University of Padova

Department of Information Engineering

Master's degree in Automation Engineering

Development of a simulator for 3D pattern recognition scanners

Author
Alessandro Rossi

Supervisor
Prof. Ruggero Carli

Company's supervisor
Ing. Roberto Polesel

July 2, 2018

Academic Year 2017/2018

My deepest gratitude to Prof. Ruggero Carli,
for his guidance and enthusiasm.
A special thanks goes to Roberto,
for the opportunities that has offered me,
and to Euclid Labs' team,
for their experience and support.
A precious thanks to my awesome family,
for untiring love and support.
To Emanuele, Marco and Maricarmen,
that have definitely made this journey unforgettable.
This work is dedicated to Alessia,
and our future together.

Abstract

Shape reconstruction using coded structured light is considered one of the most reliable techniques to recover object surfaces. Having a calibrated projector-camera pair, a light pattern is projected onto the scene and captured by the camera. Correspondences between projected and recovered patterns are found and used to extract 3D surface information. The aim of this work is to develop a simulator capable of emulating such a model. In absence of real data, an accurate simulator would allow the company to generate test datasets useful to train deep learning systems for future development of artificial intelligence algorithms. After a brief introduction on the discussed topics, a detailed explanation of the particular reconstruction method will be given. The design, together with the software implementation, of a simulator will follow. Special emphasis will be placed on its limits, highlighting possible future improvements too. Finally the algorithm is tested with different types of objects analysing quality, precision and deviation from the results obtained using real scanners. The thesis was carried out in collaboration with Euclid Labs, which shared its experience and provided the necessary tools to perform the tests.

Sommario

La ricostruzione delle superfici tramite luce strutturata è considerata una delle tecniche più affidabili per la generazione di immagini 3D. Data una coppia proiettore-camera calibrata, un pattern luminoso viene proiettato sulla scena e catturato dalla camera. Le corrispondenze tra pattern proiettato e catturato vengono poi utilizzate per estrarre le coordinate 3D della superficie. L'obiettivo di questo lavoro è di realizzare un simulatore in grado di emulare tale modello. In mancanza di dati reali, un simulatore accurato permetterebbe all'azienda di generare test dataset utili ad allenare sistemi di deep learning per l'implementazione futura di algoritmi di intelligenza artificiale. Dopo una breve introduzione dell'argomento, nella prima parte ci soffermeremo sulla descrizione del particolare metodo ricostruttivo e successivamente si sposterà l'attenzione sulla progettazione e realizzazione pratica del simulatore. Si presterà particolare enfasi ai limiti di quest'ultimo, mostrandone i miglioramenti rispetto alle soluzioni esistenti e i possibili perfezionamenti futuri. Infine si testerà l'algoritmo con diverse tipologie di oggetti analizzando qualità, precisione e scostamento dai risultati ottenuti con alcuni scanner a luce strutturata. La tesi è stata realizzata in collaborazione con Euclid Labs, che ha condiviso la sua esperienza e ha fornito gli strumenti necessari ad effettuare le prove.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Three-dimensional reconstruction constitutes a fundamental topic in computer vision, having different applications such as object recognition and classification, pick and place, in-line quality control, collision avoidance, range sensing, industrial inspection, biometrics and others. The developed solutions are traditionally categorized into contact and non-contact techniques [1]. The former have been used for a long time in industrial inspections and consist of a tool which probe the object through physical touch. The main problems of contact techniques are their slow performance and high cost of using mechanically calibrated passive arms. Besides, the fact of touching the object is not feasible for many applications. Non-contact techniques were developed to cope with these problems, and have been widely studied. They can be classified into two different categories: active and passive. In passive approaches, the reflectance of the object and the illumination of the scene are used to derive the shape information: no active device is necessary [2]. For example, think about stereo-vision, where the scene is first imaged by cameras from two or more points of view and then correspondences between the images are found. The main problem experimented when using this approach is a sparse reconstruction since density is directly related to the object texture. This complicates the process of finding correspondences in the presence of texture-less surfaces. On the other hand, in active approaches, suitable light sources are used as internal vector of information. Between those, two of the best most popular methods are definitely time-of-flight and structured light.

1.1 Structured light reconstruction

We have just seen several ways to build a 3D image, but, as the thesis title suggests, from now on we will focus on structured light scanner. The latter is a scanning device for measuring three-dimensional shape of an object using projected light models and a camera system. In this case, one of the cameras is substituted by an active device which projects a structured light pattern onto the scene. This active device is modelled as an inverse camera, being the calibration step a similar procedure to the one used in a classical stereo-vision system. It is essentially based on the projection of a sequence of codified patterns, like those shown in Figure 1.1.

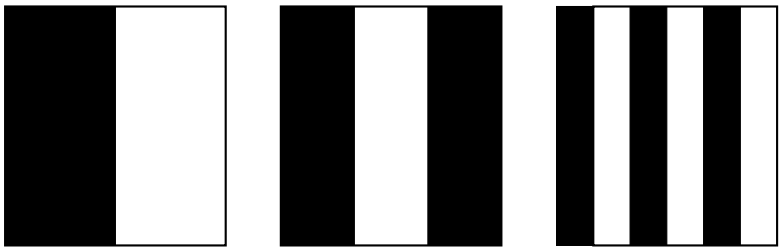


Figure 1.1: Examples of codified patterns (binary codes).

The deformation induced by the surface of the object is acquired by the camera and then used for the computation of three-dimensional coordinates. A deeper explanation of the principle behind this technique will be provided in Chapter 3.

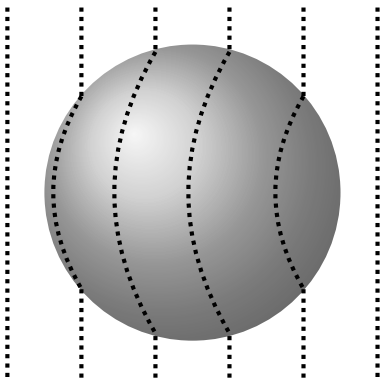


Figure 1.2: Projected grid deformation seen from camera's perspective.

Depending on the camera resolution, such a system allows digitizing several hundred thousand points at a time. Projecting a narrow band

of light onto a three-dimensionally shaped surface produces a line of illumination that appears distorted from other perspectives than that of the projector, and can be used for geometric reconstruction of the surface shape using a triangulation procedure. A faster and more versatile method is the projection of patterns consisting of many stripes at once, or of arbitrary fringes, as this allows for the acquisition of a multitude of samples simultaneously. Seen from different viewpoints, the pattern appears geometrically distorted due to the surface shape of the object. Although many other variants of structured light projection are possible, parallel stripes patterns are widely used. The latter are called *binary codes* and are those shown in Figure 1.1.

1.2 Report structure

This report is organized as follows: in Chapter 2, after a brief presentation on state of the art solutions, we will focus on a detailed description of our approach to structured light reconstruction, together with a few mathematical complements needed to understand the following chapters. In Chapter 3, we will present the design of the simulator and its implementation, highlighting the problems encountered and the solutions adopted. There we will take a closer look to the tools and libraries used for the realisation of the simulator. One of the most significant part is Chapter 4, where we will present qualitative and quantitative tests that remark pro and cons of our algorithms. We will focus on precision analysis in different contexts and difficulties. In order to validate the results obtained from the simulator, we will compare it with two real scanners. The work ends with Chapter 5, reporting conclusions and future improvements.

Chapter 2

Problem Formulation

2.1 State of the art

Over the past years, structured light projection systems have been widely studied. Shirai and Suwa in 1971, proposed a slit line projection to recognise polihedric objects [3]. In 1973, Agin and Binford generalised this idea to recognise curvilinear objects [4]. Two years later, Popplestone et al. proposed a more general system which recognises either polihedrics or curvilinear objects [5]. To improve the accuracy of the system, an alternative way is to project a grid of dots or lines over the scene to cover the entire range of the camera. Asada et al. proposed to use a pattern made by a set of vertical, parallel and equidistant, stripe lines [6]. Furthermore in 1986, in order to obtain 3D surface properties, Stockman et al. have proposed the widely known projection of a grid [7]. Then, an easier correspondence problem has to be solved, we have to identify, for each point of the imaged pattern, the corresponding point of the projected pattern. The correspondence problem can be directly solved codifying the projected pattern, so each projected light point carries some information. A detailed survey on recent progress in coded structured light is presented by [8]. Salvi et al. presents an exhaustive analysis of different coding strategies used in active structured light, focusing on the advancements presented in the last years. There, a new classification regarding the strategy used to create the pattern is proposed [1].

2.2 Our approach

As mentioned in the previous section, in the last forty years, many studies on possible projection techniques and different types of patterns

have been conducted. All of those works are focused on solving or optimizing a particular problem without providing a solid basic model. Therefore, in this work, we wanted to develop a fast algorithm to reconstruct a scene by projecting a very simple pattern, i.e. a grid of points. Our idea takes inspiration by the natural behaviour of light. Therefore, before proceeding with the simulator design, in order to give a clearer explanation, some complements are now provided.

2.2.1 Simplified natural behaviour

By simplifying what happens in nature, a light source emits a ray of light that travels, eventually, to a surface which interrupts its progress. Imagine the beam as a stream of photons travelling along the same path. By ignoring relativistic effects, in a perfect vacuum, such a ray will be a straight line. Each beam, when it hits an object, can give rise to four different phenomena: absorption, reflection, refraction and fluorescence. A surface can absorb part of the light beam, resulting in a loss of intensity of reflected or refracted light. It may also reflect all or part of the ray, in one or more directions. If the surface has any transparent properties, it refracts a portion of the light beam into itself in a different direction while absorbing some of the spectrum. Less commonly, a surface may absorb a portion of the light and fluorescently re-emit the light at a longer wavelength color in a random direction, though this is rare enough that it can be neglected from most rendering applications. At this point, reflected and/or refracted rays may hit other surfaces where their absorbent, refracting, reflecting and fluorescent properties again affect the progress of incoming rays. Some of these beams travel in such a way that they hit our eyes, causing us to see the scene and thus contributing to the final rendered image.

2.2.2 Ray casting

To simulate the model we have just described, Arthur Appel presented in 1968 the first “ray tracing” algorithm used for rendering, and has since been called “ray casting” [9]. The idea behind this algorithm, represented in Figure 2.1, is to shoot rays from the camera, one for each pixel, finding the nearest object that blocks the path of that beam. The simplifying hypothesis is made that if a surface faces a light, the latter will reach that surface and will not be in shadow. It may seem counter intuitive to send rays away from the camera rather than into it, as light does in reality, but doing so is many orders of magnitude more efficient. Since

most of the rays from a given light source do not crash directly into the camera, a "forward" simulation could potentially waste a huge amount of computation on light paths that are never recorded. One important advantage ray casting offered over older scanline algorithms[1], was its ability to easily deal with non-planar solids, such as cones and spheres. If a mathematical surface can be intersected by a ray, it can be rendered using this method.

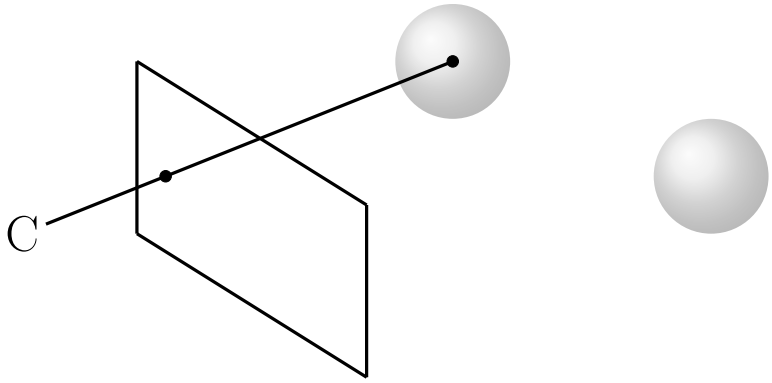


Figure 2.1: Ray casting technique.

2.2.3 Ray tracing

The next important research breakthrough came from Turner Whitted in 1979 [10]. Previous algorithms, such as ray casting, traced rays from camera to scene until they hit an object without recursively tracing the others. Whitted took inspiration from Physics and continued the process. Therefore, when a ray hits a surface, it can generate up to three new types of rays: reflection, refraction and shadow. A reflection ray is traced in the mirror-reflection direction. The closest object it intersects is what will be seen in the reflection. Refraction rays travelling through transparent material work similarly, with the addition that a refractive ray could be entering or exiting a material. A shadow ray is traced toward each light. If any opaque object is found between the surface and the light, the surface is in shadow and the light does not illuminate it. This technique is capable of producing a very high level of realism, generally higher than that provided by older methods, at the price of a greater computational cost. This makes ray tracing more suitable for applications

[1] algorithms for visible surface determination, using a particular vertices sorting.

where rendering time is negligible, as for visual effects in movies, and less suitable for real-time applications, where speed is a key factor.

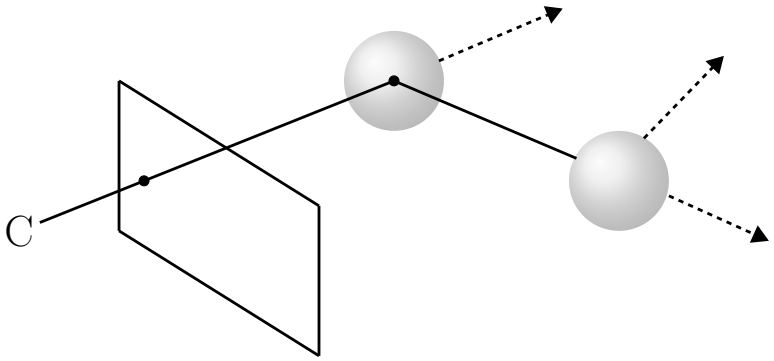


Figure 2.2: Ray tracing technique.

2.3 Mathematical complements

Before proceeding with the next chapter, concerning the simulator design, some mathematical principles, that will be useful afterwards, are now provided.

2.3.1 Ray/triangle intersection

We will see in the next chapter that for every point in the pattern grid we will throw a ray passing through it. At that point we will need to check if that ray hits a mesh in the scene. Since a mesh is a set of triangles, this operation is based on a ray-triangle intersection method; in particular we have chosen the Möller-Trumbore algorithm [11].
A ray $R(t)$ with origin O and normalized direction D is defined as

$$R(t) = O + tD$$

and a triangle is defined by three vertices V_0, V_1 and V_2. In the ray/triangle intersection problem we want to determine if the ray intersects the triangle. Previous algorithms have solved this by first computing the intersection between the ray and the plane in which the triangle lies, and then testing if the intersection point is inside the edges. Instead, Möller-Trumbore algorithm uses minimal storage, i.e only the vertices of the triangle need to be stored, and does not need any preprocessing. The memory savings

are significant, ranging from about 25% to 50%, depending on the amount of vertex sharing. A point, $T(u, v)$, on a triangle is given by

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2$$

Computing the intersection between the ray, $R(t)$, and the triangle, $T(u, v)$, is equivalent to imposing $R(t) = T(u, v)$, which yields:

$$O + tD = (1 - u - v)V_0 + uV_1 + vV_2$$

Rearranging the terms gives:

$$\begin{bmatrix} -D & V_1 - V_0 & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0 \tag{2.1}$$

This means the coordinates (u, v) and the distance, t, from the ray origin to the intersection point, can be found by solving the linear system of equations above. Denoting $E_1 = V_1 - V_0$, $E_2 = V_2 - V_0$ and $T = O - V_0$, the solution to Equation (2.1) is obtained by using Cramer's rule:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-D, E_1, E_2|} \begin{bmatrix} |T, E_1, E_2| \\ |-D, T, E_2| \\ |-D, E_1, T| \end{bmatrix} \tag{2.2}$$

From linear algebra, we know that

$$|A, B, C| = det(A, B, C) = -(A \times C) \cdot B = -(C \times B) \cdot A$$

Hence, Equation (2.2) could be rewritten as

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix}$$

where $P = D \times E_2$ and $Q = T \times E_1$. These factors can be reused in the implementation to speed up computations.

2.3.2 Snell's law

Snell's law is a formula used to describe the relationship between angles of incidence and refraction. It could be of particular interest in order to improve the realism when using a rendering algorithm like the ray casting one. In particular, we can use this law to compute the power of the reflected ray, i.e. the one which carries information about the point hit. Looking at Figure 2.3, it can be seen that the power of the reflected ray increases as the refraction angle θ_2 grows. The latter can be computed inverting formula (2.3), knowing the angle of incidence θ_1 and the refraction indices n_1 and n_2.

$$n_1 \sin\theta_1 = n_2 \sin\theta_2 \tag{2.3}$$

This happens until we reach the critical angle θ_c, defined as the angle of incidence that provides an angle of refraction of 90 degrees. In that case we have the phenomenon of total internal reflection that can be seen in the third case of Figure 2.3. This particular angle can be computed using the formula below, obtained by the Snell's law.

$$\theta_c = \arcsin\left(\frac{n_1}{n_2}\right)$$

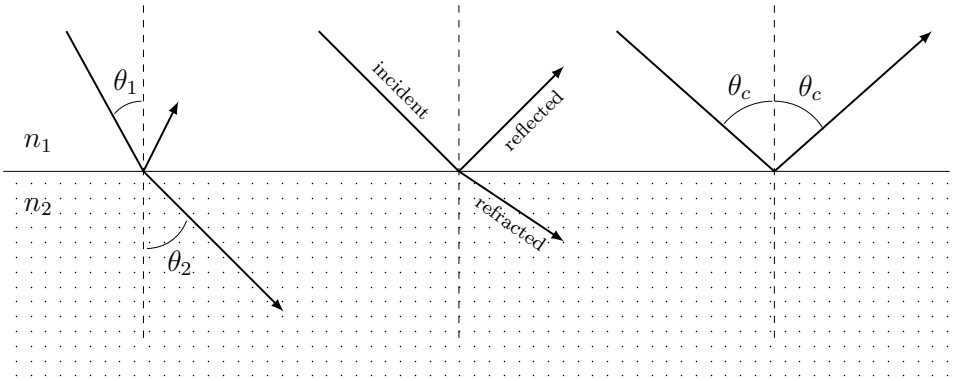


Figure 2.3: Demonstration of no refraction at angles greater than θ_c.

2.3.3 Homogeneous transformations

As will be explained in detail in the following chapter, regarding the simulator design, homogeneous transformations will play an essential role. Indeed, in order to speed up the ray/mesh intersection process and

to allow movement and rendering of objects in the scene, it is essential to be able to perform computations on sets of points lying on different reference frames. In this section, we introduce 4D vectors and the so-called "homogeneous" coordinate. As we will see, 4D vectors and 4×4 matrices are nothing more than a notational convenience for what are simple 3D operations.

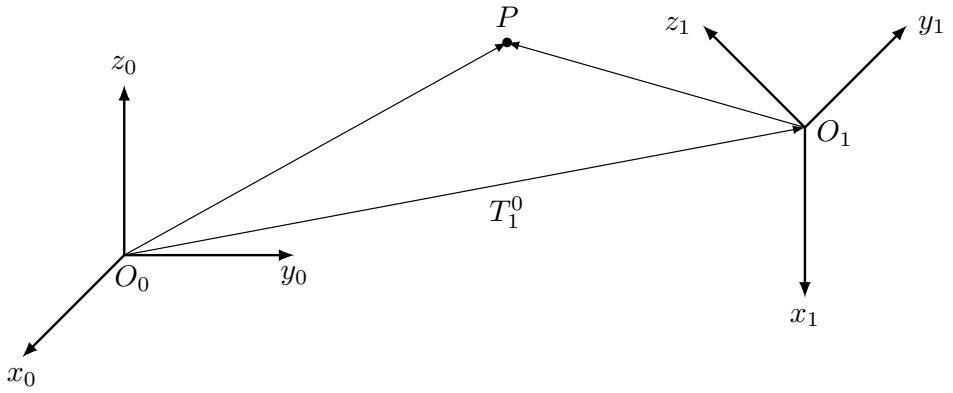


Figure 2.4: Representation of a point P in different reference frames.

Let's consider two different reference frames, $O_0 - x_0y_0z_0$ and $O_1 - x_1y_1z_1$, and an arbitrary point P in space. Let P^0 and P^1 be the vectors of coordinates of P with respect to the first and second reference frame accordingly. Let T_1^0 be the vector describing the origin of the frame 1 with respect to frame 0, and R_1^0 be the rotation matrix of frame 1 with respect to frame 0. On the basis of simple geometry, the position of P^1 with respect to the reference frame 0 can be expressed as:

$$P^0 = T_1^0 + R_1^0 P^1 \tag{2.4}$$

Hence, this equation represents the coordinate transformation of a vector between two frames. To have a compact representation of the relationship between the coordinates of the same point in two different frames, the homogeneous representation of a generic vector P can be introduced as the vector $\tilde{P}$, formed by adding a fourth unit component:

$$\tilde{P} = \begin{bmatrix} P \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

By adopting this representation for vectors, the coordinate transformation can be written in terms of a 4×4 matrix:

$$A_1^0 = \begin{bmatrix} R_1^0 & T_1^0 \\ \underline{O}^\top & 1 \end{bmatrix}$$

which is termed homogeneous transformation matrix [12]. It can be easily observed that the coordinate transformation (2.4) can be compactly rewritten as:

$$\tilde{P}^0 = A_1^0 \tilde{P}^1$$

As we will see in a moment, this transformation is the basis of the rendering pipeline and will be widely used to simplify operations on vectors and points within the 3D world. Here, we do not give details on how to build rotation or translation matrices, but we provide an example to emphasize the basic concept. Let's assume that we want to rotate a vector around the z-axis by an angle α. The corresponding rotation matrix is:

$$R_z(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Hence, the desired transformation can be expressed as:

$$\begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} = A_1^0 \tilde{P}^1 = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix}$$

2.4 Rendering pipeline

In this section we will try to understand in detail one of the core mechanics of any 3D graphics engine: the chain of matrix transformations that allow to represent a 3D object on a 2D monitor. We have already seen, in Section 2.3.3, how a transformation could be represented in matrix form. From there we will show the typical sequence of transformations needed to go from local space, where each object lives, to viewport space.

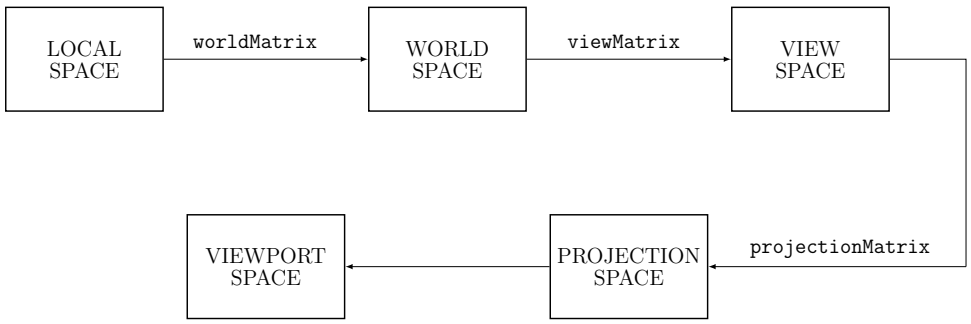


Figure 2.5: Rendering pipeline diagram.

As we can see from Figure 2.5, everything starts within the local space, which is a specific vector space where each model lives, and it's represented with the canonical 3D coordinates system. More specifically, all the vertices of a particular model are relative to the origin of its local space. The first step, when we want to render a 3D scene, is to put all the models in the same world space. Since every object lies in its own position and orientation in the world, each one has a different local-to-world transformation matrix (`worldMatrix`). With all the objects at the right place, we now need to project them to the screen, and this is usually done in three steps.
The first step moves all the objects in another space termed view space. The latter is an auxiliary space used to simplify computations and keep everything elegant and encoded into matrices. The idea is that we need to render to a camera, which implies projecting all the vertices onto the camera screen that can be arbitrarily oriented in space. The math simplifies a lot if we could have the camera centred in the origin and looking towards the z-axis. The view space does exactly this, remapping the world space, using the `viewMatrix`, so that the camera is in the origin and looks down along the z-axis.
The second step performs the actual projection starting from the view space. All we have to do is to project our scene onto the imaginary screen of the camera. Before flattening the image, we still have to move into another space, the projection one. The latter is a cuboid which dimensions are normalized between -1 and 1 for every axis. This space is very convenient for clipping, i.e anything apart from $[1, -1]$ is outside the camera view area, and simplifies the flattening operation, i.e. we just need to drop the z value to get a flat 2D image. To go from the view space into the projection space we need another matrix, termed `projectionMatrix`, and its values depend on what type of projection we

want to perform. The two most used projections are the orthographic and the perspective one. Briefly, the former is a parallel projection. Each line that is originally parallel will remain so after this transformation. Conversely, the second one is not a parallel projection and originally parallel lines will no longer be after this operation.
The last step flats the image dropping the z component, scale everything to the viewport width and height and, finally, transform all the vertices into pixel coordinates.

Summarizing, by following the chain of transformations described above, it is possible to display a 3D scene on a 2D monitor. Obviously, we could also follow the opposite procedure, with the only difference that each transformation from one space to another will be obtained by inverting the matrices shown in Figure 2.5.

Chapter 3

Simulator design

Summarizing what has been told so far, a conventional structured light technique is derived from the stereo vision. In the latter, which is inspired by the human vision, a pair of cameras is used to capture an object from two different perspectives. For every image point A captured on the first image, an algorithm tries to estimate a corresponding image point B on the second one using epipolar constraints. Having the correspondence, i.e. the position of the same object point on both images, the algorithm computes 3D position of the captured object point using the triangulation principle. Although the idea is very simple, the search for image correspondences is an ill-posed problem.
On the other hand, in the structured light approach, one of the cameras is substituted by a pattern projector, which emits a well defined structured illumination. Most often, the source of the structured illumination is a conventional 2D projector, similarly to the technology used for multimedia presentations. Camera and projector are located on a well known baseline, determined by a system calibration, and are focused towards the scanning area. A simple representation of the entire system is shown in Figure 3.1.

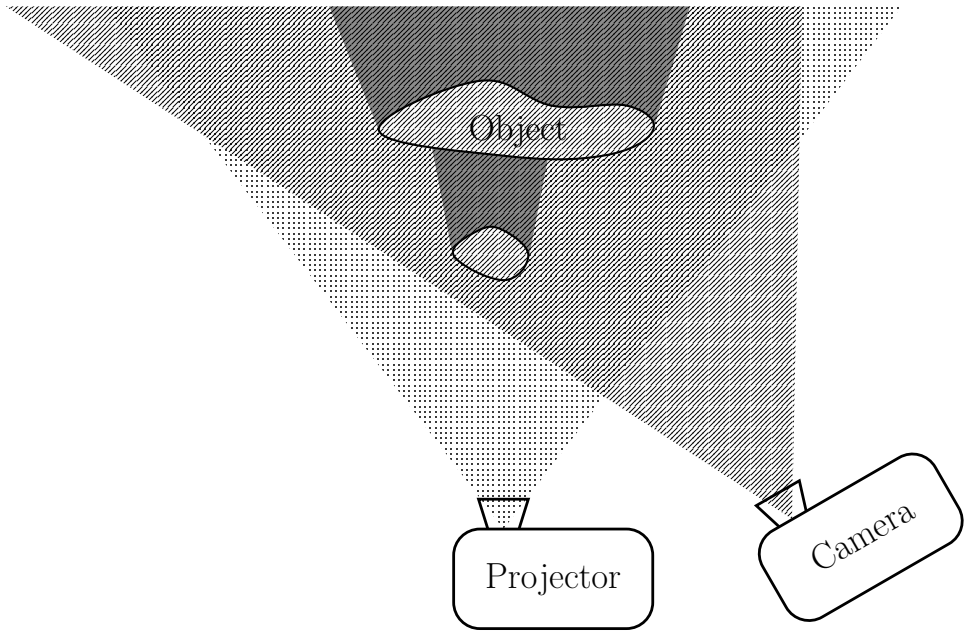


Figure 3.1: Representation of a conventional light technique.

This is the main chapter of the work. In the first part, we present the design of the simulator in a schematic way. We proceed with a digression on the development tools that we decided to use. The chapter ends with the practical implementation. In the discussion, we will focus on the main problems encountered, along with the solutions adopted.

3.1 Design

Before starting with the design of the simulator, it is very useful to describe the stages through which we arrive at the final 3D reconstruction, emulating a structured light scanner. Specifically, we can highlight five steps.

1. The projector emits a set of coding patterns projected in a specified succession, one after another. These coding patterns encodes a spatial information.

2. The camera captures the scanning area once per every projected pattern.

3. For every image point A, the algorithm decodes the spatial information encoded in the succession of intensity values captured by

the image point A, under different structured light patterns.

4. This spatial information encodes the corresponding image point B in the view of the projector.

5. With the correspondence, the algorithm computes an exact 3D position of the object point.

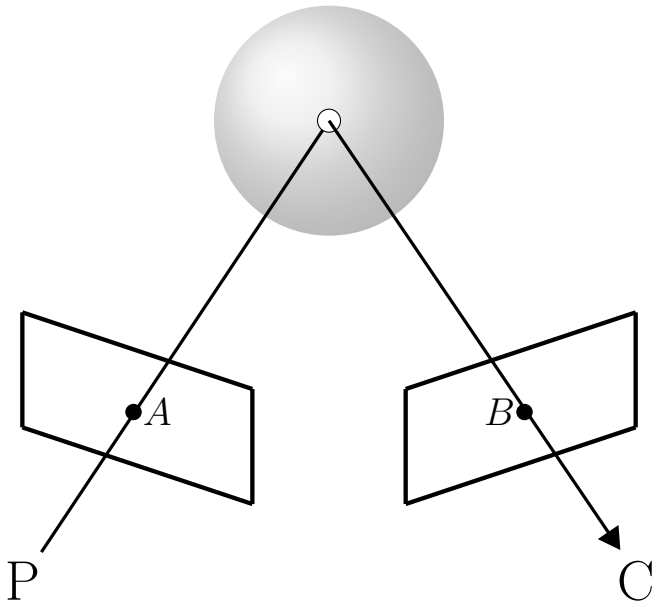


Figure 3.2: Representation of our approach.

Before going into the details of the practical implementation, we proceed illustrating framework and software development tools used to realise the simulator.

3.2 Framework

The proposed simulator, see Figure 3.3, is composed by three entities:

Core	represents all the CPU-side code that is used to manage the user interaction.
Graphics engine	represents all the GPU-side code which allows to view the world and results onto display.

Scanning engine contains all the algorithms that allow to reconstruct the scene by emulating a structured light 3D scanner. The first part of the work is done on the CPU-side and then all computations are done on GPU exploiting parallelism.

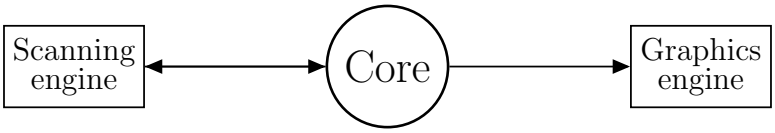


Figure 3.3: Modelling of the simulator.

The graphics engine was designed taking inspiration from the Direct3D Rendering Cookbook [13]. In this book, Stenning provides a practical guide for creating a DirectX11-based graphics engine which allows to render 3D shapes or meshes imported from external graphics files (.fbx, .stl, etc.). The framework proposed by Stenning adapts well to our needs since it was developed using the SharpDX library, a `C#` wrapper for DirectX11. However, the most interesting part of this work is represented by the core and scanning engine. Both were written in `C#` using SharpDX, although, to take advantage of the GPU power, it was necessary to write the reconstruction algorithms in `HLSL`[1]. The latter is a proprietary shading language developed by Microsoft, similar to `GLSL` for OpenGL libraries and Nvidia's CUDA language. It has a `C`-like syntax and only allows for basic operations as well as a few intrinsic functions that provide additional operations on vectors and matrices.

3.2.1 GPU computing

In computer science, with GPGPU, acronym for general-purpose computing on graphics processing units, we mean the use of a graphics processing unit (GPU) for purposes other than traditional use in computer graphics. The most attractive feature of GPGPU regards the massive theoretical power offered, and consequently the reduced processing time, when compared to similar processing carried out by the CPU. From a purely performance point of view, we can go up to ten times faster than using traditional CPUs. This explains the reason why we decided to develop algorithms using GPU computing exclusively. In order to prove

[1]High Level Shader Language

this fact we will give, in Chapter 4, some comparisons between processing times using GPU and CPU.

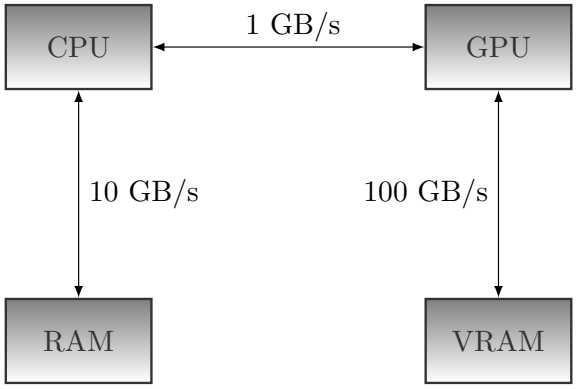


Figure 3.4: Relative bandwidth speeds between CPU, GPU and their correspondent memories.

Figure 3.4 shows relative memory bandwidth speeds between CPU and RAM, CPU and GPU, and GPU and VRAM. These numbers are just qualitative values to show the order of magnitude difference between bandwidths. It can be seen that transferring memory between CPU and GPU is the bottleneck. This is particularly important because, in GPGPU programming, the user generally needs to access the computation results back on the CPU. This requires copying the results from video memory to system memory, which is slow, but may be a negligible issue compared to the speed up from doing the computation on GPU. What we have observed explain why it is advisable to transfer all the input data, leave all the processing to the GPU, and finally transfer the results back to CPU for reading, rather than using the GPU to make a few counts each time. Instead, for graphics purposes, we typically use the computation result as an input to the rendering pipeline, so no transfer from GPU to CPU is needed.

Direct3D 11 can be used to perform extremely flexible rendering operations. However, there is a particular pipeline stage available for performing computations which may be useful for our purposes. The latter allows GPU to be used in such many applications as ray tracing and physical simulations, and in some cases, it can even be used for artificial intelligence computations. This pipeline stage is the compute shader, and it represents the implementation of a technology often referred to as DirectCompute. The greatest benefit of using DirectCompute, over other rendering APIs, such as CUDA and OpenCL, is that the performance of

a particular algorithm can easily scale with the user hardware. In other words, if a user has a high-end gaming PC with two or more high-end GPUs, then an algorithm can easily provide additional complexity to a game without needing to rewrite any code. The threading model of the compute shader inherently supports parallel processing of resources, so adding further work when more computational power is available is trivial. This is even more true for GPGPU applications, in which the user typically processes as much data as possible, as fast as possible. If an algorithm is implemented in the compute shader, it can easily scale to the current system capabilities.

3.2.2 Compute shader

It is a new type of shader, which is very similar to the existing vertex, pixel and geometry shaders, with much more general purpose processing capabilities. The compute shader is not attached specifically to any stage of the graphics pipeline, but interacts with the other stages via graphics resources such as render targets, buffers and textures. Unlike a vertex shader, which is executed once for each input vertex, or a pixel shader, which is executed once per each pixel, the compute shader does not need to have a fixed mapping between the data it is processing and the threads that are doing the processing. One thread can process one or many data elements, and the application can control directly how many threads are used to perform the computation. The compute shader also allows unordered memory access, i.e. the ability to perform writes to any location in a buffer. The last major feature is thread group shared memory. This allows threads' groups to share data, thus reducing bandwidth requirements significantly. Together, these features allow more complex data structures and algorithms to be implemented that were not previously possible in Direct3D, and can improve application performance considerably.

We will see, in Section 3.2.3, a summary of the main steps needed to compile the shader, start the GPU processes and finally read the results. Now we will provide a general intuition on how parallel GPU thread processing works. Actually, since several thousand threads can be active on the GPU simultaneously, it is important to have a solid understanding of how to harness all of these threads to fully exploit GPU power.

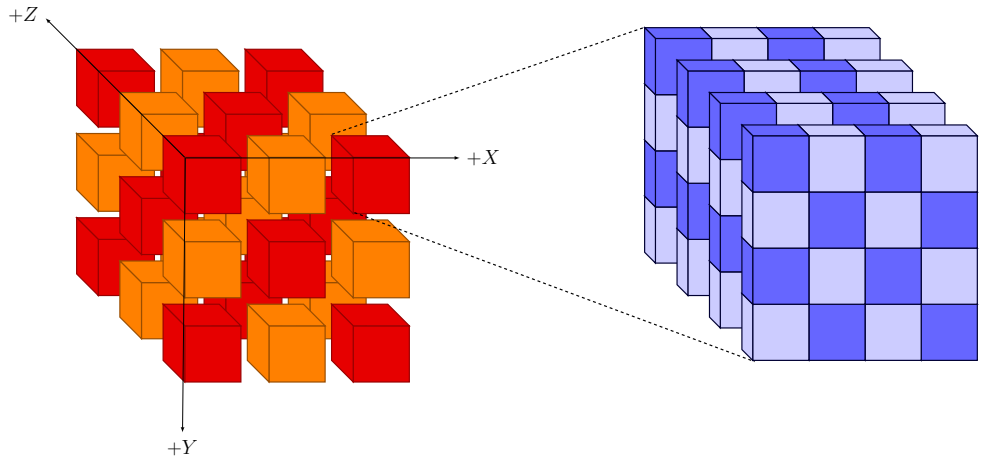


Figure 3.5: Visualization of the thread groups within a dispatch call.

Figure 3.5 helps to understand what happens when a `dispatch` command is called. It takes three unsigned integer parameters as input: x, y and z. These three parameters indicate how many groups of threads we would like to "dispatch" to execute the desired process. These three parameters provide the dimensions of a three-dimensional array of thread groups that will be instantiated, where the size of each parameter can range from 1 to 65535. In the example of Figure 3.5, the application calls `Dispatch(3,3,3)`, so a total of $3 \cdot 3 \cdot 3 = 27$ thread groups are created. Each group of threads would be identified by a unique set of indices within the specified dispatch arguments, ranging from 0 to `size` $- 1$ in each of the three dimensions. Notice that the `dispatch` call defines how many groups of threads are instantiated, and not how many threads are instantiated. The number of threads instantiated is defined by specifying how many threads will be created for each thread group with a `numthreads` function attribute preceding the compute shader `HLSL` program. As in the `dispatch` call, this statement defines the size of a three dimensional array, except that this array is made up of threads instead of thread groups. The size of each of these parameters depends on the shader model used but, for Shader Model 5 (`cs_5_0`), the x and y components must be greater than or equal to 1, the z component must be between 1 and 64, while the total number of threads $(x \cdot y \cdot z)$ cannot exceed 1024. Moreover, in order to optimize the GPU cores' occupancy, each component in `numthreads` must be multiple of 32 or 64 depending on whether Nvidia or AMD chips are used. Each of these threads can also be uniquely identified by its integer indices, ranging from 0 to `size` $- 1$ in each of the three dimensions. For example, if a compute shader calls `numthreads(32,32,1)`, a total

of $32 \cdot 32 \cdot 1 = 1024$ threads are instantiated for each thread group. If we use the dispatch call example from above, we would have a total of $27 \cdot 1024 = 27648$ threads instantiated.

A relevant problem at this point is how to access the individual threads inside the shader. We have already told how the compute shader can only read data from a memory buffer. Fortunately, each call function has a few intrinsic parameters that provide indexes for the selected group and the current thread within the group. The main ones are listed below:

`SV_GroupID` gives the indices for which thread group a compute shader is executing in. Possible values vary across the range passed as parameters to `dispatch`.

`SV_GroupThreadID` gives the indices for which an individual thread within a thread group a compute shader is executing in. Possible values vary across the range specified for the compute shader in the `numthreads` attribute.

From the geometric interpretation of the thread locations discussed above, we can also consider an overall unique identifier for each of the threads within the complete `dispatch` call. This identifier could essentially locate the thread with an X, Y, and Z coordinate within the three-dimensional grid. These indices are given by `SV_DispatchThreadID`, computed from the following formula:

$$\texttt{SV\_DispatchThreadID} = \texttt{SV\_GroupID} \cdot \texttt{numthreads} + \texttt{GroupThreadID} \quad (3.1)$$

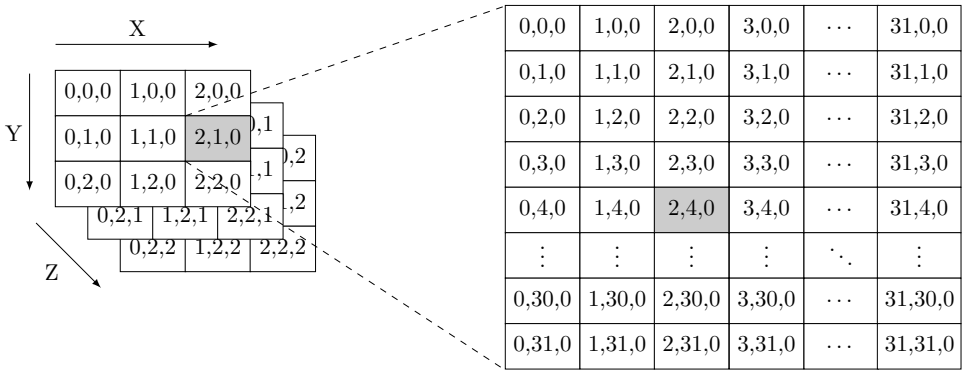


Figure 3.6: Example of a `dispatch` and `numthreads` invocations.

Retrieve the above example for which 27 thread groups were instantiated, each one containing 1024 threads. A representation of that

operation is given in Figure 3.6. Now, in order to give an example of index computation, we select a group and a thread within that group:

$$\texttt{SV\_GroupID} = (2, 1, 0)\,, \qquad \texttt{GroupThreadID} = (2, 4, 0)$$

Remember that we called `numthreads(32,32,1)`, and, using Equation (3.1), we can obtain the unique thread identifier:

$$\texttt{SV\_DispatchThreadID} = (2, 1, 0) \cdot (32, 32, 1) + (2, 4, 0) \;=\; (66, 36, 0)$$

We conclude this section, regarding the use of compute shaders, with a recommendation to the reader. The choice of allocating a number of thread groups or threads themselves is individual and very complex. As mentioned above, it also depends on which graphics chip is used as hardware. Therefore, since the optimisation is essentially based on this choice, an incorrect combination would lead to a waste of computing power. The advice is to rely on trial and error finding the combination of values that gives the best results.

3.2.3 GPGPU example

To clear up the above concepts, we now provide an example on how to use GPGPU to do a very simple task. In particular, we want to perform a McLaurin series of the exponential, given by the following formula:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

The idea is to compute all the coefficients of the series using GPU. So, let's explain the key steps for compiling and running a shader. In Algorithms 1 and 2, external `C#` and shader `HLSL` pseudo codes are shown. As already explained in the previous section, the shader function does not return anything, but it works through the use of special buffers called Unordered Access View (UAV). First and foremost, a UAV buffer and a staging copy, which will only be used to read results back on CPU, are created from the `C#` code. Afterwards, a shader program is compiled and used to create a shader object through the device interface, which can then be loaded into the compute shader stage through the device context interface. We now run the code inside the `Example.hlsl` file using the `Dispatch` command which will execute the function as many times as we want. This function does nothing more than writing the coefficients of McLaurin's series to our buffer. At the end of the `Dispatch` call, we can read the results by

making a copy of the buffer associated with the UAV on a "staging" type and doing a `Map` operation. Please note that the `Dispatch` method is asynchronous. This means that the code will continue to work even if the shader has not finished its task, unless some functions access the buffer. For example, an instruction that forces the compute shader code to end is the `Flush` method. Normally, it will be preferable to go ahead with the rendering while GPU finishes the computation.

```
struct ResultData {
    float functionResult;
    int x;
    float padding1;
    float padding2;
}
// Creating buffer and its view
accessView = CreateUAV(out buffer);
resultBuffer = CreateStaging();

// Compiling shader and loading buffer view
shaderCode = Compile("Example", "CS", "cs_5_0");
shader = new ComputeShader(device, shaderCode);
ComputeShader.SetUnorderedAccessView(0, accessView);
ComputeShader.Set(shader);

// Starting 64 × 64 thread groups
Dispatch(64, 64, 1);

CopyResource(buffer, resultBuffer);
Flush();

// Resetting compute shader when finished
ComputeShader.SetUnorderedAccessView(0, null);
ComputeShader.Set(null);

// Mapping results back to CPU for reading
MapSubresource(resultBuffer, out stream);
ResultData[] result = stream.Read();
UnmapSubresource(buffer, 0);
```

Algorithm 1: `C#` example code.

```
struct BufferStruct {
    float value;
    int x;
    float padding1;
    float padding2;
}
RWStructuredBuffer<BufferStruct> outBuffer;

// Factorial and McLaurin methods
float Factorial(int n) { ... }
float McLaurin(float x) { ... }

// Starting 32 × 32 threads for each group
[numthreads(32, 32, 1)]
void CS(uint3 ID : SV_DispatchThreadID) {
    int stride = 32 * 64;
    int idx = ID.y * stride + ID.x;
    outBuffer[idx].value = McLaurin(idx/1000);
    outBuffer[idx].x = idx;
    outBuffer[idx].padding1 = 0;
    outBuffer[idx].padding2 = 0;
}
```

Algorithm 2: `Example.hlsl` file.

3.3 Implementation

As we have already told at the beginning of Section 3.2, we will not explain in detail how the graphics engine is made. However, a detailed discussion can be found in the Direct3D Rendering Cookbook [13] which also provides the `C#` code using the SharpDX libraries. Instead, we design the core in order to allow user interaction and additional transformations with scene objects. All these computations are performed parallel to the rendering pipeline and, as soon as they are finished, are passed to the graphics engine that displays them. These operations are accessible through the menu bar on the top. From Figure 3.7a, we can find three different sub-menus:

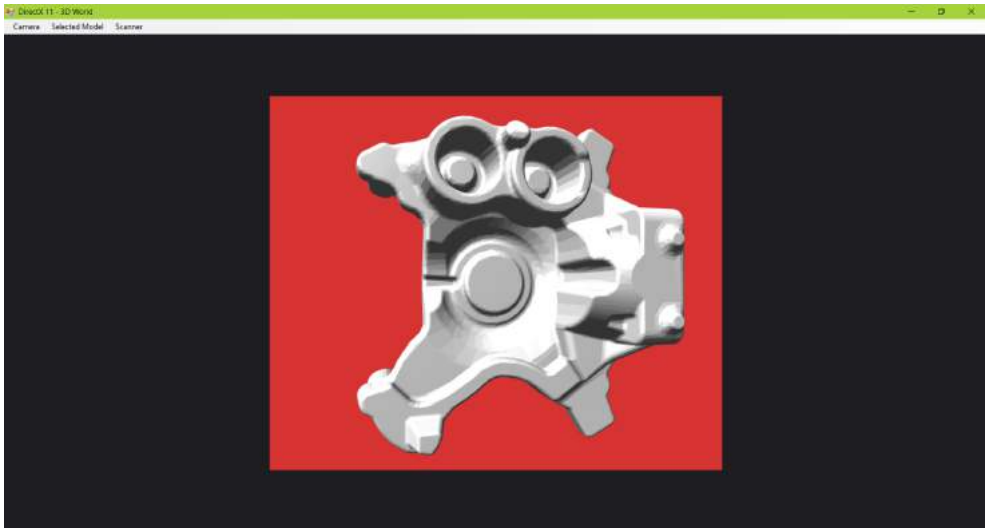
Camera allows the user to control the camera, by changing its position and orientation, and to control the lighting inside the scene.

Selected Model provides the user with a series of possible operations

and transformations on the selected mesh. For example, it is possible to resize, move and rotate it, or even to remove it from the scene.

Scanner contains commands to start the reconstruction, emulating a particular scanner, or score computation, which we will discuss in detail later.

(a) Example of user interaction.



(b) A mesh is added to the scene.

Figure 3.7: Screens of developed simulator.

To control the camera, it is straightforward to multiply the camera view matrix, discussed in Section 2.4, by a homogeneous matrix which represents the rotation or translation that we would like to perform. While, to change the lighting, we simply need to switch the pixel shader used. In Figure 3.7a we can observe a demo of the program where a 190×230 mm plane is placed at the world's origin. While, in Figure 3.7b we can see the view from the camera after having added a mesh to the scene.

The functions performed on the selected model are a bit more interesting. First of all we need to provide a method to select a mesh in the scene. This is quite simple using a `MousePicking` algorithm as the one shown in Algorithm 3, where, given the mouse coordinates, we compute the direction of the ray passing through that point.

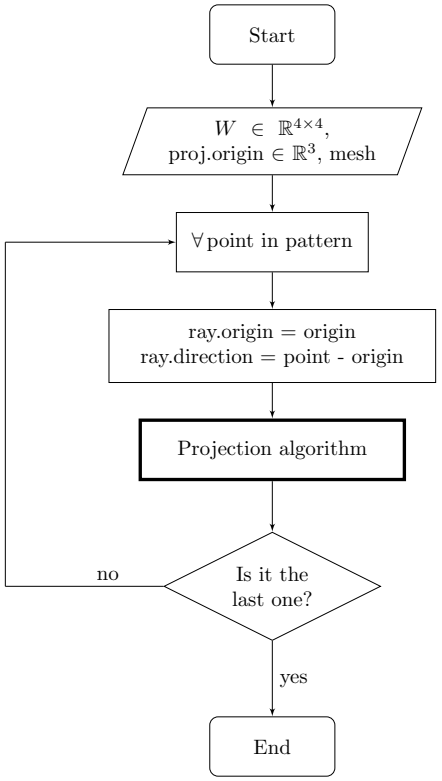
Data: `mouseX, mouseY`
Result: `Ray(origin, rayDir)`
`x = 2 · mouseX/screenWidth - 1;`
`y = 1 - 2 · mouseY / screenHeight;`
`temp = transform((x,y,-1), inverse(projectionMatrix));`
`rayDir = transform(temp, inverse(viewMatrix));`

Algorithm 3: `MousePicking` algorithm.

Once we have selected an object in the scene, we are able to resize, move and rotate it along each axis. This is done by multiplying the world matrix of the selected object, discussed in Section 2.4, by an homogeneous matrix which represents the desired scaling, translation or rotation.

The most interesting and original part concerns the scanner functions. Here we will go on a step-by-step description of what is done, providing some useful flowcharts too. First of all, we would like to inform the reader of the simplification made with respect to the model described in Section 3.1. Indeed, to simplify implementation, only one pattern, and not a succession, is projected. However, it would not be difficult to extend this functionality. The reconstruction algorithm is based on two steps: the former is to project all the pattern points on the scene, while the second one determines correspondences between the latter and the points captured by the camera.

The projection procedure is summarised in the flowchart of Algorithm 4. Given a pattern, for each point, we generate a ray passing through that point and with origin that of the projector. Then, for each mesh of the scene, the actual projection algorithm is called.



Algorithm 4: Projection procedure.

The projection algorithm, outlined in Algorithm 5, is used to search for all the intersections between ray and mesh. Basically, the algorithm does nothing but looking for intersections with all mesh triangles, applying the Möller-Trumbore method described in Section 2.3.1. Iteration on all mesh triangles is an heavy operation, as most models have tens of thousands of triangles. It is therefore very useful to firstly check if the ray hits the bounding box containing the object. Otherwise, we can immediately move to the next ray, saving a lot of computational power and drastically reducing the time. Another expedient concerns the discussion made in Section 2.4. Actually, the coordinates of the vertices of each mesh triangle are relative to the mesh reference frame, while the ray lives in the world reference frame. It is thus needed to transform the coordinates of the vertices using the `worldMatrix` of the mesh. Even better is to transform the ray's origin and direction using the inverse of the `worldMatrix`, so as to bring it into the mesh reference system. It is easy to prove that the last is the best way, since we only do one transformation compared to transforming tens of thousands of vertices each time. It is important to emphasize that, applying this trick, is then necessary to transform the

coordinates of the intersections found with respect to the world reference frame. The last thing to observe, at the end of the diagram in Algorithm 5, regards removing all redundant intersections. Indeed, each ray could intersect more than one triangle, but we are only interested in the closest point to the origin of the projector, as the others are all occluded or in shadow.

As an example, a result of the projection stage is given in Figure 3.8. The scanned object is a mechanical piece of steel and the projected pattern consists of a grid of 500×500 dots. The working distance was set to 40 cm. This choice was made in order to obtain a good level of detail but, at the same time, to cover the entire object. Indeed, the density of the projected points decreases with the distance of the scanner from the scene.

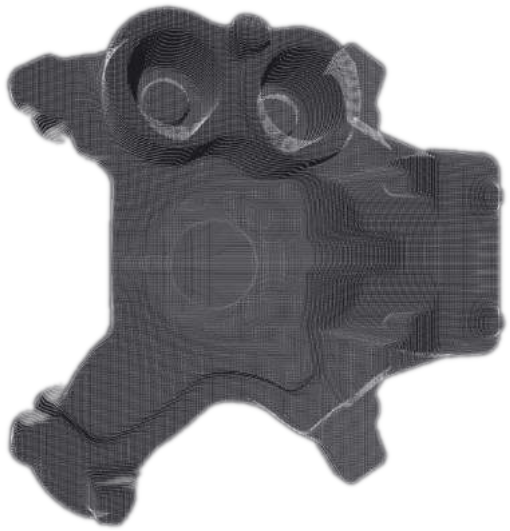
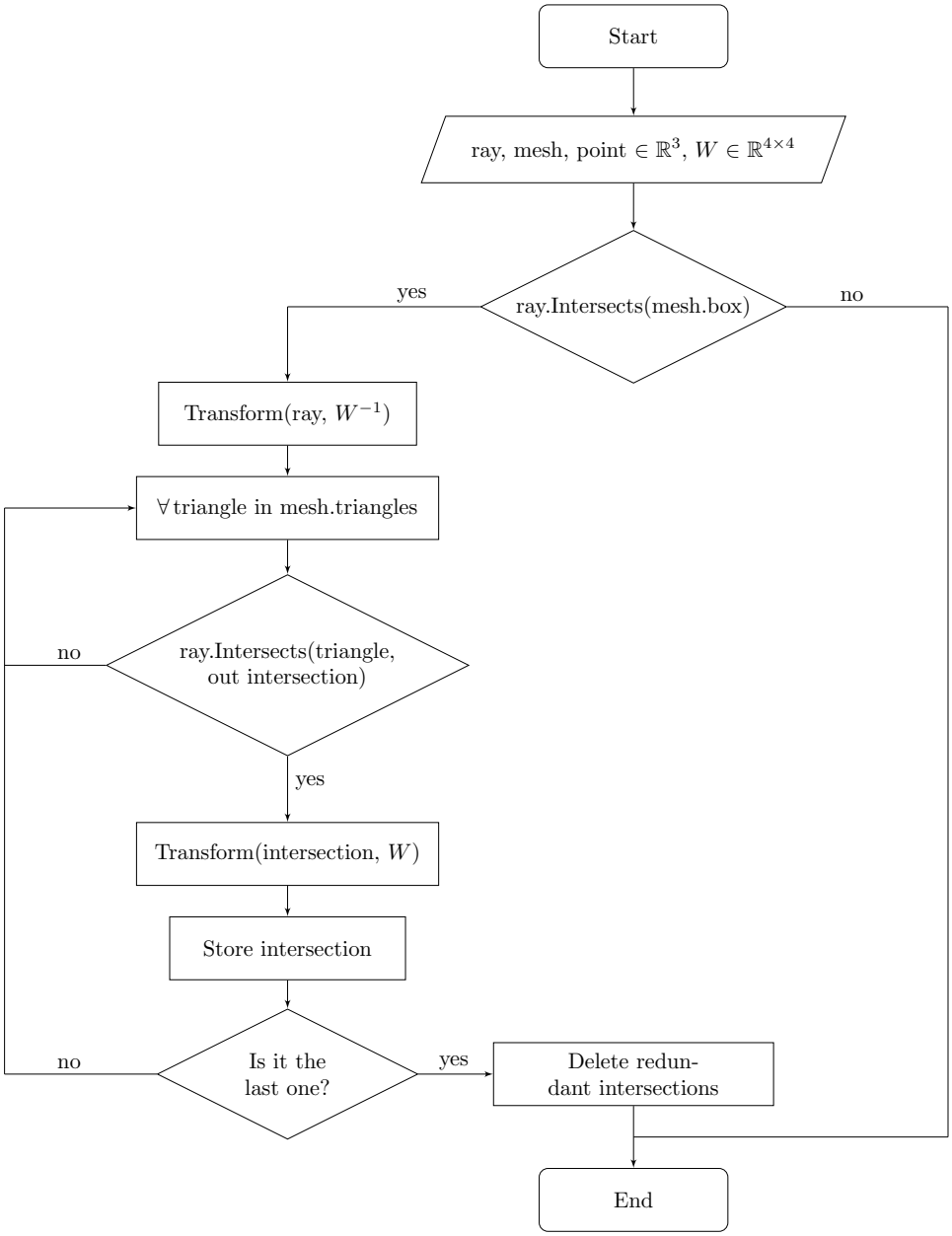
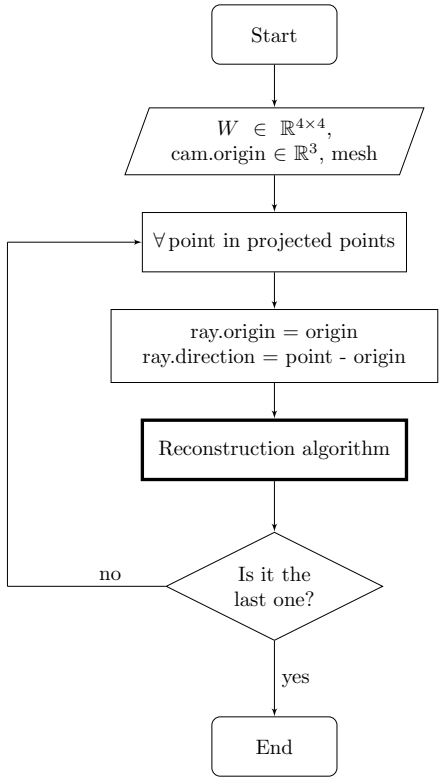


Figure 3.8: Projection result example.

When all pattern points have been projected, the first step is finished, so we move onto the reconstruction procedure, which is summarized in the diagram of Algorithm 6. Given the list of projected points, for each one, we compute the ray passing through that point and with origin that of the camera. At this point, for each mesh of the scene, the actual reconstruction algorithm is invoked.

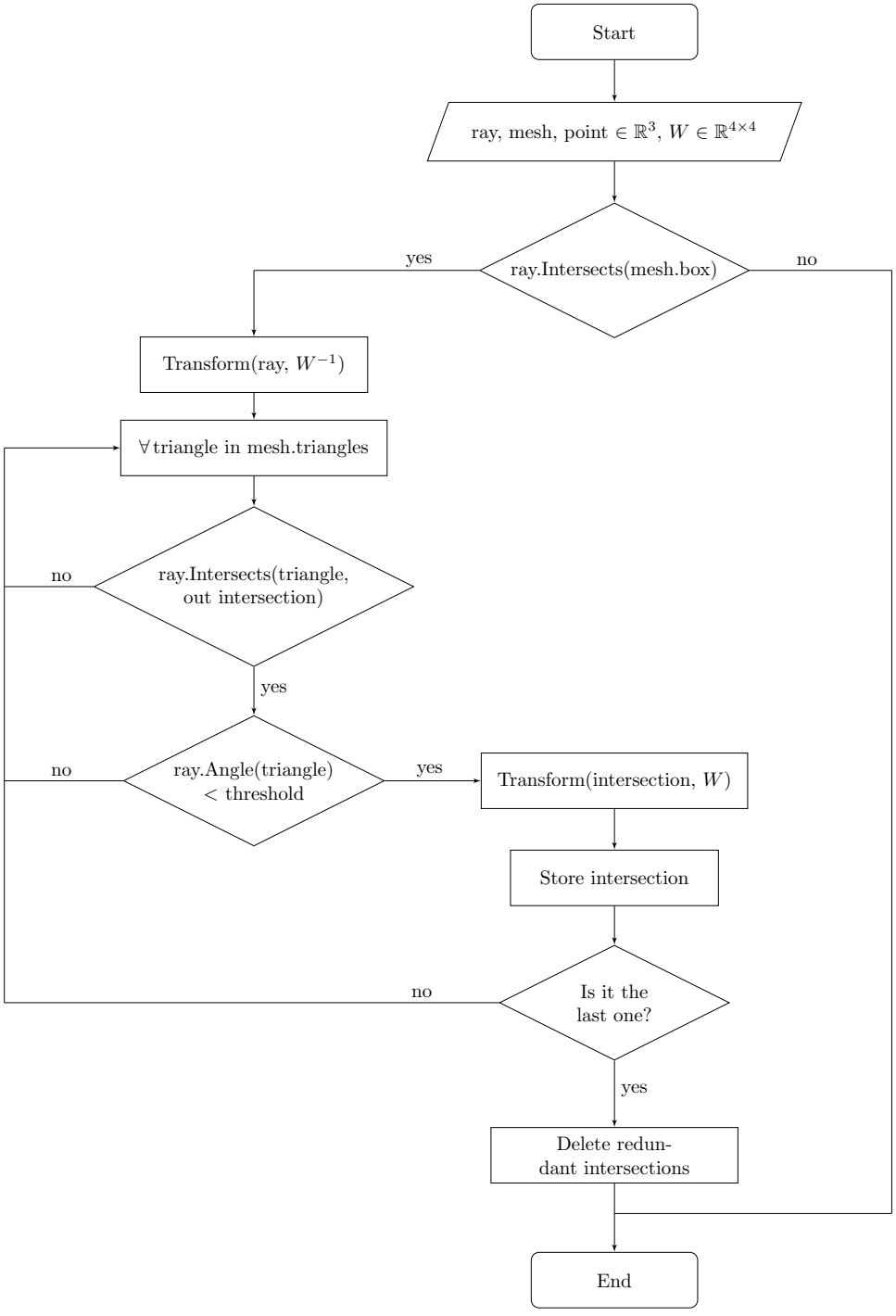


Algorithm 5: Projection algorithm.



Algorithm 6: Reconstruction procedure.

The reconstruction algorithm, outlined in Algorithm 7, is used to find the final 3D point cloud obtained by the structured light scanner. Basically, similarly to what we did in the projection algorithm, we look for all the intersections between each camera ray and all mesh triangles. Even in this case, we use the two expedients of before regarding the transformation of ray coordinates and the prior verification of incidence on the mesh bounding box. Looking at the scheme, we notice a particular difference from before. The intersection, computed by the Möller-Trumbore method, is stored only if the angle between the ray and the normal of the considered triangle is less than a certain threshold. Actually, it would be much more realistic to use Snell's law, introduced in Section 2.3.2, to calculate the power of the reflected beam, and thus, make the intersected point in the final point cloud more or less visible. To simplify this operation, we decided to discretize and evaluate a certain threshold beyond which the ray is not reflected and, consequently, does not carry the information of the intersected point to the camera. Finally, the intersection is stored in the 3D point cloud if it is more or less equal to the projected point considered.



Algorithm 7: Reconstruction algorithm.

Finally, we got a list of reconstructed points by emulating a structured light scanner. That list could be sent to the graphics engine for on-screen viewing, or it can be used to initiate a score computing, which we will discuss in detail in Chapter 4. As an example, a result of the reconstruction stage is given in Figure 3.8. The scanned object is the same used previously in the projection procedure.

Figure 3.9: Reconstruction result example.

3.4 Encountered problems

We thought of describing to the reader the main problems encountered during the design and debugging of the simulator, also listing the solutions adopted. The three problems we are going to deal with have significantly slowed down the project. Since there is not much documentation on the net, we thought it would be useful to describe them in detail.

3.4.1 Memory buffer structure

For efficiency, memory buffers are mapped such that values do not straddle GPU registers. Each register is four floats in size, i.e. 16 bytes, so buffer structures must be multiple thereof on the GPU. The `C#` structure should be padded accordingly to use it as a convenience for mapping data and to avoid misalignments in memory. In other words, we need to ensure

that the location of those bytes in memory will not differ in the CPU and GPU versions of the structure. Therefore, the solution involves inserting manually the appropriate padding to ensure proper 16-byte alignment. Let's give an illustrative example. Suppose we need the information about the bounding box of a particular mesh inside a shader function. A bounding box is characterised by the minimum and maximum points, i.e. two `float3` variables in `C#`. Each variable is an array of three floats so in total it occupies 12 bytes. The fastest way is to create a constant buffer containing the two vectors. However, the two points are represented in memory by 24 bytes, while we need a multiple size of 16 bytes. Therefore, we need two integer paddings (4 bytes each) in order to reach a size of 32 bytes. The right `HLSL` code is provided below.

```
cbuffer BoundingBox :  register(b0) {
    float3 minMeshBox;
    int padding1;
    float3 maxMeshBox;
    int padding2;
}
```

Algorithm 8: Memory alignment example.

3.4.2 Shader matrix ordering

Debugging a shader is very complex as there are no simple analysis tools for `HLSL` like the most common programming languages. It is therefore necessary to verify that each operation carried out on the data is correct. In this regard, we found that loading a 4×4 homogeneous matrix inside a buffer, the latter was automatically transposed into the shader. We then discovered this was due to the matrix ordering type defined inside the shader. Actually, data in a matrix is loaded into constant registers before a shader runs. There are two choices for how the matrix data is read: in row-major order or in column-major order. Column-major order means that each matrix column will be stored in a single constant register, and row-major order means that each row of the matrix will be stored in a single constant register. This is an important consideration for how many constant registers are used for a matrix. Therefore, since `C#` math libraries use row-major order, we had to transpose matrices, before loading into buffer, so that they were in column-major order for `HLSL`.

3.4.3 TDR issue

The main problem encountered during the debugging of the simulator concerns the overcoming of the Timeout Detection and Recovery (TDR). Actually, the GPU scheduler, which is part of the DirectX graphics kernel subsystem, detects if the GPU is taking more than the permitted amount of time to execute a particular task. The default timeout period in Windows Vista and later operating systems is 2 seconds. If the GPU cannot complete the current task within this period, the operating system suspends the GPU and diagnoses that it is frozen. There are two possible solutions to that problem. The former, which is not suitable for a final software product, consists in completely disabling TDR or increasing it as needed. Specifically, this can be done by editing the corresponding registry key in the operating system. The latter, much more elegant, consists in modifying the GPU process to avoid reaching the TDR. Specifically, to prevent timeout detection from occurring, we should ensure that graphics operations take no more than 2 seconds in end-user scenarios. Obviously, whether TDR is exceeded or not depends on the hardware used to run the program. In our case, most of the tests reported in Chapter 4 were performed using a laptop with the specifications listed in Table 3.1. By using patterns of a million or more points, we have experienced the video driver suspension. Therefore, for demonstration purposes, we decided to completely disable timeout. The same issue did not occur using the workstation offered by Euclid Labs, with specifications from Table 3.2.

CPU	Intel Core i7-4720
RAM	8 GB DDR3
GPU	Nvidia GTX 960M
	640 CUDA cores
VRAM	2 GB GDDR5

Table 3.1: Notebook specs.

CPU	Intel Core i9-7900X
RAM	64 GB DDR4
GPU	Nvidia Titan V
	5120 CUDA cores
VRAM	12 GB HBM2

Table 3.2: Workstation specs.

Chapter 4

Results

In this chapter we want to validate the goodness and robustness of our simulator through several tests. For this purpose, we will first evaluate the reconstruction on some 3D models, and then compare it with the results obtained using Zivid and Photoneo scanners. The most interesting and unedited part, reported in Section 4.5, concerns the analysis of the reconstruction by changing the fundamental parameter of each scanner, i.e. the baseline. The latter will show how the proposed simulator can be a valid tool to provide customers a quick idea of the most suitable baseline for a particular scenario, and consequently recommend a scanner rather than another. Moreover, in the future this system could be incorporated into a larger project to study and design new 3D scanners.

Two different mechanical parts were used to perform tests. The former, a $50 \times 50 \times 20\,\text{mm}$ parallelepiped, is a very simple solid and it is useful for making precision measurements and comparisons with the results obtained by the scanners. The second one is a complex mechanical piece, shown in Figure 4.2, and it is suitable for evaluating shadow and occlusion phenomena.

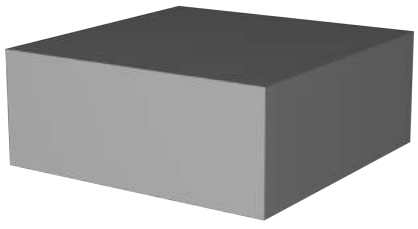


Figure 4.1: Side view of the scanned parallelepiped.

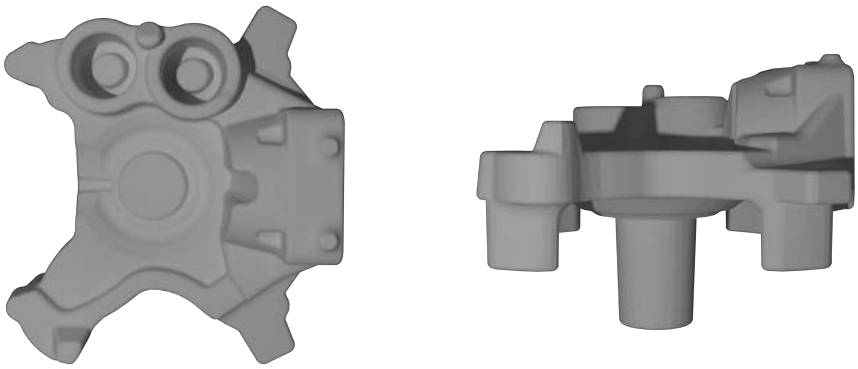


Figure 4.2: Front and side views of the scanned mechanical part.

In the following sections, we will talk about score and coverage, i.e. the tools used to evaluate the simulator and compare the results. Actually, we are interested in comparing the point cloud resulting from the reconstruction algorithm either with the 3D model or with the point cloud obtained from a real scanner. To make this comparison, we used an algorithm, developed by Euclid Labs, which tries to superimpose point clouds by moving one of the two until coverage is maximized. To be more precise, we define with R the set of simulated point and with P the set of points belonging to the real model or the real scanner reconstruction. Then, we define the intersection between the two sets with $\Omega = \{R \cap P\}$. At this point, with "score" we mean the ratio between the cardinalities of Ω and R. On the other hand, with "coverage" we mean the ratio between the cardinalities of Ω and P.

$$\text{score} = \frac{|\Omega|}{|R|}, \qquad \text{coverage} = \frac{|\Omega|}{|P|}$$

4.1 GPU vs CPU

Initially, all the work presented in this thesis was developed entirely to be performed on CPU, using parallel computing to exploit all the available power. However, we immediately realized that, by increasing the number of projected points, the time required for the reconstruction gradually became too high. So we had to move our algorithm to GPU rewriting some portions of code. In order to highlight the improvements achieved through this step, we decided to compare the execution times of the entire scanning algorithm using CPU and GPU. Figure 4.3 shows

the execution times as the number of projected points increases, from 100 thousand to 4 million. Both trends are polynomial and we can easily observe how the GPU outperforms the CPU. For example, projecting a pattern of 4 million points, the GPU takes 32 seconds against the 7.5 minutes needed by the CPU to complete the scanning. This is due to the fact that a mid-range GPU has about a thousand cores, while a common CPU only 4. Therefore, parallel computing is much more optimized using graphics cards. However, it is interesting to note that this is no longer the case when only a few points are considered. Figure 4.4 shows the execution times trend in the case of a few thousand projected points. We can immediately notice that, for less than 10 thousand points, CPU takes less time to reconstruct the scene. This is because the GPU, whatever the input, takes about a second to start the processes. In conclusion, in case of a few points it is convenient to use CPU. Otherwise, in all other cases, GPU always has the best.

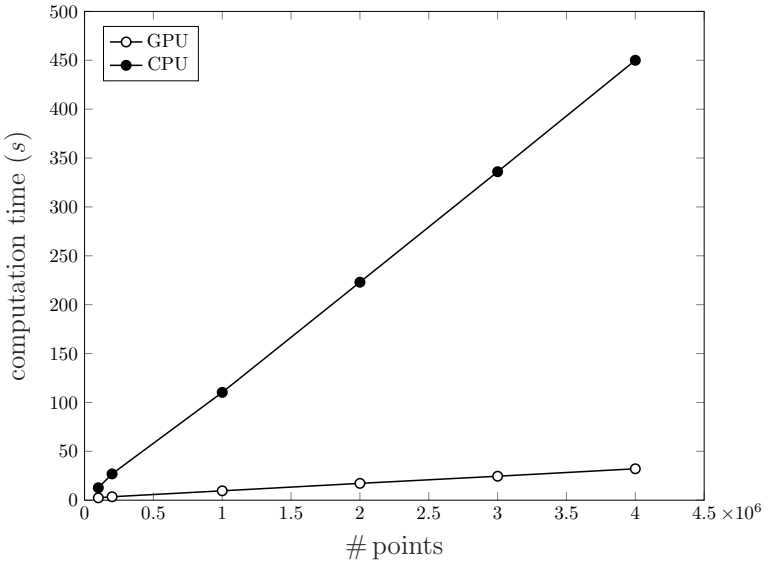


Figure 4.3: Reconstruction times using CPU and GPU.

4.2 Tests on 3D models

As already mentioned, a 3D model is nothing more than a set of triangles, while we need a point cloud in order to make a comparison. So, we need an algorithm which allows "sampling". Essentially, the latter divides each mesh triangle into smaller ones, and then generates a point

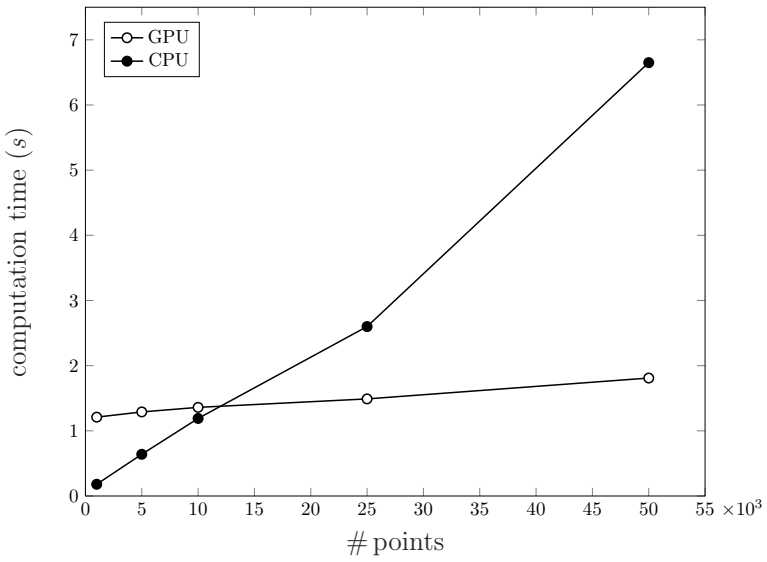


Figure 4.4: Reconstruction times using CPU and GPU (with few points).

cloud by taking all the vertices, avoiding repetitions. These first tests were used to evaluate the reconstruction capacity of the simulator. We first sampled our complex 3D model, and then launched the scanning using the algorithm described extensively in the previous chapter. The result of overlapping is shown in Figures 4.5-4.6, depending on whether Snell's law is implemented or not. In violet we find the points resulted from the sampling of the 3D model, while in orange we observe the point cloud reconstructed by the simulator. The working distance was set to 40 cm and, in the first phase, one million points were projected, arranged along a 1000×1000 grid.

The scoring algorithm provided us with the data presented in Table 4.1, depending on the number of points of the projection pattern. We got a very good score, while the coverage is low, but we can expect that since the sampling takes into account the whole object and not only the part facing the scanner. After all, this is only a qualitative assessment of how the reconstruction algorithm works. Please note that, comparing the two figures, we can observe that, using Snell's law, the reconstruction changes considerably and it is much more faithful to reality. Indeed, as the camera is placed within the scene, the power of the reflected ray at the steepest points is too weak to return to the sensor, and so, Snell's law should be implemented. However, this will be confirmed in Sections 4.3 and 4.4, where we will see the behaviour of real scanners.

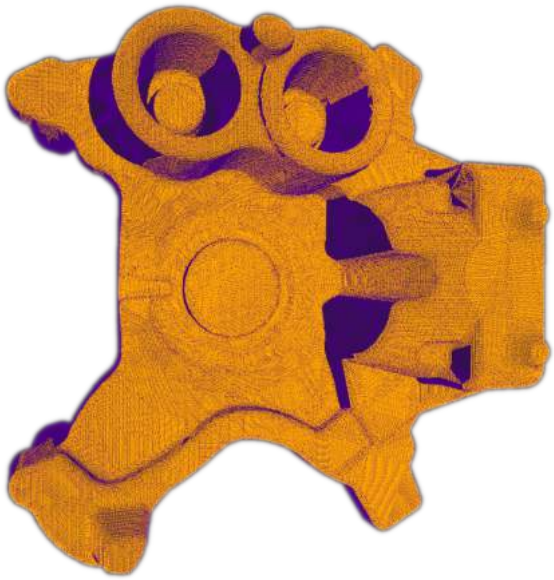


Figure 4.5: Reconstruction overlaid on 3D model without Snell's law.

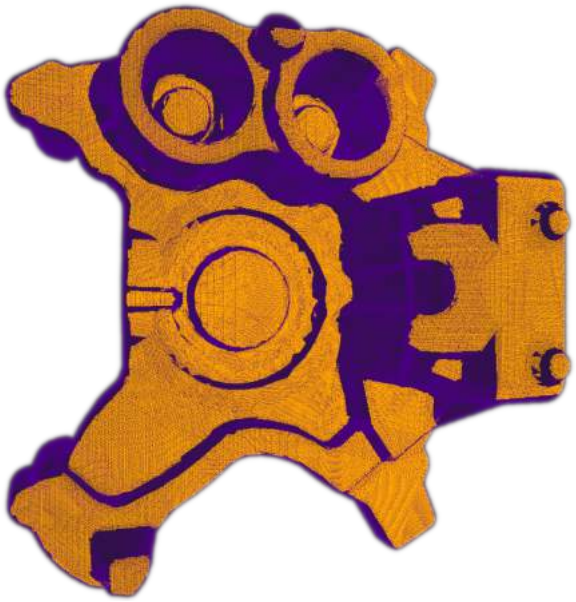


Figure 4.6: Reconstruction overlaid on 3D model using Snell's law.

# points	Score [%]	Coverage [%]
$\sim 200\,\mathrm{K}$	99.18	26.47
$\sim 1\,\mathrm{M}$	99.91	26.85
$\sim 4\,\mathrm{M}$	100	27.03

Table 4.1: Reconstruction performance on 3D model.

4.3 Comparison with Zivid scanner

Together with Euclid Labs, we decided to emulate the Zivid scanner, which currently represents the state of the art regarding 3D structured light scanners [14]. It consists of a high dynamic range full color camera (right, Fig. 4.7) and an active lighting projector (left, Fig. 4.7). Within a tenth of a second it returns a 3D point cloud with RGB colors captured with the same sensor chip.

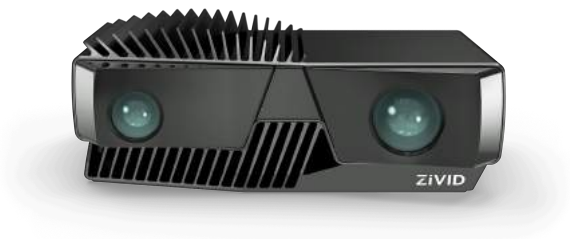
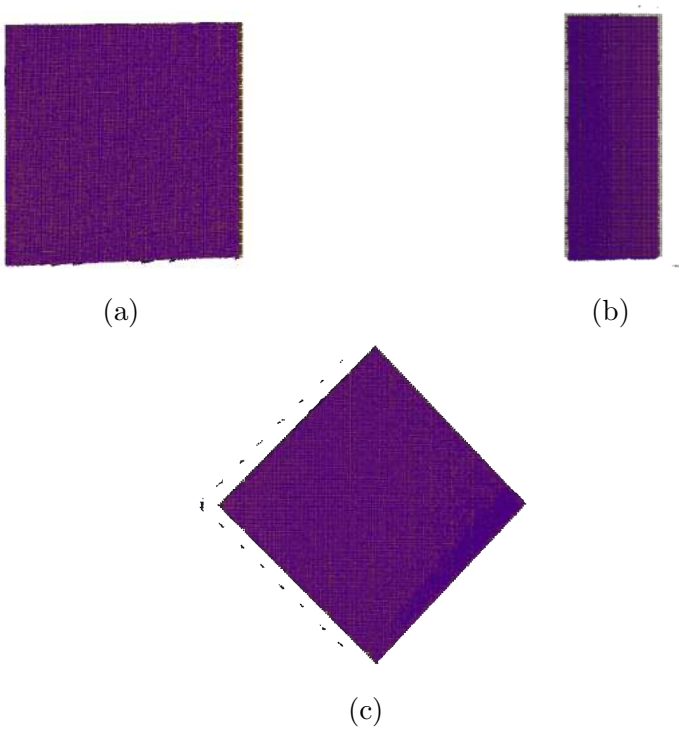


Figure 4.7: Zivid scanner.

Acquisition rate	$\sim 100\,\mathrm{ms}$
Working distance	$0.6 - 1.1\,\mathrm{m}$
Field of view	$780 \times 490\,\mathrm{mm}$ @ $1.1\,\mathrm{m}$
Depth resolution	$0.2\,\mathrm{mm}$ @ $1.1\,\mathrm{m}$
Baseline	$135\,\mathrm{mm}$

Table 4.2: Zivid data-sheet.

For this test session, the working distance was set to 60 cm. During the projection phase, we used a 1000×1000 grid for a total of one million points. In order to asses the likelihood of the simulator in emulating the Zivid scanner, we took a series of precision measurements by moving the parallelepiped along a 10×10 cm chessboard. For each pose, we captured the scene with the scanner and saved the corresponding point cloud. After that, we replicated the pose of the object in the simulator and launched the reconstruction. We evaluated the overlap using the algorithm described at the beginning of this chapter, and the results of the comparison are reported in Table 4.3. We immediately notice how, in general, the results are excellent, and confirm that the simulator emulates well the Zivid model. We also tried to evaluate performance by moving the parallelepiped further away or rotating it, and some results are visible in Figure 4.8. Even in such cases, the simulator continues to replicate the behaviour of the scanner well. In Figure 4.8c, we can see some points that do not belong to the object, recognized by the scanner but not by the simulator. This is due to light reflection and noise phenomena that are not easily replicable mathematically.



(a) (b)

(c)

Figure 4.8: Reconstruction overlaid of the parallelepiped, in three different poses, compared to Zivid scanner.

δ**x** [mm]	δ**y** [mm]	**Score** [%]	**Coverage** [%]
0	0	99.98	96.98
20	0	100	99.13
40	0	100	99.88
-20	0	100	98.61
-40	0	100	99.57
0	20	100	98.82
0	40	100	99.29
0	-20	100	98.39
0	-40	100	97.72
40	40	100	99.91
40	-40	100	99.84
-40	40	97.13	99.82
-40	-40	99.92	98.77

Table 4.3: Reconstruction performance compared to Zivid scanner.

We also compared the results obtained with the second test object, the more complex one. Looking at Figure 4.9, we can notice major differences, indeed simulator performance drops a bit. In particular, we obtained a score of 94.64% and a coverage of 99.33%. This is due to the fact that our simulator is not able to perfectly emulate the Zivid model. Actually, as expected, the choice to use ray casting method leads to simplify a bit the model. On the contrary, by following the principle of ray tracing, we could have better replicated the phenomena of reflection and refraction, which are very common in nature and "dirty" the results obtained by the scanner. However, it should be noted that it may be useful to have a noiseless result for use in detection and recognition algorithms and, in this case, the simulator is preferable to the scanner.

Figure 4.9: Reconstruction overlaid of the complex object compared to Zivid scanner.

4.4 Comparison with Photoneo scanner

Together with Euclid Labs, we decided to emulate another 3D scanner, in particular the Photoneo PhoXi 3D L, which is also based on structured light technique [15]. However, instead of using simple white light like in a normal multimedia projector, it exploits a proprietary projection system based on coherent laser radiation, which allows better access to every corner of the scanning area. From Table 4.4, showing the scanner specifications, we carefully observe that the baseline is much larger than before. Given the larger size and baseline of this scanner compared to Zivid, we can easily imagine that Photoneo is more suitable for capturing larger scenes. However, for comparison purposes, working distance and projected pattern remained unchanged. The test procedure is also identical to that described in the previous section.

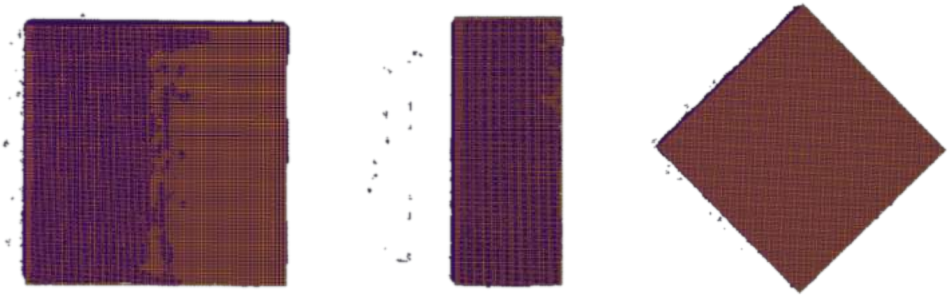
Figure 4.10: Photoneo PhoXi 3D L scanner.

Acquisition rate	$\sim 400\,\text{ms}$
Working distance	$0.87 - 2.156\,\text{m}$
Field of view	$1300 \times 975\,\text{mm}$ @ $2\,\text{m}$
Depth resolution	$\sim 1\,\text{mm}$ @ $2\,\text{m}$
Baseline	$550\,\text{mm}$

Table 4.4: Photoneo PhoXi 3D L data-sheet.

δ**x** [mm]	δ**y** [mm]	**Score** [%]	**Coverage** [%]
0	0	99.78	96.73
40	0	95.84	97.34
-40	0	95.46	96.04
0	40	99.88	99.28
0	-40	99.22	99.44
40	40	95.38	96.85
40	-40	96.91	98.40
-40	40	90.33	96.54
-40	-40	99.11	98.76

Table 4.5: Reconstruction performance compared to Photoneo scanner.



(a) (b) (c)

Figure 4.11: Reconstruction overlaid of the parallelepiped, in three different poses, compared to Photoneo scanner.

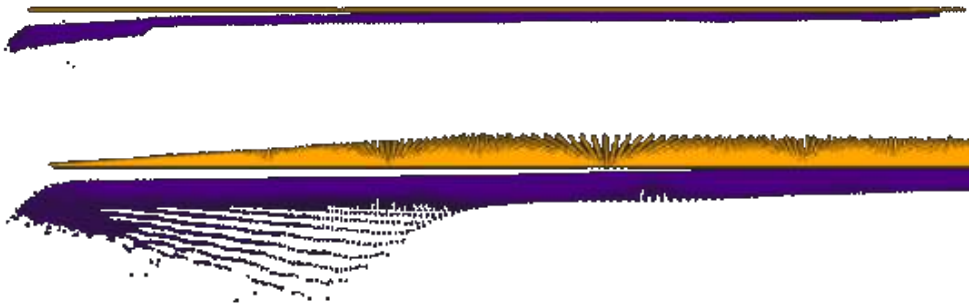


Figure 4.12: Two closer views of the Figure 4.11a.

Figure 4.11a shows the overlap with the lowest score among those listed in Table 4.5. We went deep to identify the reason for such a drop in performance and, rotating the view, we obtained the detail of Figure 4.12. We immediately observe that, in addition to the noise that is always present, the reconstruction carried out using Photoneo (in violet) has a bending error on the edge of the scanned object. This error, however, is not repeated in the simulated reconstruction (in orange) which more faithfully reproduces the squared shape of the parallelepiped.

As done previously for the Zivid scanner, we also compared the results obtained with the second test object, the more complex one. Looking at Figure 4.13, we can notice major differences, indeed score and coverage drop a bit. In particular, we obtained a score of 89.52% and a coverage of 87.96%. The reason may be addressed to the problem observed previously

in Figure 4.12. However, even in this case, the result of the simulator is to be preferred to that of Photoneo because it is noiseless. Therefore, it is more suitable for use in detection and recognition algorithms.

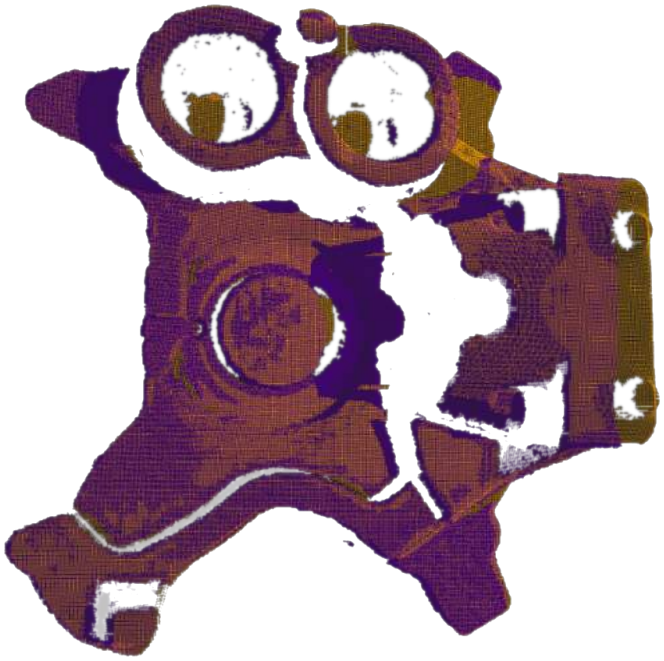


Figure 4.13: Reconstruction overlaid of the complex object compared to Photoneo scanner.

4.5 Baseline analysis

In this section we will show how the simulator can be a valid tool to evaluate occlusion and shadow phenomena. For commercial purposes, this feature is certainly the most relevant. Actually, it would save valuable time in deciding which 3D scanner is best suited to solve a particular problem without having to test it on site. Thanks to the simulator and the 3D model of the object to be recognized, it is possible to analyse different configurations. For example, we can change the baseline and/or the working distance and see the effect on the reconstruction in "real time". With the aim of showing, at least in part, this characteristic, we still take into consideration the complex object of Figure 4.2 in three separate sessions, where the working distance differs. It should be noted that the following results and graphs take account of two simplifying assumptions.

1. We consider a baseline between 50 and 800 mm. Below 50 mm would not be significant, as it would be very difficult to achieve at a constructive level. While, as will be confirmed by results, beyond 800 mm, performance drops drastically and cannot be taken into account for common industrial applications. Moreover, negative baseline values are considered. In the latter case, the camera is located on the left of the projector and no longer on the right.

2. We do not take into account the fact that, when the inclination of the camera varies, we have a different thickness of the projected lines. Indeed, a series of projected lines of equal thickness could appear as in Figure 4.14, if the angle between baseline and camera normal is very small. We would be in a situation where, by reconstructing the first line, we have a lot of well defined pixels, while, by reconstructing the furthest, we have a few blurred pixels.

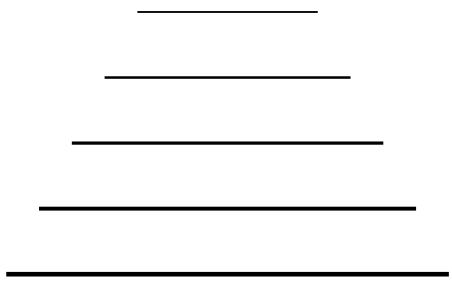


Figure 4.14: Camera tilting effect.

We also remind the reader that the following results are to be considered qualitatively, given the high number of variables involved in addition to the baseline: camera and projector angles, working distance, relative position of the scanner with respect to the object, number of points projected, and more.

First session

The first tests session was carried out by setting the working distance to 40 cm, the same as in the previous sections of the chapter.

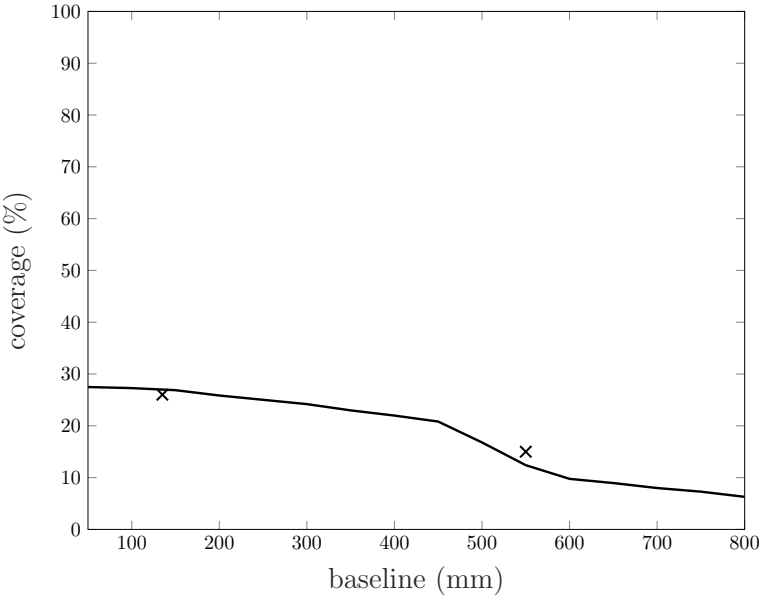
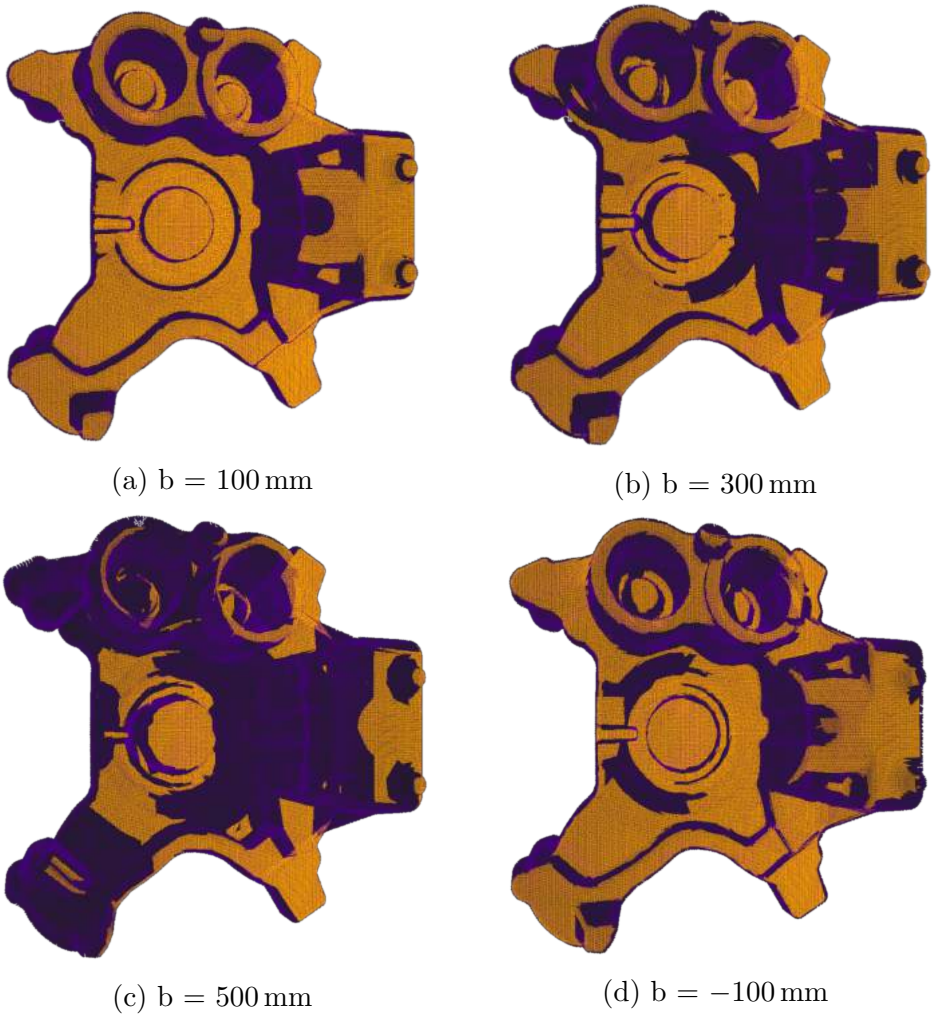


Figure 4.15: Reconstruction coverage, over different baseline values, at a working distance of 40 cm. The two crosses represent actual scanner data.

Figure 4.15 shows the coverage trend as the baseline increases from 50 to 800 mm. Remind that the coverage is generally low because the sampling, unlike reconstruction, takes into account the whole object and not only the part facing the scanner. The score values have not been added to the graph because they always stand at high values, i.e. between 99.5% and 99.8%. We immediately notice that, increasing the baseline, there is a marked reduction in coverage. This is natural since the scanner is very close to the scene and, as the camera moves away from the projector, most of the object remains occluded. Furthermore, the angle of view is greatly reduced and, according to Snell's law, the power of the reflected rays becomes weaker and weaker to the point that they are no longer caught by the camera. We can also see two crosses which represent the coverage values obtained by comparing the real scanners reconstructions with the sampled model. Remind that Zivid and Photoneo scanners have a baseline of 135 and 550 mm correspondingly. It can be seen that these two values belong approximately to the trend line obtained using the simulator. Photoneo has a lower coverage than Zivid because it is suitable for longer working distances (refer to specification tables 4.2-4.4). Always bear in mind that these results are qualitative. It was not possible to carry out measurements with a great precision and, therefore, data

coming from scanners must be taken with a grain of salt.



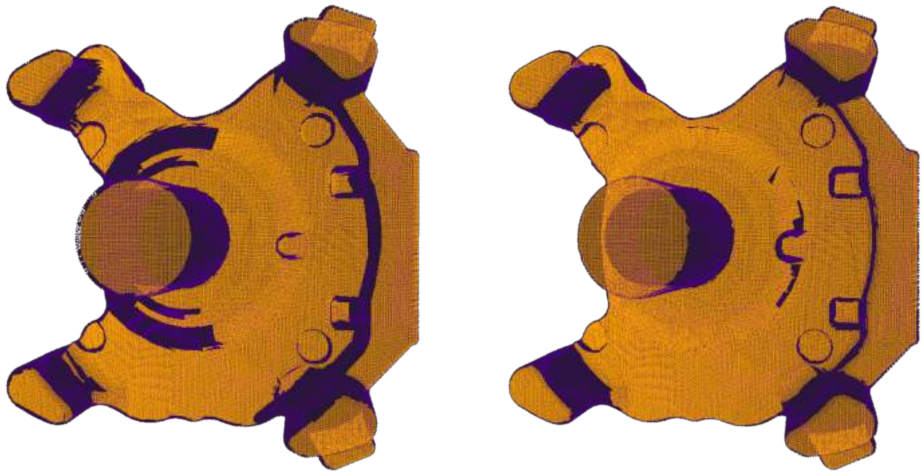
(a) b = 100 mm (b) b = 300 mm

(c) b = 500 mm (d) b = −100 mm

Figure 4.16: Reconstruction results at different baselines, at a working distance of 40 cm.

Figure 4.16 shows four special cases highlighting how shadows and occlusions affect the final result. Remembering the object shapes, reported in Figure 4.2, we notice how at a baseline of 500 mm the most protruding parts cause a large shadow zone which ruins the reconstruction. We could have guessed it even just looking at the previous plot, but it is not always good to trust the pure numerical value. A clear example is provided in Figure 4.16d, where the baseline is set to −100 mm, i.e. camera is located 100 mm on the left of the projector. The corresponding coverage rate is 27.1%, practically identical to the one obtained with a baseline

of 100 mm. This would lead us to think that the two configurations are equivalent but, finely observing the corresponding figures, we note that some characteristics of the object are correctly recognized only in the case of a positive baseline. Other times, however, the numerical results agree with the graphical ones. This is the case shown in Figure 4.17, which represents the superimpositions obtained by reconstructing the back of the object. There, we achieved 23.61% coverage with a 100 mm baseline and 25.98% coverage with a −100 mm baseline. In this case, both looking at rates or reconstructions, we deduce that it is better to use a configuration in which the camera is placed on the left of the projector.



(a) b = 100 mm (b) b = −100 mm

Figure 4.17: Reconstruction results at different baselines, at a working distance of 40 cm (backside of the object).

By superimposing the simulated point cloud on the real one, we are sufficiently able to evaluate the goodness of a particular configuration. Moreover, with a minor code update, it would be possible to select the indexes of all the unrecognised points to highlight the shadow or occluded areas. This may be useful to observe whether or not unrecognised areas are important characteristics of the tested object.

Second session

The second tests session was carried out by setting the working distance to 60 cm. The reconstruction algorithm was run several times, changing the baseline, and coverage rates were collected in the graph of

Figure 4.18. We can immediately notice how the trend is more constant compared to the previous scenario. Actually, values remain nearly always above 20%. The main reason is that, as the camera moves away from the scene, its view angle increases and, as a result, the final reconstruction is less prone to shadows and occlusions.

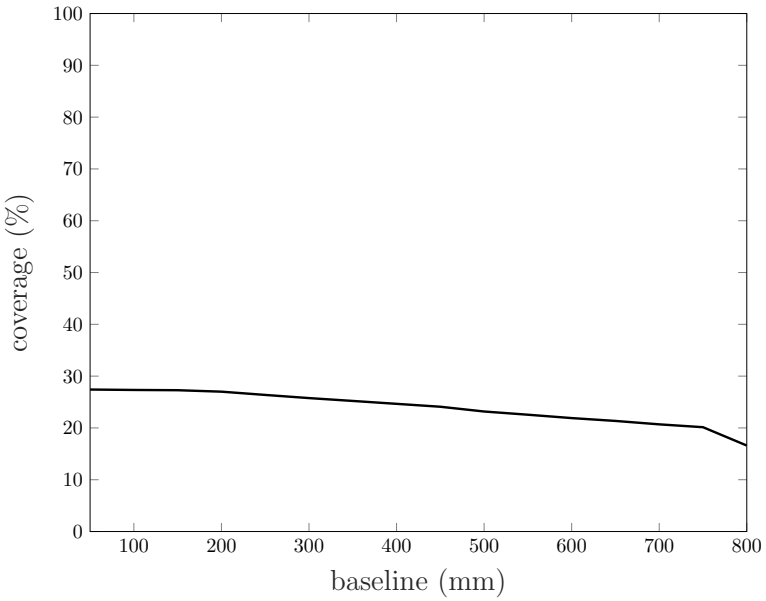
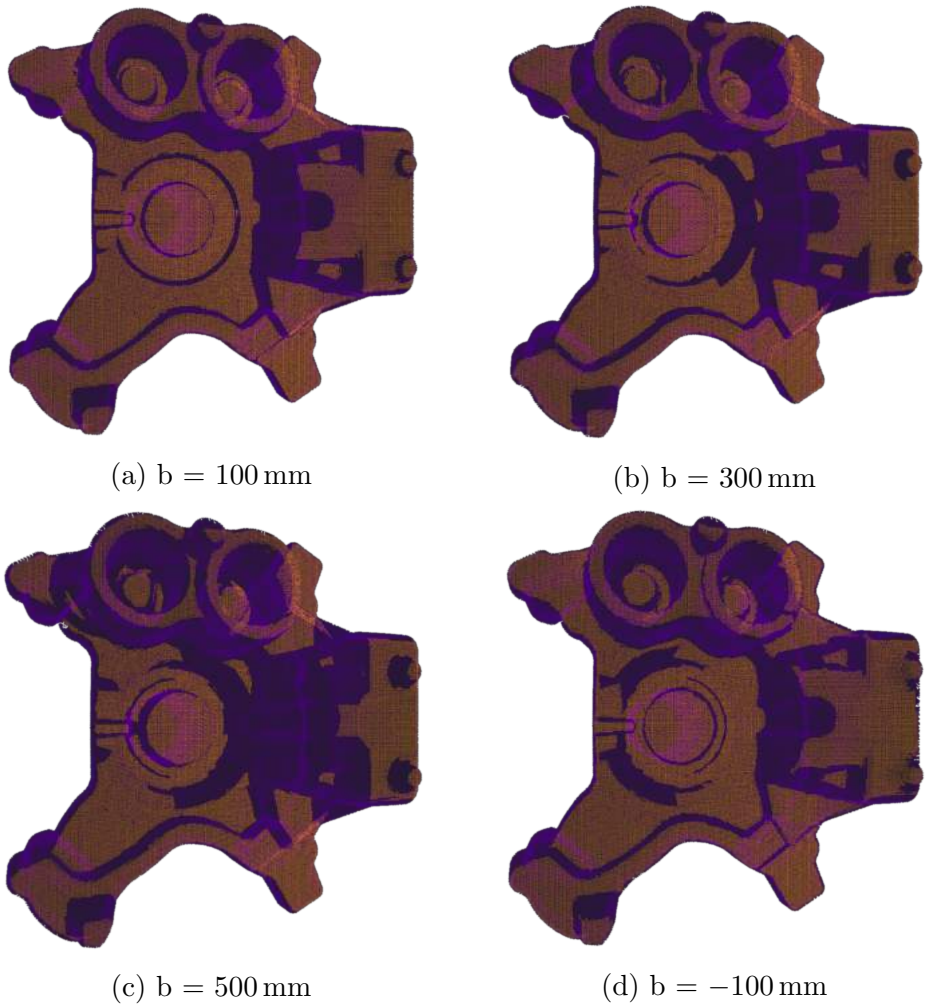


Figure 4.18: Reconstruction coverage, over different baseline values, at a working distance of 60 cm.

We can see especially from Figure 4.19c that, using a greater working distance, the reconstruction covers more regions of the object. Even if the result is better with a lower baseline, we can still recognize the most important features. A further comment concerns that, from the several figures shown so far, as the working distance increases, the points resolution significantly decreases. To confirm this, we specify that, using the same object pose and baseline, at a working distance of 40 cm we reconstruct about 40 thousand points while, at 60 cm, only 10 thousand. Actually, the points of the projected pattern are less dense as the distance between scanner and scene increases. So, in order to compensate for this loss of resolution, we are forced to increase the number of projected points, i.e. to reduce the gap between pattern lines.



(a) b = 100 mm (b) b = 300 mm

(c) b = 500 mm (d) b = −100 mm

Figure 4.19: Reconstruction results at different baselines, at a working distance of 60 cm.

Third session

The last tests session was carried out by setting the working distance to 100 cm. Looking at the plot in Figure 4.20, we can observe that now the coverage remains almost constant also by increasing the baseline. On the other hand, the points resolution has drastically decreased, i.e. points are less dense compared to previous cases. Indeed, for this reason, we decided not to report reconstruction examples like in the two previous scenarios. A longer working distance may be necessary for technical/mechanical reasons or it may be useful if the object is particularly large. Therefore, remind that, in these cases, it is preferable to increase the number of

pattern points to balance the loss of resolution.

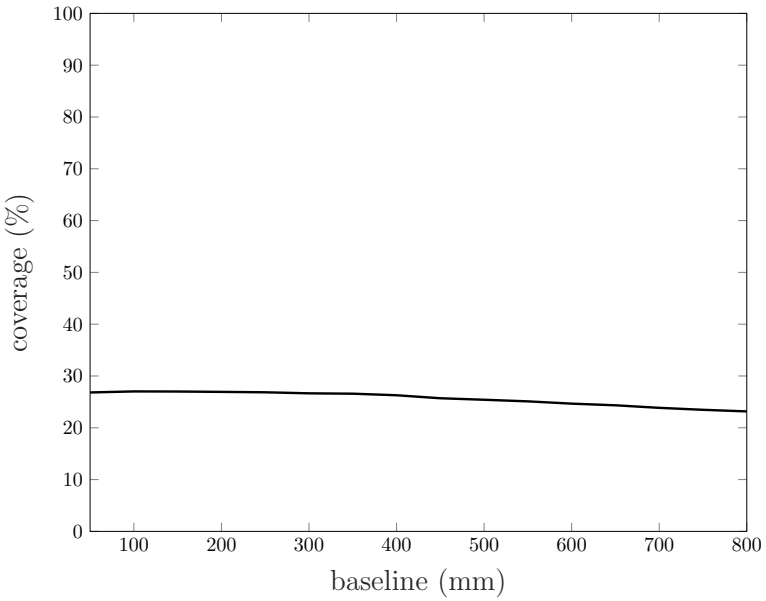


Figure 4.20: Reconstruction coverage, over different baseline values, at a working distance of 100 cm.

After observing results, however, we are not able to decide mathematically the most suitable baseline for a particular working distance. There are two main reasons. The former is that we do not own a mathematical index of "goodness" to automatically determine the best option. This index, for example, could represent the percentage of interesting features of the object that have been correctly reconstructed. For this purpose, score and coverage indices, are not sufficient. The other reason is that, as we said before, there are many variables involved in choosing the best configuration and not all of them have been considered in the proposed simulator. Therefore, we have obtained a powerful tool to evaluate in real time the result of a 3D reconstruction and then decide on the best configuration to solve a particular problem.

Chapter 5

Conclusions

Reconstruction through pattern recognition, or structured light, has become one of the preferred techniques for recognizing 3D objects in the industrial field. Indeed, thanks to the presence of a projector and a camera that recognizes the pattern emitted, this technology allows its use also in dark environments and it is not sensible to brightness changes. In this work we presented a completely new simulator capable of emulating a generic 3D structured light scanner. We emphasized the key steps in the design, highlighting problems and their possible solution, as well as the importance of developing code for GPUs.

After a wide discussion of the theory behind pattern recognition reconstruction, and a detailed presentation of the algorithms' implementation, we devoted a long test session to validate and analyse the behaviour of our simulator. We used two 3D systems to acquire scans, both based on structured light technique: PhoXi Photoneo and Zivid. Thanks to them, we were able to verify that the results obtained by the simulator were plausible. Therefore, we confirmed that model not only fits the two scanners studied, but is also able to reproduce any other based on the same technology. However, the most interesting study is certainly the one according to the baseline. We found that, increasing the baseline, coverage decreases but it still depends a lot on the working distance chosen.

5.1 Future developments

The proposed simulator can be a valid tool to provide customers a quick idea of the most suitable baseline for a particular scenario, and consequently recommend a scanner rather than another. In addition,

it could help to study occlusion and shadow phenomena in depth for the possible design of new 3D scanners. Nevertheless, there are several upgrades that can be made in order to expand its functionalities. First and foremost, a significant improvement in results could be achieved by using ray tracing principle instead of ray casting one. By using the former, as already mentioned in Chapter 2, we would trace the path of the rays even after the first collision. This would allow us to evaluate reflection and refraction phenomena on surfaces, making the model more realistic. We would certainly get a higher level of detail, but we should ensure that, by increasing computational complexity, the process does not take too long. Besides, the simulator offers a good level of interactivity to the user, but most of the variables (baseline, working distance, type of projected pattern) must be changed within the code, which is then compiled. An important improvement would be to make these variables accessible externally, so that the user can select the desired specifications before launching the reconstruction. A further step would be to incorporate the optics into the simulator. In this way, the user will be able to choose the type and specifications of the lenses, and to emulate some distortion effects, like barrel or pincushion ones, directly on the display.

Actually, the work presented is part of a larger project launched by Euclid Labs. In the field of artificial intelligence, deep learning certainly plays a fundamental role. One of the main problems regards the difficulty in finding the data needed to train these systems. With this aim in mind, Euclid Labs would like to design and build a 2D/3D image simulator in order to generate test datasets. To do this, it is necessary to emulate as many 3D scanner types as possible, not only structured light ones, and 2D cameras. Given the importance of generating images in a very short time, it will be necessary to fully exploit the power provided by the GPUs, and then design the code to avoid exceeding the TDR, as seen in Section 3.4. This project is very ambitious, but it certainly represents a fundamental step to evolve in the robot programming world.

Bibliography

[1] Joaquim Salvi, Sergio Fernandez, Tomislav Pribanic, and Xavier Llado. A state of the art in structured light patterns for surface profilometry. *Pattern Recognition*, 43(8):2666 – 2680, 2010.

[2] Giovanna Sansoni, Marco Trebeschi, and Franco Docchio. State-of-the-art and applications of 3d imaging sensors in industry, cultural heritage, medicine, and criminal investigation. *Sensors*, 9(1):568–601, 2009.

[3] Yoshiaki Shirai. Recognition of polyhedrons with a range finder. *Pattern Recognition*, 4(3):243–250, 1972.

[4] Gerald J Agin and Thomas O Binford. Computer description of curved objects. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 629–640. Morgan Kaufmann Publishers Inc., 1973.

[5] Robin J Popplestone, Christopher M Brown, A Patricia Ambler, and G Crawford. Forming models of plane-and-cylinder faceled bodies from light stripes. In *IJCAI*, pages 664–668, 1975.

[6] M Asada, H Ichikawa, and S Tsuji. Determining of surface properties by projecting a stripe pattern. In *Proc. Int. Conf. on Pattern Recognition*, pages 1162–1164, 1986.

[7] George Stockman and Gongzhu Hu. Sensing 3-d surface patches using a projected grid. In *Computer Vision and Pattern Recognition*, pages 602–607, 1986.

[8] Joan Batlle, E Mouaddib, and Joaquim Salvi. Recent progress in coded structured light as a technique to solve the correspondence problem: a survey. *Pattern recognition*, 31(7):963–982, 1998.

[9] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45. ACM, 1968.

[10] Turner Whitted. An improved illumination model for shaded display. In *ACM Siggraph 2005 Courses*, page 4. ACM, 2005.

[11] Tomas Möller and Ben Trumbore. Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*, page 7. ACM, 2005.

[12] Carli Ruggero. *Robotics, Vision and Control: lecture notes*. University of Padova, 2017.

[13] Justin Stenning. *Direct3D Rendering Cookbook*. Packt Publishing Ltd, 2014.

[14] Zivid Labs. `https://www.zividlabs.com`. [Online; accessed 8 June 2018].

[15] Photoneo. Phoxi 3d scanner technology overview. `http://www.photoneo.com/phoxi-3d-scanner/`. [Online; accessed 11 June 2018].

[16] Frank Luna. *Introduction to 3D Game Programming with DirectX 11*. Mercury Learning & Information, USA, 2012.

[17] Allen Sherrod. *Beginning DirectX 11 game programming*. Cengage Learning, 2011.

[18] Fletcher Dunn and Ian Parberry. *3D math primer for graphics and game development*. CRC Press, 2015.

[19] Pooya Eimandar. *DirectX 11.1 Game Programming*. Packt Publishing Ltd, 2013.

[20] Alexander Hornberg. *Handbook of machine vision*. John Wiley & Sons, 2007.

[21] Jason Zink, Matt Pettineo, and Jack Hoxley. *Practical Rendering and Computation with Direct3D 11*. A. K. Peters, Ltd., Natick, MA, USA, 1st edition, 2011.

[22] P. Corke. *Robotics, Vision and Control: Fundamental Algorithms In MATLAB® Second, Completely Revised, Extended And Updated Edition*. Springer Tracts in Advanced Robotics. Springer International Publishing, 2017.

[23] Cenedese Angelo. *Control System Design: lecture notes*. University of Padova, 2018.

[24] Photoneo. `http://www.photoneo.com`. [Online; accessed 11 June 2018].

[25] Photoneo. 3d scanning knowledge base photoneo wiki. `http://wiki.photoneo.com/`. [Online; accessed 11 June 2018].