LaTeX Author Guidelines for 3DV Proceedings

Anonymous 3DV submission

Paper ID ****

Abstract

The goal of this project was to create an augmented reality chess game. We used two cameras - an RGB-D camera and a thermal camera. The RGB camera is used to track a paper checkerboard with augmented reality markers which are used to estimate the pose of the camera. The video with the resulting camera matrix are used by OpenGL to augment the video with the virtual game objects. We use a thermal camera for the detection of the user input.

1. Introduction

Augmented reality (AR) is a live direct or indirect view of a physical, real-world environment whose elements are augmented by computer-generated sensory input such as sound, video or graphics.

1.1. Motivation

> On September 27, 1998 a yellow line appeared across the gridiron during an otherwise ordinary football game between the Cincinnati Bengals and the Baltimore Ravens. It had been added by a computer that analyzed the camera's position and the shape of the ground in real-time in order to overlay thin yellow strip onto the field. The line marked marked the position of the next first-down, but it also marked the beginning of a new era of computer vision in live sports, from computerized pitch analysis in baseball to automatic line-refs in tennis.

Augmented and Virtual Reality have come a long way since then and products such as Microsoft Kinect, Google Glass or the yet-to-be-released Occulus Rift or Microsoft Hololens have amazed the world. We chose this project in pursuit of understanding the challenges that have to be overcome in augmented reality and user interface engineering. Our goal was to create a simple augmented reality chess game while exploring the possibilities of augmented reality combined with real-life object interfacing through touch detection with a low-tech infrared camera on arbitrary surfaces.

1.2. Related work

For simpler augmented reality applications, such as our chess game, there is quite a simple way to accurately and robustly track the camera poses in real-time - augmented reality markers. These markers consist of an easily detectable square with a specific pattern inside that helps make the pose estimation accurate. In our project, we used Aruco [4] library which is a lightweight library based on OpenCV [5]. It defines its own set of markers and easy-to-use camera pose estimation framework. The outputted extrinsic camera parameters in combination with the camera calibration matrix can be passed into a rendering engine, which can then augment the video stream with additional virtual geometry.

Research on user input detection using thermal cameras has been done before. In [2] they show how to exploit stereo-like setup of an RGB and a thermal camera. The detection of the user input is made easy as when the user touches the interface-object, he transfers heat from his fingers onto the surface of the object. These thermal spikes are easily detectable by blob detectors. On the assumption that the geometry of the object used for infrared input detection is known, provided an accurate 3D object tracking (and pose estimation), the detected user input points can be back-projected into 3D space, intersected with the interface-object surface, providing the 3D coordinates of the touch, which can be used by the application.

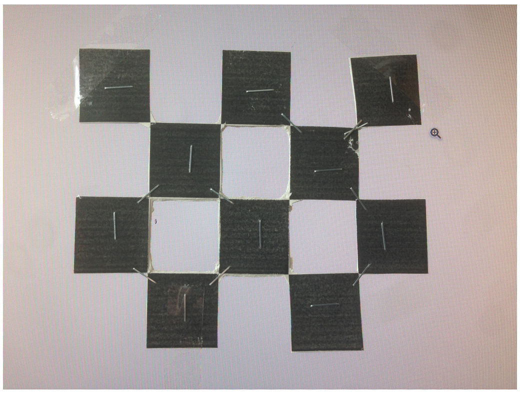
2. The problem decomposed

This section describes all the key problems that we had to solve in order to implement our game.

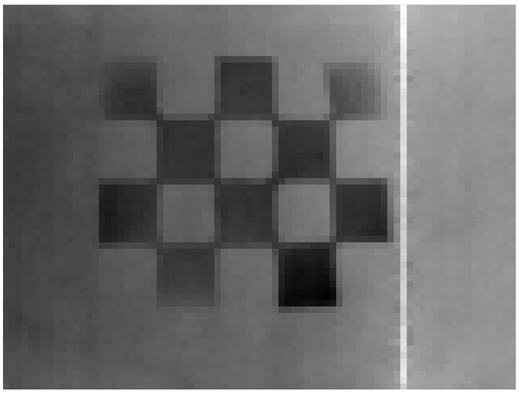
2.1. Preprocessing

The first step of creating our augmented reality application is to calibrate the cameras. Calibrating an RGB camera is easy. However, calibrating a low resolution (64x64) IR camera poses a challenge as the standard checkerboard pattern is not visible in the IR image. For this reason, we cut out the white parts of the checkerboard and taped it to a

3DV #****

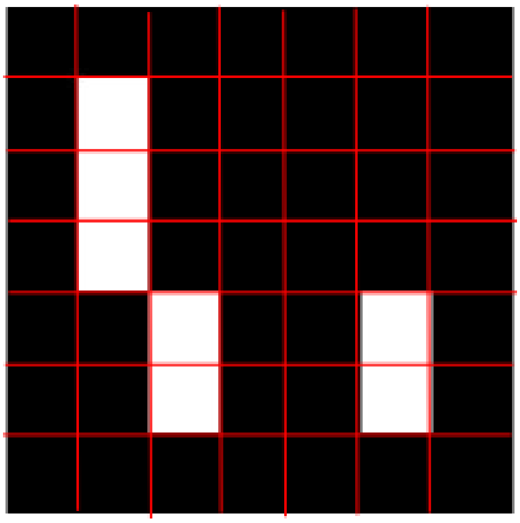


(a) RGB image of our calibration setup

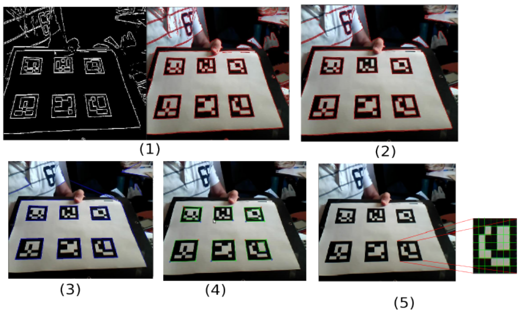


(b) thermal image of our calibration setup

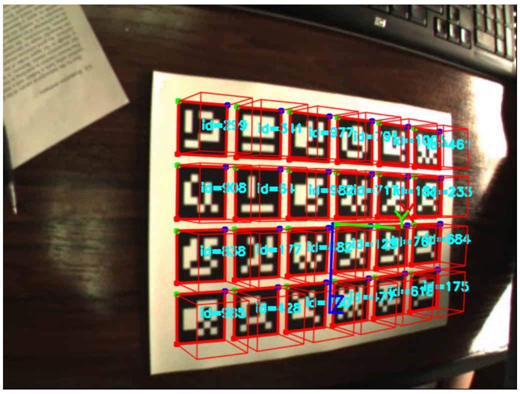
Figure 1: Calibration setup



(a) A single Aruco marker



(b) The scheme of detection of markers on one board



(c) Board with simple graphics rendered over it using the correct pose estimation

Figure 2: Aruco workflow scheme

warm screen. You can see the results of our manual work in Fig. 1. Because of the low resolution of the IR image (which is further reduced by a broken column and a brighter region on the right side of the broken column), we have not been able to estimate the initial rigid motion transform from camera to camera accurately.

2.2. Tracking and Pose Estimation

Another problem to tackle is the checkerboard detection with pose estimation. We were considering mulitple possibilities. At first we wanted to assume that out camera will be static. Then we would detect standard 8x8 checkerboard pattern to estimate the pose just once in program initialization stage. However, this simple approach would not be enough as the slightest movement of the camera or checkerboard would invalidate the camera pose and the virtual geometry would not be rendered in the right place. Therefore we decided to use a library for augmented reality - Aruco [4] , which uses a special set of augmented reality markers. The marker consists of a square border and a rotation-invariant pattern inside, which encodes the marker's ID. These markers make it easy to estimate the pose. For the detailed description of the algorithm, please refer to Aruco website. As our cameras are taped together creating a stereo setup, by knowing the pose of the RGB camera and the rigid motion transform from the RGB camera to the IR camera, we can compute the pose of the IR camera.

2.3. Input Detection

To detect the residual heat resulting from the user touching the board we use OpenCV blob detector. We filter the detected blobs by heat (pixel value) and by circularity. We have been able to tweak the parameters in such a manner that we get no false detections. In other words, only the slightly brighter touched spot gets detected and not the hand or other body parts which are much warmer and are not of circular shape. Therefore, we did not have to use the depth data from Kinect (as was initially planned), which is a very good result. Given IR camera intrinsics and extrinsics we backproject the detected point into 3D space and intersect the resulting ray with the chessboard located on the xy-plane. Then we can easily obtain the chess coordinates of the touched square.

2.4. Occlusions

For more realistic AR effect we also employ occlusion detection. We get an occlusion mask computed by Aruco. Unfortunately, the occlusion mask is very noisy and unusable for our purposes. Therefore, we exploit image opening to remove the noise (Fig. 4). Afterwards we use the mask to extract the hand and prevent the virtual object to be rendered over the occluding hand.

2.5. Result

We get an interactive 3D augmented reality chess game, which can be played against a computer AI with visually pleasing figure animations. The input detection works well without detecting false positives without the need of depth information for input validation. The pose estimation is very stable and holds even when large part of the board is occluded by the player. As a result the camera can move freely around the checkerboard and the virtual geometry stays in the right place. The only reason which prevents our game from being playable is the inaccurate thermal camera calibration and its initial pose estimation. Given a better IR camera and a proper accurate stereo calibration, our game is ready to be played.

(a) Noisy occlusion mask

(b) Denoised occlusion mask

Figure 3: An occlusion mask example

3. Application Details

This section describes the key components of our final application. Appendix A describes in detail the initial project proposal, changes that have been made, technical issues that have been encountered as well as the whole progress.

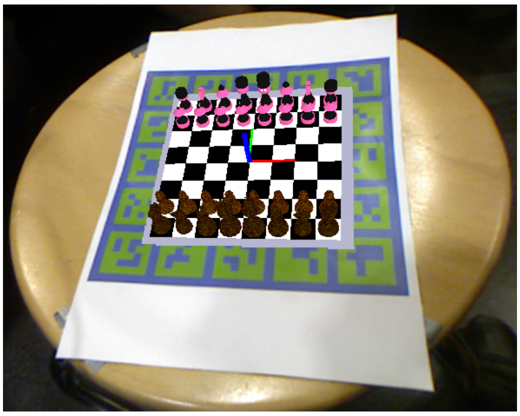
3.1. Overview

As our game runs under ROS on Ubuntu it, consists of several nodes described in the following subsections. Most of our coding is done in Python, some in C++. Our application runs in real-time. PC without a GPU or the Odroid device might have a lower (but still real-time) framerate.

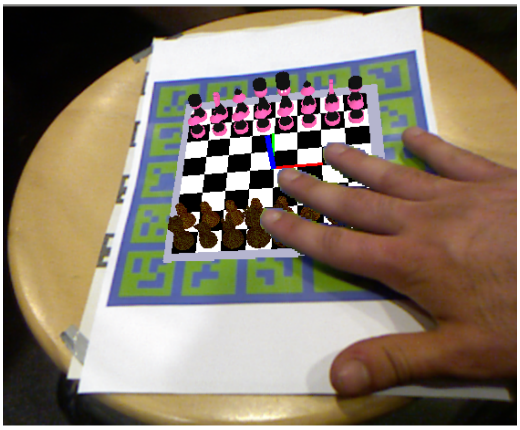
3.2. Main Game Node

Main game node is a python script. It initiates the game engine, sets the engine's projection matrix from the calibration of the RGB camera and then keeps receiving all the data processes them and passing them to the game engine. The description of the most important parts of the node follows:

- **IR listener:** This listener receives the IR image data. As our IR sensor has only resolution of 64x64, the image is first upsampled to make it usable for the input detection. To detect the residual heat resulting from the user touching the board we use OpenCV blob detector. We filter the detected blobs by heat (pixel value) and



(a) Chess game rendered on an Aruco board.



(b) Game with a hand occluding the virtual objects. Note that the virtual objects indeed do not get rendered over the hand.

Figure 4: Augmented reality chess game

by circularity. We have been able to tweak the parameters in such a manner that we get no false detections. In other words, only the slightly brighter touched spot gets detected and not the hand or other body parts which are much warmer and are not of circular shape. Therefore, we did not have to use the depth data from Kinect (as was initially planned), which is a very good result. Our RGB and IR cameras are fixed together, which means there is a rigid motion transform between them and since we know the extrinsics of the RGB camera, we can also compute the extrinsics of the IR camera. Given an accurate calibration of the IR camera, we can then easily backproject the detected input points, intersect the resulting ray with the checkerboard plane and therefore compute the 2D coordinates on the plane. These are then passed to the game engine.

- **RGB listener:** This listener receives the rectified RGB images from the OpenNI node and passes them to our game engine, where the images are used as a background over which the virtual objects are rendered.

- **Occlusion mask listener:** This listener receives the occlusion mask and passes it to the game engine. The occlusion mask is used to determine, where not to render the virtual objects. This creates a realistic effect that when a player's hand occludes the board, the virtual objects get occluded as well.

- **Pose listener:** This listener receives the extrinsics of our camera from the Ar-Sys node and passes them to our game engine, where it is used as the model view matrix for OpenGL.

Relevant file: `listener.py`
Coded by: Radek Danecek

3.3. Ar-Sys: Aruco ROS node

For the checkerboard tracking and pose estimation we use Ar-Sys [3]. It is a wrapper around Aruco library for ROS. It is used to track a special checkerboard filled with augmented reality markers. We have extended this wrapper for the purposes of this project to enable the support of the Aruco's so called "Highly Reliable Markers", which provide more stable pose estimation and also support the creation of the occlusion mask. The occlusion mask is computed by an Aruco function which uses a simple background subtraction algorithm. As the occlusion mask from the Aruco library contained many holes, we perform an image opening operation on it to fill the gaps.

Both camera pose and the occlusion masks are streamed in real-time to the main game node. By employing this library, we can move our camera freely around the board and the virtual objects get rendered exactly at the right place, which looks visually pleasing. Therefore, we have completed a secondary objective of our project, as at first we wanted to create our game with static camera only.

Relevant files: `single_board.cpp`, `single_board_occlusion.cpp`, `single_board_kinect.launch`, `single_board_kinect_occlusion.launch`

3.4. Game Engine

The graphics for the argument reality chess are completely written in python with OpenGL and GLut. For the chess figure models exist two options. The first one is to use only primitives like spheres, cones or other quadratics for figure modeling. The huge advantage is that this objects are natively supported by openGL and they improve the rendering in terms of FPS. However if the user has a graphics card he could use the second option, which load the figures as standard obj files. This files contain a set of verticies, faces, normals and texture coordinates, which are loaded in the initialization of the game. Optional it is possible to assign a material file (mtl) to an obj file. These files contain detailed information about the material properties of parts of the model. They can for example specify the texture, the ambient, specular or diffuse color.

Another feature is the GLut context menu, which allows the user to change the rendering properties during the runtime. For example it is possible to toggle shadows or basic animations.

To create the best possible AR effect we used the 3 following steps. At first the RGB frame is rendered as an orthogonal projection to get the video inside rendering. After that we render the checkerboard, and the figures in the current game state.

To create a good AR effect we update the openGL model view and projection matrix every time the listener receives a new frame. This gives us the possibility to move the camera freely around the board. In the last step we use our occlusion mask to render the players hand as an RGBA over the figures as a third layer.

The whole chess logic is computed by the open source chess engine Sunfish [1]. The engine also checks if a given move is a valid step and computes the next move for the computer AI opponent.

Another feature of the engine is that, the game is playable even if no thermal camera exists. It is possible to use the mouse as input device and click directly on the 2 squares to define a move. The 2D screen coordinates are unprojected using the model, view and projection matrix to 3D space, after that we now the structures of the checkerboard in the xy plane and can easily detected the click or touched square.

Relevant file: `GameNoLogic.py`

Coded by: Alex Lelidis

3.5. Odroid/IR camera node

For this project we have received a small low-tech IR sensor with 64x64 resolution. It runs on Odroid with ROS and Ubuntu. Together with the camera and the Odroid, we have also been provided a ROS publisher node, from which we read the IR image data.

3.6. OpenNI node

ROS node for standard OpenNI driver for Microsoft Kinect. It publishes RGB and depth data.

4. Conclusion

We have created a simple augmented reality chess game that runs in real time and uses thermal camera for input detection and Aruco library for pose estimation and checkerboard tracking. We have successfully applied and extended our knowledge in Computer Vision and Computer Graphics. We were happy that we could get our hands on quite recent hardware (the IR camera) and also extended our range of technical skills (such as working with ROS or OpenCV) and we are pleased with the overall result.

5. Appendix A: Progress

This section describes everything we have done from the initial plans, the changes we have made and our progress throughout the semester.

5.1. Project Proposal and Initial plans

The initial project proposal has been provided with this document.

5.2. Setting up the project

The ROS and OpenCV shit

5.3. Before midterm

Work on graphics and checkerboard tracking and pose estimation.

5.4. After midterm

Transition to highly reliable markers. Occlusion mask. Obtaining the thermal camera.

5.5. Final push

Problems with camera calibration and pose estimation. Fucking camera breaks all the time and stuff.

6. Appendix B: Installation

6.1. AugmentedRealityChess

The installation is tested on Ubuntu 14.04.

6.1.1 Install OpenNI

This is required for the kinect interface

```
sudo apt-get install git-core
cmake freeglut3-dev pkg-config
build-essential libxmu-dev libxi-dev
libusb-1.0-0-dev doxygen graphviz
mono-complete
```

Now clone the code and set it up

```
$ mkdir ~/kinect
$ cd ~/kinect
$ git clone
https://github.com/OpenNI/OpenNI.git
```

This thing has a bizarre install scheme. Do the following:

```
cd OpenNI/Platform/Linux/CreateRedist/
chmod +x RedistMaker
./RedistMaker
```

Now this creates some distribution.

One of the two following cases should work. Else just look for a damn compiled binary, extract it and install it.
Case 1:

```
$ cd Final
$ tar -xjf OpenNI-Bin-Dev-Linux*bz2
$ cd OpenNI- ...
$ sudo ./install.sh
```

6.1.2 Install SensorKinect

Yet another library for the Kinect `$ cd ~/kinect/`

```
$ git clone
git://github.com/ph4m/SensorKinect.git
```

Once you have the lib, go ahead and compile it in the same bizarre manner as OpenNI (well atleast they are consistent).

```
$ cd
SensorKinect/Platform/Linux/CreateRedist/
$ chmod +x RedistMaker
$ ./RedistMaker
```

Done compiling. Now install this.

```
$ cd Final
$ tar -xjf Sensor ...
$ cd Sensor ...
$ sudo ./install.sh
```

This thing has a bizarre install scheme. Do the following:

```
cd OpenNI/Platform/Linux/CreateRedist/
chmod +x RedistMaker
./RedistMaker
```

Now this creates some distribution. One of the two following cases should work. Else just look for a damn compiled binary, extract it and install it.
Case 1:

```
$ cd Final
$ tar -xjf OpenNI-Bin-Dev-Linux*bz2
$ cd OpenNI- ...
$ sudo ./install.sh
```

6.1.3 Set up OpenCV

These steps have been tested for Ubuntu 14.04 but should work with other distros as well.
Required Packages

1. GCC 4.4.x or later
2. CMake 2.8.7 or higher
3. Git
4. GTK+2.x or higher, including headers (libgtk2.0-dev)
5. pkg-config 5. Python 2.6 or later and Numpy 1.5 or later with developer packages (python-dev, python-numpy)
6. ffmpeg or libav development packages: libavcodec-dev, libavformat-dev, libswscale-dev
7. [optional] libtbb2 libtbb-dev
8. [optional] libdc1394 2.x
9. [optional] libjpeg-dev, libpng-dev, libtiff-dev, libjasper-dev, libdc1394-22-dev The packages can be installed using a terminal and the following commands or by using Synaptic Manager:

[compiler] `sudo apt-get install build-essential`
[required] `sudo apt-get install cmake git libgtk2.0-dev pkg-config libavcodec-dev libavformat-dev libswscale-dev`
[optional] `sudo apt-get install python-dev python-numpy libtbb2 libtbb-dev libjpeg-dev libpng-dev libtiff-dev libjasper-dev libdc1394-22-dev`

This thing has a bizarre install scheme. Do the following:

```
cd OpenNI/Platform/Linux/CreateRedist/
chmod +x RedistMaker
./RedistMaker
```

Now this creates some distribution. One of the two following cases should work. Else just look for a damn compiled binary, extract it and install it.
Case 1:

```
$ cd Final
$ tar -xjf OpenNI-Bin-Dev-Linux*bz2
$ cd OpenNI- ...
$ sudo ./install.sh
```

Getting OpenCV Source Code

You can use the OpenCV versio 2.4.9.
For example

```
cd ~/<my_working_directory>
git clone
https://github.com/Itseez/opencv.git
git clone
https://github.com/Itseez/opencv_contrib.git
```

Building OpenCV 2.4.9 from Source Using CMake

1. Create a temporary directory, which we denote as , where you want to put the generated Makefiles, project files as well the object files and output binaries and enter there. For example
   ```
   cd ~/opencv2.4.9
   mkdir build
   cd build
   ```

2. Configuring. Run cmake [some optional parameters] path to the OpenCV source directory
   For example
   `cmake -D CMAKE_BUILD_TYPE=Release -D CMAKE_INSTALL_PREFIX=/usr/local ..` or `cmake-gui`

- set full path to OpenCV source code, e.g. /home/user/opencv
- set full path to , e.g. /home/user/opencv/build
- set optional parameters
- run: "Configure"
- run: "Generate"

3. Description of some parameters

- build type: CMAKE_BUILD_TYPE=Release Debug
- to build with modules from opencv_contrib set OPENCV_EXTRA_MODULES_PATH to
- set BUILD_DOCS for building documents
- set BUILD_EXAMPLES to build all examples

4. Building python. Set the following python parameters:

- PYTHON2(3)_EXECUTABLE =
- PYTHON_INCLUDE_DIR = /usr/include/python
- PYTHON_INCLUDE_DIR2 = /usr/include/x86_64-linux-gnu/python
- PYTHON_LIBRARY = /usr/lib/x86_64-linux-gnu/libpython.so
- PYTHON2(3)_NUMPY_INCLUDE_DIRS = /usr/lib/python/dist-packages/numpy/core/include/

5. Build. From build directory execute make, recomend to do it in several threads For example
   `make -j7 # runs 7 jobs in parallel`

6. `sudo make install`

6.1.4 Install Ros

1. Installation
   1.1. Configure your Ubuntu repositories Configure your Ubuntu repositories to allow "restricted," "universe," and "multiverse." You can follow the Ubuntu guide for instructions on doing this.
   1.2. Setup your sources.list Setup your computer to accept software from packages.ros.org. ROS Jade ONLY supports Trusty (14.04), Utopic (14.10) and Vivid (15.04) for debian packages.`sudo sh -c echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list`
   1.3. Set up your keys `sudo apt-key adv --keyserver hkp://pool.sks-keyservers.net --recv-key 0xB01FA116`
   1.4. Installation First, make sure your Debian package index is up-to-date: `sudo apt-get update` If you are using Ubuntu Trusty **14.04.2** and experience dependency issues during the ROS installation, you may have to install some additional system dependencies. **/! Do not install these packages if you are using 14.04, it will destroy your X server:**

`sudo apt-get install xserver-xorg-dev-lts-utopic mesa-common-dev-lts-utopic libxatracker-dev-lts-utopic libopenvg1-mesa-dev-lts-utopic libgles2-mesa-dev-lts-utopic libgles1-mesa-dev-lts-utopic libgl1-mesa-dev-lts-utopic libgbm-dev-lts-utopic libegl1-mesa-dev-lts-utopic` **! Do not install the above packages if you are using 14.04, it will destroy your X server!** Alternatively, try installing just this to fix dependency issues: `sudo apt-get install libgl1-mesa-dev-lts-utopic` Desktop-Full Install: (Recommended) : ROS, rqt, rviz, robot-generic libraries, 2D/3D simulators, navigation and 2D/3D perception sudo apt-get install ros-jade-desktop-full or click here Desktop Install: ROS, rqt, rviz, and robot-generic libraries `sudo apt-get install ros-jade-desktop` ROS-Base: (Bare Bones) ROS package, build, and communication libraries. No GUI tools. `sudo apt-get install ros-jade-ros-base` Individual Package: You can also install a specific ROS package (replace underscores with dashes of the package name): `sudo apt-get install ros-jade-PACKAGE` e.g. `sudo apt-get install ros-jade-slam-gmapping` To find available packages, use: `apt-cache search ros-jade` 1.5. Initialize rosdep Before you can use ROS, you will need to initialize rosdep. rosdep enables you to easily install system dependencies for source you want to compile and is required to run some core components in ROS. `sudo rosdep init rosdep update`
1.6. Environment setup It's convenient if the ROS environment variables are automatically added to your bash

session every time a new shell is launched:

echo "source /opt/ros/jade/setup.bash" >> ~/.bashrc source ~/.bashrc If you have more than one ROS distribution installed, ~/.bashrc must only source the setup.bash for the version you are currently using.

If you just want to change the environment of your current shell, you can type:

source /opt/ros/jade/setup.bash

1.7. Getting rosinstall rosinstall is a frequently used command-line tool in ROS that is distributed separately. It enables you to easily download many source trees for ROS packages with one command.

To install this tool on Ubuntu, run:

sudo apt-get install python-rosinstall Build farm status The packages that you installed were built by ROS build farm.

6.1.5 Install ar_sys

3D pose estimation ROS package using ArUco marker boards. To install this package run `git clone https://github.com/coloss/ar_sys.git`

6.1.6 Install PyOpenGL

To be able to run the animations you new to have PyOpenGL, the quickest way to install it is using pip `$ pip install PyOpenGL PyOpenGL_accelerate`

6.1.7 Set up Augmented Reality Chess

To run the source code properly a specific file structure is needed.

1. Create a catkin workspace `cd ~; mkdir ~/catkin_ws`
2. Clone the ros part of the implementation in this directory `git clone https://github.com/alexus37/ROSARCHESS.git`
3. Clone the rendering part in an arbitary folder and link the path in the file catkin_ws/src/kinect_io/scripts/listener.py `git clone https://github.com/alexus37/AugmentedRealityChess.git`
4. Calibrate the Kinect camera using the `ros CALI BLA` to create the a cali.yml file
5. Calibrate the IR camera and create the a cali.yml file

6.1.8 Run the game

1. Run the roscore `roscore`
2. Open a new terminal and run openNi to be able to interact with the kinnect `roslaunch openni openi.launch`
3. Open a new terminal and run ros arsys to be able to track the markers `roslaunch arsys singleboardOcclusion`
4. connecte via ssh to connect to the thermal camera. `shh px4@192.168.1.2`
5. Also run the roscore on the IR cam `roscore`
6. Run the command `rosrun px4 px4`
7. Launch the video stream `roslauch leptonvideo leptonvideo`
8. Open a new terminal on your machine and run the listener `roslaunch kinectio kinectio.listner`

References

[1] T. Ahle. Sunfish chess engine, 2015. 6

[2] D. Kurz. Thermal touch: Thermography-enabled everywhere touch interfaces for mobile augmented reality applications. *Metaio GmbH*, 8(1):1–8, 2014. 2

[3] Library. Ar-sys, 2015. 5

[4] Library. Aruco, 2015. 2, 3

[5] Library. Opencv, 2015. 2