CONFIDENTIAL

SLAM for Drones

Simultaneous Localization and Mapping for autonomous flying robots

José Manuel González de Rueda Ramos

Tutor and Thesis Coordinator: Mr. Dr. Nazih Mechbal

Tutor Host University TU Muenchen: Mr. Dr. Slobodan, Ilic

SLAM FOR DRONES is a revision of the actual State-of-the-Art simultaneous localization and mapping techniques. For this approach we will focus on the super-resolution novel and award winning ISMAR 11' and ICCV 11' paper called Kinect Fusion.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

*I would like to appreciate all the feedback and support received from Arts et Métiers PARISTECH mechatronics department, specially to **Dr. Nazih Mechbal**, not only for proposing me the subject, but also for giving me the guidelines for the thesis. I would like to appreciate indeed **Dr. Michel Vergé** for his inspiration about how to contribute to science and all the stories about his earlier times in science. He is a great scientist and the 2012 promotion we will always remember him.*

*I would also like to remark how necessary has been the contact **Mr. Olaf Malassé**, from Arts et Métiers, and his predisposition for making the exchange with TU Muenchen. I thank him so much the effort for making this possible.*

*Here in TU Muenchen I just have to thank all Computer Aided Medical department not only how well they treated me, but also how they work, their professionalism. Specially, I would like to thanks **Dr. Slobodan Ilic**. who makes the complex world of computer vision a bit more easier.*

SLAM FOR DRONES

“Lo prometido, es deuda”.

LIST OF FIGURES

ABBREVIATIONS

AM	**A**mplitude **M**odulation
AGV	**A**utomated **G**uided **V**ehicle
AR	**A**ugmented **R**eality
AUV	**A**utonomous **U**nderwater **V**ehicle
BF	**B**ilateral **F**ilter
CFA	**C**olor **F**ilter **A**rray
CM	**C**enter of **M**ass
CV	**C**omputer **V**ision
DOF	**D**egrees **O**f **F**reedom
EKF	**E**xtended **K**alman **F**ilter
FLOPS	**FL**oating-point **O**perations **P**er **S**econd
FM	**F**requency **M**odulation
GPS	**G**lobal **P**ositioning **S**ystem
LIDAR	**L**ight **D**etection **A**nd **R**anging
M-Views	**M**ulticamera-**Views**
N-D	**N-D**imensional
PF	**P**article **F**ilter
SDF	**S**igned **D**istance **F**unction
SLAM	**S**imultaneous **L**ocalization **A**nd **M**apping
SoA	**S**tate **o**f the **A**rt

STD	**S**tandard **D**eviation
TSDF	**T**runcated **S**igned **D**istance **F**unction
UAV	**U**nmanned **A**erial **V**ehicle
USB	**U**niversal **S**erial **B**us

PURPOSE OF THIS THESIS

The main objective of this thesis is to be a ***reference in SLAM*** for future work in robotics. It goes from almost a zero-point for a non-expert in the field until a revision of the SoA methods.

It has been carefully divided into ***four parts***:

- The *first one* is a compilation of the *basis in computer vision*. If you are new into the field, it is recommended to read it carefully to really understand the most important concepts that will be applied in further sections.
- The *second part* will be a *full revision from zero of SLAM techniques*, focusing on the award winning KinectFusion and other SoA methods.
- The *third part* goes from a general *flying robots overview* in history until the mechanical model of a quadrotor. It has been intended to be completely apart from section two, for the case it has been determined to only focus on the vision part of this thesis.
- The *fourth part* is a pro-cons overview of the SLAM methods described, applied into flying robots.

We will finish with the *conclusions and future work* of this MSc research.

José Manuel Glez. De Rueda Ramos

INTRODUCTION

The ***importance of robots*** and robotics applications has been increasing exponentially during the last few years, allowing a top-notch industry to start commercializing products for direct consumers. From SLAM based vacuum-cleaners to affordable and mobile remote-controlled quadrotors, this is just the beginning of the extension of years of research into day-to-day applications. Some visionary people consider that the abrupt extrapolation from industrial world (that usually consider that funds are limited but non negligible) to third services/daily-life domain, it is just starting.

Figure 1: Roomba robot cleaner, capable of recognizing unknown house distributions with SLAM

According to the definition, a ***robot*** is a *system that can perform tasks automatically or with guidance* (typically remote control). The main objective of robotics is to perform tedious tasks that humans consider dangerous, difficult, or just boring. So ***robots make our lives easier and happier***.

Figure 2: Our UC3M humanoid, called "Maggie" for which I coded the neck odometry API [65]

The idea of ***automata*** originated in ancient cultures and mythologies around the world including the Ancient China, Ancient Greece and Ptolemaic Egypt [1] . Since then, many ideas, designs and prototypes where manufactured. But it is in 1920 when the interwar Czech writer, Karel Capek introduces the term ***robot*** as a humanoid capable of thinking, and being indistinguishable in a society.

A further detailed description of the history of robotics, centered on unmanned aerial vehicles (UAV's) will be given in *SECTION III*.

It is very important to notice, that robotics is a field that has one remarkable characteristic, and it is that it makes possible to work together many different ***interdisciplinary*** scientists and researchers. For example, just naming some

of the studies we could perform touching robotics field, make us an idea of how big is this science. Some of them are: *mechanical engineering, electric engineering, computer science, medicine, physics, chemistry, materials engineering, aeronautical engineering, military, design, mathematics and probably many more*.

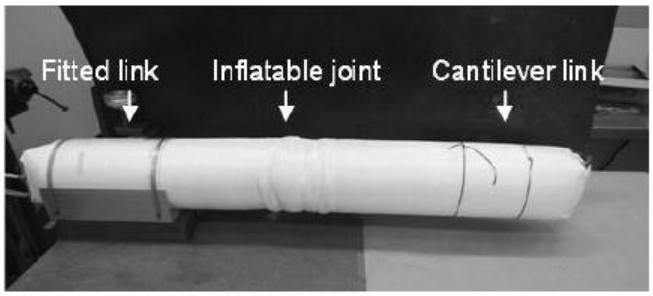


Figure 3: Novel inflatable robot prototype designed by our teachers from Arts et Métiers PARISTECH [64]

It is possible to classify robots according to different ***classifying parameters*** such as:

- **ENVIROMENT**
  - *Land or Home Robots*: They are the most commonly wheeled, but also include legged robots with two or more legs (humanoids, or resembling animals or insects)
  - ***Aerial robots* (UAV's)**[1]
  - *Underwater robots* (AUV's)
  - *Polar robots*: Designed to navigate icy, crevasse filled environments
  - *Space robots*: Designed to outperform in different gravities than Earth

- **TYPES OF MOBILE NAVIGATION**
  - *Manual remote or tele-operated*
  - *Guarded tele-operated*: Manually tele-operated robot, but with the ability to sense and avoid obstacles while navigating
  - *Line-following robot*: Some of the earliest Automated Guided Vehicles (AGVs)
  - *Autonomously randomized robot*: These are autonomous robots with random motion based on bounce off walls, whenever they are detected
  - ***Autonomously guided robot*** [1]**:** Autonomous robots are robots which do not need humans to operate. These robots base their movements in localizing themselves with sensors such as motor encoders, vision, stereopsis (3D vision), lasers, and global positioning systems (GPS). They will usually position themselves using triangulation, relative position and/or Monte-Carlo/Markov localization regarding next waypoint. We can define under this assumptions *two different cases*:

[1] SLAM based UAV's will be the object of our study

- *Featured-based SLAM robots:* Autonomous robots that know some characteristics of the environment. They will try to detect and track those environment features to have better relative position.
- *Full-SLAM robots:* Full autonomous robots that do not know by advance characteristics of the environment.

Many other classifications are possible, but those are enough for us to situate the purpose of our thesis. Now, we have to define what is SLAM. SLAM is the acronym of Simultaneous Localization and Mapping. One of the best definitions I ever read is [2] [2]:

> *SLAM are those techniques that permit robots to give an answer to these **two questions**:*
>
> *- Where am I?*
>
> *- How is the world around me?*

One of the most used sensors right now for performing SLAM due to their price versus information obtained, are cameras. We call this type of SLAM, ***Visual SLAM***. The science which studies the acquisition and image treatment is called Computer Vision (CV). State-of-the-art (SoA) techniques require studying CV in detail to outperform their possibilities.

We will present basic CV notions and concepts in the following Section.

[2] I found the reference reading my friend's Jorge García Bueno Thesis [66]

SECTION 1: BASIC CONCEPTS FOR CV

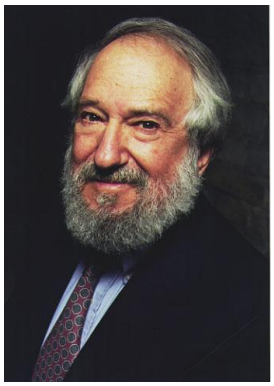


Figure 4: Seymour Papert, one of the pioneers of the AI

In 1966, one of the fathers of Artificial Intelligence, Seymour Papert, wrote a proposal for building a visual system [3]. This proposal divided a ***visual system*** into problems and subtasks that would be achieved by the MIT summer school workers in about ***two months***. The final objective was to achieve a real landmark in the field of "pattern recognition".

Surprisingly for everybody, the project was harder than they expected.

Some of those questions remain today, ***forty years after, still unsolved***.[3]

In this section, we will present the basis of computer vision we need to understand the different SLAM techniques that we will approach in next section number two. For the mathematical parts, we will follow a hybrid approach using geometric and sometimes algebraic methods depending on our needs.

1.1 WHY COMPUTER VISION IS HARD

During my internship in the TU Muenchen (Germany), I followed three courses in CAMPAR *(Computer Aided Medical Procedures for Augmented Reality)* that gave me the opportunity of learning new mathematical algorithms, theory and methods to apply in order to given a determined ***visual input, with some requirements, get an output***, e.g. face detection in a crowded city hall given the security camera video.

I also assisted to some ***presentations***, and one of them was really *inspiring* because of the afterwards discussion. The main topic of the discussion was how computer vision should *evolve* from now on. Some researchers think that emulating human based thinking in vision

[3] The introduction was inspired by the MIT CSAIL 6.869: Advances in Computer Vision course material

will put a constraint in the evolution of computer vision, mainly because computers can do it faster (=better) without programming them to think as humans do.

Thinking and talking about this with my colleagues, we just remarked that in some cases this will be totally true, which means that to ***emulate human thinking*** is a high ***complex*** way of looking up into the reality.

Just analyzing human body, the visual system has the most complex neural circuitry of all the sensory systems.

If we compare the hearing versus the visual sense, the *auditory nerve* contains about 30,000 fibers, but the *optic nerve* contains over one million. Most of what we know about the functional organization of the visual system is derived from experiments similar to those used to investigate the somatic sensory system. The similarities of these systems allow us to identify general principles governing the transformation of sensory information in the brain as well as the organization and functioning of the cerebral cortex. [4]

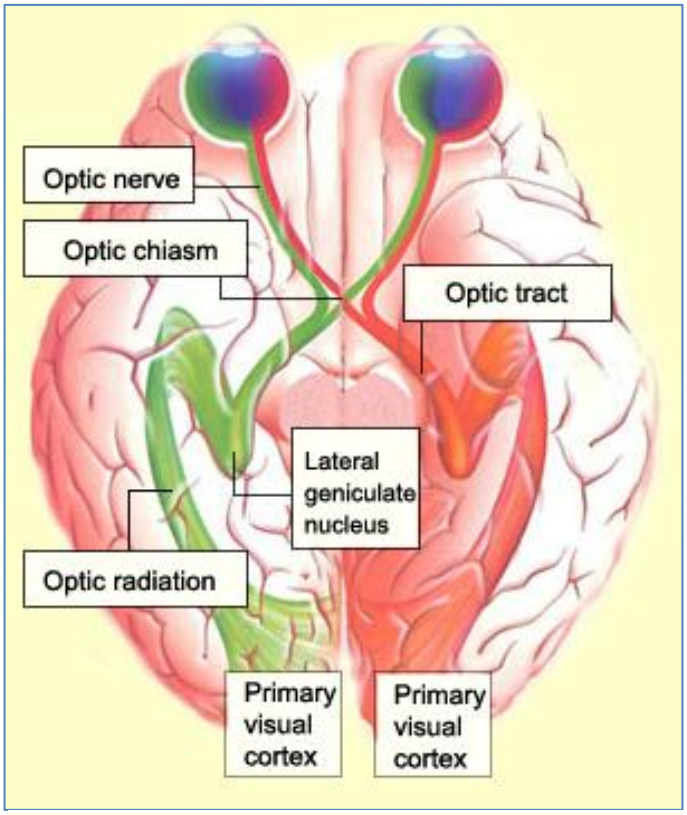


Figure 5: ***A simplified diagram of the projections from the retina to the visual areas of the thalamus (lateral geniculate nucleus) and midbrain (pretectum and superior colliculus)***

We could think that object detection is as simple as looking carefully into a scene, but in reality this is harder than that. Another problem is that we cannot compare computers structure with the innate learning human being has been acquiring through eras.

Right now, computer vision has evolved amazingly, but ***there is still no generic method to all-in-one situation.***

1.2 VISUAL SLAM STARTPOINT: IMAGES

From daily mobile phone augmented applications, to neurobiology, computer vision is a field that has evolved incredibly fast during the last years. The basic information we will need for starting computing our algorithms is an ***image*** (grayscale, rgb, depth image, m-views, etc).

We will describe in this subsection ***how an image is created***.

1.2.1 IMAGE FORMATION (GRAYSCALE - RGB)

Image formation has several components [5]:

- ***An imaging function:*** a fundamental abstraction of an image
- ***A geometric model:*** a projection of the 3D world into a 2D representation
- ***A radiometric model:*** how reflected light is captured by the sensor as raw data
- ***A color model:*** It describes how different spectral measurements are related to image colors

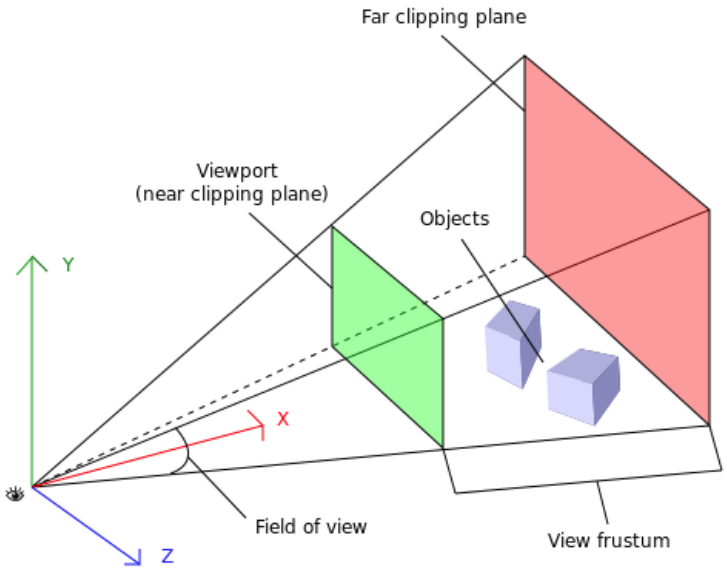


Figure 6: Image formation scheme

The ***basic model for image formation*** is the following:

The scene is illuminated by a single source, and it reflects part of this radiation, *irradiance*, towards the camera. The camera has some lenses to *focus* the image, and a *sensor* at the end that will convert this continuous radiation in a matrix of pixel values:

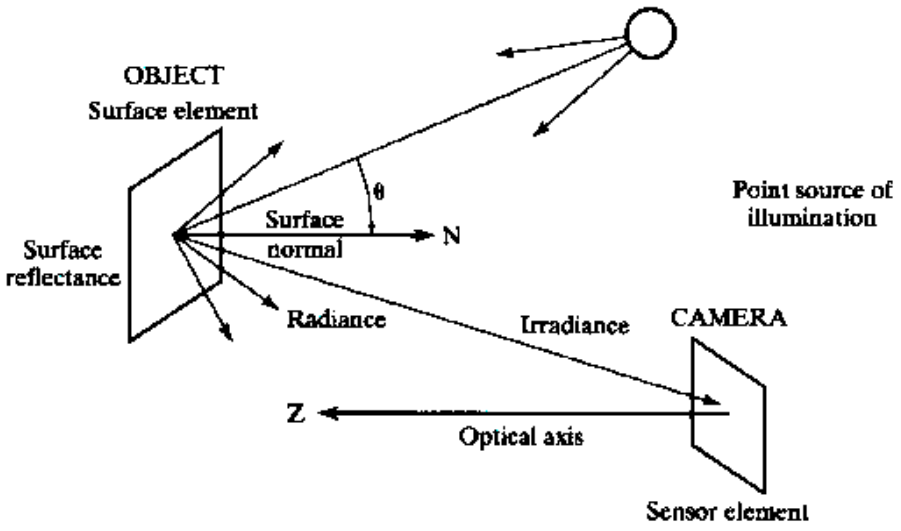


Figure 7: Basic model of a camera

The simplest device to form an image of a 3D scene on a 2D surface is the "*pinhole*" camera. Rays of light pass through a "pinhole" and form an inverted image of the object on the image plane:

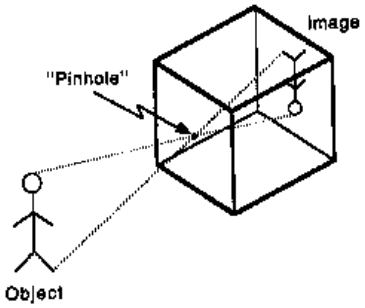


Figure 8: Camera geometry: The "pinhole" camera

In practice, the *aperture* must be larger to admit more light. Because of this, *lenses* are placed in the aperture to focus the bundle of rays from each scene point onto the corresponding point in the image plane:

A measure we have to remember from this scheme is the *focal length* that is the same of saying how strongly the system converges or diverges the incident light.

CCD cameras for example, have an array of tiny solid state cells to convert light energy into electrical charge. Manufactured chips typically measuring about *1cm X 1cm* for *a 512x512* array.

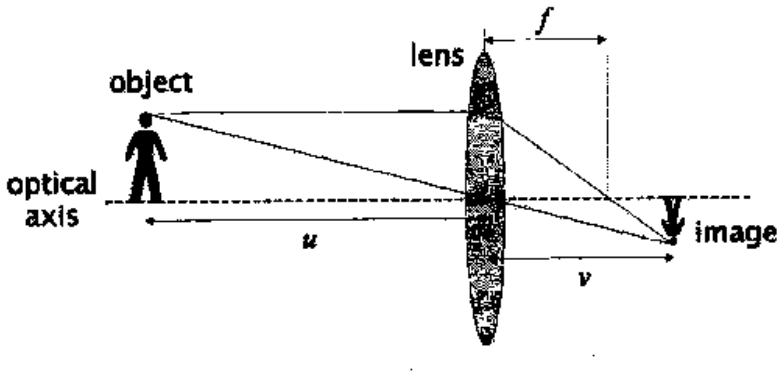


Figure 9: Lens focusing for increasing the aperture

The output of a CCD array *is a continuous electric* signal which is generated by scanning the photo-sensors in a given order and reading out their voltages.

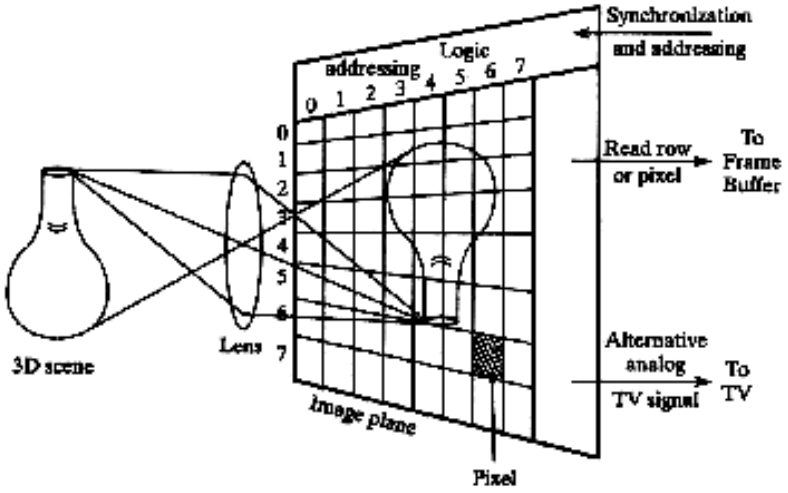


Figure 10: CCD array simulation

From the digital point of view, image formation is the process of computing an image from raw sensor data [5] [6]. The digital after process is described in the following flowing chart:

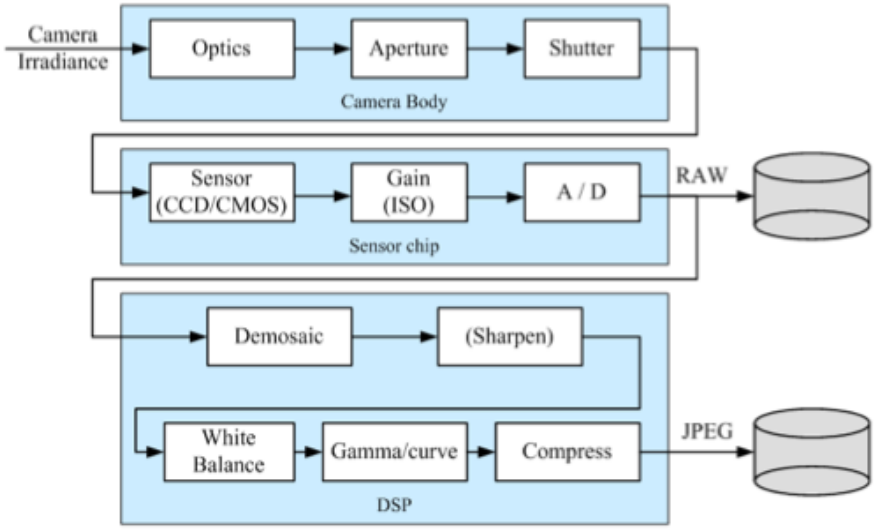


Figure 11: Image formation pipeline showing the typical digital post-processing steps. *Szeliski*

These cameras divide the sensing matrix into three main colors Red, Green and Blue (RGB) applying a Bayer Filter arrangement on the pixel array. A Bayer Filter mosaic is a color filter array (CFA) for arranging RGB colors filters on a square grid of photo sensors. [7]

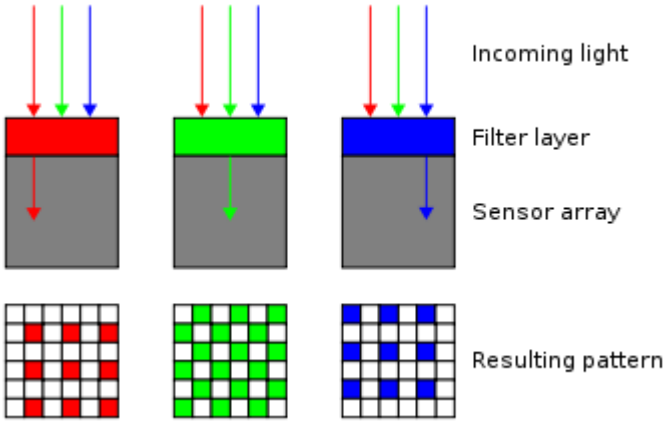


Figure 12: Profile/cross section of sensor. *Source: Wikipedia.com*

Obtaining this mosaic as a final result:

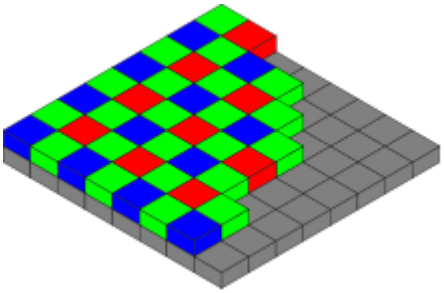


Figure 13: The Bayer arrangement of color filters on the pixel array of an image sensor. *Source: Wikipedia.com*

1.2.2 IMAGE DEPTH FORMATION

Depth information is essential to perform VISUAL SLAM techniques. Although there is a method called MonoSlam that estimates the distance basing its measurements on RGB images, in general we will save time and resources with *depth information systems*.

We can *classify* the different ***depth measurement techniques*** as following:

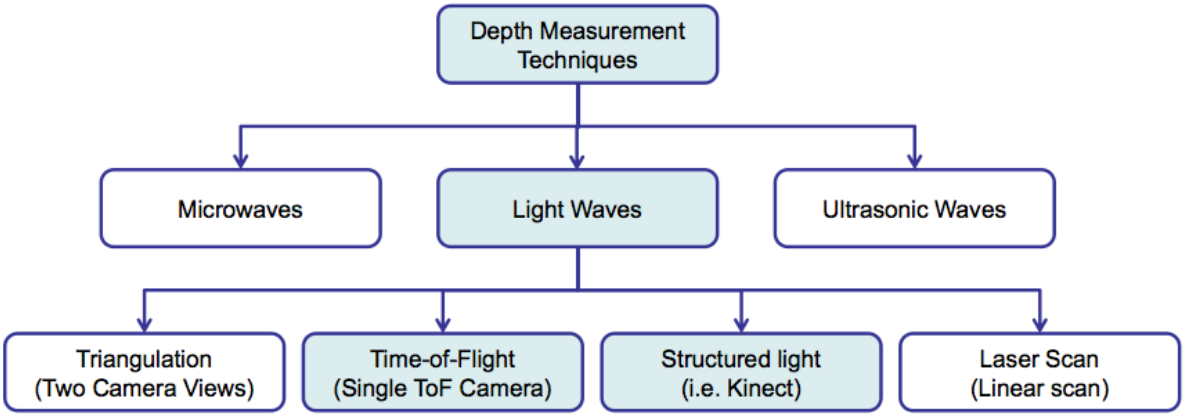


Figure 14: Depth measurement techniques. [8]

Light waves based cameras are the most extended in robotics. Inside these we can remark four methods:

- *Triangulation with two up to M-Camera views*
- *ToF camera*
- *Structured Light*
- *Linear Scanning*

i) TRIANGULATION METHODS (M-VIEWS)

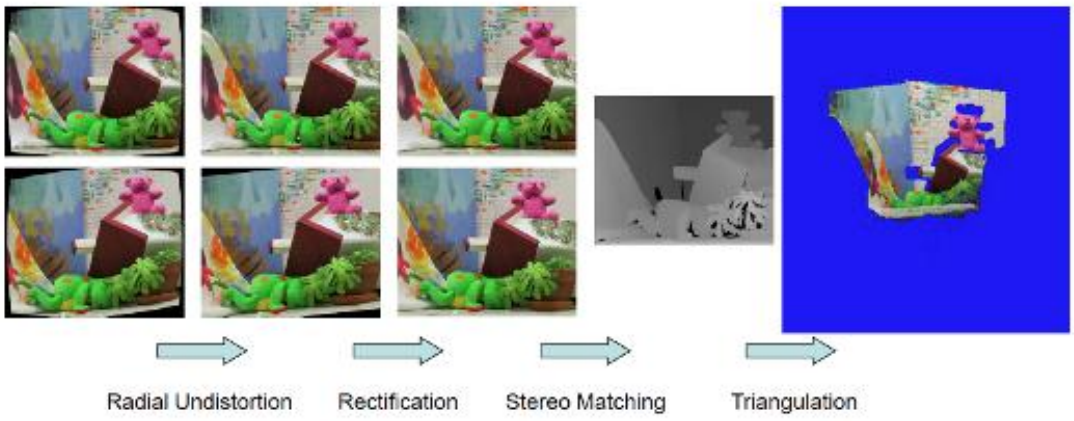


Figure 15: Triangulation process layout

In this method we will estimate distance to objects based on the *divergence* of two or more views with different position of the camera center (stereo matching). Human vision uses this technique. The main ***disadvantages*** are:

- *Need of calibrated cameras (also between them)*
- *High computational costs due to the M-Views*
- *Dependence on scene illumination*
- *Dependence on surface texturing*

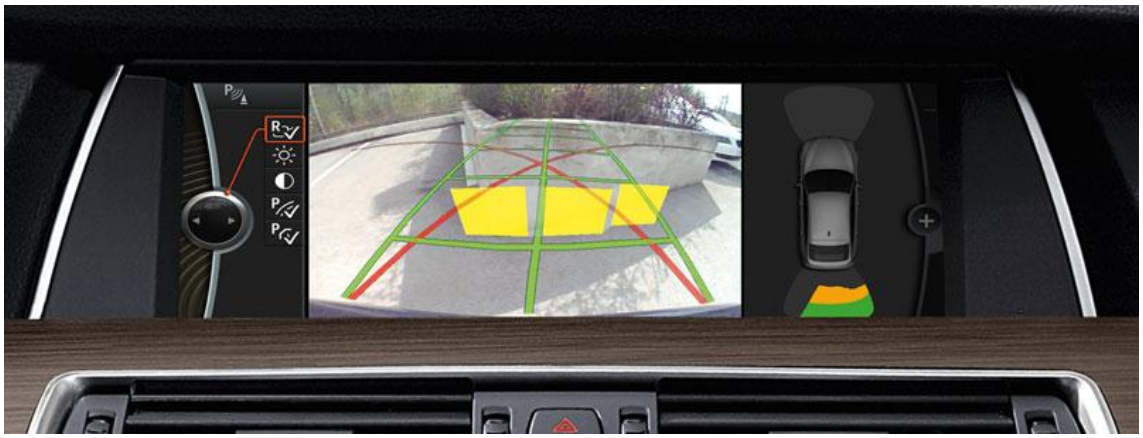


Figure 16: BMW series 5 camera system [68]

ii) TIME-OF-FLIGHT CAMERAS (TOF)

These cameras help us to measure distance to objects using one of these two techniques:

The first possible method ***PULSED MODULATION***, measures the time it takes for a beam of light (which speed is known and is $c = 3.10^8\ m/s$) emitted from the camera until the object, be reflected, and return to the camera sensor. [9]

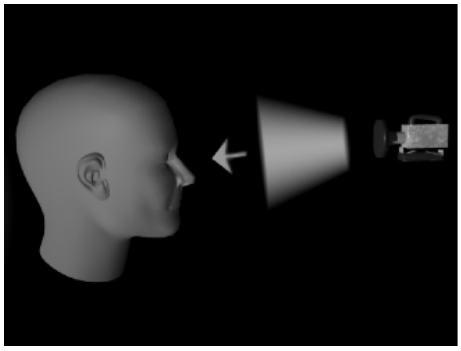


Figure 17: Light emitted from the camera

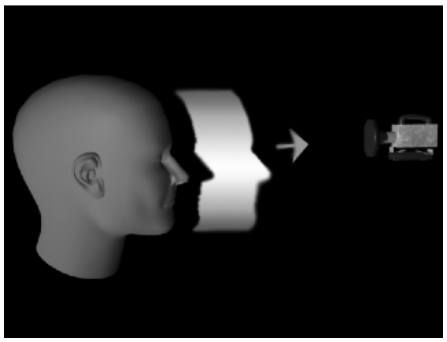


Figure 18: Light reflected from the camera

Their distance resolution ranges from sub-centimeter to several centimeters depending upon the range. The *lateral resolution* is generally low compared to standard 2D video cameras, with most commercially available devices at 320x240 pixels or less (2011) [10]. Near-infrared light $700{\sim}1400nm$ is used in this type of devices.

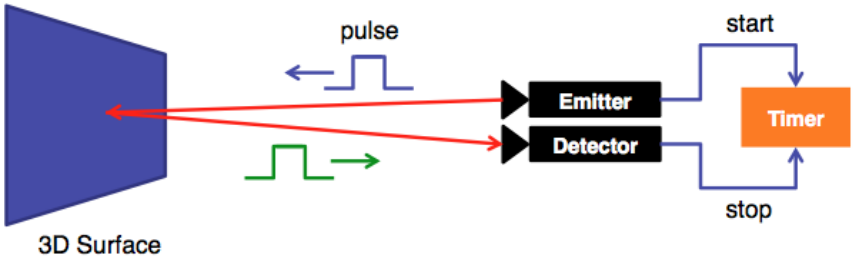


Figure 19: Pulse modulation ToF. CAMPAR Lecture

The main ***advantages*** are:

- *We will only need one camera, and not two or more as in the M-View technique*
- *Light/illumination influence is lower as beams are high-energy rays*
- *No need of external illumination source!*

The main ***disadvantages*** of this method are:

- *We will need a high-accuracy measurement of the time between the emission and detection*
- *Due to light scattering, the measurement of light is inexact*
- *It is very difficult to produce pulses with low periods of time, so we will need to be much more static than in the m-view method*

The second method used in ToF is ***CONTINUOUS WAVE MODULATION***. In this method, we will send continuous *light waves* instead of short light pulses. We typically use modulated sinusoidal waves, and detect the wave after reflection has shifted their phase. This phase shift will be proportional to the distance from the reflecting surface. A scheme is shown in next figure:

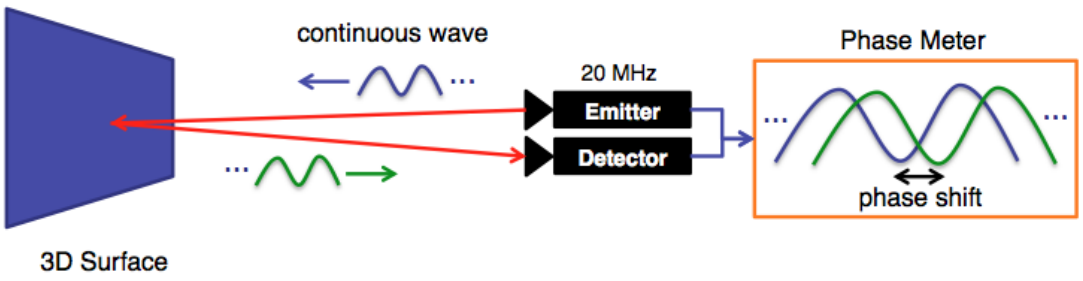


Figure 20: Continuous wave modulation. CAMPAR Lecture

The formula for calculating the *distance to an object* is:

$$d = \frac{c\phi}{4\pi\omega} \quad \text{where:}$$

$c = 3.10^8$ *is the speed light*

ϕ *is the phase shift*

ω *is the modulation frequency*

The main ***advantages*** are:

- *We can have a different types of emitted lights as (in intensity/speed) as we will focus only on the phase shift.*
- *We can use different modulation techniques (not only in frequency (FM), also in amplitude (AM)).*
- *We can obtain simultaneously range and amplitude range.*

The main ***disadvantages*** are due to the *cross-relation function* (it is used to calculate the distance to an object formula). We will need to convolve, that means integrate, input and output signals to calculate the distance to the object. This integer operation is *not fast* so we will have our limitations:

- *Frame rates are limited by this integration time*
- *We will need to reduce noise over time*
- *With long integration time, we will have motion blur*

An example of range/amplitude is presented in the next figure. We will only get amplitude images with FM, not in AM.

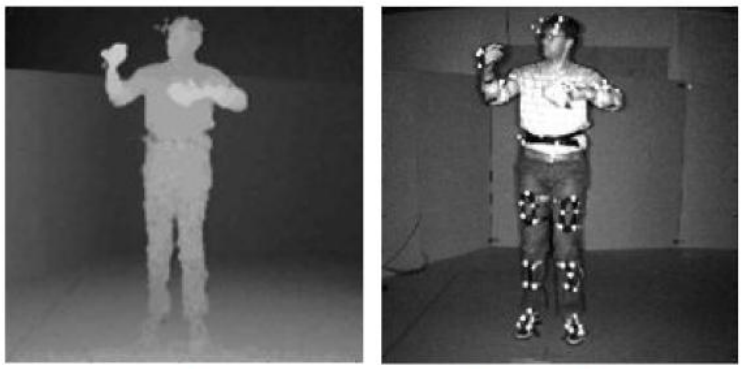


Figure 21: Range versus Amplitude images in a ToF camera

Other methods combine RGB with ToF cameras in a likehood M-View system.

Figure 22: Canesta [11] ToF camera

iii) STRUCTURED LIGHT (KINECT)

Primasense and Microsoft product *Kinect* has been a revolution for the computer vision society. Its low price allows students and researches to have depth information as well as RGB information on the same device for a *price* under two hundred euros, which is much less than typical ToF camera prices.

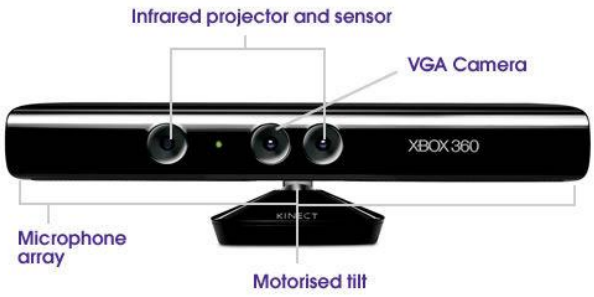


Figure 23: Microsoft Kinect

It counts with a *microphone array*, a *motor* to tilt the camera up and down, a *structured-light camera* (infrared projector and sensor 320x240 16-bit depths @ 30 frames/sec) and a *RGB* camera (VGA resolution 640x480 32-bit depth @ 30 frames/sec). It also counts with a *3-axis accelerometer* and a state multicolor *blinking led*.

Figure 24: Asus XTION camera

Other models as the ASUS camera have only the depth camera and do not need external power (only powered with the USB connection, 5Volts). [4]

The principle of structured light

The basic concept of structured light method is to send a *known band of light* onto a three-dimensionally surface (usually collinear light), and capture it from other perspectives that will seem to be different and distorted. This can be used for a geometric reconstruction of the surface shape. [12]

We can notice in the next figure that from the point of view of the left camera, the structure of the light is *parallel vertical lines*. By the other hand, for the right camera, the stripes will become *undulated lines*. This "waves" will have relation with the 3D shape of the object:

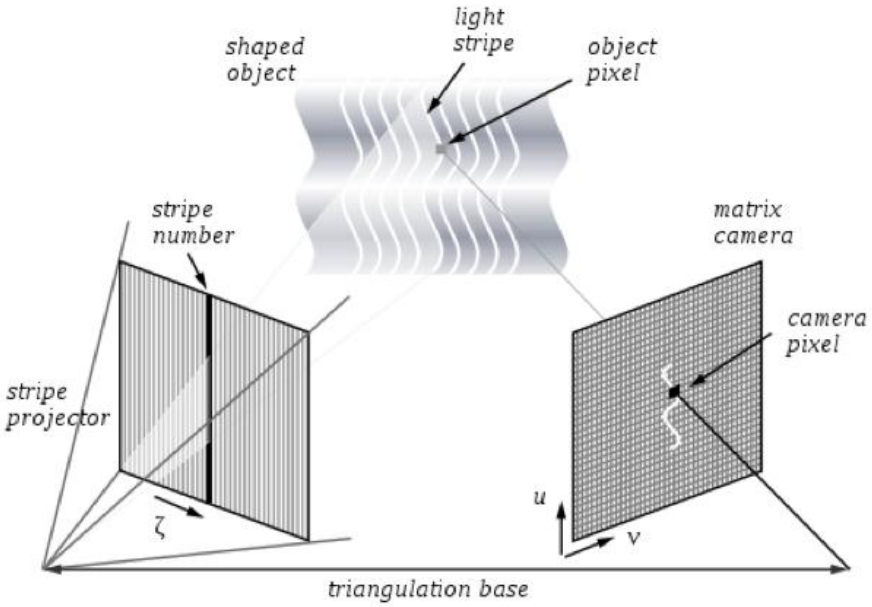


Figure 25: Structured light process example with collinear-stripe based pattern

Depending on the system, we can achieve typical ***accuracy*** figures as:

- *Planarity of 60cm wide surface, to 10µm*
- *Radius of a blade edge of e.g.* $10\mu m$*, to* $\pm 0.4\mu m$

[4] During writing the thesis a new ASUS model, the Xtion PRO LIVE has appeared on the market with a RGB camera integrated.

In example:

Figure 26: Industrial example of 3D modeling of a car seat

The main problem we will focus when dealing with ***Microsoft Kinect*** it is low accuracy from a few millimeters up to about 4cm at the maximum range of the sensor when calibrated. [13]

I recommend Khoshelham work for further references. He made several tests to calculate the camera ***depth error***, obtaining this result:

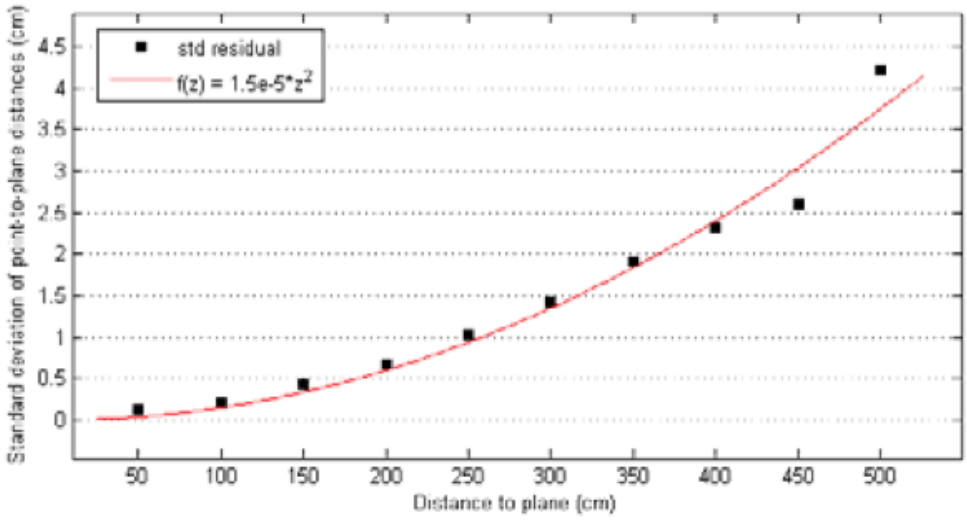


Figure 27: Std deviation of the measurements. *Khoshelham tests.*

In conclusion, when dealing with ***Kinect***, we should be concerned about the following ***problems***:

- *The **random error of depth measurements** increases quadratically with increasing distance from the sensor and reaches a top 4cm at the maximum range.*
- *The **density of points** also decreases with increasing distance to the sensor. We will have a constant around 30,000 points, which leads at very low density at large distance (7cm at the maximum range of 5m)*
- *In general, for mapping applications, the **data** should be acquired within $1{\sim}3m$ distance to the sensor. At larger distances, the quality of the data is degraded by the noise and low resolution of the measurements.*

This means that doing SLAM with Kinect will be not as dense as other structured light systems, or even 3D laser scanners.

The ***main advantages of Microsoft Kinect*** are the *price, size and all the software developed around it.*

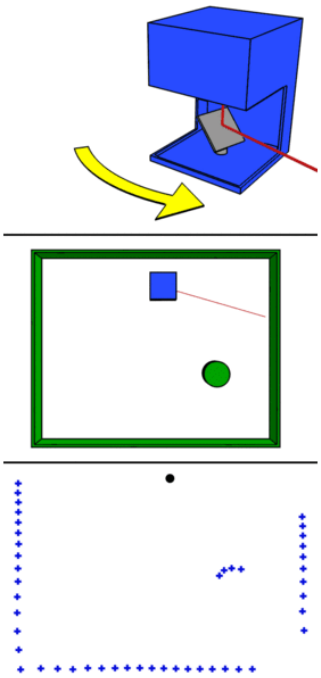


Figure 28: LIDAR system, only point-to-point capturing instead of the whole scene

iv) LASER SCAN

Laser scanner (LIDAR systems) have *more precision* than other methods, but their main ***disadvantages*** are the speed and the price of these systems.

Due to laser precision, we will be able to get high accurate models, but the main problem is that laser is a high energy wave that needs to be rotated among the space to get, point by point, all considered data. This is solved using multiple laser sources to get many points at once, but this is really expensive compare to an almost infrared pattern emitter that uses the Kinect with the Structured Light system.

Figure 29: Commercial 3D laser handheld scanner.

Price around 1500€

1.3 GEOMETRY PRIMITIVES AND TRANSFORMATIONS

We will introduce 2D and 3D primitives, namely points, lines and planes. We will also describe the process by which 3D features are projected into 2D planes. [14] [15] [16]

1.3.1 Points, lines, planes and quadrics in 2D

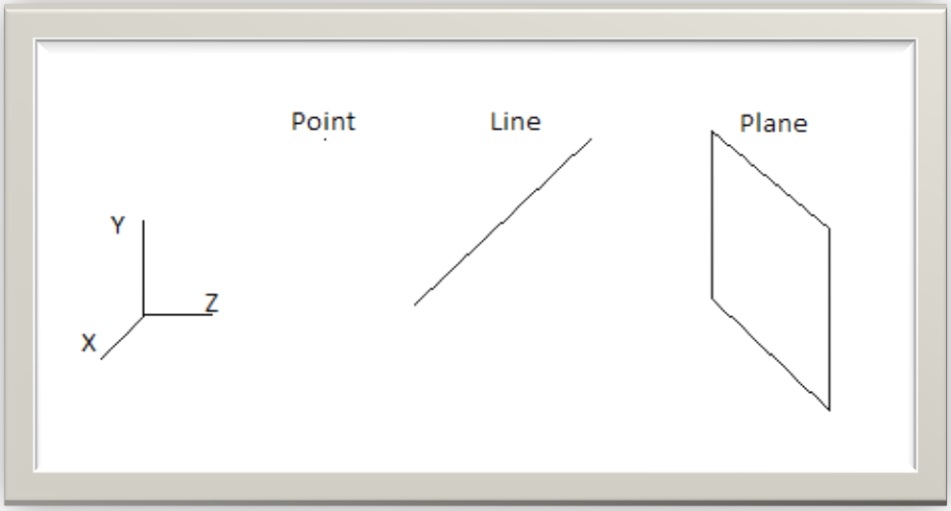


Figure 30 Point, line and plane in the space. *José Manuel Glez. De Rueda Ramos*

i) Point in a plane

2D points can be denoted using a pair of values $\boldsymbol{x} = (x, y)^T \in R^2$. We will call them "pixels" when referring to an image.

ii) Projective space P^2

Correspondence between lines and vectors is not one-to-one, since the lines $ax + by + c = 0$ and $(ka)x + (kb)y + (kc) = 0$ are the same, but two proportional vectors represent the same line, for example $(a, b, c)^T$ and $k(a, b, c)^T$ for any non-zero constant k represent the same line. So any particular vector $(a, b, c)^T$ is a representative of the equivalence class. The set of classes of vectors in $R^3 - (0,0,0)^T$ forms the projective space P^2 also called projective space.

iii) Homogeneous vector

This equivalent set of vectors is known as homogeneous vector.

iv) Homogeneous representation of lines

A line in the plane is represented by an equation such as $ax + by + c = 0$ / a,b,c are constant parameters[5]. Thus, with this approach, a line could be represented by the vector $\boldsymbol{l} = (a, b, c)^T$. These are homogeneous coordinates.

We can also normalize the line equation vector so that $\boldsymbol{l} = (\hat{n}_x, \hat{n}_y, d)$ with $\|\hat{n}\| = 1$. In this case, $\hat{\boldsymbol{n}}$ is the normal vector orthogonal to the line and *d* is its distance to the origin.

Another option is to express $\hat{\boldsymbol{n}}$ as a function of rotation angle θ, $\hat{\boldsymbol{n}} = (\hat{n}_x, \hat{n}_y) = (\cos\theta, sin\theta)$. The combination of (θ, d) is called *polar coordinates*.

A full description is in *Figure 3:*

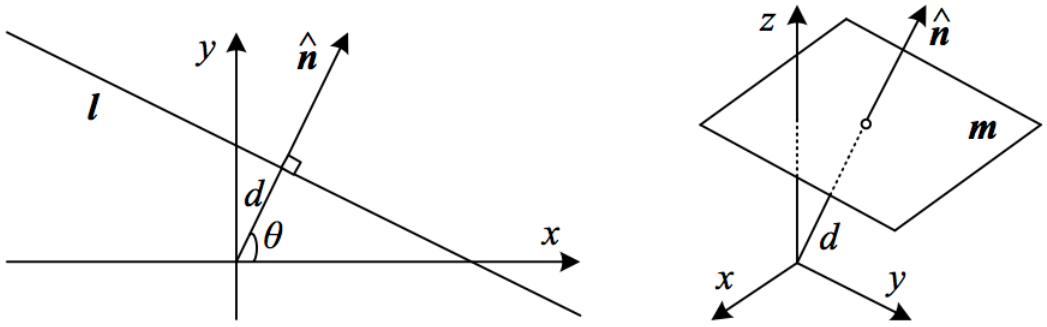


Figure 31: (left) 2D line equation (right) 3D plane equation, expressed in terms of the normal n **and the distance to the origin** d. **(*Szeliski p.33*)**

v) Homogeneous representation of points

A point $x = (x, y)^T$ lies on the line $l = (a, b, c)^T$ if and only if $ax + by + c = 0$. This can be also written in terms of an inner product of vectors representing the point as $(x, y, 1)(a, b, c)^T = 0$. This means that the point $(x, y)^T$ in R^2 is represented as a 3-vector by adding a final coordinate of 1. An arbitrary homogeneous vector representative of a point is of the form$x = (x_1, x_2, x_3)^T$, representing the point $\left(\frac{x_1}{x_3}, \frac{x_2}{x_3}, \frac{x_3}{x_3}\right)^T$

vi) Degrees of freedom (DOF)

The degrees of freedom of an object are the number of parameters needed to be specified in order to fix it in the space.

vii) Intersections

The intersection of two lines can be computed as $\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{l}}_1 \times \tilde{\boldsymbol{l}}_2$. In the same way, the line joining two points can be written as $\tilde{\boldsymbol{l}} = \tilde{\boldsymbol{x}}_1 \times \tilde{\boldsymbol{x}}_2$.

[5] The symbol "/ " means where in the mathematical language.

viii) Points in the infinity

Consider two parallel lines: $ax + by + c = 0$ and $ax + by + c' = 0$, which can be represented by the vectors $l = (a, b, c)^T$ and $l' = (a, b, c')^T$. If we try to compute their intersection (we consider that they are not coincident, so $c \neq c'$), we will get the non-sense result: $\boldsymbol{l} \times \boldsymbol{l'} = (c' - c)(b, -a, 0)^T$. Ignoring the scaling factor $c' - c$, we will get the point $(b, -a, 0)^T$.

Attempting to get the non-homogeneous representation of this point, we observe that $\left(\frac{b}{0}, -\frac{a}{0}\right)^T$, that is the limit with the tendency $(\pm\infty, \mp\infty)$ where the sign depends on the constants a, b. With this result, we can conclude that we can describe a point in the infinity with homogeneous coordinates $(x, y, 0)^T$.

ix) Line at the infinity

With the same logic, we can compute the line at the infinity as the $\tilde{\boldsymbol{l}}_\infty = \tilde{\boldsymbol{x}}_{\boldsymbol{1}\infty} \times \tilde{\boldsymbol{x}}_{\boldsymbol{2}\infty}$**.** This set lies in $\boldsymbol{l}_\infty = (0,0,1)^T$, which verifies $(0,0,1)(x_1, x_2, 0)^T = 0$.

x) Conics

A conic is a curve described by a second-degree equation in the plane. In Euclidean geometry conics are of three main types: *hyperbola, ellipse* and *parabola* (we do not consider degenerated cases now). This classification becomes from the intersection between a plane and cones. In a 2D projective geometry, all non-degenerate conics are equivalent under projective transformations.

The equation of a conic in inhomogeneous coordinates is:

$$ax^2 + bxy + cy^2 + dx + ey + f = 0 \quad (1)$$

This is a second order polynomial. In order to homogenizing this equation, we can make the variable changes $VC_1: x \to \frac{x_1}{x_3}$ and $VC_2: y \to x_2/x_3$, so the equation (1) becomes:

$$ax_1^2 + bx_1x_2 + c_2^2 + dx_1x_3 + ex_2x_3 + fx_3^2 = 0$$

In a matrix form:

$$x^T Cx = 0 \,/ \qquad C \; is \; the \; conic \; coefficient \; matrix \; c = \begin{bmatrix} a & \frac{b}{2} & \frac{d}{2} \\ \frac{b}{2} & c & \frac{e}{2} \\ \frac{d}{2} & \frac{e}{2} & f \end{bmatrix}$$

Quadric equations are really useful when studying multi-view geometry and camera calibration.

1.3.2 Points, lines, planes and quadrics in 3D

i) 3D point

A point in 3D can be expressed as $\boldsymbol{x} = (x, y, z) \in R^3$ in inhomogeneous coordinates, and as $\tilde{\boldsymbol{x}} = (\tilde{x}, \tilde{y}, \tilde{z}, \tilde{w}) \in P^3$. We can denote a 3D point as before using the augmented vector $\hat{\boldsymbol{x}} = (x, y, z, 1)$ where $\tilde{\boldsymbol{x}} = \tilde{w}\hat{\boldsymbol{x}}$.

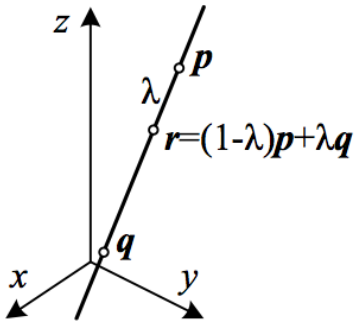


Figure 32: 3D line equation representation. *Szeliski p.34*

ii) 3D line

We can represent a 3D line using two points of itself $(\boldsymbol{p}, \boldsymbol{q})$. We can use the vector defined by those points and apply a proportional factor to define all points:

$$\boldsymbol{r} = (1 - \lambda)\boldsymbol{p} + \lambda\boldsymbol{q}$$

We can define it again, but using this time homogeneous coordinates: $\tilde{\boldsymbol{r}} = \mu\tilde{\boldsymbol{p}} + \lambda\,\tilde{\boldsymbol{q}}$

If we consider now the dof for 3D lines, we could think that it has six (three for each endpoint) instead of the four that a line truly has. If we fix the two points on the line to lie in specific planes, we obtain a representation with four degrees of freedom.[6]

iii) 3D planes

It is also possible to represent a plane in homogeneous coordinates $\tilde{\boldsymbol{m}} = (a, b, c, d)$ with a corresponding plane equation:

$$\hat{\boldsymbol{x}} \cdot \tilde{\boldsymbol{m}} = ax + by + cz + d = 0$$

We could again normalize referring this time to the normal vector $\hat{\boldsymbol{n}}$ as a function of two angles (θ, ϕ):

$$\hat{\boldsymbol{n}} = (cos\theta cos\phi, sin\theta cos\phi, sin\phi)$$

[6] For more info, consider reading chapter 2.1 of Szeliski (*Computer Vision: Algorithms and Applications,* 2010)

iv) 3D quadric

The analog of a conic section in 3D is a conic surface $\bar{\boldsymbol{x}}^{\boldsymbol{T}}\boldsymbol{Q}\boldsymbol{x} = 0$

1.3.3 2D Transformations

Geometry can be seen as the study of invariant properties under groups of transformations [17]. In R^2, we can define a projective transformation as:

$$\begin{pmatrix} x_1' \\ x_2' \\ x_3' \end{pmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

Or we can resume as: $x' = Hx$

Depending on H we can define a hierarchy of 2D coordinate transformations. These transformations are:

i) Translation

A 2D translation can be defined as $\boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{t}$ which equals $\boldsymbol{x}' = [\boldsymbol{I} \quad \boldsymbol{t}]\ \bar{\boldsymbol{x}}$. We can use homogeneous coordinates studied before to use a more compact notation:

$$\bar{x}' = \begin{bmatrix} I & t \\ 0^T & 1 \end{bmatrix} \bar{x}$$

ii) Euclidean transformation

This transformation is also known as *rigid body motion* or the *Euclidean (2D) transformation.* It can be written as $\boldsymbol{x}' = \boldsymbol{R}\boldsymbol{x} + \boldsymbol{t}$ in its non-homogeneous form. **R** is the rotation matrix $\boldsymbol{R} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$e properties of this matrix is that this matrix is orthonormal: $\boldsymbol{R}\boldsymbol{R}^{\boldsymbol{T}} = \boldsymbol{I}$ and $|\boldsymbol{R}| = 1$.

iii) Similarity transformation

This transformation can be expressed as $\boldsymbol{x}' = \boldsymbol{s}\boldsymbol{R}\boldsymbol{x} + \boldsymbol{t}$ which leads to $\boldsymbol{x}' = [s\boldsymbol{R} \quad t] = \begin{bmatrix} \boldsymbol{a} & -\boldsymbol{b} & \boldsymbol{t_x} \\ \boldsymbol{b} & \boldsymbol{a} & \boldsymbol{t_y} \end{bmatrix}$ where $a^2 + b^2$ could be in this case different than one.

iv) Affine transformation

In this transformation, parallel lines remain still parallel. This transformation can be written as $\boldsymbol{x}' = \boldsymbol{A}\boldsymbol{x}$ / **A** is an arbitrary 2x3 matrix.

v) Projective transformation

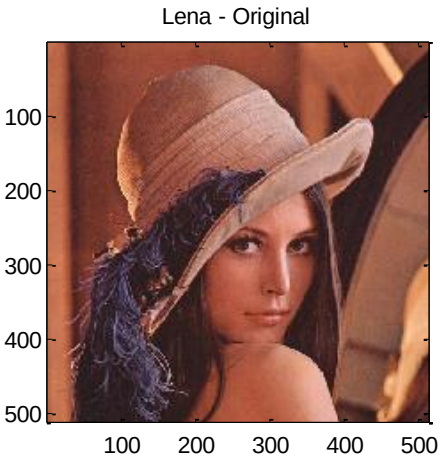
This transformation is also known as *perspective transform* or *homography*. It can be written homogenously as $\tilde{\boldsymbol{x}}' = \tilde{\boldsymbol{H}}\tilde{\boldsymbol{x}}$. Note that $\tilde{\boldsymbol{H}}$ is in this case an arbitrary 3x3 matrix, where typically $H_{31} = H_{32} = 0$ and $H_{33} = 1$.

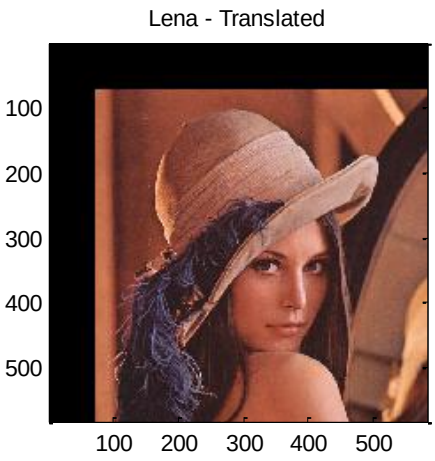
Transformation	*Matrix*	*# Dof*	*Invariant to*	*Example*
Translation	$[I\mid t]_{2x3}$	2	Orientation	
Euclidean (rigid)	$[R\mid t]_{2x3}$	3	Lengths	
Similarity	$[sR\mid t]_{2x3}$	4	Angles	
Affine	$[A]_{2x3}$	6	Parallelism	
Projective	$[\widetilde{H}]_{3x3}$	8	Straight lines	

To see the results we can use ***Matlab*** [18] software to implement these functions. The code is provided in the annex number one, at the end of this document.

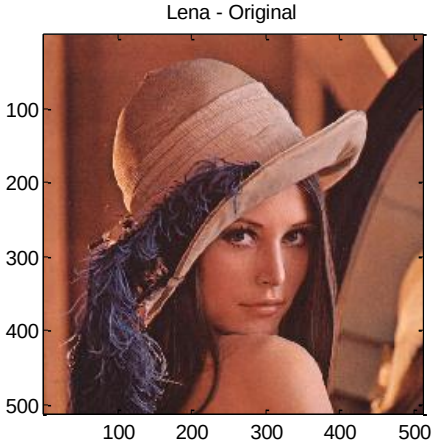
Here is the *output*:

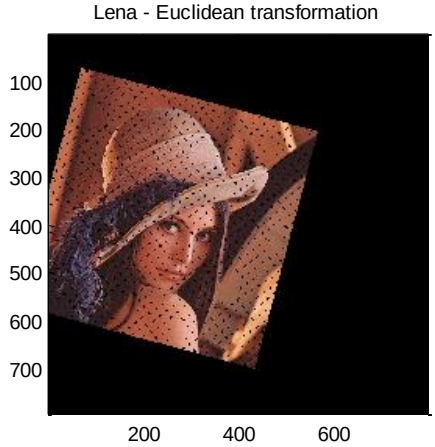
```
--- Transformations example ---

+ Initializing... init okey
+ Translation example
  Please introduce a number of pixels to translate in x: (ex: 70)
  Please introduce a number of pixels to translate in y: (ex: 70)
```



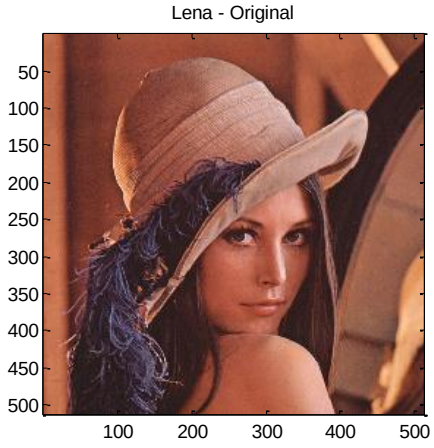


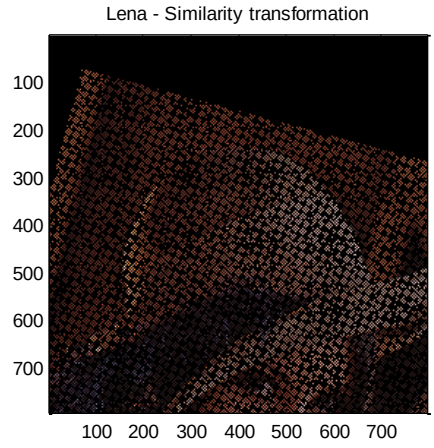
+ Euclidean transformation example
  Please introduce a number of pixels to translate in x: (ex: 70)
  Please introduce a number of pixels to translate in y: (ex: 70)
  Please introduce the rotation angle (grads): (ex: 15)



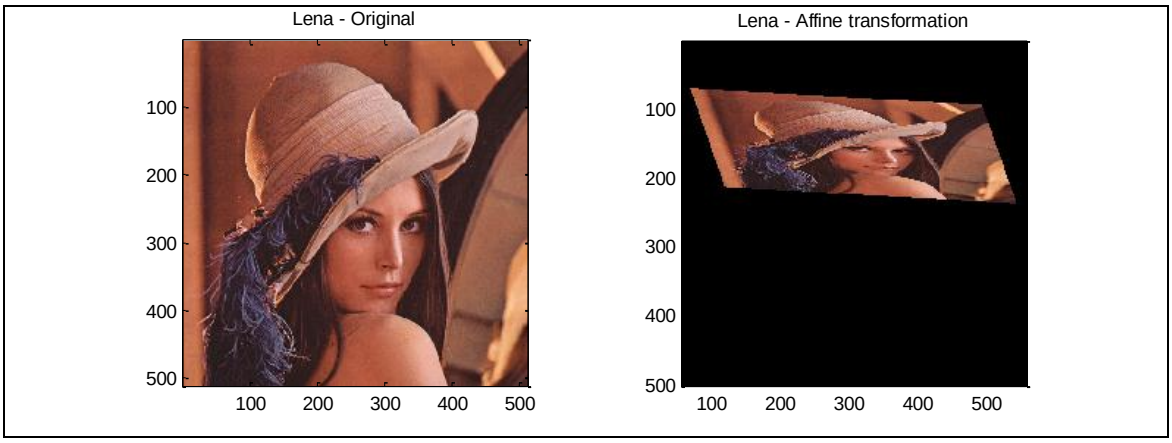


+ Similarity transformation example
  We will consider the parameters from before
  Please introduce the scaling factor (ex.: 0.5)





+ Affine transformation example

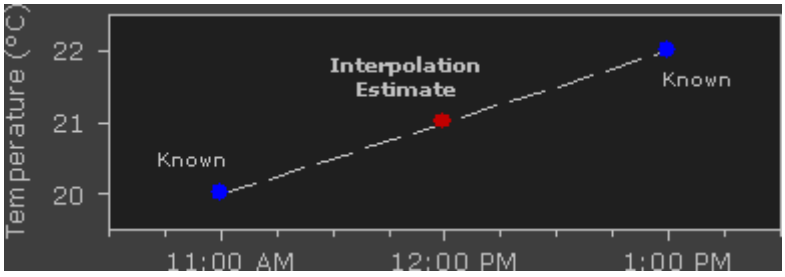


We must make two remarks; the first of all, is that as we can notice, Matlab and other software's typically establish **image origin of coordinates** in the top left corner.

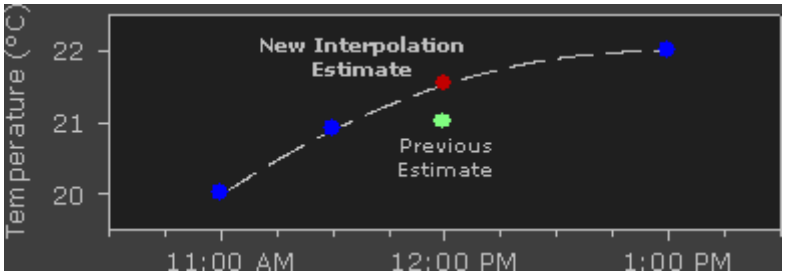
The second one, is that due to rotation, we can notice that not every point in the output image is fitted with a point of the original image. We can solve this using **interpolation** [19]:

1.3.4 Interpolation Data

Interpolation works by using known data to estimate values at unknown points. For example: If we wanted to know the temperature at noon, but only measured it at 11am and 1pm, we could estimate its value by performing a linear interpolation:



If we had now an additional measurement at 11:30am, we could notice that the bulk of the temperature rise occurred before noon. We could use in this additional data point to perform a quadratic interpolation:



i) Image resize example

Unlike air temperature fluctuations and the ideal gradient above, pixel values can change far more abruptly from one location to the next. As with the temperature example, the more information we have about the surrounding pixels, the better the interpolation it will become. So therefore, final zoomed image will deteriorate the more we stretch it. Interpolation can never add detail which is not already present in the original image.

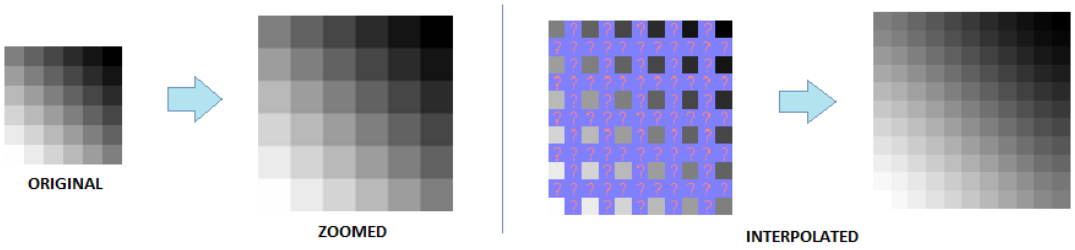


Figure 33: Image interpolation. *José Manuel Glez. De Rueda*

ii) Forward warping versus Inverse warping

In **forward warping** we will send each pixel P(x,y) to its corresponding location P'(x',y') with the projection function H(x,y). If the pixel lands in between two pixels we will add contribution to several pixels and normalize latter (Splatting).

But we have another option, called inverse (**back-warping**) that will give us better results. Instead of projecting each pixel to the final image, we will find for each pixel of the final image, the correspondence with the original pixels, interpolated in the source image. We can use for this interpolation methods as the nearest neighbor, bilinear filter, bicubic, or sinc / FINC. [20] We can implement this step easily in Matlab for improving our results using for example ***interp2*** function on source image.

1.4 BASIC STATISTICS

1.4.1 Arithmetic mean (AM) [21]

The ***arithmetic mean*** is the "standard" average, often simply called the "mean":

$$\bar{x} = \frac{1}{n} \cdot \sum_{i=1}^{n} x_i$$

The mean is the arithmetic average of set of values. It is usually defined as: $E[X] = \mu$.

1.4.2 Median

The ***median*** is described as the numerical value separating the higher half of a sample, a population, or a probability distribution, from the lower half.

1.4.3 Weighted arithmetic mean

The ***weighted arithmetic mean*** is used when we want to combine average values from samples of the same population with different sample sizes:

$$\bar{x} = \frac{\sum_{i=1}^{n} w_i \cdot x_i}{\sum_{i=1}^{n} w_i}$$

1.4.4 Mode

The ***mode*** is the most repeated value in a set.

1.4.5 Examples

For example, in the let's consider the ***series*** $\{\mathbf{1}, \mathbf{2}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{7}, \mathbf{9}\}$:

- The ***arithmetic mean*** is $\frac{1}{7}(1 + 2 + 2 + 3 + 4 + 7 + 9) = 4$.
- The ***median*** is: *3*
- The ***mode*** is $\{1, \mathbf{2}, \mathbf{2}, 3,4,7,9\}$: *2*

1.4.6 Normal distribution

The ***normal distribution*** is also called ***Gaussian distribution*** is a continuous probability distribution that has a bell-shaped probability density function. Its function is:

$$f(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Figure 34: Rose bushes are one example of normal distributions according to the number of flowers in a single plant.

Where $\mu$is the mean or expectation and σ^2 is the ***variance***. The normal distribution is considered the most prominent probability distribution in statistics. The ***central limit theorem*** states that under mild conditions, the sum of a large number of random variables is distributed approximately normally.

1.4.7 Standard deviation (STD)

σ is known as the ***standard deviation***. If we denote the average or expected value of x as $E[X] = \mu$, the STD is:

$$\sigma = \sqrt{E[(X - \mu)^2]}$$

The variance is the square value of the root, in consequence:

$$var = E[(X - \mu)^2]$$

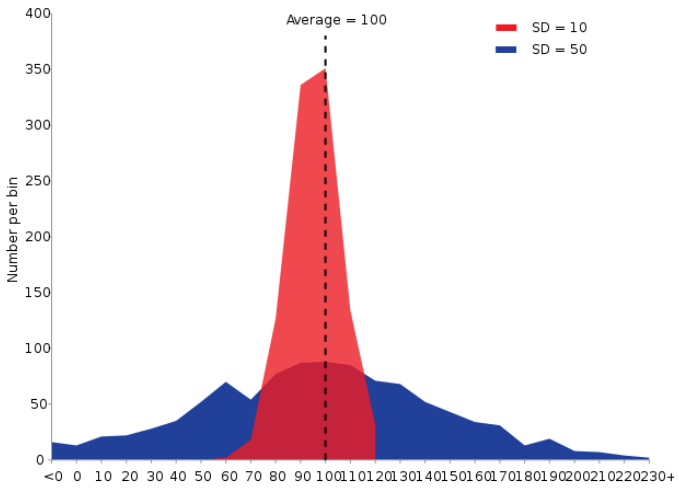


Figure 35: Example of two distributions (in red and blue) that have the same mean, but different STD values.

We can also rewrite the STD as:

$$STD = \sqrt{E\left[\left(X - E(X)\right)^2\right]} = \sqrt{E[X^2] - (E[x])^2}$$

1.4.8 Cumulative Distribution Function

For every real number x, the ***cumulative distribution function*** of a real-valued random variable X is given by:

$$F_X(x) = P(X \leq x)$$

Where the right-hand side represents the probability that the random variable X takes on a value less than or equal to x.

The ***probability*** that X lies in the interval $(a, b]$ (semi-closed interval) where $a < b$, is therefore $P(a < X \leq b) = F_X(b) - F_X(a)$

Some properties of the CDF are:

$$\lim_{x \to -\infty} F(x) = 0 \text{ and } \lim_{x \to +\infty} F(x) = 1$$

Every function with these four properties is a CDF. If X is a purely discrete random variable:

$$F(x) = P(X \leq x) = \sum_{x_i \leq x} P(X = x_i)$$

1.4.9 Bayes Theorem

Conditional probability: $\mathrm{P(A|B)}$ equals the probability of occurring the event A, known B.

Bayes theorem [22]***:***

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

$$where\ P(A) is\ called\ the\ prior, and\ is\ the\ initial\ degree\ of\ belief\ in\ A$$

$$P(A|B), the\ posterior, is\ the\ degree\ of\ belief\ having\ accounted\ B$$

$$\frac{P(B|A)}{P(B)} represents\ the\ support\ B\ provides\ to\ A$$

1.4.10 Markov process

A stochastic process has the ***Markov property*** if the *conditional probability distribution of future states of the process depends* ***only*** *upon the present state*. A process with that property is called a ***Markov process***.

1.4.11 Particle filter method [23]

The objective of a particle filter is to *estimate the sequence of hidden parameters,* $x_k\ for\ k = 0,1,2,3 \ldots etc$ ***based only on the observed data*** $y_k\ for\ k = 0,1,2,3 \ldots$etc.

Particle methods assume x_k and the observations y_k can be ***modeled*** in this form:

- $x_0, x_1, \ldots is\ a\ first\ order\ Markov\ process\ such\ that\ x_k|x_{k-1} \sim p_{x_k|x_{k-1}}(x|x_{k-1})$, and with an initial distribution $p(x_0)$.
- The observations y_0, y_1 … are conditionally independent provided that x_0, x_1, etc, are known. This means that each $y_k$only depends on x_k $(y_k|x_k \sim p_{(y|x)}(y|x_k)$.

As example is the system:

$$x_k = g(x_{k-1}) + w_k$$

$$y_k = h(x_k) + vk$$

Where both w_k and v_k are mutually independent and identically distributed sequences with known pdf. If $g(\cdot) and\ f(\cdot)$ are linear and if both $w_k and\ v_k$ are Gaussian, the Kalman filter finds the same filtering distribution. If not, a first-approximation can be done with the EKF, or even a UKF.

Particle filters are high accurate with enough particles.

1.5 GLOBAL, IMAGE AND CAMERA COORDINATES

It is very important to understand the difference between global, camera and image coordinates. Imagine that our object is the point in blue (Marker coordinates) and let´s say that

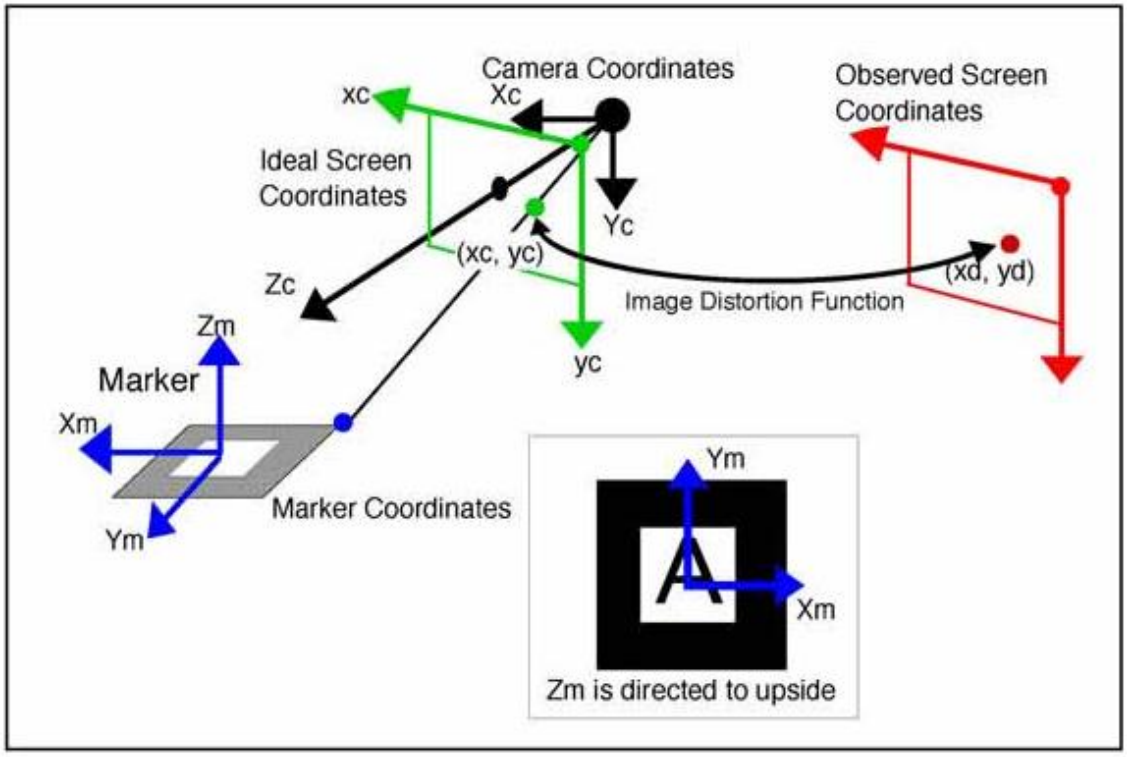


Figure 37: Schematic of global, camera and image coordinates. [67]

its value in GLOBAL COORDINATES is $Pos_{GLOBAL\ COORDINATES} = (X, Y, Z)$. If the relation between global and camera coordinates is an Euclidean transformation (rotation plus translation), we could define $Pos_{CAMERA} = T \cdot Pos_{GLOBAL\ COORDINATES}$ where T is the

Figure 36: Example of radius distortion in a camera, and its posteriors rectification

Euclidean transformation. But the reality is that CAMERA COORDINATES differ from IMAGE COORDINATES due to various factors:

- *The size of CCD sensors are not perfect squared, it means that an square perspective of size* (a, b) *is* (ka, kb)
- *The camera center is not the same as the image center*
- *The distortion factor skew is not the same in the inner center than in the external radius of the image*

It is mandatory to calibrate cameras for obtaining better results. Leading this to the general formula of getting a Pixel coordinate (x_i) from a 3D point coordinate (X_i):

$$x_i = PX_i$$

Where $P = K[R|t]$*, K is the calibration matrix, R is a rotation, and t a translation.*

We will use this formula always in CV, and we will introduce vectors $x_i$and $X_i$in homogeneous coordinates (explained in section 1.3) in the formula to impose matrixes dimensions match.

SECTION 2: SLAM

2.1 SLAM OVER THE HISTORY

As we have defined before, ***SLAM*** can be considered as the bundle of techniques that try to answer these *two questions*:

- *Where am I?*
- *How is the world around me?*

The main objective of robots performing SLAM is ***to build a map of their environment***, which is supposed to be unknown and *at the same time*, use this map to ***compute their own location***. It was at the IEEE Robotics and Automation Conference held in San Francisco in 1986 [24] when the probabilistic SLAM started to be stated as a problem. It was at that time when *probabilistic methods* emerged as new solutions to robotics. Until then, there was a misuse of them.

Chessman and others [25] [26] stated some basic concepts such as describing *relationships between landmarks* and *the geometry uncertainty problem*. Two main approaches were taken: the first one which said that there is a ***high degree of correlation between the estimates of landmarks,*** and that these correlations will grow. In parallel, works based on a ***Kalman Filter approach*** computed tests with sonar-based navigation of mobile robots.

Both approaches had much in common. [27] This paper showed that as a mobile robot moves through an unknown environment taking relative observations of landmarks, and the estimates of these landmarks are all ***necessarily correlated*** which each other because of the common error in estimated vehicle location.

The following step was to build a ***joint-state*** composed by two elements:

- *The vehicle pose*
- *All landmark positions*

The main problem of this joint-state it is *computational cost* ($\sim n^2, where\ n\ is\ the\ number\ of\ landmarks$), really high, and growing to the power of two each time we add a new landmark.

In that time, researchers did not consider the ***convergency*** of that problem, and though that map errors had random behavior with *unbound error growth*.

So they focused on making ***approximations*** of the consisted map problem assuming, even forced the *correlations between landmarks to be minimized or eliminated*. Their objective was to reduce the *FULL FILTER* into **decoupled** *landmarks vehicle filters*. In that time, mapping problem was totally ***differentiated*** from localization problem.

Sometime after, researchers discovered that the mapping plus localization problem was ***convergent***, and it was completely the opposite of what they thought until then the key solution to the problem:

More correlations between landmarks = better results

SLAM acronym an a bundle of first examples were presented in the 1995 International Symposium on Robotics Research [28]. At that time, work was focused on improving computational efficiency and addressing issues in data association or 'loop closure'.

Until now, many conferences where partially or totally (As the SLAM summer courses) dedicated to studying SLAM techniques. We will explain in the following parts the most relevant techniques applied to solve this problem

2.2 PROBABILISTIC SLAM

2.2.1 PRELIMINARIES

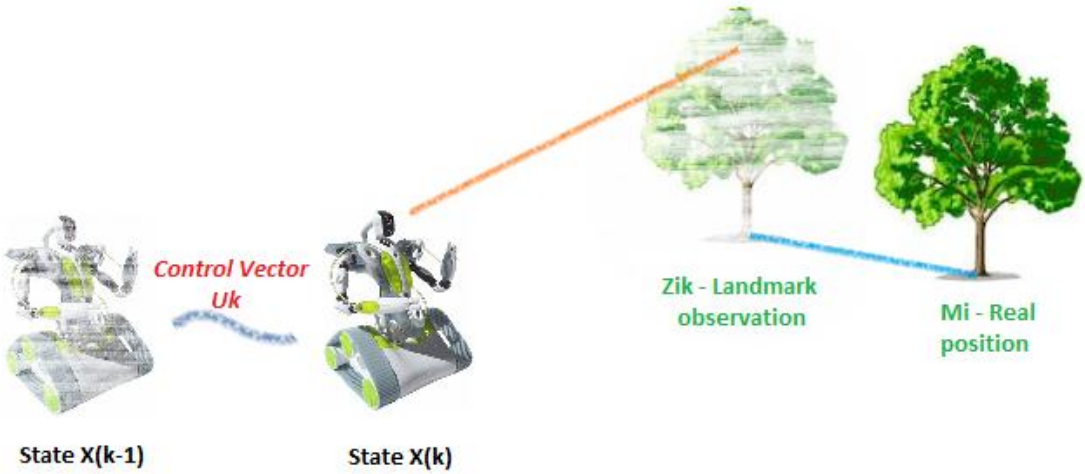


Figure 38: Essential SLAM problem. *José Manuel Glez. De Rueda Ramos.*

Imagine that we divide time in samples and we design it as ***time k.***

Consider a robot moving through a unknown environment taking ***relative observations of landmarks*** z_{ik}. These **landmarks** have a true ***absolute position*** m_i which is unknown but does not change over time, is invariant. We can describe robot location and orientation with the

state vector x_k. And finally we can describe the ***actions u_k,*** as the actions applied from last step in time *(at time k-1),* to drive the robot to the state x_k *(at time k).*

In addition, we can define sets of these data as: $X_{0:k}, U_{0:k}, m, Z_{0:k}$. Where the *subindex* represents from which state until which state is included in the set.

2.2.2 PROBLEM STATEMENT

As we explained in *Section1*, $P(A|B)$ means: *which is the probability that A occurs, known B.* With that approach, we can consider:

$$P(x_k, m|Z_{0:k}, U_{0:k}, x_0)$$

as the unveiling the ***actual position of the robot*** and the ***set of all absolute positions of all landmarks***, knowing the *relative position of the landmarks* (like the "fogged tree" in the drawing), all *the control vectors,* and the *initial state of the robot* (we can reference the origin of our map as the initial state).

A recursive approach is recommended. Let's consider an estimate of the distribution at time $\mathrm{k}-1$:

$$P(x_{k-1}, m|Z_{0:k-1}, U_{0:k-1})$$

The joint posterior will be computed applying Bayes Theorem[7], considering that we will receive these two informations: u_k because we know the control we are applying to the robot, and the new relative observations z_k.

The ***observation model*** describes the probability of making an observation z_k when the vehicle location and landmark locations are known: $P(z_k|x_k, m)$.

For the ***motion model*** it will be assumed that the process is a Markov process, which means that next state depends only on the previous state, and de applied control u_k: $P(x_k|x_{k-1}, u_k)$

The **probabilistic SLAM algorithm** is now implemented in two steps recursively: A ***prediction of the future step***, and a ***correction on those measurements***:

STEP 1: TIME-UPDATE (PREDICTION)

$$\mathrm{P}(\mathrm{x_k}, \mathrm{m}|\mathrm{Z_{0:k-1}}, \mathrm{U_{0:k}}, \mathrm{x_0}) = \int \mathrm{P}(\mathrm{x_k}|\mathrm{x_{k-1}}, \mathrm{u_k})\mathrm{P}(\mathrm{x_{k-1}}, \mathrm{m}|\mathrm{Z_{0:k}}, \mathrm{U_{0:k-1}}, \mathrm{x_0})\mathrm{dx_{k-1}}$$

STEP 2: MEASUREMENT-UPDATE (CORRECTION)

$$P(x_k, m|Z_{0:k}, U_{0:k}, x_0) = \frac{P(z_k|x_k, m)P(x_k, m\,|Z_{0:k-1}, U_{0:k}, x_0)}{P(z_k|Z_{0:k-1}, U_{0:k})}$$

[7] Bayes Theorem is also explained in Section 1

If we want two independent problems, the ***map building problem*** can be formulating as computing the conditional density $P(m|X_{0:k}, Z_{0:k}, U_{0:k})$ assuming that the location of the vehicle is known. By the other hand, assuming that the locations of the landmarks are known with certainty, this leads into $P(x_k|Z_{0:k}, U_{0:k}, m)$ that is the ***location problem***.

For next parts, we will consider $\boldsymbol{P(x_k, m|Z_{0:k}, U_{0:k}, x_0)}$ **as** $\boldsymbol{P(x_k, m|z_k) or\ even\ P(x_k, m)}$ to simplify writing and understanding the concept, but it is exactly the same conditional density.

The observation model $P(x_k, m|z_k)$ makes explicit the ***dependence of observations*** on both the *vehicle* and *landmark locations*. So we cannot partition the joint posterior as: $P(x_k, m|z_k) \neq P(x_k|z_k)P(m|z_k)$, and it was known that this lead into inconsistent maps [29].

But if we consider again the drawing before, if we had more trees, ***the error between estimated and true landmark locations would be common***, because is in fact due to the same source (for example, a 3D laser scanner to measure the distance to the tree). This leads into a very important concept, and it is that the relative position between landmarks $m_i - m_j$ could be known with high accuracy, even if the absolute location m_i is unknown / uncertain.

Mathematically, this can be translated into that the joint probability density for the pair of landmarks $P(m_i, m_j)$ is highly peaked even when the marginal densities $P(m_i)$ may be dispersed.

And to finalize with this statement problem, the most important concept insight was that the ***correlations between landmark estimates increase monotonically as*** more and more observations are made.

2.2.3 SOLUTIONS

The SLAM problem can be solved if we know the observation model and the motion model, as then it will be computing the TIME-UPDATE and the MEASUREMENT-UPDATE.

If we consider the model as a state-space model with *additive Gaussian noise* (every sensor can be considered as having Gaussian noise in a threshold), we can use ***EKF*** to solve this problem.

Another option is to describe vehicle motion as a set of samples of a *more generic non-Gaussian probability distribution*, for which we will use a ***Rao-Blackwellised particle filter***.

i) EKF-SLAM

Our objective is to obtain the motions and observation model. For this, in with the Extended Kalman Filter (EKF) we describe them as:

VEHICLE MOTIONS MODEL: $\boldsymbol{P(x_k|x_{k-1}, u_k) \leftrightarrow x_k = f(x_{k-1}, u_k) + w_k}$

$$where\ f(\cdot)\ is\ the\ kinematics\ model\ and$$

$$w_k are\ additive, zero\ mean\ uncorrelated\ Gaussian\ motion\ disturbances\ with\ covariance\ Q_k$$

OBSERVATION MODEL: $\boldsymbol{P(z_k|x_k, m) \leftrightarrow z(k) = h(x_k, m) + v_k}$

$$where\ h(\cdot)\ describes\ the\ geometry\ of\ the\ observation\ and$$

$$\boldsymbol{v_k}\ are\ additive, zero\ mean\ uncorrelated$$

$$Gaussian\ observation\ errors\ with\ covariance\ R_k$$

The standard EKF method can be applied to compute the ***mean***:

$$\begin{bmatrix} \widehat{x_{(k|k)}} \\ \widehat{m_k} \end{bmatrix} = E\begin{bmatrix} x_k \\ m \end{bmatrix} | Z_{0:k}\Big]$$

And ***covariance:*** $P_{(k|k)} = \begin{bmatrix} P_{xx} & P_{xm} \\ P_{xm}^T & P_{mm} \end{bmatrix}_{(k|k)} = E\left[\begin{pmatrix} x_k - \widehat{x_k} \\ m - \widehat{m_k} \end{pmatrix}\begin{pmatrix} x_k - \widehat{x_k} \\ m - \widehat{m_k} \end{pmatrix}^T \middle| Z_{0:k}\right]$

Of the joint posterior distribution $P(x_k, m|Z_{0:k}, U_{0:k}, x_0)$.

This joint posterior distribution comes from:

TIME-UPDATE

$$\widehat{x_{(k|k-1)}} = f(\widehat{x_{k-1|k-1}}, u_k)$$

$$P_{(xx,k|k-1)} = \nabla f P_{(xx,k-1|k-1)} \nabla f^T + Q_k$$

Where $\nabla f\ is\ the\ Jacobian\ of\ f\ evaluated\ at\ the\ estimate\ \widehat{x_{(k-1|k-1)}}$.

OBSERVATION-UPDATE

$$\begin{bmatrix} \widehat{x_{(k|k-1)}} \\ \widehat{m_k} \end{bmatrix} = \begin{bmatrix} \widehat{x_{(k|k-1)}} \\ \widehat{m_{k-1}} \end{bmatrix} + W_k[z(k) - h(\hat{x}_{(k|k-1)}, \hat{m}_{k-1})]$$

$$P_{(k|k)} = P_{(k|k-1)} - W_k S_k W_k^T$$

$$\text{Where } S_k = \nabla h P_{(k|k-1)} \nabla h^T + R_k$$

$$W_k = P_{(k|k-1)} \nabla h^T S_k^{-1}$$

And where $\nabla h\ is\ the\ Jacobian\ of\ h\ evaluated\ at\ \hat{x}_{(k|k-1)} and\ \hat{m}_{k-1}$

This EKF solution has these ***key issues***:

1.- The solution is ***convergent*** in the EKF problem properties. We can see an example of landmarks in figure from the right.

2.- ***Computational Effort***: The computational cost grows quadratically. It is important to consider this issue.

3.- ***Data Association:*** The standard formulation of the EKF is really fragile to incorrect associations of landmarks.

4.- ***Non-linearity:*** The EKF uses linearized

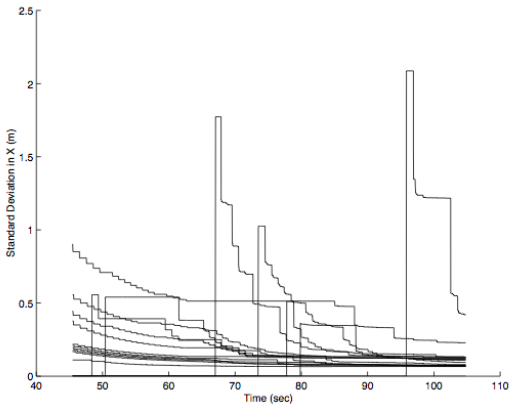


Figure 39: Landmark convergency over time

models of non-linear motions and observations models. Sometimes this leads to inconsistency in solutions.

A ***resume of the algorithm*** is presented in next figure:

Algorithm 1 SLAM:

$\mathbf{x}_0^B = \mathbf{0}$; $\mathbf{P}_0^B = \mathbf{0}$ {*Map initialization*}

$[\mathbf{z}_0, \mathbf{R}_0]$ = get_measurements

$[\mathbf{x}_0^B, \mathbf{P}_0^B]$ = add_new_features$(\mathbf{x}_0^B, \mathbf{P}_0^B, \mathbf{z}_0, \mathbf{R}_0)$

for $k = 1$ to steps **do**

$[\mathbf{x}_{R_k}^{R_{k-1}}, \mathbf{Q}_k]$ = get_odometry

$[\mathbf{x}_{k|k-1}^B, \mathbf{P}_{k|k-1}^B]$ = EKF_prediction$(\mathbf{x}_{k-1}^B, \mathbf{P}_{k-1}^B, \mathbf{x}_{R_k}^{R_{k-1}}, \mathbf{Q}_k)$

$[\mathbf{z}_k, \mathbf{R}_k]$ = get_measurements

$\mathcal{H}_k$ = data_association$(\mathbf{x}_{k|k-1}^B, \mathbf{P}_{k|k-1}^B, \mathbf{z}_k, \mathbf{R}_k)$

$[\mathbf{x}_k^B, \mathbf{P}_k^B]$ = EKF_update$(\mathbf{x}_{k|k-1}^B, \mathbf{P}_{k|k-1}^B, \mathbf{z}_k, \mathbf{R}_k, \mathcal{H}_k)$

$[\mathbf{x}_k^B, \mathbf{P}_k^B]$ = add_new_features$(\mathbf{x}_k^B, \mathbf{P}_k^B, \mathbf{z}_k, \mathbf{R}_k, \mathcal{H}_k)$

end for

Figure 40: EKF-SLAM

ii) PARTICLE FILTER (PF) SLAM

Fast-SLAM is a SLAM which is based on ***particle filtering***. Until its introduction in 2002 [30], the work was centered on performing more efficient EKF still retaining its linear Gaussian assumptions. Fast-SLAM was the first to represent the non-linear process model and non-Gaussian pose distribution, inspired by previous works [31] [32].

They key point of how to apply this algorithm was reducing the huge sample-space applying ***Rao-Blackwellisation***, whereby a ***joint state is partitioned*** as:

$$P(x_1, x_2) = P(x_2|x_1)P(x_1)$$

If $P(x_2|x_1)$can be represented analytically, only $P(x_1)\sim x_1^i$ needs to be sampled.

The SLAM joint state can be factored into two components:

$$P(X_{0:k}, m|Z_{0:k}, U_{0:k}, x_0) = P(m|X_{0:k}, Z_{0:k})P(X_{0:k}|Z_{0:k}, U_{0:k}, x_0)$$

$$as\ described\ before: P(a, b) = P(b|a)P(a)$$

And now the probability is on the ***trayectory*** $\boldsymbol{X_{0:k}}$instead of on the single pose x_k.***When conditioning on the trajectory, landmarks become independent.***

The difference with previous method is that the map is represented as a set of independent Gaussians (linear complexity), rather than a joint map covariance with quadratic complexity.

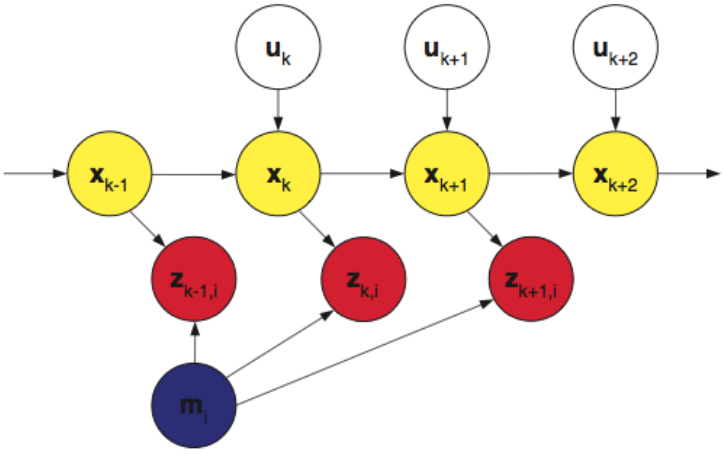


Figure 41: Fast-SLAM "pose states" conceptual idea

So the structure of this algorithm is a state where the trajectory is represented as ***weighted samples***, the joint distribution is represented by:
$\{w_k^i, X_{0:k}^i, P(m|X_{0:k}^i, Z_{0:k}) = \prod_j^M P(m_j|X_{0:k}^i, Z_{0:k}\}_i^N$ (1) where the map for each particle is composed of independent Gaussians.

*The **updating** process of the map, is exactly the same as in the EKF applied individually to each observed landmark (unobserved landmarks do not change).*

Then the ***propagation*** becomes. The methodology is inspired in sequential important sampling (SIS) where it "telescopes" the joint ***recursively***:

$$P(x_0, x_1, \dots, x_T|Z_{0:T}) = P(x_0|Z_{0:T}) \cdot P(x_1|x_0, Z_{0:T}) \cdot \dots \cdot P(x_T|Z_{0:T-1}, Z_{0:T})$$

This is an approximation that only works when the system "exponentially forget" their past, i.e., those systems whose process noise cause at time k are growing their independency respect to previous states).

So the algorithm becomes:

- At time $k-1$ we assume that the joint state is represented as (1) (with k-1, instead of k).
- For each particle, we compute a proposal distribution: $x_k^i \sim \prod(x_k|\, X_{0:k}^i, Z_{0:k}, u_k)$ we weight samples according to the importance function:
- $w_k^i = w_{k-1}^i \frac{P\left(z_k \middle| X_{0:k}^i, Z_{0:k-1}\right) P(x_k^i|x_{k-1}^i, u_k)}{\prod(x_k^i|X_{0:k}^i, Z_{0:k}, u_k)}$
  which has the observation model and the motion model expressed on the numerator.
- Resampling if needed[8], selecting particles with probability of selection proportional to theirs w_k.
- Just to end, the difference between Fast-SLAM 1.0 and 2.0 is that the second one includes in the proposal distribution, the influence of the current observation. The advantage, is that this algorithm becomes locally optimal.

[8] Some people do this on every step, some people uses a threshold for the weight variance.

2.2 STATE OF THE ART VISUAL-SLAM METHODS

Many variations of the original solutions to the SLAM problem have been developed, for example the use *neural networks* for closing the loop (when we get back into an initial already "recorded" position), *Voronoi-based systems* as they study in University Carlos III de Madrid, and many more.

But I would like to remark some **Visual SLAM** approaches that have been a huge step towards a general Visual SLAM algorithm.

2.2.1 MonoSLAM

This SLAM solution is performed using a ***single RGB camera.*** For full revision of this algorithm I recommend reading the paper [33] or its explanation in the IEEE Transactions on pattern analysis and machine intelligence [34]. A general, *non-mathematical description* will be given to understand the basis of this algorithm:

- We will ***initialize*** the algorithm with the position of some detected features, and a high uncertainty of its distance.
- For describing the system, ***quaternions*** are used combined with the typical ***rigid body equations*** for describing the camera pose. It is used the ***inverse of the distance d.***

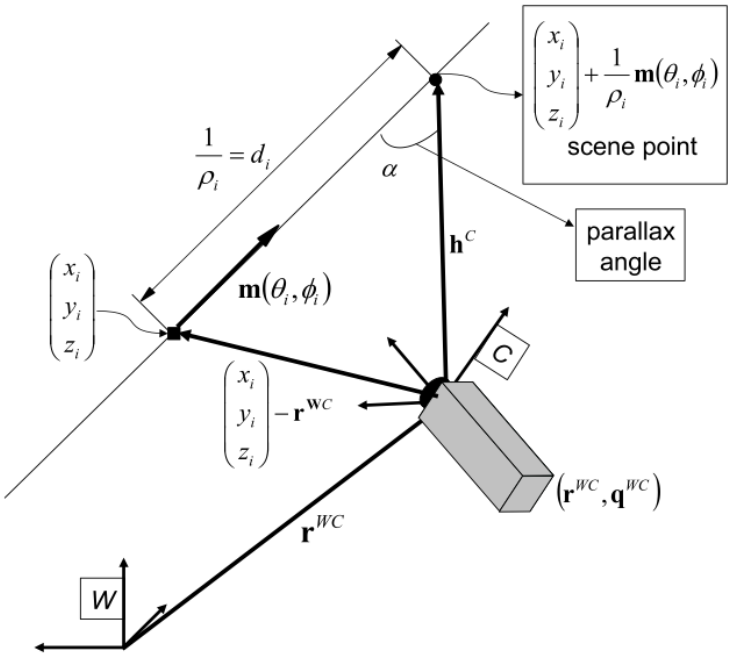


Figure 42: Problem notation

- It is important, and demonstrated on the paper, the importance of the ***parallax***[9]. If it is high, the feature depth uncertainty will be reduced, which is good. They also explain what happens when the parallax is low, as described in the next figure:

[9] The parallax is a displacement in the apparent position of a point in the space

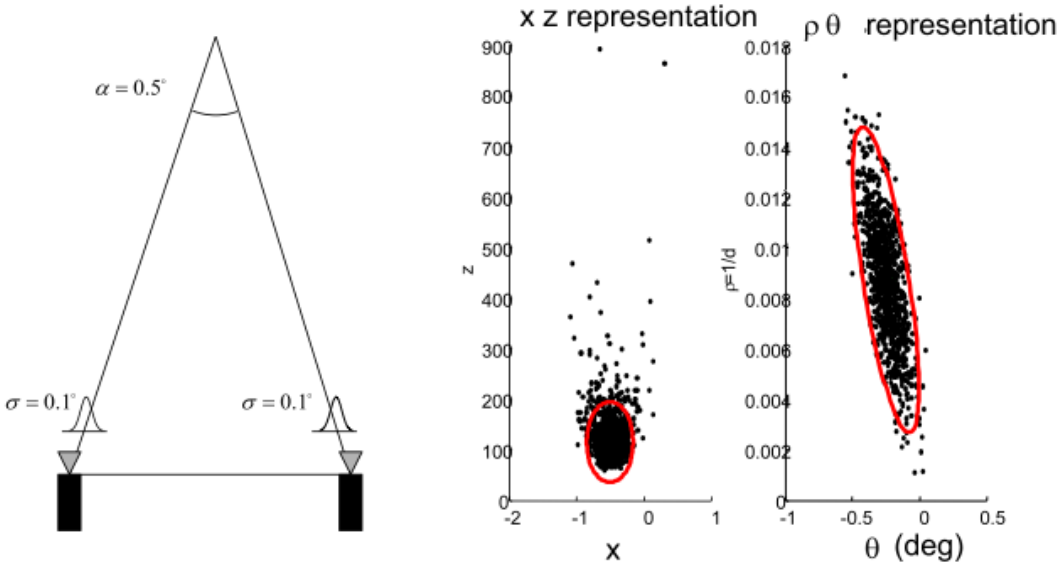


Figure 43: Low parallax means high uncertainty in the distance

They use a *prediction-update* with the ***standard EKF*** for predicting-updating the features. Detailed description about projection ray, angle derivation and covariance are given in the paper. Results can be seen in next figures:

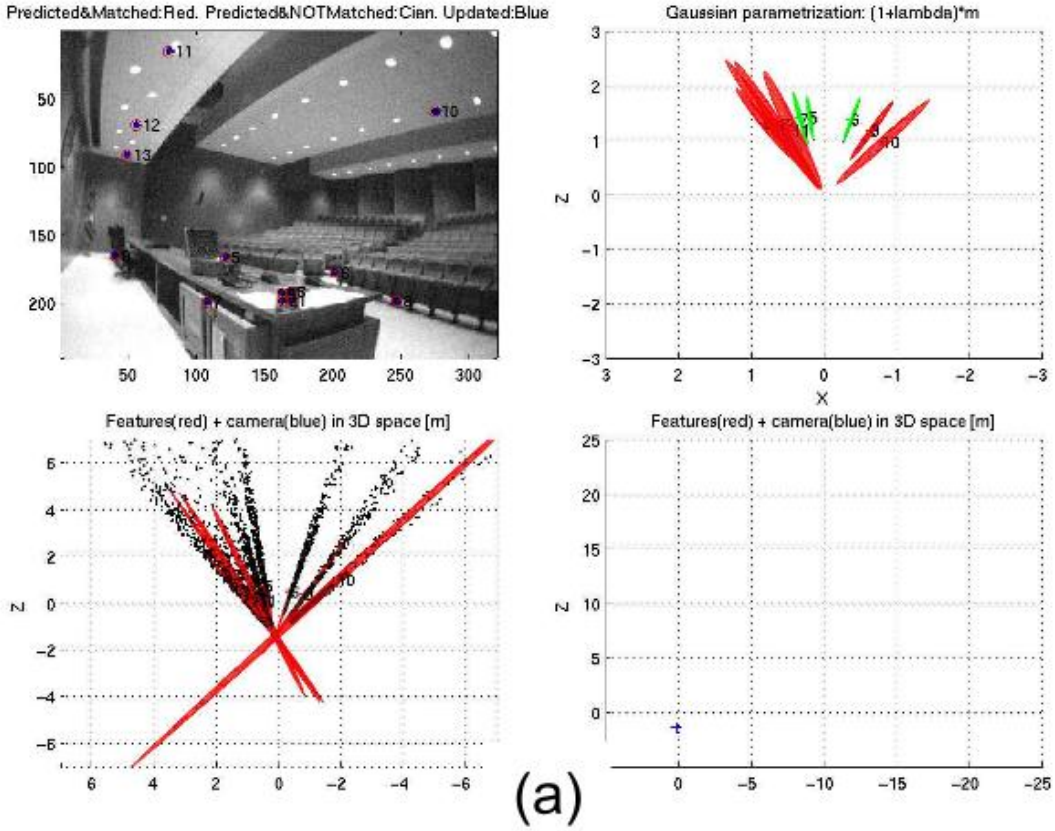


Figure 44: First steps of the demo

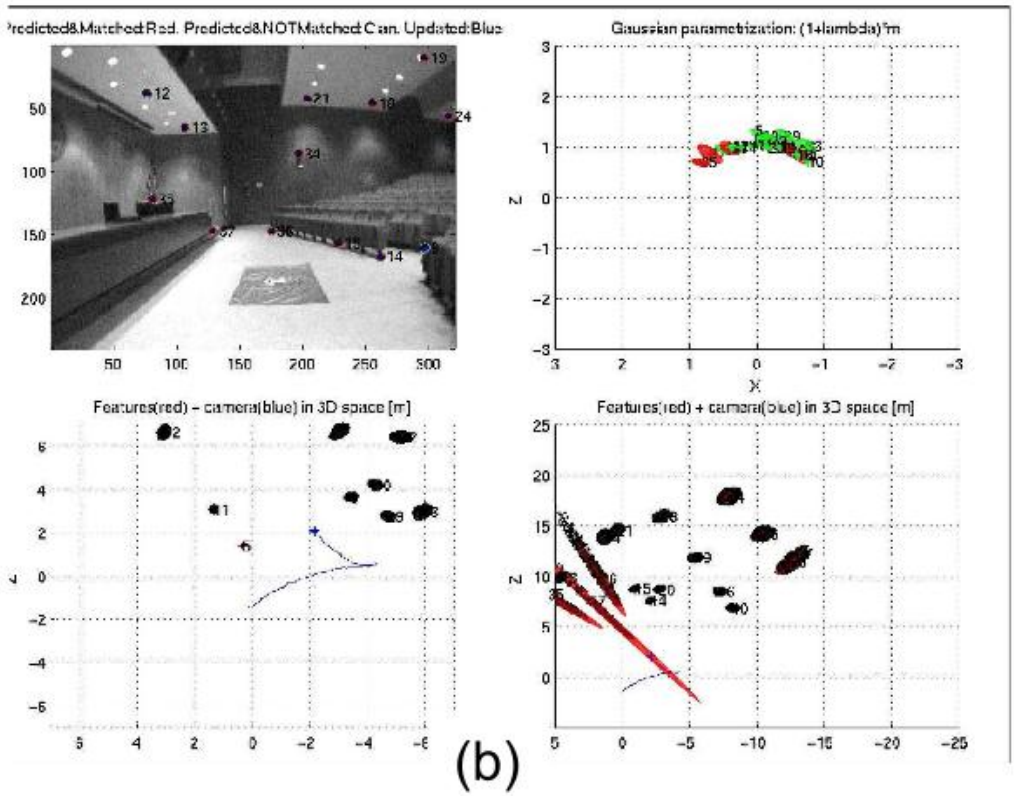


Figure 45: Last step of the demo, with the much more bounded features uncerntainty

2.2.2 PTAM

PTAM is the acronym for *Parallalel Tracking and Mapping for Small AR Workspaces* [35], presented in ISMAR 2007 and it can be summarized as:

- *Tracking* is separated in a ***different thread*** as *Mapping* (mandatory to have a dual core pc), so the algorithm becomes faster.
- ***Mapping is based on keyframes*** (they use bundle adjustment[10]).
- The map is initialized from a stereo pair with the ***5-point algorithm***.
- ***New points*** are initialized with epipolar search.
- ***Thousands of points*** are mapped

Results are incredible for that time, as next figure shows where there is a comparison between 3D trajectories between EKF-SLAM and PTAM:

[10] Bundle adjustment means refining the 3D points coordinates from a number of 3D points from different viewpoints

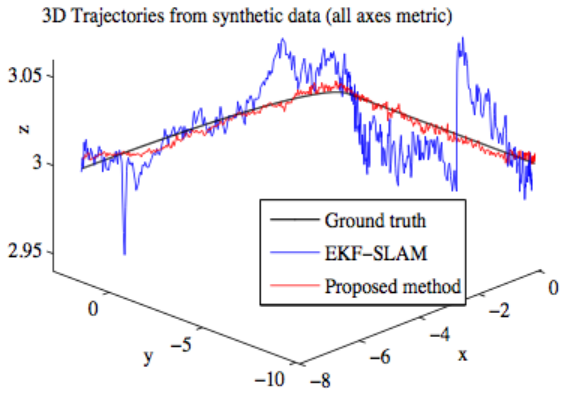


Figure 46: PTAM is clearly more effective that the typical EKF-SLAM approach

Just as a note, AR applications with virtual moving objects through the virtual reconstructed environment were presented in the conference.

2.2.3 DTAM

DTAM is the acronym for *Dense Tracking and Mapping in Real-Time* [36]. It was presented in the ICCV 2011 and won the best presentation paper. A single hand-held RGB camera is used to perform this SLAM, but this time is using ***dense models*** of the scene, a ***whole image alignment*** as the posterior KinectFusion presented this time using depth information. It uses complex energetic functions for the photometric error.

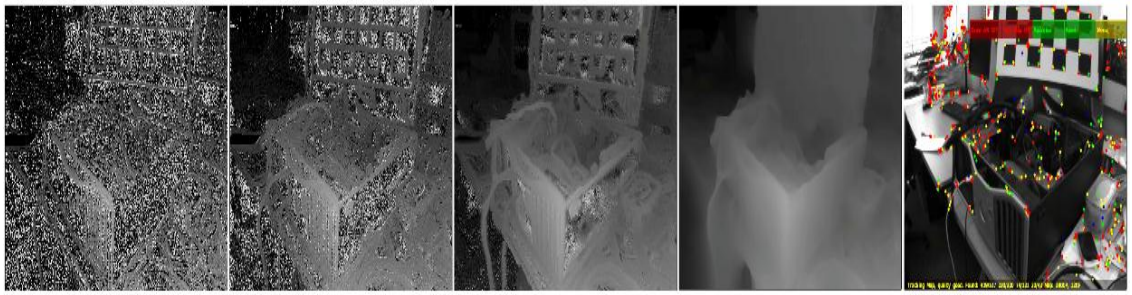


Figure 47: DTAM (First four images from the left) using full data as features (300.000 points) versus PTAM(Rigth) using around 1000 features

A video comparison between PTAM and DTAM on really difficult situations (camera shaking, defocusing, etc) is presented in the link [37].

2.3.4 HIGH SPEED VISUAL SLAM PROBLEM: BLURRING

When dealing with high speed movements, images captured from the camera can be blurred. A good reference of how to eliminate these blurs while doing SLAM was presented in the ICCV 11' as [38].

We can notice these effects in the next figure from the paper:

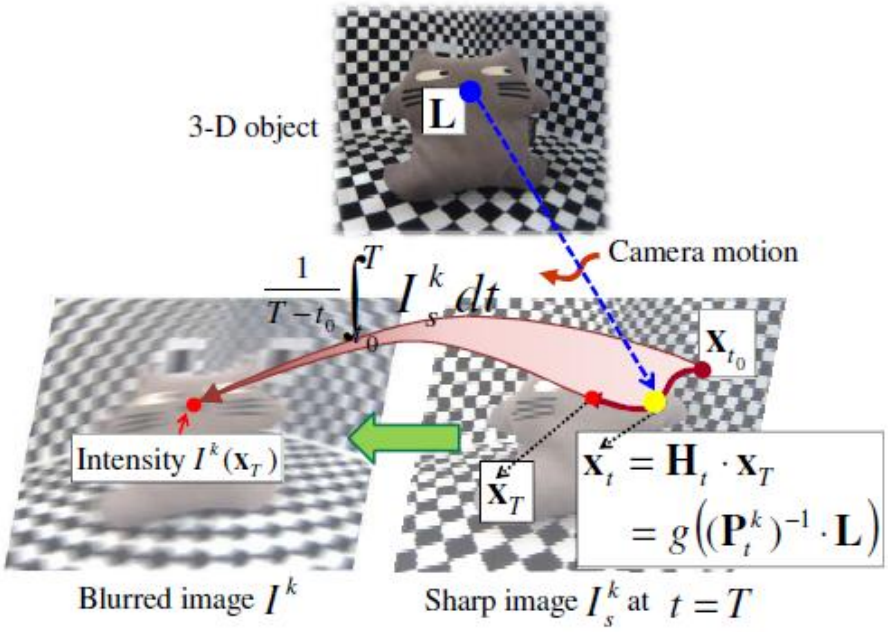


Figure 48: Example of blurred feature point

2.3 KINECTFUSION: THEORY

KinectFusion was presented in of the most prestigious conferences of this past year SIGGRAPH 11' [39] and won best vision award in ISMAR 11' [40]. Its main characteristics are the use of a single global implicit surface model, and a coarse-to-fine ICP algorithm using **all** observed data against the growing full surface model. It's main purpose is doing a fast and accurate vision SLAM for AR applications. It can be used in reversed engineering as a low cost handheld scanner. This geometry aware algorithm, introduces novel methods for segmenting physical objects, and can perform real-time interactions.

As described on previous section, we should deal with Kinect camera problems that are:

Figure 49: Examples of Kinect depth image problems

In example, we can notice ***triangulation occlusions*** in 1, ***border and random lack of measurements*** as seen on 2 and 3

2.3.1 NOVELTIES

Comparing with other previous methods, we can remark following novelties:

- ***Interactive rates:*** With desktop NVIDIA cards 30 fps are given. My tests with different implementation, but same concept, give us 4 fps with a laptop NVIDIA GTX 540m (only 96 cores)

- ***No explicit feature detection:*** Instead, we will compute a ICP with all new given data.
- High-quality reconstruction geometry: 1-3mm error 3D models are obtained instead of the 4cm (more than one order of magnitude) with a superresolution method
- ***Dynamic interaction is assumed:*** It is possible to draw lines, throw virtual masses, drive little cars on our model room, thanks to its speed
- ***Infrastructure-less:*** We do not need expensive 3D laser scanners / fully calibrated and complex systems. An inexpensive camera Kinect will be used for our tests.
- ***Room scale:*** Until now, this real-time SLAM methods with 10x less error than the camera error, could be only achieved with small volumes. With this novel algorithm, we can get a $9m^3$in seconds just moving around the Kinect camera.

2.3.2 THE ALGORITHM

i) STEP 1: DEPTH MAP CONVERSION

In the first step, our objective is to compute for each pixel $u(x, y)$ its vertex in global coordinates $v_i^g(u)$ and its normal $n_i^g(u)$

As we learned in previous section, raw depth data from Kinect camera, will have around 2cm STD error for our typical measurement distance (from 1 to 3m). In other SLAM techniques, we will use tracked features (known or learned features) to compute each new camera position over time.

But Kinect Fusion is different. Our final objective will be a super-resolution 3D model of the scene, that means that we will have much higher density in a slice of volume than with simple raw data. This will lead into a up-to-3mm STD error, instead of a 2cm error which means almost ten times more precision. And we will do all this, on real time thanks to parallel computing.[11]

1.1 Vertex validity mask

As we noticed some depth measurements are missing due to Kinect structured light occlusion. We define a vertex validity mask $M_k(u) \rightarrow 1$ for each pixel where a depth measurement transforms to a valid vertex. Otherwise, we will give to this mask a zero value $M_k(u) \rightarrow 0$.

1.2 Filtering for better results

To improve the quality of the normal maps produced, we will apply a bilateral filter to the raw data.

[11] Real-time means around 30 fps. We can reach this with NVIDIA GTX models. With the laptop NVIDIA 540M we get around 4 fps.

THEORY

GAUSSIAN FILTERING

Gaussian filter is a linear filter that is commonly used for blurring an image

$$GC[I]_p = \sum_{q \in S} G_\sigma(\|p - q\|) I_q$$

Where $G_\sigma(x)$ denotes de 2D Gaussian kernel $G_\sigma = \frac{1}{2\pi\sigma^2}\exp(-\frac{x^2}{2\sigma^2})$

We can see its results for different σ values in the next figure:

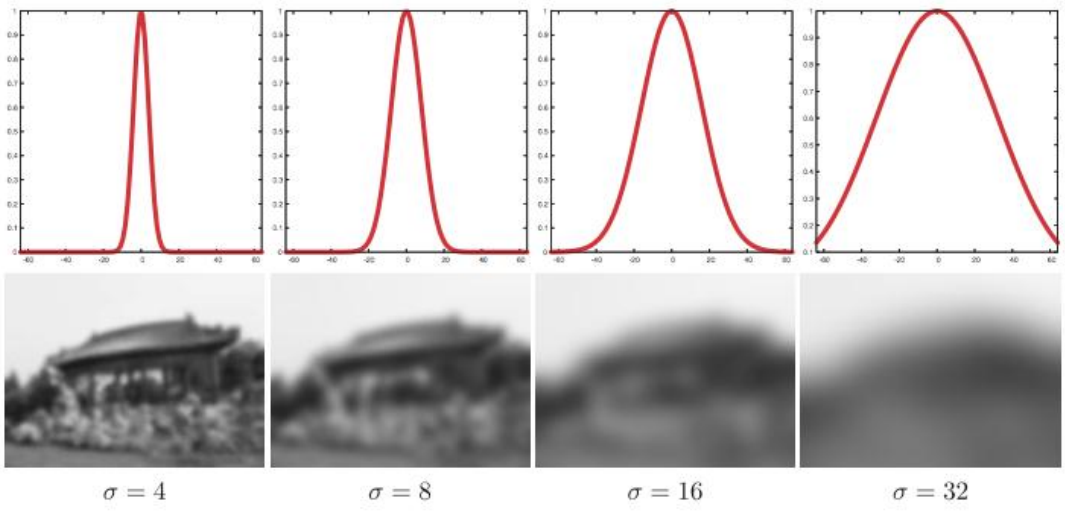


Figure 50: Gaussian blur for different values of σ [41]

THEORY

BILATERAL FILTER [12]

The ***bilateral filter*** [41] is a non-linear filter, that can blur an image when non-distorting edges. This preservation can give us a texture-less output image while preserving edges. It is widely used for de-noising applications.

[12] One of the best references I have found for the Bilateral Filter is [17]. It is very complete, so please read it if you need more information.

Figure 51: (left) Original Lena (right) BF Lena

As we can see on the right image, texture tends to smooth while preserving contours.

BF properties are:

- Each pixel is a weighted average of its neighbors. It is easy to implement.

- It depends on the size and the contrast of the features to preserve.

- It can be used in a non-iterative manner. This makes the parameters easy to set.

- It can be parallelized

The idea basically is that for influencing one pixel to another, it should not only be in a nearby location, but also have a similar value.

The formalization of this idea was presented in the literature, resulting in this formula [42]:

$$BF[I]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_S}(\|p - q\|) G_{\sigma_S}(|I_p - I_q|) I_q$$

Where W_p is a normalization factor that ensures pixel weights to sum 1:

$$W_p = \sum_{q \in S} G_{\sigma_S}(\|p - q\|) G_{\sigma_r}(|I_p - I_q|).$$

Parameters σ_S *and* σ_r will specify the amount of filtering for the image:

- *SMOOTHING FILTER*: As σ_r increases, the BF approximates gradually a Gaussian convolution, because the range Gaussian G_{σ_r} widens and flattens.

- *RANGE FILTER*: As σ_S increase, BF smooths larger features.

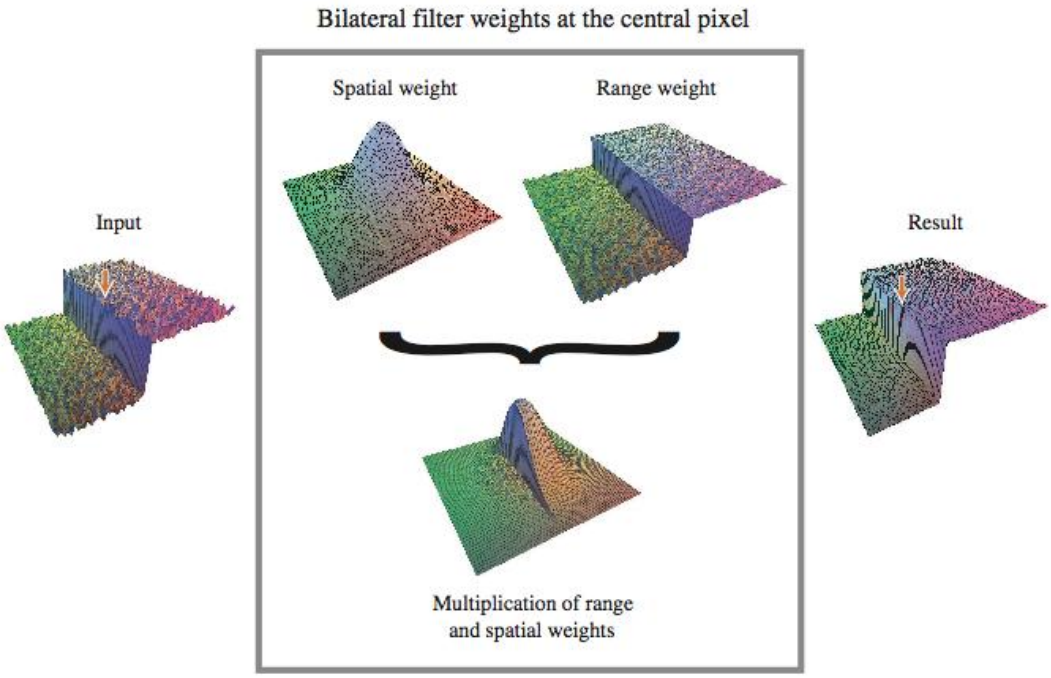


Figure 52: Visual demo of the output of a BF series of points. The height represents in this case, the intensity I. [43]

We can see different results for different values of the two parameters $\sigma_S \; and \; \sigma_r$ in the next figure:

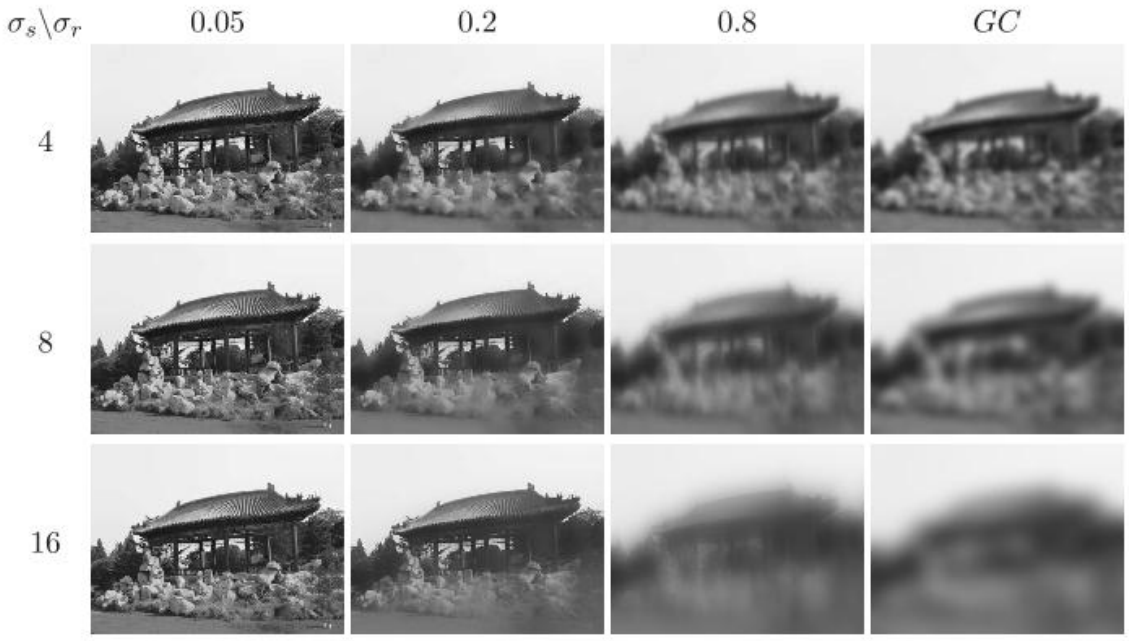


Figure 53: Different BF results only changing two parameters of the filter. Last column corresponds to a Gaussian Convolution. [41]

One of best properties of BF is de-noising the image for computing in a second step normal map. Another example of BF de-noising of a 3D model is presented in next figure:

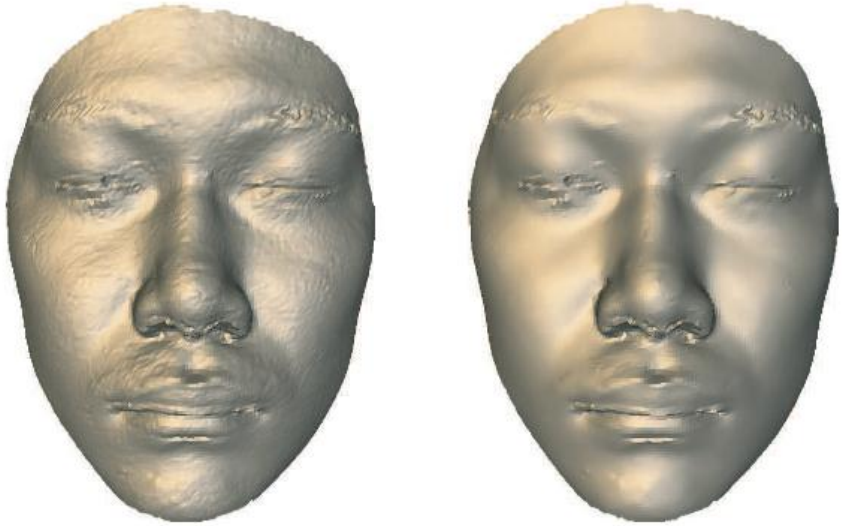


Figure 54: Face depth example after bilateral filtering [44]

A *Matlab example* written by me, and my colleague Peggy Lu is at the end in annexe 2. Part one is noise elimination comparison (some salt & pepper + Gaussian noise addition) between a median and Gaussian filter. And second part is bilateral filter example.

In our case we will use the BF for de-noising depth measures using a pyramidal approach. The concept is the same.

The bilateral filter we apply is:

$$D_k(u) = 1/W_p \sum_{q \in U} N_{\sigma_s}(\|u - q\|_2) N_{\sigma_r}(\|R_k(u) - R_k(q)\|_2) R_k(q)$$

where $N_\sigma(t) = \exp(-t^2 \sigma^{-2})$

and $Wp\ is\ a\ normalizing\ constant$

1.3 Vertex map V_i

Now that we have filtered raw data, we will transform this raw data to camera coordinates. As we know, if the intrinsic calibration matrix is K, for each pixel, its vertex will be:

$$v_i(u) = D_i(u) K^{-1}[u, 1]$$

The whole list of $v_i(u)$ is called vertex map Vi, where i is the time step.

1.4 Normals calculus

To obtain the normals, we will compute the cross product of the adjacent vectors to a point.

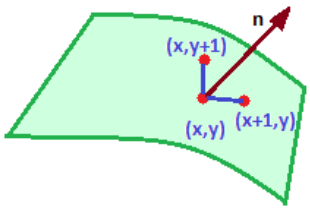


Figure 55: Normals calculation. *José Manuel Glez. De Rueda*

The formula we will apply is:

$$n_i(u) = (v_i(x+1,y) - v_i(x,y)) \times \big(v_i(x,y+1) - v_i(x,y)\big)$$

We will normalize dividing by its norm: $\|n_i(u)\|$

This will result in a single normal map N_i, computed in parallel.

1. 5 Global coordinates transform

The computation of the global coordinates will be as simple as multiplying by a rigid transform matrix the 6DOF camera pose:

$$T_i = [R_i|t_i]$$

$where\ R_i\ is\ a\ rotation\ matrix$ and $t_i\ is\ a\ 3D\ translation\ vector$

$$v_i^g(u) = T_{i-1} v_i(u)$$

$$n_i^g(u) = R_{i-1} n_i(u)$$

ii) STEP 2: CAMERA TRACKING

In this step, our objective is to estimate the global camera coordinates $T_i = [R_i\|t_i]$

We will estimate a single 6DOF transform that closely aligns the current oriented points with those of the previous frame with the ICP algorithm computed in parallel. This will give us a T^{rel}, which will be applied incrementally to give the single global camera pose T_i.

There are many different types of ICP algorithm; in our case we will use *projective data association.* Known data are the previous camera pose T_{i-1} and the each point in general coordinates v_{i-1}^g.

Pseudocode of the algorithm is explained here:

PSEUDOCODE	***EXPLANATIONS***
1: For each image pixel $u \in depth\ map\ D_i$	- We will operate in **parallel** for faster computation
2: if $D_i(u) > 0\ then$	- For this, we operate with the previous **validity mask M**
3: $T_{i-1}^{-1} v_{i-1}^g \rightarrow v_{i-1}$	- We transform last vertex from global coordinates to **image coordinates**

4:	Perspective project vertex $v_{i-1} \rightarrow p$	- Projection along the **ray**
5:	if $p \in vertex\ map\ V_i$	- If the **proyection of last** vertex is in the **new** vertex map…
6:	$T_{i-1}V_i(p) \rightarrow v$	- Estimate **new global coordinates** vertex and normals based on previous transformation matrixes (it should be around that transformation)
7:	$R_{i-1}N_i(p) \rightarrow n$	
8:	$if\ \|v - v_{i-1}^g\| < distance\ threshold$ and $n.n_{i-1}^g < normal\ threshold$ then	- If we pass a **distance** and **angle** threshold...
9:	Point correspondence found	- We found a new (good) point !

THEORY

POINT TO PLANE ICP OPTIMIZATION

The point-to-plane ICP has been shown to converge much faster than one that uses the point-to-point error metric. Typically, at each iteration of the ICP algorithm, the relative change of the camera pose is minimized with non-linear least-squares methods which are really slow. We can cite nonlinear least squares methods, such as the Levenberg-Marquardt method [45].

Kok-Lim Low had the idea of approximating the non-linear least-squares method by a linear least-squares method, when the relative change is very small. [46] We will explain his approximation from the typical ICP:

Point-to-plane ICP

In point-to-plane ICP, our objective is to minimize the sum of the squared distance between each source point ant the tangent plane at its corresponding destination point:

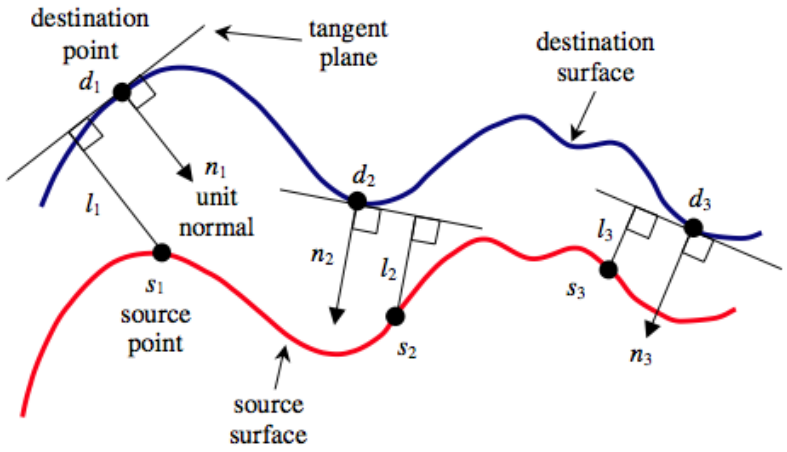


Figure 56: Point-to-plane error between two surfaces

Considering a 3D rigid-body transformation[13] M is composed of a rotation matrix $R(\alpha, \beta, \gamma)$ and a translation matrix $T(t_x, t_y, t_z)$:

$$M = T(t_x, t_y, t_z) \cdot R(\alpha, \beta, \gamma)$$

Where the translation matrix is:

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And rotation matrix is $R(\alpha, \beta, \gamma) = R_z(\gamma) \cdot R_y(\beta) \cdot R_x(\alpha)$=

$$= \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where:

$r_{11} = cos\gamma cos\beta$

$r_{12} = -sin\gamma cos\alpha + cos\gamma sin\beta sin\alpha$

$r_{13} = sin\gamma sin\alpha + cos\gamma sin\beta cos\alpha$

$r_{21} = sin\gamma cos\beta$

$r_{22} = cos\gamma cos\alpha + sin\gamma sin\beta sin\alpha$

$r_{23} = -cos\gamma sin\alpha + sin\gamma sin\beta cos\alpha$

$r_{31} = -sin\beta$

$r_{32} = cos\beta sin\alpha$

$r_{33} = cos\beta cos\alpha$

And $R_x(\alpha)$, $R_y(\beta)$ and $R_z(\gamma)$ are rotations of α, β, γ in radians around the x-axis, y-axis and z-axis respectively.

Consider also that we design the source points homogeneously as $s_i = (s_{ix}, s_{iy}, s_{iz}, 1)^T$ and destination points as $d_i = (d_{ix}, d_{iy}, d_{iz}, 1)^T$. Unit normal vectors at d_i are $n_i = (n_{ix}, n_{iy}, n_{iz}, 0)^T$.

The goal of each ICP iteration will be to find Mopt ***which minimizes:***

[13] We will consider for this section explanation same notation as in the cited paper

$$\arg\min_{M} \sum_{i} \left((M \cdot s_i - d_i) \cdot n_i \right)^2$$

As we know, this equation is a non-linear least-squares equation. We will linearize it under the assumption that for high-computational speed systems, there is almost no time to perform a rotation between iterations, so we can consider the angles $\alpha, \beta, \gamma \approx 0$. Under the assumption of a generic angle $\theta \approx 0$, $\sin\theta \approx 0$ and $\cos\theta \approx 1$, and therefore:

$$R(\alpha, \beta, \gamma) = \begin{bmatrix} 1 & \alpha\beta - \gamma & \alpha\gamma + \beta & 0 \\ \gamma & \alpha\beta\gamma + 1 & \beta\gamma - \alpha & 0 \\ -\beta & \alpha & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \approx \begin{bmatrix} 1 & -\gamma & \beta & 0 \\ \gamma & 1 & -\alpha & 0 \\ -\beta & \alpha & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \hat{R}(\alpha, \beta, \gamma)$$

And we can rewrite previous M optimization as:

$$\arg\min_{\hat{M}} \sum_{i} \left(\left(\hat{M} \cdot s_i - d_i \right) \cdot n_i \right)^2$$

Each $\left(\hat{M} \cdot s_i - d_i \right) \cdot n_i = \left(\hat{M} \cdot \begin{pmatrix} s_{ix} \\ s_{iy} \\ s_{iz} \\ 1 \end{pmatrix} - \begin{pmatrix} d_{ix} \\ d_{iy} \\ d_{iz} \\ 1 \end{pmatrix} \right) \cdot \begin{pmatrix} n_{ix} \\ n_{iy} \\ n_{iz} \\ 0 \end{pmatrix} =$

$= \left[(n_{iz}s_{iy} - n_{iy}s_{iz})\alpha + (n_{ix}s_{iz} - n_{iz}s_{ix})\beta + (n_{iy}s_{ix} - n_{ix}s_{iy})\gamma + n_{ix}t_x + n_{iy}t_y + n_{iz}t_z \right] - [n_{ix}d_{ix} + n_{iy}d_{iy} + n_{iz}d_{iz} - n_{ix}s_{ix} - n_{iy}s_{iy} - n_{iz}s_{iz}]$.

Given N pairs of point correspondences, we can arrange all $\left(\hat{M} \cdot s_i - d_i \right) \cdot n_i, 1 \le i \le N$ into a matrix expression: $Ax = b$

Where:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & n_{1x} & n_{1y} & n_{1z} \\ a_{21} & a_{22} & a_{23} & n_{2x} & n_{2y} & n_{2z} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{N1} & a_{N2} & a_{N3} & n_{Nx} & n_{Ny} & n_{Nz} \end{pmatrix}$$

$$x = (\alpha \quad \beta \quad \gamma \quad t_x \quad t_y \quad t_z)^T$$

And $b = \begin{pmatrix} n_{1x}d_{1x} + n_{1y}d_{1y} + n_{1z}d_{1z} - n_{1x}s_{1x} - n_{1y}s_{1y} - n_{1z}s_{1z} \\ n_{2x}d_{2x} + n_{2y}d_{2y} + n_{2z}d_{2z} - n_{2x}s_{2x} - n_{2y}s_{2y} - n_{2z}s_{2z} \\ \vdots \\ n_{Nx}d_{Nx} + n_{Ny}d_{Ny} + n_{Nz}d_{Nz} - n_{Nx}s_{Nx} - n_{Ny}s_{Ny} - n_{Nz}s_{Nz} \end{pmatrix}$

With $a_{i1} = n_{iz}s_{iy} - n_{iy}s_{iz}$

$$a_{i2} = n_{ix}s_{iz} - n_{iz}s_{ix}$$

$$a_{i3} = n_{iy}s_{ix} - n_{ix}s_{iy}$$

So we can say that: $\arg\min_{\widehat{M}} \sum_i \left(\left(\widehat{M} \cdot s_i - d_i \right) \cdot n_i \right)^2 = \min_x |ax - b|^2$, and we could first solve the system:

$$\boldsymbol{x_{opt} = argmin_x |ax - b|^2}$$

Which is a **standard linear least-squares problem**.

Recapping, given the previous correspondences between $v_i, v_{i-1}^g and\ n_{i-1}^g$, the output of ICP is a single relative transformation matrix T^{rel} that minimizes the point-to-plane error metric. This is defined as:

$$\arg\min \sum_{valid\ u} \left\| \left(T^{rel} v_i(u) - v_{i-1}^g(u) \right) \cdot n_{i-1}^g(u) \right\|^2$$

To resolve this system, we will linearize the system by assuming that incremental transformation occurs between frames. This means that $T^{rel} = \tilde{T}_{inc}^z \cdot T_{i-1} = [\tilde{R}^z | \tilde{t}^z\ ] \cdot T_{i-1} = \begin{bmatrix} 1 & \alpha & -\gamma & t_x \\ -\alpha & 1 & \beta & t_y \\ \gamma & -\beta & 1 & t_z \end{bmatrix} \cdot T_{i-1}$ where $\alpha, \beta, \gamma\ are\ the\ rotations\ along\ the\ axis$

The linear system is computed and summed in parallel on the GPU using Tree reduction.

The solution of this system is solved in the CPU using Cholesky decomposition.

THEORY

CHOLESKY DECOMPOSITION [47]

$Let\ A \in R^{nxn}\ be\ symmetric\ and\ positive\ definite\ which\ means\ \langle x|Ax \rangle > 0$

$for\ all\ x\ \in R^n, x \neq 0.$

$Then\ there\ exists\ a\ unique\ lower\ triangular\ matrix\ L\ R^{nxn}\ with\ strictly$

$positive\ diagonal\ entries\ and:$

$$A = LL^T$$

$-\ Solve\ LL^T x = b\ instead\ of\ Ax = b\ by\ solving\ Lz = b\ for\ z\ and\ L^T x = z\ for\ x$

$-\ If\ you\ drop\ to\ A\ symmetric, positive\ semidefinite,$

$uniqueness\ and\ the\ strictly\ positive\ diagonal\ entries\ drop\ out\ of\ the\ theorem\ as\ well.$

$-The\ cost\ of\ Cholesky\ decomposition\ is\ around \frac{n^3}{3} FLOPS$

Novel contribution: One of the novel contributions of this GPU-based camera tracking is that ICP is performed on **all the measurements provided** (over the 640x480 = 300,000 points). This type of *dense* tracking is only feasible due to this implementation, and is one of the keypoints of the algorithm.

iii) STEP 3: VOLUMETRIC INTEGRATION

In this step, our objective will be to assemble new points with older ones in the best efficient way

For accomplishing our objective, we will use a truncated signed distance function TSDF for our purposes.

THEORY

TRUNCATED SIGNED DISTANCE FUNCTION [48]

A *distance function* $d(x)$ is defined as $d(x) = \min(|x - x_I|)\ for\ all\ x_I \in \partial\Omega$. This implies that $d(x) = 0$ on the boundaries where $x \in \partial\Omega$.

A *signed distance function* (SDF) is an implicit function ϕ with $|\phi(x)| = d(x) for\ all\ x$. Thus, $\phi(x) = d(x) = 0\ for\ all\ x \in \partial\Omega, \phi = -d(x) for\ all\ x \in \Omega^-\ and\ \phi(x) = d(x) for\ all\ x \in \Omega^+$.

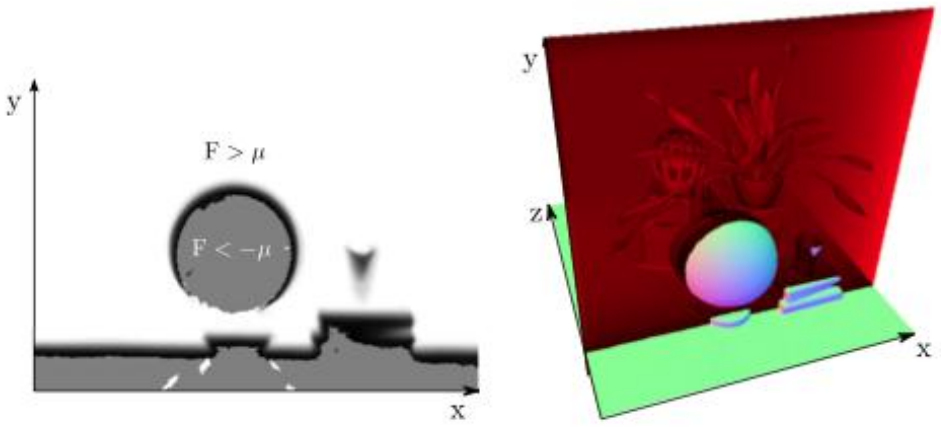


Figure 57: Signed distance function for a slice of volume

This can be easily understood by the previous figure: for every point inside the surface, the SDF value is more and more negative if we go deeper into the object. For outside points the procedure is the opposite, and their value goes higher when rising the distance to the surface.[14]

[14] Wikipedia info about the SDF is wrong as they give opposite values to the points.

In practical, we will use the **TSDF** instead of the SDF, which only stores a truncated region around the actual surface. Brian Curless and Marc Levoy [49], describe its **properties** as:

- ***Representation of range of uncertainty:*** Data typically has asymmetric error distributions with primary directions along sensor lines of sight. The integration method solves that.

- ***Utilization of all data:*** In other representations, we do not represent redundant points. With the TSDF we will reduce noise including redundant observations

- ***Incremental and order independent updating:*** It is possible to update the reconstruction after each scan, independently and not biased by early scans, which will lead into straightforward parallelization.

- ***Time and space efficiency***: When reconstructing high dense detail models, space and time efficiency is mandatory. With TSDF we will fulfill this requirement.

- ***Robustness:*** When dealing with depth algorithms, we must handle situations that could lead into catastrophic situations as holes in surfaces and self-intersecting surfaces.

- ***No restrictions on topological type:*** The algorithm should not assume that the object is of a particular form or genus.

- ***Ability to fill holes in the reconstruction:*** Given a set of range images that do not completely cover the object, the final result when overlapping will be incomplete. This algorithm can automatically fill holes with plausible surfaces yielding a model esthetically both pleasing and "watertight".

We define $d_i(x)$ $the\ signed\ distance\ profile\ i, and\ w_i(x)\ the\ weight\ of\ the\ profile\ i$.

We define the final TSDF at a point x as: $D(x) = \frac{\sum w_i(x) d_i(x)}{\sum w_i(x)}$

And its weight as: $W(x) = \sum w_i(x)$

Expressed as an incremental calculation: $D_{i+1}(x) = \frac{W_i(x) D_i(x) + w_{i+1}(x) D_{i+1}(x)}{W_i(x) + w_{i+1}(x)}$

For one dimension, the zero-crossing in the incremental formulation can be expressed as:

$R = \frac{\sum w_i r_i}{\sum w_i}$.

The main assumption we will make now is that we will perform SLAM in a determined volume (around $9m^2$ in the implementation). We will divide the volume in a grid of volume slices, find if we are "looking" this volume slice in that instant of time, and if yes, add these new depth measurements to our efficient TSDF:

PSEUDOCODE	***EXPLANATIONS***
1: For each voxel g in x,y volume slice	- We divide our general volume in slices and pick one of them each time in parallel, then sweeping from front to back…
2: while sweeping from front to back	
3: $convert\ g\ from\ grid \rightarrow v^g$	- We convert voxel position into global coordinates
4: $T_i^{-1} v^g \rightarrow v$	- Transform it to camera coordinates
5: Perspective project vertex $v \rightarrow p$	- We project the vertex and check if v is in the view frustum
6: if v is in the camera view frustum then	
7: $\|t_i - v^g\| - D_i(p) \rightarrow sdf_i$	- We check if we are in or out a surface (it is just a change of coordinates)
8: $if\ (sdf_i > 0)\ then$	- If we are outside…
9: $\min\left(1, \frac{sdf_i}{max\,truncation)}\right) \rightarrow tsdf_i$	- We put the minimum value of those two into the tsdf
10: else	- If we are inside or in the surface…
11: $\max\left(-1, \frac{sdf_i}{max\,truncation)}\right) \rightarrow tsdf_i$	- We take the maximum value from both
12: $\min(max\ weight, w_{i-1} + 1) \rightarrow w_i$	- We compute the weight, and the new tsdf average value as described before
13: $\frac{tsdf_{i-1} w_{i-1} + tsdf_i w_i}{w_i} \rightarrow tsdf^{avg}$	
14: Store w_i and $tsdf^{avg}$ at voxel g	- We store both values

Lines number 8 and 11 are just the implementation of the truncation:

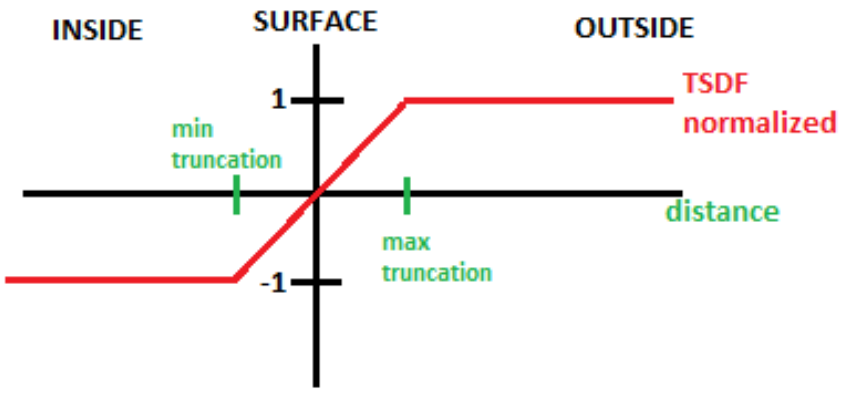


Figure 58: Plotted representation of steps 8 until 11

iv) STEP 4: RAYCASTING FOR RENDERING

Raycasting is a rendering technique that was presented in 1968 by Arthur Appel. The idea behind ray casting is ***to shoot rays from the eye***, one per pixel, and find the closest object blocking the path of that ray. Using the material properties and the effect of the lights in the scene, this algorithm can determine the shading of this object. The simplifying assumption is made that if a surface faces a light, the light will collide with the surface and not be blocked or in a shadow. One of the properties of ray casting is that it can deal with non-planar surfaces and solids easily.The main core of the algorithm is to send one ray per screen pixel and trace this ray through the volume. This can be extended to GPU programming easily allowing us to get a real-time rendering.

Figure 59: Videogames as Quake III uses this raycasting rendering techniques

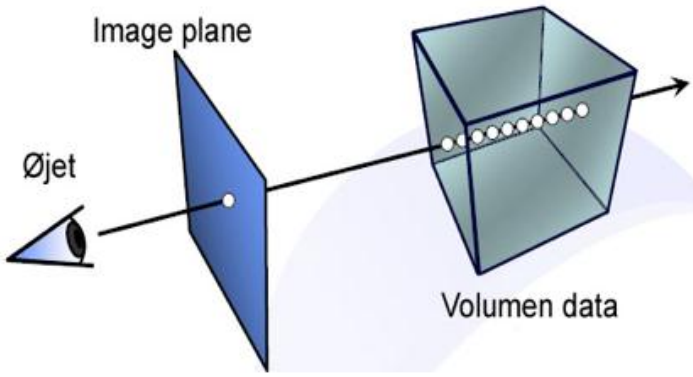


Figure 60: Raycasting idea

At each point along the ray, there is an illumination I(x,y,z) reaching the point (x,y,z) from the light source(s). The intensity scattered along the ray to the eye, depends on this value, a reflection function or phase function P, and the local density D(x,y,z). The dependence on density expresses the fact that a few bright particles will scatter less light in the eye direction than a number of dimmer particles.

One of the ***keypoints of using before the TSDF***, is that we can compute the ***surface gradient*** as $\nabla tsdf$ which will be really fast compared to other traditional methods.

(For each pixel of the final image computed in parallel)

Backproject [u,0] to get the start of the ray

Backproject [u,1] to get the end of the ray

Get the ray direction with the normalized vector (Ray end – Ray start)

We initalize an auxiliary variable RayLen = 0

*Introduce in g the **first voxel along the ray***

*Convert **global mesh vertex to grid position** and introduce it into m*

*The **distance from the beginning of the ray** until m → m_distance*

***While** the **voxel is in volume bounds** (which was predefined as a 3x3x3 m^2) …*

> *RayLen = RayLen + 1*
>
> *We store g in g_previous*
>
> *We get **next g** along the ray*
>
> ***If** we get a **zero crossing** from g to g_previous*
>
> > *We extract the **trilinear interpolated grid position** → p*
> >
> > ***Convert p from grid to global 3D position** → v*
> >
> > *Convert p to surface gradient as $\nabla tsdf(p)$ → n*
> >
> > *We give a **shade** to the pixel oriented point (v,n) or follow secondary ray for computing more complex reflections (not needed)*
>
> ***If RayLen, our auxiliary variable it is greater than m_distance***
>
> > *We **shade** the pixel using mesh maps or follow secondary ray (shadow, reflections) as before.*

It is important to consider that the physical memory we will use is not neglectable, in example for a 1024x1024x1024 map is around 4GB of hard disk.

v) STEP 5: POSSIBLE INTERACTION STEP/SEGMENTATION

As described before, one of the possibilities we get with this algorithm, is that we can compute an interaction step with the map we are building. This step was not implemented in my case because I wanted to focus on the SLAM approach and not on the Augmented Reality interactions. We can see some results as throwing particles **while** generating map in the next figure:

Figure 61: Figure showing Microsoft Research KFusion interacting step.

2.4 KINECTFUSION: IMPLEMENTATION

2.4.1 Implementation

- Code was tested under a *DEL XPS core i7 8GB RAM* provided a *NVIDIA GTX540m* (96 cores)
- The operating system *is Linux Ubuntu 11.10 64-bits* [50]
- The implementation uses *3D PCL libray* [51] *(Point Cloud Libraries) + GTK*
- Developed under *C++* and *CUDA language*

Due to my specific laptop characteristics (dual graphics card) a switch was needed to force *Optimus technology* [52] to be overpassed under Ubuntu. In my case**,** *I choose Ironhide* [53] *hacking.*

2.4.2 Results

Kinect Fusion implementation is really complicated to code due to several things:

- Mathematics below
- Parallel programming (CUDA in our case)

In our case, PCL (Point Cloud Library) parallel implementations were fundamental to get as ***results a 3x3x3 m^3 @ 4fps with 1-3mm average error.***

Figure 62: Screenshoot of myself under KF

A full real-time demo will be presented on the final dissertation.

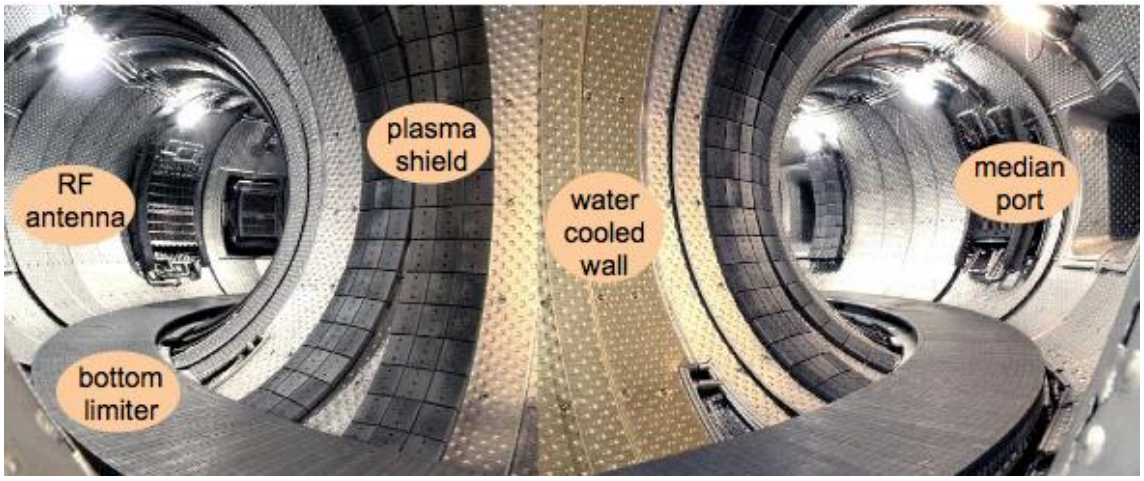
2.4.3 Extensions of this algorithm

Some extensions of this algorithm could be:

i) Module extension for inspection prototype

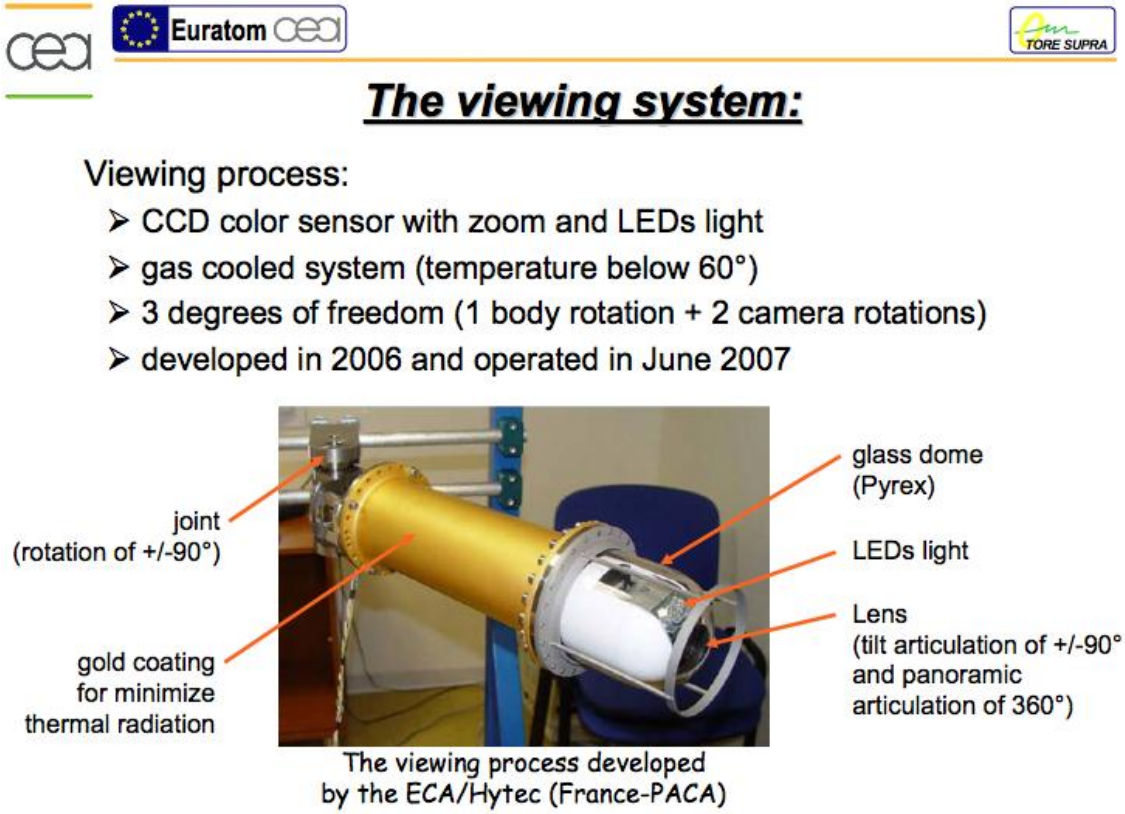
One real application could be an extension for the robot inspection prototype developed by our teachers from Arts et Métiers PARISTECH M. Nazih Mechbal and M. André Barraco. Its novel contribution is the use of ***inflatable*** modules for extending or folding the

arm.



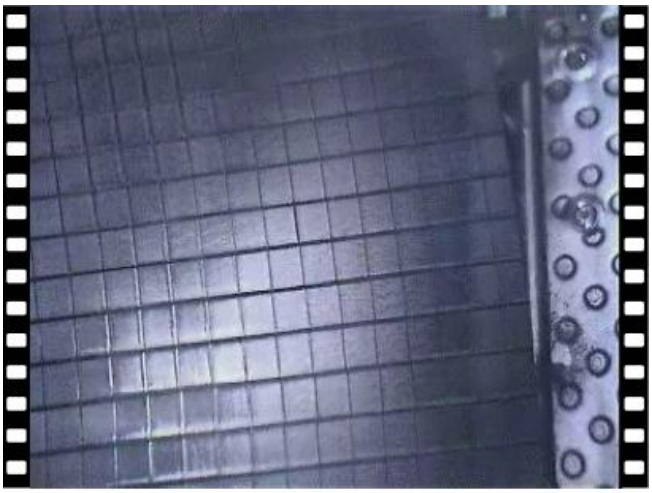
One of the possibilities of this prototype, is its use for failure-inspection as in the TORE SUPRA. The Tore Supra is a French tokamak [23], that is a device that uses a magnetic field to confine a plasma inside a volume, typicaly torus[15] shaped, for producing energy when this plasma is over 100 million degrees.

For inspecting camera (carbon particles are confined sometimes on its walls), the real arm used inside the plasma vessel of Tore Supra, had (at least in 2007) a CCD color sensor with zoom and LEDs light. [54]



The output of the AIA robot CCD camera is:

[15] A torus sometimes is informally also called *doughnut*



Video acquisition from viewing process

Some laser applications as LIBS systems (Laser Induced Breakdown Spectroscopy) were planned for developing. Instead of developing these expensive methods, implicit surface methods would allow to have ***at least the same data on real time with much lower hardware costs***.

Another advantage is that this type of depth cameras ***do not need illumination***, as described on Section 1 of this document.

ii) Reverse-engineering

Reverse-engineering could be done on real time with low-cost equipment. We could generate 3D models easily from scratch in a few seconds.

iii) AR applications

The purpose of this algorithm for Microsoft is to eliminate barriers for input data to a computer. We can write directly on a table with our finger without any need of tablet/magic tool, just a Kinect.

SECTION 3: QUADROTORS

3.1 INTRODUCTION

Human beings have always tried to go through the limits of the feasible, to convert an idea into a reality. One of this dreams, have been always the possibility of ***fly***. Due to our body limitations, it has impossible for us to fly as eagles, but science permitted us to learn the technique to *explore* and *travel* through the sky.

First flying objects date from year 400 BC when the Chinese philosopher Mo Zi tried first *kite* designs. From then, many designs have been done as famous *Leonardo Da Vinci designs* in 1485, but it was in 1903 when *Wright's brothers* tried successfully the *first plane* over some meters. More and more evolved designs were produced, remarkably in 1947 when the first *break of the sound's barrier* was produced. The career of conquering the space touched a point of inflexion when in 1969 *Apollo landed in the Moon*.

Figure 63: An Eagle flying.

Figure 64: Example of military ground base system for an UAV

All those systems had always one common limitation, ***the pilot, a huge constraint***. Fastest aircrafts are *nothing* without the expertise of a good pilot.

From 1917, during the 1st and 2nd WW, this constraint tried to be eliminated with a new definition of air vehicle: ***Unmanned Aerial Vehicles (UAV's),*** also called *Unmanned Aircraft Systems (UAS's)* or *Remotely Piloted Aircraft (RPA)*.

UAV's are aerial vehicles that do not require an inside pilot and they can be controlled by a base control located in a very far position.

One sub classification in UAV's is the ***Autonomous Unmanned Aerial Vehicles (AUAV's).*** These vehicles do not even need a pilot as they have all the "*intelligent*" side from a pilot integrated in their control system. One of the *more complex problems* until now for AUAV's has been to solve ***SLAM problem fully autonomously***. Ground competitions have been organized during last years under the *DARPA competition*, where MIT team reached the challenging task of drive a car autonomously through some undetermined map avoiding obstacles and more. In parallel the *Symposium on Indoor Flight Issues* has an economical award for indoor drones, giving increasing difficult tasks to be achieved each year.

Going back to UAV's, many classifications are possible attending type. We can remark planes, quadrotors, helicopters, balloons, coaxial rotors drones as they are developing in the Mechatronics department in Arts et Métiers PARISTECH.

These systems have special ***disadvantages*** as:

- They typically have *high vibrations* (due to the high speed of wings) that are complex to control
- *External forces* (wind) can change our equilibrium point quickly
- If control is lost, there is a high probability that the AUAV's will be damaged, lost or destroyed

But they also have some special ***advantages*** as:

- *High speed*
- *Avoidance of ground obstacles*
- *Aerial point of view*
- *Undetected at high altitudes*

To finish this introduction, we must remark how recent military conflicts have put the development of unmanned systems as combat tools in the spotlight. It has been of special interest of the mainstream media of the USA. Projects around this subject are being develop, as ANGEL ConOps from the Kentucky University (2011) [55], were a quadcopter assists militaries giving enemy position and information:

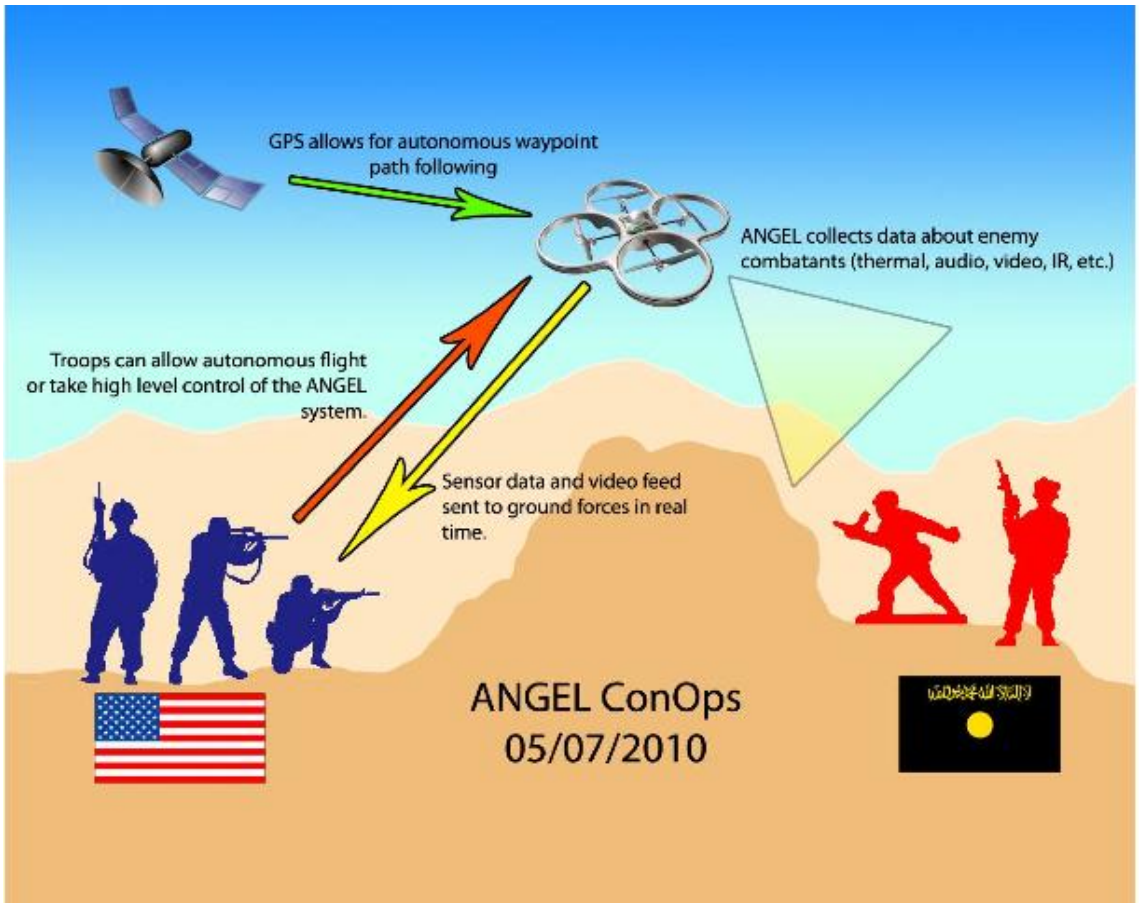


Figure 65: ANGEL ConOps scheme /with SLAM autonomous flight

3.2 SELECTION CRITERIA AND SPECIFICATIONS

When choosing type of quadrotor, I opted for choosing a quadrotor due to these ***main characteristics***:

- It can remain *static in a point in the space*, a difficult, almost impossible task for a plane
- It can be used *indoor* which is hard to achieve with balloons and planes
- It is *relatively affordable.*

Figure 66: AR.Drone from Parrot

I wanted to focus my work on making a general overview of SLAM techniques, so I acquired the ***AR.Drone model***, with a SDK already developed and two cameras integrated. The model Parrot AR.Drone can be found for about three hundred euros, end 2011, in online stores and specific resellers.

As its own name says, a quadrotor helicopter or quadcopter is an aerial vehicle propelled by four rotors. The output pitch is always constant, which makes the quadcopter mechanically simpler than an ordinary helicopter [56]. But instability problems are not neglectable, and to control it is really complicated. A big progress has been done and there are really surprising novel implementations that we will overview later.

The following table [57] is a very interesting comparison between different drones, shows that a quadcopter is also a good selection from an overall point of view:

Categories	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*
Power Cost	2	2	2	2	1	4	3	3
Control Cost	1	1	4	2	3	3	2	1
Payload/volume	2	2	4	3	3	1	2	1
Maneuverability	4	2	2	3	3	1	3	3
Mechanics Simplicity	1	3	3	1	4	4	1	1
Aerodynamics Complexity	1	1	1	1	4	3	1	1
Low Speed Flight	4	3	4	3	4	4	2	2
High Speed Flight	2	4	1	2	3	1	3	3
Miniaturization	2	3	4	2	3	1	2	4
Survivability	1	3	3	1	1	3	2	3
Stationary Flight	4	4	4	4	4	3	1	2
TOTAL	24	28	32	24	33	28	22	24

A=Single Rotor, B=Axial Rotor, C=Coaxial Rotor, D=Tandem Rotors, E=Quadrotor, F=Blimp, G=Bird-like, H=Insect-like. 1=Poor, 4=Excellent

Figure 67: Comparison between different types of drones

3.2.1 Technical specifications [58]

Figure 68: Front-camera Drone detail

The core of the AR.Drone is an ***ARM9*** *468 MHz embedded microcontroller with 128 Megabytes of RAM running the Linux operating system*. ***Communications*** are through Wi-Fi (b/g) and USB. The ***inertial guidance systems*** use a *MEMS 3-axis accelerometer, 2-axis gyrometer and a single-axis yaw precision gyrometer*. An ***ultrasonic altimeter*** with a range of 6 meters provides vertical stabilization.

The ***structure*** is constructed of *carbon-fiber tubes*. The *15W electric motors* are ***brushless*** type driving high-efficiency propellers. Power is from a rechargeable *Lithium polymer* ***battery*** *pack delivering 11.1 V, 1000 mAh*. With a ***weight*** of 380 g or 420 g (with "indoor hull") it can maintain flight for about 12 minutes with a speed of 5 m/s which is 18 km/h.

Two ***cameras*** are fitted, a wide-angle (93°) at the front which can supply live stream (640x480 pixels VGA) to a remote monitor, and a *high speed vertical camera* (64° lens) which can supply 60 frame/second.

The ***front camera*** can be used with software to detect a second AR Drone at 0.3-5 m giving validation of shots fired at enemy drones, positioning of virtual objects, and calculation of markers of virtual objects.

3.3 QUADROTOR – CONTROL

3.3.1 Coordinate axis, angle references

Typically, we can make the assumption of considering the ***physical center*** of the quadrotor, as the center of mass and gravity (and origin of our relative referencing) due to its double symmetry along the x and y-axis. We define then ***three angles*** which will be in our state-vector, in the case we want to use it for our control model. These angles are the pitch θ, the yaw ϕ and the roll η, described in the next figure:

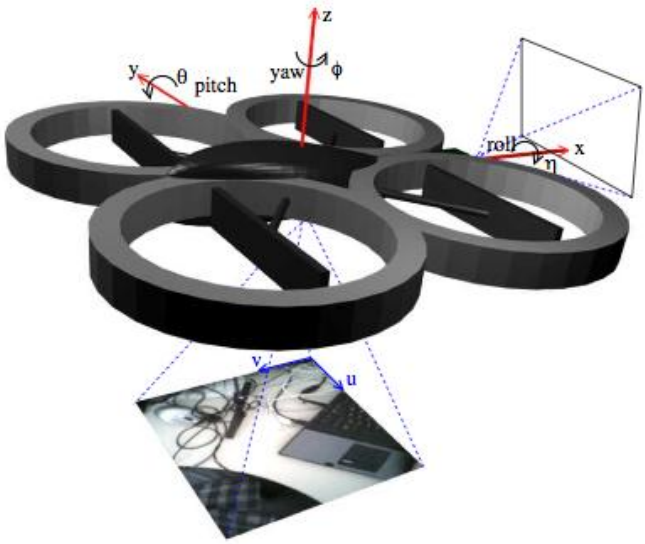


Figure 69: AR.Drone with reference coordinates and frontal/aerial camera views

If a more precise modeling is needed, a good methodology for estimating accurately these parameters would be:

Demount the quadrotor into the smaller pieces as possible→ Model them in a 3D CAD software with mathematical calculus as SolidWorks→ Weight each part→ Compute Inertia, centers of gravity and mass calculus→Build the new accurate model.

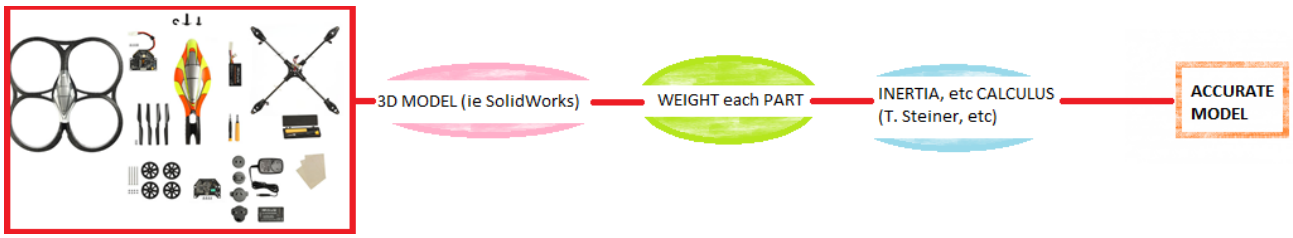


Figure 70: Precise quad modeling. *José Manuel Glez. De Rueda Ramos*

3.3.2 Basic Movements

A good classification of quadrotor ***basic movements*** is described in the next figure. First four movements a-b-c-d are *planar translations through four directions*. To accomplish this, we will reduce and increase motor speeds from two opposite rotors. Next two basic movements e-f are *vertical displacements*, which can be achieved increasing/decreasing all speeds (to go up/down). Last two basic movements, g-h are rotations of the yaw angle ϕ.

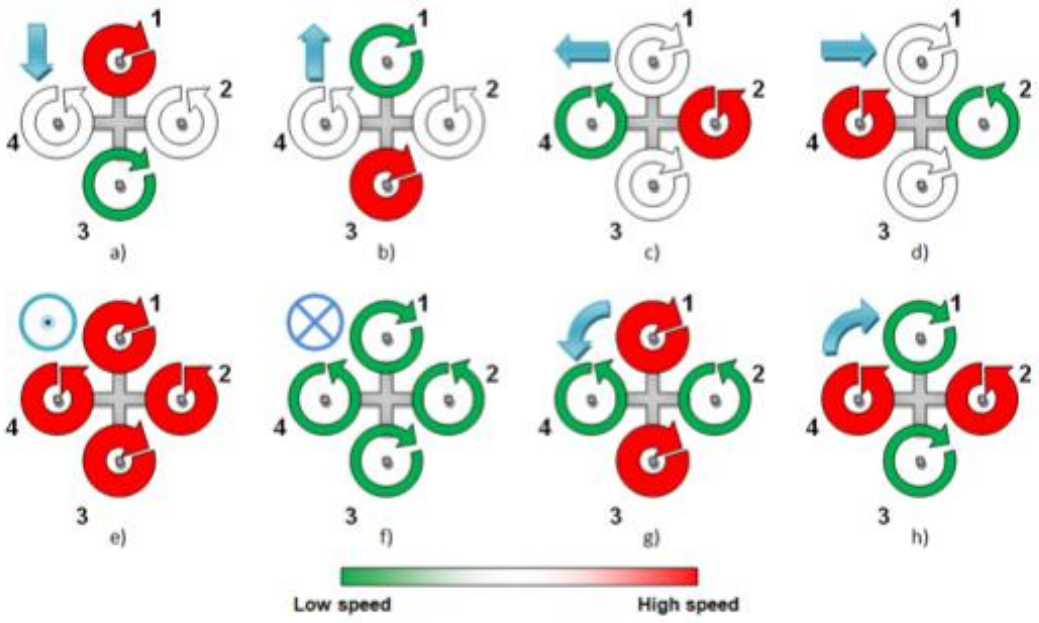


Figure 71: Quadrotor basic movements

3.3.3 Modeling

The control of a quadrotor is not the main objective of this thesis, but I think it is important at least to cite some of the SoA methods that are used right now for controlling these AUAV's.

i) AR.Drone control

The AR.Drone has a PID control system integrated implemented in C. I have tested this quadrotor under some slight wind conditions and the result is catastrophic, resulting almost always in a crash.

ii) Aggressive Maneuvers with Quadrotors

GRASP Laboratories have a good video in youtube as an example of their trajectory control systems [59].

Let´s overview their paper [60].

Starting with defining the references. Let's define the world frame W, defined by the axis x_W, y_W, z_W. The body frame B is attached to the center of mass (CM), and its defined by the axis $x_{Baxis}, y_{Baxis}, z_{Baxis}$. To perform the change of coordinate basis from world coordinates W to quadrotor coordinates B, we should perform three rotations in $Z - X - Y$ $Euler\ angles$.

The rotation matrix to transforming coordinates from $B\ to\ W$ is given by [61]:

$$R = \begin{bmatrix} c\psi c\theta & -s\phi s\psi s\theta & -c\phi s\psi & c\psi s\theta + c\theta s\phi s\psi \\ c\theta s\psi + c\psi s\phi s\theta & c\phi c\psi & s\psi s\theta - c\psi c\theta s\phi \\ -c\phi s\theta & s\phi & c\phi c\theta \end{bmatrix}$$

Where $s(\cdot)\ and\ c(\cdot)$ represent $\sin(\cdot)\ and\ \cos(\cdot)$.

The equations governing the acceleration of the CM are:

$$m\ddot{r} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R \begin{bmatrix} 0 \\ 0 \\ \sum F_i \end{bmatrix}$$

The *CM position* vector in the world frame is denoted by r.

The components of angular velocity of the robot in the body frame are $p, q\ and\ r$. These values are related to the derivatives of the roll, pitch and yaw angles according to:

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} c\theta & 0 & -c\phi s\theta \\ 0 & 1 & s\phi \\ s\theta & 0 & c\phi c\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}$$

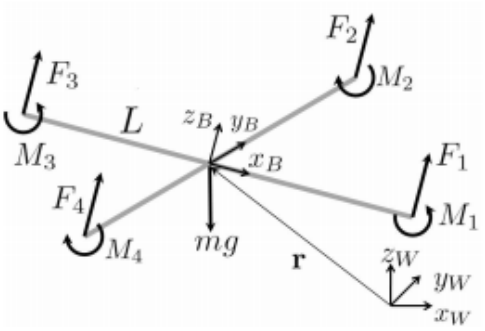


Figure 72: Coordinate systems and forces/moments acting on the quadrotor

And an angular acceleration of:

$$I \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} L(F_2 - F_4) \\ L(F_3 - F_1) \\ (M_1 - M_2 + M_3 - M_4) \end{bmatrix} - \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times I \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

Motors forces are modeled as a quadratic function of their angular speed. Their produced moments are also a quadratic function of their angular speed.

Then they define three controls

a PID ***based attitude control***,

a ***hover control*** based on as they call the "stiff hover control" for getting a rough approximation and the "soft hover control" for refining this approximation. an attitude control

a ***3D trajectory control*** with the position error defined as:

$$e_p = \left((r_T - r) \cdot \hat{n}\right) \cdot \hat{n} + \left((r_T - r) \cdot \hat{b}\right) \hat{b}$$

And the velocity error as $e_v = \dot{r}_T - \dot{r}$.

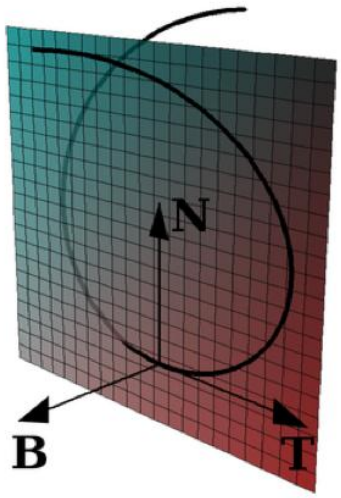


Figure 73: Normal, binormal and tangent vectors to point of a curve

Where $\hat{n}$ *and* $\hat{b}$ *are the normal and binormal directions.*

Then, five key phases are defined:

- ***Phase 1*** - hover control (stiff) to a desired position
- ***Phase 2*** - control to desired velocity vector
- ***Phase 3*** - control to desired pitch angle
- ***Phase 4 -*** control to zero pitch angle
- ***Phase 5 -*** hover control (soft) to a desired position

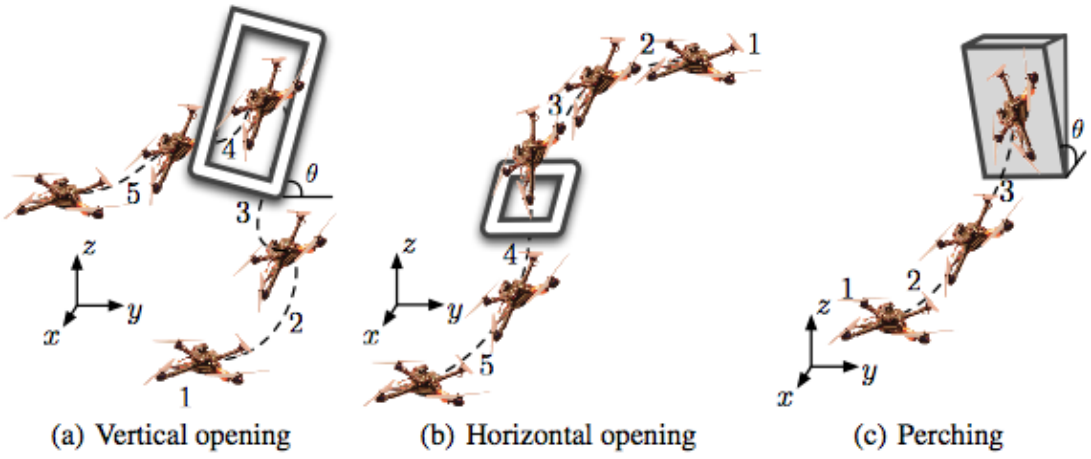


Figure 74: Different test succesfully performed

Successful test are performed where parameters are refined each time. Some of those test where to perform a triple 360 rotation, going vertically and horizontally through a window, and performing a piercing.

3.3.4 LQR

Another implementation was presented in using linear multivariable control techniques [62]. They do not compare against other techniques, but performance it is supposed to be good with wind tail and other disturbances.

3.3.5 FUTURE WORK: MULTIWORK APPROACH

GRASP Laboratories stroke again with an amazing video highly recommended uploaded 31 January 2012. The use minidrones to perform various complicated formations, and changes between formations. [63]

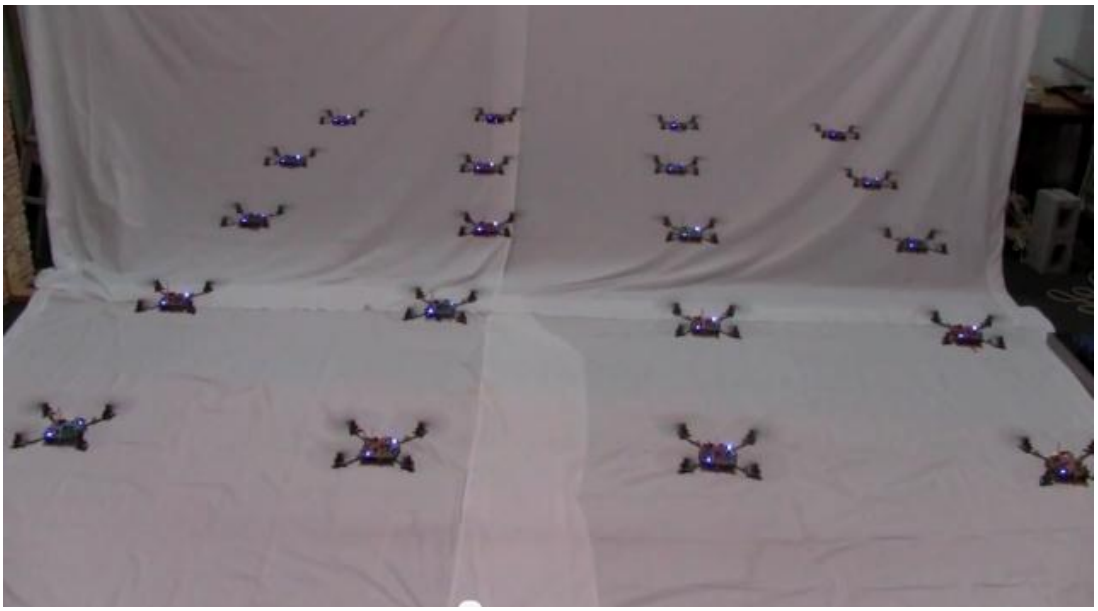


Figure 75: Robots performing different formations

SECTION 4: SLAM FOR FLYING ROBOTS

In conclusion, ***there is no an all-in-one SLAM algorithm*** that can perform a generic Simultaneous Localization and Mapping. After reading this document, you should be able to choose a right SLAM algorithm depending on your specific needs.

All descriptions are inside this document or either their respective cited papers.

Before writing a line of code, we should consider this ***diagram*** I made for choosing/developing the right algorithm.

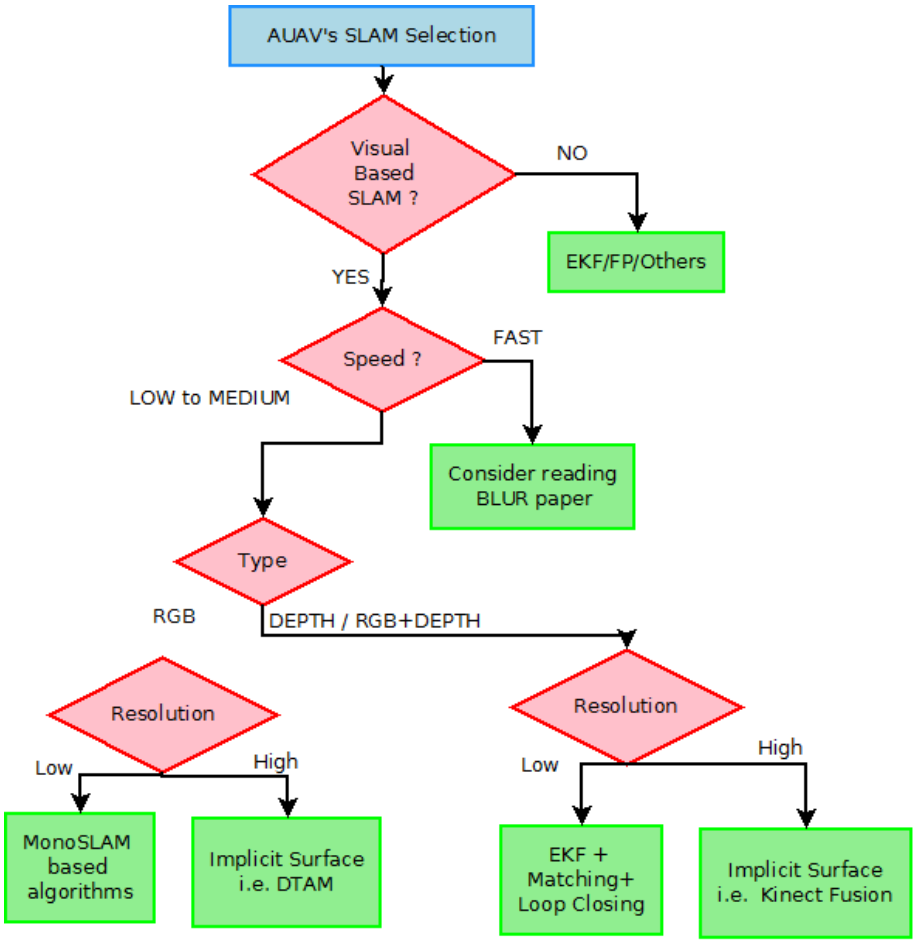


Figure 76: Global AUAV's SLAM selection

CONCLUSIONS

I reiterate my **thanks** to ARTS et METIERS PARISTECH Mechatronics department, leaded by ***Nazih Mechbal***, and giving me the oportunity of going in exchange to the university TU MUENCHEN. Reciprocically, I extend my greetings again to the mentioned university and CAMPAR department, specially to ***Ilic Slobodan***.

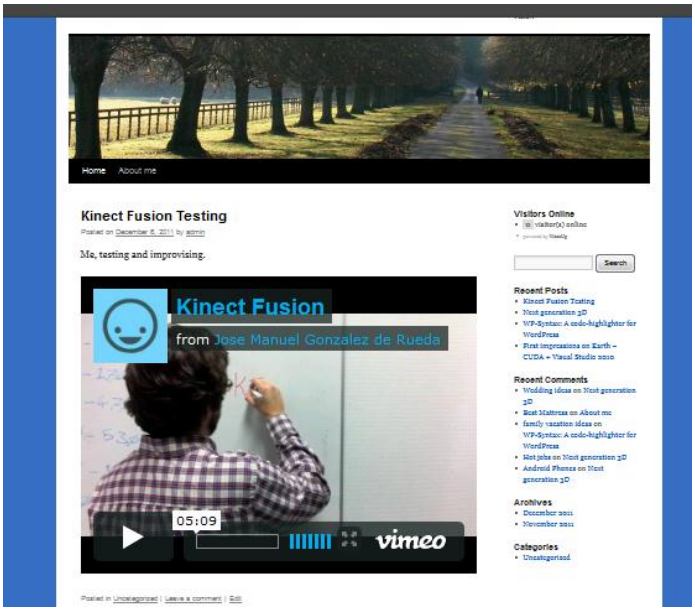


Figure 77: www.RSLAM.com screenshot

The ***global objective*** of the thesis was to overview the State of the Art of SLAM nowadays. Choosing ***KinectFusion*** as the main pillar of the thesis, gave me the opportunity of fully understanding of one of the most promising SLAM algorithms developped now (literally).

A ***general overview*** has been considered, because KinectFusion is only the last link of a huge chain that has been growing up since some decades ago.

Just for concluding, when I started the thesis in Paris, I made a ***website*** (www.rslam.com) for uploading papers. I reconsidered uploading papers due to copyright permissions, and I changed into a simplistic wordpress blog, with some extra plugins where to publish my advances. It is not neglectable that I received a total of ***860 single visits*** with a medium 3.28 pages viewed per visitors, and only 22,3% considered as marketing or spider bots since 9th December 2011, two months ago. I had no time to update the website as I wanted, but a ***demo***

Figure 78: RSLAM visitors counter plugin screenshot

of Kinect Fusion I made with my Phd colleage Byung-Kuk from South Korea, made my video appear as the *second featured video* by Google.

I hope that you have enjoyed reading this thesis, and that it has given to you a first good approach to SLAM algorithms and robotics.

Thank you for your time.

Sincerely, *José Manuel*.

BIBLIOGRAPHY

[1] A. Currie, The History of Robotics, 1999.

[2] S. Thrun, B. W. and F. D., "Probabilistic robotics," MIT Press, 2001.

[3] S. Papert, "The summer vision project," MIT AI Memo 100 - Massachusetts Institute of Technology, 1966.

[4] E. Kandel, H. Schwartz and T. Jessel, "Chapter 27: Central visual pathways," in *Principles of Neural Science, Fourth Edition*, McGraw-Hill.

[5] I. Slobodan, Writer, *Tracking and Detection for Computer Vision.* [Performance]. TUM - CAMPAR, 2011-2012.

[6] O'Sullivan, Blahut and Snyder, "Information-Theoretic Image Formation (Invited Paper)," IEEE Transactions on Information Theory, Vol 44 No 6, 1999.

[7] B. Bayer, "Bayer Filter Mosaic". Patent U.S. Patent No. 3,971,065, 1976.

[8] N. Navab, "3D Computer Vision II, Time-of-Flight and Kinect Imaging," Munich, Germany, 2011.

[9] G. J. Iddan and G. Yahav, "3D IMAGING IN THE STUDIO (AND ELSEWHERE...)," 3DV Systems Ltd., Israel.

[10] "http://en.wikipedia.org/wiki/Time-of-flight_camera," [Online].

[11] "Canesta ToF cameras," [Online]. Available: www.canesta.com.

[12] D. Fofi, T. Sliwa and Y. Voisin, "A comparative survey on invisible structured light," Le2i UMR CNRS 5158 .

[13] K. Khoshelham, "ACCURACY ANALYSIS OF KINECT DEPTH DATA," ITC Faculty of Geo-

information Science and Earth Observation, University of Twente.

[14] R. Hartley and A. Zisserman, Multiple View Geometry, Cambridge University Press, Second Edition 2003.

[15] R. Szeliski, Computer Vision: Algorithms and Applications, Springer, 2010.

[16] N. Prof. Navab, "3D Computer Vision II," TU Muenchen course, 2012.

[17] F. Klein, "Elementary Mathematics from an Advanced Standpoint," MacMillan, New York, 1939.

[18] Mathworks, "Matlab," http://www.mathworks.de/index.html.

[19] "Understandign Digital Image Interpolation," [Online]. Available: http://www.cambridgeincolour.com/tutorials/image-interpolation.html.

[20] "Yung-Ya Chuang, National Taiwan University," [Online]. Available: www.csie.ntu.edu.tw.

[21] Johnson and Kuby, Elementary Statistics.

[22] D. Lorraine, Classical Probability in the Enlightenment, Princeton University Press, 1988.

[23] Wikipedia:totamak, "wikipedia," [Online]. Available: wikipedia.com.

[24] H. Durrant-Whyte and T. Bailey, "Simoultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms," 2006.

[25] P. Cheesman and R. Smith, "On representations of spatial uncertainty," *Int. J. Robotics Research,* vol. 5, no. 4, pp. 56-68, 1987.

[26] H. Durrant-Whyte, "Uncertain geometry in robotics," in *IEEE Trans. Robotics and Automation*, 1988.

[27] S. R., M. Self and P. Cheeseman, "Estimating uncertain spatial relationships in robotics," in *Autonomous Robot Vehicles*, Springer-Verlag, 1990, pp. 167-193.

[28] H. Durrant-Whyte, D. Rye and E. Nebot, "Localisation of automatic guided vehicles," in *Robotics Research: The 7th International Symposium (ISRR'95)*, 1996.

[29] H. Durrant-White, "Uncertain geometry in robotics," in *IEEE Trans. Robotics and Automation*, 1988.

[30] M. Montemerlo, S. Thrun, D. Koller and B. Wegbreit, "Fast-SLAM: A factored solution to the simultaneous localization and mapping problem," in *AAAI National Conference on Artificial Intelligence*, 2002.

[31] K. Murphy, "Bayesian map learning in dynamic environments," *Advances in Neural Information Processing Systems,* 1999.

[32] S. Thrun, W. Bugard and D. Fox, "A real-time algorithm for mobile robot mapping with applications to multi-robot and 3D mapping," in *International Conference on Robotics and Automation*, 2000.

[33] J. Montiel, J. Civera and A. Davidson, "Unified Inverse Depth Parametrization for Monocular SLAM".

[34] A. Davidson, I. Reid, N. Molton and O. Stasse, "MonoSLAM: Real-Time Single Camera SLAM," *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, 2007,* JUNE 2007.

[35] G. Klein and D. Murray, "Parallel Tracking and Mapping for Small AR workspaces," University of Oxford.

[36] R. Newcombe, S. Lovegrove and A. J.Davidson, "DTAM: Dense Tracking and Mapping in Real-Time," Imperial College London, UK, 2011.

[37] "DTAM," [Online]. Available: http://www.youtube.com/watch?v=Df9WhgibCQA.

[38] H. Seok, J. Kwon and K. Mu, "Simultaneous Localization, Mapping and Deblurring," in *ICCV*, 2011.

[39] S. Izadi, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, A. Davidson and others, "KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera," in *SIGGRAPH 2011*, 2011.

[40] A. N. Richard, I. Shahram, H. Otmar, M. David, K. David, J. D. Andrew, K. Pushmeet, S. Jamie, H. Steve, F. Andrew, p. gonzalez and Newcombe, Richard, "KinectFusion: Real-Time Dense Surface Mapping and Tracking," *ISMAR,* 2011.

[41] S. Paris, P. Kornprobst, J. Tumblin and F. Durand, "Bilateral Filtering: Theory and Applications," *Foundations and Trends in Computer and Vision. Vol.4 No. 1,* pp. 1-73, 2008.

[42] V. Aurich and J. Weule, "Non-linear gaussian filters performing edge preserving diffusion," in *Proceedings of the DAGM Symposium*, 1995.

[43] F. Durand and J. Dorsey, "Fast Bilateral Filtering for the display of high dynamic-range images," *ACM Transactions on Graphics,* vol. 21, pp. 257-266, SIGGRAPH conference, 2002.

[44] T. Jones, F. Durand and M. Desbrun, "Non-iterative, feature-preserving mesh smoothing," *ACM Transactions on Graphics SIGGRAPH,* pp. 654-663, 2003.

[45] W. H. Press, S. A. Teukolsky, W. Vetterling and B. Flannery, Numerical Recipes in C: The Art of Scientific Computing, Second Ed, Cambridge University Press, 1992.

[46] K.-L. Low, "Linear Least-Squares Optimization for Point-to-Plane ICP Surface Registration," Technical Report TR04-004, Department of Computer Science, , University of North Carolina, 2004.

[47] T. Lasser, "Basic Mathematical Tools for Imaging and Visualization IN2124," 2011-2012.

[48] S. Osher and R. Fedwik, "Level set methods and dynamic implicit surfaces," Springer, Stanford, California.

[49] B. Curless and B. Levoy, "A Volumetric Method for Building Complex Models from Range Images," Standford University.

[50] "Ubuntu," [Online]. Available: http://www.ubuntu.com/.

[51] "Point Cloud Library," [Online].

[52] "Optimus Technology by NVidia," [Online]. Available: http://www.nvidia.com/object/optimus_technology.html.

[53] "Ironhide repository," [Online]. Available: https://launchpad.net/~mj-casalogic/+archive/ironhide/+packages.

[54] L. Gargiulo and etal, "DEVELOPMENT OF AN ITER RELEVANT INSPECTION ROBOT," EURATOM CEA and others, 2007.

[55] U. o. Kentucky, "UKnowledge," [Online]. Available: http://uknowledge.uky.edu.

[56] T. Krajn´ık, V. Von´asek, D. Fiˇser and J. Faigl, "AR-Drone as a Platform for Research in Education," *EU-Robot,* no. Springer, 2011.

[57] S. Bouabdallah and R. Siegwart, "Design and Control of a Miniature Quadrotor," *Advances in Unmanned Aerial Vehicles,* pp. 171-200, 2007.

[58] P. USA, "AR.Drone," Parrot, [Online]. Available: www.parrot.com.

[59] G. Laboratory, "Video: Aggressive Maneuvers for Autonomous Quadrotor Flight," [Online]. Available: http://www.youtube.com/watch?v=MvRTALJp8DM.

[60] D. Mellinger, N. Michael and V. Kumar, "Trajectory Generation and Control for Precise Aggresive Maneuvers with Quadrotors," GRASP Laboratory, 2007.

[61] D. Mellinger, N. Michael and V. Kumar, "Trajectory Generation and Control for Precise Aggressive Maneuvers with Quadrotors," GRASP Laboratory, University of Pennsylvania,

Philadelphia, USA.

[62] J. Cowling, Whidbourne and Cooke, "OPTIMAL TRAJECTORY PLANNING AND LQR CONTROL FOR A QUADROTOR UAV," Department of Aerospace Sciences, Cranfield University.

[63] GRASP, "Video: A Swarm of Nano Quadrotors," [Online]. Available: http://www.youtube.com/watch?feature=player_embedded&v=YQIMGV5vtd4.

[64] S. Voisembert, A. Riwan, N. Mechbal and A. Barraco, "A novel inflatable robot with constant and continous volume," in *IEEE International Conference on Robotics and Automation*, Shangai, China, 2011.

[65] R. L. UC3M, "Robotics Lab," [Online]. Available: http://roboticslab.uc3m.es/roboticslab/robot.php?id_robot=1.

[66] J. García Bueno, "Thesis: RGB-D GraphSLAM," Universidad Carlos III de Madrid, 2011.

[67] ARToolKit, "ARToolKit," [Online]. Available: http://www.hitl.washington.edu/artoolkit/.

[68] "BMW Series 5 camera Technology," [Online]. Available: www.bmw.com.

[69] "POINT CLOUD LIBRARY," [Online]. Available: http://pointclouds.org/.

APPENDIX

1. TRANSFORMATIONS MATLAB CODE

```
%%  TRANSFORMATIONS EXAMPLE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                 %
%   by Jose Manuel Gonzalez de Rueda Ramos                        %
%                                                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%% Init and loading
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all; clc;
fprintf('\n--- Transformations example ---\n\n+ Initializing');
lena = imread('lena.bmp');
lenaGray = rgb2gray(lena);
[h,w,d] = size(lena); % d is the depth of the image, in this case 3
(RGB)
fprintf('... init okey\n');

%% Translation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fprintf('+ Translation example\n');

% We ask for the translation in X and in Y
transX = input('  Please introduce a number of pixels to translate in
x: (ex: 70)', 's');
if isempty(transX)
    transX = '70';
end
transX = str2num(transX);
transY = input('  Please introduce a number of pixels to translate in
y: (ex: 70)', 's');
if isempty(transY)
    transY = '70';
end
transY = str2num(transY);

% We create the translated image
imageTr = zeros(h+transY,w+transX,d);
for i=1:h
    for j=1:w
    imageTr(i+transY,j+transX,:) = lena(i,j,:);
```

```
    end
end

% We normalize the image, to show the correct output
imageTr = imageTr/max(imageTr(:));

% We show the result
figure(1)
subplot(1,2,1), subimage(lena)
title('Lena - Original')
subplot(1,2,2), subimage(imageTr)
title('Lena - Translated')

%% Euclidean transformation (rotation + translation)
%%%%%%%%%%%%%%%%%%%%%%
fprintf('+ Euclidean transformation example\n');

% We ask for the translation in X and in Y
transX = input('  Please introduce a number of pixels to translate in
x: (ex: 70)', 's');
if isempty(transX)
    transX = '70';
end
transX = str2num(transX);
transY = input('  Please introduce a number of pixels to translate in
y: (ex: 70)', 's');
if isempty(transY)
    transY = '70';
end
transY = str2num(transY);

% We will suposse the diagonal + the distance for the translation is
the
% maximum image length
diag = ceil(sqrt(h.^2+w.^2));
imageEuc = zeros(diag + max(transX,transY), diag +
max(transX,transY),d);
[h1,w1,d1] = size(imageEuc);
% We ask for the rotation angle
angle = input('  Please introduce the rotation angle (grads): (ex:
15)', 's');
if isempty(angle)
    angle = '15';
end
angle = str2num(angle);
angle = -angle;

% We use homogeneous coordinates
H = [cosd(angle) -sind(angle) transX; sind(angle) cosd(angle) transY;
0 0 1];

for i=1:h
    for j=1:w
    % We multiply by a homogeneous vector x = (i,j,1) T
    x1 = floor(H(1,1)*i + H(1,2)*j + H(1,3)*1);
    y1 = floor(H(2,1)*i + H(2,2)*j + H(2,3)*1);
    if((x1>0)&&(y1>0))&&((x1<h1)&&(y1<w1))  imageEuc(x1,y1,:) =
lena(i,j,:);
    end
```

```
    end
end

% We normalize the image, to show the correct output
imageEuc= imageEuc/max(imageEuc(:));

% We show the result
figure(2)
subplot(1,2,1), subimage(lena)
title('Lena - Original')
subplot(1,2,2), subimage(imageEuc)
title('Lena - Euclidean transformation')

%% Similarity transformation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fprintf('+ Similarity transformation example\n');
fprintf('  We will consider the parameters from before\n');

% We ask for the scaling factor "s"
s = input('  Please introduce the scaling factor (ex.: 0.5)', 's');
if isempty(s)
    s = '2';
end
s = str2num(s);
imageSim = zeros(diag + max(transX,transY), diag +
max(transX,transY),d);

% We use homogeneous coordinates
H = [s*cosd(angle) -s*sind(angle) transX; s*sind(angle) s*cosd(angle)
transY; 0 0 1];

for i=1:h
    for j=1:w
    % We multiply by a homogeneous vector x = (i,j,1) T
    x1 = floor(H(1,1)*i + H(1,2)*j + H(1,3)*1);
    y1 = floor(H(2,1)*i + H(2,2)*j + H(2,3)*1);
    if((x1>0)&&(y1>0))&&((x1<h1)&&(y1<w1))  imageSim(x1,y1,:) =
lena(i,j,:);
    end
    end
end

% We normalize the image, to show the correct output
imageSim= imageSim/max(imageSim(:));

% We show the result
figure(3)
subplot(1,2,1), subimage(lena)
title('Lena - Original')
subplot(1,2,2), subimage(imageSim)
title('Lena - Similarity transformation')

%% Affine transformation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fprintf('+ Affine transformation example\n');
imageAff = zeros(3*diag + max(transX,transY), 3*diag +
max(transX,transY),d);
[h1,w1,d1] = size(imageAff);
```

```
% We use homogeneous coordinates
H = [rand(1) rand(1) transX; rand(1) rand(1) transY; 0 0 1];

for i=1:h
    for j=1:w
    % We multiply by a homogeneous vector x = (i,j,1) T
    x1 = floor(H(1,1)*i + H(1,2)*j + H(1,3)*1);
    y1 = floor(H(2,1)*i + H(2,2)*j + H(2,3)*1);
    if((x1>0)&&(y1>0))&&((x1<h1)&&(y1<w1))  imageAff(x1,y1,:) =
lena(i,j,:);
    end
    end
end

% We normalize the image, to show the correct output
imageAff= imageAff/max(imageAff(:));

% We show the result
figure(4)
subplot(1,2,1), subimage(lena)
title('Lena - Original')
subplot(1,2,2), subimage(imageAff)
title('Lena - Affine transformation')
```

2. BILATERAL FILTER

2.1 EX1.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%            EXERCISE  - PART I
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

echo off;
clc
clear

%% _____________________________________________ PART C ANSWER
%
% For Salt and Pepper noise-d image:
% The Gaussian filter/smoothing (also mean filter) don't do a
% good job because corrupted pixel values vary significantly from
their
% neighbours.  The Median filter does a good job of removing those
% corrupted pixels while preserving edges and details better.
%
% For Gaussian noise-d image:
% Gaussian filter/smoothing does a better job than Median filter, the
% Median filter does smoothing but with a slight degradation in image
% quality.  It is better suited for cases where you have outlier
pixels
% (e.g. salt and pepper noise).
%


tic
fprintf('\n*** Exercise 2 - Part 1 ***');
```

```
fprintf('\n- Please wait around 30seconds');
fprintf('\n-(1/9) Reading Lena');
lena = imread('lena.gif');



% Noises
fprintf('\n-(2/9) Salt&Pepper Lena');
lenaSP = student_salt_pepper(lena,0.05);

fprintf('\n-(3/9) Gaussian noise Lena');
lenaG = G_noise(lena,0.2);

% Median filters
fprintf('\n-(4/9) Filtering Original Lena - Median Filter');
lena_median_o = student_median_filter(lena,5,5);


fprintf('\n-(5/9) Filtering Salt and Pepper Lena - Median Filter');
lena_median_sp = student_median_filter(lenaSP,5,5);


fprintf('\n-(6/9) Filtering Gaussian Lena - Median Filter');
lena_median_g = student_median_filter(lenaG,5,5);


% Gaussian filters
fprintf('\n-(7/9) Filtering Original Lena - Gaussian Filter');
lena_g_o = student_gaussian(lena,1);


fprintf('\n-(8/9) Filtering Salt and Pepper Lena - Gaussian Filter');
lena_g_sp = student_gaussian(lenaSP,1);


fprintf('\n-(9/9) Filtering Gaussian Lena - Gaussian Filter\n');
lena_g_g = student_gaussian(lenaG,1);
toc

% Plotting

figure(1)
subplot(1,3,1), subimage(lena)
title('Lena Original')
subplot(1,3,2), subimage(lenaSP)
title('Lena Salt-Pepper Noise')
subplot(1,3,3), subimage(lenaG)
title('Lena Gaussian Noise')

figure(2)
subplot(1,3,1), subimage(lena_median_o)
title('Lena Original + MEDIAN FILTER')
subplot(1,3,2), subimage(lena_median_sp)
title('Lena Salt-Pepper Noise + MEDIAN FILTER')
subplot(1,3,3), subimage(lena_median_g)
title('Lena Gaussian Noise + MEDIAN FILTER')
```

```
figure(3)
subplot(1,3,1), subimage(lena_g_o)
title('Lena Original + GAUSSIAN FILTER')
subplot(1,3,2), subimage(lena_g_sp)
title('Lena Salt-Pepper Noise + GAUSSIAN FILTER')
subplot(1,3,3), subimage(lena_g_g)
title('Lena Gaussian Noise + GAUSSIAN FILTER')


% End
```

2.3.2 G_noise.m

```
% file G_noise.m
function [NewImage] = G_noise(image,var)

[height,width] = size(image);
NewImage = [image];
% var=0.2;
RandNoise = var*randn(height,width);
for i=1:height
    for j=1:width
       NewImage(i,j)=image(i,j)+(RandNoise(i,j)*100);
    end
end
```

2.3.3 salt_pepper.m

```
function [newImg] = salt_pepper(image,parameter)

[height,width] = size(image);
noiseRand = rand( height,width )

for i=1:height
    for j=1:width
        if(noiseRand(i,j)<parameter)
        newImg(i,j) = 0;
        else
            if(noiseRand(i,j)>(1-parameter))
            newImg(i,j) = 255;
            else

        newImg(i,j) = image(i,j);
            end
        end

    end
end

 % imshow(newImg);
End
```

2.3.4 student_bilateral.m

```
function [imgOut] = student_bilateral(imgIn,sigma)

    [height,width] = size(imgIn);
    imgOut = zeros(3*sigma,3*sigma);
    subImg = zeros(3*sigma,3*sigma);
```

```
    vecImg = zeros(9*sigma);
    half = (9*sigma+1)/2;

    border = (3*sigma - 1)/2
    auxImg = padarray(imgIn,[border border],'replicate','both');

    % Create waitbar.
    h = waitbar(0,'Applying bilateral filter...');
    set(h,'Name','Bilateral Filter Progress');

    for i = 1+border:height+border
        for j = 1+border:width+border

        % We select the submatrix to be ordered
        subImg(:,:) = auxImg((i-border:i+border),(j-border:j+border));
        % We extract the element of the middle (x in the formula)
        midEl = subImg(half);

        hx = 0; % The global bilateral function
        division_c = eps; % Very small amount, almost zero

        for z = 1:3*sigma
            for k = 1:3*sigma

            % index are not exactly like that, but the distance
remains the
            % same in a translation, so we don't care
            c = exp((-1/2)*((z-border)^2+(k-border)^2/(9*sigma^2)));
            s = exp((-1/2)*((subImg(z,k)-midEl)^2)/(9*sigma^2));
            division_c = division_c + c*s;
            hx = hx + subImg(z,k)*c*s;

            end
        end

        imgOut(i-border,j-border) = hx/division_c;


        end
    waitbar(i/height)
    end

    imgOut = imgOut/max(imgOut(:));
    close(h);
end
```

2.3.5 student_convolution.m

```
function [ conImg ] = student_convolution( image, mask  )
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here

% We get image & mask dimensions
[m_image, n_image] = size(image);
[m_mask, n_mask] = size(mask);

conImg = double(zeros(m_image,n_image));       % Preallocate matrix
conImg = conImg / 1.0;
```

```
% convolutedImg init for faster computation
%convolutedImg =
m_lim = (m_mask-1)/2;
n_lim = (n_mask-1)/2;
jmvalue = -3.45;
%coef = sum(abs(mask(:)));

for x=1:m_image
    for y = 1:n_image
        if (x > m_lim)&&(x<=(m_image-m_lim))
            if (y > n_lim)&&(y<=(n_image-n_lim))

                for i=1:m_mask
                    for j=1:n_mask

                        %conImg(x,y) = conImg(x,y)+
(1/coef)*mask(i,j)*image(m_lim+x-i+1,n_lim+y-j+1);



                            jmvalue =
double(mask(i,j))*double(image((m_lim+x-i+1),(n_lim+y-j+1))) ;

                            %class(mask(i,j))
                            %image((m_lim+x-i+1),(n_lim+y-j+1))
                            %class(jmvalue)

                            conImg(x,y) = double(conImg(x,y))+
double(jmvalue);
                           % fprintf('i:%d j:%d %d %d real%d mask%d
final%d   conImg(2,2)%d\n',i,j,(m_lim+x-i+1),(n_lim+y-
j+1),image((m_lim+x-i+1),(n_lim+y-
j+1)),mask(i,j),jmvalue,conImg(5,23))



                    end
                end

            else

            conImg(x,y)=0;
            end
        end
    end
end

conImg =conImg/max(conImg(:));


end
```

2.3.6 student_gaussian.m

```
function [gi] = student_gaussian( image, sigma  )
```

```
% We get image & mask dimensions
[m_image, n_image] = size(image);
gi = zeros(m_image,n_image);        % Preallocate result matrix

% We implement the mask, with general dims
mask = ones(3*sigma,3*sigma);             % Preallocate 2dmask matrix

k1 = 2*pi*sigma^2;
k2 = ((3*sigma+1)/2);
for i=1:3*sigma
    for j=1:3*sigma
        mask(i,j) = exp((-1/2)*((i-k2)^2+(j-k2)^2)/sigma^2)/k1;
    end
end


%2d convolution
gi = student_convolution( image, mask );
% gi=gi/max(gi(:));


end
```

2.3.7 student_median_filter.m

```
function [newImg] = student_median_filter(image,m,n)

    [height,width] = size(image);

    mlim = (m-1)/2;
    nlim = (n-1)/2;

    newImg = zeros(height,width);

    auxImg = padarray(image,[mlim nlim],'symmetric','both');
    subImg = zeros(m,n);

    % Create waitbar.
    h = waitbar(0,'Applying median filter...');
    set(h,'Name','Median Filter Progress');

    for i = 1+mlim:height+mlim
        for j = 1+nlim:width+nlim

            % We select the submatrix to be ordered
            subImg(1:m,1:n) = auxImg((i-mlim:i+mlim),(j-nlim:j+nlim));

            % We sort the submatrix in auxArray with the bubble method
            auxArray = student_bubble_sort(subImg);

            % We insert the middle element in newImg
            newImg(i-mlim,j-nlim) = auxArray((m*n+1)/2);

        end
    waitbar(i/height)
```

```
    end

    % As newImg class is double, and double range should be [0,1]
    % we normalize
    newImg(1:height,1:width) =
newImg(1:height,1:width)/max(newImg(:));
    close(h);
end

function init_array = student_bubble_sort(matrix)

    %ref: http://en.wikipedia.org/wiki/Bubble_sort

    init_array = matrix(:);
    n = length(init_array);

    for i = 2:n
        for j = 1:n-1
            if (init_array(j) > init_array(j+1))
                aux_val = init_array(j);
                init_array(j) = init_array(j+1);
                init_array(j+1) = aux_val;
            end
        end
    end

end
```

2.3.8 student_salt_pepper.m

```
function [newImg] = student_salt_pepper(image,parameter)

    [height,width] = size(image);
    noiseRand = rand(height,width);
    newImg = zeros(height,width);

    for i=1:height
        for j=1:width
            if(noiseRand(i,j)<parameter)
            newImg(i,j) = 0;
            else
                if(noiseRand(i,j)>(1-parameter))
                newImg(i,j) = 255;
                else
                newImg(i,j) = image(i,j);
                end
            end

        end
    end
    maxi = max(newImg(:));
    newImg(1:end,1:end) = (newImg(1:end,1:end)/maxi);

end
```

2.3.9 EXPART2.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%            EXERCISE 2 - PART II
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


echo off;
clc
clear


fprintf('\n*** Exercise 2 - Part 1 ***');
fprintf('\n- Please wait');
lena = imread('lena.gif');


% !! The bilateral filter, eliminates textures but not shapes.

% Bilateral filtering

lenaBilateral1 = student_bilateral(lena,1);
lenaBilateral5 = student_bilateral(lena,5);
lenaBilateral10 = student_bilateral(lena,9);

% Plotting

figure(1)
subplot(2,2,1), subimage(lena)
title('Lena - Original')
subplot(2,2,2), subimage(lenaBilateral1)
title('Bilateral - sigma = 1')
subplot(2,2,3), subimage(lenaBilateral5)
title('Bilateral - sigma = 5')
subplot(2,2,4), subimage(lenaBilateral10)
title('Bilateral - sigma = 10')


% The main difference between both filters is ... seems like for
% same sigma the gaussian is much more blurry than the bilateral.
hard to
% see other differences in the small images.


% Gaussian filtering
fprintf('\n- Please wait');
lenaG1 = student_gaussian(lena,1);
lenaG5 = student_gaussian(lena,5);
lenaG10 = student_gaussian(lena,9);


% Plotting

figure(2)
subplot(1,3,1), subimage(lenaG1)
title('Gaussian smoothing - sigma = 1')
```

```
subplot(1,3,2), subimage(lenaG5)
title('Gaussian smoothing - sigma = 5')
subplot(1,3,3), subimage(lenaG10)
title('Gaussian smoothing - sigma = 10')


% It is not possible to implement the bilateral filter w
% convolution masks because this is a non-linear filter !
```

3. KINECT FUSION

Filesystem tree (parenthesis indiquates a folder)

Dependencies: PCL and PCL own dependencies, CUDA libraries

(Root)

- Kinfu.cpp
- Internal.h
- **(root/cuda)**
  - Bylateral_pyrdown.cu
  - Coresp.cu
  - Device.h
  - Estimate_combined.cu
  - Estimate_transform.cu
  - Extract.cu
  - Extract_shared_buf.cu_backup
  - Image_generator.cu
  - Maps.cu
  - Normal_eigen.cu
  - Ray_caster.cu
  - Tsdf_volume.cu

Kinfu.cpp

```
/*
 * Software License Agreement (BSD License)
 *
 *  Point Cloud Library (PCL) - www.pointclouds.org
 *  Copyright (c) 2011, Willow Garage, Inc.
 *
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of Willow Garage, Inc. nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
```

```
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 */

#include <iostream>
#include <algorithm>

#include "pcl/common/time.h"
#include "pcl/gpu/kinfu/kinfu.h"
#include "internal.h"

#include <Eigen/Core>
#include <Eigen/SVD>
#include <Eigen/Cholesky>
#include <Eigen/Geometry>
#include <Eigen/LU>

#ifdef HAVE_OPENCV
#include <opencv2/opencv.hpp>
#include <opencv2/gpu/gpu.hpp>
#include "pcl/gpu/utils/timers_opencv.hpp"
#endif

using namespace std;
using namespace pcl::device;
using namespace pcl::gpu;

using Eigen::AngleAxisf;
using Eigen::Array3f;

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
template<class D, class Matx> D&
device_cast (Matx& matx)
{
  return (*reinterpret_cast<D*>(matx.data ()));
}

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
pcl::gpu::KinfuTracker::KinfuTracker (int rows, int cols) : rows_(rows),
cols_(cols), global_time_(0)
{
  rmats_.reserve (30000);
  tvecs_.reserve (30000);

  setDepthIntrinsics (525.f, 525.f);
  setVolumeSize (Vector3f::Constant (3000));

  init_Rcam_ = Eigen::Matrix3f::Identity ();// * AngleAxisf(-30.f/180*3.1415926,
Vector3f::UnitX());
  init_tcam_ = volume_size_ * 0.5f - Vector3f (0, 0, volume_size_ (2) / 2 *
1.2f);

  const int iters[] = {10, 5, 4};
  std::copy (iters, iters + LEVELS, icp_iterations_);

  const float default_distThres = 100; //mm
```

```
  const float default_angleThres = sin (20.f * 3.14159254f / 180.f);
  const float default_tranc_dist = 30; //mm

  setIcpCorespFilteringParams (default_distThres, default_angleThres);
  setTrancationDistance (default_tranc_dist);

  allocateBufffers (rows, cols);
  reset ();
}

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
void
pcl::gpu::KinfuTracker::setDepthIntrinsics (float fx, float fy, float cx, float
cy)
{
  fx_ = fx;
  fy_ = fy;
  cx_ = (cx == -1) ? cols_/2 : cx;
  cy_ = (cy == -1) ? rows_/2 : cy;
}

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
void
pcl::gpu::KinfuTracker::setVolumeSize (const Eigen::Vector3f& volume_size)
{
  volume_size_ = volume_size;
}

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
void
pcl::gpu::KinfuTracker::setInitalCameraPose (const Eigen::Affine3f& pose)
{
  init_Rcam_ = pose.rotation ();
  init_tcam_ = pose.translation ();
}

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
void
pcl::gpu::KinfuTracker::setTrancationDistance (float distance)
{
  float cx = volume_size_ (0) / VOLUME_X;
  float cy = volume_size_ (1) / VOLUME_Y;
  float cz = volume_size_ (2) / VOLUME_Z;

  tranc_dist_ = max (distance, 2.1f * max (cx, max (cy, cz)));
}

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
void
pcl::gpu::KinfuTracker::setIcpCorespFilteringParams (float distThreshold, float
sineOfAngle)
{
  distThres_  = distThreshold; //mm
  angleThres_ = sineOfAngle;
}
```

```
////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////
Eigen::Vector3f
pcl::gpu::KinfuTracker::getVolumeSize () const
{
  return (volume_size_);
}

////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////
int
pcl::gpu::KinfuTracker::cols ()
{
  return (cols_);
}

////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////
int
pcl::gpu::KinfuTracker::rows ()
{
  return (rows_);
}

////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////
void
pcl::gpu::KinfuTracker::reset()
{
  if (global_time_)
    cout << "Reset" << endl;

  global_time_ = 0;
  rmats_.clear ();
  tvecs_.clear ();

  rmats_.push_back (init_Rcam_);
  tvecs_.push_back (init_tcam_);

  device::initVolume<volume_elem_type> (volume_);
}

////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////
void
pcl::gpu::KinfuTracker::allocateBufffers (int rows, int cols)
{
  volume_.create (device::VOLUME_Y * device::VOLUME_Z, device::VOLUME_X);

  depths_curr_.resize (LEVELS);
  vmaps_g_curr_.resize (LEVELS);
  nmaps_g_curr_.resize (LEVELS);

  vmaps_g_prev_.resize (LEVELS);
  nmaps_g_prev_.resize (LEVELS);

  vmaps_curr_.resize (LEVELS);
  nmaps_curr_.resize (LEVELS);

  coresps_.resize (LEVELS);

  for (int i = 0; i < LEVELS; ++i)
```

```
  {
    int pyr_rows = rows >> i;
    int pyr_cols = cols >> i;

    depths_curr_[i].create (pyr_rows, pyr_cols);

    vmaps_g_curr_[i].create (pyr_rows*3, pyr_cols);
    nmaps_g_curr_[i].create (pyr_rows*3, pyr_cols);

    vmaps_g_prev_[i].create (pyr_rows*3, pyr_cols);
    nmaps_g_prev_[i].create (pyr_rows*3, pyr_cols);

    vmaps_curr_[i].create (pyr_rows*3, pyr_cols);
    nmaps_curr_[i].create (pyr_rows*3, pyr_cols);

    coresps_[i].create (pyr_rows, pyr_cols);
  }

  depthRawScaled_.create (rows, cols);
  // see estimate tranform for the magic numbers
  gbuf_.create (27, 20*60);
  sumbuf_.create (27);
}

////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////
bool
pcl::gpu::KinfuTracker::operator() (const DepthMap& depth_raw)
{
  setTrancationDistance (tranc_dist_);
  device::Intr intr (fx_, fy_, cx_, cy_);
  {
    //ScopeTime time(">>> Bilateral, pyr-down-all, create-maps-all");
    //depth_raw.copyTo(depths_curr[0]);
    device::bilateralFilter (depth_raw, depths_curr_[0]);

    for (int i = 1; i < LEVELS; ++i)
      device::pyrDown (depths_curr_[i-1], depths_curr_[i]);

    for (int i = 0; i < LEVELS; ++i)
    {
      device::createVMap (intr(i), depths_curr_[i], vmaps_curr_[i]);
      //device::createNMap(vmaps_curr_[i], nmaps_curr_[i]);
      computeNormalsEigen (vmaps_curr_[i], nmaps_curr_[i]);
    }
    pcl::device::sync ();
  }

  //can't perform more on first frame
  if (global_time_ == 0)
  {
    Matrix3frm init_Rcam = rmats_[0]; //  [Ri|ti] - pos of camera, i.e.
    Vector3f   init_tcam = tvecs_[0]; //  transform from camera to global coo
space for (i-1)th camera pose

    Mat33&  device_Rcam = device_cast<Mat33> (init_Rcam);
    float3& device_tcam = device_cast<float3>(init_tcam);

    Matrix3frm init_Rcam_inv = init_Rcam.inverse ();
    Mat33&   device_Rcam_inv = device_cast<Mat33> (init_Rcam_inv);
    float3 device_volume_size = device_cast<float3>(volume_size_);
```

```
    //integrateTsdfVolume(depth_raw, intr, device_volume_size, device_Rcam_inv,
device_tcam, tranc_dist, volume_);
    integrateVolume (depth_raw, intr, device_volume_size, device_Rcam_inv,
device_tcam, tranc_dist_, volume_, depthRawScaled_);

    for (int i = 0; i < LEVELS; ++i)
      device::tranformMaps (vmaps_curr_[i], nmaps_curr_[i], device_Rcam,
device_tcam, vmaps_g_prev_[i], nmaps_g_prev_[i]);

    ++global_time_;
    return (false);
  }


////////////////////////////////////////////////////////////////////////////
//////////
  // Iterative Closest Point
  Matrix3frm Rprev = rmats_[global_time_ - 1]; //  [Ri|ti] - pos of camera, i.e.
  Vector3f   tprev = tvecs_[global_time_ - 1]; //  tranfrom from camera to global
coo space for (i-1)th camera pose
  Matrix3frm Rprev_inv = Rprev.inverse (); //Rprev.t();

  //Mat33&  device_Rprev     = device_cast<Mat33> (Rprev);
  Mat33&  device_Rprev_inv = device_cast<Mat33> (Rprev_inv);
  float3& device_tprev     = device_cast<float3> (tprev);

  Matrix3frm Rcurr = Rprev; // tranform to global coo for ith camera pose
  Vector3f   tcurr = tprev;
  {
    //ScopeTime time("icp-all");
    for (int level_index = LEVELS-1; level_index>=0; --level_index)
    {
      int iter_num = icp_iterations_[level_index];

      MapArr& vmap_curr = vmaps_curr_[level_index];
      MapArr& nmap_curr = nmaps_curr_[level_index];

      //MapArr& vmap_g_curr = vmaps_g_curr_[level_index];
      //MapArr& nmap_g_curr = nmaps_g_curr_[level_index];

      MapArr& vmap_g_prev = vmaps_g_prev_[level_index];
      MapArr& nmap_g_prev = nmaps_g_prev_[level_index];

      //CorespMap& coresp = coresps_[level_index];

      for (int iter = 0; iter < iter_num; ++iter)
      {
        Mat33&  device_Rcurr = device_cast<Mat33> (Rcurr);
        float3& device_tcurr = device_cast<float3>(tcurr);

        Eigen::Matrix<float, 6, 6, Eigen::RowMajor> A;
        Eigen::Matrix<float, 6, 1> b;
#if 0
        device::tranformMaps(vmap_curr, nmap_curr, device_Rcurr, device_tcurr,
vmap_g_curr, nmap_g_curr);
        findCoresp(vmap_g_curr, nmap_g_curr, device_Rprev_inv, device_tprev,
intr(level_index), vmap_g_prev, nmap_g_prev, distThres_, angleThres_, coresp);
        device::estimateTransform(vmap_g_prev, nmap_g_prev, vmap_g_curr, coresp,
gbuf_, sumbuf_, A.data(), b.data());

        //cv::gpu::GpuMat ma(coresp.rows(), coresp.cols(), CV_32S, coresp.ptr(),
coresp.step());
```

```
        //cv::Mat cpu;
        //ma.download(cpu);
        //cv::imshow(names[level_index] + string(" --- coresp white == -1"), cpu
== -1);
#else
        estimateCombined (device_Rcurr, device_tcurr, vmap_curr, nmap_curr,
device_Rprev_inv, device_tprev, intr (level_index),
                          vmap_g_prev, nmap_g_prev, distThres_, angleThres_,
gbuf_, sumbuf_, A.data (), b.data ());
#endif
        //checking nullspace
        float det = A.determinant ();

        if (fabs (det) < 1e-15 || !pcl::device::valid_host (det))
        {
          if (!valid_host (det)) cout << "qnan" << endl;

          reset ();
          return (false);
        }
        //float maxc = A.maxCoeff();

        Eigen::Matrix<float, 6, 1> result = A.llt ().solve (b);
        //Eigen::Matrix<float, 6, 1> result = A.jacobiSvd(ComputeThinU |
ComputeThinV).solve(b);

        float alpha = result (0);
        float beta  = result (1);
        float gamma = result (2);

        Eigen::Matrix3f Rinc = (Eigen::Matrix3f)AngleAxisf (gamma,
Vector3f::UnitZ ()) * AngleAxisf (beta, Vector3f::UnitY ()) * AngleAxisf (alpha,
Vector3f::UnitX ());
        Vector3f tinc = result.tail<3> ();

        //compose
        tcurr = Rinc * tcurr + tinc;
        Rcurr = Rinc * Rcurr;
      }
    }
  }
  //save tranform
  rmats_.push_back (Rcurr);
  tvecs_.push_back (tcurr);


////////////////////////////////////////////////////////////////////////////
//////////
  // Volume integration
  float3 device_volume_size = device_cast<float3> (volume_size_);

  Matrix3frm Rcurr_inv = Rcurr.inverse ();
  Mat33&  device_Rcurr_inv = device_cast<Mat33> (Rcurr_inv);
  float3& device_tcurr = device_cast<float3> (tcurr);
  {
    //ScopeTime time("tsdf");
    //integrateTsdfVolume(depth_raw, intr, device_volume_size, device_Rcurr_inv,
device_tcurr, tranc_dist, volume_);
    integrateVolume (depth_raw, intr, device_volume_size, device_Rcurr_inv,
device_tcurr, tranc_dist_, volume_, depthRawScaled_);
  }
```

```
////////////////////////////////////////////////////////////////////////////////
///////////
  // Ray casting
  Mat33& device_Rcurr = device_cast<Mat33> (Rcurr);
  {
    //ScopeTime time("ray-cast-all");
    raycast (intr, device_Rcurr, device_tcurr, tranc_dist_, device_volume_size,
volume_, vmaps_g_prev_[0], nmaps_g_prev_[0]);
    for (int i = 1; i < LEVELS; ++i)
    {
      resizeVMap (vmaps_g_prev_[i-1], vmaps_g_prev_[i]);
      resizeNMap (nmaps_g_prev_[i-1], nmaps_g_prev_[i]);
    }
    pcl::device::sync ();
  }

  ++global_time_;
  return (true);
}

////////////////////////////////////////////////////////////////////////////////
///////////////////////////////////
Eigen::Affine3f
pcl::gpu::KinfuTracker::getCameraPose (int time)
{
  if (time > (int)rmats_.size () || time < 0)
    time = rmats_.size () - 1;

  Eigen::Affine3f aff;
  aff.linear () = rmats_[time];
  aff.translation () = tvecs_[time];
  return (aff);
}

////////////////////////////////////////////////////////////////////////////////
///////////////////////////////////
void
pcl::gpu::KinfuTracker::getImage (View& view) const
{
  getImage (view, volume_size_ * (-3.f));
}

////////////////////////////////////////////////////////////////////////////////
///////////////////////////////////
void
pcl::gpu::KinfuTracker::getImage (View& view_arg, const Eigen::Vector3f&
light_source_pose) const
{
  device::LightSource light;
  light.number = 1;
  light.pos[0] = device_cast<const float3>(light_source_pose);

  view_arg.create (rows_, cols_);
  generateImage (vmaps_g_prev_[0], nmaps_g_prev_[0], light, view_arg);
}

////////////////////////////////////////////////////////////////////////////////
///////////////////////////////////
void
pcl::gpu::KinfuTracker::getLastFrameCloud (DeviceArray2D<PointType>& cloud) const
{
```

```
  cloud.create (rows_, cols_);
  DeviceArray2D<float4>& c = (DeviceArray2D<float4>&)cloud;
  device::convert (vmaps_g_prev_[0], c);
}

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
void
pcl::gpu::KinfuTracker::getLastFrameNormals (DeviceArray2D<NormalType>& normals)
const
{
  normals.create (rows_, cols_);
  DeviceArray2D<float8>& n = (DeviceArray2D<float8>&)normals;
  device::convert (nmaps_g_prev_[0], n);
}

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
void
pcl::gpu::KinfuTracker::getCloudFromVolumeHost (PointCloud<PointType>& cloud,
bool connected26)
{
  int cols;
  std::vector<int> volume_host;
  volume_.download (volume_host, cols);

  cloud.points.clear ();
  cloud.points.reserve (10000);

  const int DIVISOR = 32767; // SHRT_MAX;

#define FETCH(x, y, z) volume_host[(x) + (y) * VOLUME_X + (z) * VOLUME_Y *
VOLUME_X]

  Array3f cell_size = volume_size_.array () / Array3f (VOLUME_X, VOLUME_Y,
VOLUME_Z);

  for (int x = 1; x < VOLUME_X-1; ++x)
  {
    for (int y = 1; y < VOLUME_X-1; ++y)
    {
      for (int z = 0; z < VOLUME_Z-1; ++z)
      {
        int tmp = FETCH (x, y, z);
        int W = reinterpret_cast<short2*>(&tmp)->y;
        int F = reinterpret_cast<short2*>(&tmp)->x;

        if (W == 0 || F == DIVISOR)
          continue;

        Vector3f V = ((Array3f(x, y, z) + 0.5f) * cell_size).matrix ();

        if (connected26)
        {
          int dz = 1;
          for (int dy = -1; dy < 2; ++dy)
            for (int dx = -1; dx < 2; ++dx)
            {
              int tmp = FETCH (x+dx, y+dy, z+dz);

              int Wn = reinterpret_cast<short2*>(&tmp)->y;
              int Fn = reinterpret_cast<short2*>(&tmp)->x;
```

```
              if (Wn == 0 || Fn == DIVISOR)
                continue;

              if ((F > 0 && Fn < 0) || (F < 0 && Fn > 0))
              {
                Vector3f Vn = ((Array3f (x+dx, y+dy, z+dz) + 0.5f) *
cell_size).matrix ();
                Vector3f point = (V * abs (Fn) + Vn * abs (F)) / (abs (F) + abs
(Fn));

                pcl::PointXYZ xyz;
                xyz.x = point (0);
                xyz.y = point (1);
                xyz.z = point (2);

                cloud.points.push_back (xyz);
              }
            }
          dz = 0;
          for (int dy = 0; dy < 2; ++dy)
            for (int dx = -1; dx < dy * 2; ++dx)
            {
              int tmp = FETCH (x+dx, y+dy, z+dz);

              int Wn = reinterpret_cast<short2*>(&tmp)->y;
              int Fn = reinterpret_cast<short2*>(&tmp)->x;
              if (Wn == 0 || Fn == DIVISOR)
                continue;

              if ((F > 0 && Fn < 0) || (F < 0 && Fn > 0))
              {
                Vector3f Vn = ((Array3f (x+dx, y+dy, z+dz) + 0.5f) *
cell_size).matrix ();
                Vector3f point = (V * abs(Fn) + Vn * abs(F))/(abs(F) + abs (Fn));

                pcl::PointXYZ xyz;
                xyz.x = point (0);
                xyz.y = point (1);
                xyz.z = point (2);

                cloud.points.push_back (xyz);
              }
            }
        }
        else /* if (connected26) */
        {
          for (int i = 0; i < 3; ++i)
          {
            int ds[] = {0, 0, 0};
            ds[i] = 1;

            int dx = ds[0];
            int dy = ds[1];
            int dz = ds[2];

            int tmp = FETCH (x+dx, y+dy, z+dz);

            int Wn = reinterpret_cast<short2*>(&tmp)->y;
            int Fn = reinterpret_cast<short2*>(&tmp)->x;
            if (Wn == 0 || Fn == DIVISOR)
              continue;
```

```
            if ((F > 0 && Fn < 0) || (F < 0 && Fn > 0))
            {
              Vector3f Vn = ((Array3f (x+dx, y+dy, z+dz) + 0.5f) *
cell_size).matrix ();
              Vector3f point = (V * abs (Fn) + Vn * abs (F)) / (abs (F) + abs
(Fn));

              pcl::PointXYZ xyz;
              xyz.x = point (0);
              xyz.y = point (1);
              xyz.z = point (2);

              cloud.points.push_back (xyz);
            }
          }
        } /* if (connected26) */
      }
    }
  }
#undef FETCH
  cloud.width  = (int)cloud.points.size ();
  cloud.height = 1;
}

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
pcl::gpu::DeviceArray<pcl::gpu::KinfuTracker::PointType>
pcl::gpu::KinfuTracker::getCloudFromVolume (DeviceArray<PointType>& cloud_buffer)
{
  if (cloud_buffer.empty ())
    cloud_buffer.create (DEFAULT_VOLUME_CLOUD_BUFFER_SIZE);

  float3 device_volume_size = device_cast<float3> (volume_size_);
  size_t size = device::extractCloud (volume_, device_volume_size, cloud_buffer);
  return (DeviceArray<PointType> (cloud_buffer.ptr (), size));
}

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
void
pcl::gpu::KinfuTracker::getNormalsFromVolume (const DeviceArray<PointType>&
cloud, DeviceArray<PointType>& normals) const
{
  normals.create (cloud.size ());
  const float3 device_volume_size = device_cast<const float3> (volume_size_);
  device::extractNormals (volume_, device_volume_size, cloud,
(device::PointType*)normals.ptr ());
}

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
void
pcl::gpu::KinfuTracker::getNormalsFromVolume (const DeviceArray<PointType>&
cloud, DeviceArray<NormalType>& normals) const
{
  normals.create (cloud.size ());
  const float3 device_volume_size = device_cast<const float3> (volume_size_);
  device::extractNormals (volume_, device_volume_size, cloud,
(device::float8*)normals.ptr ());
}

void
```

```
pcl::gpu::KinfuTracker::getTsdfVolume( std::vector<float>& volume) const
{
  volume.resize(volume_.cols() * volume_.rows());
  volume_.download(&volume[0], volume_.cols() * sizeof(int));

  for(size_t i = 0; i < volume.size(); ++i)
  {
    float tmp = ((short2*)&volume[i])->x;
    volume[i] = tmp/DIVISOR;
  }
}
```

Internal.h

```
/*
 * Software License Agreement (BSD License)
 *
 *  Point Cloud Library (PCL) - www.pointclouds.org
 *  Copyright (c) 2011, Willow Garage, Inc.
 *
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of Willow Garage, Inc. nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 */

#ifndef PCL_KINFU_INTERNAL_HPP_
#define PCL_KINFU_INTERNAL_HPP_

#include "pcl/gpu/containers/device_array.hpp"
#include "pcl/gpu/utils/safe_call.hpp"

namespace pcl
{
  namespace device
  {
```

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////
    // Types
    typedef unsigned short ushort;
    typedef DeviceArray2D<float> MapArr;
    typedef DeviceArray2D<ushort> DepthMap;
    typedef float4 PointType;

    //Tsdf fixed point divisor (if old format is enabled)
    const int DIVISOR = 32767;     // SHRT_MAX;

    enum { VOLUME_X = 512, VOLUME_Y = 512, VOLUME_Z = 512 };

    /** \brief Camera intrinsics structure
      */
    struct Intr
    {
      float fx, fy, cx, cy;
      Intr () {}
      Intr (float fx_, float fy_, float cx_, float cy_) : fx(fx_), fy(fy_),
cx(cx_), cy(cy_) {}

      Intr operator()(int level_index) const
      {
        int div = 1 << level_index;
        return (Intr (fx / div, fy / div, cx / div, cy / div));
      }
    };

    /** \brief 3x3 Matrix for device code
      */
    struct Mat33
    {
      float3 data[3];
    };

    /** \brief Light source collection
      */
    struct LightSource
    {
      float3 pos[1];
      int number;
    };


////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////
    // Maps

    /** \brief Perfoms bilateral filtering of disparity map
      * \param[in] src soruce map
      * \param[out] dst output map
      */
    void
    bilateralFilter (const DepthMap& src, DepthMap& dst);
    /** \brief Computes depth pyramid
      * \param[in] src source
      * \param[out] dst destination
      */
    void
    pyrDown (const DepthMap& src, DepthMap& dst);
```

```
    /** \brief Computes vertex map
      * \param[in] intr depth camera intrinsics
      * \param[in] depth depth
      * \param[out] vmap vertex map
      */
    void
    createVMap (const Intr& intr, const DepthMap& depth, MapArr& vmap);
    /** \brief Computes normal map using cross product
      * \param[in] vmap vertex map
      * \param[out] nmap normal map
      */
    void
    createNMap (const MapArr& vmap, MapArr& nmap);
    /** \brief Computes normal map using Eigen/PCA approach
      * \param[in] vmap vertex map
      * \param[out] nmap normal map
      */
    void
    computeNormalsEigen (const MapArr& vmap, MapArr& nmap);

    /** \brief Performs affine tranform of vertex and normal maps
      * \param[in] vmap_src source vertex map
      * \param[in] nmap_src source vertex map
      * \param[in] Rmat Rotation mat
      * \param[in] tvec translation
      * \param[out] vmap_dst destination vertex map
      * \param[out] nmap_dst destination vertex map
      */
    void
    tranformMaps (const MapArr& vmap_src, const MapArr& nmap_src, const Mat33&
Rmat, const float3& tvec, MapArr& vmap_dst, MapArr& nmap_dst);


////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
    //   ICP

    /** \brief (now it's exra code) Computes corespondances map
      * \param[in] vmap_g_curr current vertex map in global coo space
      * \param[in] nmap_g_curr current normals map in global coo space
      * \param[in] Rprev_inv inverse camera rotation at previous pose
      * \param[in] tprev camera translation at previous pose
      * \param[in] intr camera intrinsics
      * \param[in] vmap_g_prev previous vertex map in global coo space
      * \param[in] nmap_g_prev previous vertex map in global coo space
      * \param[in] distThres distance filtering threshold
      * \param[in] angleThres angle filtering threshold. Represents sine of angle
between normals
      * \param[out] coresp
      */
    void
    findCoresp (const MapArr& vmap_g_curr, const MapArr& nmap_g_curr, const
Mat33& Rprev_inv, const float3& tprev, const Intr& intr,
                const MapArr& vmap_g_prev, const MapArr& nmap_g_prev, float
distThres, float angleThres, PtrStepSz<short2> coresp);

    /** \brief (now it's exra code) Computation Ax=b for ICP iteration
      * \param[in] v_dst destination vertex map (previous frame cloud)
      * \param[in] n_dst destination normal map (previous frame normals)
      * \param[in] v_src source normal map (current frame cloud)
      * \param[in] coresp Corespondances
```

```
      * \param[out] gbuf temp buffer for GPU reduction
      * \param[out] mbuf ouput GPU buffer for matrix computed
      * \param[out] matrixA_host A
      * \param[out] vectorB_host b
      */
    void
    estimateTransform (const MapArr& v_dst, const MapArr& n_dst, const MapArr&
v_src, const PtrStepSz<short2>& coresp,
                       DeviceArray2D<float>& gbuf, DeviceArray<float>& mbuf,
float* matrixA_host, float* vectorB_host);


    /** \brief Computation Ax=b for ICP iteration
      * \param[in] Rcurr Rotation of current camera pose guess
      * \param[in] tcurr translation of current camera pose guess
      * \param[in] vmap_curr current vertex map in camera coo space
      * \param[in] nmap_curr current vertex map in camera coo space
      * \param[in] Rprev_inv inverse camera rotation at previous pose
      * \param[in] tprev camera translation at previous pose
      * \param[in] intr camera intrinsics
      * \param[in] vmap_g_prev previous vertex map in global coo space
      * \param[in] nmap_g_prev previous vertex map in global coo space
      * \param[in] distThres distance filtering threshold
      * \param[in] angleThres angle filtering threshold. Represents sine of angle
between normals
      * \param[out] gbuf temp buffer for GPU reduction
      * \param[out] mbuf ouput GPU buffer for matrix computed
      * \param[out] matrixA_host A
      * \param[out] vectorB_host b
      */
    void
    estimateCombined (const Mat33& Rcurr, const float3& tcurr, const MapArr&
vmap_curr, const MapArr& nmap_curr, const Mat33& Rprev_inv, const float3& tprev,
const Intr& intr,
                      const MapArr& vmap_g_prev, const MapArr& nmap_g_prev, float
distThres, float angleThres,
                      DeviceArray2D<float>& gbuf, DeviceArray<float>& mbuf,
float* matrixA_host, float* vectorB_host);


////////////////////////////////////////////////////////////////////////////////
///////////////////////////////////
    // TSDF volume functions

    //switch between two tsdf volume formats
    typedef short2 volume_elem_type;
    //typedef ushort2 volume_elem_type;

    /** \brief Perform tsdf volume initialization
      *  \param[out]
      */
    template<typename T>
    void
    initVolume(PtrStepSz<T> array);

    //first version
    /** \brief Performs Tsfg volume uptation (extra obsolete now)
      * \param[in] depth_raw Kinect depth image
      * \param[in] intr camera intrinsics
      * \param[in] volume_size size of volume in mm
      * \param[in] Rcurr_inv inverse rotation for current camera pose
      * \param[in] tcurr translation for current camera pose
```

```
      * \param[in] tranc_dist tsdf trancation distance
      * \param[in] volume tsdf volume to be updated
      */
    void
    integrateTsdfVolume (const PtrStepSz<ushort>& depth_raw, const Intr& intr,
const float3& volume_size,
                         const Mat33& Rcurr_inv, const float3& tcurr, float
tranc_dist, PtrStep<short2> volume);

    //second version
    /** \brief Function that integrates volume if volume element contains: 2
bytes for round(tsdf*SHORT_MAX) and 2 bytes for integer weight.
      * \param[in] depth_raw Kinect depth image
      * \param[in] intr camera intrinsics
      * \param[in] volume_size size of volume in mm
      * \param[in] Rcurr_inv inverse rotation for current camera pose
      * \param[in] tcurr translation for current camera pose
      * \param[in] tranc_dist tsdf trancation distance
      * \param[in] volume tsdf volume to be updated
      * \param[out] depthRawScaled Buffer for scaled depth along ray
      */
    void
    integrateTsdfVolume (const PtrStepSz<ushort>& depth_raw, const Intr& intr,
const float3& volume_size,
                         const Mat33& Rcurr_inv, const float3& tcurr, float
tranc_dist, PtrStep<short2> volume, DeviceArray2D<float>& depthRawScaled);

    //third version (half)
    /** \brief Function that integrates volume if volume element contains: 2
bytes for half-float(tsdf) and 2 bytes for integer weight.
      * \param[in] depth_raw Kinect depth image
      * \param[in] intr camera intrinsics
      * \param[in] volume_size size of volume in mm
      * \param[in] Rcurr_inv inverse rotation for current camera pose
      * \param[in] tcurr translation for current camera pose
      * \param[in] tranc_dist tsdf trancation distance
      * \param[in] volume tsdf volume to be updated
      * \param[out] depthRawScaled buffer for scaled depth along ray
      */
    void
    integrateTsdfVolume (const PtrStepSz<ushort>& depth_raw, const Intr& intr,
const float3& volume_size,
                         const Mat33& Rcurr_inv, const float3& tcurr, float
tranc_dist, PtrStep<ushort2> volume, DeviceArray2D<float>& depthRawScaled);

    // Dispatcher
    /** \brief Dispatched function for fast swithing between two tsdf volume
element formats
      * \param[in] depth Kinect depth image
      * \param[in] intr camera intrinsics
      * \param[in] volume_size size of volume in mm
      * \param[in] Rcurr_inv inverse rotation for current camera pose
      * \param[in] tcurr translation for current camera pose
      * \param[in] tranc_dist tsdf trancation distance
      * \param[in] volume  tsdf volume to be updated
      * \param[out] depthRawScaled buffer for scaled depth along ray
      */
    inline
    void
    integrateVolume (const PtrStepSz<ushort>& depth, const Intr& intr, const
float3& volume_size, const Mat33& Rcurr_inv, const float3& tcurr, float
tranc_dist,
```

```
                    DeviceArray2D<int>& volume, DeviceArray2D<float>&
depthRawScaled)
    {
      integrateTsdfVolume (depth, intr, volume_size, Rcurr_inv, tcurr,
tranc_dist, (PtrStep<volume_elem_type>) volume, depthRawScaled);
    }



////////////////////////////////////////////////////////////////////////////
///////////////////////////////////
    // Raycast and view generation
    /** \brief Generation vertex and normal maps from volume for current camera
pose
      * \param[in] intr camera intrinsices
      * \param[in] Rcurr current rotation
      * \param[in] tcurr current translation
      * \param[in] tranc_dist volume trancation distance
      * \param[in] volume_size volume size in mm
      * \param[in] volume tsdf volume
      * \param[out] vmap output vertex map
      * \param[out] nmap output normals map
      */
    void
    raycast (const Intr& intr, const Mat33& Rcurr, const float3& tcurr, float
tranc_dist, const float3& volume_size,
             const PtrStep<volume_elem_type>& volume, MapArr& vmap, MapArr&
nmap);

    /** \brief Renders 3D image of the scene
      * \param[in] vmap vetex map
      * \param[in] nmap normals map
      * \param[in] light poase of light source
      * \param[out] dst buffer where image is generated
      */
    void
    generateImage (const MapArr& vmap, const MapArr& nmap, const LightSource&
light, PtrStepSz<uchar3> dst);

    /** \brief Performs resize of vertex map to next pyramid level by averaging
each four points
      * \param[in] input vertext map
      * \param[out] output resized vertex map
      */
    void
    resizeVMap (const MapArr& input, MapArr& output);

    /** \brief Performs resize of vertex map to next pyramid level by averaging
each four normals
      * \param[in] input normal map
      * \param[out] output vertex map
      */
    void
    resizeNMap (const MapArr& input, MapArr& output);


////////////////////////////////////////////////////////////////////////////
///////////////////////////////////
    // Cloud extraction

    /** \brief Perofrm point cloud extraction from tsdf volumer
      * \param[in] volume tsdf volume
```

```
      * \param[in] volume_size size of the volume
      * \param[out] output buffer large enought to store point cloud
      * \return number of point stored to passed buffer
      */
    size_t
    extractCloud (const PtrStep<volume_elem_type>& volume, const float3&
volume_size, PtrSz<PointType> output);

    /** \brief Performs normals computation for given poins using tsdf volume
      * \param[in] volume tsdf volume
      * \param[in] volume_size volume size
      * \param[in] input points where normals are computed
      * \param[out] output normals. Could be float4 or float8. If for a point
normal can't be computed, such normal is marked as nan.
      */
    template<typename NormalType>
    void
    extractNormals (const PtrStep<volume_elem_type>& volume, const float3&
volume_size, const PtrSz<PointType>& input, NormalType* output);


////////////////////////////////////////////////////////////////////////////////
///////////////////////////////////
    // Utility
    struct float8 { float x, y, z, w, f1, f2, f3, f4; };

    /** \brief Conversion from SOA to AOS
      * \param[in] vmap SOA map
      * \param[out] output Array of 3D points. Can be float4 or float8.
      */
    template<typename T>
    void
    convert (const MapArr& vmap, DeviceArray2D<T>& output);

    /** \brief  Check for qnan (unused now)
      * \param[in] value
      */
    inline bool
    valid_host (float value)
    {
      return *reinterpret_cast<int*>(&value) != 0x7fffffff; //QNAN
    }

    /** \brief synchronizes CUDA execution */
    inline
    void
    sync () { cudaSafeCall (cudaDeviceSynchronize ()); }
  }
}

#endif /* PCL_KINFU_INTERNAL_HPP_ */
```

Bylateral_pyrdown.cu

```
/* * Software License Agreement (BSD License)
 *
 *  Point Cloud Library (PCL) - www.pointclouds.org
 *  Copyright (c) 2011, Willow Garage, Inc.
 *
 *  All rights reserved.
```

```
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of Willow Garage, Inc. nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 */

#include "device.hpp"

namespace pcl
{
  namespace device
  {
    const float sigma_color = 30;     //in mm
    const float sigma_space = 4.5;     // in pixels


////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
    __global__ void
    bilateralKernel (const PtrStepSz<ushort> src,
                     PtrStep<ushort> dst,
                     float sigma_space2_inv_half, float sigma_color2_inv_half)
    {
      int x = threadIdx.x + blockIdx.x * blockDim.x;
      int y = threadIdx.y + blockIdx.y * blockDim.y;

      if (x >= src.cols || y >= src.rows)
        return;

      const int R = 6;       //static_cast<int>(sigma_space * 1.5);
      const int D = R * 2 + 1;

      int value = src.ptr (y)[x];

      int tx = min (x - D / 2 + D, src.cols - 1);
      int ty = min (y - D / 2 + D, src.rows - 1);

      float sum1 = 0;
```

```
      float sum2 = 0;

      for (int cy = max (y - D / 2, 0); cy < ty; ++cy)
      {
        for (int cx = max (x - D / 2, 0); cx < tx; ++cx)
        {
          int tmp = src.ptr (cy)[cx];

          float space2 = (x - cx) * (x - cx) + (y - cy) * (y - cy);
          float color2 = (value - tmp) * (value - tmp);

          float weight = __expf (-(space2 * sigma_space2_inv_half + color2 *
sigma_color2_inv_half));

          sum1 += tmp * weight;
          sum2 += weight;
        }
      }

      int res = __float2int_rn (sum1 / sum2);
      dst.ptr (y)[x] = max (0, min (res, numeric_limits<short>::max ()));
    }


////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////
    __global__ void
    pyrDownKernel (const PtrStepSz<ushort> src, PtrStepSz<ushort> dst, float
sigma_color)
    {
      int x = blockIdx.x * blockDim.x + threadIdx.x;
      int y = blockIdx.y * blockDim.y + threadIdx.y;

      if (x >= dst.cols || y >= dst.rows)
        return;

      const int D = 5;

      int center = src.ptr (2 * y)[2 * x];

      int tx = min (2 * x - D / 2 + D, src.cols - 1);
      int ty = min (2 * y - D / 2 + D, src.rows - 1);
      int cy = max (0, 2 * y - D / 2);

      int sum = 0;
      int count = 0;

      for (; cy < ty; ++cy)
        for (int cx = max (0, 2 * x - D / 2); cx < tx; ++cx)
        {
          int val = src.ptr (cy)[cx];
          if (abs (val - center) < 3 * sigma_color)
          {
            sum += val;
            ++count;
          }
        }
      dst.ptr (y)[x] = sum / count;
    }
  }
}
```

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////
void
pcl::device::bilateralFilter (const DepthMap& src, DepthMap& dst)
{
  dim3 block (32, 8);
  dim3 grid (divUp (src.cols (), block.x), divUp (src.rows (), block.y));

  cudaFuncSetCacheConfig (bilateralKernel, cudaFuncCachePreferL1);
  bilateralKernel << < grid, block >> > (src, dst, 0.5f / (sigma_space *
sigma_space), 0.5f / (sigma_color * sigma_color));

  cudaSafeCall ( cudaGetLastError () );
};

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////
void
pcl::device::pyrDown (const DepthMap& src, DepthMap& dst)
{
  dst.create (src.rows () / 2, src.cols () / 2);

  dim3 block (32, 8);
  dim3 grid (divUp (dst.cols (), block.x), divUp (dst.rows (), block.y));

  pyrDownKernel << < grid, block >> > (src, dst, sigma_color);
  cudaSafeCall ( cudaGetLastError () );
};
```

Coresp.cu

```
/*
 * Software License Agreement (BSD License)
 *
 *  Point Cloud Library (PCL) - www.pointclouds.org
 *  Copyright (c) 2011, Willow Garage, Inc.
 *
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of Willow Garage, Inc. nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
```

```
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 */

#include "device.hpp"
#include <pcl/gpu/utils/device/block.hpp>

namespace pcl
{
  namespace device
  {
    __device__ unsigned int count = 0;

    struct CorespSearch
    {
      enum { CTA_SIZE_X = 32, CTA_SIZE_Y = 8, CTA_SIZE = CTA_SIZE_X * CTA_SIZE_Y
};

      struct plus
      {
        __forceinline__ __device__ int
        operator () (const int &lhs, const volatile int& rhs) const {
          return lhs + rhs;
        }
      };

      PtrStep<float> vmap_g_curr;
      PtrStep<float> nmap_g_curr;

      Mat33 Rprev_inv;
      float3 tprev;

      Intr intr;

      PtrStep<float> vmap_g_prev;
      PtrStep<float> nmap_g_prev;

      float distThres;
      float angleThres;

      mutable PtrStepSz<short2> coresp;

      mutable int* gbuf;

      __device__ __forceinline__ int
      search () const
      {
        int x = threadIdx.x + blockIdx.x * blockDim.x;
        int y = threadIdx.y + blockIdx.y * blockDim.y;

        if (x >= coresp.cols || y >= coresp.rows)
          return 0;

        coresp.ptr (y)[x] = make_short2 (-1, -1);

        float3 ncurr_g;
        ncurr_g.x = nmap_g_curr.ptr (y)[x];
```

```
        if (isnan (ncurr_g.x))
          return 0;

        float3 vcurr_g;
        vcurr_g.x = vmap_g_curr.ptr (y              )[x];
        vcurr_g.y = vmap_g_curr.ptr (y + coresp.rows)[x];
        vcurr_g.z = vmap_g_curr.ptr (y + 2 * coresp.rows)[x];

        float3 vcurr_cp = Rprev_inv * (vcurr_g - tprev);         // prev camera
coo space

        int2 ukr;         //projection
        ukr.x = __float2int_rn (vcurr_cp.x * intr.fx / vcurr_cp.z + intr.cx);
//4
        ukr.y = __float2int_rn (vcurr_cp.y * intr.fy / vcurr_cp.z + intr.cy);
//4

        if (ukr.x < 0 || ukr.y < 0 || ukr.x >= coresp.cols || ukr.y >=
coresp.rows)
          return 0;

        float3 nprev_g;
        nprev_g.x = nmap_g_prev.ptr (ukr.y)[ukr.x];

        if (isnan (nprev_g.x))
          return 0;

        float3 vprev_g;
        vprev_g.x = vmap_g_prev.ptr (ukr.y              )[ukr.x];
        vprev_g.y = vmap_g_prev.ptr (ukr.y + coresp.rows)[ukr.x];
        vprev_g.z = vmap_g_prev.ptr (ukr.y + 2 * coresp.rows)[ukr.x];

        float dist = norm (vcurr_g - vprev_g);
        if (dist > distThres)
          return 0;

        ncurr_g.y = nmap_g_curr.ptr (y + coresp.rows)[x];
        ncurr_g.z = nmap_g_curr.ptr (y + 2 * coresp.rows)[x];

        nprev_g.y = nmap_g_prev.ptr (ukr.y + coresp.rows)[ukr.x];
        nprev_g.z = nmap_g_prev.ptr (ukr.y + 2 * coresp.rows)[ukr.x];

        float sine = norm (cross (ncurr_g, nprev_g));

        /*if (sine >= 1 || asinf(sine) >= angleThres)
            return 0;*/

        if (/*sine >= 1 || */ sine >= angleThres)
          return 0;

        coresp.ptr (y)[x] = make_short2 (ukr.x, ukr.y);
        return 1;
      }

      __device__ __forceinline__ void
      reduce (int i) const
      {
        __shared__ volatile int smem[CTA_SIZE];

        int tid = Block::flattenedThreadId ();

        smem[tid] = i;
```

```
        __syncthreads ();

        Block::reduce<CTA_SIZE>(smem, plus ());

        __shared__ bool isLastBlockDone;

        if (tid == 0)
        {
          gbuf[blockIdx.x + gridDim.x * blockIdx.y] = smem[0];
          __threadfence ();

          unsigned int value = atomicInc (&count, gridDim.x * gridDim.y);

          isLastBlockDone = (value == (gridDim.x * gridDim.y - 1));
        }
        __syncthreads ();

        if (isLastBlockDone)
        {
          int sum = 0;
          int stride = Block::stride ();
          for (int pos = tid; pos < gridDim.x * gridDim.y; pos += stride)
            sum += gbuf[pos];

          smem[tid] = sum;
          __syncthreads ();
          Block::reduce<CTA_SIZE>(smem, plus ());

          if (tid == 0)
          {
            gbuf[0] = smem[0];
            count = 0;
          }
        }
      }

      __device__ __forceinline__ void
      operator () () const
      {
        int mask = search ();
        //reduce(mask); if uncomment -> need to allocate and set gbuf
      }
    };

    __global__ void
    corespKernel (const CorespSearch cs) {
      cs ();
    }
  }
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////
void
pcl::device::findCoresp (const MapArr& vmap_g_curr, const MapArr& nmap_g_curr,
                         const Mat33& Rprev_inv, const float3& tprev, const Intr&
intr,
                         const MapArr& vmap_g_prev, const MapArr& nmap_g_prev,
                         float distThres, float angleThres, PtrStepSz<short2>
coresp)
{
  CorespSearch cs;
```

```
  cs.vmap_g_curr = vmap_g_curr;
  cs.nmap_g_curr = nmap_g_curr;

  cs.Rprev_inv = Rprev_inv;
  cs.tprev = tprev;

  cs.intr = intr;

  cs.vmap_g_prev = vmap_g_prev;
  cs.nmap_g_prev = nmap_g_prev;

  cs.distThres = distThres;
  cs.angleThres = angleThres;

  cs.coresp = coresp;

  dim3 block (CorespSearch::CTA_SIZE_X, CorespSearch::CTA_SIZE_Y);
  dim3 grid (divUp (coresp.cols, block.x), divUp (coresp.rows, block.y));

  corespKernel << < grid, block >> > (cs);

  cudaSafeCall ( cudaGetLastError () );
  cudaSafeCall (cudaDeviceSynchronize ());
}
```

Device.hpp

```
/*
 * Software License Agreement (BSD License)
 *
 *  Point Cloud Library (PCL) - www.pointclouds.org
 *  Copyright (c) 2011, Willow Garage, Inc.
 *
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of Willow Garage, Inc. nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
```

```
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 */

#ifndef PCL_GPU_KINFU_DEVICE_HPP_
#define PCL_GPU_KINFU_DEVICE_HPP_

#include "pcl/gpu/utils/device/limits.hpp"
#include "pcl/gpu/utils/device/vector_math.hpp"

#include "internal.h"

namespace pcl
{
  namespace device
  {

////////////////////////////////////////////////////////////////////////////////
/////
    /// for old format


    #define INV_DIV 3.051850947599719e-5f

    __device__ __forceinline__ void
    pack_tsdf (float tsdf, int weight, short2& value)
    {
      int fixedp = max (-DIVISOR, min (DIVISOR, __float2int_rz (tsdf *
DIVISOR)));
      //int fixedp = __float2int_rz(tsdf * DIVISOR);
      value = make_short2 (fixedp, weight);
    }

    __device__ __forceinline__ void
    unpack_tsdf (short2 value, float& tsdf, int& weight)
    {
      weight = value.y;
      tsdf = __int2float_rn (value.x) / DIVISOR;   //*/ * INV_DIV;
    }

    __device__ __forceinline__ float
    unpack_tsdf (short2 value)
    {
      return static_cast<float>(value.x) / DIVISOR;    //*/ * INV_DIV;
    }


////////////////////////////////////////////////////////////////////////////////
/////
    /// for half float
    __device__ __forceinline__ void
    pack_tsdf (float tsdf, int weight, ushort2& value)
    {
      value = make_ushort2 (__float2half_rn (tsdf), weight);
    }

    __device__ __forceinline__ void
    unpack_tsdf (ushort2 value, float& tsdf, int& weight)
    {
      tsdf = __half2float (value.x);
      weight = value.y;
```

```
    }

    __device__ __forceinline__ float
    unpack_tsdf (ushort2 value)
    {
      return __half2float (value.x);
    }

    __device__ __forceinline__ float3
    operator* (const Mat33& m, const float3& vec)
    {
      return make_float3 (dot (m.data[0], vec), dot (m.data[1], vec), dot
(m.data[2], vec));
    }
  }
}

#endif /* PCL_GPU_KINFU_DEVICE_HPP_ */
```

Estimate_combined.cu

```
/*
 * Software License Agreement (BSD License)
 *
 *  Point Cloud Library (PCL) - www.pointclouds.org
 *  Copyright (c) 2011, Willow Garage, Inc.
 *
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of Willow Garage, Inc. nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 */

#include <pcl/gpu/utils/device/block.hpp>
#include <pcl/gpu/utils/device/funcattrib.hpp>
```

```
#include "device.hpp"

namespace pcl
{
  namespace device
  {
    struct Combined
    {
      enum
      {
        CTA_SIZE_X = 32,
        CTA_SIZE_Y = 8,
        CTA_SIZE = CTA_SIZE_X * CTA_SIZE_Y
      };

      struct plus
      {
        __forceinline__ __device__ float
        operator () (const float &lhs, const volatile float& rhs) const
        {
          return (lhs + rhs);
        }
      };

      Mat33 Rcurr;
      float3 tcurr;

      PtrStep<float> vmap_curr;
      PtrStep<float> nmap_curr;

      Mat33 Rprev_inv;
      float3 tprev;

      Intr intr;

      PtrStep<float> vmap_g_prev;
      PtrStep<float> nmap_g_prev;

      float distThres;
      float angleThres;

      int cols;
      int rows;

      mutable PtrStep<float> gbuf;

      __device__ __forceinline__ bool
      search (int x, int y, float3& n, float3& d, float3& s) const
      {
        float3 ncurr;
        ncurr.x = nmap_curr.ptr (y)[x];

        if (isnan (ncurr.x))
          return (false);

        float3 vcurr;
        vcurr.x = vmap_curr.ptr (y       )[x];
        vcurr.y = vmap_curr.ptr (y + rows)[x];
        vcurr.z = vmap_curr.ptr (y + 2 * rows)[x];

        float3 vcurr_g = Rcurr * vcurr + tcurr;
```

```
        float3 vcurr_cp = Rprev_inv * (vcurr_g - tprev);         // prev camera
coo space

        int2 ukr;         //projection
        ukr.x = __float2int_rn (vcurr_cp.x * intr.fx / vcurr_cp.z + intr.cx);
//4
        ukr.y = __float2int_rn (vcurr_cp.y * intr.fy / vcurr_cp.z + intr.cy);
//4

        if (ukr.x < 0 || ukr.y < 0 || ukr.x >= cols || ukr.y >= rows)
          return (false);

        float3 nprev_g;
        nprev_g.x = nmap_g_prev.ptr (ukr.y)[ukr.x];

        if (isnan (nprev_g.x))
          return (false);

        float3 vprev_g;
        vprev_g.x = vmap_g_prev.ptr (ukr.y       )[ukr.x];
        vprev_g.y = vmap_g_prev.ptr (ukr.y + rows)[ukr.x];
        vprev_g.z = vmap_g_prev.ptr (ukr.y + 2 * rows)[ukr.x];

        float dist = norm (vprev_g - vcurr_g);
        if (dist > distThres)
          return (false);

        ncurr.y = nmap_curr.ptr (y + rows)[x];
        ncurr.z = nmap_curr.ptr (y + 2 * rows)[x];

        float3 ncurr_g = Rcurr * ncurr;

        nprev_g.y = nmap_g_prev.ptr (ukr.y + rows)[ukr.x];
        nprev_g.z = nmap_g_prev.ptr (ukr.y + 2 * rows)[ukr.x];

        float sine = norm (cross (ncurr_g, nprev_g));

        if (sine >= angleThres)
          return (false);
        n = nprev_g;
        d = vprev_g;
        s = vcurr_g;
        return (true);
      }

      __device__ __forceinline__ void
      operator () () const
      {
        int x = threadIdx.x + blockIdx.x * CTA_SIZE_X;
        int y = threadIdx.y + blockIdx.y * CTA_SIZE_Y;

        float3 n, d, s;
        bool found_coresp = false;

        if (x < cols || y < rows)
          found_coresp = search (x, y, n, d, s);

        float row[7];

        if (found_coresp)
        {
          *(float3*)&row[0] = cross (s, n);
```

```
      *(float3*)&row[3] = n;
      row[6] = dot (n, d - s);
    }
    else
      row[0] = row[1] = row[2] = row[3] = row[4] = row[5] = row[6] = 0.f;

    __shared__ float smem[CTA_SIZE];
    int tid = Block::flattenedThreadId ();

    int shift = 0;
    for (int i = 0; i < 6; ++i)        //rows
    {
                #pragma unroll
      for (int j = i; j < 7; ++j)          // cols + b
      {
        __syncthreads ();
        smem[tid] = row[i] * row[j];
        __syncthreads ();

        Block::reduce<CTA_SIZE>(smem, plus ());

        if (tid == 0)
          gbuf.ptr (shift++)[blockIdx.x + gridDim.x * blockIdx.y] = smem[0];
      }
    }
  }
};

__global__ void
combinedKernel (const Combined cs)
{
  cs ();
}

struct TranformReduction
{
  enum
  {
    CTA_SIZE = 512,
    STRIDE = CTA_SIZE,

    B = 6, COLS = 6, ROWS = 6, DIAG = 6,
    UPPER_DIAG_MAT = (COLS * ROWS - DIAG) / 2 + DIAG,
    TOTAL = UPPER_DIAG_MAT + B,

    GRID_X = TOTAL
  };

  struct plus
  {
    __forceinline__ __device__ float
    operator () (const float &lhs, const volatile float& rhs) const
    {
      return lhs + rhs;
    }
  };

  PtrStep<float> gbuf;
  int length;
  mutable float* output;

  __device__ __forceinline__ void
```

```
      operator () () const
      {
        const float *beg = gbuf.ptr (blockIdx.x);
        const float *end = beg + length;

        int tid = threadIdx.x;

        float sum = 0.f;
        for (const float *t = beg + tid; t < end; t += STRIDE)
          sum += *t;

        __shared__ float smem[CTA_SIZE];

        smem[tid] = sum;
        __syncthreads ();

        Block::reduce<CTA_SIZE>(smem, plus ());

        if (tid == 0)
          output[blockIdx.x] = smem[0];
      }
    };

    __global__ void
    TransformEstimatorKernel2 (const TranformReduction tr)
    {
      tr ();
    }
  }
}


////////////////////////////////////////////////////////////////////////////////
///////////////////////////////////
void
pcl::device::estimateCombined (const Mat33& Rcurr, const float3& tcurr,
                               const MapArr& vmap_curr, const MapArr& nmap_curr,
                               const Mat33& Rprev_inv, const float3& tprev, const
Intr& intr,
                               const MapArr& vmap_g_prev, const MapArr&
nmap_g_prev,
                               float distThres, float angleThres,
                               DeviceArray2D<float>& gbuf, DeviceArray<float>&
mbuf,
                               float* matrixA_host, float* vectorB_host)
{
  int cols = vmap_curr.cols ();
  int rows = vmap_curr.rows () / 3;

  Combined cs;

  cs.Rcurr = Rcurr;
  cs.tcurr = tcurr;

  cs.vmap_curr = vmap_curr;
  cs.nmap_curr = nmap_curr;

  cs.Rprev_inv = Rprev_inv;
  cs.tprev = tprev;

  cs.intr = intr;
```

```
  cs.vmap_g_prev = vmap_g_prev;
  cs.nmap_g_prev = nmap_g_prev;

  cs.distThres = distThres;
  cs.angleThres = angleThres;

  cs.cols = cols;
  cs.rows = rows;

////////////////////////////

  dim3 block (Combined::CTA_SIZE_X, Combined::CTA_SIZE_Y);
  dim3 grid (1, 1, 1);
  grid.x = divUp (cols, block.x);
  grid.y = divUp (rows, block.y);

  mbuf.create (TranformReduction::TOTAL);
  if (gbuf.rows () != TranformReduction::TOTAL || gbuf.cols () < (int)(grid.x *
grid.y))
    gbuf.create (TranformReduction::TOTAL, grid.x * grid.y);

  cs.gbuf = gbuf;

  combinedKernel << < grid, block >> > (cs);
  cudaSafeCall ( cudaGetLastError () );
  //cudaSafeCall(cudaDeviceSynchronize());

  //printFuncAttrib(combinedKernel);

  TranformReduction tr;
  tr.gbuf = gbuf;
  tr.length = grid.x * grid.y;
  tr.output = mbuf;

  TransformEstimatorKernel2 << < TranformReduction::TOTAL,
TranformReduction::CTA_SIZE >> > (tr);
  cudaSafeCall (cudaGetLastError ());
  cudaSafeCall (cudaDeviceSynchronize ());

  float host_data[TranformReduction::TOTAL];
  mbuf.download (host_data);

  int shift = 0;
  for (int i = 0; i < 6; ++i)  //rows
    for (int j = i; j < 7; ++j)    // cols + b
    {
      float value = host_data[shift++];
      if (j == 6)       // vector b
        vectorB_host[i] = value;
      else
        matrixA_host[j * 6 + i] = matrixA_host[i * 6 + j] = value;
    }
}
```

Estimate_transform.cu

```
/*
 * Software License Agreement (BSD License)
```

```
 *
 *  Point Cloud Library (PCL) - www.pointclouds.org
 *  Copyright (c) 2011, Willow Garage, Inc.
 *
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of Willow Garage, Inc. nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 */

#include "device.hpp"
#include <pcl/gpu/utils/device/block.hpp>
#include <pcl/gpu/utils/device/funcattrib.hpp>
#include <pcl/gpu/utils/timers_cuda.hpp>

namespace pcl
{
  namespace device
  {
    template<typename T>
    struct TransformEstimator
    {
      enum
      {
        CTA_SIZE_X = 32,
        CTA_SIZE_Y = 8,
        CTA_SIZE = CTA_SIZE_X * CTA_SIZE_Y
      };

      struct plus
      {
        __forceinline__ __device__ T
        operator () (const T &lhs, const volatile T &rhs) const {
          return lhs + rhs;
        }
      };
```

```
PtrStep<float> v_dst;
PtrStep<float> n_dst;
PtrStep<float> v_src;
PtrStepSz<short2> coresp;

mutable PtrStep<T> gbuf;

__device__ __forceinline__ void
operator () () const
{
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;

  float row[7];
  row[0] = row[1] = row[2] = row[3] = row[4] = row[5] = row[6] = 0.f;

  if (x < coresp.cols || y < coresp.rows)
  {
    short2 ukr = coresp.ptr (y)[x];

    if (ukr.x != -1)
    {
      float3 n;
      n.x = n_dst.ptr (ukr.y                  )[ukr.x];
      n.y = n_dst.ptr (ukr.y +     coresp.rows)[ukr.x];
      n.z = n_dst.ptr (ukr.y + 2 * coresp.rows)[ukr.x];

      float3 d;
      d.x = v_dst.ptr (ukr.y                  )[ukr.x];
      d.y = v_dst.ptr (ukr.y +     coresp.rows)[ukr.x];
      d.z = v_dst.ptr (ukr.y + 2 * coresp.rows)[ukr.x];

      float3 s;
      s.x = v_src.ptr (y                  )[x];
      s.y = v_src.ptr (y +     coresp.rows)[x];
      s.z = v_src.ptr (y + 2 * coresp.rows)[x];

      float b = dot (n, d - s);

      *(float3*)&row[0] = cross (s, n);
      *(float3*)&row[3] = n;
      row[6] = b;
    }
  }

  __shared__ T smem[CTA_SIZE];
  int tid = Block::flattenedThreadId ();

  int shift = 0;
  for (int i = 0; i < 6; ++i)        //rows
  {
    #pragma unroll
    for (int j = i; j < 7; ++j)          // cols + b
    {
      __syncthreads ();
      smem[tid] = row[i] * row[j];
      __syncthreads ();

      Block::reduce<CTA_SIZE>(smem, plus ());

      if (tid == 0)
        gbuf.ptr (shift++)[blockIdx.x + gridDim.x * blockIdx.y] = smem[0];
```

```
          }
        }
      }
    };

    template<typename T>
    struct TranformReduction
    {
      enum
      {
        CTA_SIZE = 512,
        STRIDE = CTA_SIZE,

        B = 6, COLS = 6, ROWS = 6, DIAG = 6,
        UPPER_DIAG_MAT = (COLS * ROWS - DIAG) / 2 + DIAG,
        TOTAL = UPPER_DIAG_MAT + B,

        GRID_X = TOTAL
      };

      PtrStep<T> gbuf;
      int length;
      mutable T* output;

      __device__ __forceinline__ void
      operator () () const
      {
        const T *beg = gbuf.ptr (blockIdx.x);
        const T *end = beg + length;

        int tid = threadIdx.x;

        T sum = 0.f;
        for (const T *t = beg + tid; t < end; t += STRIDE)
          sum += *t;

        __shared__ T smem[CTA_SIZE];

        smem[tid] = sum;
        __syncthreads ();

        Block::reduce<CTA_SIZE>(smem, TransformEstimator<T>::plus ());

        if (tid == 0)
          output[blockIdx.x] = smem[0];
      }
    };

    __global__ void
    TransformEstimatorKernel1 (const TransformEstimator<float> te) {
      te ();
    }
    __global__ void
    TransformEstimatorKernel2 (const TranformReduction<float> tr) {
      tr ();
    }
  }
}

/////////////////////////////////////////////////////////////////////////////
////////////////////////////////
void
```

```
pcl::device::estimateTransform (const MapArr& v_dst, const MapArr& n_dst,
                                const MapArr& v_src, const PtrStepSz<short2>&
coresp,
                                DeviceArray2D<float>& gbuf, DeviceArray<float>&
mbuf,
                                float* matrixA_host, float* vectorB_host)
{
  typedef TransformEstimator<float> TEst;
  typedef TranformReduction<float> TRed;

  dim3 block (TEst::CTA_SIZE_X, TEst::CTA_SIZE_Y);
  dim3 grid (1, 1, 1);
  grid.x = divUp (coresp.cols, block.x);
  grid.y = divUp (coresp.rows, block.y);

  mbuf.create (TRed::TOTAL);
  if (gbuf.rows () != TRed::TOTAL || gbuf.cols () < (int)(grid.x * grid.y))
    gbuf.create (TRed::TOTAL, grid.x * grid.y);

  TEst te;
  te.n_dst = n_dst;
  te.v_dst = v_dst;
  te.v_src = v_src;
  te.coresp = coresp;
  te.gbuf = gbuf;

  TransformEstimatorKernel1 << < grid, block >> > (te);
  cudaSafeCall ( cudaGetLastError () );
  //cudaSafeCall(cudaDeviceSynchronize());

  TRed tr;
  tr.gbuf = gbuf;
  tr.length = grid.x * grid.y;
  tr.output = mbuf;

  TransformEstimatorKernel2 << < TRed::TOTAL, TRed::CTA_SIZE >> > (tr);

  cudaSafeCall ( cudaGetLastError () );
  cudaSafeCall (cudaDeviceSynchronize ());

  float host_data[TRed::TOTAL];
  mbuf.download (host_data);

  int shift = 0;
  for (int i = 0; i < 6; ++i)  //rows
    for (int j = i; j < 7; ++j)    // cols + b
    {
      float value = host_data[shift++];
      if (j == 6)       // vector b
        vectorB_host[i] = value;
      else
        matrixA_host[j * 6 + i] = matrixA_host[i * 6 + j] = value;
    }
}
```

Extract.cu

```
/*
 * Software License Agreement (BSD License)
```

```
 *
 *  Point Cloud Library (PCL) - www.pointclouds.org
 *  Copyright (c) 2011, Willow Garage, Inc.
 *
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of Willow Garage, Inc. nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 */

#include "device.hpp"
#include <pcl/gpu/utils/device/funcattrib.hpp>
#include <pcl/gpu/utils/device/block.hpp>
#include <pcl/gpu/utils/device/warp.hpp>

namespace pcl
{
  namespace device
  {

//////////////////////////////////////////////////////////////////////////////
///////
    ///// Prefix Scan utility

    enum ScanKind { exclusive, inclusive };

    template<ScanKind Kind, class T>
    __device__ __forceinline__ T
    scan_warp ( volatile T *ptr, const unsigned int idx = threadIdx.x )
    {
      const unsigned int lane = idx & 31;       // index of thread in warp
(0..31)

      if (lane >= 1)
        ptr[idx] = ptr[idx - 1] + ptr[idx];
      if (lane >= 2)
        ptr[idx] = ptr[idx - 2] + ptr[idx];
```

```
      if (lane >= 4)
        ptr[idx] = ptr[idx - 4] + ptr[idx];
      if (lane >= 8)
        ptr[idx] = ptr[idx - 8] + ptr[idx];
      if (lane >= 16)
        ptr[idx] = ptr[idx - 16] + ptr[idx];

      if (Kind == inclusive)
        return ptr[idx];
      else
        return (lane > 0) ? ptr[idx - 1] : 0;
    }


////////////////////////////////////////////////////////////////////////////////
///////
    ///// Full Volume Scan6

    enum
    {
      CTA_SIZE_X = 32,
      CTA_SIZE_Y = 6,
      CTA_SIZE = CTA_SIZE_X * CTA_SIZE_Y,

      MAX_LOCAL_POINTS = 3
    };

    __device__ int global_count = 0;
    __device__ int output_count;
    __device__ unsigned int blocks_done = 0;

    __shared__ float storage_X[CTA_SIZE * MAX_LOCAL_POINTS];
    __shared__ float storage_Y[CTA_SIZE * MAX_LOCAL_POINTS];
    __shared__ float storage_Z[CTA_SIZE * MAX_LOCAL_POINTS];

    struct FullScan6
    {
      PtrStep<volume_elem_type> volume;
      float3 cell_size;

      mutable PtrSz<PointType> output;

      __device__ __forceinline__ float
      fetch (int x, int y, int z, int& weight) const
      {
        float tsdf;
        unpack_tsdf (volume.ptr (VOLUME_Y * z + y)[x], tsdf, weight);
        return tsdf;
      }

      __device__ __forceinline__ void
      operator () () const
      {
        int x = threadIdx.x + blockIdx.x * CTA_SIZE_X;
        int y = threadIdx.y + blockIdx.y * CTA_SIZE_Y;

        if (__all (x >= VOLUME_X) || __all (y >= VOLUME_Y))
          return;

        float3 V;
        V.x = (x + 0.5f) * cell_size.x;
        V.y = (y + 0.5f) * cell_size.y;
```

```
int ftid = Block::flattenedThreadId ();

for (int z = 0; z < VOLUME_Z - 1; ++z)
{
  float3 points[MAX_LOCAL_POINTS];
  int local_count = 0;

  if (x < VOLUME_X && y < VOLUME_Y)
  {
    int W;
    float F = fetch (x, y, z, W);

    if (W != 0 && F != 1.f)
    {
      V.z = (z + 0.5f) * cell_size.z;

      //process dx
      if (x + 1 < VOLUME_X)
      {
        int Wn;
        float Fn = fetch (x + 1, y, z, Wn);

        if (Wn != 0 && Fn != 1.f)
          if ((F > 0 && Fn < 0) || (F < 0 && Fn > 0))
          {
            float3 p;
            p.y = V.y;
            p.z = V.z;

            float Vnx = V.x + cell_size.x;

            float d_inv = 1.f / (fabs (F) + fabs (Fn));
            p.x = (V.x * fabs (Fn) + Vnx * fabs (F)) * d_inv;

            points[local_count++] = p;
          }
      }               /* if (x + 1 < VOLUME_X) */

      //process dy
      if (y + 1 < VOLUME_Y)
      {
        int Wn;
        float Fn = fetch (x, y + 1, z, Wn);

        if (Wn != 0 && Fn != 1.f)
          if ((F > 0 && Fn < 0) || (F < 0 && Fn > 0))
          {
            float3 p;
            p.x = V.x;
            p.z = V.z;

            float Vny = V.y + cell_size.y;

            float d_inv = 1.f / (fabs (F) + fabs (Fn));
            p.y = (V.y * fabs (Fn) + Vny * fabs (F)) * d_inv;

            points[local_count++] = p;
          }
      }               /*  if (y + 1 < VOLUME_Y) */

      //process dz
```

```
            //if (z + 1 < VOLUME_Z) // guaranteed by loop
            {
              int Wn;
              float Fn = fetch (x, y, z + 1, Wn);

              if (Wn != 0 && Fn != 1.f)
                if ((F > 0 && Fn < 0) || (F < 0 && Fn > 0))
                {
                  float3 p;
                  p.x = V.x;
                  p.y = V.y;

                  float Vnz = V.z + cell_size.z;

                  float d_inv = 1.f / (fabs (F) + fabs (Fn));
                  p.z = (V.z * fabs (Fn) + Vnz * fabs (F)) * d_inv;

                  points[local_count++] = p;
                }
            }               /* if (z + 1 < VOLUME_Z) */
          }              /* if (W != 0 && F != 1.f) */
        }            /* if (x < VOLUME_X && y < VOLUME_Y) */


        ///not we fulfilled points array at current iteration
        int total_warp = __popc (__ballot (local_count > 0)) + __popc (__ballot
(local_count > 1)) + __popc (__ballot (local_count > 2));

        if (total_warp > 0)
        {
          int lane = Warp::laneId ();
          int storage_index = (ftid >> Warp::LOG_WARP_SIZE) * Warp::WARP_SIZE *
MAX_LOCAL_POINTS;

          volatile int* cta_buffer = (int*)(storage_X + storage_index);

          cta_buffer[lane] = local_count;
          int offset = scan_warp<exclusive>(cta_buffer, lane);

          if (lane == 0)
          {
            int old_global_count = atomicAdd (&global_count, total_warp);
            cta_buffer[0] = old_global_count;
          }
          int old_global_count = cta_buffer[0];

          for (int l = 0; l < local_count; ++l)
          {
            storage_X[storage_index + offset + l] = points[l].x;
            storage_Y[storage_index + offset + l] = points[l].y;
            storage_Z[storage_index + offset + l] = points[l].z;
          }

          PointType *pos = output.data + old_global_count + lane;
          for (int idx = lane; idx < total_warp; idx += Warp::STRIDE, pos +=
Warp::STRIDE)
          {
            float x = storage_X[storage_index + idx];
            float y = storage_Y[storage_index + idx];
            float z = storage_Z[storage_index + idx];
            store_point_type (x, y, z, pos);
          }
```

```
            bool full = (old_global_count + total_warp) >= output.size;

            if (full)
              break;
          }

        }         /* for(int z = 0; z < VOLUME_Z - 1; ++z) */


        ///////////////////////////
        // prepare for future scans
        if (ftid == 0)
        {
          unsigned int total_blocks = gridDim.x * gridDim.y * gridDim.z;
          unsigned int value = atomicInc (&blocks_done, total_blocks);

          //last block
          if (value == total_blocks - 1)
          {
            output_count = min ((int)output.size, global_count);
            blocks_done = 0;
            global_count = 0;
          }
        }
      }       /* operator() */

      __device__ __forceinline__ void
      store_point_type (float x, float y, float z, float4* ptr) const {
        *ptr = make_float4 (x, y, z, 0);
      }
      __device__ __forceinline__ void
      store_point_type (float x, float y, float z, float3* ptr) const {
        *ptr = make_float3 (x, y, z);
      }
    };

    __global__ void
    extractKernel (const FullScan6 fs) {
      fs ();
    }
  }
}

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
size_t
pcl::device::extractCloud (const PtrStep<volume_elem_type>& volume, const float3&
volume_size,
                           PtrSz<PointType> output)
{
  FullScan6 fs;
  fs.volume = volume;
  fs.cell_size.x = volume_size.x / VOLUME_X;
  fs.cell_size.y = volume_size.y / VOLUME_Y;
  fs.cell_size.z = volume_size.z / VOLUME_Z;
  fs.output = output;

  dim3 block (CTA_SIZE_X, CTA_SIZE_Y);
  dim3 grid (divUp (VOLUME_X, block.x), divUp (VOLUME_Y, block.y));

  //cudaFuncSetCacheConfig(extractKernel, cudaFuncCachePreferL1);
```

```
  //printFuncAttrib(extractKernel);

  extractKernel << < grid, block >> > (fs);
  cudaSafeCall ( cudaGetLastError () );
  cudaSafeCall (cudaDeviceSynchronize ());

  int size;
  cudaSafeCall ( cudaMemcpyFromSymbol (&size, output_count, sizeof(size)) );
  return (size_t)size;
}


namespace pcl
{
  namespace device
  {
    template<typename NormalType>
    struct ExtractNormals
    {
      float3 cell_size;
      PtrStep<volume_elem_type> volume;
      PtrSz<PointType> points;

      mutable NormalType* output;

      __device__ __forceinline__ float
      readTsdf (int x, int y, int z) const
      {
        return unpack_tsdf (volume.ptr (VOLUME_Y * z + y)[x]);
      }

      __device__ __forceinline__ float3
      fetchPoint (int idx) const
      {
        PointType p = points.data[idx];
        return make_float3 (p.x, p.y, p.z);
      }
      __device__ __forceinline__ void
      storeNormal (int idx, float3 normal) const
      {
        NormalType n;
        n.x = normal.x; n.y = normal.y; n.z = normal.z;
        output[idx] = n;
      }

      __device__ __forceinline__ int3
      getVoxel (const float3& point) const
      {
        int vx = __float2int_rd (point.x / cell_size.x);        // round to
negative infinity
        int vy = __float2int_rd (point.y / cell_size.y);
        int vz = __float2int_rd (point.z / cell_size.z);

        return make_int3 (vx, vy, vz);
      }

      __device__ __forceinline__ void
      operator () () const
      {
        int idx = threadIdx.x + blockIdx.x * blockDim.x;

        if (idx >= points.size)
```

```
          return;
        const float qnan = numeric_limits<float>::quiet_NaN ();
        float3 n = make_float3 (qnan, qnan, qnan);

        float3 point = fetchPoint (idx);
        int3 g = getVoxel (point);

        if (g.x > 2 && g.y > 2 && g.z > 2 && g.x < VOLUME_X - 3 && g.y < VOLUME_Y
- 3 && g.z < VOLUME_Z - 3)
        {
          float3 t;

          t = point;
          t.x += cell_size.x / 4;
          float Fx1 = interpolateTrilineary (t);

          t = point;
          t.x -= cell_size.x / 4;
          float Fx2 = interpolateTrilineary (t);

          n.x = (Fx1 - Fx2);

          t = point;
          t.y += cell_size.y / 4;
          float Fy1 = interpolateTrilineary (t);

          t = point;
          t.y -= cell_size.y / 4;
          float Fy2 = interpolateTrilineary (t);

          n.y = (Fy1 - Fy2);

          t = point;
          t.z += cell_size.z / 4;
          float Fz1 = interpolateTrilineary (t);

          t = point;
          t.z -= cell_size.z / 4;
          float Fz2 = interpolateTrilineary (t);

          n.z = (Fz1 - Fz2);

          n = normalized (n);
        }
        storeNormal (idx, n);
      }

      __device__ __forceinline__ float
      interpolateTrilineary (const float3& point) const
      {
        int3 g = getVoxel (point);

        float vx = (g.x + 0.5f) * cell_size.x;
        float vy = (g.y + 0.5f) * cell_size.y;
        float vz = (g.z + 0.5f) * cell_size.z;

        g.x = (point.x < vx) ? (g.x - 1) : g.x;
        g.y = (point.y < vy) ? (g.y - 1) : g.y;
        g.z = (point.z < vz) ? (g.z - 1) : g.z;

        float a = (point.x - (g.x + 0.5f) * cell_size.x) / cell_size.x;
        float b = (point.y - (g.y + 0.5f) * cell_size.y) / cell_size.y;
```

```
        float c = (point.z - (g.z + 0.5f) * cell_size.z) / cell_size.z;

        float res = readTsdf (g.x + 0, g.y + 0, g.z + 0) * (1 - a) * (1 - b) * (1
- c) +
                    readTsdf (g.x + 0, g.y + 0, g.z + 1) * (1 - a) * (1 - b) * c
+
                    readTsdf (g.x + 0, g.y + 1, g.z + 0) * (1 - a) * b * (1 - c)
+
                    readTsdf (g.x + 0, g.y + 1, g.z + 1) * (1 - a) * b * c +
                    readTsdf (g.x + 1, g.y + 0, g.z + 0) * a * (1 - b) * (1 - c)
+
                    readTsdf (g.x + 1, g.y + 0, g.z + 1) * a * (1 - b) * c +
                    readTsdf (g.x + 1, g.y + 1, g.z + 0) * a * b * (1 - c) +
                    readTsdf (g.x + 1, g.y + 1, g.z + 1) * a * b * c;
        return res;
      }
    };

    template<typename NormalType>
    __global__ void
    extractNormalsKernel (const ExtractNormals<NormalType> en) {
      en ();
    }
  }
}

////////////////////////////////////////////////////////////////////////////////
///////////////////////////////
template<typename NormalType> void
pcl::device::extractNormals (const PtrStep<volume_elem_type>& volume, const
float3& volume_size,
                             const PtrSz<PointType>& points, NormalType* output)
{
  ExtractNormals<NormalType> en;
  en.volume = volume;
  en.cell_size.x = volume_size.x / VOLUME_X;
  en.cell_size.y = volume_size.y / VOLUME_Y;
  en.cell_size.z = volume_size.z / VOLUME_Z;
  en.points = points;
  en.output = output;

  dim3 block (256);
  dim3 grid (divUp (points.size, block.x));

  extractNormalsKernel << < grid, block >> > (en);
  cudaSafeCall ( cudaGetLastError () );
  cudaSafeCall (cudaDeviceSynchronize ());
}

using namespace pcl::device;

template void pcl::device::extractNormals<PointType>(const
PtrStep<volume_elem_type>&volume, const float3 &volume_size, const
PtrSz<PointType>&input, PointType * output);
template void pcl::device::extractNormals<float8>(const
PtrStep<volume_elem_type>&volume, const float3 &volume_size, const
PtrSz<PointType>&input, float8 * output);
```

Extract_shared_buf.cu_backup

```
/*
* Software License Agreement (BSD License)
```

```
*
*  Copyright (c) 2011, Willow Garage, Inc.
*  All rights reserved.
*
*  Redistribution and use in source and binary forms, with or without
*  modification, are permitted provided that the following conditions
*  are met:
*
*   * Redistributions of source code must retain the above copyright
*     notice, this list of conditions and the following disclaimer.
*   * Redistributions in binary form must reproduce the above
*     copyright notice, this list of conditions and the following
*     disclaimer in the documentation and/or other materials provided
*     with the distribution.
*   * Neither the name of Willow Garage, Inc. nor the names of its
*     contributors may be used to endorse or promote products derived
*     from this software without specific prior written permission.
*
*  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
*  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
*  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
*  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
*  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
*  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
*  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
*  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
*  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
*  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
*  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
*  POSSIBILITY OF SUCH DAMAGE.
*
*  Author: Anatoly Baskeheev, Itseez Ltd, (myname.mysurname@mycompany.com)
*/
#include "device.hpp"
#include "pcl/gpu/utils/device/funcattrib.hpp"
#include "pcl/gpu/utils/device/block.hpp"
#include "pcl/gpu/utils/device/warp.hpp"

namespace pcl
{
    namespace device
    {

//////////////////////////////////////////////////////////////////////////////
///////
        ///// Prefix Scan utility

        enum ScanKind { exclusive,  inclusive } ;

        template <ScanKind Kind , class T>
        __device__ __forceinline__ T scan_warp ( volatile T *ptr , const unsigned
int idx = threadIdx.x )
        {
            const unsigned int lane = idx & 31; // index of thread in warp
(0..31)

            if ( lane >=  1) ptr [idx ] = ptr [idx -  1] + ptr [idx];
            if ( lane >=  2) ptr [idx ] = ptr [idx -  2] + ptr [idx];
            if ( lane >=  4) ptr [idx ] = ptr [idx -  4] + ptr [idx];
            if ( lane >=  8) ptr [idx ] = ptr [idx -  8] + ptr [idx];
            if ( lane >= 16) ptr [idx ] = ptr [idx - 16] + ptr [idx];
```

```
            if( Kind == inclusive )
                return ptr [idx ];
            else
                return (lane > 0) ? ptr [idx - 1] : 0;
        }


////////////////////////////////////////////////////////////////////////////////
///////
        ///// Full Volume Scan6

        enum
        {
            CTA_SIZE_X = 32,
            CTA_SIZE_Y = 8,
            CTA_SIZE = CTA_SIZE_X * CTA_SIZE_Y
        };

        __shared__ int shared_count;
        __device__ int global_count = 0;
        __device__ int global_count_dd = 0;
        __device__ int output_count;
        __device__ unsigned int blocks_done = 0;


        const static int MAX_SHARED_COUNT = CTA_SIZE*3;
        __shared__ float storage[3][MAX_SHARED_COUNT];

        __shared__ int cta_buffer[CTA_SIZE];

        struct FullScan6
        {

            PtrStep<volume_elem_type> volume;
            float3 cell_size;


            mutable PtrSz<PointType> output;


            __device__ __forceinline__ float fetch(int x, int y, int z, int&
weight) const
            {
                float tsdf;
                unpack_tsdf(volume.ptr(VOLUME_Y * z + y)[x], tsdf, weight);
                return tsdf;
            }

            __device__ __forceinline__ void operator()()const
            {
                int x = threadIdx.x + blockIdx.x * CTA_SIZE_X;
                int y = threadIdx.y + blockIdx.y * CTA_SIZE_Y;

                if (threadIdx.x == 0 && threadIdx.y == 0 && threadIdx.z == 0)
                    shared_count = 0;

                if (__all(x >= VOLUME_X) || __all(y >= VOLUME_Y))
                    return;

                float3 V;
                V.x = (x + 0.5f) * cell_size.x;
                V.y = (y + 0.5f) * cell_size.y;
```

```
                int ftid = Block::flattenedThreadId();

                for(int z = 0; z < VOLUME_Z - 1; ++z)
                {
                    float3 points[3];
                    int local_count = 0;

                    if (x < VOLUME_X && y < VOLUME_Y)
                    {
                        int W;
                        float F = fetch(x, y, z, W);

                        if (W != 0 && F != 1.f)
                        {
                            V.z = (z + 0.5f) * cell_size.z;

                            //process dx
                            if (x + 1 < VOLUME_X)
                            {
                                int Wn;
                                float Fn = fetch(x+1, y, z, Wn);

                                if (Wn != 0 && Fn != 1.f)
                                    if ((F > 0 && Fn < 0) || (F < 0 && Fn > 0))
                                    {
                                        float3 p;
                                        p.y = V.y;
                                        p.z = V.z;

                                        float Vnx = V.x + cell_size.x;

                                        float d_inv = 1.f/(fabs(F) + fabs(Fn));
                                        p.x = (V.x * fabs(Fn) + Vnx * fabs(F)) *
d_inv;

                                        points[local_count++] = p;
                                    }
                            } /* if (x + 1 < VOLUME_X) */

                            //process dy
                            if (y + 1 < VOLUME_Y)
                            {
                                int Wn;
                                float Fn = fetch(x, y+1, z, Wn);

                                if (Wn != 0 && Fn != 1.f)
                                    if ((F > 0 && Fn < 0) || (F < 0 && Fn > 0))
                                    {
                                        float3 p;
                                        p.x = V.x;
                                        p.z = V.z;

                                        float Vny = V.y + cell_size.y;

                                        float d_inv = 1.f/(fabs(F) + fabs(Fn));
                                        p.y = (V.x * fabs(Fn) + Vny * fabs(F)) *
d_inv;

                                        points[local_count++] = p;
                                    }
                            }  /*  if (y + 1 < VOLUME_Y) */
```

```
                            //process dz
                            //if (z + 1 < VOLUME_Z) // guaranteed by loop
                            {
                                int Wn;
                                float Fn = fetch(x, y, z+1, Wn);

                                if (Wn != 0 && Fn != 1.f)
                                    if ((F > 0 && Fn < 0) || (F < 0 && Fn > 0))
                                    {
                                        float3 p;
                                        p.x = V.x;
                                        p.y = V.y;

                                        float Vnz = V.z + cell_size.z;

                                        float d_inv = 1.f/(fabs(F) + fabs(Fn));
                                        p.z = (V.x * fabs(Fn) + Vnz * fabs(F)) *
d_inv;

                                        points[local_count++] = p;
                                    }
                            } /* if (z + 1 < VOLUME_Z) */
                        }  /* if (W != 0 && F != 1.f) */
                    }  /* if (x < VOLUME_X && y < VOLUME_Y) */


                    int total_warp = __popc(__ballot(local_count > 0)) +
__popc(__ballot(local_count > 1)) + __popc(__ballot(local_count > 2));
                    int offset;

                    int block_old_shared_count = shared_count;
                    __syncthreads();

                    if (total_warp > 0)
                    {
                        cta_buffer[ftid] = local_count;
                        offset = scan_warp<exclusive>(cta_buffer, ftid);

                        int lane = Warp::laneId();
                        if (lane == 0)
                        {
                            int old_shared_count = atomicAdd(&shared_count,
total_warp);
                            cta_buffer[ftid - lane] = old_shared_count;
                        }
                        int old_shared_count = cta_buffer[ftid - lane];

                        //can write
                        if (old_shared_count + total_warp < MAX_SHARED_COUNT)
                        {
                            flush2shared(points, local_count, old_shared_count +
offset);
                            total_warp = 0;
                        }
                    }
                    __syncthreads();

                     if (ftid == 0)
                        printf("b....shared_count %d %d\n", shared_count,
block_old_shared_count);
```

```
                    __syncthreads();


                    if (shared_count > MAX_SHARED_COUNT)
                    {
                        if (ftid == 0)
                            printf("flushing %d\n", shared_count);

                        if (flush2global(block_old_shared_count))
                            break;

                        __syncthreads();

                        if (ftid == 0)
                            printf("after flushing %d %d\n", shared_count,
global_count);
                    }

                    __syncthreads();

                    //repeat unsucessiful write
                    if (total_warp > 0)
                    {
                        int lane = Warp::laneId();
                        if (lane == 0)
                        {
                            int old_shared_count = atomicAdd(&shared_count,
total_warp);
                            cta_buffer[ftid - lane] = old_shared_count;

                            printf("@%d - %d\n", old_shared_count, total_warp);
                        }
                        int old_shared_count = cta_buffer[ftid - lane];

                        //sure that shared memory is enought after flushing to
global
                        flush2shared(points, local_count, old_shared_count +
offset);
                    }

                    __syncthreads();

                    if (ftid == 0)
                    {
                        //printf("....shared_count %d\n", shared_count);

                    }

                    //need in order to be sure that all unsucessiful writes
before are completed
                    __syncthreads();

                } /* for(int z = 0; z < VOLUME_Z - 1; ++z) */


                ////////////////////////
                // prepare for future scans
                if (ftid == 0)
                {
                    unsigned int total_blocks = gridDim.x * gridDim.y *
gridDim.z;
                    unsigned int value = atomicInc(&blocks_done, total_blocks);
```

```
                    //last block
                    if (value == total_blocks - 1)
                    {
                        printf("global_count => %d\n", global_count);
                        output_count = min(output.size, global_count);
                        blocks_done = 0;
                        global_count = 0;

                        printf("global_count_dd %d\n", global_count_dd);
                    }
                }

            } /* operator() */

            __device__ __forceinline__ void flush2shared(float3 points[], int
local_count, int offset) const
            {
                for(int l = 0; l < local_count; ++l)
                {
                    storage[0][offset+l] = points[l].x;
                    storage[1][offset+l] = points[l].y;
                    storage[2][offset+l] = points[l].z;
                }
            }

            __device__ __forceinline__ bool flush2global(int old_shared_count)
const
            {
                int ftid = Block::flattenedThreadId();
                int STRIDE = Block::stride();

                if (ftid == 0)
                {
                    int old_global = atomicAdd(&global_count, old_shared_count);
                    cta_buffer[0] = old_global;
                }
                __syncthreads();

                int old_global = cta_buffer[0];

                int new_length = min(output.size, old_global + old_shared_count);

                PointType *beg = output.data + old_global;
                PointType *end = output.data + new_length;


                int index = ftid;
                for(PointType* pos = beg + ftid; pos < end; pos += STRIDE, index
+= STRIDE)
                {
                    float x = storage[0][index];
                    float y = storage[1][index];
                    float z = storage[2][index];
                    store_point_type(x, y, z, pos);
                }

                bool full = (old_global + old_shared_count) >= output.size;

                __syncthreads();

                if (ftid == 0)
```

```
                    shared_count = 0;

                __syncthreads();

                return full;
            }

            __device__ __forceinline__ void store_point_type(float x, float y,
float z, float4* ptr) const { *ptr = make_float4(x, y, z, 0); }
            __device__ __forceinline__ void store_point_type(float x, float y,
float z, float3* ptr) const { *ptr = make_float3(x, y, z); }
        };

        //__global__ void extractKernel(const FullScan26 fs) { fs(); }
        __global__ void extractKernel(const FullScan6 fs) { fs(); }
    }
}

size_t pcl::device::extractCloud(const PtrStep<volume_elem_type>& volume, const
float3& volume_size, PtrSz<PointType> output)
{
    FullScan6 fs;
    fs.volume = volume;
    fs.cell_size.x = volume_size.x / VOLUME_X;
    fs.cell_size.y = volume_size.y / VOLUME_Y;
    fs.cell_size.z = volume_size.z / VOLUME_Z;
    fs.output = output;

    dim3 block(CTA_SIZE_X, CTA_SIZE_Y);
    dim3 grid(divUp(VOLUME_X, block.x), divUp(VOLUME_Y, block.y));

    cudaFuncSetCacheConfig(extractKernel, cudaFuncCachePreferL1);
    printFuncAttrib(extractKernel);

    extractKernel<<<grid, block>>>(fs);
    cudaSafeCall( cudaGetLastError() );
    cudaSafeCall(cudaDeviceSynchronize());

    printf(">>> osz %d\n", output.size);

    int size;
    cudaSafeCall( cudaMemcpyFromSymbol(&size, output_count, sizeof(size)) );
    return (size_t)size;
}



#if 0
struct FullScan26
{
    enum
    {
        CTA_SIZE_X = 32,
        CTA_SIZE_Y = 8
    };
    PtrStep<volume_elem_type> volume;
    float3 cell_size;

    __device__ __forceinline__ float fetch(int x, int y, int z, int& weight)
const
    {
        float tsdf;
```

```
    unpack_tsdf(volume.ptr(VOLUME_Y * z + y)[x], tsdf, weight);
    return tsdf;
}

__device__ __forceinline__ void operator()()const
{
    int x = threadIdx.x + blockIdx.x * CTA_SIZE_X;
    int y = threadIdx.y + blockIdx.y * CTA_SIZE_Y;

    if (x >= VOLUME_X || y >= VOLUME_Y)
        return;

    if (threadIdx.x == 0 && threadIdx.y == 0)
        shared_count = 0;

    float3 V;
    V.x = (x + 0.5f) * cell_size.x;
    V.y = (y + 0.5f) * cell_size.y;

    for(int z = 0; z < VOLUME_Z - 1; ++z)
    {
        int W;
        float F = fetch(x, y, z, W);

        if (W == 0 || F == 1.f)
            continue;

        V.z = (z + 0.5f) * cell_size.z;

        //front 3x3
        int dz = 1;
        for(int dy = -1; dy < 2; ++dy)
        {
            if (y + dy >= VOLUME_X || y + dy < 0)
                continue;

            for(int dx = -1; dx < 2; ++dx)
            {
                if (x + dx >= VOLUME_X || x + dx < 0)
                    continue;

                int Wn;
                float Fn = fetch(x+dx, y+dy, z+dz, Wn);

                if (Wn == 0 || Fn == 1.f)
                    continue;

                if ((F > 0 && Fn < 0) || (F < 0 && Fn > 0))
                {
                    float3 Vn = V;
                    Vn.x += dx * cell_size.x;
                    Vn.y += dy * cell_size.y;
                    Vn.z += dz * cell_size.z;

                    float d_inv = 1.f/(fabs(F) + fabs(Fn));
                    float3 p = (V * fabs(Fn) + Vn * fabs(F)) * d_inv;

                    if(!store(p))
                        return;
                }
            }
        }
```

```
            //middle 3x1 + 1
            dz = 0;
            for(int dy = 0; dy < 2; ++dy)
            {
                if (y + dy >= VOLUME_Y)
                    continue;

                for(int dx = -1; dx < dy * 2; ++dx)
                {
                    if (x + dx >= VOLUME_X || x + dx < 0)
                        continue;

                    int Wn;
                    float Fn = fetch(x+dx, y+dy, z+dz, Wn);

                    if (Wn == 0 || Fn == 1.f)
                        continue;

                    if ((F > 0 && Fn < 0) || (F < 0 && Fn > 0))
                    {
                        float3 Vn = V;
                        Vn.x += dx * cell_size.x;
                        Vn.y += dy * cell_size.y;
                        Vn.z += dz * cell_size.z;

                        float d_inv = 1.f/(fabs(F) + fabs(Fn));
                        float3 p = (V * fabs(Fn) + Vn * fabs(F)) * d_inv;

                        if(!store(p))
                            return;
                    }
                }
            }

        } /* for(int z = 0; z < VOLUME_Z - 1; ++z) */
    } /* operator() */

    __device__ __forceinline__ bool store(const float3& p) const
    {
        //__ffs __ballot(1);

        return true;
    }
};

#endif
```

Image_generator.cu

```
/*
 * Software License Agreement (BSD License)
 *
 *  Point Cloud Library (PCL) - www.pointclouds.org
 *  Copyright (c) 2011, Willow Garage, Inc.
 *
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
```

```
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of Willow Garage, Inc. nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 */

#include "device.hpp"

using namespace pcl::device;

namespace pcl
{
  namespace device
  {
    struct ImageGenerator
    {
      enum
      {
        CTA_SIZE_X = 32, CTA_SIZE_Y = 8
      };

      PtrStep<float> vmap;
      PtrStep<float> nmap;

      LightSource light;

      mutable PtrStepSz<uchar3> dst;

      __device__ __forceinline__ void
      operator () () const
      {
        int x = threadIdx.x + blockIdx.x * CTA_SIZE_X;
        int y = threadIdx.y + blockIdx.y * CTA_SIZE_Y;

        if (x >= dst.cols || y >= dst.rows)
          return;

        float3 v, n;
        v.x = vmap.ptr (y)[x];
        n.x = nmap.ptr (y)[x];

        uchar3 color = make_uchar3 (0, 0, 0);
```

```
        if (!isnan (v.x) && !isnan (n.x))
        {
          v.y = vmap.ptr (y + dst.rows)[x];
          v.z = vmap.ptr (y + 2 * dst.rows)[x];

          n.y = nmap.ptr (y + dst.rows)[x];
          n.z = nmap.ptr (y + 2 * dst.rows)[x];

          float weight = 1.f;

          for (int i = 0; i < light.number; ++i)
          {
            float3 vec = normalized (light.pos[i] - v);

            weight *= fabs (dot (vec, n));
          }

          int br = (int)(205 * weight) + 50;
          br = max (0, min (255, br));
          color = make_uchar3 (br, br, br);
        }
        dst.ptr (y)[x] = color;
      }
    };

    __global__ void
    generateImageKernel (const ImageGenerator ig) {
      ig ();
    }
  }
}

////////////////////////////////////////////////////////////////////////////////
///////////////////////////////////
void
pcl::device::generateImage (const MapArr& vmap, const MapArr& nmap, const
LightSource& light,
                            PtrStepSz<uchar3> dst)
{
  ImageGenerator ig;
  ig.vmap = vmap;
  ig.nmap = nmap;
  ig.light = light;
  ig.dst = dst;

  dim3 block (ImageGenerator::CTA_SIZE_X, ImageGenerator::CTA_SIZE_Y);
  dim3 grid (divUp (dst.cols, block.x), divUp (dst.rows, block.y));

  generateImageKernel << < grid, block >> > (ig);
  cudaSafeCall (cudaGetLastError ());
  cudaSafeCall (cudaDeviceSynchronize ());
}
```

Maps.cu

```
/*
 * Software License Agreement (BSD License)
 *
 *  Point Cloud Library (PCL) - www.pointclouds.org
 *  Copyright (c) 2011, Willow Garage, Inc.
```

```
 *
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of Willow Garage, Inc. nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 */

#include "device.hpp"

using namespace pcl::device;
using namespace pcl::gpu;

namespace pcl
{
  namespace device
  {
    __global__ void
    computeVmapKernel (const PtrStepSz<unsigned short> depth, PtrStep<float>
vmap, float fx_inv, float fy_inv, float cx, float cy)
    {
      int u = threadIdx.x + blockIdx.x * blockDim.x;
      int v = threadIdx.y + blockIdx.y * blockDim.y;

      if (u < depth.cols && v < depth.rows)
      {
        int z = depth.ptr (v)[u];

        if (z != 0)
        {
          float vx = z * (u - cx) * fx_inv;
          float vy = z * (v - cy) * fy_inv;
          float vz = z;

          vmap.ptr (v                 )[u] = vx;
          vmap.ptr (v + depth.rows    )[u] = vy;
          vmap.ptr (v + depth.rows * 2)[u] = vz;
        }
```

```
        else
          vmap.ptr (v)[u] = numeric_limits<float>::quiet_NaN ();

      }
    }

    __global__ void
    computeNmapKernel (int rows, int cols, const PtrStep<float> vmap,
PtrStep<float> nmap)
    {
      int u = threadIdx.x + blockIdx.x * blockDim.x;
      int v = threadIdx.y + blockIdx.y * blockDim.y;

      if (u >= cols || v >= rows)
        return;

      if (u == cols - 1 || v == rows - 1)
      {
        nmap.ptr (v)[u] = numeric_limits<float>::quiet_NaN ();
        return;
      }

      float3 v00, v01, v10;
      v00.x = vmap.ptr (v  )[u];
      v01.x = vmap.ptr (v  )[u + 1];
      v10.x = vmap.ptr (v + 1)[u];

      if (!isnan (v00.x) && !isnan (v01.x) && !isnan (v10.x))
      {
        v00.y = vmap.ptr (v + rows)[u];
        v01.y = vmap.ptr (v + rows)[u + 1];
        v10.y = vmap.ptr (v + 1 + rows)[u];

        v00.z = vmap.ptr (v + 2 * rows)[u];
        v01.z = vmap.ptr (v + 2 * rows)[u + 1];
        v10.z = vmap.ptr (v + 1 + 2 * rows)[u];

        float3 r = normalized (cross (v01 - v00, v10 - v00));

        nmap.ptr (v       )[u] = r.x;
        nmap.ptr (v + rows)[u] = r.y;
        nmap.ptr (v + 2 * rows)[u] = r.z;
      }
      else
        nmap.ptr (v)[u] = numeric_limits<float>::quiet_NaN ();
    }
  }
}

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
void
pcl::device::createVMap (const Intr& intr, const DepthMap& depth, MapArr& vmap)
{
  vmap.create (depth.rows () * 3, depth.cols ());

  dim3 block (32, 8);
  dim3 grid (1, 1, 1);
  grid.x = divUp (depth.cols (), block.x);
  grid.y = divUp (depth.rows (), block.y);

  float fx = intr.fx, cx = intr.cx;
```

```
  float fy = intr.fy, cy = intr.cy;

  computeVmapKernel << < grid, block >> > (depth, vmap, 1.f / fx, 1.f / fy, cx,
cy);
  cudaSafeCall (cudaGetLastError ());
}

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
void
pcl::device::createNMap (const MapArr& vmap, MapArr& nmap)
{
  nmap.create (vmap.rows (), vmap.cols ());

  int rows = vmap.rows () / 3;
  int cols = vmap.cols ();

  dim3 block (32, 8);
  dim3 grid (1, 1, 1);
  grid.x = divUp (cols, block.x);
  grid.y = divUp (rows, block.y);

  computeNmapKernel << < grid, block >> > (rows, cols, vmap, nmap);
  cudaSafeCall (cudaGetLastError ());
}

namespace pcl
{
  namespace device
  {
    __global__ void
    tranformMapsKernel (int rows, int cols, const PtrStep<float> vmap_src, const
PtrStep<float> nmap_src,
                        const Mat33 Rmat, const float3 tvec, PtrStepSz<float>
vmap_dst, PtrStep<float> nmap_dst)
    {
      int x = threadIdx.x + blockIdx.x * blockDim.x;
      int y = threadIdx.y + blockIdx.y * blockDim.y;

      const float qnan = pcl::device::numeric_limits<float>::quiet_NaN ();

      if (x < cols && y < rows)
      {
        //vetexes
        float3 vsrc, vdst = make_float3 (qnan, qnan, qnan);
        vsrc.x = vmap_src.ptr (y)[x];

        if (!isnan (vsrc.x))
        {
          vsrc.y = vmap_src.ptr (y + rows)[x];
          vsrc.z = vmap_src.ptr (y + 2 * rows)[x];

          vdst = Rmat * vsrc + tvec;

          vmap_dst.ptr (y + rows)[x] = vdst.y;
          vmap_dst.ptr (y + 2 * rows)[x] = vdst.z;
        }

        vmap_dst.ptr (y)[x] = vdst.x;

        //normals
        float3 nsrc, ndst = make_float3 (qnan, qnan, qnan);
```

```
        nsrc.x = nmap_src.ptr (y)[x];

        if (!isnan (nsrc.x))
        {
          nsrc.y = nmap_src.ptr (y + rows)[x];
          nsrc.z = nmap_src.ptr (y + 2 * rows)[x];

          ndst = Rmat * nsrc;

          nmap_dst.ptr (y + rows)[x] = ndst.y;
          nmap_dst.ptr (y + 2 * rows)[x] = ndst.z;
        }

        nmap_dst.ptr (y)[x] = ndst.x;
      }
    }
  }
}

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////
void
pcl::device::tranformMaps (const MapArr& vmap_src, const MapArr& nmap_src,
                           const Mat33& Rmat, const float3& tvec,
                           MapArr& vmap_dst, MapArr& nmap_dst)
{
  int cols = vmap_src.cols ();
  int rows = vmap_src.rows () / 3;

  vmap_dst.create (rows * 3, cols);
  nmap_dst.create (rows * 3, cols);

  dim3 block (32, 8);
  dim3 grid (1, 1, 1);
  grid.x = divUp (cols, block.x);
  grid.y = divUp (rows, block.y);

  tranformMapsKernel << < grid, block >> > (rows, cols, vmap_src, nmap_src, Rmat,
tvec, vmap_dst, nmap_dst);
  cudaSafeCall (cudaGetLastError ());

  cudaSafeCall (cudaDeviceSynchronize ());
}

namespace pcl
{
  namespace device
  {
    template<bool normalize>
    __global__ void
    resizeMapKernel (int drows, int dcols, int srows, const PtrStep<float> input,
PtrStep<float> output)
    {
      int x = threadIdx.x + blockIdx.x * blockDim.x;
      int y = threadIdx.y + blockIdx.y * blockDim.y;

      if (x >= dcols || y >= drows)
        return;

      const float qnan = numeric_limits<float>::quiet_NaN ();

      int xs = x * 2;
```

```
      int ys = y * 2;

      float x00 = input.ptr (ys + 0)[xs + 0];
      float x01 = input.ptr (ys + 0)[xs + 1];
      float x10 = input.ptr (ys + 1)[xs + 0];
      float x11 = input.ptr (ys + 1)[xs + 1];

      if (isnan (x00) || isnan (x01) || isnan (x10) || isnan (x11))
      {
        output.ptr (y)[x] = qnan;
        return;
      }
      else
      {
        float3 n;

        n.x = (x00 + x01 + x10 + x11) / 4;

        float y00 = input.ptr (ys + srows + 0)[xs + 0];
        float y01 = input.ptr (ys + srows + 0)[xs + 1];
        float y10 = input.ptr (ys + srows + 1)[xs + 0];
        float y11 = input.ptr (ys + srows + 1)[xs + 1];

        n.y = (y00 + y01 + y10 + y11) / 4;

        float z00 = input.ptr (ys + 2 * srows + 0)[xs + 0];
        float z01 = input.ptr (ys + 2 * srows + 0)[xs + 1];
        float z10 = input.ptr (ys + 2 * srows + 1)[xs + 0];
        float z11 = input.ptr (ys + 2 * srows + 1)[xs + 1];

        n.z = (z00 + z01 + z10 + z11) / 4;

        if (normalize)
          n = normalized (n);

        output.ptr (y        )[x] = n.x;
        output.ptr (y + drows)[x] = n.y;
        output.ptr (y + 2 * drows)[x] = n.z;
      }
    }

    template<bool normalize>
    void
    resizeMap (const MapArr& input, MapArr& output)
    {
      int in_cols = input.cols ();
      int in_rows = input.rows () / 3;

      int out_cols = in_cols / 2;
      int out_rows = in_rows / 2;

      output.create (out_rows * 3, out_cols);

      dim3 block (32, 8);
      dim3 grid (divUp (out_cols, block.x), divUp (out_rows, block.y));
      resizeMapKernel<normalize><< < grid, block >> > (out_rows, out_cols,
in_rows, input, output);
      cudaSafeCall ( cudaGetLastError () );
      cudaSafeCall (cudaDeviceSynchronize ());
    }
  }
}
```

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////
void
pcl::device::resizeVMap (const MapArr& input, MapArr& output)
{
  resizeMap<false>(input, output);
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////
void
pcl::device::resizeNMap (const MapArr& input, MapArr& output)
{
  resizeMap<true>(input, output);
}

namespace pcl
{
  namespace device
  {

    template<typename T>
    __global__ void
    convertMapKernel (int rows, int cols, const PtrStep<float> map, PtrStep<T>
output)
    {
      int x = threadIdx.x + blockIdx.x * blockDim.x;
      int y = threadIdx.y + blockIdx.y * blockDim.y;

      if (x >= cols || y >= rows)
        return;

      const float qnan = numeric_limits<float>::quiet_NaN ();

      T t;
      t.x = map.ptr (y)[x];
      if (!isnan (t.x))
      {
        t.y = map.ptr (y + rows)[x];
        t.z = map.ptr (y + 2 * rows)[x];
      }
      else
        t.y = t.z = qnan;

      output.ptr (y)[x] = t;
    }
  }
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////
template<typename T> void
pcl::device::convert (const MapArr& vmap, DeviceArray2D<T>& output)
{
  int cols = vmap.cols ();
  int rows = vmap.rows () / 3;

  output.create (rows, cols);

  dim3 block (32, 8);
  dim3 grid (divUp (cols, block.x), divUp (rows, block.y));
```

```
  convertMapKernel<T><< < grid, block >> > (rows, cols, vmap, output);
  cudaSafeCall ( cudaGetLastError () );
  cudaSafeCall (cudaDeviceSynchronize ());
}

template void pcl::device::convert (const MapArr& vmap, DeviceArray2D<float4>&
output);
template void pcl::device::convert (const MapArr& vmap, DeviceArray2D<float8>&
output);
```

Normal_eigen.cu

```
/*
 * Software License Agreement (BSD License)
 *
 *  Point Cloud Library (PCL) - www.pointclouds.org
 *  Copyright (c) 2011, Willow Garage, Inc.
 *
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of Willow Garage, Inc. nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 */

#include "device.hpp"
#include <pcl/gpu/features/device/eigen.hpp>

namespace pcl
{
  namespace device
  {
    enum
    {
      kx = 7,
      ky = 7,
      STEP = 1
```

```
    };

    __global__ void
    computeNmapKernelEigen (int rows, int cols, const PtrStep<float> vmap,
PtrStep<float> nmap)
    {
      int u = threadIdx.x + blockIdx.x * blockDim.x;
      int v = threadIdx.y + blockIdx.y * blockDim.y;

      if (u >= cols || v >= rows)
        return;

      nmap.ptr (v)[u] = numeric_limits<float>::quiet_NaN ();

      if (isnan (vmap.ptr (v)[u]))
        return;

      int ty = min (v - ky / 2 + ky, rows - 1);
      int tx = min (u - kx / 2 + kx, cols - 1);

      float3 centroid = make_float3 (0.f, 0.f, 0.f);
      int counter = 0;
      for (int cy = max (v - ky / 2, 0); cy < ty; cy += STEP)
        for (int cx = max (u - kx / 2, 0); cx < tx; cx += STEP)
        {
          float v_x = vmap.ptr (cy)[cx];
          if (!isnan (v_x))
          {
            centroid.x += v_x;
            centroid.y += vmap.ptr (cy + rows)[cx];
            centroid.z += vmap.ptr (cy + 2 * rows)[cx];
            ++counter;
          }
        }

      if (counter < kx * ky / 2)
        return;

      centroid *= 1.f / counter;

      float cov[] = {0, 0, 0, 0, 0, 0};

      for (int cy = max (v - ky / 2, 0); cy < ty; cy += STEP)
        for (int cx = max (u - kx / 2, 0); cx < tx; cx += STEP)
        {
          float3 v;
          v.x = vmap.ptr (cy)[cx];
          if (isnan (v.x))
            continue;

          v.y = vmap.ptr (cy + rows)[cx];
          v.z = vmap.ptr (cy + 2 * rows)[cx];

          float3 d = v - centroid;

          cov[0] += d.x * d.x;              //cov (0, 0)
          cov[1] += d.x * d.y;              //cov (0, 1)
          cov[2] += d.x * d.z;              //cov (0, 2)
          cov[3] += d.y * d.y;              //cov (1, 1)
          cov[4] += d.y * d.z;              //cov (1, 2)
          cov[5] += d.z * d.z;              //cov (2, 2)
        }
```

```
      typedef Eigen33::Mat33 Mat33;
      Eigen33 eigen33 (cov);

      Mat33 tmp;
      Mat33 vec_tmp;
      Mat33 evecs;
      float3 evals;
      eigen33.compute (tmp, vec_tmp, evecs, evals);

      float3 n = normalized (evecs[0]);

      u = threadIdx.x + blockIdx.x * blockDim.x;
      v = threadIdx.y + blockIdx.y * blockDim.y;

      nmap.ptr (v       )[u] = n.x;
      nmap.ptr (v + rows)[u] = n.y;
      nmap.ptr (v + 2 * rows)[u] = n.z;
    }
  }
}

/////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////
void
pcl::device::computeNormalsEigen (const MapArr& vmap, MapArr& nmap)
{
  int cols = vmap.cols ();
  int rows = vmap.rows () / 3;

  nmap.create (vmap.rows (), vmap.cols ());

  dim3 block (32, 8);
  dim3 grid (1, 1, 1);
  grid.x = divUp (cols, block.x);
  grid.y = divUp (rows, block.y);

  computeNmapKernelEigen << < grid, block >> > (rows, cols, vmap, nmap);
  cudaSafeCall (cudaGetLastError ());
  cudaSafeCall (cudaDeviceSynchronize ());
}
```

Ray_caster.cu

```
/*
 * Software License Agreement (BSD License)
 *
 *  Point Cloud Library (PCL) - www.pointclouds.org
 *  Copyright (c) 2011, Willow Garage, Inc.
 *
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
```

```
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of Willow Garage, Inc. nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 */
#include "device.hpp"

namespace pcl
{
  namespace device
  {
    __device__ __forceinline__ float
    getMinTime (const float3& volume_max, const float3& origin, const float3&
dir)
    {
      float txmin = ( (dir.x > 0 ? 0.f : volume_max.x) - origin.x) / dir.x;
      float tymin = ( (dir.y > 0 ? 0.f : volume_max.y) - origin.y) / dir.y;
      float tzmin = ( (dir.z > 0 ? 0.f : volume_max.z) - origin.z) / dir.z;

      return fmax ( fmax (txmin, tymin), tzmin);
    }

    __device__ __forceinline__ float
    getMaxTime (const float3& volume_max, const float3& origin, const float3&
dir)
    {
      float txmax = ( (dir.x > 0 ? volume_max.x : 0.f) - origin.x) / dir.x;
      float tymax = ( (dir.y > 0 ? volume_max.y : 0.f) - origin.y) / dir.y;
      float tzmax = ( (dir.z > 0 ? volume_max.z : 0.f) - origin.z) / dir.z;

      return fmin (fmin (txmax, tymax), tzmax);
    }

    struct RayCaster
    {
      enum { CTA_SIZE_X = 32, CTA_SIZE_Y = 8 };

      Mat33 Rcurr;
      float3 tcurr;

      float time_step;
      float3 volume_size;

      float3 cell_size;
      int cols, rows;
```

```
      PtrStep<volume_elem_type> volume;

      Intr intr;

      mutable PtrStep<float> nmap;
      mutable PtrStep<float> vmap;

      __device__ __forceinline__ float3
      get_ray_next (int x, int y) const
      {
        float3 ray_next;
        ray_next.x = (x - intr.cx) / intr.fx;
        ray_next.y = (y - intr.cy) / intr.fy;
        ray_next.z = 1;
        return ray_next;
      }

      __device__ __forceinline__ bool
      checkInds (const int3& g) const
      {
        return (g.x >= 0 && g.y >= 0 && g.z >= 0 && g.x < VOLUME_X && g.y <
VOLUME_Y && g.z < VOLUME_X);
      }

      __device__ __forceinline__ float
      readTsdf (int x, int y, int z) const
      {
        return unpack_tsdf (volume.ptr (VOLUME_Y * z + y)[x]);
      }

      __device__ __forceinline__ int3
      getVoxel (float3 point) const
      {
        int vx = __float2int_rd (point.x / cell_size.x);        // round to
negative infinity
        int vy = __float2int_rd (point.y / cell_size.y);
        int vz = __float2int_rd (point.z / cell_size.z);

        return make_int3 (vx, vy, vz);
      }

      __device__ __forceinline__ float
      interpolateTrilineary (const float3& origin, const float3& dir, float time)
const
      {
        return interpolateTrilineary (origin + dir * time);
      }

      __device__ __forceinline__ float
      interpolateTrilineary (const float3& point) const
      {
        int3 g = getVoxel (point);

        if (g.x <= 0 || g.x >= VOLUME_X - 1)
          return numeric_limits<float>::quiet_NaN ();

        if (g.y <= 0 || g.y >= VOLUME_Y - 1)
          return numeric_limits<float>::quiet_NaN ();

        if (g.z <= 0 || g.z >= VOLUME_Z - 1)
          return numeric_limits<float>::quiet_NaN ();
```

```
        float vx = (g.x + 0.5f) * cell_size.x;
        float vy = (g.y + 0.5f) * cell_size.y;
        float vz = (g.z + 0.5f) * cell_size.z;

        g.x = (point.x < vx) ? (g.x - 1) : g.x;
        g.y = (point.y < vy) ? (g.y - 1) : g.y;
        g.z = (point.z < vz) ? (g.z - 1) : g.z;

        float a = (point.x - (g.x + 0.5f) * cell_size.x) / cell_size.x;
        float b = (point.y - (g.y + 0.5f) * cell_size.y) / cell_size.y;
        float c = (point.z - (g.z + 0.5f) * cell_size.z) / cell_size.z;

        float res = readTsdf (g.x + 0, g.y + 0, g.z + 0) * (1 - a) * (1 - b) * (1
- c) +
                    readTsdf (g.x + 0, g.y + 0, g.z + 1) * (1 - a) * (1 - b) * c
+
                    readTsdf (g.x + 0, g.y + 1, g.z + 0) * (1 - a) * b * (1 - c)
+
                    readTsdf (g.x + 0, g.y + 1, g.z + 1) * (1 - a) * b * c +
                    readTsdf (g.x + 1, g.y + 0, g.z + 0) * a * (1 - b) * (1 - c)
+
                    readTsdf (g.x + 1, g.y + 0, g.z + 1) * a * (1 - b) * c +
                    readTsdf (g.x + 1, g.y + 1, g.z + 0) * a * b * (1 - c) +
                    readTsdf (g.x + 1, g.y + 1, g.z + 1) * a * b * c;
        return res;
      }
#if 0
      __device__ __forceinline__ void
      operator () () const
      {
        int x = threadIdx.x + blockIdx.x * CTA_SIZE_X;
        int y = threadIdx.y + blockIdx.y * CTA_SIZE_Y;

        if (x >= cols || y >= rows)
          return;

        vmap.ptr (y)[x] = numeric_limits<float>::quiet_NaN ();
        nmap.ptr (y)[x] = numeric_limits<float>::quiet_NaN ();

        float3 ray_start = tcurr;
        float3 ray_next = Rcurr * get_ray_next (x, y) + tcurr;

        float3 ray_dir = normalized (ray_next - ray_start);

        //ensure that it isn't a degenerate case
        ray_dir.x = (ray_dir.x == 0.f) ? 1e-15 : ray_dir.x;
        ray_dir.y = (ray_dir.y == 0.f) ? 1e-15 : ray_dir.y;
        ray_dir.z = (ray_dir.z == 0.f) ? 1e-15 : ray_dir.z;

        // computer time when entry and exit volume
        float time_start_volume = getMinTime (volume_size, ray_start, ray_dir);
        float time_exit_volume = getMaxTime (volume_size, ray_start, ray_dir);

        const float min_dist = 0.f;         //in mm
        time_start_volume = fmax (time_start_volume, min_dist);
        if (time_start_volume >= time_exit_volume)
          return;

        int time_curr = time_start_volume;
        int3 g = getVoxel (ray_start + ray_dir * time_curr);
        g.x = max (0, min (g.x, VOLUME_X - 1));
        g.y = max (0, min (g.y, VOLUME_Y - 1));
```

```
        g.z = max (0, min (g.z, VOLUME_Z - 1));

        float tsdf = readTsdf (g.x, g.y, g.z);

        //infinite loop guard
        const float max_time = 3 * (volume_size.x + volume_size.y +
volume_size.z);

        for (; time_curr < max_time; time_curr += time_step)
        {
          float tsdf_prev = tsdf;

          int3 g = getVoxel (  ray_start + ray_dir * (time_curr + time_step)  );
          if (!checkInds (g))
            break;

          tsdf = readTsdf (g.x, g.y, g.z);

          if (tsdf_prev < 0.f && tsdf > 0.f)
            break;

          if (tsdf_prev > 0.f && tsdf < 0.f)           //zero crossing
          {
            float Ftdt = interpolateTrilineary (ray_start, ray_dir, time_curr +
time_step);
            if (isnan (Ftdt))
              break;

            float Ft = interpolateTrilineary (ray_start, ray_dir, time_curr);
            if (isnan (Ft))
              break;

            float Ts = time_curr - time_step * Ft / (Ftdt - Ft);

            float3 vetex_found = ray_start + ray_dir * Ts;

            vmap.ptr (y       )[x] = vetex_found.x;
            vmap.ptr (y + rows)[x] = vetex_found.y;
            vmap.ptr (y + 2 * rows)[x] = vetex_found.z;

            int3 g = getVoxel ( ray_start + ray_dir * time_curr );
            //if (g.x != 0 && g.y != 0 && g.z != 0 && g.x != VOLUME_X - 1 && g.y
!= VOLUME_Y - 1 && g.z != VOLUME_Z - 1)
            {
              float3 normal;

              //extract gradient
              normal.x = readTsdf (g.x + 1, g.y, g.z) - readTsdf (g.x - 1, g.y,
g.z);
              normal.y = readTsdf (g.x, g.y + 1, g.z) - readTsdf (g.x, g.y - 1,
g.z);
              normal.z = readTsdf (g.x, g.y, g.z + 1) - readTsdf (g.x, g.y, g.z -
1);

              //normalize if volume isn't cubic
              normal.x /= cell_size.x;
              normal.y /= cell_size.y;
              normal.z /= cell_size.z;

              normal = normalized (normal);

              nmap.ptr (y       )[x] = normal.x;
```

```
            nmap.ptr (y + rows)[x] = normal.y;
            nmap.ptr (y + 2 * rows)[x] = normal.z;
          }
          break;
        }
      }          /* for(;;)  */
    }

#else
    __device__ __forceinline__ void
    operator () () const
    {
      int x = threadIdx.x + blockIdx.x * CTA_SIZE_X;
      int y = threadIdx.y + blockIdx.y * CTA_SIZE_Y;

      if (x >= cols || y >= rows)
        return;

      vmap.ptr (y)[x] = numeric_limits<float>::quiet_NaN ();
      nmap.ptr (y)[x] = numeric_limits<float>::quiet_NaN ();

      float3 ray_start = tcurr;
      float3 ray_next = Rcurr * get_ray_next (x, y) + tcurr;

      float3 ray_dir = normalized (ray_next - ray_start);

      //ensure that it isn't a degenerate case
      ray_dir.x = (ray_dir.x == 0.f) ? 1e-15 : ray_dir.x;
      ray_dir.y = (ray_dir.y == 0.f) ? 1e-15 : ray_dir.y;
      ray_dir.z = (ray_dir.z == 0.f) ? 1e-15 : ray_dir.z;

      // computer time when entry and exit volume
      float time_start_volume = getMinTime (volume_size, ray_start, ray_dir);
      float time_exit_volume = getMaxTime (volume_size, ray_start, ray_dir);

      const float min_dist = 0.f;         //in mm
      time_start_volume = fmax (time_start_volume, min_dist);
      if (time_start_volume >= time_exit_volume)
        return;

      int time_curr = time_start_volume;
      int3 g = getVoxel (ray_start + ray_dir * time_curr);
      g.x = max (0, min (g.x, VOLUME_X - 1));
      g.y = max (0, min (g.y, VOLUME_Y - 1));
      g.z = max (0, min (g.z, VOLUME_Z - 1));

      float tsdf = readTsdf (g.x, g.y, g.z);

      //infinite loop guard
      const float max_time = 3 * (volume_size.x + volume_size.y +
volume_size.z);

      for (; time_curr < max_time; time_curr += time_step)
      {
        float tsdf_prev = tsdf;

        int3 g = getVoxel (  ray_start + ray_dir * (time_curr + time_step)  );
        if (!checkInds (g))
          break;

        tsdf = readTsdf (g.x, g.y, g.z);
```

```
          if (tsdf_prev < 0.f && tsdf > 0.f)
            break;


          if (tsdf_prev > 0.f && tsdf < 0.f)           //zero crossing
          {
            float Ftdt = interpolateTrilineary (ray_start, ray_dir, time_curr +
time_step);
            if (isnan (Ftdt))
              break;

            float Ft = interpolateTrilineary (ray_start, ray_dir, time_curr);
            if (isnan (Ft))
              break;

            //float Ts = time_curr - time_step * Ft/(Ftdt - Ft);
            float Ts = time_curr - time_step * Ft / (Ftdt - Ft);

            float3 vetex_found = ray_start + ray_dir * Ts;

            vmap.ptr (y       )[x] = vetex_found.x;
            vmap.ptr (y + rows)[x] = vetex_found.y;
            vmap.ptr (y + 2 * rows)[x] = vetex_found.z;

            int3 g = getVoxel ( ray_start + ray_dir * time_curr );
            if (g.x > 1 && g.y > 1 && g.z > 1 && g.x < VOLUME_X - 2 && g.y <
VOLUME_Y - 2 && g.z < VOLUME_Z - 2)
            {
              float3 t;
              float3 n;

              t = vetex_found;
              t.x += cell_size.x / 4;
              float Fx1 = interpolateTrilineary (t);

              t = vetex_found;
              t.x -= cell_size.x / 4;
              float Fx2 = interpolateTrilineary (t);

              n.x = (Fx1 - Fx2);

              t = vetex_found;
              t.y += cell_size.y / 4;
              float Fy1 = interpolateTrilineary (t);

              t = vetex_found;
              t.y -= cell_size.y / 4;
              float Fy2 = interpolateTrilineary (t);

              n.y = (Fy1 - Fy2);

              t = vetex_found;
              t.z += cell_size.z / 4;
              float Fz1 = interpolateTrilineary (t);

              t = vetex_found;
              t.z -= cell_size.z / 4;
              float Fz2 = interpolateTrilineary (t);

              n.z = (Fz1 - Fz2);

              n = normalized (n);
```

```
              nmap.ptr (y       )[x] = n.x;
              nmap.ptr (y + rows)[x] = n.y;
              nmap.ptr (y + 2 * rows)[x] = n.z;
            }
            break;
          }

        }          /* for(;;)  */
      }

#endif
    };

    __global__ void
    rayCastKernel (const RayCaster rc) {
      rc ();
    }
  }
}


////////////////////////////////////////////////////////////////////////////////
////////////////////////////////
void
pcl::device::raycast (const Intr& intr, const Mat33& Rcurr, const float3& tcurr,
                      float tranc_dist, const float3& volume_size,
                      const PtrStep<volume_elem_type>& volume, MapArr& vmap,
MapArr& nmap)
{
  RayCaster rc;

  rc.Rcurr = Rcurr;
  rc.tcurr = tcurr;

  rc.time_step = tranc_dist * 0.8f;

  rc.volume_size = volume_size;

  rc.cell_size.x = volume_size.x / VOLUME_X;
  rc.cell_size.y = volume_size.y / VOLUME_Y;
  rc.cell_size.z = volume_size.z / VOLUME_Z;

  rc.cols = vmap.cols ();
  rc.rows = vmap.rows () / 3;

  rc.intr = intr;

  rc.volume = volume;
  rc.vmap = vmap;
  rc.nmap = nmap;

  dim3 block (RayCaster::CTA_SIZE_X, RayCaster::CTA_SIZE_Y);
  dim3 grid (divUp (rc.cols, block.x), divUp (rc.rows, block.y));

  rayCastKernel << < grid, block >> > (rc);
  cudaSafeCall (cudaGetLastError ());
  //cudaSafeCall(cudaDeviceSynchronize());
}
```

Tsdf_volume.cu

```
/*
 * Software License Agreement (BSD License)
 *
 *  Point Cloud Library (PCL) - www.pointclouds.org
 *  Copyright (c) 2011, Willow Garage, Inc.
 *
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of Willow Garage, Inc. nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 */

#include "device.hpp"

using namespace pcl::device;

namespace pcl
{
  namespace device
  {
    template<typename T>
    __global__ void
    initializeVolume (PtrStepSz<T> volume)
    {
      int x = threadIdx.x + blockIdx.x * blockDim.x;
      int y = threadIdx.y + blockIdx.y * blockDim.y;

      if (x < volume.cols && y < volume.rows)
        pack_tsdf (0.f, 0, volume.ptr (y)[x]);
    }

    template<typename T>
    void
```

```
    initVolume (PtrStepSz<T> volume)
    {
      dim3 block (32, 16);
      dim3 grid (1, 1, 1);
      grid.x = divUp (volume.cols, block.x);
      grid.y = divUp (volume.rows, block.y);

      initializeVolume << < grid, block >> > (volume);
      cudaSafeCall ( cudaGetLastError () );
      cudaSafeCall (cudaDeviceSynchronize ());
    }
  }
}

template void pcl::device::initVolume (PtrStepSz<short2> volume);
template void pcl::device::initVolume (PtrStepSz<ushort2> volume);

namespace pcl
{
  namespace device
  {
    struct Tsdf
    {
      enum
      {
        CTA_SIZE_X = 32, CTA_SIZE_Y = 8,
        MAX_WEIGHT = 1 << 7
      };

      mutable PtrStep<short2> volume;
      float3 volume_size;       //in mm

      Intr intr;

      Mat33 Rcurr_inv;
      float3 tcurr;

      PtrStepSz<ushort> depth_raw;

      float tranc_dist;

      __device__ __forceinline__ float3
      getVoxelGCoo (int x, int y, int z) const
      {
        float3 coo = make_float3 (x, y, z);
        coo += 0.5f;         //shift to cell center;

        coo.x *= volume_size.x / VOLUME_X;
        coo.y *= volume_size.y / VOLUME_Y;
        coo.z *= volume_size.z / VOLUME_Z;

        return coo;
      }

      __device__ __forceinline__ void
      operator () () const
      {
        int x = threadIdx.x + blockIdx.x * CTA_SIZE_X;
        int y = threadIdx.y + blockIdx.y * CTA_SIZE_Y;

        if (x >= VOLUME_X || y >= VOLUME_Y)
          return;
```

```
        short2 *pos = volume.ptr (y) + x;
        int elem_step = volume.step * VOLUME_Y / sizeof(short2);

        for (int z = 0; z < VOLUME_Z; ++z, pos += elem_step)
        {
          float3 v_g = getVoxelGCoo (x, y, z);            //3 // p

          //tranform to curr cam coo space
          float3 v = Rcurr_inv * (v_g - tcurr);           //4

          int2 coo;           //project to current cam
          coo.x = __float2int_rn (v.x * intr.fx / v.z + intr.cx);
          coo.y = __float2int_rn (v.y * intr.fy / v.z + intr.cy);

          if (v.z > 0 && coo.x >= 0 && coo.y >= 0 && coo.x < depth_raw.cols &&
coo.y < depth_raw.rows)           //6
          {
            int Dp = depth_raw.ptr (coo.y)[coo.x];

            if (Dp != 0)
            {
              float xl = (coo.x - intr.cx) / intr.fx;
              float yl = (coo.y - intr.cy) / intr.fy;
              float lambda_inv = rsqrtf (xl * xl + yl * yl + 1);

              float sdf = norm (tcurr - v_g) * lambda_inv - Dp;

              sdf *= (-1);

              if (sdf >= -tranc_dist)
              {
                float tsdf = fmin (1, sdf / tranc_dist);

                int weight_prev;
                float tsdf_prev;

                //read and unpack
                unpack_tsdf (*pos, tsdf_prev, weight_prev);

                const int Wrk = 1;

                float tsdf_new = (tsdf_prev * weight_prev + Wrk * tsdf) /
(weight_prev + Wrk);
                int weight_new = min (weight_prev + Wrk, MAX_WEIGHT);

                pack_tsdf (tsdf_new, weight_new, *pos);
              }
            }
          }
        }
      }
    };

    __global__ void
    integrateTsdfKernel (const Tsdf tsdf) {
      tsdf ();
    }

    __global__ void
    tsdf2 (PtrStep<short2> volume, const float3 volume_size, const float
tranc_dist, const Mat33 Rcurr_inv, float3 tcurr,
```

```
          const Intr intr, const PtrStepSz<ushort> depth_raw, const float3
cell_size)
    {
      int x = threadIdx.x + blockIdx.x * blockDim.x;
      int y = threadIdx.y + blockIdx.y * blockDim.y;

      if (x >= VOLUME_X || y >= VOLUME_Y)
        return;

      short2 *pos = volume.ptr (y) + x;
      int elem_step = volume.step * VOLUME_Y / sizeof(short2);

      float v_g_x = (x + 0.5f) * cell_size.x - tcurr.x;
      float v_g_y = (y + 0.5f) * cell_size.y - tcurr.y;
      float v_g_z = (0 + 0.5f) * cell_size.z - tcurr.z;

      float v_x = Rcurr_inv.data[0].x * v_g_x + Rcurr_inv.data[0].y * v_g_y +
Rcurr_inv.data[0].z * v_g_z;
      float v_y = Rcurr_inv.data[1].x * v_g_x + Rcurr_inv.data[1].y * v_g_y +
Rcurr_inv.data[1].z * v_g_z;
      float v_z = Rcurr_inv.data[2].x * v_g_x + Rcurr_inv.data[2].y * v_g_y +
Rcurr_inv.data[2].z * v_g_z;

//#pragma unroll
      for (int z = 0; z < VOLUME_Z; ++z)
      {
        float3 vr;
        vr.x = v_g_x;
        vr.y = v_g_y;
        vr.z = (v_g_z + z * cell_size.z);

        float3 v;
        v.x = v_x + Rcurr_inv.data[0].z * z * cell_size.z;
        v.y = v_y + Rcurr_inv.data[1].z * z * cell_size.z;
        v.z = v_z + Rcurr_inv.data[2].z * z * cell_size.z;

        int2 coo;         //project to current cam
        coo.x = __float2int_rn (v.x * intr.fx / v.z + intr.cx);
        coo.y = __float2int_rn (v.y * intr.fy / v.z + intr.cy);


        if (v.z > 0 && coo.x >= 0 && coo.y >= 0 && coo.x < depth_raw.cols &&
coo.y < depth_raw.rows)         //6
        {
          int Dp = depth_raw.ptr (coo.y)[coo.x];

          if (Dp != 0)
          {
            float xl = (coo.x - intr.cx) / intr.fx;
            float yl = (coo.y - intr.cy) / intr.fy;
            float lambda_inv = rsqrtf (xl * xl + yl * yl + 1);

            float sdf = Dp - norm (vr) * lambda_inv;


            if (sdf >= -tranc_dist)
            {
              float tsdf = fmin (1, sdf / tranc_dist);

              int weight_prev;
              float tsdf_prev;
```

```
              //read and unpack
              unpack_tsdf (*pos, tsdf_prev, weight_prev);

              const int Wrk = 1;

              float tsdf_new = (tsdf_prev * weight_prev + Wrk * tsdf) /
(weight_prev + Wrk);
              int weight_new = min (weight_prev + Wrk, Tsdf::MAX_WEIGHT);

              pack_tsdf (tsdf_new, weight_new, *pos);
            }
          }
        }
        pos += elem_step;
      }       /* for(int z = 0; z < VOLUME_Z; ++z) */
    }      /* __global__ */
  }
}


////////////////////////////////////////////////////////////////////////////////
///////////////////////////////
void
pcl::device::integrateTsdfVolume (const PtrStepSz<ushort>& depth_raw, const Intr&
intr, const float3& volume_size,
                                  const Mat33& Rcurr_inv, const float3& tcurr,
float tranc_dist,
                                  PtrStep<short2> volume)
{
  Tsdf tsdf;

  tsdf.volume = volume;
  tsdf.volume_size = volume_size;

  tsdf.intr = intr;

  tsdf.Rcurr_inv = Rcurr_inv;
  tsdf.tcurr = tcurr;
  tsdf.depth_raw = depth_raw;

  tsdf.tranc_dist = tranc_dist;

  dim3 block (Tsdf::CTA_SIZE_X, Tsdf::CTA_SIZE_Y);
  dim3 grid (divUp (VOLUME_X, block.x), divUp (VOLUME_Y, block.y));

#if 0
  //float3 cell_size;
  //cell_size.x = volume_size.x / VOLUME_X;
  //cell_size.y = volume_size.y / VOLUME_Y;
  //cell_size.z = volume_size.z / VOLUME_Z;
  //tsdf2<<<grid, block>>>(volume, volume_size, tranc_dist, Rcurr_inv, tcurr,
intr, depth_raw, cell_size);
  else
    integrateTsdfKernel << < grid, block >> > (tsdf);
#endif
  cudaSafeCall ( cudaGetLastError () );
  cudaSafeCall (cudaDeviceSynchronize ());
}


namespace pcl
{
```

```
  namespace device
  {
    __global__ void
    scaleDepth (const PtrStepSz<ushort> depth, PtrStep<float> scaled, const Intr
intr)
    {
      int x = threadIdx.x + blockIdx.x * blockDim.x;
      int y = threadIdx.y + blockIdx.y * blockDim.y;

      if (x >= depth.cols || y >= depth.rows)
        return;

      int Dp = depth.ptr (y)[x];

      float xl = (x - intr.cx) / intr.fx;
      float yl = (y - intr.cy) / intr.fy;
      float lambda = sqrtf (xl * xl + yl * yl + 1);

      scaled.ptr (y)[x] = Dp * lambda;
    }

    __global__ void
    tsdf23 (const PtrStepSz<float> depthScaled, PtrStep<short2> volume,
            const float tranc_dist, const Mat33 Rcurr_inv, const float3 tcurr,
const Intr intr, const float3 cell_size)
    {
      int x = threadIdx.x + blockIdx.x * blockDim.x;
      int y = threadIdx.y + blockIdx.y * blockDim.y;

      if (x >= VOLUME_X || y >= VOLUME_Y)
        return;

      float v_g_x = (x + 0.5f) * cell_size.x - tcurr.x;
      float v_g_y = (y + 0.5f) * cell_size.y - tcurr.y;
      float v_g_z = (0 + 0.5f) * cell_size.z - tcurr.z;

      float v_g_part_norm = v_g_x * v_g_x + v_g_y * v_g_y;

      float v_x = (Rcurr_inv.data[0].x * v_g_x + Rcurr_inv.data[0].y * v_g_y +
Rcurr_inv.data[0].z * v_g_z) * intr.fx;
      float v_y = (Rcurr_inv.data[1].x * v_g_x + Rcurr_inv.data[1].y * v_g_y +
Rcurr_inv.data[1].z * v_g_z) * intr.fy;
      float v_z = (Rcurr_inv.data[2].x * v_g_x + Rcurr_inv.data[2].y * v_g_y +
Rcurr_inv.data[2].z * v_g_z);

      float z_scaled = 0;

      float Rcurr_inv_0_z_scaled = Rcurr_inv.data[0].z * cell_size.z * intr.fx;
      float Rcurr_inv_1_z_scaled = Rcurr_inv.data[1].z * cell_size.z * intr.fy;

      float tranc_dist_inv = 1.0f / tranc_dist;

      short2* pos = volume.ptr (y) + x;
      int elem_step = volume.step * VOLUME_Y / sizeof(short2);

//#pragma unroll
      for (int z = 0; z < VOLUME_Z;
           ++z,
           v_g_z += cell_size.z,
           z_scaled += cell_size.z,
           v_x += Rcurr_inv_0_z_scaled,
           v_y += Rcurr_inv_1_z_scaled,
```

```
           pos += elem_step)
      {
        float inv_z = 1.0f / (v_z + Rcurr_inv.data[2].z * z_scaled);
        if (inv_z < 0)
            continue;

        // project to current cam
        int2 coo =
        {
          __float2int_rn (v_x * inv_z + intr.cx),
          __float2int_rn (v_y * inv_z + intr.cy)
        };

        if (coo.x >= 0 && coo.y >= 0 && coo.x < depthScaled.cols && coo.y <
depthScaled.rows)         //6
        {
          float Dp_scaled = depthScaled.ptr (coo.y)[coo.x];

          float sdf = Dp_scaled - sqrtf (v_g_z * v_g_z + v_g_part_norm);

          if (Dp_scaled != 0 && sdf >= -tranc_dist)
          {
            float tsdf = fmin (1.0f, sdf * tranc_dist_inv);

            //read and unpack
            float tsdf_prev;
            int weight_prev;
            unpack_tsdf (*pos, tsdf_prev, weight_prev);

            const int Wrk = 1;

            float tsdf_new = (tsdf_prev * weight_prev + Wrk * tsdf) /
(weight_prev + Wrk);
            int weight_new = min (weight_prev + Wrk, Tsdf::MAX_WEIGHT);

            pack_tsdf (tsdf_new, weight_new, *pos);
          }
        }
      }       // for(int z = 0; z < VOLUME_Z; ++z)
    }      // __global__

    __global__ void
    tsdf23normal_hack (const PtrStepSz<float> depthScaled, PtrStep<short2>
volume,
                  const float tranc_dist, const Mat33 Rcurr_inv, const float3
tcurr, const Intr intr, const float3 cell_size)
    {
        int x = threadIdx.x + blockIdx.x * blockDim.x;
        int y = threadIdx.y + blockIdx.y * blockDim.y;

        if (x >= VOLUME_X || y >= VOLUME_Y)
            return;

        const float v_g_x = (x + 0.5f) * cell_size.x - tcurr.x;
        const float v_g_y = (y + 0.5f) * cell_size.y - tcurr.y;
        float v_g_z = (0 + 0.5f) * cell_size.z - tcurr.z;

        float v_g_part_norm = v_g_x * v_g_x + v_g_y * v_g_y;

        float v_x = (Rcurr_inv.data[0].x * v_g_x + Rcurr_inv.data[0].y * v_g_y +
Rcurr_inv.data[0].z * v_g_z) * intr.fx;
```

```
        float v_y = (Rcurr_inv.data[1].x * v_g_x + Rcurr_inv.data[1].y * v_g_y +
Rcurr_inv.data[1].z * v_g_z) * intr.fy;
        float v_z = (Rcurr_inv.data[2].x * v_g_x + Rcurr_inv.data[2].y * v_g_y +
Rcurr_inv.data[2].z * v_g_z);

        float z_scaled = 0;

        float Rcurr_inv_0_z_scaled = Rcurr_inv.data[0].z * cell_size.z * intr.fx;
        float Rcurr_inv_1_z_scaled = Rcurr_inv.data[1].z * cell_size.z * intr.fy;

        float tranc_dist_inv = 1.0f / tranc_dist;

        short2* pos = volume.ptr (y) + x;
        int elem_step = volume.step * VOLUME_Y / sizeof(short2);

        //#pragma unroll
        for (int z = 0; z < VOLUME_Z;
            ++z,
            v_g_z += cell_size.z,
            z_scaled += cell_size.z,
            v_x += Rcurr_inv_0_z_scaled,
            v_y += Rcurr_inv_1_z_scaled,
            pos += elem_step)
        {
            float inv_z = 1.0f / (v_z + Rcurr_inv.data[2].z * z_scaled);
            if (inv_z < 0)
                continue;

            // project to current cam
            int2 coo =
            {
                __float2int_rn (v_x * inv_z + intr.cx),
                __float2int_rn (v_y * inv_z + intr.cy)
            };

            if (coo.x >= 0 && coo.y >= 0 && coo.x < depthScaled.cols && coo.y <
depthScaled.rows)         //6
            {
                float Dp_scaled = depthScaled.ptr (coo.y)[coo.x];

                float sdf = Dp_scaled - sqrtf (v_g_z * v_g_z + v_g_part_norm);

                if (Dp_scaled != 0 && sdf >= -tranc_dist)
                {
                    float tsdf = fmin (1.0f, sdf * tranc_dist_inv);

                    bool integrate = true;
                    if ((x > 0 &&  x < VOLUME_X-2) && (y > 0 && y < VOLUME_Y-2)
&& (z > 0 && z < VOLUME_Z-2))
                    {
                        const float qnan = numeric_limits<float>::quiet_NaN();
                        float3 normal = make_float3(qnan, qnan, qnan);

                        float Fn, Fp;
                        int Wn = 0, Wp = 0;
                        unpack_tsdf (*(pos + elem_step), Fn, Wn);
                        unpack_tsdf (*(pos - elem_step), Fp, Wp);

                        if (Wn > 16 && Wp > 16)
                            normal.z = (Fn - Fp)/cell_size.z;
```

```
                        unpack_tsdf (*(pos + volume.step/sizeof(short2) ), Fn,
Wn);
                        unpack_tsdf (*(pos - volume.step/sizeof(short2) ), Fp,
Wp);

                        if (Wn > 16 && Wp > 16)
                            normal.y = (Fn - Fp)/cell_size.y;

                        unpack_tsdf (*(pos + 1), Fn, Wn);
                        unpack_tsdf (*(pos - 1), Fp, Wp);

                        if (Wn > 16 && Wp > 16)
                            normal.x = (Fn - Fp)/cell_size.x;

                        if (normal.x != qnan && normal.y != qnan && normal.z !=
qnan)
                        {
                            float norm2 = dot(normal, normal);
                            if (norm2 >= 1e-10)
                            {
                                normal *= rsqrt(norm2);

                                float nt = v_g_x * normal.x + v_g_y * normal.y +
v_g_z * normal.z;
                                float cosine = nt * rsqrt(v_g_x * v_g_x + v_g_y *
v_g_y + v_g_z * v_g_z);

                                if (cosine < 0.5)
                                    integrate = false;
                            }
                        }
                    }

                    if (integrate)
                    {
                        //read and unpack
                        float tsdf_prev;
                        int weight_prev;
                        unpack_tsdf (*pos, tsdf_prev, weight_prev);

                        const int Wrk = 1;

                        float tsdf_new = (tsdf_prev * weight_prev + Wrk * tsdf) /
(weight_prev + Wrk);
                        int weight_new = min (weight_prev + Wrk,
Tsdf::MAX_WEIGHT);

                        pack_tsdf (tsdf_new, weight_new, *pos);
                    }
                }
            }
        }       // for(int z = 0; z < VOLUME_Z; ++z)
    }      // __global__
  }
}

////////////////////////////////////////////////////////////////////////////////
//////////////////////////////
void
pcl::device::integrateTsdfVolume (const PtrStepSz<ushort>& depth, const Intr&
intr,
```

```
                                const float3& volume_size, const Mat33&
Rcurr_inv, const float3& tcurr,
                                float tranc_dist,
                                PtrStep<short2> volume, DeviceArray2D<float>&
depthScaled)
{
  depthScaled.create (depth.rows, depth.cols);

  dim3 block_scale (32, 8);
  dim3 grid_scale (divUp (depth.cols, block_scale.x), divUp (depth.rows,
block_scale.y));

  scaleDepth << < grid_scale, block_scale >> > (depth, depthScaled, intr);
  cudaSafeCall ( cudaGetLastError () );

  float3 cell_size;
  cell_size.x = volume_size.x / VOLUME_X;
  cell_size.y = volume_size.y / VOLUME_Y;
  cell_size.z = volume_size.z / VOLUME_Z;

  //dim3 block(Tsdf::CTA_SIZE_X, Tsdf::CTA_SIZE_Y);
  dim3 block (16, 16);
  dim3 grid (divUp (VOLUME_X, block.x), divUp (VOLUME_Y, block.y));

  tsdf23<<<grid, block>>>(depthScaled, volume, tranc_dist, Rcurr_inv, tcurr,
intr, cell_size);
  //tsdf23normal_hack<<<grid, block>>>(depthScaled, volume, tranc_dist,
Rcurr_inv, tcurr, intr, cell_size);

  cudaSafeCall ( cudaGetLastError () );
  cudaSafeCall (cudaDeviceSynchronize ());
}

namespace pcl
{
  namespace device
  {
    __global__ void
    tsdf24 (const PtrStepSz<float> depthScaled, PtrStep<ushort2> volume,
            const float tranc_dist, const Mat33 Rcurr_inv, const float3 tcurr,
const Intr intr, const float3 cell_size)
    {
      const int x = threadIdx.x + blockIdx.x * blockDim.x;
      const int y = threadIdx.y + blockIdx.y * blockDim.y;

      if (x >= VOLUME_X || y >= VOLUME_Y)
        return;

      float v_g_x = (x + 0.5f) * cell_size.x - tcurr.x;
      float v_g_y = (y + 0.5f) * cell_size.y - tcurr.y;
      float v_g_z = (0 + 0.5f) * cell_size.z - tcurr.z;

      const float v_g_part_norm = v_g_x * v_g_x + v_g_y * v_g_y;

      float v_x = (Rcurr_inv.data[0].x * v_g_x + Rcurr_inv.data[0].y * v_g_y +
Rcurr_inv.data[0].z * v_g_z) * intr.fx;
      float v_y = (Rcurr_inv.data[1].x * v_g_x + Rcurr_inv.data[1].y * v_g_y +
Rcurr_inv.data[1].z * v_g_z) * intr.fy;
      float v_z = (Rcurr_inv.data[2].x * v_g_x + Rcurr_inv.data[2].y * v_g_y +
Rcurr_inv.data[2].z * v_g_z);

      float z_scaled = 0;
```

```
      const float Rcurr_inv_0_z_scaled = Rcurr_inv.data[0].z * cell_size.z *
intr.fx;
      const float Rcurr_inv_1_z_scaled = Rcurr_inv.data[1].z * cell_size.z *
intr.fy;

      const float tranc_dist_inv = 1.0f / tranc_dist;

      ushort2* voxel = volume.ptr (y) + x;
      const int elem_step = volume.step * VOLUME_Y / sizeof(ushort2);
      const ushort2* end_voxel = volume.ptr (y + VOLUME_Y * VOLUME_Z);

      while (voxel < end_voxel)
      {
        float inv_z = 1.0f / __fmaf_rn (Rcurr_inv.data[2].z, z_scaled, v_z);

        // project to current cam
        int2 coo =
        {
          __float2int_rn (__fmaf_rn (v_x, inv_z, intr.cx)),
          __float2int_rn (__fmaf_rn (v_y, inv_z, intr.cy))
        };

        if (inv_z > 0 && coo.x >= 0 && coo.y >= 0 && coo.x < depthScaled.cols &&
coo.y < depthScaled.rows)         //6
        {
          float Dp_scaled = depthScaled.ptr (coo.y)[coo.x];

          float sdf = Dp_scaled - sqrtf (__fmaf_rn (v_g_z, v_g_z,
v_g_part_norm));

          if (Dp_scaled != 0 && sdf >= -tranc_dist)
          {
            float tsdf = fmin (1.0f, sdf * tranc_dist_inv);

            float tsdf_prev;
            int weight_prev;
            unpack_tsdf (*voxel, tsdf_prev, weight_prev);

            //const int Wrk = 1;

            float tsdf_new = __fmaf_rn (tsdf_prev, weight_prev, tsdf) /
(weight_prev + 1);
            int weight_new = min (weight_prev + 1, Tsdf::MAX_WEIGHT);

            pack_tsdf (tsdf_new, weight_new, *voxel);
          }
        }

        v_g_z += cell_size.z;

        z_scaled += cell_size.z;

        v_x += Rcurr_inv_0_z_scaled;
        v_y += Rcurr_inv_1_z_scaled;

        voxel += elem_step;
      }       // for(int z = 0; z < VOLUME_Z; ++z)
    }      // __global__

  }
}
```

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////
void
pcl::device::integrateTsdfVolume (const PtrStepSz<ushort>& depth, const Intr&
intr,
                                  const float3& volume_size, const Mat33&
Rcurr_inv, const float3& tcurr,
                                  float tranc_dist,
                                  PtrStep<ushort2> volume, DeviceArray2D<float>&
depthRawScaled)
{
  depthRawScaled.create (depth.rows, depth.cols);
  {
    dim3 block (32, 8);
    dim3 grid (divUp (depth.cols, block.x), divUp (depth.rows, block.y));

    scaleDepth << < grid, block >> > (depth, depthRawScaled, intr);
    cudaSafeCall ( cudaGetLastError () );
  }

  {
    float3 cell_size;
    cell_size.x = volume_size.x / VOLUME_X;
    cell_size.y = volume_size.y / VOLUME_Y;
    cell_size.z = volume_size.z / VOLUME_Z;

    dim3 block (Tsdf::CTA_SIZE_X, Tsdf::CTA_SIZE_Y);
    dim3 grid (divUp (VOLUME_X, block.x), divUp (VOLUME_Y, block.y));

    cudaFuncSetCacheConfig (tsdf24, cudaFuncCachePreferL1);

    tsdf24 << < grid, block >> > (depthRawScaled, volume, tranc_dist, Rcurr_inv,
tcurr, intr, cell_size);
    cudaSafeCall (cudaGetLastError ());
  }

  cudaSafeCall (cudaDeviceSynchronize ());
}



namespace pcl
{
    namespace device
    {

    }
}
```