Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative

SUPSI

Image-Based Lighting for OpenGL

Student

Denis Beqiraj

Advisor

Peternier Achille

Tutor

Fabrizio Valsangiacomo

Industrial Partner

Bachelor

Bachelor in Computer Science

Project Code

C10520

Year

2021/2022

Date

September 02, 2022

STUDENTSUPSI

Abstract

English

The project consists in applying the image based lighting technique,so a 3D rendering technique which involves capturing an omnidirectional representation of real-world light information as an image, to the 3D Overvision graphics engine, this technique is used to make more realistic the already written physically based rendering pipeline, so the computer graphics approach that seeks to render images in a way that models the flow of light in the real world. This is generally accomplished by manipulating a cubemap environment map (taken from the real world or generated from a 3D scene) such that we can directly use it in our lighting equations: treating each cubemap texel as a light emitter.
This way we can effectively capture an environment's global lighting and general feel, giving objects a better sense of belonging in their environment.
As image based lighting algorithms capture the lighting of some (global) environment, its input is considered a more precise form of ambient lighting, even a crude approximation of global illumination.
This makes IBL interesting for PBR as objects look significantly more physically accurate when we take the environment's lighting into account.
The main goals are to complete implementation of IBL lighting in the engine, and the implementation of an extra dynamic cubemap to make more realistic reflections.
The main goals has been achieved, so the IBL pipeline has been implemented in the engine and the cubemap has been added to have reflections, now we have an engine that can handle physical materials well and make them more realistic through image based lighting.

Italian

Il progetto consiste nell'applicare la tecnica dell'image based rendering, quindi una tecnica di rendering 3D che prevede l'acquisizione di una rappresentazione omnidirezionale delle informazioni sulla luce del mondo reale come un'immagine, al motore grafico 3D Overvision, questa tecnica viene utilizzata per rendere più realistica la già scritta physically based rendering pipeline, quindi l'approccio della computer grafica che cerca di rendere le immagini il più simili possibili al mondo reale tramite il comportamento della luce sugli oggetti.
Ciò si ottiene generalmente manipolando una cubemap (presa dal mondo reale o generata da una scena 3D) in modo tale da poterla utilizzare direttamente nelle nostre equazioni di illuminazione: trattando ogni cubemap texel come un emettitore di luce.
In questo modo possiamo catturare efficacemente l'illuminazione globale e la sensazione generale di un ambiente, dando agli oggetti un migliore senso di appartenenza.
Poiché gli algoritmi di illuminazione basati su immagini catturano l'illuminazione di un determinato ambiente globale, il suo input è considerato una forma più precisa di illuminazione ambientale.
Questo rende IBL interessante per il PBR poiché gli oggetti sembrano molto più accurati fisicamente.
Gli obiettivi principali sono di completare l'implementazione dell'illuminazione IBL nel motore grafico e l'implementazione di una cubemap dinamica extra per creare riflessi più realistici.
Gli obiettivi principali sono stati raggiunti, quindi la pipeline IBL è stata implementata nel motore grafico ed è stata aggiunta la cubemap per avere riflessi, ora abbiamo un motore in grado di gestire bene i materiali fisici e renderli più realistici attraverso l'image based rendering.

Contents

List of Figures

Assigned Project

Description

Image-based lighting (IBL) refers to a series of techniques that are used in computer graphics for lighting 3D objects that are based on treating the surrounding environment as one large source of light (usually implemented via cube maps in real-time GPU-based software).

IBL can thus replace the classic, constant ambient term that is used to "simulate" global illumination by providing a much more detailed reconstruction of how the sourrounding light interacts with the 3D scene.

The goal of this project is to implement real-time IBL (possibily also with the support for on-the-fly-rendered cube maps) on top of OpenGL to improve the quality of the OverVision rendering software that we use at ISIN in various research projects and teaching activities. The IBL implementation will further extend the current physically-based rendering pipeline available in our renderer.

Tasks

- Investigate the state-of-the-art about IBL, from theory to its implementations.
- Discuss with the teacher which variant of IBL to implement as the best trade-off between computational power required, algorithmical complexity, and compatibility with the existing rendering software.
- Define the kind of information 3D assets require to work with IBL, including the identification of sample datasets to be used for development, testing, and validation.
- Integrate IBL into the OpenGL-based 3D rendering software provided by the teacher.

Goals

- Add real-time 3D IBL support to the graphics engine provided by the teacher by making sure that all the software pipelines (using direct and deferred rendering) are operational.
- If need be, extend the graphics engine plugins for the export and adaptation of 3D assets with support for IBL-related information (e.g., for storing high-dynamic range cube map textures).
- Provide a demostrator that clearly shows the advantages of IBL over the previous rendering model.

Introduction

In this chapter we will talk about the background needed to understand the whole project, we will also understand the main idea on how the project has been implemented.
The project starts from the Overvision engine, an engine provided by the advisor, this engine has many features like: point shadows, PBR pipeline, physics and moreover, but we will focus on PBR pipeline in first chapters(it will help us to understand mathematical formulas), after that we will see the implementation of IBL and how it is integrated on the engine, in the end we will talk about dynamic cubemaps.

Keywords

- **Halfway vector**: Halfway vector is a unit vector exactly halfway between the view direction and the light direction.
  The closer this halfway vector aligns with the surface's normal vector, the higher the specular contribution.[1]

- **Radiant flux**: radiant flux is the transmitted energy of a light source measured in Watts.
  Light is a collective sum of energy over multiple different wavelengths, each wavelength associated with a particular (visible) color.
  The emitted energy of a light source can therefore be thought of as a function of all its different wavelengths.[2] The radiant flux measures the total area of this function of different wavelengths.[See figure 1]

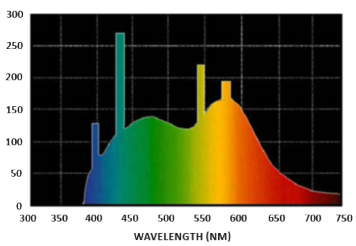

Figure 1: Daylight spectral distribution (source: [3])

- **Solid angle**: The solid angle, tells us the size or area of a shape projected onto a unit sphere.
  The area of the projected shape onto this unit sphere is known as the solid angle.[See figure 2]

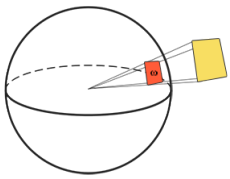

Figure 2: Solid angle (source: [3])

- **Radiant intensity**: Radiant intensity measures the amount of radiant flux per solid angle, or the strength of a light source over a projected area onto the unit sphere.[See figure 3]
  The radiant intensity is calculated by the following formula:

$$I = \frac{d\Phi}{d\omega}$$

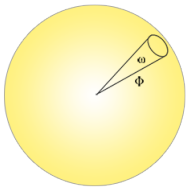

Figure 3: Radiant intensity (source: [3])

- **Refraction and reflection**:In our pbr implemention we assume that the reflection part is light that directly gets reflected and doesn't enter the surface; this is what we know as specular lighting.
  The refraction part is the remaining light that enters the surface and gets absorbed; this is what we know as diffuse lighting.

- **Radiance**: The radiance is used to quantify the magnitude or strength of light coming from a single direction.

- **Convolution**: Convolution is applying some computation to each entry in a data set considering all other entries in the data set; the data set being the scene's radiance or environment map.

- **Geometry obstruction**: approximates the amount the surface's microfacets are aligned to the halfway vector, influenced by the roughness of the surface; this is the primary function approximating the microfacets.

- **Geometry shadowing**: describes the self-shadowing property of the microfacets. When a surface is relatively rough, the surface's microfacets can overshadow other microfacets reducing the light the surface reflects.

- **Gamma correction**: Gamma correction is a nonlinear operation used to encode and decode luminance or tristimulus values in video or still image systems.
  Gamma encoding of images is used to optimize the usage of bits when encoding an image, or bandwidth used to transport an image, by taking advantage of the non-linear manner in which humans perceive light and color.
  The idea of gamma correction is to apply the inverse of the monitor's gamma to the final output color before displaying to the monitor.
  We multiply each of the linear output colors by this inverse gamma curve (making them brighter) and as soon as the colors are displayed on the monitor, the monitor's gamma curve is applied and the resulting colors become linear.
  We effectively brighten the intermediate colors so that as soon as the monitor darkens them, it balances all out.

- **BRDF**: The bidirectional reflectance distribution function is a function of four real variables that defines how light is reflected at an opaque surface.[4]

- **HDR**: Monitors are limited to display colors in the range of 0.0 and 1.0, but there is no such limitation in lighting equations.
  By allowing fragment colors to exceed 1.0 we have a much higher range of color values available to work in known as high dynamic range (HDR).
  With high dynamic range, bright things can be really bright, dark things can be really dark, and details can be seen in both.
  High dynamic range was originally only used for photography where a photographer takes multiple pictures of the same scene with varying exposure levels, capturing a large range of color values.
  Combining these forms a HDR image where a large range of details are visible based on the combined exposure levels, or a specific exposure it is viewed with.

- **Fresnel equation**: The Fresnel equation describes the ratio of surface reflection at different surface angles.

- **Cubemap**: In computer graphics, cube mapping is a method of environment mapping that uses the six faces of a cube as the map shape.

The environment is projected onto the sides of a cube and stored as six square textures, or unfolded into six regions of a single texture.
The cube map is generated by first rendering the scene six times from a viewpoint, with the views defined by a 90 degree view frustum representing each cube face.
In the majority of cases, cube mapping is preferred over the older method of sphere mapping because it eliminates many of the problems that are inherent in sphere mapping such as image distortion, viewpoint dependency, and computational inefficiency.

- **Dynamic cubemaps**:A cubemap made out of provided textures is called static cubemaps, whereas textures provided at run-time is called dynamic cubemaps.
  We use them to catch the environment and through IBL make more realistic the reflections.

State of the art

In this section we will talk about the state of the art of IBL and dynamic cubemaps especially using as example three engines:

- The Rockstar Advanced Game Engine[5]
- Frostbite engine[6]
- Cycles in Blender[7]

The Rockstar Advanced Game Engine

The Rockstar Advanced Game Engine (RAGE) is a proprietary game engine developed by RAGE Technology Group, a division of Rockstar Games Rockstar San Diego studio.
Since its first game, Rockstar Games Presents Table Tennis in 2006, released for the Xbox 360 and Wii, the engine has been used by Rockstar Games' internal studios to develop advanced open world games for consoles and computers.
We are interested on IBL and reflections in fact they have a very good implementation of them in Grand Theft Auto 5.
GTA 5 uses a deferred rendering pipeline, working with many HDR buffers.
So they as a first step, renders a cubemap of the environment.
This cubemap is generated in realtime at each frame, its purpose is to help render realistic reflections later, so this part is forward-rendered.
The environment cubemap we obtained is then converted to a dual-paraboloid map.
The cube is just projected into a different space, the projection looks similar to sphere-mapping but with 2 "hemispheres".
They even optimize decreasing the level of detail and then they have the reflections on PBR materials with IBL.
Here's a car and the environment reflections on GTA 5:[See figure 4]

Figure 4: Grand Theft Auto Five Reflection (source: [5])

Frostbite

Frostbite is a game engine developed by DICE, designed for cross-platform use on Microsoft Windows, seventh generation game consoles PlayStation 3 and Xbox 360, eighth generation game consoles PlayStation 4, Xbox One and Nintendo Switch and ninth generation game consoles PlayStation 5 and Xbox Series X/S, in addition to usage in the cloud streaming service Google Stadia.
The game engine was originally employed in the Battlefield video game series, but would later be expanded to other first-person shooter video games and a variety of other genres.
To date, Frostbite has been exclusive to video games published by Electronic Arts.
Frostbite has implemented Image Based Lighting to make the scene more realistic, in fact they use that technology to make games photorealistic without heavy loading the GPU, they apply the technique that we are currently using in Overvision, so the results are similar to ours.[See figure 5]

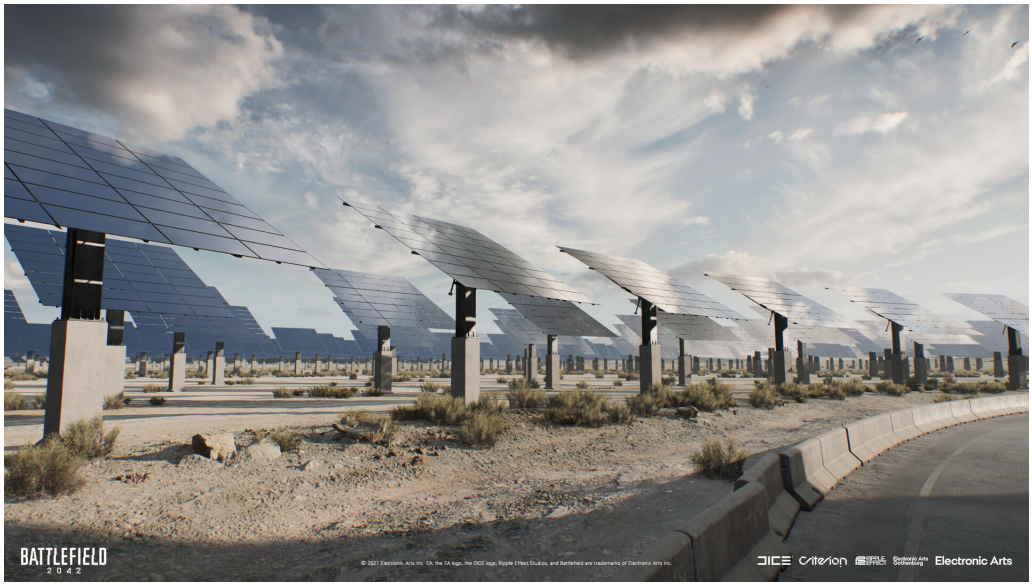

Figure 5: Battlefield 2042 photorealistic environment (source: [8])

Cycles

We have always talked about realtime rendering, but Image based lighting is extensively used in offline rendering, that's because instead of approximate results to have them immediatly, we have more time to render images or videos because it's done offline, as example we have Cycles.
Cycles is Blender's physically-based path tracer for production rendering.
It is designed to provide physically based results out-of-the-box, with artistic control and flexible shading nodes for production needs.
With that engine we have a different PBR pipeline, so a different one for IBL, that differs because we don't approximate results and we get slower, but more precise results.[See figure 6]

Figure 6: Offline rendered spheres using Cycles (source: [9])

Overvision engine

OverVision is a compact 3D graphics engine built for fast prototyping and pedagogical applications in the domain of Computer Graphics and Virtual Reality, it reduces development times and simplifies deployment by offering all its functions through one single library and one unified API. It brings many useful features as:

- Minimalistic object-oriented API written in C++
- Support of recent hardware via OpenGL 4.5 and Almost Zero Drive Overhead (AZDO) functions
- Multi-pipeline (forward, deferred and OIT rendering)
- Cross-platform (Windows, Linux)
- Dynamic scene graph management
- Dynamic lighting
- Dynamic soft shadows
- Physics and collision detection (via Bullet)
- Physically-Based Rendering (PBR)
- Post-processing (AA, HDR, bloom lighting)
- Skinning and animations
- Particle emitters

- Loading of scenes directly exported from 3D Studio MAX and Blender through custom plugins
- Support for arbitrary multiscreen projections
- Support for VR via OpenVR

Now we'll integrate in our already implemented PBR pipeline, image based lighting to make more realistic environment based reflections.

Physically based rendering

Introduction to PBR

We will treat this argument as first because all formulas used in IBL are based on physically based rendering.[3]
PBR is a collection of render techniques that are more or less based on the same underlying theory that more closely matches that of the physical world.
As physically based rendering aims to mimic light in a physically plausible way, it generally looks more realistic compared to our original lighting algorithms like Phong and Blinn-Phong. Not only does it look better, as it closely approximates actual physics, we (and especially the artists) can author surface materials based on physical parameters without having to resort to cheap hacks and tweaks to make the lighting look right.
One of the bigger advantages of authoring materials based on physical parameters is that these materials will look correct regardless of lighting conditions; something that is not true in non-PBR pipelines.
A material to be PBR must:[10] [8]

- Be based on the microfacet surface model.
- Conserve energy.
- Use a physically based BRDF.

Microfacet surface model

The theory describes that any surface at a microscopic scale can be described by tiny little perfectly reflective mirrors called microfacets.
The rougher a surface is, the more chaotically aligned each microfacet will be along the surface.
The effect of these tiny-like mirror alignments is, that when specifically talking about specular lighting/reflection, the incoming light rays are more likely to scatter along completely different directions on rougher surfaces, resulting in a more widespread specular reflection.
In contrast, on a smooth surface the light rays are more likely to reflect in roughly the same

direction, giving us smaller and sharper reflections.[See figure 7]

Figure 7: Microfacets on rough and smooth surface (source: [3])

Based on the roughness of a surface, we can calculate the ratio of microfacets roughly aligned to some vector h.
This vector h is the halfway vector that sits halfway between the light l and view v vector.

$$h = \frac{l + v}{||l + v||}$$

We can see that higher roughness values display a much larger specular reflection shape, in contrast with the smaller and sharper specular reflection shape of smooth surfaces.[See figure 8]

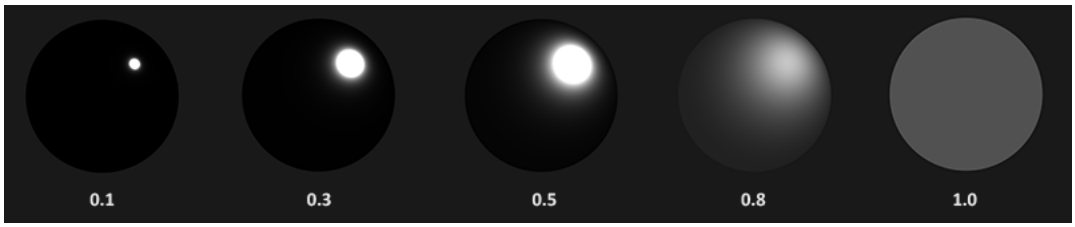

Figure 8: Sphere roughness (source: [3])

Energy conservation

The microfacet approximation employs a form of energy conservation: outgoing light energy should never exceed the incoming light energy (excluding emissive surfaces).
If we look at the [See figure 8] we notice that the specular reflection area increase, but also its brightness decrease at increasing roughness levels, and if we have the the same specular intensity at each pixel the rougher surfaces would emit much more energy, violating the energy conservation principle.
To solve this problem we use refraction and reflection, so not all energy reflects, but scatters, so the light rays re-emerging out of the surface contribute to the surface's observed color.[See figure 9]

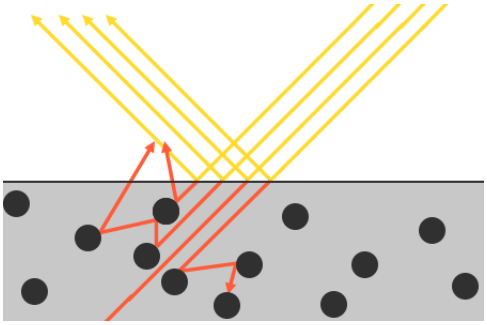

Figure 9: Surface reaction (source: [3])

There's a little difference with metallic materials, in fact metallic surfaces follow the same principles of reflection and refraction, but all refracted light gets directly absorbed without scattering.
This means metallic surfaces only leave reflected or specular light; metallic surfaces show no diffuse colors.
The energy conservation take in account another thing, that differentiate reflection and refraction, that's they're mutually exclusive, this means that whatever light energy gets reflected will no longer be absorbed by the material itself and can be simply explained by:

```
float kS = calculateSpecularComponent(...); // reflection/specular fraction
float kD = 1.0 - kS;                        // refraction/diffuse  fraction
```

This way we know both the amount the incoming light reflects and the amount the incoming light refracts, while adhering to the energy conservation principle.
Given this approach, it is impossible for both the refracted/diffuse and reflected/specular contribution to exceed 1.0, thus ensuring the sum of their energy never exceeds the incoming light energy.
So we get the render equation, precisely a specialized version that's called reflectance equation that permits us to describe how the light behaves based on material:

$$L_o(p, \omega_o) = \int_{\Omega} f_r(p, \omega_i, \omega_o) L_i(p, \omega_i) n \cdot \omega_i d\omega_i$$

The equation is based on 3 physical quantities:

- Radiant flux
- Solid angle
- Radiant intensity

Joining all these physical quantities we get the radiance, that's calculated by the total observed energy of an area A over a solid angle ω of a light intensity Φ:[See figure 10]

$$L = \frac{d^2\Phi}{dAd\omega\cos\theta}$$

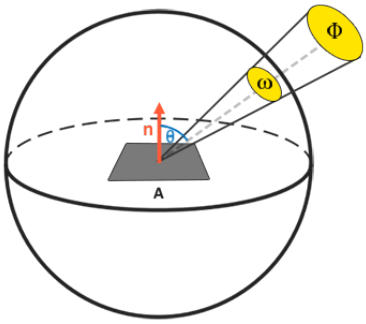

Figure 10: Radiance (source: [3])

Radiance is a radiometric measure of the amount of light in an area, scaled by the incident angle θ of the light to the surface's normal as $\cos\theta$: light is weaker the less it directly radiates onto the surface, and strongest when it is directly perpendicular to the surface.
So if we take care only of an infinitely small area A, we can use radiance to measure the flux of a single ray of light hitting a single point in space, pratically what happens when light hits a point p in our fragments.
Now we have all constructs to understand the reflectance equation because we calculate the sum of all the single lights(radiance) in the scene, therefore the irradiance:

$$L_o(p,\omega_o) = \int_{\Omega} f_r(p,\omega_i,\omega_o)L_i(p,\omega_i)n\cdot\omega_i d\omega_i$$

So L represents the radiance of some point p and some incoming infinitely small solid angle ω_i, that energy is scaled by $\cos\theta$ incident's angle to surface(it is defined by $n\cdot\omega_i$ formula part).
The reflectance equation calculates the sum of reflected radiance $L_o(p,\omega_o)$ of a point p in direction ω_o which is the outgoing direction to the viewer.
To calculate the total of values inside an area or a volume, we use an integral over all incoming directions $d\omega_i$ within the hemisphere Ω. We could think about the integral approximation through a Riemann sum.

BRDF

The BRDF, or bidirectional reflective distribution function, is a function that takes as input the incoming light direction ω_i, the outgoing view direction ω_o, the surface normal n, and a surface parameter a that represents the microsurface's roughness.
The BRDF approximates how much each individual light ray ω_i contributes to the final reflected light of an opaque surface given its material properties.
For instance, if the surface has a perfectly smooth surface (like a mirror) the BRDF function would return 0.0 for all incoming light rays ω_i except the one ray that has the same (reflected) angle as the outgoing ray ω_o at which the function returns 1.0.
BRDF approximates the material's reflective and refractive properties based on microfacet theory.
For a BRDF to be physically plausible it has to respect the law of energy conservation i.e. the sum of reflected light should never exceed the amount of incoming light.
There are many BRDF functions, but we use the Cook-Torrance BRDF:

$$f_r = k_d f_{lambert} + k_s f_{cook-torrance}$$

k_d is the ratio of incoming light energy that gets refracted with k_s being the ratio that gets reflected.
The $f_{lambert}$. This is known as Lambertian diffuse, which is a constant factor denoted as:

$$f_{lambert} = \frac{c}{\pi}$$

With c being the albedo or surface color.(That describes the diffuse part)
The $f_{cook-torrance}$ part, so the specular one, is described by the Epic Games implementation:

$$f_{CookTorrance} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}$$

The Cook-Torrance specular BRDF is composed three functions and a normalization factor in the denominator.
Each of the D, F and G symbols represent a type of function that approximates a specific part of the surface's reflective properties.
These are defined as the normal Distribution function, the Fresnel equation and the Geometry function.[11]

Normal distribution function

The normal distribution function D statistically approximates the relative surface area of microfacets exactly aligned to the halfway vector h.

That function statistically approximate the general alignment of the microfacets given some roughness parameter and the one we will be using is known as the Trowbridge-Reitz GGX:

$$NDF_{GGXTR}(n, h, \alpha) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2}$$

We can notice in [See figure 7] that when the roughness is low, so a smooth surface, a highly concentrated number of microfacets are aligned to halfway vectors over a small radius, so it looks white.
On a rough surface the microfacets are aligned in much more random directions, so the result looks more grayish.

Geometry function

The geometry function statistically approximates the relative surface area where its micro surface-details overshadow each other, causing light rays to be occluded.[See figure 11]

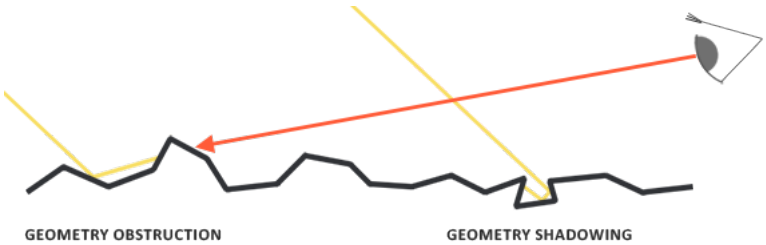

Figure 11: Geometry Shadowing (source: [3])

The geometry function we will use is a combination of the GGX and Schlick-Beckmann approximation known as Schlick-GGX:

$$G_{SchlickGGX}(n, v, k) = \frac{n \cdot v}{(n \cdot v)(1 - k) + k}$$

Here k is a remapping of α based on whether we are using the geometry function for either direct lighting or IBL lighting:

$$k_{direct} = \frac{(\alpha + 1)^2}{8}$$

$$k_{IBL} = \frac{\alpha^2}{2}$$

To effectively approximate the geometry we need to take account of both the view direction (geometry obstruction) and the light direction vector (geometry shadowing).

We can take both into account using Smith's method:

$$G(n, v, l, k) = G_{sub}(n, v, k)G_{sub}(n, l, k)$$

Here we can see an example where we variate the roughness R:[See figure 12]

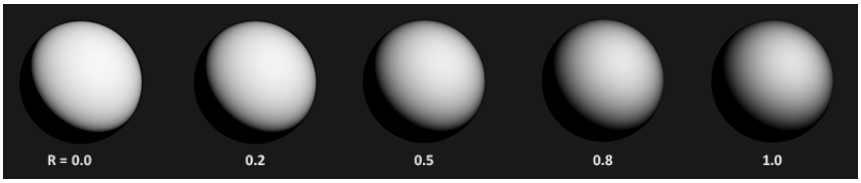

Figure 12: Geometry shadowing (source: [3])

Fresnel equation

The Fresnel equation describes the ratio of light that gets reflected over the light that gets refracted, which varies over the angle we are looking at a surface.
The moment light hits a surface, based on the surface-to-view angle, the Fresnel equation tells us the percentage of light that gets reflected.
From this ratio of reflection and the energy conservation principle we can directly obtain the refracted portion of light.

Every surface or material has a level of base reflectivity when looking straight at its surface, but when looking at the surface from an angle all reflections become more apparent compared to the surface's base reflectivity.
The Fresnel equation is approximated using the Fresnel-Schlick approximation:

$$F_{Schlick}(h, v, F_0) = F_0 + (1 - F_0)(1 - (h \cdot v))^5$$

F_0 is calculated using the indices of refraction.
As you can see on a sphere surface, the more we look towards the surface's grazing angles (90 degrees halfway-view angle), the stronger are the reflections:[See figure 13]

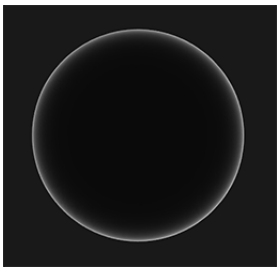

Figure 13: Fresnel reflection sphere (source: [3])

There are a few problems involved with the Fresnel equation. One is that the Fresnel-Schlick approximation is only really defined for dielectric or non-metal surfaces.
For metals, calculating the base reflectivity with indices of refraction doesn't properly hold and we need to use a different Fresnel equation, so we approximate by pre-computing the surface's response at normal incidence $F0$ at a 0 degree angle as if looking directly onto a surface.
We interpolate this value based on the view angle, as per the Fresnel-Schlick approximation, such that we can use the same equation for both metals and non-metals.

Final Cook-Torrence equation

If we put all together we get this equation:

$$L_o(p,\omega_o) = \int_{\Omega} (k_d \frac{c}{\pi} + k_s \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}) L_i(p,\omega_i) n \cdot \omega_i d\omega_i$$

But it is not completely correct in-fact F represents the ratio of light that gets reflected on a surface, that effectively is k_s, so it becomes:

$$L_o(p,\omega_o) = \int_{\Omega} (k_d \frac{c}{\pi} + \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}) L_i(p,\omega_i) n \cdot \omega_i d\omega_i$$

We left the pratical implementation of PBR to the engine, so we can focus on using these formulas on IBL's implementation.[12] [13]

Image based lighting

Introduction to IBL

IBL, or image based lighting, is a collection of techniques to light objects, not by direct analytical lights, but by treating the surrounding environment as one big light source.[14]
This is generally accomplished by manipulating a cubemap environment map (taken from the real world or generated from a 3D scene) such that we can directly use it in our lighting equations: treating each cubemap texel as a light emitter.
This way we can effectively capture an environment's global lighting and general feel, giving objects a better sense of belonging in their environment.
As image based lighting algorithms capture the lighting of some global environment, its input is considered a more precise form of ambient lighting.
The technique is combined to PBR to make materials more realistic.

Diffuse irradiance

Looking at the reflectance equation our main goal was to solve the integral of all incoming light directions w_i over the hemisphere Ω .
Solving the integral was easy as we knew beforehand the exact few light directions w_i that contributed to the integral.
This time however, every incoming light direction w_i from the surrounding environment could have some radiance making it more complex to solve.
This gives us two main requirements for solving the integral:

- Retrieve the scene's radiance given any direction vector w_i.
- Solving the integral needs to be fast and real-time.

Given a cubemap, we can visualize every texel of the cubemap as one single emitting light source.
By sampling this cubemap with any direction vector w_i, we retrieve the scene's radiance from that direction.

Solving the integral requires us to sample the environment map from not just one direction, but all possible directions w_i over the hemisphere Ω which is far too expensive for each fragment shader invocation.
To solve the integral in a more efficient way we will want to pre-process most of the computations.
So let's go deeper on the equation:

$$L_o(p,\omega_o) = \int_{\Omega} (k_d \frac{c}{\pi} + k_s \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}) L_i(p,\omega_i) n \cdot \omega_i d\omega_i$$

We can split the integral because the sum between k_d and ks is indipendent so it becomes:

$$L_o(p,\omega_o) = \int_{\Omega} (k_d \frac{c}{\pi}) L_i(p,\omega_i) n \cdot \omega_i d\omega_i + \int_{\Omega} (k_s \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}) L_i(p,\omega_i) n \cdot \omega_i d\omega_i$$

Now we have splitted the integral so we can focus on singular pieces of it and as first thing we will take in account the diffuse part(k_d).
We can simplify it by putting the constant pieces of the integral out of the equation:

$$L_o(p,\omega_o) = k_d \frac{c}{\pi} \int_{\Omega} L_i(p,\omega_i) n \cdot \omega_i d\omega_i$$

This gives us an integral that only depends on w_i (assuming p is at the center of the environment map).
With this knowledge, we can calculate or pre-compute a new cubemap that stores in each sample direction w_0 the diffuse integral's result by convolution.
So for every sample direction in the cubemap, we take all other sample directions over the hemisphere Ω into account.
To convolute an environment map we solve the integral for each output w_0 sample direction by discretely sampling a large number of directions w_i over the hemisphere Ω and averaging their radiance.
The hemisphere we build the sample directions w_i from is oriented towards the output w_0 sample direction we are convoluting.[See figure 14]

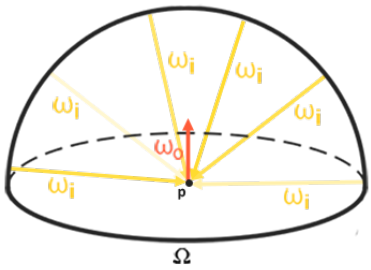

Figure 14: IBL hemisphere (source: [3])

This pre-computed cubemap, that for each sample direction w_0 stores the integral result, can be thought of as the pre-computed sum of all indirect diffuse light of the scene hitting some surface aligned along direction w_0.
Such a cubemap is known as an irradiance map seeing as the convoluted cubemap effectively allows us to directly sample the scene's irradiance from any direction.
Here's a look to a convoluted map:[See figure 15]

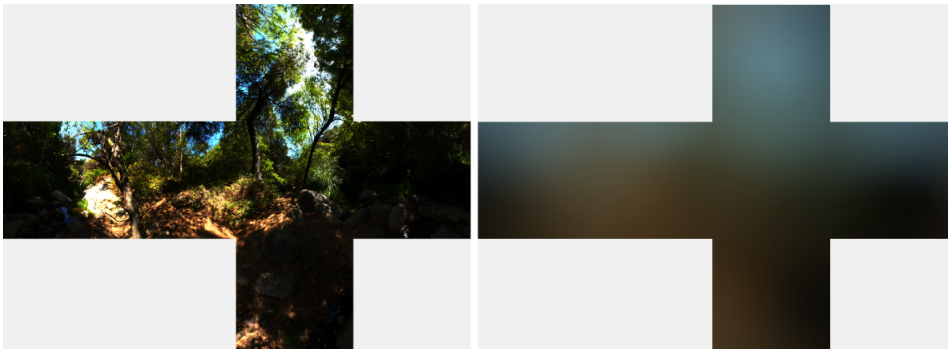

Figure 15: IBL convoluted map (Taken from Overvision engine)

Using PBR with HDR

As PBR bases most of its inputs on real physical properties and measurements it makes sense to closely match the incoming light values to their physical equivalents.
Without working in an HDR render environment it is impossible to correctly specify each light's relative intensity.
How does it all relate to image based lighting? Seeing as for image based lighting we base the environment's indirect light intensity on the color values of an environment cubemap we need some way to store the lighting's high dynamic range into an environment map.
The environment maps we have been using so far as cubemaps are in low dynamic range, so we directly used their color values from the individual face images, ranged between 0.0 and 1.0.
This may work fine for visual output, but with physical input parameters it is not going to work as well.
The radiance file format stores a full cubemap with all 6 faces as floating point data.
This allows us to specify color values outside the 0.0 to 1.0 range to give lights their correct color intensities.
The file format also uses a clever trick to store each floating point value, not as a 32 bit value per channel, but 8 bits per channel using the color's alpha channel as an exponent (this does come with a loss of precision).
This works quite well, but requires the parsing program to re-convert each color to their floating point equivalent.[See figure 16]

Figure 16: Cubemap equirectangular (source: [3])

This environment map is projected from a sphere onto a flat plane such that we can more easily store the environment into a single image known as an equirectangular map.
This does come with a small problem as most of the visual resolution is stored in the horizontal view direction, while less is preserved in the bottom and top directions, but it is a decent compromise.

Converting equirectangular to cubemap

To convert an equirectangular image into a cubemap we need to render a unit cube and project the equirectangular map on all of the cube's faces from the inside and take 6 images of each of the cube's sides as a cubemap face.
The vertex shader of this cube simply renders the cube as is and passes its local position to the fragment shader as a 3D sample vector:

```
...
localPos = aPos;
gl_Position =  projection * view * vec4(localPos, 1.0);
```

For the fragment shader, we color each part of the cube as if we neatly folded the equirectangular map onto each side of the cube.
To accomplish this, we take the fragment's sample direction as interpolated from the cube's local position and then use this direction vector and we convert from spherical to cartesian to sample the equirectangular map as if it is a cubemap itself.
We directly store the result onto the cube-face's fragment which should be all we need to do:

```
...
const vec2 invAtan = vec2(0.1591, 0.3183);
vec2 SampleSphericalMap(vec3 v)
{
    vec2 uv = vec2(atan(v.z, v.x), asin(v.y));
    uv *= invAtan;
    uv += 0.5;
    return uv;
}

void main()
{
    vec2 uv = SampleSphericalMap(normalize(localPos));
    vec3 color = texture(equirectangularMap, uv).rgb;

    FragColor = vec4(color, 1.0);
}
```

Now we have as parameter:

- Projection:

```
glm::perspective(glm::radians(90.0f), 1.0f, 0.1f, 10.0f)
```

Note that's important setting fov to 90 degrees so we make sure the viewing field is exactly large enough to fill a single face of the cubemap such that all faces align correctly to each other at the edges.

- View: The view is an array of 6 glm::lookat matrices that rotates in all 6 directions of the cube.
- Equirectangularmap: The equirectangular map is a texture2d given in input by the file.

Now we create a FBO that draws the cubemap and stores it in a 3d texture.
We do that by setting up 6 different view matrices (facing each side of the cube), set up a projection matrix with a fov of 90 degrees to capture the entire face, and render a cube 6 times storing the results in a floating point framebuffer.
Now using that texture is simple as using a normal cubemap, so we set it as usual in our skybox and we are almost done, the unique difference now is that we need to change a little bit the skybox code because we use HDR values on LDR framebuffer so we tonemap by applying gamma correction.[See figure 17]

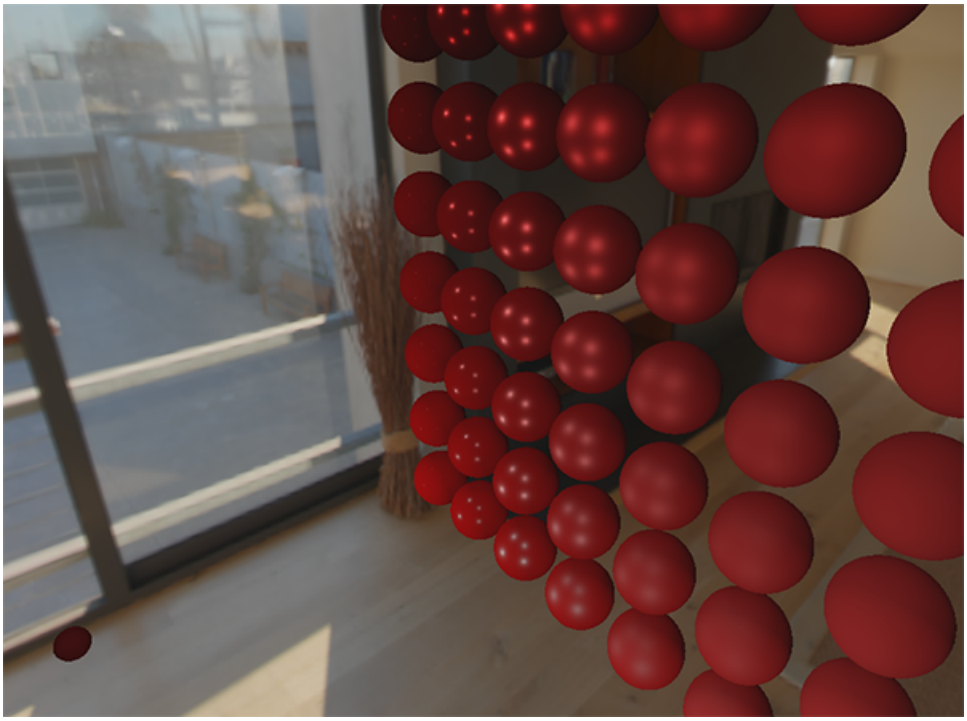

Figure 17: IBL equirectangular to cubemap (source: [3])

Cubemap convolution

To convolute the cubemap we need to solve the integral for all diffuse indirect lighting given the scene's irradiance in the form of a cubemap environment map.
We get the radiance of the scene through $L(p, w_i)$ in a particular direction by sampling an HDR environment map in direction w_i.
To solve the integral, we sample the scene's radiance from all possible directions within the hemisphere for each fragment.
To generate the irradiance map, we need to convolute the environment's lighting as converted to a cubemap.
Given that for each fragment the surface's hemisphere is oriented along the normal vector, convoluting a cubemap equals calculating the total averaged radiance of each direction w_i in the hemisphere oriented along N.
We will use a similar approach to the one we have used in equirectangular conversion, so we create an FBO that will contain the final cubemap, the code is similar, but with the difference that this time the fragment shader will draw the irradiance map.
We are going to generate a fixed amount of sample vectors for each cubemap texel along a hemisphere oriented around the sample direction and average the results.
The fixed amount of sample vectors will be uniformly spread inside the hemisphere.
To solve the integral we will discretely sample the function given by a fixed amount of sample vectors.
The integral of the reflectance equation revolves around the solid angle dw which is rather difficult to work with.
Instead of integrating over the solid angle dw we will integrate over its equivalent spherical coordinates θ and ϕ.[See figure 18]

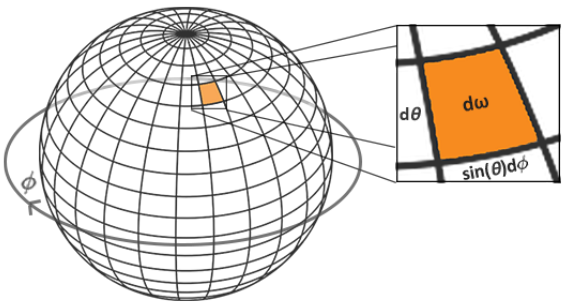

Figure 18: IBL spherical integration (source: [3])

We use the polar azimuth ϕ angle to sample around the ring of the hemisphere between 0 and 2π, and use the inclination zenith θ angle between 0 and 12π to sample the increasing rings of the hemisphere.
This will give us the updated reflectance integral:

$$L_o(p, \phi_o, \theta_o) = k_d \frac{c}{\pi} \int_{\phi=0}^{2\pi} \int_{\theta=0}^{\frac{1}{2}\pi} L_i(p, \phi_i, \theta_i) \cos(\theta) \sin(\theta) d\phi d\theta$$

Solving the integral requires us to take a fixed number of discrete samples within the hemisphere Ω and averaging their results.
This translates the integral to the following discrete version as based on the Riemann sum given $n1$ and $n2$ discrete samples on each spherical coordinate respectively:

$$L_o(p, \phi_o, \theta_o) = k_d \frac{c\pi}{n1n2} \sum_{\phi=0}^{n1} \sum_{\theta=0}^{n2} L_i(p, \phi_i, \theta_i) \cos(\theta) \sin(\theta) d\phi d\theta$$

As we sample both spherical values discretely, each sample will approximate or average an area on the hemisphere as the image before shows.
Note that the hemisphere's discrete sample area gets smaller the higher the zenith angle as the sample regions converge towards the center top.
To compensate for the smaller areas, we weigh its contribution by scaling the area by $\sin\theta$.
We can translate the code as follows:

```
vec3 irradiance = vec3(0.0);

vec3 up    = vec3(0.0, 1.0, 0.0);
vec3 right = normalize(cross(up, normal));
up         = normalize(cross(normal, right));

float sampleDelta = 0.025;
float nrSamples = 0.0;
for(float phi = 0.0; phi < 2.0 * PI; phi += sampleDelta)
{
    for(float theta = 0.0; theta < 0.5 * PI; theta += sampleDelta)
    {
        // spherical to cartesian (in tangent space)
        vec3 tangentSample = vec3(sin(theta) * cos(phi),  sin(theta)
        * sin(phi), cos(theta));
        // tangent space to world
        vec3 sampleVec = tangentSample.x * right + tangentSample.y
        * up + tangentSample.z * N;
```

```
        irradiance += texture(environmentMap, sampleVec).rgb *
        cos(theta) * sin(theta);
        nrSamples++;
    }
}
irradiance = PI * irradiance * (1.0 / float(nrSamples));
```

We specify a fixed sampleDelta delta value to traverse the hemisphere; decreasing or increasing the sample delta will increase or decrease the accuracy respectively.
From within both loops, we take both spherical coordinates to convert them to a 3D Cartesian sample vector, convert the sample from tangent to world space oriented around the normal, and use this sample vector to directly sample the HDR environment map.
We add each sample result to irradiance which at the end we divide by the total number of samples taken, giving us the average sampled irradiance.
Note that we scale the sampled color value by $\cos\theta$ due to the light being weaker at larger angles and by $\sin\theta$ to account for the smaller sample areas in the higher hemisphere areas.
The parameters view and projection that takes the vertex shader are the same as equirectangular's one, for the environmentMap we use the output of the equirectangular, so the cubemap of the environment.

Results of irradiance map

The irradiance map represents the diffuse part of the reflectance integral as accumulated from all surrounding indirect light.
Seeing as the light doesn't come from direct light sources, but from the surrounding environment, we treat both the diffuse and specular indirect lighting as the ambient lighting.
However, as the indirect lighting contains both a diffuse and specular part we need to weigh the diffuse part accordingly.
Similar to what we did in previously, we use the Fresnel equation to determine the surface's indirect reflectance ratio from which we derive the refractive ratio.
As the ambient light comes from all directions within the hemisphere oriented around the normal, there's no single halfway vector to determine the Fresnel response.
To still simulate Fresnel, we calculate the Fresnel from the angle between the normal and view vector.
However, earlier we used the micro-surface halfway vector, influenced by the roughness of the surface, as input to the Fresnel equation.
As we currently do not take roughness into account, the surface's reflective ratio will always end up relatively high.
Indirect light follows the same properties of direct light so we expect rougher surfaces to reflect less strongly on the surface edges.
Because of this, the indirect Fresnel reflection strength looks off on rough non-metal surfaces:[See figure 19]

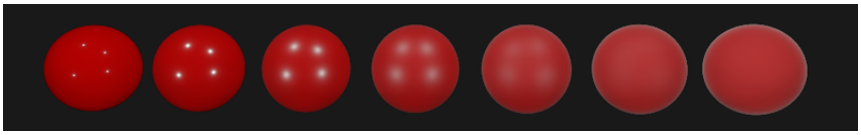

Figure 19: Fresnel without roughness parameter (source: [3])

We can alleviate the issue by injecting a roughness term in the Fresnel-Schlick equation:

```
F0 + (max(vec3(1.0 - roughness), F0) - F0) * pow(clamp(1.0 - cosTheta, 0.0, 1.0)
, 5.0);
```

So this could be a possible result by not using the ambient hardcoded light, but our calculated irradiance:[See figure 20]

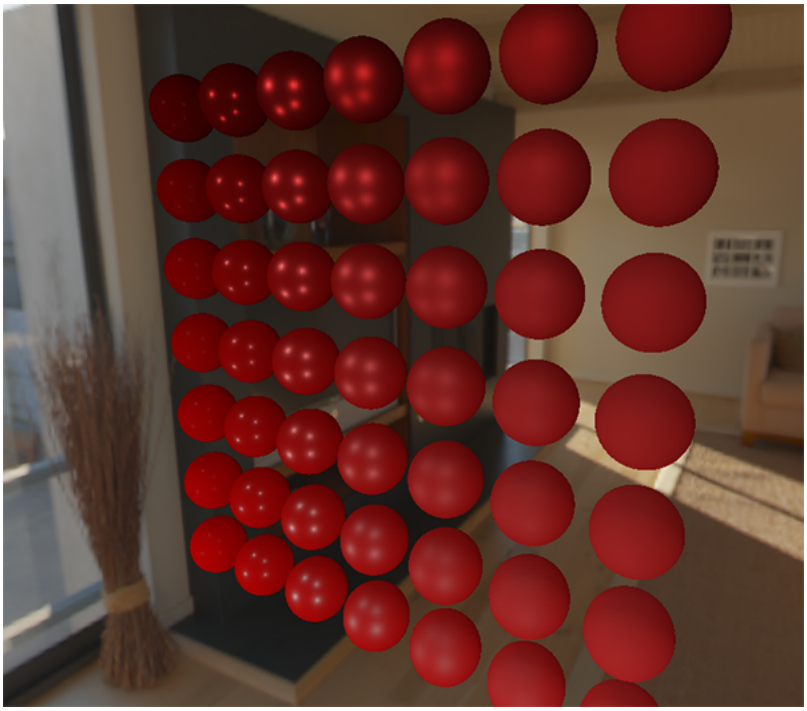

Figure 20: IBL irradiance result (source: [3])

Specular IBL

In this section we will focus on specular part of Cook-Torrence equation:

$$L_o(p,\omega_o) = \int_{\Omega} (k_d \frac{c}{\pi} + k_s \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}) L_i(p,\omega_i) n \cdot \omega_i d\omega_i$$

The Cook-Torrance specular portion (multiplied by k_s) isn't constant over the integral and is dependent on the incoming light direction, but also the incoming view direction.[15]
Trying to solve the integral for all incoming light directions including all possible view directions is a combinatorial overload and way too expensive to calculate on a real-time basis.
Epic Games proposed a solution where they were able to pre-convolute the specular part for real time purposes, given a few compromises, known as the split sum approximation.
The split sum approximation splits the specular part of the reflectance equation into two separate parts that we can individually convolute and later combine in the PBR shader for specular indirect image based lighting.
Similar to how we pre-convoluted the irradiance map, the split sum approximation requires an HDR environment map as its convolution input.
To understand the split sum approximation we will again look at the reflectance equation,

but this time focus on the specular part:

$$L_o(p,\omega_o) = \int_{\Omega} (k_s \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)} L_i(p,\omega_i) n \cdot \omega_i d\omega_i = \int_{\Omega} f_r(p,\omega_i,\omega_o) L_i(p,\omega_i) n \cdot \omega_i d\omega_i$$

For the same performance reasons as the irradiance convolution, we cannot solve the specular part of the integral in real time and expect a reasonable performance.
So preferably we'd pre-compute this integral to get something like a specular IBL map, sample this map with the fragment's normal, and be done with it.
We were able to pre-compute the irradiance map as the integral only depended on ω_i and we could move the constant diffuse albedo terms out of the integral.
This time, the integral depends on more than just ω_i as evident from the BRDF:

$$f_r(p,w_i,w_o) = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}$$

The integral also depends on w_0, and we cannot really sample a pre-computed cubemap with two direction vectors.
The position p is irrelevant here as described in the previous chapter.
Pre-computing this integral for every possible combination of ω_i and ω_o isn't practical in a real-time setting.
Epic Games' split sum approximation solves the issue by splitting the pre-computation into 2 individual parts that we can later combine to get the resulting pre-computed result we are after.
The split sum approximation splits the specular integral into two separate integrals:

$$L_o(p,\omega_o) = \int_{\Omega} L_i(p,\omega_i) d\omega_i * \int_{\Omega} f_r(p,\omega_i,\omega_o) n \cdot \omega_i d\omega_i$$

The first part is known as the pre-filtered environment map which is a pre-computed environment convolution map, but this time taking roughness into account.
For increasing roughness levels, the environment map is convoluted with more scattered sample vectors, creating blurrier reflections.
For each roughness level we convolute, we store the sequentially blurrier results in the pre-filtered map's mipmap levels.
For instance, a pre-filtered environment map storing the pre-convoluted result of 5 different roughness values in its 5 mipmap levels looks as follows:[See figure 21]

Figure 21: IBL prefiltered roughness map (source: [3])

We generate the sample vectors and their scattering amount using the normal distribution function (NDF) of the Cook-Torrance BRDF that takes as input both a normal and view direction.
As we do not know beforehand the view direction when convoluting the environment map, Epic Games makes a further approximation by assuming the view direction (and thus the specular reflection direction) to be equal to the output sample direction ω_i.
This way, the pre-filtered environment convolution doesn't need to be aware of the view direction.
This does mean we do not get nice grazing specular reflections when looking at specular surface reflections from an angle as seen in the image below; this is however generally considered an acceptable compromise:[See figure 22]

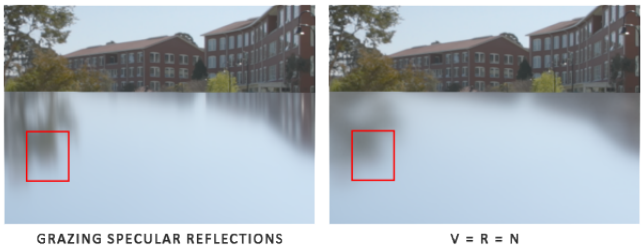

Figure 22: Specular reflection based on view (source: [3])

The second part of the split sum equation equals the BRDF part of the specular integral.
If we pretend the incoming radiance is completely white for every direction (thus $L(p, x) = 1.0$) we can pre-calculate the BRDF's response given an input roughness and an input angle between the normal and light direction ω_i, or $n * \omega_i$.
Epic Games stores the pre-computed BRDF's response to each normal and light direction combination on varying roughness values in a 2D lookup texture (LUT) known as the BRDF integration map.
The 2D lookup texture outputs a scale (red) and a bias value (green) to the surface's Fresnel response giving us the second part of the split specular integral:[See figure 23]

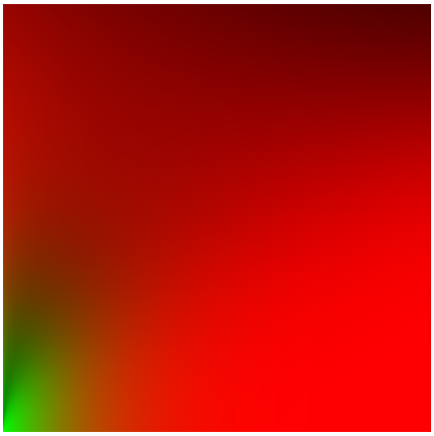

Figure 23: IBL BRDF lut (source: [3])

We generate the lookup texture by treating the horizontal texture coordinate (ranged between 0.0 and 1.0) of a plane as the BRDF's input $n * \omega_i$, and its vertical texture coordinate as the input roughness value.

Pre-filtering an HDR environment map

Pre-filtering an environment map is quite similar to how we convoluted an irradiance map. The difference being that we now account for roughness and store sequentially rougher reflections in the pre-filtered map's mip levels.
Previously we convoluted the environment map by generating sample vectors uniformly spread over the hemisphere using spherical coordinates.
While this works just fine for irradiance, for specular reflections it is less efficient.
When it comes to specular reflections, based on the roughness of a surface, the light reflects closely or roughly around a reflection vector r over a normal n, but (unless the surface is extremely rough) in every case around the reflection vector:[See figure 24]

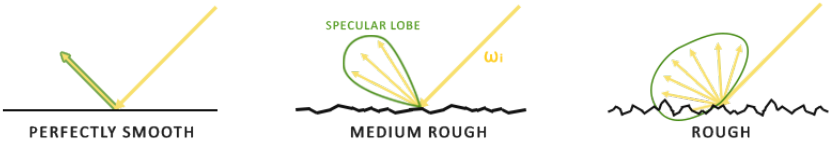

Figure 24: IBL specular lobe (source: [3])

The general shape of possible outgoing light reflections is known as the specular lobe.
As roughness increases, the specular lobe's size increases; and the shape of the specular lobe changes on varying incoming light directions.
The shape of the specular lobe is thus highly dependent on the material, so when it comes to the microsurface model, we can imagine the specular lobe as the reflection orientation about the microfacet halfway vectors given some incoming light direction, seeing as most light rays end up in a specular lobe reflected around the microfacet halfway vectors, it makes sense to generate the sample vectors in a similar fashion as most would otherwise be wasted, this process is known as importance sampling.

Pre-filtering an HDR environment map

To fully understand importance sampling we need to known Monte Carlo integration.
Monte Carlo integration revolves mostly around a combination of statistics and probability theory.
Monte Carlo helps us in discretely solving the problem of figuring out some statistic or value of a population without having to take all of the population into consideration.
The result will not be exact, but similar to the right one.
This is known as the law of large numbers.
Rather than solving an integral for all possible (theoretically infinite) sample values x, simply generate N sample values randomly picked from the total population and average.
When it comes to Monte Carlo integration, some samples may have a higher probability of being generated than others.
So far, in each of our cases of estimating an integral, the samples we have generated were uniform, having the exact same chance of being generated.
By default, each sample is completely (pseudo)random as we are used to, but by utilizing certain properties of semi-random sequences we can generate sample vectors that are still random, but have interesting properties.
For instance, we can do Monte Carlo integration on something called low-discrepancy sequences which still generate random samples, but each sample is more evenly distributed:[See figure 25]

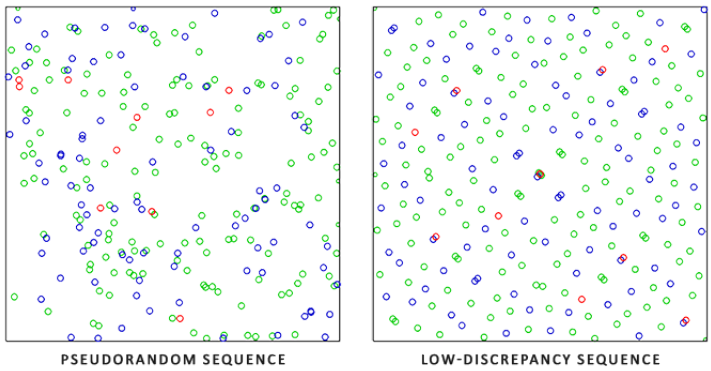

Figure 25: IBL low discrepancy (source: [3])

When using a low-discrepancy sequence for generating the Monte Carlo sample vectors, the process is known as Quasi-Monte Carlo integration.
Quasi-Monte Carlo methods have a faster rate of convergence which makes them interesting for performance heavy applications.

Given our newly obtained knowledge of Monte Carlo and Quasi-Monte Carlo integration,

there is an interesting property we can use for an even faster rate of convergence known as importance sampling.
When it comes to specular reflections of light, the reflected light vectors are constrained in a specular lobe with its size determined by the roughness of the surface.
Seeing as any (quasi-)randomly generated sample outside the specular lobe isn't relevant to the specular integral it makes sense to focus the sample generation to within the specular lobe, at the cost of making the Monte Carlo estimator biased.

This is in essence what importance sampling is about: generate sample vectors in some region constrained by the roughness oriented around the microfacet's halfway vector.
By combining Quasi-Monte Carlo sampling with a low-discrepancy sequence and biasing the sample vectors using importance sampling, we get a high rate of convergence.
Because we reach the solution at a faster rate, we will need significantly fewer samples to reach an approximation that is sufficient enough.
We will pre-compute the specular portion of the indirect reflectance equation using importance sampling given a random low-discrepancy sequence based on the Quasi-Monte Carlo method.
The sequence we will be using is known as the Hammersley Sequence as carefully described by Holger Dammertz.
The Hammersley sequence is based on the Van Der Corput sequence which mirrors a decimal binary representation around its decimal point.[16]

```
float RadicalInverse_VdC(uint bits)
{
    bits = (bits << 16u) | (bits >> 16u);
    bits = ((bits & 0x55555555u) << 1u) | ((bits & 0xAAAAAAAAu) >> 1u);
    bits = ((bits & 0x33333333u) << 2u) | ((bits & 0xCCCCCCCCu) >> 2u);
    bits = ((bits & 0x0F0F0F0Fu) << 4u) | ((bits & 0xF0F0F0F0u) >> 4u);
    bits = ((bits & 0x00FF00FFu) << 8u) | ((bits & 0xFF00FF00u) >> 8u);
    return float(bits) * 2.3283064365386963e-10; // / 0x100000000
}
// ------------------------------------------------------------------------------
vec2 Hammersley(uint i, uint N)
{
    return vec2(float(i)/float(N), RadicalInverse_VdC(i));
}
```

Instead of uniformly or randomly generating sample vectors over the integral's hemisphere, we will generate sample vectors biased towards the general reflection orientation of the microsurface halfway vector based on the surface's roughness.

The sampling process will be similar to what we have seen before: begin a large loop, generate a random (low-discrepancy) sequence value, take the sequence value to generate a sample vector in tangent space, transform to world space, and sample the scene's radiance. Additionally, to build a sample vector, we need some way of orienting and biasing the sample vector towards the specular lobe of some surface roughness.
We can take the NDF as described in the theory chapter and combine the GGX NDF in the spherical sample vector process as described by Epic Games:[17]

```
vec3 ImportanceSampleGGX(vec2 Xi, vec3 N, float roughness)
{
    float a = roughness*roughness;

    float phi = 2.0 * PI * Xi.x;
    float cosTheta = sqrt((1.0 - Xi.y) / (1.0 + (a*a - 1.0) * Xi.y));
    float sinTheta = sqrt(1.0 - cosTheta*cosTheta);

    // from spherical coordinates to cartesian coordinates
    vec3 H;
    H.x = cos(phi) * sinTheta;
    H.y = sin(phi) * sinTheta;
    H.z = cosTheta;

    // from tangent-space vector to world-space sample vector
    vec3 up        = abs(N.z) < 0.999 ? vec3(0.0, 0.0, 1.0) : vec3(1.0, 0.0, 0.0);
    vec3 tangent   = normalize(cross(up, N));
    vec3 bitangent = cross(N, tangent);

    vec3 sampleVec = tangent * H.x + bitangent * H.y + N * H.z;
    return normalize(sampleVec);
}
```

So we approximate the integral:

```
const uint SAMPLE_COUNT = 1024u;
float totalWeight = 0.0;
vec3 prefilteredColor = vec3(0.0);
for(uint i = 0u; i < SAMPLE_COUNT; ++i)
{
    vec2 Xi = Hammersley(i, SAMPLE_COUNT);
    vec3 H  = ImportanceSampleGGX(Xi, N, roughness);
```

```
    vec3 L  = normalize(2.0 * dot(V, H) * H - V);

    float NdotL = max(dot(N, L), 0.0);
    if(NdotL > 0.0)
    {
        prefilteredColor += texture(environmentMap, L).rgb * NdotL;
        totalWeight      += NdotL;
    }
}
prefilteredColor = prefilteredColor / totalWeight;
```

So once we get the prefilteredMap we can test it placing on the main cubemap, but will notice that are some artifacts:[See figure 26]

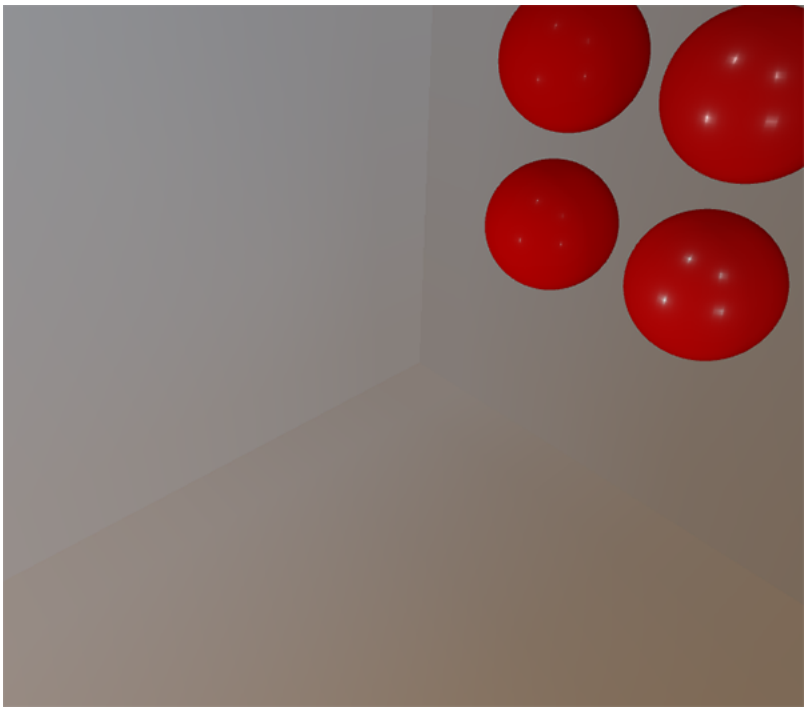

Figure 26: IBL prefilter artifacts (source: [3])

That's because sampling the pre-filter map on surfaces with a rough surface means sampling the pre-filter map on some of its lower mip levels.
When sampling cubemaps, OpenGL by default doesn't linearly interpolate across cubemap faces because the lower mip levels are both of a lower resolution and the pre-filter map is convoluted with a much larger sample lobe, the lack of between-cube-face filtering becomes quite apparent.
To solve this problem OpenGL gives us the option to properly filter across cubemap faces by enabling GL_TEXTURE_CUBE_MAP_SEAMLESS.
Another problem is an artifact that creates bright dots on the scene:[See figure 27]

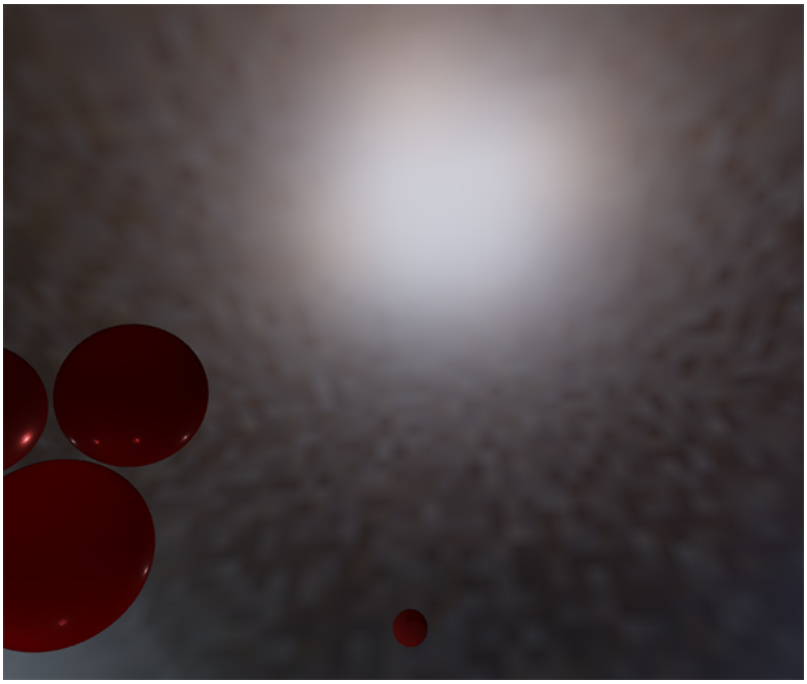

Figure 27: IBL prefilter dots (source: [3])

To reduce or remove completely the dots by (during the pre-filter convolution) not directly sampling the environment map, but sampling a mip level of the environment map based on the integral's PDF and the roughness:

```
float D   = DistributionGGX(NdotH, roughness);
float pdf = (D * NdotH / (4.0 * HdotV)) + 0.0001;
float resolution = 512.0; // resolution of source cubemap (per face)
float saTexel  = 4.0 * PI / (6.0 * resolution * resolution);
float saSample = 1.0 / (float(SAMPLE_COUNT) * pdf + 0.0001);
float mipLevel = roughness == 0.0 ? 0.0 : 0.5 * log2(saSample / saTexel);
```

Pre-computing the BRDF

In this section we can focus on the second part of the split-sum approximation: the BRDF. So let's look on the equation:

$$L_o(p,\omega_o) = \int_{\Omega} L_i(p,\omega_i)d\omega_i * \int_{\Omega} f_r(p,\omega_i,\omega_o)n \cdot \omega_i d\omega_i$$

we have pre-computed the left part of the split sum approximation in the pre-filter map over different roughness levels.
The right side requires us to convolute the BRDF equation over the angle $n * \omega_o$, the surface roughness, and Fresnel's $F0$.
This is similar to integrating the specular BRDF with a solid-white environment or a constant radiance L_i of 1.0.
Convoluting the BRDF over 3 variables is a bit much, but we can try to move $F0$ out of the specular BRDF equation:

$$\int_{\Omega} f_r(p,\omega_i,\omega_o)n \cdot \omega_i d\omega_i = \int_{\Omega} f_r(p,\omega_i,\omega_o)\frac{F(\omega_o,h)}{F(\omega_o,h)}n \cdot \omega_i d\omega_i$$

With F being the Fresnel equation.
Moving the Fresnel denominator to the BRDF gives us the following equivalent equation:

$$\int_{\Omega} \frac{f_r(p,\omega_i,\omega_o)}{F(\omega_o,h)}F(\omega_o,h)n \cdot \omega_i d\omega_i$$

Substituting the right-most F with the Fresnel-Schlick approximation gives us:

$$\int_{\Omega} \frac{f_r(p,\omega_i,\omega_o)}{F(\omega_o,h)}(F_0 + (1-F_0)(1-\omega_o \cdot h)^5)n \cdot \omega_i d\omega_i$$

Let's replace $(1-\omega_o \cdot h)^5$ by α to make it easier to solve for F_0:

$$\int_{\Omega} \frac{f_r(p,\omega_i,\omega_o)}{F(\omega_o,h)}(F_0 + 1 * \alpha - F_0 * \alpha)n \cdot \omega_i d\omega_i$$

$$\int_{\Omega} \frac{f_r(p,\omega_i,\omega_o)}{F(\omega_o,h)}(F_0 + 1 * \alpha - F_0 * \alpha)n \cdot \omega_i d\omega_i$$

$$\int_{\Omega} \frac{f_r(p,\omega_i,\omega_o)}{F(\omega_o,h)}(F_0 * (1-\alpha) + \alpha)n \cdot \omega_i d\omega_i$$

Then we split the Fresnel function F over two integrals:

$$\int_{\Omega} \frac{f_r(p, \omega_i, \omega_o)}{F(\omega_o, h)}(F_0 * (1-\alpha))n \cdot \omega_i d\omega_i + \int_{\Omega} \frac{f_r(p, \omega_i, \omega_o)}{F(\omega_o, h)}(\alpha)n \cdot \omega_i d\omega_i$$

This way, F_0 is constant over the integral and we can take F_0 out of the integral.
Next, we substitute back to its original form giving us the final split sum BRDF equation:

$$F_0 \int_{\Omega} f_r(p, \omega_i, \omega_o)(1-(1-\omega_o \cdot h)^5)n \cdot \omega_i d\omega_i + \int_{\Omega} f_r(p, \omega_i, \omega_o)(1-\omega_o \cdot h)^5 n \cdot \omega_i d\omega_i$$

The two resulting integrals represent a scale and a bias to F_0 respectively.
Note that as $f_r(p, \omega_i, \omega_o)$ already contains a term for F they both cancel out, removing F from f_r.

In a similar fashion to the earlier convoluted environment maps, we can convolute the BRDF equations on their inputs: the angle between n and ω_o, and the roughness.
We store the convoluted results in a 2D lookup texture (LUT) known as a BRDF integration map that we later use in our PBR lighting shader to get the final convoluted indirect specular result.
We take both the angle θ and the roughness as input, generate a sample vector with importance sampling, process it over the geometry and the derived Fresnel term of the BRDF, and output both a scale and a bias to F_0 for each sample, averaging them in the end.
The variable k has different interpretation on IBL in fact:

$$k_{direct} = \frac{(\alpha+1)^2}{8}$$

$$k_{IBL} = \frac{\alpha^2}{2}$$

Since the BRDF convolution is part of the specular IBL integral we will use kIBL for the Schlick-GGX geometry function.

Completing IBL

To complete IBL and integrate in PBR to the engine we get the indirect specular reflections of the surface by sampling the pre-filtered environment map using the reflection vector:

```
...
vec3 R = reflect(-V, N);

const float MAX_REFLECTION_LOD = 4.0;
vec3 prefilteredColor = textureLod(prefilterMap, R,  roughness
* MAX_REFLECTION_LOD).rgb;
...
```

Sample from the BRDF lookup texture given the material's roughness and the angle between the normal and view vector:

```
...
vec3 F        = FresnelSchlickRoughness(max(dot(N, V), 0.0), F0, roughness);
vec2 envBRDF  = texture(brdfLUT, vec2(max(dot(N, V), 0.0), roughness)).rg;
vec3 specular = prefilteredColor * (F * envBRDF.x + envBRDF.y);
...
```

And then we integrate it in our PBR part:

```
...
vec3 F = FresnelSchlickRoughness(max(dot(N, V), 0.0), F0, roughness);

vec3 kS = F;
vec3 kD = 1.0 - kS;
kD *= 1.0 - metallic;

vec3 irradiance = texture(irradianceMap, N).rgb;
vec3 diffuse    = irradiance * albedo;

const float MAX_REFLECTION_LOD = 4.0;
vec3 prefilteredColor = textureLod(prefilterMap, R,  roughness * MAX_REFLECTION_LOD).rgb;
vec2 envBRDF  = texture(brdfLUT, vec2(max(dot(N, V), 0.0), roughness)).rg;
vec3 specular = prefilteredColor * (F * envBRDF.x + envBRDF.y);

vec3 ambient = (kD * diffuse + specular) * ao;
...
```

Then we have that final result:[See figure 28]

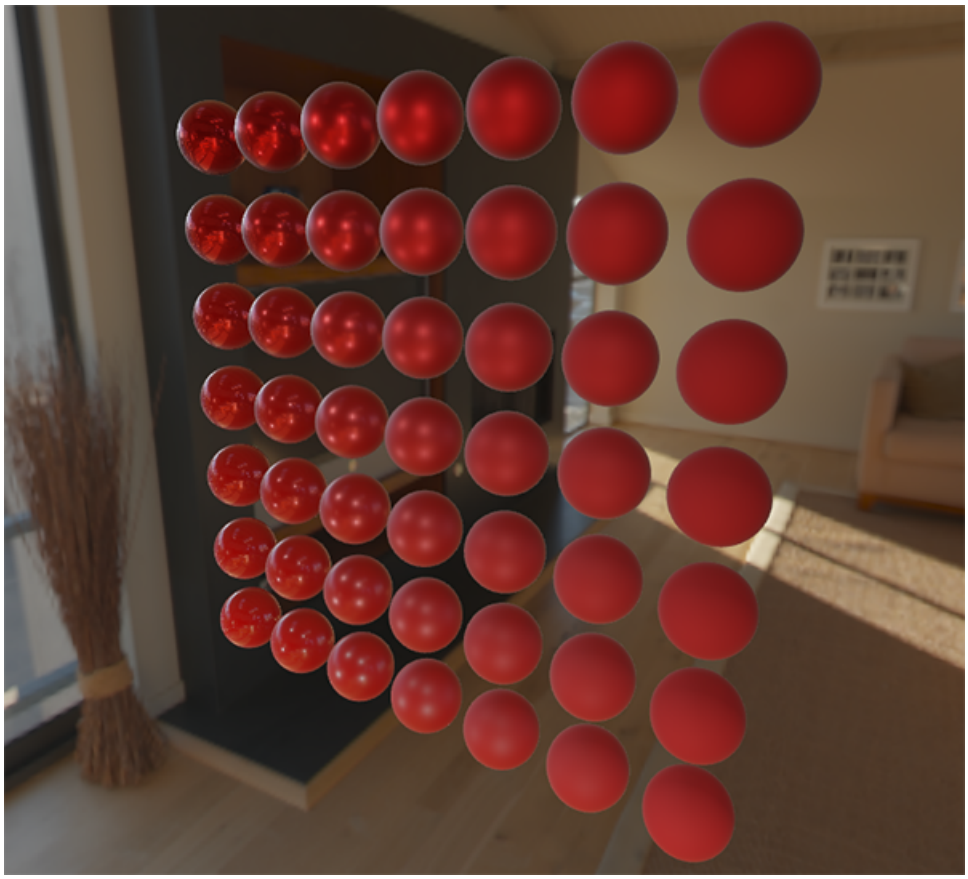

Figure 28: Final result

Implementation

i have implemented the IBL pipeline in the engine in that way:

- Read the file hdr and translate it into a OvTexture* type
- Convert it from equirectangular to cubemap
- Set the skybox
- Cache the BRDF,convolution,prefilter texture
- Set the bindless textures to engine
- Calculate on light shader file the final irradiance and the specular IBL

That scheme explains the code structure:[See figure 29]

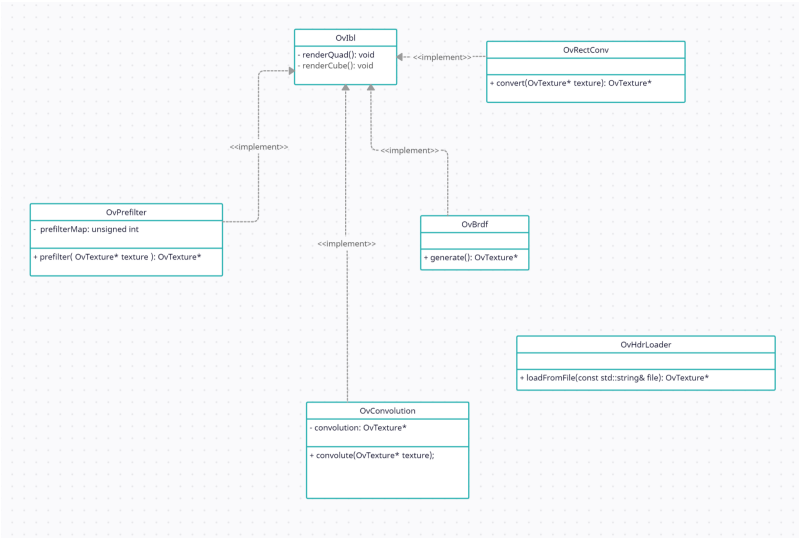

Figure 29: UML code scheme

As you can see almost all classes inherits from OvIbl that contains 2 methods that are used in all classes, these methods simply render a square in the scene and a cube, that's because almost all classes has a final cubemap where they write the result(except BRDF that has to draw in a square the texture).
Now let's go deeper in these classes:

- OvHdr requires the file where is the hdr texture and it loads it in a OvTexture*
- OvRectConv converts hdr textures in from equirectangular texture to a cubemap texture
- OvConvolution calculates the convolution texture of the cubemap

- OvPrefilter calculates the prefilter texture of the cubemap
- OvBrdf generates a BRDF texture

These classes outputs some textures that must be coherent with previous ones so we use a OpenGL debugger called Nsight.
Hdr texture:[See figure 30]

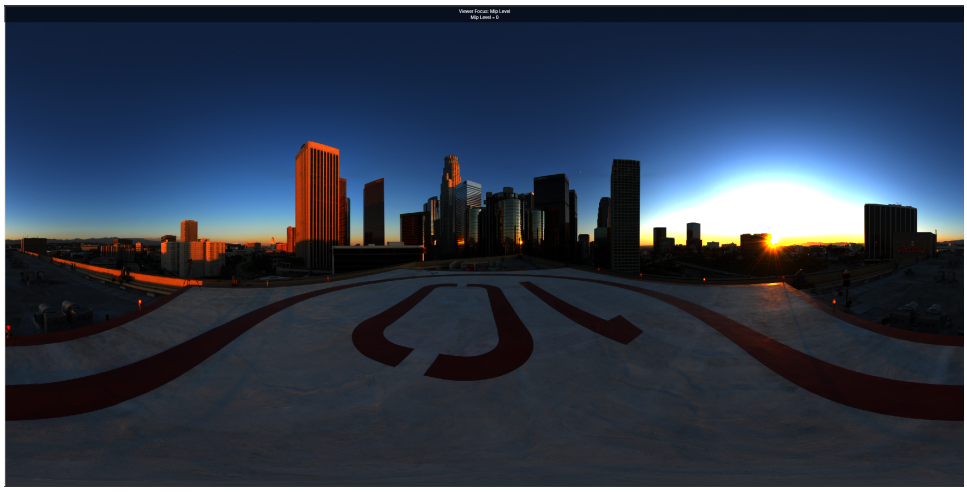

Figure 30: Hdr Texture

As we can see is a equirectangular texture.
Equirectangular to cubemap conversion:[See figure 31]

Figure 31: Cubemap conversion

The image is converted to cubemap correctly[27]

Convolution texture:[See figure 32]

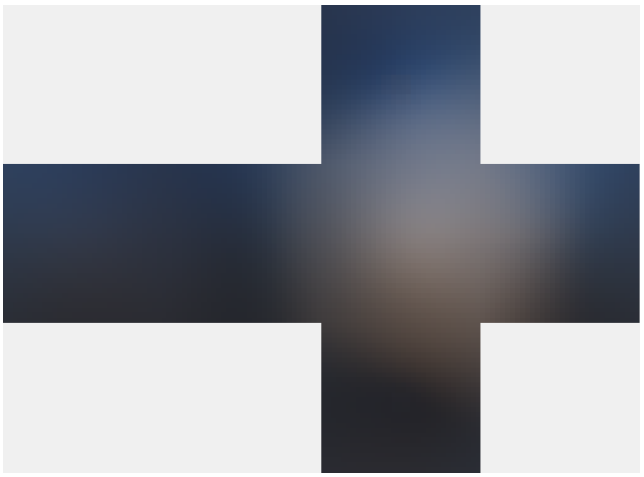

Figure 32: Convolution texture

The correct convolution texture is generated.
Prefilter texture:[See figure 33]

Figure 33: Prefilter texture

Here we see the lowest level of mipmap that means the one that has the lowest roughness.

Brdf texture:[See figure 34]

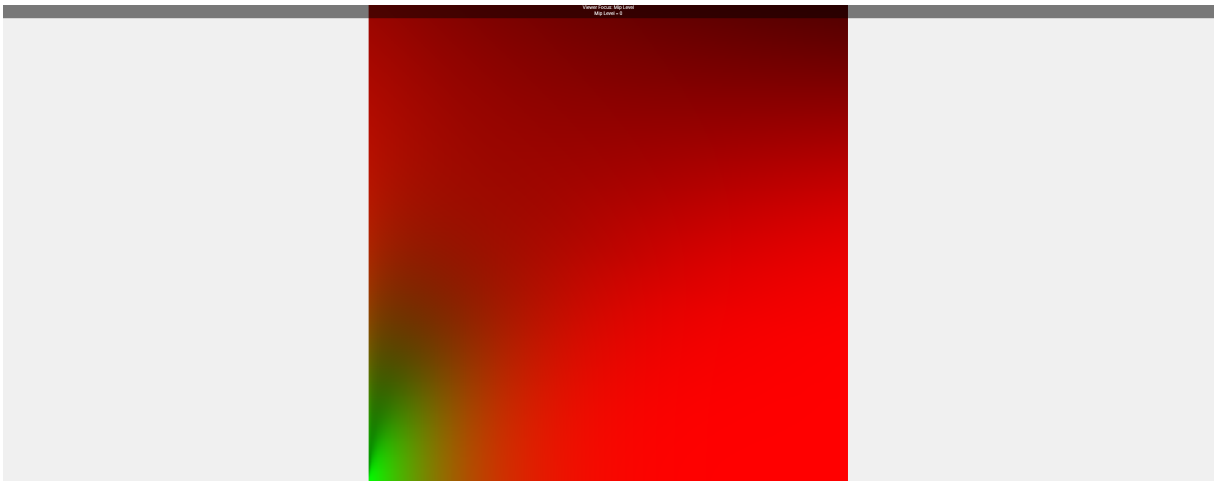

Figure 34: Brdf texture

The BRDF texture is generated exactly as [See figure 23].
Finally we get our scene with IBL implemented:[See figure 35][See figure 36]

Figure 35: Before and after the IBL implementation

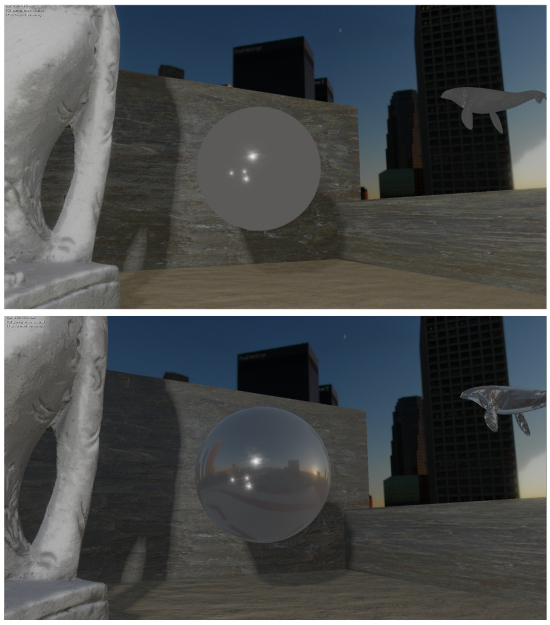

Figure 36: Before and after the IBL implementation

We can notice that with IBL the teapots looks more realistic.
The first line teapots has metalness, the second line doesn't have, going forward the roughness increases, so the reflections are rougher, the sphere is metallic and with 0 roughness so it reflects perfectly the environment.

Dynamic cubemaps

So we get the method to attach the cubemap to the surfaces, but how can we reflect perfectly the environment? Well there are many techniques, but we will use the dynamic cubemap one, that consists on render the whole scene 6 times, in a cubemap, that's a lot of work for our GPU so we will implement some optimizations that will make them become lighter.
We will take care only of forward rendering pipeline, and the scene will be rendered 6 times in a cubemap, so we will generate all IBL textures and then we will attach through shaders.
To render the scene 6 times i have implemented another class called OvReflect:[See figure 37]

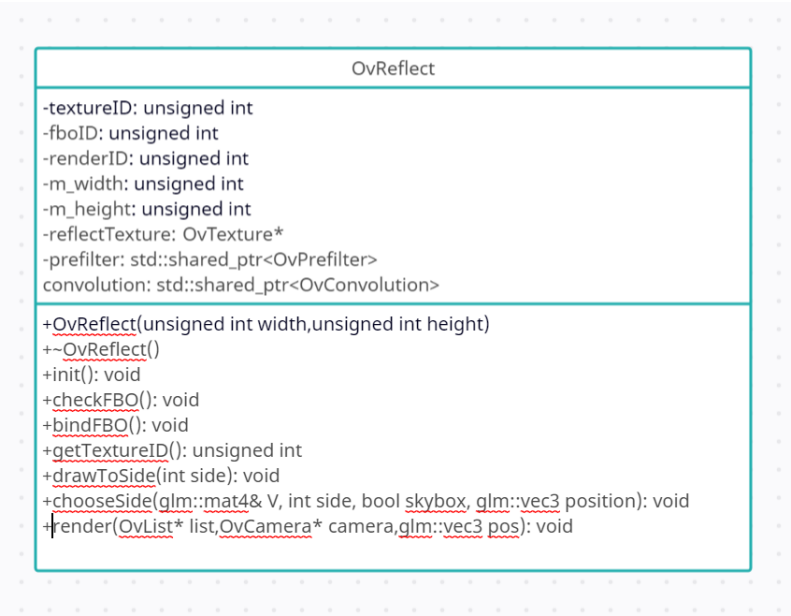

Figure 37: IBL OvReflect

The usage of this class is:

- Init the resolution
- Use forward rendering pipeline after the scene rendering, bind the fbo at the end
- Draw to side by using lookat in every side of the cube(similarly on how we have done in equirectangular part)
- Render the scene using the lookat of the side as camera matrix, so in the fbo
- Restore camera and projection matrix

Here's the results of a dynamic cubemap of resolution 512x512:[See figure 38]

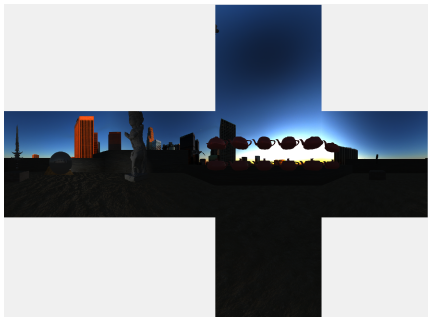

Figure 38: IBL Dynamic cubemap

Problems

This approach has some known problems:

- The result is blurry on low resolution
- The items when are dynamic so they moves, the reflection lags
- The cubemap is one and the far objects won't reflect well on it
- Parallax not corrected

The first two problems are solved by using right values of resolution(could be choose by the user), and the second one increasing the ratio of updates per second, but the third one is a little more tricky in fact we could create a cubemap per object in the scene, but is extremely expansive, because it would be $nr_{objects} * render\_time * 6$ that wastes a lot of resources, so looking at Rockstar Games solution we can precompute some environments and by the distance of the object to the cubemap position attach it to the object, that's a good solution in fact it is used in many games and newer engines.
The last problem is a parallax problem, we will go deeper:[18] We will divide cubemaps in two categories:

- Infinite cubemaps: These cubemaps are used as a representation of infinite distant lighting, they have no location.
  They can be generated with the game engine or authored by hand.
  They are perfect for representing low frequency lighting scene like outdoor lighting (i.e the light is rather smooth across the level) .
- Local cubemaps: These cubemaps have a location and represent finite environment lighting.
  They are mostly generating with game engine based on a sample location in the level.
  The generated lighting is only right at the location where the cubemap was generated, all other locations must be approximate.

More, as cubemap represent an infinite box by definition, there is parallax issue (Reflected objects are not at the right position) which require tricks to be compensated.
They are used for middle and high frequency lighting scene like indoor lighting.
The number of local cubemap required to match lighting condition of a scene increase with the lighting complexity (i.e if you have a lot of different lights affecting a scene, you need to sample the lighting at several location to be able to simulate the original lighting condition).

The common point of every parallax-correction techniques is to define an approximation of the geometry (we will call this geometry proxy) surrounding the local cubemap.
The simpler is the approximation, the more efficient will be the algorithm at the price of accuracy.
Example of geometry proxy are sphere volume , box volume or cube depth buffer.
In the shader, we perform an intersection between the reflection vector and the geometry proxy.
This intersection is then use to correct the original reflection vector to a new direction.
As the interaction must be performing in the shader, you can see how performance is linked to the choice of geometry proxy.
The hatched line is the reflecting ground and the yellow shape is the environment geometry.
A cubemap has been generated at position C, the camera is looking at the ground, the view vector reflected by the surface normal R is normally used to sample the cubemap.
Artists define an approximation of the geometry surrounding the cubemap using a box volume, that is the black rectangle in the figure.
It should be noted that the box center does not need to match the cubemap center.
We then find P, the intersection between vector R and the box volume.
We use vector CP as a new reflection vectorR' to sample the cubemap.[See figure 39]

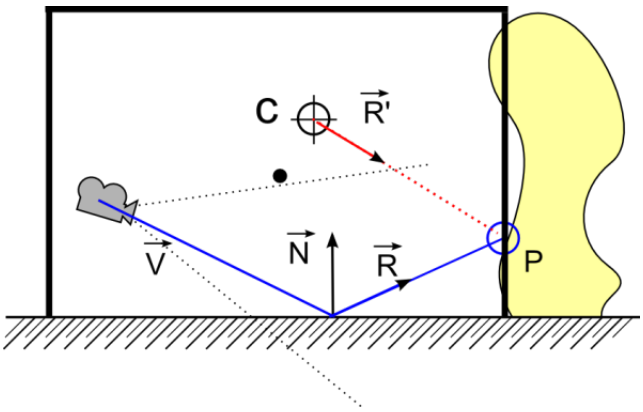

Figure 39: Parallax resolution

We have local cubemaps in our case and we solve it by calculating the reflection by the

viewer angle into a bounding box and the cubemap position with the following code:

```
...
float3 DirectionWS = PositionWS - CameraWS;
float3 ReflDirectionWS = reflect(DirectionWS, NormalWS);

// Following is the parallax-correction code
// Find the ray intersection with box plane
float3 FirstPlaneIntersect = (BoxMax - PositionWS) / ReflDirectionWS;
float3 SecondPlaneIntersect = (BoxMin - PositionWS) / ReflDirectionWS;
// Get the furthest of these intersections along the ray
// (Ok because x/0 give +inf and -x/0 give -inf )
float3 FurthestPlane = max(FirstPlaneIntersect, SecondPlaneIntersect);
// Find the closest far intersection
float Distance = min(min(FurthestPlane.x, FurthestPlane.y), FurthestPlane.z);

// Get the intersection position
float3 IntersectPositionWS = PositionWS + ReflDirectionWS * Distance;
// Get corrected reflection
ReflDirectionWS = IntersectPositionWS - CubemapPositionWS;
...
// End parallax-correction code
```

That problem is almost completely solved in our code, but it is longer than that, in fact with this method we solve a piece of environment and not the whole one, so we need to make other catching points.

Optimizations

Once done we have the whole scene in a cubemap of a particular position that's specified when we choose the side, then we create IBL cubemaps and attach them.
To have more efficient results the cubemap doesn't render shadows.
To optimize even more the rendering of the scene we call this method every 0.5 seconds and not for every frame, so the scene on the reflections will be updated every 0.5 seconds, but the performance gets benefits so much, in fact we get on a gtx 1060 6gb around 25 fps without that optimization and with that we have 110/120 fps stable so we have a great result.
Another Optimization implemented is to use a low cubemap resolution.
To optimize we can load lower lods and use a dual paraboloid map, like Rockstar games does, but the last one will be done in next implementations.

Results

In this chapter we'll talk about final results of the project, with a step-by-step image explanation.
As first thing we had the PBR environment without diffuse irradiance so we get this:[See figure 40]

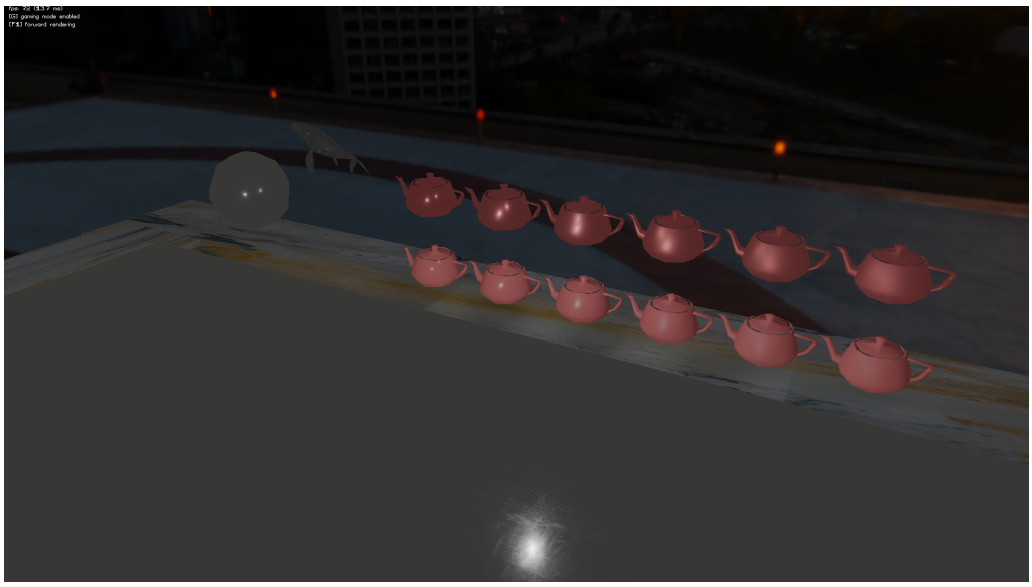

Figure 40: IBL without irradiance

It looks very metallic and greyish the floor, even the teapots has not a great visual effect. By adding the irradiance:[See figure 41]

Figure 41: IBL with irradiance

Here we see the ambient factor a little bit cleaner and it looks like as the environment, so orange/blue. Adding the specular:[See figure 42]

Figure 42: IBL with specular

Adding specular the environment looks a lot more realistic, with reflections of the skybox. Adding not corrected cubemaps:[See figure 43]

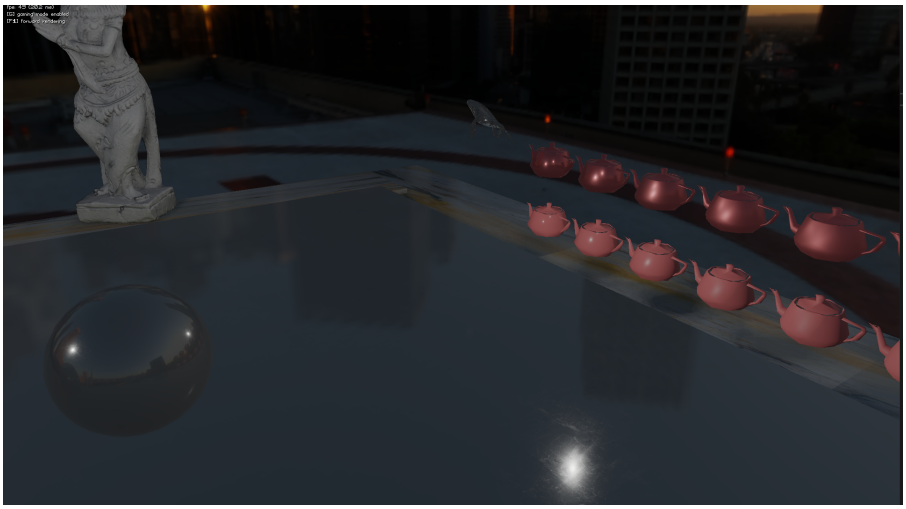

Figure 43: IBL without corrected cubemaps

We have added the environment mapping through dynamic cubemaps, and it looks great until we see the floor, that does not reflect the objects in the scene. Adding corrected cubemaps:[See figure 44]

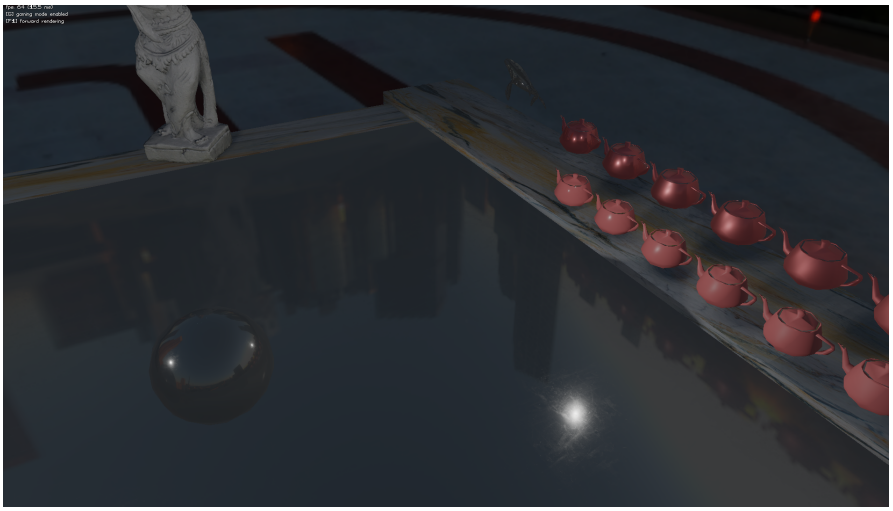

Figure 44: IBL corrected cubemaps

Finally we get corrected cubemaps and the environment has correct reflections, in fact we see teapots that reflects to the floor and the statue too.
And a closer look to another environment that has good reflections environment:[See figure 45]

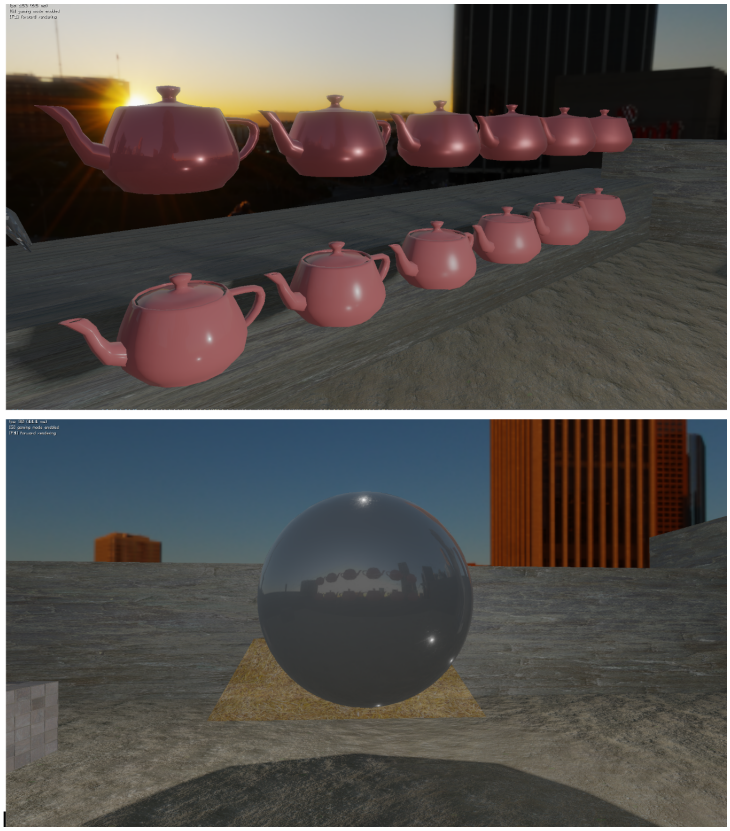

Figure 45: IBL dynamic cubemaps

Conclusions

We achieved successfully the IBL part combined with dynamic cubemaps implementations and completed the project.
The study of PBR and IBL comprehends complex maths, but step by step i have studied the logic behind and implemented in the current engine.
We used the learnopengl's approach to implement IBL, but there are many ways achieve that, some of them could be more expansive in terms of computational power or cheaper it depends on various engines.
So to recap a little bit we have as first implemented diffuse irradiance, and our environment has looked more realistic with an ambient more similar to the skybox cubemap, after that we had the specular part that gives us a way to reflect even in a better way, then we added dynamic cubemaps to catch the environment realtime and make reflections look better, finally we solved the cubemaps problem by correcting the parallax.
The parallax problem is a very common problem of this approach and many games had in past this problem, that's simply because by default, cubemaps are naively reflected on the ground and do not follow the player's perspective because of that, this causes an unrealistic-looking reflection for most surfaces so we approached as a possible solution to parallax-correct the cubemaps based on the player's camera position, using a custom shader and a bounding box trigger for a cubemap.
The dynamic cubemaps are used extensively in offline rendering and that's right because they are very heavy even on modern GPUs, so as Rockstar Games does for realtime applications would be better the usage of dual paraboloid maps.
Unreal engine 4 and 5 uses dynamic cubemaps extensively with a little bit of ray tracing to make them more realistic.
So these techniques are very useful nowadays and are extensively used in many engines that we have cited before.

Future work

We can even improve that implementation using: Dual paraboloid map so we can render the scene only 2 times as rockstar games dows, then store in files the precomputed maps so

we have faster loading time and sse reflection probes cubemap interpolation with parallax correction making almost perfect reflections.

Bibliography

[1] Dave Shreiner. “The Official Guide to Learning OpenGL”. In: *OpenGL Programming Guide*. 2013 (cit. on p. 9).

[2] Jr. Wright. “Comprehensive Tutorial and Reference”. In: *OpenGL Superbible*. 2015 (cit. on p. 9).

[4] Satheesh PV. “Master the basics of Unreal Engine 4 to build stunning video games”. In: *Unreal Engine 4 Game Development Essentials*. 2016 (cit. on p. 11).

[8] Joey de Vries. “Learn modern OpenGL graphics programming in a step-by-step fashion”. In: *Learn OpenGL*. 2018 (cit. on pp. 15, 19).

Sitography

[3] Joey de Vries. *"https://learnopengl.com/PBR/Theory"*. 2018 (cit. on pp. 9, 10, 19–22, 24, 25, 29, 30, 32, 33, 36, 37, 39–42, 45, 46).

[5] Adrian Courrèges. *"http://www.adriancourreges.com/blog/2015/11/02/gta-v-graphics-study/"*. 2013 (cit. on pp. 13, 14).

[6] Sebastien Lagarde. *"https://seblagarde.files.wordpress.com/2015/07/course_notes_moving_frostbite_to_pbr_v32.pdf"*. 2014 (cit. on p. 13).

[7] *"https://docs.blender.org/manual/en/latest/render/cycles/index.html"*. 2022 (cit. on p. 13).

[9] *"https://cgi.tutsplus.com/tutorials/hdr-image-based-lighting-in-blender-in-60-seconds–cms-31266"*. 2018 (cit. on p. 16).

[10] Naty Hoffmann. *"https://blog.selfshadow.com/publications/s2013-shading-course/hoffman/s2013_pbs_physics_math_notes.pdf"*. 2013 (cit. on p. 19).

[11] Brian Karis. *"https://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf"*. 2015 (cit. on p. 23).

[12] knarkowicz. *"https://www.shadertoy.com/view/4sSfzK"*. 2015 (cit. on p. 26).

[13] Marmoset. *"https://www.marmoset.co/toolbag/learn/pbr-theory"*. 2022 (cit. on p. 26).

[14] Joey de Vries. *"https://learnopengl.com/PBR/IBL/Diffuse-irradiance"*. 2018 (cit. on p. 27).

[15] Joey de Vries. *"https://learnopengl.com/PBR/IBL/Specular-IBL"*. 2018 (cit. on p. 37).

[16] Chetan Jags. *"https://chetanjags.wordpress.com/2015/08/26/image-based-lighting/"*. 2015 (cit. on p. 43).

[17] Trent Reed. *"http://www.trentreed.net/blog/physically-based-shading-and-image-based-lighting/"*. 2015 (cit. on p. 44).

[18] Sebastien Lagarde. *"https://seblagarde.wordpress.com/2012/09/29/image-based-lighting-approaches-and-parallax-corrected-cubemap/"*. 2018 (cit. on p. 57).