Essay

Contents

[1. Physically based rendering 3](#_Toc421194212)

[1.1. Introduction 3](#_Toc421194213)

[1.2. Benefits 3](#_Toc421194214)

[1.2.1. Realism 3](#_Toc421194215)

[1.2.2. Art workflow 3](#_Toc421194216)

[1.3. Core concepts 3](#_Toc421194217)

[1.3.1. Reflection 3](#_Toc421194218)

[1.3.2. Microsurface 4](#_Toc421194219)

[1.3.3. Fresnel 4](#_Toc421194220)

[2. Engine 4](#_Toc421194221)

[Philosophy 4](#_Toc421194222)

[Tools 4](#_Toc421194223)

[Code architecture 5](#_Toc421194224)

[3. Implementation 5](#_Toc421194225)

[Linear lighting 5](#_Toc421194226)

[Environment reflections 6](#_Toc421194227)

[4. Conclusion 6](#_Toc421194228)

[5. References 7](#_Toc421194229)

# 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.

### 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.

# 2. Engine

## Philosophy

The engine follows the philosophy that systems should specialize in doing one thing and doing it 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.

## Tools

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.

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. This interface for easy extraction of the data flow information of a shader. The Material Tool uses this for two purposes. The first thing is finding out the vertex attribute layout, which is written in the same way as in the Mesh Tool. The second thing 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. The binary compiled shaders and vertex attribute meta data are all output into the same file that the engine can read.

// 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];

};

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.

## Code architecture

The library is based around Direct3D 11, but this is abstracted away for the user.

# 3. Implementation

## Linear lighting

Most textures that are not HDR are stored in sRGB space. This means they require the *color = pow(color, 2.2);* modification to be transformed into linear space. After this, all calculations are done in linear space. When the final image gets drawn to the screen, the *color = pow(color, 1 / 2.2);* modification is used to transform the image back to sRGB space.

Instead of using the power function in the shaders, Direct3D 11 allows the user to specify that a texture is in sRGB space. This will make the hardware appropriately transform texture reads and writes.

**My implementation**

I chose to go with the shader based alternative because it demonstrates more clearly where the transformations happen and this way the gamma value could be adjusted at run-time.

## Environment reflections

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.

**Environment mapping**

This is the cheapest and simplest way to implement approximate reflections. The environment map is a texture of the environment, usually in the form of a cubemap. In the simplest implementation, this is the skybox texture where the reflection vector is used for a hardware accelerated texture lookup. A slight improvement on this is to have multiple local environment maps in different areas of the world.

**Reflection probes**

This can be extended to parallax corrected local cubemaps, which do an even better job at capturing the environment.

**Planar reflections**

Planar reflections is an expensive method to render reflections on a planar surface. Either the camera or the scene is mirrored across the plane and then rendered. This mirror image of the world is then projected onto the plane in the final render pass.

**Screen space reflections**

This is a cheap post processing effect that is good at reflecting dynamic objects such as characters or particles, but it can only reflect the things on screen. In its simplest form, this technique works by ray marching through the depth buffer in projection space. When the ray ends up behind the depth value it is registered as a hit and the corresponding pixel color is used as the reflection color.

**Image proxies**

Similarly to parallax corrected local cubemaps, these capture important scene elements that need a more precise reflection. A manually specified approximately planar part of the scene is rendered and projected onto a plane. In the shaders of nearby objects, the reflection vector is used to do a ray-plane intersection test to get the correct texel. This is not very expensive if only a few nearby image proxy planes are tested against.

**My implementation**

I have implemented pre-filtered environment map reflections and screen space reflections.

# 4. Conclusion

bla

# 5. References

http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013\_pbs\_epic\_notes\_v2.pdf

https://www.marmoset.co/toolbag/learn/pbr-theory

http://www.frostbite.com/2014/11/moving-frostbite-to-pbr/