Integrated DVS recording and tracking system in ROS

ADVANCED SEMINAR

submitted by
Jianjie Lin
Siqi Wang

NEUROSCIENTIFIC SYSTEM THEORY
Technische Universität München

Prof. Dr Jörg Conradt

Supervisor: Dipl.-Ing. Lukas Everding
Final Submission: 06.07.2016

In your final hardback copy, replace this page with the signed exercise sheet.

Abstract

Dynamic vision sensors, an advanced type of cameras which have independent pixel and able to send events so long as they detect bright changes. 3D motion capture system, which can track object's exact position with fixed couples of cameras on the walls inside a room, should be integrated with dynamic vision sensors. After combination, what we are ought to get are information of moving object's orientation and position in a constructed 3D space. In order to do a real time monitoring, synchronization is necessary. The original code in C/C++ languages for both systems are provided, and what needs to be done is converting codes for both systems to the ones can be successfully applied in Robot Operating System. Later, output all data measured from both systems together at the same time. In this paper, authors mainly talk about working principle and how to execute new code after integration.

Contents

Chapter 1

Introduction

Traditional vision sensors capture objects by a series of successive frames. Huge amount of useless information will be brought by the successive frames and such information will occupy and waste RAM, disk space, energy, time and computational power. In the meantime, every pixel has the same exposure time period, which makes these vision sensors cannot be able to work properly in dark or extremely bright environment. eDVS, which is the short form of dynamic vision sensors, is a new type of camera with capability to solve such problems. eDVS cameras use a patented technology which based on biologically inspired principle to work like our retina: the pixels operate individually and only the events like changing of local pixel-level caused by movement will be sent. By using eDVS, intensity changes can be reported as a stream of events which include the pixel address, the time instant of intensity change and its sign. In other words, brightness change on the edges of any moving object can be detected by eDVS cameras. Although eDVS screen can only display images in black, gray and white colors, in motion objects capturing point of view, eDVS performs much better than common cameras.
In many situations, only the data recorded from eDVS is not enough to fulfill the requirement. We also want to capture the exact position and orientation in space of the object like what the traditional cameras do. Actually, we are able to achieve such goal by applying a 3D motion capture system. This tracking system consists of eight cameras, which track the sensing points on the eDVS camera. The sensing points will be treated as a rigid body and displayed in a three-dimensional space constructed before. The origin point is a fixed point formed when the space model was built. Hence, the distance along hypothetic x, y, z coordinations can be easily measured. Generally speaking, main task for us in this project is to integrate two systems mentioned above and synchronize them. In addition, this new integrated system should be operated on ROS platform. With this new integrated system, users are able to get information detected from both eDVS recording and tracking systems at any particular instant time.

Chapter 2

Integrated DVS recording and tracking system in ROS

2.1 Convert the System into ROS

The Robot Operating System (ROS) is a framework for writing robot software, which involves libraries and different tools to achieve the goal of creating complex robotic behaviors. We choose ROS as an intermediate platform for the combination of eDVS and tracking system mainly due to the following three reasons: distributed nature, inter-platform operability and high efficiency when executing complex robotic commands. Distributed nature of ROS makes it possible for external users to create and publish their own packages, which greatly extends the usage and applications of ROS. Up to now, there are more than 3000 ROS packages announced and exposed to public by their programmers. These packages have a wide range, cover nearly everything either in concept proof or industrial applications. ROS has the property of inter-platform operability, which means programmers can work between very different subsystems that are probably running in different programming languages. Such property is extremely suitable for our case, which is the combination of two systems. Since distributed message system is used to connect different parts of ROS, the whole system will not crash even if there is one component crashes. So robot is more likely to continue working when it is controlled by ROS. That makes ROS become stable even if doing some huge or complex processes
As already mentioned in introduction session, this project involves two systems: eDVS-System and tracking-system, which are separately wrote in C/C++ languages, are needed to be converted to be applied in ROS system. The Figure 2.1 shows the basic GUI of the tracking system. In order to achieve this, the first step is to find out all necessary libraries in both systems. Basically, the two super libraries are needed: “tflog_libs” and “Edvs”.
In addition, under the library “tflog_libs”, there are five different sub libraries, which are interconnected with each other.
For each library, the command “add_library” is required. For example: Adding the

library “npoint” into the *CMakelist.txt*, the command should be like the following:

```
add_library(npoint
        src/tflogs_libs/npoint/src/npoint_conn.c
        src/tflogs_libs/npoint/src/npoint_logger.c
        src/tflogs_libs/npoint/src/npoint_packet.c
        )
```

Since interconnections exist between different libraries, only add files themselves will not be enough. And this may lead to the compiling errors as well. The proper method to fix the problem is to add related files from other libraries.

```
add_library(npoint
        src/tflogs_libs/npoint/src/npoint_conn.c
        src/tflogs_libs/npoint/src/npoint_logger.c
        src/tflogs_libs/npoint/src/npoint_packet.c
        src/tflogs_libs/conns/src/dvs_conn.c
        src/tflogs_libs/conns/src/tcp_client.c
        src/tflogs_libs/conns/src/udp_link.c
        src/tflogs_libs/conns/src/udp_multicast_recv.c
        src/tflogs_libs/conns/src/udp_server.c
        src/tflogs_libs/utils/src/utils.c
        )
```

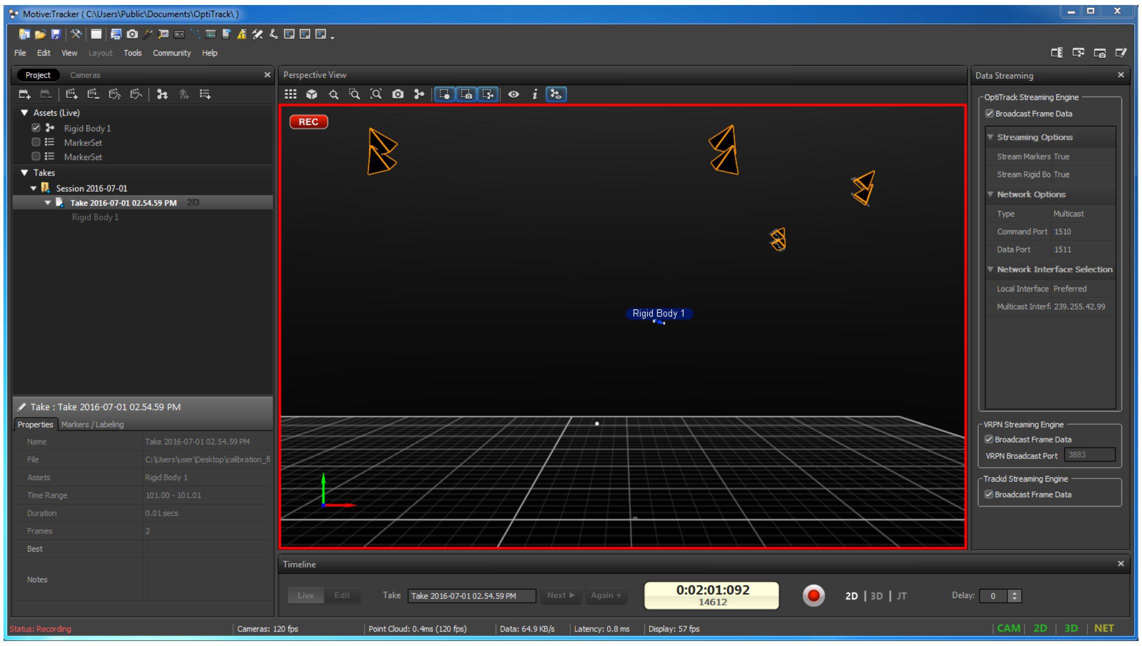
The other libraries are also needed to be wrote according to this format. Command “add_library” should be always followed by “add_dependencies”, in order to tell compiler the library’s name. e.g. “npoint”.

```
add_dependencies(npoint
        ${catkin_EXPORTED_TARGETS}
)
```

Finally, all the libraries can be linked by command “target_link_libraries”:

```
target_link_libraries(npoint
        ${catkin_LIBRARIES}
)
```

For more details about the *CMakelist.txt*, readers can refer to ROS official website. The way to compile the ROS is by using the command “catkin_make”. But before doing such behavior, the computer in the tracking room is needed to switch to the bash mode. It means that, in the terminal, command “bash” is typed.



Assets (Live): Rigid Body 1, MarkerSet, MarkerSet
Takes: Session 2016-07-01, Take 2016-07-01 02.54.59 PM, Rigid Body 1

Take : Take 2016-07-01 02.54.59 PM

Name	Take 2016-07-01 02.54.59 PM
File	C:\Users\user\Desktop\calibration_fi
Assets	Rigid Body 1
Time Range	101.00 - 101.01
Duration	0.01 secs
Frames	2
Best	
Notes	

OptiTrack Streaming Engine: Broadcast Frame Data ✓

Stream Markers	True
Stream Rigid Bo	True
Type	Multicast
Command Port	1510
Data Port	1511
Local Interface	Preferred
Multicast Interf.	239.255.42.99

VRPN Streaming Engine: Broadcast Frame Data ✓, VRPN Broadcast Port 3883
Trackd Streaming Engine: Broadcast Frame Data ✓

Take: Take 2016-07-01 02.54.59 PM Delay: 0

Status: Recording | Cameras: 120 fps | Point Cloud: 0.4ms (120 fps) | Data: 64.9 KB/s | Latency: 0.8 ms | Display: 57 fps

Figure 2.1: The GUI of the tracking system

2.2 ROS publisher and subscriber

In this project, timing and position information in both systems are interested. In addition, orientation information from tracking system is also necessary. It is obvious that converting code must be prior to combining two systems. There are many small but important points need to be treated carefully when doing the conversion. Since valid data types in C++ and ROS are different, a new data type should be defined when converting original C/C++ code to ROS ones. For example, "float" is valid in C++ while not in ROS, it should be replaced by "float64/32" in this case. In order to publish the information got from both systems, three message files ("dvs_dara_type.msg","dvs_message.msg" and "synchron.msg") have to be created. Since only the position and timing information are interested, in the message file only the variables *pos_x*, *pos_y*, *pos_z*, *quan_x*, *quan_y*, *quan_z*, *quan_w* and *time_end* are included. If there is any new requirement, readers can also insert the new variable accordingly. Readers can refer to ROS official website on how to write *CMakelist.txt*. As mentioned in the section 2.1, the advantage of the ROS-system is that, programmers can arbitrarily subscribe and publish the message from other nodes and topics. Because of the independence of two systems: eDVS and tracking system, we have to use two nodes and two topics for communication.
By means of two topics in node "Sender", different kinds of information from both systems can be published separately. Later, subscriber "listener" will receive those information, so that if there are some further requirements, it would be much easier to develop them. Figure 2.2 is a demonstration of the relationship between the

publisher and subscriber. Topic “Tracking_system” is used to publish the message

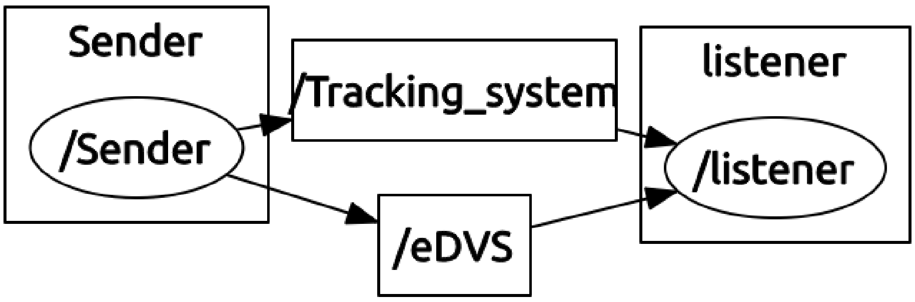


Figure 2.2: Computation Graph of Software Nodes

from the tracking system, similarly, topic “eDVS” is used to publish the message from the eDVS information. It needs to be pointed out that the sampling rate from both systems are different, so that the publish rate for two topics are also different. In this project, the baud rate of eDVS is 12000000 Hz, while the sampling rate of the tracking system is only 0.2 Hz. The procedure to run the node is like following:

1. open a terminal: roscore
2. open a new terminal: cd catkin_ws
3. rosrun synchronization synchronization_node
   if this node is not founded, the command “source ./devel/setup.bash” is needed to type in the same terminal
4. open a new terminal : rosrun synchro_listener synchro_listener_node

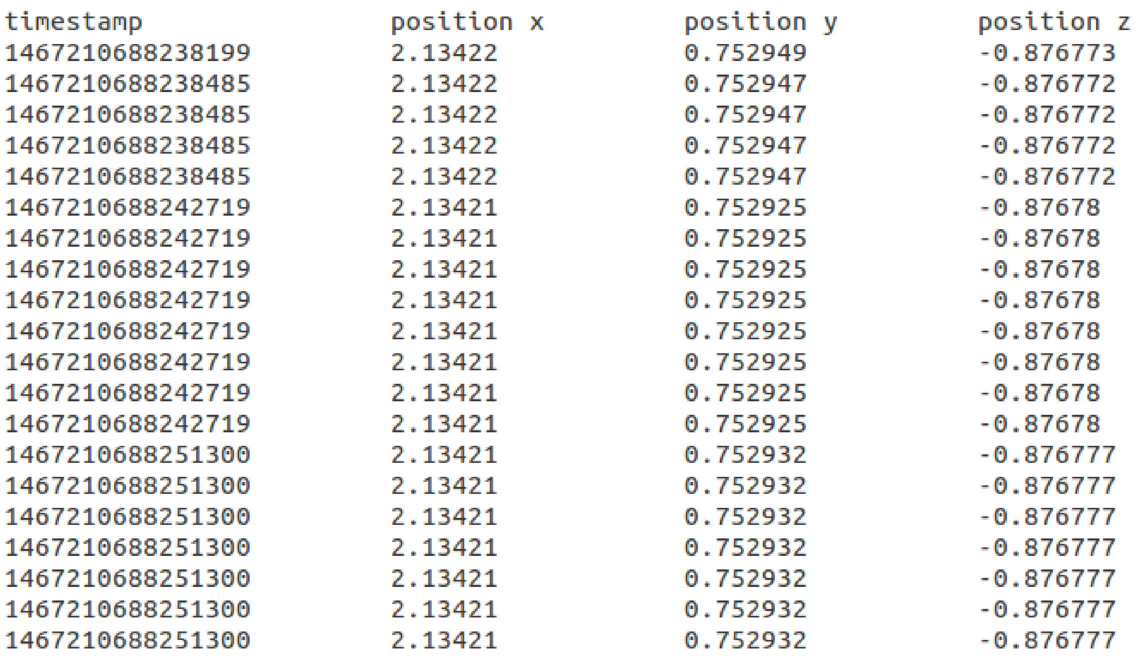
2.3 Synchronization among tracking and eDVS system

In order to combine eDVS and tracking system, firstly, synchronization is needed since it’s a bridge connects both systems. Initially, time recorded in tracking system is according to real time by a function called “gettimeofday()”. That means whenever users open the tracking system, time recording is always corresponding to real time of the computer. However, in eDVS, time recording starts to count at the moment eDVS is triggered. Therefore, every time users start the eDVS program, timer inside the program will initialize and record time from zero. So the two systems have different timing references. The approach is changing time recording method of eDVS to real time reference as well. The procedures of this approach are introduced as following:

1. The timestamps from eDVS is needed to be reset, before checking whether the stream is open or not. In the meantime, the real time at this moment is stored in the variable “reset_time”.

2. After the resetting, the eDVS counts time start from zero. The new timestamps is now added to “reset_time”: dvs_msg.dvs_x=(events[e].t+reset_time).

After these two implementations, in principle, both systems will have the same timing reference, so that synchronization can be realized. The Figure 2.3 and 2.4 show the result. Although both timestamps are already in the real time, they still



```
timestamp               position x          position y          position z
1467210688238199        2.13422             0.752949            -0.876773
1467210688238485        2.13422             0.752947            -0.876772
1467210688238485        2.13422             0.752947            -0.876772
1467210688238485        2.13422             0.752947            -0.876772
1467210688238485        2.13422             0.752947            -0.876772
1467210688242719        2.13421             0.752925            -0.87678
1467210688242719        2.13421             0.752925            -0.87678
1467210688242719        2.13421             0.752925            -0.87678
1467210688242719        2.13421             0.752925            -0.87678
1467210688242719        2.13421             0.752925            -0.87678
1467210688242719        2.13421             0.752925            -0.87678
1467210688242719        2.13421             0.752925            -0.87678
1467210688242719        2.13421             0.752925            -0.87678
1467210688251300        2.13421             0.752932            -0.876777
1467210688251300        2.13421             0.752932            -0.876777
1467210688251300        2.13421             0.752932            -0.876777
1467210688251300        2.13421             0.752932            -0.876777
1467210688251300        2.13421             0.752932            -0.876777
1467210688251300        2.13421             0.752932            -0.876777
1467210688251300        2.13421             0.752932            -0.876777
```

Figure 2.3: Data from tracking system

have a $10e^4\,\mu s$ difference. Here are several explanations:

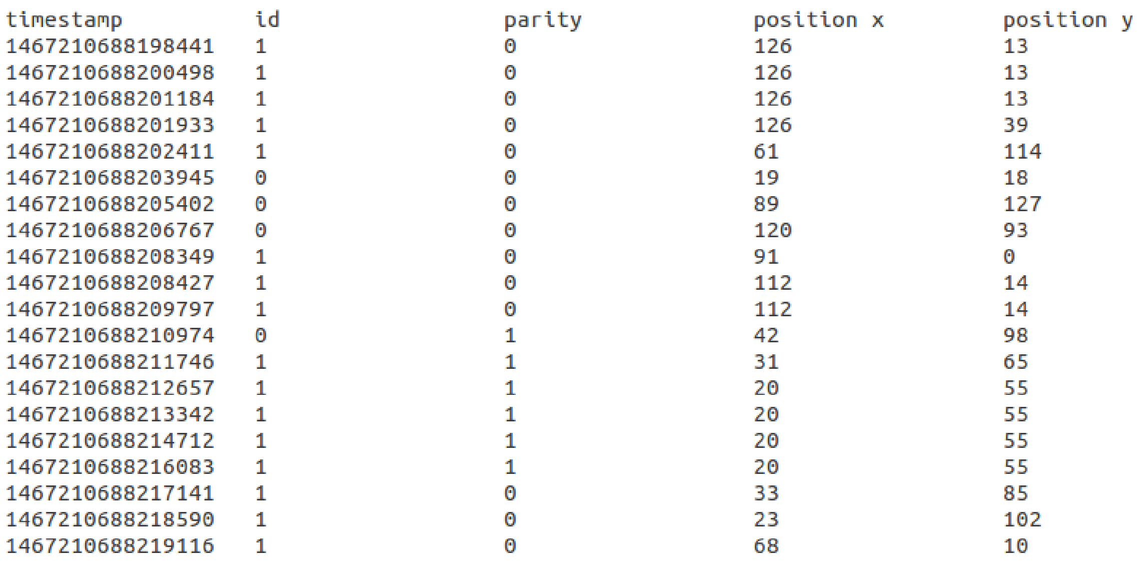
1. the sampling rate from the eDVS and tracking system is extremely different. The baud rate from the eDVS is 12000000 bit/s, or rather 12 MHz, while the sampling rate from the tracking system is 0.2 Hz.

2. Delay always exists in the eDVS and tracking system, which can also lead to such result.

The modification to make the difference smaller can be done in the future.

2.4 Camera calibration

Distortion is recognized to be the easiest aberration since it deforms the image as whole. Since straight lines in the object space are rendered as curved lines by sensors, the name curvilinear distortion is frequently encountered. The most commonly encountered distortions are radially symmetric, or approximately so, arising from the



```
timestamp           id                parity            position x        position y
1467210688198441    1                 0                 126               13
1467210688200498    1                 0                 126               13
1467210688201184    1                 0                 126               13
1467210688201933    1                 0                 126               39
1467210688202411    1                 0                 61                114
1467210688203945    0                 0                 19                18
1467210688205402    0                 0                 89                127
1467210688206767    0                 0                 120               93
1467210688208349    1                 0                 91                0
1467210688208427    1                 0                 112               14
1467210688209797    1                 0                 112               14
1467210688210974    0                 1                 42                98
1467210688211746    1                 1                 31                65
1467210688212657    1                 1                 20                55
1467210688213342    1                 1                 20                55
1467210688214712    1                 1                 20                55
1467210688216083    1                 1                 20                55
1467210688217141    1                 0                 33                85
1467210688218590    1                 0                 23                102
1467210688219116    1                 0                 68                10
```

Figure 2.4: Data from eDVS

symmetry of a photographic lens. These radial distortions can usually be classified as either barrel distortions or pincushion distortions. Camera calibration and reflection is already done routinely for “normal”, frame-based cameras. Therefore, there are many available library to tackle such tasks.

2.4.1 Calibration Model

Pinhole camera model is the fundamental of the model. Plumb Bob distortion model, which is a simple model with distortion in radial and tangential directions is used as an extended version by the calibration.
A standard 3 × 3 matrix shows focal length (fx, fy) and principal point (cx, cy) is used as the intrinsic camera matrix.

$$K = \begin{bmatrix} s_x & s_\theta & 0_x \\ 0 & s_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where s_x,s_y is the scalar to convert the point in the metric unit to the pixel coordinates, s_θ corresponds to a skew factor, and $(o_x\, o_y)$ are the coordinates (in pixels) of the principle point relative to the image reference frame. In this project, the intrinsic camera matrix is defined in the file “undistortDVSPoints.hpp” with the value:

$$K = \begin{bmatrix} 233 & 0 & 69.583 \\ 0 & 232.19 & 53.021 \\ 0 & 0 & 1 \end{bmatrix}$$

Rectification and projection matrix can be provided by calibration in the stereo

case.
The rectification matrix is the rotation matrix that aligns the camera coordinate systems to the ideal stereo image plane so that epipolar lines are parallel. The projection matrix is a 3×4 stereo extension of the intrinsic matrix. The position of the second camera's optical center will be added into the first camera's image frame.
Details information can be found in the CameraInfo ROS message documentation.
In a three-dimensional space, a point located in $[X, Y, Z]^T$, the (x, y) projection of this point on the rectified image is given by:

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \mathrm{P} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}, \text{ where } x = u/w, \text{ and } y = v/w.$$

2.4.2 Calibration method

As mentioned above, many available libraries can be used for camera calibration, e.g. openCV, matlab. In this project, the openCV is applied. The event from the eDVS with the position information $events[e].x$ and $events[e].y$ are both integers. In order to convert the integer to float type, we need to apply the intrinsic camera matrix K. In the main.cpp file, both type of position information are available. If one wants the uncalibrated position information, $events[e].x$ is suitable. If one wants the calibrated position information, dst_.at <double>(e,0), which has already been calibrated by openCV .

Chapter 3

Conclusion

eDVS, camera that only sensitive to brightness change, cannot get any information about position. To improve and solve such problem, tracking system were introduced. Therefore, the project goal is to integrate eDVS and tracking systems, later, information collected from two systems are ought to be published. To make the combination of two systems user friendly, data collection should be done together for both systems. That means synchronization is also required. According to the project requirement, the new system after integration should be suitable for Robot Operating System (ROS). Therefore, code converting is also a part of the project. So far we have already finished what we wanted to do at the beginning of the project, and achieved the aim of the project successfully. Both of the data collection and synchronization have already been realized within one new integrated system executed on ROS.
Here the command to run the program in the tracking room by chair of Neuroscientific System Theory is presented :

1. Every time a terminal is opened, check if the terminal is bash mode. If not, type "bash".
2. open a terminal: roscore
3. open a new terminal:
   (a) cd catkin_ws,
   (b) source ./devel/setup.bash
   (c) catkin_make (if no code is changed, then it's no neeed to do again)
   (d) rosrun synchronization synchronization_node
4. open a new terminal: rosrun synchro_listener synchro_listener_node

List of Figures

License