Introduction

Rocky Hockey is a data analysis project by a group of five students at the University of Applied Sciences Kempten, Germany. The topic was self-chosen as we were looking for something fun and challenging. The goal is to build a robot-controlled air hockey table, which plays against a human opponent.

Proposed Features

- computer vision to track the puck
- physics simulation to predict puck trajectory
- AI to react to player actions
- graphics projected onto the table surface
- high score list
- computer driven mechanic to move the striker
- oh, an air hockey table

Specifications

- 160x90cm playing field
- two axis high speed robot arm
- Nvidia Tegra X1 processor
- 2*180Hz 320x240px Sony Playstation Eye camera
- two 99W industrial fans from EBM Papst

Problems during the build process

Hardware Problems

- **Airflow:** The holes for the airflow in the table are too big and the space between them is to large. It would be better if we used 3mm holes and a space of less than 25mm instead of 5mm holes and a space of 30mm. This change would give us a better air flow which would reduce the jitter while the puck is moving with low velocity.
- **Rods for the striker movement:** The rods for the striker movement aren't parallel because the holes in which they are inserted are not drilled precisely enough. On the Y-Axis they diverge around 4mm towards the center of the table and on the X-Axis theres a difference of about 2mm. This results in a reduced flexibility of the striker, which means that it can't move with max velocity withput losing steps.

Software Problems

- **Millimeter to pixel ratio:** At the moment we use 2 webcams with a resolution of 320*240px each*

*for a 1600mm*900mm table. But the ratio is too big. This means, that the jitter of the puck is increased, because OpenCV doesn't always detect the exact center of the puck.

- **Finding the correct FPS for the webcam**: If we use a low framerate, it's much better for the puck prediction, because the direction vector becomes more accurate when the puck has more time between 2 frames. Sadly, the low framerate is bad for our motion controller, because we get the information where the puck ig going to be, even too late sometimes. A higher framerate would give us this information faster, but it isn't as accurate anymore (see mm to px ratio for the reason).
- **Table distortion**: Due the position of the webcams we get some unwanted distortions in the image. Before we can stitch them together we have to undistort them. We tried to use an automatic undistortion and a manual one. The automatic distortion gave us some good results, but it can only detect tables that have lines which are parallel to the cameras, which means, that the resulting image is slightly to large and has some unwanted areas. The manual stitching needs more time for calibration, because we have to take a sample image for each webcam and define the corners of the table by hand. The results there are much better and give us a very good undistorted image of the table. But due to the manual definition we faced new problems. When the webcams are being moved even slightly, which can happen very easily if you reloacate the table into another position or touch the cables, you have to readjust the values for the manual calibration.
- **Object Tracking**: We tried many ways for object tracking. Some with background subtraction some without it. Both have their benefits and problems. With the background subtraction we get the problem that the starting position of the striker would get subtracted with the background. It would be the same with the puck, but we can just take it off the table when we boot our application. With the subtraction, we sometimes get the problem that the Blobdetector detects the negative striker (position of the striker when we start the app). This results in the robot smashing the striker to the right wall with max velocity. When we disable the background subtraction we get problems with lighting and shadows in different enviroments. We noticed that the webcam generally has difficulty detecting the striker when the light conditions aren't good enough. One of our solutions we found was to use the flashlight of a smartphone to illuminate the striker. If we would have had more time, we would have added some LEDs on the top of the table to illuminate the whole table with a diffuse light. We also tried different ways/algorithms of object tracking. Some were good (HoughCircles), some were bad (nearly every Tracker from Tracking.h). We found out that we got the best results when we just used a simple blob detector, which detects just circles with a specific surface area. With some changes in the parameters of the blob detector we got a huge FPS boost, which is nice.
- **Compile Time**: We started to use the boost framework for things like the start argument parser and the serial communication with our Ardurino. Unfortunately, this increased our compilation time to about 10 times of what we had before.

Motion Controller Problems

- **GRBL**: We started by using GRBL to move the striker to the correct position with the exact number of motor steps needed. This works very well for single movements with a cooldown that waits

until every move is completed. But this doesn't fit the needs of a fast response time. And the buffer still hinders us in adjusting the stricker when the prediction value changes. We have to be able to cancel actions while moving and send commands without a cooldown. If you try this with GRBL it will save the commands in a buffer and that gives us a huge input lag. Canceling movements doesn't work very well either, if you try to do it too often it will crash the whole motion controller. To avoid these problems, we wrote our own motion controller, which just accepts a positive speed value for moving right and a negative speed value for moving left.

Compiling (Linux)

To compile the project under linux you require these libraries. * Boost (1.55 or higher) * OpenCV (3.1 or higher)

Debian 64Bit / Ubuntu 64Bit

```
apt-get install libboost-all-dev build-essentials
```

For OpenCV you have to install the openCV .deb package or compile it yourself

```
dpkg -i opencv_20170325-1_amd64.deb
```

When the required tools are installed, go to the project folder and type:

```
cmake ./
```

```
make
```

Quickstart

Before you run the application, make sure you've at leat 1 webcam plugged in. Then start the application with

```
./rocky_hockey -c 0
```

Motion Controller (Arduino-C++ Application)

The motion controller controls our motors and regulates their speed. For that we use the PWM module of our motion controller (AVR ATMega328P), we create a pulse width modulation, where the pulse width won't actually be taken into account. We will look at the frequency, or more accurately, the falling edges per second. On every falling edge, our driver will make our motor move a single step (Stepper motor). The PWM module has a big advantage, and that is that the period of the PWM can simply be set by a register and therefore won't affect the CPU at all. The set speed will be transmitted to the motion controller by the JETSON Board using our serial port (UART, 1 Start bit, 8 Data bits, 1 Stop bit, baud rate 115200). We're transferring a single byte, which displays the set

speed as complement of two. The sign (positive or negative) indicates the direction. The set value does not have a concrete value, but is only in ratio to the max speed we were using (100%: 127/-128). We found the max speed by trial and error, we were experimenting until the motor was running smoothly.

We need the speed control, or easing, because we can't gate the motors directly with the exact speed that we want, because the motors would always jam up on us. To get the set speed without problems we need to ensure a smooth acceleration. The digital controller consists of a PD control (I is implemented, but coefficient 0) and an exponential function, which generates start-up curve. We will use the previously used set point as the current actual value. Here, just like before, we found our coefficients by trial and error. When changing directions, easing does not come into effect. We try to get the striker to 0 speed immediately using the motors holding current, that doesn't work perfectly and we lose a couple of steps, but that's not a problem. We only move our striker relatively and the absolute positioning wil be done by the camera tracing the puck.

Possible improvements: * Controller: using a PT1-element instead of the exponential function that we implemented (PT1 is suitable for regulation speed control) * Controller: Installing and using an Encoder * Motor control: Using Servos instead of Stepper motors. They will allow us to get better torque and also to use analog voltage (DAC) over a MOSFET H-bridge as the driver. * Controller/Firmware: Using fixed point arithmetic, to relieve the motion controller and for possibly adding new functionalities

Aufgabenverteilung

Christopher : MotionController, C++2014 Refactoring, C++ Improvements,
David : Logo Design, Puck Tracking, Puck Prediction
Falk : Tischbau, Aufbau Motion Controller, Config, Communication Management,
Matthias : Aufbau Motion Controller, Verkabelung, Multithreading
Mirko: Puck Tracking, Puck Prediction, Multicam Setup, Management

Durch das Arbeiten direkt am Board entsprechen die Autoren der Commits nicht immer den richtigen Autoren.