Marilyn KELLER

SICOM
2016

BBC R&D
56 Wood Ln
London W12 7SB
United Kingdom

Setup of a capture system consisting of multiple video and depth cameras

from 04/04/16 to 19/08/16

Under the supervision of:

- **Company supervisor : Florian SCHWEIGER**
  florian.schweiger@bbc.co.uk
- **Phelma Tutor : Pierre-Yves COULON**
  pierre-yves.coulon@gipsa-lab.grenoble-inp.fr

Confidentiality : no

Ecole nationale
supérieure de physique,
électronique, matériaux

Phelma
Bât. Grenoble INP - Minatec
3 Parvis Louis Néel - CS 50257
F-38016 Grenoble Cedex 01

Tél +33 (0)4 56 52 91 00
Fax +33 (0)4 56 52 91 03

http://phelma.grenoble-inp.fr

Abstract

The present master thesis studies the use of RGB and depth cameras to capture and render a scene. A pair of calibrated RGB and depth camera provides a coloured point cloud. Using several calibrated camera pairs around a scene gives information on the captured objects shape. This master thesis seeks to set up such a capture system and render the data in a realistic way. The rendering of point clouds implies several issues. The observer should not be able to see through an object surface and the colours must be consistent. Different techniques are applied and compared to solve those issues. An interpolation technique to fill the holes between the points gives satisfying results for an off-line rendering, while representing the point cloud as a cloud of quads of varying sizes is a good solution for a real-time rendering. To colour the final point cloud, the information of several colour cameras can be used for a more natural illumination of the scene.

Keywords: Kinect, Depth sensor, Point cloud, Joint calibration, image-based rendering, Plenoptic sampling, Unreal engine 4

Contents

Acknowledgment

I would like to thank the IIC team leader Graham Thomas who gave me the great opportunity to do my master thesis within BBC R&D and my supervisor Florian Schweiger who supported and advised me during those five months and helped me to conclude this project successfully.

Glossary

Levenberg–Marquardt Algorithm Algorithm used to solve non-linear least squares problems. 12

Light field Vector function that describes a scene through the light flowing in every direction. 17–19

LMA Levenberg–Marquardt Algorithm 12, 13

PCA Principal Component Analysis 12

Plenoptic sampling Using the sampling theorem to determine the minimum sampling rate for light field rendering from a spectral analysis of light field signals. 17

Principal Component Analysis Statistical procedure to extract uncorrelated variables from a set of observation. 12

Singular Value Decomposition Factorisation of an $m \times n$ Matrix $\mathbf{M}$ in $\mathbf{U\Sigma V}*$ where $\mathbf{U}$ is a $m \times m$ unitary matrix, $\mathbf{U}$ is a $m \times n$ rectangular diagonal matrix and $\mathbf{U}$ is a $n \times n$ unitary matrix. 12

SVD Singular Value Decomposition 12

1 Introduction

1.1 BBC R&D

The British Broadcasting Corporation (BBC) is a public service broadcaster. Created in 1922, it is the world's oldest broadcasting company. BBC broadcasts on television, radio and online. BBC R&D is the research department of the BBC. It is located in the north lab in Salford and the south lab in London where I worked.

I was in the team Immersive & Interactive Content which focuses on the development of next-generation audio and video systems and ways in which these can offer new interactive opportunities, including virtual reality and augmented reality applications.

1.2 Objectives

The objective of this project is to set up a functional capture system consisting of multiple broadcast cameras and depth sensors. Such a capture system will then be integrated into IP Studio, a framework developed by BBC R&D for IP-based end-to-end broadcasting that allows them to produce content by combining different kind of media objects.

An example application would be the live capture of a game or quiz show in 3D. The show could then be experienced in real-time in a virtual world in which the spectators would be present as avatars and could, from their homes, using a computer, a tablet or a head mounted display, move freely around the scene and actively participate in the programme. The gameshow host could even see the remote participant's avatars and interact with them, reacting to their movement and actions.

To create a first version of such a capture system, we used Microsoft's Kinect sensor because it is a low-cost device combining a colour camera and a depth camera. Furthermore, Kinect is widely used by researchers in computer science, electrical engineering and robotics to create new kinds of Human–computer interactions [31], thus many previous works have been conducted on its use [13, 14, 25].

Once calibrated, a coloured point cloud can be computed from the Kinect data. Many works have been done on generating meshes from 3D point cloud [8, 21, 23] but those operations are quite onerous and require more precise depth information for a faithful rendition than the Kinect can provide. On the other side, with image-based rendering techniques, a new point of view on a scene can be generated from existing ones in quite simple ways.

The main idea behind this project is that knowing the depth information quite precisely, we should be able to generate an accurate new point of view with very few existing point of view and without building a mesh.

The project was divided into two main parts. First, the joint calibration of the system to get usable data from the capture system and second, the exploration of rendering algorithms to render those data as realistic as possible and to finally create a real-time renderer.

In Section 2, the Kinect device and different fundamental concepts relating to camera calibration and image-based rendering will be presented. Section 3 describes the calibration approach taken in this work and its practical implementation in detail. In Section 4, several techniques are discussed to render captured data from new viewpoints. The focus here lies on the fidelity of the rendering. Section 5, on the other hand, introduces a simpler, yet real-time capable rendering approach that has been implemented in an existing game engine.

The provisional planning can be found in Appendix A.

2 State of the art

Many publications exist in the fields linked to my project. I could thus get technical information on both Kinects and test several image-based rendering algorithms to apply to my setup.

2.1 Microsoft's Kinect sensor

Kinect [17] is a 3D sensor by Microsoft for Xbox 360 and Xbox One video game consoles. It introduced a new way of playing, using the body to interact with a game. Since its introduction in 2010, it has been widely used in the world of computer vision for 3D scene reconstruction or object reconstruction. In 2013, a second version has been introduced with improved features [18] and using another technology.

Basically, Kinect outputs an RGB image and depth information of the scene. The depth here corresponds to the distance between a point of the scene and its orthogonal projection in the camera plane (Figure 1).

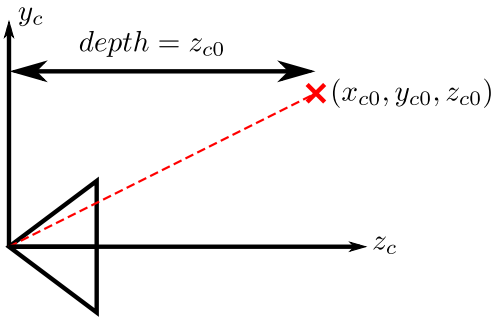


Figure 1: Depth measured by Kinect. The (x_c, y_c, z_c) axis build the camera coordinates system.

In many cases, Kinects data are used to produce meshes. Some approaches allow now to capture real-time moving meshes from multiple RGBD cameras [8,21].

Microsoft released a Software Development Kit [19] to use the Kinect on Windows which can be integrated into the game engine Unity. Thus, the Kinect can be used to interact with a virtual environment. In [7], The game engine is used more as a visualisation tool, similar to the goals of this thesis but with the difference that the camera point of view remains the same so they can just look at a coloured mesh deformed by a depth map.

2.1.1 Kinect for Xbox 360

Kinect for Xbox 360 (Kinect v1) was introduced in 2010. It is made of an RGB camera, an IR camera and an IR projector. The projector projects an IR pattern in the scene and, knowing the projected pattern, a disparity map can be computed from the distortion of the pattern seen by the IR camera. As explained later on, the disparity corresponds to an "inverse depth".

Capturing the Kinect output is achieved with libfreenect, an open source driver running on Linux which is part of the OpenKinect project [1].

For this Kinect version with this driver the obtained output has the following characteristics:

RGB Format: .ppm
Size: 640x480
Frame rate: $\simeq$ 30 Hz

Depth Format: .pgm
Size: 640x480
Frame rate: $\simeq$ 30 Hz

The frame rate is the same for the depth and RGB cameras but the disparity image and the RGB images are captured one after the other.

2.1.2 Kinect for Xbox One

For this second version (Kinect v2), the depth camera uses the time of flight camera (ToF)technology, which consists in measuring the time that it takes for an infrared ray to hit an object and go back to the sensor. It has a better resolution and fewer shadows are projected so less information is lost. This camera outputs directly the depth map of the scene, this one is captured, as well as the RGB image with libfreekinect2-record, a fork of libfreenect2. Unlike Kinect v1, the depth image and the RGB image are

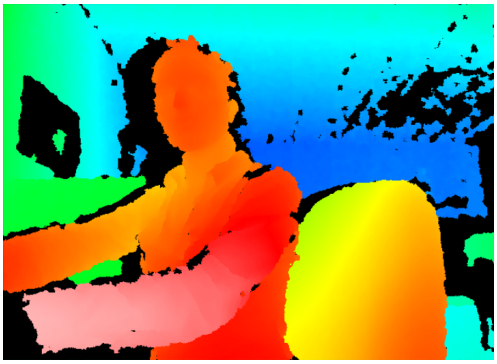
captured quasi-simultaneously, i.e. the duration between both capture is very little compared to the frame rate and both the depth and colour image get the same timestamp. The following output is obtained:

RGB Format: .png
Size: 512x424
Frame rate: $\simeq$ 30 Hz

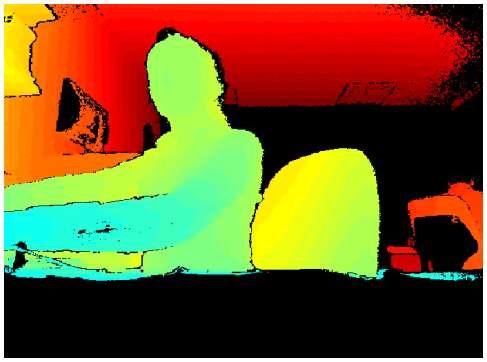
Depth Format: .png
Size: 512x424
Frame rate: $\simeq$ 30 Hz

Libfreenect2 apply fixed calibration parameters such that the depth and RGB superimpose roughly. So the RGB image output is cropped and sampled from the original image which has HD resolution .

Figure 2 shows the depth information returned by the two Kinect versions. We can see that the depth map on Figure 2b shows more precise edges than Figure 2a and no shadows are projected with the depth camera using ToF in Kinect v2. Kinect v1 can, however, see further than Kinect v2, even though the documentation specifies the same depth range (0,80m to 4,00m) for both versions.



(a) Disparity map output by Kinect v1



(b) Depth map output by Kinect v2

Figure 2: Depth information output by the two Kinect versions, black values correspond to no information.

2.2 Calibration

The camera calibration consists in estimating camera parameters so that a point in space can be reprojected in the camera's image coordinates. With the publication of [30] which propose a flexible camera calibration method, a camera can be calibrated with no special equipment, except a checkerboard that must be shown to the camera from multiple orientations.

The calibration parameters are separated in the extrinsic parameters that give the position of the camera in world coordinates and its intrinsic parameters, linked to the imaging properties of its lens and sensor.

2.2.1 Colour camera

With the extrinsic camera parameters which are a rotation matrix $\mathbf{R}_c$ and a translation $\mathbf{t}_c$, a point can be transformed from world coordinates $\mathbf{x}_w$ to camera coordinates $\mathbf{x}_c$ with Eqs.(1) [11].

$$\mathbf{x}_c = {\mathbf{R}_c}^T(\mathbf{x}_w - \mathbf{t}_c) \tag{1}$$

Then, the same intrinsic model used by Herrera [13] is used. The reprojection of a point from colour camera coordinates $\mathbf{x}_c = [x_c, y_c, z_c]^T$ to the colour image coordinates $\mathbf{p}_c = [u_c, v_c]^T$ can be computed using the following equation:

The point is first normalised $\mathbf{x}_n^c = [x_n, y_n]^T = [x_c/z_c, y_c/z_c]^T$ then the distortion is applied:

$$\mathbf{x}_g^c = \begin{bmatrix} 2k_3 x_n y_n + k_4(r^2 + 2{x_n}^2)) \\ k_3(r^2 + 2{y_n}^2) + 2k_4 x_n y_n \end{bmatrix} \tag{2}$$

$$\mathbf{x}_k^c = [x_k, y_k]^T = (1 + k_1 r^2 + k_2 r^4 + k_5 r^6)\mathbf{x}_n^c + \mathbf{x}_g^c \tag{3}$$

where $r^2 = {x_n}^2 + {y_n}^2$ and $\mathbf{k}_c = [k_1, ..., k_5]$ are distortion coefficients.

Finally, the coordinates of the point in colour image space are given by:

$$\begin{bmatrix} u_c \\ v_c \end{bmatrix} = \begin{bmatrix} f_{cx} & 0 \\ 0 & f_{cy} \end{bmatrix} \begin{bmatrix} x_k \\ y_k \end{bmatrix} + \begin{bmatrix} u_{0c} \\ v_{0c} \end{bmatrix} \tag{4}$$

where $\mathbf{f}_c = [f_{cx}\ f_{cy}]$ are the colour camera focal lengths in terms of pixel dimensions in the x and y directions respectively and $\mathbf{p}_c = [u_{0c}\ v_{0c}]$ the camera principal point.

Thus, there are 6 intrinsic and 9 extrinsic parameters to estimate to describe a colour camera model: $\mathcal{L}_c = \{\mathbf{R}_c, \mathbf{t}_c, \mathbf{k}_c, \mathbf{f}_c, \mathbf{p}_c\}$. Indeed, a rotation matrix can be represented by a 3-dimensional vector using Rodrigues Formula [30], this vector is parallel to the rotation axis and its magnitude is equal to the rotation angle.

2.2.2 Depth sensor

The transformation between world coordinates and depth image coordinates is similar to the one used for the colour camera so we have the following parameters: $\{\mathbf{R}_d, \mathbf{t}_d, \mathbf{f}_d, \mathbf{p}_{0d}\}$. However, Herrera [13] defines the distortion model as a backwards model (image to world) instead of a forward model (world to image) defined in Eqs.(2) and (3). This makes the computation of the distortion easier since we will compute a 3D point $\mathbf{x}_w$ from a depth camera pixel $[u_d\ v_d]^T$.

First, the pixel is transformed from image coordinates to get the distorted direction $\mathbf{x}_k^d = [x_k, y_k, 1]$ in depth camera coordinates with equation (5).

$$\mathbf{x}_g^d = \begin{bmatrix} x_g \\ y_g \end{bmatrix} h = \begin{bmatrix} f_{dx} & 0 \\ 0 & f_{dy} \end{bmatrix}^{-1} \begin{bmatrix} u_d - u_{0d} \\ v_d - v_{0d} \end{bmatrix} \tag{5}$$

Second, this point is undistorted by equation (6) to get the direction of the distorted point in depth camera coordinates $\mathbf{x}_n^d = [x_n, y_n, 1]^T$.

$$\mathbf{x}_g^d = \begin{bmatrix} 2k'_3 x_k y_k + k'_4(r^2 + 2{x_k}^2)) \\ k'_3(r^2 + 2{y_k}^2) + 2k'_4 x_k y_k \end{bmatrix} \tag{6}$$

$$\mathbf{x}_n^d(1 + k'_1 r^2 + k'_2 r^4 + k'_5 r^6)\mathbf{x}_k^d + \mathbf{x}_g^d \tag{7}$$

where $r^2 = {x_k}^2 + {y_k}^2$ and $\mathbf{k}_d = [k'_1, ..., k'_5]$ are distortion coefficients.

From its direction in depth camera coordinates and its depth, a 3D point can be reconstructed.

For Kinect v1, Herrera [13] models the relation between the disparity d_k and depth z_d by the equation:

$$z_d = \frac{1}{c_1 d_k + c_0} \tag{8}$$

where c_0 and c_1 are two additional depth camera intrinsic parameters to be estimated.

Herrera also puts forward the following distortion model for the disparity:

$$d_k = d + \mathbf{D}_\delta(u, v) \cdot exp(\alpha_0 - \alpha_1 d) \tag{9}$$

Where d is the distorted disparity returned by the Kinect and $D_\delta(u, v)$, α_0 and α_1 are parameters to optimise.

Thus, we get the 3D point in depth camera coordinate $\mathbf{x}_d = z_d \cdot [x_n^d, y_n^d, 1]^T$.

Finally, $\mathbf{x}_d$ is transformed to world coordinates $\mathbf{x}_w$ by Eqs. (10).

$$\mathbf{x}_w = \mathbf{R}_d \mathbf{x}_d + \mathbf{t}_d \tag{10}$$

Thus, the parameters to estimate are the following: $\mathcal{L}_d = \{\mathbf{R}_d, \mathbf{t}_d, \mathbf{k}_d, \mathbf{f}_d, \mathbf{p}_d, c_0, c_1, D_\delta(u, v), \alpha_0, \alpha_1\}$.

2.3 Pixel displacement

In [22], a method is presented to generate, from a texture and its depth map, a new texture seen from another point of view. This aims to simulate the parallax effect without needing to model all the details of a surface (for example a balcony on a facade).

To do so, the pixels are moved on the image as a function of the depth of the pixel on the depth map and the new viewpoint. The further a point lies off the plane defined by the input texture, the stronger the parallax is, so the more the pixel will be moved to get the new image. The displacement also depends on the calibration parameters of the cameras that captured the image and the virtual camera.

Basically, it is equivalent to compute, from a (u, v) position on the initial image, the point coordinates in world and then project it in a virtual camera to get a new position (u', v') for the pixel.

To fill the holes created by the pixel displacement, two 1-D interpolations are applied. For each pixel moved, an interpolation is done between this one and the last one moved. To prevent occlusions, the pixels must also be warped from the borders of the image toward the epipole corresponding to the new viewpoint.

This technique seems quite interesting for my setup but the limitation is that there is no simple way to combine several warped points of view as shown in Figure 3

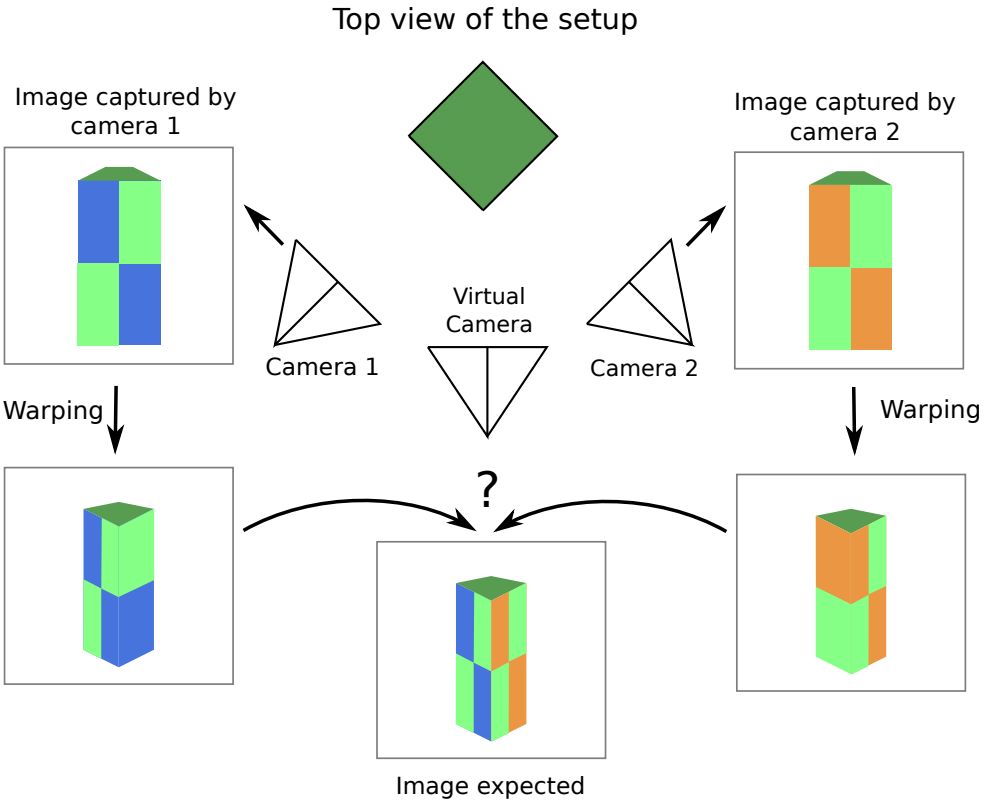


Figure 3: Here we want to generate the virtual camera image from the view of camera 1 and 2. We can wrap each image separately but then it is complicated to recombine them to keep only the valid information brought by each image

2.4 Plenoptic sampling

In this section, I tried to apply the Plenoptic sampling theory to my setup. It is an image-based rendering technique consisting in analysing the spectrum of the scene's Light field to determine the minimum sampling rate for its rendering, i.e. the number of cameras and the image resolution needed for a perfect reconstruction from a new point if view. A Light field is a representation of a 3D scene as a field of light rays described by a plenoptic function l. If the object is in free space, the Light field can be parametrised

into a 4-D plenoptic function [10, 15].

I will make the assumption that in our setup, the Kinects are placed along a circle of radius R looking toward its centre as represented in Figure 4. I also suppose that the Kinects are one simple camera with one colour value and depth value for each pixel. We aim to determine how many cameras we need in this setup to be able to generate a new point of view. Using the depth information provided by the Kinect should reduce the number of Kinect needed.

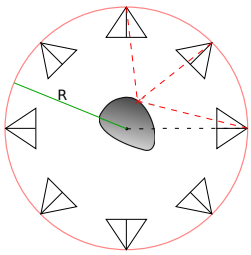


Figure 4: This is the studied setup. A point of the captured surface projects in several cameras in different pixels.

2.4.1 Cameras in a plane

The plenoptic function can be parametrised in several ways. One parametrisation can be done with two planes: a camera plane indexed by (s,t) and a focal plane indexed by (u,v) (Figure 5). Each light ray corresponds then to a value $l(u, v, s, t)$ which is equivalent to the colour of the pixel (u,v) of a camera in (s,t).

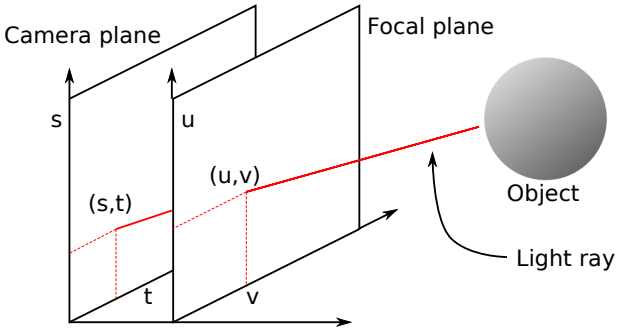


Figure 5: A light ray can be parametrised with four coordinates (u, v, s, t).

This plenoptic function can then be sampled, which is equivalent to having cameras placed at regular positions in the (s,t) plane with discrete pixel coordinates (Figure 6).

The question now is, considering a fixed resolution for the images, how far apart the cameras can be placed in order that no information is lost. In [6], a minimum sampling rate is proposed for Light field rendering, using the depth of the scene.

Light field Sampling

Let Δs and Δt be the distance between the cameras along the s and t axis and $\Delta u \times \Delta v$ the size of an image pixel, the sampled Light field is given by:

$$l_s(u, v, s, t) = l(u, v, s, t) \sum_{n1,n2,k1,k2 \in Z} \delta(u - n_1\Delta_u)\delta(v - n_2\Delta_v)\delta(s - k_1\Delta_s)\delta(t - k_2\Delta_t) \tag{11}$$

Hence its Fourier transform:

$$L_s(\Omega_u, \Omega_v, \Omega_s, \Omega_t) = \sum_{m1,m2,l1,l2 \in Z} L(\Omega_u - \frac{2\pi m_1}{\Delta_u}, \Omega_v - \frac{2\pi m_2}{\Delta_v}, \Omega_s - \frac{2\pi l_1}{\Delta_s}, \Omega_t - \frac{2\pi l_2}{\Delta_t}) \tag{12}$$

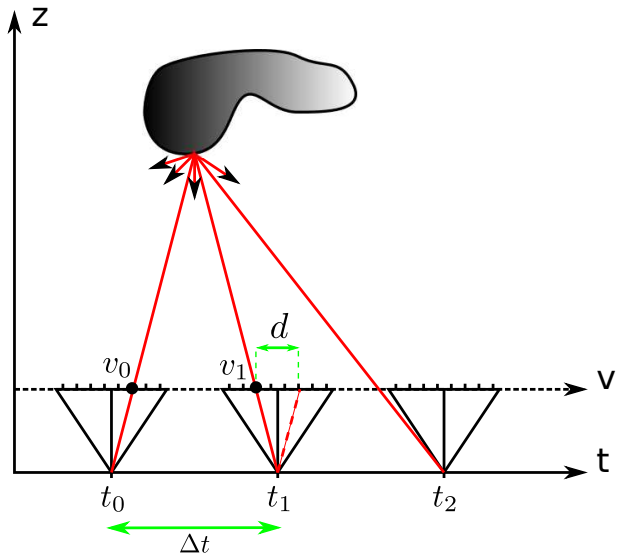


Figure 6: 2D illustration of a sampled Light field signal. The disparity d corresponds to the distance $v_1 - v_0$, where v_0 and v_1 are respectively the projection pixel coordinates in the t_0 and t_1 camera local coordinate system.

So by sampling, the spectrum of the plenoptic function is replicated in every 4D point $(\frac{2\pi m_1}{\Delta_u}, \frac{2\pi m_2}{\Delta_v}, \frac{2\pi l_1}{\Delta_s}, \frac{2\pi l_2}{\Delta_t})$, where $m1, m2, l1, l2 \in Z$.

Spectral support bounds

For a point with depth $z(v,t)$ and a camera displacement $\Delta t = t$, the disparity d between two image coordinates (see Figure 6) is given by $d = \frac{ft}{z(v,t)}$.

$$d = \frac{ft}{z(v,t)} \tag{13}$$

Therefore, the radiance received in (s,t) is

$$l(u,v,s,t) = l(u - \frac{fs}{z(u,v,s,t)}, v - \frac{ft}{z(u,v,s,t)}, 0, 0) \tag{14}$$

Thus, any frame captured from a camera can be expressed as a function of the frame captured by the reference camera (for example camera t_0 in Figure 6).

As showed by [6], using this relation, the 4D Fourier transform of the Light field function $l(u,v,s,t)$ for a constant depth z_0 gives:

$$L(\Omega_u, \Omega_v, \Omega_s, \Omega_t) = 4\pi^2 L'(\Omega_u, \Omega_v)\delta(\frac{f}{z_0}\Omega_u + \Omega_s)\delta(\frac{f}{z_0}\Omega_v + \Omega_t) \tag{15}$$

Consequently, the spectral support of the Light field signal $l(v,t)$ is defined by the line $\frac{f}{z_0}\Omega_v + \Omega_t = 0$. Thus, the spectrum of the Light field signal for a scene with non-constant depth is bounded by two lines $\frac{f}{z_{min}}\Omega_v + \Omega_t = 0$ and $\frac{f}{z_{max}}\Omega_v + \Omega_t = 0$ (Figure 7). Since the depth of the scene is limited by $z_{min} \geq f$, the spectrum support is limited by the minimum slope $\frac{\Omega_v}{\Omega_t} = -\frac{f}{z_{min}} \geq 1$, so a reconstruction filter can be defined.

Signal reconstruction

The idea presented in [6] and illustrated in Figure 8 consists in decomposing the spectral support into multiple layers and use a reconstruction filter for each layer to allow a lower sampling rate than when using only one filter.

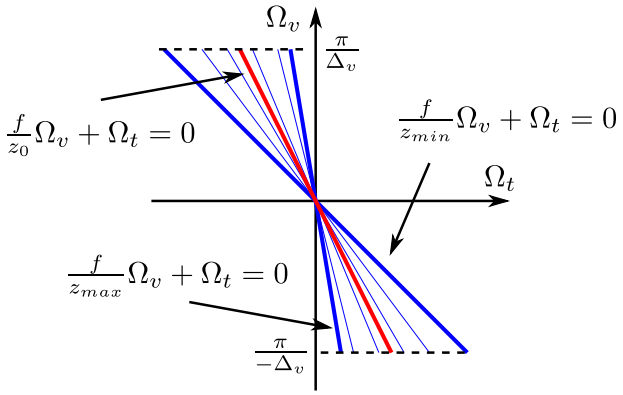


Figure 7: Fourier transform $L(\Omega_v, \Omega_t)$ of the 2D Light field signal $l(v,t)$. Each depth in the scene gives a line in the spectrum. The spectrum is bounded by the lines given by the minimum and maximum depths z_{min} and z_{max}.

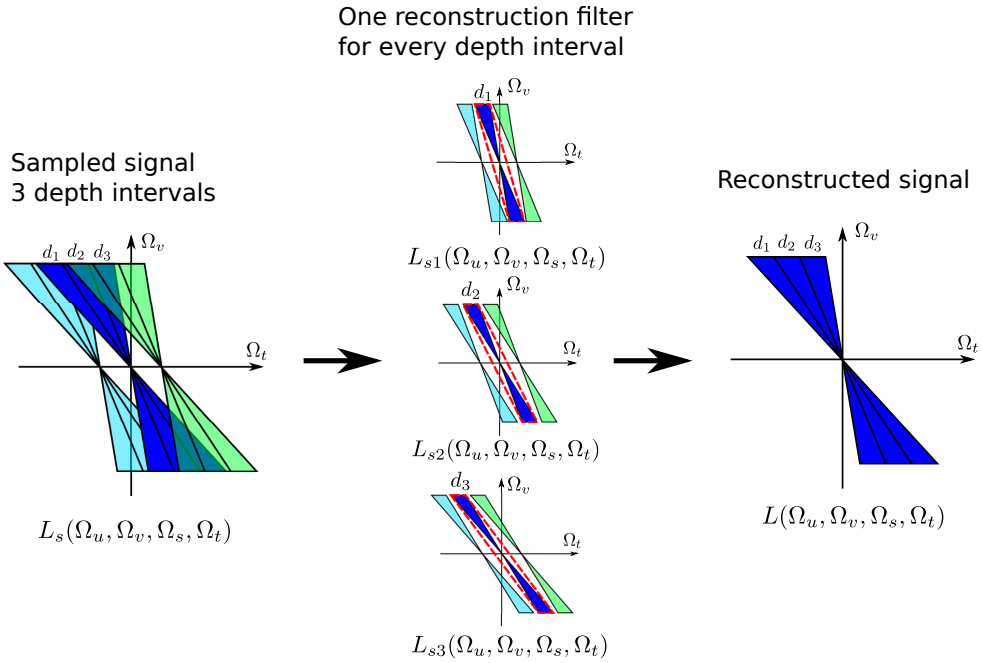


Figure 8: Example of reconstruction of a 2D Light field function using 3 layers. The scene can be decomposed in 3 scenes with 3 different depth intervals $d_1 = [z_2, z_{max}]$, $d_2 = [z_1, z_2]$ and $d_3 = [z_{min}, z_1]$, which give three Light field functions l_{s1}, l_{s2} and l_{s3}. By filtering each of those signals and summing them, the initial Light field functions $l(v,t)$ can be reconstructed even though there were aliasing on the global spectrum.

2.4.2 Concentric mosaic

To apply similar results as in a planar setup to a circular setup, we need to find a parametrisation of a light ray depending on the distance from the ray origin (point of the surface) to the cameras. In a circular setup, we can consider the distance to the centre of the capture circle. In this case, the position of a camera on the circle is defined by β while the pixels are indexed by α. For simpler notations, I will only consider a 2D setup (corresponding to the top view), such that α has only one dimension. A light ray emitted by a point of the surface can be represented by the coordinates (r, ϕ, θ) and a light ray captured by the camera is defined by (α, β).

In [4,27], an outward looking concentric mosaic is considered (Figure 9). The conclusions are the same than for the planar setup, i.e. we can determine the plenoptic function spectrum boundaries in function of the scene depth, but the approach is different. In the previous part, we associated the plenoptic function $l(v,t)$ to the function in another pixel v' of the reference camera in $t = 0$‘ to get $l(v,t) = l(v',0)$. Here, the strategy is to associate the plenoptic function $l(\alpha,\beta)$ the surface plenoptic function $l_s(s,\theta)$ where $s(r,\phi)$ is the captured surface.

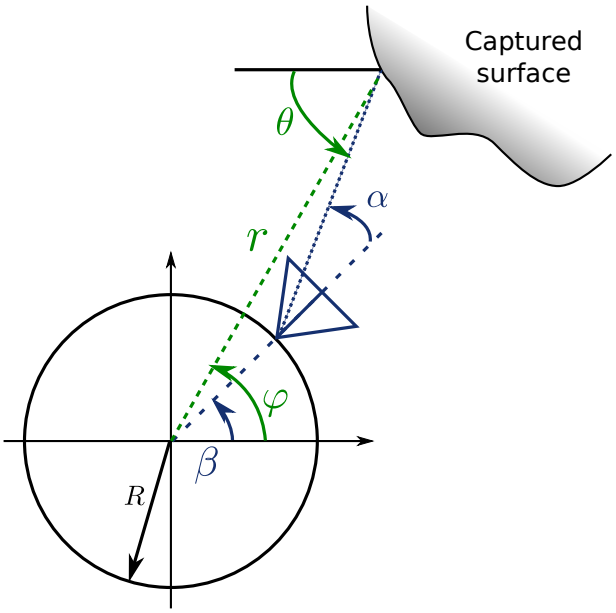


Figure 9: Outward looking concentric mosaic. The cameras are placed on a circle and capture the scene outside the circle.

A surface at constant depth is defined by:

$$\begin{cases} r = r_0 \\ \varphi = \frac{s}{r_0} \end{cases} \tag{16}$$

From Figure 9, we can establish the following relations:

$$r_0 - \frac{R\ sin\ \alpha}{sin(\alpha+\beta-\phi)} = 0 \tag{17}$$

$$\alpha + \beta + \phi = \theta \tag{18}$$

Provided that the field of view of a camera is limited, we can assume $sin\ \alpha \approx \alpha$ and since the scene must be outside the circle, we have $|\alpha + \beta - \varphi| < \alpha \Rightarrow sin(\alpha + \beta - \varphi) \approx \alpha + \beta - \varphi$.

We can deduce:

$$s = r_0(\alpha + \beta) \tag{19}$$

$$\theta = \alpha + \beta + \phi \tag{20}$$

This parametrisation enables then, with a substitution in the plenoptic function spectrum to find that the slope of the spectrum is always less than 45° [27].

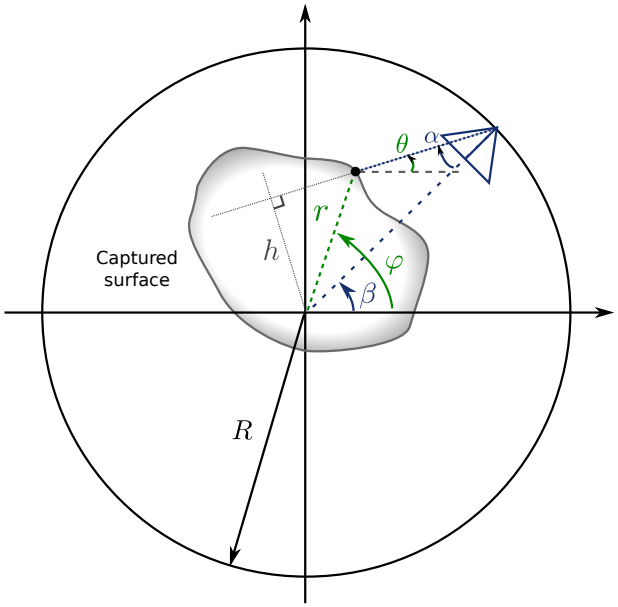


Figure 10: Inward-looking concentric mosaic. The cameras are placed on a circle and capture the scene inside the circle.

In the case of an inward-looking concentric mosaic, the equations do not change much. From Figure 10 we have:

$$h = R\ sin\ \alpha \quad (21)$$
$$= r\ sin(\alpha + \beta - \varphi) \quad (22)$$

Hence the equations:

$$r_0 + \frac{R\ sin\ \alpha}{sin(\alpha + \beta - \phi)} = 0 \quad (23)$$
$$\theta = \alpha + \beta \quad (24)$$

The issue when the scene is inside the circle is that $|\alpha + \beta - \varphi| < \alpha$ does not apply anymore. Indeed, for many point positions we will have $\beta - \varphi \gg \alpha$.

Thus, the simplification $sin(\alpha + \beta - \varphi) \approx \alpha + \beta - \varphi$ is not possible or would restrict us to a very small part of the circle. With this parametrisation, the plenoptic function spectrum can not be computed easily anymore and we have no indication on the spectrum boundaries to estimate a limit sampling rate and thus, the theoretic number of cameras needed for our setup.

In [28], Zhang confirms that the mapping between $l(\alpha, \beta)$ and the surface plenoptic function $l_s(s, \theta)$ has rarely an explicit form. It is however possible, if we know the object shape, to perform stochastic sampling on the light rays from surface Plenoptic sampling and record their corresponding (α, β) coordinates to count the frequency of the light rays falling into each pixel α of a camera β and deduce how to place the cameras.

In conclusion, we could not find a way to compute how many cameras should be used for a capture with our setup using the Plenoptic sampling theory.

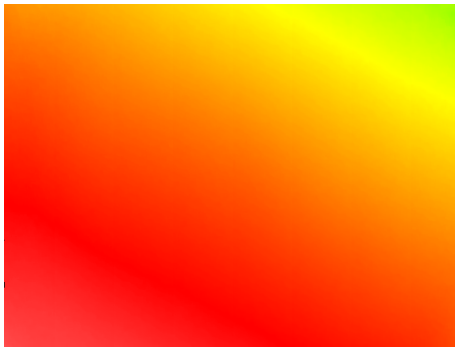
3 Kinect calibration

3.1 Using several Kinects at the same time

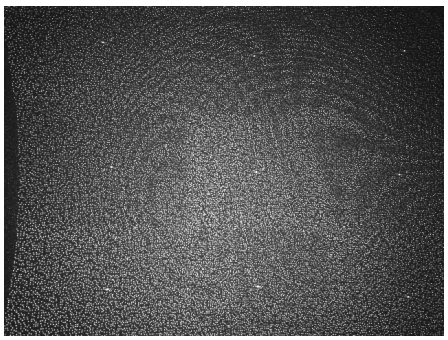
The calibration technique described in Section 2.2 assumed a single depth sensor and multiple colour cameras. In this work, several depth and colour cameras are used simultaneously which introduces several issues.

3.1.1 Interference

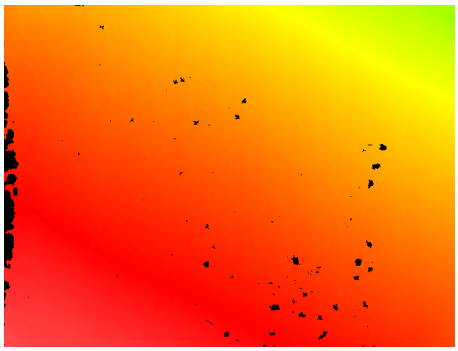
The main issue is that the Kinects v1 interfere with each other. One Kinect sees indeed the points of the other Kinect's projected pattern which creates different sorts of effects, like "holes" in the disparity map (Figure 11).



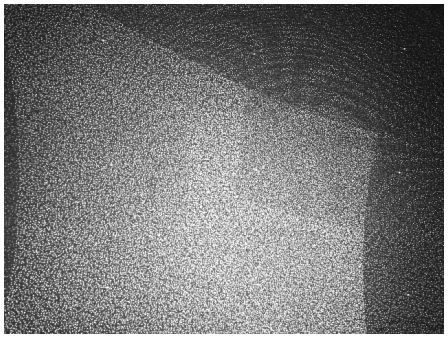
(a) Disparity map for one Kinect v1.



(b) IR projected pattern for one Kinect v1.



(c) Disparity map for two Kinects v1.



(d) IR projected pattern for two Kinects v1.

Figure 11: Figures 11a and 11b show the depth map output by a Kinect v1 when looking at a wall and the infrared points pattern it sees. Figure 11c shows that when a second Kinect points to the wall, it creates holes in the disparity map output by the first Kinect. On Figure 11d, we can see that the infrared patterns of both Kinect superimpose.

In [14], two methods are presented to eliminate interference between two Kinects. The first one is to turn off the illuminator via electronics, which is not safe since it requires interfacing with a laser diode. The second method is to build a set of mechanical shutters that hides the IR projectors with a distinct phase which would require the appropriate equipment.

For the calibration, those interference holes do not really matter as long as there are enough valid depth values to calibrate the depth camera properly. But to reconstruct a scene, we do not want holes in our generated image. The good point is that using several Kinects makes it unlikely that both generated point cloud have missing values at the same place. Moreover, placing the Kinects far from each other reduces this effect.

To remove the holes, a morphological operator can also be used since the holes are quite little and sparse. This works fine with an opening with a circular structuring element of an adapted size.

With two Kinects v2, however, we could not create interference even by placing them on top of each other or looking straight at each other. This can be explained by comparing the frame rate to the

observation duration. Kinect v2 captures a frame every 33 ms. To know how far a point is with the Time of Flight technology, an infrared signal is sent and comes back to the camera. the observation duration must then be at least as long as the light return trip from camera to furthest point in the scene. Since the maximum distance is limited to 4 metres, the camera should observe the scene for $2 \cdot 4\ m \cdot 3 \cdot 10^{-8} m/s = 0.24\ \mu s \ll 33\ ms$. So the probability that both non-synchronised Kinects observe the scene at the same time is very low.

3.1.2 Synchronisation

For the system calibration, the synchronisation is not an issue since the scenes we capture are still. But ideally, the calibration should be done by capturing a sequence and using the images of the sequence. And the synchronisation is essential to reconstruct a sequence. Eventually, the Kinects will be integrated into IP Studio, to synchronise them and handle their output easily.

3.1.3 Bandwidth

The last limitation is that the USB hubs are limited in bandwidth. I could make a Kinect v1 work on each of my four USB hub. However, on an external USB hub plugged into the computer I could only get one Kinect working properly. Sometimes two Kinects v1 worked but then, not all the channels were transmitted to the computer, for example only the sound and not the image.

However, several Kinect v2 can not be used on one computer because the bandwidth needed is much higher. This makes the use of a tool like IP studio essential to get and synchronise the data.

3.2 Calibration steps

The calibration is done with a sequence of images of a checkerboard (with known size) seen from different distances and with different orientations (Figure 12). From those images, initial values can be computed for the calibration parameters described in part 2.2. Those parameters are then refined using a non-linear minimisation algorithm.

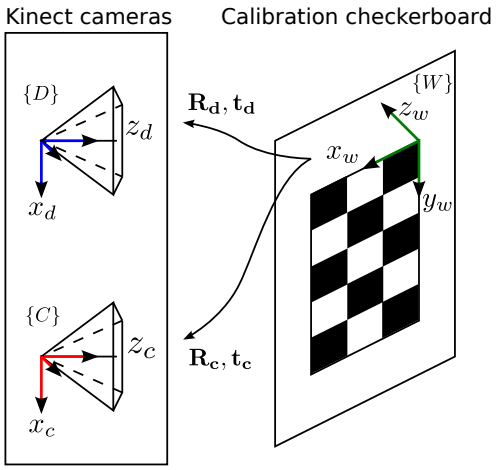


Figure 12: Calibration setup for one Kinect. There are three coordinate systems, $\{W\}$ linked to the checkerboard and $\{C\}$ and $\{D\}$ are respectively the Kinect colour camera and depth camera coordinate system. The calibration gives us $\mathbf{R_c}$, $\mathbf{t_c}$, $\mathbf{R_d}$ and $\mathbf{t_d}$ to go from a coordinate system to another.

3.2.1 Initialization

To get an initial calibration of the colour cameras, Zhang's method [29] is used. First, the corners of the checkerboard are extracted from the image. Knowing the position of the corners in world coordinates, a homography is computed between world coordinates and colour camera image coordinates. Each homography imposes constraints on the camera parameters, so those parameters can be computed by solving a linear system of equations. The initial distortion coefficients are set to zero.

Knowing the position of the Kinect colour camera, the depth camera parameters are then initialised with standard values. The depth camera is translated along the x-axis in the colour camera coordinates system, both depth and colour camera of a Kinect should look in the same direction and distortion coefficients are set to zero.

In this section and further, the world coordinate system will be defined as the camera coordinate system of the first colour camera.

$$K = \begin{bmatrix} f_{cx} & 0 & u_{0c} \\ 0 & f_{cy} & v_{0c} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 590 & 0 & 320 \\ 0 & 590 & 230 \\ 0 & 0 & 1 \end{bmatrix}$$

$\mathbf{R_d} = \mathbf{R_c}$

$\mathbf{t_d} = \mathbf{t_c} + \mathbf{R_c}[-0.025\ 0\ 0]^T$

$[c_0\ c_1] = [3.1121 - 0.0028525]$

$\mathbf{k_d} = [0\ 0\ 0\ 0\ 0]$

where $\mathbf{R_c}$ and $\mathbf{t_c}$ are the rotation and the translation matrix of the Kinect colour camera.

3.2.2 Global refinement

The global refinement step consists in minimising the weighted sum of squares of the measurement reprojection errors over all the parameters.

For colour cameras, the reprojection error is the Euclidean distance in pixel between the position of the checkerboard corners in the colour camera image $\hat{\mathbf{p}}$ and the projection of the checkerboard corners from world coordinate system to colour camera image coordinate system $\mathbf{p}$.

For depth cameras, the reprojection error is the difference between the measured disparity $\hat{d}$ and the predicted disparity d computed from the distance of the checkerboard along the optical axis of the depth camera using Eqs. (8).

Those reprojection errors have different units so they are weighted by the inverse of the corresponding measurement variance (σ_c, σ_d).

Thus, we got the following cost function:

$$c(\mathcal{L}_c, \mathcal{L}_d) = \sum_i e_{ci} \tag{25}$$

Where i indexes an image, i.e. a checkerboard position and $e_c = \frac{\sum_k \|\hat{\mathbf{p}}_\mathbf{k} - \mathbf{p}_\mathbf{k}\|^2}{{\sigma_c}^2}$ with k indexing the corners of the checkerboard.

Finally, we non-linearly minimise this cost function along all parameters with the Levenberg-Marquardt algorithm [16].

3.2.3 Joint calibration

The joint calibration of the Kinects is done in two steps.

In the initialisation step, each Kinect colour camera is calibrated individually using Zhang's method [29]. Then the colour camera parameters are refined using all Kinect colour cameras n by minimising the cost function (26). The depth camera parameters are then initialised using the position of the colour cameras and the parameters are refined for each Kinect n, using its colour camera and depth camera by minimising the cost function (27).

$$c(\mathcal{L}_{c1}, \mathcal{L}_{c2}, ..., \mathcal{L}_{cn}, ...) = \sum_i e_{ci} \tag{26}$$

$$c_i(\mathcal{L}_{cn}, \mathcal{L}_{dn}) = \sum_i e_{ci} + \sum_i e_{di} \tag{27}$$

Where $e_d = \frac{\sum_l \|\hat{\mathbf{d}}_\mathbf{l} - \mathbf{d}_\mathbf{l}\|^2}{{\sigma_d}^2}$ with l indexing the pixels belonging to the checkerboard in the depth image.

In the minimisation step, a non-linear minimisation is applied on the cost function (28) along all the parameters.

$$c(\mathcal{L}_{c1}, \mathcal{L}_{c2}, ..., \mathcal{L}_{d1}, \mathcal{L}_{d2}, ...) = \sum_i \frac{\sum \|\hat{\mathbf{p}}_\mathbf{i} - \mathbf{p}_\mathbf{i}\|^2}{{\sigma_c}^2} + \frac{\sum \|\hat{d}_i - d_i\|^2}{{\sigma_d}^2} \tag{28}$$

3.3 Implementation

To implement the Kinect joint calibration, I used the Matlab toolbox provided by [12] as a basis. Originally this toolbox can be used to calibrate a Kinect jointly with several external RGB camera. I aimed to adapt it for the joint calibration of several Kinects with external cameras and make it work with the standard version of Matlab without additional toolboxes.

3.3.1 The program

Before explaining the modifications made to the calibration toolbox, this section describes how it works and the step that must be followed by the user.

First, the RGB images and disparity map path must be given, as well as the files name format.

Second, the corners of the checkerboard must be selected on every RGB image and the size and number of squares specified. This data will then be used to calibrate the RGB camera as shown in Figure 13.

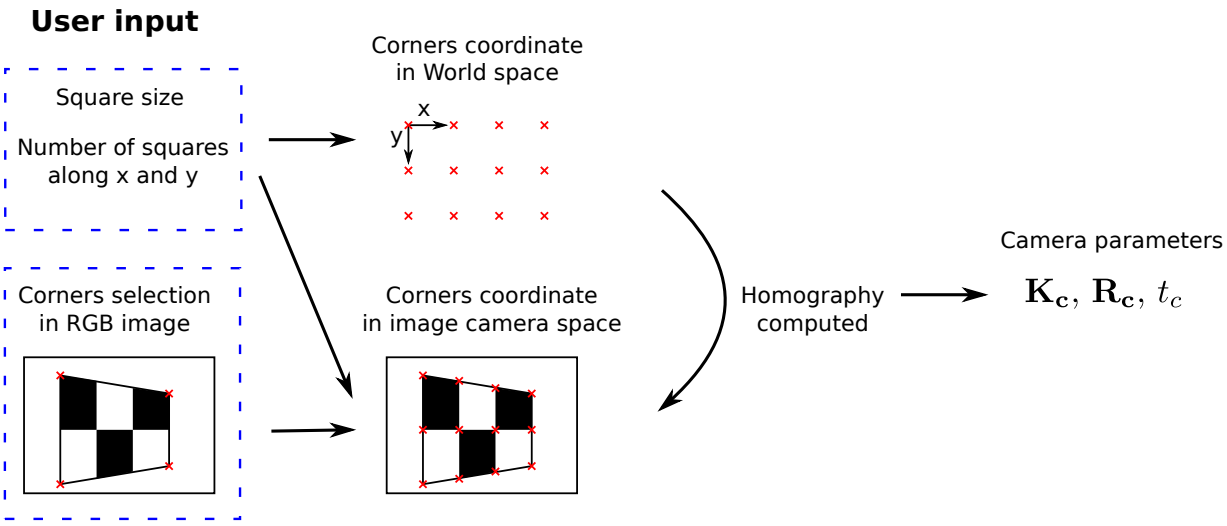


Figure 13: This is an illustration of Zhang's calibration method [29]. If the position of the checkerboard is known in world space, then the parameters of the camera can be computed from the homography between the checkerboard in world space and its camera image.

After the RGB camera calibration, a new world space can be defined as the reference camera space (coordinate space of the first colour camera). And we get the position $\mathbf{R}_{ext}(i), t_{ext}(i)$ of the checkerboard in each frame i.

Then the user has to select the plane to which the checkerboard belongs to on the depth images. The region selected will be used as a mask to know which point of the disparity map can be used to compute the disparity error as shown in Figure 14.

From that point, the rest of the calibration can be done.

3.3.2 Modifications

Supporting several kinects

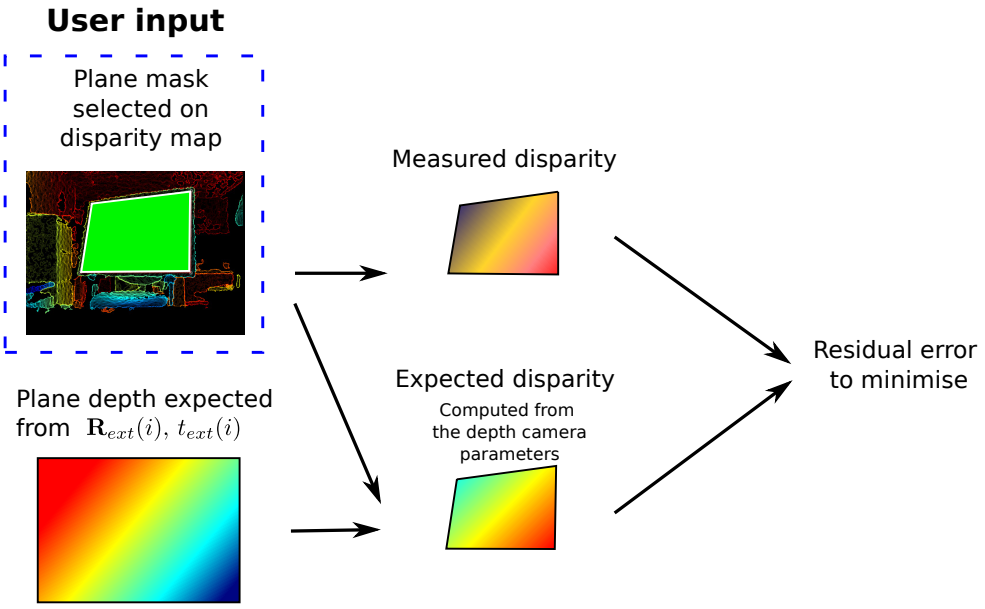


Figure 14: The plane selection indicates the points that can be used to compute the disparity error and then refine the depth calibration.

The first change to make was to modify the data structures to be able to read, store and process the data of several Kinects.

Then I had to adapt the initialisation of each depth camera position, with regard to the colour camera of the same Kinect as explained in Section 3.2.1. To refine this initial calibration, I ran the existing refinement on each depth camera, one after the other, using its corresponding RGB camera images.

Computation of the expected disparity

With several Kinects, the initial computation of the disparity error is no longer adapted. To compute this error, we need to estimate the disparity on the checkerboard to compare it to the disparity measured on the disparity image as shown in Figure 14. To compute this expected disparity, we have (R_{ext}, t_{ext}) which give the position of the checkerboard in world. From those, the normal to the plane N_w and the distance from the plane to the world origin d_w can be computed. The distance from the plane to the depth camera is then deduced by finding the normal N_d and the position d_d to the plane in the depth camera coordinate system. Those are computed with equation (29).

$$\begin{aligned} N_d &= {R_d}^T.N_w \\ d_d &= -t_d \cdot N_w + d_w \end{aligned} \tag{29}$$

Finally, from N_d and d_d, we can get the depth for each point of the plane and compute a disparity value with the calibration parameters.

Fitting a plane to a point cloud

The calibration toolbox used was coded with the image processing and the optimisation Matlab toolboxes. Without them, some functions had to be reimplemented to perform the optimisation.

To correct the depth distortion, the calibration toolbox uses depth images of the plane without a corresponding RGB image. For those frames, the position of the plane can not be deduced from the checkerboard corners so it must be computed from the 3D points reconstructed thanks to the depth camera calibration parameters and the depth map.

The original function used a Principal Component Analysis (PCA) to determine the normal to the board's plane. Indeed, the variance of the points along this axis is minimal, so the normal is given by the third eigenvector of the point cloud's covariance matrix.

Without the optimisation Toolbox, a PCA can not be applied anymore, so the Singular Value Decomposition (SVD) is used instead. The point cloud $\mathbf{X}$ is in the world coordinate and the centre of the point cloud c can be computed by calculating the mean point. Let $\mathbf{p_i}$ be the points of the point cloud and N the number of points. The centre of the point cloud is $\mathbf{c} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{p_i}$ and the matrix $\mathbf{A} = \frac{1}{N}[\mathbf{p_1} - \mathbf{c}, \mathbf{p_2} - \mathbf{c}, \ldots, \mathbf{p_N} - \mathbf{c}]$ can be defined.

The SVD $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{T}^T$ gives us the desired information. Indeed, the columns of $\mathbf{U}$ give, in decreasing order, the direction of maximum variation of the data [20]. The normal $\mathbf{n}$ to the plane corresponds then to the direction of least variation, given by the last column of $\mathbf{U}$ and the distance d from the plane to the origin can then be computed by $d = \mathbf{n} \cdot \mathbf{c}$.

Levenberg Marquardt Algorithm

The second function which had to be re-implemented is the Levenberg–Marquardt Algorithm (LMA) which is used to refine the cameras calibration parameters. To do so, a Matlab implementation of the Levenberg-Marquardt-Fletcher algorithm [2, 3] was used as a basis. Since this algorithm is more complex and did not give improved results on the calibration, I simplified it to perform the basic Levenberg–Marquardt Algorithm.

LMA is an iterative method similar to the Newton iteration method. We have the hypothesised relation $\mathbf{X} = f(\mathbf{P})$, where $\mathbf{X}$ is a measurement vector and $\mathbf{P}$ a parameter vector. $\mathbf{X}$ is an approximation of the true value $\hat{\mathbf{X}}$ and we seek the vector $\hat{\mathbf{P}}$ that satisfies $\mathbf{X} = f(\hat{\mathbf{P}}) + \epsilon$ such that $\|\epsilon\|$ is minimised [11].

In the present case, $\mathbf{P}$ is the vector of the camera parameters. Those parameters have different dimensions, they can be floats, vectors or matrix, so the Matlab calibration toolbox includes functions to store those parameters from a calibration structure type to a vector and then write the optimised values in the structure again. This includes a flags system to specify which parameters must be written and then read from the vector since we do not want to optimise all the parameters at each stage of the calibration. So first, those functions have to be modified to enable them to write and read the parameters of several depth cameras in and from a vector, making sure that in any case, a value is written and read at the same index of the vector.

In the present case, $\mathbf{X}$ is the vector of the weighted residual error for each disparity and colour images and we want to find $\hat{\mathbf{P}}$ such that $\mathbf{X}$ is as close as possible to $\hat{\mathbf{X}} = 0$.

Supposing that f is locally linear, we have $f(\mathbf{P_1}) = f(\mathbf{P_0}) + J\mathbf{\Delta}$, where J is the Jacobian matrix $\frac{\partial \mathbf{X}}{\partial \mathbf{P}}$. At each iteration, we seek the parameter vector $\mathbf{P_{i+1}} = \mathbf{P_i} + \Delta_{i+1}$ such that $\mathbf{X_i} - f(\mathbf{P_{i+1}}) = \mathbf{X_i} - f(\mathbf{P_i}) - J_i\mathbf{\Delta_{i+1}} = \epsilon_\mathbf{i} - J_i\mathbf{\Delta_{i+1}}$ is minimised.

In the Newton method, $\mathbf{\Delta_{i+1}}$ is obtained by minimising $\|\epsilon_\mathbf{i} - J_i\mathbf{\Delta_{i+1}}\|$ which is a linear minimisation problem. Thus $\mathbf{\Delta_{i+1}}$ can be obtained by solving the equation $J_i^T J_i \mathbf{\Delta_{i+1}} = J_i^T \epsilon_\mathbf{i}$. In LMA, this equation is modified to get around a badly conditioned $J^T J$ matrix. In the original calibration toolbox, the equation used by the Levenberg-Marquardt function from Matlab is Eqs. (30) [16], so I implemented this version of LMA. However many variations exist [9].

$$(J_i^T J_i + \lambda I)\mathbf{\Delta_{i+1}} = J_i^T \epsilon_\mathbf{i} \tag{30}$$

The value λ is initialised to some value (Matlab defaults value $\lambda = 0.01$ is used) and then for each iteration, if Δ leads to an increased error, λ is multiplied by 10 or if the error decreases, it is divided by ten. The algorithm is stopped when $\mathbf{P_i}$ converges (i.e. $\|\mathbf{\Delta_i}\|$ is close enough to zero), $\|\mathbf{X_i}\|_2$ is close enough to zero or when the maximum number of iteration has been reached. The code of the function can be found in Appendix C.

As shown in Section 3.4.3, this implementation of the algorithm is slower than the one from the Matlab optimisation toolbox but the residual errors are similar.

Interference

On the first calibration done with several Kinect, when there were too many holes in the depth map, the calibration did not converge. So the code was modified to systematically reject the points with invalid disparity values. The consequence is that there are fewer points used to calibrate the depth camera in this situation so the reconstructed point cloud is noisier.

Reprojection

Finally, a function was written to reproject a 3D point on any colour camera. Thus, the external RGB cameras can be used to colour the point cloud and different combination can be tested (see Section 4.4).

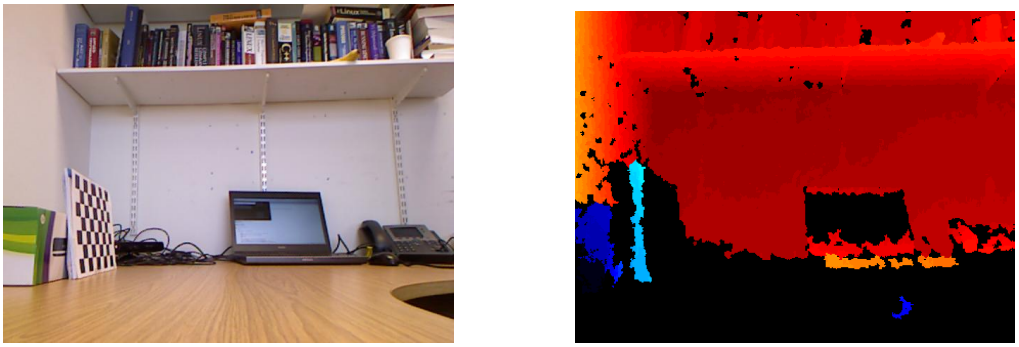
3.4 Application

I had at my disposition four Kinect v1 and one Kinect v2. In a first phase, captures were done with several Kinects v1 while the Kinect v2 was used as external RGB camera.

3.4.1 Checkerboard

To capture the data sets, scripts calling libfreenect and libfreekinect-record were written, associating the corresponding frames between the cameras and renaming the files accordingly.

Several data sets could actually not be calibrated for different reasons. Indeed, as told before, two Kinects have to be far apart enough to prevent interference between the projected pattern which leads to holes in the disparity map. But the cameras have to be close enough, such that both can see properly the checkerboard (Figure 15). If the Board is too tilted or not enough, then we do not get enough usable images where several cameras can see the plane to calibrate the system properly.



(a) RGB image (b) Disparity map

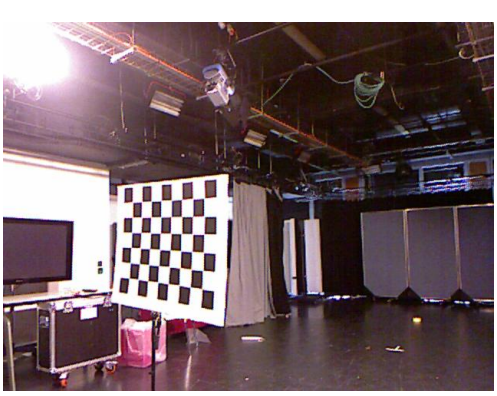
Figure 15: This Figure shows what happens when the checkerboard is too tilted with respect to the Kinect. On the RGB image, the inner corners of the checkerboard can not be detected properly, while on the disparity map, it is not even visible, probably because the angle of view makes its surface perfectly reflective to the IR rays.

We had a similar issue with the first calibration sets we captured. When the checkerboard is tilted, there are holes in the disparity map at the position of the black squares (Figure 16). This is probably because those squares are too reflective for the IR light. Consequently, the light from the IR projector is not diffused toward the IR camera, so the point pattern is not visible in those places. The same phenomenon is also observed with the Kinect v2.

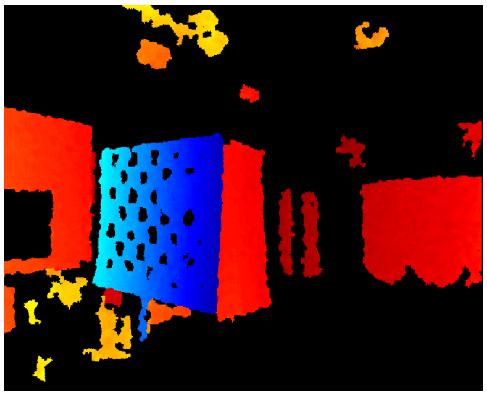
In conclusion, the checkerboard must be chosen carefully to enable a capture from a wide angle of view by both the RGB and the IR camera.

3.4.2 Lighting

The lighting caused trouble as well since the Kinect camera gave images with very different luminosity and white balance according to their position in the scene (Figure 17). Here, using external RGB cameras



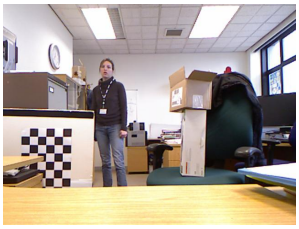
(a) RGB image



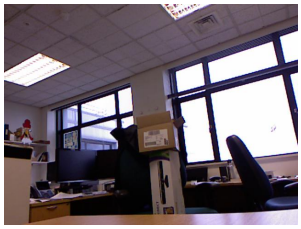
(b) Disparity map

Figure 16: There is no depth information on the black squares when the checkerboard is tilted.

is particularly interesting because their sensitivity can be fixed to prevent these effect and their output can be used to colour the points instead of using the Kinect RGB images.



(a) Kinect 0 (v1)

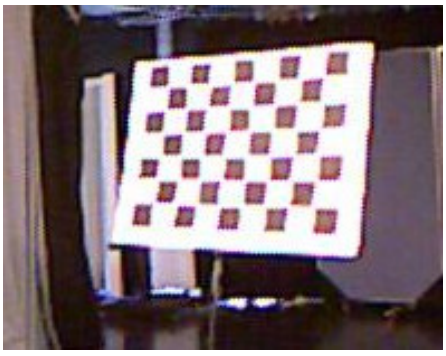


(b) Kinect 1 (v1)

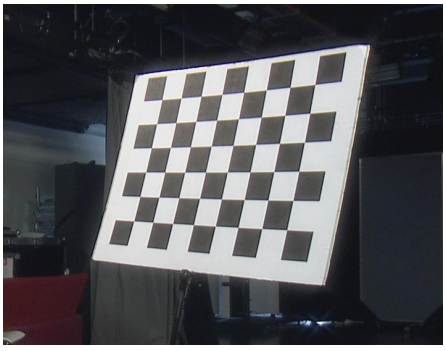
(c) Kinect 2 (v2)

Figure 17: When several Kinects are looking at the same scene, they all give RGB images with different white balance and luminosity.

Another issue caused by the light is that when the checkerboard is far from the camera and strongly illuminated, a part of the checkerboard is overexposed and the inner corners of the corner can not be located precisely (Figure 18), which deteriorates the calibration accuracy. This shows the importance of the lighting for the calibration.



(a) Kinect v1



(b) RGB camera

Figure 18: Same scene captured by a Kinect RGB camera (18a) and by a standard RGB camera (18b). On 18a, it looks like the black squares do not touch each other.

3.4.3 Results

As a conclusion, some criteria for a good calibration can be established from those captures. 15 plane pose with different inclinations seems to be sufficient for a good calibration but the checkerboard must be

seen properly and entirely by the colour camera and there must not be too many holes in the depth map of the plane.

Calibration performances

I tested the calibration on different data sets. The first one "Herrera" is the set of image provided with the Matlab calibration toolbox. Data set A is a data set captured with two Kinects v1 and one Kinect v2 positioned as shown in Figure 19. The Kinect v2 is placed on one Kinect v1 since it is used as external RGB camera at first.

In order to see the influence of the interference between Kinects on the data set A, I captured 2 depth image per Kinect for each plane position: one with the IR projector of the other Kinect occluded (no interference) and one image with interference. From now on, I will mainly use the images without interference, excepted when I precise the opposite.

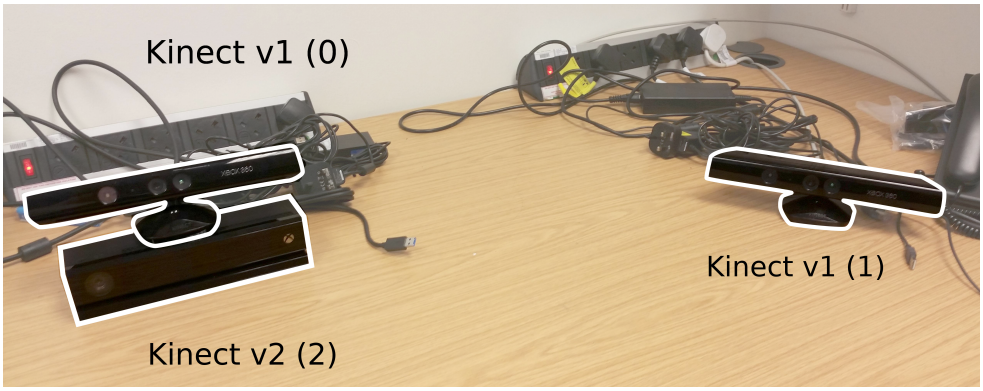


Figure 19: Data set A was captured by two Kinects v1 and one v2 and the Kinects are indexed from 0 to 2 (between brackets).

Table 1 compares the residuals errors between the original calibration toolbox and the modified version with the optimisation function re-implemented. As expected, the results are similar. However, the reimplementation of the optimisation is slower than the original Matlab implementation (about twice the time on Herrera's data set).

Calibration toolbox	Colour	Depth
Herrera's toolbox	0.28 ±0.02 px	0.77 ±0.002 kdu
Modified toolbox	0.32 ±0.02 px	0.784 ± 0.004 kdu

Table 1: Comparison of the residuals errors between the original calibration toolbox and the modified version for one Kinect on Herrera's data set.

Table 2 shows the results on one data set with different number of Kinects. We can observe that the duration of the calibration increases very quickly with the number of Kinects and is much higher for the Kinect v2.

Data set	Nb Kinects	Duration	Colour 1 ±0.04 px	Depth 1 ±0.002 kdu	Colour 2 ±0.04 px	Depth 2 ±0.003 kdu	Colour 3 ±0.03 px	Depth 3 ±0.004 kdu
H	1	2mn 17	0.32	0.784				
A	1	4mn 52	0.26	0.852				
A	1	4mn 15			0.29	1.136		
A	1	10mn 27					0.30	0.996
A	2	11mn 43	0.57	1.044	0.50	0.83		
A	3	33mn 42	0.57	0.998	0.58	0.869	0.44	1.076

Table 2: Residuals errors of the cameras after calibration on Herrera's dataset (H) and on my data set A with different number of Kinects.

Table 3 shows the residual errors on one data set with and without interference. The results are not so good with the interference and worse for the depth camera 1 than for the second one. This can be caused by the holes created by the interference, which are more present in the depth image of the Kinect 0 (Figure 20).

Data set	Colour 1	Depth 1	Colour 2	Depth 2
	±0.02 px	±0.006 kdu	±0.02 px	±0.004 kdu
A	0.57	1.044	0.50	0.83
A (interference)	0.69	2.090	0.57	1.132

Table 3

(a) Depth map seen by the Kinect 0 (b) Depth map seen by the Kinect 1

Figure 20: Same scene seen by two interfering Kinect v1.

3D reconstruction

In order to visualise the results, the reconstructed 3D points were first plotted in Matlab. Indeed, from a depth image and the calibration of the depth camera, a 3D point $\mathbf{x_w} = [x_w, y_w, z_w]^T$ can be computed for each pixel $[u_d, v_d]^T$ of the depth image as shown in Section 2.2.2.

To get the colour of a 3D point, it can be projected in a colour camera image coordinate system using equations (1) to (4). The point's colour is then the colour of the pixel obtained.

The colour can also be bilinearly interpolated between the neighbouring pixels of the projection point but this takes more time and no difference was visible in the rendering.

Figure 21 shows the point clouds obtained from the data set A with and without interference. We can see the difference of white balance between the two colour cameras and we can see on the checkerboard that the calibration without interference is more precise. In Figure 21c, the same calibration is used as in Figure 21b but the depth map used for the reconstruction has holes too because of interference. We can see the location of the holes through the colour contribution of each Kinect but there are no actual holes in the reconstructed checkerboard since the holes are at different locations in the depth map of each Kinect.

Now that we have a working calibration and can build a point cloud from the Kinects, several methods will be tested to achieve a rendering of those data as realistic as possible.

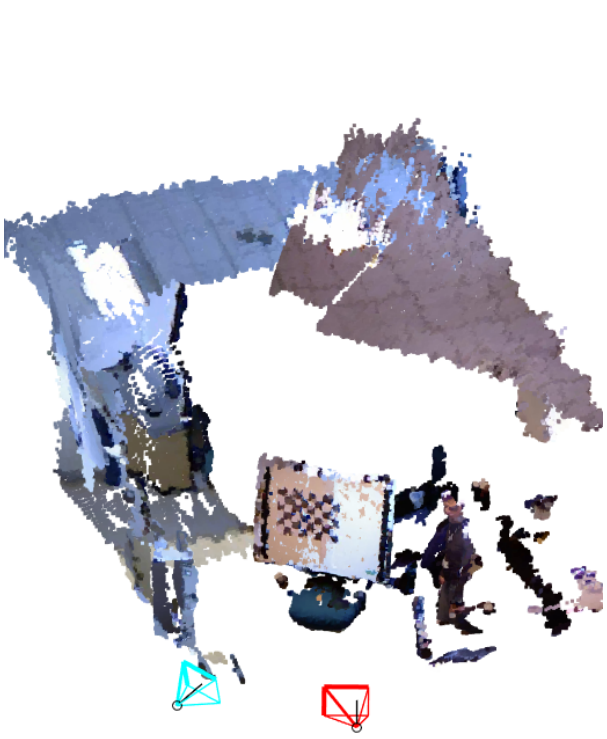
4 Off-line rendering

In a first phase, I studied several rendering techniques without considering the time constraint.

Since the colour of the points captured by the cameras on the opposite side of the circle do not give any relevant information, I will only use the cameras on the same semicircle as the virtual camera to generate the new point of view. To generate an image from a new point of view, a straightforward approach consists in projecting the coloured 3D point cloud in a virtual camera. This projection implies the two following issues: several points can project on the same pixel while on some pixels, no 3D point projects, leading to holes in the resulting image (Figure 22).

I did most of the tests on Matlab and created some virtual scene in Blender to test the implemented methods and see their limits. I captured the scenes from 360 different angles with a camera rotating

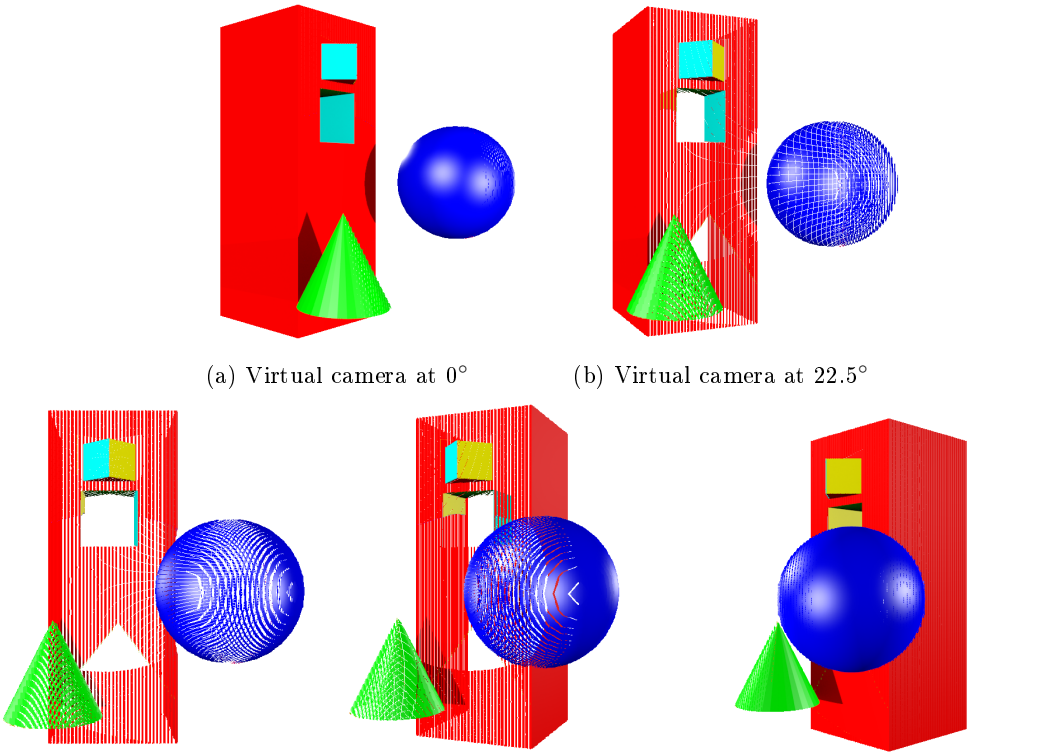
(a) Point cloud obtained with data set A using Kinect 0 and 1 without interference.



(b) Point cloud obtained with data set A using Kinect 0 and 1 calibrated with interference and images without interference.

(c) Point cloud obtained with data set A using Kinect 0 and 1 calibrated with interference and images with interference.

Figure 21



(a) Virtual camera at 0° (b) Virtual camera at 22.5°

(c) Virtual camera at 45° (d) Virtual camera at 67.5° (e) Virtual camera at 90°

Figure 22: Images obtained by projecting the point cloud in the virtual camera at different positions. The two real cameras used to build the point cloud are at 0° and 90°.

around the scene and for each capture, an RGB image is stored and the depth image where the depth is the z coordinate in virtual camera coordinate system. The scene is included in the world space x=[-1,1]m, y=[-1,1]m and the camera is rotating on a circle of radius 2m so the recorded depth values are between 1 and 3. Those depth values are then mapped between 0 and 1 such that they can be stored in an 8 bits grayscale image, which gives a depth resolution of $2m/255 = 7,8mm$.

Working with synthetic data has several advantages. First, I can use any point of view for my reconstruction, as many cameras as I want and I have a ground truth image for each angle of view to compare the generated image with. Second, those sets are simpler since the depth resolution is known and constant and the depths are captured from the same point of view than the colour which would be equivalent to having the colour and depth cameras of one Kinect at the same position. Moreover, I can deduce the camera parameters from the camera parameters in Blender.

Most of the tests have been made with 4 cameras at the positions [0°,90°,180°,270°] or 20 cameras (every 18° from 0° to 359°), and the virtual camera is placed at 45°. The capture cameras are always equidistant on the circle.

4.1 Implementation in Matlab

Since each of the methods presented needs other data, we had to think about the way to store the information. We used lookup tables to get quickly the data in the loop writing the final images and to use operations on tables for more efficiency. The Kinects were indexed by k and most of the data were stored in cell arrays. Then, for each Kinect k, the 3D points were indexed by idx. For example, for an image of size 1024x1024, $idx \in [1, 1024^2]$. Thus, I have:

- The image colour corresponding to the point indexed by idx on the Kinect k: $imc\{k\}(idx)$
- Its depth value recorded by the Kinect k: $imd\{k\}(idx)$
- The table of the valid points indices (valid depth, alpha channel of the colour to 1): $valid\{k\}$. To iterate over the points of a Kinect point cloud, we take then idx in $valid\{k\}$.

- The 3D point reconstructed from $imd\{k\}(idx)$: $X\{k\}(idx) = [x_{id}^k, y_{id}^k, z_{id}^k]$
- The colour of this point by projecting it in the camera cam: $col\{k\}\{cam\}(idx) = [r, g, b]$. Note that $col\{k\}\{k\}(idx) = imc\{k\}(idx)$.
- Its 2D coordinates in the virtual camera image: $X_i\{k\}(idx) = [u_{id}^k, v_{id}^k]$

Beside this I also store information on the camera setup:

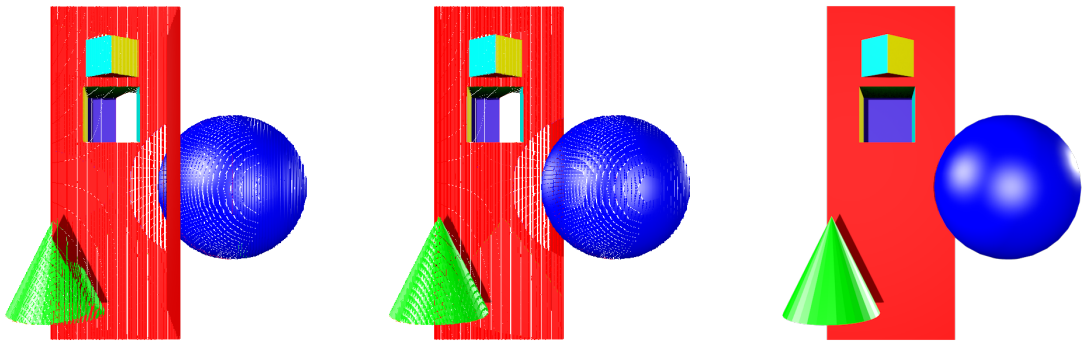
- Real cameras k parameters in a structure: $calib\{k\}.R, calib\{k\}.t, calib\{k\}.K$
- Virtual camera parameters: $vcalib\{1\}.R, vcalib\{1\}.t, vcalib\{1\}.K$
- Angular position of the camera k compared to the virtual camera: $anglevc[k]$

On synthetic data as well as on real data, I will usually use a copy of the fist camera as the virtual camera so $vcalib\{1\}.K = calib\{1\}.K$.

4.2 Handling occlusions

When projecting the point cloud in the virtual camera, several points can project on the same pixel, but we only want to see the colour of the closest point to the virtual camera. Figure 23a shows what we can get when using no depth comparison. The red box is seen in front of the blue sphere and occludes a part of the green cone. This happens because the points are projected one after the other and the points of a cloud are projected in the same order than they are read on the depth image.

To handle this, I create a depth map by storing for each pixel the radial distance of the point projected in this pixel to the virtual camera. Then I only overwrite the pixel with the projection of a new point if this new point is closer to the virtual camera than the previous one. With this method, Figure 23b is obtained.



(a) No depth comparison (b) Depth comparison. (c) Ground truth.

Figure 23: Image reconstructed by reprojection with a setup of 6 cameras.

In pseudo code, the base of the rendering algorithm is then:

```
for all cameras k in the semicircle of the virtual_camera
    for id in valid{k}
        get image projection coordinates (i,j) from Xi{id}
        if dist(X(id), virtual_camera) < depthmap(i,j)
            image(i,j)=col{k}{k}(id)
            depthmap(i,j)=dist(X(id),virtual_camera)
        end
    end
end
```

4.3 Filling holes

The second issue to solve is holes in the resulting image. Indeed, there are some pixels of the virtual camera on which no 3D point projects. We could trace a ray from those pixels and find the nearest point in 3D but this would be expensive and we aim to get an image without working in the 3D space. So I implemented two methods to fill holes in the final image. The first one uses interpolation and the second uses a quad cloud instead of a point cloud and is described in Section 5.

4.3.1 Holes size

Horizontal parallax

Before trying to fill those holes, it is interesting to estimate their size. Since the holes are mainly created by the horizontal parallax, I will do the computations in 2D and the representations in top view. Thanks to Figure 24, the relations (32) and (32) can be established between the width of a square facing a camera and the size of its projection on the camera image in pixels.

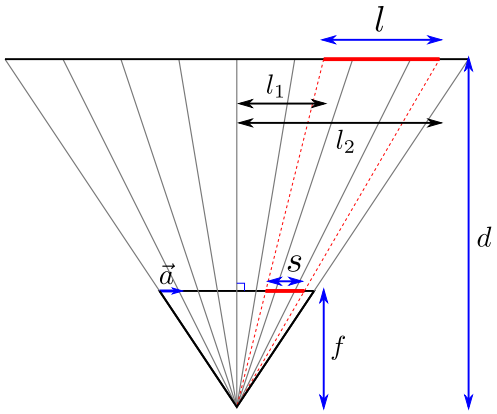


Figure 24: Equation (32) establishes the relation between the width l of a quad facing a camera of focal length f at distance d and the size of its reprojection in pixels s_{pixel}.

$$s = \frac{f}{d} \cdot l_2 - \frac{f}{d} \cdot l_1 = \frac{f}{d} \cdot l \tag{31}$$

$$s_{pixel} = \frac{s}{\|\vec{a}\|} = \frac{f \cdot l}{d \cdot \|\vec{a}\|} \tag{32}$$

With this relation, we can get an idea of the density of our spatial sampling.

For the synthetic data, $\|\vec{a}\| = 3.1250e^{-5}m$ and $f = 0.032m$. Let d be the depth of the point for the camera that captures it. We will consider the limit cases $d = 1m$ and $d = 3m$.

- For $d = 1m$: $s_{pixel} = 1pix \Leftrightarrow l(1) = 0.98mm$
- For $d = 3m$: $s_{pixel} = 1pix \Leftrightarrow l(3) = 2.94mm$
- The depth resolution does not depend on d and is $\Delta d = \frac{2m}{255} = 7.8mm$ (this only applies to the synthetic dataset)

Figure 25 is a scaled representation of the spatial sampling of the synthetic data.

In an ideal case (a valid depth value for each pixel of the depth map), for each ray through a pixel there is a unique depth value. Here this applies, so the maximum distance between two points along the x-axis Δx_{max} depends on three parameters:

- The distance d_f from which the furthest of the two points was captured
- How many depth steps n there are between both points
- From which pixel comes the nearest and the furthest point. Where a pixel is indexed by its angle α compared to the camera optic axis in the horizontal plane.

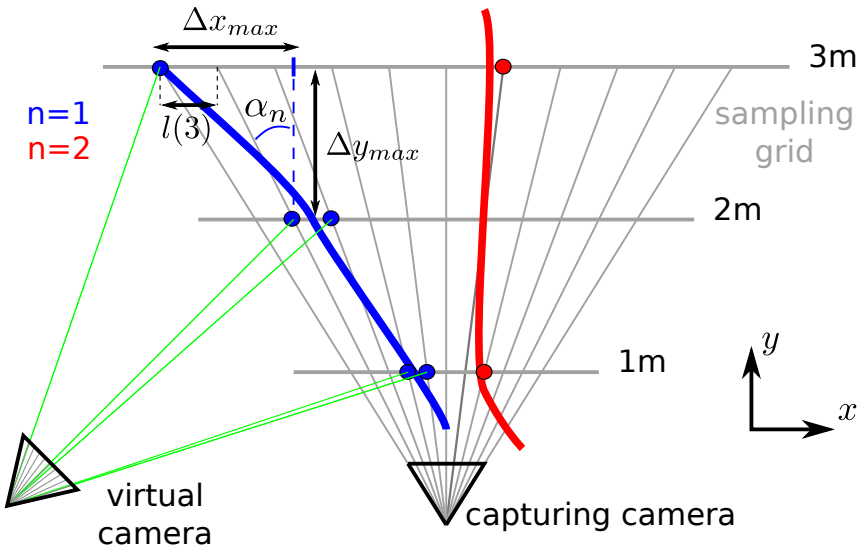


Figure 25: Representation of the spatial sampling of the synthetic data. The red and blue lines represent two surfaces and the discs mark the positions where the surface is sampled.

The distance between two points along the y-axis is then $n \cdot \Delta d$ and is independent of d so we get Equation (33) and (34). The different parameters and those distances are represented in Figure 25.

$$\Delta x(d, n) = l(d_f) + sign(|\alpha_f| - |\alpha_n|) \cdot n \cdot tan(\alpha_n)\Delta d \tag{33}$$

$$\Delta y(d, n) = -sign(\alpha_f - \alpha_n) \cdot n \cdot \Delta d \tag{34}$$

The considered cameras have a field of view of 26.6°. So for $d = 3m$, $n = 1$ and $\alpha = 13.3°$, Equations (33) and (34) give $\Delta x_{max} = 4.8\ mm$ and $y_{max} = 7.8\ mm$. Then the distance effectively seen between two points depends on the orientation of the camera. In the worst case, the perceived distance will be $l_v max = \sqrt{x_{max}^2 + y_{max}^2} = 9.15mm$. This width $l_v max$ will then project on $s_{v,1} = 9.4\ pixels$ if the points are at $d_v = 1m$ from the virtual camera, or $s_{v,1} = 3.12\ pixels$ if the points are at $d_v = 3m$.

The conclusion is that if two neighbouring points have a depth difference lower or equal to Δd (this is equivalent to the condition $n \leq= 1$), then the projection of the point cloud on a virtual camera at one metre or more will create an image with holes no wider than 10 pixels.

Holes in a pixel column

Until then, I only considered the distance between the points in the horizontal plane, but when the virtual camera looks at a couple of points closer than the virtual camera, two points that gave neighbouring pixels on a column of the capturing camera can give two non-neighbouring pixels on a column of the virtual camera. This can be quantified quite easily for two points at the same depth. In the synthetic data set, in the worst can, the virtual camera sees at 1 metres two points that were captured at 3 metres of the capture camera. If the points are seen on two neighbouring pixels by the capture camera, the distance between them in the real world is $l_{hv}(3) = 2.9\ mm$, which projected on the virtual camera gives 3.0 $pixels$.

For each couple of point we could compute on how many pixel they project in the virtual camera image, but it would be a lot of computation. Just associating a distance between points in function of the depth where the point was captured will give us satisfying results.

Kinect v1

For Kinect v1, [25] gives the quantization function Eqs. (35).

$$\Delta d(d_c) = 2.73\ {d_c}^2 + 0.74\ d_c - 0.58\ [mm] \tag{35}$$

Where Δd is the depth quantization step in mm for a point captured at depth d_c.

And from Eqs. (32), we get the equation (36).

$$l(d_c) = \frac{d_c \cdot \|\vec{a}\|}{f} \tag{36}$$

Where f is the focal length of the capture camera and $\|\vec{a}\|$ the size of a pixel.

From this we can determine an average spacing s_w between points that are captured at a depth d_c: $s_w(d_c) = \sqrt{\Delta d(d_c)^2 + l(d_c)^2}$.

This distance can then be projected on the virtual camera to get the points spacing on the camera image $s_{vc}(d_v) = \frac{s_w(d_v) \cdot f}{d_v \cdot \|\vec{a}\|}$.

So this spacing depends on the distance of the points to the capture camera and the virtual camera. Table 4 shows the spacing between points for different distances. If we assume that $d_v = d_c \pm 1\ m$, only the values in the blue cells can be considered.

d_c	$\Delta_d(d_c)$ (mm)	$l(d_c)$ (mm)	Spacing in world $s_w(d_c)$(mm)	Spacing on virtual camera $s_{vc}(d_v)$			
				$d_v = 1m$	$d_v = 2m$	$d_v = 3m$	$d_v = 4m$
1 m	2.9	1.7	3.4	2.0 px	1.0 px	0.7 px	0.5 px
2 m	12	3.4	13	7.3 px	3.7 px	2.4 px	1.8 px
3 m	26	5.1	27	16 px	7.8 px	5.2 px	3.9 px
4 m	46	6.8	46	27 px	14 px	9.1 px	6.8 px

Table 4: Spacing between points for different distances from point to the capture camera d_c (Kinect v1) and virtual camera d_v.

4.3.2 Interpolation

A common method to fill holes is interpolation. However, here a simple interpolation is not a good strategy (Figure 26a). The interpolation links separated objects and the background can be seen through some objects like the cone.

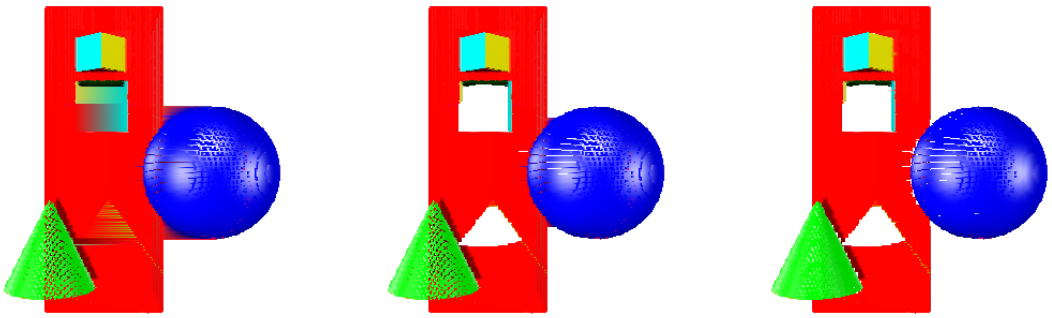
To improve the result, a constraint can by applied: the image should be interpolated only between points that belong to the same surface. This way, only holes due to the parallax will be filled and not the holes corresponding to a lack of information caused by occlusion.

We can say that two points belong to the same surface if there are less than $nbpix$ pixels between both points reprojection, and they have about the same distance to the camera. To interpolate those pixels, we use the depth map recorded for the virtual camera and consider a number of depth intervals between the minimal depth and the maximum depth. For each depth interval, the pixels belonging to this depth intervals are interpolated and the resulting pixels are written on the previous layers.

The parameters have to be chosen carefully. For the synthetic data set, we used $nbpix = 10$ since as shown before, it is the size of the biggest holes in the image and for the depth steps, we consider a depth intervals of size $l_{v,max} = 10mm$ as computed before, which gives quite good results as shown in Figure 26c.

The following pseudo code describes the horizontal interpolation and Figure 27 shows the image computed for each depth interval.

```
img_int = interpolate_horizontal(img, dmap)
 img_int=img
 for each depth interval d
   for each line l
     for each column c
       if depthmap(l,c) is in d and img(l,c) has no colour data
```



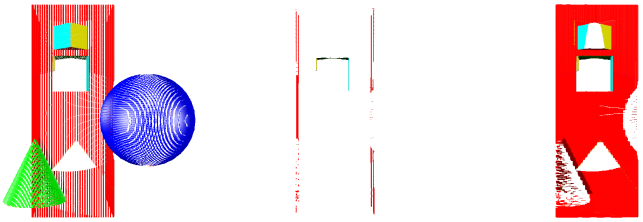
(a) Regular horizontal interpolation.

(b) Only pixels close enough (10 pixels) are interpolated.

(c) Only pixels close enough (10 pixels) and belonging to an interval of 10mm are interpolated

Figure 26: Comparison between different interpolation methods.

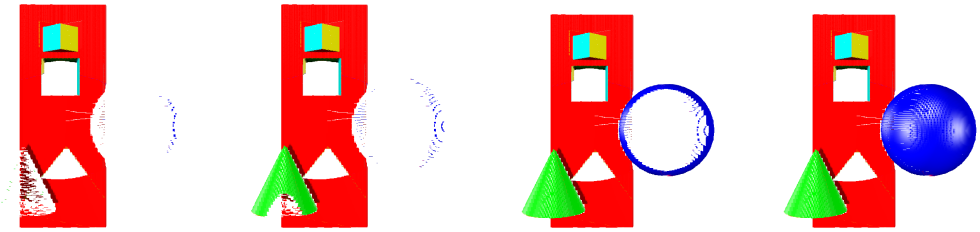
```
        cnext=c
        interpolate img between (l,cprev) and (l,cnext) and write result in img_int
      end if
      cprev=cnext
    end
  end
end
```



(a) Initial image

(b) Depth 6

(c) Depth 5



(d) Depth 4

(e) Depth 3

(f) Depth 2

(g) Depth 1

Figure 27: Here a horizontal interpolation is made on 6 depth intervals. For each depth interval, the image computed before is overwritten with the new interpolated values.

Figure 26c still shows holes because some points are seen closer than the depth at which they were captured. This can be solved by applying a vertical interpolation on the image obtained after the horizontal interpolation. To do so, we use the interpolated depth map, computed at the same time as the colour image horizontal interpolation. With $nbpix = 4pixels$, I get figure 29a with no holes.

Figure 28 shows the results on real data from two Kinects v1. The virtual camera is placed between both capture camera. I used the maximum values of Table 4 to set the maximum pixel distance and depth difference to interpolate between two pixels: 10 pixels and 50mm, and we get an image with no holes. This image was interpolated in about 4mn30.

Even though this method gives good results, we can not use it to make a real-time rendering . That's why in Section 5 we will use quads of varying size to fill the holes.

(a) Image obtained by simple reprojection

(b) Image interpolated with the parameters 10 pixels, 50mm

Figure 28: Application of the interpolation method on Kinects v1 data.

4.3.3 Conclusion

Projecting a reconstructed 3D point in a virtual camera and interpolating the resulting image is actually very similar to the pixel warping method mentioned in Section 2.3. The difference here is that we compute the 3D world coordinate of the point before reprojecting it in the virtual camera. Thus, we can get the depth information to know which pixel from one image should overwrite a pixel of the other image.

Moreover, we only perform the interpolation once all the information in the image is available. In [22], they need to compute an interpolation every time a pixel is moved and the pixels have to be moved in a certain order to prevent occlusions. Here, the depth map enables us to solve both these issues at the same time: we can interpolate all the image at once in any order, without creating occlusions.

One reason why they probably did not go through the world coordinates is that they wanted a simple and efficient hardware implementation since the hardware performances were much lower at the time when that paper was published.

4.4 Colouring points

For more realism, I tested several methods to colour the 3D points from the colour camera images. To refer to them more easily, I named them A to E. In this section, I interpolated the images got with the interpolation methods for a better visualisation of the results.

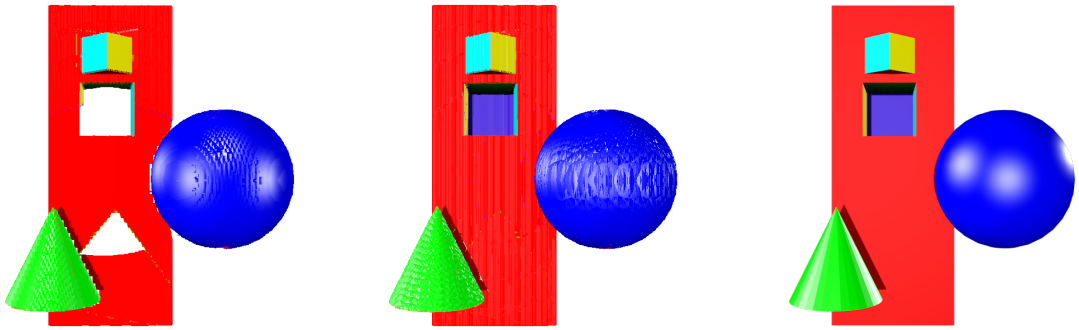
4.4.1 Kinect colour camera (Method A)

The simplest method consists in picking the colour of the point cloud reconstructed from one Kinect depth map in the RGB image output by the same Kinect. It is easy to implement, fast and with this method, colours of a non-visible object do not appear. The only issue is that two neighbouring 3D points belonging to the same surface but captured by different Kinects can be captured with a different lighting depending on the orientation of the camera with respect to the normal of the observed surface. This issue can appear when there is a specular reflection on the object, for example on the green cone and the blue sphere on Figure 29a and this is mainly an issue when a lot of cameras are used (Figure 29b).

To prevent this lighting issue, it makes sense to try colouring neighbouring points with the same RGB image. The question then is, what is the best colour camera to do it.

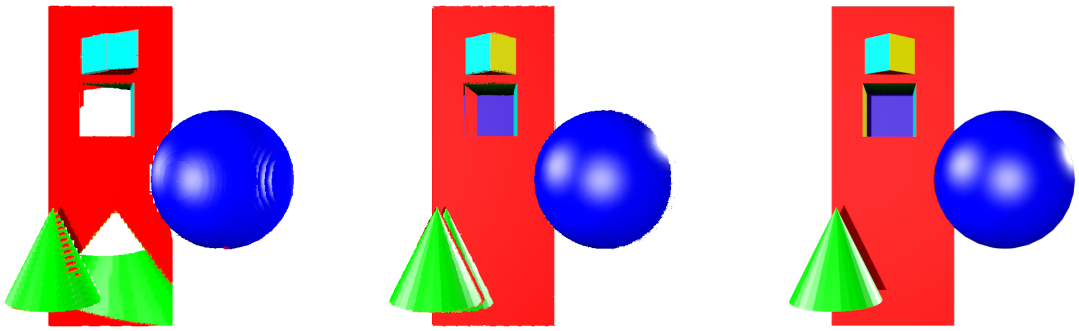
4.4.2 Closest colour camera (Method B)

The first solution I tested is to colour all of the point clouds with the colour camera the closest to the virtual camera (in angle). To do so, the closest camera is selected and all the 3D points reconstructed from all the Kinects are projected in it to get their colours. The results are shown in Figure 30.



(a) 4 cameras (b) 20 cameras (c) Ground truth

Figure 29: Here each 3D point was coloured using the RGB image of the Kinect that captured the point. Figure 29a was reconstructed using 4 cameras and Figure 29b using 2D cameras. Figure 29c is the same point of view (virtual camera at 45°) captured on Blender.



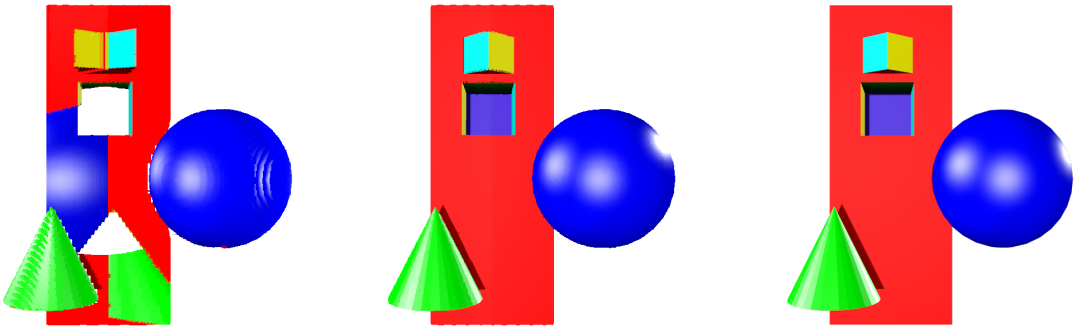
(a) 4 cameras (b) 20 cameras (c) Ground truth

Figure 30: Here each 3D point was coloured using only the colour camera the closest to the virtual camera for 4 and 20 cameras.

This suppresses the issue of the "specular noise" visible with the previous method and the sphere shows correct light reflection when there are enough cameras, but another issue appears. When the background is occluded by an object for the colour camera chosen, then those points of the background will get the colour of the occluding object as illustrated by Figure 32. For example, in Figure 30a, for the colour camera used (the one on the left of the camera), the green cone occludes the red box, so in the virtual camera point of view, the points of the box that were occluded have been coloured in green. This can be prevented by storing, for each colour, the position of the 3D point associated with this colour as explained in method D.

4.4.3 Closest colour camera per pixel (Method C)

Method B can be improved by selecting the closest camera for each point. To do so, for each 3D point, I compute and compare for each camera i the angle between the three points: virtual camera, 3D point, colour camera. Then I pick the colour camera with the smallest angle, which gives the images Figure 31.



(a) 4 cameras (b) 20 cameras (c) Ground truth

Figure 31: Each 3D point is coloured with the colour camera that has the best point of view on it for 4 and 20 cameras.

The same issues than before can be observed on Figure 31a, the blue and yellow are even inverted on the triangle. However, the result becomes very good when the number of cameras is increased and Figure 31b is hard to differentiate from the ground truth.

4.4.4 Weighted sum of colours (Method D)

A widely used reconstruction method in image-based rendering is through weighted interpolation of nearby captured light rays [5,15,24]. In [26], Zhang uses this method in a setup similar to ours. The difference is that they observe a surface while we observe a point cloud, so some light rays in our case come from the background and should not be considered but we already solved that with the depth map.

The other difference is that we have occlusions, which leads to the issues seen in the two last methods. Not to mix colours from the foreground and the background like in Figure 34c, we need to reject a camera when the point we want to colour is occluded in its view. We stored for each colour the 3D point it corresponds to. Thus, to each colour pixel of the RGB image of a Kinect k $imc\{k\}(u,v)$ corresponds a pixel on the depth map $imd\{k\}(u,v)$ which corresponds to a 3D point $X\{k\}(id)$. Thus, before using the colour $imc\{k\}(u,v)$ to colour a point X, I can check that $X\{k\}(id)$ is close enough to X, i.e. $dist(X\{k\}(id), X)$ is more little than the maximum distance between two neighbouring points (Figure 32).

To do so I have to set a margin to decide whether the points are close enough or not, because a point of space sampled by two different depth cameras will not have the same 3D coordinates as shown in Figure 33. Setting $\delta = l_{max}$ works fine. If this margin is too wide, colours of different depth of the scene are mixed.

A weight is then computed for each colour by (37) like in [26] and the pixel where the point projects is coloured by the weighted sum of the usable colours.

$$weights(cam_{id}) = 1/(cam_{angle}(cam_{id}) + \epsilon) \tag{37}$$

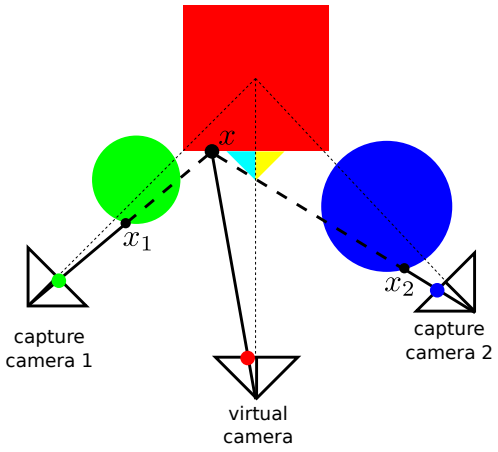


Figure 32: Top view of the scene used for the rendering. Here, by projecting the point x in camera 1, we get the colour green while the virtual camera should see a red pixel. To prevent this, the positions x_1 and x_2 are associated respectively to the colours c_1 and c_2 and a colour c_i is used only if x and x_i are close enough.

.

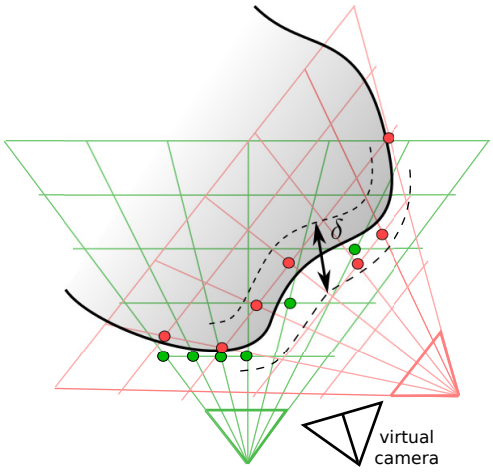


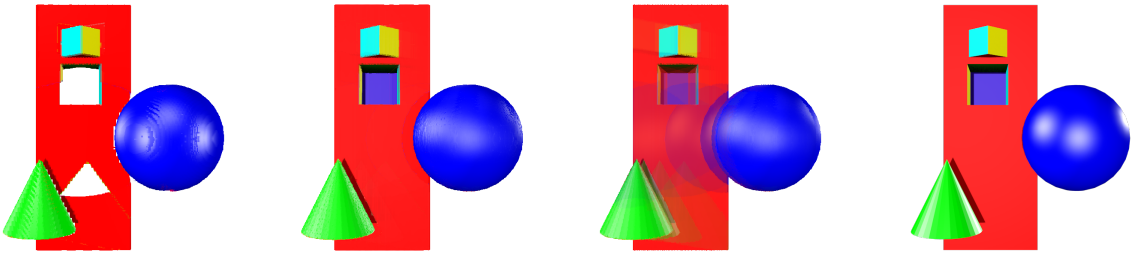
Figure 33: The surface represented by the dark line is sampled by two depth cameras (green and red). The points sampled by the cameras are respectively green and red and do not superimpose exactly on the surface.

.

Where cam_{id} is the index of the colour camera considered and cam_{angle} the angle (*virtual camera*, *3D point*, *color camera*). $\epsilon = 0.0001$ is there to limit the weight when the angle is close to zero.

The Matlab code for this method is commented in Appendix D.

On Figures 34b and 34d, we can see that the light reflection is much smoother in the generated image than in the ground truth, but the main advantage of this method is that the lighting remains consistent from one point of view to another. Indeed, while rotating around the scene, the are no jump between the colours from one frame to the next one which gives a more natural render.



(a) 4 cameras (b) 20 cameras (c) 20 cameras, no depth comparison before using the colour from a camera (d) Ground truth

Figure 34: Method mixing the colours of several cameras for 4 and 20 cameras.

4.4.5 Using points normal (Method E)

Another idea to make the lighting consistent between two close viewpoints is to colour the object with regard to the object geometry instead of the virtual camera position. To do so, the idea is to associate a normal to each point and pick the colour camera according to the angle between the normal and the colour camera's optical axis.

Each 3D point corresponds to a pixel of a depth map. The 3D points corresponding to the neighbouring pixels can be used to compute the normal to the point. Then, when a point is reprojected in the virtual camera, I can use its normal to pick the closest colour camera. But by colouring each point according to the absolute value of its normal, I get Figure 35a which shows mainly three normal values coloured in blue, red and green. This can be explained by observing the point cloud Figure 36. So, to get the normal to the real surface, I blurred the normal map obtained to get a mean normal for each point (Figure 35b). Then, I look in the blurred normal map of the virtual camera to pick the colour camera to colour the projected point.

(a) Normal map of the virtual camera (b) Blurred normal map.

Figure 35: The point clouds are reprojected on the virtual camera and each point is coloured according to its normal to get the normal map Figure 35a. This normal map is then interpolated (using the same method than for the RGB image) and a Gaussian blur of width 50 pixels and height 30 pixels is applied to get Figure 35b.

With this method, the colours should remain the same from one point of view to the other. However, because of the effect shown in Figure 36, the normal associated to a point depends on the capture point

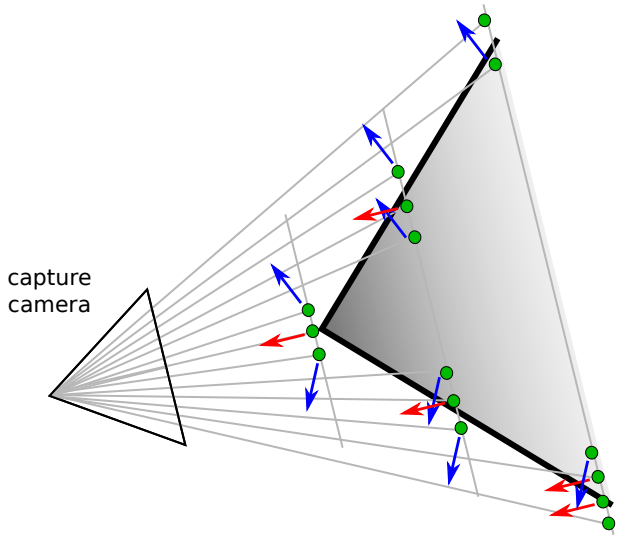


Figure 36: Top view of a depth camera sampling a tilted surface. Only three different normal values are computed.

.

of view and the colours and the normal map is computed and blurred from the virtual camera point of view so the colour will still change when the observer moves.

To get an interesting result, the normal to the points should be determined more accurately by considering for example the 10 neighbouring points to compute the normal instead of only two neighbouring points.

4.4.6 Conclusion

To conclude, we have to kinds of methods. With methods A and E, the colouration does not depend on the observer position which enables more consistent lighting when moving around the scene but in method E, the light reflection would remain the same from every point of view which is not very natural and with method A, the number of cameras must be limited and there must be no strong specularities on objects to get a visually satisfying result. Methods B, C and D deal better with light reflection but would be more adapted to render only one image off-line because they are expensive (Table 5) and the colours can jump from one value to another only by moving the virtual camera of some degrees.

Camera number	Method A	Method B	Method C	Method D
4	14s	17s	56s	61s
8	31s	39s	3mn15	3mn6
16	1mn5	1mn23	10mn11	8mn23

Table 5: Execution time of each colouring method for different number of cameras on the synthetic dataset.

For the real-time rendering we will use the Method A to pick the colours since it is the easiest to implement. Indeed, one Kinect needs then only one colour texture and one colour camera calibration.

5 Real-time rendering

As seen in part 2, numerous techniques exist to generate a 3D mesh from a point cloud, but I looked for a more straightforward way to render the calibrated Kinect's data in real-time and as realistic as possibles, using directly the point cloud.

The idea is to represent each point of the cloud captured by a Kinects by a quad of a certain size such that an observer sees no holes between points. To do so I relied on the calculations made in Section 4.3.1.

This has been implemented in the game engine Unreal Engine 4 (UE4).

5.1 Theoretical study

I first simulated the creation of quads in Matlab on Kinect v1 data to test different size algorithms and the results. Since Matlab can not plot a point cloud as quads of varying size, I only associated a quad size to each point and I computed the size of the reprojection on the virtual camera in pixels. Then, when I project a point in the virtual camera, instead of colouring one pixel in the target image, I coloured its neighbouring pixels too, using the computed size to know how many pixels to write and the depth map to decide whether I overwrite a pixel or not.

I first used the maximum size $s_w = \sqrt{\Delta d^2 + l^2}$ established in Section 4.3.1 but this gave quads far too big (Figure 38b) since it corresponds to the worst case and the depth steps are much bigger than the spacing caused by the image resolution (37).

To take this into account, I interpolate the size $s_w(d)$ between $l_c(d)$ and the maximum size $\sqrt{\Delta d(d)^2 + l(d)^2}$ according to the angle β between the capture camera and the virtual camera such that when $\beta = 0°$, $s_w(d, 0°) = l_c(d)$, and when $\beta = 45°$, $s_w(d, 45°) = \sqrt{\Delta d(d)^2 + l^2}$ (Figure 37). d corresponds to the distance between the point and the camera which captured it. This is quite an approximative calculation but it gives satisfying results as shown in Figure 38c.

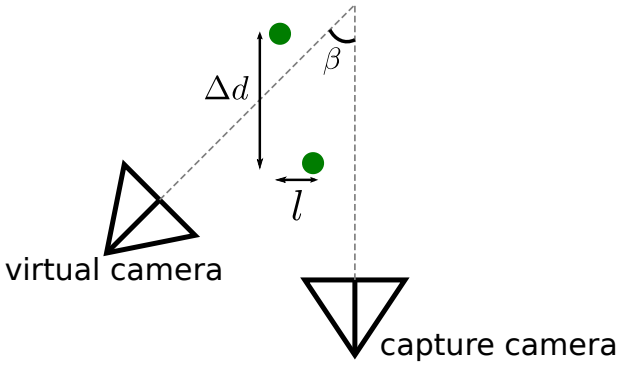


Figure 37: The spacing between two points depends on the depth sampling step Δd and the lateral spacing due to the depth image resolution l.

Figure 38 shows that with the adapted quad's size, the result is comparable to the result by interpolation and there are practically no holes in the image. Figure 39 shows several points of view between the capture Kinects generated with this method.

5.2 Coding the Kinect's data

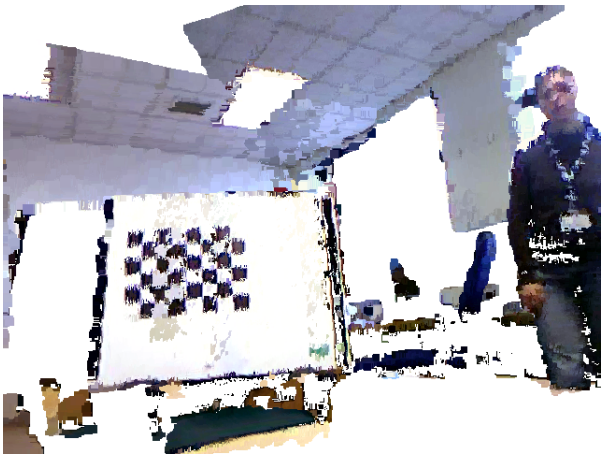
From there, we could use a second Kinect v2 so the renderer has been adapted to this version since its performances are better.

In order to pack the data output by the Kinect for each frame, the RGB image, IR and depth image are stored in one 24 bits PNG image. The colour image uses 8 bits per colour channel so 24 bits per pixels.

The depth values output by Kinect v2 are 16 bits floating point values giving the depth in millimetres. To write those values in the image, we need to distribute the bits between at least two colour channels. To make the depth values easier to decode in Unreal Engine 4, those values are converted to 32 bits unsigned integers. Indeed, the depth resolution goes from 0.5 mm to 1 mm and the depth values from 0.5 m to 4.5 m, so the depth values in micrometres go from 500 μ to $4.5 \cdot 10^6 \mu$. We have $2^{22} < 4.5 \cdot 10^6 < 2^{23}$ so a depth value can be stored in 3 bytes. The most significant byte is stored in the red channel, then the others in the green and blue channels. When the depth image is read by UE4 as a texture, it gives an 8 bits floating point value between 0 and 1 for each channel (r, g, b). The depth value can then be computed through the equation: $depth(mm) = (r \cdot 2^{24} + g \cdot 2^{16} + b \cdot 2^8)/1000$.

Thus, the Texture Figure 40 is given to UE4 and the colour image and depth map are extracted from it according to their position and size in the texture.

(a) Image obtained by simply projecting the points.

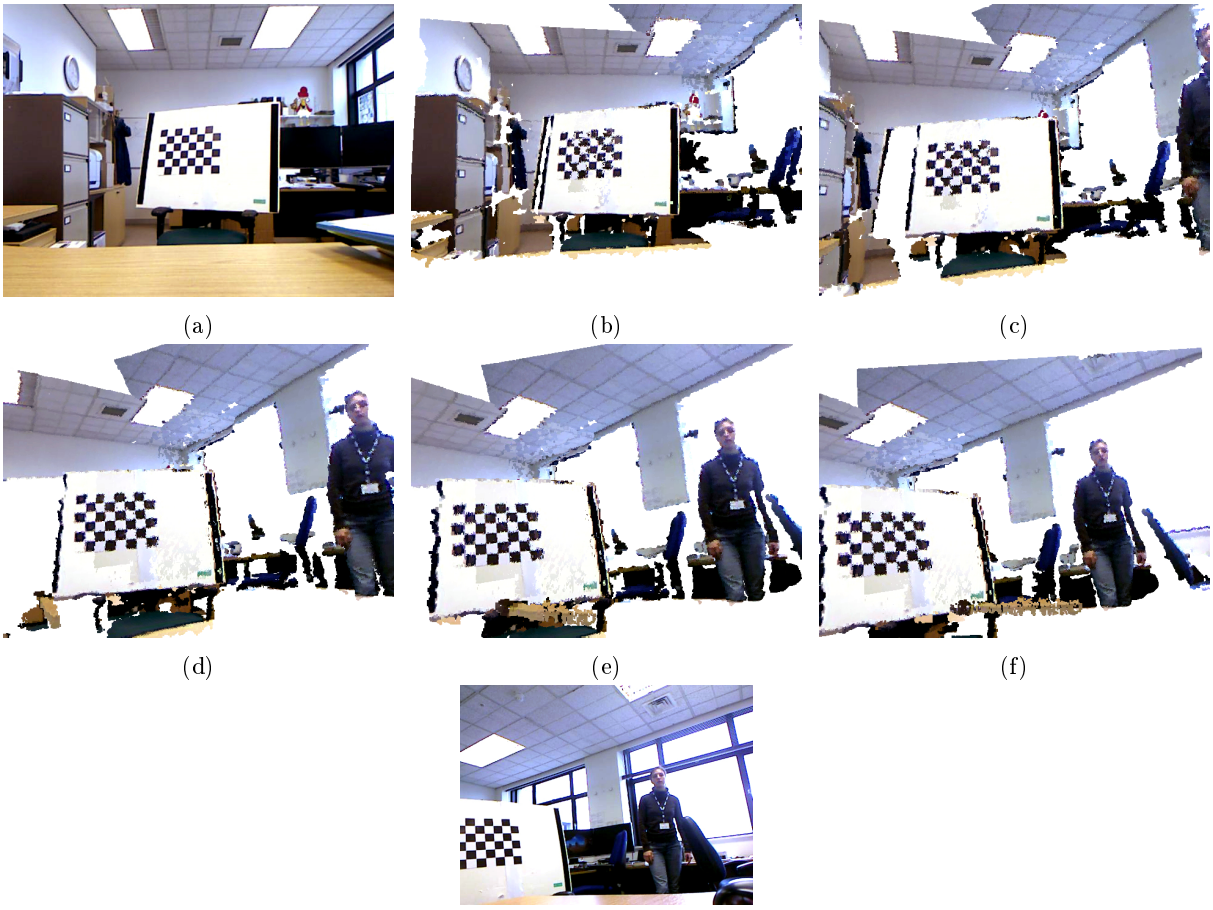


(b) Quads size computed from the capture distance d.

(c) Quads size computed from the capture distance d and β.

(d) Interpolation of Figure 38a as explained in section 4.3.2.

Figure 38: This Figure shows the image generated with different methods for a virtual camera placed between the two real Kinects.



(a) (b) (c)

(d) (e) (f)

(g)

Figure 39: Images generated for several positions of the virtual camera using quads of varying sizes.

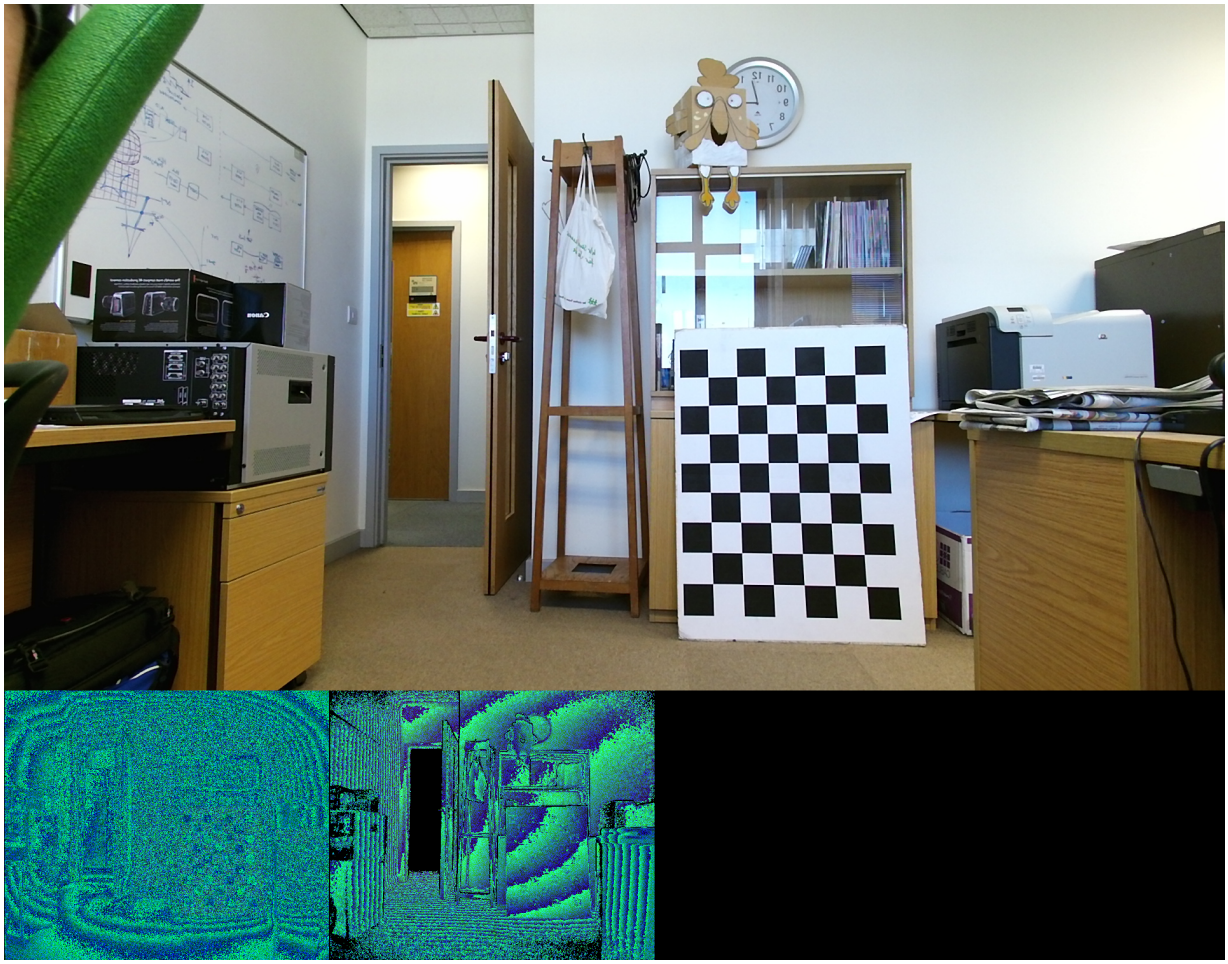
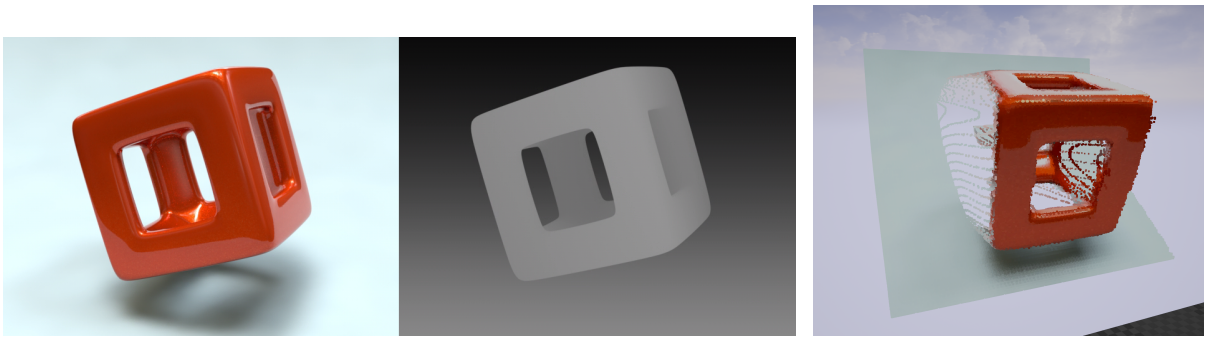


Figure 40: Input texture for the renderer in UE4. The texture contains the colour image, the infra-red image and the depth map.

.

5.3 Implementation in Unreal Engine 4

To get a real-time rendering, a good approach is to use texture lookups since they can be done really fast by the GPU. I used an implementation of a depth map render in Unreal Engine 4, which, from a texture (image or video) containing a colour image and a grayscale depth map (Figure 41a), renders a quads cloud (Figure 41b) thanks to an orthographic projection.



(a) Input video, the colour image is on the left, the depth image is on the right.

(b) The observer can move around the cube and get closer or further.

Figure 41: Result given by the initial UE4 renderer

The data of a colour camera and a depth camera can be rendered by using a dynamic mesh and a material. I had to insert the calibration data to compute a pinhole projection instead of a planar projection and handle the distortions. To do so we implemented a Matlab function to write the calibration parameters of a pair of colour camera and depth camera in an XML file and a function to read this file and store the values in the material parameters.

The mesh is a grid of infinitesimal quads built dynamically according to the input depth map size, where each quad represents a pixel of the depth map. The material applied to this mesh will give a colour to each quad, move its centre at a specific position and move the four vertices that constitute the corners of the quads such that the quad faces the virtual camera.

The final pipeline implemented in UE4 is represented in Figure 42 and 43.

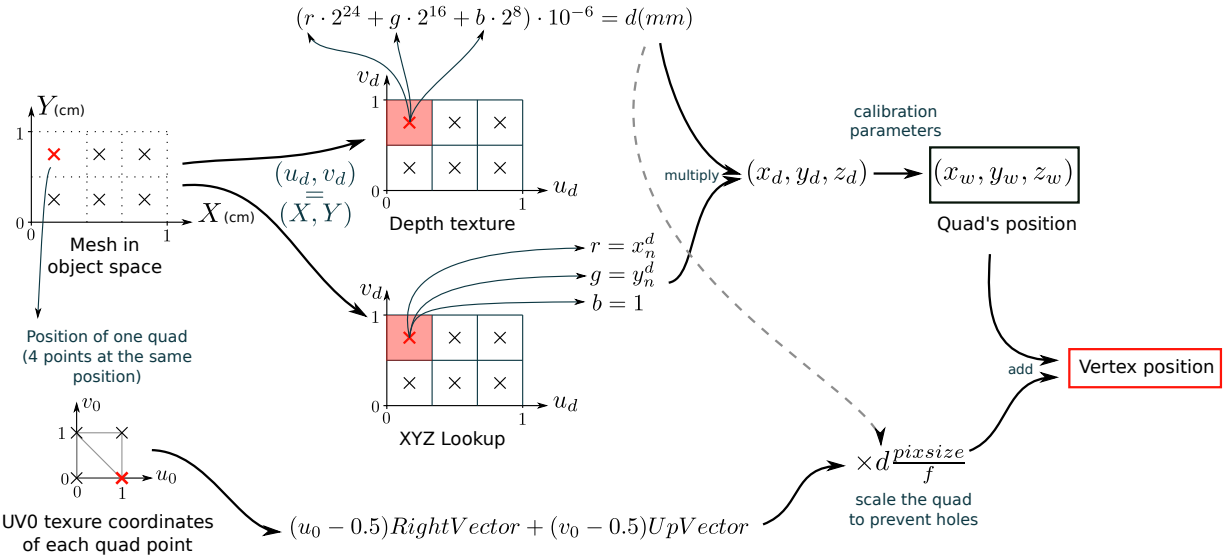


Figure 42: For each vertex of the quads grid, a world position is computed.

.

The material applies a position and a color to each vertex of the mesh. On figure 42, a vertex is highlighted in red as example.

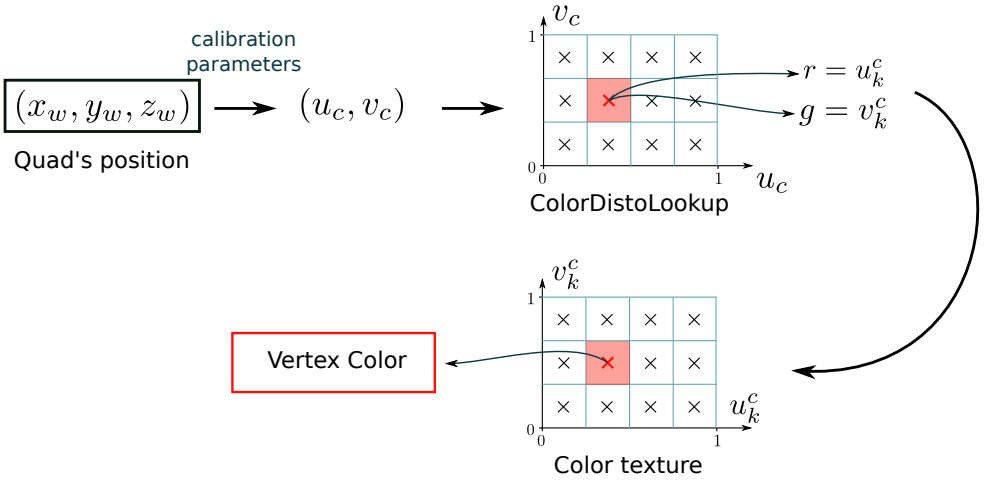


Figure 43
For each vertex of the quads grid, a world position is computed, from this world position, a colour is computed.

The (X, Y) position of the points in the mesh will be used to pick the corresponding pixels (u_d, v_d) in the depth map. From those coordinates, we can get the depth value d by look up in the depth texture and the corresponding normalised depth camera coordinate $(x_n^d, y_n^d, 1)$ in the texture XYZLookup. This texture is generated using the depth camera projection and distortion parameters. Indeed, for each depth map pixel (u_d, v_d) we can compute the corresponding point (x_n^d, y_n^d) and this texture computation only needs to be done once before rendering and not for each pixel of each new frame. By multiplying this vector by the depth d we get the point's depth camera coordinate $(x_d, y_d, z_d) = d \cdot (x_n^d, y_n^d, 1)$. This point can then be transformed into world coordinates thanks to the depth camera extrinsic parameters to get (x_w, y_w, z_w).

To get the colour of this point, it must be reprojected in the colour camera image thanks to the colour camera calibration parameters. To handle the colour camera distortion, I use a texture lookup as well. I generate once and for all a texture ColorDistoLookup such that $ColorDistoLookup(u, v) = (u_k, v_k)$ where (u_k, v_k) are the distorted image coordinates of the point. Finally, I use those values to pick up the colour in the colour texture.

This is done for each point of the mesh. Four points of one quad correspond to the same pixel and have the same position (X, Y) (since the quad's size is zero). So the four points will get the same colour. Now we need to rearrange those four points such that they form a quad facing the virtual camera. This is done by association textures coordinates $(u0, v0)$ to each point of a quad. When building the mesh, each corner of a quad gets 2D coordinates among $(1, 0)$, $(1, 1)$, $(0, 1)$ and $(0, 0)$. Those coordinates can then be used to move each vertex along the virtual camera's up and right vectors. The quad is then scaled according to the distance d at which it was captured.

The blueprint code of the material is attached in Appendix E.

5.4 Results

The final material in UE4 can be used to render a point cloud for each Kinect. Figure 44 shows the render of the scene captured by the setup A seen in part 3.4.3. An object for each Kinect and placed in the centre of the scene and the described material is associated with it, we can see that the two point clouds still superimpose. However the visual result is not so good because of the low depth resolution of Kinect v1 and the bad exposition on the camera. The non-consistency of the colours between the point clouds could be solved by using external cameras instead of the Kinect own colour camera and here the quad's sizes is here adapted to Kinect v1 so the quads can not fill the holes between points.

The ideal set up for this renderer would be several Kinects v2 and external RGB cameras, unfortunately we missed the hardware to perform a simultaneous capture with several Kinects v2. Indeed, Kinect v2 has been implemented in IP Studio but as said in part 2.1.2, each Kinect v2 needs its own computer to work and IP Studio could not be installed successfully on the available hardware.

However I could test the rendering of data provided by one Kinect v2 which gave me Figure 45 and

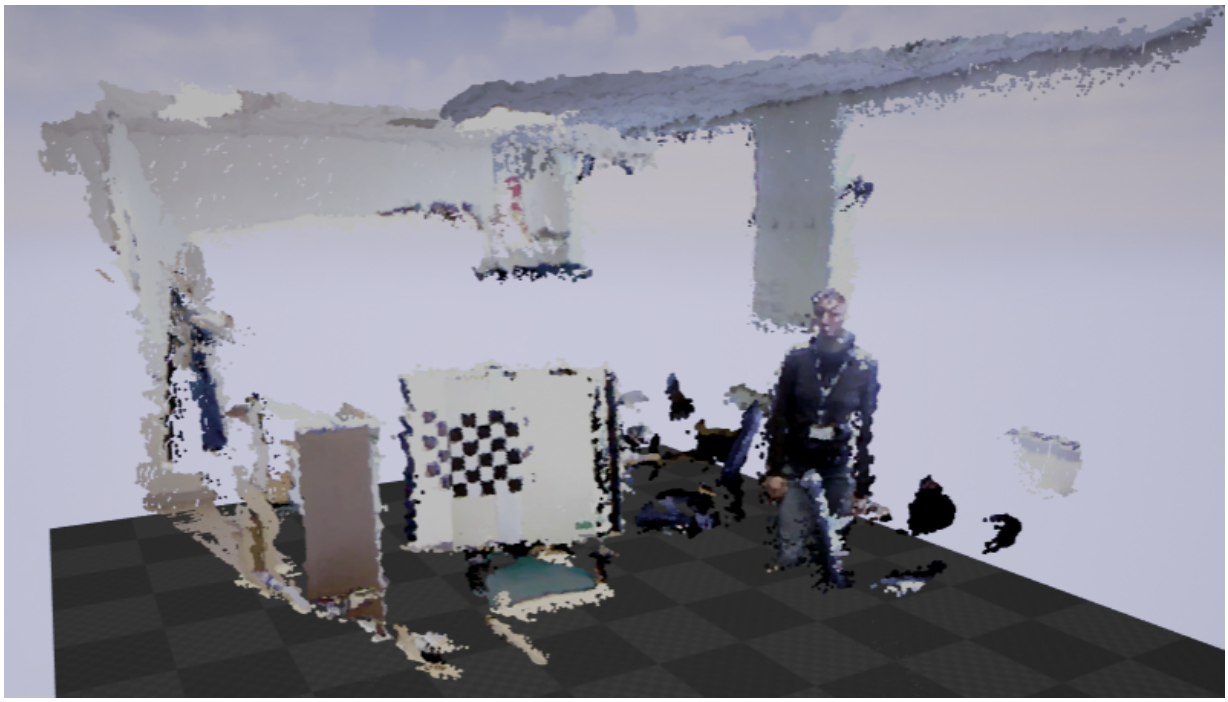


Figure 44: Visualisation of Kinect v1 data in UE4.

46.

Figure 45: Visualisation of one Kinect v2 data in UE4.

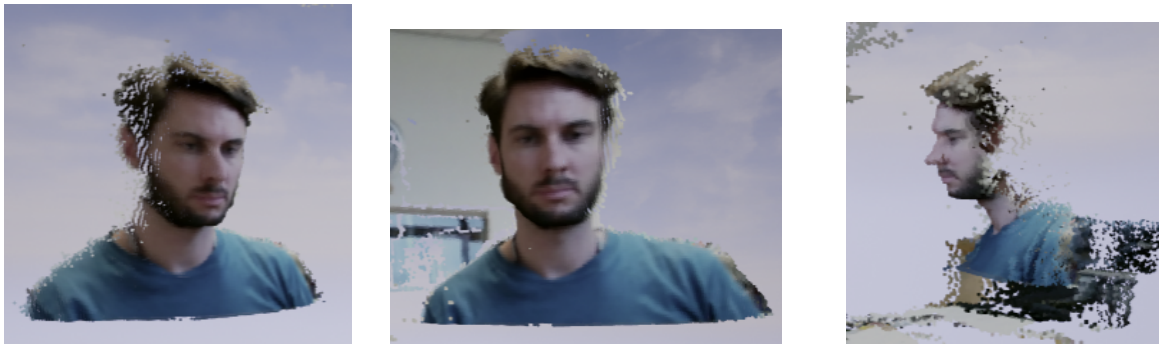


Figure 46: Visualisation of one Kinect v2 data in UE4.

On Figure 45 we can see that the result is much better with the Kinect v2, the surfaces have a better resolution and we can see no holes in the plane surface thanks to the quads.

Figure 46 show the level of details that can be achieved on a face using only the Kinect v2 without external colour camera.

With several Kinects v2, and a good calibration, we could also fill the gaps caused by occlusion and with external colour cameras, the colour would fit between two superimposing point cloud for a more realistic result.

6 Conclusion

In this project, a capture system consisting of several Kinects has been setup and several rendering techniques have been tested to finally implement one of them in an existing game engine to visualise the data captured by the system. The calibration of several Kinects has been implemented in Matlab and any captured sequence can now be rendered in Unreal Engine 4. A manual (Appendix F) has been written to detail the setup requirements, the calibration process and how to import the data in theUE4project to visualise a sequence and thus enable anyone to use the tools developed in this thesis.

The provisional planning has been followed on the whole but some unplanned stages had to be added. Indeed, we did not know precisely from the start which approach would be followed for the rendering part, so the planning has been refined after researching on the subject. Some delays have also been experienced while waiting for licences and hardware to start some parts of the project like the rendering on UE4 which requires specific software. The planning finally followed is in Appendix B.

A

Projected Gantt Diagram

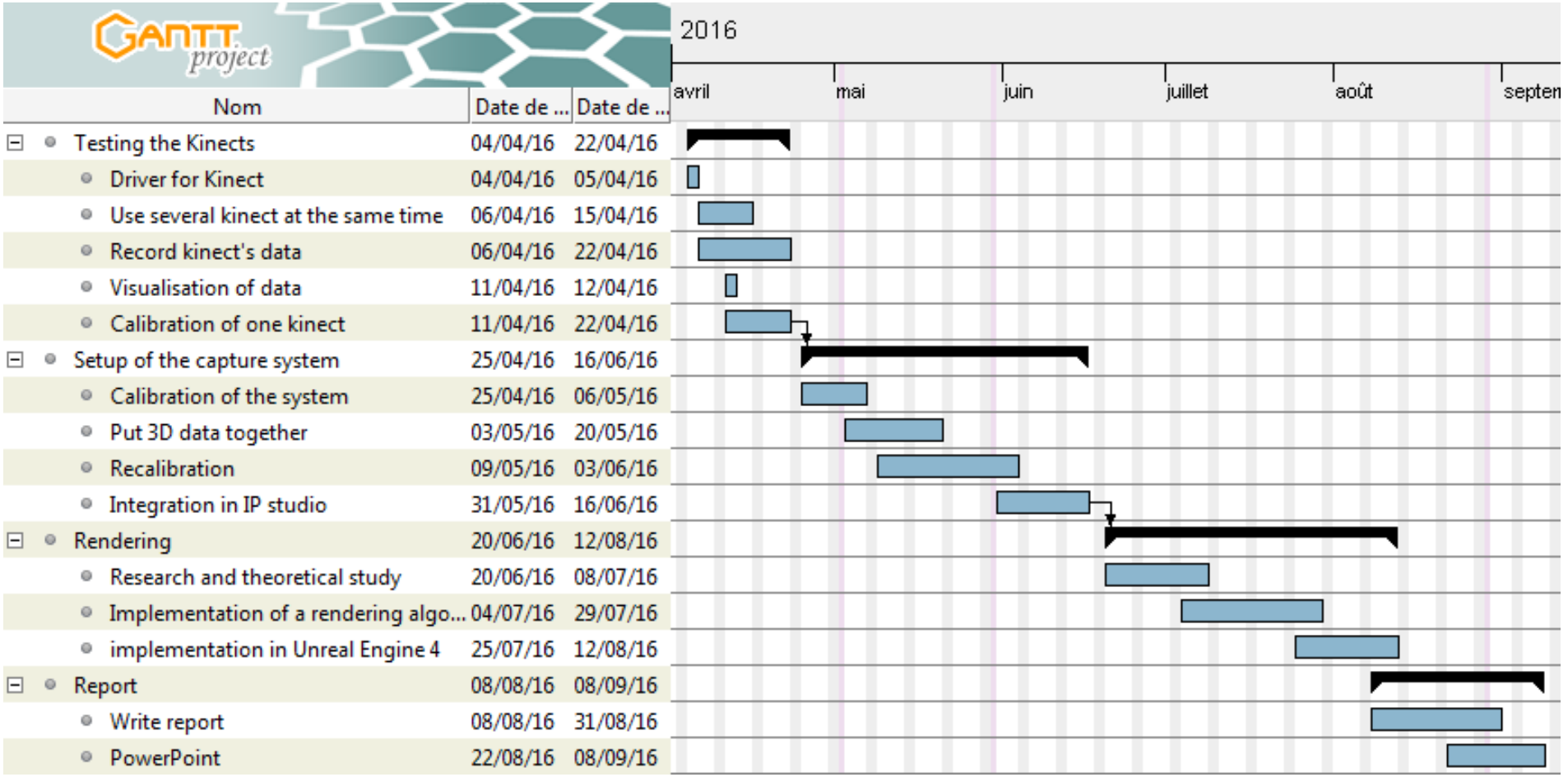


Figure 47: Gantt diagram initially projected

B

Final Gantt Diagram

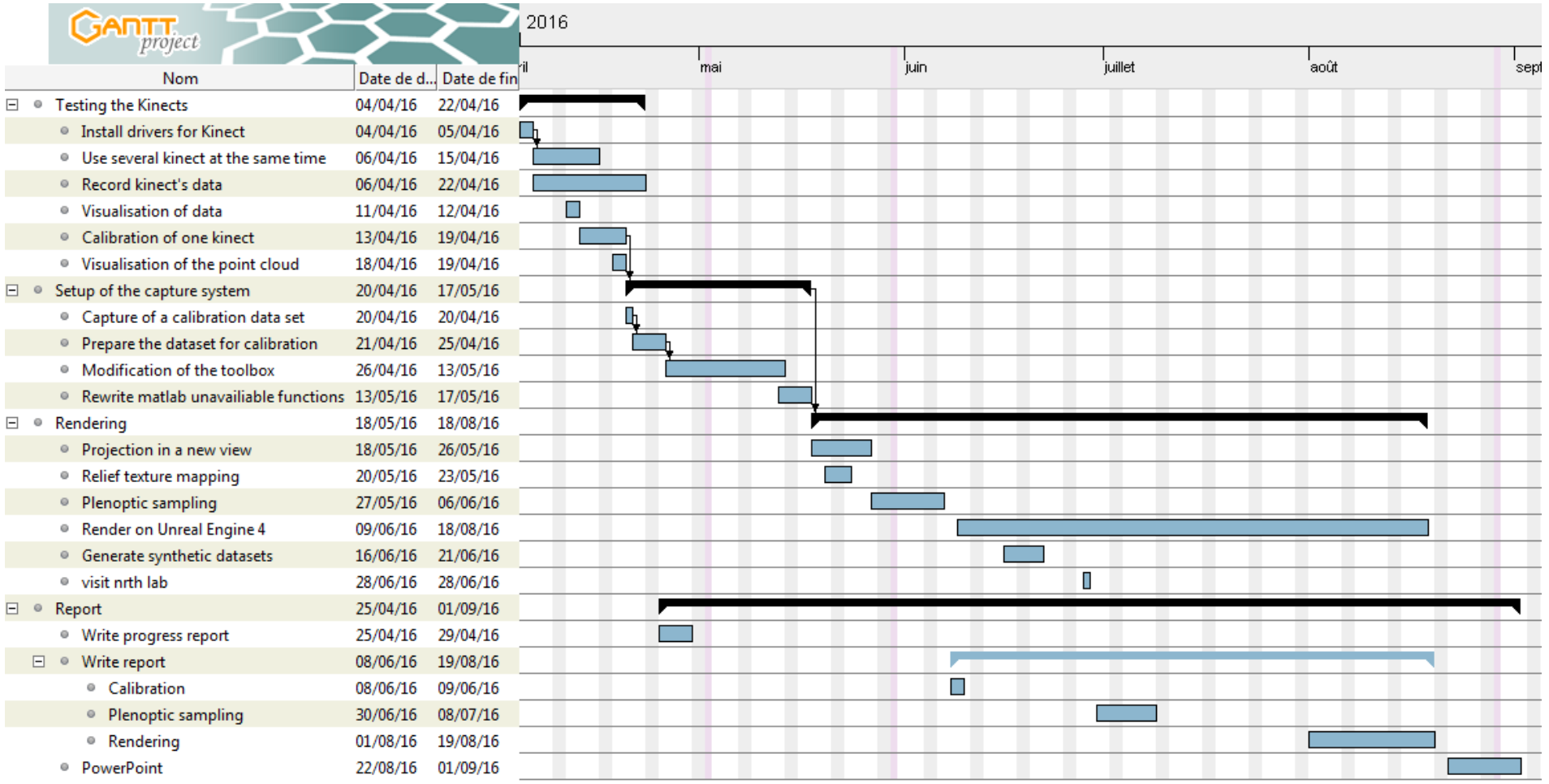


Figure 48: Gantt diagram finally followed

C
Matlab implementation of the Levenberg-Marquardt algorithm

```
function [xf, r, jac, cnt] = LMFsolve(varargin)
% LMFSOLVE  Solve a Set of Nonlinear Equations in Least-Squares Sense.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% [Xf, Res, CNT] = LMFsolve(FUN,Xo,Options)
% FUN      is a function handle or a function M-file name that evaluates
%          m-vector of equation residuals,
% Xo       is n-vector of initial guesses of solution,
% Options  is an optional set of Name/Value pairs of control parameters
%          of the algorithm. It may be also preset by calling:
%          Options = LMFsolve('default'), or by a set of Name/Value pairs:
%          Options = LMFsolve('Name',Value, ... ), or updating the Options
%                    set by calling
%          Options = LMFsolve(Options,'Name',Value, ...).
%
%    Name    Values {default}          Description
% 'Display'      integer      Display iteration information
%                              {0}  no display
%                               k   display initial and every k-th iteration;
% 'FunTol'       {1e-7}       norm(FUN(x),1) stopping tolerance;
% 'XTol'         {1e-7}       norm(x-xold,1) stopping tolerance;
% 'MaxIter'      {100}        Maximum number of iterations;
% 'ScaleD'                    Scale control:
%                value         D = eye(m)*value;
%                vector        D = diag(vector);
%                 {[]}         D(k,k) = JJ(k,k) for JJ(k,k)>0, or
%                                     = 1 otherwise,
%                                       where JJ = J.'*J
% Not defined fields of the Options structure are filled by default values.
%
% Output Arguments:
% Xf         final solution approximation
% (Ssq        sum of squares of residuals)
% Res        residuals
% Jac        Jacobian in Xf
% Cnt        >0           count of iterations
%            -MaxIter,    did not converge in MaxIter iterations

% Miroslav Balda,
% balda AT cdm DOT cas DOT cz
% 2007-07-02     v 1.0
% 2008-12-22     v 1.1 * Changed name of the function in LMFsolv
%                      * Removed part with wrong code for use of analytical
%                        form for assembling of Jacobian matrix
% 2009-01-08     v 1.2 * Changed subfunction printit.m for better one, and
%                        modified its calling from inside LMFsolve.
%                      * Repaired a bug, which caused an inclination to
%                        istability, in charge of slower convergence.
%Marilyn Keller
%2016-21-08 Simplification to get an LM algorithm implementation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%
%   PARSE OPTIONS
%%%%%%%%%%%%%
```

```
56
57 %               Default Options
58 if nargin==1 && strcmpi('default',varargin(1))
59     xf.Display   = 0;          %   no print of iterations
60     xf.MaxIter   = 100;        %   maximum number of iterations allowed
61     xf.ScaleD    = [];         %   automatic scaling by D = diag(diag(J'*J))
62     xf.FunTol    = 1e-7;       %   tolerace for final function value
63     xf.XTol      = 1e-4;       %   tolerance on difference of x-solutions
64     return
65
66 %               Updating Options
67 elseif isstruct(varargin{1}) % Options=LMFsolve(Options,'Name','Value',...)
68     if ~isfield(varargin{1},'Display')
69         error('Options Structure not correct for LMFsolve.')
70     end
71     xf=varargin{1};            %   Options
72     for i=2:2:nargin-1
73         name=varargin{i};      %   Option to be updated
74         if ~ischar(name)
75             error('Parameter Names Must be Strings.')
76         end
77         name=lower(name(isletter(name)));
78         value=varargin{i+1}; %   value of the option
79         if strncmp(name,'d',1), xf.Display  = value;
80         elseif strncmp(name,'f',1), xf.FunTol   = value(1);
81         elseif strncmp(name,'x',1), xf.XTol     = value(1);
82         elseif strncmp(name,'m',1), xf.MaxIter  = value(1);
83         elseif strncmp(name,'s',1), xf.ScaleD   = value;
84         else    disp(['Unknown Parameter Name --> ' name])
85         end
86     end
87     return
88
89 %               Pairs of Options
90 elseif ischar(varargin{1})   % check for Options=LMFSOLVE('Name',Value,...)
91    Pnames=char('display','funtol','xtol','maxiter','scaled');
92    if strncmpi(varargin{1},Pnames,length(varargin{1}))
93       xf=LMFsolve('default'); % get default values
94       xf=LMFsolve(xf,varargin{:});
95       return
96    end
97 end
98
99 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
100 %   LMFSOLVE(FUN,Xo,Options)
101 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
102
103 FUN=varargin{1};             %   function handle
104 if ~(isvarname(FUN) || isa(FUN,'function_handle'))
105    error('FUN Must be a Function Handle or M-file Name.')
106 end
107
108 xc=varargin{2};              %   Xo
109
110 if nargin>2                  %   OPTIONS
111     if isstruct(varargin{3})
112         options=varargin{3};
113     else
114         if ~exist('options','var')
```

```
115             options = LMFsolve('default');
116         end
117         for i=3:2:size(varargin,2)-1
118             options=LMFsolve(options, varargin{i},varargin{i+1});
119         end
120     end
121 else
122     if ~exist('options','var')
123         options = LMFsolve('default');
124     end
125 end
126 
127 %%%%%%%%%%%
128 %Initialisation
129 %%%%%%%%%%%
130 
131 x  = xc(:);
132 lx = length(x);
133 
134 r  = feval(FUN,x);              % Residuals at starting point
135 %~~~~~~~~~~~~~~~~~
136 S  = r'*r;
137 epsx = options.XTol(:); %minimum step value
138 epsf = options.FunTol(:);
139 if length(epsx)<lx, epsx=epsx*ones(lx,1); end
140 J = finjac(FUN,r,x,epsx);
141 %~~~~~~~~~~~~~~~~~~~~~~
142 nfJ = 2;
143 A = J.'*J;                      % System matrix
144 v = J.'*r;
145 
146 D=eye(size(A));
147 
148 Rlo = 0.25;
149 Rhi = 0.75;
150 l=0.01;       lc=.75;       is=0;
151 cnt = 0;
152 ipr = options.Display;
153 printit_less(ipr,-1);                 %   Table header
154 d = options.XTol;                    %   vector for the first cycle
155 maxit = options.MaxIter;             %   maximum permitted number of iterations
156 
157 
158 %%%%%%%%%%%
159 %Main iteration circle
160 %%%%%%%%%%%
161 
162 while cnt<maxit && ...               %
163     any(abs(d) >= epsx) && ...
164     any(abs(r) >= epsf)
165       d  = (A+l*D)\v;                %   negative solution increment
166 
167     xd = x-d;
168     rd = feval(FUN,xd);
169 %   ~~~~~~~~~~~~~~~~~~~
170     nfJ = nfJ+1;
171     Sd = rd.'*rd;
172     S = r.'*r;          %   predicted reduction
173 
```

```
174      if Sd<S
175          l=l/10;
176      else
177          l=l*10;
178      end
179
180      cnt = cnt+1;
181      if ipr~=0 && (rem(cnt,ipr)==0 || cnt==1)
182          printit_less(ipr,cnt,nfJ,S,x,d,l,lc) % print the iteration results
183      end
184
185      if Sd<S
186          S = Sd;
187          x = xd;
188          r = rd;
189          J = finjac(FUN,r,x,epsx);
190  %       ~~~~~~~~~~~~~~~~~~~~~~~~~
191          nfJ = nfJ+1;
192          A = J'*J;
193          v = J'*r;
194      end
195  end
196
197  %Print the reason for stopping
198  if cnt>maxit
199      fprintf('LMFsolve stopped because max number of iteration reached. \n');
200  elseif any(abs(d) <= epsx)
201      fprintf('LMFsolve stopped because solution converged. \n');
202  elseif any(abs(r) <= epsf)
203      fprintf('LMFsolve stopped because residual low enough. \n');
204  end
205
206  xf = x;                              %   final solution
207  if cnt==maxit
208      cnt = -cnt;
209  end                                  %   maxit reached
210  rd = feval(FUN,xf);
211  nfJ = nfJ+1;
212  Sd = rd.'*rd;
213  jac=J;
214  if ipr, disp(' '), end
215  printit_less(ipr,cnt,nfJ,Sd,xf,d,l,lc) %Print the results
```

D
Matlab code of the colouring method C

```
1
2  %Use the points captured by the cameras in the same
3  %semicircle as the virtual camera
4  for k=find(anglevc<=pi/2)
5
6      cvp=X{k}(:,xid)-vcalib.rt{1}; %vector from virtual camera to point
7      %If a point was already reprojected in the same pixel, only consider the
8      %new points if it is closer to the virtual camera than the former one
9      if cvp<dmap(i,j)
10
11         %Color given by each camera
12         col_vect=zeros(3,ccount);
13         %Angle of each camera to the virtual camera
14         angle_cams=zeros(1,ccount);
15         %Weight associated to each color
16         col_weights=zeros(1,ccount);
17
18         %Only consider the cameras in the semicircle
19         for cam=find(anglevc<=pi/2)
20
21             %Compute the weights
22             cp=X{k}(:,xid)-calib.rt{cam}; %vector from camera to point
23             angle_cams(cam)=atan2(norm(cross(cp,cvp)),dot(cp,cvp));
24
25             %Do not use colors of behind the surface
26             if col_dist{k}{cam}(xid) < depth_precision
27                 col_weights(cam)=1/(0.00001+angle_cams(cam));
28             end
29
30             % Store the color associated to the camera. If non-valid color,
31             % ignore this camera by setting the weight to zero
32             if ~isnan(col{k}{cam}(xid,:))
33                 col_vect(:,cam)=col{k}{cam}(xid,:)';
34             else
35                 col_weights(cam)=0;
36             end
37         end
38
39         %Normalize the weight coefficients
40         col_weights=col_weights/sum(col_weights);
41
42         %Compute the Weighted sum of the colors and color the pixel
43         img(i,j,:)=sum((col_vect*diag(col_weights)),2,'omitnan');
44
45         %Write the distance from point to virtual camera in the depth map
46         dmap(i, j)=cvp;
47
48     end
49 end
```

E UE4 Blueprint code

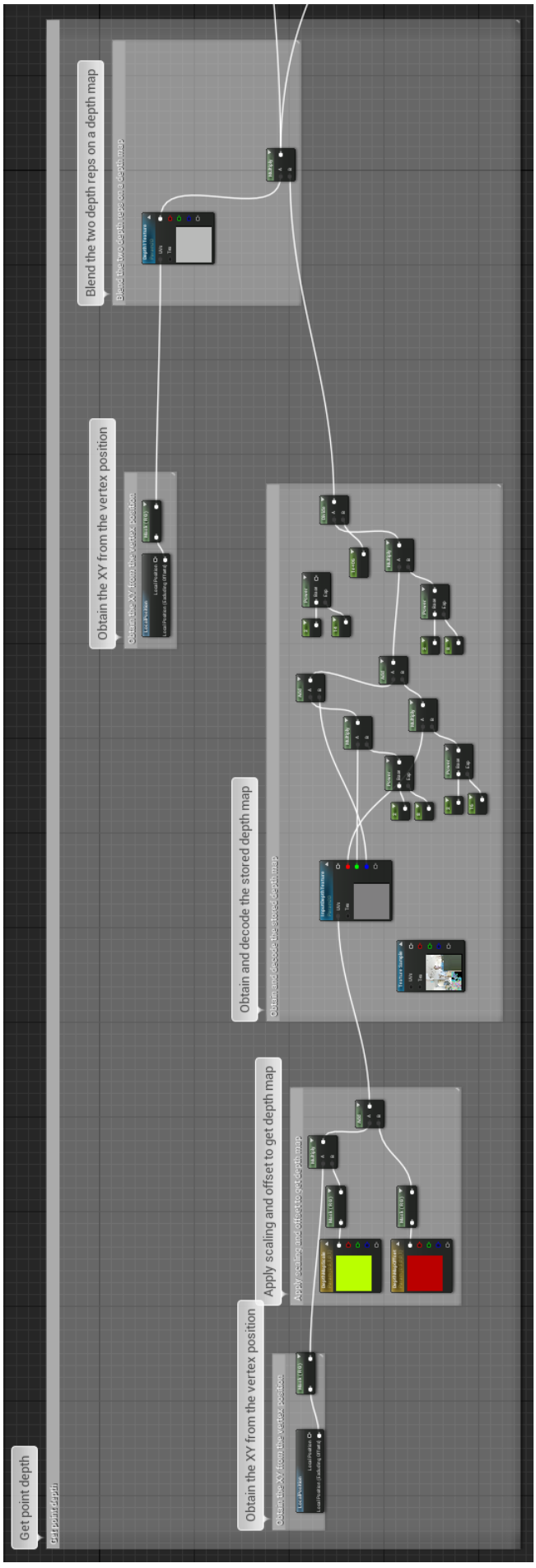


Figure 49: Outputs depth camera coordinates in function of the vertex coordinates in the mesh.

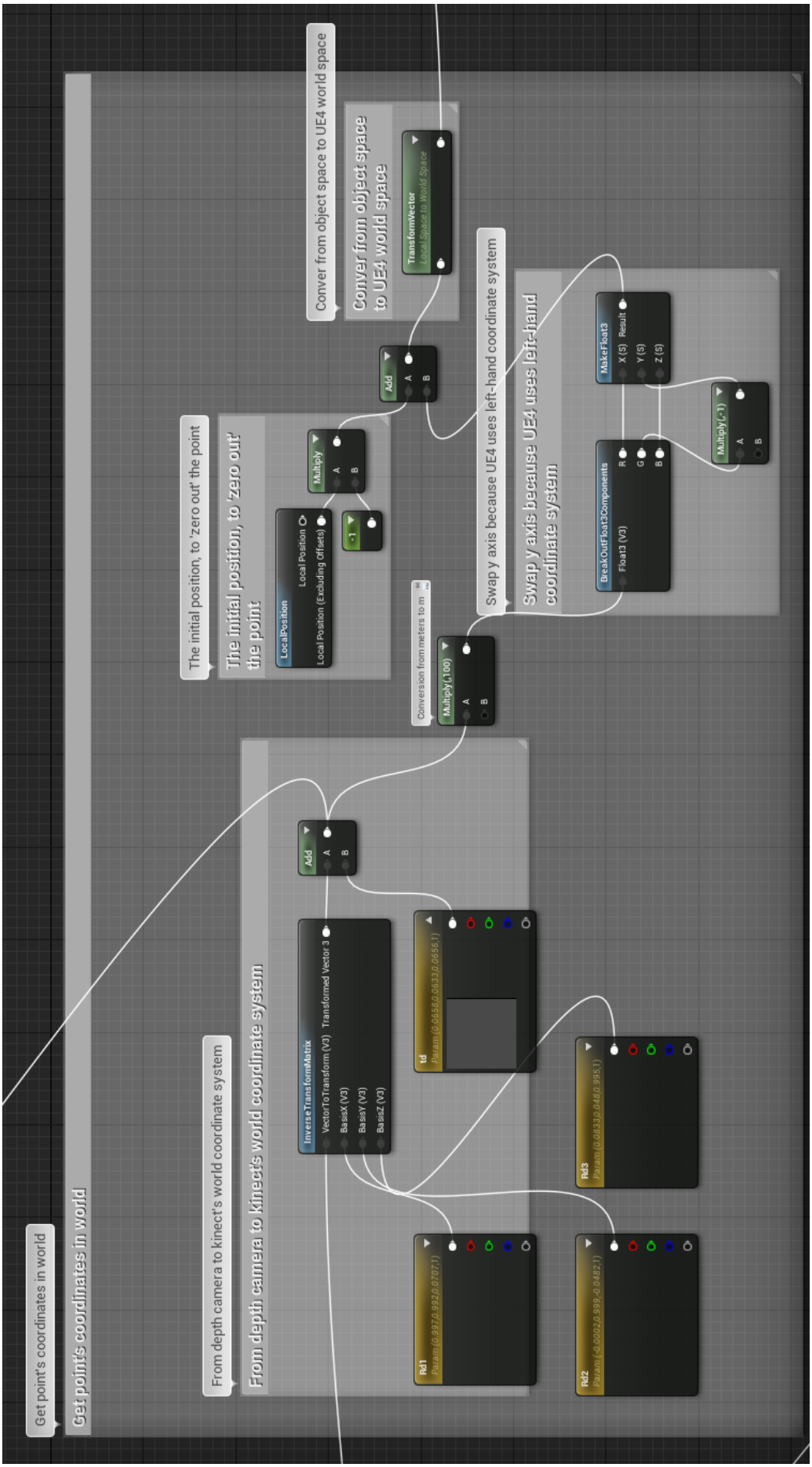


Figure 50: The block's input is the depth camera coordinates and the block transforms it in world coordinates.

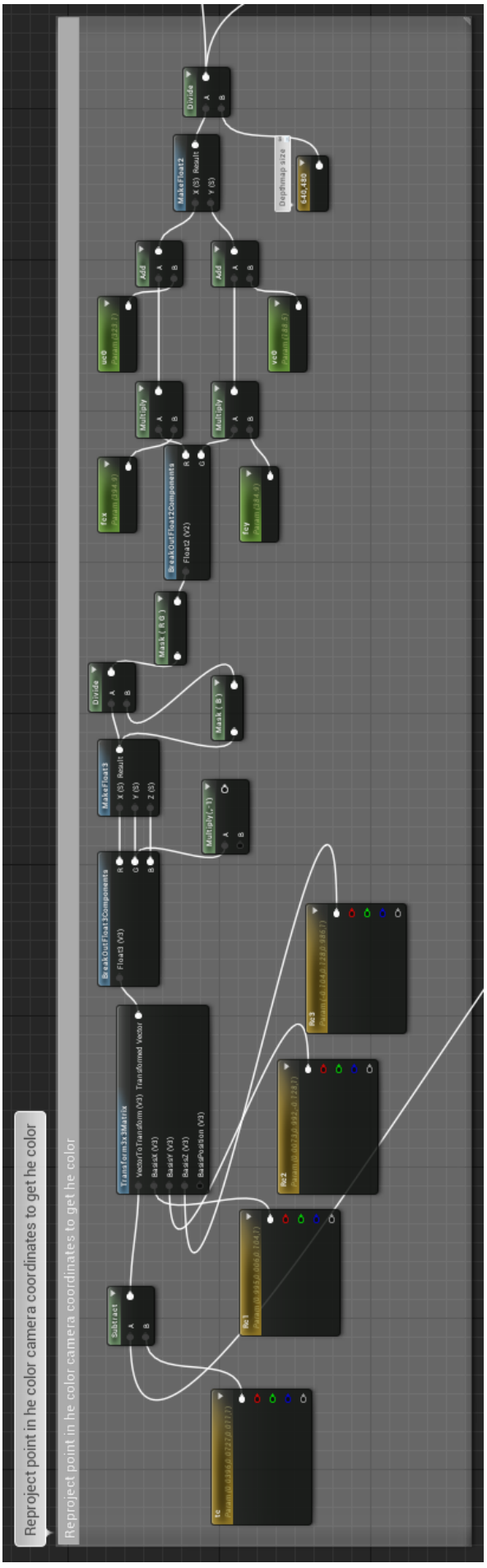


Figure 51: The block's input is the world coordinates and the block transforms it in colour camera image coordinates.

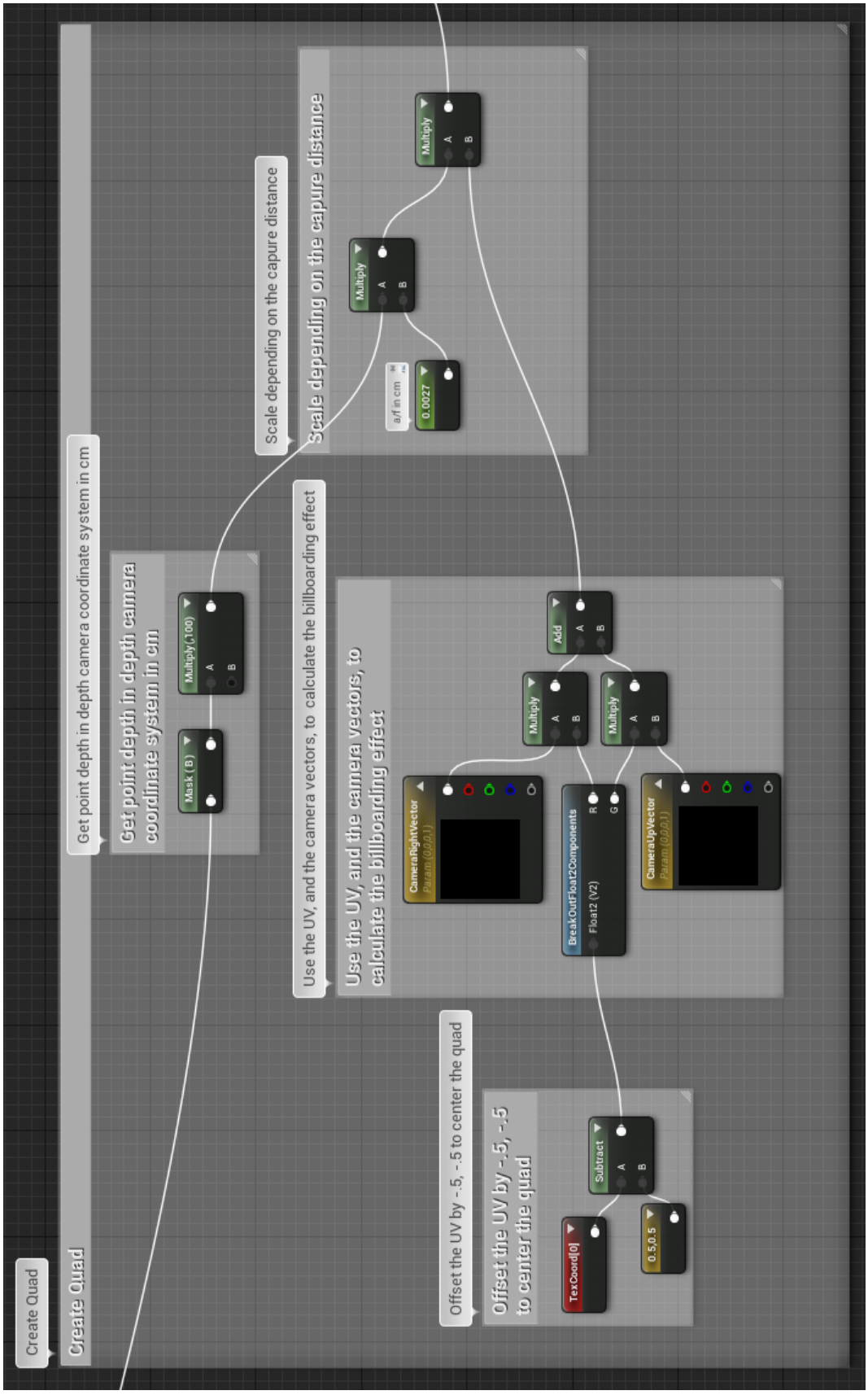


Figure 52: The block takes the depth of the point in depth camera coordinates into input and outputs the offset to apply to the vertex to scale and orientate the quad properly.

F Joint calibration and rendering manual

Calibration and Rendering with several Kinects and external colour cameras

Manual

This manual aims to explain step by step how to calibrate several Kinects and possibly some external cameras jointly with a calibration toolbox on Matlab adapted from the one provided by Herrera (http://www.ee.oulu.fi/~dherrera/kinect/) and then render a captured sequence in Unreal Engine 4.

I Acquisition of the data

1 Setup

The Kinects used should be the second version (Kinect for Xbox one). They can capture a depth range from 0.5m to 4m so they should be placed on a circle of radius about 2 meters around the scene to capture, with a reasonable angle between two Kinects (around 45 degrees).
If you want to use external colour cameras to use them instead of the Kinect internal colour camera, you should place them close to the Kinects.
The N_k Kinects will be numbered from 0 to N_k-1 and the N_c external cameras from N_k to N_k+N_c-1.

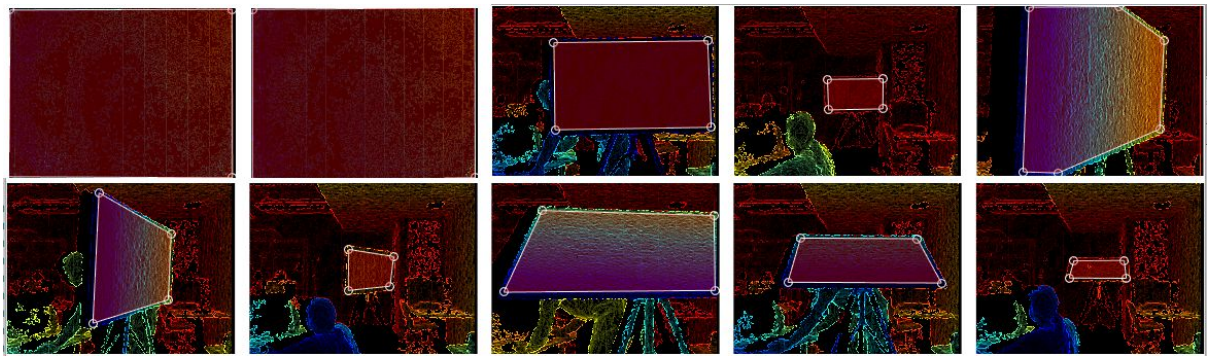
To calibrate the Kinects, a checkerboard with known dimensions must be captured by the Kinects with different positions. For example, you can use a big board (about 2x1 m) with an 8x6 A3 checkerboard stuck on it like the one used in the images below.

To know how far apart you can place the Kinects, place the checkerboard in front of two neighbouring Kinects while displaying their RGB and Depth outputs. You should be able to tilt the checkerboard with different angles and bring it closer and further to the Kinects without its surface disappearing from the depth image and the black squares should be differentiable to the white squares on the RGB image. If, because of light reflection, you can not move the checkerboard in several positions, you should either bring the Kinects closer or try using another checkerboard.
Doing this will also give you an idea of how much you can tilt the board with respect to a Kinect such that it is still seen properly.

2 Capture

10 to 20 different checkerboard positions should be sufficient to calibrate the Kinects. The checkerboard must be captured closer or further to the Kinects with different inclination. The more different the positions are, the better the calibration will be.
If you want to correct the depth distortion, you should also capture between two and five images of the board, only with the depth camera and such that the depth camera only sees the board.

The next images show the depth maps captured by one Kinect for some possible checkerboard positions. The first two depth maps have no corresponding RGB images and are useful if you want to correct the depth distortion.



There are some important things to check to get usable images. The checkerboard squares must be easy to detect so try not to make it reflect some light toward the camera and do not tilt the board too much. Moreover, **the checkerboard should be seen properly by at least two neighbouring Kinects** for most of the positions, otherwise, the joint calibration won't work properly.

Also, note that the board does not need to be seen entirely but **the checkerboard must be entirely visible on the RGB image to use this image.**

The images should be named starting from 0000-c0.jpg and 0000-d0.pgm, where the first number is the number of the capture and the second is the Kinect or external RGB camera number. "c" correspond to an RGB image and "d" to a depth image. The format must be JPG for the RGB images and PGM for the depth images. **The external RGB camera should always be numbered after the Kinects.**

For example, if we have two Kinects and one external RGB camera and capture 12 positions, we should get the following files :

- 0000-c0 to 0011-c0 and 0000-d0 to 0011-d0 for the first Kinect.
- 0000-c1 to 0011-c1 and 0000-d1 to 0011-d1 for the second Kinect.
- 0000-c2 to 0011-c2 for the external camera

II Calibration

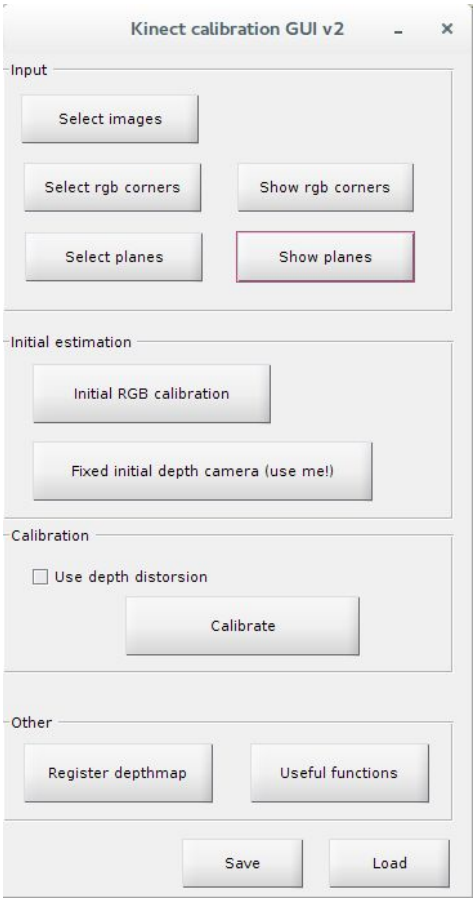
In Matlab, go to the **toolbox** folder and add the following folders to the path:

```
>> cd toolbox/v2.1/toolbox/
>> addpath('missing_fct/')
>> addpath('../../../code_fs/toolbox/Diplomarbeit/')
```

Launch **kinect_calib_gui**, a new window opens with each button launching a different calibration step.

```
>> kinect_calib_gui
```

Launch each step in the order as described below.



1 Select images

Click on the **Select images** button and follow the instructions at the prompt. If you press enter without writing a value, the default value in parenthesis will be used.

- Enter the path to the folder containing the calibration images.
- Enter the number of colour cameras (if you have 3 Kinects and one RGB camera it will be 4).
- Enter the number of depth cameras (if you have 3 Kinects and one RGB camera it will be 3).
- If the files name format differs from the expected one, enter the format you used (without quotation marks).
- Enter then the vector of the images to use. Only use an image if the whole checkerboard is visible by the RGB camera and a sufficient surface of the board is visible by the depth camera.

Example:

```
>> Path to image directory ([]=current dir):>> ../office4/calib
>> Number of color cameras ([]=3):2
>> Number of depth cameras ([]=3):1
>> Filename format for camera 1 images ([]='%.4d-c0.jpg'):
>> Filename format for camera 2 images ([]='%.4d-c1.jpg'):
>> Filename format for depth 1 images ([]='%.4d-d0.pgm'):
   26 plane poses found.
>> Select poses to use for calibration ([]=all):[1,4,6,8:16]
```

2 Select RGB corners

An automatic corner detection is possible but requires the version 2.1 of OpenCV and before launching Matlab, the path to the library must be added every time by entering in the console:
$ LD_LIBRARY_PATH=/usr/local/lib/opencv2.1:$LD_LIBRARY_PATH
or equivalent.

Click on the **Select rgb corners** Button and follow the instructions at the prompt:

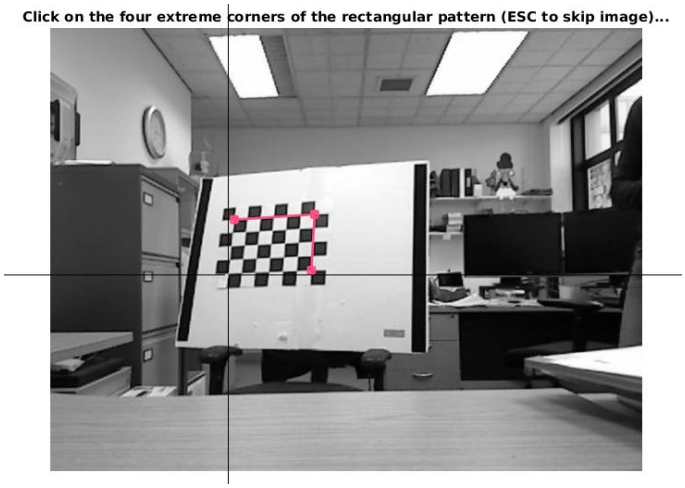
- Enter the size of the checkerboard quads in meters
- If you use the automatic corner detector, you must specify the number of inner corners of the checkerboard. On the image below it would be 7 and 5.
- The Corner finder window size can be left at 3 pixels but you can increase/decrease it if you have a better/worst image quality,
- The `Select poses` line enables you to select the checkerboard only on specific images. For example, if you realise once you are done that your selection in images 5 and 9 are wrong, you can launch a selection again by clicking on the **Select images** Button and in `Select images to process` enter `[5,7]` to only reprocess those images.

Example:

```
>> Square size ([]=0.06m): >>
>> Use automatic corner detector? ([]=true, other=false)? >>
>> Inner corner count in X direction: >> 7
```

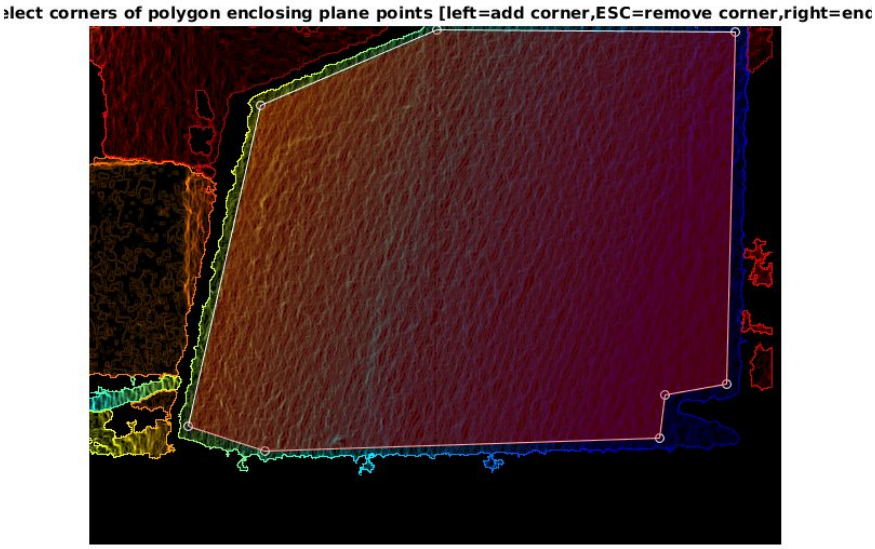
```
>> Inner corner count in Y direction: 5
Camera 1
>> Corner finder window size ([]=3px):
>> Select images to process ([]=all):
```

- When you have to select the corners, select the inner corners.



3 Select planes

Click on the **Select planes** Button and follow the instructions at the prompt. In this part you must create a mask on the plane region of the board, so you should place points **slightly inside the board** and not select the corners of the checkerboard .
You can place as many points as you want and can delete the last one placed with the ESC key.



4 Initial estimation

Clicking on the **Initial estimation** button launches the calibration of the RGB cameras. With 3 Kinects it takes about 30 seconds then the final errors are displayed. A reprojection error is given for each colour camera in pixels and should be much lower than 1 (0.20 pixels for example).

5 Fixed initial depth camera

Clicking on the **Fixed initial depth camera** initialises the depth camera parameters with the known values for Kinect version one and refines them using the colour camera of the same Kinect. With 3 Kinects it takes about 10 seconds. The errors are displayed just before the parameters obtained and look like this:

```
Stats after depth optimization with RGB camera:
Color 1: mean=0.000000, std=0.205617 [-0.011073,+0.012345] (pixels)
Color 2: mean=-0.000000, std=0.232996 [-0.012547,+0.013988] (pixels)
Color 3: mean=-0.000000, std=0.297678 [-0.016031,+0.017872] (pixels)
Depth 1: mean=-0.139493, std=2.386314 [-0.004049,+0.004062] (disparity)
Depth 2: mean=0.018600, std=1.850696 [-0.003180,+0.003191] (disparity)
Depth 3: mean=0.041492, std=2.330167 [-0.007041,+0.007082] (disparity)
```

The mean error should be very close to zero and the disparity error standard deviation should be a couple of disparity units.

6 Calibration

This performs the joint calibration and can take quite a while (up to 30 minutes for 3 Kinects and the depth distortion correction). By unchecking “use depth distortion”, you decrease the computation time.

7 Save

The **Save** button stores in a .mat file the calibration result but also all the calibration variables like the path to the files, the corners positions, the selected plane and the initial calibration, etc… So you can stop and save your calibration at any step to continue it later. This file can then be load with the **Load** button to get all the variables back.

You can load the demo file **../office4/calib/all.mat** to test the calibration steps and the functions described in part 10.
>> Calibration path or filename: ../office4/calib/calib.mat

8 Check the calibration

To check that the calibration gives good results, you can use it to reconstruct a point cloud from a calibration colour image and depth image and visualise it in Matlab with the function **plot_reconstruction.** This also displays the computed position of the cameras.

Example :

```
>> plot_reconstruction(final_calib, dfiles, rfiles, dataset_path, 13)
```

final_calib, dfiles, rfiles and **dataset_path** are four calibration variables that can be added to the workspace by entering **global_vars()** at the prompt after finishing or loading a calibration. **final_calib** is the name of the structure containing the calibration results, **dfiles** and **rfiles** are the names of the colour and depth images used for the calibration and **dataset_path** the path to those files. The last parameter is the index of the image to use to reconstruct a point cloud. All in all, if you used 10 checkerboard positions to do the calibration, you can pick one of those captures (i.e. the corresponding colour image and depth image) by entering an image index between 1 and 10.

This function also displays the position of the cameras according to the calibration results. The cameras (colour and depth) of one Kinect have the same hue and the depth camera is displayed darker than the colour camera.

9 Export the calibration

To be able to use the final calibration parameters in Unreal Engine 4, they have to be written in an XML file with the function:

```
write_xml(calib, rcam_idx, dcam_idx, filename)
```

calib is the calibration variable which can be added to the workspace by entering **global_vars()** at the prompt.

One file is needed for each point cloud, i.e. each Kinect. With two Kinects 0 and 1, two calibration files could be created with the lines:

```
write_xml(final_calib,0,0,'calib_kinect0.xml')
write_xml(final_calib,1,1,'calib_kinect1.xml')
```

One depth camera gives the position of the points and a colour camera gives the colour of the points but we can colour the point cloud of one Kinect with an external colour camera.

For example, if we have 2 Kinects (indexed 0, 1) and two external colour cameras (indexed 2, 3) with the camera 2 close to the Kinect 0 and the 3 close to the Kinect 1. We could generate two calibration files to create two point clouds coloured with the external colour cameras with the following call:

```
write_xml(final_calib,2,0,'calib_kinect0.xml')
write_xml(final_calib,3,1,'calib_kinect2.xml')
```

10 Useful functions

You can get more information on each of those functions with the **help** command followed by the function name in Matlab.

To test those functions on calibration results you can load the demo calibration with:
>> do_load_calib('../office4/calib/all.mat')
>> global_vars
The calibration results are then in the **final_calib** variable.

a) Functions to visualise and export the calibration results

global_vars
This file lists all the global variables used by the calibration toolbox. Run this file to link all variables to the current workspace and access the results.

do_save_calib
Saves the calibration input (corners & planes) data and results.

do_load_calib
Loads the calibration input (corners & planes) data and results.

plot_reconstruction(calib, rfiles, dfiles, data_path, img_idx)
Plots the reconstruction of a plane position from the calibration sequence and the calibration results. Also plots the position of the cameras found by the calibration and the expected position of the checkerboard as a red rectangle.

plot_RGBDpointcloud(calib, rcam_idx, dcam_idx, data_path, img_idx)
Plots a point cloud reconstructed from a colour and rgb image.

plot_rcam(calib,rcam_idx,color)
Plots the position of the colour camera indexed rcam_idx.

plot_dcam(calib,dcam_idx,color)
Plots the position of the depth camera indexed dcam dcam_idx.

plot_checkerboard(calib,img_idx,color)
Plots the expected position of the checkerboard in world space from a calibration image.

dispmap2RGBdepthmap(source_file, dest_file)

Loads a Kinect v1 disparity map output by libfreenect, converts it in an uint32 depth map in micrometres and writes it in a 3 x uint8 RGB image. The most significant byte is written in R and only R,G and B are used since the maximum depth is lower than 2^24 micrometres. The output corresponds to the input depth map expected by the UE4 Kinect renderer (Part 2 of this manual).

write_xml(calib, rcam_idx, dcam_idx, filename)
Writes in an xml file the calibration results stored in the variable CALIB for a pair of colour camera index RCAM_IDX and depth camera DCAM_IDX.

b) Functions to render the calibration results

[img, dmap]=**project_spec**(final_calib) Projects the point clouds got from the data in a virtual camera placed between the two real Kinects, using the calibration data in FINAL_CALIB. The resulting image is written in IMG and the corresponding depth map in DMAP. This uses the data in ../office4/calib and to load the calibration data from ../office4/calib/all.mat.

[img, dmap]=**project_spec**(final_calib, method) specifies a reprojection method, four are possible:
METHOD=

- **'none'** Default method, simply reprojects each point in a pixel, the output image has then holes.
- **'interpolate'** The final image IMG with holes is interpolated using the built depth map DMAP thanks to the function **interpolate_image**.
- **'quads_dist'** Simulates the projection of a cloud of quads instead of a point cloud. Each quad is dimensioned in function of its distance to the camera that captured it
- **'quads_dist_angle'** Same as 'quads_dist' but the quads size also depend on the virtual camera angle compared to the capture camera

You can test those by entering:

```
>> do_load_calib('../office4/calib/all.mat');
>> global_vars;
>> [img,~]=project_spec(final_calib);
>> imshow(img);
>> [img,~]=project_spec(final_calib, 'interpolate');
>> imshow(img);
```

img_int=**interpolate_image**(img,dmap,nb_pix_l,nb_pix_c,dstep,res)
Interpolates the image IMG with white holes using its corresponding depth map DMAP.

II Rendering in UE4

The calibration of the cameras as XML files can be used in the game engine Unreal Engine 4 to reconstruct a captured sequence in 3D and in real-time. First, open the DepthMapRenderer project. You can modify an existing level (they are in the **Map** folder) or create a new one (Files->new->Level).

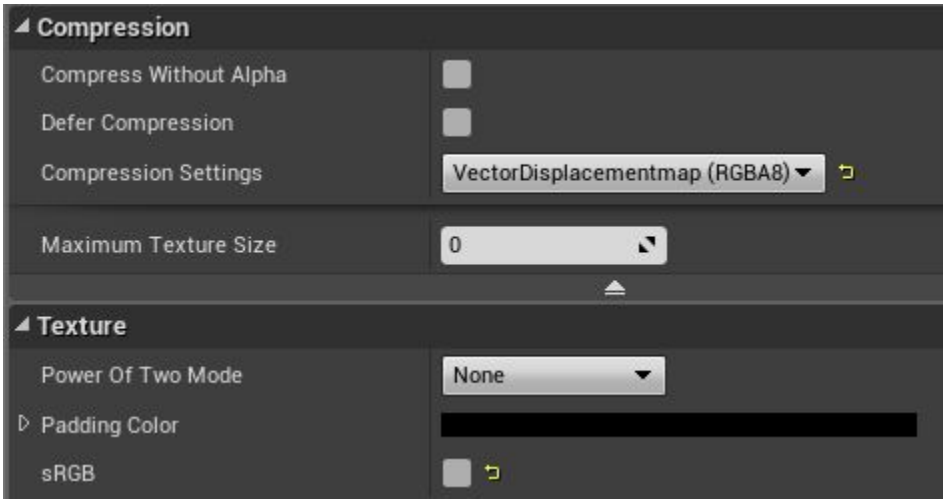
1) Add a point cloud

Each Kinect point cloud will be represented by the actor **DepthMapRenderer**. So first for each Kinect, create a **DepthMapRenderer** actor. Place it in the world at the location (0,0,0) with a rotation of 90 degrees around the x-axis in the right Panel:

Each DepthMapRenderer actor should be placed exactly this way so that the final point clouds superimpose.

You can render either a static scene from an image or a moving scene from a video.

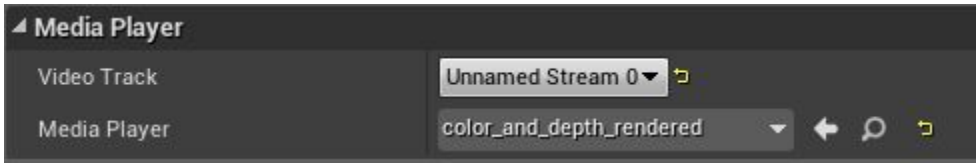
- If your input is a picture, just import it as a texture in Unreal. You then need to open the texture and set the **Compression Settings** to **VectorDisplacementmap (RGBA8) in the right panel.** Without it, the depth map could be unusable because compressed too much. Also, **uncheck the box sRGB** or the scene depth will appear very noisy and scaled down.



- If your Input is a video, you have to create a Media Texture from it. To do so create first a new **Media Player** by clicking on **Add New** in the content browser **-> Miscellaneous -> Media Player.**

Then open it and enter the path to your video in **File or URL:**

Now a media Texture can be created by clicking on **Add New** in the content browser **-> Material and Textures -> Media Texture.** Open it and set Media Player to the one you just created.



2) Create a new Material instance

A **DepthMapRenderer** actor is basically made of a dynamic mesh and a material. The material allows to place the points in space and colour them. The base material for this is **M_DepthMap**, you should find it in the content browser. To feed our images/ video and calibration in this material, a new instance of this one must be created by clicking right on the material **M_DepthMap -> Create Material Instance.** You will need to create one **Material Instance** for each DepthMapRenderer actor so one for each Kinect. For example, the Material Instance you just created can be named MI_DepthMap1 for the first Kinect.

By opening the created Material Instance, you can see a list of parameters on the left side. Some of those have to be set as follows.

ColorMapOffset: Set (R,G) to the position of the first pixel of the colour image in the Texture / Media Texture you just created (values between 0 and 1).

ColorMapScale: Set (R,G) to the size of the colour image in the Texture / Media Texture with respect to the Texture size (values between 0 and 1).

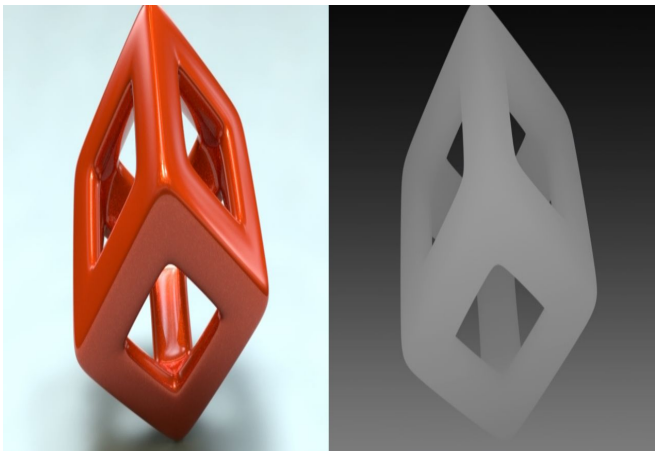
DepthMapOffset: Set (R,G) to the position of the first pixel of the depth image in the Texture / Media Texture(values between 0 and 1).

DepthMapScale: Set (R,G) to the size of the depth image in the Texture / Media Texture with respect to the Texture size (values between 0 and 1).

InputTexture: Select the Texture / Media Texture you created before

For example, with the following Texture :

We will set:
ColorMapOffset: R=0, G=0
ColorMapScale: R=0.5 G=1
DepthMapOffset: R=0.5 G=0
DepthMapScale: R=0.5 G=1

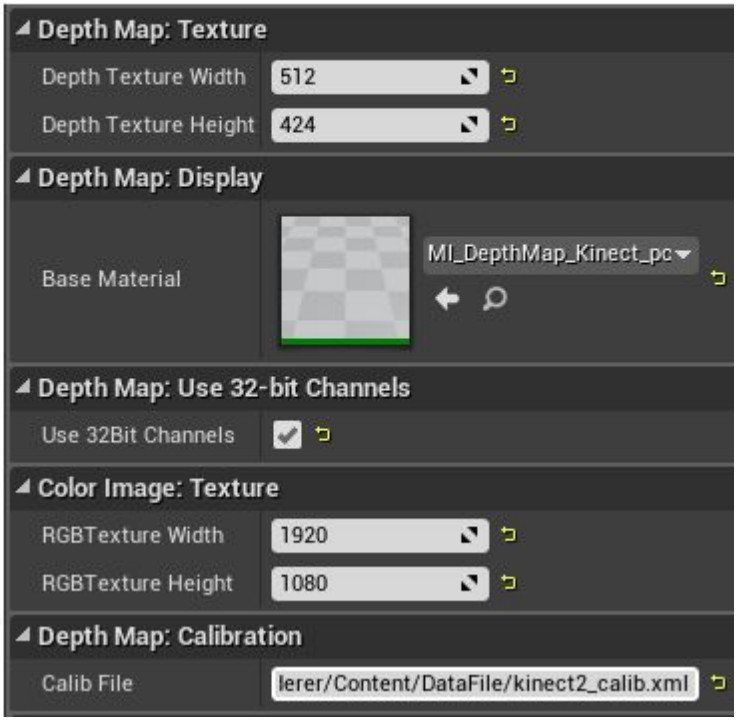


The other parameters should not be modified, they will be set by reading the provided texture and calibration file. (This is done when building the geometry, in the C++ class **ADepthMapRenderer** which is located in **Unreal Projects\DepthRenderer\Source\DepthRenderer.)**

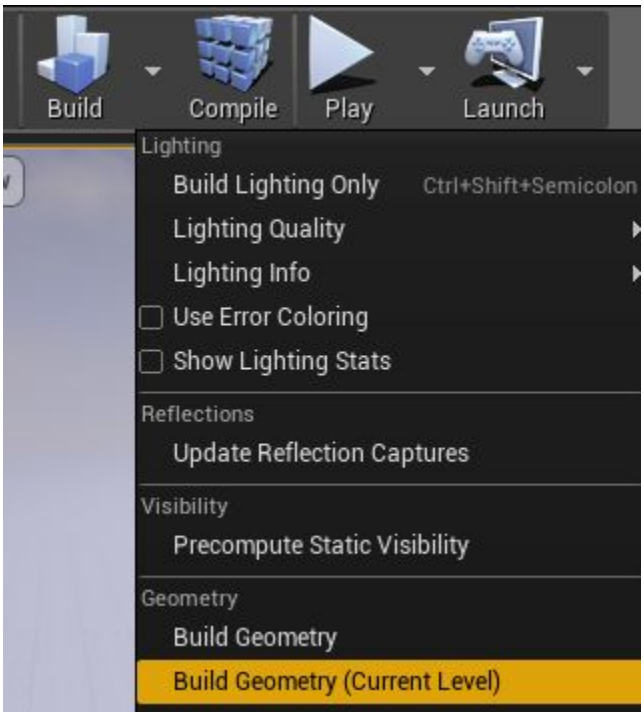
3) Use the material

Back in the level window, there are several properties to set to a **DepthMapRenderer actor.** Select the actor in the World Outliner up right, then you can edit the parameters in the right window.

- In **Depth Texture Width** and **Depth Texture Height**, specify the size of the depth map in pixels.
- In **Base Material**, select the Material Instance you created.
- In **RGB Texture Width** and **RGB Texture Height**, specify the size of the colour image in pixels.
- In **Calib File**, set the path to your .xml calibration file generated with Matlab. Note that the “\” in a path on window must be replaced by “/”.



This done you can click on the arrow right to the Build button and click on Build Geometry (Current Level) to take the changes into account.



Once a Kinect data has been added this way, the visualisation can be launched by clicking on **Play**.

References

[1] Openkinect project wiki. URL: https://openkinect.org/wiki/.

[2] M Balda. An algorithm for nonlinear least squares. In *Conference Technical Computing Prague*, 2007.

[3] M Balda. Lmfsolve. m: Levenberg-marquardt-fletcher algorithm for nonlinear least squares problems, 2009.

[4] Ingo Uwe Bauermann et al. *Acquisition and Streaming of Image-based Scene Representations*. PhD thesis, Technische Universität München, 2008.

[5] Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen. Unstructured lumigraph rendering. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 425–432. ACM, 2001.

[6] Jin-Xiang Chai, Xin Tong, Shing-Chow Chan, and Heung-Yeung Shum. Plenoptic sampling. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 307–318. ACM Press/Addison-Wesley Publishing Co., 2000.

[7] Timothée De Goussencourt and Pascal Bertolino. Using the unity® game engine as a platform for advanced real time cinema image processing. In *Image Processing (ICIP), 2015 IEEE International Conference on*, pages 4146–4149. IEEE, 2015.

[8] Mingsong Dou, Sameh Khamis, Yury Degtyarev, Philip Davidson, Sean Ryan Fanello, Adarsh Kowdle, Sergio Orts Escolano, Christoph Rhemann, David Kim, Jonathan Taylor, et al. Fusion4d: real-time performance capture of challenging scenes. *ACM Transactions on Graphics (TOG)*, 35(4):114, 2016.

[9] Henri Gavin. The levenberg-marquardt method for nonlinear least squares curve-fitting problems, 2011.

[10] Steven J Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F Cohen. The lumigraph. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 43–54. ACM, 1996.

[11] Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[12] Daniel Herrera, Juho Kannala, and Janne Heikkilä. Accurate and practical calibration of a depth and color camera pair. In *International Conference on Computer analysis of images and patterns*, pages 437–445. Springer, 2011. URL http://www.ee.oulu.fi/ dherrera/kinect/.

[13] Daniel Herrera, Juho Kannala, and Janne Heikkilä. Joint depth and color camera calibration with distortion correction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(10):2058–2064, 2012.

[14] Jeff Kramer, Nicolas Burrus, Florian Echtler, Herrera C Daniel, and Matt Parker. *Hacking the Kinect*, volume 268. Springer, 2012.

[15] Marc Levoy and Pat Hanrahan. Light field rendering. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 31–42. ACM, 1996.

[16] Donald W Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.

[17] Microsoft. Kinect for x-box 360., 2010. http://www.xbox.com/en-US/kinect.

[18] Microsoft. Kinect for xbox one., 2013. https://developer.microsoft.com/en-us/windows/kinect/hardware.

[19] Microsoft. Kinect for windows sdk., 2016. http://www.xbox.com/en-US/kinect.

[20] Neil Muller, Lourenço Magaia, and Ben M Herbst. Singular value decomposition, eigenfaces, and 3d reconstructions. *SIAM review*, 46(3):518–545, 2004.

[21] Richard A Newcombe, Dieter Fox, and Steven M Seitz. Dynamicfusion: Reconstruction and tracking of non-rigid scenes in real-time. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 343–352, 2015.

[22] Manuel M Oliveira, Gary Bishop, and David McAllister. Relief texture mapping. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 359–368. ACM Press/Addison-Wesley Publishing Co., 2000.

[23] Ruwen Schnabel, Roland Wahl, and Reinhard Klein. Efficient ransac for point-cloud shape detection. In *Computer graphics forum*, volume 26, pages 214–226. Wiley Online Library, 2007.

[24] Heung-Yeung Shum and Li-Wei He. Rendering with concentric mosaics. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 299–306. ACM Press/Addison-Wesley Publishing Co., 1999.

[25] Jan Smisek, Michal Jancosek, and Tomas Pajdla. 3d with kinect. In *Consumer Depth Cameras for Computer Vision*, pages 3–25. Springer, 2013.

[26] Cha Zhang. *On sampling of image-based rendering data*. PhD thesis, Microsoft Research, 2004.

[27] Cha Zhang and Tsuhan Chen. Spectral analysis for sampling image-based rendering data. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(11):1038–1050, 2003.

[28] Cha Zhang and Tsuhan Chen. Surface plenoptic function: a tool for the sampling analysis of image-based rendering. In *Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03). 2003 IEEE International Conference on*, volume 4, pages IV–768. IEEE, 2003.

[29] Zhengyou Zhang. Flexible camera calibration by viewing a plane from unknown orientations. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 1, pages 666–673. Ieee, 1999.

[30] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Transactions on pattern analysis and machine intelligence*, 22(11):1330–1334, 2000.

[31] Zhengyou Zhang. Microsoft kinect sensor and its effect. *IEEE multimedia*, 19(2):4–10, 2012.

Abstract

The present master thesis studies the use of RGB and depth cameras to capture and render a scene. A pair of calibrated RGB and depth camera provides a coloured point cloud. Using several calibrated camera pairs around a scene gives information on the captured objects shape. This master thesis seeks to setup such a capture system and render the data in a realistic way.

The rendering of point clouds implies several issues. The observer should not be able to see through an object surface and the colours must be consistent. Different techniques are applied and compared to solve those issues. An interpolation technique to fill the holes between the points gives satisfying results for an off-line rendering, while representing the point cloud as a cloud of quads of varying sizes is a good solution for a real-time rendering. To colour the final point cloud, the information of several colour cameras can be used for a more natural illumination of the scene.

Résume

Ce projet de fin d'étude fait l'étude de l'usage de caméras RGB et caméras de profondeur pour capturer et visualiser une scène. Une paire de caméra couleur et profondeur permet de reconstruire un nuage de point coloré. Utiliser plusieurs paires de telles caméras calibrées entre elles donne donc la forme des objets capturées sous forme de points. L'objectif de ce projet est de mettre en place un tel système de capture et de générer un rendu réaliste de ses données.

La visualisation d'un nuage de points amène plusieurs problématiques. L'observateur ne doit pas pouvoir voir à travers une surface et les couleurs de l'objet doivent concorder sur toute sa surface. Plusieurs techniques sont ici appliquées et confrontées afin de résoudre ces problèmes. Une technique d'interpolation pour combler l'espace entre les points donne un rendu satisfaisant mais long à générer, tandis que la représentation des points par des carrés de taille variable constitue une bonne solution pour une visualisation réaliste en temps réel. Pour appliquer la couleur des caméras RGB au nuage de point, les informations apportées par les différentes caméras peuvent être croisées afin d'obtenir une illumination plus naturelle de la scène.