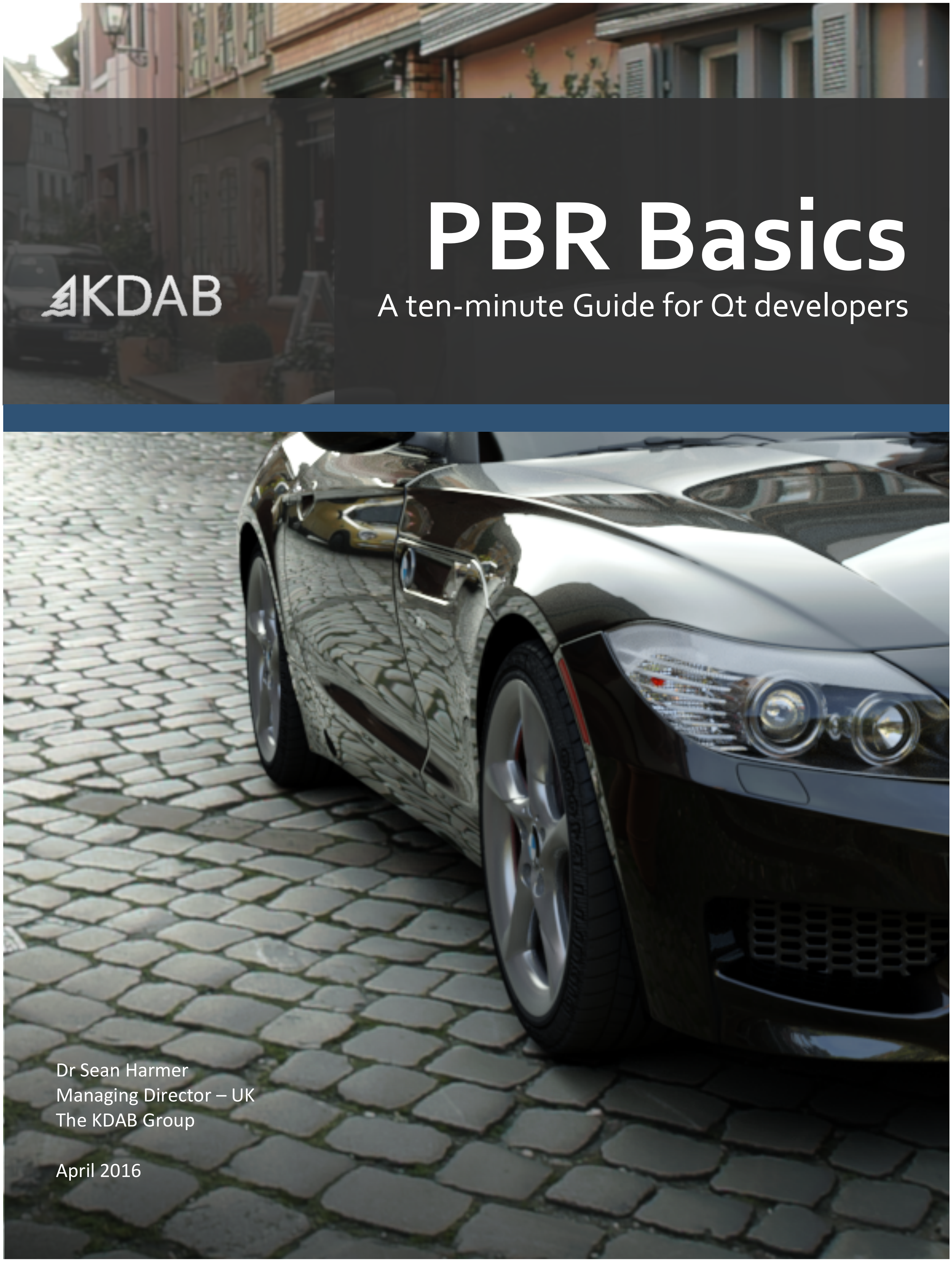

PBR Basics

A ten-minute Guide for Qt developers

Dr Sean Harmer
Managing Director – UK
The KDAB Group

April 2016

(Image courtesy of NVIDIA)

Physically based rendering (PBR) emulates the interaction between light and materials and is a trend in real-time rendering. Here's a ten-minute guide to the essentials.

PBR Basics for Qt Developers

KDAB demo of Dodge Viper using PBR

What exactly is physically based rendering (PBR)? It is the latest in a never-ending pursuit of more realistic computer generated imagery, a system that replaces common shortcuts for rendering surfaces with methodologies rooted in the physical world. Fast becoming a standard in game development circles, PBR is a general set of principles rather than a hard-and-fast standard, meaning that you can find multiple interpretations that all rightfully call themselves PBR. The primary defining feature of any PBR system though, is that it more accurately simulates the behavior of light through the methodical application of reflection, scattering, and absorption physics.

Other than enhanced realism, one of the biggest benefits of a PBR approach is that it is much more foolproof. PBR renders materials that look correct regardless of the lighting environment. Artists no longer need to tinker with esoteric parameters of their objects to get things to "look right" because the shader properly renders an object's look based on its properties and the given lighting. What's more, PBR's inherent ability to create great-looking graphics with consistently defined properties makes it easier to collaborate and share object libraries.

The words "physical" and "physics" both derive from the Latin word for nature, so it's not a coincidence that physical rendering that mimics nature requires physics. But aren't we already emulating physics in how we mimic the path of light in existing non-PBR rendering models? Yes and no. Let's take a look at the main properties that distinguish PBR from earlier models.

Material

Representing an object's texture with physically derived properties is probably the most distinguishing feature of a PBR system. While there may be differences in the properties of any given implementation, these three fundamental attributes are common to nearly all.

PBR differs from older rendering models in that the GPU shaders can create realistic rendering without needing specialized object hints or lighting tricks.

Albedo

The albedo of an object is the color of its diffuse reflected light. Think of this as the color of the object under perfectly even, omni-directional, balanced white light. You might be thinking albedo seems like just a more scientific name for “diffuse color”—what’s the difference? PBR differs from older rendering models in that shader computations take the place of texture hints (in legacy OpenGL) or non-energy conserving shader formulations. PBR does not need to tweak colors or include directional light highlights to ensure the object displays properly under the given lighting conditions. You no longer need extra “artist supplied” information to trick the rendering engine into displaying as desired. With PBR, you represent an object with real physical characteristics and a PBR texture map of an object contains its flat, unaltered color.

“By modeling physical phenomena rather than approximating observation, we can achieve more mathematically stable and photorealistic visual fidelity”

Koray Hagen, Sony

Conductivity

We’re not used to thinking about the conductivity of a surface to understand how to render it. But remembering that light is actually an electromagnetic phenomenon, it makes sense that a PBR system needs to know an object’s conductivity to predict how light will interact with its surface. A measure of conductivity—often simplified as “metalness”—specifies how strongly light will reflect, as well as whether reflected light will shift towards a new hue. Insulating surfaces (like rubber, paint, or wood) absorb most light hitting the surface, while conducting surfaces (most metals) reflect most light and potentially color it. Some PBR systems are able to also specify materials like rubies, emeralds, or sapphires as semi-conductors—something between a conductor and an insulator—allowing partial reflection, absorption, and coloration of light.

A conductivity map allows the “metalness” property to vary across an object. Conductivity textures can realistically depict painted metal by indicating where the surface is painted (and diffusely reflecting light), and where the underlying metal is visible through wear or scratches (and reflecting light in a specular manner).

One way in which the microfacet theory comes into PBR systems is in the selection of the shader algorithms used to calculate the specular reflection.

Microfacets control reflections from roughness of the carpet to smoothness of the chairs

(Image courtesy of NVIDIA)

Microfacets

Understanding a material's appearance requires an understanding of the microscopic properties of the surface. If the surface is broken into many tiny randomly oriented facets, it will spread reflected light across a wide range of angles, blurring reflections and spreading out highlights. If the surface is very smooth, reflected light will bounce back in a precise angle from the surface, resulting in sharp reflections and narrow highlights. Of course it's completely impractical to specify microscopic surface characteristics with a tremendous array of polygons, so generally a "roughness" map is used to specify the microscopic nature of a surface.

A roughness map may also be paired with a cavity map, which captures features that are larger than the surface roughness but also much smaller than the object's polygon mesh. Those small surface features have the property of trapping and absorbing light. A cavity map allows the renderer to more accurately represent features like pits, nicks, grooves or other features that are impractical to represent in triangles. In practice, this is often represented by the combination of a height map and a normal map, and used in conjunction with one of a number of possible occlusion mapping algorithms.

In addition to the roughness map, the other way in which the microfacet theory comes into PBR systems is in the selection of the shader algorithms used to calculate the specular reflection. Many such algorithms exist; each specialized to best handle certain lighting conditions or materials. Using different algorithms allows the number and orientation of a surface's microfacets to react differently to light depending on an object's desired appearance. For example, the GGX distribution gives a wider fall off for the specular highlights than the traditional Blinn-Phong model, providing a better-looking appearance for real world surfaces. There are also shaders tailored for specific use cases, such as handling rough matte surfaces like clay pots or dealing with transparent materials. Generally you'll want to minimize the total number of shaders employed, as each shader change requires a costly context shift on the GPU.

Now that we're done talking about materials, what other features distinguish a physically based renderer? There are two main behaviors of light that change within a PBR system.

PBR’s emulation of the Fresnel effect shows light at a very shallow angle creating bright reflections for “shiny” objects … and for flat and matte objects.

“Physically based rendering is great because it simplifies the asset creation process and makes it difficult for artists to create unrealistic results”

Martin Thomas, Extremeistan

Conservation of energy

This isn’t just a law from your undergraduate physics class. It’s an observation that the amount of light an object reflects can never be more than the amount of light hitting that object. Although this might seem obvious, earlier models to emulate specular highlights (such as the Blinn-Phong reflection model) often resulted in producing more light than what actually hit the surface. While it mostly works, it also doesn’t look natural.

Obeying conservation of energy in PBR means that the amount of light reflected by diffuse and specular sources should never exceed 100%. (Of course this rule doesn’t apply if the object rendered is a light emitter, like a neon sign or fluorescent tube.) Conservation of energy and surface roughness combine to control an object’s highlights and overall appearance. An object with a rough surface reflects more diffuse light, while a smooth surface reflects more highlights. But due to conservation of energy, switching an object’s texture between rough and smooth does not change the total amount of light reflected.

Fresnel effect

Have you noticed that if you look straight down into a pool you can see the bottom but if you look across its surface you’ll see a reflection of the sky? That is due to the Fresnel effect, which in plain English means the more glancing an angle of light that hits a surface, the more that surface will reflect the light. The critical thing that makes PBR different from other rendering methods is an acknowledgement that the Fresnel effect doesn’t just apply to “shiny” objects (like the surface of a pool), it applies to every object. Yes, even seemingly drab surfaces like bricks and cardboard exhibit the Fresnel effect.

Simulating the Fresnel effect with precision requires a considerable amount of complicated math, a significant speed concern in a real-time rendering system. In PBR systems that need rapid rendering, a rough and ready approximation for the Fresnel effect computes an interpolated reflected color using the angle of light, the conductor color, and the metalness map.

Getting graphics to display with a PBR rendering model comes down to using a PBR-aware engine and providing material information in your textures.

Using PBR

Getting your graphics to display with a PBR rendering model comes down to using a PBR-aware engine and providing appropriate material information in your textures. Many existing rendering engines (Unreal Engine 4, Unity 5, Frostbite) and tools (Marmoset, Sketchfab) have already made the move to PBR for you. However, if you aren't using an existing engine, you'll need to implement the necessary physics equations in your shader.

If you're using Qt, PBR can be implemented with custom shaders in Qt3D. For an example of this in action, see our Dodge Viper demo. This demo shows a mix of standard Qt controls mixed with PBR-based 3D rendering, and is made possible through a new PBR layer that KDAB is developing on top of Qt 3D. We are upstreaming this Qt PBR component into Qt Automotive as a first test deployment of the technology. The eventual goal is to make it part of a future Qt 3D release, making Qt just as capable and current as the rest of the graphics world.

Shiny slate tiles show an instance of the Fresnel effect

(Image courtesy of NVIDIA)

About the KDAB Group

The KDAB Group is the world's leading software consultancy for architecture, development and design of Qt, C++ and OpenGL applications across desktop, embedded and mobile platforms. KDAB is the biggest independent contributor to Qt. Our experts build run-times, mix native and web technologies, solve hardware stack performance issues and porting problems for hundreds of customers, many among the Fortune 500. KDAB's tools and extensive experience in creating, debugging, profiling and porting complex applications help developers worldwide to deliver successful projects. KDAB's trainers, all full-time developers, provide market leading, hands-on, training for Qt, OpenGL and modern C++ in multiple languages. Founded in 1999, KDAB has offices throughout North America and Europe.

www.kdab.com