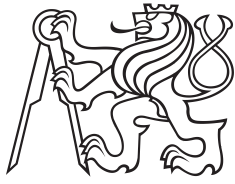


Bachelor Project



**Czech
Technical
University
in Prague**

F3

**Faculty of Electrical Engineering
Department of Computer Graphics and Interaction**

Path tracing using Vulkan API

Matvii Bunin

**Supervisor: doc. Jiří Bittner
January 2021**

Abstract

This work overviews basic techniques of ray tracing as well as the structure of Vulkan API and shows how to use them to create a path tracer. The related implementation is built from ground up and is aimed for simplicity . . .

Keywords:

Supervisor: doc. Jiří Bittner

Abstrakt

Tato práce obsahuje přehled základních technik sledování paprsků spolu s popisem struktury Vulkan API a ukazuje jak za jejich pomoci vytvořit program pro sledování paprsků. Vztahující se implementace je postavena od základu a je zaměřena na jednoduchost . . .

Klíčová slova:

Překlad názvu: Implementace sledování cest v rozhraní Vulkan

Contents

1 Introduction	1
2 Theory of ray tracing	3
2.1 Physical properties of light	3
2.2 Whitted ray tracing	6
2.3 Calculating illumination	8
2.3.1 Measuring lighting	8
2.3.2 Lambert's law	10
2.4 BSDF And The Rendering Equation	10
2.5 Defining distribution functions	11
2.5.1 Reflectance	12
2.5.2 Cook-Torrance BRDF	13
2.5.3 Physically based Phong BRDF	16
2.6 Monte Carlo integration	16
2.6.1 Importance sampling	17
2.6.2 Russian roulette	20
2.7 Path tracing	21
2.7.1 Light sources and next event estimation	24
2.8 Modelling camera	25
2.9 Other ray tracing techniques	25
2.10 Acceleration and BVH	27
3 Structure and workflow of Vulkan API	29
3.1 Memory structure	29
3.2 Rendering pipeline	30
3.2.1 Descriptor Sets	31
3.3 Command buffers	31
3.4 Synchronization	32
3.5 Ray tracing in Vulkan	33
3.5.1 Bottom and Top Level acceleration structures	33
3.5.2 Ray tracing pipeline	34
3.6 Shader binding table	35
4 Implementation	39
4.1 VulkanBase	39
4.2 RayTracerNV	42
4.3 Ray tracing shaders	44
4.4 Path tracing implementation	45
5 Results	49
Conclusion	55
Acknowledgements	55
Bibliography	57

Figures

<p>2.1 Linearly polarized light wave (a), interference pattern of monochromatic light (b) [3] 4</p> <p>2.2 ω_i, ω_r and ω_t are incoming, reflected and transmitted light directions respectively. \mathbf{n} is surface normal vector. All the vectors are normalized. 6</p> <p>2.3 Tree of rays 7</p> <p>2.4 Shadow rays 7</p> <p>2.5 The famous image produced by Whitted ray tracer (Whitted [4]). Reflection and refraction impossible to synthesize before are well visible here. 8</p> <p>2.6 The small region dA where light hits the surface is the projection of a small surface region dA^\perp orthogonal to the incoming light direction on a unit sphere. 10</p> <p>2.7 Reflections corresponding to specular, diffuse and glossy types respectively 11</p> <p>2.8 Local and subsurface scattering (Real-Time Rendering [3]) 12</p> <p>2.9 The pictures are synthesized using Phong BRDF, which does not consider Fresnel effect, while the bottom pictures are real photographs. (Kavita Bala, Cornell University) . 13</p> <p>2.10 Micro vs. macrostructure (Walter [6]) 14</p> <p>2.11 Total viewed surface of microfacets is equal to viewed differential area (Physically Based Rendering [2]) 15</p> <p>2.12 Shadow masking (Walter [6]) . . 15</p> <p>2.13 Cook-Torrance model compared to the ground truth data (Ngan [10]) 16</p> <p>2.14 Scene rendered with 100 samples, without importance sampling (top) and with importance sampling (bottom). 20</p>	<p>2.15 Pinhole camera described by near and far plane distance (near plane being the screen), vectors \vec{u}_p, \vec{side} and direction \mathbf{d}, as well as aspect ratio and FOVy angle, here presented as $\theta = \frac{FOVy}{2}$. w and h here represent the size of screen in pixels. 25</p> <p>2.16 Light tracing and ray tracing steps. Dots represent the points at which values are stored. It can be seen that some stored values may remain unused. 26</p> <p>2.17 Choosing volumes subdivision axis [2] 27</p> <p>2.18 Ray tracing on RT cores [13] . . 28</p> <p>2.19 Software emulated ray tracing [13] 28</p> <p>3.1 Memory hierarchy [13] 30</p> <p>3.2 Vulkan rendering pipeline [12] . . 30</p> <p>3.3 Command buffer life cycle [12] . . 32</p> <p>3.4 Acceleration structures hierarchy [14] 34</p> <p>3.5 Ray tracing pipeline [14] 35</p> <p>3.6 Shader binding table in DXR [1] 36</p> <p>4.1 .rgen shader pseudocode (1) 46</p> <p>4.2 .rgen shader pseudocode (2) 47</p>
--	---

Tables

5.1 Scene with two direct light sources rendered for different numbers of samples S and fixed depth $D=10$. .	50
5.2 Scene with two direct light sources rendered for different recursion depth D with number of samples $S=100$.	51
5.3 Test results for scenes of varying complexity	52



Chapter 1

Introduction

Physically based rendering is the most common approach for creating realistic images of 3D scenes implemented in a number of GPU and CPU-based renderers such as Blender Eevee, Octane, or Corona. Unlike empirical approaches, PBR aims to achieve photo realistic results by modeling physical behavior of light, which also involves modeling physical properties of various types of surfaces influencing how they interact with light. The resulting approach is consistent and capable of producing images close or indistinguishable from real photos. The reason why it was widely adopted only in the last 10 years is its computational heaviness and hence its dependence on hardware acceleration.

Light propagation in PBR is commonly simulated by the path tracing algorithm, which recursively calculates how each light ray bounces off object surfaces in the scene. Since all the light rays are independent and behavior of each light ray follows the same rules, all rays can be computed in parallel on different GPU cores via compute shader. Other ways to speedup ray tracing include storing vertex data in accelerating structures such as Bounding Volume Hierarchy (BVH) which is the part of NVIDIA RTX platform. Despite the optimizations path tracing does not run in real time on regular hardware.

In year 2018 NVIDIA released new generation of graphic cards GeForce RTX with dedicated ray tracing cores, which provide hardware acceleration for BVH and sped up ray tracing to real time performance. This makes RTX platform the technology that expands the application field of ray tracing into the game industry, and it is already supported in major games such as Fortnite, Minecraft, Cyberpunk 2077, and many others.

This work aims to demonstrate the framework capabilities via implementing basic path tracing. Features of the RTX platform can be accessed through in OptiX, DirectX and Vulkan APIs. For this work Vulkan API is chosen for advantages of being fast, cross-platform and leaving a lot of control for the user at cost of somewhat bigger complexity compared to other APIs.



Chapter 2

Theory of ray tracing

Rendering techniques have always progressed towards increasing realism, from empirical local illumination techniques to modern ray tracing, each of them trying to simulate the same phenomenon of visible light interacting with different media. Even though all the existing systems due to computational limits omit a lot of complexity of physical light behavior, it is still important to understand for seeing their limitations.

2.1 Physical properties of light

Light propagating through space can be best represented as electromagnetic waves 2.1(a), which are described by direction of propagation, speed, wavelength, amplitude and polarization. The latter is never simulated in rendering applications since it would be complicated and would not attribute to visual quality. Wavelength influences the light color, which during the simulation is represented with RGB vector. This also handles the common case of light consisting of multiple waves with different wavelengths.

Both electric and magnetic component of the light wave have the same wavelength and amplitude. Together, they contribute to average light energy per time and surface unit called *irradiance*, denoted E . This can be written as $E = ka^2$ where a is the amplitude and k is a constant factor. The electrical component of light interacts with molecules which absorb and re-emit the light mostly along the direction where it comes from, which is called *light scattering*. In simulation, light components are assumed to interact independently, which is in most cases physically correct.

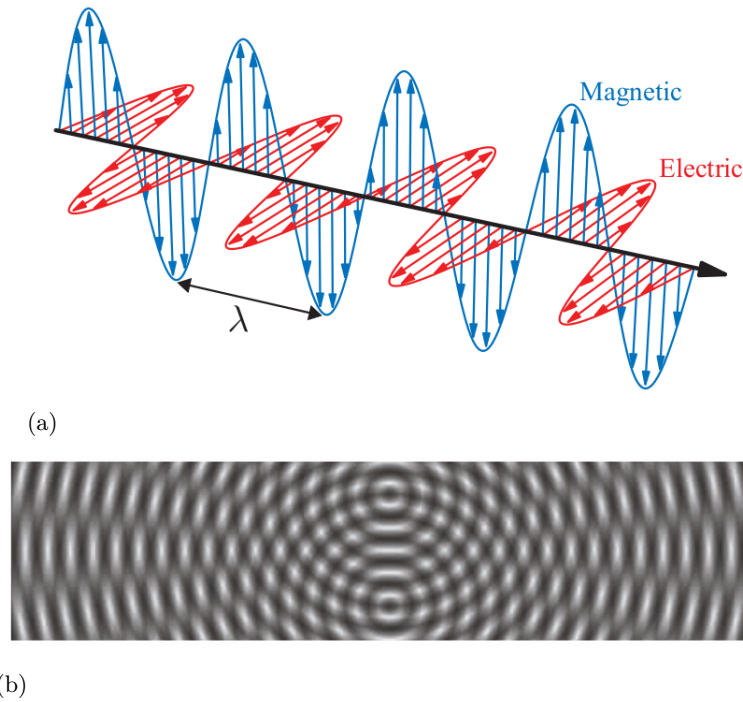


Figure 2.1: Linearly polarized light wave (a), interference pattern of monochromatic light (b) [3]

Combined together multiple waves of monochromatic light may amplify or cancel out depending on their phase offset, which is called waves *interference*. For two interfering waves, the resulting irradiance is changing quadratically $E = k(a_1p_1 + a_2p_2)^2$, where p_1 and p_2 are phase factors $\in \langle -1, 1 \rangle$. When the phase offset is constant for each point at some plane, a wave pattern is formed 2.1(b). Although this phenomenon also underlies the way light behaves in different media, it is usually ignored in rendering applications.

Abstracting away from wave particle interactions, the environment can be described with two parameters: *attenuation index* κ , which describes how much light is absorbed by the environment and *refractive index* η , describing phase velocity of light in the environment relative to vacuum (*vacuum* = 1, *air* = 1.0003, *water* = 1.33, *glass* = 1.6). When light hits a surface between two environments, part of the waves is reflected and part is transmitted into the other environment. The direction of the transmitted light is affected by light speed change and is defined by Snell's law. In metallic materials, all the transmitted light is immediately absorbed by free electrons, so no transmission takes place. The wave form of the reflected light is influenced by microscopic surface structures in the order of hundreds of wavelengths. Simpler models ignore structures that are bigger than a wavelength, while more complicated approaches like Cook-Torrance (2.5.2) to model them.

Summarizing all the simplifications, simulating light itself in a ray tracing application requires only the ray along which the wave is propagated and intensities of the three light components. The simulation will then follow the

rules of geometric optics:

- Light propagating through space is described by its flow and a corresponding wave length.
- Light travels along straight lines in the direction from the light source (*light rays*).
- Light behavior is *reciprocal*, meaning that laws of physics are the same for a light ray traveling in the opposite direction.
- Light rays do not interact with each other.
- Light rays obey laws of reflection and refraction.

The rule of reflection states that the angle of the incoming ray θ_i and the ray of ideal mirror reflection θ_r (fig. 2.2) are equal, resulting in equation 2.1 for reflected ray for $\omega_i \cdot \mathbf{n} < 0$.

$$\omega_r = \omega_i - 2\mathbf{n}(\omega_i \cdot \mathbf{n}) \quad (2.1)$$

The rule of refraction also known as Snell's law states that for angles of incoming and refracted ray holds $\eta_i \sin \theta_i = \eta_t \sin \theta_t$. The refracted ray ω_t can then be computed as 2.2, assuming $\theta_t \in (0, \frac{\pi}{2}]$ ¹.

$$\begin{aligned} \mathbf{k} &= \omega_i + \mathbf{n} \cos \theta_i \quad // \text{ projection of } \omega_i \text{ to the surface} \\ \cos \theta_i &= -\omega_i \cdot \mathbf{n} \\ \sin^2 \theta_t &= \left(\frac{\eta_i}{\eta_t}\right)^2 (1 - \cos^2 \theta_i) \\ \cos \theta_t &= \sqrt{1 - \sin^2 \theta_t} \\ \omega_t &= \frac{\mathbf{k}}{\|\mathbf{k}\|} \sin \theta_t - \mathbf{n} \cos \theta_t \\ &= \mathbf{k} \frac{\sin \theta_t}{\sin \theta_i} - \mathbf{n} \cos \theta_t \\ &= \omega_i \frac{\eta_i}{\eta_t} - \mathbf{n} (\cos \theta_i \frac{\eta_i}{\eta_t} + \cos \theta_t) \end{aligned} \quad (2.2)$$

¹ $\theta_t = \frac{\pi}{2}$ is called *critical angle*. Refraction angles above critical are handled separately as a phenomenon of *total internal reflection*.

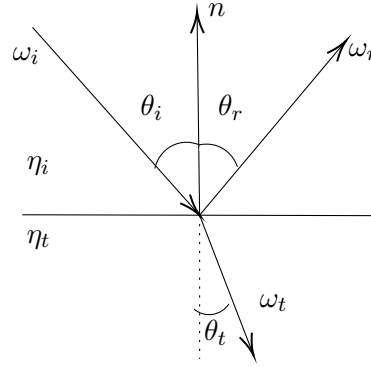


Figure 2.2: ω_i , ω_r and ω_t are incoming, reflected and transmitted light directions respectively. \mathbf{n} is surface normal vector. All the vectors are normalized.

2.2 Whitted ray tracing

Use of geometric optics for image synthesis was first suggested by Turner Whitted [4] in 1980. Before him, the Phong illumination model 2.3 was used.

$$I = I_a + k_d \sum_i^l L_{i,d}(\mathbf{n} \cdot \boldsymbol{\omega}_i) + k_s \sum_i^l L_{i,s}(\mathbf{n} \cdot \mathbf{v})^n \quad (2.3)$$

Where:

\mathbf{v} is the viewer direction ².

I_a is intensity of ambient component.

k_d and k_s are diffuse and specular coefficients respectively, making the surface look more or less glossy.

$L_{i,d}$ and $L_{i,s}$ are diffuse and specular intensities of i -th light source, respectively.

n is reflection sharpness.

l is the number of light sources.

This model works fast and produces good results for diffuse reflections, but it does not consider the case of surfaces that reflect light acting as light sources. The alternative offered by Whitted 2.4 replaces the specular component by the intensity S of light coming from the direction of specular reflection. Reflection sharpness n in the original paper is replaced by adjusting random perturbations of the \mathbf{n} vector. The model also counts with transmitted light intensity by adding T term weighted by k_t coefficient of transmitted light contribution.

$$I = I_a + k_d \sum_i^l L_{i,d}(\mathbf{n} \cdot \boldsymbol{\omega}_i) + k_s S + k_t T \quad (2.4)$$

²Often was replaced with halfway vector for i -th light source $\mathbf{h}_i = (\mathbf{n} + \boldsymbol{\omega}_i) / \|\mathbf{n} + \boldsymbol{\omega}_i\|$.

S and T intensities depend on previous light hits and hence must be calculated recursively. The approach used in Whitted ray tracing, suggested by Appel[5] is to trace rays backwards from camera to light sources, exploiting the aforementioned rule of reciprocity. Each time a ray coming from the camera hits a surface it creates two incoming rays that are traced recursively creating a tree of incoming rays (fig. 2.3). So that the process is not running infinitely, the recursion is stopped at some constant depth and in the last hit only the diffuse component is calculated.

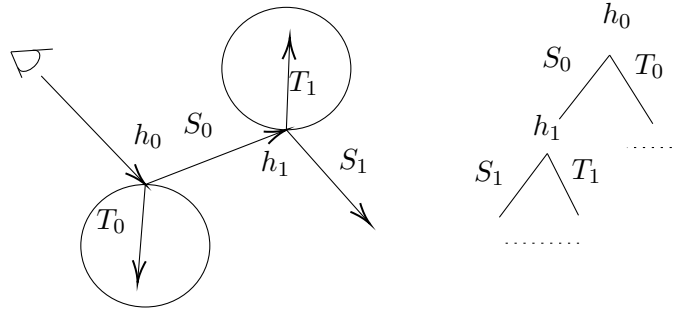


Figure 2.3: Tree of rays

At each hit point h_j light intensity is calculated using the equation 2.4. Note that for calculating hits that will be produced by rays T_1 and T_2 inside transparent objects normal vectors will have to be flipped.

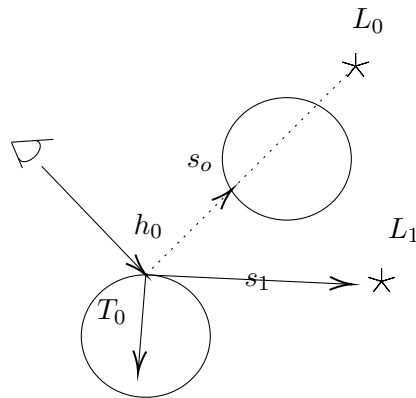


Figure 2.4: Shadow rays

Apart from reflections and refractions the model also tests visibility of each light source attributing to the diffuse term. It does so by casting shadow rays s_i (fig. 2.4) from the hit point in the direction of each light source. If a geometry hit occurs as for s_0 ray, then the light source L_0 is considered in shadow and its contribution is attenuated, e.g. as $L'_{0,d} = L_{0,d} \cdot c$, $c \ll 1$.

Whitted ray tracing produces much better results than previous techniques simulating reflections, transparency and color bleeding (one surface influencing the color of another), but does so at much bigger computational cost. Tracing all the rays takes time and the optimizations such as clipping, back face

culling, and depth tests for visibility can no longer be used because all objects in the scene can now influence each other's appearance.

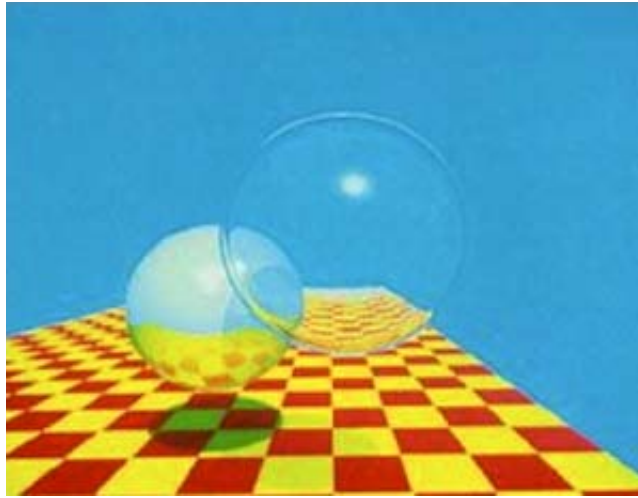


Figure 2.5: The famous image produced by Whitted ray tracer (Whitted [4]). Reflection and refraction impossible to synthesize before are well visible here.

2.3 Calculating illumination

Just as with light propagation, when calculating lighting at some point the following assumptions are made:

- Light components are independent (no fluorescence or phosphorescence takes place).
- Light system is at equilibrium and is not changing over time.
- Incident light from multiple sources is always calculated as a vector sum of influences of each light source.
- Energy conservation law is preserved, meaning that the scattered light never has more energy than the incoming light.

To describe and evaluate the contribution of each light component at some point, some further notions are required.

2.3.1 Measuring lighting

Multiple physical quantities exist for measuring the light influence, all being derived from light energy (2.5) expressed as a function of wavelength (λ) and using constants of speed of light in an environment (c) and Plank's constant $h = 6.626$.

$$Q = \frac{hc}{\lambda} \quad [J] \quad (2.5)$$

Other measures represent the distribution of light energy over different quantities:

- Radiant flux (Φ) describes the light energy distribution over time.

$$\begin{aligned}\Phi &= \frac{dQ}{dt} \quad \left[\frac{J}{s} = W \right] \\ Q &= \int_{t_0}^{t_1} \Phi(t) dt\end{aligned}\tag{2.6}$$

- Irradiance (E) and radiant exitance describe the radiant flux distribution over an area for incoming and exiting light respectively. Irradiance at some point p is calculated as a limit of differential flux over a differential area at the point. The total flux is then expressed as an integral over the area. The value of irradiance is the one used as value of light components in ray tracing applications.

$$\begin{aligned}E(p) &= \frac{d\Phi(p)}{dA} \quad \left[\frac{W}{m^2} \right] \\ \Phi &= \int_A E(p) dA\end{aligned}\tag{2.7}$$

- Intensity (I) describes the distribution of light over different directions. It can be used e.g. to describe point light sources. The directions are described as points on a unit sphere via solid angles³ denoted ω . The total flux is obtained by integrating over the set of directions Ω . The differential solid angle is defined as in eq. 2.8 using polar coordinates.

$$\begin{aligned}d\omega &= \sin \theta d\phi d\theta \\ \int_0^\pi \int_0^{2\pi} \sin \theta d\phi d\theta &= 4\pi\end{aligned}\tag{2.8}$$

$$\begin{aligned}I &= \frac{d\Phi}{d\omega} \quad \left[\frac{W}{sr} \right] \\ \Phi &= \int_\Omega I(\omega) d\omega\end{aligned}\tag{2.9}$$

- Radiance (L) is the most general quantity for describing the light energy with respect both to directions and to area.

$$L(p, \omega) = \frac{dE_\omega(p)}{d\omega} = \frac{d\Phi}{d\omega dA^\perp}\tag{2.10}$$

In the equation 2.10 radiance is calculated with respect to a surface perpendicular to the light direction, the incident light energy for an arbitrary surface is calculated using the Lambert's law.

³A notion to describe a 3D angle. Just as a unit circle has an arc angle of 2π radians, a unit sphere has a surface angle of 4π steradians (sr).

2.3.2 Lambert's law

From fig. 2.6 we see that $dA = dA^\perp \cos \theta$. Light falling to the surface at an angle θ with radiance $L(p, \omega)$ would therefore produce irradiance expressed as follows (using eq. 2.10 and 2.8).

$$E(p) = \int_{\Omega} L(p, \omega) \cos \theta d\omega = \int_0^{\frac{\pi}{2}} \int_0^{2\pi} L(p, \omega) \cos \theta \sin \theta d\varphi d\theta \quad (2.11)$$

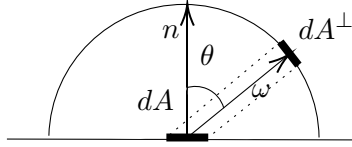


Figure 2.6: The small region dA where light hits the surface is the projection of a small surface region dA^\perp orthogonal to the incoming light direction on a unit sphere.

2.4 BSDF And The Rendering Equation

To compute an image of an object, we need to know how much light is leaving the surface in any particular direction. This notion is formally described using *bidirectional reflectance distribution function* (BRDF).

The main principle, used in the BRDF definition, is that the total amount of irradiance incident from a particular direction is getting distributed over different reflection directions. Put otherwise, the differential reflected radiance $dL_o(p, \omega_o)$ in the direction ω_o is a fraction of the differential irradiance $dE(p, \omega_i)$ coming from the direction ω_i . BRDF $f_r(p, \omega_o, \omega_i)$ is the function defining the proportionality coefficient for all incoming and outgoing directions.

$$f_r(p, \omega_o, \omega_i) = \frac{dL_o(p, \omega_o)}{dE(p, \omega_i)} = \frac{dL_o(p, \omega_o)}{L_i(p, \omega_i) \cos \theta_i d\omega_i} \quad (2.12)$$

To define the distribution of the transmitted light, the *bidirectional transmittance distribution function* is being defined analogously. The total light distribution as a combination of BRDF and BTDF is called BSDF for *bidirectional scattering distribution function*. Using the BSDF function $f(p, \omega_o, \omega_i)$ we can define the equation for radiance leaving the surface in any direction considering all the incoming directions known as *scattering equation*.

$$L_o(p, \omega_o) = \int_{S^2} f(p, \omega_o, \omega_i) L_i(p, \omega_i) |\cos \theta_i| d\omega_i \quad (2.13)$$

For a BSDF to fulfil the property of energy conservation, the following restriction should apply.

$$\int_{S^2} f(p, \omega_o, \omega_i) |\cos \theta_i| d\omega_i \leq 1 \quad (2.14)$$

Also, to fulfil the property of reciprocity, $f(p, \omega_i, \omega_o) = f(p, \omega_o, \omega_i)$ should hold for any BxDF.

When combined with light emitted by the surface in the direction ω_o , the scattering equation turns into a basic equation for radiance leaving the surface in any direction, known as the *rendering equation*.

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{S^2} f(p, \omega_o, \omega_i) L_i(p, \omega_i) |\cos \theta_i| d\omega_i \quad (2.15)$$

The absolute value around the cosine term here is a substitute of a normal flip for the transmitted light.

2.5 Defining distribution functions

Physically based BRDFs are used to capture the phenomenon of light being scattered differently by various kinds of surfaces with respect to their microscopic structure. The simplest case of reflection from a surface is an ideal mirror or pure *specular* reflection, which is characteristic for very smooth surfaces. On the other side there is pure *diffuse* reflection which scatters light equally in all directions and happens on rough surfaces e.g. brick or cotton fabric. The intermediate kind of reflection are *glossy* reflections that scatter the light around the ideal specular direction and emerge on surfaces like porcelain or car paint.

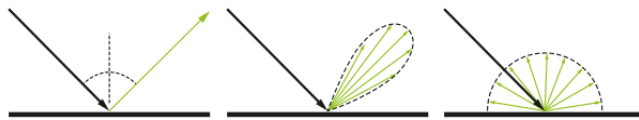


Figure 2.7: Reflections corresponding to specular, diffuse and glossy types respectively

Note that these types of BRDF are based on an assumption that the reflected ray starts at the same point where the incident ray hits a surface. This is not the case though for materials like human skin, where a significant amount of subsurface scattering takes place. Simulating such effects would require adding another term to the scattering equation and would make computations much more complex.

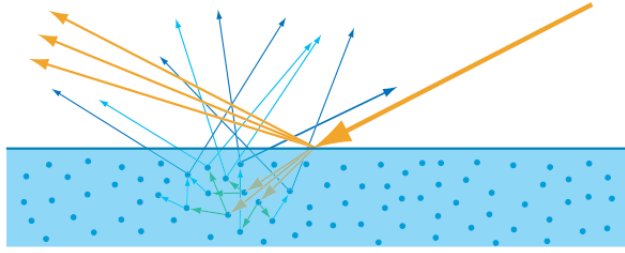


Figure 2.8: Local and subsurface scattering (Real-Time Rendering [3])

2.5.1 Reflectance

Defining BRDF functions typically requires a value called *reflectance*, describing a ratio of light energy being reflected to the light energy being absorbed by the environment. This value depends on indices of refraction, and its behavior for conductor and dielectric environments is strictly distinguished.

- Dielectrics have real indices of refraction, can transmit incoming light and some subsurface scattering may take place.
- Conductors have complex indices of refraction in form $\eta + ik$. Light energy falling on their surface gets rapidly absorbed by free electrons and never gets transmitted.

Another factor influencing the reflectance coefficient is light polarization. Assuming that light is unpolarized we can compute reflectance using Fresnel equation.

$$F(\omega) = \frac{1}{2}(r_p^2 + r_s^2) \quad (2.16)$$

where r_p and r_s represent Fresnel reflectance for light with parallel and perpendicular polarization. The reflectance values for dielectrics can be computed as follows.

$$\begin{aligned} r_p &= \frac{\eta_t \cos \theta_i - \eta_i \cos \theta_t}{\eta_t \cos \theta_i + \eta_i \cos \theta_t} \\ r_s &= \frac{\eta_t \cos \theta_i + \eta_i \cos \theta_t}{\eta_t \cos \theta_i - \eta_i \cos \theta_t} \end{aligned} \quad (2.17)$$

These formulas are largely simplified due to real indices of refraction. In case of light hitting a surface of a metal, the formulas for Fresnel reflectance are much more complex and computationally heavy. To simplify them, some good approximations are possible. The following simplified Fresnel coefficient for metals is suggested by Lazanyi [9].

$$F(\omega) = \frac{(\eta - 1)^2 + 4\eta(1 - \cos \theta)^5 + k^2}{(\eta + 1)^2 + k^2} \quad (2.18)$$

where η and k come from $\eta_t = \eta + ik$ and θ is the angle of the incident light direction with halfway vector between light and viewer direction. The reason



Figure 2.9: The pictures are synthesized using Phong BRDF, which does not consider Fresnel effect, while the bottom pictures are real photographs. (Kavita Bala, Cornell University)

for using halfway vector instead of normal comes from microfacet theory. The simplification relies on the fact that light comes from an environment with $\eta_i = 1$ e.g. air or vacuum.

In case when the incident light angle is above critical, the Fresnel formulas are not applicable and the reflectance coefficient can be considered equal to one. Using of Fresnel coefficient is used to capture Fresnel effect when reflections get more specular at near-grazing angles (2.9).

■ 2.5.2 Cook-Torrance BRDF

Distribution function is always defined as a sum of glossy (specular) and diffuse reflection. For physically based rendering, the combination of Cook-Torrance model for specular and Lambertian model for diffuse component is used (2.19).

$$f(p, \omega_o, \omega_i) = k_d f_{rd}(p, \omega_o, \omega_i) + k_s f_{rs}(p, \omega_o, \omega_i) \quad (2.19)$$

Here k_d and k_s are mixture coefficients for diffuse and specular components respectively for which the restriction $k_d + k_s \leq 1$ should hold, otherwise the amount of emitted light may exceed the amount of light coming in, breaking the law of energy conservation.

Lambert diffuse distribution is defined as an even distribution of the surface diffuse color $f_{rd} = \frac{C_d}{\pi}$.

Cook-Torrance reflectance model (R.Cook and K.Torrance, 1982) is a more complex function that was derived based on microfacet theory of surface structure (2.20).

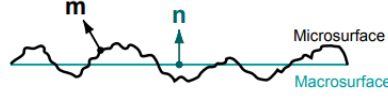


Figure 2.10: Micro vs. macrostructure (Walter [6])

$$f_{rs}(p, \omega_o, \omega_i) = \frac{D(\omega_m) \cdot G(\omega_o, \omega_i) \cdot F(\omega_i)}{4 \cos \theta_i \cos \theta_o} \quad (2.20)$$

Here the nominator consists of the distribution function D , geometry function G and Fresnel function F . Here ω_m is a half angle between ω_i and ω_o . While the D function is the distribution of light reflected directly, G is the distribution of light bouncing of microfacets and losing energy. Both are dependent on same surface parameters.

- According to the microfacet theory, any surface consists of microscopic ideal mirrors (micro facets) facing random directions (2.10). *Microfacet distribution function* is a probability distribution $D(\mathbf{m})$ of microfacet normal \mathbf{m} (2.10) facing any particular direction. It is most often defined by Backmann, Phong or GGX functions. To preserve the law of energy conservation, any microfacet distribution function should fulfill

$$\int_{H^2} D(\omega_m) \cos \theta_m d\omega_m = 1 \quad (2.21)$$

Backmann distribution for isotropic surfaces assuming $\mathbf{m} \cdot \mathbf{n} > 0$ is defined as

$$D(\mathbf{m}) = \frac{\exp\left(-\frac{\tan^2 \theta_m}{\alpha^2}\right)}{\pi \alpha^2 \cos^4 \theta_m} \quad (2.22)$$

where α is a distribution parameter set depending on slope of the microfacets.

- *Geometry or masking-shadowing function* accounts for light reflected from microfacets being occluded by other microfacets (2.12). Forward facing microfacets are visible from a viewing direction if they are not shadowed by back facing microfacets. The masking-shadowing function $G_I(\omega)$ can be then defined as a ratio of visible microfacet area to the total forward facing microfacet area in the direction ω . For the areas of forward and backward facing microfacets denoted $A^+(\omega)$ and $A^-(\omega)$ the visible microfacet area can be calculated as $A^+(\omega) - A^-(\omega)$, leading to the masking-shadowing function

$$G_I(\omega) = \frac{A^+(\omega) - A^-(\omega)}{A^+(\omega)} \quad (2.23)$$

usually written using the auxiliary function $\Lambda(\omega)$

$$G_I(\omega) = \frac{1}{1 + \Lambda(\omega)} \quad \Lambda(\omega) = \frac{A^-(\omega)}{A^+(\omega) - A^-(\omega)} \quad (2.24)$$

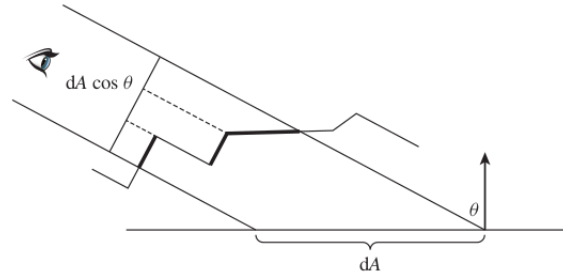


Figure 2.11: Total viewed surface of microfacets is equal to viewed differential area (Physically Based Rendering [2])

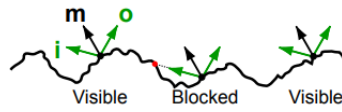


Figure 2.12: Shadow masking (Walter [6])

As can be seen from figure 2.11, the portion of microfacet area directly visible from a direction ω equals to $\cos \theta = \omega \cdot \mathbf{n}$, imposing the restriction of

$$A^+(\omega) - A^-(\omega) = \cos \theta \quad (2.25)$$

Assuming that heights microfacets close to each other do not correlate (which does not always apply in reality), the $\Lambda(\omega)$ function can be derived for a chosen $D(\omega)$. For Beckmann distribution it takes the form of

$$\Lambda(\omega) = \frac{1}{2} \left(\text{erf}(a) - 1 + \frac{e^{-a^2}}{a\sqrt{\pi}} \right)$$

where:

$$a = \frac{1}{\alpha \tan \theta} \quad (2.26)$$

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-y^2} dy$$

In order for light to be reflected from a microfacet both incident and outgoing directions should be visible. A function $G(\omega_o, \omega_i)$ representing this distribution could be defined as $G_I(\omega_i)G_I(\omega_o)$. Because the probabilities of visibility are dependent, a similar but better approximation is used:

$$G(\omega_o, \omega_i) = \frac{1}{1 + \Lambda(\omega_o) + \Lambda(\omega_i)} \quad (2.27)$$

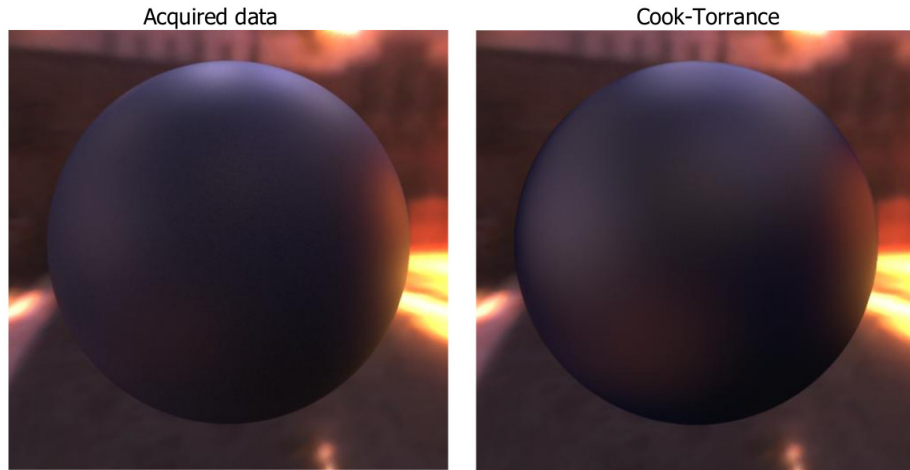


Figure 2.13: Cook-Torrance model compared to the ground truth data (Ngan [10])

2.5.3 Physically based Phong BRDF

The Cook-Torrance model yields results that are very close to reality (2.13), but is quite complex to implement. In this work, we implement a simpler model, being a modification of Phong illumination model for physically based rendering proposed by Lafortune [7]. It differs from the Cook-Torrance model by a simplified specular term reduced to the $D(\omega)$ function defined as Phong distribution.

$$f_r(p, \omega_0, \omega_i) = k_d \frac{1}{\pi} + k_s \frac{n+2}{2\pi} \cos^n \alpha \quad (2.28)$$

where α is an angle between the reflected ray and ideal specular reflection and n is specular exponent. The restriction of $k_s + k_d \leq 1$ is required to preserve the energy conservation law.

2.6 Monte Carlo integration

The most effective way to compute radiometric integrals is using the probability based Monte Carlo method, which estimates the integral of a function as an average of the function values at randomly chosen points. For N samples drawn from a random distribution p , the integral estimation can be computed as

$$F_N = \frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p(X_i)} \quad (2.29)$$

where it holds that $f(x) \neq 0 \Rightarrow p(x) \neq 0$. This integration technique is universal, does not require continuity and does not depend on an integral dimension, but it comes with a downside of slow convergence rate of $O(\sqrt{N})$. In rendering applications it manifests itself as a noise taking large amount of samples to get rid of.

Various Monte Carlo estimators can be built depending on how the samples are drawn. The main properties used for evaluating the Monte Carlo estimator are its *bias*, *variance* and *efficiency*.

- Bias of an estimator is defined as $E[\bar{I}] - I$. In the simplest case, unbiased Monte Carlo estimators are used, so they approach the actual value of the integral at infinite number of samples.
- The variance of an estimator can be computed as

$$V(\bar{I}) = \frac{1}{N^2} \sum_{i=1}^N V\left(\frac{f(X_i)}{p(X_i)}\right) = \frac{1}{N} V\left(\frac{f(X_i)}{p(X_i)}\right) \quad (2.30)$$

which means that the error of an unbiased estimator can be reduced by increasing number of samples and the closest is the $\frac{f(X_i)}{p(X_i)}$ value to a constant, the fewer samples it will take.

- The efficiency of an estimator is defined as $\epsilon(\bar{I}) = \frac{1}{V(\bar{I})T(\bar{I})}$ where $T(\bar{I})$ is the time it took to produce the variance.

Depending on the goals and type of function, various sampling techniques are used to optimize for these parameters.

■ 2.6.1 Importance sampling

The importance sampling techniques can be used to reduce the variance by assigning higher probability to samples that contribute to the result the most. Rather than using a uniform distribution, we choose some p close to p^* so that $f(x) = p^*(x)c$ where $c = \int_D f(x)dx$. Then we need to compute the CDF $F(x) = \int_{-\infty}^x p(x)dx$ and invert it, so that we can draw samples using inversion method $X = F^{-1}(u)$ where u is a variable generated from a uniform distribution.

From the equation 2.30 we can see that for $p = p^*$ the method would produce 0 variance, but computing such p is an equivalent to solving the problem. One way of computing the approximation for p is the *metropolis light transport* algorithm and the other is using some analytical approximation, as we will show for the physically based Phong model.

■ Sampling Phong BRDF

The method for sampling the BRDF is introduced by Lafortune [7]. The paper suggests rewriting the integral into the new form, separating parts of the specular and diffuse BRDFs that integrate to 1:

$$\begin{aligned} L_o(p, \omega_o) &= \int_{H^2} \left(k_d \frac{1}{\pi} + k_s \frac{n+2}{2\pi} \cos^n \alpha\right) L_i(p, \omega_i) \cos \theta_i d\omega_i = \\ &= k_d \int_{H^2} \left(\frac{1}{\pi} \cos \theta_i\right) L_i(p, \omega_i) d\omega_i \\ &+ k_s \int_{H^2} \left(\frac{n+1}{2\pi} \cos^n \alpha\right) \left(\frac{n+2}{n+1} L_i(p, \omega_i) \cos \theta_i\right) d\omega_i \end{aligned} \quad (2.31)$$

To determine the type of the reflection to sample, another uniform random variable u is used, and the appropriate component is chosen as follows:

- for $0 \leq u < k_d$ sample diffuse component;
- for $k_d \leq u < k_d + k_s$ sample specular component;
- for $k_d + k_s \leq u$ sample nothing;

For each type of reflection, we now have the p so that $f(x) = p(x)c$ function, where the constant factor c is computed by the ray tracing algorithm and used as sample value:

Diffuse component:

$$\begin{aligned} p_d &= \frac{1}{\pi} \cos \theta_i \\ c_d &= L_i(p, \omega_i) \frac{k_d}{\max(k_d)} \end{aligned} \quad (2.32)$$

Specular component:

$$\begin{aligned} p_s &= \frac{n+1}{2\pi} \cos^n \alpha \\ c_s &= L_i(p, \omega_i) \frac{n+2}{n+1} \cos \theta_i \end{aligned} \quad (2.33)$$

Evaluating the integral in closed form (both components for each reflection type) is also possible but does not lead to good results, so the authors suggest evaluating only a single selected component, setting the other one to 0.

The difficulty when sampling such BRDF functions is the need of mapping the distribution into polar coordinates and then projecting the value back to the Cartesian shading coordinates. The F function for the diffuse component can be derived from the integral through the solid angle in a way very similar to the following:

$$\begin{aligned} F_d(a, b) &= \int_{\Omega_{ab}} \frac{1}{\pi} \cos \theta \omega = \int_0^a \int_0^b \frac{1}{\pi} \cos \theta \sin \theta d\theta d\phi \\ &= \frac{1}{4\pi} \int_0^a \int_0^{2b} \sin(2\theta) d(2\theta) d\phi \\ &= \frac{1}{4\pi} \int_0^a [-\cos(2\theta)]_0^b d\phi \end{aligned} \quad (2.34)$$

We can get marginal F^{-1} functions as

$$\begin{aligned} F_d(a, b = \frac{\pi}{2}) &= \frac{1}{4\pi} \int_0^a (\cos 0 - \cos \pi) d\phi d\phi \\ &= \frac{\phi}{2\pi} F_d^{-1}(u) = 2\pi u \end{aligned} \quad (2.35)$$

$$\begin{aligned} F_d(a = 2\pi, b) &= \frac{1}{4\pi} \int_0^{2\pi} (1 - \cos(2b)) d\phi = \frac{2\pi}{4\pi} 2 \sin^2 b = \sin^2 b \\ F_d^{-1}(u) &= \arcsin \sqrt{u} \end{aligned} \quad (2.36)$$

Then the sampling direction in polar coordinates according to the paper will be

$$(\theta, \phi) = (a \cos \sqrt{u_1}, 2\pi u_2) \quad (2.37)$$

for two uniform variables u_1 and u_2 . The direction of the specular reflection can be analogously computed to be

$$(\theta, \phi) = (a \cos(u_1^{\frac{1}{n+1}}), 2\pi u_2) \quad (2.38)$$

Translating the direction to shading coordinates can be done as follows:

Diffuse component:

$$(x, y, z) = (\cos \phi \sin \theta, \cos \theta, \sin \phi \sin \theta) \quad (2.39)$$

Specular component:

$$(x, y, z) = (\mathbf{x}_r \cos \phi + \mathbf{y}_r \sin \phi) \sin \theta + \mathbf{z}_r \cos \theta \quad (2.40)$$

where $\mathbf{x}_r = \mathbf{z}_r \times \mathbf{n}$, $\mathbf{y}_r = \mathbf{x}_r \times \mathbf{z}_r$ and \mathbf{z}_r is the direction of specular reflection.

This importance sampling technique allows to significantly reduce the amount of noise in the final image. The image rendered with importance sampling, compared to the image where the two components are sampled separately but uniformly, is shown at fig. 2.14

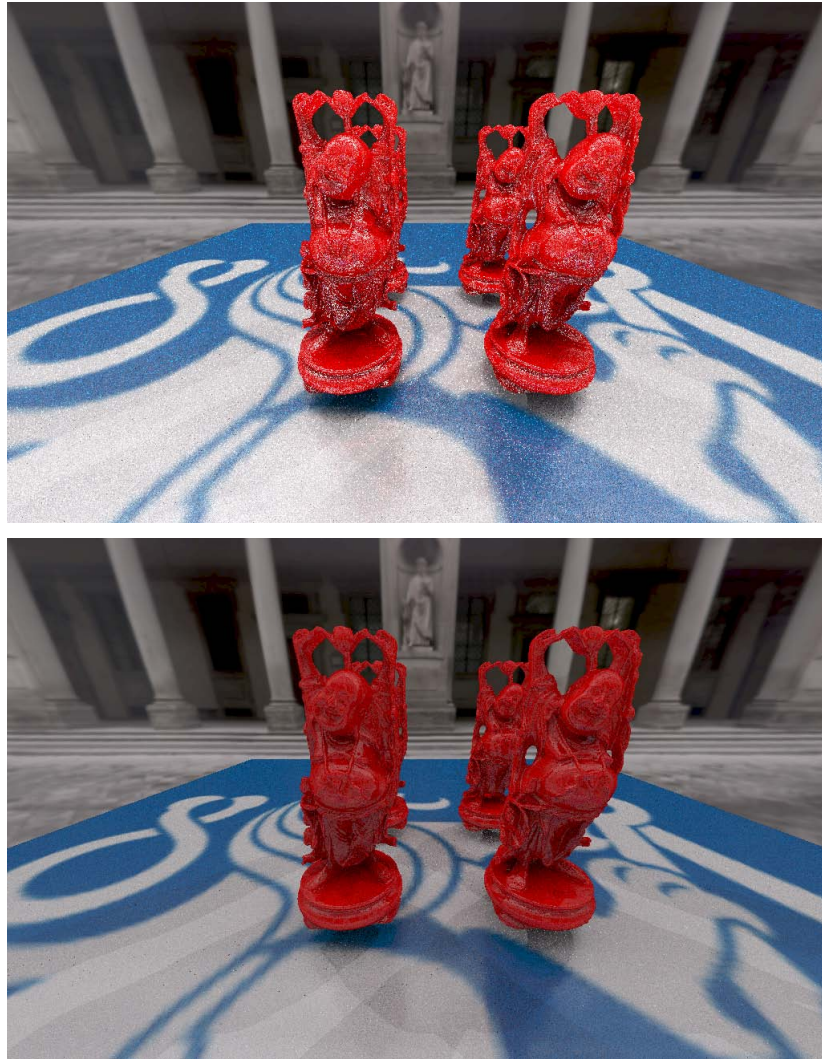


Figure 2.14: Scene rendered with 100 samples, without importance sampling (top) and with importance sampling (bottom).

2.6.2 Russian roulette

The other subject to optimization is the efficiency of the estimator, inversely proportional to the time it took to calculate the samples. While importance sampling aims to prefer samples that contribute the most, the Russian roulette method aims to cancel the computation of samples that contribute the least. This is achieved not taking a sample with probability q and when taking a sample weighting it with its probability $q - 1$. The estimator will then be updated as

$$\bar{I}' = \begin{cases} \frac{\bar{I} - qc}{1 - q}, & u > q \\ c, & \text{otherwise} \end{cases} \quad (2.41)$$

where c is a default sample value mostly set to 0.

The value of q can be set constant as well as be computed from the throughput value as

$$q = \max_{rgb} \left(\prod_{d=1}^D \frac{f(p_d, \omega_{od}, \omega_{id}) \cos \theta_{id}}{p(\omega_{id}) q_d} \right) \quad (2.42)$$

Computing q dynamically results in reduced number of "fireflies" that occur when the sampling probability is low while the sample value is high.

When using dynamic, q however, it is possible that some dark regions will never get sampled and paths with big BRDF values like mirrors will be too long. For this reason, the minimum and the maximum number of bounces should always be set.

2.7 Path tracing

One way of integrating the rendering equation is using path tracing. Unlike other ray tracing techniques, it samples only one light path for a pixel, instead of the whole tree of rays. Each time light hits the surface, the BSDF function is integrated, taking the radiance from the next hit point as an input leading to the recursive definition 2.43⁴. The value of the last sample is assumed to be 0.

$$\begin{aligned} L_o(p_n, \omega_o) &= L_e(p_n, \omega_o) + \int_{S^2} f(p_n, \omega_o, \omega_i) L_i(p_{n+1}, \omega_i) |\cos \theta_i| d\omega_i \\ L_o(p_D, \omega_o) &= L_e(p_D, \omega_o) \end{aligned} \quad (2.43)$$

We can now rewrite the integrals into Monte Carlo sums and expand the recursive definitions, taking the sum out:

$$\begin{aligned} L_o(x_0, \omega_o) &= L_e(x_0, \omega_o) + \frac{1}{N} \sum_i^N \left(\frac{f(x_0, \omega_o, \omega_i)}{p(\omega_i)} (L_e(x_1, \omega_o) + \right. \\ &\frac{1}{N} \sum_i^N \left(\frac{f(x_1, \omega_o, \omega_i)}{p(\omega_i)} \dots \right) \cos \theta_{i0}) \\ &= \frac{1}{N} \sum_i^N (L_e(x_0, \omega_o) \\ &+ \left(\frac{f(x_0, \omega_{o0}, \omega_{i0})}{p(\omega_{i0})} \cos \theta_{i0} \right) L_e(x_1, \omega_{o1})) \\ &+ \left(\frac{f(x_1, \omega_{o1}, \omega_{i1})}{p(\omega_{i0})} \frac{f(x_0, \omega_{o0}, \omega_{i0})}{p(\omega_{i0})} \cos \theta_{i1} \cos \theta_{i0} \right) L_e(x_2, \omega_{o2})) \end{aligned} \quad (2.44)$$

This equation can be then rewritten to the following pseudocode.

⁴This notion can also be formalized as a path integral proposed by Veach [11]

```

vec3 radiance(wo, x, d){
    L = emis(x); // in case of miss, sample the environment emis
    if(d == D || !is_hit(x)){
        return L;
    }
    wi, f = brdf_sample(x); // f is weighted by sample probability
    x1 = trace(x, wi);
    L += f * dot(wi, norm(x)) * radiance(wi, x1, d++);
    return L;
}
vec3 integrate(x){
    col = 0;
    prim = cameraRay(x);
    x = trace(cam, prim);
    for (i=0; i < N; i++){
        col += radiance(prim, x, 1);
    }
    return col / N;
}

```

The most common implementation of the algorithm relies on recurrent call to the radiance function. In some cases, though, the algorithm needs to be implemented in constant memory, e.g. because of hardware limitations. As can be seen from the equation 2.44, the BRDF samples and cosine terms are multiplied cumulatively. This accumulator called throughput

$$T = \prod_{d=1}^D \frac{f(p_d, \omega_{od}, \omega_{id}) \cos \theta_{id}}{p(\omega_{id})} \quad (2.45)$$

and we can use it to modify the algorithm.

```

vec3 radiance(wo, x){
    L = 0;
    T = 1;
    for (d=1; ; d++){
        L += T * emis(x);
        if (!is_hit(x) || d == D){
            break;
        }
        wi, f = brdf_sample(x);
        T *= f * dot(wi, norm(x));
        x = trace(x, wi);
    }
    return L;
}

```

We can now modify the termination condition to employ the throughput-based Russian roulette:

```

vec3 radiance(wo, x){
    L = 0;
    T = 1;
    for (d=1; ; d++){
        L += T * emis(x);
        q = max(T);
        if (d < Dmax && d >= Dmin && urand() < q){
            T /= q
        }else{
            break;
        }
        if (!is_hit(x)) break;
        ...
    }
    return L;
}

```

This is the basic path tracing implementation used in projects like *smallpt* (<http://www.kevinbeason.com/smallpt/>). Here it can be seen that all the light in the scene actually comes from emissive components. The problem with this approach is similar to the one with uniformly sampling BRDF, meaning that when light sources are small, the scene would appear dark. In this work, it was decided to use point light sources which cannot be sampled at all.

2.7.1 Light sources and next event estimation

To address the problem of light sources, the *next event estimation* technique is used. Its goal is to efficiently sample direct illumination from light sources and combine the result with indirect light when evaluating radiance. The direct light component for point lights can be simply added to the rendering equation:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{H^2} f(p, \omega_o, \omega_i) L_i(p, \omega_i) \cos \theta_i d\omega_i + E_l(p)$$

$$E_l(p) = \sum_{l \in \text{lights}} \int_{H^2} I_l V(p_l, p) f(p, \omega_o, \omega_i) \cos \theta_i d\omega_i \quad (2.46)$$

where I_l is the intensity of the light source, $V(p_l, p)$ is the visibility function equal to 1 if the point of a light source p_l is visible from the shading point.

The common practice in path tracing is to apply Monte Carlo-like method to sampling point lights as well. Only one light source is sampled at a time, and the result is divided by the sampling probability. The directions of light sources are known, and the integral is zero elsewhere, so the equation simplifies to:

$$E_l(p) = I_l V(p_l, p) f(p, \omega_o, \omega_i) \cos \theta_i * \text{num_lights} \quad (2.47)$$

where θ_i is the angle at which light arriving from the uniformly chosen random light source.

We can implement the visibility test by casting a shadow ray, hence the final modification to the ray tracing pseudocode:

```
vec3 radiance (wo, x) {
    L = 0;
    T = 1;
    for (d=1; ; d++){
        ... // Russian roulette and emission
        l = lights [urand()*num_lights];
        l_dir = dir(x, l);
        s = trace(x, l_dir);
        if (!is_hit(s)) {
            L += T * intensity(l)
                * brdf_eval(x, l_dir) * dot(l_dir, norm(x));
            * num_lights;
        }
        ... // sampling indirect light
    }
    return L;
}
```

When calculating the light source contribution, we need to use *brdf_eval* to evaluate BRDF for the light source direction. For Phong BRDF, we also

find that evaluating both components produces very noisy results, so it is better to only evaluate one of them.

We still use emissive components, but small objects cannot be used here as light sources efficiently. Next, event estimation for light sources with a surface would result in them contributing both to direct and indirect illumination. This problem can be solved for smaller light sources by disabling emissive component everywhere except the first and the last bounce, or generally by applying *multiple importance sampling* technique.

2.8 Modelling camera

Camera models are being used to specify different ways to cast the primary ray. The simplest and widely used is the pinhole camera model.

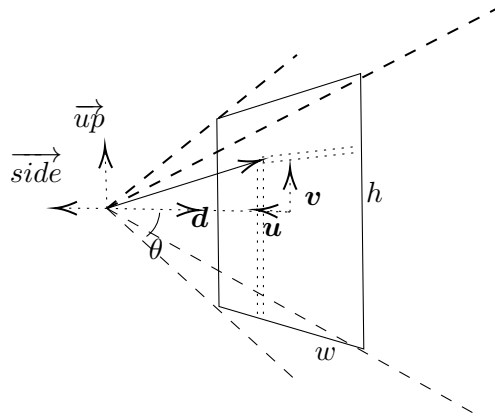


Figure 2.15: Pinhole camera described by near and far plane distance (near plane being the screen), vectors \vec{u} , \vec{side} and direction \mathbf{d} , as well as aspect ratio and FOVy angle, here presented as $\theta = \frac{FOVy}{2}$. w and h here represent the size of screen in pixels.

To calculate the primary ray, we first transform pixel coordinates to the range $[-1, 1]$, so that zero lies on the camera axis, and then get the ray direction:

$$\begin{aligned} \vec{u} &= 2 \frac{x_{pix}}{w} - 1 & \vec{v} &= 2 \frac{y_{pix}}{h} - 1 \\ \vec{r} &= \vec{side} \cdot \mathbf{u} + \tan \theta \cdot \mathbf{v} \cdot \tan \theta \cdot h \vec{u} + \mathbf{d} \end{aligned} \quad (2.48)$$

Other camera models deal with simulating camera lenses and aperture. This could produce various effects like depth of field or fish eye.

2.9 Other ray tracing techniques

The approach of tracing rays from camera back to light sources used by Whitted is referred to as backward ray tracing. As opposed to it, other ray tracing techniques exist, namely light tracing and bidirectional path tracing.

Light tracing is forward ray tracing, i.e. tracing rays from light sources until they hit the camera. Whitted-type ray tracing has hard time finding paths like EDS^+L ⁵ because the diffuse BRDF does not favor any direction, so the specular path will probably not be found. The same goes for refracted rays causing that various light caustics phenomena e.g. thin lenses will not be rendered properly. The same paths can be though effectively sampled with light tracing. The disadvantage of such approach is much higher computational intensity due to rendering points invisible from camera and being restricted by $L.S^+.E$ paths from "cutting corners" via *next event estimation*⁶. This approach is hence used mainly for computing light maps or radiosity maps and as part of bidirectional path tracing.

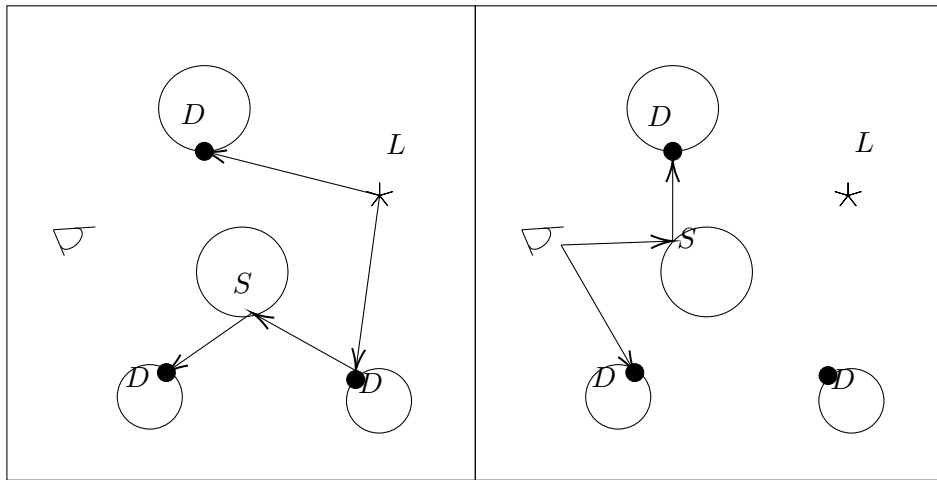


Figure 2.16: Light tracing and ray tracing steps. Dots represent the points at which values are stored. It can be seen that some stored values may remain unused.

Bidirectional path tracing is a combination of Whitted ray tracing and light tracing, first suggested in a paper by Paul Heckbert [8]. The algorithm consists of two stages (fig. 2.16):

- First comes light tracing pass, which traces paths $L(S^*D)^+$ and stores all the diffuse illumination values into a texture. In the Heckbert's approach, the diffuse illumination values are computed via radiosity algorithm, which yields much better results for diffuse reflections than ray tracing.
- Next the ray tracing pass is made sampling paths ES^*D and reading the diffuse component value stored in the texture created by the previous pass.

This results in an approach of sampling paths $L(S^*D)^+ DS^*E$, the first part representing all the components ending with D and the second part

⁵regular expression for ray path string where E stands for eye, L for light source, S and D for specular and diffuse reflections.

⁶casting the ray in the direction of interest instead of waiting for it to be cast there randomly

adding an arbitrary number of *Ses*. This can be also written as $L(S|D)^*E$, which represents all possible ray paths.

Bidirectional approach may be used to produce better results for diffuse illumination and can simulate some phenomena that backward ray tracing can not. Currently, it is the most used technique in production ray tracers.

2.10 Acceleration and BVH

The path tracing algorithm would be very expensive to compute if each ray-triangle intersection test would be calculated linearly for every triangle in the scene. For this reason, certain tree-like *acceleration structures* were designed to make intersection tests logarithmic in number of primitives. The two basic types of approaches are:

- Subdividing space, e.g. using K-d trees. Possible improvements include adaptive subdivision depending on number of triangles. Such approach tend to take bigger amounts of memory but produce slightly better results.
- Subdividing objects into *bounding volume hierarchies*(BVH) which are trees of depth $\log_{num_descendants} num_objects$ where leaves are objects wrapped, typically, into AABBs. When constructing BVH each time a subdivided axis needs to be chosen, then to create child bounding volumes we need to choose a subdividing pivot. Axis is chosen as the one with the biggest distance between primitives' centers of masses (fig. 2.17). The subdivision pivot should be chosen so that the amount of computation is evenly distributed between the node children. The heuristics can be the number of triangles or object's surface area.

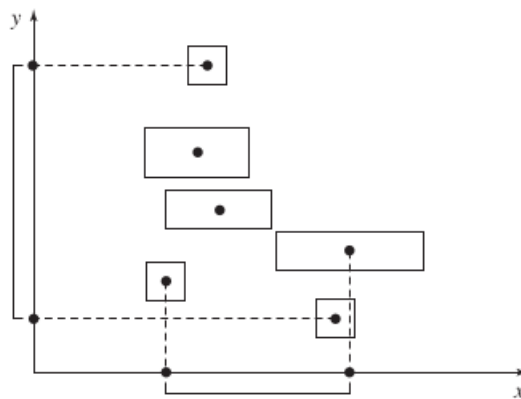


Figure 2.17: Choosing volumes subdivision axis [2]

Both techniques can significantly speed up finding intersections, but the computation still takes a lot of time. Another speedup comes from massive parallelization of acceleration structure construction and traversal. The BVH

traversal intersection tests can run in shaders (fig. 2.19), but still never in real time.

NVIDIA has recently enabled ray tracing capabilities in their drivers to build highly efficient BVH and traverse them, while the ray tracing logic is left to specialized shaders. This allowed to separate the bottleneck tasks of ray tracing and offload them to hardware that is optimized for the task, allowing for real-time performance. Such hardware, namely dedicated ray tracing cores, is a part of modern RTX series GPUs. BVH traversal and ray-triangle intersections can run on RT cores, while ray generation and hit processing is left to the shader (fig. 2.18).

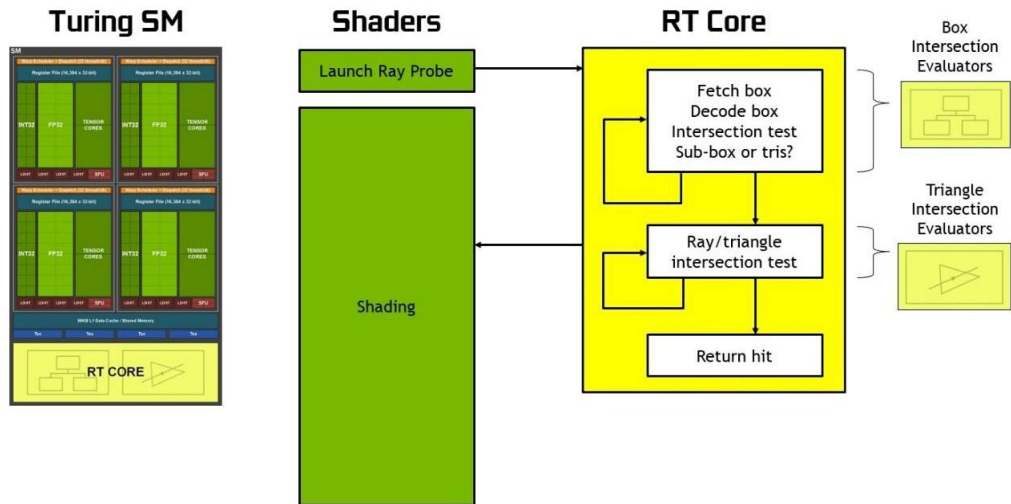


Figure 2.18: Ray tracing on RT cores [13]

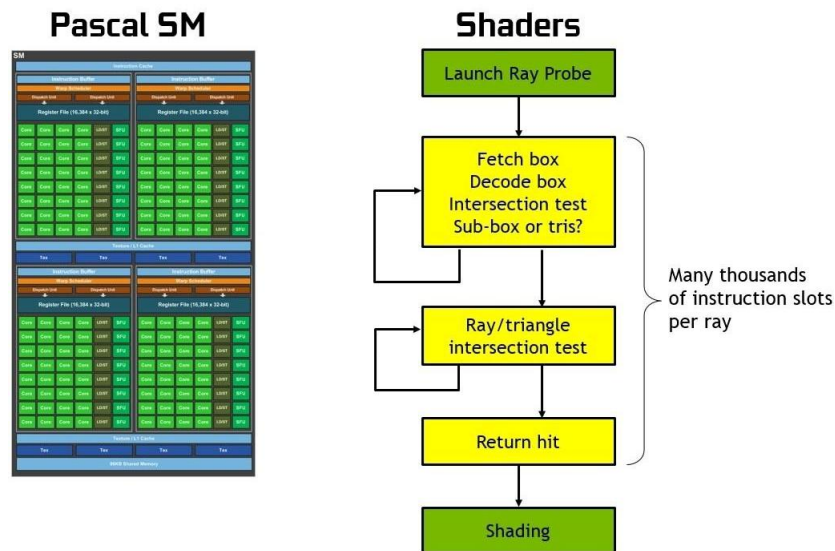


Figure 2.19: Software emulated ray tracing [13]

Chapter 3

Structure and workflow of Vulkan API

Vulkan API is a modern cross-platform graphics API designed to provide the developer with maximum control over the program execution as well as provide the system itself with as much as possible information about the user intentions for better optimization.

3.1 Memory structure

There are three types of memory for storing Vulkan objects:

- *Device memory* is memory that is managed by the developer, who is responsible for its allocation and deallocation.
- *Resource pools* are memory that is shared by objects of the same type such as `VkCommandBuffer` or `VkDescriptorSet` whose lifetime is controlled by a `Pool` object, meaning that it is on driver to allocate them.
- Memory allocated with custom allocators.

Memory is allocated from multiple heaps that depend on device and are distinguished by physical location, size, alignment and by operations they can perform. Not all memory can be mapped by the user, so for creating a vertex buffer in GPU memory it is needed to create an intermediate *staging buffer* and then copy it into the desired heap.

From the developer's perspective, the device memory is managed using buffers. When creating a buffer, the developer specifies the heap type (host or device memory, visibility, writing behavior) and how the buffer will be used (e.g. for ray tracing or as memory transfer source). After buffer creation, memory of the size of the buffer must be allocated separately and then bound to the buffer using `vkBindBufferMemory()`. Host is responsible for the buffer lifetime, so at the end it should be deallocated with `vkDestroyBuffer()`.

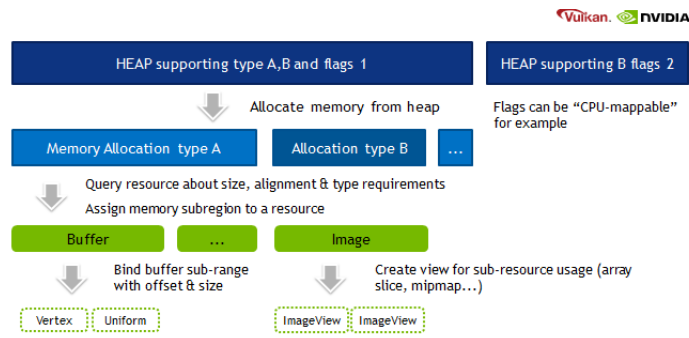


Figure 3.1: Memory hierarchy [13]

3.2 Rendering pipeline

Rendering pipeline in Vulkan is similar to OpenGL pipeline in the sense that it consists of same programmable and fixed stages.

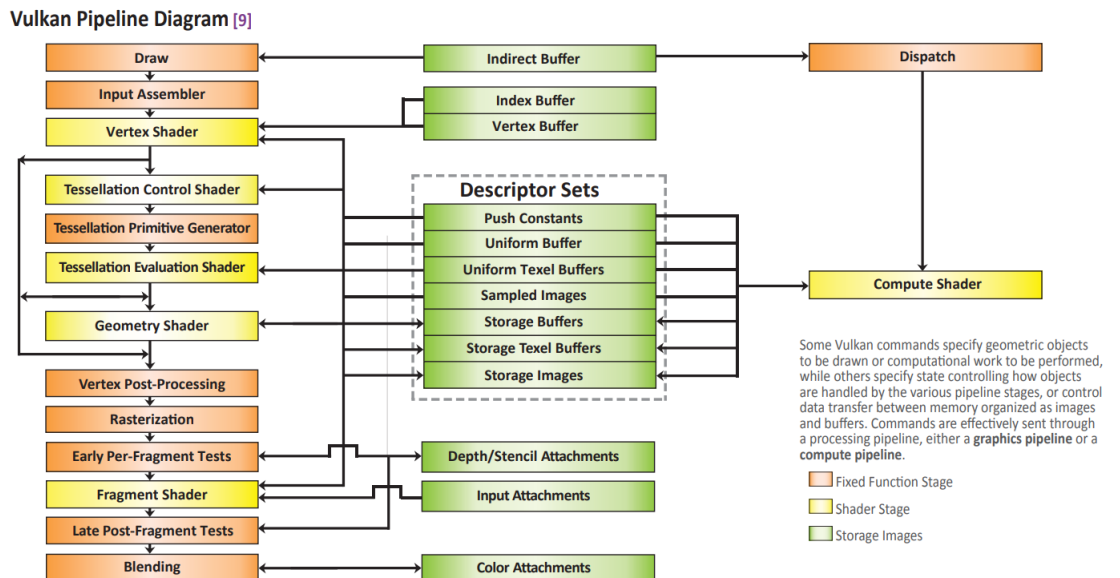


Figure 3.2: Vulkan rendering pipeline [12]

- *Input Assembler* converts vertex and index buffers into input to the vertex shader.
- *Vertex, Tessellation and Geometry Shaders* are manipulating data associated with each vertex in a programmable way. Like in OpenGL vertex shader gets access only to one vertex at a time, while tessellation and geometry shaders run on whole primitives like triangles.
- *Rasterization* stage projects primitives on screen in order from back to front discarding those that are covered, discarded by depth testing or oriented in an opposite direction and creates *fragments* to be

sent to fragment shader. During the stage creation, type of fragments (triangles/lines/points) as well as fragments orientation for face culling.

- *Fragment Shader* stage performs operation on each fragment given the interpolated fragment information and outputs the final fragment color.
- *Blending* stage defines the behavior occurring when several fragments are projected into one screen pixel. They can be combined with some fixed (e.g. 0 : 1) ratio, or alpha blending may be used.

In Vulkan after the pipeline is created it remains fixed except only the parameters provided in `VkDynamicState` during pipeline creation. To change any other parameters such as shaders, type of rasterization or blending function the pipeline needs to be re-created.

Shader input variables are passed to the pipeline by directly binding buffers with `vkCmdBindVertexBuffers()` and `vkCmdBindIndexBuffers()` before each draw call. Uniform buffers on the other hand are not bound directly, but with *descriptor sets*.

■ 3.2.1 Descriptor Sets

Descriptor sets are the primary way of transferring uniform data from CPU to GPU. Their goal is specifying the way uniform buffers bind to stages of graphics pipeline and to specific uniform variables.

Descriptor sets are allocated from a pool object. After allocation descriptor gets updated with `VkWriteDescriptorSet` structures by calling `vkUpdateDescriptorSets()`. Each structure defines an update of one or more *descriptors*, which define bindings between a uniform variable (according to layout) and some interval in the uniform buffer. By default, descriptors sets are immutable and cannot be updated while used in pipeline.

After descriptor sets are created, they can be bound to graphics pipeline before each draw call by `vkCmdBindDescriptorSets()`. Data stored in uniform buffers associated to each descriptor get updated during the bind call.

■ 3.3 Command buffers

Commands are Vulkan built-in procedures to be performed on the graphics card. For performance reasons, commands in Vulkan are pre-recorded and stored in a special buffer named `VkCommandBuffer`, which is then submitted for execution. This way, Vulkan can re-order the command sequence of execution based on given information so that they execute more efficiently.

The whole lifetime of the command buffer is made explicit to the developer. First, it is required to create `VkCommandPool` object that would manage lifetimes of command buffers. The command pool is tied to a specific queue

family, which means that it should execute only some certain set of commands. The purpose of it is to allow for parallel execution of commands that do not share resources, e.g. commands from graphics queue and from presentation queue. After that, one or multiple buffers are allocated. When allocating, the type of command buffers should be specified. There are two types of command buffers: primary buffers, that are submitted, and secondary that can not be submitted directly but have some shared commands that can be called from primary buffers.

After creating a buffer, commands should be recorded into it. Recording starts with call to `vkBeginCommandBuffer()` and ends with call to `vkEndCommandBuffer()`. The first transitions the buffer from Initial to Recording state, the second - from Recording to Executable state; all the commands in between are recorded to the buffer.

Command buffer is submitted to the corresponding queue using `vkQueueSubmit()`. After submission, the host process has no information on when the buffer starts executing and when it finishes. To wait until the buffer finishes its execution, some synchronisation primitives to wait for can be passed to `vkQueueSubmit()` or `vkQueueWaitIdle()` that make the whole application wait until all the queue is finished.

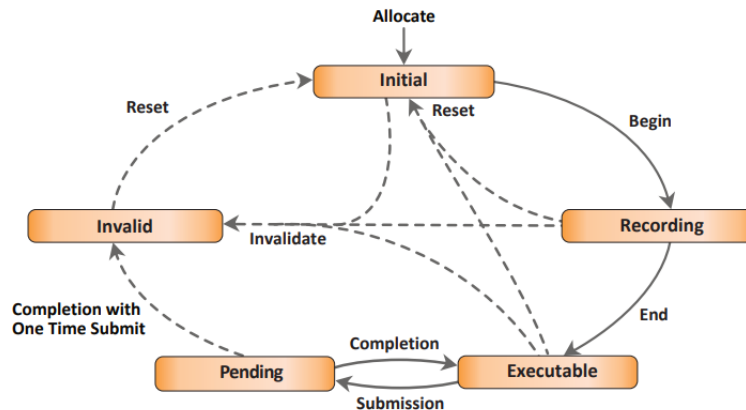


Figure 3.3: Command buffer life cycle [12]

3.4 Synchronization

When some commands are submitted to a queue, it is generally not known in which order they will be executed. Although even if submitted from different command buffers, commands on the same queue will not be executed in parallel, they will probably be executed out-of-order. That's why programming in Vulkan is largely about enforcing execution order via synchronization primitives.

The most basic synchronization primitive, arising in different forms, is barrier, set e.g. by `vkCmdPipelineBarrier(srcStageMask, dstStageMask)`. When this command is submitted, all commands submitted after it will wait at the `dstStage` until all the commands submitted before the barrier finish the

execution of `srcStage`. This allows to set dependencies between commands, ensuring that commands from one set *happen-before* commands in the other.

A slightly modified version of barrier `vkCmdMemoryBarrier(srcStageMask, dstStageMask, srcAccessMask, dstAccessMask)` may be used to not only order the commands with shared resources but also ensure that the resources are transferred correctly. After the commands at `srcStageMask` finish, the memory should be made *available*, meaning, that all the dirty L1 caches from stages in `srcStageMask` modified as `srcAccessMask` need to be written to the L2 cache. Also, before the `dstStageMask` commands start executing the memory will be made visible to the stages and `dstAccessMask`, meaning that the corresponding L1 caches will be invalidated.

Similar approach is used when transferring image layout for different usages via `vkCmdImageBarrier(srcStageMask, dstStageMask, srcLayout, dstLayout)`.

When creating the rendering pipeline, it is necessary to synchronize across different queues and with the host, e.g. graphics queue needs to signal for the presentation and to the host to acquire the next frame. For such purposes, semaphores and fences are created. Vulkan offers multiple ways for sequencing the execution of commands, which often relate to synchronization options passed to. One of them is specifying `pWaitDstStageMask` property to wait for certain pipeline stages before starting the buffer execution. Other ways involve own synchronization primitives of the framework, such as fences, semaphores and barriers.

- `VkFence` is used for synchronization of commands execution with host application. Fences can be only signalled by the device i.e. when passed to the `VkQueueSubmit()`, can be waited for using `vkWaitForFences()` and then reset later with `vkResetFences()`.
- `VkSemaphore` is similar to `VkFence` except that the synchronization only runs on device, and they are reset automatically at submission.

■ 3.5 Ray tracing in Vulkan

■ 3.5.1 Bottom and Top Level acceleration structures

One way to speed up the ray tracing computation is to exclude parts of the scene that a ray will definitely not intersect. This can be solved by using acceleration structures that store geometry information efficiently:

- Bottom level acceleration structure (**BLAS**)
Is built from the geometry vertex and index buffers. Contains triangles and bounding volumes for geometries.
- Top level acceleration structure (**TLAS**)
Stores pointers to BLAS structures along with per-instance data such as transformation and shader binding table offset. Multiple instances may point to the same BLAS for reusing the geometry data.

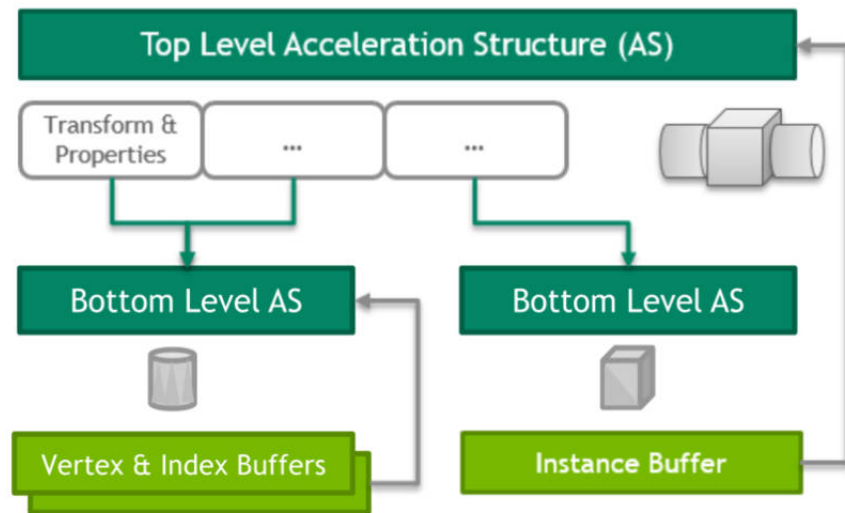


Figure 3.4: Acceleration structures hierarchy [14]

Data for the TLAS in Vulkan are provided as an array of *GeometryInstance* structures. The structure is defined as

```
struct GeometryInstance {
    float transform [12];
    uint32_t instance_custom_index : 24;
    uint32_t mask : 8;
    uint32_t instance_offset : 24;
    uint32_t flags : 8;
    uint64_t acceleration_structure_handle;
};
```

In this structure, *instance_custom_index* and *instance_offset* provide the shader information in a way we will discuss later. *GeometryInstance.mask* can be used to identify the geometry instance, for example to cull it while testing visibility, passing a corresponding argument to *traceNV* call in shader.

■ 3.5.2 Ray tracing pipeline

As already mentioned, ray tracing is a massively parallel task of computing ray intersections with scene objects. In Vulkan (similarly to DirectX) this process of tracing rays is implemented in a dedicated pipeline. Unlike the graphics pipeline, it is recursive and contains significantly more programmable parts.

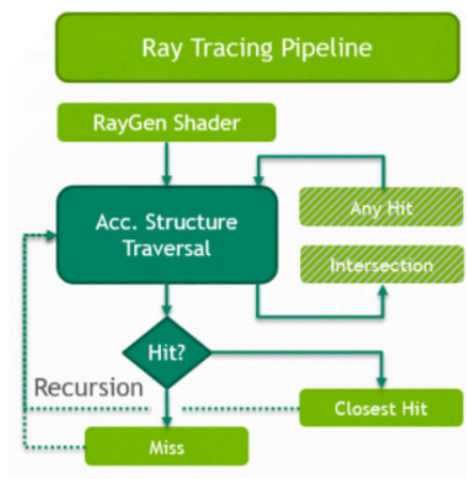


Figure 3.5: Ray tracing pipeline [14]

The whole pipeline is started after submitting `vkCmdTraceRaysNV` command, which creates a number of threads that generate rays and traverse the acceleration structure. Behavior of acceleration structure traversal is defined in Any Hit and Intersection shaders. There are also Closest Hit and Miss shaders that determine the final result of tracing a ray. Shader functions are as follows:

- **Ray Gen** (*Required*) generates rays and starts tracing. Must be implemented in shader function `traceNV`.
- **Intersection** allows for defining intersections with arbitrary primitives. If an intersection occurs. If not implemented, the default triangle intersection will be used.
- **Any Hit** is invoked for all ray intersections with primitives.
- **Closest Hit** (*Required*) is invoked for the closest intersections with primitive after all Any Hit calls. Returns lighting in the intersection point.
- **Miss** (*Required*) is called if no intersection was found in the given range. Returns color of the environment.

All the three shaders are bound to the pipeline via Shader Binding Table (SBT), which allows for quick switching between shaders, e.g. for different types of objects.

■ 3.6 Shader binding table

The table is a buffer of shader records. Each record consists of a shader handle and some optional parameters passed to it. The same shader handle may be added multiple times with different parameters. Shaders are added

to the table in the form of a group, a set of shaders with the same binding parameters. There are separate groups for generation, intersection, hit, and miss shaders. Most of the groups would contain only one shader, but it is common for a hit group containing both closest hit and any hit shader. To run the pipeline, generation, hit, and miss groups must be bound.

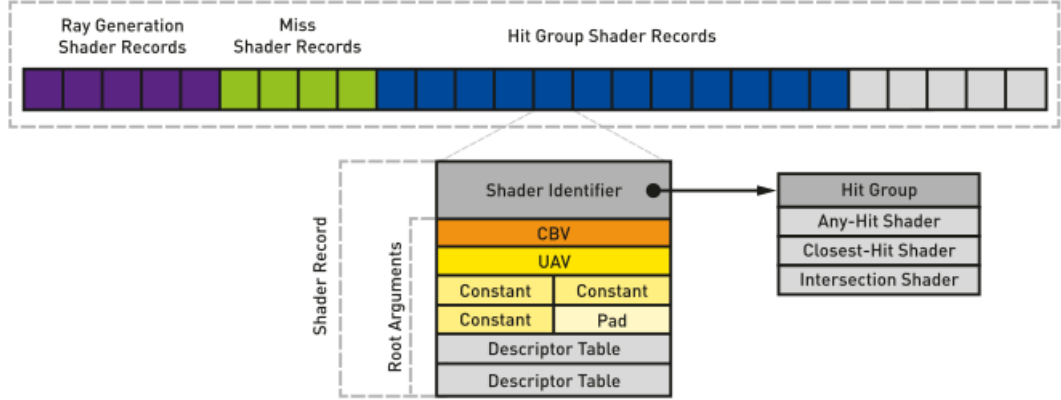


Figure 3.6: Shader binding table in DXR [1]

Different instances may have multiple shader groups attached. The common use case for this is adding separate Miss and Hit shaders for occlusion testing to reduce the payload. To use a different shader, it is then enough to specify a different shader offsets to the traceNV call in the shader.

The SBT address of the hit shader associated with the instance is calculated by the following formula.

$$H = HG[0] + H_{stride} \cdot (R_{offset} + R_{stride} * G_{id} + I_{offset}) \quad (3.1)$$

where:

$HG[0]$ points to the start of the hit groups buffer.

H_{stride} is the shader handle size that we can get from the `VkPhysicalDevicePropertiesRayTracingNV` structure.

R_{offset} is the index of the shader group to be used (the type of shader). Set from the traceNV call.

R_{stride} is the number of shader groups (the type of shader). Also set from the traceNV call. Can be set to 0 for all the instances to use the same shaders.

G_{id} is the geometry index, set either in the GeometryInstance or as the instance index in the array.

I_{offset} is the offset of shader subset of the instance. Can also be specified in the GeometryInstance.

The miss shader offset calculation is simpler:

$$M = \&MG[0] + M_{stride}R_{miss} \quad (3.2)$$

where R_{miss} is the parameter of the traceNV and M_{stride} is the size of the shader handle and $MG[0]$ shows the start of the miss groups buffer.

There can also be multiple gen shaders, although they cannot be changed "on the fly" but only by the new command buffer submission.

The layout of the SBT slightly differs across different APIs. It brings more flexibility in working with multiple shaders, but is quite difficult to be used correctly.

Chapter 4

Implementation

The related implementation was created in bare Vulkan API. It aimed to be as simple as possible and did not introduce much abstraction over the API.

Ray tracing is not a standard part of Vulkan. In this implementation, *VK_NV_ray_tracing* extension was used. This extension was meant as a demo for the *VK_NVX* NVIDIA extension that actually uses RTX technology. The resulting code can easily be converted for *VK_NVX* or *VK_KHR* extensions, since all the calls are similar.

The resulting program consists of three main parts:

- *VulkanBase* class containing all the basic initialization, scene creation, event handling and swapchain management code.
- *RayTracerNV* derives from *VulkanBase* and provides code related to ray tracing pipeline through virtual methods.
- *RayTracerNVOverlay* derives from *RayTracerNV* and provides rasterized gui (text) rendering on top of the ray tracer output image.

To start the application, either *RayTracerNV* or *RayTracerNVOverlay* instance can be used. In the following sections, we will discuss the function of these classes in detail.

4.1 VulkanBase

The abstract class *VulkanBase* is designed to provide the functionality that is independent on whether the application uses ray tracing or the classical rasterization pipeline.

First, the window needs to be initialized. For this purpose, we use the GLFW library designed for OpenGL applications. When initializing the window GLFW should be explicitly told not to create an OpenGL context, instead, we create a *Vulkan instance* by ourselves and use a *rendering surface* provided by GLFW.

We create an instance using `vkCreateInstance`. In the corresponding structure, we specify the list of instance extensions required by the GLFW returned by `glfwGetRequiredInstanceExtensions` function and enable validation layers if they are supported.

Then we need to discover the GPU to run on and check whether all the required extension are supported. Then we create a logical device with these extensions enabled. The extension include *VK_KHR_swapchain*, *VK_NV_ray_tracing* and helper extensions *VK_KHR_maintenance3* for additional device properties and *VK_EXT_descriptor_indexing* for simplifying work with descriptors. When creating the logical device, we also create two command queues: the presentation queue that will be used for showing the swapchain image and graphics queue that will be used for the rest of commands.

For on-line rendering, we need to initialize a swapchain of images to draw to. The API provides maximum and minimum number of swapchain images that can be used; the minimum value is commonly 2, so that one image can be drawn into while the other is being shown to the user. Although since graphics and presentation commands can run in parallel, it is better to have a third image prepared not to wait until the other image is released. Hence, in this application we will use a swapchain of three images, the approach which is also called *triple buffering*.

Finally, we record the drawing commands to command buffers. Because commands are pre-recorded, we would need a separate command buffer for each swapchain image. Some commands for timing are recorded in the base class, while the actual drawing commands that depend on a particular pipeline are recorded in the overridden method *VulkanBase::recordPipelineDrawCommands*.

After the initialization, the *VulkanBase::run()* method is called to start rendering frames. The logic of drawing a frame is located in *VulkanBase::drawFrame* method ¹ and can be described by the following pseudocode.

¹The method is based the tutorial code freely available at https://vulkan-tutorial.com/code/15_hello_triangle.cpp.

```

#define IMG_COUNT 3
#define MAX_FRAMES_IN_FLIGHT 10
VkSemaphore rendSem[MAX_FRAMES_IN_FLIGHT];
VkSemaphore swapSem[MAX_FRAMES_IN_FLIGHT];
VkFence frameFences[MAX_FRAMES_IN_FLIGHT];
//Initialize the previous three arrays
...
VkFence imageFences[IMG_COUNT];
int frameInd = 0;
void drawFrame(){
    // Pick the next image to render to
    int imgInd;
    vkAcquireNextImageKHR(..., &imgInd,
        swapSem[frameInd], ...)
    if(/*resized*/){
        windowResized(); // re-initialize
            swapchain and pipeline
        return;
    }
    // Wait till the previous frame is
        rendered to this image
    if(imageFences[imageInd] != 0){
        vkWaitForFences(...,
            &imageFences[imageInd], ...);
    }
    imageFences[imageInd] =
        frameFences[frameInd];
    // Do some per-frame operations
    ...
    // Then on the device:
    // 1. Execute render commands.
    // 2. At the image writing stage wait for
        the swapchain to release the image.
    // 3. Write the image, signal the
        semaphore for the presentation queue,
        and signal the fence to start rendering
        the next frame to the image.
    :

```

```

vkResetFences (... , &imageFences [ imageInd ] );
vkSubmitQueue ( graphicsQueue , ... ,
{ ...
pCommandBuffers =
    &commandBuffers [ imageIndex ] ,
pWaitDstStageMask = * _ATTACHMENT_OUTPUT_* ,
pWaitSemaphores = &swapSem [ frameInd ] ,
pSignalSemaphores = &rendSem [ frameInd ] } ,
imageFences [ imageInd ] )
// Wait for image to get written and start
  presenting it
vkQueuePresentKHR ( presentQueue ,
{ pWaitSemaphores = &rendSem [ frameInd ] ,
pImageIndices = &imageInd ,
pSwapchains = &swapchain } );
// Switch between frames in circle
frameInd = ( frameInd + 1 ) %
    MAX_FRAMES_IN_FLIGHT;
}
:

```

The reason for having all the synchronization primitives is that CPU is able to submit work much faster than GPU can process it, and therefore waiting for the pipeline to get empty is inefficient. Instead, we constantly feed the pipeline with commands for different images and then wait for the swapchain images to become available. The number of command buffers submitted to the pipeline at any time is the number of so-called *frames in flight*. The optimal maximum number of frames in flight depends on the speed of the particular hardware but comes with little overhead, so it should rather be set bigger.

4.2 RayTracerNV

This class built on top of the *VulkanBase* creates a ray tracing pipeline using the API provided by the *VK_NV_ray_tracing* extension.

Unlike in the rasterization pipeline, input to the ray tracing pipeline is provided in the form of acceleration structures, which we need to first build. The process of building acceleration structure of the scene consists of the following steps:

1. For each scene object, pass the geometry index and vertex buffers info to the *vkCreateAccelerationStructureNV* and get an opaque acceleration structure object.
2. Using the method *vkGetAccelerationStructureMemoryRequirementsNV*

query the type and the maximum size of memory needed for the AS. Then we allocate the required memory and assign it to the acceleration structure via *vkBindAccelerationStructureMemoryNV*.

3. Go through the same steps for the TLAS except that instead of geometry data, we pass data for each instance. Instances data are specified in a standard framework *GeometryInstance* structure, which is not included in the extension and should be defined by the user. In the structure, we specify the BLAS reference and the world transformation. Multiple *GeometryInstances* may reference one BLAS providing a way of instancing.
4. Build the BLASes using *vkCmdBuildAccelerationStructureNV* command. The command would require a scratch buffer for work that we also should allocate. The buffer should be on device memory of type *__USAGE_RAY_TRACING_BIT_NV* and should be at least as big as any of the acceleration structures.
5. Insert the pipeline barrier on the acceleration structure build stage so that all the BLASes building happens-before the TLAS building.
6. Build the TLAS acceleration structure with the same command as for BLAS also passing a handle to a buffer containing *GeometryInstances*.

If when creating BLASes the *__ALLOW_COMPACTION_BIT_NV* is set in flags, the actual size of the ASes after they are built is smaller than the allocated memory. They can be therefore reallocated to save some GPU memory, but let us leave it in a space of possible improvements.

After the ASes are built, we can create the pipeline. The pipeline does not require specifying any fixed-function stages, so we should only provide shader bindings and descriptor set layout. The shaders are pre-compiled SPIR-V files, which we load to create *VkShaderModules*. A shader binding table is then created from the pipeline.

Passing data to the pipeline is done using descriptor sets that are extended by enabling *VK_KHR_descriptor_indexing* extension which allows passing arrays of values to descriptors, which we can use for all the per-instance data and light sources.

Each frame, the command buffer is submitted to the graphics queue. Aside from performance measurements, the commands written in pseudocode look as follows:

4.4 Path tracing implementation

For implementation of the algorithm it was chosen to use path tracing without recursion described in the Path Tracing section, since in GLSL shaders recursion is not supported. Although it is possible to implement recursion via calling the ray casting function from the Hit shader, such approach can only reach certain maximum hardware dependent recursion depth.

For tracing rays we use the traceNV function provided by the framework that takes the parameters d , t_{min} , t_{max} and o of the ray described by equation $\mathbf{r} = \mathbf{o} + \mathbf{d} * t$, where $t \in [t_{min}, t_{max}]$, and then returns the data passed to it by miss or hit shaders depending on whether miss or hit occurred. This function should only be called from the gen shader, so we cast each next reflected ray (or do not cast in case of a miss) based on payload accepted from the miss or hit shader.

The algorithm of the path tracer the shaders are implementing can be described as follows:

1. Trace a ray from a camera till it reaches a geometry.
2. If a hit has occurred, identify the geometry ID and interpolate normal and UV coordinates from the hit triangle using barycentric coordinates. Write all of these to the ray payload.
3. If no hit has occurred, set a specific payload value telling that the environment map should be sampled.
4. Cast a shadow ray to a randomly selected light source.
5. If a geometry hit has occurred (light source is shadowed) then set resulting ray color to 0.
6. Otherwise, calculate the illumination value and multiply it into the throughput.
7. Then sample the next ray direction from the BRDF function and proceed with 2.

This can be otherwise written in the following pseudo-code, describing the contents of the gen shader.

```

Ray primaryRay = traceNV(<parameters of primary
    ray cast from camera>);
traceSecondary(primaryRay);

void traceSecondary(Ray curRay){
    vec3 endColor = vec3(0);
    vec3 throughput = vec3(1);
    for(i = 0;; i++){
        if(curRay.miss){
            endColor += throughput * envEmission();
            break;
        }
        Point hp = curRay.hitPoint;
        //cast the shadow ray and get the color of
        a visible light source
        Light lightSource = randomLightSource();
        vec3 lightDir = lightSource.direction(hp);
        shadowRay = traceNV(
            direction = lightDir ,
            origin = hp,
            Tmin = 0,
            Tmax = lightSource.distance(hp)
        );
    }
}

```

Figure 4.1: .rgen shader pseudocode (1)

```

//now get the current illumination value
//and the next tracing direction
Samp sampe = BRDFsample( hp.norm,
    hp.material, curRay.dir )
//Phong BRDF (section 2.11) may randomly
//choose not to sample
if (!sampe.sampled || i == RECURSION_DEPTH){
    break;
}
throughput *= sampe.value;
curRay = traceNV(
    direction = sampe.direction,
    origin = hp,
    Tmin = 0,
    Tmax = inf.
);
}
return endColor;
}
:

```

Figure 4.2: .rgen shader pseudocode (2)

The previous pseudocode is the basic implementation with fixed depth and zero emission components for all objects except for the environment. These features could be easily implemented though by small modifications and passing additional material parameters. There is a problem with light sources of high intensity because bright colors are automatically clamped to 1. It can be solved by applying tone mapping based on average radiance, which would take slightly more effort.

Another problem with the algorithm that the big loop over samples is sometimes uncontrollably unwrapped by the compiler for optimization purposes. That is why in our implementation we need to split the samples into smaller patches and compute new color values based on previous ones using the following formula.

$$col = \frac{col \cdot sampleInd + frameSamplesSumColor}{sampleInd + \min(totalSamples - sampleInd, sampPerFrame)} \quad (4.1)$$

In our application we are reading and writing colors to the same image, which is the one being displayed either by copying to the swapchain or as a texture if rasterization overlay is used. Depending on the case we are in, we may or may not need to reorder color components and perform gamma correction. For this reason, it is necessary to call a reordering and gamma correction function when we write and call their inverses when we read to always work with the same color space.

Functions that perform the common operations like image reads and writes, uniform random number generation, primary rays casting etc. are moved to a helper file *gen_helpers.rgen* to let the *gen.rgen* shader only integrate the rendering equation. All the shader interface defines are moved to *gen_interface.rgen* and structure definitions are moved to the *shaders_shared.h* C header file to enable further sharing with the C++ code. This refactoring is made possible by enabling *GL_GOOGLE_include_directive* extension.

Chapter 5

Results

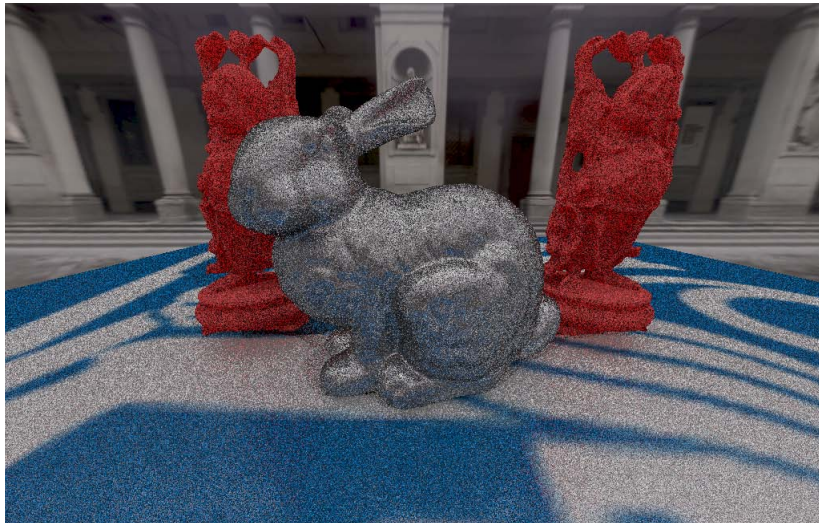
The path tracer provides realistic results for varying material parameters. Results shown here are measured using the implementation from the previous chapter, using no transmitted rays and fixed recursion depth. Also, for the direct light, only the diffuse BRDF component was evaluated, but the light sources' contribution is insignificant compared to the environment light, so it should not be too noticeable.

The table 5.1 shows how the image quality is progressively improving with increasing number of samples. The image at 1000 samples does not differ a lot from the image at 100 samples. The reason for it is the rate of convergence of Monte Carlo integration, which is \sqrt{N} in number of samples. The quality of the random generator influences the results as well.

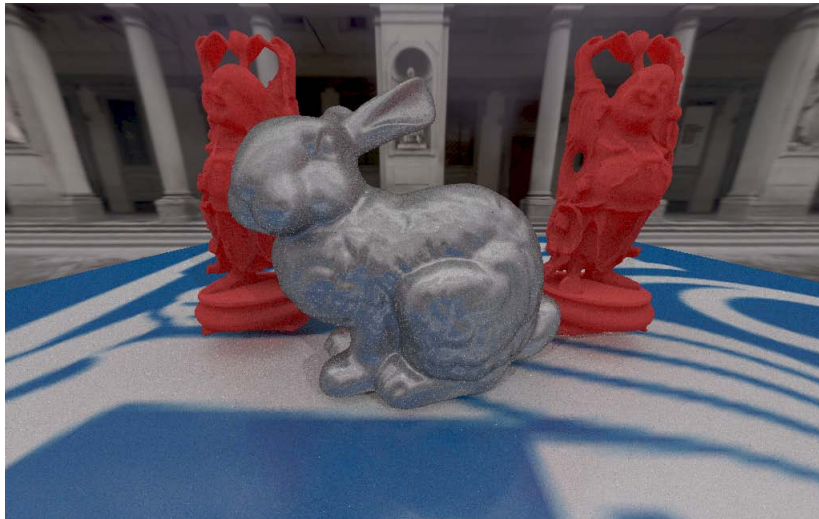
How the rendered image changes depending on the recursion depth is shown in the table 5.2. At depth 1 only camera ray and shadow ray is cast, and no indirect illumination is sampled, that is why the image appears darker. As the number of bounces increases, more orders of reflections are computed and the indirect light contribution gets bigger.

The implementation performance was measured on two NVIDIA graphics cards: TITAN Xp, the high-end GPU from the year 2017 based on Pascal microarchitecture, and the recent GeForce RTX 3060 based on Ampere architecture with RT cores. The measurements of the *secondaryrays/second* for four different scenes are shown in the table 5.3. This is the main metric we will use for evaluating the ray tracing performance. The other two values are the total CPU time T it took to produce the image and the number of secondary rays cast per sample.

S=10



S=100

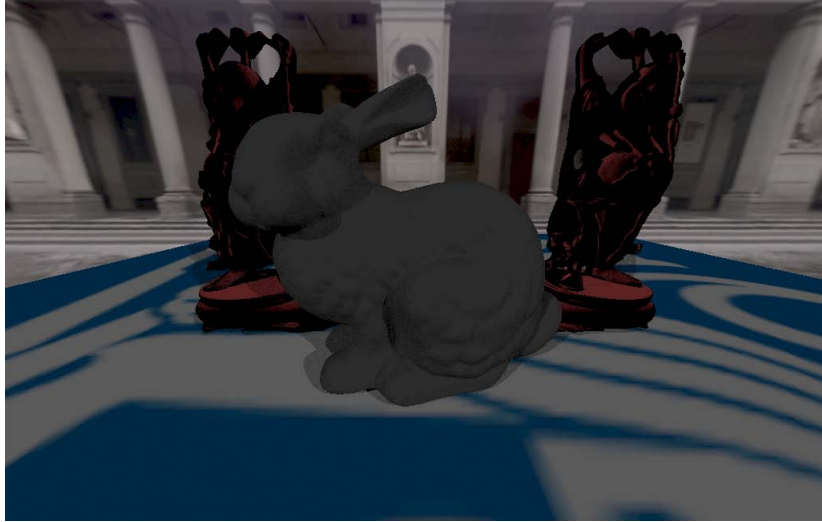


S=1000



Table 5.1: Scene with two direct light sources rendered for different numbers of samples S and fixed depth $D=10$

D=1







D=2



D=10



Table 5.2: Scene with two direct light sources rendered for different recursion depth D with number of samples $S=100$

Scene	Monkeys	Bunny	Happy Buddhas	Happy Buddhas Instanced
				
lights	2	2	2	2
triag.	5810	2.3 mil.	4.3 mil.	4.3 mil.
unique triag.	5810	1.2 mil.	4.3 mil.	1 mil.
TLAS (Kb)	2.1	1.75	1.88	1.88
BLAS total (Kb)	400	$8.16 * 10^4$	$2.88 * 10^5$	$7.2 * 10^4$

NVIDIA TITAN Xp

for S=10

D	T[s]	10^6 r/s	10^3 sr/sp	T[s]	10^6 r/s	10^3 sr/sp	T[s]	10^6 r/s	10^3 sr/sp	T[s]	10^6 r/s	10^3 sr/sp
1	0.04	887.47	0.00	0.029	332.80	0.00	0.016	166.40	0.00	0.016	204.80	0.00
2	0.03	242.04	101.95	0.036	76.07	122.13	0.037	73.96	124.44	0.037	78.31	124.44
3	0.02	177.49	113.36	0.045	60.51	160.49	0.051	53.25	158.31	0.047	56.65	158.31
5	0.03	156.61	120.26	0.051	53.25	193.99	0.06	45.13	208.21	0.054	50.23	216.88
10	0.03	110.93	124.67	0.052	52.20	212.64	0.067	40.34	240.39	0.07	38.59	227.94
50	0.04	63.39	185.13	0.048	56.65	222.80	0.075	35.98	236.35	0.09	29.91	259.04
100	0.04	80.68	133.43	0.057	46.71	187.22	0.066	40.34	235.14	0.063	42.26	230.19

for D=3

S	T[s]	10^6 r/s	10^3 sr/sp	T[s]	10^6 r/s	10^3 sr/sp	T[s]	10^6 r/s	10^3 sr/sp	T[s]	10^6 r/s	10^3 sr/sp
2	0.01	-	0.00	0.024	-	0.00	0.029	-	0.00	0.024	-	0.00
10	0.02	221.87	113.36	0.037	71.96	154.41	0.041	66.56	162.88	0.04	66.56	158.31
50	0.14	9.64	22.67	0.35	0.17	31.22	0.4	6.14	31.66	0.39	6.36	31.66
100	0.3	9.76	11.34	0.78	4.10	15.44	0.85	3.55	15.83	0.82	3.68	15.83
500	1.5	0.50	2.27	4.1	0.71	3.09	4.5	0.72	3.17	4.3	0.75	3.17
1000	3	0.47	1.13	8.3	0.03	1.54	9.1	0.02	1.58	8.8	0.02	1.58

NVIDIA GeForce RTX 3060

for S=10

D	T[s]	10^6 r/s	10^3 sr/sp	T[s]	10^6 r/s	10^3 sr/sp	T[s]	10^6 r/s	10^3 sr/sp	T[s]	10^6 r/s	10^3 sr/sp
1	0.071	887.47	0.00	0.071	665.60	0.00	0.071	295.82	0.00	0.07	332.80	0.00
2	0.071	887.47	101.57	0.07	221.87	121.88	0.071	110.93	124.14	0.07	126.78	124.14
3	0.07	887.47	112.90	0.07	140.13	154.08	0.07	76.07	157.97	0.07	80.68	157.97
5	0.069	665.60	119.70	0.07	106.50	175.12	0.069	66.56	208.06	0.069	70.06	199.99
10	0.07	887.47	124.01	0.07	98.61	182.32	0.07	63.39	235.37	0.07	64.94	229.52
50	0.07	665.60	131.55	0.071	98.61	182.65	0.07	61.92	237.18	0.071	66.56	232.09
100	0.073	532.48	132.72	0.073	91.81	182.65	0.071	61.92	237.40	0.071	68.27	233.12

for D=3

S	T[s]	10^6 r/s	10^3 sr/sp	T[s]	10^6 r/s	10^3 sr/sp	T[s]	10^6 r/s	10^3 sr/sp	T[s]	10^6 r/s	10^3 sr/sp
2	0.038	-	0.00	0.037	-	0.00	0.048	-	0.00	0.043	-	0.00
10	0.074	887.47	112.90	0.071	110.93	154.08	0.075	66.56	164.62	0.07	91.81	157.97
50	0.34	36.54	22.58	0.34	23.54	30.82	0.36	6.86	31.59	0.34	8.24	31.59
100	0.67	37.59	11.29	0.67	7.59	15.41	0.78	3.80	15.80	0.67	4.62	15.80
500	3.3	0.24	2.26	3.3	1.03	3.08	4.1	0.71	3.16	3.4	0.85	3.16
1000	6.7	0.14	1.13	6.7	0.74	1.54	8.2	0.46	1.58	6.9	0.54	1.58

Table 5.3: Test results for scenes of varying complexity

We see that on all the scenes, the performance of the RTX GPU for different recursion depths are from roughly one third to ten times better on all the scenes. The best performance gain is demonstrated on Monkeys scene. The monkey models take up a small part of the screen and most of the rays do not hit anything, hence the biggest speedup probably comes from an efficient acceleration structure traversal by the RT cores.

When it comes to tests for different numbers of samples, though, the GeForce RTX tests may even run slower as the value reaches 500. The reason for this is that not all samples are calculated at once, but the intermediate results are stored and presented to the user, which becomes the actual performance bottleneck. Unfortunately, our application failed to measure the GPU time for the two samples case, but we can still imagine that the number is quite big.

When comparing results across different scenes, we see that the performance on the RTX card is more dependent on the number of visible triangles. The Bunny scene has a big bunny model with a relatively small number of triangles taking up a lot of view space, while the four Buddha models in the Happy Buddhas scene are much more complex. On TITAN the two scene show almost the same complexity for secondary rays, while on the RTX card the Bunny scene is rendered almost twice as fast.

Interestingly, the instanced version of the Happy Buddhas scene is always rendered slightly faster. The only way it differs is by the TLAS structure pointing to the same Buddha model rather than to different models, so the performance increase might come from some caching while accessing the BLAS memory.

The CPU time in the two tables is different since the application was tested on different machines and different environments, the second one being tested on a notebook. The same applies for the number of rays per sample. The latter would probably not differ if the random sequence generator for this implementation would not use a periodic function that can be evaluated differently on different hardware, and would use for example LFSR for generating pseudo-random numbers.

The ray tracing performance on modern GPUs can actually be done in real time for some smaller number of samples. The goal of a ray tracing programs now is to rather optimize passing the data to the GPU and try to reduce the host-device synchronization and CPU overhead.



Conclusion

In this work we have gone through the basic theory of path tracing from ground up as well as some more advanced concepts like Cook-Torrance illumination model. The implementation of even a basic path tracer in Vulkan, especially with additional profiling capabilities, turned out to be a highly non-trivial task. Because Vulkan gives such a low abstraction and broad opportunities for optimization, it is quite complex to get even the simplest things running in bare API. The application that we have built offers a framework for further experimentation with ray tracing, so some basic path tracers can be implemented just by writing and compiling relatively simple shaders. The framework still has a lot of space for improvements, such as tone mapping, more advanced materials, separate shadow shaders, support for more complex light sources and more.



Acknowledgements

This work would not be possible without multiple blog posts, tutorials and projects that allow to find a way through the API functions, which would be nearly impossible to understand purely from the specification. Also, it is necessary to acknowledge the crucial contribution of the work's supervisor doc. Jiří Bittner for the recommendation to choose this topic, as well as regular feedback, encouragement and direction which largely defined the final result.



Bibliography

- [1] Haines et al. Ray Tracing Gems, Apress, 2019.
- [2] Pharr et al. Physically Based Rendering 3rd edition, Morgan Kaufmann, 2017.
- [3] Tomas Akenine-Möller et al. Real-Time Rendering (4th edition) CRC Press, 2018.
- [4] Turner Whitted. An Improved Illumination Model for Shaded Display. Bell Laboratories, 1980.
- [5] A. Appel. Some techniques for shading machine renderings of solids, IBM, 1968.
- [6] Bruce Walter et al. Microfacet Models for Refraction through Rough Surfaces, Cornell University, 2007.
- [7] E. Lafortune and Y. Willems. Using the modified Phong reflectance model for physically based rendering, K.U. Leuven, 1994.
- [8] Paul S. Heckbert. Adaptive Radiosity Textures for Bidirectional Ray Tracing, SIGGRAPH, 1990.
- [9] I. Lazany, L. Szirmay-Kalos. Fresnel Term Approximation for Metals, Budapest University of Technology, 2014.
- [10] A. Nugan et al. Experimental Validation of analytical BRDF models, SIGGRAPH, 2004.
- [11] E. Veach, Robust Monte Carlo methods for light transport simulation, Stanford University, 1997.
- [12] Vulkan short reference, www.vulkan.org.
- [13] NVIDIA Developer, developer.nvidia.com.
- [14] Ray tracing in Vulkan, www.khronos.org/blog/ray-tracing-in-vulkan.