Integrated Vision Framework for a Robotics Research and Development Platform

by

Julián Hernández Muñoz

B.S. in Computer Science and Engineering
Massachusetts Institute of Technology (2009)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2011

Certified by .....
Dr. Cynthia Breazeal
Associate Professor of Media Arts and Sciences
Thesis Supervisor

Accepted by .....
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Integrated Vision Framework for a Robotics Research and Development Platform

by

Julián Hernández Muñoz

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2011, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents the design of a vision framework integrated into a robotics research and development platform. The vision system was implemented as part of the software platform developed by the Personal Robots Group at the MIT Media Lab. Featuring representations for images and camera sensors, this system provides a structure that is used to build robot vision applications. One application shows how to merge the representations of two different cameras in order to create a camera entity that provides images with fused depth-color data. The system also allows the integration of computer vision algorithms that can be used to extract perceptual information from the robot's surroundings. Two more applications show detection and tracking of human face and body pose using depth-color images.

Thesis Supervisor: Dr. Cynthia Breazeal
Title: Associate Professor of Media Arts and Sciences

Acknowledgments

I would like to start by thanking Dr. Cynthia Breazeal for giving me the opportunity of being part of the Personal Robots Group and for guiding me through the completion of this thesis. Almost fifteen years ago I read a newspaper article about Kismet and her, and it inspired me to venture into the field of engineering. It has been an honor working with her on this project.

I would also like to thank Philipp Robbel and Siggi Adalgeirsson, who have helped me from the moment I started at the group. To Philipp, for taking me as his UROP and advising me in early projects. To Siggi, for always lending a hand whenever I needed it.

Thanks to Dr. Peter Szolovits, my longtime academic advisor, for his support and mentoring throughout my bachelors and masters. I would also like to thank Dr. Berthold K.P. Horn and Dr. Antonio Torralba for sparking my interests in computer vision.

Thanks to my parents, Omar and Arzoris, for always being a source of great inspiration and for their constant motivation to strive for excellence. To my brother and sister, Javier Arturo and Arzoris Marian, for giving me the support that only they can provide.

Finally, I would like to thank the Institute for six rewarding years. Getting an education from MIT is indeed like taking a drink from a fire hose.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Robotics is a fast-growing field in academia, industry, and open-source communities, with many universities and corporations joining efforts in order to create frameworks that promote and advance the development of robotic systems. These systems are usually composed of both hardware and software platforms. As the field moves forward, it becomes necessary to describe the components that these platforms should have.

A vision framework is required by any system that needs to acquire, handle, and process images from cameras. In the same way cameras are typical sensors in robots, a vision framework should be a common component in any robotics software platform. Color cameras, for example, are widely available and the primary choice for the "eyes" of a robot. Other types of cameras, like depth cameras, are becoming cheaper and more accessible.

This thesis provides a technical description of the vision system integrated into `r1d1`, the Java software platform developed and used by the Personal Robots Group at the MIT Media Lab. The `r1d1` platform defines a cognitive architecture for interactive agents that allows designing real-time interactions between robots and persons. It features, among other things, flexible real-time integration of multiple perceptual inputs.

With this vision system, `r1d1` applications have access to flexible image and camera representations. The image representation abstracts the source of the image, while the camera representations abstract the process of image acquisition. The modularity

of these entities allows building applications without worrying about writing code to interface with the cameras. This system is introduced in Chapter 2.

Furthermore, the vision system lets the user define new representations by possibly combining existing ones. Chapter 3 explores how to fuse the data captured from two different types of cameras by merging their representations. The goal is to combine 2-dimensional information from a color camera with 3-dimensional information from a depth camera in order to produce fused depth-color images.

The motivation for merging these two sensors comes from the camera setup found in the second generation of the MDS robotic platform developed by the Personal Robots Group. The MDS is a humanoid robot that combines mobility, dexterity, and social interaction abilities [12]. In its head, the MDS has a color camera and a depth camera, both positioned in the forehead one below the other.

The vision framework also integrates computer vision algorithms that can be used to gain perceptual information from the robot's surroundings. Chapter 4 presents face and body tracking capabilities that work on top of the image and camera representations. These algorithms serve to exemplify the potential of the proposed vision system.

Although the framework is described as part of `r1d1`, its design is meant to be modular and non-platform dependent. Most of the direct dependencies with other `r1d1`'s components occur to take advantage of the intra-application communication system built into `r1d1`. In general, the core concepts in the design of this vision system can be extracted and implemented in other research and development platforms.

Chapter 2

Integrated Vision System

This chapter presents an integrated vision system for the `r1d1` robotics research and development platform. The proposed system introduces new representations for images and cameras in the `r1d1` environment with designs that prove to be modular and flexible. In this system, not only images can vary in size, pixel depth, and color model, but they can be easily manipulated both within and outside `r1d1` through applications with access to the native system memory.

Moreover, this system presents a core structure for a camera that can be replicated into different camera types, including, as shown in this chapter, a color camera and a 3D time-of-flight camera. These representations are flexible enough to allow a camera be either a device directly connected to the computer running `r1d1`, or a device connected remotely that transfers the image data through the network.

Section 2.1 of this chapter discusses the image representation in the integrated vision system, while Sections 2.2 and 2.3 discuss the base camera representation. Sections 2.4 and 2.5 show the flexibility of the system by presenting the implementations for two types of cameras. Finally, the last three sections of the chapter, Sections 2.6, 2.7, and 2.8, further discuss some of the entities used in the cameras' implementations.

2.1 Image Buffer

The `ImageBuffer` class represents an image in the `r1d1` environment. An instance of this class is defined by four fields. The first three are the image's width, height, and pixel depth. The fourth field can be either the image's color model or its number of channels.[1] The different possible combinations of values for these fields allow a flexible description of an image inside the programming environment. Tables 2.1 and 2.2 show the available values for the depth and color model fields, as described by the `ImageBuffer.Depth` and `ImageBuffer.ColorModel` enum types.

Table 2.1: Pixel depth values in the `ImageBuffer.Depth` enum

`ImageBuffer.Depth`

Depth	Description
`BYTE`	Unsigned 8-bit integer
`SHORT`	Unsigned 16-bit integer
`FLOAT`	32-bit floating-point single precision
`DOUBLE`	64-bit floating-point double-precision

Table 2.2: Color model values in the `ImageBuffer.ColorModel` enum

`ImageBuffer.ColorModel`

Color Model	Description
`BGR`	Blue-Green-Red channels
`BGRA`	Blue-Green-Red-Alpha channels
`RGB`	Red-Green-Blue channels
`RGBA`	Red-Green-Blue-Alpha channels
`GRAY`	Grayscale channel

In an image buffer, the image data is stored in an underlying `java.nio.ByteBuffer` object. Part of the flexibility of the `ImageBuffer` class is attributed to the versatility of a byte buffer with respect to the data it can hold. Furthermore, the byte buffer within an image buffer is created to be direct. A direct byte buffer in Java allocates its space in native system memory, outside of the Java virtual machine's memory

[1] Not all images require a color model. An example is the depth image acquired by a range sensor.

heap. This allows other programs outside Java to use native code to access the same allocated memory space. An example of the advantage of using direct buffers is the implementation of an interface between `r1d1` and the OpenCV computer vision open source library. This interface enables performing OpenCV functions directly onto the Java images through native code.

Pixels in an image buffer are stored row by row with interleaved channels, starting at the top-left pixel. The methods to access and manipulate the image data are specific to the image's pixel depth. For example, the methods of the form `getBytePixel` are used to access pixels in an image of depth `BYTE`. The different versions of the same getter method trade off convenience for performance. Getter methods that accept an array as input speed up a loop through the image's pixels by avoiding the overhead of creating the output array on every call.

2.2 Camera

The `Camera` abstract class is the base representation of a camera in the `r1d1` environment. As listed in Table 2.3, this class defines methods to access information about a camera (`getWidth`, `getHeight`, `getTimestamp`, `getFrameRate`), methods to capture images from the camera (`startCapturing`, `grabImage`), and methods to perform camera calibration (`startCalibration`).

Table 2.3: Public methods in the `Camera` class

Camera
startCapturing
startCalibration
grabImage
getWidth
getHeight
getTimestamp
getFrameRate

By providing an abstract implementation, the `Camera` class lets its subclasses de-

scribe different types of cameras while specifying a common structure for them. In this framework, `ColorCam` and `SwissRangerCam` are the main examples of extending the `Camera` class to represent cameras that capture different types of data: the first one represents a conventional color camera while the second one represents a 3D time-of-flight camera (see Sections 2.4 and 2.5). This design allows a flexible definition of what a camera is and lets the user choose the representation of the data captured by a specific sensor.

Furthermore, an object of type `Camera` is a subtype of `DataProvider`. A data provider is an object that can publish user defined datatypes in `r1d1`'s intra-application communication system, allowing other objects called data handlers to subscribe to and receive the provided data. In the case of a `Camera` object, the provided data is represented by the abstract class `CameraData`. Subclasses of `CameraData` define the datatypes captured by different camera sensors. For example, `ColorCamData` and `SwissRangerCamData` represent the data provided by `ColorCam` and `SwissRanger-Cam`, respectively.

2.3 Camera Receiver

The `CameraReceiver` abstract class defines the base structure of the mechanism that will acquire and handle data from the camera sensors. It implements two `r1d1` interfaces. The first one is the `iUpdateable` interface, which describes classes that have an `update` method. The main loop of an `r1d1` application calls `update` on all the objects that have been registered as updatable. For the camera receiver this means that the `update` method can be used to grab an image from a camera on every loop.

The second interface used by the `CameraReceiver` class is the `iDataRecord-Handler`. This interface is part of `r1d1`'s intra-application communication system. It is used to define a class that can handle `iDataRecord` objects being sent from another application. A class that implements this interface has a `handleDataRecord` method that handles the incoming data record objects. The camera receiver can use this method to receive records containing image data.

Table 2.4 lists the two abstract methods declared in the `CameraReceiver` class. These methods are used to start the image receiving process and query the receiver if it has image data available to be accessed or retrieved.

Table 2.4: Public methods in the `CameraReceiver` class

CameraReceiver
startReceiving isDataAvailable

2.4 Color Camera

The `ColorCam` class extends `Camera` to represent a color camera. In the `r1d1` environment, a color camera can be a camera sensor connected directly to the computer running the system, or a camera sensor connected to some other device that sends the images over the network to `r1d1`. This versatility is provided by `ColorCam` through the `ColorReceiver` class. While the color receiver is in charge of acquiring and handling the images from the camera, a color camera provides access to the image buffer, the image view, and the camera calibration tool. Additionally, it provides the implementation of all abstract methods defined by the `Camera` class. Table 2.5 lists the methods that a color camera adds to the base camera representation.

Table 2.5: Public methods in the `ColorCam` class

ColorCam extends Camera
getImage getImageView getCalibrationTool

There are two ways of accessing the camera's images. One way is through the `ColorCam` object's internal image buffer, which stores a copy of the last grabbed image. This image buffer is obtained by calling the `getImage` method. The second way takes

advantage of r1d1's intra-application communication system and consists of subscribing to the data objects of type ColorCamData through the DataHandler interface. The handleData method of the data handler receives the data objects that contain an image buffer with the copy of the grabbed image.

Therefore, every time a call to grabImage returns successfully (i.e. returns a true value) two things happen: (1) the acquired image is stored in the internal image buffer and (2) it is published through the DataProvider interface. Accessing the images from the image buffer gives the user more precision when it is necessary to process the image right after it is grabbed. Receiving a published image contains the overhead of going through the data dispatcher mechanism, but simplifies the communication between r1d1 applications.

The ColorCam class adds a second version of the grabImage method. This second version takes as input a value that represents the desired timestamp for the grabbed image. This method is useful when synchronizing two or more cameras: given the timestamp of an image from one of the cameras this grabImage method can be used to retrieve the closest corresponding image from the other cameras.

A color camera also provides the option of displaying the image stream using r1d1's graphics display system. This is achieved by instantiating an ImageView object linked to the color camera's internal image buffer. Every time an image is acquired the image view needs to be refreshed in order to display the new data on the buffer. This image view is accessed by calling the getImageView method.

Finally, the methods to perform camera calibration and retrieve the camera's parameters are available through the CalibrationTool object that is associated with every instance of ColorCam. Upon the creation of the color camera, a calibration tool object is constructed and linked to the camera's internal image buffer. When the startCalibration method is called, the calibration tool is started. Then, the user can use the methods in the calibration tool, obtained by calling the getCalibrationTool method, to calibrate the camera (see Section 2.6).

The following sections present and discuss the modules that compose the ColorCam class. The module dependency diagram in Figure 2-1 shows how these classes are

The file `ColorCam.ps` hasn't been created from `ColorCam.dot` yet.
Run '`dot -Tps -o ColorCam.ps ColorCam.dot`' to create it.
Or invoke LaTeX with the `-shell-escape` option to have this done automatically.

Figure 2-1: `ColorCam` module dependency diagram. Arcs with white arrows represent subtype relations (A ▷ B = A extends B) while arcs with black arrows represent implementation relations (A ▶ B = A uses B). Gray rectangles represent abstract classes. The `ImageBuffer` class is omitted from this diagram.

related to each other.

2.4.1 Color Receiver

The `ColorReceiver` class extends `CameraReceiver` to provide the mechanism that interfaces with a color camera. It allows the user to acquire images directly from a Firewire camera or to receive the color images via a network transfer. Image acquisition from the camera is achieved using the libdc1394 API, a library of functions to control Firewire cameras that conform to the 1394-based Digital Camera Specifications [4]. The network transfer is achieved through IRCP, the network communication protocol used by `r1d1` [5]. Table 2.6 lists the methods that the `ColorReceiver` class adds to the camera receiver representation.

Table 2.6: Public methods in the `ColorReceiver` class

ColorReceiver extends CameraReceiver
getImage handleDataRecord update

Upon initialization, the color receiver first tries to connect to a Firewire camera. The `update` method is in charge of acquiring the images using the `DC1394JavaAcquire` methods. If the color receiver does not find a camera or if it fails when attempting to connect to one, it will proceed to start the IRCP network connection by setting up a `ColorPacketHandler` (the user has the option of forcing this network connec-

tion). As discussed in Section 2.3, the images received by the packet handler are sent to the color receiver through the `iDataRecordHandler` interface. The color receiver calls the `handleDataRecord` method to handle the incoming data record objects.

The `ColorReceiver` class is flexible with the type of image it can handle. The image type is specified by a set of public static variables that the user can modify. Table 2.7 contains the list of all the variables available to the user. When running on libdc1394 mode the user can change the size of the image captured from the camera. When running on network mode the user can modify the image size, pixel depth, and color model according to what is being transferred over the network.

Table 2.7: User-modifiable static variables in the `ColorReceiver` class

`ColorReceiver`

Variable	Description
`WIDTH`	The image width
`HEIGHT`	The image height
`DEPTH`	The image pixel depth
`COLOR_MODEL`	The image color model
`CAMERA_GUID`	The camera GUID
`DC1394_FRAME_RATE`	The camera capturing frame rate
`DC1394_ISO_SPEED`	The camera ISO speed
`UNDISTORT`	Flag to undistort the image
`COMPRESSED_IMAGE`	Flag if image is compressed
`STREAM_CAPACITY`	The receiver image stream size
`VISION_MINOR_TYPE`	The IRCP minor type
`PACKET_HANDLER_NAME`	The packet handler name
`PACKET_HANDLER_KEY`	The packet handler key

Besides handling a real time image stream, the color receiver can store a user-defined number of last seen images in memory. The mechanism that processes the image stream works like a First-In-First-Out queue: fresh images are stored at the tail of the stream while old images are retrieved from the head of the stream. The size of the stream determines the number of images that can be saved in memory. This system allows to synchronize two or more cameras by finding in their streams the images with matching (or closest) timestamp. The `ImageStream` class contains the implementation of this mechanism (see Section 2.7).

2.4.2 Color Packet Handler

The `ColorPacketHandler` class is a subtype of `SafePacketHandler`, a class part of the receiving end in the IRCP communication system. The `ColorReceiver` object creates a color packet handler to manage the packets sent over the network that correspond to image data from a color camera. Therefore, the color packet handler needs to know how the color information is encoded in the incoming packet.

In order to manage a large number of sent packets, some of which can get lost or arrive out of sequence, the color packet handler implementation uses a `PacketOrganizer` object to organize the incoming data (see Section 2.8). The packet organizer takes the received packets, which consist of subparts of the whole image, and informs the color packet handler once there are images ready to be retrieved.

Table 2.8 contains the method that the color packet handler overrides from the safe packet handler class.

Table 2.8: Public methods in the `ColorPacketHandler` class

`ColorPacketHandler` extends `SafePacketHandler`
`safeHandle`

2.4.3 DC1394 Java Acquire

The `DC1394JavaAcquire` class establishes the Java interface to the libdc1394 library. Since libdc1394 is written in C, this class uses the Java Native Interface (JNI) framework to access the library functions. The native methods declared in the Java layer are implemented in a separate C layer, and this implementation is packaged in the `r1d1` JNI library called libdc1394JavaAcquire.

The `DC1394JavaAcquire` class simplifies the access to a Firewire camera by declaring the methods listed in Table 2.9. The `getGUIDs` method returns a list of GUIDs from the available Firewire cameras, and `startCapturing`, `stopCapturing`, and `getImage` are used to control and acquire images from them. The last two methods in the list

are specific to Firewire cameras produced by Point Grey, and they are used to convert captured grayscale images to color images.

Table 2.9: Public methods in the DC1394JavaAcquire class

DC1394JavaAcquire
getGUIDs
startCapturing
stopCapturing
getImage
retrievePointGreyBayerTile
convertPointGreyMonoToBGR

2.5 SwissRanger Camera

The SwissRangerCam class and its components are very similar to the classes that describe a color camera in r1d1. This proves that the system can represent cameras following a similar core structure. Like ColorCam, the SwissRangerCam class extends Camera. In this case it represents a SwissRanger sensor, a 3D time-of-flight camera produced by Mesa Imaging [10]. The SwissRanger can capture depth information (z-coordinates) from a scene, as well as x-coordinates and y-coordinates for each of the depth points. Associated with this data is the amplitude and the confidence map, which provide measurements of the quality of the acquired data.

In the representation, the SwissRanger can be connected directly to the computer running the system, or it can be connected to some other device that sends the images over the network to r1d1. This versatility is provided through the SwissRangerReceiver class. While the SwissRanger receiver is in charge of acquiring and handling the images from the camera, the SwissRanger camera provides access to the image buffers, the image views, and the camera calibration tool, as well as the implementation of all abstract methods defined by the Camera class. Furthermore, it contains the information about the modulation frequency on which the SwissRanger is running, and static methods to create visualizations for the depth and amplitude data. Table

2.10 lists the methods that the SwissRanger class adds to the base camera representation.

Table 2.10: Public methods in the `SwissRangerCam` class

SwissRangerCam extends Camera
getDepth getAmplitude getX getY getConfidenceMap getDepthDisplay getAmplitudeDisplay getDepthView getAmplitudeView getCalibrationTool getModulationFrequency createDisplays threshold toMeters

There are two ways of accessing the image data captured by the SwissRanger. One way is through the camera object's internal image buffers, which store copies of the last grabbed images. The depth data is obtained by calling the `getDepth` method, and the amplitude data is obtained by calling `getAmplitude`. The x-coordinate, y-coordinate, and confidence map data are obtained by calling `getX`, `getY`, and `getConfidenceMap`, respectively.

The second way takes advantage of `r1d1`'s intra-application communication system and consists of subscribing to the data objects of type `SwissRangerCamData` through the `DataHandler` interface. A SwissRanger data object contains the same number of image buffers as described before, each with a copy of the last grabbed image. The `handleData` method of the data handler receives these data objects.

Similar to the `ColorCam` class, when the `grabImage` method returns successfully the acquired images are stored in the internal image buffers and then are published through the `DataProvider` interface. Accessing the images directly from the image

buffers avoids the overhead of going through the data dispatcher mechanism and allows the user to process the images right after they are grabbed. However, receiving the images published through the data dispatcher simplifies the communication between `r1d1` applications.

Furthermore, the `SwissRangerCam` class also adds a second version of the `grabImage` method. The reason for this method becomes clearer now. A Firewire color camera could be synchronized with a SwissRanger, but differences in the devices and their capturing mechanisms might not give the user the same capturing frame rate nor the same timestamp for images captured at the same time. Since this second version of the `grabImage` method takes as input a desired timestamp for the grabbed image, it is possible to get a timestamp from one of the cameras and retrieve the closest corresponding image from the second camera taking into account a timestamp offset.

The `SwissRangerCam` class provides the option of displaying visualizations for the depth image stream and the amplitude image stream. The methods `getDepthDisplay` and `getAmplitudeDisplay` return image buffers with BGR color model and `BYTE` pixel depth containing the image data of the visualizations. These image buffers are linked to `ImageView` objects and are obtained through the `getDepthView` and `getAmplitudeView` methods. The method `createDisplays` is used to create the visualizations from the raw depth and amplitude data.

The `CalibrationTool` class is again used to provide the methods that perform camera calibration and retrieve the camera's parameters. Since the amplitude image is visually close to a grayscale image, the calibration tool object is linked to the internal amplitude image buffer of the `SwissRangerCam` instance. The user can access this object through the `getCalibrationTool` method, and then calibrate the camera by using the methods discussed in Section 2.6.

The following sections present and discuss the modules that make up the `SwissRangerCam` class. Figure 2-2 shows a module dependency diagram that indicates how these classes are related to each other.

The file SwissRangerCam.ps hasn't been created from SwissRangerCam.dot yet.
Run 'dot -Tps -o SwissRangerCam.ps SwissRangerCam.dot' to create it.
Or invoke LaTeX with the -shell-escape option to have this done automatically.

Figure 2-2: SwissRangerCam module dependency diagram. Arcs with white arrows represent subtype relations (A ▷ B = A extends B) while arcs with black arrows represent implementation relations (A ▶ B = A uses B). Gray rectangles represent abstract classes. The ImageBuffer class is omitted from this diagram.

2.5.1 SwissRanger Receiver

The SwissRangerReceiver class extends CameraReceiver to provide the mechanism that interfaces with a SwissRanger camera. It allows the user to acquire images directly from a SwissRanger or to receive depth, amplitude, x-coordinate, y-coordinate, and confidence images via a network transfer. Image acquisition from the camera is achieved using the SwissRanger driver API that is provided by Mesa Imaging. The network transfer is achieved through IRCP. Table 2.11 lists the methods of the SwissRanger receiver class.

Table 2.11: Public methods in the SwissRangerReceiver class

SwissRangerReceiver extends CameraReceiver
getDepth getAmplitude getX getY getConfidenceMap handleDataRecord update

Similar to the color receiver, the SwissRanger receiver first tries to connect to a SwissRanger camera. The update method, called on each system loop, grabs the images from the camera using the SwissRangerJavaAcquire methods. If the receiver fails to connect to a camera it proceeds to start the IRCP network connection by setting up a SwissRangerPacketHandler (again, the user has the option of forcing a

network connection). The images received by the packet handler are sent to the SwissRanger receiver, which calls the `handleDataRecord` method to handle the incoming data record objects.

The `SwissRangerReceiver` class is flexible with the type of image it can handle and it provides a set of public static variables that the user can modify. However, in practice these variables are not changed because the SwissRanger sensor itself does not provide this flexibility. Table 2.12 contains the list of all the variables available to the user. As this table shows, the user has the option of thresholding the depth image. This thresholding is performed based on depth, amplitude, and confidence map values.

Table 2.12: User-modifiable static variables in the `SwissRangerReceiver` class

`SwissRangerReceiver`

Variable	Description
WIDTH	The image width
HEIGHT	The image height
DEPTH	The image pixel depth
NUMBER_OF_CHANNELS	The image number of channels
MODULATION_FREQUENCY	The modulation frequency of the camera
INITIAL_DEPTH_HIGH_THRESHOLD	The initial depth high threshold
INITIAL_DEPTH_LOW_THRESHOLD	The initial depth low threshold
INITIAL_AMPLITUDE_THRESHOLD	The initial amplitude threshold
INITIAL_CONFIDENCE_THRESHOLD	The initial confidence threshold
MAX_AMPLITUDE_THRESHOLD	The maximum threshold for the amplitude
MAX_CONFIDENCE_THRESHOLD	The maximum threshold for the confidence
UNDISTORT	Flag to undistort the image
THRESHOLD	Flag to threshold the depth image
STREAM_CAPACITY	The receiver image stream size
VISION_MINOR_TYPE	The IRCP minor type
PACKET_HANDLER_NAME	The packet handler name
PACKET_HANDLER_KEY	The packet handler key

The SwissRanger receiver handles the different image streams using the First-In-First-Out queue mechanism used by the color receiver: fresh images are stored at the tail of the stream while old images are retrieved from the head of the stream. This allows the receiver to store a user-defined number of last seen images in memory and to synchronize two or more cameras by locating in their streams the images with match-

ing (or closest) timestamp. Like in the ColorReceiver class, the implementation of this mechanism is provided by an ImageStream object (see Section 2.7).

2.5.2 SwissRanger Packet Handler

The SwissRangerPacketHandler class contains the information about how the SwissRanger data is encoded in incoming network packets. It is a subtype of SafePacketHandler, and it is instantiated by the SwissRanger receiver in order to manage the SwissRanger image data transferred over the network.

Similar to the ColorPacketHandler, this class uses a PacketOrganizer in its implementation to sort and organize the large number of incoming packets (see Section 2.8). The packet organizer puts together the received images' pieces and informs the SwissRanger packet handler once they are ready to be retrieved. Table 2.13 lists the method that the SwissRanger packet handler overrides from the SafePacketHandler class.

Table 2.13: Public methods in the SwissRangerPacketHandler

SwissRangerPacketHandler extends SafePacketHandler
safeHandle

2.5.3 SwissRanger Java Acquire

The SwissRangerJavaAcquire class establishes the Java interface to the SwissRanger driver. It uses the Java Native Interface (JNI) framework to access the native functions in the driver's library. The r1d1 Java layer declares native methods that are implemented in a C layer packaged in the r1d1 JNI library called libsrJavaAcquire.

The SwissRangerJavaAcquire class simplifies the access to a SwissRanger camera by declaring the method listed in Table 2.14. Unlike DC1394JavaAcquire, this class does not declare startCapturing nor stopCapturing methods. The driver's library does not contain functions equivalent to these operations since the camera is

capturing from the moment it is created. The `getImages` method is used to retrieve the images for all data types at the same time.

Table 2.14: Public methods in the `SwissRangerJavaAcquire` class

`SwissRangerJavaAcquire`
`getImages`

2.6 Calibration Tool

The `CalibrationTool` class provides methods to calibrate a camera, undistort the camera images using the output of the calibration, and save the output into a file. A calibration tool object is created by providing the image buffer associated with the camera that is to be calibrated. Table 2.15 lists all methods available in the `CalibrationTool` class.

The implementation of the `CalibrationTool` is based on the calibration functions of the OpenCV open source library, which in turn are based on Jean-Yves Bouguet's implementation of Zhang's calibration method [1, 17]. The calibration process is started by calling the `start` method, and at any time it can be restarted by calling `reset` followed by `start`. The process consists of detecting a checkerboard pattern (Figure 2-3) on multiple images until acquiring a preset number of calibration images. The information about the position of the checkerboard's corners is extracted from the images and then fed to the calibration algorithm. The algorithm for one loop of the calibration process, which runs on each user call to the method `calibrate`, is described in Table 2.16.

The algorithm starts by checking if the count of acquired images is less than the number of required images. If it is, it calls the `findAndDrawCorners` method to search the image for the position of the calibration pattern's corners (line 3). If the complete set of corners is found, `addCornersToList` is called in order to save their positions into a list (line 6), and then the count of acquired images is incremented (line 7). There-

Table 2.15: Public methods in the `CalibrationTool` class

CalibrationTool
start
reset
isCalibrating
calibrate
findAndDrawCorners
addCornersToList
calibrateCamera
initUndistortMap
undistort
setCameraParameters
setChessboardDimensions
setChessboardSquareSize
setRequiredSamples
setWindowDimensions
getCameraParameters
getCameraParametersPath
getChessboardNumberOfColumns
getChessboardNumberOfRows
getChessboardSquareSize
getRequiredSamples
getWindowWidth
getWindowHeight
loadCameraParameters
saveCameraParameters
getVectorFromUndistortedCameraImage

fore, the user must call `calibrate` until the number of acquired images reaches the number of required images. Once it reaches it, an additional call to `calibrate` runs the camera calibration algorithm by calling the `calibrateCamera` method (line 9). At any point of the calibration process the user can call `isCalibrating` to check if the calibration tool is waiting for more images.

Once the camera calibration is performed, the results are saved into an object of type `CameraParameters` which is accessed through the `getCameraParameters` method. This object holds the intrinsic parameters (focal length, principal point, distortion coefficients) and extrinsic parameters (rotation and translation vectors) that describe the

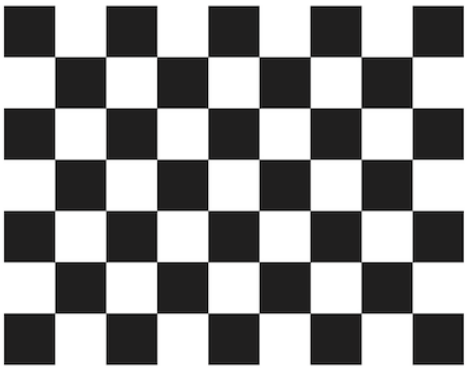

Figure 2-3: Checkerboard pattern used for camera calibration

Table 2.16: Algorithm for the `calibrate` method in `CalibrationTool`

```
CALIBRATE (currentImages, requiredImages):
1  If (currentImages < requiredImages)
2     Then:
3        Search checkerboard pattern in the image;
4        If the pattern is found
5           Then:
6              Extract the corners and save them;
7              Increment currentImages counter;
8     Else:
9        Run calibration algorithm with saved corners;
```

camera. The camera parameters can be saved into or read from an XML file using the `saveCameraParameters` and `loadCameraParameters` methods, respectively.

Finally, the `CalibrationTool` class provides the methods to undistort the camera images given the camera's parameters values. First, the x-coordinate and y-coordinate pixel undistortion maps need to be created by calling the `initUndistortMap` method. This method assumes that the camera parameters object has been set, either by running the calibration process, loading the parameters from a file, or setting the object using the `setCameraParameters` method. After initializing the undistortion maps, each call to the `undistort` method will undistort the image in the calibration tool's image buffer.

2.7 Image Stream

The ImageStream class is used by the ColorReceiver and SwissRangerReceiver classes to handle the image streams from the cameras (see Sections 2.4.1 and 2.5.1). Section 2.7.1 describes the superclass of ImageStream, the Stream abstract class. Section 2.7.2 describes the ImageStream class itself.

2.7.1 Stream

The Stream abstract class represents a stream of buffer objects. A buffer object can be any mutable object used to store data. Since the Stream class is defined using a generic type declaration, it allows the user decide what will the buffer object be in an specific stream implementation.

When deciding on a buffer object, the user has to make sure it can be created and manipulated using the abstract methods listed in Table 2.17. These methods should be implemented by the user in the Stream's subclasses, where T is substituted by the type of the buffer object. The buffer argument in the writeBuffer and readBuffer methods represents the buffer where data is going to be written or from where data is going to be read, respectively. Conversely, the data argument represents the data that will be written into the buffer or where the data read from the buffer will be copied into, respectively.

Table 2.17: Abstract methods that define the buffer object in the Stream class

Stream
T createBuffer() writeBuffer(T buffer, T data) readBuffer(T buffer, T data)

The Stream object is created with a finite capacity that remains constant during the lifetime of the object. Therefore, it creates a finite amount of buffer objects that are instantiated only once and always reused afterwards. The Stream class' design allows to operate similar to a First-In-First-Out queue, with the incoming data being added

at the tail of the stream and the old data being polled from the head of the stream. Furthermore, the data is available to be retrieved only when all the buffers in the stream have been filled, ensuring in that way that all buffers contain meaningful continuous data that the user can use at any given time.

The `Stream` class contains the implementation of the public methods listed in Table 2.18. It also provides some non-public helper methods that are used by the subclasses to add, poll, and peek the stream. The user must declare in the subclasses the formal methods to access the data from the stream according to the required design.

Table 2.18: Public methods in the `Stream` class

Stream
isDataAvailable
capacity
get
clear

2.7.2 Image Stream

The `ImageStream` class extends `Stream` to represent a stream of images. An `ImageBuffer` (see Section 2.1) is used as the buffer object to holds image data. Consequently, the constructor of an `ImageStream` requires as argument the width, height, pixel depth, and color model information that describes the type of image it needs to handle.

Given the specified capacity for the stream, an `ImageStream` object will instantiate that number of image buffers, and will only use those during the lifetime of the object. The advantage of this design becomes clear when considering that the `ImageBuffer` class performs the expensive operation of allocating space in native memory. Instead of reallocating that space multiple times, the buffer is created once and reused afterwards.

An image stream behaves like a First-In-First-Out queue. The images recently acquired by a camera are added at the tail of the stream using the `add` method listed in Table 2.19. When the stream has reached maximum capacity, each call to `add` polls an

old image from the head of the stream before adding the new image at the tail. In this way, at any given time, the stream stores a number of consecutive images equal to the capacity of the stream.

Table 2.19: Public methods in the `ImageStream` class

`ImageStream` `extends Stream`
`add` `peekLast`

The `ImageStream` class provides the `peekLast` method to access the most recently added image from the stream. Given that the stream holds a history of past images, the user has realtime access to the image data by using the `peekLast` method. A second version of the `peekLast` method takes as input the desired timestamp for the retrieved image, and searches in the stream for the image with matching (or closest) timestamp. This method is used by the `ColorReceiver` and `SwissRangerReceiver` classes to get image data given a specified timestamp (see Sections 2.4.1 and 2.5.1).

2.8 Packet Organizer

When sending packets through the network, packets might arrive at the receiver end out of order, or they might get dropped and not be received at all. Furthermore, the sender might divide the packets into smaller parts, and send those to the receiver in the form of subpackets. The receiver must sort the subpackets and know when the whole packet has been received. The receiver should also decide when to stop waiting for a dropped subpacket.

The `PacketOrganizer` class provides a mechanism to handle these situations. When creating a packet organizer, the user decides how many buffers will be collecting subpackets at the same time, each buffer being assigned to a different packet. A received subpacket must contain header information about its position in the subpacket

stream, its length, the timestamp of the packet to which it belongs, and the total number of subpackets that form this packet.

Table 2.20: Public methods in the `PacketOrganizer` class

`PacketOrganizer`

`put`
`getRecord`
`getBuffer`
`ready`
`releaseBuffer`

Table 2.20 lists the methods available in the `PacketOrganizer` class. The `put` method is used to insert a received subpacket into its corresponding packet buffer. The `ready` method indicates if a packet buffer has received all the expected subpackets. Once a packet buffer is ready, the methods `getRecord` and `getBuffer` are used to retrieve the data from that buffer. The user should copy the data out of the buffer because buffers are reused during the lifetime of the packet organizer. Once the data has been retrieved, the method `releaseBuffer` tells the packet organized that the buffer can be used to collect subpackets from another incoming packet.

Chapter 3

Depth-Color Fusion

Chapter 2 presented an integrated vision system that introduces representations for images and camera sensors in the `r1d1` development platform. In this chapter these representations are used to create new vision-related entities. It focuses specifically on merging the representations of the color and 3D time-of-flight cameras with the goal of creating a new camera that provides color and depth information.

To achieve this, it is necessary to describe an algorithm that finds correspondences between points in the color camera and points in the 3D camera. The first part of this chapter, Section 3.1, gives an overview of the setup and discusses the type of data acquired by these cameras. Then, Section 3.2 provides a detailed description of the depth-color fusion algorithm, using as reference the work by Linarth *et al.* [9]. This algorithm consists of extracting the cameras' parameters through synchronized camera calibration, computing the relative transformation between both cameras, and finding the correspondences between color and depth pixels.

Finally, Section 3.3 discusses the implementation of the fusion algorithm as part of the integrated vision system. This section introduces the new depth-color camera representation, as well as the entities that are needed to perform the synchronized camera calibration and the fusion between the depth and color images.

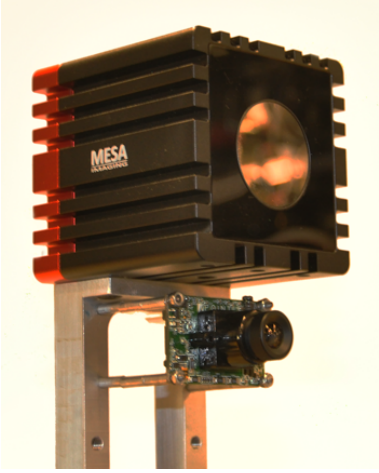

Figure 3-1: SwissRanger depth camera (top) and Firefly color camera (bottom) setup.

3.1 Cameras Setup

Figure 3-1 shows a setup of two camera sensors fixed on a frame, one below the other. The camera at the bottom is a Firefly, a Firewire camera manufactured by Point Grey [13]. It captures grayscale images that can be converted to color using functions from the libdc1394 library (see Section 2.4.3). The size of the images captured by a Firefly is 640×480 pixels.

The camera on top is a SwissRanger SR4000, a 3D time-of-flight camera manufactured by Mesa Imaging. It captures x-coordinate, y-coordinate, and z-coordinate information from the scene in front of the camera. This chapter focuses on the z-coordinate data, which will be referred to from this point on as the *depth image*.[2] The SwissRanger also produces an amplitude image, which corresponds to the amplitude of the modulated signal used in the depth measurements. The amplitude image can be used both for generating a grayscale image representing the captured scene and for measuring the quality of the depth values [10]. The size of all images captured by the SwissRanger is 176×144 pixels.

Depending on the lens in the Firefly, the field of view of the color camera can be inside the field of view of the SwissRanger, or vice versa. Since the Firefly is more flexible in the matter of changing its field of view, this setup uses a wide angle lens on the

[2] Similarly, the SwissRanger camera will also be referred to as the *depth camera*.

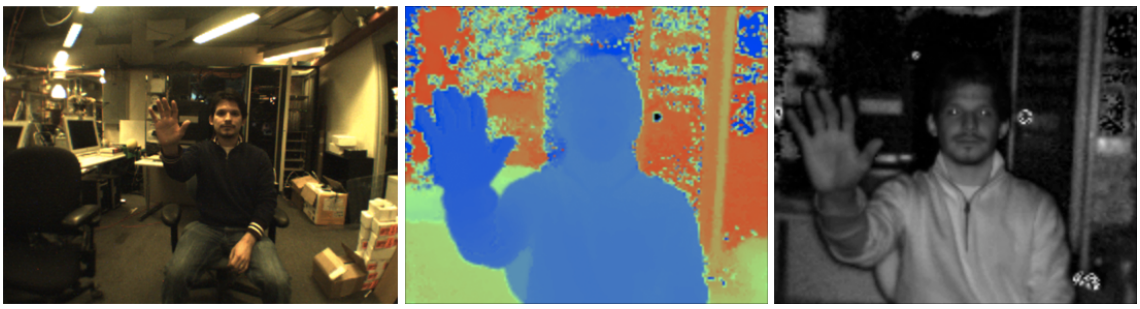

Figure 3-2: Image types captured by the depth-color camera setup. From left to right: color image, depth image, and amplitude image. Notice the difference between the field of views of both cameras.

color camera that makes it encompass the field of view of the depth camera. Figure 3-2 shows an example of the three types of images captured by this depth-color camera setup.

3.2 Fusion Algorithm

The goal of the depth-color fusion algorithm is to find the correspondences between the pixels of a depth image and the pixels of a color image that have been acquired with a depth-color camera setup like the one described in Section 3.1. The first step of the algorithm is to retrieve the intrinsic and extrinsic parameters from both cameras. This is achieved by performing synchronized camera calibration.

Given the cameras' parameters, the next step computes the relative transformation between the depth and color cameras, which is described in terms of a rotation and a translational offset. The relative transformation is used to transform world points in the depth camera's coordinate system to world points in the color camera's coordinate system. The correspondences between points in the depth image and points in the color image are retrieved from an image point to world point transformation, followed by a world point to image point transformation.

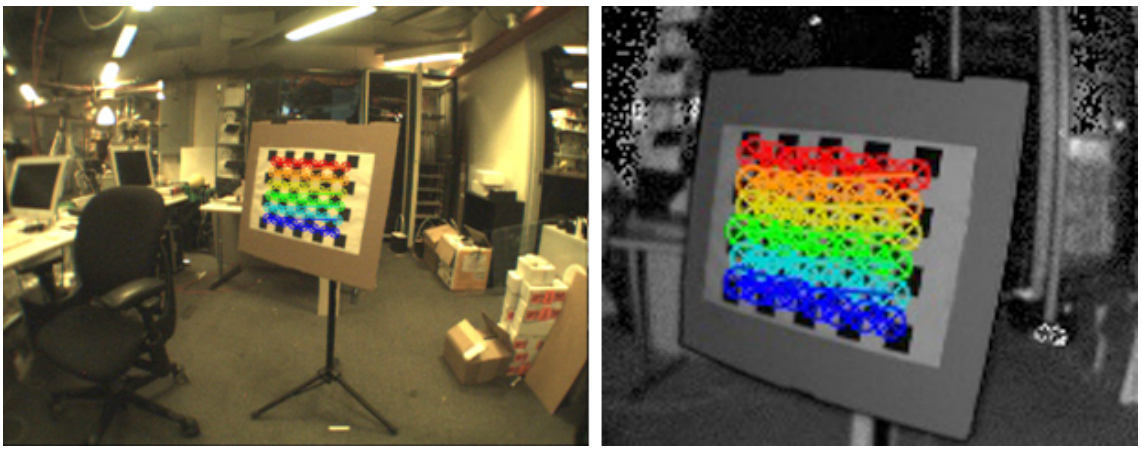

Figure 3-3: Example of a calibration image pair. An image pair is composed of a color image (left) and an amplitude image from the depth camera (right).

3.2.1 Cameras Calibration

The depth and color cameras need to be calibrated in order to retrieve their intrinsic and extrinsic parameters. Since the goal is to find how the cameras are positioned and oriented one with respect to the other, the calibration of the cameras needs to be performed synchronously. In other words, there needs to exist a positional correspondence between the calibration images taken with both cameras. This is achieved by setting the calibration pattern in a fixed position with respect to the camera setup and capturing one image from each camera at the same time. The two synchronized images are called an *image pair* (see Figure 3-3).

As observed in Figure 3-3, the image pair consists of a color image and an amplitude image. Since the amplitude image is visually similar to a grayscale image, it is possible to use it in the calibration process to find the calibration pattern. The colored circles in the figure indicate the location of the corners in the pattern (see Section 2.6). Besides knowing the correspondence between the images in an image pair, the synchronized calibration algorithm needs to know the correspondence between the calibration points in the two images.

The calibration process described in Section 2.6 computes the camera's focal length and principal point. If f is the focal length of a camera, and m_x, m_y are the ratios of pixel width and pixel height per unit distance, respectively, then (f_x, f_y) represents the

focal length expressed in units of horizontal and vertical pixels. That is,

$$(f_x, f_y) = (f\,m_x, f\,m_y) \tag{3.1}$$

Furthermore, (x_0, y_0) represents the principal point of the camera. Assuming that the skew coefficient between the x and y axes is zero, the intrinsic matrix of this camera is given by

$$\mathbf{K} = \begin{bmatrix} f_x & 0 & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

The calibration process also provides a rotation matrix and a translation vector. These parameters describe the transformation between the world's coordinate system and the camera's coordinate system. If $\mathbf{R}$ denotes the 3×3 rotation matrix, and $\mathbf{t}$ denotes the 3×1 translation vector, the extrinsic matrix of this camera is given by the 3×4 matrix

$$\begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \tag{3.3}$$

Therefore, the synchronized calibration algorithm outputs two intrinsic matrices, $\mathbf{K}_d$ and $\mathbf{K}_c$, and two extrinsic matrices, $[\mathbf{R}_d\ \mathbf{t}_d]$ and $[\mathbf{R}_c\ \mathbf{t}_c]$, where the d and c subscripts are used to differentiate between the depth camera's parameters and the color camera's parameters. While the intrinsic matrices contain information about the internals of the camera, the extrinsic matrices hold information about how the cameras are positioned in space. This information is used in the next section to compute the relative transformation between both cameras.

3.2.2 Relative Transformation

When computing the relative transformation between the cameras, the direction of the transformation is chosen to be from the depth camera to the color camera. As dis-

cussed in Section 3.1, the field of view of the depth camera is within the field of view of the color camera. Therefore, every point in the depth image will have a corresponding point in the color image, but not necessarily vice versa.

If $\mathbf{P} = (X, Y, Z)^T$ is a point in world coordinates, the position of $\mathbf{P}$ in the depth camera's coordinate system is given by $\mathbf{q}_d$. Similarly, the position of $\mathbf{P}$ in the color camera's coordinate system is given by $\mathbf{q}_c$. The points $\mathbf{q}_d$ and $\mathbf{q}_c$ can be expressed in terms of the cameras' extrinsic parameters by equations (3.4) and (3.5), respectively.

$$\mathbf{q}_d = \mathbf{R}_d\mathbf{P} + \mathbf{t}_d \tag{3.4}$$

$$\mathbf{q}_c = \mathbf{R}_c\mathbf{P} + \mathbf{t}_c \tag{3.5}$$

Considering now the image of $\mathbf{P}$ in the depth image as having coordinates (x_d, y_d), this point can be expressed in homogeneous coordinates as $\mathbf{p}_d = (w\,x_d, w\,y_d, w)^T$, for some constant w. Using the depth camera's intrinsic parameters, $\mathbf{p}_d$ can be expressed by the equation:

$$\mathbf{p}_d = \mathbf{K}_d\mathbf{q}_d \tag{3.6}$$

This automatically reveals another expression for $\mathbf{q}_d$:

$$\mathbf{q}_d = \mathbf{K}_d^{-1}\mathbf{p}_d \tag{3.7}$$

Combining the two expressions for $\mathbf{q}_d$ (equations (3.4) and (3.7)), and solving for $\mathbf{P}$ gives an equation for point $\mathbf{P}$:

$$\begin{aligned} \mathbf{K}_d^{-1}\mathbf{p}_d &= \mathbf{R}_d\mathbf{P} + \mathbf{t}_d \\ \mathbf{R}_d\mathbf{P} &= \mathbf{K}_d^{-1}\mathbf{p}_d - \mathbf{t}_d \\ \mathbf{P} &= \mathbf{R}_d^{-1}\mathbf{K}_d^{-1}\mathbf{p}_d - \mathbf{R}_d^{-1}\mathbf{t}_d \end{aligned} \tag{3.8}$$

This expression for $\mathbf{P}$ can be substituted in equation (3.5) to get a new expression

for $\mathbf{q}_c$:

$$\begin{aligned}\mathbf{q}_c &= \mathbf{R}_c(\mathbf{R}_d^{-1}\mathbf{K}_d^{-1}\mathbf{p}_d - \mathbf{R}_d^{-1}\mathbf{t}_d) + \mathbf{t}_c \\ &= \mathbf{R}_c\mathbf{R}_d^{-1}\mathbf{K}_d^{-1}\mathbf{p}_d - \mathbf{R}_c\mathbf{R}_d^{-1}\mathbf{t}_d + \mathbf{t}_c \end{aligned} \tag{3.9}$$

Using equation (3.7), equation (3.9) simplifies to:

$$\mathbf{q}_c = (\mathbf{R}_c\mathbf{R}_d^{-1})\,\mathbf{q}_d + (\mathbf{t}_c - \mathbf{R}_c\mathbf{R}_d^{-1}\mathbf{t}_d) \tag{3.10}$$

Equation (3.10) reveals how the world points in the depth camera's coordinate system are related to the world points in the color camera's coordinate system. As seen from the equation, this transformation is given in terms of the cameras' extrinsic parameters. Therefore, the relative transformation between the depth and color cameras is defined by the rotation matrix in equation (3.11) and the translation vector in equation (3.12).

$$\mathbf{R}_r = \mathbf{R}_c\mathbf{R}_d^{-1} \tag{3.11}$$

$$\mathbf{t}_r = \mathbf{t}_c - \mathbf{R}_r\mathbf{t}_d \tag{3.12}$$

3.2.3 Point Correspondences

The fusion algorithm must convert every point (x_d, y_d) in the depth image into a point (x_c, y_c) in the color image. This is achieved by first computing the world point $\mathbf{q}_d$ from the depth image point (x_d, y_d). Then, $\mathbf{q}_d$ is transformed into $\mathbf{q}_c$ using the results from Section 3.2.2. Finally, world point $\mathbf{q}_c$ is converted into a color image point (x_c, y_c).

If (f_x, f_y) and (x_0, y_0) are the focal length and principal point of the depth camera, respectively, the world point $\mathbf{q}_d = (X, Y, Z)^T$ can be related to the image point (x_d, y_d)

through the perspective projection equations (3.13) and (3.14).

$$x_d = f_x \frac{X}{Z} + x_0 \tag{3.13}$$

$$y_d = f_y \frac{Y}{Z} + y_0 \tag{3.14}$$

The depth camera provides the z-component of $\mathbf{q}_d$.[3] The x and y components are computed by solving equations (3.13) and (3.14) for X and Y, respectively:

$$X = \frac{Z}{f_x}(x_d - x_0) \tag{3.15}$$

$$Y = \frac{Z}{f_y}(y_d - y_0) \tag{3.16}$$

Using the color camera's intrinsic parameters, $\mathbf{p}_c$ can be expressed by the equation

$$\mathbf{p}_c = \mathbf{K}_c \mathbf{q}_c \tag{3.17}$$

Furthermore, equation (3.10) gives an expression for $\mathbf{q}_c$. Therefore, combining (3.17) and (3.10) results in a new equation for $\mathbf{p}_c$:

$$\mathbf{p}_c = \mathbf{K}_c(\mathbf{R}_r \mathbf{q}_d + \mathbf{t}_r) \tag{3.18}$$

By expressing $\mathbf{q}_d$ in homogeneous coordinates (that is, $\mathbf{q}_d^{'} = (X, Y, Z, 1)^T$), equation (3.18) can be rewritten as

$$\mathbf{p}_c = \mathbf{K}_c[\mathbf{R}_r \ \mathbf{t}_r]\mathbf{q}_d^{'} \tag{3.19}$$

The image coordinates (x_c, y_c) are obtained by dividing the first and second components of $\mathbf{p}_c$ by its third component. That is, if $\mathbf{p}_c = (x, y, z)^T$, then

$$(x_c, y_c) = \left(\frac{x}{z}, \frac{y}{z}\right) \tag{3.20}$$

[3]The depth camera can actually provide all components as discussed in Section 3.1. However, the noise in these measurements is high and recomputing the x and y components delivers better results.

The file DepthColorCam.ps hasn't been created from DepthColorCam.dot yet.
Run 'dot -Tps -o DepthColorCam.ps DepthColorCam.dot' to create it.
Or invoke LaTeX with the -shell-escape option to have this done automatically.

Figure 3-4: DepthColorCam module dependency diagram. Arcs with white arrows represent subtype relations (A ▷ B = A extends B) while arcs with black arrows represent implementation relations (A ▶ B = A uses B). Gray rectangles represent abstract classes. The ImageBuffer class is omitted from this diagram.

3.3 Implementation in the Vision System

The vision system described in Chapter 2 provides the core structure needed to implement the depth-color fusion algorithm. The implementation is introduced in the form of a new camera object that provides fused depth-color images. The following sections discuss the classes that make up this representation, and the module dependency diagram in Figure 3-4 illustrates how they are related to the components of the vision system.

3.3.1 Depth-Color Camera

The DepthColorCam class represents a camera that provides fused depth-color images. It extends the Camera abstract class, the base camera representation discussed in Section 2.2. The implementation of the DepthColorCam class, however, differs from that of the ColorCam and SwissRangerCam classes (see Sections 2.4 and 2.5). The principal reason for the difference comes from the fact that instead of representing an actual sensor device, this class represents the merging of a depth camera with a color camera through the algorithmic process discussed in Section 3.2.

In order to respond to this representation, the DepthColorCam class instantiates a SwissRangerCam object and a ColorCam object. As discussed before, this means the acquired depth and color images can either come directly from the cameras or be transferred through the network. Before obtaining the fused images, the depth-color camera needs to be calibrated using the DepthColorCalibrationTool. This calibration tool provides the cameras' parameters needed by the DepthColorFusion class to

fuse the acquired depth and color images.

Table 3.1 lists the methods of the DepthColorCam class. The depth-color camera object provides access to the fused image through the getFusedColor and getDepth methods. The first method returns the image buffer containing the color information of the fused image. The second method returns the image buffer containing the depth information. The camera object also provides access to the original color image (getOriginalColor) and to the amplitude image associated with the depth measurements (getAmplitude).

Table 3.1: Public methods in the DepthColorCam class

DepthColorCam extends Camera
getFusedColor
getDepth
getAmplitude
getOriginalColor
getDepthDisplay
getAmplitudeDisplay
getFusedColorView
getDepthView
getAmplitudeView
getOriginalColorView
getFusionTool
getCalibrationTool
getModulationFrequency
setCameraParameters
setupDepthColorFusion

Like the SwissRangerCam class, the depth-color camera provides image buffers with the color visualizations of the depth and amplitude images. These visualizations are obtained through the getDepthDisplay and getAmplitudeDisplay methods. Furthermore, all image streams (fused color, depth, amplitude, and original color) are displayed in r1d1 using the ImageView objects that are associated with each image buffer.

The rest of the methods are used to retrieve the modulation frequency on which

the SwissRanger camera is operating, to gain access to the `DepthColorCalibrationTool` and `DepthColorFusion` objects associated with the depth-color camera, and to setup the depth-color fusion object after the camera has been calibrated.

When capturing the image stream from the depth and color cameras, it is important to make sure that the pair of acquired images belongs to the same point in time. Since the camera devices are different, the time response of their capturing mechanisms is not identical. Furthermore, the transfer of the images from the cameras to the computer running `r1d1` also accounts for the time difference between the images.[4]

To solve this synchronization problem, the depth-color camera uses the version of the `grabImage` method in the `ColorCam` and `SwissRangerCam` objects that takes as input the desired timestamp of the image that will be grabbed (see Sections 2.4 and 2.5). By using this method it is possible to retrieve the two images closest in time while taking into account the timestamp offset. Table 3.2 describes the algorithm used by the `grabImage` method in the depth-color camera to retrieve the depth and color images and produce the fused image.

Table 3.2: Algorithm for the `grabImage` method in `DepthColorCam`. The method returns `true` if the image was grabbed successfully and `false` otherwise.

```
GRAB_IMAGE :
1   Grab image from color camera;
2   If the image is grabbed successfully
3      Then:
4         Grab image from SwissRanger camera using as
-            input the timestamp of the color image;
5         If the image is grabbed successfully
6            Then:
7               Fuse the color and depth images;
8               Return true;
9            Else:
10              Return false;
11     Else:
12        Return false;
```

Another problem that arises with the depth-color camera involves the noisy nature

[4]For example, the computer's processor speed might affect the image transfer time.

Figure 3-5: Fused color image without noise reduction

of the SwissRanger images. As mentioned in Section 3.2.3, the noisy values for the x and y coordinates provided by the SwissRanger camera are discarded and recalculated using equations (3.15) and (3.16). The depth values, however, cannot be recalculated. The noise from the depth data produces a fused color image like the one shown in Figure 3-5. In order to reduce this noise, some depth values are discarded based on depth and amplitude thresholds. Points that are too close, points that are too far, and points with small amplitude value are discarded. This produces areas with no depth and color information in the final fused depth-color image.

Figure 3-6 illustrates the result provided by the depth-color camera. It shows the original depth and color images, and next to them the fused color image. The black pixels in the depth and fused color images indicate the points that were discarded due to data noise. It is worth noting that by thresholding the depth image it is possible to segment the object of interest from the background.

3.3.2 Depth-Color Calibration Tool

The `DepthColorCalibrationTool` class implements the steps of the depth-color fusion algorithm that are described in Sections 3.2.1 and 3.2.2. Analogous to the `DepthColorCam` class, the implementation uses two `CalibrationTool` objects, one for each of the cameras that is to be calibrated. The idea behind it consists of performing the calibration of both cameras in parallel, accepting an image pair only when the calibration pattern is found in both images.

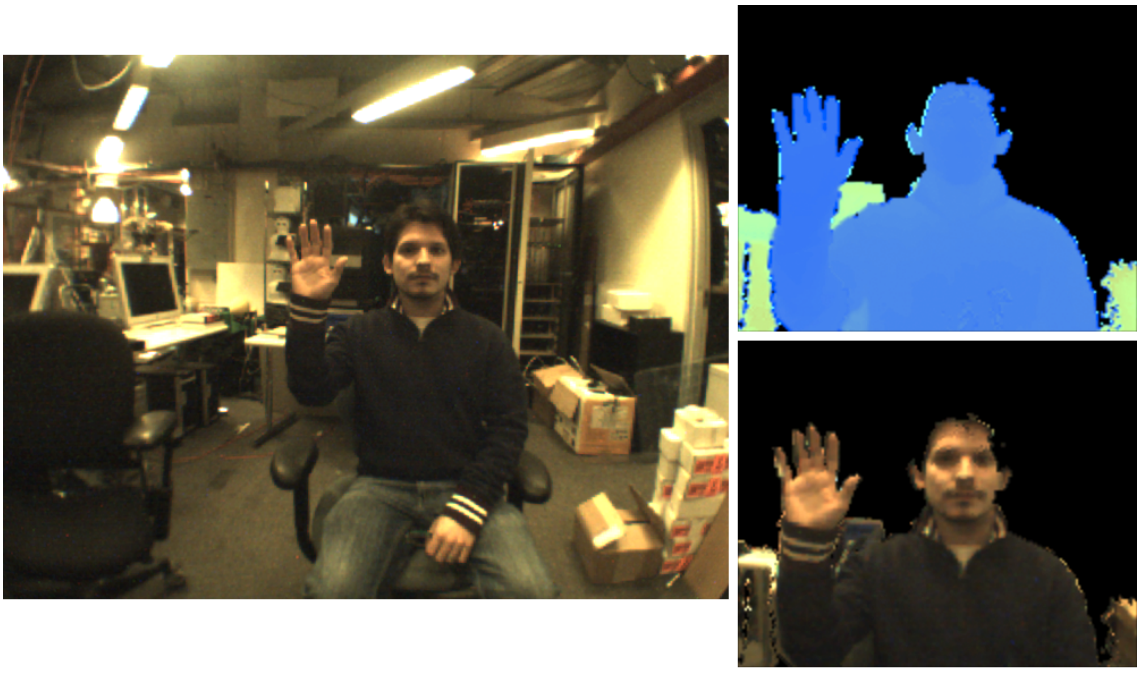

Figure 3-6: `DepthColorCam`'s images: the original color image (left), the depth image (upper right) and the fused color image (bottom right).

Table 3.3 lists the methods of the `DepthColorCalibrationTool` class. Similar to a `CalibrationTool` object, a depth-color calibration tool has methods to start, reset, and retrieve the status of the calibration process. It also provides methods to load and save the cameras' parameters, and to change the default values of some of the parameters in the calibration process.

The algorithm for the `calibrate` method in the `DepthColorCalibrationTool` class is also similar to the one seen in Section 2.6. However, in this case it handles two calibration processes in parallel. Throughout the steps of the algorithm described in Table 3.4, the methods `findAndDrawCorners`, `addCornersToList`, and `calibrateCamera` are called simultaneously on the two calibration tool objects used by the depth-color calibration tool.

This synchronized calibration takes advantage of the underlying OpenCV calibration functions. As mentioned in Section 2.6, the calibration is performed using the functions in the OpenCV open source library. The function `cvFindChessboardCorners`, for example, locates the corners of a checkerboard (chessboard) pattern in an

Table 3.3: Public methods in the DepthColorCalibrationTool class

DepthColorCalibrationTool
start
reset
isCalibrating
calibrate
setColorCameraParameters
setDepthCameraParameters
setChessboardSquareSize
setChessboardDimensions
setRequiredSamples
getColorCameraParameters
getDepthCameraParameters
getChessboardNumberOfColumns
getChessboardNumberOfRows
getChessboardSquareSize
getRequiredSamples
getRelativeRotation
getRelativeTranslation
getRelativeTransformation
loadCameraParameters
saveCameraParameters

image [2]. Since it returns the corners ordered in row-major order starting from the upper-left corner, it is easy to know the correspondence between the calibration points in the depth image and the calibration points in the color image.

As seen in Table 3.4, the success of the calibration loop depends on finding the calibration pattern in both images of the image pair. When the calibration is completed for both cameras, the relative transformation is calculated using equations (3.11) and (3.12). The parameters of the transformation can be accessed through three different methods. The `getRelativeRotation` and `getRelativeTranslation` methods return the 3×3 rotation matrix and 3×1 translation vector, respectively. The `getRelativeTransformation` method returns a 3×4 matrix in the form of an extrinsic matrix as described by equation (3.3). Both the combination of the first two methods and the third method by itself provide all the necessary information to describe the transformation in 3D space between the depth camera and the color camera.

Table 3.4: Algorithm for the `calibrate` method in `DepthColorCalibrationTool`

```
CALIBRATE (currentImages, requiredImages):
1   If (currentImages < requiredImages)
2      Then:
3         Search checkerboard pattern in color image;
4         Search checkerboard pattern in amplitude image;
5         If the pattern is found in both images
6            Then:
7               Extract corners from color image and save them;
8               Extract corners from amplitude image and save them;
9               Increment currentImages counter;
10     Else:
11        Run color camera calibration with saved corners;
12        Run depth camera calibration with saved corners;
13        Compute relative transformation with computed parameters;
```

3.3.3 Depth-Color Fusion

The `DepthColorFusion` class implements the steps of the depth-color fusion algorithm described in Section 3.2.3. The constructor of this class takes as input the cameras' parameters computed with the `DepthColorCalibrationTool` object. From these parameters it can calculate the relative transformation between the depth and color cameras using equations (3.11) and (3.12). Then, method `fusePoints`, listed in Table 3.5, can be used to get the point correspondences between the depth and color images.

Table 3.5: Public methods in the `DepthColorFusion` class

DepthColorFusion
fusePoints
getDepthCameraParameters
getColorCameraParameters
getDepthToColorMap
getColorToDepthMap

Table 3.6 describes the algorithm for the `fusePoints` method. The method takes

as input the depth image, the color image, and the image buffer where it will store the fused color information. For each pixel in the depth image it finds the correspondent pixel in the color image. Then, the algorithm retrieves the color value from the pixel in the color image and stores it in the pixel of the fused color buffer with position equal to the position of the depth image pixel. This fusion operation can be performed in real time with live image streams.

Table 3.6: Algorithm for the `fusePoints` method in `DepthColorFusion`

```
FUSE_POINTS (depthImage, colorImage, fusedImage):
1  For each pixel (x_d, y_d) of depthImage:
2     Compute the world point q_d using the depth value
-        and equations (3.15) and (3.16);
3     Compute homogeneous image point p_c using equation (3.19);
4     Compute color image coordinate (x_c, y_c) using equation (3.20);
5     Retrieve color value from pixel (x_c, y_c) of colorImage;
6     Place retrieved color value in pixel (x_d, y_d) of fusedImage;
```

The final output of the depth-color fusion algorithm is a fused image that is composed of the original depth image and the fused color image. There is a one-to-one correspondence between the pixels of these images with same x and y position. Furthermore, the `DepthColorFusion` object stores the point correspondence information in the form of two hash maps, one that maps points in the depth image to points in the original color image, and another one that maps points in the original color image to points in the depth image. These hash maps can be accessed through the `getDepthToColorMap` and `getColorToDepthMap` methods, respectively.

Chapter 4

Face and Body Pose Tracking

Chapter 2 presented an integrated vision system for `r1d1`, and Chapter 3 showed that it is possible to build new camera representations on top of it. Along with capturing and handling images, a vision system should be able to retrieve information from them. This chapter integrates two algorithms in the field of object detection and tracking that let the system gather information about humans in the camera's surroundings.

Section 4.1 presents a face tracking algorithm that operates on the depth-color images introduced in Chapter 3. This algorithm uses the depth information to segment the person of interest from the background and consequently reduce the number of false positives detected by a traditional face detector. The algorithm is shown to be robust to different face positions and orientations.

Section 4.2 introduces a representation for the Microsoft's Kinect camera and integrates an open source framework that uses the Kinect's images to detect and track a person's body pose. The algorithm implemented in this framework operates on the depth images captured by the Kinect and outputs a body skeleton indicating the position and orientation of various joints in the human body.

In this chapter, the Kinect becomes the second depth-color camera that forms part of the integrated vision system. In conjunction with the two tracking algorithms, this serves to point out the new importance of these sensors in the field of computer vision.

4.1 Face Tracker

The images from a depth-color camera can be used to get better results from traditional face detection algorithms. The false positives from such algorithms come in part from pixels in the background that are thought to be faces. With the depth-color images, the depth information can be used to remove pixels from the background based on distance thresholds.

The `FaceTracker` class implements an algorithm that uses the Viola-Jones object detection framework [16] to find faces in depth-color images. Furthermore, after detecting the faces it uses the Camshift (Continuously Adaptive Mean Shift) algorithm [3] to track them based on color information. The `FaceTracker` uses the OpenCV library's implementations of the Viola-Jones and Camshift algorithms. [5]

The idea of the `FaceTracker` algorithm consists of discarding the pixels in the fused color image that are below or above certain depth thresholds. The value for these pixels are replaced with a zero value (a black pixel) creating uniform areas in the image that do not contain color information. Figure 3-6 already showed an example of a fused color image after thresholding based on depth. It revealed that this technique can successfully segment the object of interest from the background, given that there is a significant distance between the two.

Table 4.1 describes the algorithm of the `findFaces` method, the main method of the face tracker class. This method performs the task of running the Viola-Jones detector and the Camshift tracker, and integrating the results from both algorithms.

The algorithm gives priority to the Viola-Jones detector over the Camshift tracker, mainly because Camshift only relies on tracking a patch of pixels with color distribution similar to that of a reference patch. Furthermore, this design follows the assumption that a person, the object of interest for the `FaceTracker`, will be facing the camera most of the time, prompting in that way a high rate of successful face detections. Therefore, the detector is ran on every call to `findFaces` (line 1), while the tracker is used as a backup for when the detector fails to find a face (line 11).

[5] OpenCV implements the extended version of the Viola-Jones detector described by Lienhart and Maydt [8].

Table 4.1: Algorithm for the `findFaces` method in `FaceTracker`

```
FIND_FACES (faceStream, successThreshold):
1   Run Viola-Jones face detector;
2   If face is detected
3      Then:
4         Add face to faceStream;
5         If the number of consecutive found faces
-         is larger than successThreshold
6            Then:
7               Initialize Camshift with the detection
-               window of the found face
8      Else:
9         If faceStream contains a face for the last image
-         and Camshift has been initialized
10           Then:
11              Run the Camshift tracker;
12              If face is successfully tracked
13                 Then:
14                    Add face to faceStream;
```

The Camshift tracker needs to "initialized" before it can track a face (line 7). This initialization consists of building a histogram of the color information in a given reference window. The color information is the hue channel of the image and the reference window is the face detection window outputted by the Viola-Jones detector. Afterwards, every run of the tracker (line 11) calculates the back projection of the input image's hue channel using the reference histogram, and uses that to find the new location of the reference face. It is not necessary to reinitialize the tracker on every successful detection because on consecutive image frames the position of the face does not change drastically. However, after some time the information in the reference histogram might need to be refreshed, making necessary to reinitialize the tracker with a new face detection window (lines 5 to 7).

The faces found by the `FaceTracker` algorithm are added to a `FaceStream` object (lines 4 and 14). The `FaceStream` class extends `Stream` and is similar in concept to the image stream representation discussed in Section 2.7. In this case, the stream uses objects of the type `java.util.List` as buffer objects. In these lists the stream can

store multiple instances of the Face class, a simple data structure that holds the face's position and dimension. The advantage of storing the found faces in a face stream is that the user can define other algorithms based on the stored result of consecutive calls to findFaces.

Table 4.2 lists the methods of this class. The detectFaces, initializeTrackers, and trackFaces methods are called by findFaces to run the Viola-Jones detector, initialize Camshift, and run the Camshift tracker, respectively. The getFacesInFused method can be used to access the faces found on the last call to findFaces. The actual face stream is accessed through the getFacesInFusedStream method.

Table 4.2: Public methods in the FaceTracker class

FaceTracker
findFaces
detectFaces
initializeTrackers
trackFaces
mapFace
getFacesInFused
getFacesInColor
getFacesInFusedStream
getFacesInColorStream
drawFaceInColor
drawFaceInFused
findFaceInterior
mapFaceInterior

Furthermore, since the DepthColorFusion object contains the mapping from pixels in the depth image to pixels in the color image (see Section 3.3.3), it is possible to know the position of the face in the original color image. The mapFace method is used to map faces from the fused color image to the original color image. This method is called automatically by the face tracker in order to provide an additional stream of faces. The getFacesInColor and getFacesInColorStream methods are used to access the faces in the original color image.

The face tracker takes advantage of this mapping by providing the capability of de-

Figure 4-1: Output of the `FaceTracker`'s algorithm on a sequence of depth-color images. The red squares indicate the position and dimension of the faces.

tecting the eyes, nose, and mouth of a face using the Viola-Jones detector. Since the fused color image does not provide the necessary resolution to detect face parts, the face tracker uses the mapped face to run the detector on the region of interest of the original color image. The `findFaceInterior` method performs this operation, and the `mapFaceInterior` method can be used to map the positions of the eyes, nose, and mouth back to the depth-color image.

Figure 4-1 shows a sequence of fused color images that have passed through the `FaceTracker`'s algorithm. The red squares indicate the position and dimension of the face that was found on each of the images. Once the face tracker detects a person facing the camera, it can start tracking the face through different head movements. The sequence of images shows that the algorithm is robust to change in face position (including distance from the camera) and orientation.

4.2 Body Pose Tracker

The vision system also provides a way of tracking the body pose of a person. This is achieved through the OpenNI open source framework, which provides support for detecting and tracking the body of a human in depth images [11]. OpenNI achieves this by attempting to fit a skeleton on a potential human body. The information about the skeleton's joints is used to extract the position and orientation of the body.

The OpenNI module that contains the skeleton fitting capabilities is designed to work exclusively with PrimeSense derived sensors [15, 14]. An example of such sensor is the Microsoft's Kinect camera. Similar to the depth-color camera setup described in Chapter 3, the Kinect is designed to capture depth and color images, everything on a single piece of hardware. The Kinect's images can be retrieved using the PrimeSense interface provided by OpenNI.

In order to take advantage of OpenNI's body tracking support, the vision system introduces the `KinectCam` class. This class serves both as a representation of the Kinect camera and an interface to the OpenNI framework. The `KinectCam` class is implemented following the same structure as the `ColorCam` and `SwissRangerCam` classes (see Sections 2.4 and 2.5). The `KinectJavaAcquire` class establishes the Java interface to OpenNI and as such is used to retrieve the images and body skeleton information. The `KinectPacketHandler` class receives image and skeleton data that is transferred over the network. The `KinectReceiver` uses instances of the two previous classes to handle the data streams from the Kinect. A Kinect receiver is used by the `KinectCam` class to gain access to the image and skeleton data.

This design reveals that, in the `r1d1` environment, the Kinect camera is seen not only as a sensor that provides depth and color images, but also as a sensor that provides body skeleton data for every image frame. The skeleton data is represented with an object of type `KinectSkeleton`. This object contains the position and orientation of every joint in the body skeleton generated by OpenNI. The Kinect receiver handles the skeleton objects using an instance of the `KinectSkeletonStream` class, a subtype of `Stream`. The `KinectSkeletonStream` class represents a sequence of detected

body poses, and its design is similar to the design of the `FaceStream` class discussed in Section 4.1.

Table 4.3 lists the methods that the `KinectCam` class adds to the base camera representation. The `getSkeletons` method further exemplifies the idea that in this system the Kinect captures image and body skeleton data. The `createDisplay` method is used to create a color visualization of the raw depth data, and the `threshold` method serves to threshold the values of the depth image.

Table 4.3: Public methods in the `KinectCam` class

KinectCam extends Camera
getColor getDepth getSkeletons getDepthDisplay getColorView getDepthView createDisplay threshold

Figure 4-2 shows a sequence of depth images captured with the Kinect camera along with superimposed red skeletons that illustrate the output of OpenNI's body tracking algorithm. As it has been seen before, thresholding the image based on the depth values can segment the body of a person from the background assuming there is a significant distance between the two. The top left image shows the position of the body that is required to calibrate and start the tracking algorithm. The other images show that the algorithm is robust to different body positions and orientations.

One of the principal advantages of the body skeleton data is that it can be used as input to gesture recognition algorithms. For example, the skeleton can provide information about the position of a person's arms with respect to the rest of the body. This information can be used to train algorithms that classify certain arm movements into gestures, making possible to recognize, for example, an arm waving, pointing, or reaching out.

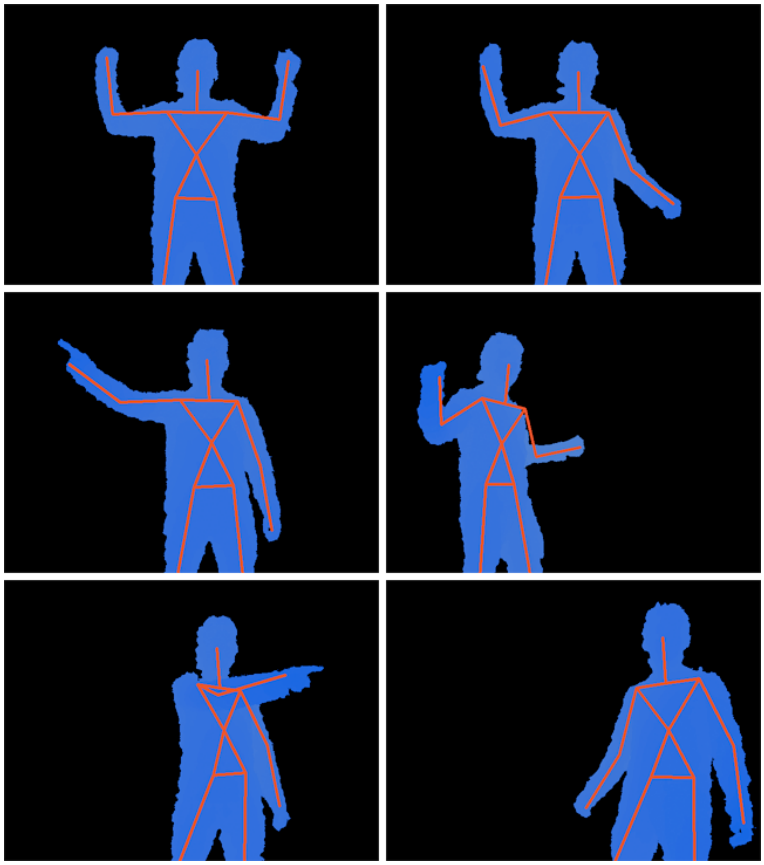

Figure 4-2: Output of OpenNI's body pose tracking algorithm on a sequence of depth images. The red lines indicate the position of the skeleton's joints.

Chapter 5

Conclusion

This thesis presented an integrated vision system for a robotics research and development platform. The system was described as part of `r1d1`, the software platform developed and used by the Personal Robots Group at the MIT Media Lab. Nevertheless, its design can be generalized and implemented in other platforms.

The proposed vision system featured representations for images and camera sensors. These representations were shown to be versatile. For example, an `ImageBuffer` can describe images of any size and with a variety of pixel depths and color models. Moreover, the `Camera` and `CameraReceiver` classes can be extended into subclasses that represent any type of camera. This thesis presented the implementations of three different types of cameras: a color camera, a 3D time-of-flight camera (or depth camera), and a Microsoft's Kinect camera.

Then, it was shown that it is possible to build new representations on top of the existing ones. The example discussed in this thesis was the `DepthColorCam` class. The camera represented by this class merged the representations of the depth and color cameras in order to provide images with fused depth-color information. To achieve this, it was necessary to describe an algorithm capable of performing the fusion of the depth and color images in real-time.

At the end, this thesis discussed two applications that use depth-color images to retrieve information from a scene. The first one used the images from the `DepthColorCam` object to detect and track human faces. The second application used the

images from the Kinect, another depth-color camera, to detect and track human body pose. The algorithms that performed these operations proved to be robust to different positions and orientations of the face and body.

The system described in this thesis sets the basic requirements for the vision system of any robotics development platform. Color cameras have been the "eyes" of robots for many years. Now, depth-color cameras are becoming prime sensors in robotics. Robots like the MDS now have access to enhanced image data that can be used to design new computer vision algorithms with superior performance at retrieving information about the robot's surroundings.

Many joint efforts are involved in promoting the use of depth-color images in different applications. OpenNI is led by industry leaders like PrimeSense and Willow Garage. The RGB-D Project [7], a joint research effort between Intel Labs Seattle and the University of Washington, is another example of the potential these sensors have in future computer vision research.

Appendix A

Image Buffer Performance

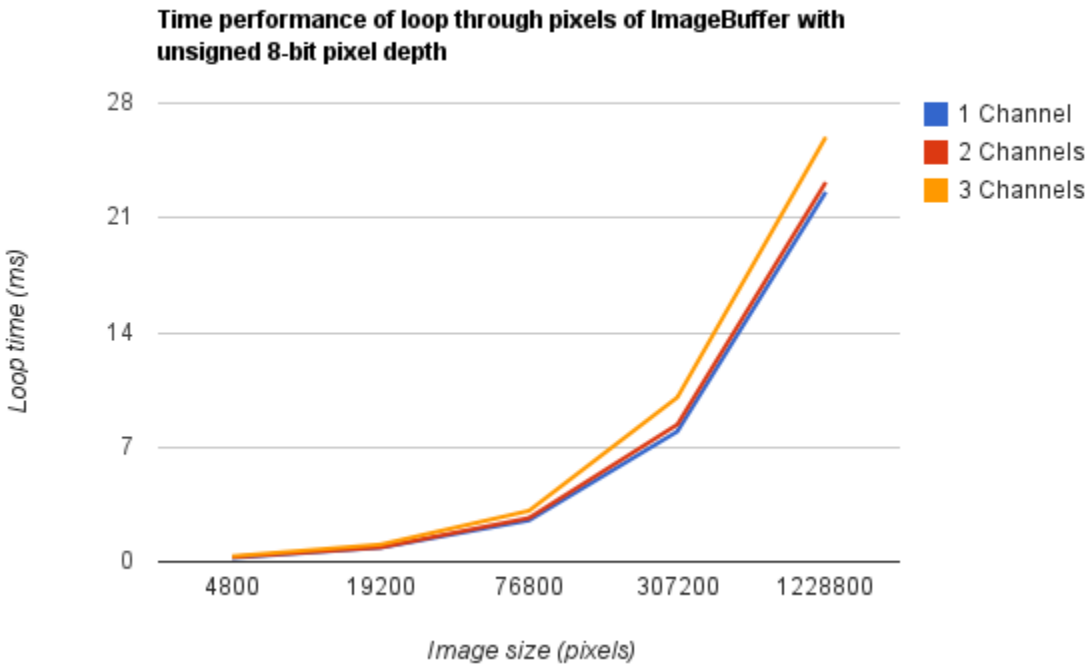

Figure A-1: Time performance of loop through all the pixels of an image buffer. The tests were ran on image buffers of pixel depth BYTE, with one, two, and three channels. The tested image sizes were 80 x 60, 160 x 120, 320 x 240, 640 x 480, and 1280 x 960.

Appendix B

Depth-Color Fusion Performance

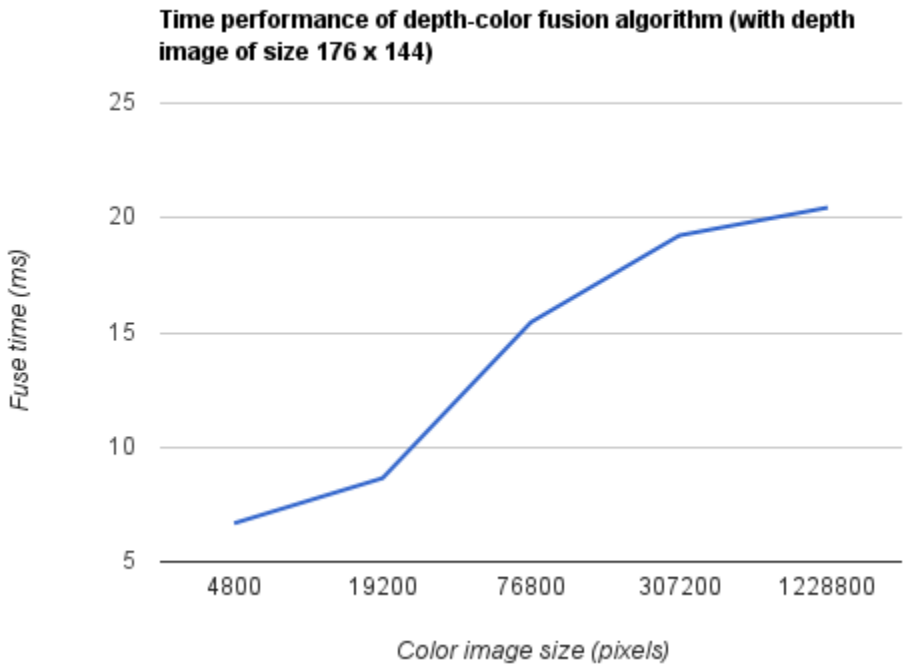

Figure B-1: Time performance of the depth-color fusion algorithm. The tests were ran changing the size (resolution) of the color image while keeping the size of the depth image constant. The tested image sizes were 80 x 60, 160 x 120, 320 x 240, 640 x 480, and 1280 x 960.

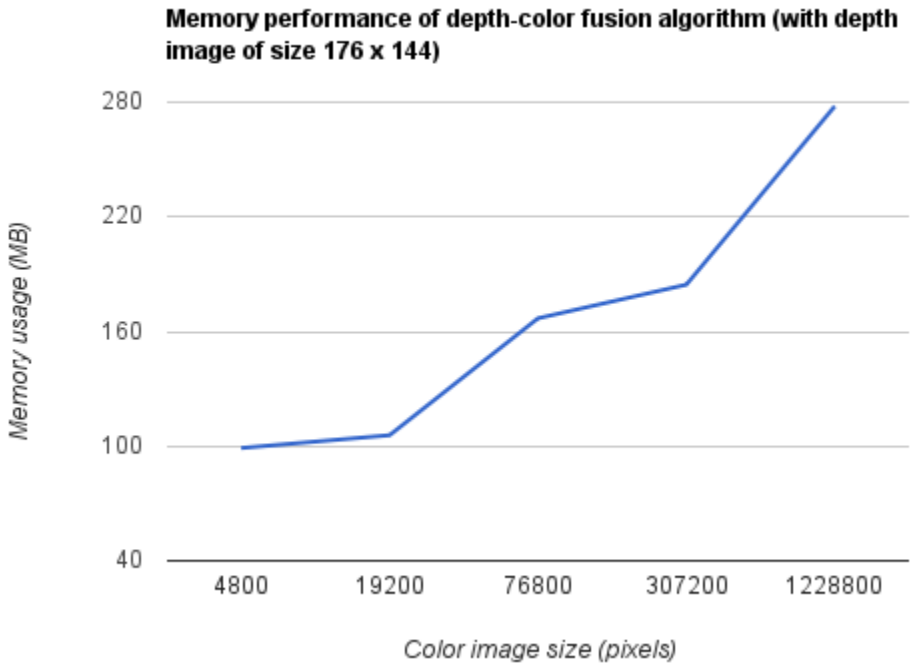

Figure B-2: Memory performance of the depth-color fusion algorithm. The tests were ran changing the size (resolution) of the color image while keeping the size of the depth image constant. The tested image sizes were 80 x 60, 160 x 120, 320 x 240, 640 x 480, and 1280 x 960.

Bibliography

[1] J.-Y. Bouguet. Camera calibration toolbox for matlab. [Online]. Available: http://www.vision.caltech.edu/bouguetj/calib_doc/index.html

[2] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*. Sebastopol, CA: O'Reilly, 2008.

[3] G. Bradski, "Computer vision face tracking for use in a perceptual user interface," *Intel Technology Journal, Q2*, 1998.

[4] D. Douxchamps. (2006, Oct.) libdc1394 homepage. [Online]. Available: http://damien.douxchamps.net/ieee1394/libdc1394/

[5] M. Hancher, "A motor control framework for many-axis interactive robots," Master of Engineering thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2003.

[6] B. K. P. Horn, *Robot Vision*. Cambridge, MA: MIT Press, 1986.

[7] RGB-D: Techniques and usages for kinect style depth cameras. Intel Labs Seattle. Seattle, WA. [Online]. Available: http://ils.intel-research.net/projects/rgbd

[8] R. Lienhart and J. Maydt, "An extended set of Haar-like features for rapid object detection," in *IEEE ICIP*, 2002, pp. 900–903.

[9] A. Linarth, J. Penne, B. Liu, O. Jesorsky, and R. Kompe, "Fast fusion of range and video sensor data," in *Advanced Microsystems for Automotive Applications 2007*, J. Valldorf and W. Gessner, Eds. Berlin: Springer, 2007, pp. 119–134.

[10] *SR4000 User Manual*, SR4000_Manual.pdf, Mesa Imaging, Zurich. [Online]. Available: http://www.mesa-imaging.ch/prodview4k.php

[11] *OpenNI User Guide*, OpenNI_UserGuide.pdf, OpenNI. [Online]. Available: http://www.openni.org/documentation

[12] MDS. Personal Robots Group, MIT Media Lab. Cambridge, MA. [Online]. Available: http://robotic.media.mit.edu/projects/robots/mds/overview/overview.html

[13] *Firefly MV Product Datasheet*, fireflymv.pdf, Point Grey. [Online]. Available: http://www.ptgrey.com/products/fireflymv/fireflymv.pdf

[14] *PrimeSensor NITE 1.1 Middleware Datasheet*, FMF_3.pdf, PrimeSense. [Online]. Available: http://www.primesense.com/?p=515

[15] *PrimeSensor Reference Design 1.08 Datasheet*, FMF_2.pdf, PrimeSense. [Online]. Available: http://www.primesense.com/?p=514

[16] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *IEEE CVPR*, 2001, pp. 511–518.

[17] Z. Zhang, "Flexible camera calibration by viewing a plane from unknown orientations," in *ICCV*, 1999, pp. 666–673.