POLITECNICO MILANO 1863

SCHOOL OF INDUSTRIAL AND INFORMATION ENGINEERING

Master of Science in Computer Science and Engineering

Physically Based Rendering of Animated Point Clouds

Supervisor

Prof. Marco Gribaudo

Candidate

Matteo Pozzi

Matr. 945685

Abstract

Point cloud 3D models are becoming more and more popular thanks to the diffusion of scanning systems employed in many fields, like autonomous vehicles and robotics. When used in rendering, point clouds are usually displayed with their original color acquired at scan-time without taking into consideration the lighting condition of the scene where the model is placed. This leads to a lack of realism in many contexts, especially in case of animated point clouds where it would be desired to have the model reacting to incoming light and integrating with the environment.

This thesis proposes to apply the rendering technique known as Physically Based Rendering, widely used in Computer Graphics applications, to animated point cloud models to give them a photorealistic and physically accurate look under any lighting condition. An available animated point cloud model will be imported in Unity and rendered with a developed shader implementing Physically Based Rendering. Then, the point cloud using Physically Based Rendering will be compared to the same point cloud rendered with the standard, commonly used shader when placed in different environments characterized by different lighting conditions and it will be shown how, with Physically Based Rendering, a point cloud better integrates to the surrounding environment with respect to the counterpart using a basic, unlit shader. Moreover, it will be shown that with this rendering technique it is possible to render different kind of materials, by exploiting the features of Physically Based Rendering to use the point cloud as a perfect mirror reflecting the environment.

Sommario

Le nuvole di punti 3D stanno diventando sempre più popolari grazie alla diffusione dei sistemi di scansione utilizzati in vari settori, come nei veicoli autonomi e nella robotica. Quando vengono usate nel rendering, le nuvole di punti vengono solitamente visualizzate con il loro colore originale ottenuto durante la scansione senza tenere in considerazione le condizioni di illuminazione della scena in cui il modello viene collocato. Ciò risulta in una mancanza di realismo in molti contesti, soprattutto nel caso di nuvole di punti animate dove sarebbe desiderabile che il modello reagisca alla luce incidente e si integri con l'ambiente.

Questa tesi propone di utilizzare la tecnica di rendering nota come Physically Based Rendering, ampiamente utilizzata in applicazioni di Computer Grafica, sui modelli di nuvole di punti per conferire loro un aspetto fotorealistico e fisicamente accurato sotto qualsiasi condizione di illuminazione. Un modello di nuvola di punti animata verrà importato in Unity e renderizzato con uno shader che implementa il Physically Based Rendering. Successivamente, la nuvola di punti che utilizza il Physically Based Rendering verrà collocata in diversi ambienti caratterizzati da varie condizioni di illuminazione e confrontata con la stessa nuvola di punti renderizzata con lo standard shader tipicamente utilizzato, all'interno dello stesso ambiente. Verrà dimostrato come, utilizzando il Physically Based Rendering, una nuvola di punti si integra meglio con l'ambiente circostante rispetto alla controparte che utilizza uno shader base e non illuminato. Inoltre, verrà mostrato che con questa tecnica di rendering è possibile creare diversi tipi di materiali, sfruttando le caratteristiche del Physically Based Rendering per utilizzare la nuvola di punti come uno specchio perfetto che riflette l'ambiente circostante.

Acknowledgements

I would like to thank first my parents, whose unconditioned support allowed me to overcome all the obstacles I encountered along the way to reach this important milestone in my life. I will always be grateful to them, and I hope to be able to repay them soon for all their support.

Other special thanks go to all my friends and loved ones for their constant presence and their patience, always there ready to make me laugh, without whom I would have gone mad a long way ago.

Thanks to all the colleagues I had the opportunity to work with, for the time spent together on studying and sharing ideas during this journey, and for all their support.

Another thank goes to Prof. Marco Gribaudo for his availability throughout the whole development process of this work, being a guide in helping me to look towards the right direction and dispelling my doubts in many circumstances during our calls.

Finally, I would also like to thank all the professors I had during my journey at PoliMi for all their teachings and methods that allowed me to become an engineer.

Table of Contents

Abstract....................i

Sommario....................iii

Acknowledgements....................v

Table of Contents....................vi

List of Figures....................viii

List of Abbreviations....................xii

1 Introduction....................1

2 State of the Art....................5

2.1 Point Clouds....................5

2.1.1 Point cloud acquisition technologies....................7

2.1.1.1 Laser Scanning with LiDAR....................7

2.1.1.2 Photogrammetry....................8

2.1.1.3 Videogrammetry....................8

2.1.1.4 RGB-D cameras....................9

2.1.1.5 Stereo cameras....................10

2.1.2 The MPEG-PCC standard....................10

2.1.3 Video-based Point Cloud Compression....................11

2.1.4 Geometry-based Point Cloud Compression....................15

2.1.5 Other studies....................18

2.1.5.1 End-To-End Learned lossy compression....................18

2.1.5.2 Geometric distortion measure for PCC....................19

2.2 Rendering....................21

2.2.1 Physically Based Rendering....................25

2.3 Conclusion 30
3 Physically Based Rendering of Animated Point Clouds 33
3.1 Point cloud normals check 33
3.2 Importing models into Unity 37
3.3 Towards PBR 40
3.4 Test scene: background 360° video of an indoor basketball court 47
3.5 Test scene: indoor room with different lights 51
3.6 Test scene: outdoor basketball court in a daylight environment 53
3.7 Test scene: outdoor basketball court in a night environment 58
3.8 Test scene: playing with metalness and smoothness parameters 63
4 Applications 67
4.1 Learning 67
4.2 Advertising 69
4.3 Entertainment 70
4.4 Conclusion 70
5 Conclusion & Future Works 73
Bibliography 75

List of Figures

FIGURE 2.1 - A POINT CLOUD EXAMPLE OF A MONKEY 5

FIGURE 2.2 - A POINT CLOUD OF WASHINGTON, DC CAPTURED BY A LIDAR WITH HEIGHT INFORMATION 6

FIGURE 2.3 - THREE TYPES OF LIDAR ACCORDING TO THEIR WORKING PLATFORM. FROM [8] 9

FIGURE 2.4 - MICROSOFT KINECT V2, AN RGB-D CAMERA 10

FIGURE 2.5 - GENERATION OF 3D PATCHES FROM THE POINT CLOUD. FROM [3] 11

FIGURE 2.6 - AN EXAMPLE OF OCCUPANCY MAP (LEFT), GEOMETRY (MIDDLE) AND TEXTURE (RIGHT) IMAGE. 12

FIGURE 2.7 - PROGRESS OF V-PCC PERFORMANCE SINCE CFP BASED ON D1 GEOMETRY DISTORTION. FROM [13] 13

FIGURE 2.8 – AN OVERVIEW OF THE V-PCC ENCODER. 14

FIGURE 2.9 - AN OVERVIEW OF THE G-PCC ENCODER. FROM [14]. 15

FIGURE 2.10 - THE FIRST TWO ITERATIONS OF OCTREE GENERATION. FROM [3] 16

FIGURE 2.11 - LEVEL OF DETAIL (LOD) GENERATION PROCESS 17

FIGURE 2.12 - VISUAL COMPARISON OF "SOLDIER" BETWEEN DIFFERENT COMPRESSION SCHEMES. FROM [19] 20

FIGURE 2.13 - THE RENDERING EQUATION 22

FIGURE 2.14 - THE PHONG SHADING MODEL WITH ALL ITS COMPONENTS. FROM [22] 22

FIGURE 2.15 - SPECULAR LIGHTING OF PHONG AND BLINN-PHONG REFLECTION MODELS. FROM [23] 23

FIGURE 2.16 - DIFFERENT ITERATIONS OF THE RADIOSITY ALGORITHM. PATCHES ARE VISIBLE AS SQUARES ON WALLS AND FLOOR. 23

FIGURE 2.17 - DIFFERENCE BETWEEN STANDARD DIRECT ILLUMINATION AND RADIOSITY 24

FIGURE 2.18 - AN IMAGE RENDERED WITH MONTE CARLO PATH TRACING 24

FIGURE 2.19 - PATH TRACING WITH INCREASING NUMBER OF LIGHT RAY SAMPLES PER PIXEL 25

FIGURE 2.20 - A MATERIAL WITH INCREASING VALUE OF ROUGHNESS 26

FIGURE 2.21 - DIFFUSION AND SPECULARITY MUTUAL EXCLUSION. SPECULAR MATERIALS SHOW LESS DIFFUSE COLOR 27

FIGURE 2.22 - THE REFLECTANCE EQUATION SOLVED BY PBR SPLIT IN DIFFUSE PART (LEFT INTEGRAL) AND SPECULAR PART (RIGHT INTEGRAL) 27

FIGURE 2.23 - OVERVIEW OF THE CONVOLUTION OPERATION OF THE DIFFUSE INTEGRAL. FROM [27] 28

FIGURE 2.24 - THE SPLIT-SUM APPROXIMATION OF THE SPECULAR INTEGRAL 29
FIGURE 2.25 - AN ENVIRONMENT CUBEMAP (LEFT) WITH THE RESULTING IRRADIANCE CUBEMAP OBTAINED BY CONVOLUTION (RIGHT) 29
FIGURE 2.26 - PRE-FILTERED ENVIRONMENT MAPS OF 5 LEVELS OF INCREASING ROUGHNESS. 29
FIGURE 2.27 - THE BRDF'S INTEGRATION MAP. THE COLORS REPRESENT THE SCALE (RED) AND THE BIAS (GREEN) WITH RESPECT TO THE FRESNEL RESPONSE OF THE SURFACE. FROM [28] 30
FIGURE 2.28 - A HUMAN FACE DIGITALLY RENDERED IN THREE DIFFERENT LIGHTING CONDITIONS WITH LIGHTSTAGE. FROM [30] 31
FIGURE 3.1 - A FRAME OF THE BASKETBALL_PLAYER SEQUENCE VISUALIZED WITH MESHLAB 34
FIGURE 3.2 - DIFFERENT FRAMES OF THE BASKETBALL_PLAYER SEQUENCE WITH CORRECT (LEFT) AND ERRONEOUS (RIGHT) NORMAL ESTIMATION... 35
FIGURE 3.3 - MESHLAB FILTER TO COMPUTE VERTEX NORMALS OF FACELESS MODELS 36
FIGURE 3.4 - THE SAME MODELS AS FIGURE 3.2 AFTER NORMAL RE-COMPUTATION...................... 36
FIGURE 3.5 – A MODEL WITH NORMALS COMPUTED WITHOUT SMOOTHING (LEFT) AND WITH 15 ITERATIONS OF SMOOTHING (RIGHT) WITH K = 10.. 37
FIGURE 3.6 - BASIC SCENE CONSISTING OF A SINGLE FRAME PLACED ON A SURFACE IN UNITY 38
FIGURE 3.7 - A FRAME RENDERED WITH THE BASIC SHADER (LEFT) AND THE NEW SHADER WITH HARD SHADOWS (RIGHT) 39
FIGURE 3.8 – BASIC UNLIT SHADER ALLOWING HARD SHADOW CASTING 39
FIGURE 3.9 - A FRAME RENDERED WITH (LEFT) AND WITHOUT (RIGHT) COLOR SPACE CONVERSION TO LINEAR SPACE 40
FIGURE 3.10 - NORMAL MAP SHADER APPLIED TO A SPHERE IN FRONT (LEFT) AND BACK (RIGHT) VIEW 41
FIGURE 3.11 - NORMAL MAP APPLIED TO THE IMPORTED MODEL WITH THE REFERENCE SPHERE. NORMALS ARE NOT CORRECTLY IMPORTED... 42
FIGURE 3.12 - NORMAL MAP SHADER APPLIED TO MODELS WITH THE MODIFIED IMPORTER IN FRONT(LEFT) AND BACK(RIGHT) VIEW, VIEW THE REFERENCE SPHERE 43
FIGURE 3.13 - DEFAULT LIT SHADER IN SHADER GRAPH 44
FIGURE 3.14 - GRAPH OF THE PBR SHADER THAT WILL BE USED TO RENDER THE BASKETBALL PLAYER MODEL IN A PHYSICALLY PLAUSIBLE WAY 45
FIGURE 3.15 - MODEL RENDERED WITH THE PBR SHADER SEEN FROM DIFFERENT VIEWPOINTS UNDER THE SAME LIGHTING CONDITION.. 46
FIGURE 3.16 - COMPARISON BETWEEN PBR AND UNLIT SHADED POINT CLOUDS UNDER DIFFERENT LIGHTING CONDITIONS... 46
FIGURE 3.17 - A SNAPSHOT OF THE INDOOR BASKETBALL COURT VIDEO IN EQUIRECTANGULAR FORMAT 47

Figure 3.18 – PBR Point cloud model placed in a scene with the indoor 360 video rendered on the skybox 48
Figure 3.19 - Background depth issue: figures in background don't follow the point cloud behavior 48
Figure 3.20 - Fluctuation issue: when rotating the camera it looks like the player fluctuates over the court 49
Figure 3.21 - Depth issue: the point cloud is rendered on top of a character that should be in front of it 49
Figure 3.22 - The insertion of a plane in the scene covers the background video 50
Figure 3.23 - Representation of a room with three different light sources of different color 51
Figure 3.24 - Comparison between the visual look of the PBR and standard point clouds placed in different locations of the scene 52
Figure 3.25 - 3D model of the outdoor basketball court with a directional light 54
Figure 3.26 - Snapshot of the 360 degrees beach video in equirectangular format 54
Figure 3.27 - First comparison between PBR and unlit point clouds in daylight condition 55
Figure 3.28 - Comparison between PBR and unlit point clouds from different viewpoints 56
Figure 3.29 - Comparison between the models when occluded by the basket support and the grate 58
Figure 3.30 - Frame of the night square video in equiangular format and its cubemap projection 59
Figure 3.31 - Basketball court under night lighting conditions 60
Figure 3.32 - First comparison between PBR and unlit point clouds in night condition 60
Figure 3.33 - Comparison between PBR and unlit point clouds when they face one of the spotlights 61
Figure 3.34 - Point clouds placed in different locations of the court 62
Figure 3.35 - Point cloud rendered as a perfect mirror placed in different environments 65
Figure 4.1 - Digital coach showing how to execute an exercise 68
Figure 4.2 - Clothing advertising project from [41]. The Kinect sensor detects a user standing in front of the installation (a). The user interacts with the application, by Kinect or smart screen (b). The user can simulate the fitting of the dress concept created (c) 69

List of Abbreviations

LiDAR	Light Detection and Ranging
MPEG	Moving Pictures Experts Group
PCC	Point Cloud Compression
L-PCC	LIDAR Point-Cloud-Compression
S-PCC	Surface Point Cloud Compression
V-PCC	Video-based Point Cloud Compression
G-PCC	Geometry-based Point Cloud Compression
sRGB	Standard Red Green Blue color space
YUV	Luminance-Bandwidth- Chrominance color space
BD-Rate	Björntegaard Delta Rate
PBR	Physically Based Rendering
BRDF	Bidirectional Reflective Distribution Function
IBL	Image Based Lighting
URP	Universal Render Pipeline

1 Introduction

Rendering is a key concept in the Computer Graphics field since its very first years. Many different techniques have been developed through the years, going from simple algorithms generating 8-bit “pixelated” colors to the most recent ones, showing realistic lighting effects. Physically Based Rendering is a technique born in the ‘80s aiming to achieve photorealistic lighting and is currently widely used in many applications such as videogames, design and many other fields involving the creation of digital images.

Historically, Computer Graphics pipelines have been optimized to work with polygonal meshes, which are 3D models represented by a collection of vertices, edges connecting such vertices and polygonal faces (in most cases triangles) obtained by closed sets of connected edges. Meshes are still the most widely used 3D models in rendering, as hardware and accelerators have been highly optimized to work with such primitives in a fast way.

In recent years, however, with the spread of scanning systems using radar or laser scanners employed on autonomous cars, drones or mounted on air vehicles to acquire aerial scans, have led to the diffusion of a different kind of 3D model using only point primitives, which have been called *point clouds*. Point clouds have several advantages over polygonal meshes, for example the fact that they can be automatically acquired in a rapid way, saving long and tedious work of designers, having to model and check meshes by hand. Another advantage is that since a point cloud has no connectivity information, points can be stored and transmitted in any order, as long as the whole set is considered. This suggests that point clouds could be heavily used in Computer Graphics applications by substituting polygonal meshes, although there is still no optimized hardware to work with them.

At present day most applications using point clouds use simple rendering techniques to display only the color of the model acquired during scanning. In some cases this is fine, since there is no practical advantage in using complicated rendering techniques to visualize the model and the original color is all what is needed. An example is a point cloud acquired by an autonomous car, which doesn't need to be rendered in a photorealistic way but just needs colors to distinguish obstacles on its path (semaphores, crosswalks, other cars etc.). However there exist other applications involving animations where the point cloud needs to be better contextualized within an environment and we would like to obtain from the model a photorealistic look, as in movies, immersive experiences in 360 degrees Virtual Reality such as virtual tours and games, or a mix between traditional 3D models and VR. In such scenarios the basic rendering technique is not enough, and we could use, instead, Physically Based Rendering to give the animated point cloud a photorealistic look under any external lighting condition, allowing to relocate it in any environment and still maintaining the feel that the model belongs to it.

In this work we will show how Physically Based Rendering can be applied to animated point clouds as to polygonal meshes to obtain a photorealistic appearance of the model under various lighting conditions. The document at hands follows this structure:

- In Chapter 2 the main concepts and topics subject of this study are analyzed in detail, starting from point cloud acquisition techniques, the ongoing standardization efforts on point cloud compression addressed by the MPEG group and ending with a brief analysis of trending rendering techniques, with focus on Physically Based Rendering.
- Chapter 3 is the core of this document. It contains all the steps followed to import a point cloud animation sequence into the engine used for rendering and the development of a shader implementing Physically Based Rendering which is then applied to the point cloud. Then a series of

different scenes is presented, which aim to recreate various environments to study how the point cloud rendered with the PBR technique behaves compared to the point cloud rendered by just displaying the original color, and some of the features provided by PBR are explored.

- In Chapter 4 a short overview on potential applications of (animated) point clouds with the support of Physically Based Rendering is presented.
- Chapter 5 contains some final considerations about the obtained results and discusses the limitations of the study, leaving space for future developments on the subject.

2 State of the Art

2.1 Point Clouds

As the name suggests, point clouds are a set of high-density individual points used to represent volumetric visual data, which can be computer-generated or directly captured from the real world. All these points carry various attributes about the properties of the object they are representing, like the basic position in the space (x, y, z coordinates) and eventually its color, surface normal, etc. [1]. They can be seen as an alternative to polygonal meshes when representing 3D models, with the advantage that they can be directly sampled from the real world with cameras without the need of reconstructing the surface and they can be processed in real time.

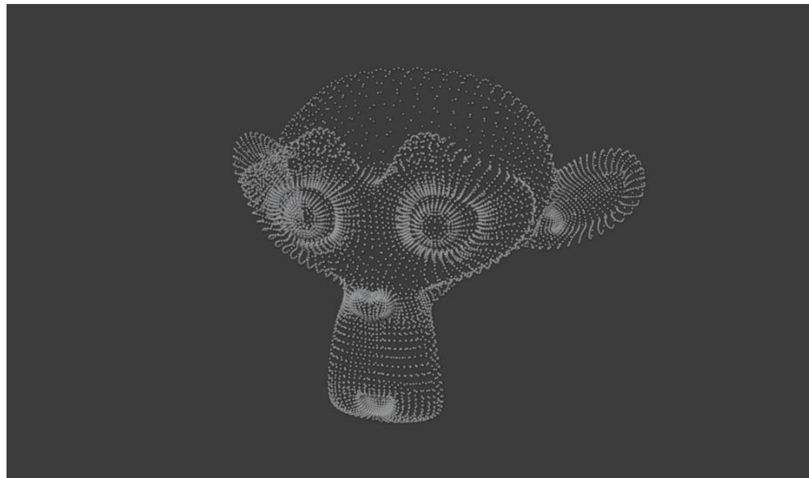


Figure 2.1 - A point cloud example of a monkey

Usually, the 3D coordinates of points are represented by floating-point values, but they can also be quantized into integer values by creating a grid in 3D space and mapping each point residing within a sub-grid volume to the sub-grid center, referred to as *voxel*. The process is hence called *voxelization*. Voxels are often seen as a 3D extension of pixels and are very accurate 3D building blocks allowing simulation techniques that wouldn't be possible with other modelling methods. However, current computer hardware is optimized for rendering polygons and

specialized hardware to render high-resolution voxels is not available yet [2]. Also, the space precision may affect the perceived quality of the voxelized point cloud [3].

Since point clouds lack information about vertices connectivity, more points are needed both to “fill” the model (which otherwise may result in having holes) and to get a better level of detail that would be lost when removing the information about faces from the polygonal mesh, but having too many points might also result in adding details not present in the original model. However, since there is no connectivity information, storage and transmission of point clouds are simpler, as points can be acquired, stored and transmitted in any order as long as the whole set of points is considered, while a polygonal mesh needs to preserve the order of points to ensure that the connectivity of vertices is kept unaltered during compression and transmission [4]. Even though polygonal meshes are still widely used in Computer Graphics applications due to their integration with graphics pipelines and surface representations, point clouds are getting more and more popular in applications like virtual and mixed reality (AR/VR/MR) thanks to their flexibility [1]. As point clouds also provide immediate depth information, they also find use in applications like self-driving cars and, in general, autonomous

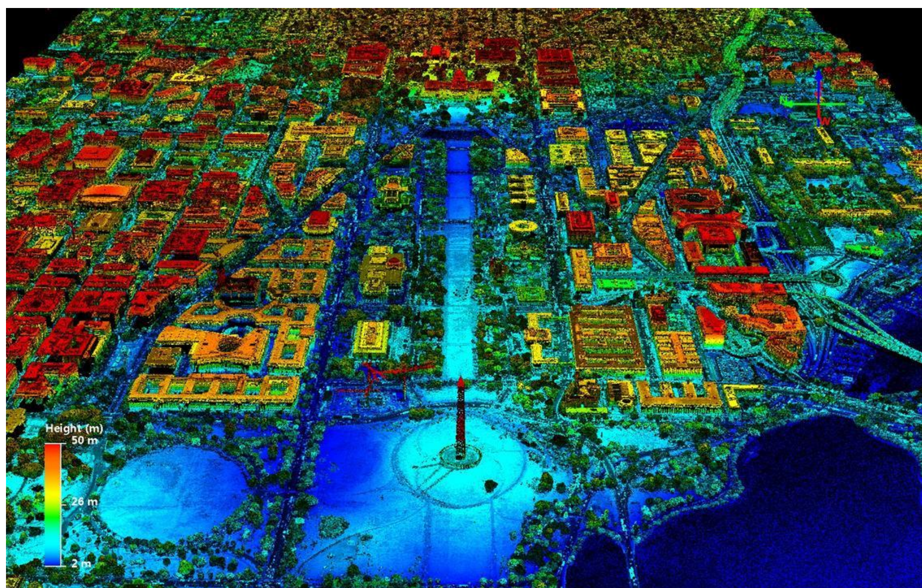


Figure 2.2 - A point cloud of Washington, DC captured by a LiDAR with height information

machines, helping to detect objects and allowing navigation and localization [5], and also in the construction industry, to reconstruct an original 3D model of a building from its point cloud representation, for geometry quality inspection tasks and as an help to repair and maintain cultural heritage buildings [6].

2.1.1 Point cloud acquisition technologies

There are a few ways in which a point cloud can be generated from the real world. The most common solutions are LiDAR-based scanning and photogrammetry, but other alternatives are possible, for example videogrammetry, RGB-D cameras and stereo cameras. A short overview will be given for each.

2.1.1.1 Laser Scanning with LiDAR

Laser scanners measure the distance from an object by emitting laser beams and detecting the reflected beam from the object. The distance is then evaluated with the Time-of-Flight principle or Phase-Shift of the wave. Scanners using the Time-of-Flight principle have a higher maximum range of measurement than the counterpart, while Phase-Shift gives higher ranging accuracy and measurement speed. Different solutions exist on the market, going from very expensive mechanical LiDARs (up to $100,000) to cheaper solid-state LiDARs (less than $1,000). For example, the iPhone12 Pro is equipped with a solid-state LiDAR with short sensing range [5, 7].

Laser scanners can be furtherly divided into three categories [8]:

- Terrestrial Laser Scanner (TLS), also known as ground LiDAR, which is mounted on tripods placed on the ground. Since during operation it is still, this is the solution with highest accuracy and used for surveying and monitoring buildings and infrastructures.
- Airborne Laser Scanner (ALS), also known as aerial laser scanner, which is mounted on aircrafts during flight. Its main advantage is the high mobility

and is mainly used to capture point clouds of terrains without the need of being very accurate.

- Mobile Laser Scanner (MLS), which is mounted on ground mobile platforms, such as vehicles. They are mostly adopted for 3D city mapping and as sensing systems of autonomous cars.

Examples of LiDARs are available in Figure 2.3. With multiple LiDAR scans it is possible to create very detailed and accurate measurements, with millions of measurements at each laser pulse [9].

2.1.1.2 Photogrammetry

Photogrammetry is defined as the art, science and technology of obtaining reliable information about physical objects and the environment through the process of recording, measuring and interpreting photographic images and patterns of electromagnetic radiant imagery and other phenomena. So, photogrammetry gets data from photographs instead of light rays. Many photos must be taken from different angles and overlapped to capture the geometry of the object to represent as a point cloud. These photos can be captured even with simple cameras, making the approach more affordable in terms of availability and costs, but is less accurate than 3D scanners and it can be difficult if a multi-camera setup is not available. However, the advantage is the straightforward capability of photogrammetry to represent objects with full color, directly taken from photos [9].

2.1.1.3 Videogrammetry

Videogrammetry can be seen as an extension of photogrammetry, taking as input a video stream instead of a collection of images. With videogrammetry it is possible to progressively reconstruct a point cloud by stacking the information obtained from a video frame to the previous frames, and gradually increasing the accuracy and detail of the model. Also, the need of human intervention is limited since the reconstruction process can search for points by tracking features from consecutive frames [8].

Ground LiDAR mounted on a tripod

Mobile LiDAR mounted on a car

Airborne LiDAR mounted on
an aircraft

Figure 2.3 - Three types of LiDAR according to their working platform. From [8]

2.1.1.4 RGB-D cameras

An RGB-D camera consists of an RGB camera (red, green, blue) plus a depth sensor. Images taken by this camera are normal RGB images augmented with depth information at pixel level. Colored 3D point clouds can be generated by just mapping the image to the depth. A very popular RGB-D camera is the Microsoft Kinect (Figure 2.4), released firstly in 2009 and widely used in many applications such as robotics and computer vision.

A study made by [10] proposes an architecture for an automated indoor scanning system that uses multiple RGB-D cameras facing different directions with a slightly overlapping field of view and mounted on a rotating support. This solution doesn't need human support, such as manually moving the sensors in the room, for scanning indoor environments.

Figure 2.4 - Microsoft Kinect v2, an RGB-D camera

2.1.1.5 Stereo cameras

A stereo camera has two or more lenses and an image sensor. Knowing the relative position and orientation of lenses, it is possible to obtain 3D point clouds from the acquired 2D images, after a previous fully automated calibration process based on the same images.

2.1.2 The MPEG-PCC standard

The first standardization activity for Point Cloud Compression was initiated in 2014 by the Moving Picture Experts Group (MPEG), known for other standardizations in the field of multimedia technologies. In 2017 they came up, through a Call for Proposals (CfP), with three different technologies for three targeted categories:

- LiDAR Point Cloud Compression (L-PCC) for dynamically acquired data.
- Surface Point Cloud Compression (S-PCC) for static point cloud data.
- Video-based Point Cloud Compression (V-PCC) for dynamic content.

The final standard came out in 2020 and consists of two approaches:

- *Video-based*, appropriate for point clouds with a uniform distribution of points.

- *Geometry-based*, equivalent to the combination of L-PCC and S-PCC, appropriate for sparser distributions.

2.1.3 Video-based Point Cloud Compression

The idea behind V-PCC comes from the great success of 2D video compression that is widely used thanks to the spread of video coding standards. To take advantage of such technologies, PCC may convert a point cloud from 3D to 2D and then code it with 2D video encoders [3]. The proposal was to divide the point cloud into connected regions named 3D patches (clusters) and then project each

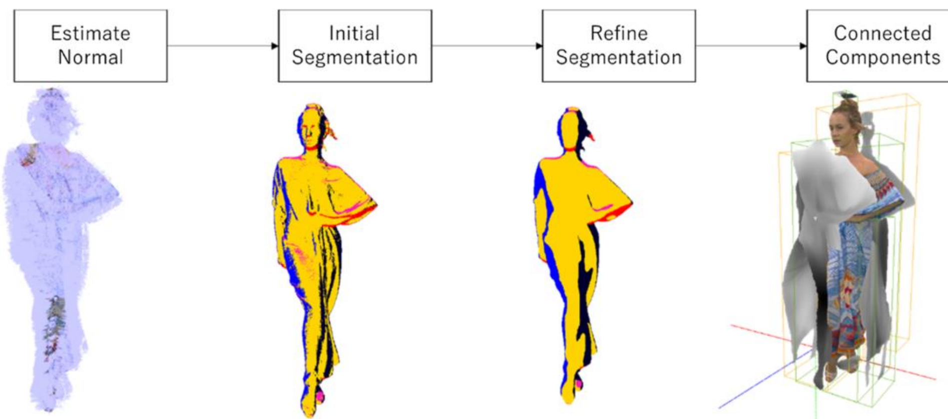


Figure 2.5 - Generation of 3D patches from the point cloud. From [3]

of them independently into a 2D patch with orthogonal projections, which are then packed into images that can be compressed with any existing or future video codec (for example, MPEG-4 or AVI). The approach also helps to reduce self-occlusions and distortions that may be present in the original point cloud. The objective of creating patches is to obtain a temporally coherent, low-distortion, injective mapping, which would assign each point of the 3D point cloud to a cell of a 2D grid [11]. The mapping between the point cloud and the 2D regular grid is then created by packing the projected patches, with a strategy that can be different from encoder to encoder. The compression efficiency can be improved by mapping patches with similar content to similar positions [3].

After the patch-packing process is completed, other images are generated:

- A *geometry image* containing the depth information of the point cloud (distance between each point's position and the projection plane). Since a patch may have different points being projected to the same pixel, the standard allows the encoder to use more layers ordered from lowest to highest depth value to store overlapping points.
- A binary image called *occupancy map* signaling whether a pixel is occupied by a valid 3D projected point or not.
- Other attribute images containing information like the texture (color) of each point, the material or user-defined attributes.

Figure 2.6 - An example of occupancy map (left), geometry (middle) and texture (right) image.

The occupancy map is used to disambiguate pixels used for 3D reconstruction of the point cloud from pixels that, instead, are unused and inserted by the padding procedure. Both lossless and lossy coding are possible for occupancy maps. The padding function is applied to geometry images to fill the spaces between patches and obtain a piecewise smooth image, improving the video compression efficiency furtherly.

The V-PCC bitstream is finally created by concatenating into a single stream all the encoded information.

The decoding process is split into two different phases: information decoding and point cloud reconstruction. The first phase generates the 2D video frames together with patches information associated to each frame from the encoded

information, while the second phase recreates the 3D point cloud from the video frames. However, while the decoding result is generally bit-exact, the reconstruction can lead to a slightly different geometry with respect to the original one, introducing artifacts due to quantization errors. Studies to remove such artifacts are ongoing. For example, in [12] a solution using deep learning based on a U-Net architecture is presented.

Since the reconstructed geometry might be different from the original one, the information about color is transferred from the original point cloud to the decoded point cloud and these new colors are used for transmission [3].

Since the first Call for Proposals evaluation, the performance of V-PCC has been constantly improving. In Figure 2.7 the progress is shown in terms of coding performance, based on the D1 geometry distortion, with also the originally proposed technology added as reference [13].

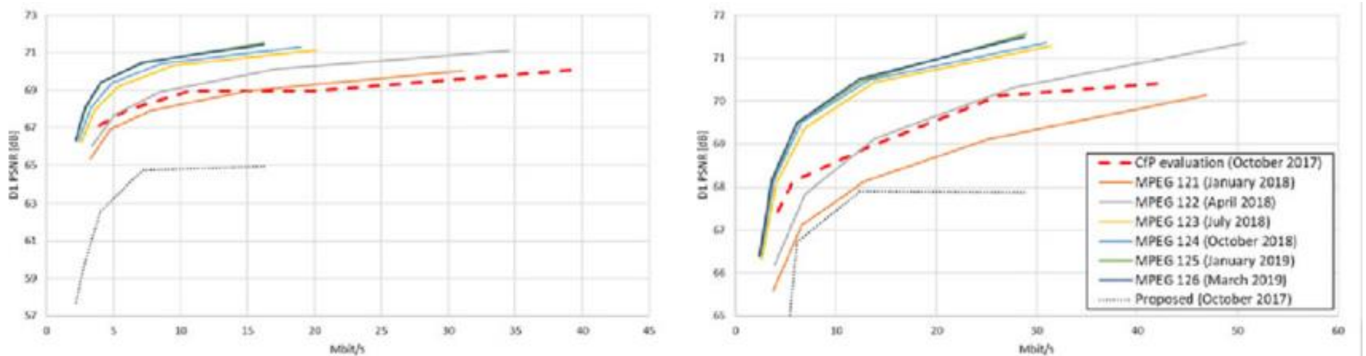


Figure 2.7 - Progress of V-PCC performance since CfP based on D1 geometry distortion. From [13]

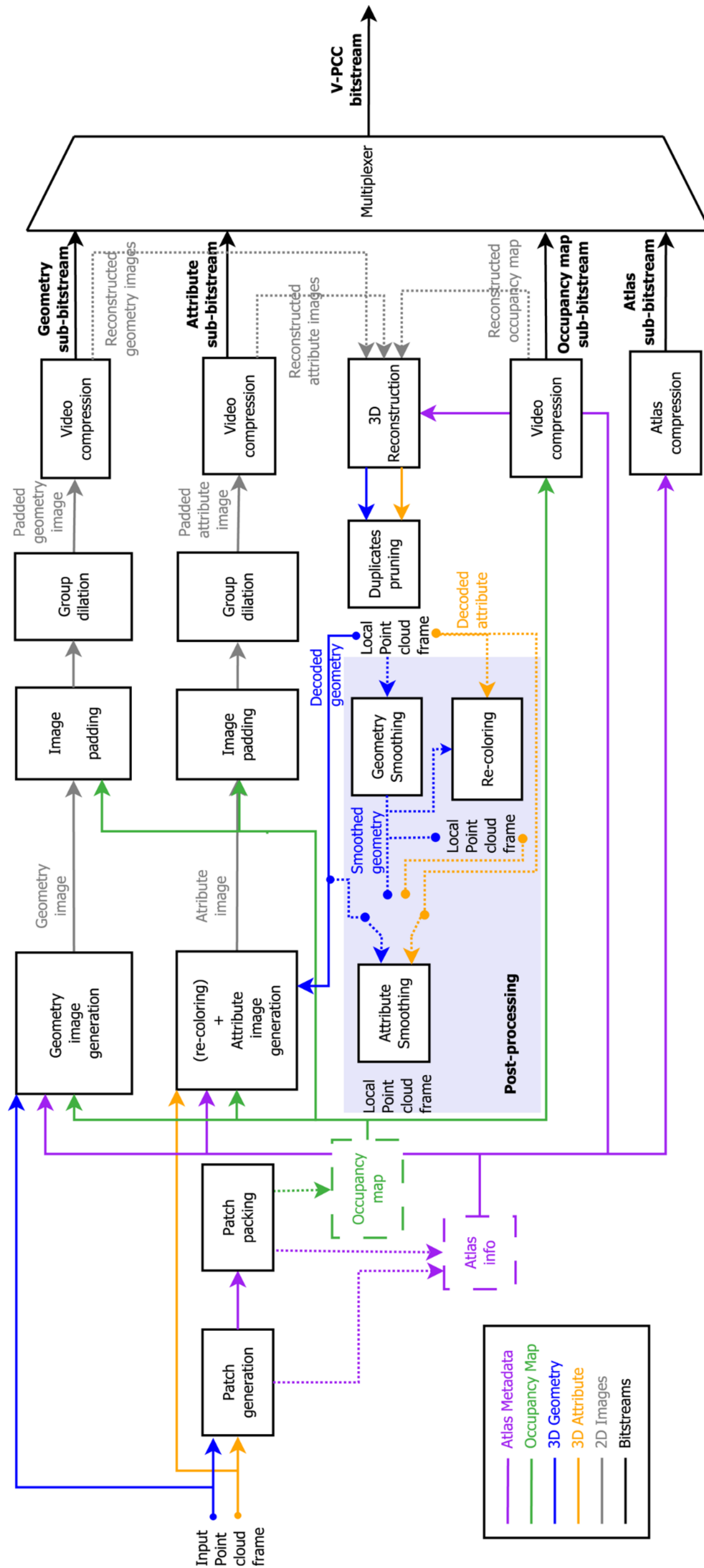


Figure 2.8 – An overview of the V-PCC encoder.

2.1.4 Geometry-based Point Cloud Compression

The first difference between V-PCC and G-PCC is that while video-based compression uses video coding formats to project a 3D point cloud into a 2D representation, geometry-based encoding instead directly encodes the model in 3D by using a data structure called *octree* describing a point localization in 3D space. Also, the approach makes no assumption on the coordinate representation used by the input point cloud and points have an internal integer-based representation obtained with a conversion from a floating-point representation. The conversion is conceptually like voxelization [3].

G-PCC also allows for parallel coding functionalities using *slices* and *tiles*. A slice is a set of points, with geometry and attribute information, that can be independently encoded and decoded, while a tile is a group of slices with bounding box information. Tiles may overlap with each other, and specific slices may be accessed to decode a specific area of the point cloud only.

An overview of the core modules of the G-PCC encoder is shown in Figure 2.9. The first thing to be noted is that geometry and attributes are encoded separately, but since attribute encoding depends on geometry encoding, geometry is encoded first [14].

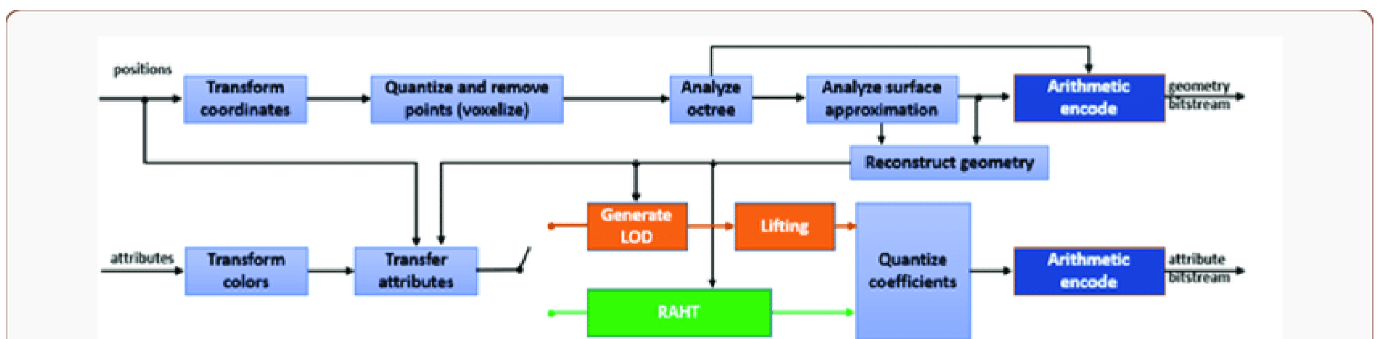


Figure 2.9 - An overview of the G-PCC encoder. From [14].

The first operation made in both geometry and attribute encoding is a conversion into a different representation: point positions are converted from floating-point values into integers with a coordinate transformation followed by voxelization, while colors are converted from RGB to YUV color space, which is more compression friendly. In fact, it is shown in [15] that compressing an image in YUV format leads to a reduction of the root mean squared error to 78.65% of the same image with RGB coding.

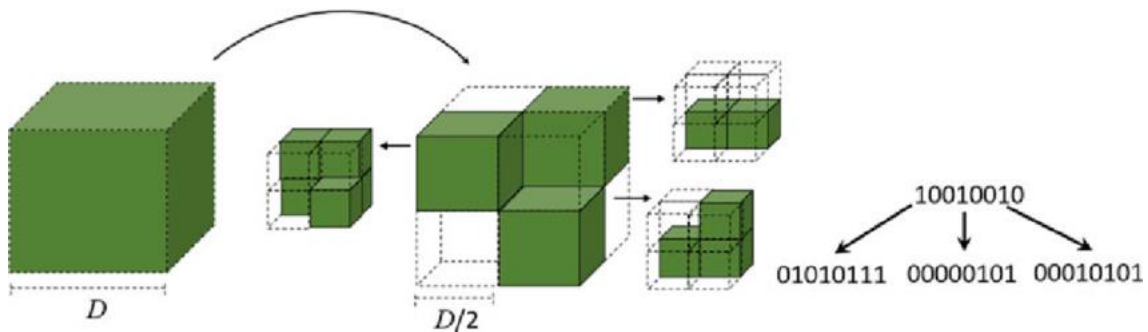


Figure 2.10 - The first two iterations of octree generation. From [3]

The following step is the geometry analysis of the octree. Two schemes are possible:

- *Octree coding:* assuming that the quantized volume occupied by the point cloud is $D \times D \times D$ voxels, this is initially horizontally and vertically partitioned into eight sub-cubes of $\frac{D}{2} \times \frac{D}{2} \times \frac{D}{2}$ voxels. The process is recursively applied to occupied sub-cubes until D becomes equal to 1. Occupied blocks are marked by a 1 while empty ones are marked by a 0. The generated octets at each step represent an occupancy state stored in one-byte words. The first two steps of the process can be visualized in Figure 2.10.
- *Trisoup surface approximation:* the geometry is represented by a pruned octree constructed until a chosen level where leaves represent sub-cubes with a higher dimension than a voxel. The object surface is then approximated by a series of triangles without any connectivity information between each other (a "triangle soup", or trisoup, giving the name to the approach).

In general, only 1% of voxels are occupied and this makes the octree representation very convenient. Since more points can be mapped to the same sub-cube, the number of points for each sub-cube is also arithmetically encoded [16].

The reconstructed geometry is then used to transfer attributes in order to minimize distortions between the input and reconstructed point cloud. Three different methods are available for attribute coding:

- *Region-Adaptive Hierarchical Transform (RAHT):* use the attribute value in a lower octree level to predict the value of the next level, starting from leaves and heading up to the root. For more details about the transform see [3].
- *Predicting Transform:* a distance-based prediction scheme relying on a Level of Detail (LoD) representation distributing points in sets of different refinement levels based on a deterministic Euclidean distance. Attributes are encoded following the prediction determined by the LoD order.
- *Lifting Transform:* built on top of the Predicting Transform, it adds an update operator and an influence weight to each point. Since points in lower LoDs are used more often in prediction, they have a higher impact on the process.

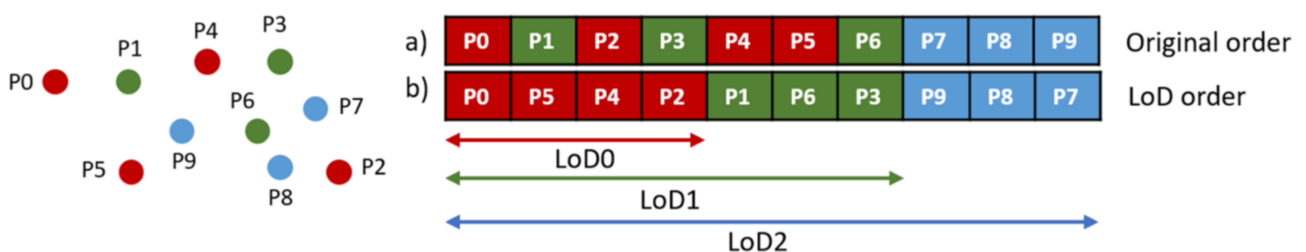


Figure 2.11 - Level of Detail (LoD) generation process

For a more detailed overview on G-PCC standard the reader can refer to [17].

Although G-PCC can achieve remarkable performances, it might still lead to serious artifact issues during the attribute compression task, especially in low bitrate scenarios. A first study was made in [18] where a solution using a Multi-Scale Graph Attention Network is used to remove artifacts on compressed attributes. Experiments showed that on average the proposed solution reaches a 9.28% BD-rate reduction (a measure of rate-distortion performance).

2.1.5 Other studies

In this section a short overview of other studies carried out independently from the MPEG standardization effort is given. In particular, attention will be given to an end-to-end framework based on a deep neural network to efficiently compress a point cloud geometry [19] and a point-to-plane metric to measure geometric distortions of point clouds [20].

2.1.5.1 End-To-End Learned lossy compression

The idea to try to use a deep learning approach to compress point clouds' geometry comes from the emerging of analogous applications in 2D compression of images and videos. Since redundancy in 2D images can be exploited by stacked 2D convolutions, the idea was to try to use 3D convolutions to have a compact representation of a point cloud. The proposed framework consists in three main operations:

1. The point cloud firstly goes through a pre-processing pipeline, where it is voxelized and partitioned into non-overlapping cubes, to reduce the computational cost that would be required to process the entire point cloud at a time. The position of occupied cubes is specified with an octree decomposition and the number of occupied voxels in each cube is also

transmitted to allow later point cloud reconstruction. Each cube is processed independently, allowing massive parallelism of the task.

2. The obtained volumetric point cloud is then fed into a Variational AutoEncoder (VAE) architecture composed by a stack of three consecutive 3D convolutions with integrated downsampling to generate hyperpriors and have a compact representation. Convolutions are interleaved by a Voxception-ResNet (VRN) to capture the essential information of the representation. The hyperpriors are later used to increase the conditional probabilities of latent features. To train the network, a Weighted Binary Cross-Entropy loss function has been used to optimize distortion, where the non-occupied voxels are weighted more than the occupied ones. The reconstruction task is treated as a classification problem, where the target is a voxel to be classified as occupied or not occupied.
3. A final post-processing phase, where voxels classification and extraction are made. Decoded voxels coming from the neural network are floating-point numbers in the interval [0, 1] and need to be classified as occupied or not occupied. At inference time the used threshold is not fixed, but an adaptive threshold is used based on the number of occupied points belonging to the original cube fed to the network (information contained in each cube as metadata). Finally, voxelization is reverted and points are extracted from the volumetric representation.

The study showed that this method outperforms the G-PCC standard with a good margin of at least 60% BD-Rate. The results also qualitatively look better, as can be noted in Figure 2.12. For further details about the framework the reader can refer to [19].

2.1.5.2 Geometric distortion measure for PCC

Classic metrics used by the MPEG standard to measure geometry distortion are based on point-to-point or point-to-surface distances. Regarding the point-to-point framework, firstly for each point of the original point cloud a corresponding

point in the compressed point cloud is identified. Then the average or maximum Euclidean distance between such couple of points is used as basis to measure distortion. However, the approach fails to consider that points in a point cloud often represent surfaces, so a point-to-surface approach has ben developed. In this

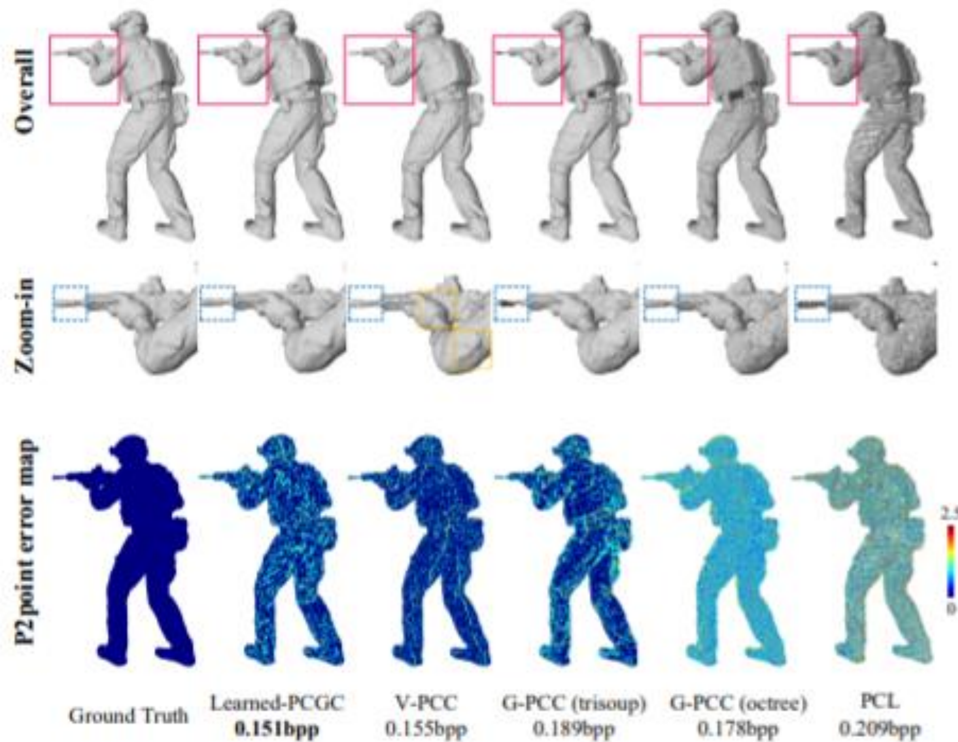


Figure 2.12 - Visual comparison of "soldier" between different compression schemes. From [19]

second approach a mesh is constructed from the original point cloud and then the distances between the compressed point cloud and the corresponding mesh are computed. The framework, however, strongly depends on the algorithm used to obtain the mesh from a point cloud and is difficult to use.

The proposed approach uses a point-to-plane measure, that resides between the point-to-point and point-to-surface approaches [20].

Firstly, for each point of the reference point cloud the corresponding point in the compressed one is identified. Such corresponding point is determined as the nearest neighbor of the point itself. Then the unit normal vector of the reference point is considered if available or, if not, estimated on the fly. The point-to-point error is then computed by connecting the two considered points. The final,

proposed point-to-plane error is obtained by projecting the point-to-point error onto the normal vector of the reference point.

For point clouds in which surfaces are represented, the proposed point-to-plane error is better aligned to the perceived quality of the point cloud with respect to the point-to-point metric. In addition, instead of considering the mean square error (MSE), the values are converted into Peak Signal-to-Noise Ratio (PSNR) numbers to normalize the metrics with respect to a peak value, which is chosen as the intrinsic resolution of the input point cloud.

The point-to-plane metric requires lightweight computation and is demonstrated to better track visual qualities of a compressed point cloud than the classic point-to-point metric.

2.2 Rendering

Rendering is a key concept in Computer Graphics. It is the process of automatically creating a 2D or 3D image from a scene defined by a series of objects. It involves many information, such like object geometry, textures, shading, lighting, shadows, materials, reflectance, transparency and so on.

Rendering is, in general, a very expensive task in terms of calculation and time. The key is to find a good balance between image quality and rendering speed determining how many frames can be processed in a certain period of time [21]. The higher is the quality of the image we expect from rendering, the more time will be required to compute it. A good choice is to use algorithms that produce images with an acceptable perceived quality for the specific application we intend to use them for and don't require too much time to be computed.

Through the years many different techniques have been developed to try to provide an approximate solution to the rendering equation and give a realistic look

to scenes (an exact computation is still infeasible since it would be needed to follow the path of every single ray of light in the scene, which are infinite). Some examples of photo-realistic lighting models are:

$$L(\mathbf{x},\vec{\omega}_o)=L_e(\mathbf{x},\vec{\omega}_o)+\int_S f_r(\mathbf{x},\vec{\omega}_i \rightarrow \vec{\omega}_o)L(\mathbf{x}',\vec{\omega}_i)G(\mathbf{x},\mathbf{x}')V(\mathbf{x},\mathbf{x}')d\omega_i$$

,where $L(\mathbf{x},\vec{\omega}_o)$ = the intensity reflected from position $\mathbf{x}$ in direction ω_o

$L_e(\mathbf{x},\vec{\omega}_o)$ = the light emitted from $\mathbf{x}$ by this object itself

$f_r(\mathbf{x},\vec{\omega}_i \rightarrow \vec{\omega}_o)$ = the BRDF of the surface at point $\mathbf{x}$, transforming incoming light ω_i to reflected light ω_o

$L(\mathbf{x}',\vec{\omega}_i)$ = light from $\mathbf{x}'$ on another object arriving along ω_i

$G(\mathbf{x},\mathbf{x}')$ = the geometric relationship between $\mathbf{x}$ and $\mathbf{x}'$

$V(\mathbf{x},\mathbf{x}')$ = a visibility test, returns 1 if $\mathbf{x}$ can see $\mathbf{x}'$, 0 otherwise

Figure 2.13 - The Rendering Equation

- The *Phong lighting model* [22], that considers three lighting components: ambient, diffuse and specular lighting. Ambient lighting is a constant term that always gives the scene some color to simulate the fact that objects are never completely dark; diffuse lighting simulate the directional impact of light hitting an object, the more a face is aligned with the light the brighter it becomes; specular lighting is based on the reflection properties of surfaces and the view direction. It simulates the bright spot of a light appearing on shiny objects. In Figure 2.14 an overview of all the lighting

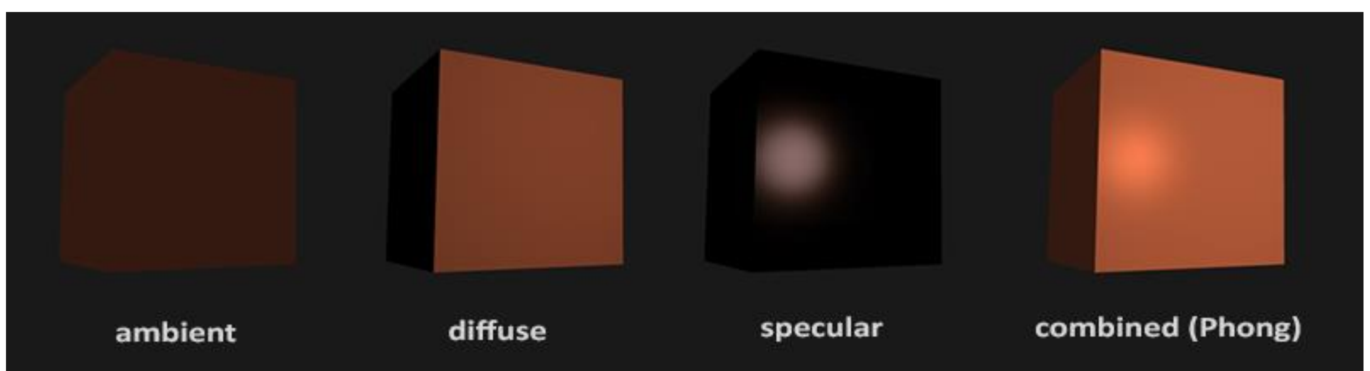


Figure 2.14 - The Phong shading model with all its components. From [22]

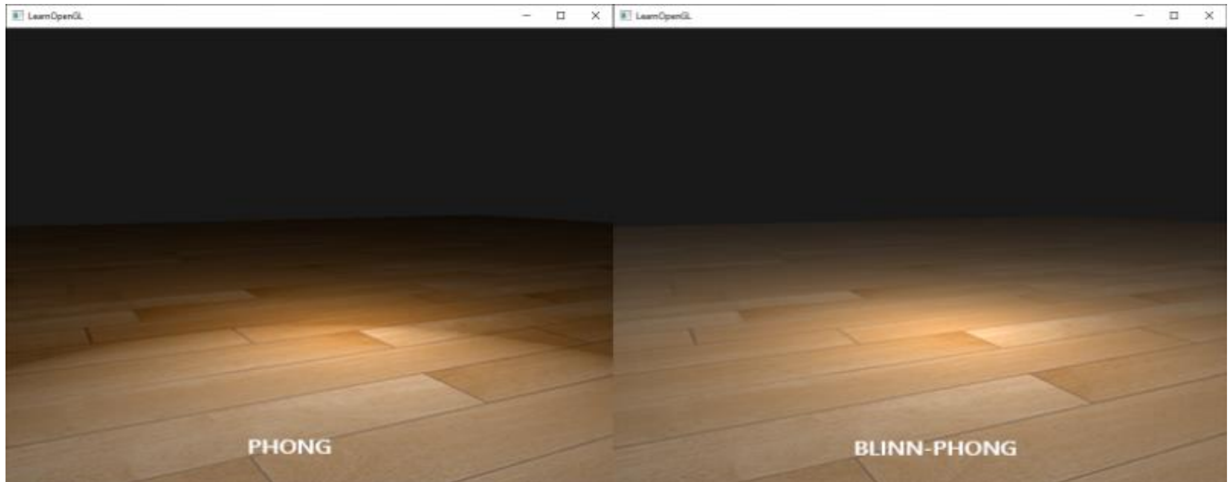


Figure 2.15 - Specular lighting of Phong and Blinn-Phong reflection models. From [23]

terms and how they combine together is available. Obviously, all terms can also be represented by textures rather than just raw colors. The model however suffers from an issue about specular light when the angle between the reflected light vector and the viewing direction is greater than 90 degrees, which is solved by the Blinn-Phong lighting model, that introduces a halfway vector. For details, see [23].

- The *radiosity* method, which is an alternative to the Phong model that tries to better approximate the interaction of diffuse surfaces [24]. It is a view independent algorithm that simulates indirect illumination effects by considering light rays leaving a light source and reflected some number of times on surfaces before reaching the eye. All objects in the scene are divided into patches with a single unknown per patch, which is the radiosity of the object (the "amount" of light hitting the object). Light sources instead are implemented as patches emitting radiosity. The rendering

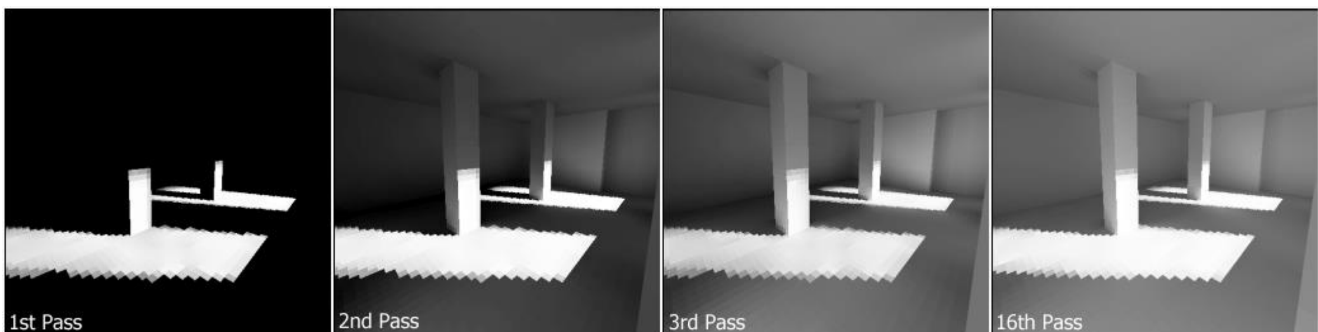


Figure 2.16 - Different iterations of the radiosity algorithm. Patches are visible as squares on walls and floor.

equation is transformed into a very large system of linear equations which can be solved with an iterative approach. The method is well suited for indoor scenes due to its capability to mimic real-world phenomena of light reflection, as can be seen in Figure 2.17. However, a major limitation of the method is that it was designed to simulate purely diffuse environments and lighting effects such as specularity of materials like glass and metal can't be

Figure 2.17 - Difference between standard direct illumination and radiosity

rendered properly and need to be considered in some way by extending the solution provided by standard radiosity.

- *Monte Carlo techniques* which use simulation and sampling to approximate the solution of the rendering equation. Many algorithms exist using this technique, an example is *path tracing* which integrates over all the illuminance hitting a specific point on a surface, for every point. The

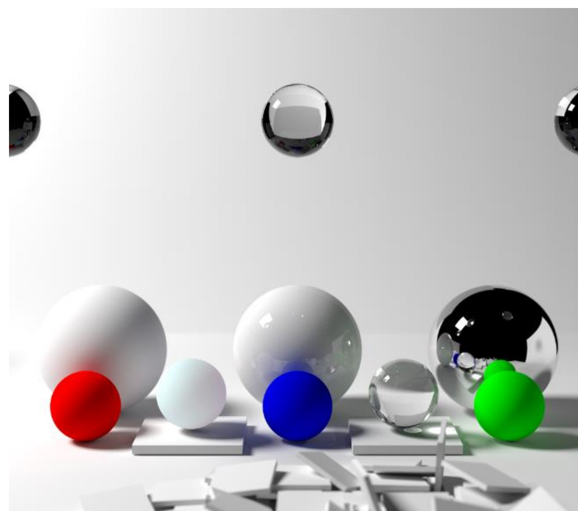


Figure 2.18 - An image rendered with Monte Carlo path tracing

algorithm can give a very realistic appearance to images and naturally simulates effects like soft shadows, ambient occlusion and depth of field, but it is rather inefficient. A lot of rays need to be followed to have a high-quality result and without noise artifacts, which are a constant issue of all methods based on Monte Carlo simulation.

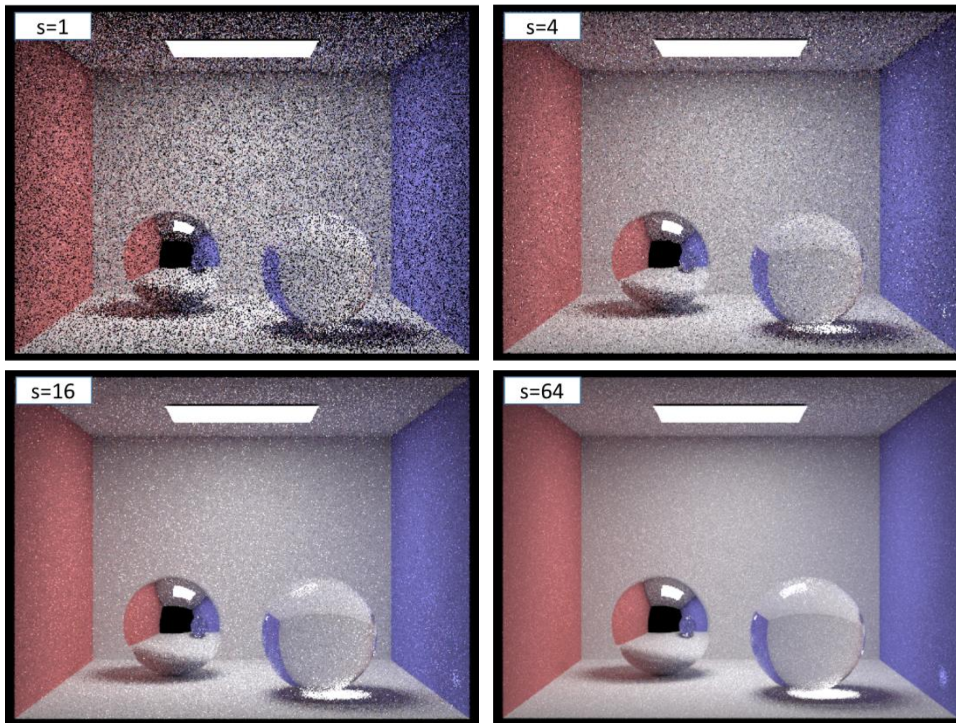


Figure 2.19 - Path tracing with increasing number of light ray samples per pixel

2.2.1 Physically Based Rendering

Physically Based Rendering, or PBR, is an interesting and currently widely used collection of rendering techniques used in real time rendering. It tries to mimic how light interacts with surfaces in a physically plausible way, making the result more photorealistic with respect to other shading models [25]. Another advantage is that it is possible to create different materials by changing physical parameters and making them look correctly under any lighting condition without the need to resort to coding hacks.

To be considered as physically based, a lighting model must fulfill three conditions:

1. Be based on the microfacet surface theory, according to which any surface can be described at a microscopic scale by very small perfectly reflective mirrors called microfacets. The rougher a surface is, the more dispersive will be the scattering of light when bouncing on it, generating widespread specular reflections, while a smooth surface will reflect light approximately in the same direction. The roughness of the material is one of the physical parameters that can be tweaked to obtain different behaviors on the surface (Figure 2.20).
2. Satisfy the energy conserving principle, which states that the light energy leaving a surface never exceeds the incoming light energy (except for emissive surfaces). A distinction must be made between diffuse and specular light: when hitting a surface, light splits in a *refracted* and a *reflected* part. Refracted light is absorbed by the surface and scatter, producing diffuse lighting, while reflected light does not enter the surface and produce specular lighting. The conclusion is that reflection and diffusion are *mutually exclusive:* reflected light will be no longer available to get absorbed by the surface. A consequence is that highly reflective surfaces will show little to no diffuse light, as well as surfaces with strong diffusion won't be much reflective [26]. A note must be made on metallic materials: when light hits a metallic surface, all refracted light is directly absorbed without scattering, meaning that no diffuse light is shown.

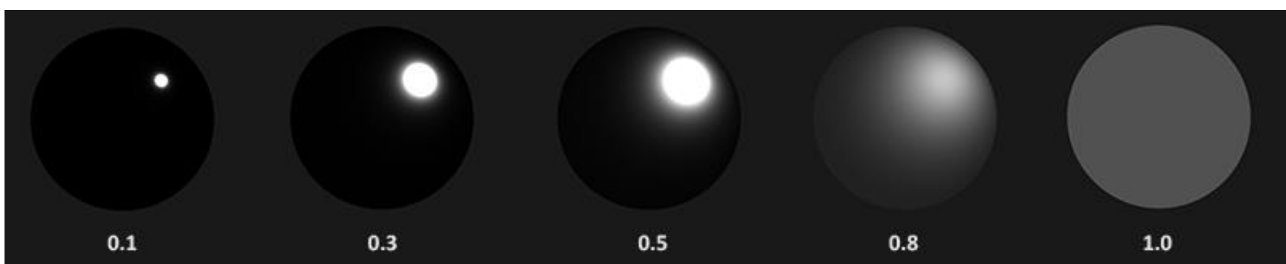


Figure 2.20 - A material with increasing value of roughness

Figure 2.21 - Diffusion and specularity mutual exclusion. Specular materials show less diffuse color

Metalness is another parameter that can be tweaked in a material to obtain different behaviors.

3. Use a physically based Bidirectional Reflective Distribution Function (BRDF). The BRDF is a function that approximates how a material reflects and refracts incoming light, based on the microfacet theory. To be physically based, the BRDF must adhere to the energy conservation principle. Blinn-Phong, which is also a BRDF, is not physically based since it is not energy conserving. Most of the real-time rendering pipelines based on PBR use the Cook-Torrance BRDF.

PBR is often used in combination with Image Based Lighting to produce even more realistic and physically accurate results [27]. This is usually done by transforming the environment into a cubemap that can be used in the lighting equations as a big light source. In this way it is possible to capture the environment's global illumination. The lighting equation solved by PBR (which is a more specialized version of the rendering equation, called *reflectance equation*) considers diffuse and specular lighting separately as they are independent from each other. Such equation is available in Figure 2.22 where the Cook-Torrance BRDF appears explicitly.

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

Figure 2.22 - The reflectance equation solved by PBR split in diffuse part (left integral) and specular part (right integral)

Diffuse integral

As the first term is constant with respect to the integral variables (c is the color, k_d is the refraction ratio, the whole term is known as the diffuse *Lambert term*), it is moved outside the integral, which now depends only on the direction ω_i of the incoming radiance, assuming p being the center of the environment map. From the integral, a new cubemap is pre-computed, storing at each texel ω_0 the result of the diffuse integral obtained by *convolution*. The convolution operation is done by sampling a high number of incoming directions ω_i over the hemisphere Ω and computing the average radiance. The hemisphere is oriented towards the direction ω_0 over which the convolution is being computed (Figure 2.23). The resulting cubemap is known as *irradiance map* (Figure 2.25) and allows to directly sample the pre-computed irradiance from any direction ω_0.

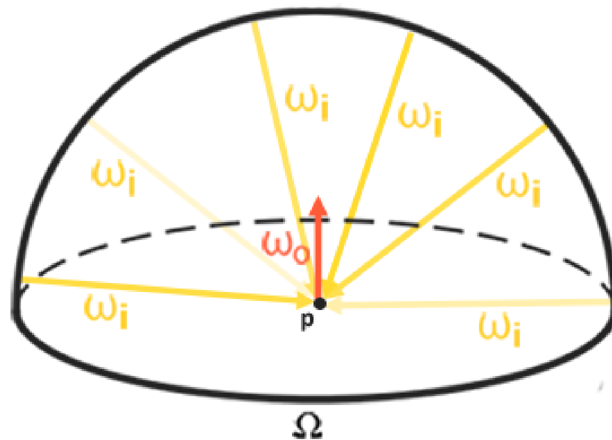


Figure 2.23 - Overview of the convolution operation of the diffuse integral. From [27]

Specular integral

The specular integral is more complicated to deal with, since it does not depend only on the incoming light direction, but also on the view direction [28]. To solve it, an approximation called split-sum is used to pre-convolute the specular part into two separate components that can be later recombined in real-time PBR. The split-sum approximation splits the specular integral into two integrals (Figure 2.24). The first integral is a pre-filtered environment map, which is similar to the

irradiance map, but it takes into consideration also roughness. The environment map is convoluted with more scattered vectors for increasing roughness, generating blurred reflections. For each level of roughness, a smaller environment map is stored as mipmap level (Figure 2.26).

$$L_0(p, \omega_0) = \int_{\Omega} L_i(p, \omega_i) d\omega_i * \int_{\Omega} \frac{DFG}{4(\omega_0 \cdot n)(\omega_i \cdot n)} n \cdot \omega_i d\omega_i$$

Figure 2.24 - The split-sum approximation of the specular integral

Figure 2.25 - An environment cubemap (left) with the resulting irradiance cubemap obtained by convolution (right)

Figure 2.26 - Pre-filtered environment maps of 5 levels of increasing roughness.

The second integral of the split sum approximation represents the BRDF of the surface. By pretending that $L(p, x) = 1.0$ (white radiance for every direction) it is possible to pre-calculate the BRDF's response given an input roughness and an input angle between the surface normal and the incoming light direction $n \cdot \omega_i$. A 2D lookup texture known as *BRDF integration map* is pre-computed, containing the BRDF's response to each normal and light direction combination, with varying roughness. The texture is available in Figure 2.27 with the horizontal axis being the input angle $n \cdot \omega_i$ and the vertical axis being the input roughness.

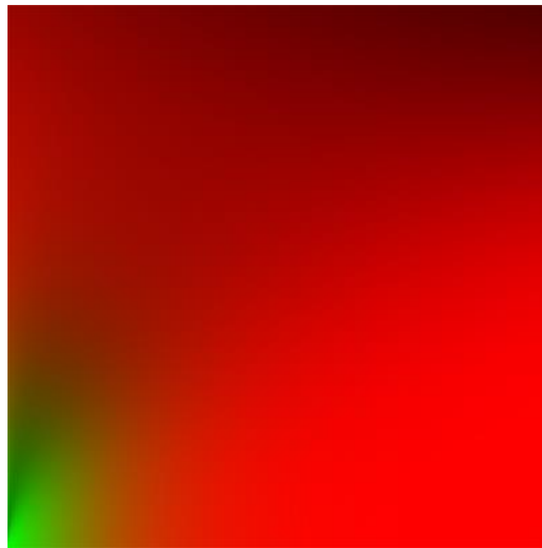


Figure 2.27 - The BRDF's integration map. The colors represent the scale (red) and the bias (green) with respect to the Fresnel response of the surface. From [28]

2.3 Conclusion

In this chapter an overview on the current trends about point clouds and rendering was given. When it comes to applications, most of the times point clouds are simply rendered as they were scanned, meaning that they look exactly as if they were under the same lighting conditions as during scan-time and the used shaders in rendering do not take into consideration incoming light at all. In many cases this is fine since there is no need to have a realistic look of the point cloud under different lighting conditions, however for other applications, for example related

to Virtual, Augmented and Mixed Reality (VR/AR/MR) it could be needed to render a model in different lighting conditions as realistically as possible to have a good integration of the model inside the scene, both for static and animated models. PBR with IBL looks a good suit for this task.

Some steps in this direction have already been made. The most notable studies have been carried out by Paul Debevec, researcher and professor at the University of Southern California's Institute for Creative Technologies [29], who developed with his team a scanning technology called LightStage which is able to scan human faces when lit from any possible lighting direction and render them faithfully in different realistic virtual conditions, obtaining quite impressive results (Figure 2.28). LightStage was launched in 2008 for commercial use and has been widely adopted in blockbusters films and triple-A game titles and allows the creation of photorealistic digital images as they would appear in any lighting condition [30, 31]. The goal of this paper is to extend the use of advanced rendering techniques in any lighting conditions to any model, not only to human faces subject of studies by Debevec's team.

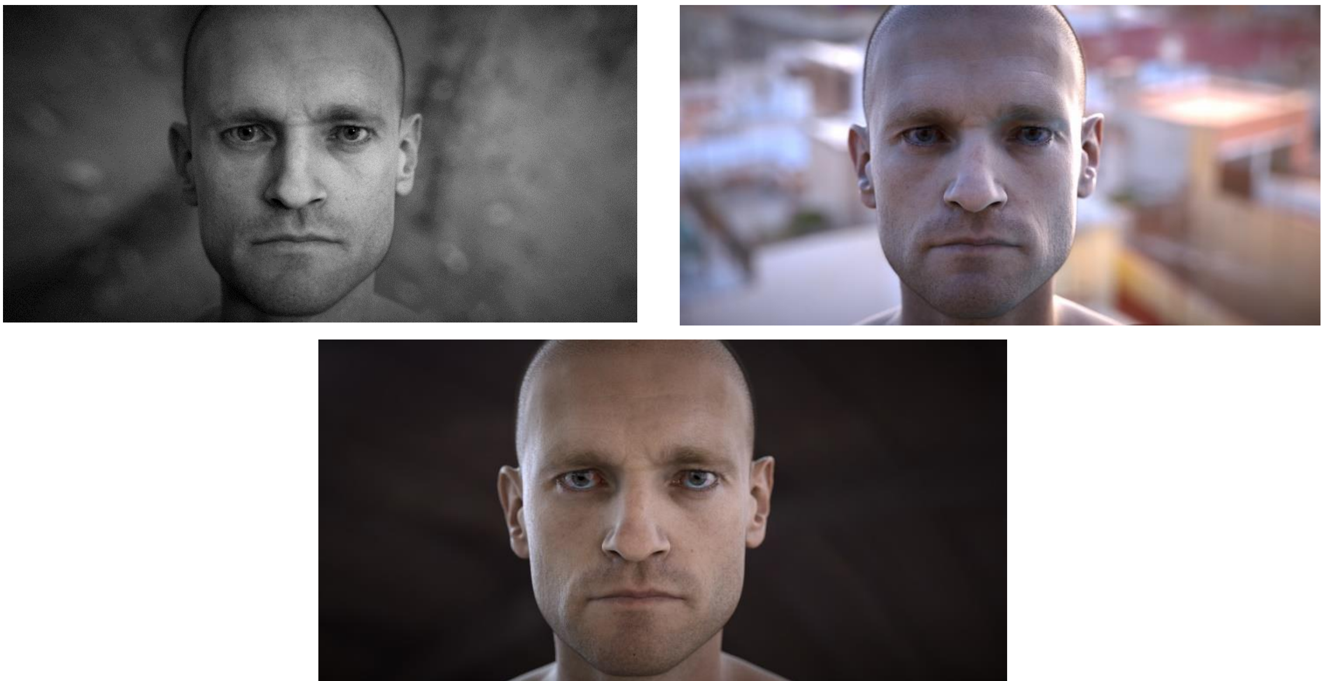


Figure 2.28 - A human face digitally rendered in three different lighting conditions with LightStage. From [30]

3 Physically Based Rendering of Animated Point Clouds

This chapter will provide a detailed description of the process used to visualize animated point clouds rendered with PBR and a comparison of the obtained results with classical, unlit point clouds. To conduct the experiments an animated point cloud will be used, where each frame is stored in a different file, to better study the impact of lighting on a non-static model. The specific models which are going to be used as reference come from the *basketball_player* sequence provided by [32], which is a collection of 600 files stored in .ply format acquired at 30 frames per second, thus resulting in an animation of 20 seconds. The software used for the experiments are:

- MeshLab and PyMeshLab 2021.10 for point cloud manipulation and fixing of normal vectors.
- Unity3D with the Long-Term Support version 2020.3.22f1 as the engine to render the scenes and compare results.

3.1 Point cloud normals check

The first step is to make sure that all the available point clouds come out with correctly estimated normals. Depending on the approach used to acquire and compress the point cloud, normals could have already been estimated or not. For example, in the case of V-PCC normal estimation happens during patch generation (as can be seen in Figure 2.5) and they can be saved and transmitted in that circumstance. Another approach is to estimate them right after scan-time in some way. An example can be found in [33] where normals are estimated from images acquired with the Time-of-Flight principle by depth cameras.

To make this check the software MeshLab is used, which allows to visualize the whole point cloud and the normals associated to its points. Also, it provides many filters that can be applied to any 3D model among which the one called “compute normals for point sets”, which tries to estimate the normal vector of each point in models without information about triangle connectivity, which is exactly what is needed for the purpose of this work.

Considering the *basketball_player* sequence it turns out that many frames of the animation are decorated with correct normal vectors but, in some frames, these have been erroneously estimated and are oriented inward the model, as can be seen in Figure 3.2. For these frames it is necessary to re-compute the normals to make them look correctly. However, since estimated normals for “correct” frames still make the models look a bit rough and “squared” (see as example Figure 3.1), all frames have been subjected to normal vectors re-computation, to make the models look as smooth as possible.

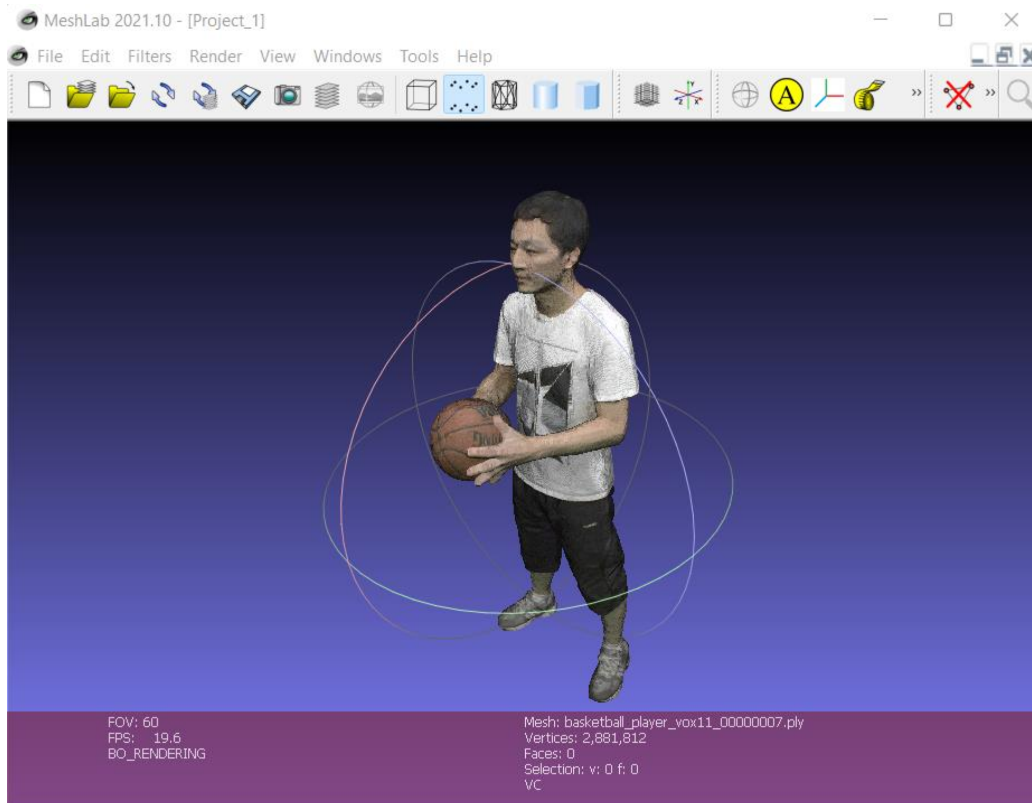


Figure 3.1 - A frame of the basketball_player sequence visualized with MeshLab

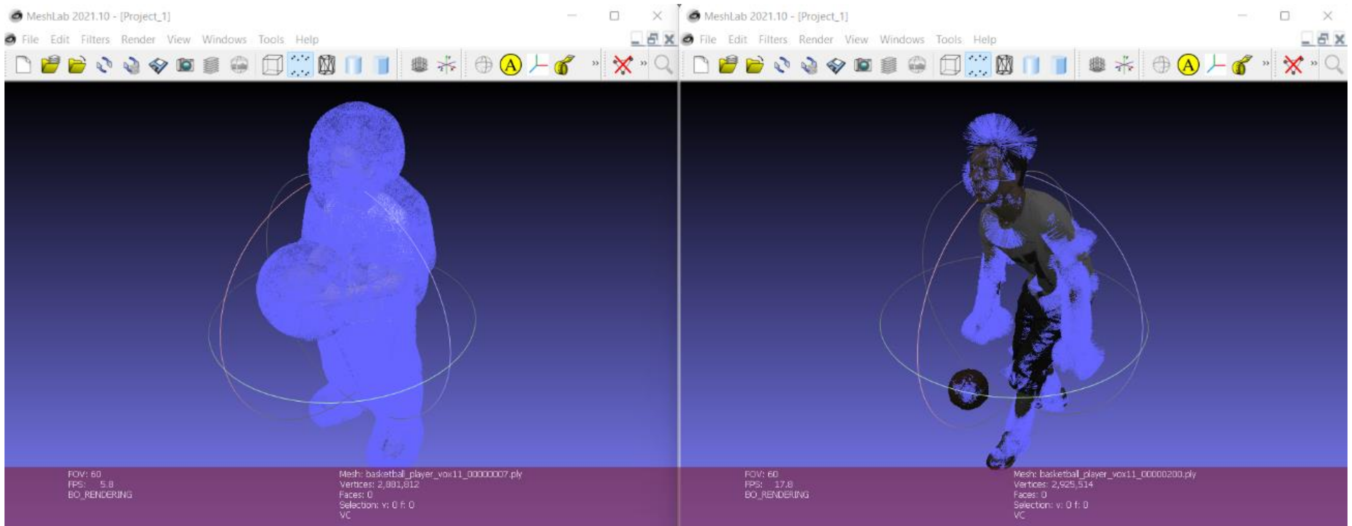


Figure 3.2 - Different frames of the basketball_player sequence with correct (left) and erroneous (right) normal estimation

The operation is performed with the already cited filter called "compute normals for point sets" available in MeshLab. The steps followed by the filter can be synthetized as:

1. *kNN computation:* For each point of the point cloud, get its k nearest neighbors, where k is an input parameter of the procedure chosen before launching the filter.
2. *Normal estimation:* Estimate the plane tangent to the model surface using the neighborhood of the query point and compute its normal. This can be easily done through Principal Component Analysis (PCA).
3. *Orientation disambiguation:* Pick a random point, choose an orientation (inside/outside) and propagate it to nearby points.

The last step is critical since generally there is no easy way to disambiguate normal orientation in an automatic way. If we had a single viewpoint to observe the model from, the problem would not arise since all normals would be consistently oriented towards the viewpoint itself. This solution is not suited for this application as the goal is being able to cast light on the model from any possible direction.

Regarding the choice of the parameter k defining the size of the neighborhood to fit the tangent plane, low values are usually suggested to better capture the local curvature of the model. If k is too large, points from different surfaces might be

included in the neighborhood, resulting in distortion of the plane and badly estimated normals, especially around corners. MeshLab puts as default $k = 10$ and suggests values between 10 and 30 for a good estimation.

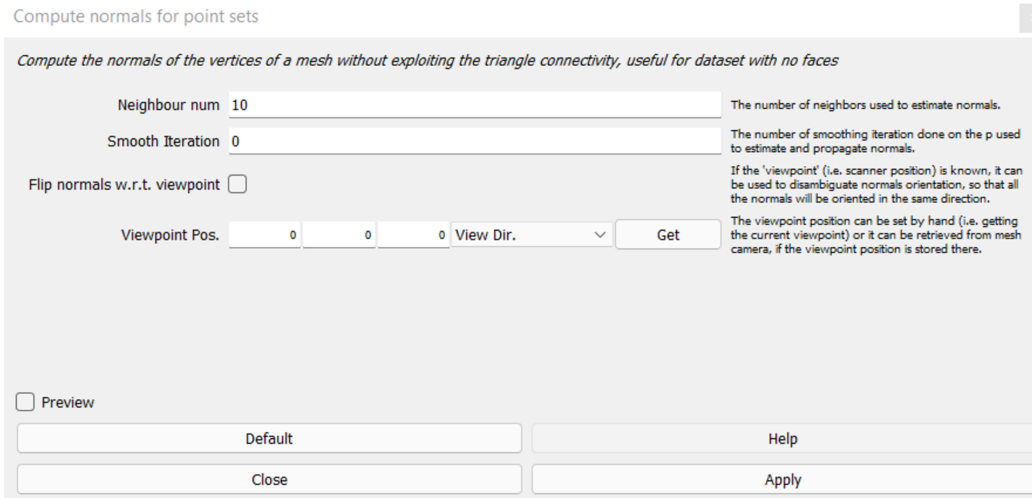


Figure 3.3 - MeshLab filter to compute vertex normals of faceless models

For this work, the suggested values for k worked fine. Also, some smoothing iterations have been performed to obtain better results. To be more precise, a number of smoothing iterations between 10 and 15 have been used, giving the model a smoother appearance (Figure 3.5).

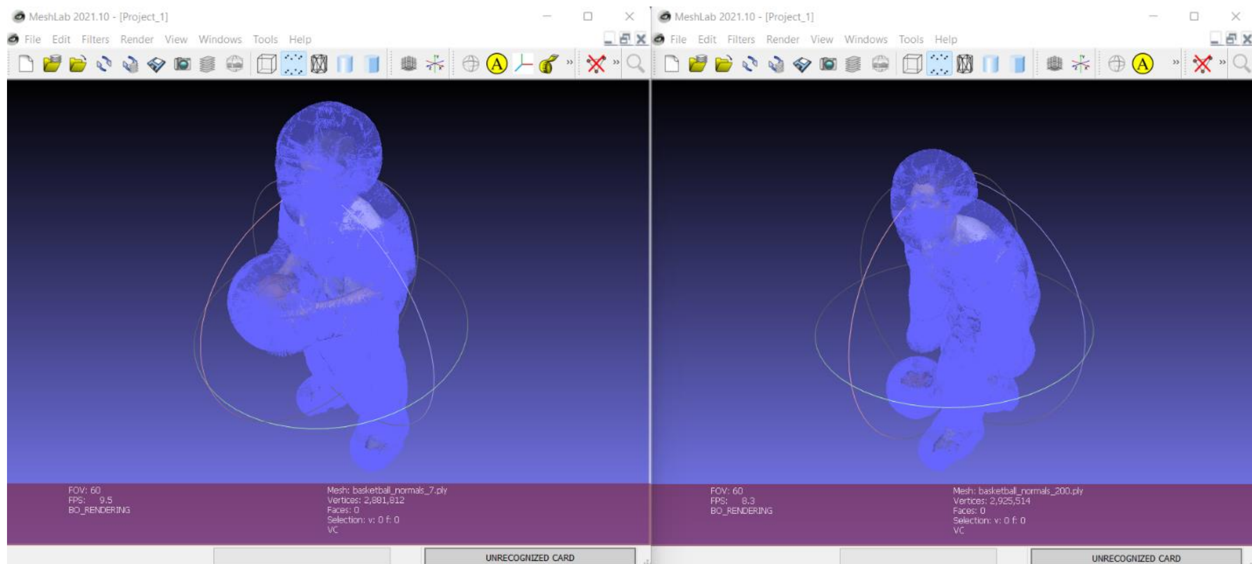


Figure 3.4 - The same models as Figure 3.2 after normal re-computation

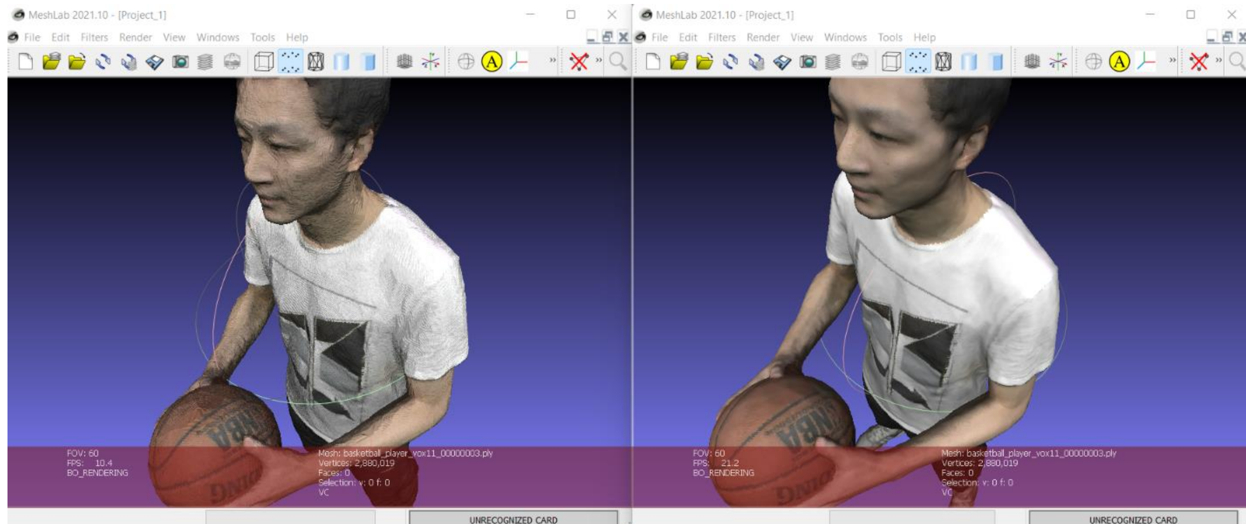


Figure 3.5 – A model with normals computed without smoothing (left) and with 15 iterations of smoothing (right) with $k = 10$

At the end of this step all frames from the *basketball_player* sequence have correct normal vectors associated to their points and are stored as .ply files ready to be imported in Unity and rendered in a scene.

3.2 Importing models into Unity

As previously mentioned, the point clouds are stored in .ply binary format, which has as advantage a relatively small requirement in terms of storage occupancy. However, Unity has no native package allowing to import and directly use in a project a .ply asset, so it is necessary to use a custom importer to load the point clouds into the project. For this purpose, a package available on GitHub called Pcx [34] developed by the user Keijiro has been added to the project and used to import the models.

The package provides useful tools to work with .ply files and, more specifically, point clouds inside Unity. The importer reads a binary little endian .ply file and parses its content into an object the engine can work with. After parsing, the model can be wrapped into three different containers provided by Unity:

- A *Mesh:* the point cloud is treated as a standard 3D mesh with just vertices and colors (no faces) and can be rendered with a standard MeshRenderer component with any shader.
- A *ComputeBuffer:* a buffer for data mostly used with compute shaders, which are a special kind of shaders used to compute arbitrary information not directly related to rendering.
- A *Texture:* points are baked into 2D textures (a position map and a color map) that can be used as attribute maps for visual effects.

For the purpose of this work the most suited container to wrap point clouds in and which is going to be used is the Mesh. The other two containers will not be considered any further.

The package also provides scripts to manage imported meshes from the inspector window in Unity and two basic shaders to render point clouds. A first basic scene with an imported frame placed on a simple planar surface and rendered with the basic shader is available in Figure 3.6.

The first thing that can be noted is that the shader provided in the imported package does not draw any kind of shadow of the model on the plane, since the work of the author was meant just to visualize point clouds as they were scanned without integrating them in a different environment.

Figure 3.6 - Basic scene consisting of a single frame placed on a surface in Unity

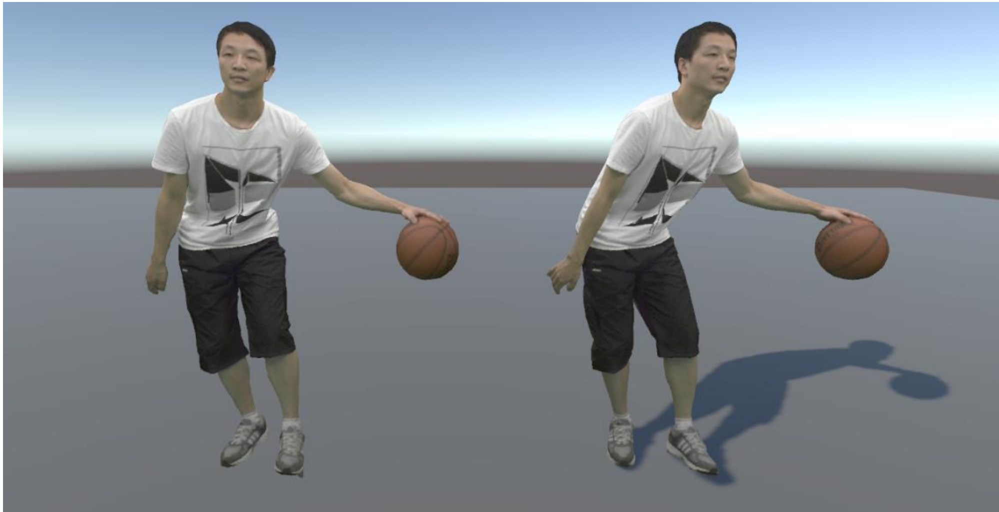


Figure 3.7 - A frame rendered with the basic shader (left) and the new shader with hard shadows (right)

Therefore, to have a slightly better baseline to allow a better visual understanding of the lighting conditions and to compare the PBR shader that will be developed later with, another shader has been created and used that behaves exactly as the one provided in the Pcx package with the addition of shadows.

To speed up the writing process of shaders and have an easier control over various rendering parameters, the Universal Render Pipeline (URP) with Shader Graph support has been enabled. In Figure 3.7 the two shaders are applied to the same

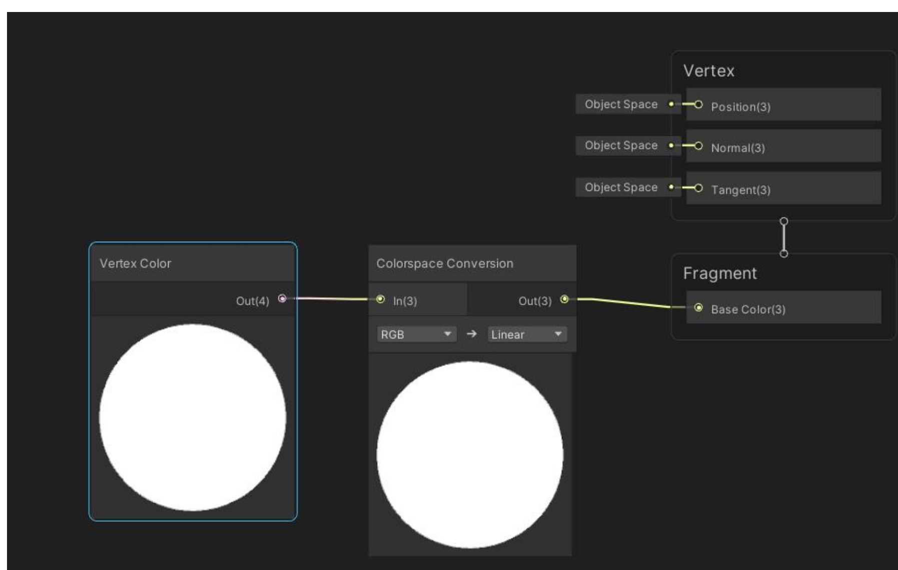


Figure 3.8 – Basic unlit shader allowing hard shadow casting

frame to highlight the difference when rendered, while the shader graph can be viewed in Figure 3.8.

The Colorspace Conversion node is necessary since URP is being used instead of the standard Unity built-in pipeline. While the built-in render pipeline uses a linear color space with sRGB light intensity, URP applies linear light intensity and since vertex colors stored in the used models are RGB their look results very bright and need to be corrected with the conversion to linear space. The difference between the two color spaces is shown in Figure 3.9.

Figure 3.9 - A frame rendered with (left) and without (right) color space conversion to linear space

3.3 Towards PBR

Having set a reference baseline to compare the final results with, from now on the focus will be on developing a PBR material to render the model in a physically plausible way in a scene.

Vertex normals are a key element to implement the technique correctly, so in order to make sure that these are imported and used as expected by the engine, another shader is built first to generate a normal map of the model. To visualize a smoother

color transition, instead of using pure normals as vertex color, a normalization is applied to transform all normal components in the [0, 1] interval as following:

$$N(x, y, z) = \begin{cases} n'_x = \dfrac{n_x + 1}{2} \\ n'_y = \dfrac{n_y + 1}{2} \\ n'_z = \dfrac{n_z + 1}{2} \end{cases}$$

As reference model a sphere will be used, which when rendered with the normal map shader looks as in Figure 3.10.

Unfortunately, when the shader is applied to the point cloud model, the behavior is not the desired one, but the shape comes out with a plain, fixed blueish color (Figure 3.11). This means that normal vectors are not correctly acquired or imported. The issue was not evident before since, when using an unlit shader, the model is rendered by just taking into consideration the color of vertices without taking into any consideration light and view directions and, consequently, normal vectors. Before proceeding to develop a correct PBR shader, this issue must be inspected and solved.

Figure 3.10 - Normal map shader applied to a sphere in front (left) and back (right) view

The issue most likely arises when the model is imported into Unity, since in MeshLab it was possible to render on screen all normal vectors stored on the point cloud vertices as done in Figure 3.3 and Figure 3.4. As a double-check, one of the frames has been imported again into MeshLab and saved in ASCII format instead

of binary and manually inspected, confirming that normal vectors are stored as expected. Therefore, the problem is due to the .ply importer not parsing normal vectors information and thus it needs to be extended to be able to save and use them in Unity.

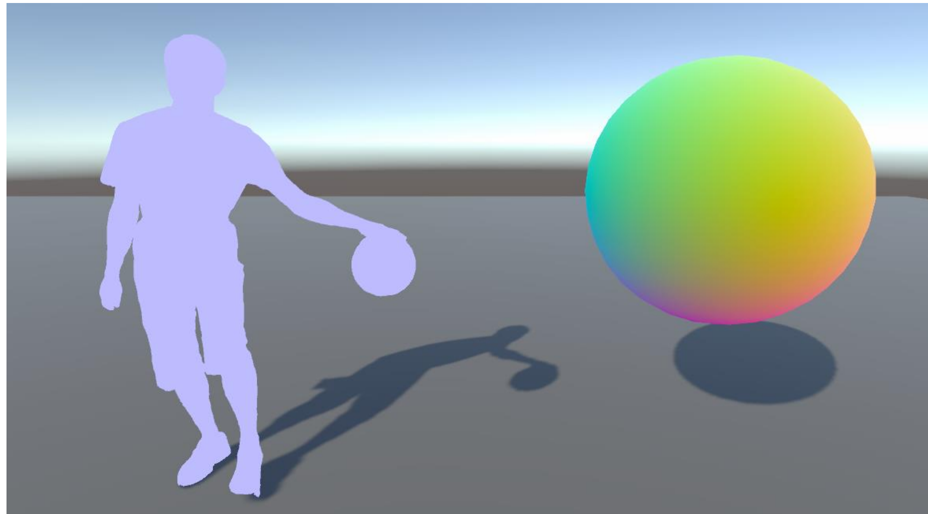


Figure 3.11 - Normal map applied to the imported model with the reference sphere. Normals are not correctly imported.

In its original version, the importer is built to process just vertices position coordinates (x, y, z) and RGBA color (red, green, blue, alpha), whereas all properties not corresponding to the mentioned ones are marked as invalid and skipped. The parser has been modified to recognize the presence of normal vectors information from the file header and parse the values associated to normal components, storing them in a list of 3D vectors later assigned to the Mesh object, coherently with respect to how the author dealt with positions and colors. Since the work focuses on the use of meshes only, this is the only container to which the modification has been made, while ComputeBuffers and Textures have been ignored.

After having adapted the importer, the point clouds must be imported again to allow Unity to store normals together with position and color information and use them for rendering. By reapplying the normal mapping shader to the reimported models, the behavior is now the expected one, as Figure 3.12 demonstrates.

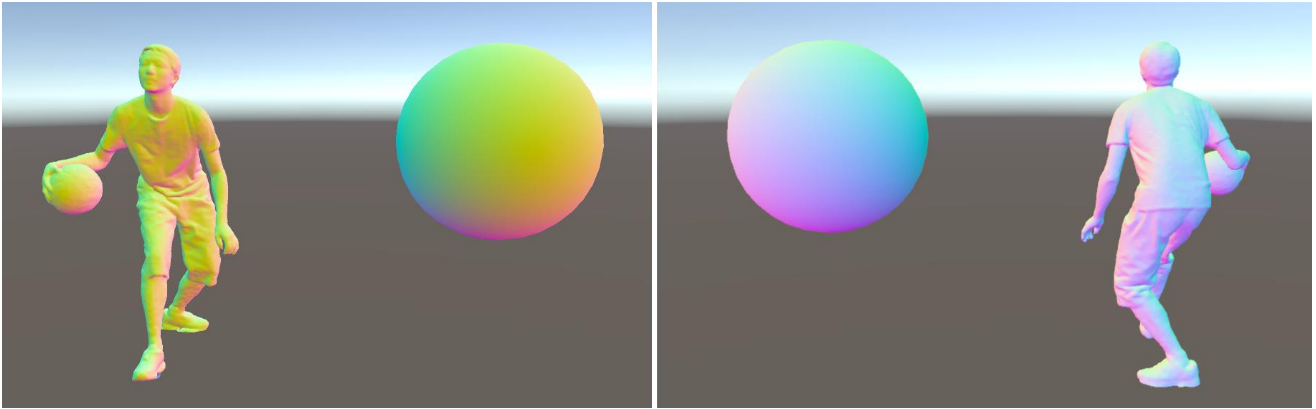


Figure 3.12 - Normal map shader applied to models with the modified importer in front(left) and back(right) view, view the reference sphere

Having fixed the issue about normal vectors, it is now possible to proceed with the development of a PBR shader and apply it to the material assigned to the basketball player model for photorealistic rendering.

A new lit shader graph is hence created. The fragment stage node for a lit shader contains all the standard parameters cited during the discussion about PBR rendering in Section 2.2.1:

- The *base color* parameter, which is the albedo of the material, also present in unlit shaders (Figure 3.8).
- The *normal vectors* of the material set as default in tangent space, but they can also be represented in object or world space.
- A *metalness* parameter defining how much metallic the surface of the material is. As default it is represented by a floating-point value between 0 and 1, but it can also be sampled from a texture to render more complex objects with points of varying metalness along the surface.
- A *smoothness* parameter, which is the inverse of roughness, defining how smooth the surface is. As for metalness it is by default defined by a floating-point value between 0 and 1 but can also be sampled from a texture.
- An *ambient occlusion* parameter measuring how obscured the object is from light by other objects in the scene. Again, it can be represented by a floating-point value or a texture.

- An *emission* parameter defining the emitted light color by the material. It takes as input an HDR color with intensity. For non emissive materials, a black color is used as emission. For complex objects having different emitted colors, textures can be used.

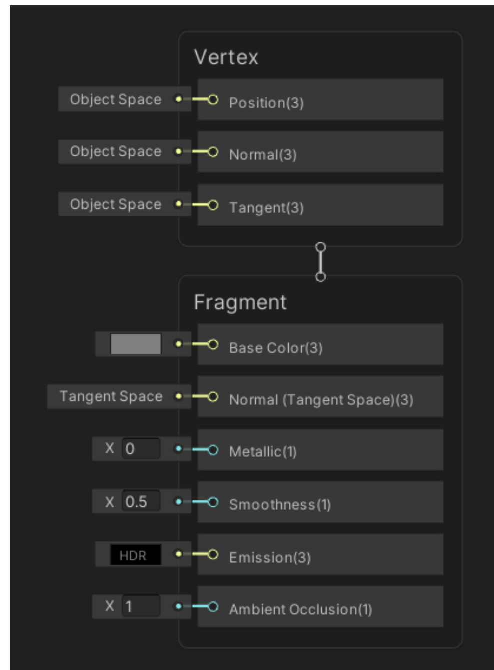


Figure 3.13 - Default lit shader in Shader Graph

Regarding the shader used for this work, the parameters have been set as follow:

- The albedo is set the same as for the previously built unlit shader, with the color given by vertices and colorspace conversion to linear space.
- The fragment normal space is set in Object space.
- Metalness and smoothness are set as properties to allow easier and faster customization from the Unity inspector and linked to a Clamp node to limit the values in the [0, 1] interval. As default both values have been put to 0, which is quite reasonable to realistically render the available basketball player model.
- There is no interest in showing an emissive color from the used model, so the emission color is put to full black.

➢ Ambient occlusion is also set as a property and will be tweaked accordingly to the scene in which the model is placed to better simulate the lighting intensity.

The final PBR shader, which is rather simple but fully complies with our needs, is shown in Figure 3.14. The point cloud rendered with a material using such shader is available in Figure 3.15. Even in a simple scene like the one used for this very first example, a notable increase in realism can be appreciated, with many little shadows occurring on the whole model. Also, many small details that were less evident with the standard unlit shader previously used can now more easily distinguished, like the folds of the player's t-shirt seen from the back view. Most importantly, the model now looks different with respect to the incoming light direction and intensity, as opposed to the previous one which instead looks exactly the same under any lighting condition (Figure 3.16).

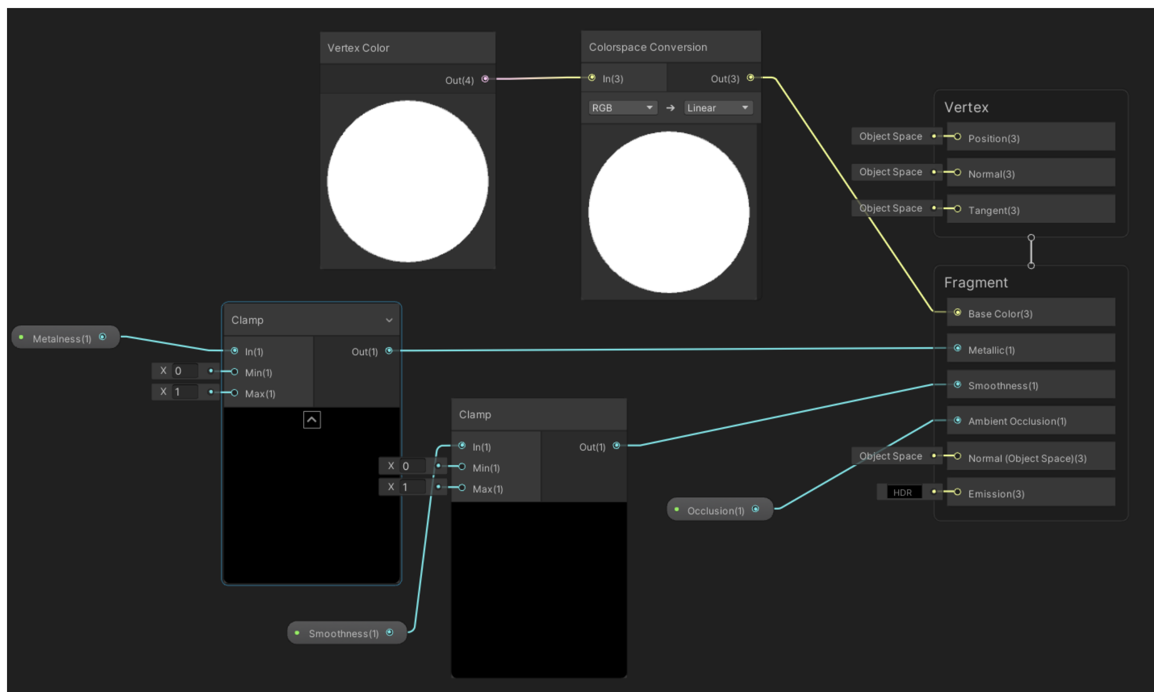


Figure 3.14 - Graph of the PBR shader that will be used to render the basketball player model in a physically plausible way

In the next sections the focus will be on trying to fit the PBR shaded point cloud in different scenes and backgrounds with various kind of lighting conditions and demonstrate how the model rendered with this technique integrates better with respect to a standard unlit shader normally used for point clouds.

Figure 3.15 - Model rendered with the PBR shader seen from different viewpoints under the same lighting condition

Figure 3.16 - Comparison between PBR and unlit shaded point clouds under different lighting conditions

3.4 Test scene: background 360° video of an indoor basketball court

In this first test scene the objective is to try to better contextualize the point cloud model by inserting it into an environment it might be found in. The background choice fell on a 360 degrees video uploaded on YouTube by the Columbia International University representing an indoor basketball court with some people playing around [35]. The main goal is to check if it is possible to achieve realism for an application like Virtual Reality by placing the lit point cloud in a pre-recorded environment with the possibility to observe the model from any viewpoint.

The video comes up in the equirectangular format, so it can be easily rendered on the scene's skybox using a Render Texture and the built-in Skybox/Panoramic shader. By creating a new material using this shader and applying it to the skybox, the video can be played by a video player component placed in the scene and visualized in 360 degrees as in a Virtual Reality setting.

Figure 3.17 - A snapshot of the indoor basketball court video in equirectangular format

The next step is to create a script that takes the frames of the model and generates the point cloud animation at runtime. Since each frame of the animation is stored in a different mesh container, the easiest solution to create the animation is to load all meshes in a list data structure and update at each frame the mesh used by the

Figure 3.18 – PBR Point cloud model placed in a scene with the indoor 360 video rendered on the skybox

MeshFilter component in Unity to change the mesh drawn on the screen and use a variable to store the current index of the mesh used in the list. This results in changing the rendered model at each frame, thus recreating the animation.

Now that the background and animation are set, it is possible to take the animated point cloud with PBR shader and insert it into the environment to check how it fits.

A few issues in the rendered scene can be spotted related to the integration of the point cloud with the video:

- Although the figures belonging to the background video slightly show some weak shadows, regarding the point cloud no shadow at all is visible.

Figure 3.19 - Background depth issue: figures in background don't follow the point cloud behavior

Figure 3.20 - Fluctuation issue: when rotating the camera it looks like the player fluctuates over the court

- When the camera is moved towards or away from the point cloud, there is no sense of depth coming from the video: while the point cloud correctly becomes smaller or bigger accordingly to the camera movement, the background remains exactly the same (Figure 3.19).
- Another issue related to camera movement, if the viewpoint is rotated around the point cloud to see the animation from a different angle there is no feeling of the model being on a fixed spot but there is an illusion of fluctuation of the figure over the court. The issue can already be observed in Figure 3.19 where the camera is just moved forward but it is more evident by rotating the camera as in Figure 3.20.

Figure 3.21 - Depth issue: the point cloud is rendered on top of a character that should be in front of it

- Another issue related to depth, being the video rendered on the skybox its content will always be drawn behind the 3D models placed in the scene. So, if one of the characters in the video walks “in front” of the point cloud, the point cloud will still be rendered in the foreground (Figure 3.21).

The shadows related issue could be solved by putting other 3D models in the scene, like a plane, where the shadow would be projected on. This would, however, break the realism of the scene since the newly added models would be rendered on top of the background, resulting in a very strange look (see Figure 3.22 as an example). For the same reason, in this setting nothing can be done to solve the other issues as well. The Unity’s skybox represents the content of the scene placed on an infinite distance from the camera and it will always be rendered behind any other model inserted in the scene.

Figure 3.22 - The insertion of a plane in the scene covers the background video

In conclusion, it is not possible to render a realistic scene by just using a background video on the skybox and a 3D point cloud with a dynamic camera with six degrees of freedom. The camera translation would break the realism of the scene, while a camera with only rotation enabled might still work in some circumstances. To produce realistic scenes, other 3D models need to be placed together with the point cloud, with the background having a content on a sufficient distance to prevent unrealistic behaviors when the images overlap.

3.5 Test scene: indoor room with different lights

Before testing another scene with a background video, another scene is tested, with just 3D models besides the point cloud. By taking inspiration from the typical room used to test global illumination algorithms (like the one shown in Figure 2.19), an indoor room is created by linking together some cubes. Then, three planes emitting light of different colors are used, which act as the only light sources in the scene. The goal is to demonstrate how the point cloud rendered with PBR is influenced by the light emitters while there is no effect on the unlit point cloud. An overview of the scene without the models is available in Figure 3.23.

In the figures on the next page it will be shown, by placing the two models in different positions of the room, how the PBR point cloud integrates well with the environment by showing a color tending to nearby lights whereas the other point cloud is not influenced by light and doesn't look realistic. It is shown on the left of each figure the PBR point cloud while on the right the unlit point cloud. Also, to provide a better look to the scene, some post processing effects have been added, like Tonemapping and Bloom.

As already shown in the very first scene used to compare PBR and unlit point clouds (Figure 3.15-16), the model rendered with PBR is effectively able to react

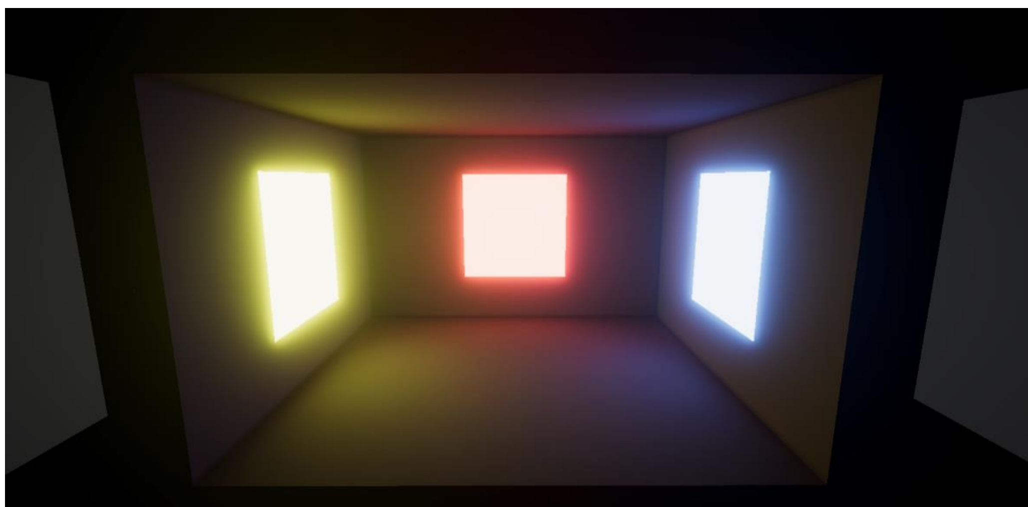


Figure 3.23 - Representation of a room with three different light sources of different color

to environment lighting when placed nearby a light source, getting brighter or darker according to the distance from lights.

In the next scenes more complex environments will be used, with a 3D model representing a full basketball court and background videos providing better context and global illumination.

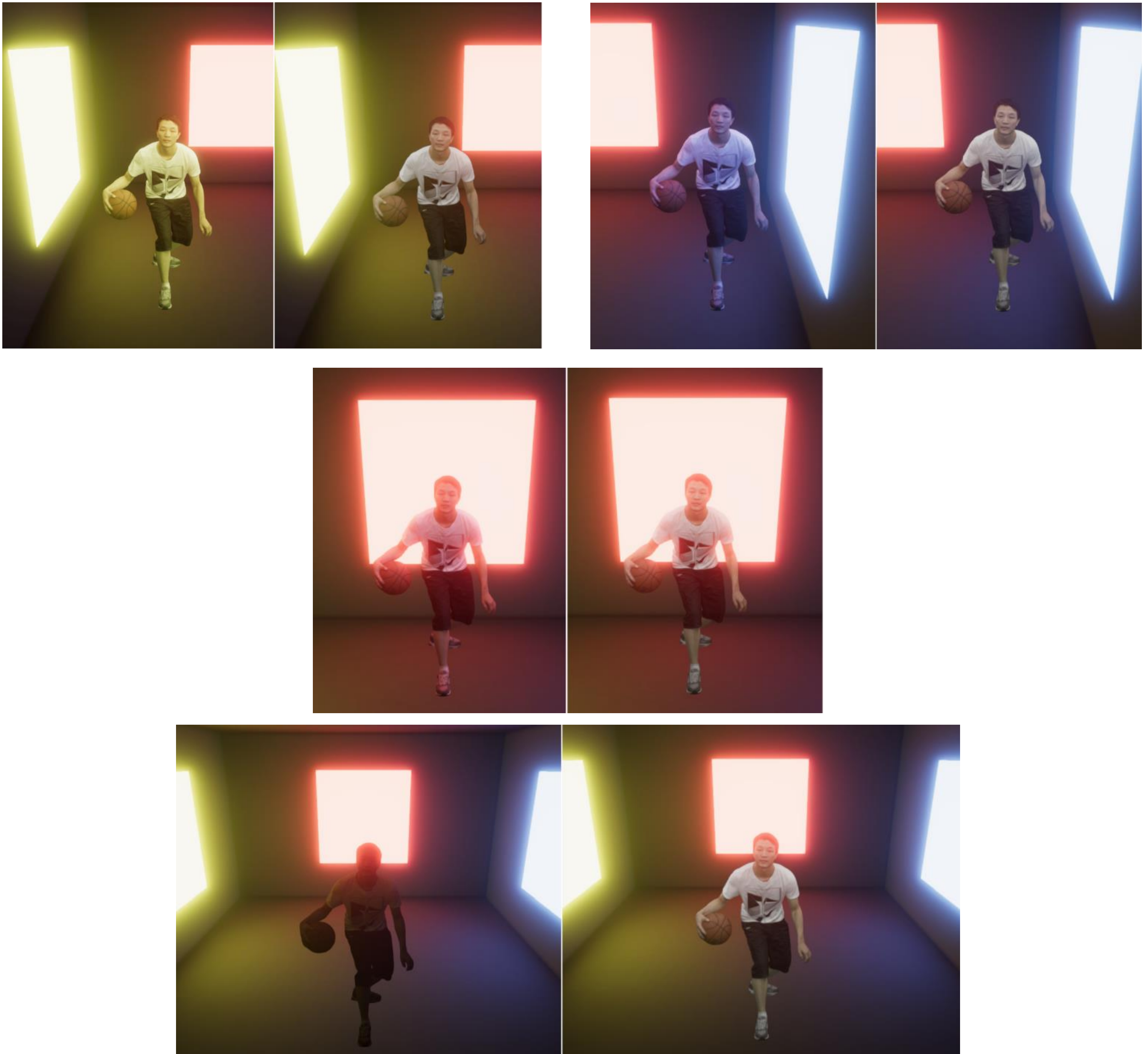


Figure 3.24 - Comparison between the visual look of the PBR and standard point clouds placed in different locations of the scene

3.6 Test scene: outdoor basketball court in a daylight environment

As previously stated, to obtain a realistic scene a good context is needed to place the point cloud in. Since a background video alone is, as demonstrated in Section 3.4, unusable due to the issues it introduces, other 3D models are needed to be placed in fore-midground to contextualize the point cloud model, while the background needs to contain subjects distant enough not to interfere with the models.

Regarding the 3D model, the choice went to an outdoor basketball court created by the SketchFab user Klieg3D [36], and the same model will be used for all the next scenes. The model is interesting for a series of reasons, the most important being:

- It is a basketball court, which is the natural environment in which the specific point cloud model in use would be found in.
- It has been ideated as an outdoor court, which gives wider alternative choices about the background to use in the scene, being outdoor videos less likely to interfere with the animations (e.g. in closed environments it would be more likely to have people walking too "close" to the 3D models and overlap).
- It is surrounded by a grate, which allows to produce interesting shadow patterns under some lighting conditions.
- It has four lamps, which gives the opportunity to play with very different lighting conditions switching from daylight to artificial lighting provided by such lamps in a night setting.

For this scene a daylight setting will be used and as background a 360 degrees video recorded on a beach in California with some beach volley fields has been chosen [37]. Like the previous video used in Section 3.4, this is also in equirectangular format and can be rendered in the skybox in the same way.

Figure 3.25 - 3D model of the outdoor basketball court with a directional light

The lights and lighting settings of the scene are applied as follows:

- Only a single, directional light is used as light source, except for ambient lighting given by the skybox. The light direction is oriented in a way to make it consistent with the position of the sun in the background video. Also, the environmental lighting intensity given by the skybox is increased to make the scene a little brighter and better simulate daylight.
- The occlusion parameter of the lit point cloud material is increased to 0.45 to make it a little brighter as well.

Figure 3.26 - Snapshot of the 360 degrees beach video in equirectangular format

- Post processing effects are applied to the scene, in particular:
    - Tonemapping with ACES mode.

- Bloom with an intensity of 10 to make the sunlight glow.
- A low intensity vignette effect.
- Some post exposure from the color adjustments effect, to furtherly increase the global brightness of the scene.

With these settings, the point clouds with the two different materials subjects of comparison are finally inserted in the scene for visualization. In Figure 3.27 the point cloud with PBR material is placed on the left, while the one on the right is unlit. Although the unlit point cloud, being very bright, suits quite well the surrounding environment, it still is not as realistic as the point cloud rendered with PBR which shows a darker appearance due to occlusions and self-occlusions of the model, noticeable on the neck of the player, on the t-shirt and on the lower part of the ball. In the figures on the next page we also try to change viewpoint and zoom in the models to visualize and compare smaller details. As in the previous comparisons, the point cloud rendered with PBR is placed on the left while the unlit point cloud is on the right.

As already pointed out in previous sections, when using PBR many smaller details become more distinguishable, like the t-shirt folds which due to the harder shadows produced by self-occlusion become more highlighted, for example regarding the upper part of the clothing in the first, second and fourth comparison.

Figure 3.27 - First comparison between PBR and unlit point clouds in daylight condition

Figure 3.28 - Comparison between PBR and unlit point clouds from different viewpoints

In the first comparison it is also clear how, with PBR, the incoming light direction is projected on the model, being the left side of the player in shadow, while this is not happening on the unlit point cloud. The same happens in the third comparison, where the point cloud rendered with PBR appears darker as light hits the model from the opposite direction. Self-occlusions are also visible in other parts of the model rendered with PBR, for example in the second comparison where the player's arm projects a shadow on the t-shirt, and in the fourth comparison where the player's arm also projects a stronger shadow on the ball with respect to the model using the unlit shader.

Additional comparisons can be made to check how the model reacts when it's occluded by other objects in the scene. In Figure 3.29 two different comparisons are made.

In the first one the model is placed under the basket support, trying to shield the player from light as much as possible. It can be clearly noted how the PBR model becomes darker due to the projected shadow except for small areas on the player's hand and leg, while the behavior of the second model doesn't correspond to what expected in this condition.

The same happens in the second comparison, where the light direction is slightly modified to stretch the shadow produced by the grates surrounding the court and the overall ambient light intensity has been lowered to highlight the shadows even more. When placed in the shadowed zone produced by the grate, the PBR model produces an interesting shadow pattern which corresponds to what happens in a real setting, while the second model does not show the same behavior.

In conclusion we can say that in a daylight environment the PBR point cloud is more suited to realistically reflect the external lighting condition with respect to a point cloud using a standard shader, both by exploiting self-occlusions to create photorealistic shadows and occlusions due to other objects in the scene.

Figure 3.29 - Comparison between the models when occluded by the basket support and the grate

3.7 Test scene: outdoor basketball court in a night environment

In this new scene the lighting conditions and background are modified in order to recreate a night environment. Regarding the background, a video recorded in an empty city square at night is used [38]. However, this video is not stored in equirectangular format as the previous ones, but in a different one called *equiangular* format, which projects a 360 image on a cubemap instead of a sphere. Thus, the video cannot be rendered on the skybox as done in the previous cases, but it is necessary to unpack the cubemap and assign its faces in the correct positions to recreate the original 360 video during rendering. The actual shader implementation is taken from GitHub [39] with a minor change to render a 2D

projection instead of 3D. By using a material associated to this shader, the background can now be rendered on the skybox as done in the previous scenes.

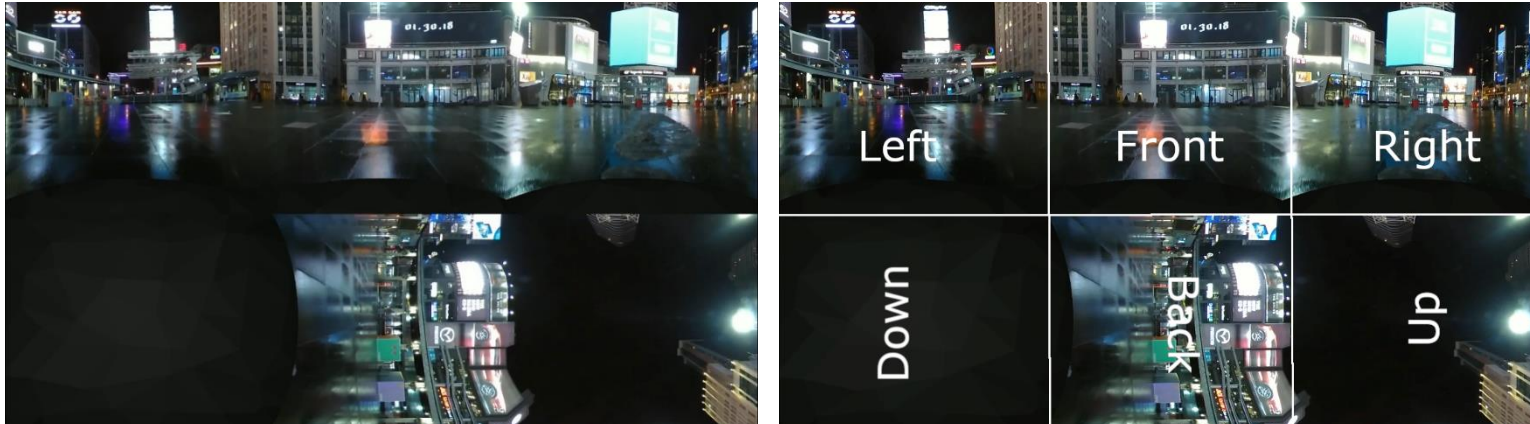


Figure 3.30 - Frame of the night square video in equiangular format and its cubemap projection

The scene lights and settings are applied as follows:

- No directional lights are used. Instead, we take advantage of the 3D court model structure and a spotlight is placed on each of the four lamps on the court sides. To illuminate the court in the best possible way, the spotlights are not completely oriented in the vertical direction, but they are slightly directed towards the center of the court. We also place a strong point light at the source position of each spotlight to better simulate the glowing effect of light emitted from lamps.
- Ambient lighting intensity given by the background is set slightly higher than 1.0 to avoid having the areas not reached by light completely black.
- The occlusion parameter of the PBR material is kept unaltered from the previous scene.
- Post processing effects are applied:
  - Tonemapping with ACES mode.
  - Bloom with a higher intensity than in the previous scene (around 50.0)
  - Vignette with the same parameters as the previous scene.
  - No post exposure is applied.

Figure 3.31 - Basketball court under night lighting conditions

After light baking the final look of the scene is shown in Figure 3.31. Everything is now set up and ready for the insertion of the models and make the usual comparisons.

For the first comparison the two models are put on the center of the emission cone of one of the spotlights. It is immediately evident how the point cloud using the unlit shader does not really fit with the environment, being very bright, while with PBR the point cloud is highly influenced by the dark lighting condition, with strong shadows on the right side, where no direct lighting reaches the player from. If the orientation of the model is slightly modified to face the light, as expected the model using PBR becomes more evenly bright.

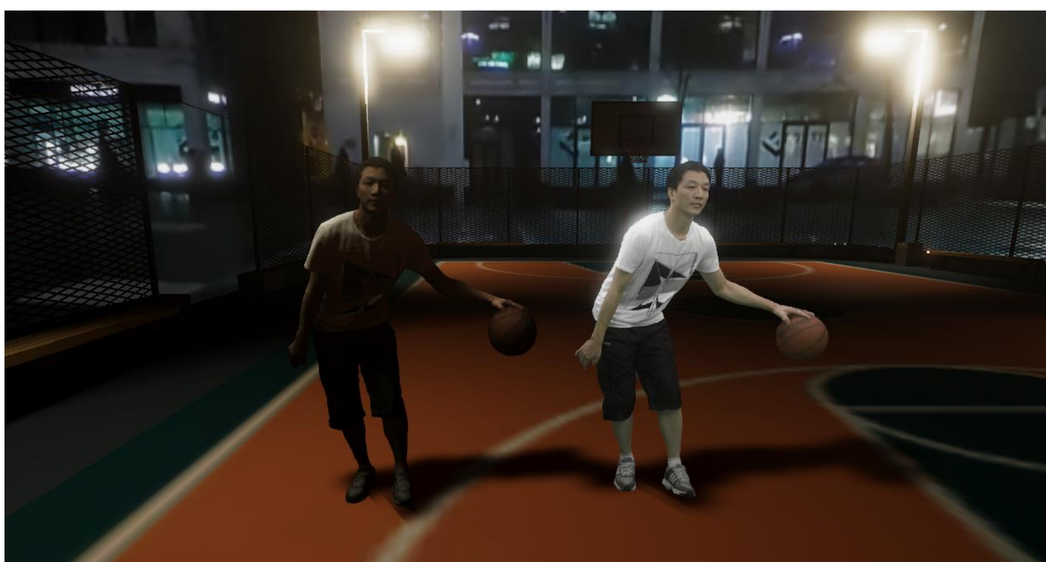


Figure 3.32 - First comparison between PBR and unlit point clouds in night condition

Figure 3.33 - Comparison between PBR and unlit point clouds when they face one of the spotlights

Then the two models can be placed in different positions of the court to see how light fades out by moving away from light sources, until the PBR model gets almost black at the middle of the court, and then back illuminated on the other side.

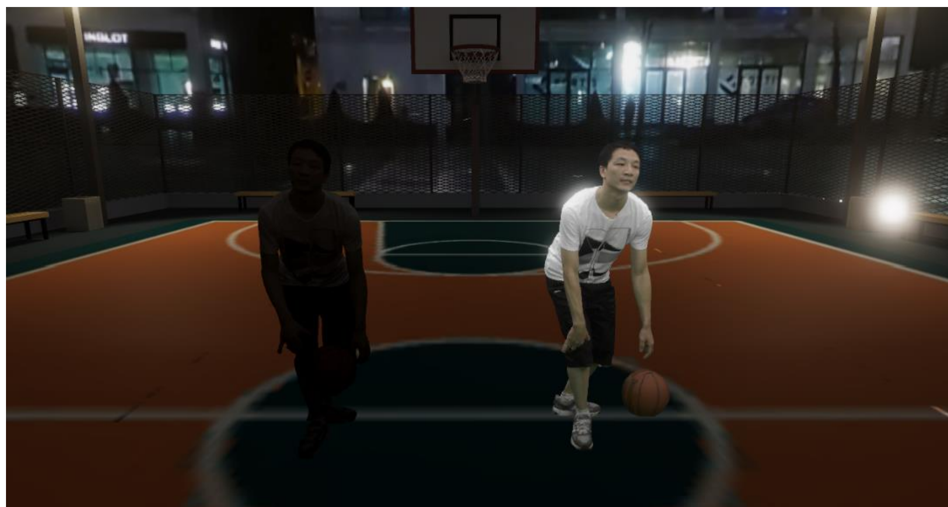


Figure 3.34 - Point clouds placed in different locations of the court

In conclusion we can say that also in a night environment a point cloud rendered with PBR is better integrated into the scene with respect to the same point cloud rendered with a standard shader, which results in being too bright and not following the expected behavior in such lighting conditions as opposed to PBR which, instead, does.

3.8 Test scene: playing with metalness and smoothness parameters

As mentioned in Section 2.2.1, one of the most powerful features provided by PBR is its capability of emulating metallic and glass-like materials, thanks to the microfacet theory and the energy-conserving principle. Until now we have been using a PBR point cloud with both metalness and smoothness parameters equal to zero, which is reasonable with respect to the nature of the represented model and the purpose of the previous scenes, which was trying to integrate a point cloud rendered with PBR in an environment to obtain a photorealistic result.

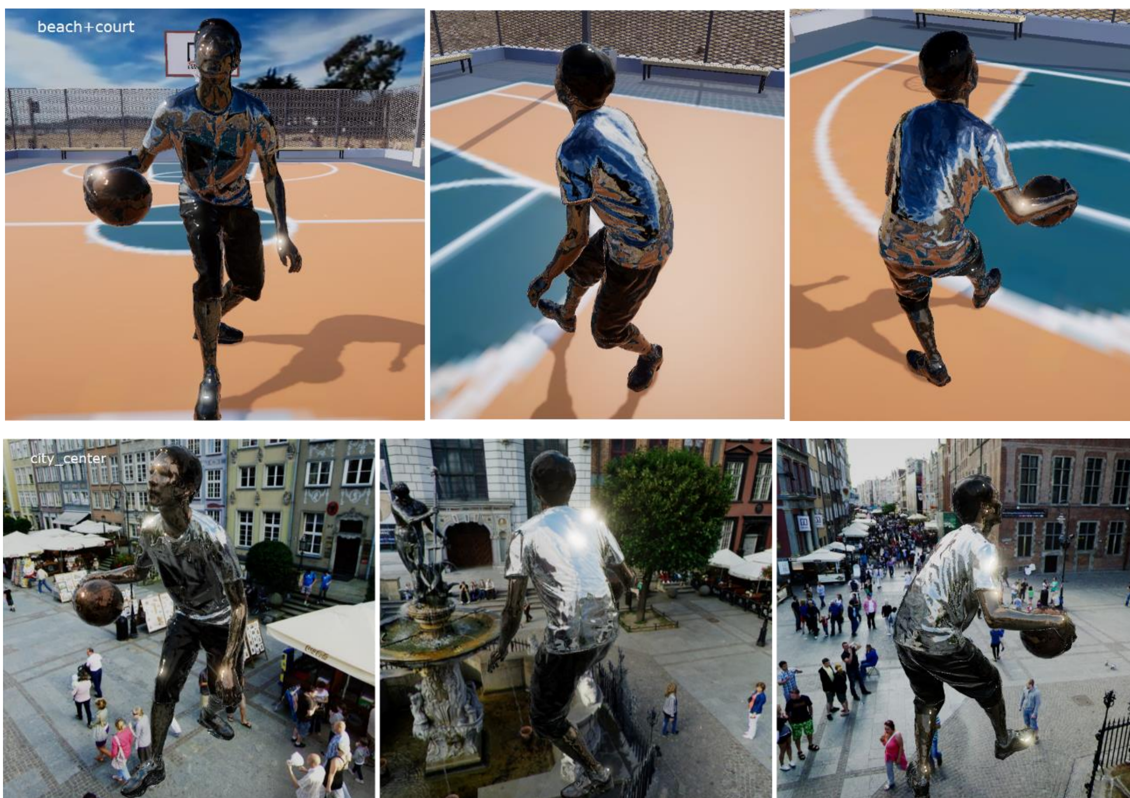
In this scene the focus will not be on integrating a point cloud inside an environment but on demonstrating how point clouds, as meshes, can be used to render different type of materials, for example perfect mirrors. To do this, we will firstly reuse the daylight scene analyzed in Section 3.6 and then we will use as background various environment textures rendered on the skybox without the use of the basketball court model. The used environment textures are freely available on [40].

To set up the scene the following modifications are made:

- To render a material representing a perfect mirror, the smoothness and metalness parameters are both modified to 1.0.
- The occlusion parameter of the material is also increased to approximately 0.7-0.8 to have a brighter model allowing to better distinguish the environment reflection.
- A reflection probe is placed above the model in the scene, to capture the environment to be reflected by the point cloud.
- Bloom is reduced in intensity to avoid annoying strong shining effects due to light reflection from the point cloud, which in turn prevent to easily distinguish the environment reflection.

In the next series of figures the point cloud reflecting the environment as a perfect mirror is shown for various environments. Note how the reflection is not perfect since the point cloud still uses its original color as albedo, so pixels that have a black color do not show any reflection, whereas white pixels well reflect the surroundings. Some imperfections are also due to the nature of the model, which is not a completely smooth surface as a sphere but represents a human figure with varying curvature which wears a t-shirt presenting many folds as analyzed in previous scenes.

Also note how the effect of the directional light bouncing on the model changes with respect to the previous settings, due to the different nature of the point cloud material. Being the smoothness at its maximum, when light hits the object a shiny effect is produced which, in this case, is slightly amplified by the bloom effect, still used to a limited extent. In this scene we can also appreciate the power of Image Based Lighting, described in Section 2.2.1. It is evident how the point cloud, when placed in different environments showing different lighting dynamics, still looks physically accurate regardless of the specific environment.



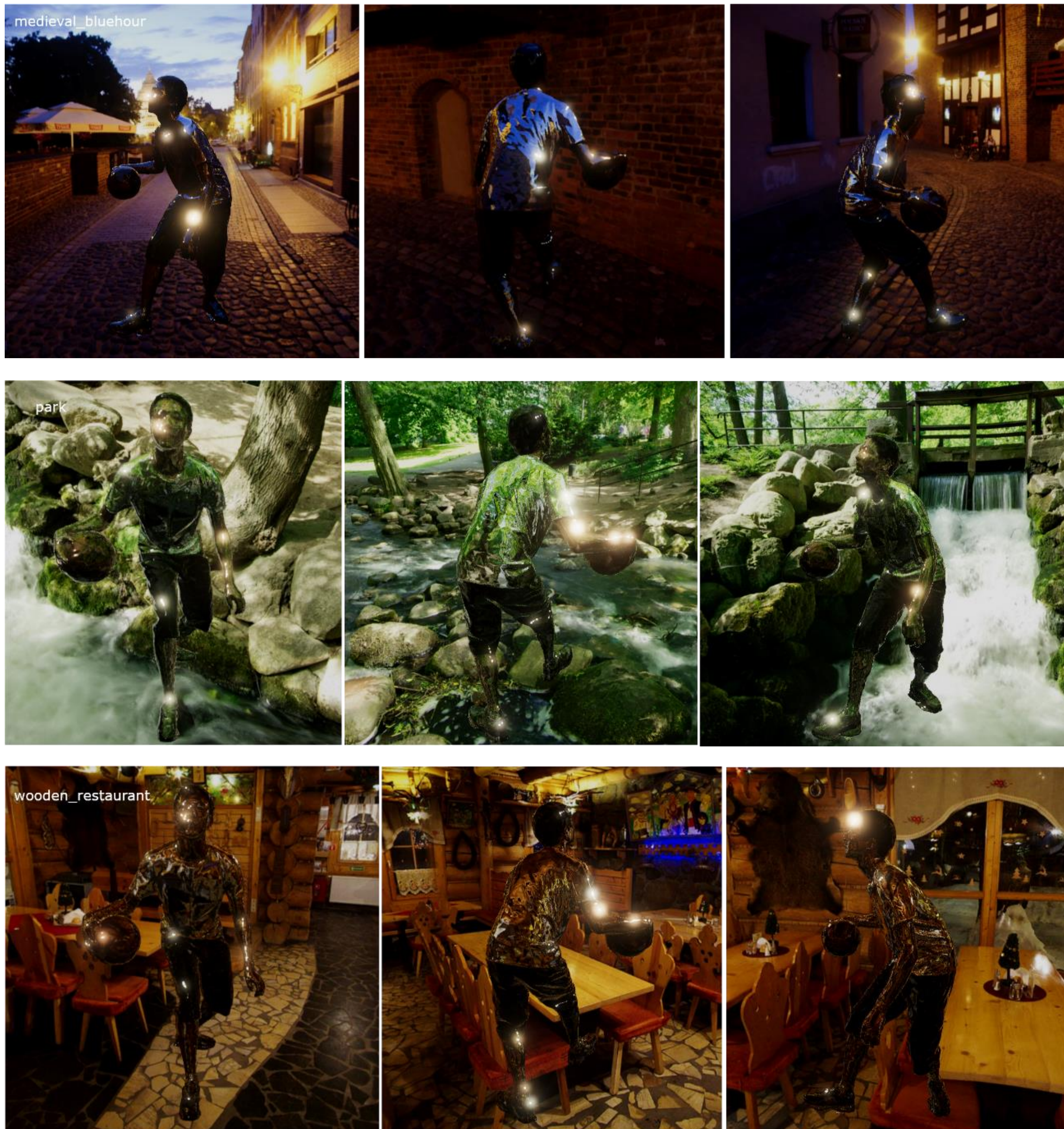


Figure 3.35 - Point cloud rendered as a perfect mirror placed in different environments

Beware that it is not possible to zoom in too much the models to appreciate very small details. Remember that the model used for these scenes is still a point cloud, so if it is observed from too close it will result in being able to distinguish the distinct points the model is made of and see through the holes between them. This is valid in any application using point clouds, not just for this specific scene.

4 Applications

In this chapter a short overview of potential applications where animated point clouds coupled with PBR may come up handy is given. Although at present day their use might still not be very affordable in many circumstances due to the high cost, both economic and computational, of producing them which needs many scanners placed around the objects and many computation systems to parallelize the effort and produce the result in a fast way, there are many fields that could potentially benefit from a wide exploitation of such models. The focus will be mainly on three macro areas: *learning*, *advertising* and *entertainment*.

4.1 Learning

There are many areas where a point cloud might be used to teach something to a user, going from classic environments as school and academic settings to sports, where the learning target are specific movements or gestures. Some examples of applications where animated point clouds can be used in this field are:

- *Guided Tours:* tours are constrained by the presence of physical guides which show and explain the particularities of the tour subject (which can be a museum, an archeological site or historical cities and events) in a certain order and under possible constraints like the time to reach a location or weather conditions. Guides can potentially be acquired as an animated point cloud with audio that can be rendered in Augmented Reality to allow higher flexibility on the tour and giving tourists the possibility to decide which items to visit and in which order with less constraints regarding time or otherwise.
- *Surgery simulations:* to teach students how to conduct a particular surgery, a sample intervention can be acquired as an animated point cloud that can

be visualized on a digital screen of any kind and as many times as needed by each student to fully understand it. Moreover, it allows to see the intervention from different viewpoints around the operating bed, giving a deeper insight on what is going on without the possible issue of overcrowding caused by an in-presence demonstration of a teacher in front of several students that would constrain free mobility of students in the environment and keeping a good sight on the teacher's movements.

- *Digital personal trainers:* a lot of smartphone applications have been developed to allow users to practice gym exercises from home. A feature of these apps is the presence of a 3D model of a coach which shows the user the correct execution of the specific exercise under practice. The 3D model can be replaced with an animated point cloud which behaves exactly the same thing but represents a real coach doing the exercise and allows the user to rotate and zoom the viewpoint to check specific details of the correct position to assume during the exercise.
- *Other sports* where specific movements need to be learnt, like martial arts, dance. The use of a point cloud representing a trainer of the discipline can be used to effectively study the movements to reproduce the target technique. Exercises from different sports can also be represented, for example the animated point cloud of the basketball player widely used in this document could be used to show how to execute a specific dribble drill in an effective way, or team schemes could also be scanned and proposed during training.

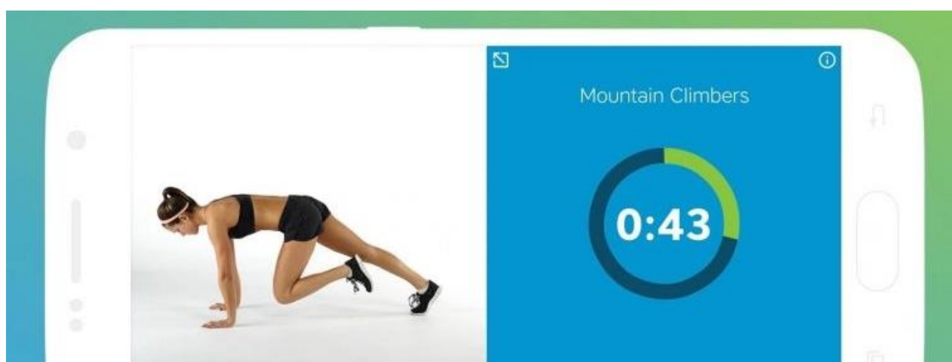
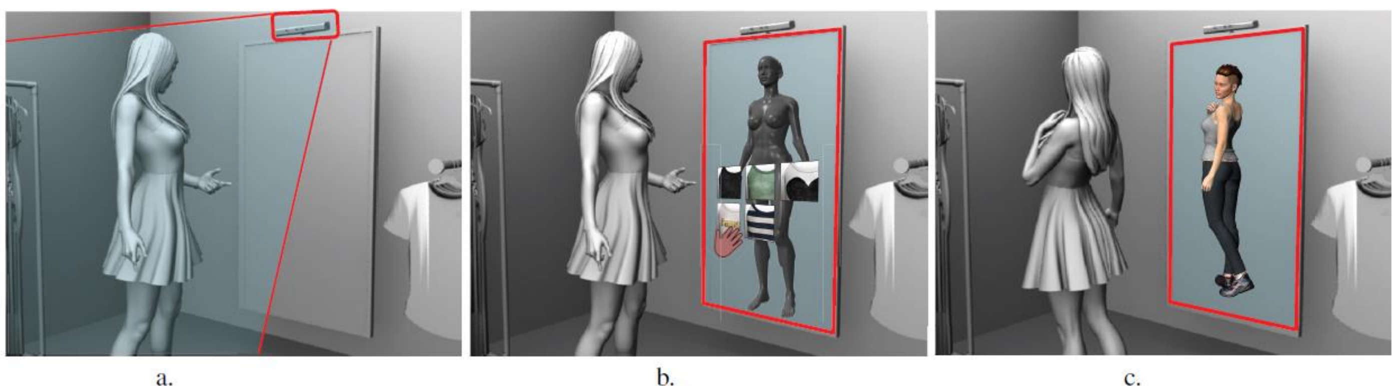


Figure 4.1 - Digital coach showing how to execute an exercise

4.2 Advertising

When a product is advertised, somehow it needs to be shown to customers to let them know what it is about. For some products there is no need to haven available 360 view, but there are cases in which it could come up handy and point clouds can be effectively used in these scenarios. For example, possible use cases are:

- *Clothing advertisement:* to sponsor clothing online, many pictures are usually needed to show the item as seen from different angulations to the potential customer. All these pictures can be substituted by a single point cloud which can be rotated and zoomed in different ways to satisfy the customer's needs. As a further development, it could be possible to have an animated avatar wearing the item or, with the help of Augmented Reality, the customer itself who can see herself wearing the clothing at the mirror, showing how it fits. The concept can be applied to any kind of clothing item, including shoes and caps.
- *Any other product advertising* that might benefit in showing the good from various viewpoints. Examples might be:
    - Vehicles, which could be visualized in 360 degrees instead of a single slightly rotated picture.
    - Furniture, to better see an item from any perspective and check if it would fit an apartment space.



a. b. c.

Figure 4.2 - Clothing advertising project from [41]. The Kinect sensor detects a user standing in front of the installation (a). The user interacts with the application, by Kinect or smart screen (b). The user can simulate the fitting of the dress concept created (c)

- Entire houses, allowing a potential customer to navigate inside the rooms from remote.

4.3 Entertainment

The entertainment field is probably the one which would most benefit from the use of point clouds due to the widespread use of digital media in many applications. The most suited uses would be in:

- *Videogames:* probably the application which most drives the development of Computer Graphics, animated point clouds could be used to substitute 3D meshes and save a lot of work needed to create animations by designers.
- *Movies:* most modern movies make extensive use of Computer-Generated Imagery (CGI), and animated point clouds could be used in this context. They could also be used in the production process of animated films, by substituting 3D models which are currently used.
- *Immersive experiences:* in such applications, users are typically inserted into an environment which is created in some way and visualized with the help of Extended Reality. The environment could be scanned from the real world and represented as an animated point cloud, allowing to render the whole background as a single model.

4.4 Conclusion

In this chapter a not exhaustive overview of how animated point clouds could be used in some fields. These models are very powerful, and they can potentially substitute other commonly used 3D models in any application thanks to the easier process of creation and acquisition. However, at present day, technology is still far from being suited for an extensive use of animated point clouds since they require a considerable computational effort to be acquired and processed in a fast way.

Also note that the use of PBR is not mandatory in any of the mentioned applications, but as shown above in this document by using point clouds with just their own color results in having a model disconnected from the environment in which is placed and gives a strange look to the scene. By using PBR, the point cloud fits better into the scene and gives a better overall appearance to the rendered image.

5 Conclusion & Future Works

In this work the goal was to demonstrate how an animated point cloud rendered with PBR could achieve better realism during visualization with respect to a point cloud rendered with a basic unlit shader using just the original vertices colors of the model. We first had to make sure that all vertices belonging to the model were associated to the estimated surface normal vectors to allow a proper lighting behavior, then models were imported in Unity and, through a sequence of different scenes with different environments and lighting conditions we highlighted the differences between using PBR or not.

It has been observed how with PBR a point cloud better integrates with the surrounding environment in any lighting condition, by casting shadows due to direct lighting and self-occlusions. It was also shown that it is possible, for a point cloud using PBR, to render different kind of materials as glass-like and mirrors in a physically plausible way. As point clouds are becoming more and more popular, using PBR to render them would be a benefit for many applications, going from simple model visualization to immersive experiences in Virtual or Mixed Reality. Using as an example the proposed scene in the night environment (Section 3.7), an immersive experience using an unlit shader would break the realism and immersivity of the entire scene by showing a very bright object in such a dark environment. With PBR the issue can be avoided, by providing realistic lighting in any condition.

This work was intended to show the potential of using PBR to render point clouds but there are many areas and topics that were left uncovered and beyond its purpose, which can be subject of future developments for improvement:

- The method we used to visualize the animation of the basketball player is rather inefficient. This is partially due to the fact that each frame of the animation was stored as a different mesh model, resulting in the necessity

to remove from the scene the previous model and draw the next one from scratch at each frame, which is a very expensive operation considering that each model of the *basketball_player* sequence comprises of about three million vertices with color and normal vectors information (about 80MBs per model). Although we have never imported and used all the 600 frames of the animation due to storage limitations, since at runtime the models are stored together in a list, even using just 20 frames of the animation results in occupying about 1.5 GBs of RAM. Consequently, by increasing the number of frames used, when the animation is played it results in being definitely slow. By using about 30 frames an average of 12 frames per seconds, touching a minimum of 5 frames per seconds, was estimated on the testing machine. Hence the efficiency of the process is a good area to explore for future works.

- In this work we used pre-recorded videos or textures rendered on the skybox as background, however there might be the possibility to acquire other point clouds and use them as background as well. In this way the background would also be a 3D model and we could obtain a similar result to what we were trying to achieve in our first scene (Section 3.4) without the related discovered issues when the background was placed on the skybox.
- The execution of the process in real-time, potentially useful for autonomous or remote-control devices, which would need intelligent cameras for a correct normal estimation on-the-fly to avoid the necessity of doing it by hand as a post-scan operation of the point cloud.

Bibliography

[1] MPEG-PCC, "Introduction to the MPEG-PCC project," 2019. [Online]. Available: https://mpeg-pcc.org/.

[2] F. Poux, "How to Automate Voxel Modelling of 3D Point Cloud with Python," 13 Dec 2021. [Online]. Available: https://towardsdatascience.com/how-to-automate-voxel-modelling-of-3d-point-cloud-with-python-459f4d43a227.

[3] D. Graziosi, O. Nakagami, S. Kuma, A. Zaghetto, T. Suzuki and A. Tabatabai, "An overview of ongoing point cloud compression standardization activities: video-based (V-PCC) and geometry-based (G-PCC)," *APSIPA Transactions on Signal and Information Processing,* vol. 9, p. e13, 2020.

[4] E. Zerman, C. Ozcinar, P. Gao and A. Smolic, "Textured Mesh vs Coloured Point Cloud: A Subjective Study for Volumetric Video Compression," in *Twelfth International Conference on Quality of Multimedia Experience (QoMEX)*, Athlone, Ireland, 2020.

[5] Y. Zhu and H. Rushmeier, "Point Clouds are Eating the World, One Application at a Time," 5 January 2021. [Online]. Available: https://www.sigarch.org/point-clouds-are-eating-the-world/.

[6] Q. Wang and M.-K. Kim, "Applications of 3D point cloud data in the construction industry: A fifteen-year review from 2004 to 2018," *Advanced Engineering Informatics,* vol. 39, pp. 306-319, 2019.

[7] M. Wilson, "What is a LiDAR scanner, the iPhone 12 Pro's camera upgrade, anyway?," TechRadar, 15 Jul 2021. [Online]. Available:

https://www.techradar.com/news/what-is-a-lidar-scanner-the-iphone-12-pros-rumored-camera-upgrade-anyway. [Accessed 19 Jan 2022].

[8] Q. Wang, Y. Tan and Z. Mei, "Computational Methods of Acquisition and Processing of 3D Point Cloud Data for Construction Applications," *Archives of Computational Methods in Engineering volume,* vol. 27, pp. 479-499, 2020.

[9] C. Thomson, "Reality capture 101: point clouds, photogrammetry and LiDAR," Vercator, 19 Nov 2019. [Online]. Available: https://info.vercator.com/blog/reality-capture-101-point-clouds-photogrammetry-and-lidar. [Accessed 19 Jan 2022].

[10] D. Medda, Y. M. Anoffo, C. Perra and D. Giusto, "Automated point cloud acquisition systemusing multiple RGB-D cameras," in *Proc. SPIE 11353, Optics, Photonics and Digital Technologies for Imaging Applications VI,* 2020.

[11] E. S. Jang, M. Preda, K. Mammou, A. M. Tourapis, J. Kim, D. Graziosi, S. Rhyu and M. Budagavi, "Video-Based Point-Cloud-Compression Standard in MPEG:," *IEEE Signal Processing Magazine,* vol. 36, no. 3, pp. 118-123, May 2019.

[12] A. Akhtar, W. Gao, L. Li, Z. Li, W. Jia and S. Liu, "Video-based Point Cloud Compression Artifact Removal," *IEEE Transactions on Multimedia,* 2021.

[13] S. Schwarz, N. Sheikhipur, V. Fakour Sevom and M. M. Hannuksela, "Video coding of dynamic 3D point cloud data," *APSIPA Transactions on Signal and Information Processing,* vol. 8, Dec 2019.

[14] C. Cao, M. Preda and T. Zaharia, "What's new in Point Cloud Compression?," *Global Journal of Engineering Sciences,* vol. 4, no. 5, 2020.

[15] H. Nobuhara and K. Hirota, "Color Image Compression/Reconstruction by YUV," in *IEEE Annual Meeting of the Fuzzy Information*, Banff, AB, Canada, 2004.

[16] S. Schwarz and al, "Emerging MPEG Standards for Point Cloud Compression," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems,* vol. 9, 2019.

[17] M. Preda, "G-PCC codec description," WG 7, MPEG 3D Graphics Coding, 2021.

[18] X. Sheng, L. Li, D. Liu and Z. Xiong, "Attribute Artifacts Removal for Geometry-based Point Cloud Compression," 1 Dec 2021. [Online]. Available: https://arxiv.org/pdf/2112.00560.pdf.

[19] J. Wang, H. Zhu, H. Liu and Z. Ma, "Lossy Point Cloud Geometry Compression via End-to-End Learning," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 31, no. 12, pp. 4909 - 4923, 2021.

[20] D. Tian, H. Ochimizu, C. Feng, R. Cohen and A. Vetro, "Geometric distortion metrics for point cloud compression," in *2017 IEEE International Conference on Image Processing (ICIP),* Beijing, China, 2017.

[21] Autodesk, "Introduction to rendering," Autodesk, 09 Sep 2014. [Online]. Available: https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2015/ENU/Maya/files/About-rendering-and-renderers-Introduction-to-rendering-htm.html. [Accessed 21 Jan 2022].

[22] "Basic lighting", LearnOpenGL, [Online]. Available: https://learnopengl.com/Lighting/Basic-Lighting. [Accessed 21 Jan 2022].

[23] "Advanced Lighting", LearnOpenGL, [Online]. Available: https://learnopengl.com/Advanced-Lighting/Advanced-Lighting. [Accessed 21 Jan 2022].

[24] A. Martin, "Radiosity," Worcester Polytechnic Institute, 1999. [Online]. Available: https://web.cs.wpi.edu/~matt/courses/cs563/talks/radiosity.html. [Accessed 23 Jan 2022].

[25] "PBR Theory", "LearnOpenGL," [Online]. Available: https://learnopengl.com/PBR/Theory. [Accessed 24 Jan 2022].

[26] J. Russell, "Basic Theory of Physically-Based Rendering," Marmoset, 5 Nov 2020. [Online]. Available: https://marmoset.co/posts/basic-theory-of-physically-based-rendering/. [Accessed 25 Jan 2022].

[27] "Diffuse Irradiance", "LearnOpenGL," [Online]. Available: https://learnopengl.com/PBR/IBL/Diffuse-irradiance. [Accessed 26 Jan 2022].

[28] "Specular IBL", "LearnOpenGL," [Online]. Available: https://learnopengl.com/PBR/IBL/Specular-IBL. [Accessed 26 Jan 2022].

[29] P. Debevec, "Paul Debevec Home Page," [Online]. Available: https://www.pauldebevec.com. [Accessed 27 Jan 2022].

[30] Otoy, "LightStage," Otoy, [Online]. Available: https://home.otoy.com/capture/lightstage/overview. [Accessed 27 Jan 2022].

[31] T. Sun, Z. Xu, X. Zhang, S. Fanello, C. Rhemann, P. Debevec, J. T. Barron, Y.-T. Tsai and R. Ramamoorthi, "Light Stage Super-Resolution:

Continuous High-Frequency Relighting," *ACM Transactions on Graphics (TOG),* vol. 39, no. 6, pp. 1-12, 2020.

[32] Y. Xu, Y. Lu and Z. Wen, "Owlii Dynamic human mesh sequence dataset," in *ISO/IEC JTC1/SC29/WG11 m41658, 120th MPEG Meeting,* Macau, 2017.

[33] S. Molnar, B. Kelenyi and L. Tamas, "ToFNest: Efficient normal estimation for time-of-flight depth cameras," in *2021 IEEE/CVF International Conference on Computer Vision Workshops (ICCVW)*, Montreal, BC, Canada, 2021.

[34] Keijiro, "Pcx," GitHub Repository, 2021. [Online]. Available: https://github.com/keijiro/Pcx.

[35] Columbia International University, "YouTube," 16 Aug 2021. [Online]. Available: https://www.youtube.com/watch?v=0NSAKO-YQak.

[36] Klieg3D, "SketchFab," 30 Aug 2020. [Online]. Available: https://sketchfab.com/3d-models/basketball-court-d2ea5bc76e094f1a9e6aa15891bd6885.

[37] FluentEsl, "YouTube," 16 Dec 2020. [Online]. Available: https://www.youtube.com/watch?v=u7GPUqT3E_c.

[38] LuckyKid88, "YouTube," 23 Jan 2018. [Online]. Available: https://www.youtube.com/watch?v=0KXDZd6rF58.

[39] Hakanai, "EACSkyboxShader," GitHub Repository, 31 May 2018. [Online]. Available: https://github.com/hakanai/EACSkyboxShader.

[40] Texturify, "Environment Panoramas," [Online]. Available: https://texturify.com/category/environment-panoramas.html.

[41] P. Cremonesi, F. G. M. Garzotto and P. I. M. Piazzolla, "Toward a New Fashion Concepts Design Tool: The vMannequin Framework," in *Workshop on Business Models and ICT Technologies for the Fashion Supply Chain*, 2017.