ISBN 978-987-4417-90-9

NATUS. A Physically-Based Rendering Engine with Real-Time Global Illumination

Ignacio del Barrio[1], María Luján Ganuza[1,2], and Silvia M. Castro[1,2]

[1] VyGLab Research Laboratory, DCIC, Universidad Nacional del Sur,
ignacioa.del.barrio@gmail.com, {mlg, smc}@cs.uns.edu.ar
http://vyglab.cs.uns.edu.ar
[2] Inst. for Computer Science and Engineering, ICIC (CONICET-UNS),
San Andrés 800, 8000 Bahía Blanca, Argentina

Abstract. Real-time rendering applications are difficult software engineering development projects, due to the complexity of the implemented algorithms required to achieve interactive frame rates. To provide a faster development process, the rendering engines implement many of these algorithms to support developers. In this article, we present a real-time physically-based rendering engine supporting global illumination. It supports diffuse indirect illumination, glossy reflections, refractions (transparency), soft shadows, and light emitted from emissive surfaces at interactive frame rates.

Keywords: physically-based rendering, real-time rendering, rendering engine, global illumination, directx, cone tracing, voxels

1 Introduction

Real-time rendering is the process of synthesizing three-dimensional data as images on a computer at interactive frame rates [1]. For high-quality virtual rendered scenes, Global Illumination (GI) techniques are required to simulate the light exchanges of indirect illumination in those scenes. While these techniques allow a lot of realism to be added, calculating that GI, in real-time, is a difficult challenge.

Traditionally, indirect illumination has been too costly to compute under real-time constraints. Like most real-time rendering applications, access to the graphic accelerator cards (GPUs) is required. Rendering engines provide an abstraction layer on top of the graphic APIs and, in most cases, extra features that speed up the development of graphic applications. These characteristics should facilitate the engine to synthesize high-quality images with GI. By adhering to Physically-Based Rendering (PBR) models, a more accurate representation of how light interacts with surfaces could be provided.

In this paper, we present NATUS, an advanced rendering engine to assist developers in the design of realistic graphic applications. This is based on a PBR model, that allows users to easily configure the material appearance of the objects in their scenes, being assured that these will behave correctly under different lighting conditions. To achieve a high level of realism, the engine also provides a GI model. Its implementation allows dynamic scene objects and light sources, that would affect the final result of the indirect illumination.

Its architecture makes it easily extensible. New scenes can be created and the rendering pipeline modified. NATUS Engine is implemented in C++ and executable demos are available at `http://natus.io/project/natus-engine`.

2 Related Work

In the last couple of years several hybrid approaches have been developed, many of which are executed on the GPU, and have been implemented in commercial graphic engines like Unreal, CryEngine and Unity3D. Unreal Engine 4 [2] is the current iteration of one of the first major game engines to have come out to the public [3]. It uses DirectX, OpenGL, as well as WebGL. Unreal Engine 4 supports Screen Space Global Illumination (SSGI), a feature that aims to create natural-looking lighting by adding dynamic indirect lighting to objects within the screen view. CryEngine is another industry-level game engine [4]. It is known to produce state-of-the-art graphics and performance and supports Voxel-Based Global Illumination and soft shadows. Unity 3D [5] uses also a middleware for Realtime GI. At the moment, this solution is deprecated and Unity 3D is developing a new solution for real-time global illumination [6].

Most of the mentioned Engines require desktop GPUs with large amounts of memory and are suited for high-end desktop systems. But despite these limitations, it's thanks to the use of simplified reconstructions of the 3D scene coupled with clever algorithms and optimizations, that simulation of global illumination is practicable in the real-time graphic applications of today.

3 NATUS Rendering Engine Architecture

Like most engines, our system is built in layers [7]. Its different functionalities are carefully partitioned by topic and level of complexity. The upper layers of the system depend on the functionalities defined on the lower layers to create a more sophisticated and complex set of features. An overview of the architecture is shown in Fig. 1, and the details are described below.

Middleware

The middleware is the lowest layer of the system architecture and provides the interface to the GPU via the DirectX 12 API. It contains all the third party libraries and APIs that make up the foundation of a windows-based rendering engine. Two of the most important programming interfaces are the DirectX 12 API and the Win32 API [8]. In addition, the application supports the libraries Tiny OBJ [9], to load 3D Meshes from files, and Dear ImGUI [10], for the construction of the application UI.

Low-Level Renderer

The low-level renderer manages the drawing of the primitives by communicating with the graphics API. At this level, the design is focused on rendering a collection of *render items* as quickly as possible, without much regard to visibility issues. According to Gregory [7], this renderer should be completely agnostic as to the type of spatial subdivision or scene graphic used. This allows for the design of a system that is specifically suited to the needs of different types of applications. This module is integrated by six components.

The *Graphics Device Interface* main task is to initialize DirectX (configure the rendering pipeline, prepare the render buffers, etc.) for our engine framework and to manage the communication with the GPU.

The main components of a scene, *Cameras*, *Lights*, 3D objects and *Materials*, are defined in this engine layer. It supports two standard types of cameras available, a *Free Camera*, and a *Spherical Camera*, three different types of light sources, that is, point, spot and directional lights, and *Meshes* to represent

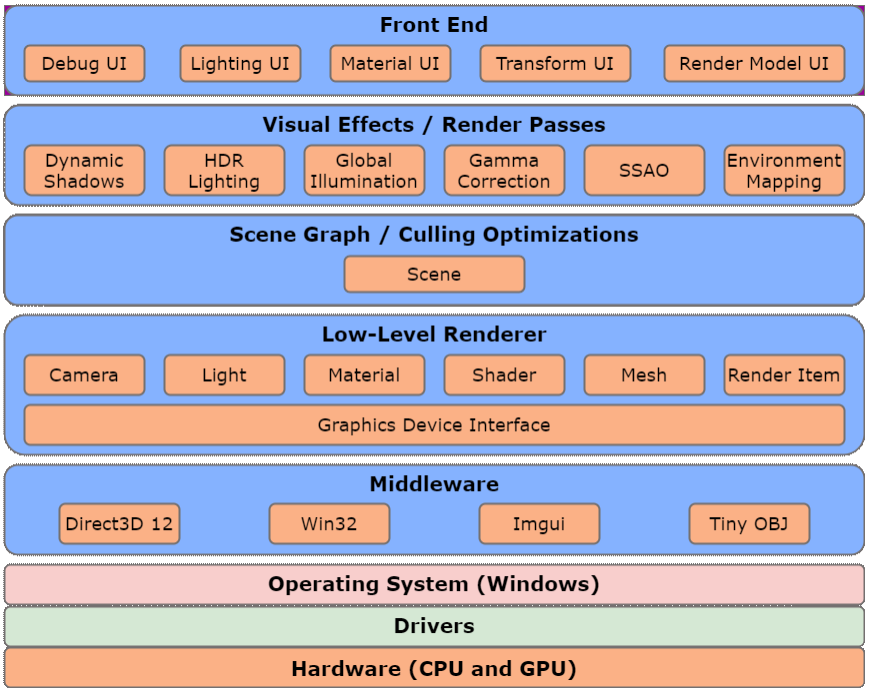


Fig. 1: The layered architecture of the rendering engine

the 3D objects. The *Materials* are lightweight components that describe how a surface should react under different light conditions, by including references to the texture resources it uses, color tints, and other physical properties. A detailed description of the material model is given in section 4. Materials only hold the attributes that describe the behavior given their interaction with light, but the actual computation of this behavior is done on the shaders.

The rendering engine supports four types of *Shaders*, that is, vertex, geometry, pixel, and compute. There is a predefined collection of shaders for different purposes, including texture filtering and clearing, geometry voxelization, voxels rendering, sky rendering ,and global illumination. Finally, the *Render Item* is a lightweight component that keep rendering information from each scene object, such as a reference to the mesh and the material objects, the transformation matrix, and so on, in order to submit a draw call to the graphics API.

Scene Graph

This layer manages which contents will be submitted to the low-level renderer based on some sort of visibility determination. The Scene class holds the scene information, that is, the main camera, collections of 3D Meshes, Render Items, textures and materials, and Light Sources. When the Rendering Engine starts up, it initializes the current scene instance by configuring the camera location followed by loading of textures and 3D Meshes, construction of materials, assembling of render items, and finalizing with lights configuration.

Visual Effects

This layer is where all of the advanced rendering effects that our engine is capable of simulating are implemented, based on the functionality provided by the low-level renderer. These effects can be enabled or disabled on the custom scenes defined by the programmer. The most relevant effects, *Ambient Occlu-*

sion (AO) and *Global Illumination (GI)*, are described in more detail in section 5. The engine also implements *Dynamic Shadows* to calculate shadows from directional light sources with the traditional shadow map scheme proposed by Williams[11]. In addition to these, it provides *Gamma Correction*, performed at the end of all lighting calculations to ensure that the values in the final image are properly gamma-corrected, and *Environment Mapping*, to simulate reflections and refractions coming from the distant background. NATUS also implements *HDR Lighting*, in order to have a wider range of color values to calculate the final illumination. Once finished, it transforms the HDR values back to the LDR using the Hable [12] algorithm.

Front End

This is the uppermost layer defined in the engine architecture, that is, the 2D user interface. To develop this interface we use a third party library for C++, Dear ImGUI. It is fast, portable, renderer agnostic, and has no external dependencies, particularly suited for integration in game engines and real-time 3D applications. The UI includes the *lighting panel*, the *material panel* and *transform panel*.

The *Lighting Panel* panel is the main control point for the engine lighting features and is divided into two sections, one for the GI settings and another for configuring the light sources in the scene. The *Lighting Panel* is shown in Fig. 2. The GI section has settings for the scene voxelization, different effects

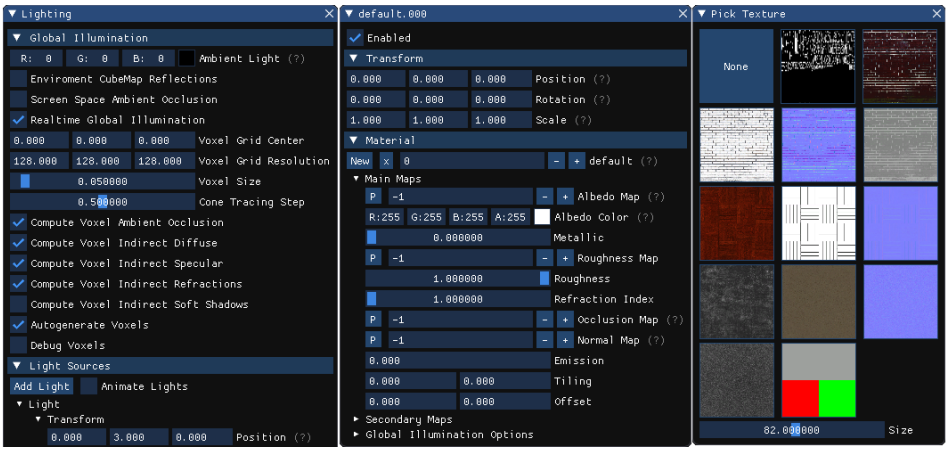


Fig. 2: The lighting panel (left). Material and transform panels (center and right).

computed via voxel cone tracing, environment reflections, and ambient light. The light sources section contains a list of light sources in the scene, whose properties can be modified via this panel. Lights can be added or removed from the scene through this panel. The *Material Panel* allows the user to control the appearance of objects in the scene, by manipulating the material properties of the selected object. This panel and its texture picking window are shown in Fig. 2. The *Transform Panel* controls the position, orientation and scale of objects, and lights in the scene relative to the world. It is shown in Fig. 2.

4 Physically-Based Materials

Our rendering engine materials follow a physically-based model implemented by our *standard shader*, which incorporates the advanced lighting algorithms and cal-

culations to simulate realistic surface lighting interaction. Physics-based rendering (PBR) is a methodology with no defined standard. The two most common workflows are metal/roughness and specular/glossiness. One of the first defined workflows, the metallic workflow, has been explored by Disney [13] and Adobe [14] and has been evolving, now emerging as a standard. We adopted this workflow, also used for real-time rendering by Epic Games in the Unreal Engine 4 [15]. This workflow is defined by a set of physical attributes, which are fed to the standard shader as textures or values. These attributes are diffuse albedo, metallic, and roughness:

- **Albedo.** The albedo or base color map is an RGB texture map that can contain the diffuse reflected color for dielectrics and the reflectance values for metals [16, 14]. It should not contain any lighting information. The alpha value of the albedo component color controls the transparency level for the material.
- **Metallic.** This map allows to define which areas of a material denote raw metal. As a grayscale map, it describes which areas in the base color should be interpreted as the reflected color (dielectric) and which ones as metal reflectance values, representing 1.0 (white) a raw metal.
- **Roughness.** This map describes the surface irregularities that cause light scattering. While rougher surfaces will have larger and dimmer-looking reflections, smoother surfaces will have concentrated specular reflections. On this map, white (1.0) represents a rough surface and black (0.0) a smooth one. The roughness map plays an integral part in the Cook-Torrance illumination model [1].

Other maps are usually attached. NATUS also supports *ambient occlusion (AO) map*, *Normal map* and *Emission* property. The *AO map* takes into account how much of the ambient environment lighting is accessible to a surface point. It only affects the diffuse contribution and should not occlude the specular contribution. In our implementation, the indirect diffuse lighting provided by the GI is multiplied by the AO. The *Normal map* is integrated into the PBR Material in order to simulate surface details. The *Emission* property controls the intensity of light emitted from the surface; when a material has a non-zero emission value it appears to be self-illuminated. During the real-time GI process, the emission value is used for the indirect diffuse lighting calculations to affect the illumination of nearby objects.

5 Voxel-Based Global Illumination

The voxel cone tracing is a GI technique introduced by Crassin et al. [17]. It calculates an approximation of the indirect lighting, generated by one or two bounces of light, for fully dynamic scenes in interactive applications, allowing both diffuse and glossy reflections with very realistic visual results. Instead of working on the actual geometry, a scene voxel representation is created and stored in the GPU. This representation can be created once for static geometry and per frame for dynamic objects. This algorithm is the central point of the GI Pipeline whose implementation is presented in Fig. 3.

Shadow Mapping The first step in the GI pipeline is to generate the corresponding shadow maps,that are going to be used later in the light voxelization step. We use *percentage closer filtering* [18], an improvement of the classic shadow mapping technique that allows smoothing the shadow edges. The map resolution is a trade off between speed and quality, and we use a screen resolution of 2048x2048 to produce an image without visual artifacts.

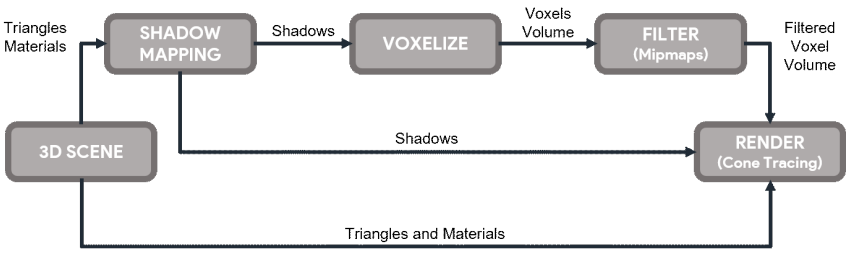


Fig. 3: Overview of the global illumination (GI) pipeline

Voxelization During the *voxelization step*, we use geometry shaders to output a voxelized representation of the lighting in the original scene geometry. We use the GPU hardware rasterizer to voxelize the scene geometry into a regular grid of voxels. Based on the Crassin et al. [17] approach, the voxel grid is stored in a 3D Texture. The scene illumination data is then saved into the grid voxels to effectively store the direct lighting from the scene. Each 3D texture voxel is represented by a single RGBA16 value; the alpha channel is used for opacity.

Filtering In order to perform voxel cone tracing of the 3D texture for full GI, it is necessary to filter it to generate a *mipmap chain*. The *filtering pass*, use the filtering GPU capabilities to create that mipmap. The 3D mipmap level generation is an extension of the 2D implementation by Nils Daumann [19].

Voxel Cone Tracing In this step, paths are traced through the filtered voxel structure in order to gather and approximate the indirect lighting for a point. A cone ray samples the volume on each step. The sampled volume is traced from the apex of a cone, located in a point p_0 on the object surface, and along its axis oriented in the desired direction (p_d) with an aperture angle θ. The sampling region is increased on every step, based on the diameter of the cone. This expansion of the volume being sampled is approximated by taking samples from different mipmap levels generated during the filtering stage; the traced distance t, allows deciding the mipmap level to sample [17]. The whole process of tracing cones is done entirely on the pixel shader. Opacity α_i and color c_i of each sample i from the voxel structure are integrated front-to-back along the cone to approximate the incoming indirect light at point p_0 and the corresponding occlusion value α.

Render The *illumination* at a point p combines the direct illumination, calculated using the physically-based Cook-Torrance BRDF, and the indirect diffuse illumination, reflections, and refractions from voxel cone tracing.

The *indirect illumination* at a surface point p is described by the hemisphere integral of the rendering general equation. To compute this integral efficiently,

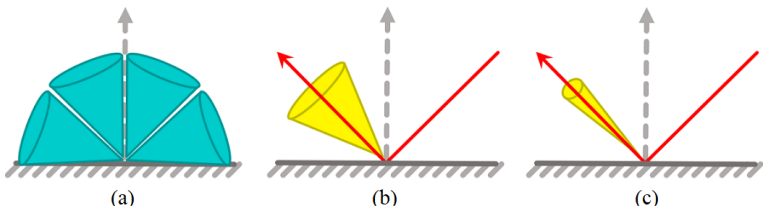
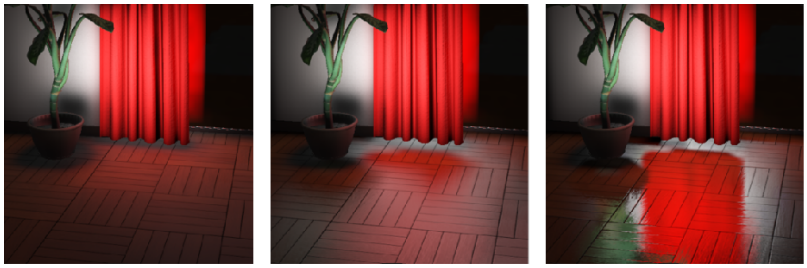


Fig. 4: Cone distributions approximate different phenomena. a) Multiple cones approximate indirect diffuse light. b) Wide cone approximates rough specular reflection. c) Narrow cone approximates fine specular reflections.

Crassin et al. [17] observed that the hemisphere can be partitioned into a sum of integrals, each of them gathering the indirect light from the scene for a cone. We use nine cones oriented over the hemisphere in order to gather all the indirect light from the scene for the diffuse reflection, as shown in Fig. 4. For the specular reflection and refraction we use a single cone in the reflected, or the refracted, direction, and its aperture is specified based on the surface material roughness. Specular cone apertures for different material roughness are shown in Fig. 4.



(a) (b) (c)

Fig. 5: Different material roughness on the same surface. (a) Completely rough surface with diffuse reflection only. (b) Slightly rough surface with diffuse and specular reflections. (c) Very smooth surface with high specular reflection.

The visual appearance of the cone tracing configurations are shown in Fig. 5. An advantage of voxel cone tracing is that the same cones distribution can be used to approximate the indirect diffuse reflection (combining color values), and the ambient occlusion (combining occlusion values). The accumulated occlusion value α returned from cone tracing represent the final occlusion value. Soft shadows can be casted by tracing a cone from each surface point p towards the light sources.

6 Evaluation

The application was tested on a mid-range desktop PC. We used a NVIDIA GeForce GTX 960 (2GB) GPU, an Intel Core i5 8400 CPU and 16GB of RAM.

The **default scene** used to evaluate our rendering engine is based on the classic Cornell Box scene. We placed a point light slightly above the center, illuminating the entire box. The objects placed inside have different geometrical complexity and their appearances were configured to showcase different material properties (Table 1).

Object	**Diffuse** RGBA format	**Metallic** (0.0-1.0)	**Roughness** (0.0-1.0)	**IOR** ≥ 0.0	**Emission** ≥ 0.0
Dragon	[1.0, 1.0, 1.0, 1.0]	1.0	0.1	1.0	0.0
Buddha	[0.0, 0.0, 0.0, 0.0]	0.0	0.1	3.0	0.0
Suzanne	[1.0, 1.0, 1.0, 1.0]	0.0	0.1	1.0	0.0
Teapot	[0.0, 0.5, 0.0, 1.0]	0.0	1.0	1.0	0.0

Table 1: Material settings for each object in the test scene. IOR: Index Of Refraction.

The default scene was tested using a 128^3 voxel grid, and a 1024x768 screen resolution. We tested the following lighting settings, with whom the scenes in Fig. 6 were rendered:

- **Test 1.** Only direct illumination (No GI).
- **Test 2.** GI, including reflections and refractions.
- **Test 3.** GI and soft shadows.
- **Test 4.** GI, soft shadows and ambient occlusion.
- **Test 5.** GI, with only emissive wall.

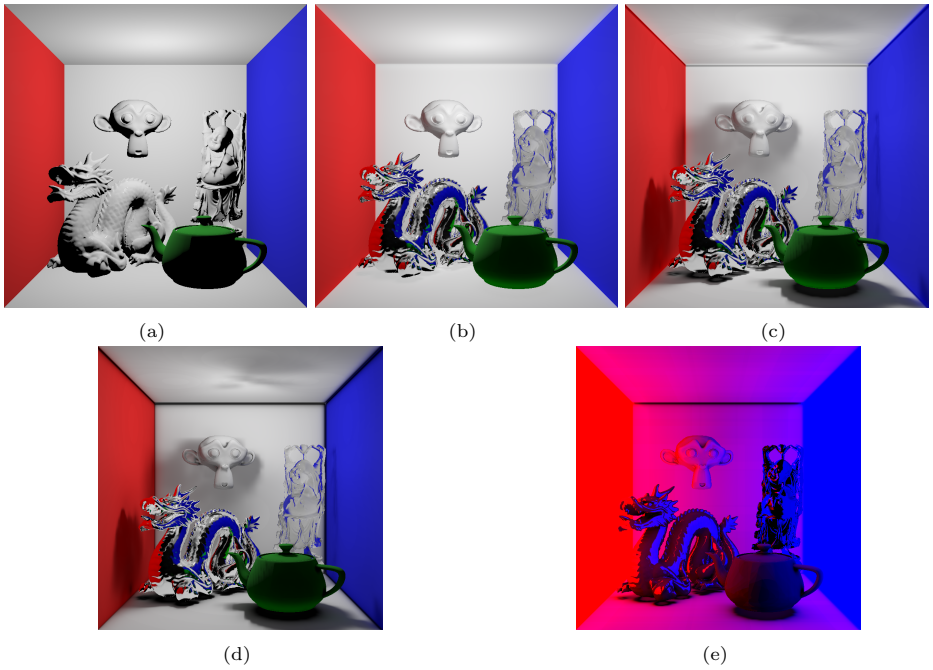


Fig. 6: Cornell Box scene with (a) only basic direct lighting, (b) direct and indirect lighting, (c) full GI,including soft shadows, (d) full GI and ambient occlusion, and (e) emissive wall materials, and no other light sources.

6.1 Results and discussion

We ran the tests described in section 6. All of them performed very well, with frame rates over 30 fps even on full HD (1920x1080) resolutions and all GI effects enabled. These results differ significantly from those obtained with Blender's Cycles renderer. Although the visual results of path tracing are slightly superior in Blender, the performance penalty compared to our engine is very significant. It takes about 20 ms to our application to generate one frame with full GI. Under the same hardware, Cycles needs 18 minutes to generate it. This is around 67.500 times higher. Fig. 7 shows the comparison between both rendering engines at 1024x768 resolution. AO is not enabled on our application to more closely match the reference.

Table 2 shows the average time to render one frame for different voxel grid resolutions and fixed 1024x768 screen resolution. The rows marked with an asterisk are the averages when voxelization is not run on each frame. This Table shows the performance implications of running GI in our rendering engine. It can be seen that the automatic voxelization of the scene on each frame has an impact on the performance, specially visible on Test 1 where no GI is being calculated. Automatic voxelization could be disabled for scenes with no dynamic lighting or

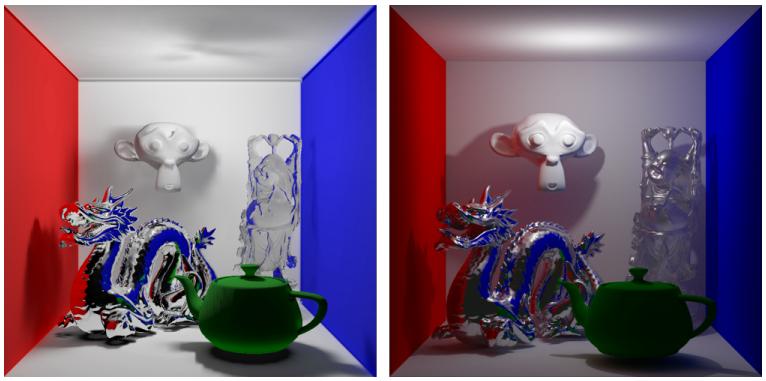


Fig. 7: (a) Voxel Cone Tracing in NATUS (20ms). (b) Path Tracing in Blender (18 min)

Voxel Resolution	Test 1	Test 2	Test 3	Test 4
256	6.62	24.05	29.02	29.01
256(*)	1.72	19.36	24.26	24.26
128	4.09	18.23	22.36	22.34
128(*)	1.79	16.23	20.36	20.36
64	4.92	15.41	18.45	18.44
64(*)	1.77	12.45	15.53	15.52

Table 2: Average time (ms) per frame for each test scene under different voxel grid resolutions. Rows with asterisk indicate that voxelization does not run on each frame.

objects. That allows to gain a small performance boost. Anyway, we get real-time performance (over than 33fps) in all test cases. We observe a strong performance decrease when GI is activated on Test 2, which is completely reasonable. Adding traced soft shadows in Test 3 also increases the render times by an average of 3 ms, but there is no penalty for enabling traced AO. As it is obtained using the same cones traced for indirect diffuse lighting, both effects can be calculated at the same time.

Voxel Resolution	Shadow Map	Voxelize	Filter	Cone Trace
256	0.31	3.01	1.72	28.96
128	0.30	1.96	0.23	22.83
64	0.30	3.14	0.06	16.22

Table 3: Average time (ms) per step for test scene 4 under several voxel grid resolutions

Table 3 shows the average time it takes to perform each step of the full GI algorithm, with indirect diffuse lighting, reflections, refractions and soft shadows. It can be seen that the most expensive step in the GI algorithm is cone tracing the voxel structure. It is interesting to see that voxelization of a 64^3 voxel grid takes more time than that of a 128^3 voxel grid. Generating the mipmap chain on the filtering step is quite fast, and the shadow mapping step does not really depends on voxel resolution which is why it doesn't really change.

7 Conclusions and Future Work

In this article, we present a real-time rendering engine capable of synthesizing high quality images with full Global Illimunation. It was designed following a layered

architecture, where each layer has a specific responsibility within the application which also allows it to be easily extended.

Main features of our engine are the introduction of PBR materials coupled with a simulation of full GI. Our PBR material model is based on the metallic workflow which is a *de-facto* standard on most of the real-time rendering applications used nowadays, allowing for the authoring of materials in third party applications like Substance Designer [20].

The voxel cone tracing allows a wide range of GI effects such as indirect diffuse reflections, ambient occlusion, glossy reflections, transparency and soft shadows to be achieved in real-time with high quality results. While this method yields very good results for dynamic scenes, there are still a few issues that need to be addressed. The major downside of this technique is that it only performs well under small scenes. A similar technique, used for shadow mapping, known as Cascaded Voxelization [21] could be used to extend this solution for bigger scenes.

References

1. T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-time rendering*. CRC Press, 2019.
2. Epic Games. Unreal engine. `www.unrealengine.com/`, 1998. Last acc. July 2020.
3. A. Andrade. Game engines: a survey. *EAI Endorsed Trans. on Serious Games*, 2(6), 11 2015.
4. Crytek. Cry engine, 2002. http://cryengine.com/, online July 2020.
5. Unity Technologies. Unity engine, 2005. `https://unity.com`, online July 2020.
6. Unity3D.com. Realtime global illumination using enlighten, 2002. https://docs.unity3d.com/Manual/realtime-gi-using-enlighten.html, Acc Jul 2020.
7. J. Gregory. *Game engine architecture*. CRC Press, 2018.
8. Windows Developer Center. Windows api. `https://docs.microsoft.com/en-us/windows/win32/apiindex/windows-api-list`, 2020. Last acc. June 2020.
9. S. Fujita. Tinyobj:tiny but powerful single file wavefront obj loader. `https://github.com/syoyo/tinyobjloader`, 2020. Last acc. December 2019.
10. O. Cornout. Dear imgui, 2019. github.com/ocornut/imgui, online June 2020.
11. Lance Williams. Casting curved shadows on curved surfaces. In *Proce. of the 5th annual Conf. on Computer Graphics and Int. Techniques*, pages 270–274, 1978.
12. J. Hable. Uncharted 2: Hdr lighting. In *Game Developers Conf.*, page 56, 2010.
13. B. Burley. Physically-based shading at disney, part of practical physically based shading in film and game production. In *SIGGRAPH Courses*, 2012.
14. W. McDermott. *The PBR Guide*. Allegorithmic, 2018.
15. B. Karis. Real shading in unreal engine 4. *Proc. Physically Based Shading Theory Practice*, 4:3, 2013.
16. S. Lagarde. Feeding a physically-based shading model. `https://seblagarde.wordpress.com/2011/08/17/feeding-a-physical-based-lighting-mode/`, 2011. Last acc. June 2020.
17. C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann. Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, volume 30, pages 1921–1930. Wiley Online Library, 2011.
18. Randima Fernando et al. *GPU gems: programming techniques, tips, and tricks for real-time graphics*, volume 590. Addison-Wesley Reading, 2004.
19. N. Daumann. D3d12 texture mipmap generation. `https://slindev.com/d3d12-texture-mipmap-generation/`, 2017. Last acc. July 2020.
20. J. Jantunen. Creating procedural textures for games: with substance designer. 2017.
21. J. McLaren. Cascaded voxel cone tracing in the tomorrow children. In *Computer Entertainment Developers Conference*, 2014.