Master Erasmus Mundus in
Color in Informatics and Media Technology (CIMET)

A survey of techniques for depth extraction in films

Master Thesis Report

Presented by

Victor Medina

and defended at

Gjøvik University College

Academic Supervisor(s): Prof. Simon McCallum

Jury Committee: Prof. Alain Tremeau
Prof. Pertti Silfsten

A survey of techniques for depth extraction in films

Victor Medina

2012/07/15

Abstract

The proliferation of 3D-enabled displays in the last decade is increasing the demand for 3D content. There are several approaches to produce 3D content: Live camera capture, Computer-generated content and 2D-to-3D conversion. Creating 3D content directly from cameras requires the use of expensive equipment and facilities, so many content providers prefer to create their material in 2D, and then convert them into 3D. Similarly, these conversion techniques can also be used to convert already existing 2D material (movies, cartoons, etc.) into 3D, so they can take advantage of the new display technologies. The conversion of films from 2D to 3D requires three stages: Assigning depth to each picture element (pixel), generating a stereo pair of images, and filling in missing information. One of the most time consuming parts of this process is assigning a depth value for each pixel of each image, as this process is usually done manually.

Two of the open research questions related to converting 2D content are the integration of multiple depth cues and the automation of the conversion process. In this project we present an automatic process that can successfully estimate the depth range of a film scene, as well as the number of depth planes and their relative positions in relation to that range.

In addition, we propose a new method to evaluate the quality of the results by comparing them with depth map data generated with the Microsoft Kinect's depth sensor.

Keywords: 3D conversion, 3D films, Depth estimation, Point cloud, Kinect

Preface

First of all, I would like to thank the Erasmus Mundus Program, as well as the four partner universities of the CIMET Consortium: Jean-Monnet University (France), University of Granada (Spain), Gjøvik University College (Norway) and University of Eastern Finland (Finland), and their corresponding coordinators: Alain Tremeau, Javier Hernandez-Andrés, Jon Yngve Hardeberg and Jussi Parkkinen, for their constant efforts in organizing and improving the CIMET Master program. I also want to thank the administrative coordinators Helene Goodsir and Hilde Bakke, for helping us with many issues during this time, administrative and otherwise, and all the professors and lecturers that have taught us throughout this program.

I would like to thank my supervisor, Professor Simon McCallum (PhD) at Gjøvik University College, for creating this project and supporting me all along. He was able to come up with new suggestions when some of the original ideas did not work out, and provide a lot of helpful feedback during the course of this project. Thanks to all this help I improved a lot my knowledge of research and writing techniques, but also my knowledge of the English language. I would also like to thank Professor Jayson Mackie (Gjøvik University College) for dedicating a lot of his time helping me during the initial software development stages.

I also would like to thank all the people who have studied with me during these two years. Together we have gone through good and bad moments and, even though I have coincide with some of them only for a short time, I feel like we have all been together all along, and hopefully will stay in touch for much longer. A very special mention goes to all the friends I have made in these two years, some of whom have become very close ones; I have had great times with them and this has helped me cope with the stress of the the studies.

And last but not least, I would like to thank my family for always supporting me in my decision of studying and working abroad, despite the big effort that I know it means for them.

Contents

List of Figures

List of Tables

1 Introduction

Display and graphic processing technologies have undergone many changes in recent years. Average graphic processors are now capable of performing operations that some years back could only be done by super computers. These advances have made possible a variety of new graphical applications, one of the most important being 3D visualization.

3D visualization has been utilized for a long time in many areas such as simulation or medicine; however, its high cost did not allow it to extend to more general applications. Presently, much higher quality graphics can be generated at a low cost, which is the reason behind the increasing popularity of 3D content. Indeed, the use of 3D visualization today is no longer restricted to applications for scientific visualization and research, and it is now accesible to the average user, especially for entertainment purposes. The game, film and, in general, the entertainment industries, are arguably the main areas where the use of three dimensional content is proliferating, favored by three major events that took place in the year 2009. The release in January of the "Nvidia 3D Vision Gaming Kit"[1] and the release shortly thereafter, by the Samsung corporation, of the first 3D monitor compatible with Nvidia 3D Vision[2], followed by the release in December of the movie *Avatar*, directed by James Cameron, had a key role in promoting the current capabilities of 3D technology to the average public, which generated an increasing demand for such technology[3]. The film industry soon reacted to this new market demand[4] by increasing the production of 3D content which, in turn, given the novelty of this technology, required a lot of investments in research and development.

Until recently, display technologies worked mainly with 2D images, leaving it up to the viewer's brain to infer the third dimension. We have grown accostumed to watching this type of image, and during that time our brain has adapted in a way that it is capable of deducing the third dimension from other cues, so that it looks natural to us even though some information is missing. There is an open debate about whether the use of the third dimension in films really improves the viewing experience, or it is just a gimmick for movie theaters to increase ticket sales and stay competitive with the other multiple entertainment options available[5]. According to Matthew Jeppsen from FreshDV.com[6] the answer to this question is "a question of content". 3D should not be used in just any film and, in any case, those films which do use it should ensure that it is done in a proper manner, so that the viewing experience is indeed improved and not damaged.

There are several approaches to producing 3D content: Live camera capture, Computer-generated (CG) content and 2D-to-3D conversion – a process also referred to as *Dimensionalization*[7]. For the first approach, which consists of creating 3D content directly from cameras, production companies must make an important investment, given that special cameras and sets are required to film in three dimensions, so only big-budget productions can afford to be filmed in 3D. As a cheaper alternative, many content providers use 2D to 3D conversion techniques to create 3D versions of their 2D material, which can also be used to convert already existing 2D material (movies, cartoons, etc.) into 3D, so they can take advantage of the new displays.

The process that is currently used requires a great deal of post-production, such as manually selecting the objects' outlines (known as rotoscoping) and hand inpainting [1] occluded areas[9](see Figure 1); and yet, the quality of the results is far below that of natively produced 3D content. Two of the open research questions related to dimensionalizing 2D content are the integration of multiple depth cues and the automation of the conversion process. In this project, we propose to develop and analyse different techniques that can contribute to the automatization of the 2D-to-3D conversion process, thus minimizing the amount of human interaction.

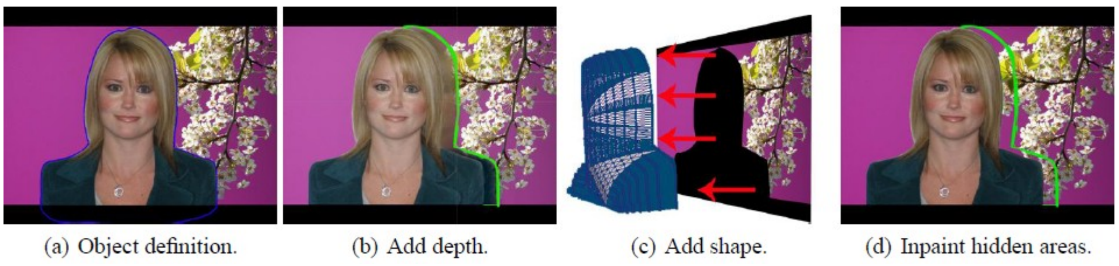

(a) Object definition. (b) Add depth. (c) Add shape. (d) Inpaint hidden areas.

Figure 1: The Dimensionalization process[7]

There are many different methods to extract depth information from a video. A first approach consists of using information from visual cues such as the horizon line, vanishing points, aerial perspective or texture gradients, on individual video frames alone, to infer the depth of a scene. However, when videos are considered as a whole set of images which are related with each other, rather than as independent frames, it is possible to use the temporal dimension to provide additional cues such as image displacement (also known as parallax) or object occlusion. These temporal cues can be used to mimic the stereoscopic properties of the human visual system (HVS), thus achieving much more accurate results. In the specific case of films, the analysis of certain cinematographic techniques such as motion blur, depth of field or element arrangement, can help to further estimate the depth.

Some of the existing solutions to the problem of extracting depth from films are limited to only one of the afore mentioned categories, whereas other use a hybrid approach where several types of cues are integrated to obtain the final depth. For example, Battiato et al.[10] use information only from the vanishing lines to generate the corresponding depth maps for static images; Alvarez et al.[11] exploit more geometrical cues apart from the vanishing lines; other solutions provided as well, among others, by Battiato et al.[12], apply a previous segmentation based on the color properties of the image before calculating the vanishing lines; Dalmia and Trivedi[13] take into account the temporal dimension found in videos, extracting depth information from the epipolar geometry of the points. More advanced solutions integrate several of the previous approaches into one, obtaining more reliable results; this is the case, for example, of the solution presented by Benini et al. [14], which integrates information from color histogram variance, geometrical cues, motion vector estimation and face tracking.

In this work we propose a method to automaticaly estimate the three dimensional structure

[1]Inpainting: *the technique of modifying an image in an undetectable form*[8]

of a scene from a monocular video, based on the concept of structure-from-motion (SfM), as explained later in Section 2.3. The proposed process can successfully estimate the depth range of a scene, as well as the number of depth planes and their relative positions in relation to that range. First, a sparse set of 3D points (point cloud) is generated from SfM information. Then, we obtain a denser point cloud, and apply statistical clustering methods to estimate the location of the main depth planes. Finally, RANSAC is used to perform an additional mapping process that assigns each point to its corresponding plane. [2]

Since there is no ground truth depth information associated with traditional films, which we could use to analyze the quality of our results, we propose a new quality evaluation method, using Microsoft's kinect to generate test cases with two purposes: to check the validity of the dense point clouds, and to measure the deviation error in the estimated depth planes. Our results show that we can estimate depth planes with a small deviation error of up to a 0.67%.

Chapter 2 presents a discussion on the current and future 3D technology, some of its advantages and disadvantages, as well as some techniques that are typically used in 2D films to achieve the sensation of depth. It also introduces the reader to the problem of 3D reconstruction and the most common approaches to this problem, and describes some of the most important aspects of 3D reconstruction to take into account when converting films to 3D. Chapter 3 describes how the project has been implemented as well as the main libraries used to manage the data. This is followed by a detailed description of each of the stages of the proposed method (Chapter 4), and an analysis of the results (Chapter 5). The final chapters will be dedicated to giving a global overview, as well as proposing some future directions in this field (Chapters 6 and 7, respectively).

[2]If needed, see Section 2.1 for a definition of some important terms in this context that may not be clear to the reader.

2 Background

2.1 Terminology

It is possible that some readers are not familiar with some of the terms used throughout this report. Therefore, to avoid confussion and make sure that everyone understands all the concepts, we describe in this section some of the most common terms used in this project. If the reader is already familiar with these terms, they can skip this section and continue reading from Section 2.2.

2.1.1 Point Cloud

A point cloud is a set of vertices, where each vertex represents the position in 3D space (coordinates x, y and z) of a point in the scene. Additionally, a vertex can also store information about other properties such as color, brightness or normal direction. Point clouds are typically created by some type of 3D imaging system (e.g., 3D scanners), or exported from a CAD or 3D application, as an efficient way to store 3D models. We will say that a point cloud is sparse or dense, depending on the number of vertices it contains. In this project, we talk about sparse point clouds when referring to those point clouds containing only vertices for the main features in an image, whereas dense point clouds are those that contain a vertex per each pixel in the image – and, therefore, a much higher number of vertices.

2.1.2 Depth map

Given the three-dimensional structure of point clouds, it is difficult to display them in 2D space. Therefore, a common alternative method to represent the depth of the elements in an image in 2D space is through the use of depth maps. These are grayscale images where the depth of each pixel is stored as a gray value, with darker values typically representing closer values, although it can also be used the other way around depending on the model used. This model – a standard in computer graphics – is used to represent 3D graphic features in 2D space and, although it is most typically used for depth maps, it is also used in other types of maps such as bump maps, displacement maps and disparity maps. Depth maps are a simple and convenient method to store and analyze the 3D structure of a scene; since the gray values represent the depth of each pixel, the dynamic range of these maps is, in turn, a measure of the depth range of a frame, which allows to study the depth of a scene using simpler image processing techniques. Moreover, there is a direct relationship between depth maps and point clouds (see Section 2.1.1) so that, in most cases, one can be obtained from the other using a straighforward conversion.

2.1.3 Depth planes

Given a 3D scene, the objects in the scene can be found at different depths; however, given the structure of a film scene described in Section 2.4.1, there will always be some specific positions in the depth axis that contain a much higher concentration of objects than others. In this sense, we call depth planes to the different depth values at which most of the relevant objects in the scene are located. Given a scene, the number of depth planes is not unique, so it has to be prefixed in advance, and the depth of the scene analyzed according to the pre-established number of planes.

Far and Near depth planes

Regarding the depth of a scene, the far plane is the depth plane that is located the farthest from the camera plane. Similarly, the near plane is the closest one to the camera. These planes are important because the determine the total depth range of a scene.

2.1.4 Depth Range

The depth range of a scene is determined by the distance between the far and near planes, and it is used as a normalization reference to compute the relative distances between each depth plane in the scene.

2.2 3D display and capture technology

When an object is viewed from two different perspectives, its position appears to be different for the viewer. This is known as the parallax effect. In many animals, including humans, the eyes are located in different positions on the head, so that different views with overlapping visual fields are perceived by both eyes simultaneously. The parallax between these two views is used by the brain to estimate distances and perceive depth, a process known as stereopsis, which is the main concept behind the working mechanism of current 3D displays[15]. Along with the parallax, the Human Visual System (HVS) uses many other cues to estimate the depth of objects in a 3D scene, which can be divided into two categories: physiological and psychological.[16]

The four main physiological mechanisms used by the HVS to provide the neccesary cues to perceive depth are the binocular parallax (also known as binocular disparity), monocular motion parallax – even when we close one of our eyes, we can still perceive depth from similar consecutive views of the scene by moving our head, which is known as *look around*[17] – accomodation and convergence. When objects are located within a certain distance from an observer, these physiological cues are used to estimate their depth; however, when the observer is located sufficiently far from the perceived objects, the visual system is not capable of calculating exact distances using the aforementioned physiological cues, so psychological depth cues must be used instead. Some examples of psychological cues are the texture and color gradient, shadows, relative size or aerial perspective. Furthermore, a series of studies presented by Sỳkora et al.[18] show that humans can accurately tell whether parts of an object are in front of another, and viceversa, but fail to specify absolute depths. This is the reason why some methods for creating the illusion of depth in 2D – such as the one used in traditional animation consisting of stacking layers on top of each other, or the matte painting technique that is widely used in films – work in most of the cases (in other cases, the camera motion or the excesive bad quality of artificially created elements can reveal this “fake” depth). Nonetheless, if all three physiological cues mentioned above are not present, it is not possible to perceive true depth and, therefore, true 3D.

2.2.1 3D technology overview

If we look at the existing technologies for generating three dimensional imagery, we can distinguish three main categories: stereoscopy, holography and volumetry.

Stereoscopy is currently the most popular technique used for visualizing films in 3D. It exploits the afore-mentioned process of stereopsis by projecting slightly offset frames for each eye consecutively, and incorportaing some mechanism to allow each eye to view only the frame destined to it. Although there are many different variations on how this mechanism is implemented[16], the most popular one for viewing films is the one known as stereo-pair display. Stereo-pair displays require the use of some viewing device, typically a pair or googles or glasses, which can be active or passive, depending on whether the image synchronization mechanism is included in the viewing device itself or the display, respectively. In turn, depending on the method use to discriminate the images for each eye, we can find several types of stereoscopic devices. The most commonly used for films are those that use polarization – the images for each eye are sent with a different polarization, typically circular or linear – or color filters –usually red/cyan pairs,– also known as anaglyphs.

Health Risks of Stereoscopy

There are several issues inherent to stereoscopy that prevent an even faster popularization of this technology. Since the images are projected onto a screen, there is always a disparity between the depth interpretation generated in our brain of the visual content, and the real depth of the projected image - which is flat[19]. Indeed, when we look at the sterescopic images presented by the 3D display, the eyes converge to a point at the apparent distance of the 3-D image -this is known as eye vergence,- which is the correct angle for the real objects, but they accomodate to focus the display plane on the retina. This results in a visual conflict between the depth information from convergence and accomodation that sometimes causes eyestrain, which can be more or less severe depending on the sensitivity of the viewer, and the strength and duration of the discrepancy (see Figure 2). Additionally, watching stereoscopic 3D is reported to cause other symptons such as headaches or nausea, effects which are also attributed to the mentioned visual conflict[20, 21, 22].

This conflict cannot be resolved by technical means, so the only solution is to either limit scene depth to very short sequences, or to artificially reduce the depth of the scene so that it falls within the so-called comfort depth range, at approximately 20/30% of the distance between the viewer and the display, where the human eye can tollerate some mismatch. In general, it is recommended to watch 3D displays from a distance at least three times the display's height[20].

Most modern productions try to minimize these risks by following several rules of thumb to obtain compelling results. The most important rule is to avoid extreme depth ranges, so that the depth budget mimics the depth of the original scene but it is scaled down to reduce discomfort as well as the adverse effects of extreme compositions; the second most important rule consists of maintaining depth continuity, so that consecutive frames do not contain transitions from very shallow shots to very deep shots, or viceversa[23].

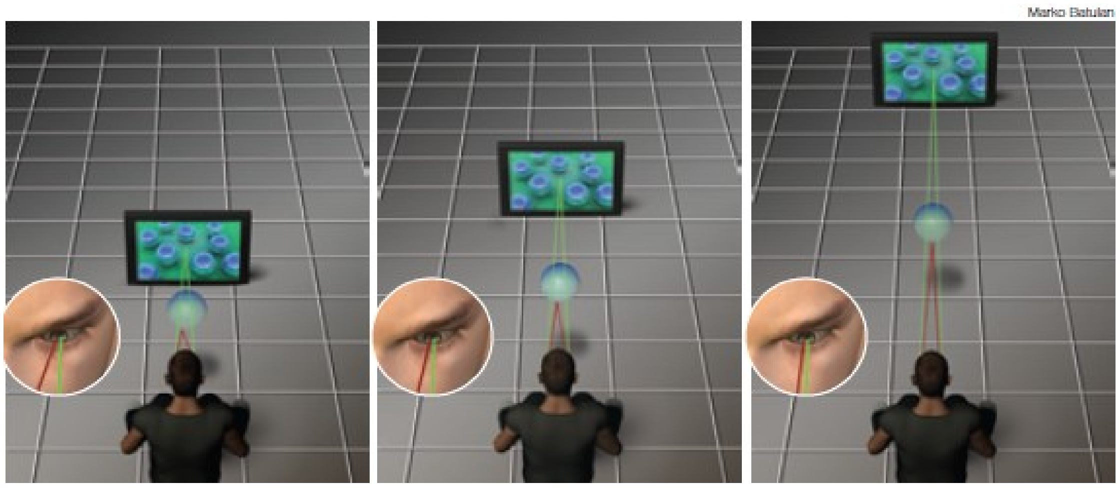


Figure 2: The closer the object, the more the eyes must converge, and the larger the visual disparity between the 3-D object and the screen. Eyestrain increases with the degree of visual disparity and the exposure time, so headaches are most likely to occur after watching intense 3-D for long period on a close-up screen.[20]

Alternatives to Stereoscopy

Holography and Volumetry are the most advanced alternatives to stereoscopy for 3D visualization, although they are still far from reaching the commercial status of stereoscopy. However, given all the issues mentioned above regarding stereoscopy, the amount of research on these alternative display technologies is increasing and, specially holography, presents itself as a serious alternative to substitute stereoscopy in the future[24, 25, 26].

We mentioned earlier that, to generate true three dimensional images, a given technology must be able to provide the four physiological cues needed by the human visual system to perceive depth. Due to the explained conflict between vergence and accomodation, stereoscopic displays fail to provide a coherent depth information, and therefore they do not show true 3D, but a virtual approximation of the natural viewing conditions[27]. Holograpic and volumetric displays, on the other hand, are capable of producing true three dimensional images, without the health risks explained above.

There are many solutions that convert films from 2D to stereoscopic 3D. Some of them use anaglyphs and some others use standard full-color stereo pair technology, typically by duplicating the frames and creating an artificial offset generating depth at the location estimated by some algorithm. However, if we want to obtain the true depth of the films we must reconstruct its entire 3D structure, and more exhaustive processing is required. In this project we are trying to contribute to this type of reconstruction, and we will describe the process in the following chapters.

2.2.2 Full 3D vs. stereoscopic 3D

One of the first things to consider before designing a new system to dimensionalize films (or any other type of video) is the target visualization model, since this model will determine the properties that the final film must have. As explained earlier, the most common visualization models for 3D films are currently based on the stereoscopic model, so many of the current approaches to extract depth from films use stereoscopic 3D as the target system. Of course, when stereo cameras are used to capture the video, the solution to this problem is relatively straighforwad, since the depth can be easily extracted from the stereo pairs; on the other hand, if monocular videos are used instead, models aiming to convert a film to 3D typically analyze the depth of the film using different criteria, and then use this information to generate “fake” stereo images from the monocular input.

These stereo films, as explained in Section 2.2.1, are not in “true” 3D but, rather, 2D video with additional depth information added to it – also known as 2.5D or pseudo-3D. We say that it is not “true” because the depth information attached to it does not correspond perfectly with the depth existing when the video was filmed; instead, the depth was estimated from the video content, so a certain error is most likely to exist. The problem that we find when converting films to 2.5D is that, although they are good enough for stereoscopic technologies, they would not work well with other technologies that use true 3D such as holographic or volumetric displays, because the result would look rather flat, given that those types of displays do not project the images onto a screen, but on the real three-dimensional space[16]. Therefore, as 3D visualization technologies evolve, it becomes more neccessary to find more robust methods to estimate and represent the 3D structure of a film.

2.3 3D reconstruction

The problem of recovering the 3D structure of a given environment has been around for a long time, being a task that is commonly required as part of the pipeline in remote sensing applications. Recently, however, due to the proliferation of digital cameras, common reconstruction techniques have also started to be used for video reconstruction purposes. The following sections give an overview of the 3D video reconstruction problem applied to films and the most common tasks and methods used in this area.

2.3.1 Simultaneous Localization and Mapping

Simultaneous localization and mapping (SLAM) is defined as the process by which a mobile robot can build a map of an environment and, at the same time, use this map to deduce its own location, without the need for any a priori knowledge of the environment[28]. In robotics, SLAM is a common technique used to locate a robot within an unknown environment. As the name implies, the process consists of two main tasks: localization and mapping. Mapping is the process of representing the environment from the information gathered from the sensor data, whereas localization is the process of finding the position of the robot within that environment. These two tasks are intrinsically dependent on one another since, to be able to estimate the pose of a robot within an environment, a representation (or map) of that environment must be created first but, the map can only be created by moving the robot through the environment – for which the robot must have a map. This way, a map is build incrementally as the robot advances through the environment.

SLAM is an old problem, considered to have originated back in the year 1986[28], that is often presented in probabilistic form. Although many different solutions have been provided to this problem over the years, the most common ones are considered to be EKF-SLAM and Fast-SLAM. EKF-SLAM represents the problem as a state-space model with additive Gaussian noise, whereas Fast-SLAM uses a different representation where the vehicle motion is considered as a set of samples from a non-gaussian distribution. Hugh Durrant-Whyte and Tim Bailey describe these two approaches in more detail[28], as well as some more advanced methods to solve the SLAM problem[29].

As mentioned earlier, SLAM was initially originated as a problem in the field of robotics. Nonetheless, the solutions to this problem did not consider the use of cameras until very recently, due to their high cost and the amount of time required to convert data coming from them into reliable maps; instead, other sensors such as laser range-finders and sonar were used[30]. Today, however, the improvements in camera technology and the low prices have made them a more attractive choice as SLAM sensor.

2.3.2 Structure from motion

The methods for depth extraction from video content can be divided into two categories, depending on the followed approach: monocular reconstruction and stereo reconstruction. Monocular reconstruction techniques, as mentioned earlier, use information from several static visual cues. In contrast, stereo reconstruction techniques typically use a pair of images (or videos) to infer depth using cues such as binocular parallax or object overlapping. Stereoscopic techniques tend to perform better than monocular ones; however, in cases where a stereoscopic source is not available, a hybrid approach can be used, consisting in using frames from the same video, sufficiently spaced out in time, as a substitute for stereoscopic frames. This last approach is known as Structure from Motion (SfM) – also referred to as Monocular SLAM, or MonoSLAM – and it is the most commonly used in computer vision, and also the approach that we will follow in this project.

The camera pose, movement and calibration parameters are essential factors in determining the method that must be used to reconstruct the 3D structure of the scene. The simplest case is that where the camera motion, pose and calibration parameters are known; in this category we can find, for instance, 3D scanners with a constrained motion. If the calibration parameters are unknown the setup is similar to, for example, that of the Proforma system[31], where a webcam (uncalibrated) is used to scan an object that rotates always over the same known axis. The most problematic case is that of a freely-moving uncalibrated camera, where neither the camera movement nor the calibration parameters are known. When converting already existing 2D films into 3D, we do not have access to the calibration parameters of the camera used when the film was made, so we can consider this problem as a problem of uncalibrated structure from motion(USfM).

2.3.3 Camera pose estimation

In situations where the calibration parameters of the camera are unknown, it is neccesary to automatically track the camera motion and to estimate the camera viewing parameters. Several algorithms exist to estimate such parameters and, depending on the application, we may choose or another.

Parallel Tracking and Mapping (PTAM) is a common algorithm used for camera pose estimation in Augmented Reality (AR) applications where a map of the environment is not available, using an USfM approach. To speed up the calculations, PTAM takes advantage of new multi-core processors to solve the SLAM problem, by splitting the localization and mapping tasks in parallel threads[32]. Separating the processes in threads removes the dependency between them, which also removes some important limitations of the original problem. Since the threads are now independent, the tracking process has more resources for itself, so it can do more extensive computation, thus yielding better results. Similarly, the mapping process is no longer dependent on tracking so it does not need to work with every consecutive frame, which allows it to reduce the number of processed frames, avoiding unnecesary redundancies such as unchanged frames, and dedicating the computation to only a reduced set of relevant keyframes. These optimizations allow to increase the quality of the results while maintaining a real-time performance. Additionally, the incremental mapping required by the original SLAM can be replaced by a batch method, such as bundle adjustment[33], which requires more computation but provides more accurate results[32].

A more recent optimization of PTAM is the algorithmm DTAM (Dense Tracking and Mapping). Instead of using features like PTAM, DTAM operates on every pixel of the images to generate dense maps; this, of course, requires a much higher amount of computation time and resources, but using parallelization and optimization techniques it keeps the execution time within real-time boundaries.

2.3.4 Keyframe selection

Trying to estimate the camera pose for each frame of a video requires an extremely high amount of time that is almost impossible to do in real-time. However, to display smooth motion, videos typically contain between 23 to 50 frames per second, most of which are very similar to each other – especially in sections with little or no motion. Therefore we can increase the performance of the reconstruction process greatly by selecting only those frames where relevant changes occurr (known as keyframes). Indeed, keyframe selection is a very critical part of the reconstruction process which affects, not only its running time, but also the quality of the final results. According to Sourimant[34], there are several methods for keyframe extraction; in general, keyframes are chosen to maximize the distance between the positions of the features in two consecutive keyframes, while maintaining the number of matches above a certain threshold. This method ensures that the camera motion and the number of matches between consecutive keyframes is enough to allow for a robust reconstruction.

2.4 Depth in Films

Before we continue describing in more detail the process that we have followed in this project to infer depth from a film, we must first describe how traditional films create and use depth, to understand its purpose and the role that it plays as an additional resource to tell a story. The following sections explain how the concept of depth was originally introduced in films, some different depth models that exist, and ways in which depth can be typically used to convey a specific message or feeling in a film.

2.4.1 Film structure

The history of films dates as far back as the 1890's, when the first films were created. At that time, films meant a step forward from theater plays, overcoming some of the limitations imposed, among other factors, by the restricting dimensions of the stage[35]; getting rid of the stage meant gaining freedom of movement, as well as much wider viewing angles and deeper depths[36, 37]. Nonetheless, scenes continued on being structured much in the same way that they used to in theater plays, given the dependency of cinematography on camera lenses. The focal length and focal point of the camera lens determine which parts of the image will be on focus and which parts will be out of focus; therefore, the arrangement of the elements in a film shot (also known as staging) generally follows a known layout, consisting of parallel planes defined by their distance to the camera. Although the number and space coverage of these planes may vary depending on the specific properties of the lenses, there are typically three main planes: a foreground, which is the closest plane to the camera, a middle ground, and a background that is usually much farther from the viewer[38]. How the elements and the actions are distributed along those planes will depend on the specific type of shot and the depth of composition used in it (see Section 2.4.3)[39]. As long as the planes in such layout are roughly parallel to each other and, in turn, parallel to the camera plane, estimating the 3D structure of the scenes is a relatively easy task. However, this is not always the case, and that is partly the reason why the dimensionalization process is still done manually.

The distribution of depth planes mentioned above was the foundation of traditional animation or animated cartoons (or just cartoons for short), where several transparent layers (overlays and underlays) are stacked on top of each other to create depth[40]; varying the content of each individual layer generates the impression of movement with respect to one another. One can easily see the parallelism between this setup and the ones used in theater, where there is normally a foreground consisting of the actors on the stage (upstage), a middle ground formed by the props placed on stage(middlestage), and one or several painted backgrounds (downstage)[38]. Although it has changed considerably over time, especially in big productions, the underlying model used by films today still follows the same setup.

A typical exception to the described standard depth layout can be found in the recent 3D release of Disney's classic animated film *Beauty and The Beast(2010)*. This scene, shown in Figure 3, has a very compelling composition, but the camera motion and the perspective used differs greatly from most scenes we are accustomed to, which is precisely one of the reasons why it was so acclaimed. When the engineering team at WaltDisney created the initial version of this shot in 3D, they quickly found after a few minutes of trial and error, that it was not neccesary to complicate the shot by adding unusual depth planes. Instead, they replaced the two characters, as well as the chandelier, by CG models, so that they could be effectively treated separately from the rest of the elements in the scene, which were merely given standard depth and volume for the final composition, using less complex techniques. This solution was found to provide "*an appealing stereoscopic rendition of the shot without conflicting depth cues and with minimal effort*"[23].

However, as mentioned, this process becomes more complicated than that in live-action films, given that many other resources such as depth of field, motion blur, etc (see Section 2.4.3), may affect the perception of depth and, therefore, complicate the process of depth estimation.

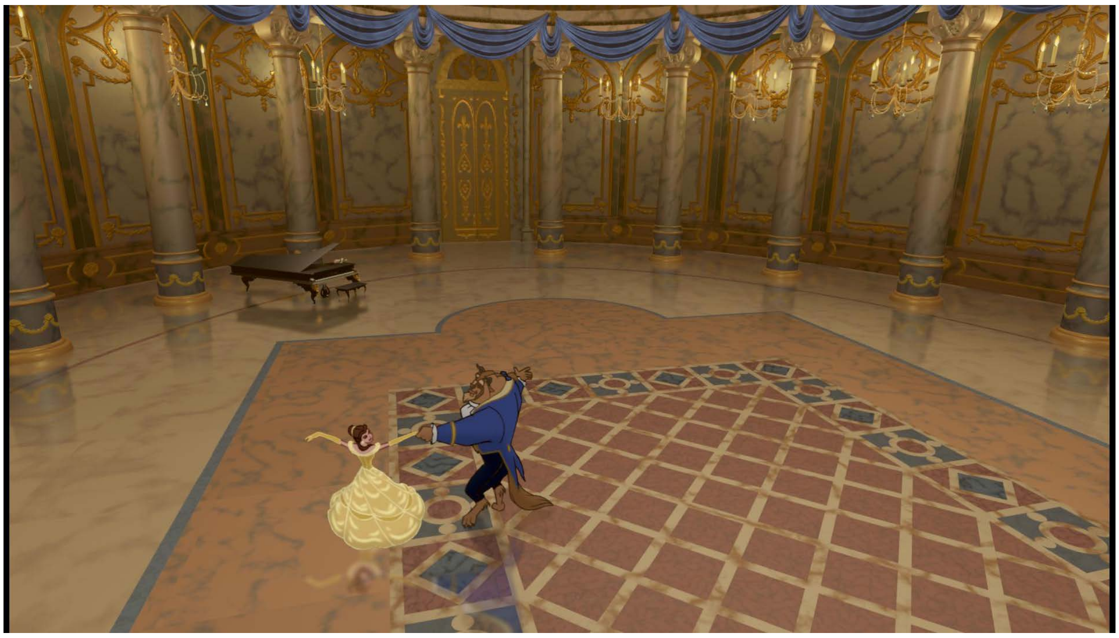


Figure 3: The acclaimed Ballroom sequence from *Beauty and the Beast(2010)* was one of the few scenes that was converted to 3D using a hybrid approach with CG models

2.4.2 The planar depth model

In films the z-axis, that is the line that runs from foreground to background, is what carries the illusion of depth. Following the direction of the z-axis the depth of a scene is divided into three main planes: the foreground, middleground and background[38], which we will hereafter refer to as the planar depth model. Filmmakers use these planes to exploit the frame depth by positioning different story elements in different planes[38]. More recent films are starting to use more complicated depth arrangements using a higher number of planes that the traditional three[41, 42], partly because filmmakers have to keep in mind a later conversion to 3D, but they still use the planar model. For this reason, in this project we have chosen to assume the planar depth model to estimate the depth of the scene, given its simplicity. We explained in previous sections that it is recommendable to avoid extreme depth ranges as well as unneccesary depth artifacts; in this sense, using the planar model ensures that we can control the amount of depth by restricting it to a maximum number of planes. Of course this assumption, just like any other, reduces the quality of the results since we are simplifying the depth of the objects to only one dimension. However, the assumption is more than sufficient for the purpose of this research, since we are only interested in finding the main depth levels in the scene, rather than the exact 3D shape of each and every element.

2.4.3 The role of depth in story-telling

Story telling in films is for the most part done by means of a script. The script establishes the main plot of the story as well as the events that take place in it. This is indeed true for most films but, given that cinema is supposed to improve the mechanisms for story telling available in theater, filmmakers normally do not limit themselves to merely tell stories through words, but also use the additional resources offered by the medium. There are certainly many different resources that a filmmaker can use to convey a story other than just a script, and the one we are most concern about in this project is the use of depth as a creative tool.

The use of depth for creative purposes is a common tool that has been used by filmmakers for a long time as a way to refine the initial composition of a shot. The most common way to do this is through the use of depth of field to draw the viewer's attention to certain areas in the image (see Figure 4), in such way that important areas remain in focus while less relevant ones are left more or less out of focus[43]. Indeed, the depth of field is an important factor in defining the relation and distances between the foreground, mid ground and background of a shot. A deep focus keeps all the different planes of the image in focus, giving the same relevance to all of them, and it is thought to provide a more realistic representation of space; on the other hand, a shallow focus keeps only one depth plane in focus, and it is used to direct the viewer's attention towards specific elements of a scene and to create proximity between the characters[39].

Nonetheless, although the use of depth of field is the most common form of depth modification employed by filmmakers, there are many other resources, such as the use of motion blur [1], special lenses or zooming techniques, that can affect how we perceive the arrangement of the elements in a film scene.

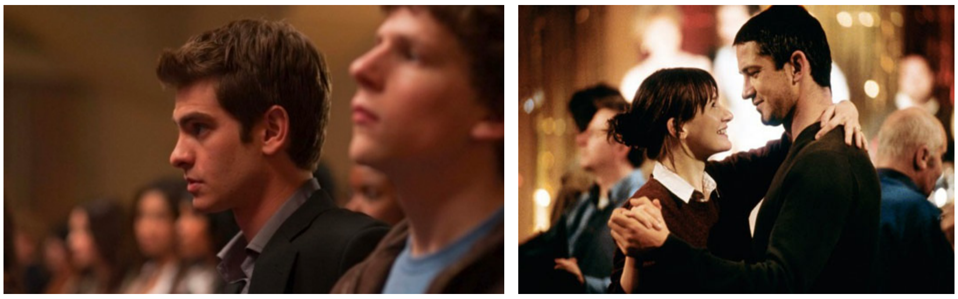


Figure 4: Use of depth of field to define the importance of each character in the film *The Social Network(2010)*(left) and to create an intimate scene in the film *Dear Frankie(2004)*(right). (Pike[43])

Although all of the above is true for films that were originally filmed in 2D, the visual narrative is very different when it comes to films made in 3D. In 2D films, the viewer is always aware that there is a screen hanging from a wall in front of them; in 3D, however, the screen acts more as if there were a hole in the wall or, as it is commonly referred to, "a window to the world", with elements of the world on either side of it[44]. Regarding this interpretation, there are different criteria amongst directors as to where the objects should be placed: some consider the screen strictly as a window and place all objects within it, so that they are all visible to the viewer, whereas others prefer to extend the "world" beyond the screen; but typically both interpretations are mixed within the same films depending on the scene[7].

[1]Motion Blur: *artifact created by blurring several consecutive frames in a film. It can appear naturally while filming, due to rapid movement or long exposure, or it can also be added post-production, typically to increase the sensation of motion*

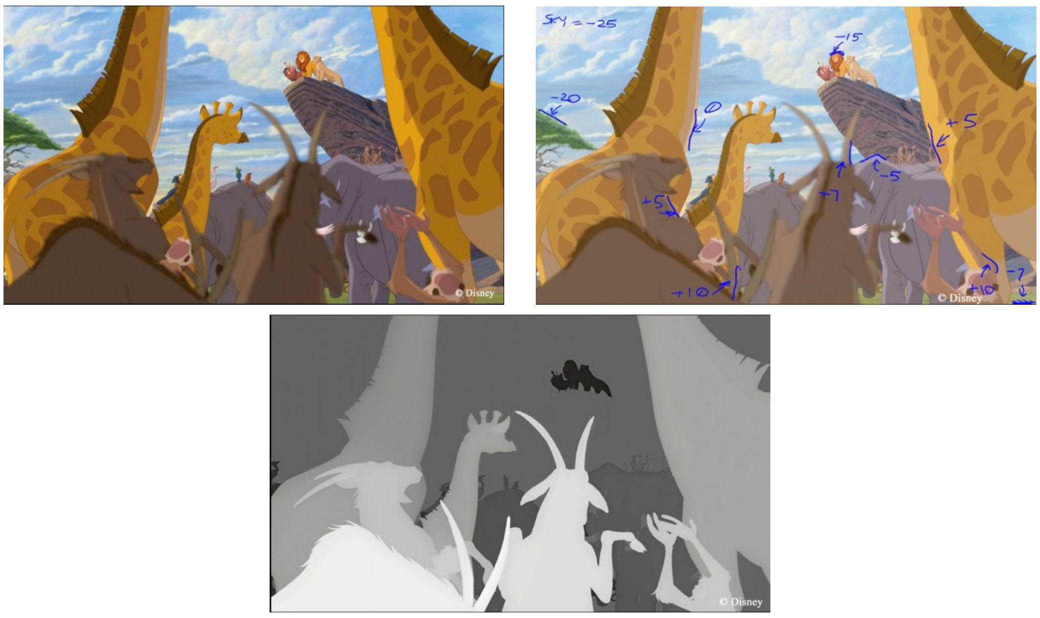
As the reader will probably understand, using techniques like depth of field to draw the attention is something that makes no sense in 3D. Indeed, the sensation of depth in 3D is transmitted by actually placing elements in different depths and letting the visual system to interpret that information in an appropiate way so, in order to give more importance to certain elements in the picture, the filmmakers must now modify their actual location in the 3D space, so that important elements are closer to the viewer.

Not keeping these strong differences between 2D and 3D storytelling techniques may cause conflicts after dimensionalization. For example, when a scene with strong use of depth of field is converted to 3D it may be the case that some elements are more out-of-focus than they should given the resulting depth in the 3D space, which would most likely look ackward to the viewer; the same thing would happen for scenes with strong motion blur. Similarly, given that 3D films try to convey the feeling that the screen does not exist, to make the viewers feel as if they were inside it, it is not desirable to place closeby objects near the edges of the screen because then it would give away the screen. However, cutting elements on the edges of the screen presents more serious issues than just revealing the screen. An actor that is partially cut by the screen looks normal in 2D, because the viewer understands that the rest of the actor is hidden “behind” the frame. But if you present the audience with the exact same shot in 3D, with the character located in front of the screen, on the audience side, they would see only half a character floating in the air. This is an example of what is called a “window violation”; it is intimately related with the vergence-accomodation problem described in Section 2.2.1 and, as such, it can cause eyestrain, headaches and, if there is a lot of motion, even induce vomiting[44]. Brian Gardner[44], stereoscopic consultant for the film *Coraline(2009)*, describes the causes of this effect in in the following way:

> *Part of your perceptual system says, “I can only see half of the person, because the other half of him is behind the window, so he’s behind the window.” Then your stereo system says, “The window frame is behind him, but half of him is way in FRONT of the window! That’s not possible!”* (Brian Gardner, on the effect of window violations[44])

The chances of running into these problems are higher when the conversion is done over a live-action film. Traditionally or CG animated films can typically achieve better results in the conversion process given that all the original material can be easily manipulated, and new footage can be tailored or even re-generated for specific situations when needed, thus reducing the risk of running into conflicts. However, we can find issues like the ones mentioned above, for example, in dimensionalized films like *The Lion king 3D(2011)*. In this case they “pushed vertical objects that violated the stereo window into the negative parallax and as a result created eye strain”, as commented by Russell McGee[45], a student of 3D filmmaking at Indiana University, who also mentions that they “employed a technique of projection of the 2D image onto geometry or 3D modeling which created unusual and warped effects on the 2D perspective of the original artwork, which in my opinion was a major gaff and was very distracting as a viewer”. This seems to be the general feeling of many other viewers’s after watching some of the films that have been re-released in 3D recently.

3D films, in general, are not becoming as popular as the industry thought initially – Engber[47]



a	b
c	

Figure 5: Images from the 3D conversion process of the film *The Lion King 3D (2011)*. a) Original keyframe, b) depth detail markings and c) final depth map. (Graham[46])

shows a very exhaustive analysis of the reasons behind this fact – and this is partly due to the poor results obtained by the current process of dimensionalization which, as we have explained in previous sections, can cause a discomforting feeling in the viewers while watching the film, as well as several other uncomfortable side effects (see Section 2.2.1), which is keeping many viewers away from 3D films. Therefore, it becomes neccesary to find more robust alternatives to the current dimensionalization process to avoid such problems and improve the film viewing experience.

3 Implementation Environment

In this project, rather than a fully integrated piece of software, we propose a solution consisting of a workflow, that is, a combination of several pieces of code obtained from different sources. Given the wide variety of environments and languages used to develop research tools, this approach just came out naturally as different libraries and algorithms became neccesary during the implementation stage.

Once Bundler was chosen as the tool to generate the initial point cloud, it became neccesary to find the right programming library to be able to work comfortably with some of the standard point cloud formats generated by it. To begin with, we needed to choose a programming language whicch had libraries for the required tasks, while at the same time being efficient and fast. For this reason, we chose to combine C++ and Matlab, which are two of the most common languages used in the computer vision and graphics research community. This way, the efficiency and high number of libraries available for C++ is complemented with the simplicity to work with matrix structures – which is how images and point clouds are typically stored – and the even higher number of specialized libraries available for Matlab. As a result, most of the code is done with C++, except for the depth clustering algorithm (see Section 4.4.1 and Appendix A), which is done in Matlab and included into our code as a Dynamic-Linked Library (DLL).

The library that we have used to work with the point clouds is the recently released C++ Point Cloud Library (PCL)[48], which offers a toolbox to simplify the processing of point clouds in several formats, among them PLY, which is one of the output formats generated by Bundler. The most important functions used in the depth estimation process (see Listing A.2 in Appendix A) belong to the PCL library. Some of these functions are read/write methods for PLY files, filtering algorithms such as the Voxel grid filter (see Section 4.3.3) or the RANSAC-based projection filter used to fit the 3D points to our planar model (Section 4.5.2).

Apart from the point cloud PLY files, Bundler also produces what is known as the “bundle file” (with extension .out), which contains the final state of the scene after registering all the images used as input. The bundle file contains information about each camera (rotation, translation, focal length and radial distortion) and each point in the scene (position, color and feature index and position in each frame). This file contains very important information for the reconstruction process, so it is important to find a way to store it somehow in the program, so that we can easily access it from the methods that require it. For this purpose, we created a set of classes that allow the developer to store and access information about the scene extracted from the bundle files, as well as performing several operations such as perspective transformations or distance calculations. Figure 6 shows all the bundle classes and the dependencies between them, and their description is shown in Table 1.

The BundleReader class is used to parse the content of the Bundle file and store the information in an instance of the BundlerScene class, which creates a list with the corresponding number of cameras and points. The BundlerScene class has methods to switch from one view to another,

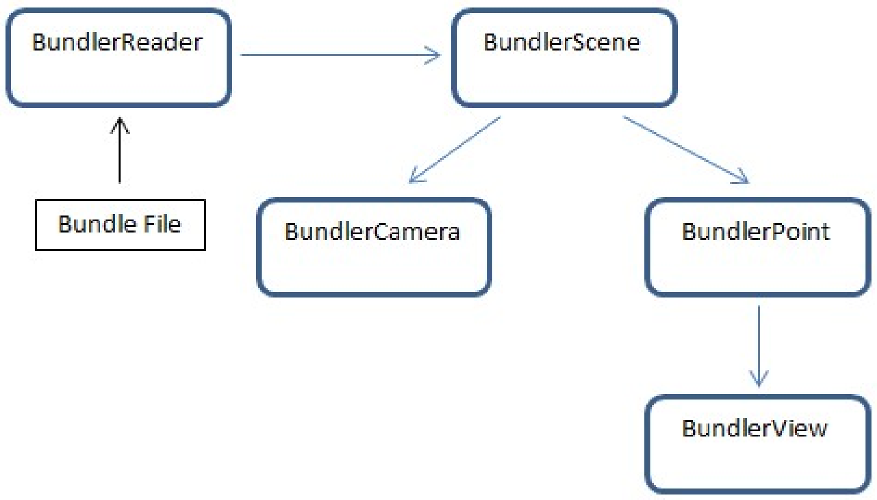


Figure 6: Dependency diagram of the Bundle classes

Class Name	Description
BundleReader	Reads the content from a Bundle file, and stores the information in the corresponding BundleScene instance.
BundleScene	Stores and manages the list of cameras and points in the scene. From this class we can access each of the individual cameras and points.
BundleCamera	Stores and manages the information about a specific camera, such as rotation, translation, focal length and radial distortion.
BundlePoint	Stores and manages the information about a specific point, such as position and color. It also keeps a list of views, which is a list of all the cameras under which the point is visible, and the conditions under which it is visible.
BundleView	Stores the camera corresponding to the view, and the index and pixel coordinates of the feature corresponding to the visible point this view is associated to.

Table 1: Set of classes created to manage the information about a scene stored in the Bundle file.

where each view corresponds to a different frame. The BundlerCamera class stores information about individual cameras, and also provides methods to project 3D point coordinates onto pixel coordinates under that camera. Similarly, the BundlerPoint class also stores information about inidividual points, providing additional methods to check is a point is visible under a certain view or to find the number of cameras that can see the point, as well as methods to compute distances between points in both the 2D and 3D space; each instance of this class keeps a list of all the views under which the point is visible, to be able to access information about each view from them. By point view, we refer to the properties of a point when seen by a specific camera; these properties are stored in instances of the BundlerView class, which contain information about what camera number is associated with the point view, as well as information about the feature associated to the point under that camera, such as the index in the feature array, and the pixel coordinates in the corresponding frame.

As we mentioned earlier, the depth plane clsutering algorithm was implemented in Matlab because it resulted more simple than doing it in OpenCV. Therefore, the neccesary static and dynamic libraries were compiled from a matlab script using the MATLAB Compiler Runtime (MCR). Appendix A shows the two variants that were implemented for the depth plane estimation process, using clustering and a kernel density estimator.

Additionally to the libraries and custom classes mentioned above, the OpenCV C++ library was also used in many early tests that were abandoned. Espectifically, several methods for the generation of depth maps and disparity maps were implemented (see Section 4.3.1), as well as some color correction and image segmentation algorithms such as gamma correction, contrast enhancing and edge detection. Similarly, our own feature detection and camera calibration methods were implemented but later abandoned after it was decided to replace them by Bundler. The tests made with the Kinect were all made using a modification of the Kinect-depth sample included with the OpenCV distribution. Many of these methods have been kept in the code for future reference and possible additions.

4 Project description

During this chapter we will be examining in detail the stages that this project went through, the decisions that had to be made when choosing one or another method, the alternatives available in each case, and the advantages and disadvantages of each one.

The initial stages of the project were dedicated to background research. This is a very new area where a lot of new methods are being researched and developed, so before implementing a solution it is neccesary to study what the available methods are, the conditions under which they can be successfully applied, and whether or not they are a valid solution to the problem at hands. This background research helps in delimiting the scope of the project and finding the right question that will be answered throughout the thesis, thus avoiding unnecessary work that would fall out of the established scope. The following sections present a description of the problem that we are trying to solve in this project, the areas that the research is most focused on, and the main steps that form the proposed solution. This is followed by a more detailed description of each of the stages of the project and the theoretical concepts used at each point.

4.1 Scope of the research

There is indeed an extensive amount of research being done in the field 3D reconstruction, and since the problem of estimating the 3D structure of a film scene is important task within that research area, there are plenty of solutions available to solve this problem. However, given the complexity of this problem, solutions are typically provided for specific cases very well-defined conditions. In the case of films there are several different approaches but, as we mentioned several times earlier, most of these approcahes require some amount of human interaction, which reduces the performance and convenience of the methods in some situations. In this project we propose a new approach to automate the depth estimation process while obtaining results that closely resemble the real values. In the following pragraphs, we will explain the reasons why we decided to follow this approach.

Since we are working with films, by definition, the calibration parameters of the camera are unknown, because we do not have access to the parameters used back when the film was made. Similarly, the input video will always be of a monocular nature. Therefore, we can reformulate our particular problem as the estimation of depth planes from a monocular video. Initially, this thesis was supposed to analyze and compare the different methods available, using DTAM as the main algorithm for the comparison. As we explained in Section 2.3, DTAM calculates depth for every pixel in the scene, so a full dense reconstruction of the entire scene can be obtained easily in a few steps. However, these are all very recent algorithms which are not publicly available yet, so we could not manage to find the source code to implement any of the proposed methods and had to implement our own solution to the problem using pieces from different projects, plus our own code. Thus, the goal of the research became to analyze the advantages and disadvantages of using one or another approach in each of the stages of the depth estimation process, and to propose a combination of those methods that allows to estimate the depth of a scene automatically, yielding values as close as possible to those present in the original scene.

As mentioned earlier, we chose to follow an approach based on Monocular SLAM, extracting the scene depth by matching a set of features between pairs of consecutive keyframes. Kien[49] describes in detail all the neccesary steps of the 3D reconstruction process. Donate and Liu[50] propose a solution using this approach that closely resembles the problem we are trying to solve. Rather than finding a global solution for the entire film, they choose to use different methods for the detection of specific elements, such as people – who are reconstructed by mapping movement of elements in the scene to a pre-existing CG model. Indeed, finding an universal method that can applied to every type of scene is a very difficult task, so most reconstruction methods must make certain assumptions about the conditions under which the solution works, or combine several methods into one solution. Given the time constraints, in this project we have chosen to make assumptions about the type of scene that we can reconstruct, choosing to focus on scenes containing horizontal camera panning, avoiding more complicated camera motions such as rotations or zoom (see Section 5.1 for more details). In the following sections, we will describe each of the steps followed to obtain depth from a given scene, which are also summarized in Figure 7.

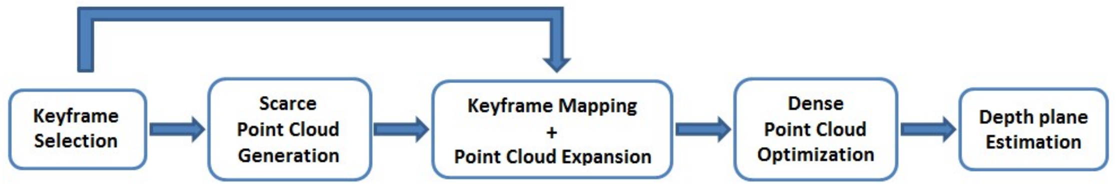

Keyframe Selection
Scarce Point Cloud Generation
Keyframe Mapping + Point Cloud Expansion
Dense Point Cloud Optimization
Depth plane Estimation

Figure 7: Diagram summarizing the steps we have followed in this project to estimate the depth structure from a film scene.

4.2 Camera Calibration

Reconstructing the 3D structure of a film scene is typically a problematic task. We do not have access to the calibration parameters of the cameras and, additionally, the cameras can move freely in space, which results in a problem with uncalibrated and unconstrained cameras. Before we proceed any further, we must explain what a calibration matrix is, and why we need it.

The pixels that we see in a 2D image are the result of a projection transform that maps points in 3D space to pixels in a 2D image. When we project a 3D scene onto the sensor of a camera, many geometrical properties such as shape, distance, and length, are not preserved after the projection. The coordinates of a point in the projective 3D space (denoted by (X,Y,Z)), which is always defined picking out one point as the origin of the coordinate system, are typically expressed in homogeneous coordinates relative to the origin. We can easily convert the coordinates of a point in one 3D projective space to a different projective space with a different origin, by applying the following linear transformation of homogeneous coordinates[51]:

$$X' = H_{(n+1)\times(n+1)}X \tag{4.1}$$

where X are the coordinates of the point in world coordinates, X' are the coordinates of the same point in the camera space, and H is the camera matrix. We may think of the conversion between spaces as having the first space translated and rotated to a different position, the resulting operation is known as a Euclidean transform; the matrix H in equation 4.1 above contains the coefficient of such translation and rotation. Each frame in the sequence is considered as a different projective space with origin at the point where the camera is located; therefore, it is easy to see that, knowing the camera matrices for each view (frame), we can convert the coordinates of a given point from one frame to another. Once we have the point coordinates in the camera space, we can project the point onto image coordinates, by dividing the first two coordinates, X and Y, by the Z coordinate. However, this equation only gives us a projective reconstruction of the scene, meaning that the positions of the points cannot be determined uniquely due to a certain ambiguity that only allows for a reconstruction at best up to a similarity transformation of the world[51]. If we want to obtain a correct reconstruction of the scene, minimizing the projection ambiguity, we must determine some information about the calibration parameters of the cameras. The calibration parameters account for the effects introduced by the camera itself, such as focal length and radial distortion of the lens. Provided we have such information, we can obtain the final pixel coordinates of the 3D points as follows[52]:

$$P = R \times X + t \tag{4.2}$$

$$p = -P/P.z \tag{4.3}$$

$$p' = f \times r(p) \times p \tag{4.4}$$

where equation 4.2 converts the coordinates of point X from world to camera coordinates, equation 4.3 applies a perspective transformation by removing the third coordinate, and equation 4.4 converts the mapped coordinates to pixel coordinates. The function r(p) in equation 4.4 calculates a scaling factor to undo the radial distortion, based on the radial distortion values, k1 and k2, obtained from the camera, as

$$r(p) = 1.0 + (k1 \times \|p\|^2) + (k2 \times \|p\|^4). \tag{4.5}$$

where $\|p\|$ represents the norm of 2D point p.

The camera matrices are calculated from a set of matching points in several images. The method used to obtain the camera matrices from the point correspondences will depend highly upon the number of views. In the most simple case, which is the calibration from two views, a 3x3 matrix of rank-2, known as the fundamental matrix, represents the contraints between both views neccesary to determine the camera matrices for each view(see Figure 8); in the case of three views, the coordinates of the corresponding points are related by 3x3x3 matrix, known as the trifocal sensor.

For each image point X with a correspodning pair x-x', we can visualize the epipolar geometry as shown in Figure 9. The line connecting the camera centers, C and C', is called the baseline. Image points x and x' are the results of mapping point X onto the image planes of cameras C and C' respectively. The plane formed by the baseline CC' and the mapping lines XC and XC' is called the epipolar plane. The points where the image planes cross the baseline are the epipoles,

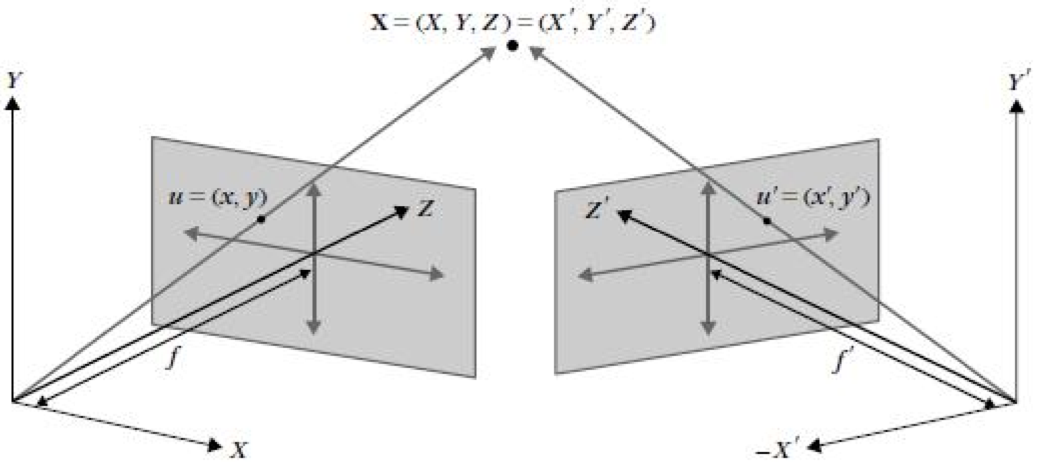


Figure 8: Camera imaging model for two cameras. Object point X is mapped onto both image planes, resulting in image point x(x,y) and x'(x',y') respectively. Points x and x' are called point correspondences. (Aghajan and Cavallaro[53])

and correspond to the image of one view as "seen" by the other camera. Lines l and l' are the epipolar lines, and they represent the intersection of the image planes with the epipolar plane. Ie we knew the location of point x, but not x', we could use this epipolar geometry to find x', since we know tha it must lie somethere on the line l'.[51]

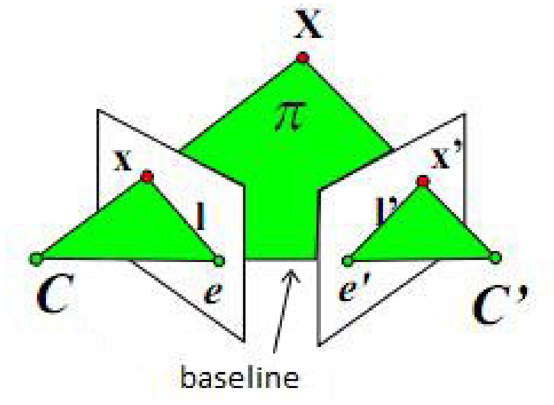


Figure 9: Epipolar geometry.

Several methods were developed to perform the camera calibration process. Initially, the idea was to obtain a dense reconstruction with matches for all the points of the scene, although this method proved impossible given the amount of points in a film picture, as well as the fact that a lot of the points were not visible from all the views – especially when a high number of frames were used in the reconstruction. The first problem, regarding the high amount of points, was resolved by using only a set of features that were present in all the views; the problem regarding the fact the most points were not visible in all the views was resolved by matching features between pairs of frames, and then using a bundle adjustment approach[33] to add new points to the reconstruction as they were found. However, the performance of the implemented solution was not good enough due to other issues whose resolution was out of the scope and the time constraints of this project. Therefore, a simpler approach was chosen, and it was decided to use an existing piece of software which implements the camera calibration process in a similar way. This software is Bundler, which is based on the work of (and often used by) several known authors in this field suchs as Pollefeys[54, 55, 56] and Koch[57, 58, 59].

Bundler – which was initially developed by Noah Snavely as part of the Photo Tourism project[60] and then distributed as a stand-alone program – first uses the Scale-Invariant Feature Transform (SIFT)[61] to detect and match the features across several images (in our case, keyframes), and then estimates the fundamental matrix for each pair of images using RANSAC[62]. Each keyframe is considered as a different view, so that there is exactly one camera per frame. Two main types of files are generated when the execution is finished: calibration files (with extension ".out") and point cloud files –several point cloud formats are available, although in this case we have chosen to use PLY files, given that it is a common format with the C++ programming library used, PCL. The .out calibration files contain information about each camera –rotation and translation, focal length and radial distortion,– and information about the points visible in each one –position, color and feature location. The program processes the frames sequentially; each time new points (features) are found in a frame, a new point cloud is generated with all the points that are visible within the current camera's field of view, so the number of resulting point clouds varies for each video sequence. More information on how it works can be found in the User's manual[52].

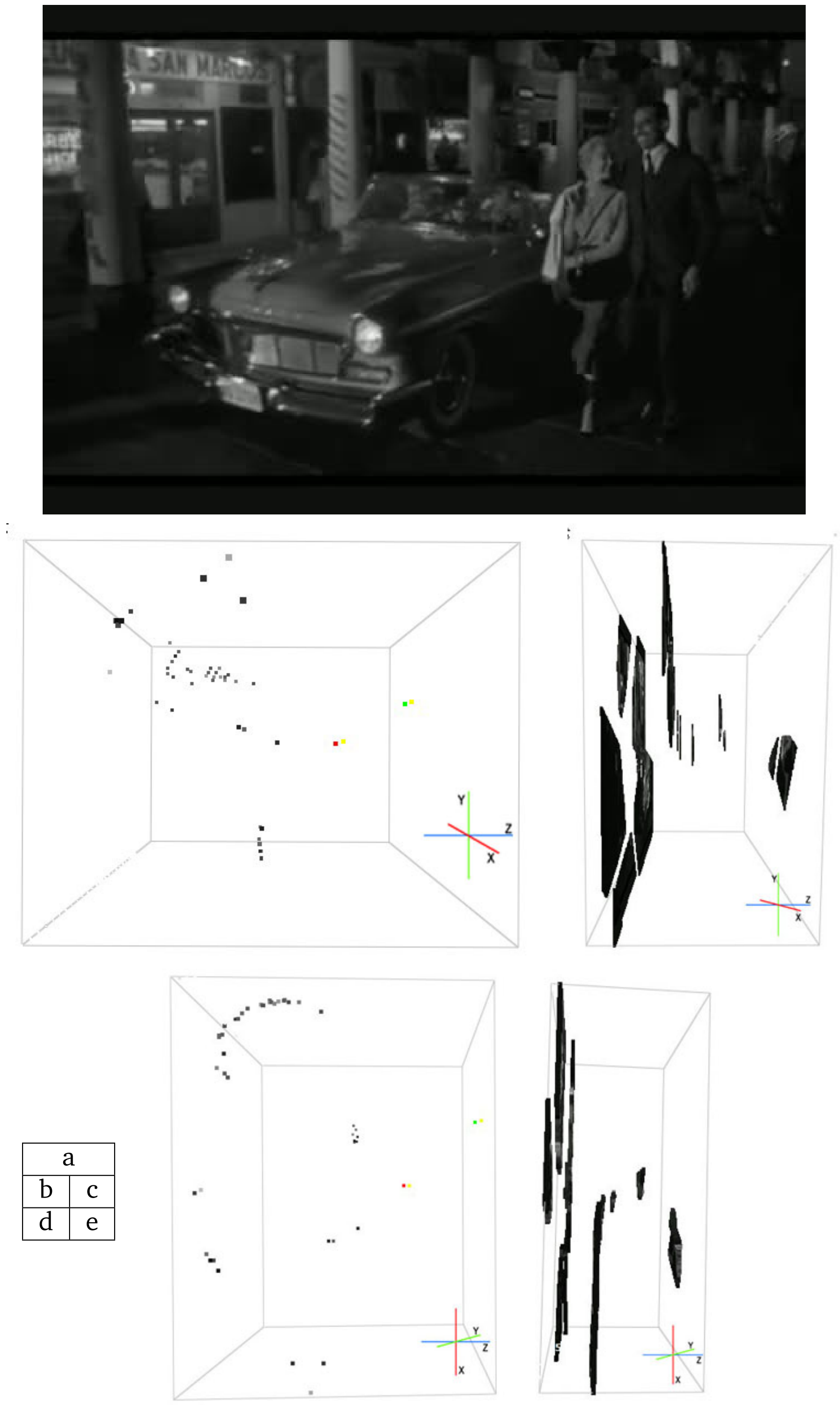
4.3 Point Cloud generation

In this section we will detail the method implemented to construct a dense point cloud from several input frames (see Figure 10). We use as input a series of keyframes from the film sequence that we are trying to reconstruct, and use Bundler to estimate the camera parameters and create an initial scarce point cloud from a set of matching features. A representative keyframe from the input sequence is then chosen to expand the point cloud dataset, which will be refined later on to remove duplicates. We will also comment several methods that could be used to improve the results of the depth estimation algorithm.

4.3.1 Initial point cloud generation

As we mentioned in earlier sections, the original idea was to use a dense reconstruction of the scenes, using each pixel in the frames to calculate the correspondences between the views. The DTAM algorithm was considered to obtain the ground truth data to compare other algorithms against. However, this algorithm has not yet been released and, given the time constraints, a custom implementation was out of the scope of this thesis.

Once the initial idea was discarded, an alternative solution was to compute depth maps – grayscale version of an image, where each gray level represents a depth value, with white values



a	
b	c
d	e

Figure 10: Steps required to generate the final point cloud from a sequence of the film *Touch of Evil(1958)*. a) sample keyframe used to expand the point cloud data set, b,c) side view of the original point cloud as generated by Bundler (left) and the expanded dataset (right), d,e) top view of the original point cloud as generated by Bundler (left) and the expanded dataset (right). The camera is facing towards the negative z-axis, which points left in the side and top views.

representing the points closest to the camera and black values representing the points furthest away from it, although sometimes it may be the other way around – from the selected keyframes, and manually extract the depth information for each pixel to create the corresponding point cloud. Several methods for computing stereo correspondence depth maps were developed, including a method based on the semi-global block matching algorithm[63], and another one based on the Graph-Cut algorithm[64], both implementations included in the OpenCV toolbox. However, a solution based on depth maps is inherently full of problems when working with footage from real cameras[65]. Due to the type of camera motion that is usually present in films, it is often the case when it is not possible to obtain the epipoles from two selected keyframes. This generates errors during the feature matching process which, in turn, results in a wrong or incomplete stereo-matching process. Therefore, the resulting depth maps do not provide a reliable depth information that can be used later to infer the position of each of the elements in the scene.

For all of the reasons above, it was decided to represent the depth of the scene directly as a point cloud. Since we cannot obtain a dense reconstruction of the scene, generating a dense point cloud is of course out of the question for the very same reasons. Therefore, the point cloud has to be generated from the same features used in Section 4.2 to obtain the camera matrices. There is no need to implement this solution, given that the Bundler software also generates a scarce point cloud once the cameras have been calibrated.

4.3.2 Scarce point cloud expansion

Given the difficulty to find matching features amongst keyframes in a film sequence, Bundler tends to return very scarce point clouds, which is not sufficient to do any calculations and obtain valid results. However, since we have already calculated the camera matrices and calibration parameters for each view, we can easily use equations 4.2 to 4.4 in inverse order, to obtain the corresponding 3D coordinates for each pixel in a picture. This way, we will extend the scarce point cloud calculated by Boundler into a dense point cloud.

The expansion uses a given keyframe as input, as well as the reconstructed scarce point cloud pcl. For each pixel px_i in the keyframe we calculate the euclidean distance to every point in pcl – previously converted to pixel coordinates using equations 4.2 to 4.4 – and assign the depth of the closest point to px_i. This way, we end up with a dense point cloud where each point has the same depth as its closest feature. The algorithm could be summarized as follows:

Algorithm 1 Scarce point cloud expansion from keyframes.

```
INPUT: keyframe, pcl;
OUTPUT: pcl2;

for each pixel px in keyframe
    for each feature f in pcl
        Compute 2D coordinates f2d from $(f_x, f_y, f_z)$;
        Find euclidean distance d(f2d,px);
        If d is minimum
            fmin = f;
    end
    p = $(px_x, px_y, fmin_z)$;
    Add p to pcl2;
end
return pcl2;
```

4.3.3 Refining point cloud data

We now have a dense point cloud but, before we can proceed to do any computations, we must refine the dataset to avoid duplicated or redundant points that might bias the depth extraction process. For this purpose, we applied a voxelized grid approach, that consists in creating a 3D voxel grid with a very small leaf size over the point cloud data. A voxelgrid can be thought of as a set of tiny 3D boxes in space, which approximates all the points with coordinates inside it to their centroid. Typically a voxelgrid is used to downsample a point cloud dataset, to reduce its size and complexity; in this case, by using a voxelgrid with a very small size, we can remove all duplicates, while minimizing the amount of useful points that get removed. In fact, our experiments showed that al duplicates were removed from the input cloud, and only a small amount of points, smaller than a 0.01% of the amount of input points, were removed.

4.3.4 Improving results with object segmentation

If we observe Figure 10, we can see that the shapes in the resulting point cloud (shown in Subfigures c and e) is composed by separate chunks of the image, rather than separate objects. We explained in Section 4.3.2 that the expansion is done by assigning to each point the depth of its closest feature, which means that all the points located in the neighborhood around each feature will be in the same depth layer. For this project, we are only interested in finding the main depth levels in the scene, so this approach is sufficient for our purposes. However, this method is certainly not ideal if we were to perform a total reconstruction of the depth of each element in the scene.

To improve the quality of the results, we could implement a simple approach consisting of a combination of edge detection with image segmentation. First, an edge detection such as Sobel[66] is applied to the keyframe image to obtain the main elements in the scene; then, a simple image segmentation algorithm can loop through the pixels in the edge image and obtain the depth for each object. Every time the algorithm finds a new object in the scene – which occurs when an edge is crossed – it checks if this object has been previously visited; if it has not been visited before, the algorithm finds the feature from the scarce point cloud that is closest to the object; otherwise, it assigns the same depth as the last time it was visited. Of course, the algorithm does not need to do any search for features as long as it stays in the same object; therefore, time-consuming computation is only required in the areas around the edges, which reduces considerably the computation time while improving the reconstruction results. More advanced segmentation algorithms could be used in cases where a more exhaustive reconstruction is needed. Additionally, motion flow information could also be used when there is more than one keyframe, to avoid re-evaluating static objects and further improve the performance[67].

We can easily see that the improvements in the performance of the algorithm described above are directly proportional to the size of the objects. As the average object size increases, the number of objects in the scene decreases and, therefore, so does the amount of computation. For example, imagine that we have a frame containing only three objects and a background: in this case, the algorithm will have to find the closest feature for each of the three objects and the background only once, the first time each object is visited; each successive time, the algorithm will recognize the object as already visited and will assign the same depth as the previous time, which would be much more efficient that having to compute the distances for each and every point in the scarce point cloud for every pixel in the keyframe (as in Algorithm 1 above).

4.4 Depth level calculation

Once we have obtained a dense point cloud from the keyframes selected from the scene that we wish to reconstruct, we need to analyze the depth of the point cloud dataset to find the depth values which are most densely populated. We must reiterate here the need to avoid unneccesary depth levels, which could add discomfort in the viewer. We can therefore control the depth fluctuations in the scene by restricting the amount of depth to only the main depth levels. For this purpose, we apply several statistical methods to compute the main depth planes in the point cloud which, in turn, will correspond with the depth planes of the reconstructed scene.

Oehler et al.[68] propose a 3D plane segmentation method based on clustering surfaces according to the orientation of their normals; in the same line, Borrmann et al.[69] analyze variations of the Hough transform to detect planes in the 3D space. In contrast, Wahl et al.[70] and Yang and Foerstner[71] present alternative methods based on RANSAC (Random Sample Consensus) (see Section 4.5.1): the former uses RANSAC to reduce the computational cost associated with the Hough transform, as part of a high-performance point cloud rendering system; the latter proposes to combine RANSAC with MDL to detect planes in situations where the plane orientation is not clearly defined. Yet one more method is presented by Bauer et al.[72], consisting of the combination of RANSAC and vanishing line detection to detect the shape of the most significant elements on the facade plane of buildings. Additionally, Yang and Foerstner[71] also present a very good summarized review of the main approaches to plane detection depending on its application.

RANSAC is one of the most common methods for plane detection in computer vision, given that it can be applied to detect planes in both 2D and 3D, even in the presence of many outliers. However, although solutions based on both RANSAC and the Hough transform were implemented in this project, they both find planes in multiple orientations, which is not what we require for our project. We mentioned in Section 2.4.1 that, in order to simplify our solution, we chose to make the assumption that all of the planes in our scene will be parallel to the camera – which, as we explained, adjusts quite well to the reality. Therefore, we do not follow any of the common approaches for plane detection but, instead, present our own method, that proved to perform better for our specific setup.

A point cloud from a given film scene can be thought of as having many different depth levels, which can vary from as little as just one level, to as many as the number of distinct depth values in the point cloud. Therefore, to define the depth of a scene we must first decide several parameters such as the number of depth levels or the distance threshold from a point to a plane location for that point to be considered as part of that depth plane. We consider an acceptable number of depth planes to be within an interval from three to five, so that enough depth can be perceived, without incurring in some of the issues mentioned in Section 2.4.3. In the following sections, we will describe some of the methods that we have used to find the most adequate location for each of the depth planes, while bounding the number of planes within the specified limits.

4.4.1 Statistical principles

Given the chosen model, where all planes are supposed to be parallel to the camera plane, finding the position of the depth planes in a given scene consists simply on finding those values along the z-axis of the point cloud with the highest concentration of points. There are several statistical techniques that we can apply to achieve the desired result; we implemented some of the solutions, but only two of them seemed to achieve good results, using a Kernel Density Estimator (KDE) and Clustering.

Kerner Density Estimator

A KDE is a variation of the histogram that tries to find the curve that best adjusts to a given data density distribution by means of data smoothing, and it is defined as follows:

$$\hat{f}_h(x) = \frac{1}{n}\sum_{i=1}^{n} K_h(x - x_i) \quad = \frac{1}{nh}\sum_{i=1}^{n} K\Big(\frac{x - x_i}{h}\Big), \quad K_h(x) = \frac{1}{h}K\Big(\frac{x}{h}\Big) \tag{4.6}$$

where K is a nonnegative function, integrating to 1, called the kernel function, and h is the *bandwith*, a positive parameter that controls the smoothness of the curve. K_h is called the *rescaled kernel* function and, provided that K(x) is a standard normal density, $K_h(x - x_i)$ represents the normal denstity with mean x_i and standard deviation h[73]. There are several types of kernel functions, depending on the type of density distribution, such as uniform, triangular, gaussian, normal, etc; the normal kernel is the most commonly used due to its mathematical properties, and it is the one we have used for our tests. The bandwidth is equivalent to the bin width in a histogram and, as such, it is directly related to the number of modes in the resulting curve and, in our case, the maximum number of depth planes. Generally, the bandwith is selected keeping in mind the shape of the resulting curve, finding a compromise between a ragged and an oversmoothed curve; however, we are more interested in controlling the resulting number of maxima in the curve, so that it is always within a given interval –in our case, we choose this value to be between 3 and 5, which are the minimum and maximum number of depth planes that we want to obtain. We implemented our depth estimation algorithm using the KDE described by Botev et al.[74], performed several tests to find the right bandwidth, and then used the same bandwidth in all our tests. The graphs in Figure 11 show the result of using a KDE of bandwidth 16 – which corresponds exactly with the square root of the maximum number of desired depth planes minus 1 – for the point clouds in Figure 10.

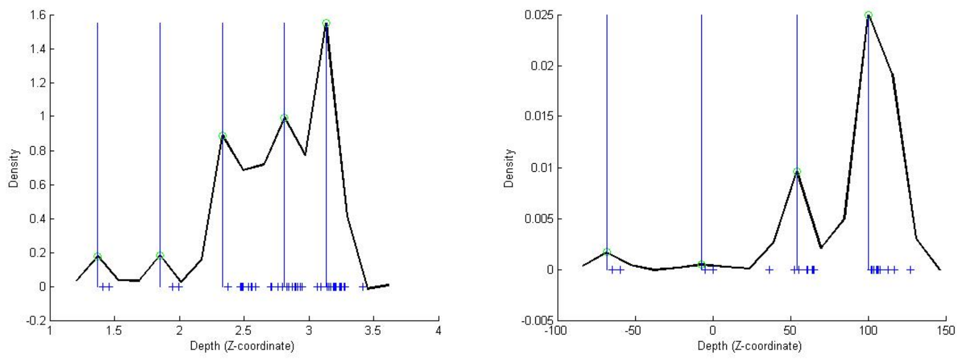


Figure 11: Depth planes found using a KDE with a bandwidth of 16, for the point clouds in Figure 10.b,d (left) and 10.c,e (right), respectively. The depth values in the right graph have been scaled by 100 for visualization purposes.

We can see that the number of detected depth planes is different when using the scarce point cloud than when using the dense point cloud. In the scarce point cloud, the points are much more scattered, so the peaks in the density curve are not as clearly marked as in the dense curve. The latter case is a more realistic situation, since we want to extract the depth planes when all the points are present in the cloud, and not just a scarce set of features. However, the results would be more accurate if we used some additional method for depth reconstruction, like the one explained in Section 4.3.4.

Depth clustering

Data clustering is an unsupervised data analysis technique that tries to find groups (or clusters) in a data distribution, based on some similarity criterion among the objects or variables, so that the intra-group similarity is higher than the inter-group similarity. Typically, each object in a given partition belongs to one and only one cluster, whereas each cluster must contain at least one object[75]. The clustering process separates regions in the pattern space in which the patterns are dense (modes), from regions of low pattern density. Each cluster is assigned a center and each pattern is then assigned to the cluster with the closest center[76].

To find the areas of the z-axis containing the highest concentration of points, we apply a data cluster analysis over a data distribution containing the depths of all the points in the point cloud. We use an approach based on the k-means cluster analysis, which partitions the initial set into k clusters and minimizes the sum of squared distances (classification "error") of the samples about the cluster means[77]. We could say that k has a similar role in cluster analysis as the bandwith in density estimation; using k-means with 5 clusters provides a straightforward solution to our limitation in the number of depth planes that result from the classification. Besides the number of clusters, k, we use an additional parameter, threshold T, which determines the minimum distance allowed between two clusters as a percentage of the total depth range. Initially, random seeds are chosen uniformly from the distribution, and repeated during 5 iterations; if two clusters are closer than the specified threshold, the second cluster is removed, and a new iteration is done to redistribute the orphan values amongst the remaining clusters. The Matlab code used to achieve this task can be found in Appendix A

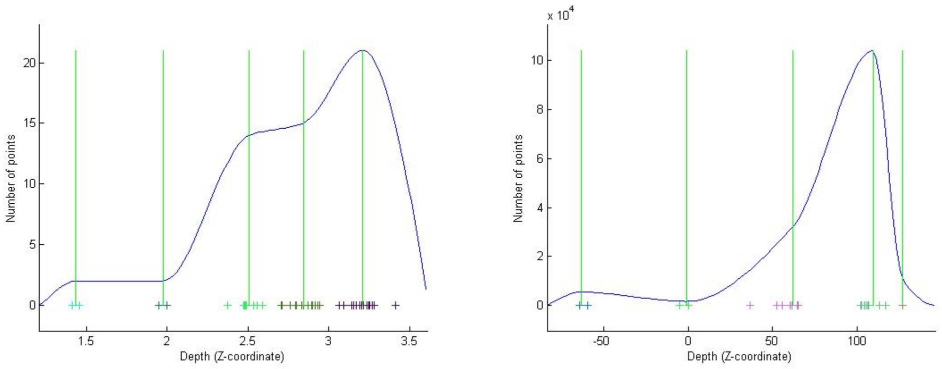


Figure 12: Depth planes found using k-means cluster analysis with 5 clusters and a threshold of 4, for the point clouds in Figure 10.b,d (left) and 10.c,e (right), respectively. The depth values in the right graph have been scaled by 100 for visualization purposes.

The curves displayed in Figure 12 represent a histogram of the density in each cluster – as opposed to the graphs in Figure 11, which show the actual density distribution from the entire population – and that is the reason why they are much smoother than those in Figure 11, even though a similar number of planes were detected.

The reason why we chose cluster analysis over density estimation is because the former gives us more flexibility in deciding the location of the the final planes. The location of the planes found by the kde depend mostly on the density distribution, so no realocation can be made based on additional factors; with cluster analysis, however, we were able to introduce the threshold parameter, to obtain a more realistic distribution of depth planes.

4.5 Fitting data to depth planes

The final step in the process of depth estimation – after we have calculated the amount of depth planes, as well as their most optimal location – is to project the points in the point cloud onto some type of predefined parametric model (a plane in our case), so that we can assign their corresponding depth to each point in the scene. In the following sections we will describe the RANSAC algorithm which is the most commonly used for this type of solutions. We will also describe the parametric model and, especially, the plane model, which is the one that best adapts to our scene setup

4.5.1 Point projection

We commented in Section 4.4 that two common solutions to the detection of 3D planes in point cloud data were the Hough transform and the RANSAC algorithm. We chose not to use any of these two solutions because they find planes in all possible orientations; however, at this stage, we know the parametric equation of the surface we want to project our data to, so we can specify the coefficients of the geometric model, instead of running RANSAC blindly.

Random Sample Consensus(RANSAC)[62, 78] is an algorithm that can sucesfully detect different basic shapes (such as planes, spheres, cylinders, or cones) in point clouds, in both 2D and 3D, and presents a strong tolerance to outliers. RANSAC manages to search the best plane among a 3D point cloud in less iterations than other algorithms, even for a large number of points. The algorithm initially takes three random point from the point cloud and calculates the parameters of the plane that connects them; then, it finds which of the remaining points in the point cloud belong to this plane, according to a certain distance tolerance threshold. This procedure is repeated N times, saving the best of the N results[71].

Since we know the parameters of the planes in advance, there is no need to choose the three initial random points; instead, we start with the parametric equation of the plane, so we just need to find all the points in the point cloud that belong to that plane, according to a previously specified distance threshold. We calculate the distance threshold dynamically so that, for each plane, only the points closer than half the distance to each neigboring plane (on both sides) are considered inliers; the rest of the points will be rejected as outliers and will be projected to a different plane (see Figure 13).

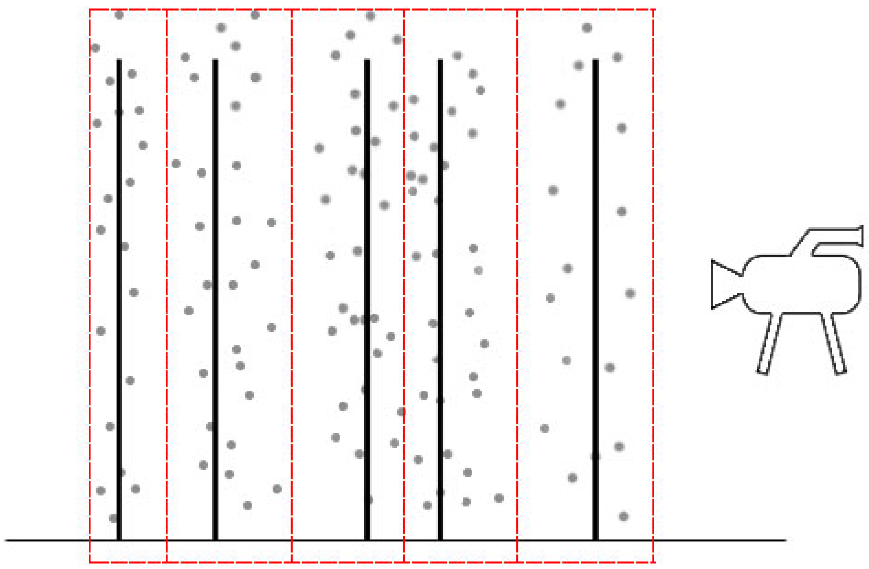
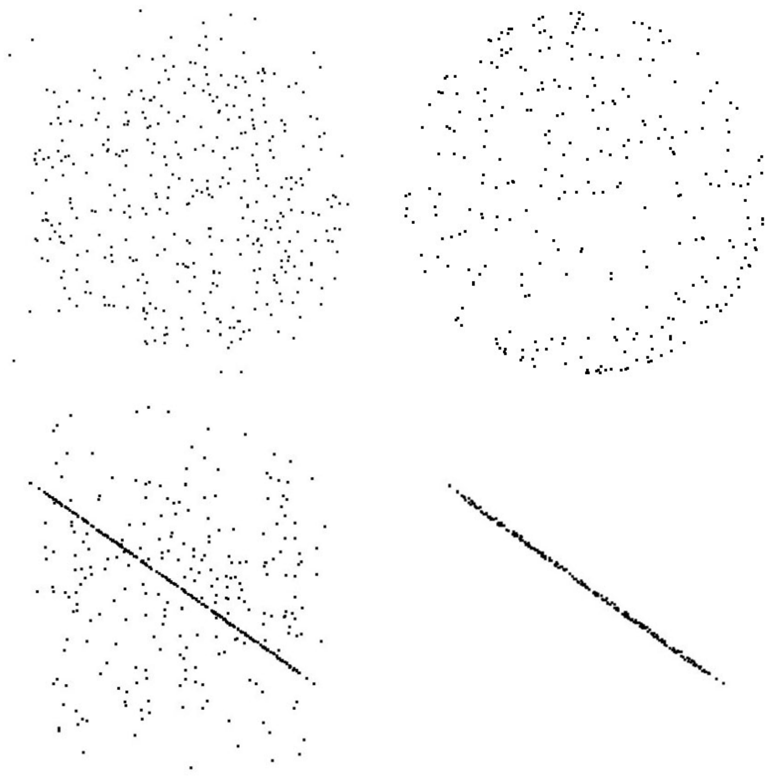


Figure 13: Selected distance threshold for a sample distribution of depth planes. The dashed line represents the area containing the points projected to each plane.

4.5.2 Geometric fitting

Fitting data to a curve/surface (also called geometric fitting) is a very common task in areas such as computer vision or computer graphics to extract shape patterns from a set of points in 2D and 3D space[79]. Given a set of points, we can use a parametric model to find the shape that best adjust to our distribution, thus separating useful points from outliers. The parametric model is typically described by a set of coefficients which, in the case of geometric fitting, corresponds to the equation that defines the specific shape.



a	b
c	d

Figure 14: Example of geometrical fitting of a point cloud to a sphere (a,b) and a line (c,d). The left and right columns show the point cloud before and after the projection respectively.

Figure 14 shows the result of projecting a point cloud onto a line and sphere. Given the depth model that we have been describing, we need to project the points to a 3D plane, which is given by its parametric equation $ax + by + cz + d = 0$. Therefore, we need planes that are parallel to the camera plane which, in other words, means that the planes will all be lying on the X-Y plane, at different depths. The equation of the X-Y plane is $z = 0$, obtained by setting coefficients a, b and d to zero and c to one. Similarly, the planes parallel to X-Y at depth 1, 2, etc, will be given by the equations $z = 1$, $z = 2$, etc, so that we can obtain the equation for any depth plane by setting coefficients a and b to zero, and c to one, giving coefficient d the additive inverse of the corresponding depth value.

Data classification and parameter estimation are two factors that strongly depend on each other[80]; however, since we calculate the location of the depth planes in advance, we already have all the parameters neccesary to fully define each of the plane equations. We use RANSAC to classify the point cloud points according to several parametric models – one per each depth plane found in the previous step – so that those points located at around each of the depth planes, within a certain distance threshold, are assigned the same depth as the plane. The rest of the points that do not fit into one plane model are considered outliers for that model, and will be projected onto another plane.

5 Experimental results

In the previous sections we have explained some of the techniques that can be used to detect depth planes in a film scene, as well as the methods we have implemented to accomplish such task. To asses the quality of the results obtained with such methods, we have performed a series of experiments that evaluate specific aspects of these results. Since we are using the Microsoft Kinect to generate depth reference data to evaluate our results, the first experiment we performed was to obtain the equivalence between the depth units returned by the Kinect and the real depth values present in the scene, as well as to check the linearity of such equivalences as objects move away from the sensor. Once this was done, we performed two additional experiments to evaluate the quality of the results, both quantitatively (comparing the depth results against the Kinect) and qualitatively (assessing the visual response of human observers).

5.1 Assumptions

Throughout the previous sections, we have mentioned several of the assumptions that we have made in this project. Some of them have been imposed by the specific solution that we have implemented in each case, whereas some others have just been chosen for simplification purposes. In this section we will describe these assumptions formally, so we can clearly delimit the scope of the research.

The first simplification we do is to see the depth in a film scene as consisting of planes that are parallel to each other and, in turn, parallel to the field of view of the camera. We explained this model in much more detail in Section 2.4.1 but, to summarize, this assumption simplifies the depth extraction model, because we know in advance the parameters of the fitting model that we have to use to estimate the depth planes, avoiding the additional computation associated with having to find each possible orientation.

The other important simplification that we have made in this project is to reduce the number of possible camera trajectories, and focus exclusively in the reconstruction of scenes where only a camera panning [1] is present. As described in Section 4.2, this simplification reduces the chances of errors during the camera calibration process due to feature mismatch and, at the same time, simplifies the keyframe selection process.

We decided to make these two main assumptions due to time contraints because, although an implementation of the proposed solution would also be possible without them, it would require additional steps and much longer computation times. Therefore, we made these simplifications to reduce the complexity of the problem at hands.

[1]Although, in cinema, the term camera panning sometimes implies some amount of rotation in the camera, in this case we only use it when referring to a lateral displacement, with no rotation.

5.2 Ground truth data

As with any other scientific experiment, we must have some ground truth data to compare our results with, to be able to evaluate the quality of our solution. Finding ground truth data about films is a very difficult task, because it means that we have information about the position of the elements in the scene at the time when the film was made, which is obviously something that we cannot have access to. Although it is possible to find rough information about the location of the elements in a given shot, exact measurements are not neccesary in most films and, therefore, not available. Thus, in order to compare our results, we must find some alternative way to find that ground truth information.

We explained in Section 4.2 that, due to the camera calibration process, the 3D reconstruction always has a certain projection ambiguity, so it is not possible to find the exact distances between the elements in the scene; however, this is not required in film reconstruction, because the relative distances can always be scaled up or down. We chose to obtain ground truth data by setting up our own scene, noting down the real distances between the objects, and comparing these values with our results. Although the properties of the testing scene – namely, an “artificial” [2] static scene with hand-positioned objects – do not stricly correspond with those found in an actual film, we can use these results to measure the precision of our solution, in terms of how similar the relative distances between the reconstructed depth planes are to the actual distances.

5.3 Microsoft Kinect

The performance evaluation that we have done so far only considers relative distances between the depth planes, due to the projective ambiguity. Then, to measure the magnitude of this ambiguity, and also as an additional test, we used the Microsoft Kinect to obtain additional ground truth data with real distance values. The Kinect was first released by Microsoft in 2010 as a motion sensing input device for their Xbox 360 video game console. A Windows version as well as a software development kit (SDK) was released soon thereafter – apart from the Kinect SDK from Microsoft (whose official version was released in February 2012), some other implementations exist, such as the OpenNI SDK, or the open source libfreenect library – which, along with its low cost, has contributed to making it a popular device in the computer vision community, as a cheap substitute for the more expensive traditional 3D cameras, such as time-of-flight (ToF) cameras.[81]

The Kinect sensor consists of three main components: a color VGA camera (the RGB camera) a depth sensor (an infrared projector and a CMOS camera with an IR-pass filter which simultaneously captures the projected image), and a multiarray microphone (an array of four microphones that allows to isolate the voices of the players from the ambient noise)[82]. Table 2 shows the current hardware specifications of the Kinect.

The depth values returned by the depth sensor represent distances from the camera-laser plane, rather than from the sensor itself (see Figure 15), so the (x,y,z)-coordinates of 3D objects can be obtained directly from the values returned by the depth sensor[81]. However, the values

[2]We will sometimes talk about an artificial scene when referring to a scene that was created for the sole purpose of generating ground truth data, as opposed to the standard film scene that is the type of data that our solution is oriented to.

Property	**Value**
Angular Field-of-View	57° horz., 43° vert.
Framerate approx.	30 Hz
Nominal spatial range	640 x 480 (VGA)
Nominal spatial resolution (at 2m distance)	3 mm
Nominal depth range	0.8 m - 3.5 m
Nominal depth resolution (at 2m distance)	1 cm
Device connection type	USB (+ external power)

Table 2: Current hardware specifications of the Microsoft's Kinect.

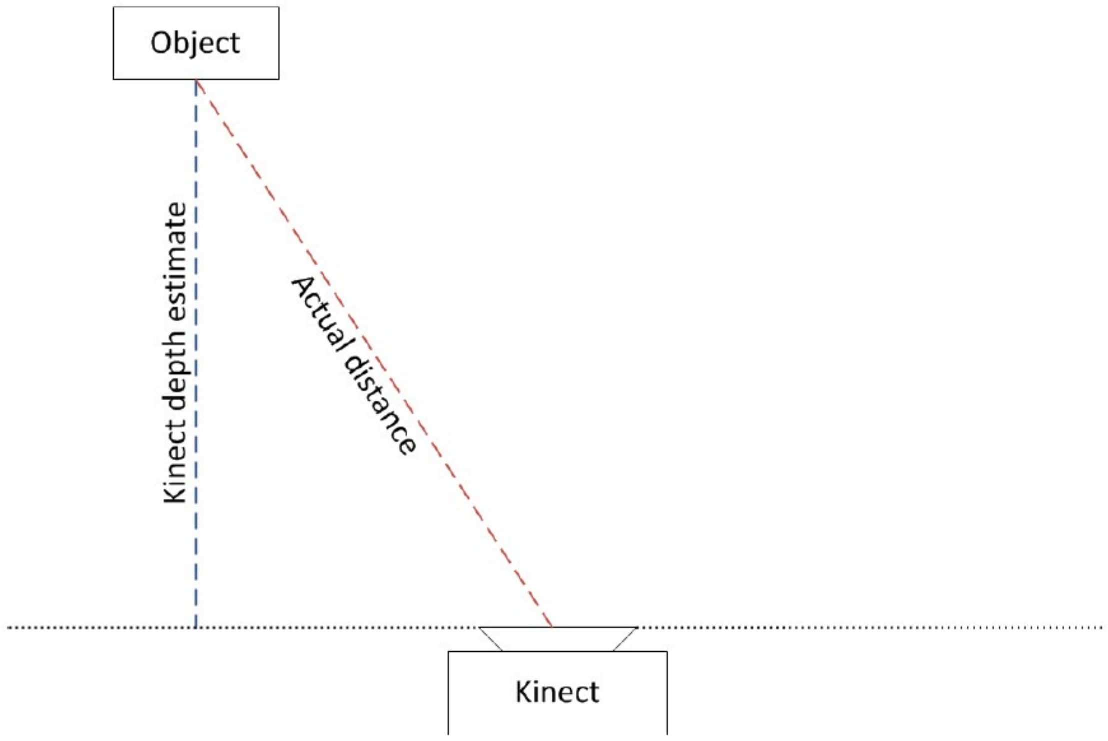


Figure 15: The depth calculated by the Kinect depth sensor is measured as the distance from the object to the camera-laser plane rather than the actual distance from the object to the sensor.[81]

returned by the depth sensor are expressed in depth units rather than metric units and, therefore, we must find the equivalence between depth units and real distances in metric units to be able to compare the results with our solution. The hardware specifications in Table 2 are only theoretical values, so we performed our own experiment to obtain the real values instead of the theoretical ones.

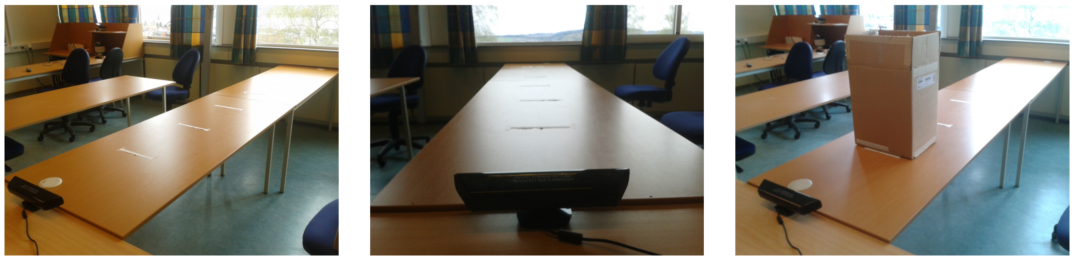


Figure 16: Setup of the experiment conducted to obtain the equivalencies between depth units and actual distance for the Kinect.

To obtain the equivalencies between depth units measured by the sensor and actual distance, we performed a simple experiment. Over a long table, we placed several distance marks, from 30 to 325 centimeters, every 25 centimeters, on a line perpendicular to the field of view of the Kinect (see Figure 18); then a cardboard box was placed on the table at each of the marks, in front of the Kinect's field of view, and the corresponding color and depth images were captured with the Kinect. Then a region inside the cardboard box was chosen, and the depth of the pixels inside it was averaged and compared to the actual distance. The results are shown in Table 3 and Figure 17. If we observe those results, we can see that the real distance, in centimeters, corresponds roughly to twice the amount of depth units – some small differences are acceptable due to small deviations introduced by averaging and noise. The point cloud, on the other hand, does store absolute distances (in meters), so that the depth of each point in the point cloud is in fact the real distance to that point from the camera. Then, we can obtain the real distance between two objects by using the Kinect and, therefore, we can rely on it as a reference device for obtaining ground truth data. Similar experiments have been conducted by Andersen et al.[81] and Crock[83].

Actual Distance(cms.)	**Kinect Depth**	**...Actual Distance(cms.)**	**...Kinect Depth**
30	0	200	100.4562
50	13.6591	225	113.1760
75	38.0000	250	126.3545
100	50.0232	275	138.9910
125	62.3390	300	151.6228
150	75.0047	325	164.5770
175	87.8286	358	181.4329

Table 3: Equivalencies between depth units returned by the Kinect and actual distance. The graph corresponding to these values is shown in Figure 17.

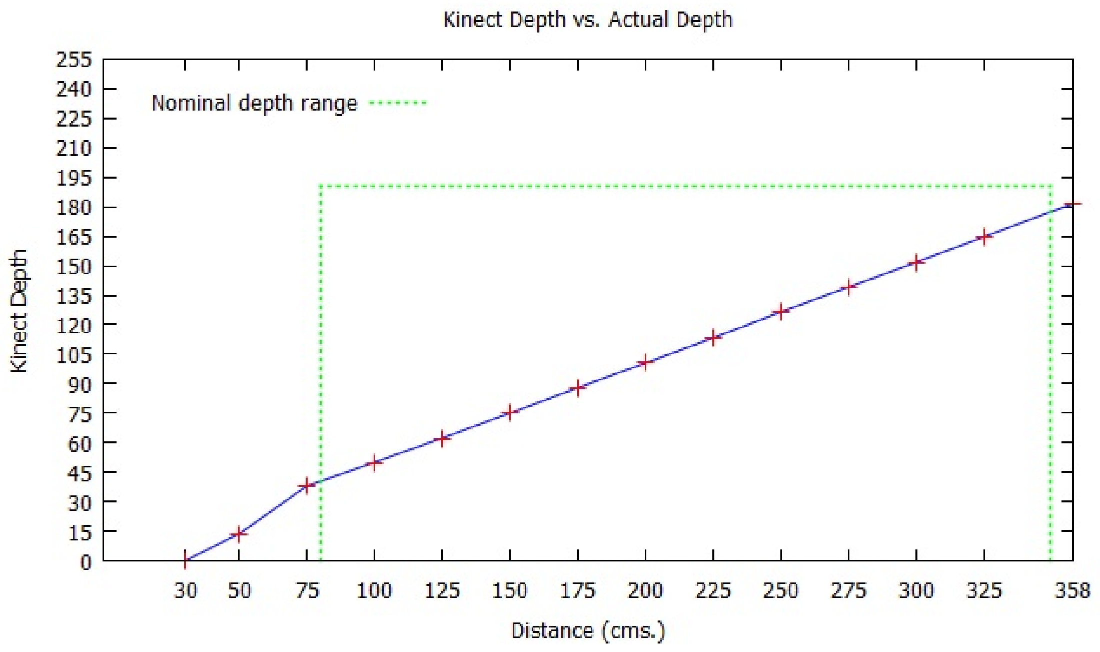


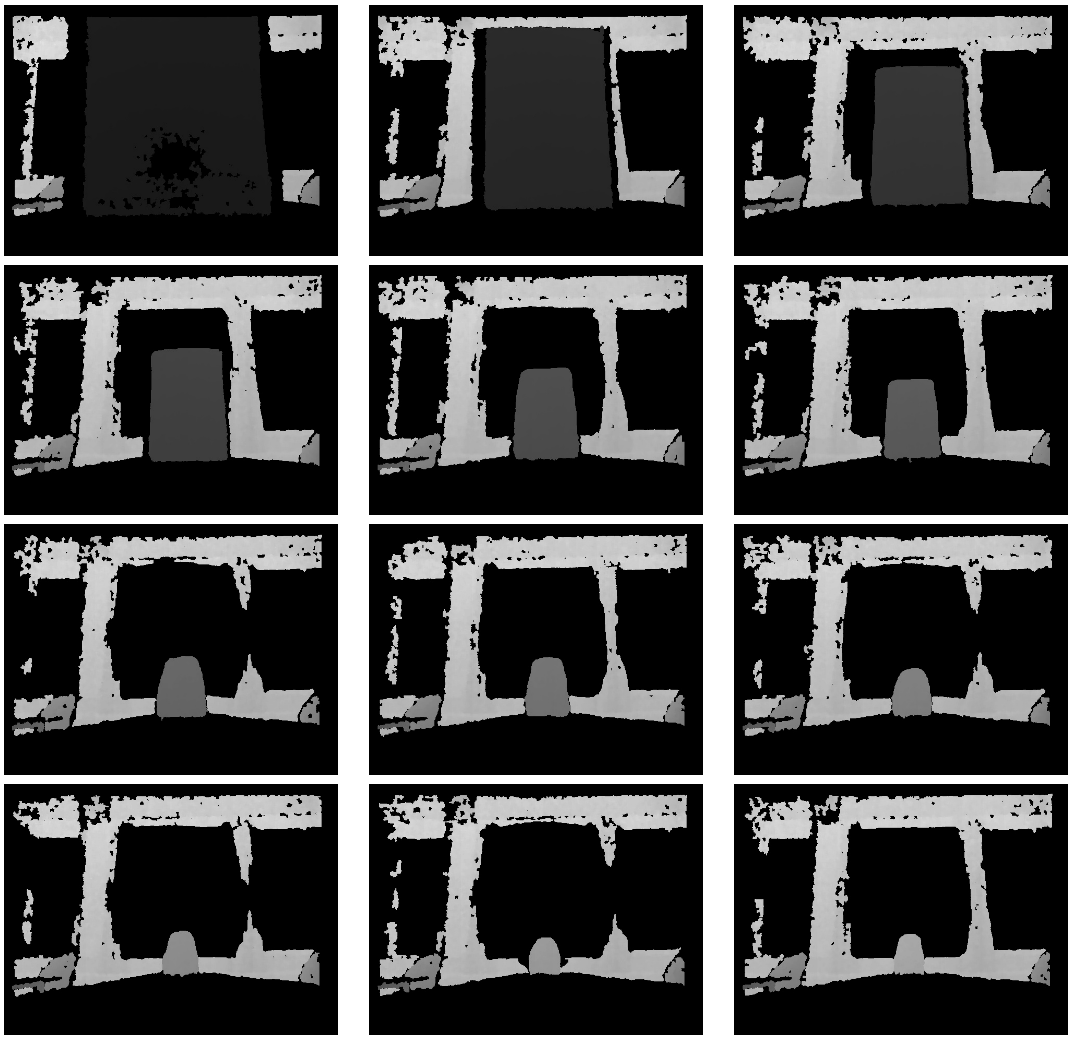
Figure 17: Equivalencies between depth units returned by the Kinect and actual distance.

Looking at the graph in Figure 17 we see that there is a nearly perfect linear correspondence between depth units and actual distance for any distance within the Nominal depth range, which means that the values obtained with the Kinect are reliable, without any loss of precision, for any object within this range. The graph also shows that, for values below the nominal range (80cm), the correspondence is not linear, which is due to the fact that the sensor does not detect all the points on the box properly. If we look at the images in Figure 18 we can see that, indeed, the maps obtained for distances closer than 75 cms. have some “holes” in them (the depth map corresponding to a distance of 30 cms. was not included because it is a completely black image), meaning that the sensor cannot measure the distance correctly; similarly, object that are located on the far background are shown as black because they fall out of the depth range.

5.4 Results

We have already described several ways to obtain the ground truth data that will allow us to evaluate the performance of our solution. In this section we will show a fully working example and will discuss the goodness of the results, comparing them with the depth data generated by the Kinect as explained in Section 5.3.

In the same setup used for the Kinect’s depth experiment described above, we positioned several objects at five different depths, which were measured manually, including foreground, middleground and background (see Figure 19). Then, the Kinect was positioned in front of the scene (looking towards the window), and used to obtain a depth image and a point cloud of the scene, which would be used as ground truth (Figure 20). Next, a short video sequence of the



a	b	c
d	e	f
g	h	i
j	k	l

Figure 18: Depthmaps obtained with the Kinect for an object located at 50(a), 75(b), 100(c), 125(d), 150(e), 175(f), 200(g), 225(h), 250(i), 275(j), 300(k) and 325(l) centimeters, respectively.

scene was recorded by panning a camera sideways along the Kinect's sensor plane, and several keyframes were extracted from this video sequence and used as input for our program.

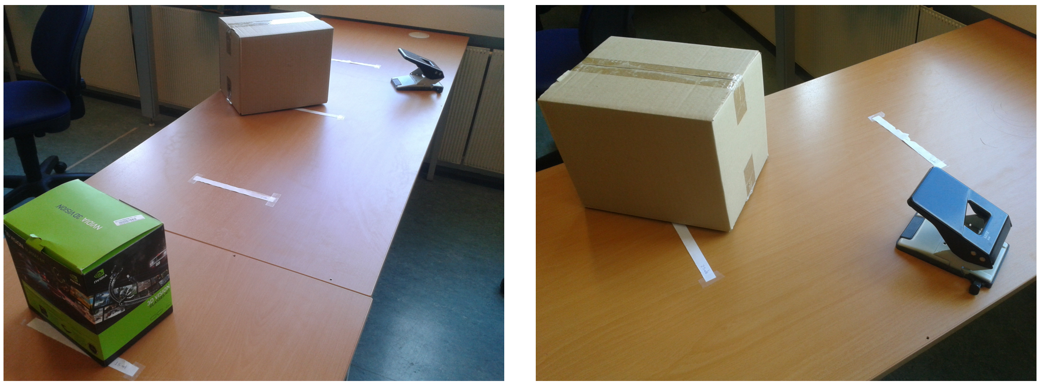


Figure 19: Position of the objects for a simple study case experiment. The left image shows all three objects, located in the fore and middleground; the right image shows a close-up of the two objects in the midleground, so that the distance between them can be appreciated more clearly. Objects outside the room, visible through the window, are considered to be in the bakground.

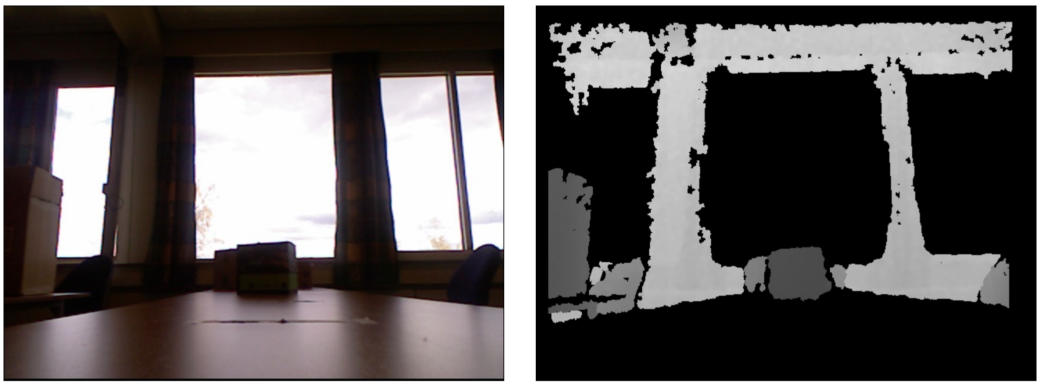
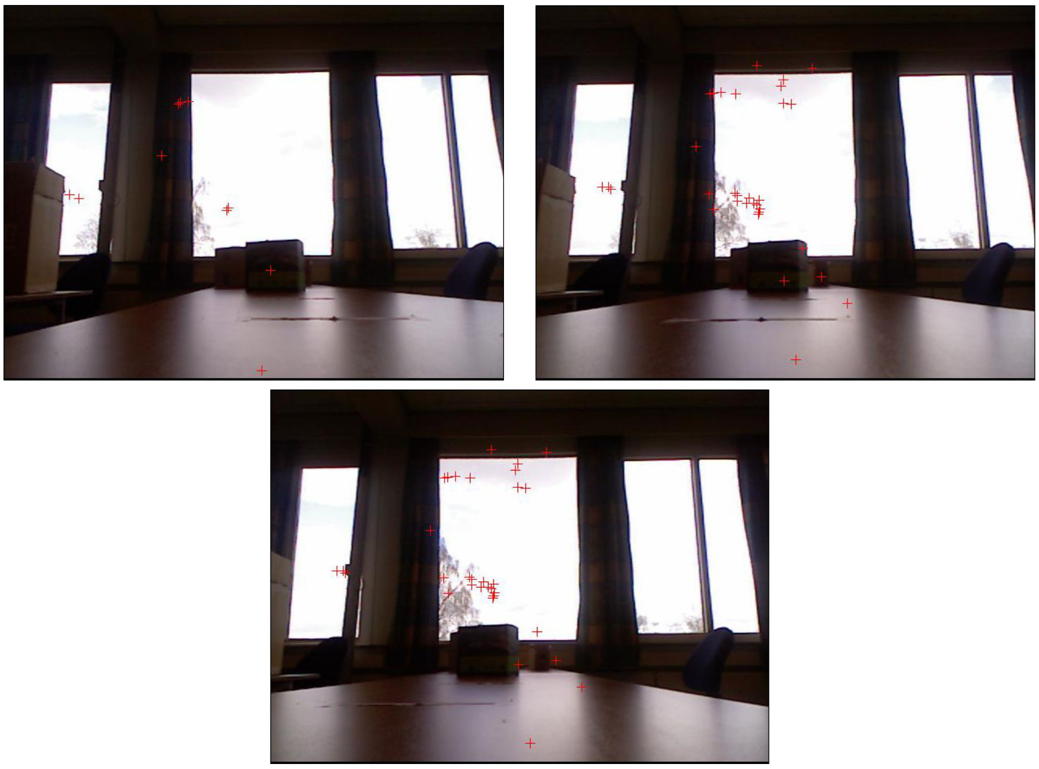


Figure 20: Kinect capture of the scene to use as ground truth: a)rgb image (left) and b)depthmap (right).

Due to time constraints, we have not implemented any method to select the keyframes automatically from a video sequence. However, since most of our tests have been done for short video sequences, we can afford to extract the keyframes manually. For this example, we used a total of 11 keyframes from the camera panning sequence. These keyframes are used as input to Bundler, which will then find the main features using SIFT, match them across the frames, estimate the matrices and calibration parameters for each camera, and generate a series of scarce point cloud files. Figure 21 shows three sample frames, along with the features found by Bundler in each one of them. Depending on the keyframe that we want to reconstruct in 3D, we have to choose the corresponding point cloud to use – since a new point cloud file is created every time new features are found, it is possible that there is not a point cloud for each keyframe, in which case we will choose the latest update generated before reaching the keyframe – along with the keframe, to estimate the dense point cloud with one point for each pixel in the frame.



a	b
c	

Figure 21: Features extracted from keyframes 3 (a), 5 (b) and 6 (c).

The next step, after generating the dense point cloud, is to apply the statistical methods described in Section 4.4 to estimate the location of the main depth planes present in the scene. Figure 22 shows the real distance between each one of the objects used in the experiment.

Figures 23 and 24 show the clusters as well the estimated location of the depth planes in the point clouds obtained with the Kinect and our solution, respectively –these depth values are the same as those shown in the depth map in figure 20.b. As we can see in the figures, in the Kinect the camera is looking down the positive z-axis – so the depth of the points is always positive – as opposed to our solution, where the camera is looking down the negative z-axis – and, therefore, the depths are negative. This means that, to keep the consistency when comparing the results obtained for both datasets, we must keep this fact into account when matching the planes between both datasets.

To compare the quality of our solution, we have to calculate how similar the distances between each plane are. As we explained earlier, we are interested in obtaining relative distances, rather than absolute locations, so to do the comparison, we calculate the normalized distance –in a [0-1] interval– between each one of the planes in relation to the total depth range of the

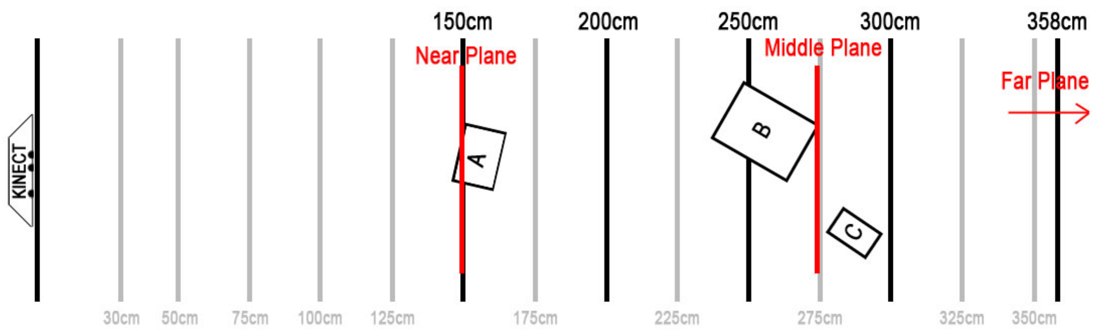


Figure 22: Distribution and real distances of the objects used for the experiment. A rough manual estimation of the location of the depth planes, based on the location of the objects, is shown in red.

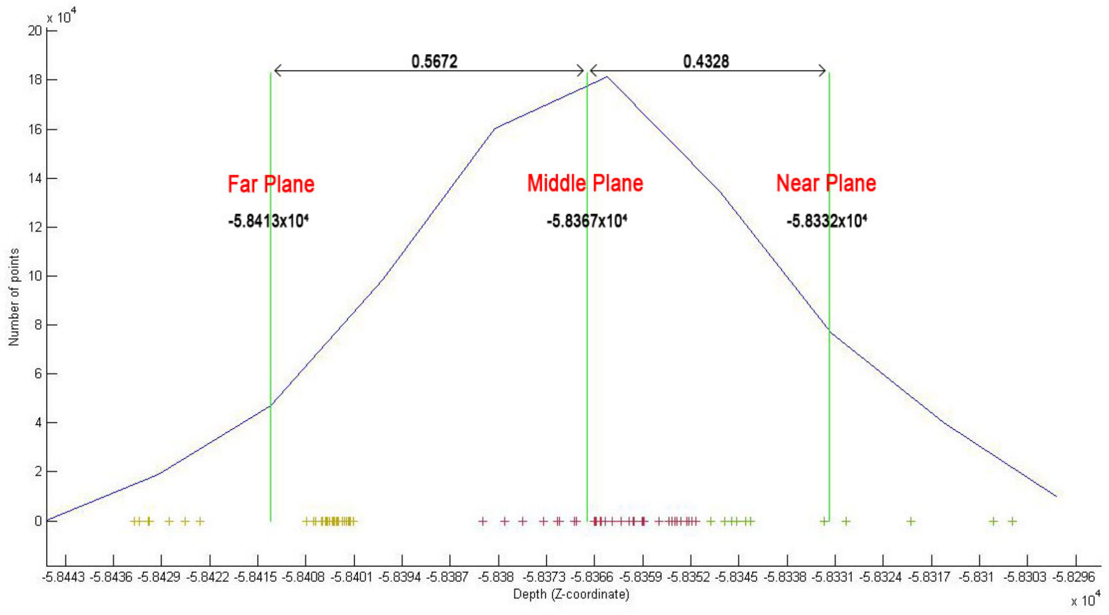


Figure 23: Depth clustering of the point cloud obtained, with our solution, for the camera panning sequence. The image shows the estimated position of the depth planes, as well as the normalized distances between them. The camera is positioned to the right-hand side of the graph. The units are artificial depth units.

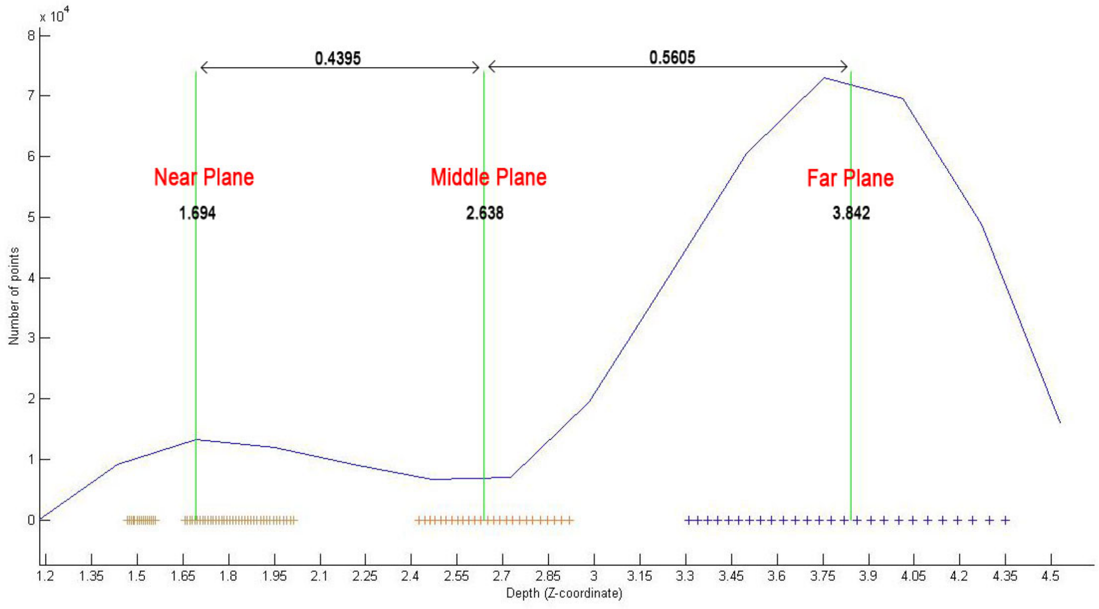


Figure 24: Depth clustering of the point cloud obtained, with the Kinect, for the camera panning sequence. The image shows the estimated position of the depth planes, as well as the normalized distances between them. The camera is positioned to the left-hand side of the graph. The units represent real distances in meters.

scene – that is, the absolute distance between the camera's far and near planes. The normalized distance is computed as follows:

$$\mathrm{distance}_i = \frac{(\mathrm{plane}_i) - (\mathrm{plane}_{i-1})}{(\mathrm{far\ plane}) - (\mathrm{near\ plane})} \tag{5.1}$$

In this case, the Kinect has only generated three depth levels since, due to its nominal depth range, points located very close to the camera or in the far background are not detected correctly. Therefore, to be able to compare this data with our results, we have also limited the number of planes in the point cloud obtained with our solution to three. Both figure 23 and 24 show these three clusters and the location of the three corresponding depth planes, as well as the relative normalized distances between planes. In both cases, the near plane corresponds to the first object of the three (the one closest to the camera), the middle plane includes the two remaining objects, and the far plane corresponds to the points in the wall, window, and some objects outside the window. In this example, our solution has estimated the depth planes at similar relative distances to those obtained by the kinect, with a deviation error of only a 0.67% of the depth range, which can be considered a small error. We mentioned that the point cloud stores real absolute depths so, if the estimation of the depth planes were correct, the location of the planes in figure 24 should correspond roughly with the the location of the planes in figure 22, namely, 1.5 meters, 2.75 meters, and somewhere close to 3.6 meters – since, as we mentioned, some elements within the far plane are not detected correctly and, therefore, the depth of these points is not very accurate.

Since our method estimates the locations of the three depth planes at 1.69, 2.63 and 3.84 meters, we can conclude that our solution is capable of providing a satisfactory estimation of the depth of this scene. Of course, these results cannot be extrapolated to any scene, given the specific characteristics of each video, but they show that this method can provide a reliable estimation of the depth planes within a certain margin of error.

5.5 Visual Experiment

Until now, we have shown the results achieved by our solution in quantitative terms, using as reference the measurements obtained by the Kinect. However, good numerical results do not neccesarily correlate with good visual quality. Therefore, to be able to judge the quality of the results from a qualitative perspective, it becomes neccesary to analyze whether or not these results are in fact visually appealing to human observers. For this purpose, we performed a simple experiment to assess the response of the observers to the depth levels obtained by our solution, in comparison with the depth levels obtained by the Kinect.

We used an input sequence obtained by panning the Kinect horizontally accross a room, with a total of 113 frames, to estimate the location of the depth planes in the scene. As opposed to what we did in the quantitative experiment describe in section 5.4, in this case we estimated a different number of depth planes for each method, so that we could also compare the visual quality of using different numbers of depth planes. Then, one frame was chosen from an intermediate point in the sequence (see Figure 25), and converted to 3D to perform the experiment.

Our method estimated a total of four planes, with the two middle planes located at a normalized relative distance of 0.35 and 0.97 units from the near camera plane, respectively (the near and far planes are supposed at distances 0 and 1, respectively). Since our current solution is not yet capable of segmenting the images to assign each object their corresponding depth, we made the depth allocation manually, segmenting the image into its main objects, and assigning each segment their depth according to the values obtained with our solution. We then saved the result as a grayscale image to be used as depth map. This process is illustrated in Figure 26.

The Kinect, as mentioned earlier, computes only a maximum of three depth planes; these planes were found at a distance of 2.02, 4.69 and 8.89 meters from the camera plane. The depth maps returned from the Kinect are incomplete (a lot of pixels are not properly computed), so we inpainted the remaining “holes” manually according to the values calculated by the Kinect in each image segment, creating a smoothed depth map, rather than separate depth planes as we did with the depth map calculated with our method (see Figure 27). Comparing a plane-based depth map against a smoothed one allows us to judge how much using non-smoothed transitions between the depth planes affects the perception of the final image.

Once the two depth maps were created, we made a stereoscopic 3D version of the input image from each map, using a horizontal displacement filter (keeping the same settings for both methods). These two 3D images were then shown to a set of ten observers, who were asked to compare them in terms of visual comfort, natural depth and overall visual quality.

After analyzing the results of the experiment, we can conclude that the observers tended to feel more comfortable looking at the kinect image. Many of them affirmed that the abrupt depth changes between depth levels in the image obtained with our method makes it look unrealistic.

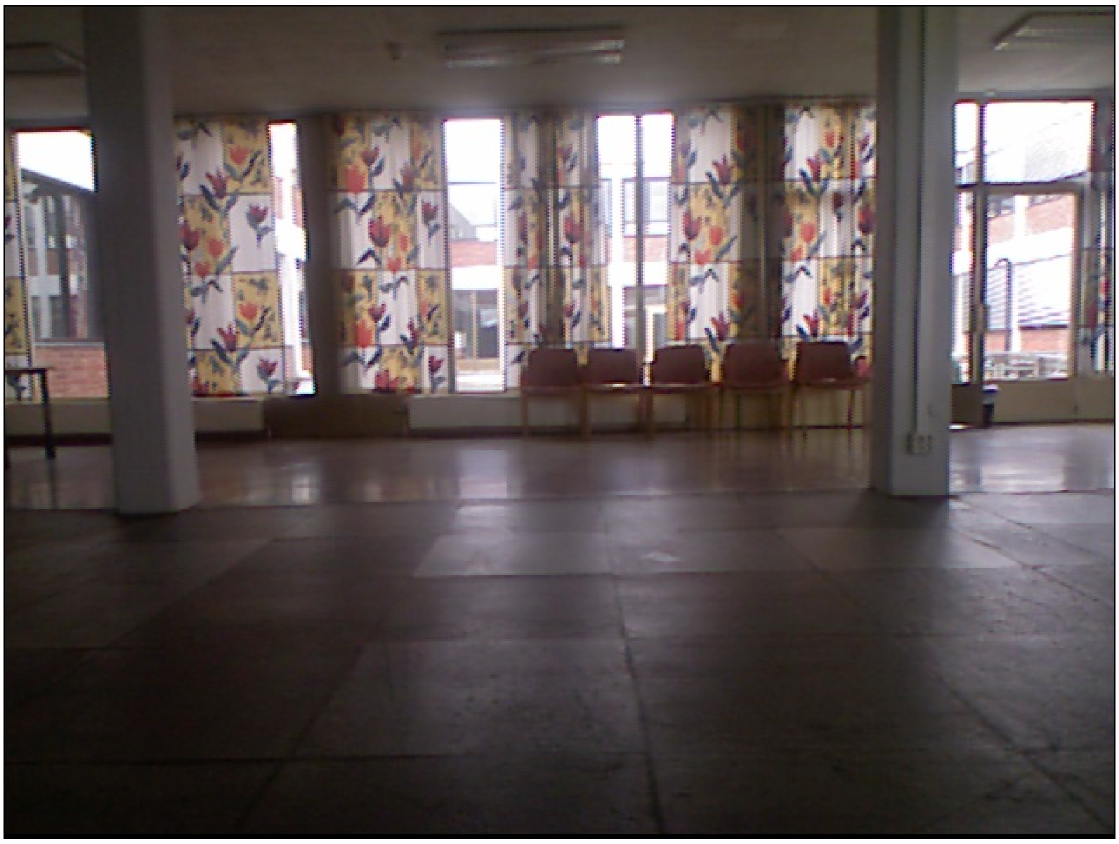


Figure 25: Input image used in the visual experiment.

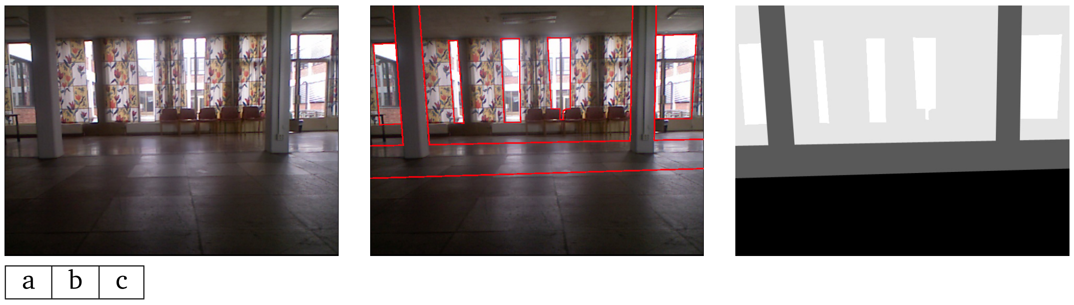
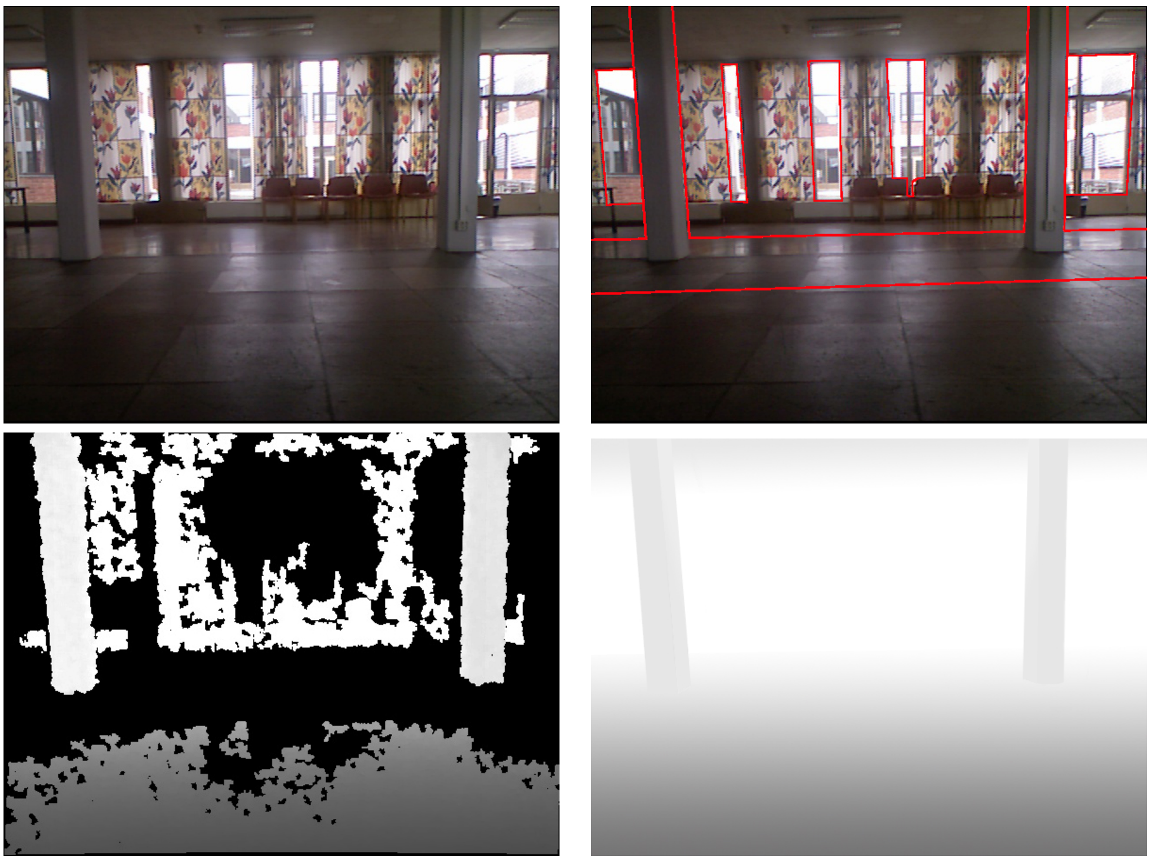


Figure 26: Stages of the manual depth map creation process from the values computed with the proposed method for depth estimation. From left to right: a) original image, b) segmented image and c) final depth map. Each gray level in the depth map corresponds to a different depth plane.



a	b
c	d

Figure 27: Stages of the depth map creation process from the values computed by the kinect. a) original image, b) segmented image, c) kinect depth map and d) modified final depth map according to the depth levels found by the Kinect. Each gray level in the depth map corresponds to a different depth plane.

On the other hand, all the observers also agreed that the image obtained using the kinect depth map looks rather flat, whereas the image obtained with our method shows a more realistic distribution of depth levels. As we will comment later on this report (see Chapter 7), this issue could be solve, for example, by increasing the number of estimated depth planes. Nonetheless, the fact that the observers found the depth levels estimated by our method more visually appealing, demonstrates that our solutions does indeed a good job in estimating such planes.

In terms of global visual quality, the image converted with the Kinect depth map was given an average score of 6.6, with a standard deviation of 1.7764, whereas the one converted with our method was given an average score of 5.6, with a standard deviation of 1.0750. Although our method received a lower average score, we consider that the difference is within an acceptable interval, considering that the goal of our method is not to obtain better results than those obtained manually, but to provide a fast automatic method which can obtain results that are visually close to the ones obtained manually.

6 Discussion

Finding the 3D structure of the elements in a video has always been an important but challenging task in computer vision. Computer vision has been used for a long time in many scientific areas such as robotics, medicine or biology; however, the technological advances and cost reductions undergone in recent years, especially in the area of digital cameras, graphic processors and displays, has opened new application areas for this technology, making it available to the general public as well, mainly for entertainment applications such as video games, or 3D films. As it is usually the case with fast growing markets such as this, companies are trying to get ahead of their competition by investing a lot of of effort and resources to develop new methods which, in turn, is impulsing the amount of research activity in those new areas. Many new 3D films are being made every year but, due to the high cost of making a film in 3D, many production companies choose to create their films in the traditional way and then convert them into 2D, or simply to convert already existing 2D films.

There are many pieces of software capable of converting a film to 3D, using different techniques. For example, Ward et al.[9] present a method that finds which objects are on top of which in the image, calculating the neccesary amount of disparity, and creating the two stereo frames required for stereoscopic displays to show the correct depth. It is common for production houses to develop their own 3D conversion software solutions, and customize the methods for their specific purposes; nonetheless, all the existing solutions share a common drawback, that is, they all require an important amount of manual interaction, which largely increases the conversion time.

Additionally, the goal of most of the solutions mentioned above is to produce results that are visually appealing, without taking into consideration the real depth of the scene – that is, the real location of the scene objects when the film was made. We showed in Section 2.2 that true-depth displays are improving considerably over the last years and, at some point in the future, they could be replacing the current stereoscopic technology. Therefore, it becomes neccesary to find solutions that, do not only add depth, but are capable of fully interpreting a scene and finding the right structure that was present at the time of filming. Otherwise, although the results produced with the current methods may be visually appealing when viewed on the current stereoscopic displays, they might not look right on future true-depth displays.

During the course of this report we have described some of the most common mechanisms that are currently used for the 3D reconstruction of video, and defined some important concepts in the areas of filmmaking and video reconstruction, in order to explain the choices that we have made for our solution. Additionally, we have proposed a method to combine several individual techniques, to create a workflow for depth estimation that can be included as part of the 3D reconstruction process. The final workflow has been the result of an exhaustive trial-and-error research, where several different methods have been implemented and tested, and many others had to be abandoned halfway through or discarded after a full implementation, due to several reasons such as bad performance or incompatibilities with some of the other methods being used.

Some state-of-the-art methods were originally considered as part of our solution, but the lack of an existing implementation and the lack of time for us to implement our own, prevented us from using them. In research projects such as this, it is common to modify the initial approach, as newer and better methods are found during the research process. In this sense, the problem of 3D estimation was approached in several different ways along the project, as they were found unfeasible. Some of the implemented methods may not be the best choice individually but, when combined, they have been found to yield the best results of all the tests we have performed.

It must be noted that the goal of this research is not to find an universal method the can convert a 2D film to 3D. Instead, we focus on finding an automatic method that can successfully estimate the depth of a scene, in a way such that it resembles the original depth of the scene, and the objects in it maintain the same relative positions and distances as in the original scene when the film was made. The converted films are typically shown in displays of varying sizes, so computing the real depth values is not important; instead, we estimate the relative distances between each depth plane, so that the structure is conserved regardless of the size of the scene.

An important issue for this project was finding a way to assess the quality of the results, given the lack of ground truth data to match the numerical results against. To overcome this problem, we have presented an innovative approach in this field, which consists of using the Microsoft's Kinect depth sensor to evaluate the performance of our method. Two different experiments were conducted to assess the results, both quantitatively and qualitatively, in comparison with those obtained by the Kinect. Both experiments yielded acceptable results that were within a reasonable interval, as discussed in Section 5.4.

In the next section we will comment some possible ways in which the proposed solution could be improved, and we will also discuss some future directions in the area of 3D film conversion.

7 Future directions

The depth estimation method that we propose in this research project has been shown to produce satisfactory results in reconstructing the depth from a film scene. This solution could be included as part of a film 3D conversion pipeline, aiding the reconstruction process in estimating the depth of each of the objects. Nonetheless, there are still many modifications that could be added to this solution to improve the quality of the results.

The proposed method is already capable of estimating succesfully the location of each of the depth planes in selected scenes. However, the use of Bundler to generate the initial point cloud required by our method transforms the process into a black box that reduces our flexibility to control the features that get selected from the input frames and, in turn, reduces the control that we have over the density of the computed point cloud. Adding our own methods for camera calibration, feature matching, point registration and point cloud computation would provide much more control over the results. Other possible improvements would be the addition of some type of object segmentation algorithm to the point cloud expansion process, to improve the quality of the pixel-to-feature mappings (see Section 4.3.4), or the addition of a keyframe selection algorithm to reduce the amount of work required to generate the initial point cloud, improving the overall performance of the method (see Section 2.3.4).

In Section 5.1 we described the most important characteristics that a scene must have for our method to obtain good results. This is usually an important task, because the selected scene will determine the quality of the results. Therefore, we have studied the possibility of adding an additional module that can estimate the quality of the input scene. Similarly, we could also develop a piece of software to automatically select, from a video library, those scenes for which our solution will perform well.

Although the solution proposed in this work is still far from being perfect, we believe that the techniques presented in this report, as well as the proposed workflow, can be of great interest as a starting point for future researchers who are interested in performing similar tasks. Moreover, we are planning on making this project publicly available through open source license on GitHub [1], so that everyone who is interested can continue our work by adding new improvements, hoping that this decision might help, eventually, to render our method into a practical solution for depth estimation.

[1]GitHub - https://github.com/

Bibliography

[1] Nvidia Corporation. January 2009. Nvidia announces 3d vision-the world's first high-definition 3d stereo solution for the home. Press Release.

[2] Samsung Electronics America, Inc. January 2009. Samsung unveils 22-inch, 120hz 3d-monitor compatible with nvidia's® geforce® 3d vision. Press Release.

[3] Hartley, A. July 2011. In depth: The future of 3d tv content. 3DRadar.

[4] Han, F. January 2012. Ces supersession - spotlight on 3d content. NetShelter.com.

[5] October 2008. 3d movies: Adding depth or falling flat? *Knowledge@Wharton*.

[6] Jeppsen, M. March 2009. Does 3d enhance or hinder the moviegoing experience? FreshDV.com.

[7] Van Pernis, A. P. & DeJohn, M. S. March 2008. Dimensionalization: converting 2d films to 3d. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 6803 of *Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference*.

[8] Bertalmio, M., Sapiro, G., Caselles, V., & Ballester, C. 2000. Image inpainting. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, 417–424, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

[9] Ward, B., Kang, S. B., & Bennett, E. jan.-feb. 2011. Depth director: A system for adding depth to movies. *Computer Graphics and Applications, IEEE*, 31(1), 36 –48.

[10] Battiato, S., Capra, A., Curti, S., & La Cascia, M. 2004. 3d stereoscopic image pairs by depth-map generation. In *3D Data Processing, Visualization and Transmission, 2004. 3DPVT 2004. Proceedings. 2nd International Symposium on*, 124–131.

[11] Alvarez, J. M., Gevers, T., & Lopez, A. 2010. 3d scene priors for road detection. In *IEEE Conference on Computer Vision and Pattern Recognition*, 57–64.

[12] Battiato, S., Curti, S., Cascia, M. L., Tortora, M., & Scordato, E. Depth-map generation by image classification.

[13] Dalmia, A. K. & Trivedi, M. 1996. Depth extraction using a single moving camera: an integration of depth from motion and depth from stereo. *Machine Vision and Applications*, 9, 43–55. 10.1007/BF01214359.

[14] Benini, S., Canini, L., & Leonardi, R. 19-23 July 2010. Estimating cinematographic scene depth in movie shots. In *In Proceedings of the IEEE International Conference on Multimedia & Expo (ICME)*, Singapore.

[15] Steinman, S., Steinman, B., & Garzia, R. 2000. *Foundations of binocular vision: a clinical perspective*. McGraw-Hill.

[16] Mcallister, D. F. 2006. Display technology: Stereo & 3d display technologies. *In: Hornak, J. 2005. Wiley Encyclopaedia on Imaging Science and Technology. Wiley Interscience: Bognor Regis, West*, 1327–1344.

[17] Teittinen, M. 1993. Depth cues in the human visual system. *see http://www. hitl. washington. edu/scivw/EVE/III. A*, 1.

[18] Sỳkora, D., Sedlacek, D., Jinchao, S., Dingliana, J., & Collins, S. 2010. Adding depth to cartoons using sparse depth (in) equalities. In *Computer Graphics Forum*, volume 29, 615–623. Wiley Online Library.

[19] Hoffman, D. M., Girshick, A. R., Akeley, K., & Banks, M. S. 2008. Vergence-accommodation conflicts hinder visual performance and cause visual fatigue. *Journal of vision*, 8(3).

[20] Hecht, J. Feb 2011. 3-d tv and movies: Exploring the hangover effect. *Opt. Photon. News*, 22(2), 20–27.

[21] Child, B. August 2011. 3d no better than 2d and gives filmgoers headaches, claims study. *The Guardian*.

[22] Long, D. April 2010. Science's health concerns over 3d films. *PC & Tech Authority*.

[23] Turner, T. H. 2010. Case study: Beauty and the beast 3d: benefits of 3d viewing for 2d to 3d conversion. In *Proceedings SPIE 7524, 75240B*, Walt Disney Animation Studios, 500 South Buena Vista Street, Burbank, CA, USA 91521.

[24] Jr., V. B., Smithwick, Q., Barabas, J., & Smalley, D. 2005. Is 3-d tv preparing the way for holographic tv?

[25] Razutis, A. 2007. More 3d and '4d' ('can you feel it?').

[26] V. M. Bove, J. 2011. Live holographic tv: From misconceptions to engineering. In *Proc. 2011 SMPTE International Conference on Stereoscopic 3D for Media and Entertainment*.

[27] Haussler, R., Schwerdtner, A., & Leister, N. 2008. Large holographic displays as an alternative to stereoscopic displays. *Proceedings of SPIE*, 6803(1), 68030M–68030M–9.

[28] Durrant-Whyte, H. & Bailey, T. 2006. Simultaneous localisation and mapping (slam): Part i the essential algorithms. *IEEE Robotics and Automation Magazine*, 2, 2006.

[29] Bailey, T. & Durrant-whyte, H. 2006. Simultaneous localisation and mapping (slam): Part ii state of the art. *Computational Complexity*, 13(3), 1–10.

[30] Davison, A. J., Reid, I. D., Molton, N. D., & Stasse, O. June 2007. Monoslam: Real-time single camera slam. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(6), 1052–1067.

[31] Pan, Q., Reitmayr, G., & Drummond, T. September 2009. ProFORMA: Probabilistic Feature-based On-line Rapid Model Acquisition. In *Proc. 20th British Machine Vision Conference (BMVC)*, London.

[32] Klein, G. & Murray, D. November 2007. Parallel tracking and mapping for small AR workspaces. In *Proc. Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07)*, Nara, Japan.

[33] Triggs, B., McLauchlan, P. F., Hartley, R. I., & Fitzgibbon, A. W. 2000. Bundle adjustment - a modern synthesis. In *Proceedings of the International Workshop on Vision Algorithms: Theory and Practice*, ICCV '99, 298–372, London, UK, UK. Springer-Verlag.

[34] Sourimant, G. Depth maps estimation and use for 3dtv. Technical Report RT-0379, INRIA, February 2010.

[35] Sontag, S. 1966. Film and theatre. *The Tulane Drama Review*, 11(1), 24–37.

[36] Kurowska, M. *Peter Shaffer's play Amadeus and its film adaptation by Milos Forman*. Diplomarbeit, Johannes Gutenberg-Universitat Mainz, 1998. p.17.

[37] Brandt, C. June 2011. Theater, movies - what's the difference? */One/*, June.

[38] Van Sijll, J. 2005. *Cinematic storytelling: The 100 most powerful film conventions every filmmaker must know*. Michael Wiese.

[39] Prunes, M., Raine, M., & Litch, M. August 2002. Film analysis guide. Online Manual.

[40] Fowler, M. S. November 2003. Animation layout: Overlay (ol) and overlay/underlay (ol/ul). Online Magazine.

[41] Kaufman, D. March 2012. Stereo d converts abraham lincoln: Vampire hunter to 3d. Online Magazine.

[42] Maupin, T. 2011. Artistic uses of 3d in film. Stereolith.com.

[43] Pike, C. November 2010. Using depth of field for storytelling. DSLR Video Shooter.

[44] Gardner, B. February 2009. Perception and the art of 3d storytelling. Creative COW magazine: Stereoscopic 3D.

[45] McGee, R. January 2012. The dimensionalization of "the lion king" vs "beauty and the beast". 3D IU Students.

[46] Graham, B. August 2011. The lion king 3d conversion images show off depth of field; stereographer robert neuman talks about the process. Collider.com.

[47] Engber, D. September 2011. Who killed 3-d?: A box-office whodunit. Slate.

[48] Rusu, R. B. & Cousins, S. 2011 2011. 3d is here: Point cloud library (pcl). In *International Conference on Robotics and Automation*, Shanghai, China.

[49] Kien, D. T. A review of 3d reconstruction from video sequences - mediamill3d technical reports series. Technical Report 1.107, Intelligent Sensory Information Systems, Department of Computer Science, University of Amsterdam, The Netherlands, 2005.

[50] Donate, A. & Liu, X. june 2010. 3d structure estimation from monocular video clips. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*, 17 –24.

[51] Hartley, R. I. & Zisserman, A. 2004. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition.

[52] Snavely, N. 2008. Bundler user's manual.

[53] Aghajan, H. & Cavallaro, A. 2009. *Multi-Camera Networks: Principles and Applications*. Academic Press.

[54] Pollefeys, M., Koch, R., Vergauwen, M., & Gool, L. V. 1998. Metric 3d surface reconstruction from uncalibrated image sequences. In *3D STRUCTURE FROM MULTIPLE IMAGES OF LARGE SCALE ENVIRONMENTS. LNCS SERIES*, 138–153. Springer-Verlag.

[55] Pollefeys, M., Gool, L. V., Vergauwen, M., Cornelis, K., Verbiest, F., & Tops, J. 2002. Video-to-3d. In *Proc. Photogrammetric Computer Vision 2002 (ISPRS Commission III Symposium), International Archive of Photogrammetry and Remote Sensing.*

[56] Pollefeys, M. 2004. Visual 3d modeling from images. In *VMV*, 3.

[57] Koch, R., Pollefeys, M., & Van Gool, L. 1998. Multi viewpoint stereo from uncalibrated video sequences. *Computer Vision-ECCV'98*, 55–71.

[58] Koch, R., Pollefeys, M., & Gool, L. V. 1999. Realistic 3-d scene modeling from uncalibrated image sequences. In *ICIP'99, Kobe: Japan*, 500–504.

[59] Koch, R., f. Evers-senne, J., m. Frahm, J., & Koeser, K. 2005. 3d reconstruction and rendering from image sequences. In *In: WIAMIS 05*.

[60] Snavely, N., Seitz, S. M., & Szeliski, R. 2006. Photo tourism: Exploring photo collections in 3d. In *SIGGRAPH Conference Proceedings*, 835–846, New York, NY, USA. ACM Press.

[61] Lowe, D. 1999. Object recognition from local scale-invariant features. 1150–1157.

[62] Fischler, M. A. & Bolles, R. C. 1981. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6), 381–395.

[63] Hirschmuller, H. February 2008. Stereo processing by semiglobal matching and mutual information. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(2), 328–341.

[64] Boykov, Y., Veksler, O., & Zabih, R. November 2001. Fast approximate energy minimization via graph cuts. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(11), 1222–1239.

[65] Scharstein, D. & Szeliski, R. 2001. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47, 7–42.

[66] Sobel, I. & Feldman, G. 1973. A 3x3 isotropic gradient operator for image processing. *Pattern Classification and Scene Analysis*, 271–272.

[67] Ozden, K. E., Schindler, K., & Gool, L. J. V. 2007. Simultaneous segmentation and 3d reconstruction of monocular image sequences. In *ICCV*, 1–8.

[68] Oehler, B., Stückler, J., Welle, J., Schulz, D., & Behnke, S. 2011. Efficient multi-resolution plane segmentation of 3d point clouds. In *ICIRA (2)*, 145–156.

[69] Borrmann, D., Elseberg, J., Lingemann, K., & Nüchter, A. March 2011. The 3d hough transform for plane detection in point clouds: A review and a new accumulator design. *3D Res.*, 2(2), 32:1–32:13.

[70] Wahl, R., Guthe, M., & Klein, R. October 2005. Identifying planes in point-clouds for efficient hybrid rendering. In *The 13th Pacific Conference on Computer Graphics and Applications*.

[71] Yang, M. Y. & Foerstner, W. Plane detection in point cloud data. Technical Report TR-IGG-P-2010-01, Department of Photogrammetry, University of Bonn, 2010.

[72] Bauer, J., Karner, K., Schindler, K., Klaus, A., & Zach, C. 2003. Segmentation of building models from dense 3d point-clouds. In *In 27th Workshop of the Austrian Association for Pattern Recognition*, 253–259.

[73] Rice, J. 2006. *Mathematical statistics and data analysis*. Thomson Learning.

[74] Botev, Z., Grotowski, J., & Kroese, D. 2010. Kernel density estimation via diffusion. *The Annals of Statistics*, 38(5), 2916–2957.

[75] Frank, I. & Todeschini, R. 1994. *The data analysis handbook*. Number 14 in Data Handling in Science and Technology. Elsevier Amsterdam.

[76] Jain, A. & Dubes, R. 1988. *Algorithms for clustering data*. Prentice-Hall, Inc.

[77] Schwaiger, M. & Opitz, O. 2003. *Exploratory data analysis in empirical research: proceedings of the 25th Annual Conference of the Gesellschaft für Klassifikation eV, University of Munich, March 14-16, 2001*, volume 25. Springer Verlag.

[78] Zuliani, M. January 2012. Ransac for dummies.

[79] Ahn, S. 2008. Geometric fitting of parametric curves and surfaces. *Journal of Information Processing Systems*, 2.

[80] Danuser, G. & Stricker, M. March 1998. Parametric model fitting: From inlier characterization to outlier detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(3), 263–280.

[81] Andersen, M., Jensen, T., Lisouski, P., Mortensen, A., Hansen, M., Gregersen, T., & Ahrendt, P. Kinect depth sensor evaluation for computer vision applications. Technical report ECE-TR-6, Department of Engineering, Aarhus University (Denmark), February 2012.

[82] Crawford, S. July 2010. How microsoft kinect works. HowStuffWorks.com.

[83] Crock, N. March 2011. Kinect depth vs. actual distance. Mathnathan.com.

A Source code

Listing A.1: Depth clustering algorithm by clustering (Matlab)

```matlab
function C = clusterValues(data, maxClusters, T)
%% This function groups the values in the 'data' vector into clusters
 % according to the squared Euclidean distance criterium.
 % If two clusters are closer than a certain threshold T,
 % they are merged into one and a new clustering round is done
 % to redistribute the values from the merged clusters.
 %
 % INPUT:
 %              data: the data vector.
 %       maxClusters: maximum number of clusters to find (depending on T,
 %                    the resulting clusters may be less if they are too
 %                    close).
 %                 T: Threshold which determines the minimum distance
 %                    allowed between two clusters. It is specified as a
 %                    percentage of the range (max(data)-min(data)).
 %
 % OUTPUT:
 %                 C: a vector containing the centroid of each cluster.
 %
 %
 % Example:     C = clusterValues(data, 5, 4);
 %
 %


     if nargin < 2
         maxClusters = 5;
         T = 4;
     end
     if nargin < 3
         T = 4;
     end

     numClusters = maxClusters;

     T = T / 100;

     [IDX,C]=kmeans(data, numClusters, 'distance', 'sqEuclidean',...
     'emptyaction', 'singleton', 'start', 'uniform', 'replicates', 5);
     C=sort(C);

     range = max(data) - min(data);

    % Merge planes that are very close to each other.
     j=1;
     for i=1:numel(C)
         if ((i==1) || ((i>1) && (abs(C(i)-C2(j-1))>(T*range))))
             C2(j)=C(i);
             j = j+1;
         end
     end

     % Re-cluster the data according to the new locations.
     if (numel(C2)<numel(C))
         numClusters = numel(C2);
         [IDX,C]=kmeans(data, numClusters, 'distance', 'sqEuclidean',...
           'emptyaction', 'singleton', 'start', C2');
         C=sort(C);
     end
 end
```

Listing A.2: Depth clustering algorithm by Kernel Density Estimation(Matlab) – Unused

```
function planes = getPlaneLocations (x, n)
    if nargin<2
        n=2^12;
    end

    [h, fhat, xgrid] = kde(x, n);
    [pks,locs] = findpeaks(fhat);

    planes = xgrid(locs);
end
```

Listing A.3: Depth planes estimation (C++)

```
/**
 * @file scan_image.cpp
 * @author Victor Medina
 * @version 1.0
 *
 * @section DESCRIPTION
 *
 * This program reads an input pointcloud from a PLY file, and assigns each
 * vertex to a corresponding depth plane based on the value of its z-coordinate.
 * Each of the generated point clouds are saved onto a ply file for future use.
 * It's also possible to display the resulting point clouds in a built-in PCL
 * viewer.
 */

// C++ libraries
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

// OpenCV libraries
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

// PCL libraries
#include <pcl/point_types.h>
#include <pcl/ModelCoefficients.h>
#include <pcl/common/common_headers.h>
#include <pcl/console/parse.h>
#include <pcl/features/normal_3d.h>
#include <pcl/filters/extract_indices.h>
#include <pcl/filters/passthrough.h>
#include <pcl/filters/project_inliers.h>
#include <pcl/filters/voxel_grid.h>
```

```
#include <pcl/io/pcd_io.h>
#include <pcl/io/ply_io.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/sample_consensus/ransac.h>
#include <pcl/sample_consensus/sac_model_plane.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/surface/concave_hull.h>
#include <pcl/visualization/cloud_viewer.h>
#include <pcl/visualization/pcl_visualizer.h>

// 3rd Party libraries
#include <boost/thread/thread.hpp>
#include <flann/flann.h>

// Custom libraries
#include "libmtools.h"
#include "BundlerReader.h"

// Constants
#define INF 99999.0
#define MAX_NUM_PLANES 5

// Global Variables
std::vector<pcl::ModelCoefficients> DEPTH_PLANES;
std::vector<double> DEPTH_VALUES;
int NUM_PLANES;
BundlerScene SceneInfo;


/** \brief Creates a depthmap image from a point cloud.
 *
 * Creates a grayscale depth map image where the maximum gray value
 * is the maximum depth in the point cloud, and the minimum is the minimum depth.
 * Each pixel's gray value is computed according to the corresponding normalized
 * depth within the resulting range.
 *
 * \param[in] p_src a pointer to the point cloud.
 * \param[out] the depth map is stored in an image file named "depthmap.jpg".
 */

void pclToDepthmap (pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud)
{
  cv::Mat depthmap = cv::Mat(cloud->height, cloud->width, CV_8S);
  int a,b;

  unsigned char *data= reinterpret_cast<unsigned char *>(depthmap.data);

  for (int i=0; i<cloud->points.size(); i++)
  {
    int x = cloud->points[i].x;
    int y = cloud->points[i].y;
    double depth = cloud->points[i].z;
    depthmap(cv::Range(x,x),cv::Range(y,y)) = depth;
  }

  cv::imwrite("depthmap.jpg",depthmap);
}

/** \brief Maps a 3D point to its corresponding  pixel coordinates.
 *
 * Maps a point cloud point in 3D coordinates to its corresponding 2D pixel coordinates
 * using the camera matrices computed during the camera calibration process. Since we
 * want to keep the depth information, the resulting point is a 3D point where the
    first
 * two coordinates are indicated in pixels coordinates, and the third coordinate is the
 * original depth of the point.
 *
 * \param[in] p_src the point in 3D coordinates.
 * \param[out] point a 3D point where the first two coordinates are indicated in pixels
 *                  coordinates, and the third coordinate is the original depth of the
    point.
 */

cv::Point3d map3Dto2Dcoord (cv::Point3d p_src)
{
  cv::Point2d p;
  cv::Point3d P;
  cv::Point3d point;
  double tmp[3];

  double R[3][3];
  double t[3];
  double f;
  double k1, k2, r;
  int activeView;
  BundlerCamera cam;

  activeView = SceneInfo.getActiveView();
  cam = SceneInfo.getCamera(activeView);

  cam.getRotation(R);
  cam.getTranslation(t);

  f = cam.getFocalLength();
  k1 = cam.getRadialDistortion(1);
  k2 = cam.getRadialDistortion(2);


  //P = (R*X) + t
  tmp[0] = (R[0][0]*p_src.x + R[0][1]*p_src.y + R[0][2]*p_src.z) + t[0];
  tmp[1] = (R[1][0]*p_src.x + R[1][1]*p_src.y + R[1][2]*p_src.z) + t[1];
  tmp[2] = (R[2][0]*p_src.x + R[2][1]*p_src.y + R[2][2]*p_src.z) + t[2];

  //p = -P./P[3]
  p = cv::Point2d(-tmp[0]/tmp[2] , -tmp[1]/tmp[2]);


  r = 1.0 + (k1 * pow(cv::norm(p),2)) + (k2 * pow(cv::norm(p),4));

  p = f * r * p;

  // Keep the original depth.
  point = cv::Point3d(p.x, p.y, tmp[2]);

  return point;
}

/** \brief Finds the closest point in a point cloud to an image pixel.
 *
 * Given a 2D point in pixel coordinates, it finds the closest point in a
 * given point cloud using the Euclidean distance. The point cloud must contain
 * at least a subset of the pixels in the image, so that a match can be found.
 *
 * \param[in] p_src the 2D point in pixel coordinates for which we want to find
 *                  the closest point in the point cloud.
 * \param[in] cloud a pointer to the point cloud where we want to search.
 * \param[out] closest_point 3D coordinates of the closest point in the point cloud.
 */

cv::Point3d findCorrespondingPoint(cv::Point p_src, pcl::PointCloud<pcl::PointXYZRGB>::
    Ptr cloud)
{
  pcl::PointXYZRGB tmp;
  double dist;
  double min_dist = 99999999.999999;
  cv::Point3d p, closest_point;

  for (int i=0; i<cloud->points.size(); i++)
  {
    tmp = cloud->points[i];
    p = map3Dto2Dcoord(cv::Point3d(tmp.x,tmp.y,tmp.z));

    dist = sqrt(std::pow(p.x - p_src.x, 2) + std::pow(p.y - p_src.y, 2));

    if (dist < min_dist)
    {
      closest_point = p;
      min_dist = dist;
    }
  }

  return closest_point;
}

/** \brief Converts a non-rgb point cloud to an rgb one.
 *
 * Converts a point cloud which does not contain color information about the vertices
 * to an rgb point cloud where each point is black. This is used so that all the
    processed point
 * clouds contain color information, to avoid incompatibilities issues when processing
    different
 * types of point clouds.
 *
 * \param[in] cloud a pointer to the non-rgb point cloud.
 * \param[out] new_cloud a pointer to the new rgb point cloud.
 */

pcl::PointCloud<pcl::PointXYZRGB>::Ptr convertCloudToRGB (pcl::PointCloud<pcl::PointXYZ
    >::Ptr cloud)
{
  pcl::PointCloud<pcl::PointXYZRGB>::Ptr new_cloud(new pcl::PointCloud<pcl::PointXYZRGB
    >);

  new_cloud->width  = cloud->width;
  new_cloud->height = cloud->height;
  new_cloud->points.resize (new_cloud->width * new_cloud->height);

  for (int i=0; i<cloud->points.size(); i++)
  {
    new_cloud->points[i].x = cloud->points[i].x;
    new_cloud->points[i].y = cloud->points[i].y;
    new_cloud->points[i].z = cloud->points[i].z;

    new_cloud->points[i].r = 0;
    new_cloud->points[i].g = 0;
    new_cloud->points[i].b = 0;
  }

  return new_cloud;
}

/** \brief Maps the pixels in an image to their corresponding depth value.
 *
 * Converts a scarce point cloud into a dense one from a given image. For each pixel
 * in the image, it finds the closest point in the scarce point cloud, and assigns
 * its depth to that pixel. Then creates a new point cloud containing all the pixels
 * from the image with their corresponding 3D coordinates.
 *
 * \param[in] src the reference image whose pixels we want to map. It has to belong to
    the
 *                sequence that generated the point cloud.
 * \param[in] cloud a pointer to the scarce point cloud that contains the refence
    points
 *                  for the mapping.
 * \param[out] new_cloud a pointer to the point cloud where we want to store the
    resulting
 *                       dense point cloud.
 */

pcl::PointCloud<pcl::PointXYZRGB>::Ptr mapImage (cv::Mat src , pcl::PointCloud<pcl::
    PointXYZRGB>::Ptr cloud)
{
  cv::Point2d pixel2d_new, pixel2d_orig;
  cv::Point3d corr_point;
  pcl::PointXYZRGB pixel3d;
  double pixel_depth;
  pcl::PointCloud<pcl::PointXYZRGB>::Ptr new_cloud(new pcl::PointCloud<pcl::PointXYZRGB
    >);

  new_cloud->width  = src.rows * src.cols;
  new_cloud->height = 1;
  new_cloud->points.resize (new_cloud->width * new_cloud->height);

  int i=0;
  int w= src.cols;
  int h = src.rows;

  unsigned char label1, label2=255;

  for (int u=-(w/2); u<=(w/2)-1; u++)
  {
    for (int v=(-(h/2))+1; v<=(h/2)-1; v++)
    {
      int a,b;
      a = u + w/2.0;
      b = abs(v - h/2.0);

      pixel2d_new = cv::Point(u,v);
      const cv::Vec3b& bgr = src.at<cv::Vec3b>(b,a);

      if ((u==-(w/2) && v==-(h/2)+1) || label1 != label2)
      {
        corr_point = findCorrespondingPoint( pixel2d_new , cloud );
        pixel_depth = corr_point.z*100.0;
      }

      pixel3d = pcl::PointXYZRGB(bgr[2],bgr[1],bgr[0]);
      pixel3d.x = pixel2d_new.x;
      pixel3d.y = pixel2d_new.y;
      pixel3d.z = pixel_depth;

      new_cloud->points[i++] = pixel3d;
    }
  }

  return new_cloud;
}

/** \brief Merges all the point clouds in the input vector into one.
 *
 * Receives a vector of point clouds, and merges their content into one.
 *
 * \param[in] cloudList the vector of pointers to the point clouds that we want to
     merge.
 * \param[in] cloud a pointer to the point cloud where we want to store the merged
     point cloud.
 */

void mergeClouds (std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr> cloudList, pcl::
    PointCloud<pcl::PointXYZRGB> &new_cloud)
{
  if (cloudList.size()>0)
  {
    new_cloud  = *cloudList[0];
    for (int i=1; i < cloudList.size(); i++)
      new_cloud  += *cloudList[i];
  }
}

/** \brief Removes duplicates to reduce the amount of points in the point cloud.
 *
 * Filters the point cloud with a VoxelGrid of a very small leaf size, so that only
 * duplicated values are removed from the cloud. ASince we are working with feature
     point
 * clouds, we are not interested in reducing the resolution and, in fact, we should try
      to
 * keep as many as possible.
 *
 * \param[in] cloud a pointer to the point cloud that we want to modify.
 */

void removeDuplicates(pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud)
{
  sensor_msgs::PointCloud2::Ptr cloud2 (new sensor_msgs::PointCloud2 ());
  sensor_msgs::PointCloud2::Ptr cloud_filtered (new sensor_msgs::PointCloud2 ());

  pcl::toROSMsg(*cloud, *cloud2);

  // Create the filtering object
  pcl::VoxelGrid<sensor_msgs::PointCloud2> sor;
  sor.setInputCloud (cloud2);
  sor.setLeafSize (0.00001f, 0.00001f, 0.00001f);
  sor.filter (*cloud_filtered);

  pcl::fromROSMsg(*cloud_filtered, *cloud);
}

/** \brief Stores the z coordinate of each point in the point cloud for easier access.
 *
 * Copies the value of the z-coordinate of each PointXYZ in the point cloud
 * to the global vector DEPTH_VALUES, so that C++ built-in routines can be used
 * to perform operations such as sorting, comparing or adding. This vector is
 * sorted in ascending order, so the positions will not match those of the points in
 * the original cloud.
 *
 * \param[in] cloud a pointer to the point cloud whose points' depth we want to store.
 * \param[out] DEPTH_VALUES The content of the global variable DEPTH_VALUES is modified
     .
 */

void extractDepthValues(pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud)
{
  pcl::PointCloud<pcl::PointXYZRGB>::iterator it;

  for(it = cloud->points.begin(); it != cloud->points.end(); it++)
    DEPTH_VALUES.push_back(it->z);

  sort(DEPTH_VALUES.begin(), DEPTH_VALUES.end());
}

/** \brief Finds the highest z-coordinate in the point cloud.
 *
 * Finds the z-coordinate of the furthest point from the camera in the point cloud.
 *
 * Returns:
 *  * The z-coordinate of the furthest point in the point cloud.
 * \param[in] cloud a pointer to the point cloud whose furthest point we want to find.
 */

double max_depth(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud)
{
  return DEPTH_VALUES[DEPTH_VALUES.size()-1];
}

double max_depth(pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud)
{
  return DEPTH_VALUES[DEPTH_VALUES.size()-1];
}

/** \brief Finds the closest z-coordinate in the point cloud.
 *
 * Finds the z-coordinate of the closest point to the camera in the point cloud.
 *
 * Returns:
 *  * The z-coordinate of the closest point in the point cloud.
 * \param[in] cloud a pointer to the point cloud whose closest point we want to find.
 */

double min_depth(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud)
{
  return DEPTH_VALUES[0];
}

double min_depth(pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud)
{
  return DEPTH_VALUES[0];
}

/** \brief Adds the z-coordinate of all the points in the point cloud.
 *
 * Adds the z-coordinate of all the points in the point cloud. This is used only
 * as a middle step required for other operations.
 *
 * Returns:
 *   * The addition of the z-coordinate of all the points in the point cloud.
 * \param[in] cloud a pointer to the point cloud whose points we want to add up.
 */

double sum (pcl::PointCloud<pcl::PointXYZ>::Ptr cloud)
{
  double sum = 0.0;

  for (int i = 0; i < cloud->points.size(); i++)
    sum += cloud->points[i].z;

  return sum;
}

double sum (pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud)
{
  double sum = 0.0;

  for (int i = 0; i < cloud->points.size(); i++)
    sum += cloud->points[i].z;

  return sum;
}

/** \brief Finds the average depth of the points in the point cloud.
 *
 * Finds the average depth of the points in the point cloud, using the
 * z-coordinate as the depth.
 *
 * Returns:
 *   * The average depth of the points in the point cloud.
 * \param[in] cloud a pointer to the point cloud whose depth we want to average.
 */

double average (pcl::PointCloud<pcl::PointXYZ>::Ptr cloud)
{
  return sum(cloud)/cloud->points.size();
}

double average (pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud)
{
  return sum(cloud)/cloud->points.size();
}

/** \brief Finds the standard deviation of the depth of the points in the point cloud.
 *
 * Finds the standard deviation of the depth of the points in the point cloud,
 * using the z-coordinate as the depth. It uses the average depth previously calculated
     with the
 * average function, in order to avoid duplicating calculations since, in many cases,
     both values
 * are needed.
 *
 * Returns:
 *   * The standard deviation of the depth of the points in the point cloud.
 * \param[in] cloud a pointer to the point cloud whose standard deviation we want to
     calculate.
 * \param[in] center the distribution center from which we want to calculate the
     standard deviation,
 *                    typically the average or the median.
 */

double deviation (pcl::PointCloud<pcl::PointXYZ >::Ptr cloud, double center)
{
  double u;
  double sum=0.0;

  for (int i = 0; i < cloud->points.size(); i++)
    sum += pow((cloud->points[i].z - center),2);

  u = sqrt(sum/cloud->points.size());

  return u;
}

double deviation (pcl::PointCloud<pcl::PointXYZRGB >::Ptr cloud, double center)
{
  double u;
  double sum=0.0;

  for (int i = 0; i < cloud->points.size(); i++)
    sum += pow((cloud->points[i].z - center),2);

  u = sqrt(sum/cloud->points.size());

  return u;
}

/** \brief Finds the median depth of the points in the point cloud.
 *
 * Finds the median depth of the points in the point cloud, using the
 * z-coordinate as the depth. This measure is much more robust than the
 * average, because it's not affected so much by outliers.
 *
 * Returns:
 *   * The median depth of the points in the point cloud.
 * \param[in] cloud a pointer to the point cloud whose median depth we want to obtain.
 */

double median(pcl::PointCloud<pcl::PointXYZ >::Ptr cloud)
{
  return DEPTH_VALUES[(int)(DEPTH_VALUES.size()/2)];
}

double median(pcl::PointCloud<pcl::PointXYZRGB >::Ptr cloud)
{
  return DEPTH_VALUES[(int)(DEPTH_VALUES.size()/2)];
}

/** \brief Finds the Median Absolute Deviation (MAD) of the depth of the points in the
    point cloud.
 *
 * Finds the Median Absolute Deviation (MAD) of the depth of the points in the point
     cloud,
 * using the z-coordinate as the depth. This measure is much more robust than the
     standard deviation,
 * and it's its equivalent measure when using the median instead of the average.
 * It uses the median depth previously calculated with the median function, in order to
      avoid
 * duplicating calculations since, in many cases, both values are needed.
 *
 * Returns:
 *   * The Mean Absolute Deviation of the depth of the points in the point cloud.
 * \param[in] cloud a pointer to the point cloud whose standard deviation we want to
     calculate.
 * \param[in] center the distribution center from which we want to calculate the
     standard deviation,
 *                   typically the average or the median.
 */

double medianAbsDev (pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, double center)
{
  double u;
  std::vector<double> absDev;

  for (int i = 0; i < cloud->points.size(); i++)
    absDev.push_back(cloud->points[i].z - center);

  std::sort(absDev.begin(), absDev.end());
  u = absDev[(int)(absDev.size()/2)];

  return u;
}

double medianAbsDev (pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud, double center)
{
  double u;
  std::vector<double> absDev;

  for (int i = 0; i < cloud->points.size(); i++)
    absDev.push_back(cloud->points[i].z - center);

  std::sort(absDev.begin(), absDev.end());
  u = absDev[(int)(absDev.size()/2)];

  return u;
}

/** \brief Reads a point cloud from a PLY file.
 *
 * Reads a point cloud data from the specified PLY file and stores it into a
 * new cloud.
 *
 * Returns:
 *   * A pointer to the newly created point cloud object.
 * \param[in] file_name The path of the PLY file.
 */

pcl::PointCloud<pcl::PointXYZ>::Ptr readCloudFromPLY(const std::string file_name)
{
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::PointXYZ>);

  // Read in the cloud data
  if (pcl::io::loadPLYFile<pcl::PointXYZ> (file_name, *cloud) == -1) //* load the file
  {
    PCL_ERROR ("Couldn't read file %s\n", file_name);
    return cloud;
  }


  return cloud;
}
```

```
/** \brief Reads a point cloud from a PCD file.
 *
 * Reads a point cloud data from the specified PCD file and stores it into a
 * new cloud.
 *
 * Returns:
 *  * A pointer to the newly created point cloud object.
 * \param[in] file_name The path of the PCD file.
 */

pcl::PointCloud<pcl::PointXYZ>::Ptr readCloudFromPCD(const std::string file_name)
{
  pcl::PCDWriter writer;
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud;

  // Read in the cloud data
  pcl::io::loadPCDFile(file_name,*cloud);

  return cloud;
}

/** \brief Writes a point cloud onto a PLY file.
 *
 * Writes a point cloud data onto the specified PLY file.
 *
 * \param[in] file_name The path of the PLY file.
 * \param[in] cloud a pointer to the point cloud that we want to write.
 */

void writeCloudToPLY(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, const std::string
    file_name)
{
  pcl::PLYWriter writer;

  if (cloud->points.size()>0)
    writer.write (file_name, *cloud, false);
}

void writeCloudToPLY(pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud, const std::string
    file_name)
{
  pcl::PLYWriter writer;

  if (cloud->points.size()>0)
    writer.write (file_name, *cloud, false);
}

/** \brief Writes a point cloud onto a PCD file.
 *
 * Writes a point cloud data onto the specified PCD file.
 *
 * \param[in] file_name The path of the PCD file.
 * \param[in] cloud a pointer to the point cloud that we want to write.
 */

void writeCloudToPCD(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, const std::string
    file_name)
{
  pcl::PCDWriter writer;

  if (cloud->points.size()>0)
    writer.write<pcl::PointXYZ> (file_name, *cloud, false);
```

```
}

void writeCloudToPCD(pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud, const std::string
    file_name)
{
  pcl::PCDWriter writer;

  if (cloud->points.size()>0)
    writer.write<pcl::PointXYZRGB> (file_name, *cloud, false);
}

/** \brief Divide the point cloud in several sub-clouds at different depth levels.
 *
 * Creates several depth planes based on the statistical distribution of the depths
 * of the points in the point cloud. Then finds the closest points to each plane and
     fits
 * their depth to the corresponding plane.
 *
 * Returns:
 *   * A vector of pointers to the generated sub-clouds for each depth plane.
 * \param[in] cloud a pointer to the point cloud that we want to divide.
 */

std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr> fitCloudToPlanes(pcl::PointCloud<
    pcl::PointXYZRGB>::Ptr cloud)
{
  std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr> cloudList;
  int i;
  double avg, dev, minZ, maxZ, med;

  // Compute statistics from the cloud
  minZ = min_depth(cloud);
  maxZ = max_depth(cloud);

  std::vector <double> plane_positions;
  std::vector <double> plane_limits;

  mwArray planes;

  mwArray depth_values(1, DEPTH_VALUES.size(), mxDOUBLE_CLASS);
  depth_values.SetData(DEPTH_VALUES.data(), DEPTH_VALUES.size());

  clusterValues( 1, planes, depth_values, mwArray( MAX_NUM_PLANES ), mwArray( 4.0 ) );

  NUM_PLANES = planes.NumberOfElements();

  cout << planes <<endl;

  plane_positions.resize(NUM_PLANES);
  planes.GetData(plane_positions.data(), NUM_PLANES);

  std::sort(plane_positions.begin(), plane_positions.end());

  plane_limits.resize(NUM_PLANES+1);
  plane_limits[0] = ((minZ <= plane_positions[0])? minZ : plane_positions[0]);
  plane_limits[NUM_PLANES] = ((maxZ >= plane_positions[NUM_PLANES-1])? maxZ :
      plane_positions[NUM_PLANES-1]);
  for (i=1; i<NUM_PLANES; i++)
  {
    plane_limits[i] = plane_positions[i-1] + ((plane_positions[i] - plane_positions[i
        -1])/2.0);
  }
  for (i=0; i<NUM_PLANES; i++)
  {
    pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_filtered (new pcl::PointCloud<pcl::
        PointXYZRGB>);
    pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_projected (new pcl::PointCloud<pcl::
        PointXYZRGB>);

    // Use a passthrough filter to select the points belonging to each depth plane
    pcl::PassThrough<pcl::PointXYZRGB> pass;
    pass.setInputCloud (cloud);
    pass.setFilterFieldName ("z");
    pass.setFilterLimits (plane_limits[i], plane_limits[i+1]);
    pass.filter (*cloud_filtered);

    // Create the coefficients that define the depth plane (X,Y=0; Z=plane_position).
    pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients ());
    coefficients->values.resize (4);
    coefficients->values[0] = coefficients->values[1] = 0;
    coefficients->values[2] = 1.0;
    coefficients->values[3] = plane_positions[i];

    // Save the coefficients of the plane for future use.
    DEPTH_PLANES.push_back(*coefficients);

    // Create the projection object
    pcl::ProjectInliers<pcl::PointXYZRGB> proj;
    proj.setModelType (pcl::SACMODEL_PLANE);
    proj.setInputCloud (cloud_filtered);
    proj.setModelCoefficients (coefficients);
    proj.filter (*cloud_projected);

    cloudList.push_back(cloud_projected);
  }

  return cloudList;
}

/** \brief Creates an instance of the built-in PCL viewer PCLVisualizer.
 *
 * Creates an instance of the built-in PCL viewer PCLVisualizer and sets some
 * of its parameters.
 *
 * Returns:
 *   * The newly created viewer object.
 *
 */

boost::shared_ptr<pcl::visualization::PCLVisualizer> createViewer(void)
{
  boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer (new pcl::visualization::
      PCLVisualizer ("3D Viewer"));
  viewer->setBackgroundColor (0, 0, 0);

  viewer->addCoordinateSystem (0.1);
  viewer->initCameraParameters ();

  return viewer;
}

/** \brief Displays a point cloud in an existing viewer.
 *
 * Displays a point cloud in an existing viewer, which was created previously.
 *
 * \param[in] viewer an existing viewer where we want to display the point clouds.
 * \param[in] cloudList a vector of pointers to all the point clouds that we want to
     display.
 */

void displayCloud(boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer, std::
    vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> cloudList)
{
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::PointXYZ>);

  for (int i=0; i<NUM_PLANES; i++){
    cloud = cloudList[i];
    std::stringstream ss;
    pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> single_color(cloud,
         (rand()%255)+1, (rand()%255)+1, (rand()%255)+1);

    ss.str(std::string());
    ss << "cloud" << i;
    viewer->addPointCloud<pcl::PointXYZ> (cloud, single_color, ss.str());
    viewer->setPointCloudRenderingProperties (pcl::visualization::
        PCL_VISUALIZER_POINT_SIZE, 3, ss.str());

    i++;
  }
}

/** \brief Displays all the planes stored in DEPTH_PLANES in an existing viewer.
 *
 * Displays all the planes stored in DEPTH_PLANES in an existing viewer, which was
     created
 * previously. Of course, the planes must have been created previously when calling the
      function
 * fitCloudToPlanes.
 *
 * \param[in] viewer an existing viewer where we want to display the point clouds.
 */

void displayPlanes(boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer)
{
  for (int i=0; i<DEPTH_PLANES.size(); i++)
  {
    std::stringstream plane_id;
    plane_id << "plane"<<i;
    viewer->addPlane(DEPTH_PLANES[i],plane_id.str());
  }
}

/** \brief project_inliers entrypoint.
 *
 * This is the main method of this program.
 * All the primary functions are called within
 * this method.
 *
 * Returns:
 *   * 0 on success
 *   * <>0 on error
 */

int main (int argc, char** argv)
{
```

```
std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr> cloudList;
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_orig_gray (new pcl::PointCloud<pcl::
    PointXYZ>);
pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_orig (new pcl::PointCloud<pcl::
    PointXYZRGB>);
pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_mapped (new pcl::PointCloud<pcl::
    PointXYZRGB>);
pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_merged (new pcl::PointCloud<pcl::
    PointXYZRGB>);
boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer;
pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients ());
int num_planes;
int key=0;
BundlerReader br;

cv::Mat image;
cv::Mat src, src_gray, dst;

// Load an image
src = cv::imread( "./frames/rgb100.jpg" );
if( !src.data ) { return -1; }

// Initialize the MATLAB Compiler Runtime global state
  if (!mclInitializeApplication(NULL,0))
  {
      std::cerr << "Could not initialize the application properly."
                << std::endl;
      return -1;
  }

  // Initialize the libmtools library
  if( !libmtoolsInitialize() )
  {
      std::cerr << "Could not initialize the library properly."
                << std::endl;
      return -1;
  }


// Read the file and separate the points into depth planes.
cloud_orig_gray = readCloudFromPLY("./frames/points113.ply");
cloud_orig = convertCloudToRGB(cloud_orig_gray);
br.load("./frames/bundle.out", SceneInfo);
SceneInfo.setActiveView(100); //this index is 0-based


ofstream myfile;
myfile.open ("./frames/feature_keys.txt");

for (int i=0; i<SceneInfo.numPoints(); i++){
  BundlerPoint p = SceneInfo.getPoint(i);
  for (int j=0; j<p.numViews(); j++){
    BundlerView v = p.getView(j);
    if (v.camera == SceneInfo.getActiveView()){
      cout << v.key << "," << v.x << "," << v.y << endl;
      myfile << v.key << "," << v.x << "," << v.y << endl;
    }
  }
}

myfile.close();
writeCloudToPLY(cloud_orig, "./frames/cloud_orig.ply");
```

```
  cloud_mapped = mapImage (src, cloud_orig);
  writeCloudToPLY(cloud_mapped, "./frames/mapped.ply");

  extractDepthValues(cloud_mapped);
  cloudList = fitCloudToPlanes(cloud_mapped);

  mergeClouds(cloudList, *cloud_merged);
  writeCloudToPLY(cloud_merged, "./frames/merged.ply");

  if (!mclTerminateApplication())
    {
        std::cerr << "Could not terminate the matlab application properly." << std::
            endl;
        return -1;
    }


  return (0);
}
```

Listing A.4: Test code 1 (C++) – Abandoned

```
// opencv_test.cpp : Defines the entry point for the console application.
// This program captures video from the computer's webcam and displays it.
// Press Esc to finish.
//


//c
#include <stdio.h>
#include <string.h>

//c++
#include <iostream>
#include <string>
#include <utility>

//cv
#include <omp.h>
#include <opencv/cv.h>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2\imgproc\imgproc_c.h>
#include <opencv2\features2d\features2d.hpp>

//pcl
#include <pcl/visualization/cloud_viewer.h>


bool DEBUG=false;

using namespace std;
using namespace cv;


template<class T> class Image
{
  private:
  IplImage* imgp;
  public:
  Image(IplImage* img=0) {imgp=img;}
  ~Image(){imgp=0;}
  void operator=(IplImage* img) {imgp=img;}
  inline T* operator[](const int rowIndx) {
    return ((T *)(imgp->imageData + rowIndx*imgp->widthStep));}
};

typedef struct{
  unsigned char b,g,r;
} RgbPixel;

typedef struct{
  float b,g,r;
} RgbPixelFloat;

typedef Image<RgbPixel>       RgbImage;
typedef Image<RgbPixelFloat>  RgbImageFloat;
typedef Image<unsigned char>  BwImage;
typedef Image<float>          BwImageFloat;



float Ek100_1000h[] = {0.95,0.01,0.01,0.01,0.81,0.05,0.01,-0.01,0.85};
float Ek100_1000h2[] = {0.12,0.04,0.02};

float Ek100_1800h[] = {0.75,0.05,0.00,0.05,0.70,0.08,0.02,0.0,0.73};
float Ek100_1800h2[] = {0.22,0.07,0.03};

float Ek100_2600h[] = {0.54,0.08,0.01,0.06,0.62,0.10,0.03,0.01,0.61};
float Ek100_2600h2[] = {0.33,0.09,0.04};


//float Ek100_1000h[] = {1,1,1,1,1,1,1,1,1};
//float Ek100_1000h2[] = {0,0,0};

void capture1(void);
//void capture_match(void);
IplImage* colorDegradation(IplImage* img);
void colorDegradation2(IplImage* img);

int main(int argc, char* argv[])
{
  //capture1();
  CvMemStorage* storage = cvCreateMemStorage(0);
  cvNamedWindow("Image", 1);
  int key = 0;
  static CvScalar red_color[] ={0,0,255};

  CvCapture* capture = cvCreateCameraCapture(0);
  CvMat* prevgray = 0, *image = 0, *gray =0;

  while( key != 'q' )
  {
    int firstFrame = gray == 0;
    IplImage* frame = cvQueryFrame(capture);
    //IplImage* frame = cvLoadImage("apple.jpg",1);
    if(!frame)
      break;
    if(!gray)
    {
      image = cvCreateMat(frame->height, frame->width, CV_8UC1);
    }
    //Convert the RGB image obtained from camera into Grayscale
    cvCvtColor(frame, image, CV_BGR2GRAY);
    //Define sequence for storing surf keypoints and descriptors
    CvSeq *imageKeypoints = 0, *imageDescriptors = 0;
    int i;

    //Extract SURF points by initializing parameters
    //CvSURFParams params = cvSURFParams(500, 1);
    //cvExtractSURF( image, 0, &imageKeypoints, &imageDescriptors, storage, params );
    //printf("Image Descriptors: %d\n", imageDescriptors->total);

    //draw the keypoints on the captured frame
    /*
    for( i = 0; i < imageKeypoints->total; i++ )
    {
      CvSURFPoint* r = (CvSURFPoint*)cvGetSeqElem( imageKeypoints, i );
      CvPoint center;
      int radius;
      center.x = cvRound(r->pt.x);
      center.y = cvRound(r->pt.y);
      radius = cvRound(r->size*1.2/9.*2);
      cvCircle( frame, center, radius, red_color[0], 1, 8, 0 );
    }
    */

    // image = Grayscale image
    // frame = Color image

    // Apply color dye simulation to "frame", according to the formula in page 16 of
        paper.
    frame = colorDegradation(frame);
    cvShowImage( "Image", frame );

    cvWaitKey(30);
  }

  cvDestroyWindow("Image");

 return 0;
}

void colorDegradation2(IplImage* img){
  CvMat *im = cvCreateMat(img->height,img->width,CV_32FC3 );
  CvMat *im_cmy = cvCreateMat(img->height,img->width,CV_32FC3 );

  IplImage *img_cmy = cvCloneImage( img );

  cvConvert( img, im );
  cvConvert( img_cmy, im_cmy );

  cvScale(im,im_cmy,-1.0);
  cvAddS(im_cmy,cvScalar(255.0,255.0,255.0),im_cmy);

  cvConvert(im,img);
  cvConvert(im_cmy,img_cmy);

  cvShowImage( "rgb image", img);
  cvShowImage( "cmy image", im_cmy);
  //Mat im = imread("peppers.tif");
  //Mat im_cmy = 255 - im;
}

IplImage* colorDegradation(IplImage* img){
  IplImage *imgT;

  IplImage *img_cmy;
  IplImage *img_cmy_deg;
  IplImage *img_deg;

  int height,width,step,channels;
  uchar *data, *data2;
  int i,j;
  float cmy [3];
  float cmy_test [3];

  img_cmy = cvCloneImage( img );
  img_cmy_deg = cvCloneImage( img_cmy );

  CvMat M1 = cvMat( 3, 3, CV_32F, Ek100_2600h );
  CvMat M2 = cvMat( 3, 1, CV_32F, Ek100_2600h2 );

  height    = img_cmy_deg->height;
  width     = img_cmy_deg->width;
  channels  = img_cmy_deg->nChannels;
  step      = img_cmy_deg->widthStep;
  data      = (uchar *)img_cmy_deg->imageData;

#pragma omp parallel for private(j, cmy)
  for (i=0; i<height; i++){
    for (j=0; j<width; j++){
      CvScalar s=cvGet2D(img_cmy_deg,i,j); //Get value of pixel (i,j)
      cmy[0]=255-s.val[0];
      cmy[1]=255-s.val[1];
      cmy[2]=255-s.val[2];

      CvMat *CMY2 = &cvMat(3,1,CV_32FC1,cmy);
      CvMat *CMY1 = &cvMat(3,1,CV_32FC1,cmy);

      cvMatMul(&M1, CMY1, CMY2); //Multiply by the bleaching matrix.
      cvAdd(CMY2,&M2,CMY2);      //Add the Base density correction.
      data[i*step+j*channels+0] = (float)(CMY2->data.fl+0*(CMY2->step/sizeof(float)))
          [0];
      data[i*step+j*channels+1] = (float)(CMY2->data.fl+0*(CMY2->step/sizeof(float)))
          [1];
      data[i*step+j*channels+2] = (float)(CMY2->data.fl+0*(CMY2->step/sizeof(float)))
          [2];
    }
  }

  img_deg = cvCloneImage( img_cmy_deg );

  data2     = (uchar *)img_deg->imageData;

#pragma omp parallel for private(j, cmy)
  for (i=0; i<height; i++){
    for (j=0; j<width; j++){
      CvScalar s=cvGet2D(img_cmy_deg,i,j); //Get value of pixel (i,j)
      cmy[0]=255-s.val[0];
      cmy[1]=255-s.val[1];
      cmy[2]=255-s.val[2];

      data2[i*step+j*channels+0] = 255 - data2[i*step+j*channels+0];
      data2[i*step+j*channels+1] = 255 - data2[i*step+j*channels+1];
      data2[i*step+j*channels+2] = 255 - data2[i*step+j*channels+2];
    }
  }

  return img_deg;
}

//http://opencv.willowgarage.com/documentation/c/
    camera_calibration_and_3d_reconstruction.html#findstereocorrespondencegc
pair<CvMat*,CvMat*> calcDepthmap(IplImage* cvImgLeft, IplImage* cvImgRight, int
    numberOfDisparities, int maxIters) {

    CvSize size = cvGetSize(cvImgLeft);
    CvMat* disparity_left = cvCreateMat( size.height, size.width, CV_16S );
    CvMat* disparity_right = cvCreateMat( size.height, size.width, CV_16S );
    CvStereoGCState* state = cvCreateStereoGCState( numberOfDisparities, maxIters );
    cvFindStereoCorrespondenceGC( cvImgLeft, cvImgRight, disparity_left,
        disparity_right, state, 0 );
    cvReleaseStereoGCState( &state );

    CvMat* disparity_left_visual = cvCreateMat( size.height, size.width, CV_16S );
    CvMat* disparity_right_visual = cvCreateMat( size.height, size.width, CV_16S );
    cvConvertScale( disparity_left, disparity_left_visual, -16 );
    cvConvertScale( disparity_right, disparity_right_visual, -16 );

    if (DEBUG) {
        cvSave("disparity_left.pgm", disparity_left_visual );
        cvSave("disparity_right.pgm", disparity_right_visual );
    }

    return pair<CvMat*,CvMat*>(disparity_left_visual, disparity_right_visual );
}

CvMat* getDisparityMap(IplImage* cvImgLeft, IplImage* cvImgRight, int
    numberOfDisparities=16, int maxIters=2){
  //set numberOfDisparities
    if (numberOfDisparities<1) numberOfDisparities=1;

    //set maxIters
    if (maxIters<1) maxIters=1;

    //create depthmap
    pair<CvMat*,CvMat*> disparity = calcDepthmap(cvImgLeft, cvImgRight,
        numberOfDisparities, maxIters);

  //return one of the resulting disparity maps.
    return disparity.first;
}


void capture1(void){
  int c;
  // allocate memory for an image
  IplImage *img;
  // capture from video device #1
  CvCapture* capture = cvCaptureFromCAM(1); // capture from video device #1
  // create a window to display the images
  cvNamedWindow("mainWin", CV_WINDOW_AUTOSIZE);
  // position the window
  cvMoveWindow("mainWin", 5, 5);
  while(1)
  {
    // retrieve the captured frame
    img=cvQueryFrame(capture);
    // show the image in the window
    cvShowImage("mainWin", img );
    // wait 10 ms for a key to be pressed
    c=cvWaitKey(10);
    // escape key terminates program
    if(c == 27)
    break;
  }
}
/*
void capture_match(void){
  CvMemStorage* storage = cvCreateMemStorage(0);
  cvNamedWindow("Image", 1);
  int key = 0;
  static CvScalar red_color[] ={0,0,255};

  CvCapture* capture = cvCreateCameraCapture(0);
  CvMat* prevgray = 0, *image = 0, *gray =0;
  while( key != 'q' )
  {
    int firstFrame = gray == 0;
    IplImage* frame = cvQueryFrame(capture);
    if(!frame)
      break;
    if(!gray)
    {
      image = cvCreateMat(frame->height, frame->width, CV_8UC1);
    }
    //Convert the RGB image obtained from camera into Grayscale
    cvCvtColor(frame, image, CV_BGR2GRAY);
    //Define sequence for storing surf keypoints and descriptors
    CvSeq *imageKeypoints = 0, *imageDescriptors = 0;
    int i;

    //Extract SURF points by initializing parameters
    CvSURFParams params = cvSURFParams(500, 1);
    cvExtractSURF( image, 0, &imageKeypoints, &imageDescriptors, storage, params );
    printf("Image Descriptors: %d\n", imageDescriptors->total);

    //draw the keypoints on the captured frame
    for( i = 0; i < imageKeypoints->total; i++ )
    {
      CvSURFPoint* r = (CvSURFPoint*)cvGetSeqElem( imageKeypoints, i );
      CvPoint center;
      int radius;
      center.x = cvRound(r->pt.x);
      center.y = cvRound(r->pt.y);
      radius = cvRound(r->size*1.2/9.*2);
      cvCircle( frame, center, radius, red_color[0], 1, 8, 0 );
    }
    cvShowImage( "Image", frame );
    cvWaitKey(30);
  }
  cvDestroyWindow("Image");
}*/
```

Listing A.5: Test code 2 (C++) – Abandoned

```
// opencv_test.cpp : Defines the entry point for the console application.
// This program captures video from the computer's webcam and displays it.
// Press Esc to finish.
//
```

```

//c
#include <stdio.h>
#include <string.h>

//c++
#include <iostream>
#include <string>
#include <utility>
#include <vector>

//cv
#include <omp.h>
#include <opencv/cv.h>
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2\imgproc\imgproc_c.h>
#include <opencv2\features2d\features2d.hpp>

//pcl
#include <pcl/visualization/cloud_viewer.h>


bool DEBUG=false;

using namespace std;
using namespace cv;


template<class T> class Image
{
  private:
  IplImage* imgp;
  public:
  Image(IplImage* img=0) {imgp=img;}
  ~Image(){imgp=0;}
  void operator=(IplImage* img) {imgp=img;}
  inline T* operator[](const int rowIndx) {
    return ((T *)(imgp->imageData + rowIndx*imgp->widthStep));}
};

typedef struct{
  unsigned char b,g,r;
} RgbPixel;

typedef struct{
  float b,g,r;
} RgbPixelFloat;

typedef Image<RgbPixel>       RgbImage;
typedef Image<RgbPixelFloat>  RgbImageFloat;
typedef Image<unsigned char>  BwImage;
typedef Image<float>          BwImageFloat;



float Ek100_1000h[] = {0.95,0.01,0.01,0.01,0.81,0.05,0.01,-0.01,0.85};
float Ek100_1000h2[] = {0.12,0.04,0.02};

float Ek100_1800h[] = {0.75,0.05,0.00,0.05,0.70,0.08,0.02,0.0,0.73};
float Ek100_1800h2[] = {0.22,0.07,0.03};

float Ek100_2600h[] = {0.54,0.08,0.01,0.06,0.62,0.10,0.03,0.01,0.61};
float Ek100_2600h2[] = {0.33,0.09,0.04};

IplImage* colorDegradation(IplImage* img);
Mat colorDegradation2(Mat img);
pair<CvMat*,CvMat*> calcDepthmap(IplImage* cvImgLeft, IplImage* cvImgRight, int
    numberOfDisparities, int maxIters);
CvMat* getDisparityMap(IplImage* cvImgLeft, IplImage* cvImgRight, int
    numberOfDisparities, int maxIters);
Mat calcDepthmap_BM(Mat img1, Mat img2, int numberOfDisparities=32, int maxIters=2);

int main(int argc, char* argv[])
{
  int key = 0, i;
  boolean retVal=true;
  Mat frame;
  //vector<Mat *>::iterator it;
  Mat disparity, disp_deg;
  Mat img1, img2;
  int frameOffset;
  int numberOfDisparities=16;

  VideoCapture cap("touchofevil.avi");  // Create an object from a video file.
  //VideoCapture cap("tron_sample.avi");  // Create an object from a video file.

  if(!cap.isOpened()) return -1;  // check if we succeeded

  Mat frame_deg, frame_deg2;
  IplImage* test;
  namedWindow("img1",1);
  //namedWindow("img1 deg",1);
  //namedWindow("depth map",1);
  //namedWindow("depth map deg",1);


  int nframes = cap.get(CV_CAP_PROP_FRAME_COUNT);
  Mat **preLoadedVideo = new Mat *[nframes];  //create nframes pointers to Mat.

  // Preload Videos into a Vector for faster Access.
  cout << "Preloading Frames" << endl;

  for (i=0; i<nframes; i++){
    preLoadedVideo[i] = new Mat;
    cap >> frame;
    if (!frame.data){
      nframes=i-1;
      break;
    }
    *(preLoadedVideo[i]) = frame.clone();
    cout << "Loading frame " << i << endl;
  }
  cout << "Finished Preloading Video" << endl;

  frameOffset=10;
//#pragma omp parallel for private(img1, img2, disparity, disp_deg, frame_deg,
    frame_deg2)
  for (i=0; i<nframes-frameOffset; i++){
    img1 = *(preLoadedVideo[i]);
    img2 = *(preLoadedVideo[i+frameOffset]);
    //disparity = calcDepthmap_BM(img2, img1, numberOfDisparities);
    //frame_deg = colorDegradation2(img1);
    //frame_deg2 = colorDegradation2(img2);
    //disp_deg = calcDepthmap_BM(frame_deg2, frame_deg, numberOfDisparities);
    //imshow("img1 deg", frame_deg);
    //imshow("depth map deg", disp_deg);
    cout << "Showing frame " << i << endl;
    imshow("img1", *(preLoadedVideo[i]));
    //imshow("depth map", disparity);
    if(waitKey(30) >= 0) break;
  }

  // the camera will be deinitialized automatically in the VideoCapture destructor
  return 0;
}

Mat findFeatures(Mat img){
  //Define sequence for storing surf keypoints and descriptors
  CvSeq *imageKeypoints = 0, *imageDescriptors = 0;
  int i;

  CvArr *image=CvArr(&img);

  //Extract SURF points by initializing parameters
  CvSURFParams params = cvSURFParams(500, 1);
  cvExtractSURF( image, 0, &imageKeypoints, &imageDescriptors, storage, params );
  printf("Image Descriptors: %d\n", imageDescriptors->total);

  //draw the keypoints on the captured frame

  for( i = 0; i < imageKeypoints->total; i++ )
  {
    CvSURFPoint* r = (CvSURFPoint*)cvGetSeqElem( imageKeypoints, i );
    CvPoint center;
    int radius;
    center.x = cvRound(r->pt.x);
    center.y = cvRound(r->pt.y);
    radius = cvRound(r->size*1.2/9.*2);
    cvCircle( frame, center, radius, red_color[0], 1, 8, 0 );
  }
}

Mat calcDepthmap_BM(Mat img1, Mat img2, int numberOfDisparities, int maxIters) {

    CvStereoBMState* BMState;
  StereoSGBM sgbm;
  StereoBM bm;
  int SADWindowSize = 0;

  //disparity = cvCreateMat( size.height, size.width, CV_16S );

  Mat disp, disp8;

  //state = cvCreateStereoBMState( numberOfDisparities, maxIters );

  Size img_size = img1.size();
  numberOfDisparities = numberOfDisparities > 0 ? numberOfDisparities : ((img_size.
      width/8) + 15) & -16;

  Rect roi1, roi2;

  bm.state->roi1 = roi1;
    bm.state->roi2 = roi2;
    bm.state->preFilterCap = 31;
    bm.state->SADWindowSize = SADWindowSize > 0 ? SADWindowSize : 9;
    bm.state->minDisparity = 0;
    bm.state->numberOfDisparities = numberOfDisparities;
    bm.state->textureThreshold = 10;
    bm.state->uniquenessRatio = 15;
    bm.state->speckleWindowSize = 100;
    bm.state->speckleRange = 32;
    bm.state->disp12MaxDiff = 1;

  sgbm.preFilterCap = 63;
    sgbm.SADWindowSize = SADWindowSize > 0 ? SADWindowSize : 3;

    int cn = img1.channels();

    sgbm.P1 = 8*cn*sgbm.SADWindowSize*sgbm.SADWindowSize;
    sgbm.P2 = 32*cn*sgbm.SADWindowSize*sgbm.SADWindowSize;
    sgbm.minDisparity = 0;
    sgbm.numberOfDisparities = numberOfDisparities;
    sgbm.uniquenessRatio = 10;
    sgbm.speckleWindowSize = bm.state->speckleWindowSize;
    sgbm.speckleRange = bm.state->speckleRange;
    sgbm.disp12MaxDiff = 1;
    sgbm.fullDP = false;


  int64 _time1 = cvGetTickCount();

  //cvFindStereoCorrespondenceBM( cvImgLeft, cvImgRight, disparity, BMState );
    //cvReleaseStereoBMState( &BMState );
  sgbm(img1, img2, disp);

  int64 _time2 = cvGetTickCount();
  printf("Time elapsed: %f\n", float(_time2 - _time1)/cvGetTickFrequency()*1e-6);

  //disparity_visual = cvCreateMat( size.height, size.width, CV_16S );
  //cvConvertScale( disparity, disparity_visual, -16 );
  disp.convertTo(disp8, CV_8U, 255/(numberOfDisparities*16.));

    if (DEBUG) {
    //cvSave("disparity.pgm", disparity_visual );
    imwrite("disparity.pgm", disp8);
    }

    //imwrite(disparity_filename, disp8);
  return disp8;
}

/*
void showCloud ()
 {
   pcl::PointCloud<pcl::PointXYZRGB> cloud;
   //... populate cloud

   pcl_visualization::CloudViewer viewer("Simple Cloud Viewer");
   viewer.showCloud(cloud);
   while (!viewer.wasStopped())
   {
   }
 }
*/
```

```
Mat colorDegradation2(Mat img){
  Mat img_cmy, img_deg;
  Mat CMY;
  int i,j;
  float cmy[3];

  img_cmy = cvScalar(255,255,255) - img; // convert to cmy

  //Mat M1 = Mat( 3, 3, CV_32F, Ek100_2600h );
  //Mat M2 = Mat( 3, 1, CV_32F, Ek100_2600h2 );

  CvMat M1 = cvMat( 3, 3, CV_32F, Ek100_2600h );
  CvMat M2 = cvMat( 3, 1, CV_32F, Ek100_2600h2 );

  img_deg = img_cmy;

#pragma omp parallel for private(j, cmy)
  for (i=0; i<img_deg.rows; i++){
    for (j=0; j<img_deg.cols; j++){
      Vec3b p = img_deg.at<Vec3b>(i,j);

      cmy[0]=p.val[0];
      cmy[1]=p.val[1];
      cmy[2]=p.val[2];

      CvMat *CMY2 = &cvMat(3,1,CV_32FC1,cmy);
      CvMat *CMY1 = &cvMat(3,1,CV_32FC1,cmy);

      cvMatMul(&M1, CMY1, CMY2); //Multiply by the bleaching matrix.
      cvAdd(CMY2,&M2,CMY2);      //Add the Base density correction.

      img_deg.at<Vec3b>(i,j)[0]=(CMY2->data.fl+0*(CMY2->step/sizeof(float)))[0];
      img_deg.at<Vec3b>(i,j)[1]=(CMY2->data.fl+0*(CMY2->step/sizeof(float)))[1];
      img_deg.at<Vec3b>(i,j)[2]=(CMY2->data.fl+0*(CMY2->step/sizeof(float)))[2];
    }
  }

  img_deg = cvScalar(255,255,255) - img_deg; // convert back to rgb

  return img_deg;
}


//http://opencv.willowgarage.com/documentation/c/
    camera_calibration_and_3d_reconstruction.html#findstereocorrespondencegc
pair<CvMat*,CvMat*> calcDepthmap(IplImage* cvImgLeft, IplImage* cvImgRight, int
    numberOfDisparities, int maxIters) {

    CvSize size = cvGetSize(cvImgLeft);
    CvMat* disparity_left = cvCreateMat( size.height, size.width, CV_16S );
    CvMat* disparity_right = cvCreateMat( size.height, size.width, CV_16S );
    CvStereoGCState* state = cvCreateStereoGCState( numberOfDisparities, maxIters );
    cvFindStereoCorrespondenceGC( cvImgLeft, cvImgRight, disparity_left,
        disparity_right, state, 0 );
    cvReleaseStereoGCState( &state );

    CvMat* disparity_left_visual = cvCreateMat( size.height, size.width, CV_16S );
    CvMat* disparity_right_visual = cvCreateMat( size.height, size.width, CV_16S );
    cvConvertScale( disparity_left, disparity_left_visual, -16 );
    cvConvertScale( disparity_right, disparity_right_visual, -16 );

    if (DEBUG) {
        cvSave("disparity_left.pgm", disparity_left_visual );
        cvSave("disparity_right.pgm", disparity_right_visual );
    }

    return pair<CvMat*,CvMat*>(disparity_left_visual, disparity_right_visual );
}

CvMat* getDisparityMap(IplImage* cvImgLeft, IplImage* cvImgRight, int
    numberOfDisparities=16, int maxIters=2){
  //set numberOfDisparities
    if (numberOfDisparities<1) numberOfDisparities=1;

    //set maxIters
    if (maxIters<1) maxIters=1;

    //create depthmap
    pair<CvMat*,CvMat*> disparity = calcDepthmap(cvImgLeft, cvImgRight,
        numberOfDisparities, maxIters);

  //return one of the resulting disparity maps.
    return disparity.first;
}
```