Essay

# Physically based rendering

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. Brian Karris writes in the documentation of the Unreal 4 engine (which uses a physically based shading system): "Physically based shading means we approximate what light actually does as opposed to approximating what we intuitively think it should do."

The changes that PBR brings affect both programmers and artists.

**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.

# 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 binary compiled shaders and vertex attribute meta data are all output into the same file that the engine can read.

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.

# PBR 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.