\mathbf{H.N})^n \qquad \text{E[2.2-5]}$$

$d, s \in \langle 0,1 \rangle$	Ratio of diffuse resp. specular behavior of the surface. The light which is not reflected diffusely is reflected specularly so setting $d + s = 1$ is appropriate.
$C^D, C^S \in \langle 0,1 \rangle$	The fractions of incident light reflected in a diffuse resp. specular way.

However, Phong's illumination model formulation as proposed in [Phong75] is less known then the more common formulation w[2.2-5]. The main differences between these two formulations are:

- The specular reflectivity term C^S is given by the function of incident angle $W(\theta_i)$ in the original Phong's formulation. That means the reflected light has the same color as its source. This is the characteristic behavior of plastic materials. Setting $C^S = C^D$ is more appropriate to achieve a more metallic appearance.

- The use of $\mathbf{H}$ instead of $\mathbf{R}$. The computation of the vector $\mathbf{R}$ is quite complex, $\mathbf{R} = 2(\mathbf{N} * \mathbf{L.N}) - \mathbf{L}$, so the more efficiently computed vector $\mathbf{H}$ is used instead. $\mathbf{H}$ is computed as $\|\mathbf{L} + \mathbf{V}\|$ and represents the normal vector of the plane which reflects light into the viewer's direction. The difference in the behavior of $\mathbf{H}$ and $\mathbf{R}$ is that $(\mathbf{H.N})^n$ has a slower falloff than $(\mathbf{V.R})^n$ for the same n as illustrated in the Fig. 2-2. Once the efficiency is concerned then even more efficient specular falloff function can be found

in [Schl94]. Cosine highlight function is substituted with expression E[$\rho = dC^D\mathbf{N.L} + sC^S \frac{h}{h - hp + p}, \quad h = \mathbf{H.N}$ E[2.2-6]. It has slightly different falloff than the cosine function but still provides acceptable results as illustrated in Fig. 2-3.

$$\rho = dC^D\mathbf{N.L} + sC^S \frac{h}{h - hp + p}, \quad h = \mathbf{H.N} \qquad \text{E[2.2-6]}$$

$p \in \langle 1, \infty)$ Highlight falloff coefficient. Larger values make sharper highlight.

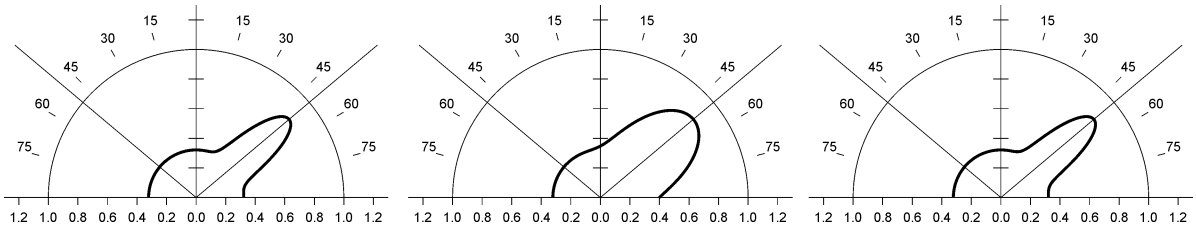

Fig. 2-2: The Phong's model with s = .5, d = .5. The BRDF of the R based highlight with the exponent set to 30 – the left diagram. The BRDF of the H based highlight with the exponent set to 30 – the middle diagram. The BRDF of an H based highlight with the exponent set to 120 – the right diagram.

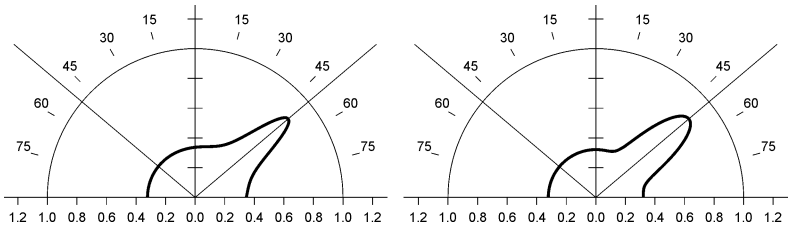

Fig. 2-3: Phong's model with s = .5, d = .5. The BRDF of the Schlick's highlight function with the coefficient set to 300 – the left diagram. The BRDF of the cosine based highlight with the exponent set to 120 – the right diagram.

From the view of spectral rendering is the Phong's illumination model suitable only for the fast previews of color matching. It can be evaluated fast enough which is very important for the spectral rendering due to its massive increase of illumination operations. On the other hand this model is unable to utilize information about a wavelength in any way. However, there is one modification of the Phong's model introduced in [Hall-Greenberg83] where the wavelength dependant Fresnel term was added into the specular term definition E[2.2-7]. This model is capable of the utilization of information about a wavelength thus making Phong's illumination model more useful.

$$\rho = dC^D\mathbf{N.L} + sF(i)(\mathbf{H.N})^n \qquad \text{E[2.2-7]}$$

$F(i)$ The Fresnel term relates the amount of light reflected and the amount absorbed.

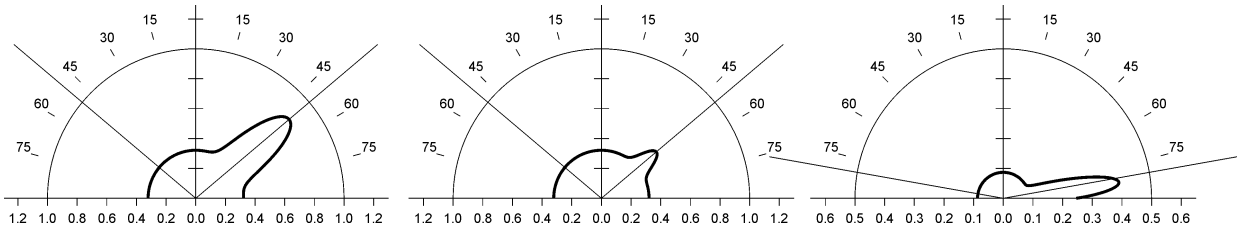

Fig. 2-4: Phong's model with s = .5, d = .5. The BRDF of an H based highlight with the exponent set to 120 – the left diagram. The effect of the Fresnel term with 0.32 normal incidence reflection – the middle diagram. A situation for the 80 degree incident angle – the right diagram.

2.2.1.1 Fresnel Term

This term express relation between the reflected and the absorbed fraction of the incident light on a perfect mirror. The term determines the strength of specularly reflected light as the function of an incident angle and a wavelength. This wavelength dependence determines the color of a specular highlight. This term also causes the rising of reflection amount as the angle of incidence increases because less light is absorbed and more is reflected. As the incident angle approaches $\pi/2$ the reflectivity approaches 100% for all wavelengths and the highlight's color is then identical with the color of a light source. This increase of reflectivity at a large incident angle also shifts the specular lobe's peak off the mirror reflection direction as can be seen in Fig. 2-4.

The complete formulation of the Fresnel term E[2.2-8] for unpolarized light as the function of the cosine of the angle of incidence is quite complex [Schl94] (the angle is usually obtained by the dot product of two vectors, so an inverse cosine function doesn't have to be evaluated). Fortunately the extinction coefficient equals zero for dielectrics and is very small for metals and then the expression becomes much simpler E[2.2-9].

$$F(u)=\frac{1}{2}\frac{(a-u)^2+b^2}{(a+u)^2+b^2}\left(\frac{(a+u-1/u)^2+b^2}{(a-u+1/u)^2+b^2}+1\right)$$

$$a^2=\frac{1}{2}\left(\sqrt{\left(n_\lambda^2-k_\lambda^2+u^2-1\right)^2+4n_\lambda^2k_\lambda^2}+n_\lambda^2-k_\lambda^2+u^2-1\right) \qquad \text{E[2.2-8]}$$

$$b^2=\frac{1}{2}\left(\sqrt{\left(n_\lambda^2-k_\lambda^2+u^2-1\right)^2+4n_\lambda^2k_\lambda^2}-n_\lambda^2+k_\lambda^2-u^2-1\right)$$

n_λ The index of refraction of surfaces' interface.

k_λ The material's extinction coefficient.

$$F(u)=\frac{1}{2}\frac{(a-u)^2}{(a+u)^2}\left(\frac{(u(a+u)-1)^2}{(u(a-u)+1)^2}+1\right) \qquad \text{E[2.2-9]}$$

$$a=\sqrt{n_\lambda^2+u^2-1}$$

Fresnel term has one disadvantage which is that n_λ is seldom known. On the other hand, there are measured spectral distributions f_λ of reflectance at normal incidence for a lots of materials and n_λ can be computed from those spectral distributions. Fresnel equation for normal incidence and $k_\lambda=0$ is E[2.2-10] and n_λ can be then expressed as E[2.2-11]. For faster evaluation the Fresnel term can be approximated with polynomial E[2.2-12] proposed

in [Schl94]. The comparison of the complete and approximated Fresnel term formulation is shown in the Fig. 2-5.

$$f_\lambda = \left(\frac{n_\lambda - 1}{n_\lambda + 1} \right)^2 \qquad \text{E[2.2-10]}$$

$$n_\lambda = \frac{1 + \sqrt{f_\lambda}}{1 - \sqrt{f_\lambda}} \qquad \text{E[2.2-11]}$$

$$F_\lambda(u) = f_\lambda + (1 - f_\lambda)(1 - u)^5 \qquad \text{E[2.2-12]}$$

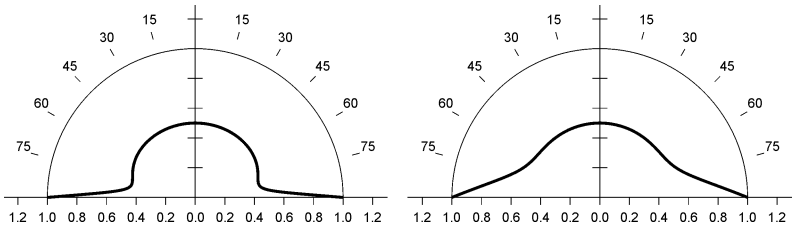

Fig. 2-5: The original Fresnel term – the left diagram and its polynomial approximation – the right diagram.

2.2.2 Torrance & Cook's Illumination Model

Torrance & Cook's illumination model E[2.2-13] is based on the previous work of Torrance and Sparrow done in physics. It is advanced and accurate simulation of light's interaction with a surface based on ray optics, although it contains the Fresnel term which belongs to the wave optics. The model as defined in [Cook81] uses real world units and uses the BRDF in the form defined in the section E[2.1-3]. This model is designed for the renderer systems which perform true integration, such as radiosity, because the solid angle subtending the incident light is needed for evaluation. The BRDF is again separated into the specular and the diffuse term. While the diffuse part is very simple – it follows the Lmabert's cosine low of illumination already used in the Phong's illumination model, the specular term contains all the complexity of this model.

$$\rho = d\,\frac{C^D}{\pi} + s\left(\frac{F(i)}{4\pi}\,\frac{D(h)}{\mathbf{N}.\mathbf{L}}\,\frac{G(j,k)}{\mathbf{N}.\mathbf{V}} \right) \qquad \text{E[2.2-13]}$$

$d, s \in \langle 0,1 \rangle$	Ratios of diffuse resp. specular behavior of the surface. To fulfill energy conservation law: $d + s = 1$.
$F(i)$	Fresnel term describing reflection from the perfect mirror.
$D(h)$	Facet slope distribution function. Could be dependent also on ϕ if material is anisotropic.
$G(j,k)$	Geometric attenuation factor GAF, which is taking masking and shadowing effects into consideration.

The Fresnel term is described in the section 2.2.1.1. The facet slope distribution function is used because the surface is modeled as the set of microfactes which are treated as perfect mirror reflectors. The distribution function then returns the reflectivity of the facets whose normals point in the same direction as **H** vector. The term **N.V** is expressing the fact that with an increasing viewing angle a larger area is projected into a viewing solid angle and

more facets contributes into a total surface radiance. Simultaneously this term is counteracted by the GAF, because the effect of masking becomes more dominant as the viewing angle approaches a grazing angle.

The Torrance & Cook's illumination model incorporates the Fresnel term so it can utilize the wavelength information which is available in a full spectral renderer. It satisfies the requirements of physical accuracy and if precise color calculations are required then it can provide reliable results.

2.2.2.1 Slope distribution function

The Trorrance & Cook's illumination model incorporates a microfacet surface roughness model. This model describes a surface as the set of v-shaped microfacets. These microfacets are oriented randomly. The slope distribution function then provides information about the fraction of the microfacets whose normals point in the same direction as **H** vector. Those facets are reflecting light directly into the direction of a viewer. To fulfill the energy conservation law the condition E[2.2-14] should be satisfied.

$$\int_0^1 2hD(h)dh = 1 \qquad \text{E[2.2-14]}$$

There are several distribution functions which can be used. Gaussian distribution E[2.2-15] was also mentioned in [Cook81] but the Beckman's distribution function E[2.2-16] was preferred. Beckman's function is similar to the Gaussian in shape and moreover gives the absolute magnitude of the reflectance without scaling constant. Also this function is energy conserving – it fulfills the condition E[2.2-14].

$$D(h) = ce^{-(h/\sigma)^2} \qquad \text{E[2.2-15]}$$

c	A scale constant. The distribution function must be scaled appropriately to get absolute reflectance.
σ	The roughness of a surface – small values for smooth surfaces and high values for rough surfaces.

$$D(h) = \frac{1}{\sigma^2 h^4} e^{\frac{h^2-1}{\sigma^2 h^2}} \qquad \text{E[2.2-16]}$$

Although the Beckman's distribution function seems to be complex it's still an approximation. Therefore a simpler function E[2.2-17] was proposed in [Schl94], which is easier to evaluate and still fulfills the condition E[2.2-14]. Comparison of those two functions is shown in Fig. 2-6. Some types of materials happen to have more then one facet distributions with different roughness then the sum of two or more distribution functions can be used but should be weighted so that the sum of all weights is one.

$$D(h) = \frac{h^2}{\left(\sigma h^4 - \frac{1}{2\sigma} h^4 + \frac{1}{2\sigma}\right)^2} \qquad 2.2\text{-}17$$

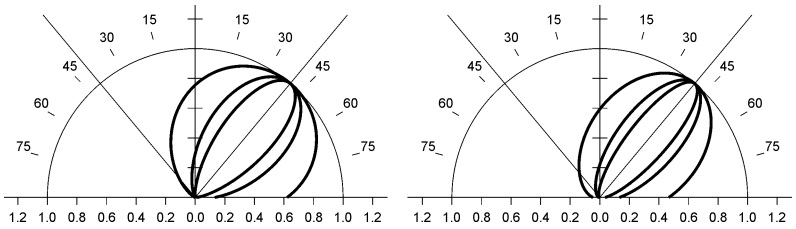

Fig. 2-6: Normalized slope distribution function for σ = 0.2, 0.3, 0.5. The Beckman's function – the left diagram and the Schlick's function – the right diagram.

2.2.2.2 Geometric attenuation factor

The geometric attenuation factor simulates the reduction of the light's amount that is reflected to a viewer due to a surface roughness. This roughness masks the incident and reflected light, see Fig. 2-7. As a result, the reflected light intensity is lowered at the grazing angles of θ and θ'.

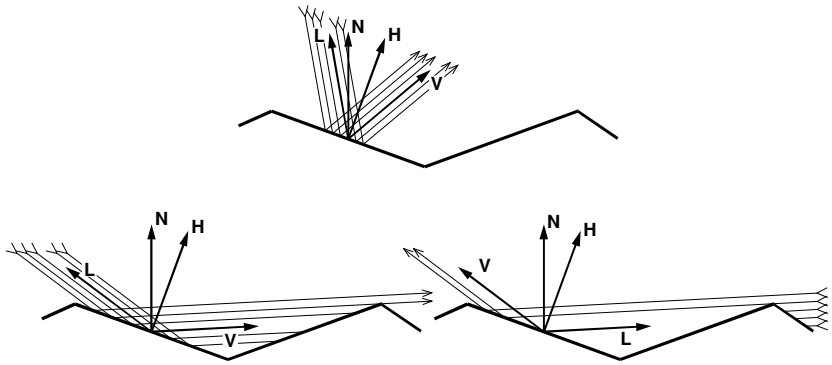

Fig. 2-7: Effect of masking (left) and shadowing (right). Picture taken form [Blinn77].

In [Cook81] was proposed use of the formulation E[2.2-18] that was derived originally by Torrance and Sparrow. However, this formulation has a several disadvantages:

- It is independent on the surface's roughness.
- Its first derivative is not continuous.
- It is not invariant to rotation around normal vector.

Better formulation E[2.2-18] of the GAF can be found in [He91], where the original formulation was rewritten in more compact version. It does not have disadvantages of the previous GAF formulation and it has been experimentally validated. Despite its complicated formulation the shape of the function is quite simple and therefore a simple approximation E[2.2-20] was proposed in [Schl94] to be use instead. In Fig. 2-8 is shown the comparison of the Torrance & Cook's and Shlick's GAF.

$$G(j,k) = \min\left[1, 2\frac{(h)(k)}{i}, 2\frac{(h)(j)}{i}\right] \qquad \text{E[2.2-18]}$$

$$G(j,k) = G(j)G(k)$$

$$G(x) = \frac{g}{g-1} \qquad \text{E[2.2-19]}$$

$$g = \sqrt{h\pi}\left(2 - erfc\sqrt{h}\right) \quad h = \frac{x^2}{4\sigma^2\left(1-x^2\right)}$$

$$G(x) = \frac{x}{x - kx + k} \quad k = \sqrt{\frac{2\sigma^2}{\pi}}$$ E[2.2-20]

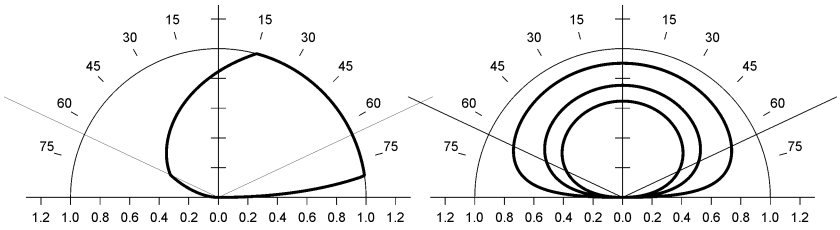

Fig. 2-8: GAF at 65 degree. The Torrance Cook GAF – the left diagram. The Shclick's GAF for roughness 0.1, 0.3 and 0.5 – the right diagram.

2.2.3 Blinn's Illumination Model

James F. Blinn took model as proposed by Torrance & Sparrow and adapted it for purposes of the CG. According to the [Blinn77] the illumination model is defined as E[2.2-21]. The Fresnel factor and the GAF can be used without changes as they are described in the previous section. The slope distribution function is tailored for the simplified BRDF requirements so the function is proportional. This means that its values are from range $\langle 0,1 \rangle$. Blinn in his work discussed the use of the Gaussian distribution with scale coefficient set to one however he chose the function E[2.2-22] proposed in [Trow75] for efficiency and accuracy reasons.

$$\rho = dC^D \mathbf{N.L} + s\frac{F(i)D(h)G(j,k)}{\mathbf{N.V}}$$ E[2.2-21]

$$D(h) = \left(\frac{\sigma^2}{h^2(\sigma^2 - 1) + 1}\right)^2$$ E[2.2-22]

$\sigma \in (0,1)$ The measure of a surface's roughness. The value 0 corresponds with more specular reflection and the value 1 with more diffuse reflection.

The Blinn's illumination model is designed as a simplified BRDF but it is functionally almost identical with the Torrance & Cook's illumination model so the notes mentioned in the previous section concerning its use in a spectral renderer are applicable in this case as well.

2.2.4 Strauss's Illumination Model

One disadvantage of the Toorance & Cook's or the Blinn's illumination model is that their parameters are not very intuitive and their values for particular material are usually acquired by measuring. Setting the parameters of those models to create a new material is an awkward task. Another problem is the possibility of setting contradictory values for some parameters e.g. a specular exponent should raise with the smoothness of a material, i.e. the specular coefficient. The value of a specularity coefficient is often unbounded and thus non-intuitive. With these issues on mind [Straus90] developed a user friendly nevertheless physically accurate illumination model.

There are two parameters controlling material appearance – smoothness *o* and metalness *p*. Smoothness modifies a reflection type form the perfect specular to the perfect diffuse. It also controls the brightness of a specular highlight. The metallness controls the color of the specular highlight and the amount of the diffuse reflection. The definition of the

BRDF E[2.2-23] is formally similar to the Phong's model definition E[2.2-5]. Different is the setting of the coefficients *d*, *s* and *n*. Those are not set by the user but are calculated form the values of metalness and smoothness parameters. The diffuseness coefficient *d* is computed as E[2.2-24], the exponent *n* is computed as E[2.2-25] and finally the specularness coefficient *s* is computed as E[2.2-26].

$$\rho = dC^D(\mathbf{N.L}) + sC^S(\mathbf{R.V})^n \qquad \text{E[2.2-23]}$$

$$d = r_d(1 - po) \qquad \text{E[2.2-24]}$$

$r_d = (1 - o^3)$ Amount of light reflected diffusely.

$$n = \frac{3}{1 - o} \qquad \text{E[2.2-25]}$$

$$s = \min[1, r_s + (r_s + k_s)\tau] \qquad \text{E[2.2-26]}$$

$r_s = 1 - r_d$ Amount of specularly reflected light.

$\tau = F(\theta)G(\theta)G(\theta')$ Fresnel and GAF terms.

k_s Off specular peak approximation for rough surface. Value 0.1 should provide reasonable results.

The Fresnel term as defined in 2.2.1.1 and the GAF as defined in 2.2.2.2 aren't used in the original Strauss model but are used their approximations E[2.2-27] instead. Fresnel term is formulated so that the reflectivity at normal incidence is set to a zero, which is an appropriate approximation for the materials which have a small index of refraction on their interface with air, e.g. water or glass. The tweaking constants are recommended according to [Straus90] to have values 1.5 for k_f and 1.1 for k_g. Arguments for both functions are from interval $\langle 0,1 \rangle$ and therefore have to be divided by $\pi/2$ first. Strauss also mentioned modified functions which have the cosine of angles as arguments but those functions weren't presented. The plot of both functions is shown in the Fig. 2-9.

$$F(x) = \frac{\frac{1}{(x - k_f)^2} - \frac{1}{k_f^2}}{\frac{1}{(1 - k_f)^2} - \frac{1}{k_f^2}}, \quad G(x) = \frac{\frac{1}{(1 - k_g)^2} - \frac{1}{(x - k_g)^2}}{\frac{1}{(1 - k_g)^2} - \frac{1}{k_g^2}} \qquad \text{E[2.2-27]}$$

k_f, k_g Functions' tweaking constants.

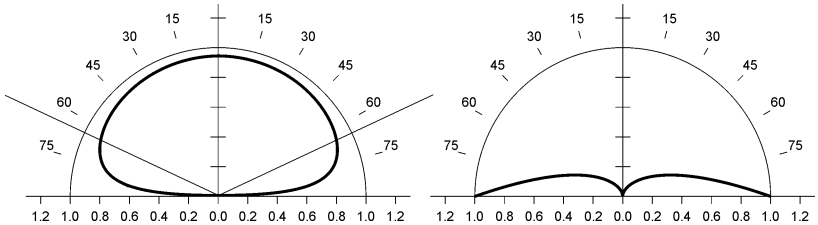

Fig. 2-9: Plot of the Strauss's GAF (left) and Fresnel (right) function.

Specular reflectivity C^S, which controls the highlight's color, is set in relation with the material's metalicity. In the case of dielectrics the specular highlight has usually the same

color as the light source which produced it. C^S is then set to 1 for all components. The situation is a little bit more difficult in the case of metals. The color of a highlight depends on a material's color and light's incident angle because of the Fresnel term, which increases reflectivity to one as the incident angle approaches a grazing angle. This effect can be approximated by the interpolation of surface reflectivity and 100 % reflectivity.

$$C^S = C^1 + p(1 - F(\theta))(C^D - C^1) \qquad \text{E[2.2-28]}$$

C^1 100 % reflectivity.

Strauss's model is a compromise between the accuracy and efficiency. Although it contains some advanced features like Fresnel term and GAF, they are implemented in an empirical way.

2.2.5 Oren & Nayar's Model

All discussed illumination models so far used the simple Lmbert's law of illumination for the evaluation of the diffuse part of the BRDF. This Lambert's law assumes ideal diffuse reflection which means uniform in every direction. However, the ideal diffuse reflection doesn't exist in a real world. In a real world the diffuse reflection is directional which means that the amount of reflected light depends on a viewer's position. The model is based on the same the same microfacet surface model as it was presented in the section 2.2.2.1. The effect of the roughness on the diffuse reflection is shown in the figure Fig. 2-10.

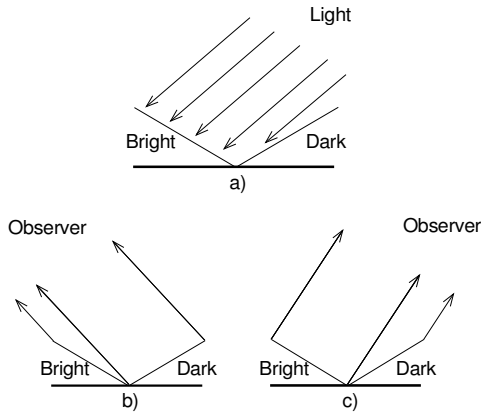

Fig. 2-10: The non lambertian behavior of a rough surface. In b) the dark part contributes to total radiance more than the bright part and vice versa in the c). [Oren-Nayar94a]

In [Oren-Nayar94a] was proposed a generalization of the lambertian reflection from rough surfaces. The complete model is quite sophisticated and a full derivation and description can be found in [Oren92]. For more practical use of the model was proposed an efficient and effective approximation E[2.2-29]. Significant property is that the model is reduced back to the lambertian if $\sigma = 0$. Comparison of the lambertian model and the Oren & Nayar's model is shown in the Fig. 2-11.

This model does not provide any special improvements for a spectral rendering. It is just an interesting alternative to the lambertian treating of the diffuse reflection.

$$\rho^D = d\left(A + B\max[0, C]\sin\alpha\tan\beta\right)\mathbf{N}.\mathbf{L}$$

$$A = 1.0 - 0.5\frac{\sigma^2}{\sigma^2 + 0.33} \qquad \text{E[2.2-29]}$$

$$B = 0.45\frac{\sigma^2}{\sigma^2 + 0.09}$$

$\alpha = Max(\theta', \theta)$

$\beta = Min(\theta', \theta)$

C — The cosine of the angle between vectors **L** and **V** projected into the plane with the normal **N.**
$C = (\mathbf{N}\times(\mathbf{L}\times\mathbf{N}))\cdot(\mathbf{N}\times(\mathbf{V}\times\mathbf{N}))$.

$\sigma \in \langle 0,1 \rangle$ — The roughness coefficient of a surface. 0 for smooth surfaces and 1 for rough surfaces.

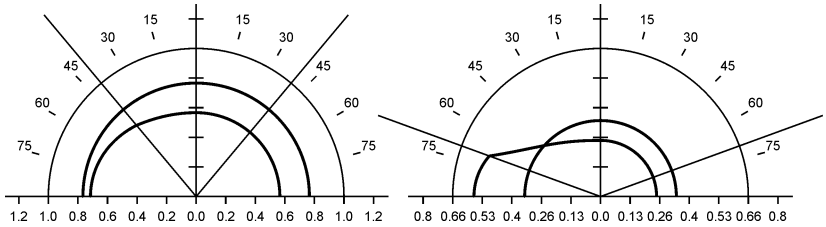

Fig. 2-11: Comparison of the Oren & Nayar diffuse model for roughness coefficient set to value 0.6 and ideal diffuse (roughness coefficient is 0.0). Angle of incidence is 40° on left and 70° on right.

2.2.6 Ward's Anisotropic Illumination Model

All discussed illumination models assumed that materials are isotropic, which means that their BRDF is not a function of the angle ϕ. Ward in his work [Ward92] proposed a gaussian anisotropic microfacet distribution function E[2.2-30]. Schlick in [Schl94] modified the distribution function so it fits into Torrance & Cook's model. Term h^4 was added into E[2.2-30] to fulfill the condition E[2.2-14]. According to [Schl94] the anisotropy can be implemented also into the simpler illumination model such as the Pohong's one if E[2.2-32] is used as an exponent of a highlight function.

Anisotropic treating of the specular reflection improves realism, but it does not inflict new possibilities in spectral rendering.

$$D(h,m) = \frac{1}{ab}e^{\frac{h^2-1}{h^2}\left(\frac{m^2}{a^2}+\frac{1-m^2}{b^2}\right)} \qquad \text{E[2.2-30]}$$

a, b — The roughness coefficients in a tangent respective binormal direction.

$$D(h,m) = \frac{1}{abh^4}e^{\frac{h^2-1}{h^2}\left(\frac{m^2}{a^2}+\frac{1-m^2}{b^2}\right)} \qquad \text{E[2.2-31]}$$

$$n = \frac{ab}{a - am^2 + bm^2} \qquad \text{E[2.2-32]}$$

2.2.7 He's Illumination Model

He and his allies designed very sophisticated illumination model [He91] which covers most of the effects occurring during light's interaction with a surface. It uses vector form of Kirchhoff diffraction theory so the model is capable to treat polarization and directional Fresnel effects properly. Surface tangent approximation was enhanced so it averages not only height distribution but the slope distribution as well.

Effect of shadowing and masking described in section 2.2.2.2 was incorporated into the surface tangent approximation which results into the introduction of the effective roughness of a surface. Effective roughness can be significantly smaller than the statistical roughness because of the shadowing and masking effect, especially at the grazing angles of incidence or reflection. Term describing the shadowing and masking effect was also enhanced so it has appropriate smoothness and symmetry.

This model is also separated into specular and diffuse parts. In contrast to the previously discussed models diffuse part consists of directional diffuse part, which is caused by the diffraction and interference effects when wavelength is comparable with the roughness elements, and the uniform diffuse part, which is caused by the multiple surface reflections and the subsurface reflection. Complete formal description of this model is included in [He91] and since it's very complicated its presentation wouldn't be much illustrative. However, those more brave ones are encouraged to explore it.

This model should be used in applications where quality of the final image is more important than the computation time. The model can handle polarization and is wavelength dependent so together with the spectral renderer it creates a very good solution for a wide selection of materials. However, its complexity makes it unsuitable for the real time implementation of the full spectral rendering.

2.3 Shading methods

Object geometry is often represented as a polygonal mesh and in that case is all information about surface's properties gathered in the vertices of such mesh and some sort of shading method must be used to compute the color inside the polygon.

There are two types of shading methods. The method of the first type evaluates an illumination model at several specific places, e.g. polygon vertices, and then the other values are interpolated from the known ones. Constant, Gouraud's and quadratic shading are representatives of this type. The advantage of this method is speed and the disadvantage is spatial aliasing. Aliasing is caused by the insufficient sampling rate of large polygons. The most obvious visual artifact is complete absence of the highlight if it's smaller than the polygon and none of the vertices interferes with it.

The shading method of the second type evaluates the illumination model for each rendered point of the polygon. This eliminates the problem of missing highlight but the illumination model must be evaluated at each pixel. Representative method of this type is Phong's shading.

2.3.1 Constant Shading

The constant shading is the simplest shading method possible. One color is used for all pixels of the rasterized polygon. Color can be estimated from the illumination model value at some point on the polygon or from average of the illumination model values at vertices.

Disadvantage of this method is discontinuity between adjacent polygons, which approximates curved surface. This method can be successfully used for fast preview which provides sufficient information about shape, position and rotation of the objects in a scene.

2.3.2 Gouraud's Shading

Gouraud's shading offers much better result than the constant shading because it utilizes linear interpolation to estimate intensity variation on a curved surface. One of the possible interpolation schemes is illustrated in the Fig. 2-12. It is used when scanline raster conversion of a triangle is utilized. Advantage of such interpolation is that it can be done using just one addition per pixel. Disadvantage is that it is not invariant to a rotation. This method smoothes the edges of adjacent polygons but not enough to avoid the Mach's bands effect, which is the optical rise of the intensity at places where intensity change its value to suddenly. However, this method provides sufficient results at low computing cost and is implemented in every HW graphics accelerator.

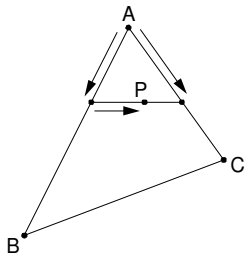

Fig. 2-12: Linear interpolation

2.3.3 Quadratic Interpolation

This method uses quadratic interpolation scheme to derive the pixel intensity. It increases the smoothness of the intensity change which makes Mach's band effect less obvious. Quadratic interpolation needs more samples. In the case of triangle there are 6 samples LMNQRS, see Fig. 2-13. Three samples are at the vertices and three are in the middle of each edge. From these samples are derived six coefficients ABCDEF used for quadratic interpolation scheme E[2.3-1]. There are a several methods for computing the coefficients. According to [Hast02], the best is method is E[2.3-3]. Quadratic interpolation provides better results than the Gouraud's shading due to double sample count and the speed of the evaluation is still acceptable if forward differences are used.

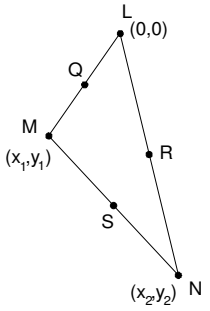

Fig. 2-13: Situation for the quadratic interpolation

$$\Phi = Ax^2 + By^2 + Cxy + Dx + Ey + F \qquad \text{E[2.3-1]}$$

$$
\begin{aligned}
&w_1 = x_1 y_2, \\
&w_2 = x_2 y_1, \\
&w_3 = w_1 - w_2, \\
&T = L + M - 2Q \\
&U = L + N - 2R \\
&V = L + S - R - Q \\
&G = 2Q - 2L - T \\
&H = 2R - 2L - U \\
&A = 2\left(T y_2^2 + U y_1^2 - 2V y_1 y_2\right) / w_3^2 \\
&B = 2\left(T x_2^2 + U x_1^2 - 2V x_1 x_2\right) / w_3^2 \\
&C = 4\left(V\left(w_1 + w_2\right) - U x_1 y_2 - T x_2 y_2\right) / w_3^2 \\
&D = \left(G y_2 - H y_1\right) / w_3 \\
&E = \left(H x_1 - G x_2\right) / w_3 \\
&F = L
\end{aligned}
\qquad \text{E[2.3-2]}
$$

2.3.4 Phong's Shading

Phong's shading described in [Phong74] evaluates illumination model in each pixel of the rasterized polygon. The normal of a surface needed for evaluation is interpolated from the normals in the polygon vertices similar to the color in Gouraud's shading. This method is the most accurate and the most expensive one. However, with the introduction of programmable pipeline in graphics accelerators this method is applicable even in real time applications.

3 Spectral rendering

The important application of CG is synthesis or rendering of the real world's images. These images are then used for preview and presentation purposes in architecture, design or for entertainment in movies or games. The process of rendering is a very complex task involving light transport simulation followed by evaluation of light's interaction with surfaces. The light which reaches a virtual camera then determines the pixel's color in the synthesized image. As one might expect the accuracy of light's behavior simulation along its path to a camera greatly influences the realism of the synthesized image. However the accuracy is significantly reduced if the simulation is performed only for three wavelengths of the visible spectrum. Those three wavelengths usually correspond to the red green and blue color sensation and it is widely used among the render systems. Justification for such under sampling is trichromatic color representation described in the section 3.1.2.

The trichromatic color calculation would be accurate if all light sources radiate only at the same wavelengths as the ones used as primaries of the used trichromatic system. However that would be very special case. Light sources in a real world have continuous emission spectrum and materials have continuous spectral reflectance. Now, if some interesting interaction occurs outside the sampled wavelengths then it is completely missed and the result of the calculation is a wrong color. Applications where color matching under different light sources is essential, such as interior design or architecture, have to perform color calculation which takes the whole spectrum range into consideration.

Practically the spectral rendering is only extension of the common trichromatic rendering. Illumination is simply evaluated at more then three wavelengths. As one might expect, spectral rendering is more expensive then the trichromatic one. Visible light's wavelength falls into the range approximately from 400 nm to 700 nm. If sampling step 5 nm is used then a total of 61 samples have to be computed. It makes the spectral computation 20 times more expensive then the trichromatic one.

There are several methods which try to make representation of the spectral information as simple as possible and thus speed up the illumination calculations. One such method uses polynomial representation but it is adequate solution only for the smoothly varying spectra which are rather rare. More generic method proposed in [Peer93] uses several ortho-normal basis functions to represent any spectrum in a form of their weighted sum. The weights are then the only information used for illumination purposes. Method is described in detail in the section 3.3.1.

Another interesting method is to pre-compute the materials' apparent color's trichromatic representation under some dominant light source of a scene and then use this representation as a reflectance for the standard trichromatic renderer. Method is described in the section 3.3.2.

Once the light's full spectral characteristic is computed for each pixel of the synthesized image then it can be converted into its trichromatic representation. This representation is more convenient since most of the displaying systems also use trichromatic color representation so the color values are displayed directly only with some minor modifications. Also trichromatic representation has smaller storage requirements.

Conversion from spectral to trichromatic representation is not computationally simple. It involves integration for each primary color and once it's done the values must be

normalized and modified to reflect the specifics of the human eye's color perception that includes chromatic and intensity adaptation. The mentioned modification is necessary because the image provided by human eye is often very different form the physical reality. Specifics of the human vision are described in the section 3.1.1.

The text above implies that the rendering is all about color. Color perception is very complex phenomena and color description and representation is a complete scientific area. Some of the basics are described in the following section.

3.1 Color Theory

Color is essentially bonded with the human's vision system. Sensation of a color starts in an eye, which is sensitive to an electromagnetic radiation in a range of wavelengths from 380 nm to 780 nm called light. Light is registered by the photoreceptors inside the eye and transformed into electric impulses which are then processed by a brain. Details concerning human eye can be found in the section 3.1.1.

Different wavelength evokes different color sensation. This can be demonstrated by decomposition of a white or sun light with a prism or diffraction grating. Color sensation varies from red on the longer wavelengths side of a spectrum corresponding to 650 nm across orange at 600 nm, yellow at 580 nm, green at 550 nm, cyan at 500 nm, blue at 450 to violet at 400 nm.

The white color decomposition also shows the effect of color mixing. Mix of all wavelengths evokes sensation of the white color, mix of the red and green light then evokes sensation of the yellow color and there are a lot of other combinations. This feature is the base for the trichromatic color representation.

Color perception is clearly subjective. There are several institutions which try to standardize color perception and description so it can be used as a reference for industrial needs. One such institution is CIE - Commission Internationale de l'Eclairage. Its standard observers and standard illuminants are widely used. CIE colorimetry is described in the section 3.1.2.3.

3.1.1 Human Eye

The interface between the real world and its subjective human vision is an eye. The eye receives light and converts it into the signals which are then analyzed and interpreted by the brain as color information. Full description of the process of light's information to the color sensation is required in order to calculate and present colors properly. Unfortunately, this description is available only partially and some parts are only estimations of the real thing. Presented information was taken from [Wysz00], [Palm03] and [Kraus98].

Anatomy of the human eye is illustrated in the Fig. 3-1. Light enters the eye through cornea a transparent protective layer, which is attached to the sclera an outer protective envelope of the eye. Light is then focused on the retina by the lens. Amount of light reaching the retina is controlled by the iris. Between the cornea and the lens is a space filled with a clear liquid, the aqueous humor, and the space between the lens and the retina is vitreous body, which consists of a transparent jelly.

The retina is the place where photoreceptors are placed. It is a complex and multilayered structure covering most of the area of the choroid, the vascular and pigmented

layer attached to the sclera. In the first layers are located two kinds of photoreceptors, the cones and rods. The second layer of intermediate neurons contains bipolar, horizontal and amacrine cells. Finally the third layer contains ganglion cells.

Between the first and the second layer the first synaptic layer is located, which interconnects photoreceptors and intermediate neurons. There is also the second synaptic layer between the second and the third layer, which interconnects intermediate neurons and the ganglion cells. There are about 2×10^8 nerve cell of different kinds located in retina. They are all involved with the processing of the visual information reaching the retina.

Incoming light goes through all the layers of the retina until it finally reaches the outer segment of the photoreceptors where it is absorbed. This event triggers the complex processing of the signal through the neural network. Signal then continues from the ganglion cells in the optic nerve fibers to the brain where it is interpreted as a color sensation. Signal is processed differently depending on the type of the photoreceptor which produced it.

Signal generated by the rod photoreceptor is transferred through the synapses to a horizontal cell, synaptic body of a cone receptor and primarily to the rod-bipolar cell. Consequently the signal is transmitted through the synaptic body of the rod-bipolar cell to the diffuse ganglion cell. There is the signal further processed along with the other signals arriving from other cell of the neural network and finally continues to the brain.

Several neighboring rods feed their signal into a single rod-bipolar cell. The same rods are also connected with one horizontal cell along with other rods from the different area of the retina. The presence of the bipolar cells, which are not connected with other cells in the intermediate neurons layer and ganglion cells, indicates that the lateral processes may exist in the first synaptic layer.

Cone photoreceptors have considerably larger synaptic body then those of the rod pohotoreceptors'. Synapses emerge from the synaptic body and through one such synapse is the signal transferred to a cone-bipolar cell called a midget bipolar. The signal is transferred through another synapse to a horizontal cell described above.

Several cones, 7 to 12 are interconnected by another cone-bipolar cell called flat bipolar. Signals from the midget and flat bipolar cell continues to the second synaptic layer and then to the ganglion cells. Midget bipolar cells are usually connected with a midget ganglion cell while flat bipolar cells are connected with both the diffuse and midget ganglion cells.

The last not mentioned kind of cells are the amacrine cells, which have the similar purpose in the second synaptic layer as the horizontal cells in the first synaptic layer. They are introducing lateral processing in the second synaptic layer.

As mentioned before, there are two types of the photoreceptors, the cones and rods. The main difference between them is, except the shape and their wiring with the neural network described above, in their outer segment, where the interaction of visual pigment and incident light occurs. The visual pigment called rhodopsin has different properties for each type of the photoreceptor. Furthermore there are three types of cone pigments with different absorptance maxima in the short, middle and long wavelength region (approx. S - 445 nm, M - 535nm, L - 575nm).

In general rods are preferred source of information when an eye is adapted to a dark, i.e. luminance is lower then 0,01 cd/m^2. This kind of vision is then called scotopic. Cones are then preferred source of information when eye is adapted to light, i.e. luminance is greater then 3 cd/m^2. This kind of vision is then called photopic [Palm03]. Three types of the cone photoreceptors makes the cones responsible for the color vision although the synaptic connections between cones and rods described previously suggests that some sort of interaction may be possible.

There are approximately $5\times10^6 - 6\times10^6$ cones in the retina. Their biggest concentration is at a place called fovea. The fovea is a spot with about 1.5 mm in diameter located around visual axis. In that spot is the biggest density of the cones. There are packed about $1\times10^5 - 1.5\times10^5$ cones and other layers of the retina are thinner there. In the center of the fovea is a spot with 0.4 mm in diameter called foveola and is referenced as the center of vision. There are approximately 2.10^4 cones and no rods. The cones in the foveola are longer then the cones in other places on retina, the number of pathways to brain is greater and the other layers of retina are almost absent. These features make foveola the most sensitive spot of the retina.

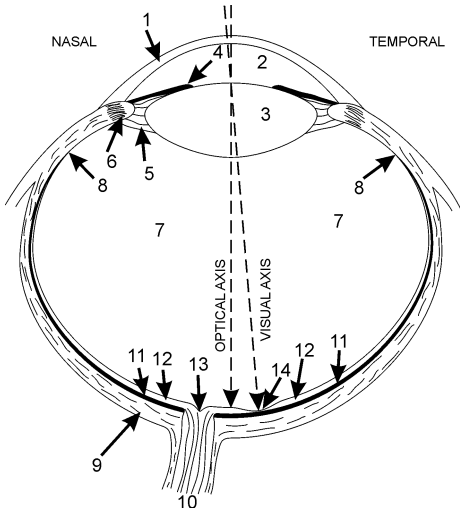

Fig. 3-1: Schematic near-horizontal median section of the right eye seen from above. (1) cornea; (2) aqueous humor; (3) eye lens; (4) iris; (5) zonule fibers; (6) ciliary muscle; (7) vitreous body; (8) ora serrata (front edge of the retina); (9) sclera (outer coat of the eyeball); (10) optic nerve; (11) choroids; (12) retina; (13) optic disk (papilla); (14) fovea. Figure taken from [Wysz00].

There are approximately 1.2×10^8 rods in retina. Rods are quite uniformly distributed across the retina except the area around foveola where they are missing completely. They are responsible for the peripheral vision. Rods lack color information and has lower resolution than cones at fovea. On the other hand, they are more sensitive then the cones. They are capable to react practically on one photon.

From the brief description of the signal processing in the retina's neural network is obvious that it is very complex system and it is difficult to explore it and describe it appropriately. There are several theories, which are based on speculations and estimations. Following information taken from [Wysz00]. One such theory called three components or trichromatic theory was commonly accepted. The method assumes that three different cone types exist with different spectral sensitivity. Signals generated form these cones are then transmitted directly into the brain where color sensation directly related to those three cone signals is experienced. However this theory is incapable to explain why an observer sees

yellow color as a result of the appropriate mixture of the red and green stimulus when yellow is clearly different sensation compared with the pure red one and the pure green one. Similar situation is the mixture of the blue and yellow which leads to the white stimulus.

Another widely accepted theory is the opponent colors theory. It is more successful in explanation of the color perception. This theory assumes existence of two opposite kinds of neural signals. These two signals provide variety of hues of varying brightness and saturation however combination limited. In other words, system provides hue attributes described by names: red, green, yellow, blue, furthermore hues corresponding to red-yellow, yellow-green, green-blue and blue-red combinations. Other combinations like reedish-green or yellowish-blue are never experienced.

Nevertheless neither of the presented theories strictly taken is capable of explanation of all color vision phenomena. However, the situation is much better if those two theories are merged into one theory called a zone theory. This theory explains and predicts many color vision phenomena.

The zone theory assumes that in the first zone there are located three independent types of cones. The color vision is initiated there by the absorption of light in the photo pigments and consequent conversion into electrical signals. This part is in match with the trichromatic theory. In the second zone are the signal processed by the neural network and three new signals are generated; one achromatic signal and two antagonistic chromatic signals (see Fig. 3-2). This is the part form the opponent color theory. It is assumed that other stages of signals' processing are performed along the way to the brain but those stages are not yet explored. The final zone is located in the cortex where the signals are interpreted in the context of other visual information received at the same time and in the context of previously accumulated visual experience.

Fig. 3-2: Two zones of the visual pathway. Two chromatic channels on the left and achromatic signal is on the right. Figure taken from [Kraus98].

Human eye operates over a 10^{10} range of light luminance although the dynamic range over which neurons operate is about 10^2. This is possible because of the high adaptability of the eye. One of the causes of his adaptability is existence of two receptoral systems - rods and cones described above. Then there are mechanisms, not yet completely understood, which are controlling signals strength along their way to the brain.

The sensitivity of the first zone, in which are signals determined by the excitation of cones, varies with the level of excitation of the cones. It is a protection from signal overload. System is adjusting signal gain according to the level of the input. Effect of this regulation can be demonstrated by staring at a colored pattern for a while. Then if a gaze is moved on a field of uniform white then the after image of the pattern is seen in complementary colors.

The sensitivity of the second zone is altered according to the history of the adaptation mechanisms. If the observer is exposed to a light varying between maximally saturated opposing colors like red and green then its more difficult for the observer to see changes of

saturation in the red and green directions but not in the yellow and blue directions and vice versa. [Kraus98]

From the text above it is obvious that color vision system is not linearly related to the radiometric quantities. The color vision system is actually modifying reality. For this reason it is very difficult to synthesize image with the exactly same colors as they would be perceived by the eye. And even if the image is finally somehow synthesized then the colors may appear to be different if the image is viewed under different viewing conditions.

3.1.2 Colorimetry

The colorimetry is concerned with description of the color of a physically defined visual stimulus [Wysz00]. The description is done by the means of experimental laws of color matching covered in the trichromatic generalization. The trichromatic generalization states that many of the color stimuli can be matched in color with an additive mixture of three primary stimuli whose radiant powers have been adequately adjusted. Other color stimuli have to be mixed with one of the primary before a match with the mixture of the other two stimuli is obtained and other stimuli must be mixed with two primaries before the match with the third stimuli is obtained. The primary stimuli can be chosen quite freely as long as no primary stimuli can be matched by mixing the other two primary stimuli.

The configuration of the color matching experiment is following. There is a light patch of such size that it subtends the required viewing angle. The light patch is divided into halves. The matched color stimulus is placed on the first half and the current mixture of the primary stimuli is placed on the second one. Then the intensity of each primary is adjusted until the complete match is obtained.

To make the color matching more objective the observing condition and observer's parameters must be taken into consideration. The subtending viewing angle is usually set so that the image of the patch is projected on the area of the fovea only and therefore only cones are involved. Other factors influencing the matching procedure are related with the adaptability of the eye. Previous exposure of the observer to light may affect the precision of the color matching.

Finally, the color match of two stimuli is subjective to the observer. This subjectivity is eliminated by the introduction of the standard observer, which is specified as an average of the normal observers. Such observer was specified by the Commission Internationale de l'Eclairage - CIE in 1931. CIE standard observer is discussed in the section 3.1.2.3.

This section is mostly based on [Wysz00].

3.1.2.1 Color Matching

The color matching, as introduced above, is a process of finding appropriate amounts of three primary stimuli **R**, **G**, **B** such that their mixture matches the color stimuli **Q** – E[3.1-1]. The amounts *R*, *G*, *B* are called tristimulus values of **Q**. Those stimuli which have to be mixed with one of the primary stimuli before the match with the mixture of the other two is obtained and those stimuli which have to be mixed with two of the primary stimuli to obtain the match with the remaining primary stimulus have one or two tristimulus values negative – E[3.1-2] and E[3.1-3].

There is a geometrical interpretation of the equations E[3.1-1], E[3.1-2] and E[3.1-3]. The primary stimuli **R**, **G**, **B** define a three dimensional space. Stimulus **Q** is represented as a vector in that space and the tristimulus values *R*, *G*, *B* of **Q** are the vector's components. The length of the vector then represents the intensity of the stimulus **Q**. The three dimensional case is usually converted into two dimensional case in the unit plane $R + G + B = 1$. Each vector **Q** or its extension must intersect this plane. The intersection point *Q*, called the point of chromacity, in the plane then determines the direction of the **Q** but information about the intensity is lost. Relation of coordinates *r, g, b* of *Q*, called chromacity coordinates, with the tristimulus values *R*, *G*, *B* is expressed in E[3.1-4]. Because of the property $r + g + b = 1$ only two coordinates sufficiently describe the chromacity and those coordinates are plotted in the chromacity diagram where *r* and *g* coordinate axis are perpendicular, see Fig. 3-4.

$$\mathbf{Q} = R\mathbf{R} + G\mathbf{G} + B\mathbf{B} \qquad \text{E[3.1-1]}$$

$$\mathbf{Q} + R\mathbf{R} = G\mathbf{G} + B\mathbf{B} \Rightarrow \mathbf{Q} = -R\mathbf{R} + G\mathbf{G} + B\mathbf{B} \qquad \text{E[3.1-2]}$$

$$\mathbf{Q} + R\mathbf{R} + G\mathbf{G} = B\mathbf{B} \Rightarrow \mathbf{Q} = -R\mathbf{R} - G\mathbf{G} + B\mathbf{B} \qquad \text{E[3.1-3]}$$

$$r = \frac{R}{R+G+B}, \quad g = \frac{G}{R+G+B}, \quad b = \frac{B}{R+G+B}, \quad r+g+b = 1 \qquad \text{E[3.1-4]}$$

$$\mathbf{E}_\lambda = \bar{r}(\lambda)\mathbf{R} + \bar{g}(\lambda)\mathbf{G} + \bar{b}(\lambda)\mathbf{B} \qquad \text{E[3.1-5]}$$

$$\mathbf{Q}_\lambda = R_\lambda\mathbf{R} + G_\lambda\mathbf{G} + B_\lambda\mathbf{B} \qquad \text{E[3.1-6]}$$

$$\mathbf{Q}_\lambda \equiv \Phi(\lambda)d\lambda\mathbf{E}_\lambda = \Phi(\lambda)d\lambda\bar{r}(\lambda)\mathbf{R} + \Phi(\lambda)d\lambda\bar{g}(\lambda)\mathbf{G} + \Phi(\lambda)d\lambda\bar{b}(\lambda)\mathbf{B} \qquad \text{E[3.1-7]}$$

$$R = \int_{380}^{780}\Phi(\lambda)\bar{r}(\lambda)d\lambda, \quad G = \int_{380}^{780}\Phi(\lambda)\bar{g}(\lambda)d\lambda, \quad B = \int_{380}^{780}\Phi(\lambda)\bar{b}(\lambda)d\lambda \qquad \text{E[3.1-8]}$$

The color stimulus **Q** has a spectral power distribution Φ_Q. It is defined on the range of the visible light wavelengths (380 – 780). If this range is divided into *n* wavelength intervals then the quantity $\Phi_Q(\lambda)\Delta\lambda$ represents the power in the wavelength interval $\Delta\lambda$ and defines a monochromatic stimulus of wavelength λ labeled as $\mathbf{Q}_\lambda$ (λ is usually the center of the $\Delta\lambda$ interval). For each monochromatic stimulus $\mathbf{Q}_\lambda$ applies the equation E[3.1-6] and the tristimulus values R_λ, G_λ, B_λ are then called spectral tristimulus values.

There is one special set of the spectral tristimulus values obtained from stimulus **Q** whose spectral stimuli obeys the condition: $\forall\lambda : \mathbf{Q}_\lambda = 1$. Such stimulus is called the equal-energy stimulus denoted by **E** and the spectral stimulus $\mathbf{E}_\lambda$ is matched as in E[3.1-5]. The labeling of the spectral tristimulus values $\bar{r}(\lambda), \bar{g}(\lambda), \bar{b}(\lambda)$ used in E[3.1-5] is standard for this special case. The spectral tristimulus values characterize the color matching properties of the observing eye for the set of primary stimuli **R**, **G**, **B** and therefore are called color-matching functions.

If both sides of E[3.1-5] are multiplied by $\Phi(\lambda)d\lambda$ then the equation E[3.1-7] represents the color matching of monochromatic stimulus $\mathbf{Q}_\lambda$ of radiant power $\Phi(\lambda)d\lambda$. If

spectral power distribution $\Phi(\lambda)$ is non-negative continuous function within the visible wavelength range then E[3.1-7] can be integrated and the tristimulus values *R*, *G*, *B* for stimulus **Q** are computed as E[3.1-8].

$$\begin{aligned}\mathbf{R}' &= a_{11}\mathbf{R} + a_{21}\mathbf{G} + a_{31}\mathbf{B}\\ \mathbf{G}' &= a_{12}\mathbf{R} + a_{22}\mathbf{G} + a_{32}\mathbf{B}\\ \mathbf{B}' &= a_{13}\mathbf{R} + a_{23}\mathbf{G} + a_{33}\mathbf{B}\end{aligned} \qquad \text{E[3.1-9]}$$

$$\begin{aligned}R &= a_{11}R' + a_{12}G' + a_{13}B'\\ G &= a_{21}R' + a_{22}G' + a_{23}B'\\ B &= a_{31}R' + a_{32}G' + a_{33}B'\end{aligned} \qquad \text{E[3.1-10]}$$

$$\begin{aligned}R' &= b_{11}R + b_{12}G + b_{13}B\\ G' &= b_{21}R + b_{22}G + b_{23}B\\ B' &= b_{31}R + b_{32}G + b_{33}B\end{aligned} \qquad \text{E[3.1-11]}$$

It is possible to choose the primary stimuli quite freely as long as none of the primaries can be matched with the mix of the other two. Any two sets of primary stimuli are linearly related and the transformation from one set **R**, **G**, **B** to the other set **R'**, **G'**, **B'** is governed by the equations E[3.1-9] – E[3.1-12].

The primaries of one set can be matched with the mixture of the primaries of the other set and the transformation can be expressed as matrix $\mathbf{A}^*$ with a coefficients a_{ki} (i,k = 1, 2, 3) – E[3.1-9]. Rows of the matrix $\mathbf{A}^*$ are tristimulus values of **R'**, **G'**, **B'** in **R**, **G**, **B**. Relation between tristimulus values *R*, *G*, *B* of a color stimulus in **R**, **G**, **B** and tristimulus values *R'*, *G'*, *B'* in **R'**, **G'**, **B'** are related as E[3.1-10] where the matrix **A** is a transpose of $\mathbf{A}^*$.

The opposite transformation of tristimulus values *R*, *G*, *B* to *R'*, *G'*, *B'* is then described in E[3.1-11] where the coefficients b_{ik} (i,k = 1, 2, 3) are the elements of the matrix $\mathbf{A}^{-1}$, which is the inverse matrix of the matrix **A**. The matrix $\mathbf{A}^{-1}$ can be also used for the special set tristimulus values $\bar{r}(\lambda), \bar{g}(\lambda), \bar{b}(\lambda)$ as shown in E[3.1-12]. From E[3.1-11] can be expressed the chromacity coordinates *r'*, *g'*, *b'* in the terms of *r*, *g*, *b* – E[3.1-13]. Using the identities $r + g + b = 1$ and $r' + g' + b' = 1$ the relation E[3.1-14] is obtained and the inverse transformation is expressed using the E[3.1-15].

$$\begin{aligned}\bar{r}'(\lambda) &= b_{11}\bar{r}(\lambda) + b_{12}\bar{g}(\lambda) + b_{13}\bar{b}(\lambda)\\ \bar{g}'(\lambda) &= b_{21}\bar{r}(\lambda) + b_{22}\bar{g}(\lambda) + b_{23}\bar{b}(\lambda)\\ \bar{b}'(\lambda) &= b_{31}\bar{r}(\lambda) + b_{32}\bar{g}(\lambda) + b_{33}\bar{b}(\lambda)\end{aligned} \qquad \text{E[3.1-12]}$$

$$\begin{aligned}r' &= \frac{b_{11}r + b_{12}g + b_{13}b}{(b_{11} + b_{21} + b_{31})r + (b_{12} + b_{22} + b_{32})g + (b_{13} + b_{23} + b_{33})b}\\ g' &= \frac{b_{21}r + b_{22}g + b_{33}b}{(b_{11} + b_{21} + b_{31})r + (b_{12} + b_{22} + b_{32})g + (b_{13} + b_{23} + b_{33})b}\\ b' &= \frac{b_{31}r + b_{32}g + b_{33}b}{(b_{11} + b_{21} + b_{31})r + (b_{12} + b_{22} + b_{32})g + (b_{13} + b_{23} + b_{33})b}\end{aligned} \qquad \text{E[3.1-13]}$$

$$r' = \frac{\beta_{11} r + \beta_{12} g + \beta_{13}}{\beta_{31} r + \beta_{32} g + \beta_{33}}$$
$$g' = \frac{\beta_{21} r + \beta_{22} g + \beta_{23}}{\beta_{31} r + \beta_{32} g + \beta_{33}} \qquad \text{E[3.1-14]}$$

$$\beta_{11} = b_{11} - b_{13}, \beta_{12} = b_{12} - b_{13}, \beta_{13} = b_{13}$$
$$\beta_{21} = b_{21} - b_{23}, \beta_{22} = b_{22} - b_{23}, \beta_{23} = b_{23}$$
$$\beta_{31} = (b_{11} - b_{13}) + (b_{21} - b_{23}) + (b_{31} - b_{33})$$
$$\beta_{32} = (b_{12} - b_{13}) + (b_{22} - b_{23}) + (b_{32} - b_{33})$$
$$\beta_{33} = b_{13} + b_{23} + b_{33}$$

$$r = \frac{\begin{vmatrix} r' & \beta_{12} & \beta_{13} \\ g' & \beta_{22} & \beta_{23} \\ 1 & \beta_{32} & \beta_{33} \end{vmatrix}}{\begin{vmatrix} \beta_{11} & \beta_{12} & \beta_{13} \\ \beta_{21} & \beta_{22} & \beta_{23} \\ \beta_{31} & \beta_{32} & \beta_{33} \end{vmatrix}}, g = \frac{\begin{vmatrix} \beta_{11} & r' & \beta_{13} \\ \beta_{21} & g' & \beta_{23} \\ \beta_{31} & 1 & \beta_{33} \end{vmatrix}}{\begin{vmatrix} \beta_{11} & \beta_{12} & \beta_{13} \\ \beta_{21} & \beta_{22} & \beta_{23} \\ \beta_{31} & \beta_{32} & \beta_{33} \end{vmatrix}} \qquad \text{E[3.1-15]}$$

3.1.2.2 Color Metamerism

In colorimetry two stimuli **Q** and **Q'** are in match if they have identical tristimulus values. However, the spectral power distributions Φ_Q and $\Phi_{Q'}$ do not need to be the same. Such color stimuli are then called metameric color stimuli often referred as metamers. This effect is used almost in every color reproduction applications.

Metamerism is significant factor when colors are evaluated in illumination model because two materials with different spectral distributions of a reflectance may happen to have the same apparent color under one illuminant but different colors under other illuminant and similarly two illuminants with different spectral power distributions may happen to have the same color but one material can have different color when illuminated by each one of them. These effects cannot be simulated in trichromatic renderer.

3.1.2.3 CIE Standard Observer

The applied colorimetry is built on standards created by CIE. This organization produced in 1931 the CIE 1931 Standard Colorimetric Observer. This observer is defined by the color-matching functions $\bar{x}(\lambda), \bar{y}(\lambda), \bar{z}(\lambda)$ (see Fig. 3-3). Matching functions were obtained from the color-matching experiments where the viewing angle subtended 2 degrees to avoid the participation of the rod receptors. Such matching functions are suitable for applications where viewing angle is from one to four degrees.

In 1964 CIE recommended an alternative color matching functions $\bar{x}_{10}(\lambda), \bar{y}_{10}(\lambda), \bar{z}(\lambda)_{10}$, see Fig. 3-3, for applications where angular subtense larger then four degrees is needed. These color-matching functions, based on the color matching experiment performed with 10 degrees viewing angle, define the CIE 1964 Supplement Standard Colorimetric Observer.

The matching functions $\bar{r}(\lambda), \bar{g}(\lambda), \bar{b}(\lambda)$ yield negative values for some wavelengths. This is inconvenient when equation E[3.1-8] is evaluated. Also determination of the photometric quantities such as luminance needs integration using the luminous efficiency function $V(\lambda)$, which is a function in photometry used as brightness matching function. For this reasons were **R**, **G**, **B** primary stimuli replaced with the **X**, **Y**, **Z**.

The **X**, **Y**, **Z** stimuli are imaginary however the triangle formed by the chromacity points encloses the spectrum locus and the purple line completely therefore the chromacity coordinates *x*, *y*, *z* and the corresponding tristimulus values *X*, *Y*, *Z* of any real color stimulus are never negative. Another important property of the **X**, **Y**, **Z** stimuli is that the matching function $\bar{y}(\lambda)$ is identical with the luminous efficiency function $V(\lambda)$.

Relation between chromacity coordinates in RGB and in XYZ is given by the equation E[3.1-16]. This conversion is valid for monochromatic primary stimuli **R** = 700.0 nm, **G** = 546.1 nm and **B** = 435.8 nm. Relation of the chromacity coordinates x, y, z and the tristimulus values X, Y, Z is expressed in equation E[3.1-17] and color matching functions $\bar{r}(\lambda), \bar{g}(\lambda), \bar{b}(\lambda)$ are obtained from equation E[3.1-18].

$$x = \frac{0.49000r + 0.31000g + 0.20000b}{0.66697r + 1.13240g + 1.20063b}$$
$$y = \frac{0.17697r + 0.81240g + 0.01063b}{0.66697r + 1.13240g + 1.20063b} \qquad \text{E[3.1-16]}$$
$$z = \frac{0.00000r + 0.01000g + 0.99000b}{0.66697r + 1.13240g + 1.20063b}$$

$$X = \frac{x}{y}V, \quad Y = V, \quad Z = \frac{z}{y}V \qquad \text{E[3.1-17]}$$

$$\bar{x}(\lambda) = \frac{x(\lambda)}{y(\lambda)}V(\lambda), \quad \bar{y}(\lambda) = V(\lambda), \quad \bar{z}(\lambda) = \frac{z(\lambda)}{y(\lambda)}V(\lambda) \qquad \text{E[3.1-18]}$$

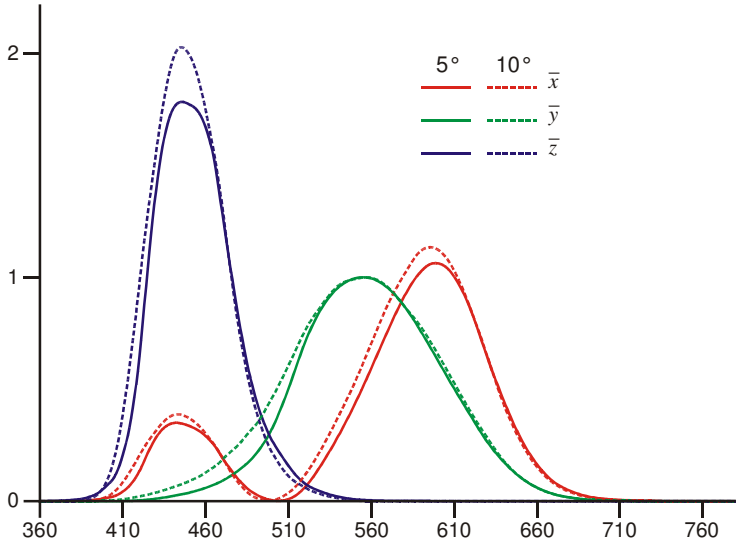

Fig. 3-3: Matching functions for 10° and 5° CIE observers.

3.1.2.4 Chromacity Diagram

As already mentioned in the section 3.1.2.1 the vectors representing a color stimulus in the three dimensional space defined by the primary stimuli or its extension intersects the unity plane $R + G + B = 1$. This intersection point called the chromacity point has chromacity coordinates r, g, b. Because of the property r + g + b = 1, each coordinate can be computed from the other two. Therefore the chromacity points can be plotted in the diagram with the orthogonal axis r and g. Such diagram is called chromacity diagram, see Fig. 3-4.

$$r(\lambda)=\frac{\bar{r}(\lambda)}{\bar{r}(\lambda)+\bar{g}(\lambda)+\bar{b}(\lambda)},\quad g(\lambda)=\frac{\bar{g}(\lambda)}{\bar{r}(\lambda)+\bar{g}(\lambda)+\bar{b}(\lambda)}$$
$$b(\lambda)=\frac{\bar{b}(\lambda)}{\bar{r}(\lambda)+\bar{g}(\lambda)+\bar{b}(\lambda)},\quad r(\lambda)+g(\lambda)+b(\lambda)=1 \qquad \text{E[3.1-19]}$$

Not every point in the diagram represents a real stimulus. Points representing real stimuli are closed by the boundary consisting of the spectral locus and the purple line. The chromacity coordinates of the points of the locus are obtained from the matching functions $\bar{r}(\lambda), \bar{g}(\lambda), \bar{b}(\lambda)$ using the equation E[3.1-19]. Purple line then connects the two ends of the spectral locus. It represents color stimuli obtained from the mixture of the extremely long stimulus (700 nm) and the extremely short stimulus (380 nm).

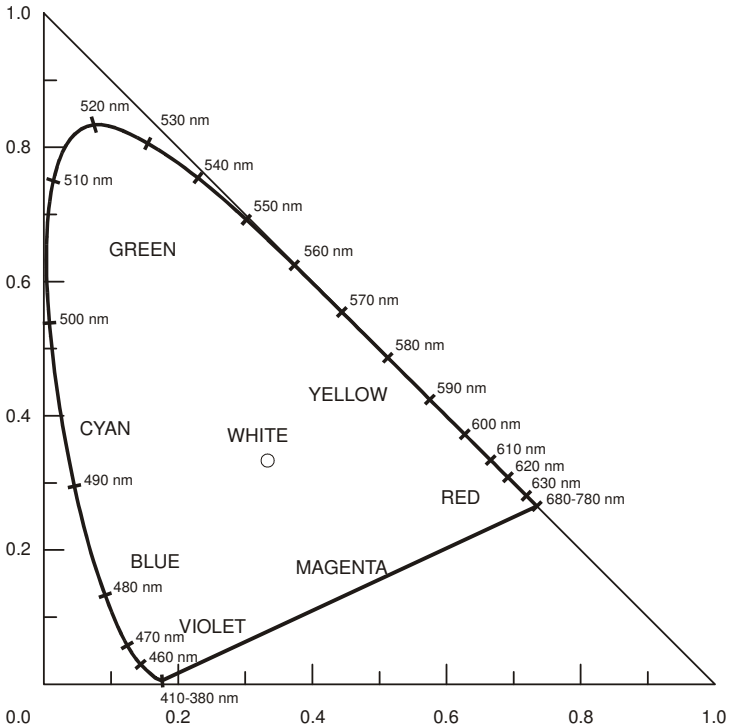

Fig. 3-4: The chromacity diagram of 2° CIE Observer.

In a chromacity diagram the principles of additive mixing can be easily demonstrated. All color stimuli created as a mixture of two color stimuli $\mathbf{Q_1}$ and $\mathbf{Q_2}$ are located in chromacity diagram on the straight line interconnecting the chromacity points Q_1 and Q_2. The chromacity point Q_{12} is computed using the mixing coefficients a, b ($a + b = 1$) as $Q_{12} = aQ_1 + bQ_2$.

Using this property color stimuli whose mixture produces the white color stimuli can be easily found. If the straight line between two chromacity points Q_1 and Q_2 intersects the

white point then the colors represented by the Q_1 and Q_2 are called complementary colors. Among complementary colors belongs commonly known pairs such as blue – yellow, green – magenta and red – cyan.

If more then two primary stimuli are used for mixing then the set of chromacity points of the color stimuli obtained from the mixture of the primary stimuli is represented as convex polygon in chromacity diagram. Each vertex of the polygon corresponds with the chromacity point of one primary stimulus. The polygon is sometimes referred as color gamut.

Direct mixing of light beams is not the only one possible. Other two types of mixing are temporal and spatial mixing. Temporal mixing is achieved by fast altering of a color, for example rotating disc colored with two colors. Because of a persistence of vision of human eye, only one additively mixed color is perceived. The spatial additive mixing is achieved by placing small enough color points close to each other. When viewing from larger distance, the perceived color is an additive mixture of those used for points. This technique is used for almost all color reproducing devices such as monitors or printers.

To describe some color's appearance, one needs some terms to do so. It is usual to describe a color using three attributes [Nas98]. In different literature these attributes have different names but have very similar meaning. The first attribute is hue or dominant wavelength or chromatic color. It specifies the spectral color or non-spectral purple color. It can be found in chromacity diagram by extending the line originating from the white point and going through chromacity point representing the color stimulus to the boundary of the diagram. Position of the intersection then determines the corresponding hue.

The second attribute is called saturation or purity or tone. It is a measure of a white present in a color. In the chromacity diagram is saturation expressed as a ratio of the distance between white point and the corresponding hue and the distance of the chromacity point of the color stimulus from the white point.

The last attribute is brightness, value, lightness or luminance. It expresses different levels of a color although those terms are not interchangeable.

3.2 Spectral Pipeline

The process of the full spectral rendering consists of three significant steps.

- Spectral data gathering and processing.
- Illumination simulation.
- Conversion of the spectral representation of a color into the trichromatic one, which is more suitable for displaying and storing.

The first step is the most important one because the accuracy of the light simulation is directly dependent on the accuracy of the input data. Spectral power distribution is needed for each light source and at least the spectral reflectance distribution for each material is needed. Other spectral data, such as spectral transparency distribution or excitation spectra for fluorescence, can be used to widen the material's description.

In the second step is performed illumination using one of the illumination models described in the chapter 2 and in the third step the spectral power distribution obtained for each pixel of the image is scaled appropriately, converted into trichromatic representation and

quantized. It is also suggested to convert the color from the used device independent trichromatic system into the device dependent one before displaying.

3.2.1 Spectral Data

The input of the full spectral renderer consists of geometry, material and light description. While the geometry determines a shape the material and light spectral description determine the appearance of the geometry. There are two kinds of spectral data. The spectra of the first kind are the spectra describing lights' power distribution. These spectra should consist of absolute values in Watts. However, it is usual to express them relatively because the relations between values of the spectrum characterize the chromacity of the light source and the absolute values can be easily obtained by scaling the relative values appropriately. Therefore the light is described by its relative spectral power distribution and a scaling factor. Such organization has smaller memory requirements because only one instance of the spectrum is needed for each scene.

The second kind of the spectral data is spectral distribution of the materials' wavelength dependant properties. Properties such as reflectance or transmittance expressed as the ratio of light reflected ore transmitted and light incident on a surface modifies the incident light. The reflectance and transmittance are related in terms of the Fresnel term (see 2.2.1.1). Fresnel term is a function of index of refraction, which is a function of wavelength. Therefore the spectral distribution of the index of refraction is considered as another input spectrum. Naturally, direct spectral index of refraction distribution is used only if it is available. In other cases is used the equation E[2.2-11] for calculation of the index of refraction form the normal incidence reflectance.

Another significant material's property is fluorescence. The effect of absorption of the short wavelength, i.e. ultra violet, radiant energy by the material and its reemission as light with longer wavelength in visible range is referred as fluorescence. The fluorescence is described as the set of spectral distributions $F_{\lambda_i}(\lambda)$. The $F_{\lambda_i}(\lambda)$ relates the amount of incident energy at wavelength λ_i and the amount of emitted energy at wavelength λ. There is one such spectrum for each wavelength taken into consideration. The power Φ_E emitted at some wavelength λ_E due to some light source with spectral power distribution Φ is computed according to E[3.2-1].

$$\Phi_E(\lambda_E) = \int_\mu F_\mu(\lambda_E)\Phi(\mu)d\mu \qquad \text{E[3.2-1]}$$

Spectral data are usually measured by spectrophotometers at a finite number of wavelengths so their output is point sampled version of the actual continuous spectrum. The number of the samples per spectrum should be large enough to catch the spectrum's specifics so it can be reconstructed appropriately. According to the Nyquist sampling theorem the sampling frequency should be twice higher then the highest one present in the sampled signal, e.g. spectrum.

As a result, the acquired spectra may have different sampling step which is inconvenient when a scene is rendered because the illumination process is performed using the piecewise additions and multiplications of the spectra. It is needed to have all spectra defined on the processed wavelength range and sampled with the same frequency before the rendering process may begin.

Inconsistence in the sampling rate and alignment of the sample positions can be treated by interpolation. The order of the used interpolation depends on the sampling frequency. New values of spectra sampled with higher frequency, e.g.10 nm step, are sufficiently interpolated by the linear interpolation. On the contrary, spectra sampled with lower frequency may be worth of higher order interpolation.

An alternative to the discrete representation is analytical representation of spectra by polynomials. However, the polynomials are able to describe efficiently only smoothly varying spectra, e.g. florescent light sources introduce high and narrow spikes in their spectral power distributions and therefore representing such spectrum using polynomials would be awkward. Another such extreme cases are one wavelength absorber or emitter that is usual for gasses.

For the point sampling speaks its generality and also the fact that matching function used for the conversion of the spectra representation to the trichromatic one are sampled as well. Implementation of the point sampled spectra into the existing trichromatic systems is also easier, because trichromatic systems are just special case of the full spectral system, i.e. they use three samples per spectrum only. For these reasons the discrete representation is the only one applied in this thesis.

With the representation of the spectral data is closely related the process of transformation from spectral representation to the trichromatic representation. This process involves integration of the product of the spectral power distribution and the appropriate matching function, see E[3.1-8]. If integrated function, i.e. spectrum, is represented as $N+2$ evenly spaced samples sampled at $\lambda_0, \lambda_1 \ldots \lambda_N$ then the integral can be approximated using the Riemann summation E[3.2-2].

$$
\begin{aligned}
X &= \sum_{i=0}^{N-1} \bar{x}(\lambda_i)\Phi(\lambda_i)\Delta\lambda_i \\
Y &= \sum_{i=0}^{N-1} \bar{y}(\lambda_i)\Phi(\lambda_i)\Delta\lambda_i \\
Z &= \sum_{i=0}^{N-1} \bar{z}(\lambda_i)\Phi(\lambda_i)\Delta\lambda_i \\
\Delta\lambda_i &= \lambda_{i+1} - \lambda_i
\end{aligned}
\qquad \text{E[3.2-2]}
$$

3.2.2 Spectrum to Tristimulus Values Conversion

Spectral representation is useful only for illumination purposes. After obtaining spectral power distribution for each pixel of the final image the use of the trichromatic representation is more convenient for displaying and storing. It is less memory extensive and its proper presentation involves only one matrix multiplication, i.e. transformation matrix from device independent *XYZ* to device dependent *RGB*. The principle of the conversion is described in the section 3.1.2.1.

The *XYZ* tristimulus values of the spectrum power distribution Φ represented as $N+1$ sampled values at $\lambda_0, \lambda_1 \ldots \lambda_N$ are computed using expression E[3.2-3]. The scale factor ξ is introduced to scale the computed values into the quantization range. This range is usually $\langle 0.0, 1.0 \rangle$ and the value of ξ is set so that the value of *Y* component of the chosen white

stimulus is scaled onto the value 1.0. The component *Y* is chosen because it corresponds to the luminance value, see the section 3.1.2.3.

The ξ scale factor can be avoided if all initial spectral power distributions in a scene, i.e. spectral power distributions of light sources, are scaled appropriately. The spectra are scaled in any case because they are usually specified relatively and therefore they have to be scaled into the appropriate absolute values. By modifying the scale factor as proposed the *Y* component of the spectrum will yield the already normalized value.

The scale factor of a spectral power distribution is related with the luminance level which is assigned to the spectrum. This luminance level may be specified relatively so that the value 1.0 represents the highest value yet in quantization range. Once the luminance is specified the scale factor is computed form expression E[3.2-4]. Each value of the spectrum is then divided by that factor.

$$X = \xi \sum_{i=0}^{N-1} \bar{x}(\lambda_i)\Phi(\lambda_i)\Delta\lambda_i$$
$$Y = \xi \sum_{i=0}^{N-1} \bar{y}(\lambda_i)\Phi(\lambda_i)\Delta\lambda_i \qquad \text{E[3.2-3]}$$
$$Z = \xi \sum_{i=0}^{N-1} \bar{z}(\lambda_i)\Phi(\lambda_i)\Delta\lambda_i$$

ξ Scale factor.

$$\xi = L_v \Big/ \sum_{i=0}^{N-1} \bar{y}(\lambda_i)\Phi(\lambda_i)\Delta\lambda_i \qquad \text{E[3.2-4]}$$
$$\Delta\lambda_i = \lambda_{i+1} - \lambda_i$$

L_v The luminance level assigned to Φ.

Another important aspect of the conversion is a chromatic adaptation ability of an eye. The chromatic adaptation is a dynamic mechanism of the human vision system which compensates the color of a light source and therefore approximately preserves the material's appearance. The chromatic adaptation is the reason why white object appear white under two illuminants with different white point, e.g. CIE D65 and CIE A. If this adaptation is not considered then the synthesized image may appear wrong if viewing conditions are different from the light conditions within the image. It should be noted that the presented information about chromatic adaptation is taken form [Suss01] if not said otherwise.

The chromatic adaptation transformation is mostly based on the von Kreis method [Wysz00], see equation E[3.2-5]. The CIE X',Y',Z' tristimulus values are linearly transformed by matrix $\mathbf{M}_{CAT}$ to derive the cones post-adaptation response R',G',B' under the first illuminant. The resulting values are independently scaled to obtain the post-adaptation response R'',G'',B'' under the second illuminant and the values are transformed by the $\mathbf{M}_{CAT}^{-1}$ matrix to obtain the adapted X'',Y'',Z'' tristimulus values. The adaptation scaling coefficients are mostly based on the illuminants' white point post-adaptation responses R'_w,G'_w,B'_w and R''_w,G''_w,B''_w.

$$\begin{bmatrix} X'' \\ Y'' \\ Z'' \end{bmatrix} = \mathbf{M}_{CAT}^{-1} \begin{bmatrix} R''_w/R'_w & 0 & 0 \\ 0 & G''_w/G'_w & 0 \\ 0 & 0 & B''_w/B'_w \end{bmatrix} \mathbf{M}_{CAT} \begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix}$$

$$\begin{bmatrix} R''_w \\ G''_w \\ B''_w \end{bmatrix} = \mathbf{M}_{CAT} \begin{bmatrix} X''_w \\ Y''_w \\ Z''_w \end{bmatrix} \quad \begin{bmatrix} R'_w \\ G'_w \\ B'_w \end{bmatrix} = \mathbf{M}_{CAT} \begin{bmatrix} X'_w \\ Y'_w \\ Z'_w \end{bmatrix} \qquad \text{E[3.2-5]}$$

There are known several transformation matrices $\mathbf{M}_{CAT}$. They vary in the underlying chromatic adaptation theory used for their derivation and in the consequent adherence to the real chromatic adaptation mechanics. In [Suss01] were enumerated and tested matrices $\mathbf{M}_{CAT1} - \mathbf{M}_{CAT4}$ and as the most accurate transformation matrices were indicated the matrices $\mathbf{M}_{CAT3}$ and $\mathbf{M}_{CAT4}$.

$$\mathbf{M}_{CAT1} = \begin{bmatrix} 0.3897 & 0.6890 & -0.0787 \\ -0.2298 & 1.1834 & 0.0464 \\ 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{M}_{CAT2} = \begin{bmatrix} 0.8951 & 0.2664 & -0.1614 \\ -0.7502 & 1.7135 & 0.0367 \\ 0.0389 & -0.0685 & 1.0296 \end{bmatrix}$$

$$\mathbf{M}_{CAT3} = \begin{bmatrix} 1.2694 & -0.0988 & -0.1706 \\ -0.8364 & 1.8006 & 0.0357 \\ 0.0297 & -0.0315 & 1.0018 \end{bmatrix} \qquad \mathbf{M}_{CAT4} = \begin{bmatrix} 0.7982 & 0.3389 & -0.1371 \\ -0.5918 & 1.5512 & 0.0406 \\ 0.0008 & 0.0239 & 0.9753 \end{bmatrix}$$

3.3 Acceleration Methods

Direct use of discretely represented spectra means that for each sampled wavelength illumination is evaluated and then the *XYZ* tristimulus values are computed and converted into a desired display device dependent color space. As one might expect such process is computationally expensive it is many times more expensive then the trichromatic system, e.g. if sampling step of 5 nm is used on the interval form 400 nm to 700 nm then 59 in total samples have to be evaluated. It is 20 times more then trichromatic renderer even without other operations required for conversion into trichromatic representation. Without some more efficient technique the spectral rendering is in a raw form far too unpractical.

3.3.1 Linear Color Representation

The acceleration method presented in [Peer93] is based on an idea of representing spectral values as a linear combination of *m* ortho-normal basis functions. The coefficients of this linear combination are all what is needed to perform illumination and conversion into the trichromatic representation.

The illumination is evaluated using pre-computed transformation matrices which transform the coefficients of the incident light into the coefficients of the reflected or transmitted or absorbed light. After illumination evaluation the coefficients can be transformed into the tristimulus values using another pre-computed transformation matrix.

Equation E[3.3-1] expresses the linear combination of *m* ortho-normal basis functions E. Two functions E_i and E_j are ortho-normal if condition E[3.3-2] applies. The combination

coefficients ε can be computed using the ortho-normal property as E[3.3-3]. The process of illumination evaluation is demonstrated on computation of coefficients of a reflected light. The coefficients of a transmitted or absorbed light are computed analogically. When the light, with spectral power distribution described by the coefficients $\varepsilon_i \quad i = 1 \ldots m$, is reflected from a surface with a reflectance *R* then the resulting spectrum Φ_O is computed according to E[3.3-4].

$$\Phi(\lambda) = \sum_{i=1}^{m} \varepsilon_i E_i(\lambda) \qquad \text{E[3.3-1]}$$

$\Phi(\lambda)$	Spectral power distribution.
ε_i	i[th] combination coefficient.
$E_i(\lambda)$	i[th] orthonormal basis function.

$$\forall i, j \in \langle 1, m \rangle, i \neq j : \int_\lambda E_i(\lambda) E_j(\lambda) d\lambda = 0 \qquad \text{E[3.3-2]}$$

$$\varepsilon_i = \int_\lambda \Phi(\lambda) E_i(\lambda) d\lambda \qquad \text{E[3.3-3]}$$

However the resulting spectrum can be also represented as a linear combination of the basis functions and one gets E[3.3-5]. Using equations E[3.3-3] and E[3.3-4] the coefficients of the reflected spectrum can be expressed as E[3.3-6]. Coefficients $R_{ij} \quad i, j = 1 \ldots m$ emerged form E[3.3-6] form a transformation matrix which transforms coefficients of the incident spectrum into the coefficients of the reflected spectrum. Equation E[3.3-6] can be then rewritten using the matrix form E[3.3-7].

$$\Phi_O(\lambda) = \sum_{i=1}^{m} \varepsilon_i^I R^I(\lambda) E_i(\lambda) \qquad \text{E[3.3-4]}$$

$\Phi_O(\lambda)$	Reflected spectral power distribution.
ε_i^I	i[th] coefficient of an incoming light.
$R^I(\lambda)$	Spectral distribution of a material's reflectance.

$$\Phi_O(\lambda) = \sum_{i=1}^{m} \varepsilon_j^O E_i(\lambda) \qquad \text{E[3.3-5]}$$

ε_j^O	Coefficients of a reflected spectral power distribution.

$$\varepsilon_j^O = \int_\lambda \Phi_O(\lambda) E_j d\lambda = \int_\lambda \sum_{i=1}^{m} \left(\varepsilon_i^I R^I(\lambda) E_i(\lambda) \right) E_j d\lambda = \sum_{i=1}^{m} \varepsilon_i^I R_{ij}^I$$
$$R_{ij}^I = \int_\lambda R^I(\lambda) E_i(\lambda) E_j(\lambda) d\lambda \qquad \text{E[3.3-6]}$$

$$\begin{bmatrix} \varepsilon_1^O \\ \vdots \\ \varepsilon_m^O \end{bmatrix} = \begin{bmatrix} R_{11}^I & \cdots & R_{1m}^I \\ \vdots & \ddots & \vdots \\ R_{m1}^I & \cdots & R_{mm}^I \end{bmatrix} \times \begin{bmatrix} \varepsilon_1^I \\ \vdots \\ \varepsilon_m^I \end{bmatrix} \qquad \text{E[3.3-7]}$$

The described process can be repeated also for transmission or absorption. The result of this method is a set of matrices of size $m \times m$ where *m* denotes the number of used ortho-normal basis functions. If some of material's property depends on an angle of light's incidence then several matrices have to be computed, each for particular angle. Then during the evaluation of illumination at some arbitrary incident angle the matrices computed at the closest angles are chosen. The coefficients at the wanted angle can be interpolated from the coefficients computed form those matrices. If the utilized illumination model offers a possibility of having different reflectance for ambient, diffuse and specular reflection then the matrices for each type of reflection are computed. Coefficients of the total reflected light are a piecewise sum of the computed sets of coefficients.

One great advantage of this method is a fact that coefficients of some spectral power distribution can be transformed directly to *XYZ* tristimulus values by matrix multiplication E[3.3-9]. If linear representation of the spectral power distribution is substituted into the equation for tristimulus value calculation then the construction of the coefficients of the transformation matrix is apparent form equation E[3.3-8]. This matrix can be multiplied with the transformation matrix which represents additional transformations as described in section 3.2.2.

$$X = \int_{\lambda} \bar{x}(\lambda)\Phi(\lambda)d\lambda = \int_{\lambda} \bar{x}(\lambda)\sum_{i=1}^{m}(\varepsilon_i E_i(\lambda))d\lambda = \sum_{i=1}^{m}\varepsilon_i T_{xi}$$
$$T_{xi} = \int_{\lambda} \bar{x}(\lambda)E_i(d\lambda) \qquad \text{E[3.3-8]}$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} T_{x1} & \cdots & T_{xm} \\ T_{y1} & \cdots & T_{ym} \\ T_{z1} & \cdots & T_{zm} \end{bmatrix} \times \begin{bmatrix} \varepsilon_1 \\ \vdots \\ \varepsilon_m \end{bmatrix} \qquad \text{E[3.3-9]}$$

One important aspect of this method was not yet mentioned. It is a selection of the basis functions. Selection of a proper basis function is clearly essential for this method. Basis function must be able to catch all subtleties of each spectral distributions used in a scene. Basis functions can be selected manually but it is not easy task especially if low error value is required together with keeping details of highly various spectra.

In [Peer93] singular value decomposition SVD was proposed as an automatic basis selection system. All possible spectra shapes present in a scene, represented as a column of values, are composed into a matrix. Then the SVD is performed on this matrix. Output of the SVD is *n*-dimensional ortho-normal basis where *n* denotes number of columns of a matrix. First *m* columns are then taken from this basis and they are used as a basis for a given scene. The result of the SVD is structured so that the first *m* basis functions are the best ortho-normal basis for the input spectra. Term "the best" means that en error E[3.3-10] is minimized for the first *m* basis functions. This error approaches zero as the number of used basis functions approaches *n*.

$$Err = \sum_{\Phi}\int\left(\Phi(\lambda) - \sum_{i=1}^{m}\varepsilon_i E_i(\lambda)\right)^2 d\lambda \qquad \text{E[3.3-10]}$$

The base for SVD is theorem that every matrix **A** of dimension $m \times n$ can be expressed as a product $\mathbf{A} = \mathbf{USV}^T$ where matrix **U** is of dimension $m \times m$, matrix **S** is of dimension $m \times n$ and matrix **V** is of dimension $n \times n$. **U** and **V** are orthogonal matrices and **S** is a diagonal matrix. Columns of **U** are then used as the ortho-normal basis functions for the linear representation of spectral distributions. Elements on the diagonal of the **S** matrix are singular values of the matrix **A** ordered in descending order.

To calculate the matrices **U** and **V** the eigenvectors and eigenvalues of the matrices $\mathbf{A}^T\mathbf{A}$ and $\mathbf{AA}^T$ must be calculated first. Eigenvectors of the matrix $\mathbf{A}^T\mathbf{A}$ creates columns of **U** and eigenvectors of the matrix $\mathbf{AA}^T$ creates the rows of **V**. The eigenvalues of $\mathbf{A}^T\mathbf{A}$ are the squares of singular values of the matrix **A** and are located on the main diagonal of the matrix **S** in descending order.

One thing must be considered when performing the SVD. Decomposition is very computationally demanding and it may even take more time to perform than the naive spectral illumination calculations.

3.3.2 Color Pre-filtering

Color pre-filtering is an acceleration method presented in [Ward92]. It reduces all spectral data, i.e. spectral power distributions, material's reflectance, into their trichromatic representation with regard to the selected dominant illuminant. The trichromatic representation is then used for rendering and display using the standard trichromatic render systems.

This method assumes that most scenes contain only one dominant illuminant, i.e. the light sources within the scene have the same spectral power distribution. There may be lights with other spectral power distributions, however those are considered to have minor influence on appearance of materials. Another assumption justifying this method is that the direct illumination component is the most defining and its accuracy determines the accuracy of the consequent steps.

If the scene contains more light sources with identical spectral power distributions but with different luminance then the light source with the highest luminance is selected as the dominant illuminant.

For each material its color is computed under the selected light source, i.e. the reflectance is multiplied with the spectral power distribution of the light, and converted into the trichromatic representation. The trichromatic representation of the spectral power distribution of the light sources is determined using the following rules:

- The light sources which have spectral power distribution and luminance identical to the dominant light source are replaced with trichromatic representation of white, i.e. $R, G, B = 1$.
- The light sources which have spectral power distribution identical to the spectral power distribution of the dominant light source but with the luminance lower then the luminance of the dominant light source are replaced with trichromatic representation of appropriately scaled white, e.g. if the luminance of the dominant light source is 2.0 and the luminance of the other light source is 1.0 then the trichromatic color of the other light source should be $R, G, B = 0.5$.

- The light sources which have the spectral power distribution different form the spectral power distribution of the dominant light source are replaced with the trichromatic representation of their own spectral power distribution. The trichromatic representation is scaled according to the relation of their luminance and the luminance of the dominant light.

Once the color pre-filtering is done, all spectral information within a scene has been reduced into the trichromatic equivalent and the standard trichromatic render sequence can take place.

Advantage of this method is that its implementation into existing trichromatic renderer is very simple, for the color pre-filtering is separated from the rendering process. The disadvantage of this method is that the complete wavelength information, which could be used in advanced illumination models, is missing.

3.4 Real Time Spectral Rendering

The graphics processing units – GPU are becoming more powerful with each new generation. The GPU were recently equipped with the programmable graphics pipeline which provides higher level of flexibility. Since there is a computational power and the tools for the calculation customization the spectral rendering can be transferred form CPU to GPU. There are still some limitations in that transfer but the obtained results from GPU calculations are sufficient for fast color's preview.

3.4.1 Programmable Graphics Pipeline

The introduction of the graphics pipeline into the graphics accelerators made them more universal tool for visualization tasks. The graphics pipeline has two programmable stages, namely a vertex processing stage and a pixel processing stage. A program for the vertex stage is commonly called vertex shader and a program for the pixel stage is commonly called pixel shader alias fragment shader.

The vertex shader modifies data linked with vertices. The data may be the position, normal, texture coordinates, color, etc. Vertices outputted from the vertex stage are expected to be transformed into a projection space, texture mapped and if per vertex illumination is performed then the computed color should be outputted in a color property.

The pixel shader is modifying color of a pixel generated during a triangle rasterization. The properties of the rasterized triangle's vertices are interpolated and they are available for pixel shader evaluation purposes. Output of a pixel shader is pixel's color and depth.

Program for each stage is limited with number of instruction slots supported by hardware. The vertex programs should have 256 instruction slots available in a version 2.0 however the number of executed instructions can be higher due to the loop operation support. The total number of executed instructions is restricted by the capabilities of the particular hardware. Pixel programs are evaluated more often then the vertex programs so they have only 96 instruction slots available divided into 64 arithmetic and 32 texture instructions for a version 2.0.

Higher versions of the vertex and pixel shaders provide more instruction slots and wider instruction sets nevertheless their hardware implementation was not widely available at the time of writing of this paper.

3.4.2 Cg and HLSL

Cg and HLSL, i.e. high level shading language, are a high level C like programming languages designed for programmable graphics pipeline. Before those languages were introduced, all programming had to be performed using an assembler language. Programs written in assembly languages are hardware dependent and therefore not so efficient on newer HW. If Cg or HLSL language is used instead of the assembler then the program code is just recompiled on the compiler for the particular HW.

The HLSL and the Cg are almost identical languages except of several minor differences. The Cg is developed by the NVidia and the Cg API has to be linked with the program which is using it. The HLSL is developed by the Microsoft and its API is incorporated directly into the DirectX graphics library that provides debugging support as well as software emulation for older HW. This is the very reason why the HLSL was chosen for the implementation of methods presented in this paper. Specifics of the language is presented in the Appendix E.

4 Implementation

Three methods were implemented in total for verification and performance comparison. They are the raw point sampling method, the linear method described in the section 3.3.1 and the color pre-filtering method described in the section 3.3.2. Methods were implemented in non interactive rendering system and in real time rendering system as well. The methods are compared in terms of speed, color accuracy and capabilities. Each method's implementation specifics are described in the separate sections, i.e. 4.2 – point sampling method, 4.3 – linear color representation and 4.4 – color pre-filtering, in detail. The common issues are discussed in the section 4.6.

The illumination calculations in both rendering systems use illumination models described in the section 2.2. Models are tested on their influence on the evaluation speed and on the realism of their output. The model's contribution to the capabilities of a spectral rendering is also explored. If possible the illumination models are incorporated into the programmable graphics pipeline. The summary of results is presented in the section 4.5.

For the implementation of the real time rendering system was used multimedia library Microsoft DirectX, more concretely, its managed version for the .NET platform. Programs for programmable pipeline was created in the HLSL and processed using the HLSL API included in the DirectX. Simple introduction to the DirectX programming is presented in the section 4.1.

All testing and measuring was performed on the Intel Celeron 2.4 GHz system with 512 MB RAM equipped with ATI Sapphire Radeon 9600SE Edition graphics card with 128 MB RAM.

4.1 Managed DirectX

DirectX is a multimedia library for Microsoft Windows system currently in the version 9.0. It contains solutions for 2D and 3D graphics, network communication and sound playback and manipulation. Naturally, only the 3D capabilities are employed for the spectral rendering methods. Managed version for .NET platform was introduced with the version 9.0 and this version was used for implementation of the real time spectral rendering methods.

4.1.1 DirectX Initialization

The fundamental class for 3D graphics is `Device`. It is responsible for graphics initialization and for scene rendering and presentation. The easiest DirectX initialization is presented in the Sample Code 1. The `PresentParameters` class contains basic definition of the presentation behavior of the `Device` instance which includes whether the full screen or windowed mode is required, back buffer format or the swapping method used for back buffer presentation. `SwapEffect.Discard` used in the sample code is the most efficient one but does not guarantee that the back buffer's contents remain intact by the swap operation, so it must be redrawn completely for each frame.

The parameters of the `Device` constructor are the adapter's number, device type, the reference on form window, set of settings flags and the present parameters described above. The adapter's number determines the actual graphics hardware which will be used for acceleration and for presentation. It is usually zero, for there is only one graphics card installed on most computers. Device type is mostly set to `DeviceType.Hardware` value, which makes the device to use graphics hardware for all possible drawing tasks. There is one

other option and it is `DeviceType.Reference`. This value indicates that the device will use software emulation for drawing. This option is used when some feature of DirectX is needed but it is not supported by the particular hardware.

The next parameter used in constructor is a reference on a form window. The created device will use that window for presentation of the rendered image. The set of `CreateFlags` flags provided to the constructor contains additional settings for the constructed device. The `CretaeFlags.SoftwareVertexProcessing` flag used in the sample code indicates that the vertices are processed by the CPU instead of the GPU. If the construction fails for some reason then the `DirectXException` exception is thrown.

```
//Reference to the created device
Device device;

try
{
     //PresentParameters contains basic requirements on the device
     PresentParameters presentParams = new PresentParameters();
     presentParams.Windowed = true;
     presentParams.SwapEffect = SwapEffect.Discard;

     //device is created using the constructor.
     //The parameters are adapter number, type of the device, Form
     //reference used for displaying the rendered image, processing
     //method of vertices and the PresentParameters structure filled
     //in the code above.
     device = new Device(0, DeviceType.Hardware, renderTarget,
          CreateFlags.SoftwareVertexProcessing, presentParams);
}
catch (DirectXException)
```

Sample Code 1: DirectX inicialization.

Initialization as described above is rather a simple one. It is sufficient for some easy experiments but the complete capabilities enumeration of a particular hardware should be performed to find out if required features are supported. Such complete enumeration is a quite long program to fit into this brief introduction but it can be found in the application's source code – class `DirectXView`.

4.1.2 Vertex Buffer

The geometry of a scene is stored in vertex buffers. Vertex buffer is an array of numbers representing information about vertices. The number of values describing one vertex and their ordering is quite arbitrary but it must contain position of the vertex at least. The meaning of each value in relation to the vertex is described using the `VertexFormats` enumeration. The most common additional vertex data are the normal, texture coordinates or color. There are pre-defined several most common vertex formats contained within the class `CustomVertex`.

Vertex buffer is represented by the `VertexBuffer` class. Creation of the vertex buffer and filling it with data is shown in the Sample Code 2. The runtime type of the class containing data of one vertex, number of vertices to allocate memory for, the buffer's owning device, set of `Usage` flags describing buffer's properties, `VertexFormats` value of the used vertex format and the buffer's `Pool` is passed to the constructor of the `VertexBuffer` class.

The flag `Usage.WriteOnly` indicates that the vertex buffer won't be changed thus it can be stored in local memory, which makes the operations over the buffer's data faster. The `Pool` flag specifies the memory used for storing the particular resource. If it is set to `Pool.Default` then the most appropriate memory type is chosen by the device.

Once the vertex buffer is successfully created then it can be filled with data. The memory block of the vertex buffer must be locked first before it can be accessed. It is done using the `Lock()` method. The first parameter of this method is the offset of locked memory in bytes; the second one is the size of the memory in bytes to lock and the last one is one or more `LockFlags` flags specifying the details of the lock operation.

By setting all three parameters to zero the whole memory block of the buffer is locked as a result. The result of the lock operation is a reference to the `GraphicsStream` stream, which can be used for writing the data in to the vertex buffer using the stream's `Write()` method. The operation of filling the vertex buffer with data is completed by unlocking the locked memory by the `Unlock()` method.

```
//Vertex buffer construction. The parameters are runtime type of the class
//describing the data layout, number of vertices in the buffer, owning
//device, Usage flags, VertexFormats value of the used vertex data layout
//and resource pool.
VertexBuffer vb = new VertexBuffer(typeof(CustomVertex.TransformedColored),
                  3, device, Usage.WriteOnly,
                  CustomVertex.TransformedColored.Format, Pool.Default);
//Lock of the vertex buffer's memory block to access it. First parameter
//is offset of the locked memory, second one is a number of bytes to lock -
//whole memory block is locked if zero is provided and the third one is
//LockFlag flag. Zero is LockFlag.None and means std lock operation.
GraphicsStream stm = vb.Lock(0, 0, 0);

CustomVertex.TransformedColored[] verts =
     new CustomVertex.TransformedColored[3];

verts[0].X=150; verts[0].Y=50; verts[0].Z=0.5f; verts[0].Rhw=1;
verts[0].Color =Color.Aqua.ToArgb();
verts[1].X=250; verts[1].Y=250; verts[1].Z=0.5f; verts[1].Rhw=1;
verts[1].Color = Color.Brown.ToArgb();
verts[2].X=50; verts[2].Y=250; verts[2].Z=0.5f; verts[2].Rhw=1;
verts[2].Color = Color.LightPink.ToArgb();

//Writing the data into the vertex buffer.
stm.Write(verts);
//Finishing the operation by unlocking the buffer's memory.
vb.Unlock();
```

Sample Code 2: The creation of a vertex buffer and its filling with data.

4.1.3 Scene Rendering

Once the device is initialized and geometry of a scene is prepared in vertex buffers then the actual frame rendering may proceed. Example of a simple rendering routine is shown in the Sample Code 3.

The back buffer, z-buffer and stencil buffer should be cleared first to get rid of the data they may contain. It is done using the `Clear()` method of the `Device` class. The first parameter of the method contains the combination of the `ClearFlags` flags, which marks

the buffers for clearing. The possible flags are `ClearFlags.Target`, `ClearFlags.ZBuffer` and `ClearFlags.Stencil` corresponding with the back buffer, z-buffer and stencil buffer respectively. The selected buffers are filled with the values provided as second, third and fourth parameter.

The calling of `BeginScene()` method must precede all drawing methods. The selected vertex buffer is then marked as a current data source using the `SetStreamSource()` method. The device must be noted about the format of vertices stored inside the vertex buffer by setting the format into the `VertexFormat` property. Then the drawing method `DrawPrimitives()` is called to draw the desired primitives onto the back buffer.

The parameters of the method are the `PrimitiveType` value determining the type of primitives to draw, index of the vertex to start from and the number of primitives to draw. The drawing of the scene is concluded by calling the `EndScene()` method and the content of back buffer is presented using the `Present()` method. All mentioned methods are members of the `Device` class.

```
//clear the back buffer to a blue color
device.Clear(ClearFlags.Target | ClearFlags.ZBuffer,
System.Drawing.Color.Blue, 1.0f, 0);

//begin of drawin
device.BeginScene();

//set vertexBuffer as a current data source
device.SetStreamSource(0, vertexBuffer, 0);
//tell to device, what format are the vertices of
device.VertexFormat = CustomVertex.PositionColored.Format;
//draw the triangle,
device.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);

//end of drawing
device.EndScene();
//present the contents of the back buffer on a screen
device.Present();
```

Sample Code 3: Scene rendering

4.1.4 Effects

Graphics effects in DirectX are achieved by combining the textures, vertex and/or pixel shader and adequate render state values. Setting all these parameters every time the effect is used is tedious so the effect framework has been included into the DirectX. Effect encapsulates all mentioned parameters and those can be set all at once by calling appropriate method. Furthermore, one effect may contain several techniques for achieving the particular effect. Different technique may be used with regards to the available hardware capabilities, for example. Effects are usually stored in external text files, so that the main application does not have to be rebuilt when the effect file is modified.

Example of a simple effect file is shown in the Sample Code 4. It may contain the code of vertex and pixel shader in both HLSL and assembly language, and then there is defined arbitrary number of techniques. Each technique has one or more pass, which contains

rendering state values. When the effect is active and the pass is selected then the current render state is replaced by the one stored in the active pass.

The effect framework is demonstrated in the Sample Code 5. Effects are constructed using one of the `FromString()`, `FromStream()`, `FromFile()` static methods of the `Effect` class. The parameters of each of them are the owning device, effect's source code as a string, stream or name of the text file containing the source code, instance of the `Include`, the set of `ShaderFlags` flags specifying additional compile options, instance of the `EffectPool` class and the string out parameter containing eventual effect compiler's error.

The instance of the `Include` class must be provided if preprocessor directive `#include` is present in the effect file otherwise the `null` value should be provided. The flag of `ShaderFlags` is usually set to the `ShaderFlags.None` flag. The `EffectPool` is provided only if resource sharing across different effects feature is required otherwise the `null` value should be provided.

```
//Global shader variables, may be accessed both form vertex and pixel
//shader programs. They can be set by the application using the
//Effect.SetValue() method
matrix view_matrix;
matrix projection_matrix;
matrix view_proj_matrix;
float4 viewPos;
float4 lightPos;
float4 diffuse;
float4 specular;

struct VS_OUTPUT
{
     float4 Pos: POSITION;
     float3 Normal: TEXCOORD0;
     float3 WorldPos: TEXCOORD1;
};

//vertex shader main function
VS_OUTPUT vs_main(float4 Pos: POSITION, float3 Normal: NORMAL)
{
     VS_OUTPUT Out;
     //vertex position and vertex normal are expected
     //to be already transformed into the world space
     Out.WorldPos = Pos;
     Out.Normal = Normal;
     //vertices are expected to be transformed into the projection
     //space before leaving the vertex shader
     Out.Pos = mul (view_proj_matrix, Pos);

     return Out;
}
//pixel shader main function
float4 ps_main(float4 Diffuse: COLOR0, float3 Normal: TEXCOORD0,
                    float3 WorldPos: TEXCOORD1) : COLOR0
{
     float3 N = normalize(Normal);
     float3 L = normalize(lightPos - WorldPos);
     float3 V = normalize(viewPos - WorldPos);
     float3 H = normalize(L + V);
```

```
     //diffuse part
     float d = max (dot(N, L), 0);
     //specular part
     float s = max (dot(H, N), 0);

     return d * diffuse + pow(s, 30) * specular;
}

technique t0
{
     pass p0
     {
          VertexShader = compile vs_1_1 vs_main();
          PixelShader = compile ps_2_0 ps_main();
     }
}
```

Sample Code 4: Simple effect code implementing per pixel illumination in the Phong's illumination model style.

It should be noted that all mentioned methods from this point forward are members of the `Effect` class. From the successfully compiled effect can be retrieved handles to effect's global variables using the `GetParameter()` method. The method takes the name of the variable as a parameter and returns `EffectHandle()` if the variable is found. This handle is used when the value is assigned to the variable using the `SetValue()` method, which takes the variable's handle and its value. The method `SetValue()` is overloaded for most of the data types known to the HLSL language. There is one alternative method `SetValueTranspose()` reserved for matrices which are transposed first and assigned after.

```
//Create the effect. The parameters are the owning device, effect file
//name, eventual Include instance, set of ShaderFlags flags, eventual
//EffectPool instance and the string variable where the compiler's error
//is stored.
Effect effect = Effect.FromFile(device, "effect.fx", null,
               ShaderFlags.None, null, out errors);

//retrieve a handle on a global variable defined in effect file
EffectHandle hMatrix = effect.GetParameter(null, "viewMatrix");

…

//setting the value of the global variable using a its handle
//matrices has to be transposed because DirectX treats vectors as rows
//while in graphics HW are vectors treated as columns
effect.SetParameterTranspose(hMatrix, viewMatrix);

//selecting the technique
effect.Technique = effect.GetTechnique("t0");
//inicialize effect
effect.Begin(0);
//select pass – number of available passes are returned by the Begin method
effect.Pass(0);

//draw some primitives
device.SetStreamSource(0, vertexBuffer, 0);
device.VertexFormat = CustomVertex.PositionColored.Format;
device.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);
```

```
//end of the effect
effect.End();
```

Sample Code 5: Effect creation and effect rendering.

Before the effect can be set the technique must be selected first. It is done by assigning its handle obtained from the `GetTechnique()` method into the `Technique` property. The required technique may be specified into the `GetTechnique()` method both by its name and its index. Once the technique is selected then the effect can be initialized by calling the `Begin()` method. The method has one parameter, which determines if the current render state is saved and then restored when `End()` method is called, and returns the number of passes existing within the selected technique. Render state values stored in a single pass are then set by calling the `Pass()` method with the number of the pass as a parameter. The primitive drawing may then proceed. The effect influence is ended by calling the `End()` method.

4.2 Point Sampling Method Implementation

This method is a direct extension of the method used by the trichromatic renderers. Instead of working with the vectors of three elements the vectors with the arbitrary number of elements are used instead. Illumination calculation then proceed as usual, i.e. the spectra of incoming light are multiplied in a piecewise manner with the reflectance, transmittance or with any other material's properties also represented as vectors of the same dimension.

The computed values are then scaled according to the used illumination model. At the end the spectral power distribution of light reaching the viewer is known. The spectrum's values are converted to the *XYZ* tristimulus values using the principles described in the section 3.1.2 and then converted to the chosen color space most appropriate for displaying, e.g. **RGB**. Steps involved in the process are illustrated in the Fig. 4-1.The Source code D-2 performing the conversion of a spectral power distribution to RGB tristimulus values is included in Appenidx D. Performance of the point sampling and trichromatic method is compared in the Table 4.2-1.

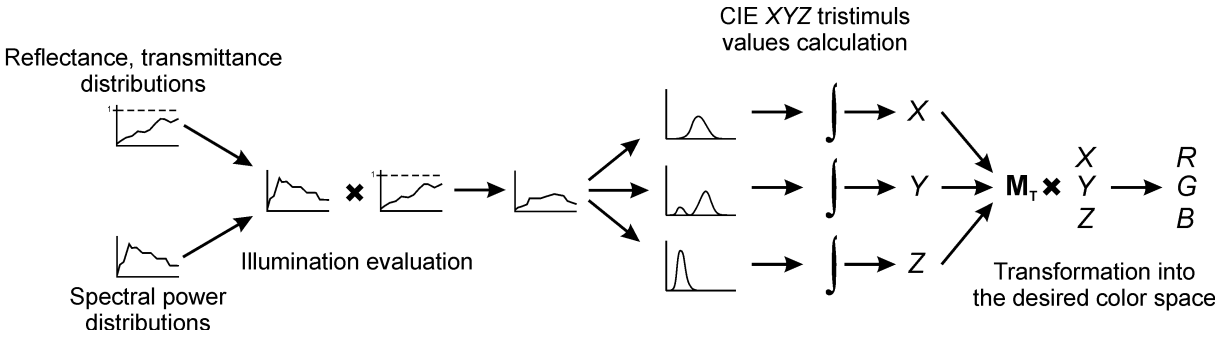

Fig. 4-1: Process of spectral rendering using the point sampling method.

The defining factor of this method's accuracy is a number of used samples. The more samples are used the more accurate result one gets. Although the quality gain from certain number of samples is usually not worth of the increased computation time. If there is some prior knowledge of the used spectra's shapes then the required number of samples and their location should be based on that knowledge. For example, the fluorescent light sources happen to have a quite smooth spectrum except of the several high spikes at several wavelengths. If such spectrum is not sampled in those significant wavelengths then one gets quite different spectrum and consequently a different colors. See Fig. E-1 in Appendix E for examples.

The other option is to sample the spectra uniformly across the wavelength range. Advantage of this method is its simple implementation, for it does not need no data survey and preprocessing. The presented solution provides a partial solution of this problem. The solution is capable to handle non-uniformly spaced samples, for it requires the corresponding wavelengths of the samples as an input. However, it is up to the client to provide adequate data including the optimal sample positions.

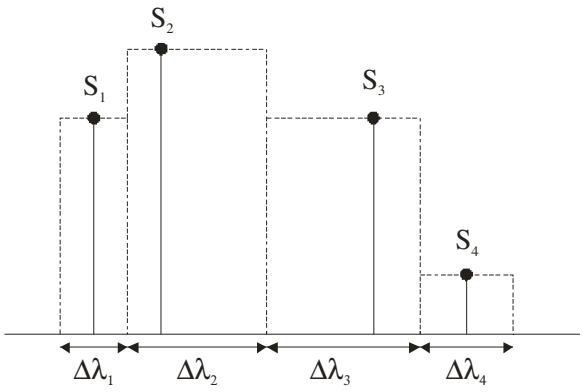

Fig. 4-2: Wavelength steps as assigned to non-uniformly spaced sample points.

With regard to sampling positions the wavelength steps each sample represents are obtained as shown in the Fig. 4-2. The length of each step is given as an average of the distances form a given sample point position to the position of the previous and the following sample point. The corresponding step of the last and the first sample point is identical as the distance to the position of the previous and the following sample point. The steps computed in this way are used for Riemann summation when tristimulus values are computed, see expression E[3.2-2].

Point samples	Time [s]
3 – trichromatic eval.	47.0
7 samples	53.0
16 samples	54.0
61 samples	61.0
301 samples	75.0

Table 4.2-1: Performance comparison of the trichromatic method and the point sampling method for various number of sample points. Scene was computed using the ray tracing however only direct illumination component was evaluated.

4.2.1 Real time implementation

The implementation of the point sampling method for the GPU system is not as straightforward as it is for non-interactive system. The GPU has limited number of operation available for pixel shader and the conversion of the spectral information into the trichromatic representation won't fit into that limit which is currently 64 arithmetic operations for pixel shader of version 2.0, although these numbers may change in a future.

The solution is to use multiple passes. In each pass are processed four samples and the results are stored in a temporary float texture for further processing. There are four samples per one pass because the registers of the GPU also consist of four float values and thus the vectors of four float components may be considered as the GPU's native data type. The float texture format, denoted as `Format.A32B32G32R32F` in DirectX, provides sufficient precision to store the computed values correctly.

Those four samples in each pass are in the graphics pipeline of the GPU treated as ordinary RGBA color information except that the A channel is not considered as transparency but as another color channel. Light's color and appropriate number of parameters describing the material's properties, i.e. diffuse reflectance, specular reflectance, transmittance etc., are usually needed for the evaluation of the illumination model. The material's properties may be provided as a constant value or as a texture. What justifies this method is the presence of wavelengths of the currently processed samples. This information is used in advanced illumination models so greater realism can be achieved. The whole frame is rendered into the float texture and it is stored there for further processing.

$$X = \sum_{t=1}^{3}\sum_{s=1}^{4} v_{ts}\Delta\lambda_{ts}x_{ts},\ Y = \sum_{t=1}^{3}\sum_{s=1}^{4} v_{ts}\Delta\lambda_{ts}y_{ts},\ Z = \sum_{t=1}^{3}\sum_{s=1}^{4} v_{ts}\Delta\lambda_{ts}z_{ts} \qquad \text{E[4.2-1]}$$

v_{ts}	Value of sample number *s* in texture *t*.
$\Delta\lambda_{ts}$	Corresponding wavelength step of the sample number *s* in texture *t*.
x_{ts}, y_{ts}, z_{ts}	Values of the xyz matching functions at the wavelength of the sample number *s* in texture number *t*.

As soon as three textures are computed using the process described above, i.e. 12 sample points were processed, they may be added into the texture where *XYZ* tristimulus values are accumulated. The values in the textures cannot be added directly. There are three sums E[4.2-1] computed first and then their value are added to the previously accumulated values stored in the texture. Pixel shader code performing this task is listed in the Source code D-4 as a function `ps_combine3_main`.

The number of temporary textures was chosen as a compromise between memory demands of a float texture and the number of summation passes. If there is only one temporary texture than there would be the same number of summation passes as there are rendering passes. If there are more temporary textures then the memory demands would be very high, for one float texture of resolution 640×480 occupies approximately 5 MB of memory.

The pixel operations performed over a texture or textures in a screen space are in the GPU performed using a specially configured geometry. The geometry consists of two triangles whose vertices positions and texture coordinates are calculated using the vertex shader program listed in the Source code D-4 as a function `vs_align_main`. As a result, the triangles cover the whole area of a render target independently on view or projection transformation and the pixels position obtained from the triangles rasterization match with the pixels position in a texture computed using the texture coordinates provided the resolution of the render target and the mapped texture are the same.

The need of two textures for *XYZ* tristimulus values accumulation comes from the fact that it is not possible to write values directly into a texture within a pixel shader code. However it is possible to set a texture as a render target and therefore the pixel shader's output values are written into that texture. Hence two textures are used for values accumulation. They are swapped periodically so that the texture used as a render target contains the most actual values and is used in the next step as inputting texture while the other one is used as the next render target.

Once all samples are processed, then the values in the accumulation texture are converted into the native **RGB** representation using the pixel shader program listed in the Source code D-4 as a function `ps_xyz2rgb_main`. The *RGB* values are obtained by multiplying the *XYZ* tristimulus values with the chromatic adaptation transformation, which is an identity matrix if this transformation is not desired, and consequently with the RGB transformation matrix. The final values are clamped to the quantization range, i.e. 0.0 – 1.0.

Sample points	Passes	Frames per second
301	100	0.4
61	22	1.85
16	6	7.0
7	3	14.5
3 – trichromatic eval.	1	44.0

Table 4.2-2: Frame rate in relation with the number of used sample points.

The described process is capable to handle the interval from 400 nm to 700 nm sampled by 10 nm steps in eleven rendering passes and the most recent GPU's are capable to perform that task while maintaining interactive frame rate, see the Table 4.2-2. Disadvantage of the method is that it needs five float textures of the render target's resolution hence it has high memory requirements. Next disadvantage is the speed of the evaluation, for one frame has to be rendered more than once. On the other hand, advantage of the method is that the wavelength information is present during the illumination process and may be used in advanced illumination models. Another advantage is that the method doesn't need pre-processing hence the spectral data may be changed dynamically without efficiency penalty.

4.3 Color pre-filtering

The color pre-filtering is a simple technique described in the section 3.3.2. It is a pre-processing technique and is done only once before any rendering is performed, however it has to be redone if the lighting conditions within a scene have changed. What follows next is a standard trichromatic rendering procedure.

4.3.1 Real Time Implementation

There is no special treatment of this method for a real time rendering system. The colors were already computed in the pre-processing stage and the following rendering is completely the same as usual. This is the biggest advantage of this method. Disadvantage of this method is that the preprocessing must be repeated if lighting conditions are changed. However, the necessity to repeat the preprocessing if lighting conditions change is not so fatal because from the measured data is apparent that it can be done very quickly, see the Table 4.3-1 for results.

Another disadvantage of this method is the loss of the continuous wavelength information which could be used in advanced illumination models' evaluation. Although the primaries of the used trichromatic system may have corresponding wavelengths only three of them are not sufficient for a comprehensive simulation.

The color pre-filtering is quite suitable for real time applications because it provides reasonably accurate results if only direct illumination component is considered which is true for most of the real time applications. The dynamic lighting changes can be still handled in a real time if the number of materials in a scene is not enormously large.

Sample points in the input data	Pre-processing time [s]
301	0.00035
61	0.0001
31	0.00008

Table 4.3-1: The pre-processing times in relation with the number of sample points of the input spectral data. Test scene contained two lights and 35 different materials.

4.4 Linear Color Representation

Linear color representation is an acceleration method for full spectral rendering introduced in the section 3.3.1. Spectra are replaced by their coordinates in the pre-computed ortho-normal basis and the illumination process is expressed using the matrix multiplication.

The most important part of this method is the selection of the basis function. They are selected automatically by the singular value decomposition of the matrix obtained by concatenating the involved spectra represented as columns. The spectra included in that matrix are taken from the spectral power distribution of the light sources present in the scene. Then the spectra obtained by multiplying the spectral power distributions with the material's reflectance or transmittance are also included.

The previous procedure handles the direct illumination component. If required, higher order components can be included by repeating previous procedure step with the newly acquired spectral power distributions. Clearly, each spectral power distribution has to be multiplied with reflectance or transmittance of each material in the scene. As a consequence the number of acquired spectra rises very quickly and the resulting matrix becomes too large to process in a reasonable time.

Another aspect of the singular value decomposition effectiveness is the relation between the error E[3.3-10], dimension of the matrix and the number of selected basis functions. Experiments show that the basis function acquired form the SVD performed over the matrices of dimension $m \times n$ where $m << n$ are less representative, i.e. the error E[3.3-10] is larger for the same number of basis functions, than the basis acquired form the SVD performed over the matrices where $m \leq n$ or $m > n$. Results of the experiment are presented in the Fig. 4-3.

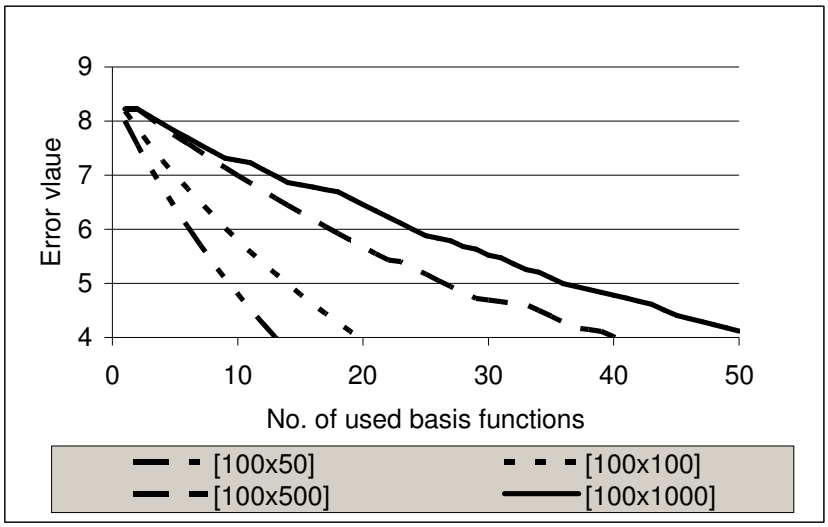

Fig. 4-3: Graph representing the dependency of the error E[3.3-10] on dimensions of a matrix.

If higher precision is required then more basis functions have to be picked. Of course that increases the number of operations required for matrix multiplication during the illumination process. If the time required for the pre-processing is summed with the time of the actual illumination calculation then it may exceed the time required for point sampling method and that practically invalidates the method. The measured times of the pre-processing are presented in the Table 4.4-2. The rendering times for different number of basis functions are compared with the rendering time of the same scene rendered using the point sampling method with 301 sample points in the Table 4.4-3.

Matrix Dimension	SVD evaluation time [s]
100×100	0.215
100×200	0.585
200×200	1.934
200×400	6.485
400×400	29.065

Table 4.4-1: Singular value decomposition performance.

Dimension of a matrix	Time [s]
301×75	0.2519
61×75	0.0671
31×75	0.0414
16×75	0.0342
7×75	0.0291

Table 4.4-2: Preprocessing times for different number. The test scene contained two lights and 35 different materials. The times correspond to the pre-processing of four basis functions and includes the time required for performing the SVD.

The actual SVD is a quite complex task and its efficient implementation is difficult. The SVD code presented in [Press04] was adapted for the purposes of this paper. The listing of the code can be found in the Appendix D. The times required for computation of different matrices dimension are presented in the Table 4.4-1.

Basis functions used	Time [s] direct component only	Time [s] traced reflections
1	52	86
2	52	87
3	53	88
4	54	88
5	54	88
301 sample points	75	118

Table 4.4-3: Performance of the linear color representation in relation with the number of used basis functions. The last row contains the referential values obtained form the point sampling method.

From the Fig. E-3 in Appendix E is evident that the color precision improvements are significant for three and more used basis functions with comparison with point sampling method that use the appropriate number of sample points.

4.4.1 Real Time Implementation

The specifics of the real time implementation of this method are similar with the color filtering, for this method is also preprocessing system and once done the actual rendering is

simple. The preprocessing involves the calculation of the basis functions, the coordinates of light sources, materials' matrices and transformation matrix transforming the linear coordinates into the XYZ tristimulus values. The actual illumination evaluation is listed in the Source code D-3 as a function `LightPhongLinear`.

From the sample code is evident that the only difference from the standard trichromatic illumination evaluation is that the linear method is multiplying the material's matrix with the light's coordinates instead of material's color with the light's color. Other coefficients like the diffuse reflection coefficient or the specular highlight coefficient are computed as usual. The final coordinates are converted into the *XYZ* tristimulus values by multiplying them by the pre-computed transformation matrix.

Performance of this implementation is not dependant on the number of basis functions because there are always considered 4 of them. If less then four basis functions are available then the unused components in matrices and light's coordinates have to be set to zero. The method maintained approximately 30 frames per second.

4.5 Illumination Models Implementation

Three illumination models were implemented for comparison. They are:

- Phong's illumination model, see section 2.2.1
- Strauss's illumination model, see section 2.2.4
- Blinn's illumination model, see section 2.2.3

These models were chosen because they are defined as the alternative BRDF, see section 2.1.3, and because they vary from a simple empirical model – Phong's model – to a complex physically based model – Blinn's illumination model.

The first examined aspect was evaluation efficiency of each illumination model because the evaluation is performed many times in each rendered frame of a scene. The measured data are presented in the Table 4.5-1. The Phong's illumination model is the easiest to evaluate and thus it achieved the shortest evaluation time. The Blinn's model has more complex specular part and thus it achieved the longest evaluation time. The Strauss's model has placed second. It is no surprise because the Strauss's model is a compromise between the simplicity of the Phong's model and the accuracy of the Blinn's model.

Illumination model	Rendering time reflections included [s]
Phong's model	118
Blinn's model	166
Strauss's model	122

Table 4.5-1: Performance of the tested illumination models in ray tracing that used point sampling method with 301 sample points. Reflections were ray traced to increase the number of illumination evaluations.

The second examined aspect was the visual accuracy and the ability of a model to profit from the extended material's properties definition of the spectral rendering system. From the comparison of the images rendered using the available illumination models, see Fig. E-12 and E-13, is apparent, that the differences are only subtle.

4.5.1 Real time implementation

There are two ways of computing the illumination from the programmable pipeline view. The first way is to evaluate the illumination model at the vertices within the vertex shader and the other points are interpolated, see section 2.3. The advantage of this way is that the vertex stage has available sufficient instruction slots, sees section 3.4.1, to implement the advanced illumination models, e.g. Blinn's illumination model. The disadvantage of this way is that the shading may not provide visually acceptable results if the geometry tessellation is too sparse, see Fig. E-13.

The second way is to evaluate the illumination model at each pixel within the pixel shader. The visual improvement is significant, see Fig. E-14, but the pixel stage has much less instruction slots available then the vertex stage has. This restricts the number of processed light sources and the most advanced illumination models might not fit at all.

Illumination model	Per vertex illumination [fps]	Per pixel illumination [fps]
Phong's	44.5	27.0
Blinn's	26.0	27.0 – one light source
Strauss's	22.0	n/a

Table 4.5-2: Performance of the illumination models' HLSL implementation.

The functions implementing the illumination models are listed in the source code D-3. The function `LightPhong` implements Phong's illumination model. It is the shortest one and it can be evaluated for two light sources even in the pixel stage. The function `LightBlinn` implements the Blinn's illumination model. Although this model is the most advanced from the considered three models it is not the longest one. Therefore, it is possible to evaluate one light source in the pixel stage. The function `LightStrauss` implements the Strauss's illumination model. This model consumed the highest number of instructions for implementation. The number of instructions actually exceeded the number of available number of instruction slots in the pixel stage. Therefore, only per vertex lighting is available for this model.

Performance of each model is recorded in the Table 4.5-2. The corresponding outputs are illustrated in the Fig. E-13 and E-14 in Appendix E.

4.6 Common Issues

There are several problems related with the spectral rendering which have to be solved sufficiently to obtain the most accurate results. Three the most important are chromatic adaptation compensation, the scale of the luminance values and the observer's specification.

4.6.1 Chromatic Adaptation

The effect of the chromatic adaptation along with the appropriate solution is presented in the section 3.2.2. The *XYZ* tristimulus values are transformed using the transformation matrix. The transformation matrix is obtained form the expression E[3.2-5]. The required data are the tristimulus values of the white of the target system and the tristimulus values of the source white stimulus used in the scene. The source white stimulus is computed using the spectral power distribution of the dominant light. The selection of the dominant light is left on a user and it is identified from its first position in a scene description file.

Three chromatic adaptation matrices are available. They are the $\mathbf{M}_{CAT2}$, $\mathbf{M}_{CAT3}$ and $\mathbf{M}_{CAT3}$ enumerated in the section 3.2.2. Effect of the Chromatic adaptation is illustrated in the Fig. E-6 and E-7 in Appendix E.

4.6.2 Normalization

The normalization of the inputting spectral data was discussed in the section 3.2.2. All spectral data are scaled appropriately so that luminance value 1.0 is mapped on the upper quantization range, which is usually 1.0 as well. Specialized tool was designed that performs this normalization. The description of this tool can be found in the section 4.7.

4.6.3 Observer Parameters

There are specified two observers, two degree and ten degree CIE observer, see 3.1.2.3. The two degree observer is valid for color matching of small objects while the ten degree observer is valid for matching of the larger objects.

Large objects of a solid color are rather rare in the usual scenes from a real word nad thus the two degree observer is considered as default. However, the option to use the ten degree observer was implemented. Comparison of the scene rendered using the two degree and ten degree is shown in the Fig. E-8 and E-9 in Appendix E.

4.7 Spectral Rendering Application

There is one application that is used for scene files editing and rendering. Both real time and off-line render system are handled from this application. Additionally it provides a tool that resamples the used spectral data and a tool that compares the selected images and visualizes their differences in the CIE Luv space.

All programs were developed for Microsoft .NET platform in the Microsoft Visual Studio .NET system using the C# language.

4.7.1 Spectra Manager

The spectral data used in the spectral rendering are required to be aligned, i.e. the sample points are the same for all spectra. That means that the spectra stored in files have to be resampled first. This is done using the Spectra Manager tool, which is a part of the spectral rendering application. The values at the desired wavelength positions are simply linearly interpolated from the two closest neighboring sample points.

Another capability of this tool is the conversion of the spectral representation into the trichromatic representation. This capability was implemented for rendering pure RGB images used as a reference. The spectral power distributions are normalized first, see the paragraph below, and then the *XYZ* tristimulus values are computed. Those are simply transformed into the desired **RGB** space using the selected transformation matrix. The spectral properties are treated differently. They

The tool normalize the spectral power distributions so that the values does not have to be scaled before they are converted into the *RGB* tristimulus values, see section 3.2.2 for details. To each spectral power distribution is assigned a luminance value which should be the same as the *Y* component of the tristimulus representation obtained by integrating the power distribution weighted by the $\bar{y}$ matching function. If the computed luminance value is

different form the assigned one then each value of a spectrum is weighted in the ratio of the assigned and the computed luminance. This normalization is performed for each spectral power distribution before the resampling and the conversion into the trichromatic representation is initiated. It is not performed on the reflectance or any other property distribution.

There are two spectral data file formats recognized by the application both are text-based. The first one assumes uniform sampling and therefore on the first line is a triplet representing lower boundary, upper boundary and the sampling step. After this line follow the values at the sample points. The second data file is more general because it contains pairs representing the wavelength and the corresponding value, see Fig. 4-4. The wavelength values are assumed to be in nanometers.

```
400 700 100
0.2
0.3
0.6
0.8
```

```
400 0.2
500 0.3
600 0.6
700 0.8
```

Fig. 4-4: Examples of the spectral data files.

5 Conclusion

Possibilities and benefits of the full spectral rendering were investigated in this thesis. It is assumed that the full spectral rendering should provide more accurate colors then the more common trichromatic renderer because the actual spectral power distribution, which is needed for a comprehensive illumination evaluation, can't be reconstructed form the its trichromatic representation.

The experiments shows that the results of the illumination simulation performed with the use of trichromatic representation are often very close to those obtained from the illumination simulation preformed with the full spectral data. The exceptions to this conclusion are metameric colors, see 3.1.2.2.

If an application depends on the accurate color calculation, e.g. design industry, then the spectral rendering is capable to provide reliable data providing the inputting spectral data are also reliable. If an application does not depend on the accurate colors calculations but on the visual believability, e.g. games, the full spectral rendering is unnecessary and standard trichromatic renderer suffices.

There is one significant disadvantage of the spectral rendering and that is the lack of the reliable spectral data. There are no freely available spectral libraries and therefore there remain only two other options: purchase the commercial library or measure the data by spectrophotometer. Obviously, both methods are inconvenient for common user.

5.1 Illumination Models

In the chapter 2 was presented an overview of the common illumination models because of their close relation with the rendering. Physical accuracy and computational efficiency of the presented models varied from the simple empirical models – Phong's model – to the comprehensive physical simulation involving almost every aspect of light's interaction with objects' surface – He's model.

The ability of each model to profit from the extended color information provided by full spectral rendering system and thus the ability to provide more realistic results was examined. Three illumination models, namely Phong's, Strauss's and Blinn's illumination model, were implemented for comparison. The example images used in the comparison are presented in the Appendix E – Fig. E-10 to E-14.

From the comparison is evident that the improvement of the realism due to the spectral color representation is only subtle. Therefore from the rendering point of view the selection of the appropriate illumination model should be based rather on visual accuracy and efficiency then on physical accuracy. This conclusion enables to choose simpler model which is easier to evaluate.

5.2 Acceleration Methods

There were tested two methods of full spectral rendering acceleration. The first one is the linear color representation described in the section 3.3.1. This method proved to be very efficient and elegant because it converts the whole spectral rendering pipeline into the matrix multiplication. It provides sufficiently accurate results yet for three basis functions, see Fig. E-3 and E-4 in the Appendix E.

The second acceleration method is the color pre-filtering described in the section 3.3.2. This method is the fastest and the easiest to implement. It is the simplest way to perform full spectral rendering because the actual rendering is performed using the trichromatic representation. The color pre-filtering can handle the metamerism and therefore it practically makes the trichromatic rendering equivalent to the spectral rendering.

5.3 Real Time Spectral Rendering

The possibility of acceleration of the spectral rendering using the programmable graphics hardware was examined in this thesis. As a result, three real time techniques of spectral rendering were developed. The first technique used for reference purpose implements the spectral rendering using the point sampling method. The principle of this method is described in the section 4.2.1. From the performance measurement is evident that this technique is unable to maintain reasonable frame rate for higher number of sample points. On the other hand, the method is general and it serves well as a reference for other techniques.

The second real time technique is an implementation of the linear color representation acceleration method. The number of basis functions is limited to 4 because it is the maximal dimension of matrices handled natively. However from the output comparison is evident that it is sufficient number for most of the scenes, see Fig. E-3 and E-4 in the Appendix E. From the performance measurement is evident that this technique can maintain reasonably high frame rate.

The third real time technique is an implementation of the color pre-filtering method. Actually, this technique does not require any specialized code because the pre-filtering is purely pre-processing method and the procedure afterwards is identical with the trichromatic pipeline.

Three illumination models were experimentally implemented. They are the Phong's, Strauss' and Blinn's illumination models. The limited capabilities of the programmable pipeline enabled fully implement the models only at the vertex stage. Only the Phong's illumination model was fully implemented at the pixel stage. Blinn's model was restricted at the pixel stage only to evaluation of one light source and the Strauss' illumination model didn't fit to the pixel stage at all. Example images Fig. E-13 and E-14 of the model's real time implementation are presented in the Appendix E.

Notations

Abbreviations

2D	two dimensional
3D	three dimensional
BRDF	Bi-directional Reflectance Distribution Function
CG	Computer Graphics
CIE	Commission Internationale de l'Eclairage
GAF	Geometric Attenuation Factor
GPU	Graphics Processing Unit
HLSL	High Level Shading Language
IOR	Index Of Refraction
RGB	Red Green Blue
SVD	Singular Value Decomposition

Illumination Model Description

L	Directon to a light source
N	Normal of a surfaře
R	Mirror reflection direction of incident light
V	Direction to a viewer
H	Bisector of L and V
T	Tangent of a surfaře
H'	Projection of H onto a surfaře
P	Point on a surfaře
$h = \mathbf{H.N}$	
$i = \mathbf{H.V}$	
$j = \mathbf{N.L}$	
$k = \mathbf{N.V}$	
$l = \mathbf{R.V}$	
$m = \mathbf{H'.T}$	
$\alpha = \cos^{-1} h$	
$\beta = \cos^{-1} i$	
$\theta = \cos^{-1} j$	
$\theta' = \cos^{-1} k$	
$\psi = \cos^{-1} l$	
$\phi = \cos^{-1} m$	
p	metallness
o	smoothness
σ	roughness
d	diffuseness
s	specularness
$F(i)$	Fresnel term
$D(h)$	facet slope distribution function
$G(j,k)$	geometric attenuation factor
C^D	diffuse reflectivity
C^S	specular reflectivity

Colorimetry

XYZ	tristimulus values
$R_\lambda\ G_\lambda\ B_\lambda$	spectral tristimulus values
$\mathbf{XYZ}$	primary stimuli
$\mathbf{Q}$	color stimulus
Q	point of chromacity
rgb	chromacity coordinates
Φ_Q	spectral power distribution of the stimulus $\mathbf{Q}$
λ	wavelength
$\mathbf{Q}_\lambda$	monochromatic stimulus
$\bar{r}(\lambda), \bar{g}(\lambda), \bar{b}(\lambda)$	matching functions
$V(\lambda)$	luminous efficiency function
$\mathbf{M}_{CAT}$	transformation matrix to the post adaptation cone response space

Literature

[Blinn77] Blinn, J. F.: Models of Light Reflection for Computer Synthesized Pictures. *Proceedings of Siggraph '77*.

[Cook81] Cook R. L, Torrance K. E.: A Reflectance Model for Computer Graphics. *Computer Graphics, v15, n3, August 1981*.

[Hast02] Hast, A., Barrera T., Bengtsson, E.: Faster Bivariate Quadratic Shading through Simplified Setup. *Licentiate Thesis No. 7, Uppsala University 2002.*

[Havr00] Havran, V.: Heuristic Ray Shooting Algorithms. *Dissertation Thesis. Czech Technical University 2000.*

[He91] He, X. D., Torrance, K. E., Sillion, F. X., Geenberg, D. P.: A Comprehensive Physical Model for Light Reflection. *Computer Graphics, v25, n4, July 1991*.

[Kozl04] Kozlowski, T.: Theory of Color. *http://semmix.pl/color/*, *July 2004*.

[Kraus98] Krauskopf, J.: Color Vision. *Color for Science, Art and Technology, Elsevier Science B.V., 1998*.

[Lars97] Larson, G. W., Rushmeier, H., Piatko, C.: A Visibility Matching Tone Reproduction Operator for High Dynamic Range Scenes. *IEEE Transactions on Visualization and Computer Graphics, v3, n4. October – December 1997.*

[Marc98] Marcus, R. T.: The measurement of Color. *Color for Science, Art and Technology, Elsevier Science B.V., 1998*.

[Nas98] Nassau, K.: Fundamentals of Color Science. *Color for Science, Art and Technology, Elsevier Science B.V., 1998*.

[Oren92] Oren, M., Nayar, S. K.: Generalization of the lambertian model and implications for machine vision. *Technical Report CUCS-057-92, Department of Computer Science, Columbia University, New York, 1992.*

[Oren94] Oren, M., Nayar, S. K.: Generalization of Lambert's Reflectance Model. *Proceedings of SIGGRAPG '94.*

[Palm03] Palmer J. M.: Radiometry and Photometry FAQ. *http://www.optics.arizona.edu/Palmer/rpfaq/rpfaq.htm, April 2004*.

[Peer93] Peercy, M. S.: Linear Color Representation for Full Spectral Rendering. *Proceedings of Siggraph '93*.

[Phong75] Phong, B. T.: Illumination for Computer Generated Pictures. *Comm. of the ACM, v18, n6, p449-455, 1975*.

[Press04] Press, W. H., Teukolsky, S. A, Vetterling, V. T., Flannery, B. P.: Numerical Recipes in C. Second Editon. *Cambridge University Press. Available online at http://www.library.cornell.edu/nr/bookcpdf.html.*

[Schl94] Schlick C.: A Survey of Shading and Reflectance Models. *Computer Graphics Forum, June 1994*.

[Straus90] Strauss, P. S.: A Realistic Lighting Model for Computer Animators. *Computer Graphics 1990*.

[Suss01] Süsstrung, S., Holm, J., Finlayson G. D.: Chromatic Adaptation Performance of Different RGB Sensors. *IS&T/SPIE Electronic Imaging. SPIE Vol. 4300. January 2001*.

[Trow75] Trowbridge, T. S., Reitz, K. P.: Average irregularity representation of a roughened surface for ray reflection. *J. Opt. Soc. Am. 65, 5(May 1975) 531-536*

[Ward92] Ward, G., Eydelberg-Vileshin, E.: Picture Perfect RGB Rendering Using Spectral Prefiltering and Sharp Color Primaries. *Thirteenth Eurographics Workshop on Rendering, 2002*.

[Wysz00] Wyszecki, G., Stiles, W. S.: Color Science Concepts and Methods, Quantitative Data and Formulae. Second Edition. *John Wiley & Sons. 2000*.

Data sources

[1] http://cvrl.ioo.ucl.ac.uk/, Color Vision Research Laboratories, April 2004.

[2] http://www.cis.rit.edu/mcsl/online/cie.shtml, Munsell Color Science Laboratory, April 2004.

[3] http://speclib.jpl.nasa.gov/, ASTER spectral library, June 2004.

Appendix A: User's Manual

Features

- Real time and off-line spectral rendering of a scene described in a text file.
- Editing of scene files.
- Resampling of spectral data files.
- Computation of trichromatic representation.
- Image comparison.

Software Installation

There is no special installation procedure needed for this application. It is sufficient to copy the directory containing the application's binaries and all subdirectories from CD on hard disk to reduce the loading times.

Running the Application

The application needs windows XP, Microsoft .NET framework version 1.1 and DirectX version 9.0b for running. The redistributable version of the .NET framework and the DirectX can be found on the accompanying CD.

After all necessary components are installed the application can be launched from its location using the executable `RenderClient.exe`. The main window of the application should appear after successful launch, see Fig. A-1.

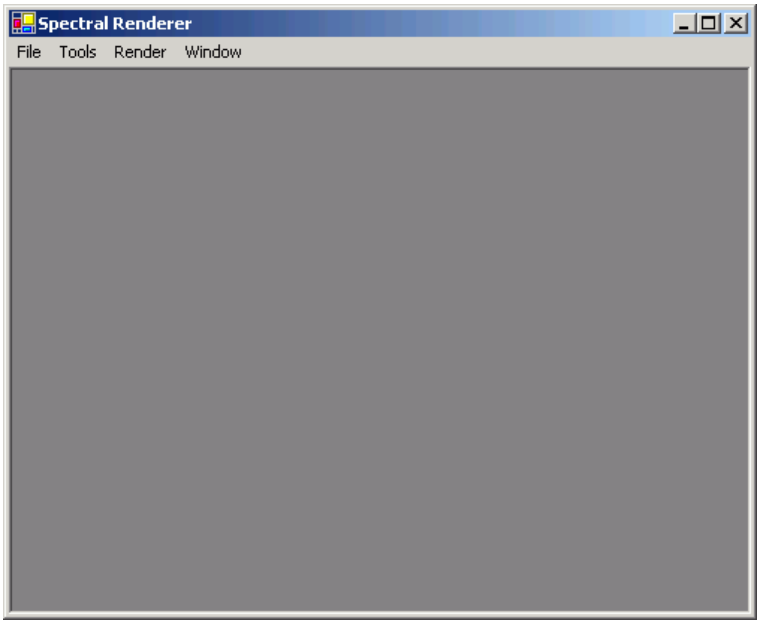

Fig. A-1: Main Windows of the application.

Main Menu

The application is controlled using the menu bar. It contains these items:

- File
  - New – creates an empty scene file.
  - Open – opens existing scene file.
  - Save – saves currently edited and modified scene file.
  - Save as – saves the currently edited scene file using a Save file dialog.

    - Exit – quits the application.
- Tools
    - Spectra Manager – launches the spectra manager tool.
    - Image Compare – launches the image compare tool.
- Render
    - Render Settings – opens the render settings dialog.
- Window – maintains the current list of the opened windows.

If a scene file is opened then these additional menu items are available:

- Render
    - Raytracer – triggers the off-line rendering of the current scene.
    - DirectX – triggers the real time rendering of the current scene.

While the window of the real time rendering system is opened then the View menu item is available:

- Device
    - HAL – switch the DirectX device into the HAL mode if possible.
    - Ref – switch the DirectX device into the reference mode.
- View
    - Show info – switch between showing and not showing the information objects, i.e. method name, frame rate and the axis icon on a screen
    - PS Lighting – switch the per pixel lighting on
    - VS Lighting – switch the per vertex lighting off

Scene Parsing Dialog

The scene parsing dialog appears after the rendering is launched, see Fig. A-2. It visualizes the progress of a scene parsing. The process can by canceled by pushing the cancel button. After the parsing is completed the OK button is enabled if pushed it starts the actual rendering. Also the preprocessing time of the selected spectral rendering method is displayed once it is available.

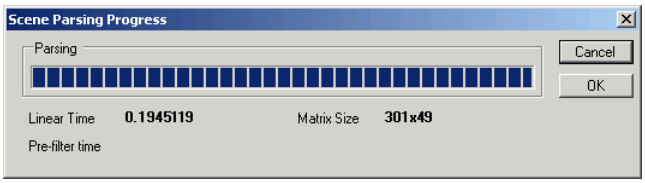

Fig. A-2: Scene file parsing dialog.

Render in Progress Dialog

The Render in progress dialog visualizes the progress of the off-line rendering, because it takes a while to complete. It contains the progress bar showing the progress and the active configuration. Rendering can be canceled by pushing the Cancel button or paused by the Pause check box. Once the rendering is completed, the cancel Button becomes OK button if pressed then the computed image is displayed.

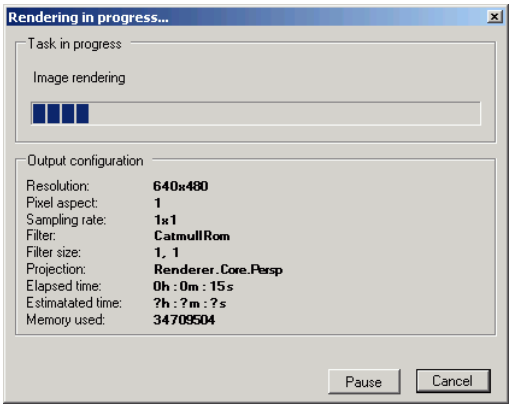

Fig. A-3: Render in progress dialog.

Render Setting Dialog

The render setting dialog, see Fig. A-4, can be used for changing the global render settings valid for both rendering system. The available values for each feature are enumerated in the combo boxes. The changes are confirmed by pushing the OK button.

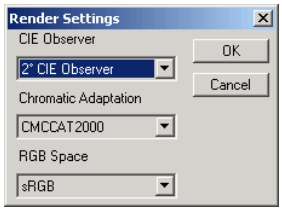

Fig. A-4: Spectra manager dialog.

Spectra Manager Tool

The spectral manager dialog consists of the material properties section and the light spectra section, see Fig. A-5. The files containing the spectral data are added into the list by pushing the corresponding Add button. Then the Open file dialog appears and the desired files can be selected. The unwanted files can be removed from list by selecting them in the list and pushing the corresponding Remove button.

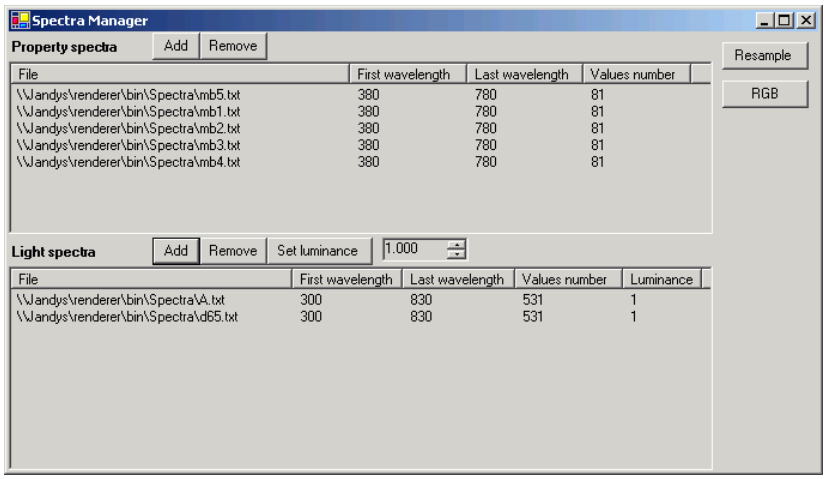

Fig. A-5: Spectra manager dialog.

The lists of the loaded spectra files show several statistical information. They are the name of the data file, a wavelength range of the spectrum a total number of samples contained

in the spectrum. Light spectra have additional information, which is the luminance assigned to the spectrum. It is initially set to 1.0 for all loaded spectra but it can be changed by selecting the spectra, typing the new luminance value into the text box and pushing the Set luminance button.

The loaded spectra then can be resampled by bushing the Resample button. Then the resample dialog appears, see Fig. A-6. The spectra can be resampled uniformly by selecting the starting wavelength, the ending wavelength and the sampling step. After pushing the Uniform button the resampling is performed and the values are written into the rich text edit box. The other option is to enter the desired sampling wavelengths into the text box beside the Select button. After busing the Select button, the sampled values are again written into the rich text box.

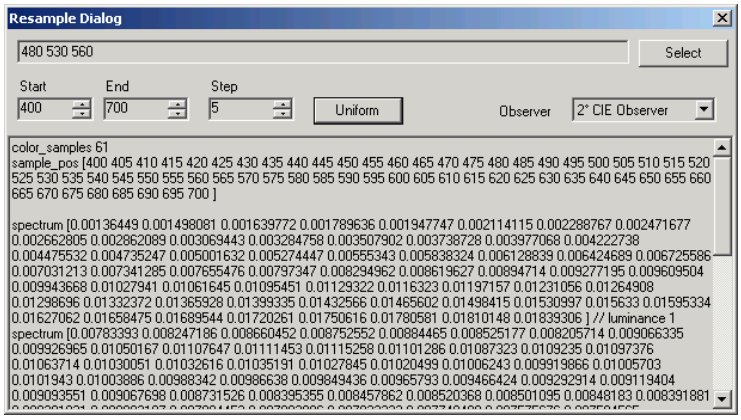

Fig. A-6: Resample dialog.

The light spectra are always written first and are normalized using the luminance value that each spectrum has assigned. The $\bar{y}$ matching function used for normalization is determined by the selected observer in the combo box Observer.

The spectra manger is also capable to compute the trichromatic representations of the loaded spectra. After bushing the RGB button the RGB dialog appears, see Fig. A-7. After selection of the conversion parameters using the Observer and RGB Transform combo box the computation is triggered by pushing the Go button. The computed values are written into the rich text box. Again, the light spectra are written as first.

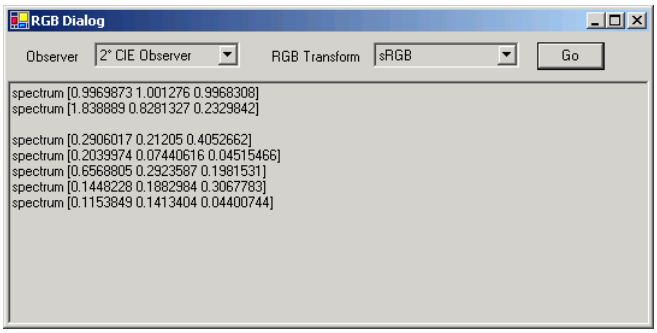

Fig. A-7: Resample dialog.

Image Compare Tool

The images to compare are selected by pushing the long buttons, see Fig. A-8. After the button is pushed the Open file dialog appears and the image can be selected. After both

images are selected the Compare button is enabled and can be pushed. If both images are of the same size then the window with the comparison is displayed or error message is displayed otherwise.

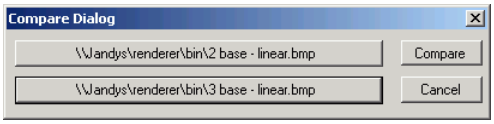

Fig. A-8: Resample dialog.

Appendix B: Programmer's manual

Application Architecture

The application is divided into two parts. The first part represents the rendering core system and it is stored in a class library. The second part is a client application which provides a graphics user interface for the rendering. This division enables to work on both parts independently provided the communication interface remains unchanged.

Appendix C: HLSL Basic Description

Presented text and the code samples were extracted from DirectX9.0 SDK documentation. Although minor modifications were made the whole Appendix E should be considered as a quotation.

Data Types

The following scalar types are supported:

- bool true or false (Boolean)
- int 32-bit signed integer
- half 16-bit floating point value
- float 32-bit floating point value
- double 64-bit floating point value

Integer, half and double values are emulated using float if the particular hardware does not have native support for those types.

Each of the scalar type has a vector alternative. A vector is a special data structure that contains between one and four components, see HLSL Code C-1. The integer immediately following the data type is the number of components on the vector.

```
Bool2   bVector;   // vector containing 2 Boolean
Int2    iVector;   // vector containing 2 int
float3  fVector = { 0.2f, 0.3f, 0.4f };   // vector containing 3 floats
```

HLSL Code C-1: HLSL - Vector data types

Vectors contain up to four components, each of which can be accessed using one of two naming sets:

- The position set: x,y,z,w
- The color set: r,g,b,a

Naming sets can use one or more components, but they cannot be mixed, see HLSL Code C-2. Specifying one or more vector components when reading components is called swizzling and masking when writing components.

```
// Given
float4 pos = float4(0,0,2,1);
float4 f_4D;
float2 f_2D;

f_2D = pos.xy  // valid
f_2D = pos.rg  // valid

f_2D = pos.xg  // NOT VALID because the position and color sets were used.

f_2D = pos.xx;   // components can be read more than once

f_4D.xz = pos.xz;  // write two components
f_4D.zx = pos.xz;  // change the write order

f_4D.xzyw = pos.w; // write one component to more than one component
f_4D.wzyx = pos;
```

HLSL Code C-2: Accessing the vectors components, swizzling and masking

Frequently used data type in the CG is a matrix. A matrix is a data structure that contains rows and columns of data. The data can be any of the scalar data types, however, every element of a matrix is the same data type. The number of rows and columns is specified with the "row by column" string that is appended to the data type. The maximum number of rows and/or columns is 4; the minimum number is 1.

```
int2x1    iMatrix;   // integer matrix with 2 rows, 1 column
int1x4    iMatrix;   // integer matrix with 1 row, 4 columns

float2x2 dMatrix { 0.0f, 0.1, // row 1
                   2.1f, 2.2f // row 2
                 }; // float matrix with 2 rows, 2 columns
float3x3 dMatrix;   // float matrix with 3 rows, 3 columns
```

HLSL Code C-3: Matrix type declaration.

A matrix contains values organized in rows and columns, which can be accessed using the structure operator "." followed by one of two naming sets.

- Zero-based, row-column position:
  - _m00, _m01, _m02, _m03
  - _m10, _m11, _m12, _m13
  - _m20, _m21, _m22, _m23
  - _m30, _m31, _m32, _m33
- One-based, row-column position:
  - _11, _12, _13, _14
  - _21, _22, _23, _24
  - _31, _32, _33, _34
  - _41, _42, _43, _44

A matrix can also be accessed using array access notation, which is a zero-based set of indices. Each index is inside of square brackets. A 4x4 matrix is accessed with the following indices:

- [0][0], [0][1], [0][2], [0][3]
- [1][0], [1][1], [1][2], [1][3]
- [2][0], [2][1], [2][2], [2][3]
- [3][0], [3][1], [3][2], [3][3]

Array access notation cannot use swizzling to read more than one component. However, array accessing can read a multicomponent vector. Matrix accessing rules are summed in the HLSL Code C-4.

```
float4x4 worldMatrix = float4({0,0,0,0}, {1,1,1,1}, {2,2,2,2}, {3,3,3,3});
float4x4 tempMatrix;
float2 temp;
float4 row;

temp = worldMatrix._m00_m11
temp = worldMatrix._m11_m00
temp = worldMatrix._11_22
temp = worldMatrix._22_11

temp.x = worldMatrix[0][0] // single component read
temp.x = worldMatrix[0][1] // single component read

row = worldMatrix[0] // read the first row
tempMatrix._m00_m11 = worldMatrix._m00_m11; // write two components
tempMatrix._m23_m00 = worldMatrix.m00_m11;
```

HLSL Code C-4: Matrix accessing rules.

The HLSL provide several object types. They are:

- sampler
- texture
- vertexshader
- pixelshader
- structure

A sampler contains sampler state. Sampler state specifies the texture to be sampled, and controls the filtering that is done during sampling. Four types of samplers: "sampler1D", "sampler2D", "sampler3D", and "samplerCUBE" are supported. Texture lookups for each of these are performed by their corresponding intrinsic function: "tex1D", "tex2D", "tex3D", and "texCUBE".

A "vertexshader" and “pixelshader” data type represents a vertex shader and pixel shader object. The "vertexshader" and “pixelshader” data type can be assigned when an assembly-language vertex shader is assembled. The "vertexshader" and “pixelshader” type can also be assigned when an HLSL vertex shader is compiled.

```
vertexshader vs =
  asm
  {
    vs_2_0
    dcl_position v0
    mov oPos, v0
  };

pixelshader ps =
  asm
  {
    ps_2_0
    mov oC0, c0
  };

pixelshader ps = compile ps_2_0 psmain();
vertexshader vs = compile vs_2_0 vsmain();
```

HLSL Code 6: Vertex shader object.

The "texture" data type represents a texture object. The data type is used in an effect to set a texture in a device. Once the texture variable is declared, it can be referenced by a sampler.

```
texture tex0;
sampler2D s_2D;
{
  texture = (tex0);
};


float2 sample_2D(float2 tex : TEXCOORD0) : COLOR
{
  return tex2D(s_2D, tex);
}
```

HLSL Code 7: Texture declaration and sampler inicialization.

The `struct` keyword defines a structure type. Any of the HLSL basic data types can be used in a structure. The structure operator "." is used to access members. Once a structure has been defined, it can be referenced by name with or without the `struct` keyword.

```
struct vertexData
{
    float3 pos;
    float3 normal;
};

struct vertexData data = { { 0.0, 0.0, 0.0 },
                           { 1.1, 1.1, 1.1 }
                         };
data.pos = float3(1,2,3);
data.pos = {1,2,3};
float3 temp = data.normal;
```

HLSL Code C-5: Structure definition and its member accessing.

Expressions and Statements

HLSL expressions are sequences of variables and literals, punctuated by operators. Expressions are composed of literals, variables, and operators. A literal is an explicit data value, such as 1 for an integer or 2.1 for a floating-point number. Literals are often used to assign a value to a variable. Operators determine how the variables and literals are combined, compared, selected, etc... The operators include:

- Assignment: `=`, `+=`, `-=`, `*=`, `/=`
- Unary: `!`, `-`, `+`
- Additive and multiplicative: `+`, `-`, `*`, `/`, `%`
- Boolean math: `&&`, `||`, `?:`
- Comparison: `<`, `>`, `==`, `!=`, `<=`, `>=`
- Prefix or postfix: `++`, `--`
- Cast: `(type)`
- Comma: `,`
- Structure member selection: `.`
- Array member selection: `[i]`

Many of the operators are "per component." This means that the operation is performed independently for each component of each variable. For example, a single component variable has one operation performed. On the other hand, a four-component variable has four operations performed, one for each component.

HLSL statements determine the order in which expressions are evaluated. Statements range in complexity from simple expressions to blocks of statements that accomplish a sequence of actions. Flow-control statements determine the order statements are executed. Statements are built from one or more of the following building blocks:

- An expression
- A statement block
- A return statement
- Flow-control statements
    - `o` if

- do
- for
- while

Expressions are built from operators, variables and literals. Any expression followed by a semicolon ";" is a statement.

A statement block is a group of one or more statements. A statement block also indicates subscope. Variables declared within a statement block are only recognized within the block.

The `if` statement chooses which statement block to execute next based on the result of a comparison. The comparison is followed by the statement block. An `if` statement can also use an optional `else` block. If the `if` expression is true, the code in the statement block associated with the `if` statement is processed. Otherwise, the statement block associated with the `else` block is processed.

The `do` statement executes a statement block, and then evaluates a conditional expression to determine whether to execute the statement block again. This is repeated until the conditional expression fails.

A `for` statement implements a loop that provides static control over the number of times a statement block will be executed. It contains an initialization expression, a comparison expression, and an increment (or decrement) expression, followed by a statement block. The statement block is executed each time that the comparison expression succeeds.

The `while` statement implements a loop, which evaluates an expression to determine whether to execute a statement block.

```
//if statement
if (dot(Normal, LightDirection) > 0 )
{
    // face is lit, so add a diffuse color component for example
}

//do statement
do
{
  // one or more statements
  color /= 2;
}
while ( color.a > 0.33f )

//for statement
for ( int i = 0; i < 2; i++)
{
   // one or more statements
}

//while statement
while ( color.a > 0.33f )
{
    color /= 2;
}
```

HLSL Code C-6: The flow-control statements demonstartion.

Vertex and Pixel Shader Semantics

The HLSL semantics identify where data comes from. Semantics are optional identifiers that identify shader inputs and outputs. Semantics appear in one of three places:

- After a structure member.
- After an argument in a function's input argument list.
- After the function's input argument list.

The use of the semantics is demonstrated in the HLSL Code C-7. The input structure identifies the data from the vertex buffer that will provide the shader inputs. This shader maps the data from the "POSITION" and "NORMAL" elements of the vertex buffer into vertex shader registers. The input data type for HLSL does not have to exactly match the vertex declaration data type. If it doesn't exactly match, the vertex data will automatically be converted into the HLSL's data type when it is written into the shader registers. For instance, if the "NORMAL" data were defined to be a `UINT` by the application, it would be converted into a `float3` when read by the shader.

```
struct VS_INPUT
{
    float4 vPosition : POSITION;
    float3 vNormal : NORMAL;
};

struct VS_OUTPUT
{
    float4  vPosition : POSITION;
    float4  vDiffuse : COLOR;

};

float4x4 mWld1;

float4 vLight;


VS_OUTPUT VS_Skinning_Example(const VS_INPUT v)
{
    VS_OUTPUT out;

    // Output stuff
    out.vPosition    = mul(mWld1,v.vPosition);
    out.vDiffuse  = dot(vLight,vNormal);

    return out;
}

float4 PixelShader_Sparkle(VS_OUTPUT In) : COLOR
{
    float4 Color = (float4)0;

    Color.rgb = In.vDiffuse;
    Color.w   = 1;

    return Color;
}
```

HLSL Code C-7: Use of vertex a pixel shader semantics.

The output structure identifies the vertex shader output parameters: position and color. These outputs will be used by the pipeline for triangle rasterization (in primitive processing). The output marked "POSITION" denotes the position of a vertex in screen space. In screen space, -1 and 1 are the minimum and maximum *x* and *y* values of the boundaries of the viewport, while *z* is used for *z*-buffer testing. As a minimum, a vertex shader must output "POSITION" data.

Input semantics for pixel shaders map values into specific hardware registers for transport between vertex shaders and pixel shaders. Each register type has specific properties. Because there are currently only two valid input semantics ("TEXCOORD" and "COLOR"), it is common for most data to be marked as "TEXCOORD", even when it is not.

Just like input semantics, output semantics identify data usage for pixel shader output data. Many pixel shaders write to only one output, "COLOR0". Pixel shader output colors must be of type `float4`. When writing multiple colors, all output colors must be used contiguously. In other words, "COLOR1" cannot be an output unless "COLOR0" has already been written. Pixel shader depth output must be of type `float1`.

The language defines the following vertex shader input semantics:

- POSITION[n] position
- BLENDWEIGHT[n] blend weights
- BLENDINDICES[n] blend indices
- NORMAL[n] normal vector
- PSIZE[n] point size
- COLOR[n] diffuse and specular color
- TEXCOORD[n] texture coordinates
- TANGENT[n] tangent
- BINORMAL[n] binormal
- TESSFACTOR[n] tessellation factor

where n is an optional integer between 0 and the number of resources supported. For example, PSIZE0, COLOR1, etc.

The vertex shader output semantics are:

- POSITION Position
- PSIZE Point size
- FOG Vertex fog
- COLOR[n] Color (example: COLOR0)
- TEXCOORD[n] Texture coordinates (example: TEXCOORD0)

where n is an optional integer between 0 and the number of registers supported. For example, texcoord0, texcoord1, etc.

The language defines the following pixel shader input semantics:

- COLOR[n] Diffuse or specular color (example: COLOR0)
- TEXCOORD[n] Texture coordinates (example: TEXCOORD0)

where n is an optional integer between 0 and the number of resources supported. For example, texcoord0, texcoord1, etc.

The pixel shader output semantics are:

- COLOR[n] Color (example: COLOR0)
- TEXCOORD[n] Texture coordinates (example: TEXCOORD0)
- DEPTH[n] Depth (example: DEPTH0)

where n is an optional integer between 0 and the number of registers supported. For example, COLOR0, TEXCOORD1, DEPTH0 etc.

Functions

High-level shader language (HLSL) functions are similar to C functions in several ways: They both contain a definition and a function body and they both declare return types and argument lists. Like C functions, HLSL validation does type checking on the arguments, argument types, and the return value during shader compilation.

Unlike C functions, HLSL entry point functions use semantics to bind function arguments to shader inputs and outputs (HLSL functions called internally ignore semantics). This makes it easier to bind buffer data to a shader, and bind shader outputs to shader inputs.

A function contains a declaration and a body, and the declaration must precede the body. A function declaration contains:

- A return type
- A function name
- An argument list (optional)
- An output semantic (optional)
- An annotation (optional)

The return type can be any of the HLSL basic data types such as a `float3`. The return type can be a structure that has already been defined. If the function does not return a value, `void` can be used as the return type. The return type always appears first in a function declaration.

An argument list declares the input arguments to a function. Parameter values are always passed by value. The in parameter usage indicates that the value of the parameter should be copied in from the calling application before the function begins. The out parameter usage indicates that the last value of the parameter should be copied out, and returned to the calling application when the function returns. The inout parameter usage is simply a shorthand for specifying both in and out.

```
float4 Light(float3 LightDir : TEXCOORD1,
             uniform float4 LightColor,
             float2 texcrd : TEXCOORD0,
             uniform sampler samp) : COLOR
{
```

```
    float3 Normal = tex2D(samp,texcrd);

    return dot((Normal*2 - 1), LightDir)*LightColor;
}
```

HLSL Code C-8: Example of the function declaration.

The function declared in the HLSL Code C-8 returns a final color, that is a blend of a texture sample and the light color. The function takes four inputs. Two inputs have semantics, "LightDir" has the "TEXCOORD1" semantic, and "texcrd" has the "TEXCOORD0" semantic. The semantics mean that the data for these variables will come from the vertex buffer. Even though the "LightDir" variable has a "TEXCOORD1" semantic, the parameter is probably not a texture coordinate. The "TEXCOORDn" semantic type is often used to supply a semantic for a type that is not pre-defined (there is no vertex shader input semantic for a light direction).

The other two inputs "LightColor" and "samp" are labeled with the "uniform" keyword. These are uniform constants that will not change between draw calls. The values for these parameters either come from shader global variables.

HLSL have a wade set of intrinsic functions. The several most interesting are:

- `cos(x)` Returns the cosine of `x`.
- `sin(x)` Returns the sine of `x`
- `tan(x)` Returns the tangent of `x`
- `cross(a, b)` Returns the cross product of two 3-D vectors `a` and `b`.
- `dot(a, b)` Returns the • product of two vectors, `a` and `b`.
- `lerp(a, b, s)` Returns `a + s(b - a)`.
- `mul(a, b)` Performs matrix multiplication between a and b.
- `normalize(v)` Returns the normalized vector `v / length(v)`.
- `tex1D(s, t)` 1-D texture lookup. `s` is a sampler or a `sampler1D` object. `t` is a scalar.
- `tex2D(s, t)` 2-D texture lookup. `s` is a sampler or a `sampler2D` object. `t` is a 2-D texture coordinate.
- `tex3D(s, t)` 3-D volume texture lookup. `s` is a sampler or a `sampler3D` object. `t` is a 3-D texture coordinate.
- `texCUBE(s, t)` 3-D cube texture lookup. `s` is a sampler or a `samplerCUBE` object. `t` is a 3-D texture coordinate.

Appendix D: Source Code Samples

```
//This code was adapted from the
//NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING
private static float Pythag(float a, float b)
{
     float absa,absb;
     absa = Math.Abs(a);
     absb = Math.Abs(b);

     if (absa > absb)
          return absa * (float) Math.Sqrt(1.0f + (absb / absa)
               * (absb / absa));
     else
          return (absb == 0.0f) ? 0.0f : absb *
               (float) Math.Sqrt(1.0f + (absa/absb) * (absa/absb));
}
//This code was taken from the
//NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING
//Matrix A is decomposed into U W and V so that A = UWV^T
public static void SingularValueDecomposition(float[,] A, out float[,] U,
out float[,] W, out float[,] V)
{
     U = null;
     V = null;
     W = null;
     int m = A.GetLength(0);
     int mb = m -1;
     int n = A.GetLength(1);
     int nb = n - 1;

     U = new float[m,n];
     V = new float[n,n];
     W = new float[n,n];

     CopyMatrix(A, out U);

     int flag,i,its,j,jj,k,l = 0, nm = 0;
     float anorm,c,f,g,h,s,scale,x,y,z;
     float[] rv1;
     rv1= new float[n];

     g = scale = anorm = 0.0f; //Householder reduction to bidiagonal form.
     for (i = 0; i < n; i++)
     {
          l = i + 1;
          rv1[i] = scale * g;
          g = s = scale = 0.0f;
          if (i <= mb)
          {
               for (k = i; k < m; k++)
                    scale += Math.Abs(U[k, i]);

               if (scale != 0.0f)
               {
                    for (k = i; k <m ; k++)
                    {
                         U[k, i] /= scale;
                         s += U[k,i] * U[k,i];
                    }
```

```
                f = U[i, i];
                g = -Math.Sign(f) * (float) Math.Sqrt(s);
                h= f * g - s;
                U[i,i]=f - g;
                for (j=l; j < n; j++)
                {
                     for (s = 0.0f, k=i; k < m; k++)
                          s += U[k,i] * U[k,j];
                     f = s / h;
                     for (k = i; k < m; k++)
                          U[k,j] += f * U[k,i];
                }
                for (k = i; k < m; k++)
                     U[k,i] *= scale;
           }
      }
      W[i, i] = scale * g;
      g = s = scale = 0.0f;

      if (i <= mb && i != nb)
      {
           for (k = l; k < n; k++)
                scale += Math.Abs(U[i,k]);
           if (scale != 0.0)
           {
                for (k=l; k < n; k++)
                {
                     U[i, k] /= scale;
                     s += U[i, k] *U [i, k];
                }
                f = U[i, l];
                g = -Math.Sign(f) * (float) Math.Sqrt(s);
                h = f * g - s;
                U[i, l] = f-g;
                for (k = l; k < n; k++)
                     rv1[k] = U[i, k] / h;
                for (j = l; j < m; j++)
                {
                     for (s = 0.0f, k = l; k < n; k++)
                          s += U[j,k] * U[i,k];
                     for (k = l; k < n; k++)
                          U[j,k] += s * rv1[k];
                }
                for (k = l; k < n; k++)
                     U[i, k] *= scale;
           }
      }
      anorm = Math.Max(anorm,(Math.Abs(W[i, i]) + Math.Abs(rv1[i])));
 }

 for (i = nb; i >= 0; i--)
 { //Accumulation of right-hand transformations.
      if (i < nb)
      {
           if (g != 0.0f)
           {
                for ( j = l; j < n; j++)
                     V[j, i]=(U[i, j] / U[i, l]) / g;
                for (j = l; j < n; j++)
                {
                     for (s = 0.0f, k=l; k<n; k++)
```

```
                         s += U[i, k] * V[k, j];
                    for (k = l; k < n; k++)
                         V[k, j] += s * V[k, i];
               }
          }
          for (j=l;j<n;j++)
               V[i, j] = V[j,i] = 0.0f;
     }
     V[i, i] = 1.0f;
     g = rv1[i];
     l = i;
}
for (i= Math.Min(mb,nb); i >= 0; i--)
{ //Accumulation of left-hand transformations.
     l = i + 1;
     g = W[i, i];
     for (j=l; j < n; j++) U[i, j] = 0.0f;
     if (g != 0.0f)// || g < 0.0) ***
     {
          g = 1.0f / g;
          for (j=l; j < n; j++)
          {
               for (s = 0.0f, k=l; k < m; k++)
                    s += U[k, i] * U[k, j];
               f = ( s / U[i, i]) * g;
               for (k = i; k < m; k++)
                    U[k, j] += f * U[k, i];
          }
          for (j = i; j < m; j++)
               U[j, i] *= g;
     }
     else
          for (j = i; j < m; j++)
               U[j, i] = 0.0f;
     ++U[i,i];
}
for (k = nb; k >= 0; k--)
{
//Diagonalization of the bidiagonal form: Loop over singular values,
//and over allowed iterations.
     for (its = 1; its <= 30; its++)
     {
          flag = 1;
          for (l = k; l >= 0; l--)
          {
               //Test for splitting.
               nm = l-1; //Note that rv1[1] is always zero.
               if ((Math.Abs(rv1[l]) + anorm) == anorm)  //change
               {
                    flag = 0;
                    break;
               }
               if ((Math.Abs(W[nm, nm]) + anorm) == anorm) //change
                    break;
          }
          if (flag != 0)
          {
               c = 0.0f; //Cancellation of rv1[l]
               s = 1.0f;
               for (i = l; i <= k; i++)
               {
```

```
                    f = s * rv1[i];
                    rv1[i] = c * rv1[i];
                    if ((Math.Abs(f) + anorm) == anorm)
                         break;
                    g=W[i, i];
                    h = Pythag(f,g);
                    W[i,i] = h;
                    h = 1.0f / h;
                    c = g * h;
                    s = -f * h;
                    for (j=0; j < m; j++)
                    {
                         y=U[j, nm];
                         z=U[j, i];
                         U[j, nm] = y * c + z * s;
                         U[j, i] = z * c - y * s;
                    }
               }
          }
          z = W[k, k];
          if (l == k)
          { //Convergence.
               if (z < 0.0f)
               { //Singular value is made nonnegative.
                    W[k, k] = -z;
                    for (j = 0 ; j < n; j++)
                         V[j, k] = -V[j,k];
               }
               break;
          }
          x = W[l, l]; //Shift from bottom 2-by-2 minor.
          nm = k - 1;
          y = W[nm, nm];
          g = rv1[nm];
          h = rv1[k];
          f =((y - z) * (y + z) + (g - h) * (g + h)) / (2.0f * h * y);

          g = Pythag(f,1.0f);
          f = ((x - z) * (x + z) + h * ((y / (f + Math.Sign(f) * g))
               - h)) / x;
          c = s = 1.0f; //Next QR transformation:
          for (j = l; j <= nm; j++)
          {
               i = j + 1;
               g = rv1[i];
               y = W[i,i];
               h = s * g;
               g = c * g;
               z = Pythag(f,h);
               rv1[j] = z;
               c = f / z;
               s = h / z;
               f = x * c + g * s;
               g = g * c - x * s;
               h = y * s;
               y *= c;
               for (jj=0; jj < n; jj++)
               {
                    x = V[jj, j];
                    z = V[jj, i];
                    V[jj, j] = x * c + z * s;
```

```
                        V[jj, i] = z * c - x * s;
                    }
                    z = Pythag(f,h);
                    W[j, j] = z; //Rotation can be arbitrary if z = 0.
                    if (z > 0.0f)
                    {
                        z = 1.0f / z;
                        c = f * z;
                        s = h * z;
                    }
                    f = c * g + s * y;
                    x = c * y - s * g;
                    for (jj = 0; jj < m; jj++)
                    {
                        y = U[jj, j];
                        z = U[jj, i];
                        U[jj, j] = y * c + z * s;
                        U[jj, i] = z * c - y * s;
                    }
                }
                rv1[l] = 0.0f;
                rv1[k] = f;
                W[k, k] = x;
            }
        }
}
```

Source Code D-1: Singular Value Decomposition. Code adapted from Numerical Recipes in C: The Art of Scientific Computing.

```
public class Spectrum2RGB : ColorModule
{
     float[] rgbMatrix;

     public Spectrum2RGB(SceneDatabase data): base(data)
     {
          rgbMatrix = GlobalRenderSettings.RGBSpaceTransform;
     }

     public override Color GetColor(Pixel pixel)
     {
          float x = 0,y = 0,z = 0;
          float r, g, b;
          int ir, ig, ib, iw;
          float[] steps = scene.displayAttributes.Steps;
          float temp;

          for (int i = 0; i < scene.xbar.Length; i++)
          {
               temp = pixel.color.components[i] * steps[i];
               x += scene.xbar[i] * temp;
               y += scene.ybar[i] * temp;
               z += scene.zbar[i] * temp;
          }
          if (GlobalRenderSettings.ChromaticAdaptation !=
               ChromaticAdaptation.None)
          {

               float[] adaptedXYZ = Utils.ColorTransform(new float[]
```

```
                    {x,y,z}, scene.catMatrix);
                x = adaptedXYZ[0];
                y = adaptedXYZ[1];
                z = adaptedXYZ[2];
            }

            r = x * rgbMatrix[0] + y * rgbMatrix[1] + z * rgbMatrix[2];
            g = x * rgbMatrix[3] + y * rgbMatrix[4] + z * rgbMatrix[5];
            b = x * rgbMatrix[6] + y * rgbMatrix[7] + z * rgbMatrix[8];

            ir = (int) (255.0f * r + 0.5f);
            ig = (int) (255.0f * g + 0.5f);
            ib = (int) (255.0f * b + 0.5f);
            iw = (int) (pixel.alpha * 255.0f + 0.5f);

            if (ir < 0) ir = 0; if (ir > 255) ir = 255;
            if (ig < 0) ig = 0; if (ig > 255) ig = 255;
            if (ib < 0) ib = 0; if (ib > 255) ib = 255;
            if (iw < 0) iw = 0; if (iw > 255) iw = 255;

            return Color.FromArgb(iw, ir, ig, ib);
        }
}
```

Source Code D-2: Transformation of the spectral representation into the RGB trichromatic representation.

```
public class LinearColor : ColorModule
{
     float[] rgbMatrix;

     public LinearColor(SceneDatabase data) : base (data)
     {
          rgbMatrix = GlobalRenderSettings.RGBSpaceTransform;
     }

     public override Color GetColor(Pixel ie)
     {
          float[] col = ie.color.components;

          float[] xyz = new float[3];

          for (int i = 0; i < col.Length; i++)
          {
               xyz[0] += col[i] * scene.tristimulusMatrix[0, i];
               xyz[1] += col[i] * scene.tristimulusMatrix[1, i];
               xyz[2] += col[i] * scene.tristimulusMatrix[2, i];
          }

          if (GlobalRenderSettings.ChromaticAdaptation !=
               ChromaticAdaptation.None)
          {
               xyz = Utils.ColorTransform(xyz, scene.catMatrix);
          }

          col = Utils.ColorTransform(xyz, rgbMatrix);

          int ir = (int) (255.0f * col[0] + 0.5f);
          int ig = (int) (255.0f * col[1] + 0.5f);
          int ib = (int) (255.0f * col[2] + 0.5f);
          int iw = (int) (ie.alpha * 255.0f + 0.5f);

          if (ir < 0) ir = 0; if (ir > 255) ir = 255;
          if (ig < 0) ig = 0; if (ig > 255) ig = 255;
          if (ib < 0) ib = 0; if (ib > 255) ib = 255;
          if (iw < 0) iw = 0; if (iw > 255) iw = 255;

          return Color.FromArgb(iw, ir, ig, ib);
     }
}
```

Source Code D-3: Transformation of the linear coefficinets into the RGB trichromatic representation.

```
//////////////////   COMMON   //////////////////
float viewport_inv_width;
float viewport_inv_height;
matrix world_matrix;
matrix view_matrix;
matrix projection_matrix;
matrix view_proj_matrix;

float4 light0Color;
float4 light1Color;
float4 light2Color;

float4 material0Color;  //diffuse
float4 material1Color;  //specular
float4 material2Color;  //user

float3 light0Pos;
float3 light1Pos;
float3 light2Pos;

float3 viewPos;

const float PI = 3.14159265358979323846f;

float diffuseness;
float specularness;
float roughness;
float shininess;
float metalness;
//float smoothness;
float3x3 rgbMatrix;
float3x3 catMatrix;

//////////////////   POINT SAMPLING   //////////////////

texture one;
texture two;
texture three;
texture four;
texture five;

float3x4 xyz0;
float3x4 xyz1;
float3x4 xyz2;

float4 wavelength;

float4 steps;

sampler Texture1 = sampler_state
{
    Texture = (one);
};

sampler Texture2 = sampler_state
{
    Texture = (two);
};

sampler Texture3 = sampler_state
{
```

```
     Texture = (three);
};

sampler Texture4 = sampler_state
{
   Texture = (four);
};

sampler Texture5 = sampler_state
{
   Texture = (five);
};

//////////////////   LINEAR COLOR REP.   //////////////////

matrix illumMatrix0;
matrix illumMatrix1;
matrix illumMatrix2;

float4x3 tristimulusMatrix;

//////////////////   STRUCTURES    //////////////////


struct VS_OUTPUT_A {
   float4 Pos: POSITION;
   float2 texCoord: TEXCOORD0;
};

struct VS_OUTPUT_SIMPLE {
   float4 Pos: POSITION;
   float4 Diffuse: COLOR0;
};

struct VS_OUTPUT_COMPLEX {
     float4 Pos: POSITION;
     float3 Normal: TEXCOORD0;
     float3 WorldPos: TEXCOORD1;
};

//////////////////   COMMON   //////////////////

float4 fresnel(float x, float4 rn)
{
     return rn + (1.0f - rn) * pow (1.0f - x, 5.0f);
}

float gaf(float x, float y, float m)
{
    float t = sqrt(2.0f * m * m / PI);
     return (x / (x - t*x + t)) * (y / (y - t*y + y));
}

float blinnSlope(float x, float m)
{
     float mm = m*m;
     return pow ((mm / (x*x*(mm-1.0f) + 1.0f)), 2.0f);
}

float SFresnel(float angle)
{
```

```
     float x = angle / 1.57079632679489661923132169163 98;
     float res = (x - 1.5);
     res *= res;
     res = 1.0f / res;

     return (res - 0.444444444444444444444444444444444) * 0.28125f;
}

float SGAF(float angle)
{
     float x = angle / 1.57079632679489661923132169163 98;
     float res = (x - 1.1);
     res *= res;
     res = 1.0f / res;

     return (100.0 - res) * 0.010083333333333333333333333333333;
}

//////////////////   POINT SAMPLING   //////////////////

VS_OUTPUT_A   vs_align_main(float4 Pos: POSITION)
{
     VS_OUTPUT_A Out;
     // Clean  up inaccuracies
     Pos.xy =  sign(Pos.xy);
     Out.Pos = float4(Pos.xy, 0, 1);
     // Image-space
     Out.texCoord.x = 0.5 * (1 + Pos.x + viewport_inv_width);
     Out.texCoord.y = 0.5 * (1 - Pos.y + viewport_inv_height);

     return Out;
}

float4 ps_combine1_main( float4 inDiffuse: COLOR0 , float2 inCoord:
TEXCOORD0) : COLOR0
{
   float4 color1 = tex2D(Texture1, inCoord);
   float3 color4 = tex2D(Texture4, inCoord);

   float3 color5 = mul(xyz0, color1) + color4;

   return float4(color5, 1.0);
}

float4 ps_combine2_main( float4 inDiffuse: COLOR0 , float2 inCoord:
TEXCOORD0) : COLOR0
{
   float4 color1 = tex2D(Texture1, inCoord);
   float4 color2 = tex2D(Texture2, inCoord);

   float3 color4 = tex2D(Texture4, inCoord);

   float3 color5 = mul(xyz0, color1) + mul(xyz1, color2) + color4;

   return float4(color5, 1.0);
}

float4 ps_combine3_main( float4 inDiffuse: COLOR0 , float2 inCoord:
TEXCOORD0) : COLOR0
{
```

```
   float4 color1 = tex2D(Texture1, inCoord);
   float4 color2 = tex2D(Texture2, inCoord);
   float4 color3 = tex2D(Texture3, inCoord);

   float3 color4 = tex2D(Texture4, inCoord);

   float3 color5 = mul(xyz0, color1) + mul(xyz1, color2) + mul(xyz2,
color3) + color4;

   return float4(color5, 1.0);
}

float4 ps_xyz2rgb_main( float4 inDiffuse: COLOR0 , float2 inCoord:
TEXCOORD0) : COLOR0
{
   float3 color = tex2D(Texture1, inCoord);
   float3 result = mul(rgbMatrix, mul(catMatrix, color));

   result = saturate(result);

   return float4(result, 1.0);
}

float4 LightPhong(float3 pos, float3 light, float3 N, float4 lightColor)
{
     float3 L = normalize(light - pos);
     float3 V = normalize(viewPos - pos);
     float3 H = normalize(L + V);

     float d = max(dot(N, L), 0.0);
     float s = max(dot(N, H), 0.0);
     return (diffuseness * material0Color * d + specularness *
material1Color * pow(s , shininess)) * lightColor;
}

float4 LightPhongLinear(float3 pos, float3 light, float3 N, float4
lightCoords)
{
     float3 L = normalize(light - pos);
     float3 V = normalize(viewPos - pos);
     float3 H = normalize(L + V);

     float d = max(dot(N, L), 0.0);
     float s = max(dot(N, H), 0.0);
     return  mul(illumMatrix0, lightCoords) * diffuseness * d +
mul(illumMatrix1, lightCoords) * specularness * pow(s , shininess);
}

float4 LightBlinn(float3 pos, float3 light, float3 N, float4 lightColor)
{
     float3 L = normalize(light - pos);
     float3 V = normalize(viewPos - pos);
     float3 H = normalize(L + V);

     float d = dot(N, L);

     float4 s;
     if (d > 0.0) s = gaf(dot(N, L), dot(N, V), roughness) *
                blinnSlope(dot(N, H), roughness) *
                fresnel(dot(L, H), material1Color) / dot(N,V);
                else s = 0.0;
```

```
     d = max(d, 0.0f);

     return (diffuseness * material0Color * d + specularness * s) *
lightColor;
}

float4 LightStrauss(float3 pos, float3 light, float3 N, float4 lightColor)
{
     float3 L = normalize(light - pos);
     float3 V = normalize(viewPos - pos);
     float3 H = normalize(L + V);

     float d = dot(N,L);

     float NH = dot(N, H);
     //float rd = (1.0f - smoothness * smoothness * smoothness);
     //float dd = rd * (1.0f - metalness * smoothness);
     //float n = 6.0f / (1.0f - smoothness);
     //float rs = 1.0f - rd;
     float ANL = acos(d);
     float fVal = SFresnel(ANL);
     float s;

     if (d < 0.0)
        s = 0.0;
     else
        s = min(1.0f, specularness + (specularness + 0.1f) * fVal *
SGAF(ANL) * SGAF(acos(dot(N, V))));

     float4 specularRef = lerp( material0Color, 1.0, fVal); //1.0f +
metalness * (1.0f - fVal) * (material0Color - 1.0f );

     d = max(d, 0.0);

     return lightColor * (material0Color * d * diffuseness + specularRef *
s * pow(NH, shininess));
}

VS_OUTPUT_COMPLEX vs_complex(float4 Pos: POSITION, float3 Normal: NORMAL)
{
     VS_OUTPUT_COMPLEX Out;
     Out.WorldPos = mul (world_matrix, Pos).xyz;
     //only translation and rotation is considered
     Out.Normal = mul (world_matrix, float4(Normal,0)).xyz;
     Out.Pos = mul ( projection_matrix, mul(view_matrix,
mul(world_matrix,Pos)));

     return Out;
}

VS_OUTPUT_SIMPLE vs_phong_simple_point(float4 Pos: POSITION, float3 Normal:
NORMAL)
{
     VS_OUTPUT_SIMPLE Out;

     float3 wPos = mul (world_matrix, Pos);
     float3 wNormal = mul(world_matrix, float4(Normal, 0)).xyz;
     Out.Pos = mul ( view_proj_matrix, float4(wPos,1));
     Out.Diffuse = (LightPhong(wPos, light0Pos, wNormal, light0Color)  +
LightPhong(wPos, light1Pos, wNormal, light1Color)) * steps;
```

```
    return Out;
}

VS_OUTPUT_SIMPLE vs_phong_simple_color(float4 Pos: POSITION, float3 Normal:
NORMAL)
{
    VS_OUTPUT_SIMPLE Out;

    float3 wPos = mul (world_matrix, Pos);
    float3 wNormal = mul(world_matrix, float4(Normal, 0)).xyz;
    Out.Pos = mul ( view_proj_matrix, float4(wPos,1));
    Out.Diffuse = LightPhong(wPos, light0Pos, wNormal, light0Color)  +
LightPhong(wPos, light1Pos, wNormal, light1Color);

    return Out;
}

VS_OUTPUT_SIMPLE vs_blinn_simple_color(float4 Pos: POSITION, float3 Normal:
NORMAL)
{
    VS_OUTPUT_SIMPLE Out;

    float3 wPos = mul (world_matrix, Pos);
    float3 wNormal = mul(world_matrix, float4(Normal, 0)).xyz;
    Out.Pos = mul ( view_proj_matrix, float4(wPos,1));
    Out.Diffuse = LightBlinn(wPos, light0Pos, wNormal, light0Color)  +
LightBlinn(wPos, light1Pos, wNormal, light1Color);

    return Out;
}

VS_OUTPUT_SIMPLE vs_blinn_simple_point(float4 Pos: POSITION, float3 Normal:
NORMAL)
{
    VS_OUTPUT_SIMPLE Out;

    float3 wPos = mul (world_matrix, Pos);
    float3 wNormal = mul(world_matrix, float4(Normal, 0)).xyz;
    Out.Pos = mul ( view_proj_matrix, float4(wPos,1));
    Out.Diffuse = (LightBlinn(wPos, light0Pos, wNormal, light0Color)  +
LightBlinn(wPos, light1Pos, wNormal, light1Color)) * steps;

    return Out;
}

VS_OUTPUT_SIMPLE vs_strauss_simple_color(float4 Pos: POSITION, float3
Normal: NORMAL)
{
    VS_OUTPUT_SIMPLE Out;

    float3 wPos = mul (world_matrix, Pos);
    float3 wNormal = mul(world_matrix, float4(Normal, 0)).xyz;
    Out.Pos = mul ( view_proj_matrix, float4(wPos,1));
    Out.Diffuse = LightStrauss(wPos, light0Pos, wNormal, light0Color) +
LightStrauss(wPos, light1Pos, wNormal, light1Color);

    return Out;
}

VS_OUTPUT_SIMPLE vs_strauss_simple_point(float4 Pos: POSITION, float3
Normal: NORMAL)
```

```
{
     VS_OUTPUT_SIMPLE Out;

     float3 wPos = mul (world_matrix, Pos);
     float3 wNormal = mul(world_matrix, float4(Normal, 0)).xyz;
     Out.Pos = mul ( view_proj_matrix, float4(wPos,1));
     Out.Diffuse = (LightStrauss(wPos, light0Pos, wNormal, light0Color) +
LightStrauss(wPos, light1Pos, wNormal, light1Color)) * steps;

     return Out;
}

VS_OUTPUT_SIMPLE vs_phong_simple_linear(float4 Pos: POSITION, float3
Normal: NORMAL)
{
     VS_OUTPUT_SIMPLE Out;

     float3 wPos = mul (world_matrix, Pos);
     float3 wNormal = mul(world_matrix, float4(Normal, 0)).xyz;
     Out.Pos = mul ( view_proj_matrix, float4(wPos,1));
     float4 temp = LightPhongLinear(wPos, light0Pos, wNormal, light0Color)
+ LightPhongLinear(wPos, light1Pos, wNormal, light1Color);
     Out.Diffuse = float4(mul(rgbMatrix, mul(catMatrix,
mul(tristimulusMatrix, temp).xyz)).xyz, 1);

     return Out;
}

float4 ps_phong_point(float3 normal: TEXCOORD0, float3 wPos: TEXCOORD1) :
COLOR0
{
     float3 N = normalize(normal);

     return (LightPhong(wPos, light0Pos, N, light0Color) + LightPhong(wPos,
light1Pos, N, light1Color)) * steps;
}

float4 ps_phong_color(float3 normal: TEXCOORD0, float3 wPos: TEXCOORD1) :
COLOR0
{
     float3 N = normalize(normal);

     return (LightPhong(wPos, light0Pos, N, light0Color) + LightPhong(wPos,
light1Pos, N, light1Color));
}

float4 ps_phong_color_linear(float3 normal: TEXCOORD0, float3 wPos:
TEXCOORD1) : COLOR0
{
     float3 N = normalize(normal);

     return (LightPhongLinear(wPos, light0Pos, N, light0Color) +
LightPhongLinear(wPos, light1Pos, N, light1Color));
}

float4 ps_blinn_point(float3 normal: TEXCOORD0, float3 wPos: TEXCOORD1) :
COLOR0
{
     float3 N = normalize(normal);

     return LightBlinn(wPos, light0Pos, N, light0Color) * steps;
```

```
}

float4 ps_blinn_color(float3 normal: TEXCOORD0, float3 wPos: TEXCOORD1) :
COLOR0
{
     float3 N = normalize(normal);

     return LightBlinn(wPos, light0Pos, N, light0Color);
}

float4 ps_strauss_color(float3 normal: TEXCOORD0, float3 wPos: TEXCOORD1) :
COLOR0
{
     float3 N = normalize(normal);

     return LightStrauss(wPos, light0Pos, N, light0Color);
}

float4 ps_strauss_point(float3 normal: TEXCOORD0, float3 wPos: TEXCOORD1) :
COLOR0
{
     float3 N = normalize(normal);

     return LightStrauss(wPos, light0Pos, N, light0Color) * steps;
}

technique point
{
     pass combine1 //0
     {
          VertexShader = compile vs_1_1 vs_align_main();
          PixelShader = compile ps_2_0 ps_combine1_main();
     }
     pass combine2 //1
     {
          VertexShader = compile vs_1_1 vs_align_main();
          PixelShader = compile ps_2_0 ps_combine2_main();
     }
     pass combine3 //2
     {
          VertexShader = compile vs_1_1 vs_align_main();
          PixelShader = compile ps_2_0 ps_combine3_main();
     }
     pass convert //3
     {
          VertexShader = compile vs_1_1 vs_align_main();
          PixelShader = compile ps_2_0 ps_xyz2rgb_main();
     }

     pass phongV //4
     {
          VertexShader = compile vs_2_0 vs_phong_simple_point();
          PixelShader = null;

          ZEnable = true;
          CullMode = NONE;
     }

     pass phongP //5
     {
          VertexShader = compile vs_2_0 vs_complex();
```

```
          PixelShader =  compile ps_2_0 ps_phong_point();

          ZEnable = true;
          CullMode = NONE;
     }

     pass blinnV //6
     {
          VertexShader = compile vs_2_0 vs_blinn_simple_point();
          PixelShader = null;

          ZEnable = true;
          CullMode = NONE;
     }

     pass blinnP //7
     {
          VertexShader = compile vs_2_0 vs_complex();
          PixelShader = compile ps_2_0 ps_blinn_point();

          ZEnable = true;
          CullMode = NONE;
     }

     pass straussV //8
     {
          VertexShader = compile vs_2_0 vs_strauss_simple_point();
          PixelShader = null;

          ZEnable = true;
          CullMode = NONE;
     }
   //too many instructions - not used
     pass straussP //9
     {
          VertexShader = null;
          PixelShader = null;

          ZEnable = true;
          CullMode = NONE;
     }
}

technique filter
{
     pass phong //0
     {
          VertexShader = compile vs_1_1 vs_phong_simple_color();
          PixelShader = null;

          ZEnable = true;
          CullMode = NONE;
     }

     pass blinn //1
     {
          VertexShader = compile vs_1_1 vs_blinn_simple_color();
          PixelShader = null;

          ZEnable = true;
          CullMode = NONE;
```

```
	}

	pass strauss //2
	{
		VertexShader =  compile vs_2_0 vs_strauss_simple_color();
		PixelShader = null;

		ZEnable = true;
		CullMode = NONE;
	}

	pass phongP //3
	{
		VertexShader = compile vs_2_0 vs_complex();
		PixelShader =  compile ps_2_0 ps_phong_color();

		ZEnable = true;
		CullMode = NONE;
	}
	pass blinnP //4
	{
		VertexShader = compile vs_2_0 vs_complex();
		PixelShader =  compile ps_2_0 ps_blinn_color();

		ZEnable = true;
		CullMode = NONE;
	}
	//too many instructions - not used
	pass straussP //5
	{
		VertexShader = null;
		PixelShader =  null;

		ZEnable = true;
		CullMode = NONE;
	}
}

technique linear
{
	pass phong
	{
		VertexShader = compile vs_2_0 vs_phong_simple_linear();
		PixelShader = null;

		ZEnable = true;
		CullMode = NONE;
	}
	//not implemented
	pass blinn
	{
		VertexShader = null;
		PixelShader = null;

		ZEnable = true;
		CullMode = NONE;
	}
	//not implemented
	pass strauss
	{
		VertexShader = null;
```

```
            PixelShader = null;

            ZEnable = true;
            CullMode = NONE;
        }
}
```

Source Code D-4: Effect code, implementing thr real time spectral rendering including the illuminaion models.

Appendix E: Figures

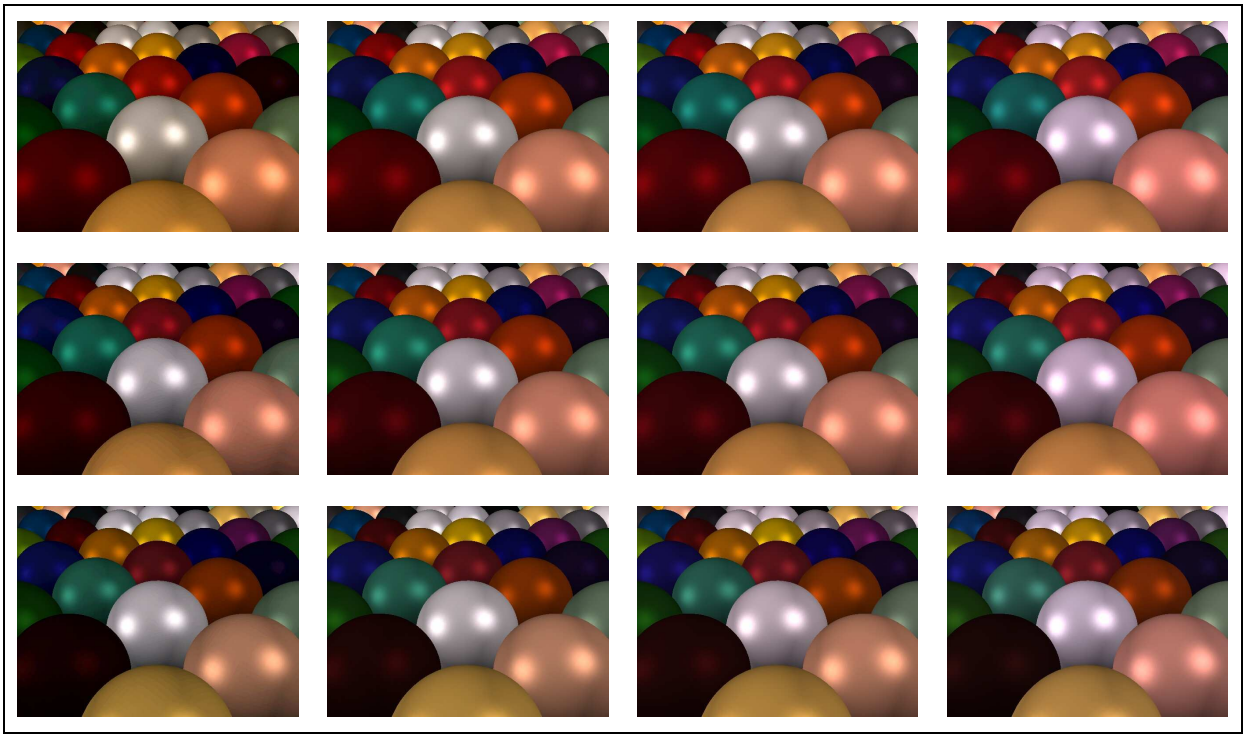

Fig. E-1: From left to right, result of point sampling method for 301, 61, 16 and 7 sample points uniformly distributed on the wavelength interval of range from 400 nm to 700 nm. Scene in a first row is illuminated by the CIE A illuminant, scene in the second row is illuminate by the CIE D65 illuminant and the scene in the third raw is illuminated by fluorescent illuminant.

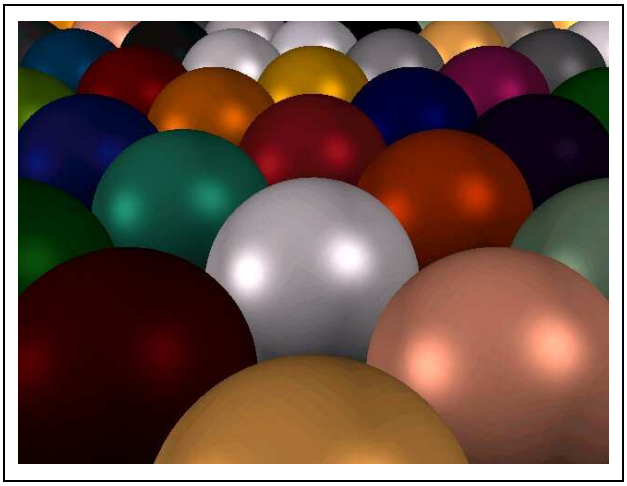

Fig. E-2: Point sampling method, 400 nm – 700 nm, 1 nm step, CIE D65 illuminant – real time version. Quantization error.

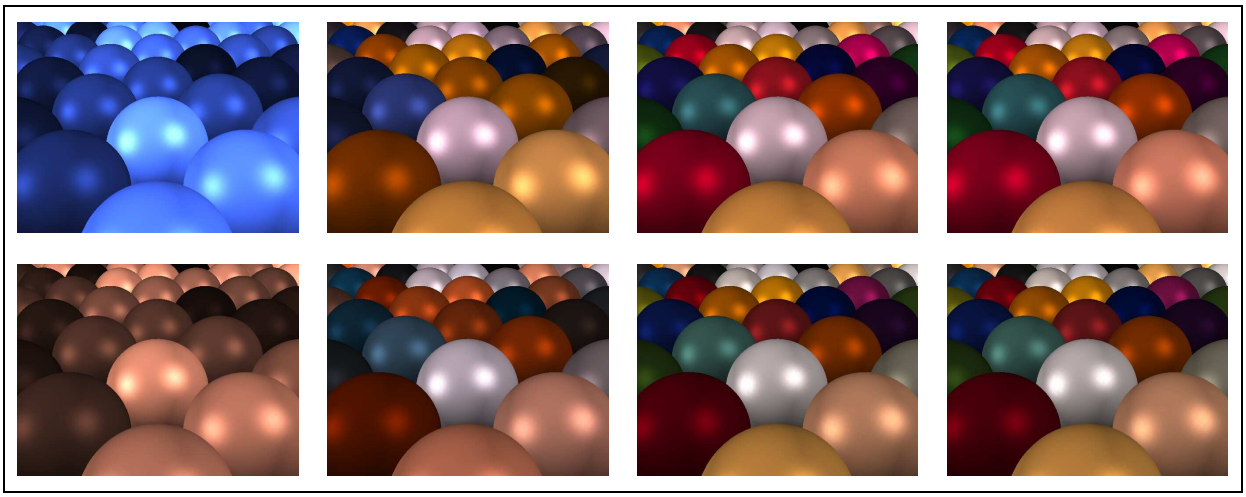

Fig. E-3: From left to right, result of using one, two, three and four basis function for linear color representation method. Scene in the upper row is illuminated by the CIE A illuminant and the scene in the lower row is illuminated by the CIE D65 illuminant.

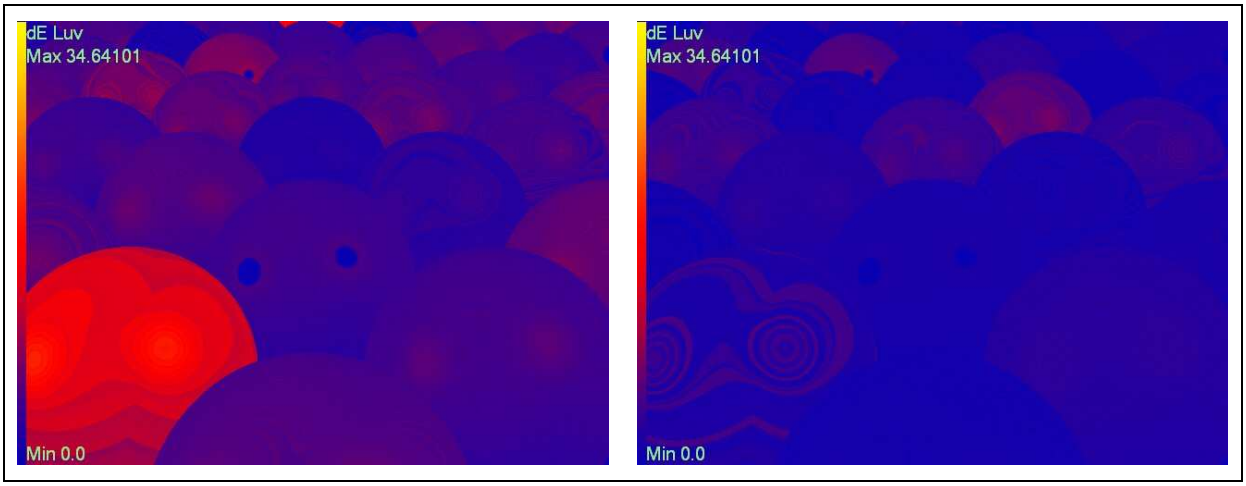

Fig. E-4: The difference of pictures computed using the linear color representation method with four basis functions and the picture computed using the point sampling method. Scenes compared in the left picture are illuminated by the CIE A illuminant and the scenes compared in the right picture are illuminated by the CIE D65 illuminant.

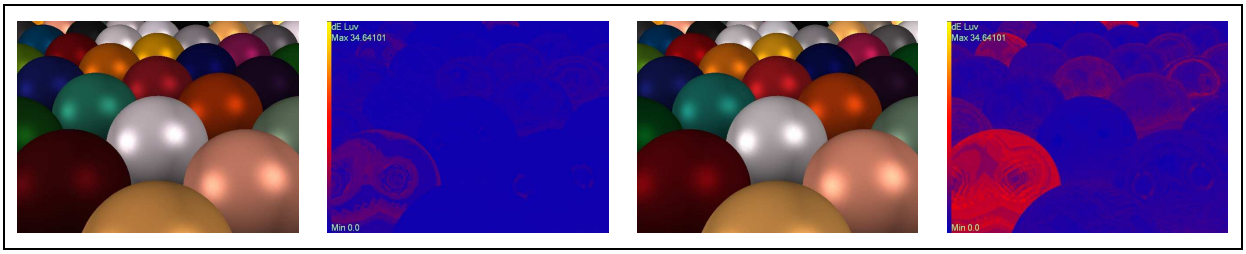

Fig. E-5: Pictures computed using the color pre-filtering method and their difference from the picture computed using the point sampling method. The scene in the first pair of pictures is illuminated by the CIE D65 illuminant and the scene in second pair of pictures is illuminated by the CIE A illuminant.

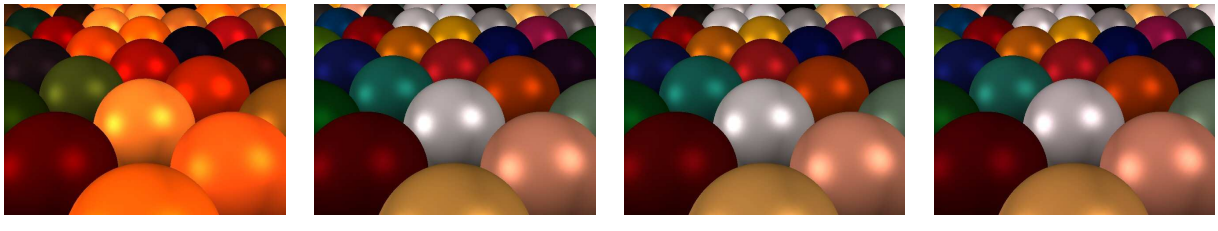

Fig. E-6: From left or right, scene illuminated by the CIE A illuminant. First picture without chromatic adaptation transformation, in the second picture is used $\mathbf{M}_{CAT2}$ transformation matrix, $\mathbf{M}_{CAT3}$ in the third and $\mathbf{M}_{CAT4}$ in the fourth picture.

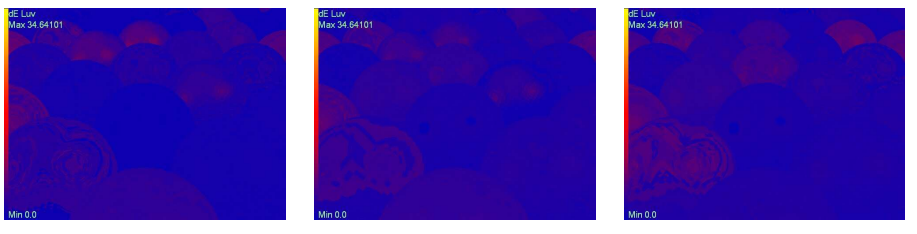

Fig. E-7: From left to right, difference of the $\mathbf{M}_{CAT2}$ and $\mathbf{M}_{CAT3}$, $\mathbf{M}_{CAT2}$ and $\mathbf{M}_{CAT4}$, $\mathbf{M}_{CAT3}$ and $\mathbf{M}_{CAT4}$.

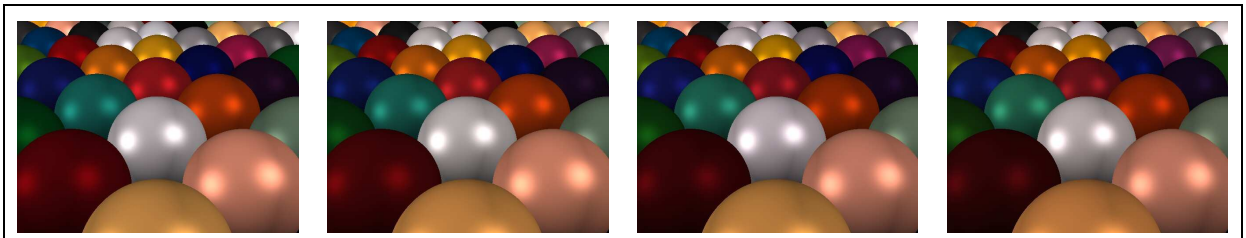

Fig. E-8: From left two right, comparison of the CIE 2° Standard observer and CIE 10° Standard observer. In the first two pictures is a scene illuminated by the CIE A illuminant. In the other pictures is a scene illuminated by the CIE D65 illuminant. The odd pictures are computed using the CIE 2° Standard observer and the even pictures are computed using the CIE 10° Standard observer.

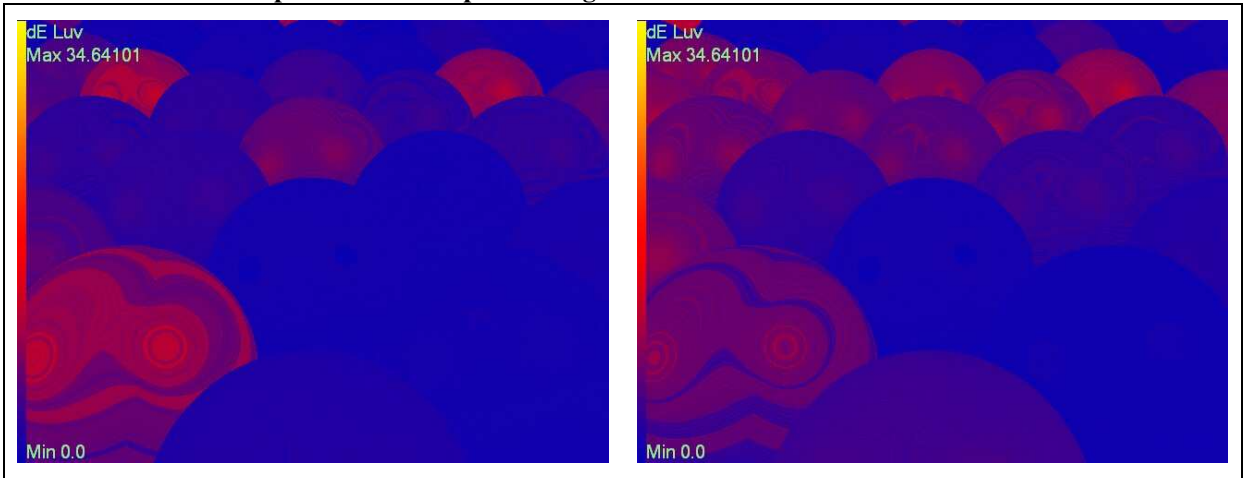

Fig. E-9: Difference of the CIE 2° Standard observer and the CIE 10° Standard observer, scenes compared in the first picture are illuminated by the CIE A illuminant and the scene compared in the second picture are illuminated by the D65 illuminant.

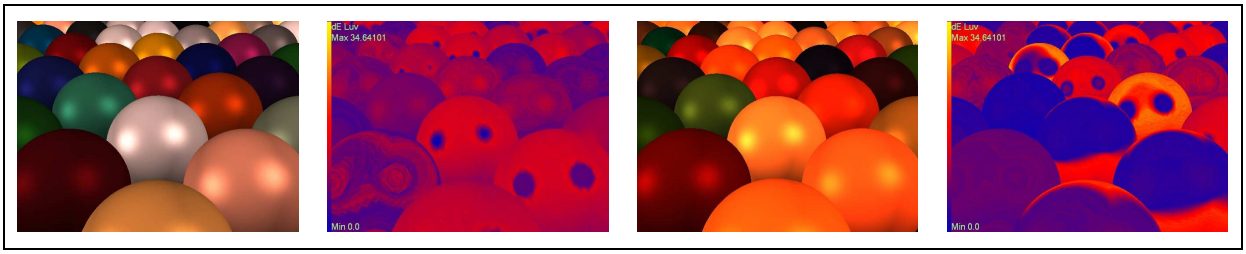

Fig. E-10: Pictures computed in standard RGB and its difference from their full spectral version. Scene in the first pair of pictures is illuminated by the CIE D65 illuminant and the scene in the second pair is illuminated by the CIE A illuminant.

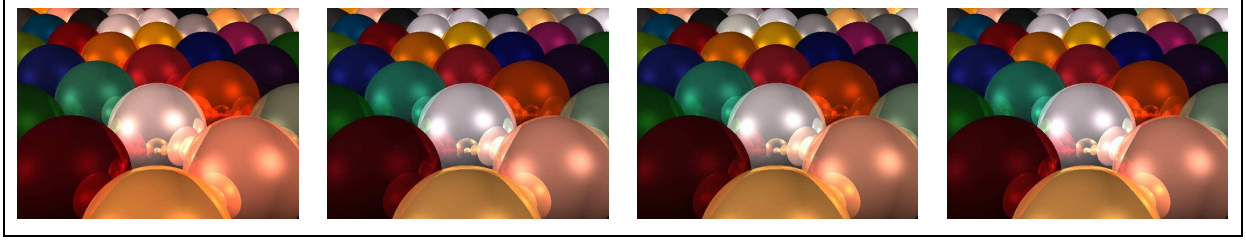

Fig. E-11: Higher order illumination component. From left to right, standard RGB, point sampling method with 301 sample points, linear color representation method with five basis functions, color pre-filtering method. Scenes are illuminated by the CIE D65 illuminant.

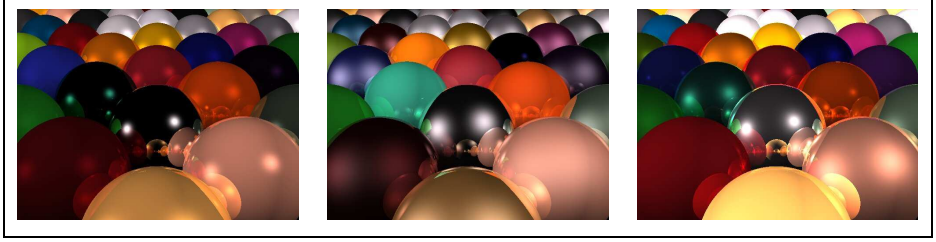

Fig. E-12: From left to right, Phong's illumination model, Strauss' illumination model and Blinn's illumination model. Scenes are illuminated by the CIE D65 illuminant and were computed using the point sampling method.

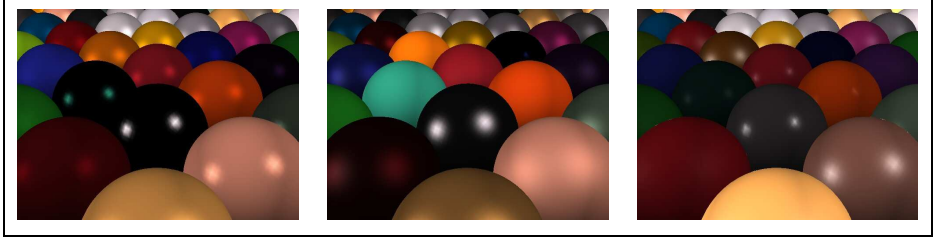

Fig. E-13: Illumination models implemented in the HLSL within the vertex shader. From left to right, Phong's illumination model, Strauss' illumination model and Blinn's illumination model. Scenes are illuminated by the CIE D65 illuminant and were computed using the color pre-filtering method.

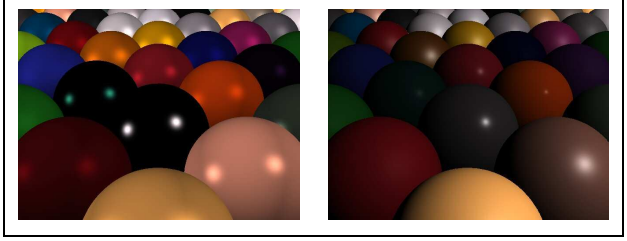

Fig. E-14: Illumination models implemented in the HLSL within the pixel shader. From left to right, Phong's illumination model, Blinn's illumination model. Scenes are illuminated by the CIE D65 illuminant and were computed using the color pre-filtering method.

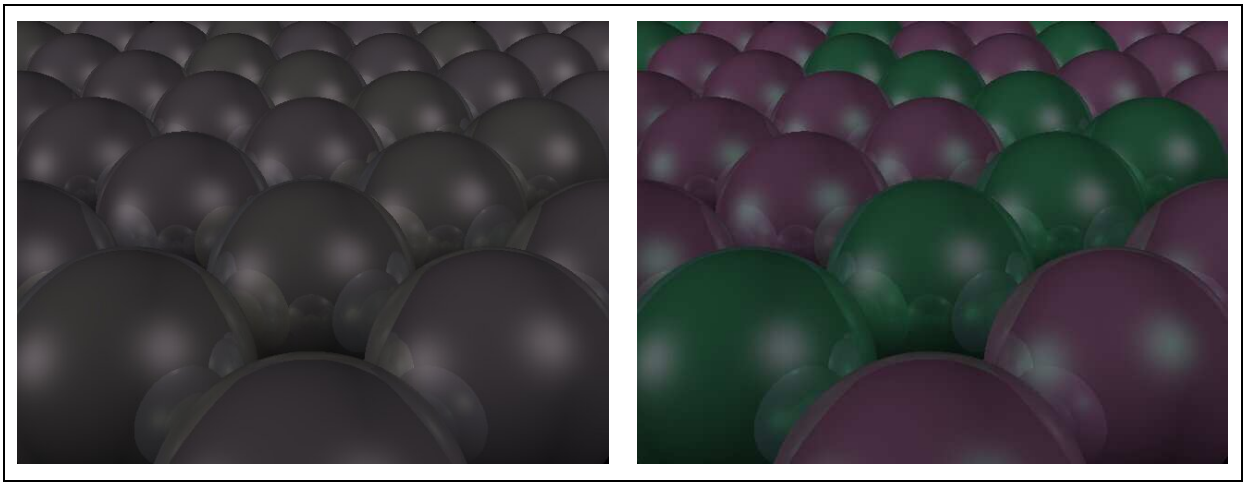

Fig. E-15: Example of metamerism. The since in the left picture is illuminated by the CIE D65 illuminant and the scene in the picture on the right is illuminated by the CIE A illuminant.