Introduction to the Microsoft Kinect for Computational Photography and Vision

Marina von Steinkirch, steinkirch@gmail.com
State University of New York at Stony Brook

May 9, 2013

Abstract

We introduce the depth sensor Microsoft Kinect and show how to start its use for developing applications on Linux. We discuss the options for frameworks and drivers and show an (original) installation tutorial on Fedora. Finally, we explain the calibration and we introduce tools to manipulate the depth data using point clouds.

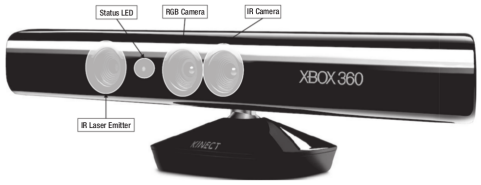

Figure 1: The Kinect hardware (Xbox model).

Introducing the Kinect

Microsoft released the *Kinect for Xbox 360* input-device in November of 2010, as the first large-scale, commercial release of a **depth camera device**(a camera that can see in 3D!). A very similar sensor, now specific for developing, the *Kinect for Windows*, was released in February of 2012. These sensor come with a proprietary algorithms for **feature selection**, **scene analysis**, **motion tracking**, **skeletal tracking**, and **gesture recognition** [1]. In addition, the company that developed the, the *PrimeSense Ltd.* [2], has also released another sensor with the similar technology, the *ASUS Xtion* [3].

Depth Sensing

The depth sensing works with the principle of **structured light**[1]. The system consists of two parts, the **IR laser emitter** and the **IR camera**. The emitter creates a known noisy pattern of structure IR light at 830 nm [4]. The output of the emitter has a pattern of nine bright dots, caused by the imperfect filtering of light[2]. The dots are recorded by the IR camera and then compared to a know pattern, where any disturbances are to be the variations in the surface, detected as closer or further away. The field of view i s 58^o horizontal and 45^o vertical, and the operational range is between 0.8 to 3.5 meters. The sensor has roughly 1 cm depth (z) resolution.

Color pixels are made of different amounts of red, green, and blue, while greyscale images are a single value representing brightness of that pixel. The Kinect provides a color image from its RGB camera and grey-scale image from its depth camera. Both images have a resolution of 640x480 and both cameras operates at a (max) of 30 fps.

[1]Process of projecting a known pattern of pixels on to a scene looking to the way they deform when meeting surfaces, which allows vision systems to calculate the depth and surface information of the objects.

[2]This scattering innovation, allowing a higher powered laser diode to be used and being eye safe was developed by *PrimeSense*, which has a patent on this process.

Available Framework and Drivers

Up to the date, there are two platforms for the Kinect sensor available for Linux [3]:

OpenNI (Open Natural Interaction) [6]:

A middleware delivered from a collaboration of PrimeSense to develop "natural interaction" software and devices. It has skeletal tracking and gesture recognition. However, the OpenNI's features on user tracking (in which an algorithm processes the depth image to determine the position of all of the joints of any users within the camera's range) is not covered by the OpenNI LGPL license. Instead, it is provided by an external module, called NITE which is not available under an open source license.

OpenKinect [7]:

This driver is known by its library name `libfreenect` and was the first Kinect driver available for general use and is fully open-source (dual Apache 2.0/GPL 2.0 license), cross-platform, and derived purely from reverse-engineering efforts. `libfreenect` implements low-level access to the hardware by directly communicating with the Kinect's USB endpoints.

The OpenKinect driver was chosen for this project based upon its simple software interface, low-level hardware access, cross-platform support, and support for multiple languages. In addition, `libfreenect` has the ability to retrieve uncalibrated depth images from the Kinect, for the depth calibration.

Installing the Kinect

Hardware Requirement

All you need to install the Kinect to your computer is a USB 2.0 hub (Kinect will take 70% of it to transmit data), a graphic card capable of handing OpenGL, and a machine that can handle 20MB/second of data. An additional power supply is needed for the Kinect Xbox 360 [4].

[3]The Microsoft SDK is only available for Windows [1].

Installing the OpenKinect on Fedora

1. Install all the dependences (`yum install...`):
   - `git-core, cmake, gcc, gcc++`.
   - `freeglut, freeglut-devel` (library provided with Mesa for creating windows containing OpenGL contexts).
   - `pkg-config` (tool used when compiling applications and libraries).
   - `libXmu, libXmu-devel` (X Window System library).
   - `libXi, libXi-devel` (X Window System client interface to the XINPUT extension to the X protocol).
   - `libusb1, libusb1-devel` (library that gives applications access to USB devices).
2. Create the files "51-kinect.rules" and "66-kinect.rules" in `/etc/udev/rules.d/` (see git repository [8]).
3. Do `git clone https://github.com/OpenKinect/libfreenect.git` and then, inside the `mkdir build` dolder,do `cmake ..`, `make`, and `make install`.
4. `sudo ldconfig /usr/local/lib64/`, to create the links and cache to the shared libraries to the destination.
5. `sudo adduser [YourUserName] video`, to add yourself to the video sudoers list.
6. Finally, test: `sudo glview` or `./bin/glview`. This should show the RGB camera capture and an attempt at fitting the color camera to the 3D space.

Libraries: OpenCV and PCL

OpenCV (Open Source Computer Vision) [9]: Pre-built library of real-time computer vision functions developed by Intel in the 90s. The library has over 500 functions in different areas of vision and image analysis including: gesture recognition, facial recognition, motion tracking, and 3D vision. It's the base of my cameras' calibration.

PCL (the Point Cloud Library) [10] PCL handles data acquired from many modern visual sensors. The scene is created by points, and each point contains a position in space (x,y,z) and optional RGB color or gray-scale. Point clouds create a three dimensional picture and PCL merges point-clouds together, filtering out uninteresting data points, identifying key points in a scene, and sorting data into tree-hierarchies.

A bridge between the OpenKinect driver and PCL is in my git repo [8].

Calibration

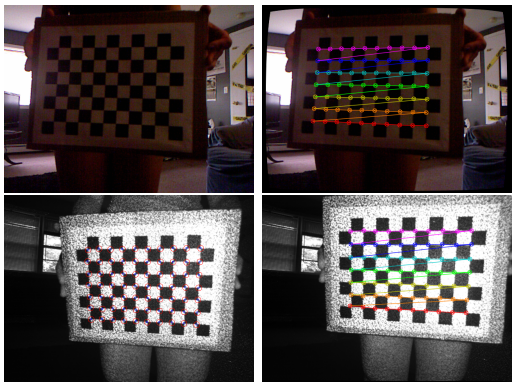

Figure 2: My calibration board. (top) Here we see the process with the IR emitter blocked, with the corners. (bottom) calibration process for the IR image.

Without calibration, the color and depth outputs from the cameras are useless data. The camera's intrinsic and extrinsic parameters are factory set and it is necessary to determine a translation from the camera frame to the world frame. The intrinsic parameters are: focal length, camera sensor size, lens distortion measured as a radial distortion, and image plane displacement measured as displacement from the optical axis. In the other hand, the extrinsic parameters are the camera location (vector) and the camera rotation (as Euler angles). The cameras are calibrated using the *pinhole model*, where the view of the scene is created by projecting a set of 3D points onto the image plane via perspective transformation.

We can create a depth annotation of a chessboard rig by physically offsetting the chessboard from its background. The offset creates sharp edges in depth along the perimeter of the chessboard mounting which remain fixed with respect to the chessboard's interior corners. By coinciding the chess- board's exterior corners with the sharp edges in depth, there is no need to measure the positioning of the chessboard with respect to the mounting; only the dimension of a single chessboard square is needed.

The calibration steps for the IR and the RGB cameras we proceed as follow:

1. I created a calibration target: a chessboard printed on a4.pdf, where each square is 0.023 m^2, as in Fig. 2.
2. I took 20 with the IR emitter blocked (for the stereo camera calibration) and 20 with none of the cameras blocked, making sure the corners of the target are always in the image and getting different angles and placements.
3. I run a code written with the OpenCV calibration functions (see git repo [8]) for the above images, aiming a pixel reprojection error smaller than 1. The OpenCV library contains built-in support for locating the interior corners of a chess-board calibration rig (e.g. `findChessboardCorners`). The set of pairings between raw depth values and object depths obtained this way does the calibration of raw depth values to meters.
4. In the last step, I generated a `yml` calibration file, that can be used in applications (e.g. OpenCV). Opening the viewer with this file I see the calibrated image: not distorted at the edges and the distances are accurate (Fig. 3). The calibration file contain:

   `rgb-intrinsics` and `depth-intrinsics`: the camera's intrinsic parameters in pixel units, a matrix with the principle point (usually the image center), $c_{x,y}$ and the focal length, $f_{1,2}$

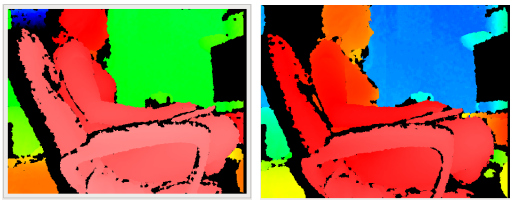

Figure 3: Depth image results from the cameras calibration (left) me in my desk in a uncalibrated camera and (right) much happier and calibrated after.

   `rgb-distortion` and `depth-distortion`: lenses also have some distortion (radial, $k_{1,2,3}$, and tangential, $P_{1,2}$).

R and T: the translation and rotation of the projected point from the world into a coordinate frame in reference to the camera.

$$\begin{bmatrix} r_1 1 & r_1 2 & r_1 3 & t_1 \\ r_2 1 & r_2 2 & r_2 3 & t_2 \\ r_3 1 & r_3 2 & r_3 3 & t_3 \end{bmatrix} \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

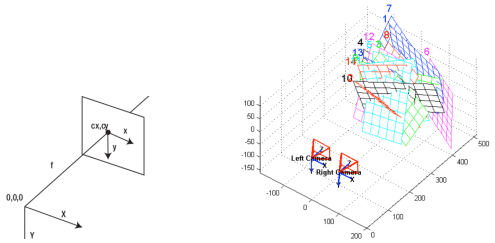

Figure 4: (top) The camera intrinsic matrix and the combined R-T matrix, (bottom) the pinhole model and stereo calibration [4].

2D and 3D Data

A first analysis of the depth data is treating it as if it were two-dimensional. For each pixel in the depth image, we think of its position within the image as its (x,y) coordinates plus a grayscale value. This last value corresponds to the depth of the image in front of it, e.g. this value will represent the pixel's z-coordinate. The depth map returns a flat array that contains 307,200 (or 640 times 480) integers arranged in a single linear stack, Fig. 6.

$$\underbrace{\begin{pmatrix} x \\ y \\ w \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ W \end{pmatrix} = \begin{pmatrix} f_x X + c_x Z \\ f_y Y + c_y Z \\ Z \end{pmatrix}}_{\text{Camera projection 3D to 2D}} \qquad \underbrace{\begin{aligned} X &= \frac{(x - c_x)Z}{f_x} \\ Y &= \frac{(y - c_y)Z}{f_y} \end{aligned}}_{\text{Inverse relation for X and Y}}$$

Figure 5: Transformations between 2D and 3D data.

Once we convert all the two-dimensional grayscale pixels into three-dimensional points in space, we have a *point cloud*, i.e. many disconnected points floating near each other in three-dimensional space in a way that corresponds to the arrangement of the objects and people in front of the Kinect, Fig. 7.

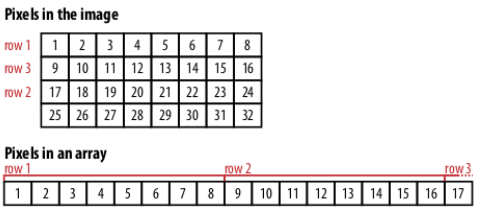

Figure 6: Depth data as a flat array.

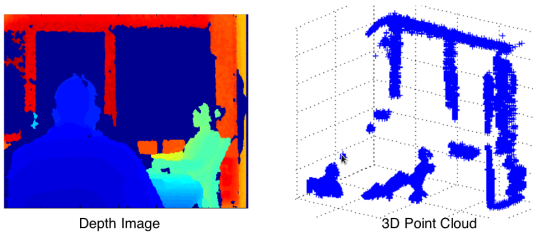

Figure 7: Depth data with point clouds.

Outline and Next Steps

This paper intended to be an introduction to how to set and calibrate the Microsoft Kinect for Linux, namely in the Fedora distribution. The described framework allows the development of many applications (and powerful!) on computer photography and computer vision. My project results, which is beyond the scope of this paper, explore some and can be acquired from the git repository [8].

References

[1] *http://www.microsoft.com/en-us/kinectforwindows/*

[2] *http://www.primesense.com/*

[3] *http://www.asus.com/Multimedia/Xtion_PRO/*

[4] *Hacking the Kinect*, Apress, 2012

[5] *Making Things See*, 2012

[6] *http://www.openni.org/*

[7] *http://openkinect.org*

[8] *https://bitbucket.org/steinkich/kinect-hacks-and-projects*

[9] *http://opencv.org/*

[10] *http://pointclouds.org/*