UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Barcelona School of Computer Science
Polytechnic University of Catalonia (BarcelonaTech)

Master Thesis

Motion Capture Based on RGBD Data from Multiple Sensors for Avatar Animation

Miguel A. Vico Moya

Supervisor: Nuria Pelechano
Advisor: Pere Brunet

Department of Computer Science

I would like to thank my supervisor Nuria Pelechano and advisor Pere Brunet for their guidance and patience throughout the whole development and writing process of this thesis.
Also to my wife, whom has been very supportive and kept me up and going until the end of this project.

Contents

Abstract

With recent advances in technology and emergence of affordable RGB-D sensors for a wider range of users, markerless motion capture has become an active field of research both in computer vision and computer graphics.

In this thesis, we designed a POC (Proof of Concept) for a new tool that enables us to perform motion capture by using a variable number of commodity RGB-D sensors of different brands and technical specifications on constraint-less layout environments. The main goal of this work is to provide a tool with motion capture capabilities by using a handful of RGB-D sensors, without imposing strong requirements in terms of lighting, background or extension of the motion capture area. Of course, the number of RGB-D sensors needed is inversely proportional to their resolution, and directly proportional to the size of the area to track to.

Built on top of the OpenNI 2 library, we made this POC compatible with most of the non-high-end RGB-D sensors currently available in the market. Due to the lack of resources on a single computer, in order to support more than a couple of sensors working simultaneously, we need a setup composed of multiple computers. In order to keep data coherency and synchronization across sensors and computers, our tool makes use of a semi-automatic calibration method and a message-oriented network protocol.

From color and depth data given by a sensor, we can also obtain a 3D pointcloud representation of the environment. By combining pointclouds from multiple sensors, we can collect a complete and animated 3D pointcloud that can be visualized from any viewpoint. Given a 3D avatar model and its corresponding attached skeleton, we can use an iterative optimization method (e.g. Simplex) to find a fit between each pointcloud frame and a skeleton configuration, resulting in 3D avatar animation when using such skeleton configurations as key frames.

Part I

Introduction

Chapter 1

Introduction

1.1 Motivation and objectives

Despite being a big step forward over marker-based motion capture systems, markerless setups are rarely used for professional productions such as movies or video-games. On markerless systems, actors do not require wearing special suits or rigs that sometimes difficult actors performance. However, these kinds of setups usually require the use of many cameras and a studio with controlled lighting and easy removable background.

Markerless motion capture has been studied for many years both in computer vision and computer graphics, but it is still an active field of research due to the existing room for improvement. Additionally, with the emergence of more affordable RGB-D sensors in the market, many other solutions to address the same problem have arisen, as both the open source community and not-so-big research labs have been able to acquire these type of devices.

Throughout this work, we present a new POC tool that allows to perform offline motion capture by using a variable number of commodity RGB-D sensors of different brands and technical specifications on constraint-less layout environments. Built on top of the OpenNI2 library [1], we leverage its multi-sensor support in order to manage both Kinect v1 and v2 sensors using the same API. Additionally, other sensors could be used as long as they are supported by OpenNI2.

Our tool provides a framework that lets the user define a multi-Kinect capture environment. A calibration procedure and algorithm has been designed to find the extrinsic parameters of all sensors so a global reference frame can be used to process their data. By using a distributed-system model, our tool will capture color, depth, and infrared streams from several sensors connected to different computers. Later, it lets the user post-process the captured data in order to synchronize all recorded clips.

Then, an algorithm removes static objects in the scene, and computes a set of raw point-

clouds of the actor's performance per sensor and per frame.

Finally, the processed data is merged to compute a final 3D-pointcloud clip of the captured data.

The main goal of this work is to have a reliable 3D representation of the character that could be then tracked in real-time to animate a virtual avatar.

As part of this project we have written an OpenNI2 driver on top of Microsoft 'Kinect for Windows' [2] driver and SDK in order to add Kinect v2 support to the OpenNI2 framework.

Additionally, an avatar creator plugin for MakeHuman [3] was implemented so a 3D-pointcloud from our tool can be used as a template while creating an avatar in real-time. Such avatar would be used in the motion capture data extraction procedure.

1.2 Document walk-through

In chapter 2, we briefly introduce OpenNI, its different versions, and discuss some of its technical details and current status. Also, we make a general introduction of how support for Microsoft proprietary Kinect hardware is added to the OpenNI2 framework, and discuss some of the technical details and differences of both Microsoft Kinect v1 and v2 sensors [2] used in our experiments.

Chapter 3 gives a brief description of the state-of-the-art of both motion capture and 3D reconstruction fields of study.

Part II is divided into three separate chapters:

- Chapter 5: OpenNI2 Microsoft Kinect v2 driver
- Chapter 4: Capture System Prototype
- Chapter 6: Avatar Template Creator plugin for MakeHuman

In chapter 5 we give implementation details of the OpenNI2 Kinect v2 driver such as what 'Kinect for Windows' APIs were used or how several non-native features were supported.

A deeper description of our capture tool is found in chapter 4. Each of its modules (calibration, network layer...) and post-processing algorithms (clips synchronization, background removal...) are presented.

In chapter 6 a brief introduction to the MakeHuman avatar template creator plugin is given.

Finally, in part III we present some of the captured clips and resulting 3D-pointcloud animations. Also, we propose future lines of development along the lines of this project,

as well as a description of some of the missing parts we were not able to finish.

Chapter 2

Background

2.1 OpenNI

OpenNI (Open Natural Interaction) is an open source software widely used across the open source community focused on providing interoperability of natural user interfaces and organic user interfaces for natural interaction devices [4]. Thus, among other things, it facilitates access to RGB-D sensors of many different brands and technical specifications such as those made available by Primesense or Microsoft.

From a low-level perspective, it implements different drivers to manage each of the supported sensors. Those drivers are handled as plugins by the OpenNI library, enabling manufacturers to easily add support for new sensors through OpenNI.

2.1.1 PrimeSense

One of the main promoters and maintainers of the OpenNI project was PrimeSense. PrimeSense was an Israeli 3D sensing company founded on 2005 and based in Tel Aviv [5]. It is the company behind the depth sensing IPs Microsoft Kinect is based on.

Some of the PrimeSense proprietary RGB-D sensors available nowadays are:

- Carmine 1.08 (Kinect-like)
- Carmine 1.09 (Short range)
- Capri 1.25 (embedded)

On November 24th 2013, PrimeSense was bought by Apple. Apparently, Apple had some plans to improve his line of living room products such as Apple TV by adding support for natural interaction interfaces. Purchase of PrimeSense was a strategic move forward

in order to compete with Microsoft, which first version of their Kinect device generated huge benefits for the company (see section 2.2.1).

On April 23rd 2014, Apple shutdown the OpenNI project, discontinuing OpenNI support. Fortunately, former partners of PrimeSense and contributors of the project kept documentation and continued giving support on their corresponding branches of the OpenNI library (see section 2.1.2).

2.1.2 OpenNI 2

Although the main fork of the OpenNI 2 SDK was shutdown when Apple decided to discontinue the OpenNI project, other forks were kept alive along with OpenNI documentation. One of the most actively supported OpenNI 2 forks is the one owned by Occipital [6], one of the former partners of PrimeSense which remains active in the depth sensing business.

OpenNI 2 SDK is a good start point to base the development of this thesis off. Among other things, it provides the following desirable features:

- Drivers for several RGB-D sensors in the market
- Complete and standardized API to:
  - Access color data
  - Access depth data
  - Access IR (infrared) data
  - Record any of the data streams above in ONI format (OpenNI-specific)
  - Playback previously recorded data by creating a virtual device, with can be handled as an actual physical device

Additionally, OpenNI 2 SDK design and documentation allows developers to easily extend its functionality by either adding support for new sensors (add drivers) or increasing the number of features to better suit specific use cases.

As part of this thesis, we have contributed to the OpenNI 2 project by writing a complete OpenNI 2 driver on top of the Microsoft Kinect for Windows [2] driver in order to add support for Kinect v2 devices. See chapter 5 for more details.

2.2 Kinect sensor support

As previously stated, throughout the development of this project we have used both Kinect v1 and v2 sensors. Unlike first version of OpenNI which gave support for Kinect v1 devices

by using reversed engineered libfreenect drivers [7], OpenNI 2 supports it through an OpenNI driver written on top of the official Microsoft Kinect v1 driver.

However, by the time this project was started, OpenNI 2 did not provide support for Kinect v2 devices. As part of this thesis, we wrote an OpenNI 2 driver on top of the Microsoft Kinect for Windows driver in order to add support for Kinect v2 sensors. In chapter 5 we thoroughly describe how such driver was implemented and how we made a contribution to the OpenNI 2 open source project.

Following subsections briefly describe some of the technical aspects of both Microsoft Kinect v1 and v2 sensors.

2.2.1 Kinect version 1

Microsoft Kinect v1 sensor was first announced as a game controller for Xbox 360 console on June 1st 2009 at E3 (Electronic Entertainment Expo) in Los Angeles, USA. Over the following years, it gained interest not as a game controller device, but as a depth sensing device by itself, allowing people all around the world to use it as mocap device or a 3D scanner, among other applications. By February 2013, 24 million units were sold by Microsoft [8].

Some features that make it attractive and suitable to satisfy this project requirements are:

- Generates color and depth data frames at 30 fps
- Can be used at distances up to 4.5 meters from the subject
- Price: $100

In addition to the above, multiple Kinect v1 devices can be connected to the same computer as long as each device is plugged into a different USB 2.0 controller. See comparison table 2.1 for some more specification details.

2.2.2 Kinect version 2

Microsoft Kinect v2 sensor was first released on November 22nd 2013. Overall, it offers an improvement over its predecessor in terms of technical specifications. Both color and depth sensors generate higher resolution data, and from empirical testing, depth precision is higher on the Kinect v2 device. The improvement on color and depth quality comes at the expense of requiring GPU hardware acceleration, which makes plugging multiple Kinect v2 devices to the same computer impossible.

Table 2.1 details the technical specifications differences between both versions of the Kinect sensor.

Feature	**Kinect v1**	**Kinect v2**
Color	640x480 @30fps	1920x1080 @30fps
Depth	320x240	512x424
Max depth distance	4.5m	4.5m
Min depth distance	40cm (near mode)	50cm
Horizontal field of view	57 degrees	70 degrees
Vertical field of view	43 degrees	60 degrees
Tilt motor	Yes	No
USB standard	2.0	3.0
Requires GPU hardware acceleration	No	Yes
Multi-device	Yes. 1 per USB controller	No
OS	Win 7, Win 8	Win 8
Price	$100	$200

Table 2.1: Comparison between Kinect v1 and v2 sensors

Chapter 3

Previous Work

3.1 Introduction

Thanks to the emergence of RGB-D cameras, such as Microsoft Kinect or Primesense devices, and their affordable cost, many researchers have found their way through on different topics related to computer graphics and computer vision which make use of these sensors.

Although different models of cameras/devices may present different features, most of them provide real-time (30+ fps) color and depth data. It enables researchers to use such devices for motion capture, 3D reconstruction, or gesture recognition.

While a huge amount of works have been focused on skeleton pose estimation either by using color-only sensors or depth sensors [9], [10], [11], [12], [13], [14], [15], many applications require the extraction of 3D surface as well. 3D estimation methods have been also proposed [16], [17], [18], [19]. Tracking the full geometry over time is a more complex task and most of the existing methods rely on a first individual 3D reconstruction in order to be able to accurately estimate both pose and geometry changes.

This document briefly walks through some of the relatively recent works done in the fields of motion capture and 3D reconstruction, both from a generic and avatar/human body reconstructions points of view.

3.2 Motion capture

3.2.1 Model initialization

Many of the markerless motion capture techniques rely on having a first skeleton estimation or a 3D approximation of the individuals to be tracked. These skeletons or 3D models are

commonly generated at a pre-processing stage and taken by the algorithm as an input. Most of the algorithms for pose estimation continue to use a manually initialized generic model. For instance, different approaches addressed this problem by estimating the body pose from manually selected joint locations [20], [21], [22], [23].

Others have explored the extraction of the skeleton structure from 3D surfaces reconstructed from multiple views. *Cheung et al.* [24] initialize the skeleton from the visual-hull of a person moving each joint independently. A full-skeleton together with the shape of each body part is obtained by alignment of the segmented moving body parts with the visual-hull model in a fixed pose. *Menier et al.* [25] present an automated approach to 3D human pose estimation from the medial axis of the visual-hull.

With the appearance of modern and fast RGB-D sensors such as Kinect, many researchers started to use the depth information to obtain better 3D shape approximations and skeleton representations. *Tong et al.* present a capturing system consisting in three Kinect sensors in order to increase the overall quality of the reconstruction [26]. They use two Kinects to capture the upper and lower parts of the body (far enough of each other in order to avoid overlapping) while using the third one to capture the middle part from the opposite direction. Using a single sensor to address the same problem have also been explored [27], [28]. They basically ask the user to adopt a fixed pose from different points of view. By leveraging the well-known *KinectFusion* algorithm (see subsection 3.3.1) they obtain super-resolution scans in a very fast and efficient way. Then, they make use of different registration methods to join the super-resolution scans.

3.2.2 Pose estimation

Pose estimation refers to the process of estimating the configuration of the underlying skeletal articulation structure of a person. Pose estimation algorithms may be separated into three main categories:

1. *Model free*: These are methods where there is no explicit a previous model. A recent trend to overcome limitations of tracking over long sequences has been the investigation of direct pose detection on individual image frames. Two different approaches fall into this category: Probabilistic assemblies of parts and Example-based methods.

   Forsythe and Fleck [29] introduced the concept of "body plans" to represent people or animals as a structured assembly of parts learned from images. Following this direction [30], [31], [32] used pictorial structures to estimate 2D body part configurations from image sequences. Combinations of body part detectors have been also used to address the related problem of locating multiple people in cluttered scenes with partial occlusion [33], [34].

2. *Indirect model*: These methods use a prior model as a reference or look-up table to guide the interpretation of measured data. *Mikic et al.* [35] present an inte-

grated system for automated recovery of both human body model and motion from multi-view image sequences. Model acquisition is based on a hierarchical rule-based approach to body part localization and labelling. Previous knowledge of body part shape, relative size, and configuration is used to segment the visual-hull. A Kalman filter is then used for human motion reconstruction between frames. A voxel labelling procedure is used to allow large inter-frame movements. *Cheung et al.* [24] first reconstruct a model of the skeleton structure, shape, and appearance of a person and then use it to estimate the 3D movement. Tracking is performed by hierarchically matching the approximate body model to the visual-hull using color matching along the silhouette boundary edge.

3. *Direct model*: These methods use an explicit 3D model representation of human shape and skeleton structure to reconstruct the pose. Most of the approaches employ an analysis-by-synthesis methodology to optimize the similarity between the model projection and observed images. Most of the approaches employed deterministic gradient descent techniques to estimate changes in pose. For instance, *Plankers and Fua* [36] proposed an upper body tracking method of arm movements with self-occlusion using stereo and silhouette cues. A common limitation of gradient descent approaches is the use of a single pose which is updated at each time step. Thus, if there is a rapid movement or visual ambiguities then the pose estimation may fail. To achieve more robust tracking, techniques which employ a deterministic or stochastic search of the pose state space have been investigated [37], [11].

3.2.3 Geometry tracking

Going a bit further on "Direct model" pose estimation methods, we find recent works that make use of actual 3D model templates of the actors in order to track motion of both skeleton and geometry [38]. *Zhang et al.* [39] present a system for recovering the 3D characters motion from multiple image sequences that supports automatic body model acquisition. A subject-specific voxel body model that properly fits to the shape of the subject being tracked is generated from the multi-view volume data. Then a hierarchical pose search method is employed to estimate the pose by matching the voxel model to image features. They use a particle-based stochastic search algorithm and introduce a robust metric, which is incorporated with joint limits, physical constraints and combines multiple 3D cues such as volume spatial and 3D scene flow motion information.

Other authors have also explored how segmentation may help to both pose estimation and geometry tracking. *Liu et al.* [40] propose a method that supports tracking of multiple characters by using multiview image segmentation. A probabilistic appearance, shape and pose framework is used to segment input images to assign each pixel uniquely to one actor. Given the articulated template models of each of the actors and the labeled pixels, both skeleton motion and geometry changes are tracked by employing an optimization scheme one by one to each individual. In a similar follow-on work by *Wu et al.* [41] authors exploit detailed BRDF information and scene illumination to improve both pose

tracking and surface refinement in general scenes. Their proposed system is able to work under uncontrolled background and lighting conditions by using a single moving stereo camera.

Using RGB-D sensors for motion capture have also been explored. *Berger et al.* [42] investigate on reducing or mitigating the noisy effects of using multiple RGB-D sensors, hence allowing motion capture from all angles. They systematically evaluate the concurrent use of one to four Microsoft Kinect sensors, including either calibration, error measures and analysis, and present a time-multiplexing approach.

3.3 3D reconstruction

3.3.1 General reconstruction

Mu et al. performed one of the first contributions to the field [43]. In their work they propose a complete 3D model reconstruction system by using a combination of two different depth sensors, but using different sensor implementations: the first is based on a phase-measuring time-of-flight principle by emitting near-infrared light, and the second is based on "light-wall" generation by a square laser pulse. By rotating an object in front of the capture device, they obtain a sequence of color and depth images. They align and merge the data of each frame into the whole 3D model by using the well-known ICP algorithm and the volumetric method. They claim to end up having errors lower than 1% between the reconstruction and the ground truth. The limitations of their system are problems of specular reflections and light inter-reflection (due to the use of infrared light).

Cui et al. present a similar work using a Kinect device [44]. Either rotating an object in front of a static Kinect or moving the Kinect around an object, they obtain the color and depth information. First, they apply a super-resolution approach to each chunk of 10 frames. Then, each superresolution chunk is globally aligned using ICP as well. Finally, they generate a 3D mesh by using Poisson reconstruction.

A novel interactive approach for 3D modeling of indoor environments is proposed by *Du et al.* [45]. They present a tool which allows a non-expert user holding a Kinect device to freely move around an indoor environment while a 3D reconstruction is generated. The systems automatically detects failures in the capture process and asks the user to re-scan the corresponding part of the environment. Moreover, the user can explore the 3D model looking for incomplete spots and guide the application to scan those missing parts. Figure 3.1 shows a 3D reconstruction of an indoor environment provided by their tool.

A performance boost in 3D reconstruction using a Kinect sensor is given by *Newcombe et al.* [46] [47]. They present a new system called *KinectFusion* which generates 3D reconstructions in real-time using a single moving Kinect and commodity graphics hardware. The overall idea is similar to the previous ones by *Mu et al.* and *Cui et al.*, but they propose a novel pipeline that uses the parallelization power of the GPU. They utilize

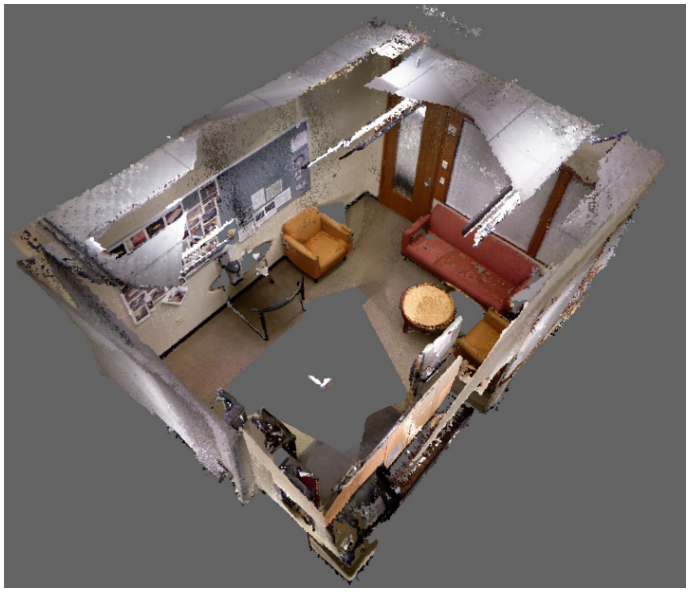


Figure 3.1: 3D indoor reconstruction

coarse-to-fine ICP not only to align and merge the 3D data over time, but also to compute the current sensor pose to allow surface prediction. These improvements allow the system to perform in real-time and obtain even better results (see figure 3.2).

Follow-on works demonstrates *KinectFusion* system has several limitations. Even though frame-to-model registration is more accurate than frame-to-frame, it is not perfect and accumulated errors over long capture trajectories may break the 3D reconstruction due to lack of loop closure handling (amongst others). *Henry et al.* apply global optimization to minimize those errors [48]. Loop closures are detected by matching features from both depth and color data. A graph that connects all pairs of consecutive frames and closes the loops is built. Then, frame-to-frame registration is carried out and global optimization is performed in order to globally distribute the errors. This method however, does not handle complex trajectories.

Zhou et al. present an offline registration and integration pipeline based on points-of-interest detection in order to deal with complex trajectories [49]. They detect densely scanned points of interest and preserve the geometric detail in the surroundings during the global optimization process. Figure 3.3 shows a results comparison between *KinectFusion*, RGB-D SLAM and their approach.

3.3.2 Avatar/Human body reconstruction

Many researchers have specialized into avatar/human body 3D reconstructions. *Zollhöfer et al.* present an algorithm to create personalized avatars from 3D face scans taken with

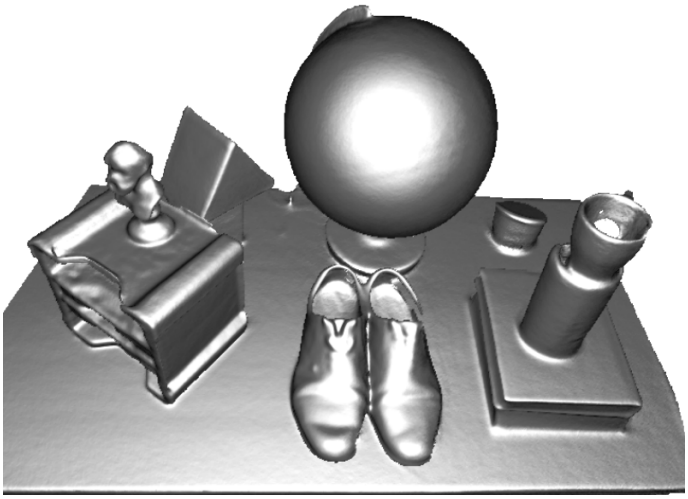


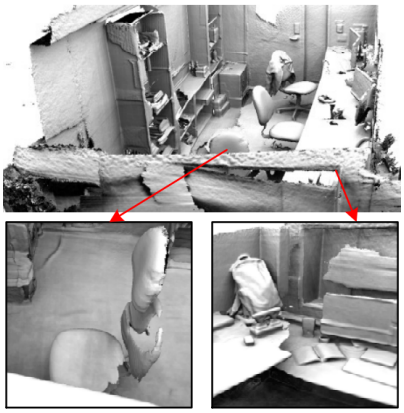
Figure 3.2: KinectFusion 3D reconstruction

a Kinect device [50]. Basically, they obtain both depth and color data of the user's face and make use of it to deform and texture a morphable face model. They first register the depth data with the average face model, then perform the corresponding deformation which best fits to the depth map, and finally, apply the RGB texture. The result is a high-quality personalized face model which can be analysed, animated or modified as well as integrated into a pre-existing avatar model.

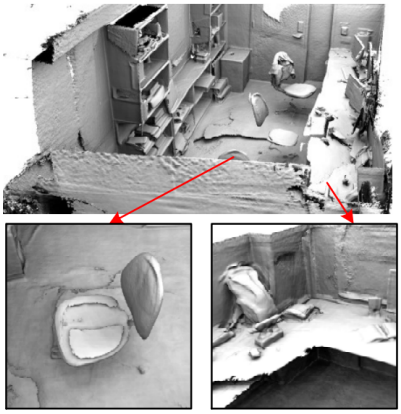
Rather than only reconstructing the human face, *Tong et al.* present a capturing system consisting in three Kinect sensors in order to increase the overall quality of the reconstruction [26]. They use two Kinects to capture the upper and lower parts of the body (far enough of each other in order to avoid overlapping) while using the third one to capture the middle part from the opposite direction. This way they try to avoid interference between Kinect sensors. The three sensors are attached to a self-turning platform where the user will stand (as rigid as possible) during the capture task. Figure 3.4 shows some of the results obtained by their approach.

Trying to avoid the burden of using multiple sensors or very complex setups, and aiming to reach a wider number of users, other authors present single-sensor capture systems. *Wang et al.* present a system where the user turns in front of a fixed Kinect device [51]. They make use of a cylindrical representation of body parts in order to detect four key poses from the captured data. These key poses are the only ones being used to generate a 3D mesh after a registration process based on matching the nodes of the cylindrical representation tree. The results of their work are promising but lack of fine detail.

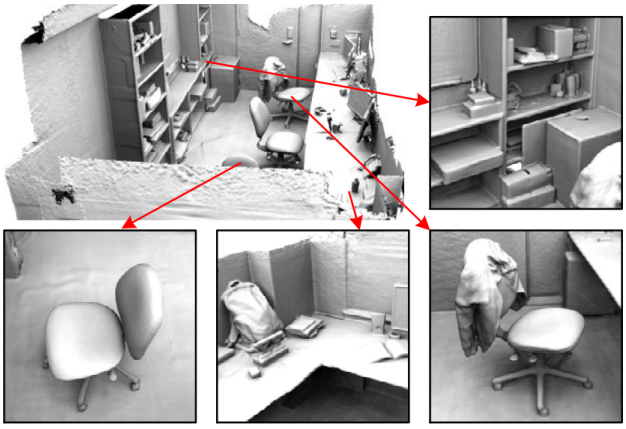
Another work in the same line is carried out by *Cui et al.* [52]. They present KinectAvatar, an automatic system to produce full human body reconstructions. As in the previous cited work, the user turns in front of a fixed Kinect sensor. They take into account color constraints to produce super-resolution depth scans which are then aligned by using



(a) KinectFusion



(b) RGB-D SLAM



(c) Point-of-interest optimization

Figure 3.3: KinectFusion, RGB-D SLAM and point-of-interest optimization approaches comparison

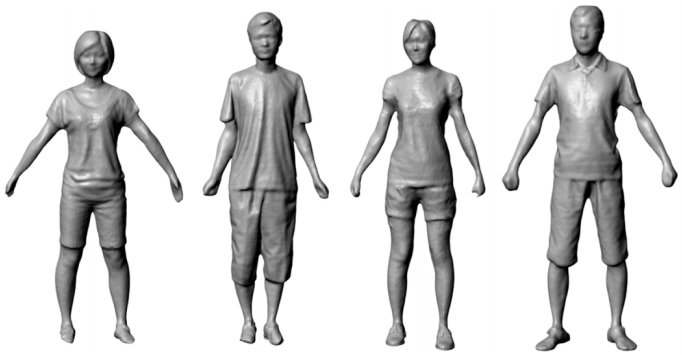


Figure 3.4: Reconstruction results from *Tong et al.* approach

rigid and non-rigid registration procedures under a probabilistic model. Even though super-resolution depth scans improve the results, the fact the user needs to be two meters apart from the sensor harms the final quality of the reconstruction. Some results of their approach are shown in figure 3.5.

Figure 3.5: Reconstruction results from *Cui et al.* approach

Li et al. address the single-sensor reconstruction again in [27]. They aim their work to be used by any user at home to obtain self 3D portraits. They ask users to adopt any desired pose in front of the Kinect. The user must turn (keeping the same pose) about 45 degrees per scan so the system captures a complete spin. A difference of this work is they make use of the tilt motor of the Kinect device to achieve a higher field of view. This allows the user to stand closer to the sensor, thus improving the final quality of the scans.

They apply an ICP algorithm to fuse the different frames of a view capture and multi-view non-rigid registration in order to align the different views. This approach generates outstanding results as can be seen in figure 3.6, even allowing to 3D-printing the resulting models.

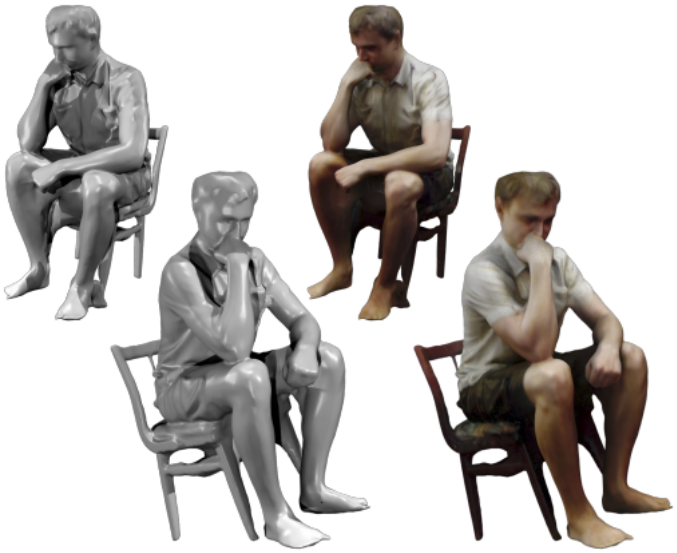


Figure 3.6: 3D self-portrait

Shapiro et al. leveraged the previous cited work to build a rapid personalized avatar capture system in order to use them in animation and simulation [28]. They improved the system by only asking the user to adopt a fixed pose from four points of view 90 degrees apart from each other. In the per-view scan step, they lifted the *KinectFusion* algorithm described in the previous section in order to obtain super-resolution scans in a very fast and efficient way. Additionally, rather than using ICP for the registration step, they make use of a contour-coherence optimization method. They claim this approach produces a high quality personalized avatar in about four minutes that can be animated by using pre-existing animations.

Part II

Development

Chapter 4

Capture System Prototype

We have designed a motion capture tool prototype built on top of OpenNI2 in order to support capture from multiple Kinect v1 and v2 sensors, although nothing would keep a user from using different devices as long as they are supported by OpenNI2.

The construction of this prototype follows a modular design, where several modules with specific functionalities each, interact with each other providing the whole sequence of steps and features to successfully capture a 3D representation of the movement of a certain actor.

Additionally, in order to support multi-sensor setups, multiple interconnected computers are needed. The proposed tool also acts as a distributed system over by using a very simple network module and protocol to synchronize several instances of the tool.

Diagram 4.1 gives an overview of all modules in our system and how they interact.

The following sections will describe each of the capture system modules in more detail, as well as the performance script that must be followed in order to satisfy some modules' requirements.

4.1 Performance script

In order to simplify the logic of different modules of our capture system, we imposed several restrictions to how a performance must be recorded.

First, a user-given number of seconds of the capture area without including any actor is recorded. These frames only including static objects of the scene will be used by the background removal module (subsection 4.4.1) in order to isolate actor's performance.

Once the backgroud is captured, the recording is paused several seconds (user-given value) to let the actor enter the capture area before starting the performance. Pausing the

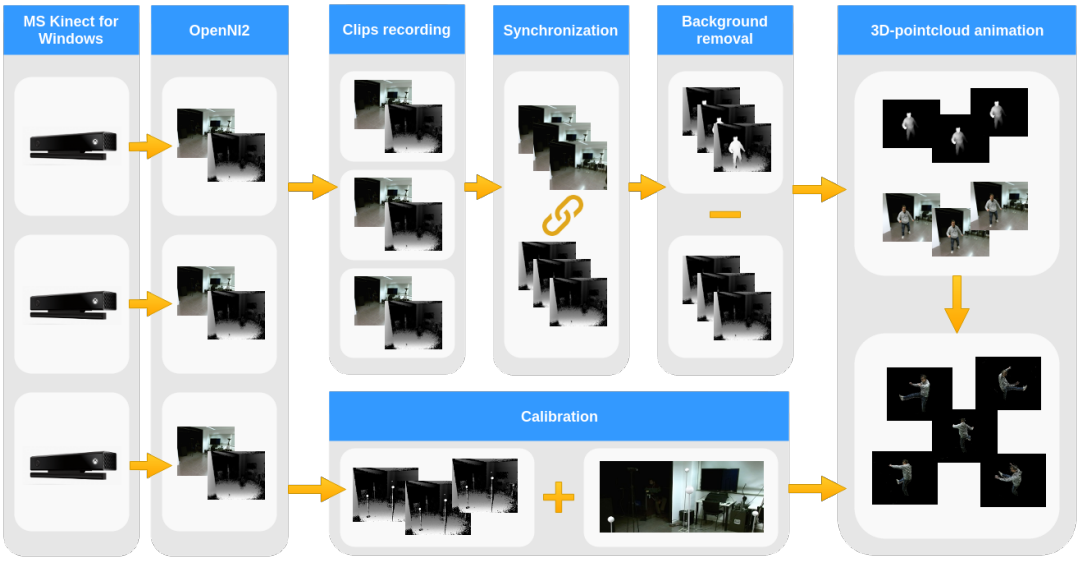


Figure 4.1: Capture system overview

recording will introduce a rather big jump in frames timestamps, resulting in several blank frames (completely black) if we were to reproduce the whole clip timespan. Such blank frames are used by the synchronization module (subsection 4.3.1) in order to estimate a sync point.

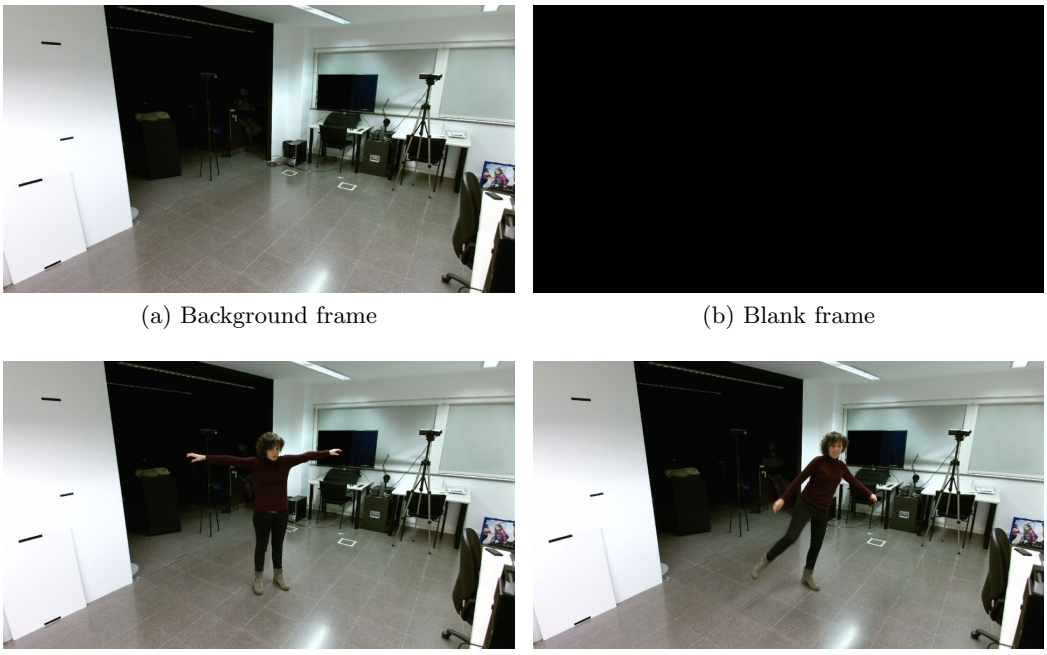
Actor's performance must always start by the actor adopting the T-pose for a few seconds. That will help later in the post-processing stage to both generate a template avatar (see chapter 6) and estimate the first avatar-pointcloud matching pose to be an input of the motion capture data extraction algorithm (see section 8.1).

After the above is satisfied, the actor is free to start performing. Figure 4.2 shows a frame of each of the mentioned script checkpoints.

4.2 Environment setup

One of the key points of our tool to support constraintless environment layouts is redundancy. It must support several Kinect v1 and v2 sensors working simultaneously. However, each sensor 3D data will be generated using its sensor local reference system.

In order to process multiple sensors 3D data, we need them to be expressed in terms of a common reference frame. Thus, a 3D calibration module is required to find a coordinate system common to all sensors.



(a) Background frame (b) Blank frame

(c) T-pose (d) Performance

Figure 4.2: Performance script checkpoints

4.2.1 Calibration

In order to have all generated 3D data represented on the same common coordinate system, we run a calibration algorithm to compute the extrinsic parameters of each sensor. The employed calibration algorithm is based on solving the absolute orientation problem as the well-known ICP algorithm does [53]. Given a source and target lists of 3D points S and T where every point S_i is linked to a target point T_i, the algorithm minimizes following energy:

$$\sum_i ||R \cdot S_i + T - T_i||^2$$

Where R and T are the rotation matrix and translation vector that determine the extrinsic parameters of the sensor being calibrated.

Currently, finding source and target 3D points is a manual task performed by the user. Both depth streams of the sensor being calibrated and the reference sensor are displayed on separate windows. The user then must select at least three different points on the source window and the corresponding target points on the reference window.

Both source and target 3D points might be automatically found by using state-of-the-art computer vision algorithms, but we left such integration for future work (8.3).

In our experiments, we found that using some sort of physical setup can facilitate the calibration task and improve the results. Figure 4.3 shows a very simple rig used in our experiments, consisting in four styrofoam balls distributed all over the capture area at different heights.

(a) Kinect #1

(b) Kinect #2

(c) Kinect #3

Figure 4.3: Calibration setup

4.3 Clips recording

Our proposed motion capture data extraction method from 3D generated pointclouds, using multiple Kinect sensors, is not a doable task in real-time. Processing of a pointcloud from a single sensor to find motion data is a time-consuming task on its own. It is aggravated by the fact that several pointclounds are actually used from multiple sensors. Additionally, given that multiple computers are required, centralization of the huge chunks of generated data is required. Making all data available to the motion capture data extraction algorithm in real-time would involve complicated compression algorithms and network protocols.

Therefore, our prototype first records users performances in a suitable format that can be processed offline. Making use of OpenNI2 built-in recording functionality eases such task, producing ONI files that can be rendered later on. It allows to record color, depth, and infrared data streams, and play the resulting ONI file by creating a virtual OpenNI2

device which simulates a physical sensor.

Some of the additional pros of working with ONI files are:

- Playback speed can be modified to suit processing or debugging needs
- A recorded clip can be paused at any time, allowing tool users to inspect every frame separately
- It also allows manual rendering of recorded frames, enabling frame synchronization of several clips (4.3.1) or spending as much processing time as needed per frame

4.3.1 Synchronization

Dealing with multiple recorded clips from different sensors means a synchronization mechanism is required. Given an instant of the record, we want data from all sensors to represent such instant of time.

Although all Kinect sensors are expected to feed OpenNI2 with frames at 30fps, different sensors have different physical clocks, and different computers may perform in a different way. Both the rate at which frames are recorded, and when each sensor actually starts to capture data, are affected by such differences in hardware.

For instance, in a two-Kinect setup, frame 200th of sensor 1 and sensor 2 will not represent the same precise instant of time.

Fortunately, frame timestamps within an ONI file can be queried through the OpenNI2 set of APIs. Using frame timestamps our tool can both:

- Find a initial synchronization point across all clips, an use each of the timestamps as T_0 for each clip
- Manually update each clip to render the closest frame to a given T_i. That is, for a two-Kinect setup:

$$\{F_{a,j}, F_{b,k}\} | a \neq b, T_j < T_i < T_{j+1}, T_k < T_i < T_{k+1}$$

  where

  - $F_{a,j}$ is the jth frame of sensor a
  - T_j the instant of time $F_{a,j}$ was recorded
  - $F_{b,k}$ is the kth frame of sensor b
  - T_k the instant of time $F_{b,k}$ was recorded

Currently, finding the initial synchronization point across all clips is done by looking for the first blank frame (completely black) after the background frames (see performance

script in section 4.1). Our system also allows the user to manually synchronize the clips if needed.

More sophisticated synchronization mechanisms are described in subsection 8.4.

4.4 Pointcloud data

Given depth and color frames from a certain sensor at a given time of the recorded actor performance, a 3D colored pointcloud of the scene is generated. Each valid pixel from the depth frame (depth value within the valid range according to sensor specifications) will become a 3D point in the space. Thanks to depth-to-color image registration (see subsection 5.3.1), corresponding pixel (same normalized image coordinates) from the color frame, will encode the color of the 3D point.

Basically, in order to get the 3D coordinates of a given depth pixel, we apply the following formula:

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = d \times U_d \times N_d \times \begin{pmatrix} x_d \\ y_d \\ 1 \end{pmatrix}$$

where

- $(x_w, y_w, z_w)^T$ are the 3D coordinates in world space (sensor reference frame)
- $(x_d, y_d, 1)^T$ are the homogeneous depth pixel coordinates
- d is the pixel's depth value in millimeters
- U_d is the depth sensor inversed projection matrix (unprojection matrix)
- N_d is a normalization matrix to transform from pixel coordinates to normalized depth image coordinates

- where N_d is defined in terms of depth frame width (w_d) and height (h_d):

$$N_d = \begin{pmatrix} 1/w_d & 0 & -0.5 \\ 0 & 1/h_d & -0.5 \\ 0 & 0 & 1 \end{pmatrix}$$

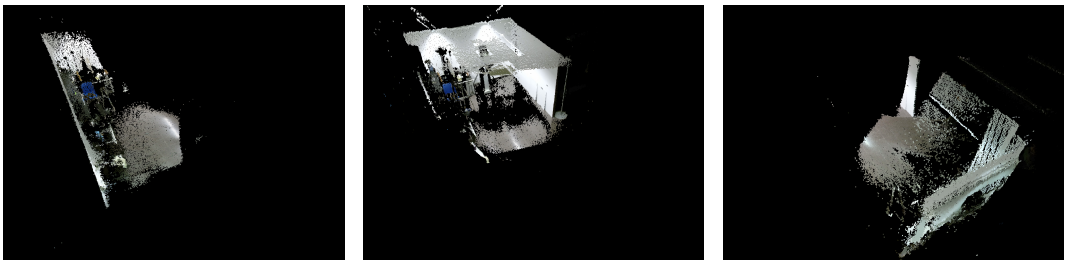
, and U_d can be estimated from both sensor horizontal (fov_h) and vertical (fov_h) field of view values:

$$U_d = \begin{pmatrix} 2 \cdot \tan(0.5 \cdot fov_w) & 0 & 0 \\ 0 & 2 \cdot \tan(0.5 \cdot fov_h) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

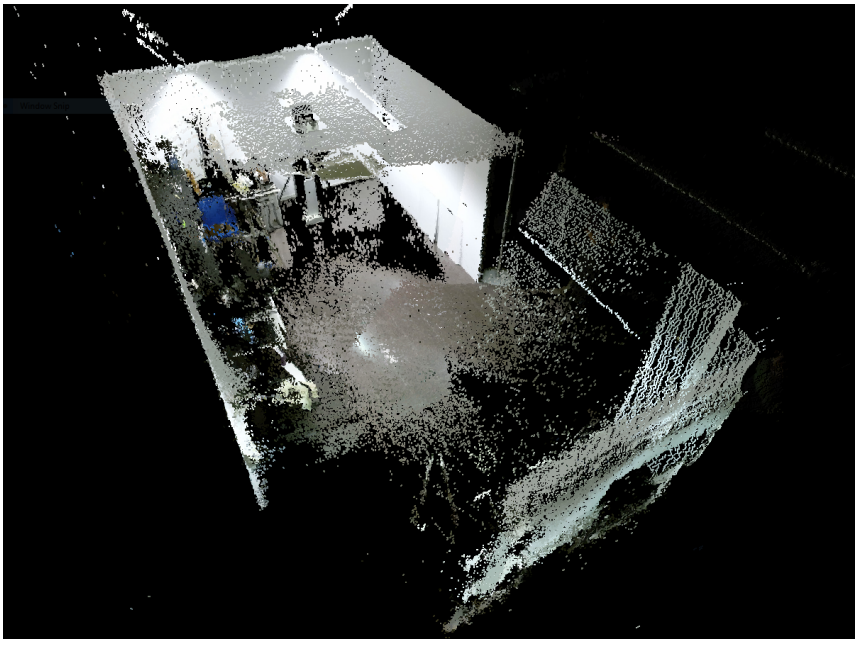
Corresponding 3D pointclouds from other sensors can also be generated. By using the previously computed extrinsic parameters matrix at calibration time (see subsection 4.2.1), we

can then transform all generated pointclouds to be on the same reference frame, resulting in a whole scene 3D pointcloud representation.

Figure 4.4 shows pointclouds generated from three different sensors and the resulting combination of them.



(a) Kinect #1 (b) Kinect #2 (c) Kinect #3



(d) Combined 3D-pointcloud

Figure 4.4: 3D-pointclouds

4.4.1 Background removal

Kinect sensors will capture the whole scene geometry of the environment. In order to isolate dynamic objects (actor), we apply a simple background removal algorithm based on discarding depth values.

Since the first frames of a clip only contain static geometry (see section 4.1), i.e. background, a mask of minimum depth values can be pre-computed at clip loading time. Such mask basically encodes the closest-to-the-camera depth value per pixel when only static objects are captured. Around 50 frames are used in order to minimize errors due to noise produced by sensor scan patterns.

Whenever background removal is enabled in our tool, static objects are filtered out from every generated depth frame according to the following equation:

$$fd_{i,j} = \begin{cases} d_{i,j}, & \text{if } t \cdot b_{i,j} > d_{i,j} \\ 0, & \text{otherwise} \end{cases}$$

where

- $fd_{i,j}$ is the filtered depth pixel (i, j)
- $b_{i,j}$ is the background mask pixel (i, j)
- $d_{i,j}$ is the original depth pixel (i, j)
- t is a background removal factor controlled by the user. From our experiments, a factor compressed within $[0.9, 1.1]$ range will give best results.

Figure 4.5 shows the resulting depth frame after applying the background removal algorithm.

4.5 Distributed capture system

Due to lack of resources on a single computer, in order to support more than a couple of sensors working simultaneously, we run into the need of using several instances of the tool, running on different computers. To work around this issue, our tool uses a message-oriented network layer to synchronize different instances of the application.

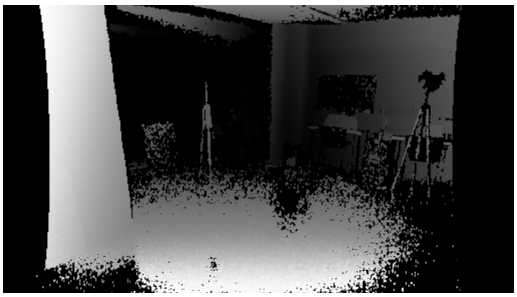
We opted for a master-slave scheme. One of the application instances will take the master role, and all others will be slaves. Slaves will be configured to connect to the master, which will be the only instance they will be able to communicate with.

4.5.1 Network protocol

Upon application initialization, two different TCP sockets are opened:

- Socket at port 5560 is configured in a one-way communication fashion using a publisher-subscriber pattern. Slaves will subscribe to this port, and master (the publisher) will send commands over this socket.

(a) Background color



(b) Background depth

(c) Foreground color

(d) Foreground depth

(e) Removed background from depth frame

Figure 4.5: Background removal

- Socket at port 5561 uses a two-way communication request-reply pattern instead. Slaves will send command requests to master over this socket, and master will acknowledge received requests when it forwards the command to all slaves.

All command messages have a very simple structure:

- Command token: a 4-byte field that indicates what command the message encodes.
- Size: a 4-byte field that indicates the byte-length of the command data.
- Data: an arbitrary length field that contains specific data of the command.

Thus, decoding commands would be as simple as identifying the command token and passing its data along to the proper handling method.

Current implementation of the network protocol includes following 0-data commands:

- *ACK*: Acknowledge slave requests when they are processed.
- *START_RECORD*: Start a new recording session.
- *PAUSE_RECORD*: Pause the current recording session.
- *STOP_RECORD*: Stop current recording session and dump the generated ONI file.

In section 8.6, we propose several extensions of the network protocol in order to improve interaction between master and slaves application instances.

Chapter 5

OpenNI2 Microsoft Kinect v2 driver

As mentioned in section 2.1.2, as part of this project we were required to write an OpenNI2[1] driver for Microsoft Kinect v2 sensor[2].

As of the time this tool was developed, only the Microsoft 'Kinect for Windows' driver and SDK were available in order to manage a Kinect v2 sensor. Therefore, the OpenNI2 driver was written on top of the Microsoft one, which implies it only has support for Windows.

Nowadays, there seems to also be an open-source reversed-engineered driver out there with support for Linux[54]. This document does not cover details about such driver.

This chapter describes some of the implementation details of the driver and how the underlying Microsoft 'Kinect for Windows' API has been used to implement the required OpenNI2 set of internal structures and functions.

5.1 Kinect for Windows API

Microsoft 'Kinect for Windows' driver and SDK offer a set of structures and methods for managing Kinect v2 sensors. Following, a brief description of the minimal set of structures and methods required to provide support for Kinect v2 through OpenNI2 is given.

5.1.1 IKinectSensor

`IKinectSensor` object represents a physical Kinect v2 sensor. It is the top level structure and entry point to manage a Kinect sensor and all its different data streams (color, depth, and infrared).

Unlike drivers and SDKs for previous versions of the Kinect sensor, with Kinect v2 driver and SDK we can manage only a single Kinect sensor connected to the computer. Thus, a global API is provided to retrieve the corresponding `IKinectSensor` instance:

- `::GetDefaultKinectSensor(IKinectSensor* sensor /* Out */)`

Once the sensor object is obtained, among others, it offers several APIs to connect to/disconnect from the physical device

- `IKinectSensor::Open()`
- `IKinectSensor::Close()`

, get its unique string identifier

- ```
  IKinectSensor::get_uniqueKinectId(UINT   stringSize /* In */,
                                    WCHAR* stringId   /* Out */)
  ```

, obtain the global coordinate mapper instance

- `IKinectSensor::get_CoordinateMapper(ICoordinateMapper* mapper /* Out */)`

, and get the different frame sources

- ```
  IKinectSensor::get_{Color,Depth,Infrared}FrameSource(
          IColorFrameSource** source /* Out */)
  ```

5.1.2 I{Color,Depth,Infrared}FrameSource

A frame source (color, depth and infrared) gives access to the corresponding data stream. It allows to perform several queries on the given source, but most importantly, it lets clients access the specific frame reader to acquire frames with.

Within the OpenNI2 driver implementation, frame sources are only used to both obtain corresponding frame description and frame reader objects:

- ```
  I{Color,Depth,Infrared}FrameSource::get_FrameDescription(
          IFrameDescription** desc /* Out */)
  ```
- ```
  I{Color,Depth,Infrared}FrameSource::OpenReader(
          I{Color,Depth,Infrared}FrameReader** reader /* Out */)
  ```

5.1.3 IFrameDescription

The `IFrameDescription` object lets us query several details about a specific data stream such as bytes per pixel, height and width, or the horizontal and vertical field of view values.

Within the OpenNI2 driver implementation following self-explanatory methods were used:

- `IFrameDescription::get_HorizontalFieldOfView(float* hfov /* Out */)`
- `IFrameDescription::get_VerticalFieldOfView(float* vfov /* Out */)`

5.1.4 I{Color,Depth,Infrared}FrameReader

Frame readers and sources are closely related, but while a frame source offers more general APIs over a specific data stream, a frame reader provides the means to acquire and read the individual generated frames from the stream.

Although frame readers offer both callback-based and query-based frame acquisition mechanisms, the OpenNI2 driver implementation only makes use of the query-based one. 'Kinect for Windows' clients will call into the following set of functions whenever they want to retrieve a new frame:

- ```
  I{Color,Depth,Infrared}FrameReader::AcquireLatestFrame(
          I{Color,Depth,Infrared}Frame** frame /* Out */)
  ```

It is a blocking call, i.e. it will block until a new non-already-acquired frame is available.

5.1.5 I{Color,Depth,Infrared}Frame

Encapsulates the underlying buffer of a generated frame. It offers different buffer-related APIs such as data format queries or format conversion methods.

Color frames handling is a bit trickier than depth or infrared due to supporting different color formats. While depth and infrared buffers are accessed by simply calling into

- ```
  I{Depth,Infrared}Frame::AccessUnderlyingBuffer(
          UINT* bytes     /* Out */,
          UINT16** buffer /* Out */)
  ```

, color frame object provides methods to query the color format and access the underlying buffer directly if it is of desired format, or convert to other formats otherwise:

- `IColorFrame::get_RawColorImageFormat(ColorImageFormat* format /* Out */)`
- ```
  IColorFrame::AccessRawUnderlyingBuffer(
          UINT* bytes     /* Out */,
          UINT16** buffer /* Out */)
  ```
- ```
  IColorFrame::CopyConvertedFrameDataToArray(
          UINT             bytes  /* In */,
          BYTE*            buffer /* Out */,
          ColorImageFormat format /* In */)
  ```

5.1.6 ColorImageFormat

Color format enumeration. It can be one of:

- None
- Rgba
- Yuv
- Bgra
- Bayer
- Yuy2

5.1.7 ICoordinateMapper

`ICoordinateMapper` represents a global object that provides means to translate 2D points from one reference system to another. E.g. translate coordinates from color image space to depth image space in order to fetch the color pixel corresponding to a certain depth pixel.

In the OpenNI2 driver, only mapping from depth points to color points is required. Such translation is achieved by using

- ```
  ICoordinateMapper::MapDepthFrameToColorSpace(
          UINT             depthPointsCount, /* In */
          const UINT16*    depthBuffer,      /* In */
          UINT             colorPointsCount, /* In */
          ColorSpacePoint* colorCoordinates  /* Out */)
  ```

5.1.8 ColorSpacePoint

Trivial object that represents a 2D-space point.

5.1.9 Miscellaneous methods

Other methods from the Microsoft SDK are used as well. For instance, in order to perform time measuring for timestamps computation, CPU counters are used:

- `::QueryPerformanceCounter(LARGE_INTEGER* ticks /* Out */)`
- `::QueryPerformanceFrequency(LARGE_INTEGER* freq /* Out */)`
  ```

Thus, a given instant of time in second units can be computed like

$$t_i = ticks_i / freq_i$$

and the elapsed time between two events would be

$$elapsed = t_i - t_j, i > j$$

5.2 OpenNI2 minimal required features

OpenNI2 accommodates a set of base driver classes and methods a specific driver implementation must be based off in order to provide support for a certain hardware. Kinect v2 driver case is no different. The minimal logic and set of structures that must be implemented to add support for Kinect v2 is composed by:

- Driver class
  - Driver initialization and shutdown functions
  - Device enumeration mechanism
  - Device connection and disconnection methods
- Device class
  - Data stream enumeration mechanism
  - Data stream connection methods
- Data streams classes
  - Data buffer acquisition and handling functions

Following subsections describe in more detail each of the implemented classes and methods.

5.2.1 Kinect2 driver implementation

The Kinect2 driver class implements the required logic to enumerate and connect to the available Kinect v2 sensors.

Upon driver initialization, the available sensors are enumerated and identifiers are cached into a device information array.

The 'Kinect for Windows' driver only allows connecting to a single Kinect v2 sensor. Therefore, in order to fill up the device information array with sensor information, it is temporarily fetched and open to query its unique sensor identifier.

Following snippet of code demonstrates how it is done using the APIs described in section 5.1.1 (note that some parts of the code were omitted in favor of readability):

```
OniStatus Kinect2Driver::initialize(...) {

    ...Initialize base driver...

    // Get sensor instance and connect
    ::GetDefaultKinectSensor(&pKinectSensor);
    pKinectSensor->Open();

    pKinectSensor->get_IsAvailable(&available);
    if (!available) {
        return ONI_STATUS_NO_DEVICE;
    }

    // Get sensor info
    pKinectSensor->get_UniqueKinectId(ONI_MAX_STR, sensorId);

    ...Cache sensor vendor, name, and URI (unique ID)...

    pKinectSensor->Close();

    return ONI_STATUS_OK;
}
```

Later on, a specific device connection can be requested from the application by using the corresponding URI (unique ID). The driver will check the given URI against the cached ones at initialization time, and if a match is found, it opens the corresponding sensor, leaving it opened this time though, and returns a OpenNI2 device instance wrapping the native sensor:

```
DeviceBase* Kinect2Driver::deviceOpen(const char* uri, ...) {
    'for_all_cached_device_uris' {
        if (iterator->uri == uri) {
            ::GetDefaultKinectSensor(&pKinectSensor);
            pKinectSensor->Open();

            ...Make sure the sensor ID actually matches the given URI...

            return new Kinect2Device(pKinectSensor);
        }
    }
    return NULL;
}
```

Upon device disconnection request or driver shutdown, all native sensor resources are properly closed and the OpenNI2 objects destroyed.

5.2.2 Kinect2 device implementation

The Kinect2 device class is in charge of listing available sensor data streams and provide means to connect to each of them.

At device creation time, supported sensor data streams and their supported video-modes (frame buffer size, format, and stream output rate in frames per second) lists are initialized.

Although it is not an OpenNI2 driver requirement, the Kinect2 device implementation supports infrared data streaming along with color and depth.

See subsection 5.3.2 for more information about supported video-modes.

Whenever an application wants access to a certain data stream, the following device function will create and return an instance of an OpenNI2 Kinect2 stream:

```
StreamBase* Kinect2Device::createStream(OniSensorType type) {
    if (type == color)    return new Kinect2ColorStream();
    if (type == depth)    return new Kinect2DepthStream();
    if (type == infrared) return new Kinect2IRStream();
}
```

5.2.3 Kinect2 stream implementation

The base Kinect2 stream class implements several functions and mechanisms common to all specific streams. It will basically implement the whole stream logic but the frame handling.

First, at initialization time, it allocates the frame buffer where generated frames will be dumped, opens the corresponding frame reader by calling into the appropriate 'Kinect for Windows' API, and sets the default video-mode settings. Following code snippets give some hints about how frame buffers and frame readers are created (note that declarations and error handling were omitted in favor of readability):

```
void Kinect2BaseStream::createFramebuffer() {
    m_framebuffer = new BYTE[<max_width>*<max_height>*<bpp>];
}
```

Where:

- `<max_width>`: 1920 pixels for color streams and 512 pixels for depth and IR streams
- `<max_height>`: 1080 pixels for color streams and 424 pixels for depth and IR streams

- <bpp>: 4 bytes (BGRA) for color streams and 2 bytes for depth and IR streams

```
void Kinect2BaseStream::openFrameReader()
    if (m_sensorType == ONI_SENSOR_COLOR) {
        m_pKinectSensor->get_ColorFrameSource(&frameSource);
        frameSource->OpenReader(&m_pFrameReader.color);
    }
    else if (m_sensorType == ONI_SENSOR_DEPTH) {
        m_pKinectSensor->get_DepthFrameSource(&frameSource);
        frameSource->OpenReader(&m_pFrameReader.depth);
    }
    else { // ONI_SENSOR_IR
        m_pKinectSensor->get_InfraredFrameSource(&frameSource);
        frameSource->OpenReader(&m_pFrameReader.infrared);
    }
}
```

Second, it implements several common property setter and getter functions. The common properties list includes:

- Video-mode settings: Hard-coded as-per device specifications (see subsection 5.3.2 for more details).
- Data stream type: OpenNI2 inherent property.
- Horizontal and Vertical field of view: Queried through `IFrameDescription` (5.1.3).
- Cropping settings: OpenNI2 inherent property.

Finally, in order to acquire frames as they are generated by corresponding data streams, `Kinect2BaseStream` provides the means to start and stop a separate thread that will be in charge of calling into the blocking `AcquireLatestFrame()` function from the corresponding frame reader (5.1.4), and properly handling the given fram by passing control to the specific stream implementation. Whenever an OpenNI2 stream is created by the application, a new thread will be launched and kept running as long as its associated stream is valid.

5.2.4 Kinect2 color frame handling

Whenever a color frame needs to be handled, the `Kinect2ColorStream` class jumps in. Currently, BGRA format is used by 'Kinect for Windows' driver for its underlying color buffer. Thus, if generated frame format matches BGRA (most problably), the underlying buffer is accessed directly and copied to the previously allocated frame buffer; otherwise, it is converted to BGRA format first to ease the frame handling process:

```
frame->get_RawColorImageFormat(&imageFormat);
if (imageFormat == ColorImageFormat_Bgra) {
```

```
        frame->AccessRawUnderlyingBuffer(&bufferSize, &data);
        memcpy(m_pFrameBuffer.color, data, 1920*1080*sizeof(RGBQUAD));
    }
    else {
        frame->CopyConvertedFrameDataToArray(1920*1080*sizeof(RGBQUAD),
                                             m_pFrameBuffer.color,
                                             ColorImageFormat_Bgra);
    }
```

Thereafter, a new OpenNI2 color frame is created and emitted so the application can use it. OpenNI2 color frame creation involves the following:

- Setting frame metadata
  - Frame size
  - Cropping region
  - Video-mode settings
  - Frame index
  - Timestamp
- Cropping the frame buffer if needed
- Converting the BGRA frame buffer to RGB format (only color format offered by the OpenNI2 Kinect2 driver).

Following code snippet shows the unified loop to both apply cropping settings and convert to RGB:

```
const int xStride = width/m_videoMode.resolutionX;
const int yStride = height/m_videoMode.resolutionY;
const int frameX = pFrame->cropOriginX * xStride;
const int frameY = pFrame->cropOriginY * yStride;
const int frameWidth = pFrame->width * xStride;
const int frameHeight = pFrame->height * yStride;
for (int y = frameY; y < frameY + frameHeight; y += yStride) {
    for (int x = frameX; x < frameX + frameWidth; x += xStride) {
        RGBQUAD* iter = data_in + (y*width + x);
        data_out->b = iter->rgbBlue;
        data_out->r = iter->rgbRed;
        data_out->g = iter->rgbGreen;
        data_out++;
    }
}
```

5.2.5 Kinect2 depth frame handling

Unlike color frame case (5.2.4), whenever a depth frame needs to be handled, the underlying can be directly accessed (no format conversion required).

Additionally, besides cropping and setting frame metadata, the frame buffer might need special treatment if depth-to-color image registration is set. See 5.3.1 for more details.

5.2.6 Kinect2 Infrared frame handling

Handling infrared frames is easier than either color or depth frames: The underlying frame buffer can be directly accessed with no format conversion required, and creation of an OpenNI2 infrared frame will only involve cropping and setting metadata (no special treatment of the frame buffer data).

5.3 OpenNI2 extra features

Some non-OpenNI2-essential features were also implemented in order to simplify the tool written on top of the OpenNI2 driver for this project.

The most important one implements depth-to-color image registration, but other minor features such as addition of non-natively supported video-modes were also required to overcome performance challenges.

5.3.1 Depth-to-color image registration

A tool using both color and depth data streams from a Kinect device will likely want to access both RGB and depth values of a certain pixel (x, y) in order to generate a colored 3D point of the captured scene.

However, generated color and depth frames have different resolutions as well as different local reference systems due to physical location of the sensors within the Kinect v2 device. That means $Color_{(x,y)}$ and $Depth_{(x,y)}$ will not correspond to color and depth of the same point in the real scene.

Depth-to-color image registration will allow OpenNI2 clients to generate depth frames with same local reference system used by color frames.

In order to achieve the above, whenever `openni::IMAGE_REGISTRATION_DEPTH_TO_COLOR` image registration mode is set through `openni::Device::setImageRegistrationMode()` API, following conversion is applied to depth frames prior to emmiting them:

$$DRI = DCM \times DIDCM = P_c \times U_d$$

where

- DRI represents the resulting "Depth Registered Image"
- DI represents the source "Depth Image"
- P_c is the color sensor projection matrix
- U_d is the depth sensor inversed projection matrix (unprojection matrix)

We could compute both P_c and U_d from both color and depth sensors intrinsic parameters (focal length, image sensor format, and principal point). However, those are hard-coded within the 'Kinect for Windows' driver.

Alternatively, we could estimate both projection matrices from horizontal and vertical field of view values, but 'Kinect for Windows' offers a coordinate mapper class that will make the translation from one reference system to another easier (5.1.7).

By using `ICoordinateMapper::MapDepthFrameToColorSpace()`, we can obtain the array of corresponding color coordinates from a depth frame.

Following code snippet shows how a depth frame is generated with depth-to-color image registration:

```
coordinateMapper->MapDepthFrameToColorSpace(width*height,
                                            data_in,
                                            width*height,
                                            m_colorSpaceCoords);

const ColorSpacePoint* mappedCoordsIter = m_colorSpaceCoords;
for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
        const float fX = mappedCoordsIter->X*xFactor;
        const float fY = mappedCoordsIter->Y*yFactor;
        const int cx = static_cast<int>(fX + 0.5f);
        const int cy = static_cast<int>(fY + 0.5f);
        if (cx >= 0 && cy >= 0 && cx < width && cy < height) {
            unsigned short* iter = data_in + (y*width + x);
            const unsigned short d = *iter;
            unsigned short* const p = data_out + cx + cy * width;
            if (*p == 0 || *p > d) *p = d;
        }
        mappedCoordsIter++;
    }
}
```

where

- `data_in` is the source depth frame buffer
- `data_out` is the registered depth frame buffer
- `xFactor` is $width_{depth}/width_{color}$
- `yFactor` is $height_{depth}/height_{color}$

Note that above algorithm produces both horizontal and vertical gaps in the resulting depth frame due to the different aspect ratios between color and depth frames. An extra image dilation morphological step is performed to fill all empty pixels with the average value of its 1-neighboring pixels.

5.3.2 Non-native color videomodes

'Kinect for Windows' driver provides native support for a single video-mode per sensor data stream. That is:

- Color sensor
  - Resolution: 1920x1080 (*FullHD*)
  - FPS: 30
- Depth sensor
  - Resolution: 512x424
  - FPS: 30
- Infrared sensor
  - Resolution: 512x424
  - FPS: 30

Recording of the color stream through the OpenNI2 record module is nevertheless low-performant at *FullHD* resolutions, even making the whole driver to randomly crash.

In order to work around the performance issue, an extra non-native color video-mode was added to the OpenNI2 Kinect2 driver:

- Resolution: 960x540
- FPS: 30

which will generate color images of 1/4 the original size, allowing the OpenNI2 recording module to perform normally.

In order to produce 960x540 frames, a scale down step without interpolation is also performed along with cropping and format conversion in the unified color frames handling loop (5.2.4). Note that lines

```
const int xStride = width/m_videoMode.resolutionX;
const int yStride = height/m_videoMode.resolutionY;
```

define the pixel increment applied at each iteration and will discard pixels whenever the video-mode resolution is lower that the original (`m_videoMode.resolutionX < width` or `m_videoMode.resolutionY < height`).

Chapter 6

Avatar Template Creator plugin for MakeHuman

MakeHuman is an open-source 3D computer graphics tool designed for the prototyping of photo-realistic humanoid avatars [3]. As part of this project we have developed a MakeHuman plugin to import 3D-pointclouds in PLY format [55] generated by our motion capture tool (see chapter 4).

The goal of this plugin is to provide hints about an actor's complexion to the avatar designer by importing the actor's generated 3D-pointcloud of a certain frame of the performance animation. A designer will most likely import a T-pose 3D-pointclound. Using the imported 3D-pointcloud, the designer will be able to better estimate actor's complexion attributes in order to create a template avatar that could be used to generate motion capture data to animate random avatars.

The avatar template creator plugin consists of three different parts, all written in python language:

- PLY format importer. A externally developed ply loader is used to import the 3D-pointclouds.
- User interface elements to help the designer editing the avatar template (see figure 6.1):
  - Load button to import a certain PLY file.
  - Checkbox to show/hide the imported pointcloud.
  - Point size controller
  - Apply geometric transformations (translations and rotations only) to the imported pointcloud so it can be moved around.
  - Reset button to restore geometric transformations to their defaults.

– Save and load transformation buttons so the designer can save the work and continue later on.

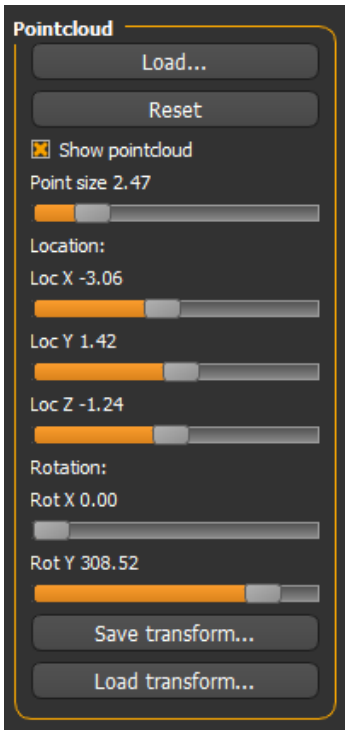


Figure 6.1: MakeHuman avatar template creator user interface controls

- OpenGL renderer interaction. The plugin must interact with MakeHuman's OpenGL engine in order to render the imported 3D-pointcloud.

Figure 6.2 shows a created avatar template and the corresponding pointcloud in MakeHuman.

(a) Overlapped

(b) Side-by-side

Figure 6.2: MakeHuman avatar template creation

Part III

Results

Chapter 7

Results

The proposed tool provides an easy way to set up multi-sensor motion capture environments. The capturing and visualization algorithms are efficient and permit both real-time raw data recording and inspection of the resulting 3D point-cloud animation. Thus, the user can easily evaluate the quality of the captured clips and restart the recording session if needed.

The result of our framework is a 3D-pointcloud animation clip of an actor's performance. Such animation can be reproduced, inspected or processed at a later stage.

Figure 7.1 presents several renders from different camera view points of a frame extracted from a recorded performance session. Despite having some remaining outliers, the point-cloud rendered could be used as-is for far away visualizations.

(a) (b) (c) (d) (e)

Figure 7.1: Resulting 3D-pointcloud performance frame from different view points

It is worth mentioning the main areas or error are concentrated around actor's feet when they are in contact with the floor. It is hard for our background removal algorithm to correctly discriminate between static and dynamic objects at such small distances.

Nevertheless, the achieved results are good enough for avatar fitting (section 8.1) or avatar reconstruction (section 8.2) as described in the future work chapter.

Two different captured performances can be watched on video by clicking on the image previews of figure 7.2 (*digital version only*). Aside from the 3D-pointcloud performance

animation, all three color and depth streams of the sensors employed are also played in sync.

(a) https://youtu.be/EGu34K5ck7c

(b) https://youtu.be/x50rutqoedA

Figure 7.2: Resulting 3D-pointcloud performance animations of two different actors

Note that with the proposed setup based on three strategically located sensors, we can capture the 3D-pointcloud to render it later on from any view point. Our setup allows us to correctly eliminate self-occlusion problems that typically arise in optical mocap systems. Therefore, despite the early stage of our results, we can predict the potential of this system to develop affordable mocap systems.

Chapter 8

Future Work

8.1 Avatar fitting to extract motion capture data

One of the biggest enhancements to be done to our tool is the addition of a motion capture data extraction algorithm so random avatars can be animated according to the actor's performance.

In order to extract motion capture data, we would perform an optimized fitting of a 3D avatar template model (with its corresponding skeleton) on the per-frame pointclouds. The resulting skeleton configuration would be used as key-frames to animate other avatars in real-time.

Our proposal is to use a variation of the well-known Simplex iterative optimization method [56]. Using our tool, a user would provide a set of avatar templates with different complexities. The algorithm would automatically find the most suitable one for the current pointcloud as a preliminary step. The user would interactively position the selected avatar to overlap the pointcloud on the first frame, which would be the input for the first iteration of the Simplex algorithm. For frames other than the first one, the resulting pose of the previous frame would be taken.

The Simplex method would run once per frame in order to find the most suitable skeleton configuration (joint angles plus root position) that minimizes the least squares distance between each point of the pointcloud and resulting avatar mesh (or any other fitting metric). Along with the resulting previous frame fitting, each of the free variables of the skeleton configuration can be given some noise perturbation. This method is expected to rapidly converge into an optimal solution according to a certain epsilon value provided by the user.

8.2 Avatar mesh reconstruction

Another enhancement to our tool would be the ability to generate real-life avatars from the resulting 3D-pointclouds animations. Generating an avatar close enough to the actor's appearance, and then animating it with motion capture data from the avatar-fitting step, would add realism to the virtual reproduction a performance.

8.3 Automatic calibration

Currently, our calibration mechanism is not fully automatic as it requires some user interaction in order to select correspondence points between two sensors to calibrate. Once the correspondence points are selected the algorithm to find the extrinsic parameters of a camera runs in a fully automatic way.

By using visual computing algorithms we could make our calibration module to automatically find the correspondence points: Instead of using white styrofoam balls to assist on the calibration process, we would use colored balls. An image processing algorithm could easily distinguish corresponding balls on sensor color streams. Since we use depth-to-image registration in order to make both color and depth images use the same local coordinate systems, the algorithm would easily fetch the depth values to compute each ball 3D-space coordinates. The rest of the calibration algorithm would remain the same, taking all balls 3D positions as input.

Additionally, better calibration results could be achieved by using the center of mass of every ball instead of a random point on their surfaces.

8.4 Synchronization

In order to synchronize the different recorded clips from different sensors we look for the first blank frame on all of them as a initial estimation of a syncpoint. Then the user can manually refine such approximation. Note that our approach assumes that frames will arrive at a steady rate, so a single syncpoint will suffice. That is not always the case as different hardware configurations and the system load at a given time will affect the processing of frames in the driver.

To better handle this problem, we should re-synchronize all clips every few seconds. A possible solution would be to use a blinking LED powerful enough to be seen by all sensors as part of the setup. Setting the LED to blink at a steady rate would give reliable syncpoints at a constant rate. By using state-of-the-art image processing algorithms, we could automatically detect all syncpoints on the color streams from all sensors, and re-synchronize all clips accordingly.

8.5 Background removal

Even though we obtained very good results with a very simple algorithm using only depth data, our background removal technique is not flawless. In areas where dynamic object are in contact with static objects, we have seen a portion of the dynamic object goes missing. For instance, the feet of an actor standing on the floor.

We believe that a more elaborated approach using state-of-the-art background removal techniques using color images as input, and combining the results with our approach, would lead to a more accurate description of the static geometry in the scene. We could then detect those false positives and recover the missing parts, while also helping to reduce some of the outliers caused by depth noise.

8.6 Network protocol

Of course, in order to improve the user interaction with our distributed tool, new commands could be added to our network protocol design. Keeping the user interface and settings of all application instances in sync would be useful. If the user wanted to make an adjustment to any of the settings before a recording session, it could be done on one of the instances of the tool and it would broadcast to the rest. Currently, only start/pause/stop recording actions are spread across the distributed system.

Chapter 9

Conclusions

In this project we have presented a proof-of-concept tool that performs offline motion capture from a variable number of commodity RGB-D sensors on constraintless layout environments. Our approach starts by capturing a set of synchronized color and depth clips from different sensors. Then, an algorithm computes a set of raw pointclouds per sensor and per frame. Static objects are filtered out by using a simple background removal algorithm using depth data as the only input. Finally, per frame processed 3D data is merged, using previously computed calibration data. The result is a 3D-pointcloud animation of an actor's performance.

Additionally, as part of this project we have written an OpenNI2 Kinect v2 driver on top of Microsoft 'Kinect for Windows' driver. All three color, depth, and infrared streams are processed to comply with the OpenNI2 framework requirements. We have extended the basic driver functionality by implementing depth-to-color image registration so both color and depth generated frames are advertised using the same local coordinate system.

We have contributed with our driver implementation to the OpenNI2 open-source project, which has been greatly appreciated by the community [57]. For the time our implementation has been publicly available, many other researchers and Kinect enthusiasts have made use of the OpenNI2 Kinect v2 driver.

Ultimately, we have also provided a MakeHuman plugin to create humanoid avatars using 3D-pointclouds as a template. This was in preparation for implementing an avatar fitting optimization algorithm to extract motion capture data to animate random virtual characters, as noted in the future work chapter.

Despite the inability of completing all our initially considered milestones for this project, and come up with a fully featured motion capture tool, we believe our work will pave the way for future lines of development regarding both motion capture and 3D reconstruction.

Our system provides a low-budget alternative to high-end mocap setups such as VICON

systems [58] at the expense of losing some precision. While VICON setups costs may amount to hundreds of thousands of euros, our 3-sensor capture system would only cost around 3,500 euros (150€/sensor plus 1000€/computer).

Bibliography

[1] Occipital, . URL `http://structure.io/openni`. Cited at pages 13 and 43.

[2] Microsoft. URL `https://developer.microsoft.com/en-us/windows/kinect/develop`. Cited at pages 14 (2) , 18, and 43.

[3] MakeHuman.org. Cited at pages 14 and 57.

[4] Wikipedia, . URL `https://en.wikipedia.org/wiki/OpenNI`. Cited at page 17.

[5] Wikipedia, . URL `https://en.wikipedia.org/wiki/PrimeSense`. Cited at page 17.

[6] Occipital, . URL `http://structure.io/`. Cited at page 18.

[7] OpenKinect.org, . URL `https://openkinect.org/wiki/Main_Page`. Cited at page 19.

[8] Wikipedia, . URL `https://en.wikipedia.org/wiki/Kinect`. Cited at page 19.

[9] D M Gavrila and L S Davis. 3-D model-based tracking of humans in action: a multi-view approach. In *Computer Vision and Pattern Recognition*, pages 73–80, 1996. ISBN 0-8186-7258-7. doi: 10.1109/CVPR.1996.517056. Cited at page 21.

[10] Christoph Bregler, Jitendra Malik, and Katherine Pullen. Twist based acquisition and tracking of animal and human kinematics. *International Journal of Computer Vision*, 56:179–194, 2004. ISSN 09205691. doi: 10.1023/B:VISI.0000011203.00237.9b. Cited at page 21.

[11] Jonathan Deutscher and Ian Reid. Articulated body motion capture by stochastic search. *International Journal of Computer Vision*, 61:185–205, 2005. ISSN 09205691. doi: 10.1023/B:VISI.0000043757.18370.9c. Cited at pages 21 and 23.

[12] Luca Ballan and Guido Maria Cortelazzo. Marker-less motion capture of skinned models in a four camera set-up using optical flow and silhouettes. *3Dpvt*, 37, 2008. URL `http://www.cc.gatech.edu/conferences/3DPVT08/Program/Papers/paper178.pdf`. Cited at page 21.

[13] Radu Horaud, Matti Niskanen, Guillaume Dewaele, and Edmond Boyer. Human motion tracking by registering an articulated surface to 3D points and normals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31:158–164, 2009. ISSN 01628828. doi: 10.1109/TPAMI.2008.108. Cited at page 21.

[14] Stefano Corazza, Lars Mündermann, Emiliano Gambaretto, Giancarlo Ferrigno, and Thomas P. Andriacchi. Markerless motion capture through visual hull, articulated ICP and subject specific model generation. *International Journal of Computer Vision*, 87:156–169, 2010. ISSN 09205691. doi: 10.1007/s11263-009-0284-3. Cited at page 21.

[15] Ross Girshick, Jamie Shotton, Pushmeet Kohli, Antonio Criminisi, and Andrew Fitzgibbon. Efficient regression of general-activity human poses from depth images. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 415–422, 2011. ISBN 9781457711015. doi: 10.1109/ICCV.2011.6126270. Cited at page 21.

[16] J. Starck and A. Hilton. Model-based multiple view reconstruction of people. *Proceedings Ninth IEEE International Conference on Computer Vision*, 2003. doi: 10.1109/ICCV.2003.1238446. Cited at page 21.

[17] Kiran Varanasi, Andrei Zaharescu, Edmond Boyer, and Radu Horaud. Temporal surface tracking using mesh evolution. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5303 LNCS, pages 30–43, 2008. ISBN 3540886850. doi: 10.1007/978-3-540-88688-4-3. Cited at page 21.

[18] Edilson de Aguiar, Carsten Stoll, Christian Theobalt, Naveed Ahmed, Hans-Peter Seidel, and Sebastian Thrun. Performance capture from sparse multi-view video, 2008. ISSN 07300301. Cited at page 21.

[19] Cedric Cagniart, Edmond Boyer, and Slobodan Ilic. Probabilistic deformable surface tracking from multiple videos. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6314 LNCS, pages 326–339, 2010. ISBN 364215560X. doi: 10.1007/978-3-642-15561-1_24. Cited at page 21.

[20] C. Barron and I.A. Kakadiaris. Estimating anthropometry and pose from a single image. *Proceedings IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2000 (Cat. No.PR00662)*, 1, 2000. ISSN 1063-6919. doi: 10.1109/CVPR.2000.855884. Cited at page 22.

[21] Carlos Barrón and Ioannis A. Kakadiaris. On the improvement of anthropometry and pose estimation from a single uncalibrated image. *Machine Vision and Applications*, 14:229–236, 2003. ISSN 09328092. doi: 10.1007/s00138-002-0088-8. Cited at page 22.

[22] V. Parameswaran and R. Chellappa. View independent human body pose estimation from a single perspective image. *Proceedings of the 2004 IEEE Computer Society*

Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004., 2, 2004. ISSN 1063-6919. doi: 10.1109/CVPR.2004.1315139. Cited at page 22.

[23] C.J. Taylor. Reconstruction of articulated objects from point correspondences in a single uncalibrated image. *Proceedings IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2000 (Cat. No.PR00662)*, 1, 2000. ISSN 1063-6919. doi: 10.1109/CVPR.2000.855885. Cited at page 22.

[24] K.M.G. Cheung, S. Baker, and T. Kanade. Shape-from-silhouette of articulated objects and its use for human body kinematics estimation and motion capture. *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, 1, 2003. ISSN 1063-6919. doi: 10.1109/CVPR.2003.1211340. Cited at pages 22 and 23.

[25] Clement Menier, Emond Boyer, and Bruno Raffin. 3D skeleton-based body pose recovery. In *Proceedings - Third International Symposium on 3D Data Processing, Visualization, and Transmission, 3DPVT 2006*, pages 389–396, 2007. ISBN 0769528252. doi: 10.1109/3DPVT.2006.7. Cited at page 22.

[26] Jing Tong, Jin Zhou, Ligang Liu, Zhigeng Pan, and Hao Yan. Scanning 3D Full Human Bodies Using Kinects, 2012. ISSN 1077-2626. Cited at pages 22 and 26.

[27] Hao Li, Etienne Vouga, Anton Gudym, Linjie Luo, Jonathan T. Barron, and Gleb Gusev. 3D self-portraits. *ACM Transactions on Graphics*, 32:1–9, 2013. ISSN 07300301. doi: 10.1145/2508363.2508407. URL `http://dl.acm.org/citation.cfm?id=2508363.2508407`. Cited at pages 22 and 28.

[28] Ari Shapiro, Andrew Feng, Ruizhe Wang, Hao Li, and Mark Bolas. Rapid Avatar Capture and Simulation using Commodity Depth Sensors. In *Conference on Computer Animation and Social Agents*, 2014. URL `http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Rapid+Avatar+Capture+and+Simulation+using+Commodity+Depth+Sensors#0`. Cited at pages 22 and 29.

[29] D.A. Forsyth and M.M. Fleck. Body plans. *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1997. ISSN 1063-6919. doi: 10.1109/CVPR.1997.609399. Cited at page 22.

[30] P.F. Felzenszwalb and D.P. Huttenlocher. Efficient matching of pictorial structures. *Proceedings IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2000 (Cat. No.PR00662)*, 2, 2000. ISSN 1063-6919. doi: 10.1109/CVPR.2000.854739. Cited at page 22.

[31] S. Ioffe and D. Forsyth. Finding people by sampling. *Proceedings of the Seventh IEEE International Conference on Computer Vision*, 2, 1999. doi: 10.1109/ICCV.1999.790398. Cited at page 22.

[32] S. Ioffe and D. Forsyth. Human tracking with mixtures of trees. *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, 1, 2001. doi: 10.1109/ICCV.2001.937589. Cited at page 22.

[33] Anuj Mohan, Constantine Papageorgiou, and Tomaso Poggio. Example-based object detection in images by components. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23:349–361, 2001. ISSN 01628828. doi: 10.1109/34.917571. Cited at page 22.

[34] Bo Wu and Ram Nevatia. Detection of multiple, partially occluded humans in a single image by bayesian combination of edgelet part detectors. In *Proceedings of the IEEE International Conference on Computer Vision*, volume I, pages 90–97, 2005. ISBN 076952334X. doi: 10.1109/ICCV.2005.74. Cited at page 22.

[35] Ivana Mikić, Mohan M Trivedi, Edward Hunter, and Pamela Cosman. Human Body Model Acquisition and Tracking Using Voxel Data. *International Journal of Computer Vision*, 53:199–223, 2003. ISSN 09205691. doi: 10.1023/A:1023012723347. URL `http://www.springerlink.com/content/q25728g2q821627j`. Cited at page 22.

[36] Ralf Plänkers and Pascal Fua. Articulated soft objects for multiview shape and motion capture. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25:1182–1187, 2003. ISSN 01628828. doi: 10.1109/TPAMI.2003.1227995. Cited at page 23.

[37] J MacCormick and M Isard. Partitioned Sampling, Articulated Objects, and Interface-quality Hand Tracking. In *Computer Vision?ECCV 2000*, pages 3–19, 2000. ISBN 978-3-540-67686-7, 978-3-540-45053-5. URL `http://link.springer.com/chapter/10.1007/3-540-45053-X_1`. Cited at page 23.

[38] Daniel Vlasic, Ilya Baran, Wojciech Matusik, and Jovan Popović. Articulated mesh animation from multi-view silhouettes, 2008. ISSN 07300301. Cited at page 23.

[39] Zheng Zhang, Hock Soon Seah, Chee Kwang Quah, and Jixiang Sun. A markerless motion capture system with automatic subject-specific body model acquisition and robust pose tracking from 3D data. In *Proceedings - International Conference on Image Processing, ICIP*, pages 525–528, 2011. ISBN 9781457713033. doi: 10.1109/ICIP.2011.6116397. Cited at page 23.

[40] Yebin Liu, Juergen Gall, Carsten Stoll, Qionghai Dai, Hans-Peter Seidel, and Christian Theobalt. Markerless motion capture of multiple characters using multiview image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 35(11):2720–35, November 2013. ISSN 1939-3539. URL `http://www.ncbi.nlm.nih.gov/pubmed/24051731`. Cited at page 23.

[41] Chenglei Wu, Carsten Stoll, Levi Valgaerts, and Christian Theobalt. On-set performance capture of multiple actors with a stereo camera. *ACM Transactions on*

Graphics, 32:1–11, 2013. ISSN 07300301. doi: 10.1145/2508363.2508418. URL http://dl.acm.org/citation.cfm?doid=2508363.2508418. Cited at page 23.

[42] Kai Berger, Kai Ruhl, Yannic Schroeder, Christian Bruemmer, Alexander Scholz, and Marcus Magnor. Markerless Motion Capture using multiple Color-Depth Sensors. *Sensors Peterborough NH*, 2011:317–324, 2011. doi: 10.2312/PE/VMV/VMV11/317-324. URL http://graphics.tu-bs.de/media/publications/multikinectsMocap.pdf. Cited at page 24.

[43] Guangyu Mu, Miao Liao, Ruigang Yang, Dantong Ouyang, Zhiwen Xu, and Xiaoxin Guo. Complete 3D model reconstruction using two types of depth sensors. *Intelligent Computing and Intelligent Systems (ICIS), 2010 IEEE International Conference on*, 1, 2010. doi: 10.1109/ICICISYS.2010.5658733. Cited at page 24.

[44] Y Cui and D Stricker. 3D Shape Scanning with a Kinect. *Artificial Intelligence*, page 57, 2011. doi: 10.1145/2037715.2037780. URL http://dl.acm.org/citation.cfm?id=2037780. Cited at page 24.

[45] Hao Du, Peter Henry, Xiaofeng Ren, Marvin Cheng, Dan B Goldman, Steven M Seitz, and Dieter Fox. Interactive 3D Modeling of Indoor Environments with a Consumer Depth Camera. *Science*, pages 75–84, 2011. doi: 10.1145/2030112.2030123. URL http://dl.acm.org/citation.cfm?doid=2030112.2030123. Cited at page 24.

[46] Shahram Izadi, Andrew Davison, Andrew Fitzgibbon, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, and Dustin Freeman. Kinect Fusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera. In *Proceedings of the 24th annual ACM symposium on User interface software and technology - UIST '11*, page 559, 2011. ISBN 9781450307161. doi: 10.1145/2047196.2047270. URL http://dl.acm.org/citation.cfm?id=2047270$\delimiter"026E30F$nhttp://dl.acm.org/citation.cfm?doid=2047196.2047270. Cited at page 24.

[47] Richard A. Newcombe, Andrew J. Davison, Shahram Izadi, Pushmeet Kohli, Otmar Hilliges, Jamie Shotton, David Molyneaux, Steve Hodges, David Kim, and Andrew Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 127–136, 2011. ISSN ¡null¿. doi: 10.1109/ISMAR.2011.6092378. Cited at page 24.

[48] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox. RGB-D mapping: Using Kinect-style depth cameras for dense 3D modeling of indoor environments, 2012. ISSN 0278-3649. Cited at page 25.

[49] Qian-Yi Zhou and Vladlen Koltun. Dense scene reconstruction with points of interest. *ACM Transactions on Graphics*, 32:112:1—-112:8, 2013. ISSN 07300301. doi: 10.1145/2461912.2461919. URL http://dl.acm.org/citation.cfm?id=2461912.2461919. Cited at page 25.

[50] M Zollhöfer and Michael Martinek. Automatic reconstruction of personalized avatars from 3D face scans. *Computer Animation and Virtual Worlds*, pages 195–202, 2011. ISSN 1546-4261. doi: 10.1002/cav. URL `http://onlinelibrary.wiley.com/doi/10.1002/cav.405/full`. Cited at page 26.

[51] Ruizhe Wang, J Choi, and Gerard Medioni. Accurate Full Body Scanning from a Single Fixed 3D Camera. *3D Imaging, Modeling, ...*, 2012. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6375025`. Cited at page 26.

[52] Yan Cui, Will Chang, T Nöll, and D Stricker. KinectAvatar: fully automatic body capture using a single kinect. In *ACCV 2012 Workshops*, pages 133–147, 2013. doi: 10.1007/978-3-642-37484-5_12. URL `http://link.springer.com/chapter/10.1007/978-3-642-37484-5_12`. Cited at page 26.

[53] P. J. Besl and H. D. McKay. A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, Feb 1992. ISSN 0162-8828. doi: 10.1109/34.121791. Cited at page 35.

[54] OpenKinect.org, . URL `https://github.com/OpenKinect/libfreenect2`. Cited at page 43.

[55] Greg Turk. URL `http://paulbourke.net/dataformats/ply/`. Cited at page 57.

[56] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965. doi: 10.1093/comjnl/7.4.308. URL `http://comjnl.oxfordjournals.org/content/7/4/308.abstract`. Cited at page 65.

[57] Occipital, . URL `https://github.com/occipital/OpenNI2/commits/kinect2`. Cited at page 69.

[58] VICON. URL `http://www.vicon.com/`. Cited at page 70.

List of Figures

List of Tables