Real-Time Physically Based Rendering

122675 *Tom Veltmeijer* *3GA-2PR*

Contents

1. Physically based rendering

1.1. Introduction

Physically based rendering (PBR) is a loosely defined term that describes a certain art pipeline and shading system. Before PBR, many approximations in real-time rasterization were based on observation and artistic intuition. The aim of PBR is to make these approximations based on the measured physical properties of materials and light, which results in more realistic images and an easier, more consistent art workflow.

"Physically based shading means we approximate what light actually does as opposed to approximating what we intuitively think it should do." - Unreal Engine 4 documentation

1.2. Benefits

1.2.1. Realism

The images are more realistic because the materials behave how they would in the real world. The material parameters manually specified by an artist will never produce a more realistic result than scientifically measured parameters. PBR also pushes the programmer to implement an advanced system for reflections, because in the real world, all materials are at least a little reflective.

1.2.2. Art workflow

The art asset creation workflow becomes simpler and more consistent because PBR takes out a lot of the guesswork and manual tweaking. The artists can build a library of materials with the physically measured properties. Then, when they need a model to contain a certain material, they can copy the values from that library. For example, every time an artist requires a model to contain copper, they can look up copper in the library and copy its values. This way, all coppers in the game world have the same, correct material properties. This is especially useful if many artists are creating assets because it creates consistency. If the artists had to guess the values, they would probably all end up with slightly different results. Different lighting conditions also change how a material looks. An artist might tweak the values so the material looks good in one environment, but when the lighting changes it might look off again. The improved artist workflow saves a lot of time and money, especially for big AAA game studios.

1.3. Core concepts

1.3.1. Reflection

Diffuse reflection is when light rays reflect off a surface in many different directions. Specular reflection is when the light rays reflect off a surface in a direction that is close to the mirror image of the incoming direction. Diffuse and specular were already part of most shading systems, however, PBR imposes a few restrictions on them in order to produce more physically plausible results.

The first restriction that was not present in most older shading systems is that the sum of the diffuse term and the specular term cannot be bigger than 1. For example, if the diffuse intensity equals 1, then the specular intensity must be 0, and if the specular term is 0.2, the diffuse term cannot be greater than 0.8.

This restriction is called energy conservation and without it, a surface could potentially reflect more light than it was hit by in the first place. A single ray of light can either reflect by diffuse reflection or by specular reflection, not both at the same time.

1.3.2. Microsurface

Microscopic imperfections of a surface cannot be seen individually, but they change the overall look of a material. A surface with many imperfections, also known as a rough surface, will reflect light in a less predictable way than a perfectly smooth surface. The slight randomness of the reflection vector's direction, creates a visually more blurry specular reflection.
Another manifestation of the previously mentioned energy conservation is that a rough surface shows wider but duller reflections than a smooth surface. This is because the rough surface reflects the same amount of light, but over a larger spread of directions. Old shading techniques such as Phong shading do not take this into account but they can be easily modified to do so. For example, Phong is made energy conserving by scaling it by `(a + 2) / (2 * PI)`, where `a` is the specular power.

1.3.3. Fresnel

When a ray of light hits a surface, Fresnel's law states that the bigger the incident angle, the bigger the chance the ray will reflect by specular reflection. Different materials have different base levels of specular reflectivity, but the reflectivity goes up as the angle increases. At 90 degrees, all materials reach 100% specular reflection. The reason stone walls do not look like perfect mirrors when viewed from a grazing angle, is because of the previously mentioned microsurface. Even though the general surface area of the wall seems to be at a right angle, the microscopic surface details all point in different directions because stone is a rough material. This results in a much blurrier and much duller reflection.

1.3.4. Metals and insulators

There are a few key differences between metals and insulators (non-metals). In general, metals are highly reflective (as in specular reflectivity) while insulators are only slightly reflective. Some metals reflect only certain wavelengths, while insulators reflect all colors.
These are only general observations, but because they apply to most materials they are used in many physically based renderers in order to simplify the material definitions. For example, assume that all metals have no diffuse component and all insulators do not have tinted reflections. Now you only need to store one color per material, because for metals this color can be used as the specular color and for insulators this can be used as the diffuse color. Some renderers, including mine, will therefore take a color parameter and a scalar "metalness" parameter, rather than a diffuse color and a specular color.

2. Engine

2.1. Design philosophy

The engine follows the philosophy that systems should specialize in doing one thing and do that thing well. In this case, the engine should only handle rendering. That means loading files, handling the scene graph and providing the engine with a window are all things that the user is responsible for. The engine has its own model and material format in order to efficiently interpret and render the data, but the user

is in charge of loading these files. The contents of these files are then sent to the engine as an array of bytes that are then appropriately interpreted by the engine.
The engine's interface is designed around Direct3D 11, but this is abstracted away from the user. The interface is on a higher level than Direct3D, which means it exposes less functionality. However, the functionality that is exposed is much simpler and cleaner, because the low level details are handled internally. For example, to resize a texture, my engine only requires a single function call on a Texture object.

2.2. Tools

2.2.1. Mesh tool

The Mesh Tool is a command line tool that converts common model formats into the model format the engine uses. First, the model is loaded using the Assimp library. The tool then checks which vertex attributes the model is made up of. For example, some models only have vertex positions and normals, while other meshes may contain one or more sets of texture coordinates. Finally, the tool outputs a file in the custom model format. This format lets the engine know which vertex attributes are present in the model and how they are laid out in memory.

2.2.2. Material tool

The Material Tool is also a command line tool. This tool converts an HLSL file into a pre-compiled custom material format to be read by the engine. First, the tool loads the HLSL file. The shader describes both the vertex and the pixel shader. Both of these shaders are compiled using the Direct3D interface, and then extra information is collected by using the Direct3D shader reflection interface. The Material Tool uses this information for two purposes. The first purpose is finding out the vertex attribute layout, which is written is stored in the same way as in the Mesh Tool. The second is scanning the constant buffers the shader uses.
These constant buffers are then written to a class in a C++ header file, using the appropriate data types for the members and including padding where necessary. This allows the user to include the header and use this class to intuitively edit members of a constant buffer. The code below demonstrates the header generation.

```
// Constant buffer in shader code
cbuffer Material : register(b1) {
    float4 DiffuseColor;
    float Time;
}

// Automatically generated header with equivalent types and padding where necessary
class Cb_materials_basic_Ps {
public:
    Clair::Float4 DiffuseColor;
    float Time;
    float __padding0__[3];
};
```

2.2.3. Benefits

These tools are used at asset build time, which improves run-time performance and reduces dependencies. For example, the engine itself does not need to know how to read model files and how to scan their vertex attributes, because the Mesh Tool has already done this at asset build time and has saved this as meta data. Because the Material Tool has also put the vertex attribute meta data inside the

custom material format, the engine can quickly compare a model and a material to see if they are compatible. This is all very efficient because most of the work has already been done before the program is even compiled.

3. Implementation

The following sections explain on a high level how I implemented the major elements of my deferred physically based renderer. For more details, refer to the source code and the GitHub page: https://github.com/TomVeltmeijer/D3D11Renderer

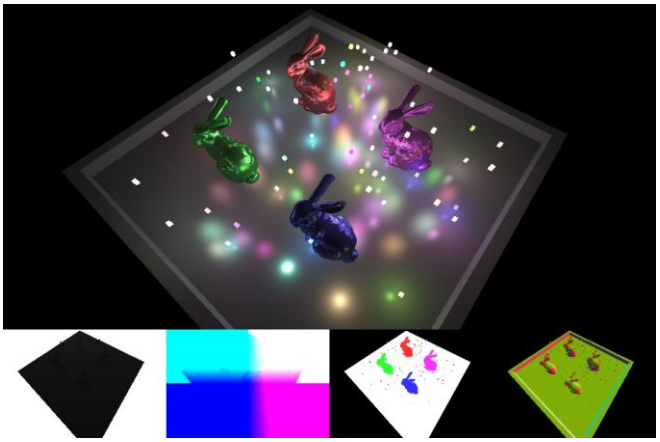
3.1. Linear lighting

Most low dynamic range textures are stored in sRGB space. The channels in this color space assign more values to dark colors than to bright colors. If calculations are done in this warped space, the results will be incorrect.

Colors in this space require `color = pow(color, 2.2)` to be transformed into linear space. After this, all shading calculations are done in linear space. When the final image gets drawn to the screen, `color = pow(color, 1 / 2.2)` is used to transform the image back to sRGB space.

3.2. Deferred rendering

Deferred rendering is a huge topic on its own, so it will not be explained in detail here. There are multiple benefits to deferred rendering, simplification and unification of the shading pipeline being the main reasons I implemented it. All shading is done in a single post-processing shader, clearly demonstrating all the steps taken to produce the final image.



3.3. Environment reflections

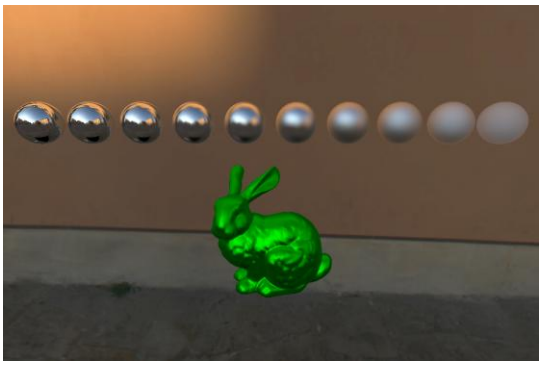
3.3.1. Introduction

There are many different techniques for approximating reflections in current AAA rendering engines. Usually, multiple techniques work together because they all have different strengths and weaknesses. Below are some commonly used methods for simulating reflections. Examples of commonly used techniques are environment mapping, parallax-correct local cubemaps, image proxies and screen space reflections. I have implemented pre-filtered environment map reflections and screen space reflections.

3.3.2. Pre-filtered environment maps

In my implementation, the environment map is an HDR image that I convert to a cubemap, with a certain number of mip maps. I create a render target for each mip map slice of this cubemap. In initialization, I filter each mip slice and render it to the next mip slice. This filtering is done in a pixel

shader that samples the input mip slice according to a bidirectional reflectance distribution function (BRDF).
I picked the Phong BRDF because it is easy to understand and implement and it has good performance.
I implemented the sampling by using an algorithm that samples uniformly on a hemisphere. This hemisphere is oriented with the current direction being evaluated and the size is dependent on the roughness. The roughness to be used when sampling is equal to `currentMip / (totalNumMips - 1)`.



When the reflection of a material with a certain roughness needs to be calculated, the reflection vector is used to get the correct texel of the environment map. The roughness is used to index the correct mip level of the environment map. The spheres in the image on the left shows reflections with different roughness values.

3.3.3. Screen space reflections

I implemented screen space reflections using some of the techniques described by Michal Valient (2014). A ray march is performed by stepping across the reflection ray in projection space. If the ray hits, the hit position's x and y coordinates are used to get the pixel color from the previous frame. Because the previous frame's color information is used, reflections can recurse infinitely.

The screen space reflections are rendered to a separate buffer. This buffer is then filtered in the same way as the pre-filtered environment maps, except it uses fewer samples because it has to filtered every frame.

3.3.4. Combining different techniques

Because the environment reflections and the screen space reflections use the same filtering, they can be blended seamlessly. The specular reflection of the energy conserving Phong shaded point lights also blends with the filtered reflections, because the reflections were filtered according to the Phong BRDF.

4. Conclusion and future work

Physically based rendering has already been widely adopted because of its many benefits. However, even though the inputs to our rendering equations are now based on real world values, realistically modeling the behavior of light in real-time is still a problem.

In order to fully commit to a PBR pipeline, every aspect of light-surface interactions should be modeled. Currently, features such as global illumination, which includes reflections and bounce lighting, area lights and soft shadows are not able to be rendered in real-time. This means that to create a realistic look, artists still have to make manual adjustments such as placing small secondary light sources to simulate bounce light. This goes against the goal of PBR, but rendering technology has not yet advanced to the level where these tweaks are not necessary anymore.

Overall, physically based rendering is a step forward, but rendering technology still requires a lot of improvement before we can take full advantage of it.

5. References

Gritz, L., d'Eon, E. (2007). The Importance of Being Linear.
http://http.developer.nvidia.com/GPUGems3/gpugems3_ch24.html

Karis, B. (2013). Real Shading in Unreal Engine 4.
http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf

Lagarde, S., De Rousiers, C. (2014). Moving Frostbite to Physically Based Rendering.
http://www.frostbite.com/2014/11/moving-frostbite-to-pbr/

Mittring, M. (2009). A bit more deferred – CryEngine3.
http://www.crytek.com/cryengine/cryengine3/presentations/a-bit-more-deferred---cryengine3

Russell, J. Basic Theory of Physically-Based Rendering
https://www.marmoset.co/toolbag/learn/pbr-theory

Valient, M. (2014). Reflections and Volumetrics of Killzone Shadow Fall.
http://advances.realtimerendering.com/s2014/index.html#_REFLECTIONS_AND_VOLUMETRICS